DATA LOCALIZATION IN PARALLEL COMPUTER SYSTEMS

by

Mann-Ho Lee

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

February 1991

Acknowledgements

I am deeply grateful to Dr. Dennis Gannon for his help as my advisor. He introduced me to and taught me about the area of Parallel Processing and I am still fascinated by the subject. He helped me with new ideas and always gave me a patient hearing when I had a problem or a new idea. I also wish to express my thanks to Professors Robert Glassey, Christopher Haynes, Gregory Shannon, and Dirk Van Gucht for agreeing to be on my committee. They helped me improve the quality of this thesis by making invaluable comments.

I wish to express my special appreciation to the fellows in the Sigma Group: Daya Atapattu, Jenq Kuen Lee, and Bruce Shei. They helped me with technical assistance, were always ready to discuss anything, and gave encouragement and helpful suggestions. My thanks also go to my friend Myunggoo Choi who helped with comments on English writing.

Last but not least, I thank my family both here and in Korea. To my wife Dong-Hee and son Yoon-Soo, I express my gratitude for their understanding, patience, and sacrifices throughout this difficult period. I would like to express my appreciation to my parents-in-law, Byung-Kwon Kim and Joeng-Sook Park, for their encouragement and financial assistance. Without their support, I could neither last for such a long time nor complete this dissertation. Finally, I owe a special debt of gratitude to my mother Soon-Ja Pang for praying for me over the years.

Abstract

In parallel computer systems, the efficient use of local memory by each processor often affects overall program performance. This is especially true with most scientific computations, where many nested loops process data that is usually stored in shared or distributed memory. If references are repeatedly made to array elements placed in external memory, the amount of time spent accessing the external data will be greater than the time required to move the data into local memory and to access it there. This is especially true if block transfer instructions are used to copy data from external to local memory. This thesis develops algorithms to generate instructions that make local copies of external data, and provides results of experiments on a parallel machine.

These algorithms are based on the *tessellation process* which divides the set of d-dimensional array elements referenced in a nested loop of depth n into $\binom{n}{d}$ subsets. Each subset provides information about the locations of the array elements in the subset. The information from all subsets is used to determine whether all the referenced array elements form contiguous blocks, and to make local copies of external data in the case of non-contiguous blocks. The tessellation process simplifies these tasks by reducing the problem size. If the referenced array elements form contiguous blocks, external data can be localized by means of fast block transfer. If not, external data can be copied word-by-word into local memory with low overhead using the information from the tessellation process. Experimental results are presented that show the algorithms' effectiveness in localization of external data.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Parallel Architectures

Computer architectures, based on the multiplicity of instruction and data streams [23], are classified in four classes:

- Single instruction stream single data stream (SISD)
- Single instruction stream multiple data stream (SIMD)
- Multiple instruction stream single data stream (MISD)
- Multiple instruction stream multiple data stream (MIMD)

The SISD is the architecture of traditional sequential computers. The MISD is so abstract that no real machine has been developed [41]. Most parallel machines built today fall into the other two categories: either the SIMD or MIMD class. The SIMD machines, called array processors because most of them are used in processing arrays of data, consist of an array of processors under the general supervision of a front-end processor that is usually a sequential machine. The front-end processor executes scalar operations, while the rest of the processors handle the vector operations. The front-end processor issues commands that cause the array processors to operate on different data simultaneously. Illiac IV [12], Goodyear MPP [29] and the Connection Machine [35] are all in the SIMD class. However, many contemporary parallel computers

can be found in the MIMD class which consists of several SISD class machines that may be coupled, either tightly or loosely. Processors in an MIMD machine execute different computations at the same time, communicating with each other through shared memory or by some other communication protocols. In this thesis, the MIMD machine is the target machine and thus the terms "parallel computer" and "MIMD machine" are used interchangeably.

A memory hierarchy is built into many parallel computer systems. The hierarchy exists because it is often necessary to coordinate the different processing speeds between processors and memory modules, especially in high speed supercomputers. Major memory components in the hierarchy are the (scalar) registers, vector registers, cache memory, local memory and shared memory; some of them are hierarchically organized according to their processing speed, i.e., from fast and low level memory to slow and high level memory. These memory components are managed by hardware or software. Cache memory is generally managed by hardware, whereas the other components are manipulated by software. In fact, registers are generally handled by the compiler at the machine instruction level, and local and shared memories are controlled by the programmer through variable declaration or dynamic allocation features that are provided by high level programming languages. Therefore programming in a high level language is mainly concerned with shared and local memory allocation, but not with the register allocation problem, which is the job of compilers [1].

Parallel computer systems are characterized by the structure of the memory hierarchy, which produces the following classification [44]:

- Shared memory system
- Distributed memory system
- Hybrid system

A shared memory system has a global memory at the highest level in the hierarchy. Shared memory is accessible to all processors; therefore, processors can communicate with each other by using a location in the shared memory. It usually consists of different parts called banks. For example, there are 128 banks in Cray-2 [17]. This bank architecture allows different processors to access the shared memory at the same time, which makes data interleaving (see [41] for example) possible for better performance, provided that the processors do not try to access the same memory bank simultaneously. Shared memory and processors are connected via a bus or switch network. For example, the Alliant FX/8 has a bus connected shared memory [6]; the NYU Ultracomputer uses a type of multistage interconnection network, called an Omega network, to connect between shared memory modules and processors [30]. Both systems provide a logically complete connection between memory banks and processors. In such an organization, all shared data can be stored in shared memory from which parallel processes fetch the data required for computation. However, this type of organization has its own drawbacks, particularly when all processors issue requests for data in shared memory at nearly the same time. Then, the connections will most likely suffer either from bus saturation or from hot spot contention [60] [51]. To overcome these potential problems, the memory system is organized with either cache or local memory. For example, the ETA-10 has 32M bytes of local memory for each processor [20], and the Cray-2 has 16K 64-bit words of local memory for each processor [66]. The Alliant has a shared cache, and the IBM 3090/VF provides a cache memory for each processor [67].

A distributed memory system, sometimes called a message passing system, does not have a global shared memory. Instead, each processor has its own local memory that is not accessible to any other processor. All processors are interconnected via a network through which processors can communicate. In this architecture, all shared data are distributed over memories each of which is exclusively accessed by its partner

processor. Running parallel processes, processors generate and send a message to the processor that has the required data, and wait for the response. The access time to a data in a remote processor depends on the network traffic and the length of the path between processors. On the average, the access time is less than $O(\log_2 n)$ in a hypercube network, $O(\log_2 n)$ in a multistage interconnection network, and $O(n)$ for a ring network. The ring network is used by the CDC Cyberplus [22], mainly because of its simplicity. Since the access time largely depends on the network topology, however, a hypercube network is widely chosen for better performance, in spite of its hardware complexity. Notable examples of machines using this network are the Intel iPSC [31], Ametek S-14 [7], NCUBE/10 [58], and FPS-T [34].

A hybrid system has properties of both shared and distributed memory systems. Each processor retains its own local memory, while the set of all local memories forms a single shared memory. For example, the BBN Butterfly GP1000 [11] consists of up to 256 processors, and each processor has 4M bytes memory module local to itself. Here all local memory modules serve as a shared memory that is interconnected via a switch network. (See Appendix A for details.) Another example is the IBM RP3 [59]. The RP3 has up to 512 processor/memory elements (PME) that are connected via an Omega-network as defined by Lawrie [50]. Each PME contains 2M or 4M bytes of memory. Part of the memory in each PME is allocated to global memory, the rest being used as local memory which is rapidly accessible by its partner processor. The boundary between global and local memory is configured at run time. Therefore, the RP3 can be treated as a shared memory system if the whole memory in each PME is allocated to global memory, as a distributed memory system if the whole memory in each PME is allocated to local memory, or as a mixture of both. In a hybrid architecture, programs can be coded as if the memory system were shared; however, the performance behavior is still similar to that of distributed memory system.

When a processor is running a certain set of parallel processes concurrently with other processors, it may require the data stored in local as well as non-local memory. Since a processor and local memory constitute a sequential processor, the data can be accessed very quickly if it is stored in local memory. However, if it happens to be in non-local (i.e., in external) memory, it obviously must traverse a longer path between external and local memory. Further, as discussed above, external data accesses can easily result in bus saturation or hot spot contention, further degrading system performance. Therefore, it is very important to utilize local memory. This thesis thus will only consider parallel computer systems where processors are equipped with local memories.

1.2 Parallel Programming

There are three different approaches in parallel programming:

1. An extended language approach
2. A portable parallel programming environment approach
3. A portable parallel programming language approach

In an extended language approach, there are two types of extensions. One type is an extension incorporating various library procedures for generating and activating parallel processes, or for passing message between processors. Examples of this type include Concentrix C [4] for the Alliant FX, Uniform System C [10] and Uniform System Fortran [9] for the Butterfly GP1000, Extended C for the Balance and iPSC [43], and Multitasking Fortran for the Cray [43]. The other type uses compiler directives. Examples are Alliant FX Fortran [5], Sequent Balance Fortran, and Cray Microtasking Fortran [45]. Using compiler directives, a programmer can give the compiler an instruction indicating how to parallelize the constructs that immediately follow the

compiler directives. Extended languages are usually provided by the manufacturers of parallel machines. Because parallel programming tools provided by manufacturers are specific to a particular architecture, writing a parallel program with them is often machine dependent. Therefore, developing parallel code is comparable to programming in a low level language for sequential computers [52]. As a result, programming styles vary a great deal even for the same class of parallel systems.

Better alternatives are found in the second and third approaches to parallel programming, both of which emphasize machine independent abstractions. A portable parallel programming environment approach provides an environment that does not particularly depend on machine characteristics, because machine dependences are hidden by means of abstractions. The Force [42], which was developed for shared memory systems, uses macro definitions to hide machine dependences. Macros are divided into several classes according to such features required for parallel execution as shared/private variable declarations, parallel executable loops, synchronizations, and so on. A Minimalist approach [54], which is efficiently implemented on the Denelcor HEP, a shared memory system, embodies Hoare's monitor concept [36]. Machine dependences are encapsulated in macros that are the lowest level in the abstraction. All features for parallel execution are implemented with monitors that are written in macros. The SCHEDULE [19] is a program package that provides a parallel programming environment for shared memory machines. In this package, data dependences and parallel structures are specified in terms of subroutine calls to the SCHEDULE subroutines. In other words, machine dependences are buried inside a package of machine dependent subroutines. The SCHEDULE allows the same user code to run without modification on different machines with minor, but in some cases difficult, modifications of the package itself. In contrast to the above languages specifically designed for shared memory machines, Linda [28] can be executed on both shared

and distributed memory machines. Such a machine independent characteristic is realized by implementing a novel communication mechanism, called the Tuple Space, that is accessible to all processes in the distributed program. Even though this approach may provide programmers with a portable parallel programming environment, implementing the environment on various machines is not easy.

In the third approach, which is the most desirable of all, new parallel programming languages are developed that have constructs to express parallelism. Many such languages have been proposed, and are based on a number of different mechanisms. The language Communicating Sequential Processes (CSP) [37] has a simple parallel command to create a fixed number of parallel processes that have local variables. These processes can communicate with each other by means of a synchronous message passing mechanism. The language Occam [55] is modeled on Hoare's CSP [37] and designed for the Transputer of Inmos, a micro processor which is often used in distributed machines. In Ada [26], parallelism is based on sequential processes, called tasks, that can run in parallel and communicate through the rendezvous mechanism. Concurrent C [27] is an extension of the C language [46] with additional features for distributed programming based on Ada's rendezvous model. Whereas CSP, Occam, Ada and Concurrent C are designed for distributed machines, Concurrent Pascal [33] is designed for shared memory machines, in that communications between its parallel processes are done through shared memory. Blaze [56] is a scientific parallel programming language that has a Pascal-like syntax, but its procedure invocation mechanism is similar to functional languages. Cedar Fortran [32] was developed for the Cedar machine [48] whose architecture provides two levels of loop parallelism: cluster loop and spread loop parallelism. However, none of these languages is popularly used, probably because they all tend to incorporate various machine dependent mechanisms in one way or another.

Given a parallel machine, an efficient parallel language program can be written by a competent programmer. Achieving the same efficiency may not be possible, however, when the program is run on another machine, unless the program is modified significantly to utilize the different hardware characteristics of the new host machine [44]. This is illustrated in [45] by comparing 12 programs to compute π written in 12 Fortran dialects for parallel machines.

In an effort to make a language widely acceptable for parallel programming, like Fortran 77 for sequential programming, PCF Fortran [64] has been developed and is recommended by the Parallel Computing Forum of several manufacturers and user communities. Basically, the language is an extension of Fortran 77 with a set of primitives added in order to program shared memory machines. Since Fortran is often called for in scientific programming, its parallel equivalent, PCF Fortran, is also expected to be gradually put into use for scientific parallel programming.

There are five levels of parallelism in parallel program execution [40]:

- Level 1: Independent jobs and programs
- Level 2: Job steps and related parts of programs
- Level 3: Routines, subroutines and co-routines
- Level 4: Loops and iterations
- Level 5: Statements and instructions

The higher the level, the finer the granularity (the unit of parallelism). Parallelism at level 1, commonly known as multiprogramming, is realized in traditional single processor systems. That of level 2 may be easily found, for example, in compilations of many procedures being carried out in parallel. Parallelism at level 3 is supported by extended languages with libraries for parallel programming. At level 5, parallelism is also being implemented in various machines, the CDC-6600 being a prominent

example, with look-ahead techniques using multiple functional units. Parallelisms at levels 1 and 2 are the main concerns of the operating system; that at level 5 is a task for hardware functional units; and expressing parallelism at level 3 largely relies on the skill of programmers and on general algorithm development. But there is still a great deal to be resolved at level 4, which therefore remains one of the major research areas in parallel programming.

Most parallel programming languages have similar constructs to declare parallelism at the level of loops, such as `forall` of Blaze, `cdoall` and `sdoall` of Cedar Fortran, `parallel do` and `spread do` of PCF Fortran, `parallel loop` of IBM Fortran for the 3090 series, `par for` of Occam, `scheduled do` of the Force, compiler directives of Alliant Fortran and Cray microtasking, and so on. The syntax and semantics of these constructs are very similar. Therefore, much research has focused on parallelism in loops, as described in detail in the next section. In this thesis, we also focus on parallelism at the level of loops, and use `forall` as a parallel construct to express the parallelism.

1.3 Program Transformation

For many traditional programmers, writing programs for parallel computers is more difficult than for sequential computers, not only because humans tend to think sequentially rather than concurrently, but also because there are many issues that make parallel programming inherently difficult: non-determinism, race conditions, synchronization, and scheduling, just to name a few [52].

It is also true that ever since the inception of modern electronic computers, an enormous number of very valuable programs have been written for sequential computers. Out of necessity, therefore, many automatic parallelization techniques have been

proposed in an attempt to run sequential programs on parallel machines. Such techniques are intended to detect parallelism in sequential programs and then generate corresponding parallel machine code [49] [3] [57]. Automatic parallelization compilers, such as CFT for the Cray X-MP, KAP205 [39], and KAP/S-1 [16], have also been developed. Automatic parallelization proved particularly successful for vectorization as well as for SIMD parallelism. Many parallelization methods may also be applied to program restructuring for MIMD parallelism [61].

The analysis of data dependences in the program is a very important tool used for detecting parallelism and restructuring programs. There are three types of data dependences: flow-, output-, and anti-dependence [47]. It is said that there exists a flow-dependence from a statement S_1 to S_2 if S_1 defines a variable and S_2 then uses it without defining the variable again between S_1 and S_2 in the execution sequence; an output-dependence if S_2 defines the same variable again; and an anti-dependence if S_1 first uses the variable and S_2 defines it afterward. If all the data dependences in a loop do not cross the loop iteration boundary, then the loop is said to be parallelizable. Further, in order to increase the possibility of detecting parallelism, data dependences annotated with direction or distance vectors, which are expressed in terms of the iteration space of the loops, have been proposed [73]. With the help of annotated data dependence analysis, even an unparallelizable program may become parallelizable by applying some transformations, such as *scalar expansion*, *loop interchanging* [69], and *loop skewing* [70], on the source. These transformations should not violate the data dependence rules [73] mandating that the data dependence should be forward directed in the execution sequence.

Program restructuring techniques are used to enhance the performance of programs written in parallel programming languages, whereas parallelization techniques are used to detect parallelism in sequential programs. Different restructuring techniques are applied to different architecture classes. For MIMD architecture machines,

it is important to make parallelism of as large granularity as possible, in order to reduce the system overhead in generating parallel processes. *Loop fusion* [53] is one technique used to create large granularity of parallel processes. For machines equipped with vector processors, such techniques as *vectorization* [2], *strip mining* [53], *loop fission* [57] and *loop collapsing* [62] may enhance the performance of vector processors. For machines equipped with cache memory, *loop interchanging* and *loop tiling* [71] [72] [68] can be applied to improve the data locality. Most of these transformations are based on dependence analysis [25].

Data locality refers to a program's tendency to refer to a subset of its address space, called a working set [18], during any time interval and to that subset's tendency to change its members slowly [15]. Even parallel computer systems have these properties, because each processor has its own execution stream. These properties are strong justification for building hierarchical memory systems. Specifically, virtual memory and cache memory are designed and implemented based on the properties of data locality.

1.4 Data Localization

Cache memory can be used efficiently through program transformations, as discussed in the previous section, because its effective utilization depends on the data locality of a program [63]. In fact, many parallel computers, and even sequential computers, are equipped with cache memory for improved system performance. However, there are other machines, like the Cray, ETA-10 and Butterfly GP1000, that do not have cache memory. Improving data locality alone will not significantly enhance the performance of these machines, even though there might be slight enhancement due to virtual memory effects. On these machines, local memory serves as an important memory component that affects the performance of parallel computers.

The old models of the Cray series do not have local memory, but the Cray-2 is manufactured with 16K 64-bit words of local memory per processor in place of cache memory [21]. The ETA-10 has 32M bytes of local memory per processor [40]. In machines with cache memory, the size of cache memory is usually small compared to local memory. For example, the RP3 has 32K bytes of cache memory and up to 4M bytes of local memory per processor [13]. Hence, restructuring techniques for improving data locality have been developed to reduce the size of the working set. If the working set fits into cache memory, then utilization of local memory, which is ineffective, is not needed. Sometimes, however, working set size can not be reduced to fit into cache memory by restructuring alone. In that case, cache memory generates frequent cache miss interrupts that request data from memory at a higher level of hierarchy. We can utilize local memory to improve the performance by putting it between cache memory and shared or external memory. This will reduce the time to service cache miss interrupts.

The programmer has explicit control over local memory through local variable declaration or dynamic allocation features that are provided by high level languages. It is the programmer's responsibility to use local memory efficiently. In complex problems, efficient algorithms aimed at utilizing local memory are often called for to get good performance [65]. Brewer et al. [14] developed a useful tool to help in programming parallel algorithms. It displays memory access patterns of one or two dimensional arrays that are being accessed in a program. With the information made available by using this tool, the programmer can decide which part of the data should be in local memory for better performance.

Most scientific computations store data in array structures, and can be run in parallel. On parallel machines, the data are usually shared between processors and are thus stored in shared or distributed memory, i.e., external or remote memory. If there is no data access conflict between sets of data accessed by processors, it may

```
forall i = 1, n
   for j = 1, n
      c(i, j) = 0
      for k = 1, n
         c(i, j) = c(i, j) + a(i, k) * b(k, j)
      endfor
   endfor
endforall
```

(a) Input to Localization System

```
forall i = 1, n
   Declare la(1 : n, 1 : n), lb(1 : n, 1 : n), lc(1 : n, 1 : n)
   Copy b(1 : n, 1 : n) to lb(1 : n, 1 : n)
loop
   Copy a(i, 1 : n) to la(i, 1 : n)
   for j = 1, n
      lc(i, j) = 0
      for k = 1, n
         lc(i, j) = lc(i, j) + la(i, k) * lb(k, j)
      endfor
   endfor
   Copy lc(i, 1 : n) to c(i, 1 : n)
endforall
```

(b) Output from Localization System

Figure 1: Example of Data Localization

be helpful to copy data required by a processor from external memory into its local memory and access it there, or vice versa. We call the process of defining a variable in local memory and inserting such copy operations *data localization.* For example, the input to, and the output from, the data localization system developed here are shown in Figure 1 for a matrix multiplication program.

The complete data localization system discussed in this thesis is shown in Figure 2. The system takes a parallel loop as input. It does not matter whether the parallel loop is directly coded by a programmer, is the result of a parallelizing system, or is itself restructured to improve data locality. As output, the system produces a

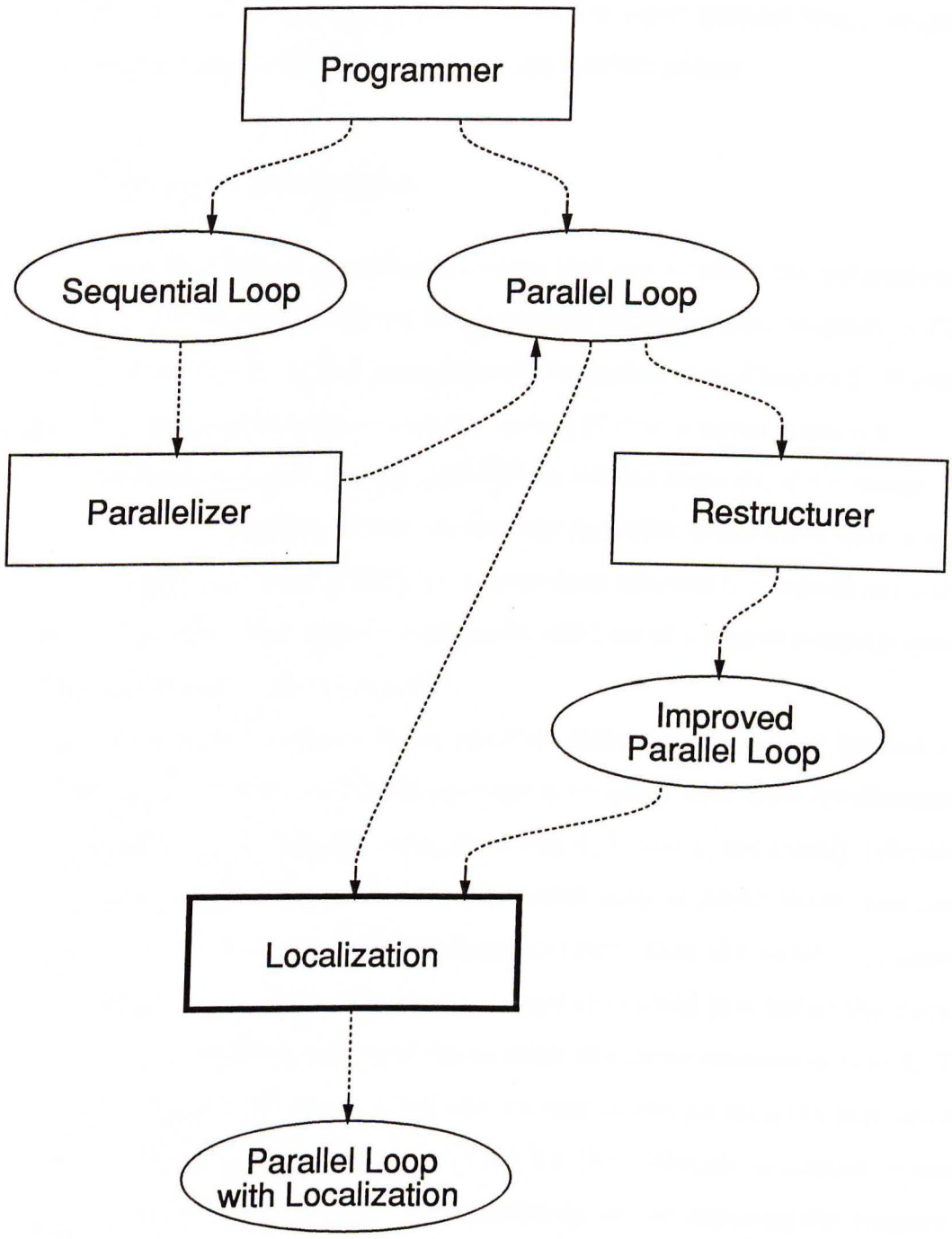


Figure 2: Loop Transformation Systems

parallel loop with local copy operations, as shown in Figure 1. The parallel loops thus generated have the same loop structure as the input parallel loop, except that data localization statements are inserted in appropriate places.

1.5 Thesis Overview

This thesis discusses a data localization system that can improve the performance of parallel loops. An example is offered by the matrix multiplication program in Figure 1 (a), whose detailed syntax and semantics will be explained in Chapter 2. When the parallel loop runs on a distributed memory system of P processors, a processor makes $\frac{1}{P}$ of array accesses to local memory and $\frac{P-1}{P}$ to remote memory, if we assume that all arrays are evenly distributed over all memory modules. Since most data accesses are made remotely, the delay caused by remote data accesses is a significant part of the computation time. The above statement is also true of a shared memory system, if all arrays are stored in shared memory.

A better version of a program is the one that utilizes local memory for fast data access. The object of this thesis is to generate a program with data localization as shown in Figure 1 (b). In (a), the array variables, a, b, and c, are mostly referenced by an assignment statement enclosed by the nested loop of depth three, and many elements of these arrays are stored in remote memory; thus the number of remote memory accesses is $O(n^3)$. However, in (b), arrays are copied just inside the `forall` loop by a row of n elements, and thus the number of remote accesses is $O(n^2)$. The primary effect of data localization is that the number of remote memory accesses are reduced from $O(n^3)$ for program (a) to $O(n^2)$ for (b). Because a remote memory access takes a much longer time than a local memory access, reducing the number of remote memory accesses is crucial for improving performance. For example, remote access takes eleven times longer than local access on a Butterfly GP1000.

The problem to be studied here can be stated as follows: How can a compiler automatically generate data localization in such a way that the resulting code is nearly optimal? We shall consider three aspects of this problem:

1. Where to put copy operations
2. How much data to copy
3. What method to use to make a local copy

Before any further discussion, we need to define the *reference mode* of a variable in statements. In a statement, we say that the reference mode of a variable is in *read mode* if its value is used, in *write mode* if its value is modified, and in *read/write mode* if its value is used and modified. In the example code, arrays a and b are used in read mode, and c is used in write mode in one statement and in read/write mode in the other statement.

With respect to the question of where to put the copy operations, the location depends on the reference mode and index function of the variable to be localized. The reference mode determines whether the copy goes before the variable reference or after (or both). The index function determines the loop level of localization. If the reference mode of a variable is in read mode, it should be localized before the computation; if in write mode, after computation; if in read/write mode, it must go both before and after the computation. Because a and b are both in read mode, they should be localized before the computation. The index function for referencing b, (k, j), is independent of `forall` loop variable i. The whole array b can be localized before starting parallel tasks. The index function for a, (i, k), does not have j, but has the term i; i.e., each parallel task accesses only one row of array a. Each parallel task needs to make a local copy of the row it may access. Therefore, a can be localized before the `for` loop j. Unlike a and b, c is used in read/write mode in the code. However, since the first usage of c is in write mode, the original values are not used.

Thus, the local array variable *lc* is used for computation and later copied into *c*. Since the index function for *c*, (i, j), has `forall` loop variable j, it is localized just after `for` loop j.

The program with localization shown in Figure 1 (b) has localization statements in the right places according to the algorithms developed in this thesis. One might ask why *a* is not localized along with *b*, even though *a* is used in read mode. Should we do so, we could still get correct results. However, we also have to take into account a performance problem that might result. Since the whole array *a* is not needed by each processor, copying unnecessary elements would waste local memory, and local memory may not be large enough to contain the whole array. If *a* is localized like *b*, all processors may access the same data almost at the same time, possibly causing network congestion and memory hot spot problems. Therefore, it is necessary to scatter localization requests throughout the parallel execution to avoid those unwanted effects on performance.

In the process of data localization, we need to copy data from external memory into local memory, or vice versa. If the data to be copied are *contiguous*, i.e., if the data are stored in consecutive areas in memory, then the data can be copied as a block, which may reduce the localization time. For example, the Butterfly GP1000 provides such a method, called *block transfer*, which can move contiguous data by blocks of up to 64 4-byte words. It takes approximately 72 μ second to move 64 4-byte words by block transfer, 7 μ second to read a 4-byte word from remote memory, and 0.38 μ second to write a 4-byte word into local memory [8]. Therefore, in localizing 64 4-byte words from remote to local memory, the block transfer is 6.56 times faster than word-by-word transfer. (See Appendix A for more details about the Butterfly GP1000.) Thus, determining whether the data are contiguous is very important if fast block transfer is to be used for better performance. Suppose that the statement

for the matrix computation of Figure 1 is replaced with

$$c(i, 2j) = c(i, 2j) + a(i, 2k) * b(2k, 2j).$$

Every other element on a row of arrays is referenced. Therefore, the data are not contiguous, and word-by-word localization will be used, provided that the time to localize and access data in local memory is less than the time to access data without localization. This case raises the question of why a whole row should not be copied using fast block transfer, if row major array allocation is assumed. For a and b, such a transfer does not cause any problem, but there may be a problem with c. If a row is copied back from a local array, lc, to the original remote array, c, then half of the original values may contain garbage values. Also, in some cases, each parallel task may modify a different set of elements on a row.

More problem definitions and assumptions for the issues of this thesis are described in Chapter 2.

The compilation steps of a data localization system are shown in Figure 3. We propose new concepts, the *tessellation process* in Chapter 3 and *q-contiguity* in Chapter 4, to solve the problems discussed above. The tessellation process takes a parallel loop as input and divides the set of d-dimensional array elements referenced in a nested loop of depth n into $\binom{n}{d}$ subsets. By dividing the referenced elements into several small pieces, the tessellation process simplifies the other procedures discussed in this thesis. Further, each subset provides information about the locations of the array elements in the subset that will be used by other parts of the algorithms. (The usage of the term "tessellation" differs here from that of Hudak et al. in their paper [38], where tessellation divides the iteration space into P partitions for P processors, and each partition is allocated to a processor.) Currently, the tessellation process is restricted to 1- and 2-dimensional arrays. It has not yet been generalized to higher

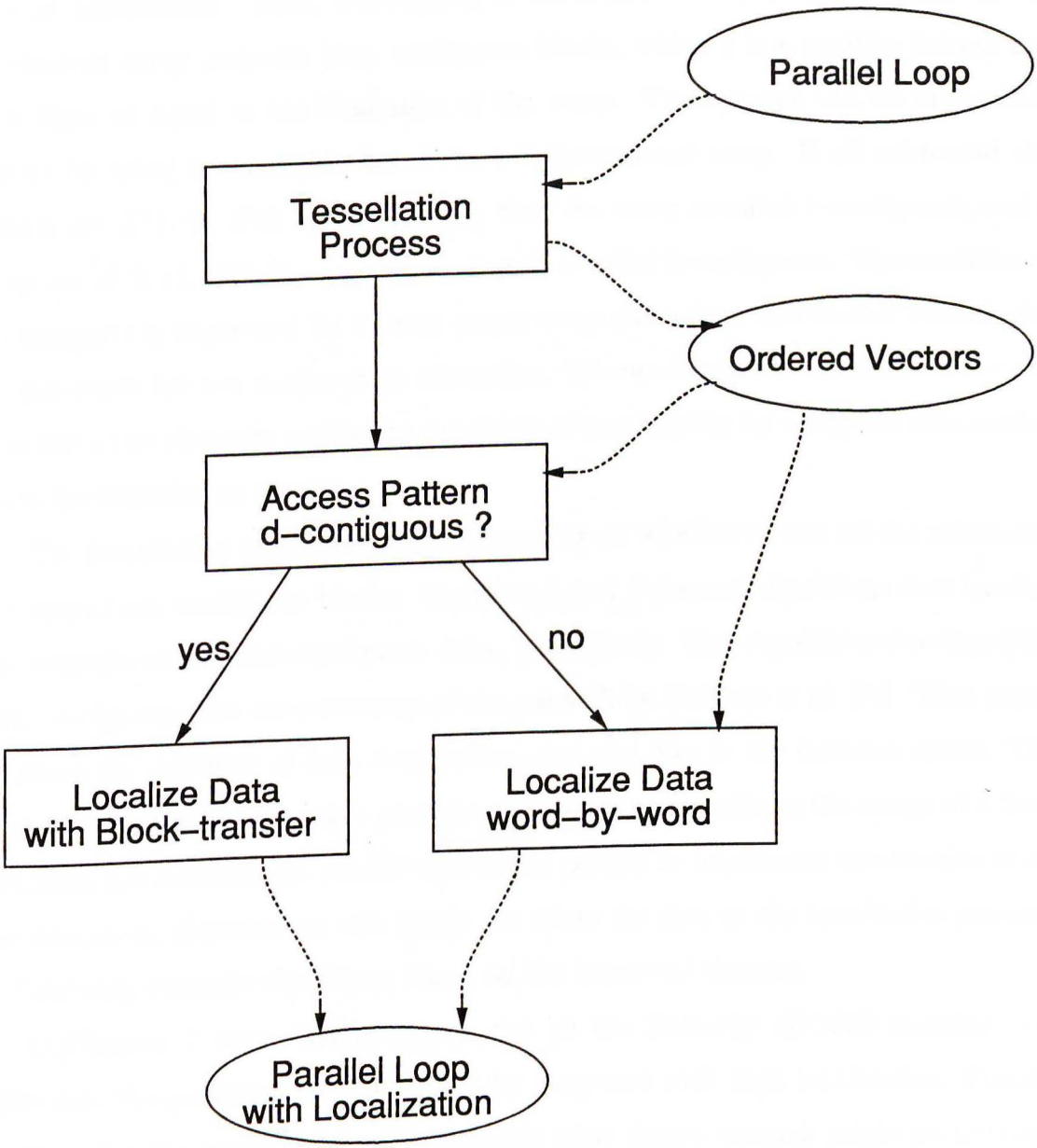


Figure 3: Data Localization System

dimensional arrays.

As mentioned before, contiguity of referenced elements is very important for efficient localization. First, *q-contiguity* is introduced to determine whether all the referenced array elements form contiguous blocks, where q is a positive integer and less than or equal to the dimension of the array. The concept can be understood easily by using an example. Let X be a 2-dimensional array. If all referenced elements are $X(1,2)$, $X(2,2)$, ..., $X(9,2)$, then the array is called 1-contiguous, and if they are $X(2,1)$, $X(2,2)$, ..., $X(2,9)$, then it is called 2-contiguous. The condition of 1-contiguity is important for column major array allocation, and that of 2-contiguity is important for row major array allocation. We can determine whether all the referenced array elements satisfy the condition of q-contiguity by using the information from the tessellation process.

The localization algorithms differ depending on whether or not all the referenced elements form contiguous blocks. Chapters 5 and 6 present algorithms that localize the contiguous and non-contiguous data, respectively. The algorithms developed for non-contiguous data are extensions of the research by Gallivan et al. [24]. That paper provides the concepts of data localization, and of a *face* in the iteration space. The data localization in that paper provides the concept of localizing the image of a face, but does not consider the precise algorithms needed to implement localization of all the referenced elements. In this thesis, we relate the face to the tessellation process, and develop concrete algorithms based on the improved theories.

In Chapter 7, some test programs, run on the Butterfly GP1000 machine, will show how the performance is improved by programs with data localization. Finally, Chapter 8 summarizes and further suggests what future research might be required to improve the algorithms developed in this thesis.

Chapter 2

Problem Definitions

Most programming languages provide loop constructs which have similar syntax and semantics. Before we start developing algorithms, we need to define the syntax and semantics of a loop construct for a clear description of algorithms, and we need to explain the assumptions of this thesis.

2.1 Loop Constructs

Most widely used programming languages have constructs, called *loops*, to denote the repetitive execution of a group of instructions, such as `do` in Fortran, `for` in Pascal and C, and so forth. Each loop consists of a *loop variable* whose value is updated on every iteration, *loop bounds* that are the lower and upper bounds of the loop variable, a *stride* which is the value added to the current value of the loop variable for the next iteration, and a *loop body* which is a group of instructions that are executed repeatedly for different values of the loop variable on every iteration. It is also possible to enclose several loops within the body of another loop, called a *nested loop*. In this case, the *nesting depth* of a loop is defined as the number of enclosing loops.

With the development of parallel computers, many programming languages such as Cedar Fortran [32] have been extended with constructs for parallel execution. Examples of those constructs used in parallel programming are `doall` and `forall`, designated using the suffix `all` with the names of loop constructs. In this thesis, `for` will be used for serial loops and `forall` will be used for parallel loops. The

```
forall i = l, u
    ⋮ initialization
loop
    ⋮ loop body
endforall
```

Figure 4: Syntax of `forall` Construct

syntax of the `forall` construct used in this thesis looks like Figure 4. *Initialization* may consist of variable declaration statements and/or executable statements, or it may be null. When there is no statement in *initialization*, the keyword `loop` may be omitted. The *loop body* contains a sequence of executable statements. When a `forall` construct is run on a parallel computer, each processor allocates variables declared in *initialization* to its local memory and executes the *loop body* accordingly. After initialization, $(u - l + 1)$ parallel tasks are generated which execute the *loop body* for $i = l, \ldots, u$ in parallel.

2.2 Assumptions

The first concern of this thesis is to develop data localization algorithms in a nested loop whose outermost loop is a parallelized `forall` loop which looks like Figure 5 (a). If the keyword `loop` is omitted, we assume that there is no initialization. Our first assumption is that *there is no synchronization problem* with the array variable we want to localize. If a variable (a scalar or an array element) needs to be synchronized, it may be accessed in write mode by several processors, or given a value by one iterate that is needed by a later iterate, and thus it is not eligible for data localization.

To simplify the initial process of algorithm development, we assume that *all occurrences of references to an array have the same index function.* (The resulting algorithms, we presume, will serve as a base for considering cases of different index

```
forall i_0 = l_0, u_0
  for i_1 = l_1, u_1
    ...
      for i_n = l_n, u_n
        X(c_10 i_0 + ... + c_1n i_n, ..., c_d0 i_0 + ... + c_dn i_n)
      endfor
    ...
  endfor
endforall
```

(a) Parallelized nested loop

```
for i_1 = l_1, u_1
  ...
    for i_n = l_n, u_n
      X(c_11 i_1 + ... + c_1n i_n, ..., c_d1 i_1 + ... + c_dn i_n)
    endfor
  ...
endfor
```

(b) Standard nested loop

Figure 5: Nested Loops

functions in future research.) For example, if the following statements are coded inside the innermost loop,

$$Y(i_1, i_1 + i_2) = Z(i_1, i_2) + X(i_1, i_2 + 1)$$
$$Z(i_1, i_2) = Z(i_1, i_2) + X(i_1, i_2 + 1)$$

then the two references to X have the same index function (i_1, i_2+1), Y has (i_1, i_1+i_2), and the three references to Z have (i_1, i_2). We handle each array independently, as if there were only one reference to each array. Thus, only *one* array reference, which represents all occurrences of references to that array, will be coded in a nested loop like Figure 5 (a).

In this example, X is used in read mode, Y in write mode, and Z in read and write mode. When all occurrences of the references to Z are represented as one reference, the reference mode is regarded as read/write mode. However, we will not specify the array reference mode, whether that be read, write, or read/write mode, because each

data localization process is quite simple and straightforward. Clearly, if the reference mode is read mode, the remote data should be copied first into local memory before they can be used. If it is write mode, then the opposite is true, in that the data will be modified in local memory and then sent back to remote memory. If it is read/write mode, both of the above operations must be carried out. For simplicity, however, from now on we will assume that *the reference mode is read mode*, unless otherwise specified.

The constant terms in index functions do not affect the algorithm development, because they imply the simple translation of referenced elements and we assume that all references to an array have the same index function. Thus they can be ignored in developing algorithms. Moreover, in the nested loop of Figure 5 (a), the loop variable i_0 of the parallelized `forall` loop behaves like a constant inside that loop. When a processor fetches one parallel task and runs it, the value of i_0 remains fixed until the process terminates. Therefore, we can also ignore the term i_0 in index functions, and omit the outermost `forall` loop, as shown in Figure 5 (b). Consequently, we assume that *there is no constant term in the index function.*

Little difficulty is presented by the stride. Many loop statements have a stride of 1, and even a loop with a stride of another value can be easily transformed to have a stride of 1 by the following normalization process:

```
for i = l, u, s                    for i = 0, ⌊(u-l)/s⌋, 1
    X(i)              ⟹               X(l + si)
endfor                             endfor
```

Therefore, we can assume that the *loop stride is 1*, and thus omit specifying the stride in the loop structures.

We also assume that *loop bounds are independent of other loop variables.* To be sure, many problems in linear algebra are programmed using nested loops whose loop bounds depend on other loop variables. However, these cases, for which different kinds of tessellation process may be developed in the future, are not considered in

this thesis.

The index functions of array references are assumed to be *linear combinations of loop variables* on integer domain and range. With the exception of sparse matrix representations, we seldom see index functions that are not linear combinations of loop variables. This assumption makes it possible to handle index functions in matrix form.

Throughout this thesis, we assume that the depth of a nested loop is n, the loop variables are $i_1, \ldots, i_n$, inwards from the outermost loop, and the dimension of the array to be localized is d. Here, the algorithms are developed for the cases $n \geq d$. If $n \geq d$, the number of accesses to the array is not less than the number of referenced elements. As the difference between n and d becomes larger, the number of accesses is much greater than that of the referenced elements, and the benefit of data localization becomes greater. In fact, the case of $n \geq d$ is common in parallel programming, and many significant applications are based on matrix multiplication types of array operations, as shown in Figure 1. The case of $n < d$ is not supported by the tessellation process, unless at least $d - n$ indices of the index function are constant.

Finally, it is assumed that *arrays are allocated in row major*, but all algorithms can be applied to the column major allocation scheme with minor modifications.

To summarize the assumptions made thus far:

- There is no synchronization problem.
- All occurrences of references to an array have the same index function.
- There are no constant terms in index functions.
- The loop stride is 1.
- Loop bounds are independent of other loop variables.

```
for i1 = 0,3
   for i2 = 0,4
      for i3 = 0,5
         X(i1 + i2, i2 + i3)
      endfor
   endfor
endfor
```

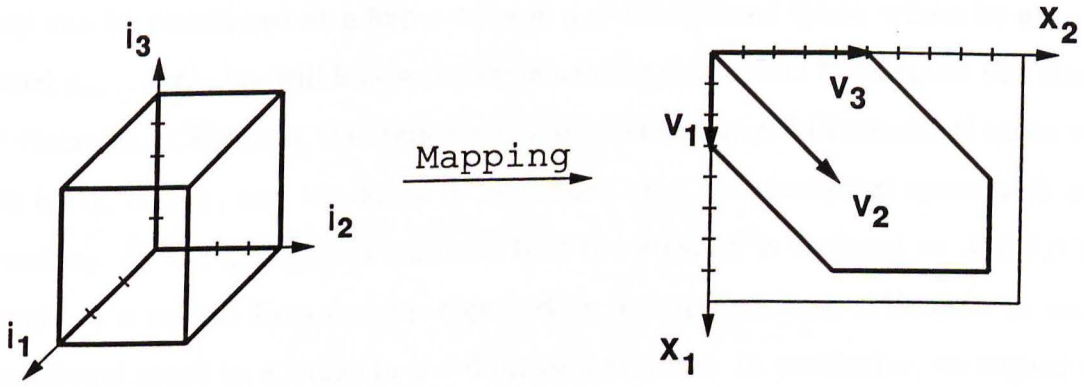


Figure 6: Nested Loop and Vector Space

- Index functions are linear combinations of loop variables.
- The depth of a nested loop is greater than or equal to the dimension of the array to be localized; i.e., $n \geq d$.
- Arrays are allocated in row major.

In Figure 5, the parallelized loop of (a) satisfies all the assumptions, and the standard nested loop of (b) has the outermost `forall` loop omitted from the code of (a). Since all parallelized loops have a `forall` loop enclosing a nested loop, we will use the standard nested loop of (b) for algorithm development and omit the outermost `forall` loop.

2.3 Nested Loops and Vector Spaces

We start developing algorithms by matching a nested loop structure with a mapping from and to vector spaces.

In a nested loop of depth n, an iteration space is defined on an n-dimensional space, where its axes are defined by the loop variables of the nested loop and its boundaries are defined by the loop bounds of the loop variables. A d-dimensional array can be considered as a hyper-cube in a d-dimensional space, where its axes are named $x_1, \ldots, x_d$. We will follow the same naming convention throughout this thesis. For example, in Figure 6, the iteration space is defined on a 3-dimensional space with axes i_1, i_2, and i_3, and the array X is defined on a 2-dimensional space with axes x_1 and x_2. In the figure, it is assumed that the array X is declared as $X(0{:}8,0{:}10)$. Therefore, a nested loop can be regarded as a mapping from a domain in an n-dimensional space to a range in a d-dimensional space. In particular, we regard the spaces as the modules Z^n and Z^d. The domain is the iteration space of the nested loop, and the bounds of loop variables specify the boundaries of the domain. The range is defined by the array elements referenced by the nested loop. The mapping function is characterized by the index function, that is, by the coefficients of the iteration variables in indices of the index function for the array reference. Then the mapping function of the nested loop of Figure 5 (b), H, can be represented in the following matrix form.

$$H = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{d1} & \cdots & c_{dn} \end{pmatrix} : Z^n \longrightarrow Z^d.$$

The matrix H of the nested loop in Figure 6 is represented as

$$H = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}.$$

The domain of each iteration variable is denoted by

$$D_k = \text{domain}(i_k) = [l_k, u_k], \quad \text{for } k = 1, \ldots, n.$$

The domain, i.e., the iteration space D, and the range R can be represented as

$$\begin{aligned} D &= D_1 \times \cdots \times D_n \subset Z^n, \\ R &= H(D). \end{aligned}$$

For example, the domain of the nested loop in Figure 6 is

$$D = D_1 \times D_2 \times D_3 = [0,3] \times [0,4] \times [0,5].$$

The domain resembles a hyper-cube in an n-dimensional space with axes named by the iteration variables. Likewise, the range is a set in a d-dimensional space with axes $x_1, \ldots, x_d$. Therefore we may think of a unit vector of the k^{th} iteration variable, i.e., e_k, as a unit vector of the i_k axis, for $k = 1, \ldots, n$,

$$e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \cdots, \quad e_n = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}.$$

To denote values of loop variables at an instant, n-tuple values for $i_1, \ldots, i_n$ are represented by a vector

$$\vec{i} = (i_1, \ldots, i_n) = \begin{pmatrix} i_1 & \ldots & i_n \end{pmatrix}^T,$$

and the corresponding n-tuple of lower and upper bounds by

$$\vec{l} = (l_1, \ldots, l_n) = \begin{pmatrix} l_1 & \ldots & l_n \end{pmatrix}^T,$$
$$\vec{u} = (u_1, \ldots, u_n) = \begin{pmatrix} u_1 & \ldots & u_n \end{pmatrix}^T.$$

For example, in Figure 6,

$$\vec{l} = (0,0,0) = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T, \quad \vec{u} = (3,4,5) = \begin{pmatrix} 3 & 4 & 5 \end{pmatrix}^T.$$

The coefficient in the r^{th} row and c^{th} column of H represents the amount of the contribution of the iteration variable i_c to the r^{th} index of the array reference. The r^{th} row is the sequence of coefficients of iteration variables in the r^{th} index of the index function for the array reference, and the c^{th} column is the sequence of coefficients of the iteration variable i_c from the indices of the index function. Let us denote the k^{th} row in the row vector form:

$$g_k = \begin{pmatrix} c_{k1} & \cdots & c_{kn} \end{pmatrix}, \quad \text{for } k = 1, \ldots, d.$$

Each index of the index function for the array reference is denoted by

$$g_k(\vec{i}) = g_k \cdot \vec{i} = c_{k1} i_1 + \cdots + c_{kn} i_n, \quad \text{for } k = 1, \ldots, d.$$

The column vectors of H are the images of unit vectors under mapping H, and they are represented as

$$h_k = H(e_k) = H \cdot e_k = \begin{pmatrix} c_{1k} \\ \vdots \\ c_{dk} \end{pmatrix}, \quad \text{for } k = 1, \ldots, n.$$

Thus, H may be represented in the following equivalent forms:

$$H = \begin{pmatrix} g_1 \\ \vdots \\ g_d \end{pmatrix} = \begin{pmatrix} h_1 & \cdots & h_n \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{d1} & \cdots & c_{dn} \end{pmatrix} : Z^n \longrightarrow Z^d.$$

The row and column vectors of the nested loop in Figure 6 are

$$g_1 = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix}, \quad g_2 = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix},$$

$$h_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad h_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad h_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Since e_k is a unit vector of iteration variable i_k, under H, h_k is the image of e_k, and $(u_k - l_k)h_k$ is the image of D_k. Let the image vector of $D_k = (u_k - l_k)h_k$, be v_k, that is,

$$v_k = (u_k - l_k)h_k.$$

For example, in Figure 6,

$$v_1 = 3\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}, \quad v_2 = 4\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \end{pmatrix}, \quad v_3 = 5\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 5 \end{pmatrix}.$$

These image vectors, v_1, v_2, and v_3, are illustrated in Figure 6.

2.4 Terminology

In this section, we define some terminology that is used very often throughout this thesis.

In a nested loop with loop variables $i_1, \ldots, i_n$, we define a *nested loop induced by*

$\{i_{j_1}, \ldots, i_{j_t}\}$ as the nested loop whose loop variables are $i_{j_1}, \ldots, i_{j_t}$, and whose other loop variables are set to a value within the loop bounds. For example, the following code is a nested loop induced by $\{i_1, i_3\}$ at $i_2 = 0$ from the nested loop of Figure 6.

```
for i1 = 0,3
    i2 = 0
    for i3 = 0,5
        X(i1 + i2, i2 + i3)
    endfor
endfor
```

In a nested loop with an array reference, the collection of the array elements referenced by the nested loop is called the *footprint* of the nested loop, and a *subfootprint* is a subset of a footprint. The *base-element* is the array element referenced by the nested loop at $\vec{i} = \vec{l}$; that is, $H(l_1, \ldots, l_n)$ is the coordinate of the base-element. The *base-space* for $\{i_{j_1} = \iota_{j_1}, \ldots, i_{j_t} = \iota_{j_t}\}$, which is also a subfootprint, is defined as the array elements referenced by the nested loop induced by a set of loop variables $\{i_{k_1}, \ldots, i_{k_s}\}$, when loop variables $i_{j_1}, \ldots, i_{j_t}$ are set to $\iota_{j_1}, \ldots, \iota_{j_t}$, respectively, within the loop bounds of those loop variables, where $t + s = n$ and $\{j_1, \ldots, j_t\} \cap \{k_1, \ldots, k_s\} = \phi$.

Chapter 3

The Tessellation Process

When a d-dimensional array variable is referenced in a nested loop of depth n as in Figure 5, the footprint of a nested loop can be divided into $\binom{n}{d}$ subfootprints. Each subfootprint is the footprint of a nested loop induced by a set of d loop variables, and the other $(n-d)$ loop variables are set to either the lower or upper bound of the loop variables. The subfootprints are mutually disjoint except at the boundaries which are the intersections of the adjacent subfootprints. (We use the term *boundary* in two ways: here, it represents the intersection of the adjacent subfootprints, and later it will be used to represent the border of the iteration space.)

For an example of a 1-dimensional array referenced in a nested loop, refer to Figure 7. Since its nesting depth is 3, there are $\binom{3}{1} = 3$ disjoint subfootprints. There are two possible ways to divide the footprint of the nested loop, as shown in (a) and (b) of the diagram. In (a), the interval BC is the footprint of the nested loop: the subinterval OA is the subfootprint for i_1=0 to 5, $i_2 = i_3 = 0$; OB for i_2=0 to 6, $i_1 = i_3 = 0$; and AC for i_3=0 to 7, i_1=5, i_2=0. In (b), the interval QR, which is the same as BC in (a), is the footprint of the nested loop: OP is the subfootprint for i_3=0 to 7, $i_1 = i_2 = 0$; OQ for i_2=0 to 6, $i_1 = i_3 = 0$; and PR for i_1=0 to 5, i_2=0, i_3=7. In both (a) and (b), the points O, A, and P are the boundary points.

For an example of a 2-dimensional array referenced in a nested loop, refer to Figure 8. Since its nesting depth is 3, there are $\binom{3}{2}$=3 disjoint subfootprints that are parallelograms. Here, too, there are two possible ways to divide the footprint of the

```
for i1 = 0,5
   for i2 = 0,6
      for i3 = 0,7
         X(i1 − i2 + i3)
      endfor
   endfor
endfor
```

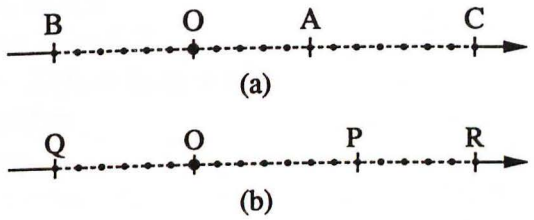


Figure 7: Tessellation of a 1-dimensional Array

nested loop, as shown in (a) and (b). In (a), the parallelogram A is the subfootprint for i_1=0 to 5, i_2=0 to 6, i_3=0; the parallelogram B for i_2=0 to 6, i_3=0 to 7, i_1=0; and the parallelogram C for i_1=0 to 5, i_3=0 to 7, i_2=6. In (b), the parallelogram P is the subfootprint for i_1=0 to 5, i_3=0 to 7, i_2=0; the parallelogram Q for i_2=0 to 6, i_3=0 to 7, i_1=5; and the parallelogram R for i_1=0 to 5, i_2=0 to 6, i_3=7. In both (a) and (b), there are three boundary lines between subfootprints.

Furthermore, there are some other important aspects in the examples offered by Figure 7 and 8. Using the notation defined in Chapter 2, we know of Figure 7 that for a 1-dimensional array,

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \end{pmatrix},$$

$$v_1 = (u_1 - l_1)h_1 = \begin{pmatrix} 5 \end{pmatrix}, \quad v_2 = (u_2 - l_2)h_2 = \begin{pmatrix} -6 \end{pmatrix}, \quad v_3 = (u_3 - l_3)h_3 = \begin{pmatrix} 7 \end{pmatrix}.$$

Clearly, in both (a) and (b), there are three vectors that span the whole footprint: namely, v_1, v_2, and v_3. In (a), v_1 is $\vec{OA}$, v_2 is $\vec{OB}$, and v_3 is $\vec{AC}$. The base-points of the subfootprints are $H(0,0,0)$ for OA and OB, and $H(5,0,0)$ for AC. In other words, the subfootprints OA, OB, and AC are $H(D_1 \times l_2 \times l_3)$, $H(l_1 \times D_2 \times l_3)$, and $H(u_1 \times l_2 \times D_3)$, respectively. Similarly, in (b), the subfootprints OP, OQ, and PR are $H(l_1 \times l_2 \times D_3)$, $H(l_1 \times D_2 \times l_3)$, and $H(D_1 \times l_2 \times u_3)$, respectively. Notice that $D_1 \times l_2 \times l_3$, $l_1 \times D_2 \times l_3$, $u_1 \times l_2 \times D_3$, $l_1 \times l_2 \times D_3$, and $D_1 \times l_2 \times u_3$ are

```
for i1 = 0,5
   for i2 = 0,6
      for i3 = 0,7
         X(i1 + i2, i2 + i3)
      endfor
   endfor
endfor
```

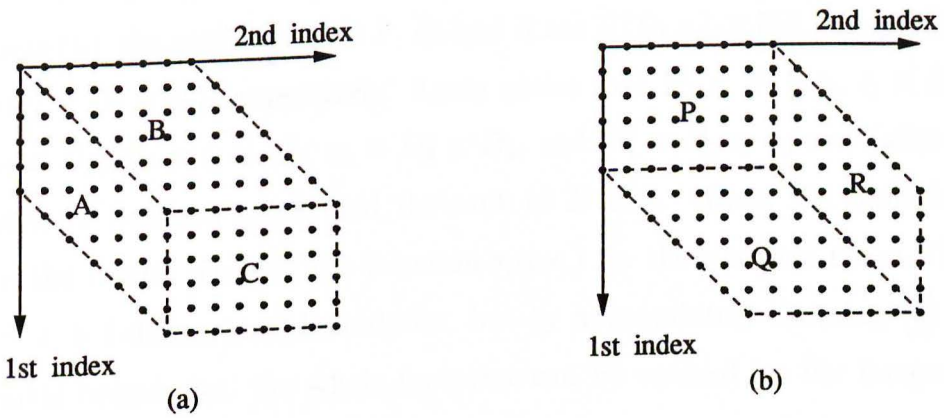


Figure 8: Tessellation of a 2-dimensional Array

1-dimensional boundaries of the iteration space, and the rank of H is 1. (Here, the term *boundary* refers to the border line of the iteration space.) In the iteration space, there are $\binom{3}{1}2^{(3-1)} = 12$ 1-dimensional boundaries, but by choosing the appropriate number of 1-dimensional boundaries — in this case, $\binom{3}{1} = 3$ — the whole footprint can be covered by the images of the chosen boundaries.

Similarly, in Figure 8, we have, for a 2-dimensional array,

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix},$$

$$v_1 = (u_1 - l_1)h_1 = \begin{pmatrix} 5 \\ 0 \end{pmatrix}, \quad v_2 = (u_2 - l_2)h_2 = \begin{pmatrix} 6 \\ 6 \end{pmatrix}, \quad v_3 = (u_3 - l_3)h_3 = \begin{pmatrix} 0 \\ 7 \end{pmatrix}.$$

In both (a) and (b), there are three vectors that span the whole footprint: namely,

v_1, v_2, and v_3. In (a), the parallelogram A is spanned by v_1 and v_2; the parallelogram B by v_2 and v_3; and the parallelogram C by v_1 and v_3. The base-points of the subfootprints spanned by two vectors are $H(0,0,0)$ for the parallelograms A and B, and $H(0,6,0)$ for the parallelogram C. In other words, the parallelograms A, B, and C are $H(D_1 \times D_2 \times l_3)$, $H(l_1 \times D_2 \times D_3)$, and $H(D_1 \times u_2 \times D_3)$, respectively. Similarly, in diagram (b), the parallelograms P, Q, and R are $H(D_1 \times l_2 \times D_3)$, $H(u_1 \times D_2 \times D_3)$, and $H(D_1 \times D_2 \times u_3)$, respectively. Again notice that $D_1 \times D_2 \times l_3$, $l_1 \times D_2 \times D_3$, $D_1 \times u_2 \times D_3$, $D_1 \times l_2 \times D_3$, $u_1 \times D_2 \times D_3$, and $D_1 \times D_2 \times u_3$ are 2-dimensional boundaries of iteration space, and the rank of H is 2. (Here, the term *boundary* refers to the border plane of the iteration space.) In the iteration space, there are $\binom{3}{2}2^{(3-2)} = 6$ 2-dimensional boundaries, but by appropriately choosing $\binom{3}{2} = 3$ 2-dimensional boundaries, the whole footprint can be covered by the images of the chosen boundaries.

The boundaries in the above two examples are called *faces* of the domain (iteration space). In this chapter, a method, called the *tessellation process*, is developed by which we can divide the range (footprint) of the nested loop given in the form of Figure 5 (b) into disjoint areas (subfootprints). The tessellation process is equivalent to the method of choosing faces, the union of whose images covers the whole footprint.

Definition 1 *Given a nested loop in the form of Figure 5 (b), let the rank of H be r. An r-dimensional boundary of the iteration space is called a* **face**, *and denoted by F.*

An example of a face is $D_1 \times \cdots \times D_r \times \eta_{r+1} \times \cdots \times \eta_n$, where $\eta_j = l_j$ or u_j for $j = r+1, \ldots, n$.

The iteration space of a nested loop is defined on an integer domain, and the index function maps the iteration space to an integer range, since the coefficients are integers. For the purposes of the tessellation process we consider in this chapter, they are assumed to be real numbers. In the following chapters, this process will be used

to divide the footprint of the nested loop, and then algorithms will be developed on an integer domain and range.

Since H, the matrix form of the index function for an array reference in a nested loop, is linear, the range of H can be represented as

$$\begin{aligned}
R &= H(D) = \begin{pmatrix} h_1 & \cdots & h_n \end{pmatrix}(D_1 \times \cdots \times D_n) \\
&= \sum_{j=1}^{n} D_j h_j \\
&= \{\sum_{j=1}^{n} a_j h_j \mid l_j \le a_j \le u_j\} \\
&= \{\sum_{j=1}^{n} (a_j h_j + l_j h_j) \mid 0 \le a_j \le u_j - l_j\} \qquad \text{(by using } v_j = (u_j - l_j)h_j\text{)} \\
&= \{\sum_{j=1}^{n} (a_j v_j + l_j h_j) \mid 0 \le a_j \le 1\} \\
&= \{\sum_{j=1}^{n} a_j v_j + H(\vec{l}) \mid 0 \le a_j \le 1\}.
\end{aligned}$$

Note, in the last expression, that the range R may be represented by a set of linear combinations of $v_1, \ldots, v_n$, with the coefficients between 0 and 1, and with a displacement $H(\vec{l})$ which is the coordinate of the base-element. This representation is the main one used in this chapter. The term $H(\vec{l})$, being a constant, will be omitted for simplicity in the following construction.

Definition 2 *Let d vectors $v_1, \ldots, v_d$ be in a d-dimensional space. A* **cone** *is defined by*

$$cone(v_1, \ldots, v_d) = \{\sum_{j=1}^{d} a_j v_j \mid 0 \le a_j\}.$$

In the definition of *cone*, only d vectors are given in a d-dimensional space; in other words, all vectors are given in a d-dimensional half space. Therefore, we can think of a *cone* as restricted to a half space. Although a *cone* can be defined on any dimensional

spaces, in the light of the complexity of the problem in higher dimensional spaces, as well as our belief that the 1- and 2-dimension cases cover most examples, our present algorithms will only cover the range of the index functions of 1- or 2-dimensional spaces.

The tessellation process provides the information about the locations of the array elements referenced in a nested loop. The locations are represented in terms of the vectors, $v_k = (u_k - l_k)h_k$ for $k = 1, \ldots, n$, which are characterized by the column vectors of the mapping matrix and the loop bounds. The following definition defines an ordering of those vectors in the 1- or 2-dimensional space. The ordering is used to represent the locations of the subfootprints.

Definition 3 *Let n non-zero vectors $v_1, \ldots, v_n$ be in a d-dimensional half space, where $v_j = (u_j - l_j)h_j$. First, reorder $v_1, \ldots, v_n$ according to the following criteria: for any pair of $i, j \in \{1, \ldots, n\}$,*

1. *v_i precedes v_j if the angle between v_i and the x_1-axis is less than the angle between v_j and the x_1-axis.*
2. *When the angles are equal, v_i precedes v_j if $\|h_i\| < \|h_j\|$, where $\|\cdot\|$ is the Euclidean norm.*
3. *When the norms are equal, v_i precedes v_j if $i < j$.*

Let $v_{k_1}, \ldots, v_{k_n}$ be a new ordering. Then, v_{inside} is defined for $d = 1$ or 2 as follows:

- *When $d = 1$, for any integer $t \in \{1, \ldots, n\}$, define $v_{inside}^{(k_t)}$ by*

$$v_{inside}^{(k_t)} = v_{k_1} + \cdots + v_{k_{t-1}}.$$

- *When $d = 2$, for any two integers $b, t \in \{1, \ldots, n\}, b < t$, define $v_{inside}^{(k_b, k_t)}$ by*

$$v_{inside}^{(k_b, k_t)} = v_{k_{b+1}} + \cdots + v_{k_{t-1}}.$$

For example, when $d = 1$, let the index function be

$$H = \begin{pmatrix} 2 & 1 & 3 & 2 \end{pmatrix}.$$

Since the norms of the column vectors are

$$\|h_1\| = \|h_4\| = 2, \quad \|h_2\| = 1, \quad \|h_3\| = 3,$$

vector v_2 can move to the first place, vectors v_1 and v_4 to the next two places, and v_3 to the last place. Since the norms of v_1 and v_4 are equal, according to the subscripts, v_1 is placed on the second place. Now all vectors are completely reordered as (v_2, v_1, v_4, v_3). Then

$$\begin{aligned} v_{inside}^{(2)} &= 0, & v_{inside}^{(4)} &= v_2 + v_1, \\ v_{inside}^{(1)} &= v_2, & v_{inside}^{(3)} &= v_2 + v_1 + v_4. \end{aligned}$$

The vector $v_{inside}^{(t)}$ is the sum of the vectors that precede v_t in the reordering.

For example, when $d = 2$, let the index function be

$$H = \begin{pmatrix} 2 & 1 & 0 & 1 \\ 2 & 0 & 1 & 1 \end{pmatrix}.$$

Then the angles between the x_1-axis of the range space and v_j vectors are

$$v_1, v_4 : \frac{\pi}{4}, \quad v_2 : 0, \quad v_3 : \frac{\pi}{2}.$$

The norms of the column vectors h_1 and h_4 are

$$\|h_1\| = (2^2 + 2^2)^{1/2} = \sqrt{8}, \quad \|h_4\| = (1^2 + 1^2)^{1/2} = \sqrt{2}.$$

The ordering should be (v_2, v_4, v_1, v_3). Then

$$\begin{aligned} v_{inside}^{(2,4)} &= 0, & v_{inside}^{(2,1)} &= v_4, \\ v_{inside}^{(4,1)} &= 0, & v_{inside}^{(4,3)} &= v_1, \\ v_{inside}^{(1,3)} &= 0, & v_{inside}^{(2,3)} &= v_4 + v_1. \end{aligned}$$

The vector $v_{inside}^{(b,t)}$ is the sum of the vectors that are inside $cone(v_b, v_t)$.

As we can see in the above example, the notation is a bit too complex because of the two level subscript indices. In order to avoid using double subscripts, all vectors $v_1, \ldots, v_n$ are assumed to have already been properly ordered according to the reordering scheme of Definition 3.

Definition 3 uses a half space, but we may extend it to the full space as well. When n vectors $v_1, \ldots, v_n$ are given in the range of a d-dimensional space, and the vectors are properly ordered as described in Definition 3, v_{inside} for $d = 1$ or 2 is defined as follows:

- When $d = 1$, for any integer $t \in \{1, \ldots, n\}$, collect vectors that have the same direction as v_t, and let them be $v_{k_1}, \ldots, v_{k_s}$. Then $v_{k_1}, \ldots, v_{k_s}$ are in the same half space. Therefore, Definition 3 can be applied to those vectors.

- When $d = 2$, for any two integers $b, t \in \{1, \ldots, n\}, b < t$, we can divide the space into the two half spaces so that the two vectors v_b and v_t are in the same half space, and then apply Definition 3 to the vectors in that half space.

Definition 4 *Let n non-zero vectors $v_1, \ldots, v_n$ be in a d-dimensional half space, where $v_j = (u_j - l_j)h_j$. Assume that they are properly ordered as described in Definition 3. Let a vector v be a linear combination of $v_1, \ldots, v_n$ with coefficients in $[0, 1]$. Then, v_{diff} is defined for $d = 1$ or 2 as follows:*

- *When $d = 1$, for any integer $t \in \{1, \ldots, n\}$, define the difference vector between*

v and $v^{(t)}_{inside}$ by

$$v^{(t)}_{diff} = v - v^{(t)}_{inside}.$$

- *When $d = 2$, for any two integers $b, t \in \{1, \ldots, n\}, b < t$, define the difference vector between v and $v^{(b,t)}_{inside}$ by*

$$v^{(b,t)}_{diff} = v - v^{(b,t)}_{inside}.$$

Using the above definitions, we may start building the tessellation process at last. Because of the different notations for the 1- and 2-dimensional cases, however, the tessellation process for each case will be developed separately.

3.1 Tessellation of a 1-dimensional Array

In this section, the tessellation process is developed for a nested loop that has a 1-dimensional array reference in the loop body of the innermost loop. If the rank of the index function is 0, the index function does not include any loop variable; i.e., the nested loop accesses only one element of the array. In this section, therefore, it is assumed that the rank of the index function is 1 (the same as d).

When n non-zero vectors, $v_1, \ldots, v_n$, are given in a 1-dimensional half space, then the 1-dimensional space is divided into two disjoint cones at the boundary point $v^{(n)}_{inside}$. For any arbitrary vector v that is a linear combination of the n vectors with coefficients between 0 and 1, the head of a vector v is located inside one of those two cones. The following theorem formalizes this concept.

Theorem 1 *Let n non-zero vectors $v_1, \ldots, v_n$, which are properly ordered, be in a 1-dimensional half space, where $v_j = (u_j - l_j)h_j$. Then for any vector $v \in R$, the vector v can be represented as one of the following two cases:*

1. $v = \alpha_n v_n + v_{inside}^{(n)}$, *if* $v_{diff}^{(n)} \in S_1 = cone(v_n)$

2. $v = \beta_1 v_1 + \cdots + \beta_{n-1} v_{n-1}$, *if* $v_{diff}^{(n)} \in S_2 = cone(-v_{inside}^{(n)})$

where $\alpha_j, \beta_j \in [0, 1]$.

Proof Note that S_1 and S_2 have a point in common, when $\alpha_n = 0$ and $\beta_1 = \cdots = \beta_{n-1} = 1$. If the boundary point is excluded, S_1 and S_2 may be regarded as a partition of a 1-dimensional space. The boundary point can be handled with any set in which it is included.

Case 1: When $v_{diff}^{(n)} \in S_1$,

$$v_{diff}^{(n)} = v - v_{inside}^{(n)} = \alpha_n v_n, \quad \text{where } 0 \leq \alpha_n,$$

$$v = \alpha_n v_n + v_{inside}^{(n)}.$$

In this representation, $\alpha_n \in [0, 1]$, since otherwise v is not a member of R.

Case 2: When $v_{diff}^{(n)} \in S_2$,

$$v_{diff}^{(n)} = v - v_{inside}^{(n)} = \beta(-v_{inside}^{(n)}), \quad \text{where } \beta \in [0, 1],$$

$$v = \beta(-v_{inside}^{(n)}) + v_{inside}^{(n)} = (1 - \beta) v_{inside}^{(n)}.$$

In this representation, $(1 - \beta) \in [0, 1]$. Hence the vector v may be rewritten as

$$v = \beta_1 v_1 + \cdots + \beta_{n-1} v_{n-1}, \quad \text{where } \beta_j \in [0, 1].$$

□

For an example of the above theorem, see Figure 9. When

$$v_1 = \vec{OA}, \quad v_2 = \vec{AB}, \quad v_3 = \vec{BC}, \quad v_4 = \vec{CD}$$

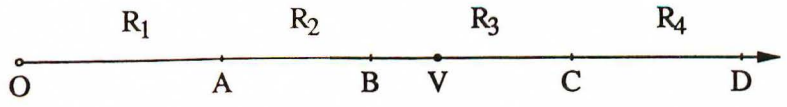


Figure 9: 1-dimensional Tessellations

are given, the head of an arbitrary linear combination of those four vectors is to be between points O and D. Let v be such a linear combination,

$$v = a_1v_1 + a_2v_2 + a_3v_3 + a_4v_4 = \vec{OV}, \quad \text{where } a_j \in [0,1].$$

Then

$$cone(v_4) = \vec{CD}, \qquad cone(-v^{(4)}_{inside}) = cone(-(v_1+v_2+v_3)) = \vec{CO}.$$

The location of the head of v is determined by the direction of the vector

$$v^{(4)}_{diff} = v - v^{(4)}_{inside} = v - (v_1+v_2+v_3) = \vec{CV}.$$

In the diagram, it is inside $cone(-v^{(4)}_{inside})$.

Applying Theorem 1 recursively for n non-zero vectors in a 1-dimensional half space, we can divide a 1-dimensional space into n disjoint areas except at the $(n-1)$ boundaries $v^{(2)}_{inside}, \ldots, v^{(n)}_{inside}$. Therefore, the set of linear combinations of n vectors with coefficients between 0 and 1 can be divided into n disjoint subsets.

Theorem 2 *Let n non-zero vectors $v_1, \ldots, v_n$, which are properly ordered, be in a 1-dimensional half space, where $v_j = (u_j - l_j)h_j$. For any $t \in \{1, \ldots, n\}$, let*

$$R_t = \{\alpha_t v_t + v^{(t)}_{inside} \mid \alpha_t \in [0,1]\}$$

be the area spanned by v_t with coefficients in [0,1] at the point $v_{inside}^{(t)}$. Then

$$R = \bigcup_{t=1}^{n} R_t.$$

Proof To prove $R \subseteq \bigcup_{t=1}^{n} R_t$, it must be shown that

$$\forall v = \sum_{j=1}^{n} a_j v_j \in R, \quad \exists t \in \{1, \ldots, n\}, \alpha_t \in [0,1] \text{ such that } v = \alpha_t v_t + v_{inside}^{(t)}.$$

Initially $v = \sum_{j=1}^{n} a_j v_j$. From Theorem 1, the vector v can be represented as one of the following two possible linear combinations:

1. $v = \alpha_n v_n + v_{inside}^{(n)}$, if $v_{diff}^{(n)} \in S_1 = cone(v_n)$
2. $v = \beta_1 v_1 + \cdots + \beta_{n-1} v_{n-1}$, if $v_{diff}^{(n)} \in S_2 = cone(-v_{inside}^{(n)})$

where $\alpha_j, \beta_j \in [0,1]$.

If v is as in case 1, then $v \in R_n$.

If v is as in case 2, the vector v is represented with $(n-1)$ vectors. The number of vectors is decreased by one. We can repeat this process until for some value of t, the represented satisfies case 1; i.e.,

$$v = \alpha_t v_t + v_{inside}^{(t)} \in R_t.$$

Eventually, this process terminates with $t = 1$. Therefore, we can find $t \in \{1, \ldots, n\}$ such that $v \in R_t$.

To prove $R \supseteq \bigcup_{t=1}^{n} R_t$, let

$$v = \alpha_t v_t + v_{inside}^{(t)} = \alpha_t v_t + \sum_{j=1}^{t-1} v_j \in R_t.$$

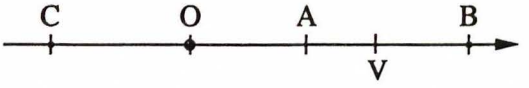


Figure 10: Dividing 1-dimensional Space into Half Spaces

Setting

$$a_1 = \cdots = a_{t-1} = 1, \quad a_t = \alpha_t, \quad a_{t+1} = \cdots = a_n = 0,$$

we can say that $v \in R$.

□

So far, we have proved the case when all vectors are in one half space. The next theorem generalizes the above theorem to the full space.

Theorem 3 *Let n non-zero vectors $v_1, \ldots, v_n$, which are properly ordered, be in a 1-dimensional space, where $v_j = (u_j - l_j)h_j$. Then*

$$R = \bigcup_{t=1}^{n} R_t.$$

Proof For an arbitrary vector $v = \sum_{j=1}^{n} a_j v_j \in R$, there exists a half space such that all vectors in the half space are in the direction of the vector v and all vectors in the other half space are in the opposite direction to the vector v. Then the vector v can be represented with the vectors in the half space containing the vector v. Finally, by applying Theorem 2, the proof is complete.

□

For example, refer to Figure 10. The point O is the origin, $v_1 = \vec{OA}$, $v_2 = \vec{AB}$, $v_3 = \vec{OC}$, and $v = \vec{OV}$. Then the vectors v_1 and v_2 are in the direction of the vector v, but the vector v_3 is in the opposite direction. Therefore, the vector v can be represented with a linear combination of v_1 and v_2, which are in the half space to the right of the point O.

From the process of the construction of the R_t, it is clearly seen that the R_t are

mutually disjoint except at the boundary points when the coefficient α_t is 0 or 1. It may also be noted in the above theorem that any linear combination of vectors $v_1, \ldots, v_n$ falls into one of R_t. This implies that the images of $\binom{n}{1}$ faces cover the whole image of the domain under H, since $v_k = (u_k - l_k)h_k$ is an image of $(u_k - l_k)e_k$ under H. When all vectors $v_1, \ldots, v_n$ are properly ordered as described in Definition 3, the faces are

$$\eta_1 \times \cdots \times \eta_{k-1} \times D_k \times \eta_{k+1} \times \cdots \times \eta_n, \quad \text{for } k = 1, \ldots, n$$

$$\text{where } \eta_j = \begin{cases} u_j & \text{if } v_j \text{ is part of } v_{inside}^{(k)} \\ l_j & \text{otherwise} \end{cases}$$

The images of the faces, $H(\eta_1 \times \ldots \times \eta_{k-1} \times D_k \times \eta_{k+1} \times \ldots \times \eta_n) = R_k$, for $k = 1, \ldots, n$, are mutually disjoint except at the boundaries. The images of the faces are called *tessellations* for the 1-dimensional space, and the process of constructing tessellations is the *tessellation process.*

As a final example with the case of a 1-dimensional array, consider the nested loop of Figure 7. From the code,

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \end{pmatrix},$$

$$v_1 = (u_1 - l_1)h_1 = \begin{pmatrix} 5 \end{pmatrix}, \quad v_2 = (u_2 - l_2)h_2 = \begin{pmatrix} -6 \end{pmatrix}, \quad v_3 = (u_3 - l_3)h_3 = \begin{pmatrix} 7 \end{pmatrix}.$$

According to the reordering scheme described in Definition 3, the vectors should be reordered as

$$(v_1, v_3, v_2).$$

Since v_1 and v_3 have the same direction, v_1 and v_3 are in the same half space, but v_2

is in the other half space. From the reordering,

$$v_{inside}^{(1)} = 0, \quad v_{inside}^{(3)} = v_1, \quad v_{inside}^{(2)} = 0,$$

and the faces for tessellations are

$$D_1 \times l_2 \times l_3, \quad l_1 \times D_2 \times l_3, \quad u_1 \times l_2 \times D_3.$$

So the figure of the tessellations is like Figure 7 (a).

With the tessellation process, we can locate all the array elements referenced in a nested loop using the concept of $v_{inside}^{(t)}$.

3.2 Tessellation of a 2-dimensional Array

In this section, the tessellation algorithm is developed for a nested loop that has a 2-dimensional array reference in the loop body of the innermost loop. If the rank of the index function is 0, the index function does not include any loop variables; i.e., the nested loop accesses only one element of the array. If the rank is 1, the problem may be reduced to the 1-dimensional case. In this section, therefore, it is assumed that the rank of the index function is same as $d = 2$.

When n non-zero vectors, $v_1, \ldots, v_n$, are given in a 2-dimensional half space, the 2-dimensional space can be divided into three disjoint cones, except at the boundaries of v_1, v_n, and $-v_{inside}^{(1,n)}$, at the head of $v_{inside}^{(1,n)}$. For any arbitrary vector v that is a linear combination of the n vectors with coefficients between 0 and 1, the head of a vector v is located inside one of those three cones. The following theorem formalizes this concept.

Theorem 4 *Let n non-zero vectors $v_1, \ldots, v_n$, which are properly ordered, be in a 2-dimensional half space, where $v_j = (u_j - l_j)h_j$. Then for any vector $v \in R$, the*

vector v can be represented as one of the following three cases:

1. $v = \alpha_1 v_1 + \alpha_n v_n + v_{inside}^{(1,n)}$, *if* $v_{diff}^{(1,n)} \in S_1 = cone(v_1, v_n)$
2. $v = \beta_1 v_1 + \cdots + \beta_{n-1} v_{n-1}$, *if* $v_{diff}^{(1,n)} \in S_2 = cone(v_1, -v_{inside}^{(1,n)})$
3. $v = \gamma_2 v_2 + \cdots + \gamma_n v_n$, *if* $v_{diff}^{(1,n)} \in S_3 = cone(v_n, -v_{inside}^{(1,n)})$

where $\alpha_j, \beta_j, \gamma_j \in [0, 1]$.

Proof Note that each pair of S_1, S_2, and S_3 has a vector in common. If the boundary vectors are excluded, S_1, S_2, and S_3 may be regarded as a partition of a 2-dimensional space. The boundary vectors can be handled with any set in which they are included.
Case 1: When $v_{diff}^{(1,n)} \in S_1$,

$$v_{diff}^{(1,n)} = v - v_{inside}^{(1,n)} = \alpha_1 v_1 + \alpha_n v_n, \quad \text{where } 0 \leq \alpha_1, \alpha_n,$$

$$v = \alpha_1 v_1 + \alpha_n v_n + v_{inside}^{(1,n)}.$$

In this representation, $\alpha_1, \alpha_n \in [0, 1]$; otherwise, v is not a member of R.
Case 2: When $v_{diff}^{(1,n)} \in S_2$,

$$v_{diff}^{(1,n)} = v - v_{inside}^{(1,n)} = \beta_1 v_1 + \beta(-v_{inside}^{(1,n)}), \quad \text{where } 0 \leq \beta_1, \beta,$$

$$v = \beta_1 v_1 + \beta(-v_{inside}^{(1,n)}) + v_{inside}^{(1,n)} = \beta_1 v_1 + (1 - \beta) v_{inside}^{(1,n)}.$$

In this representation, $\beta_1, (1 - \beta) \in [0, 1]$. Hence the vector v can be rewritten as

$$v = \beta_1 v_1 + \cdots + \beta_{n-1} v_{n-1}, \quad \text{where } \beta_j \in [0, 1], \quad \text{for } j = 1, \ldots, n-1.$$

Case 3: When $v_{diff}^{(1,n)} \in S_3$, the vector v can be

$$v = \gamma_2 v_2 + \cdots + \gamma_n v_n, \quad \text{where } \gamma_j \in [0, 1], \quad \text{for } j = 2, \ldots, n.$$

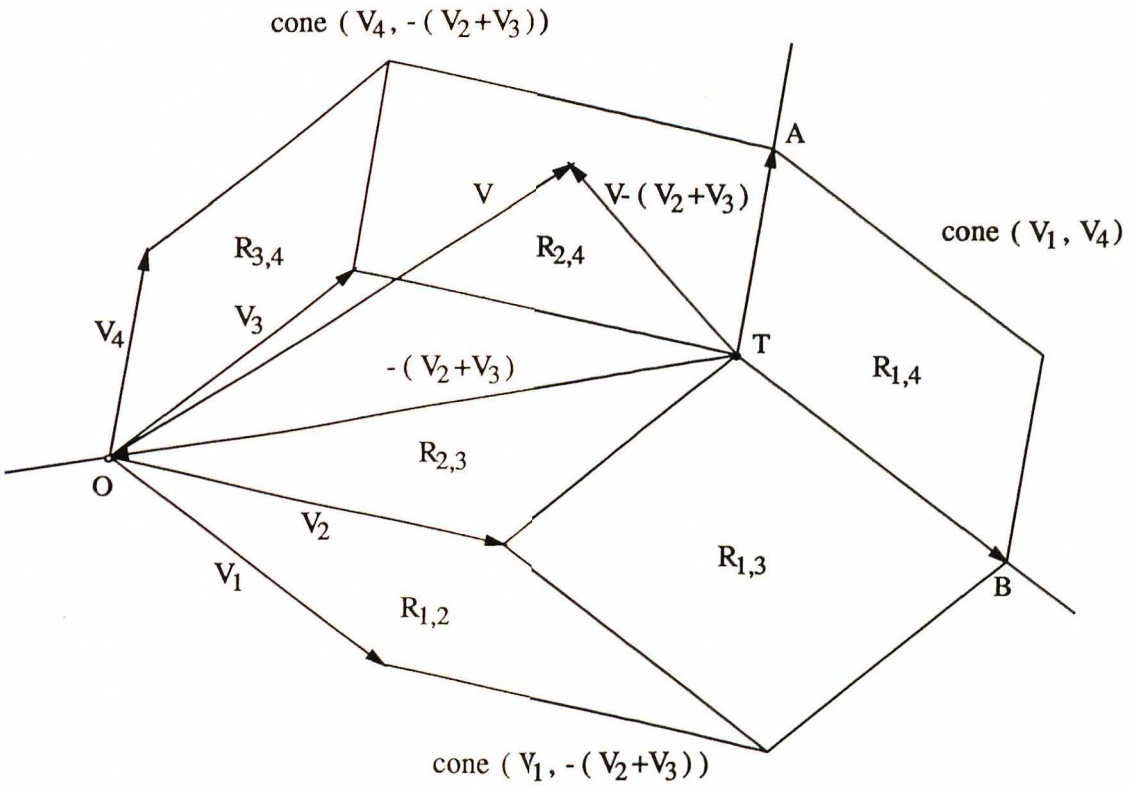


Figure 11: 2-dimensional Tessellations

□

For an example of the above theorem, see Figure 11. When v_1, v_2, v_3, and v_4 are given as in the diagram, the head of an arbitrary linear combination of those four vectors is to be inside the convex area. Let v be such a linear combination,

$$v = a_1v_1 + a_2v_2 + a_3v_3 + a_4v_4, \quad \text{where } a_j \in [0,1].$$

Then

$$\begin{aligned} cone(v_1, v_4) &= cone(\vec{TA}, \vec{TB}), \\ cone(v_1, -v_{inside}^{(1,4)}) &= cone(\vec{TO}, \vec{TB}), \end{aligned}$$

$$cone(v_4, -v_{inside}^{(1,4)}) \;=\; cone(\vec{TO}, \vec{TA}).$$

The location of the head of v is determined by the direction of the vector

$$v_{diff}^{(1,4)} = v - v_{inside}^{(1,4)} = v - (v_2 + v_3).$$

In the diagram, it is inside $cone(v_4, -(v_2 + v_3))$, that is, $cone(v_4, -v_{inside}^{(1,4)})$.

Applying Theorem 4 recursively for n non-zero vectors in a 2-dimensional half space, we can divide a 2-dimensional space into $\binom{n}{2}$ disjoint areas, except at the boundary vectors, at the points $v_{inside}^{(b,t)}$ for $1 \leq b < t \leq n$. Therefore, the set of linear combinations of n vectors with coefficients between 0 and 1 can be divided into $\binom{n}{2}$ disjoint subsets.

Theorem 5 *Let n non-zero vectors $v_1, \ldots, v_n$, which are properly ordered, be in a 2-dimensional half space, where $v_j = (u_j - l_j)h_j$. For any $b, t \in \{1, \ldots, n\}$ such that $b < t$, let*

$$R_{b,t} = \{\alpha_b v_b + \alpha_t v_t + v_{inside}^{(b,t)} \mid \alpha_b, \alpha_t \in [0,1]\}$$

be the area spanned by v_b and v_t with coefficients in [0,1] at the point $v_{inside}^{(b,t)}$. Then

$$R = \bigcup_{b<t}^{n} R_{b,t}.$$

Proof To prove $R \subseteq \bigcup_{b<t}^{n} R_{b,t}$, it must be shown that

$$\forall\, v = \sum_{j=1}^{n} a_j v_j \in R, \quad \exists\, b, t \in \{1, \ldots, n\}, \alpha_b, \alpha_t \in [0,1]$$

$$\text{such that} \quad v = \alpha_b v_b + \alpha_t v_t + v_{inside}^{(b,t)}.$$

Initially it is clear that v is inside $cone(v_1, v_n)$ because v is a linear combination of $v_1, \ldots, v_n$ and $v_2, \ldots, v_{n-1}$ are inside $cone(v_1, v_n)$. From Theorem 4, the vector v can

be represented as one of the following three possible linear combinations:

1. $v = \alpha_1 v_1 + \alpha_n v_n + v_{inside}^{(1,n)}$, if $v_{diff}^{(1,n)} \in S_1 = cone(v_1, v_n)$
2. $v = \beta_1 v_1 + \cdots + \beta_{n-1} v_{n-1}$, if $v_{diff}^{(1,n)} \in S_2 = cone(v_1, -v_{inside}^{(1,n)})$
3. $v = \gamma_2 v_2 + \cdots + \gamma_n v_n$, if $v_{diff}^{(1,n)} \in S_3 = cone(v_n, -v_{inside}^{(1,n)})$

where $\alpha_k, \beta_k, \gamma_k \in [0, 1]$.

If v is as in case 1, then $v \in R_{1,n}$.

If v is as in case 2 or 3, the vector v is represented with $(n-1)$ vectors. The number of vectors decreases by one. We can repeat this process until for some pair of (b, t), the representation falls into case 1; i.e.,

$$v = \alpha_b v_b + \alpha_t v_t + v_{inside}^{(b,t)} \in R_{b,t}.$$

Eventually, this process terminates with $t - b = 1$. Therefore, we can find b and t in $\{1, \ldots, n\}$ such that $v \in R_{b,t}$.

To prove $R \supseteq \bigcup_{b<t}^{n} R_{b,t}$, let

$$v = \alpha_b v_b + \alpha_t v_t + \sum_{j=l+1}^{u-1} v_j \in R_{b,t}.$$

Setting

$$a_1 = \cdots = a_{l-1} = a_{u+1} = \cdots = a_n = 0, \quad a_b = \alpha_b, a_t = \alpha_t, \quad a_{l+1} = \cdots = a_{u-1} = 1,$$

we can say that $v \in R$.

□

So far, we have proved the case when all vectors are in one half space. Just as Theorem 2 has been extended to Theorem 3 for a 1-dimensional space, Theorem 5

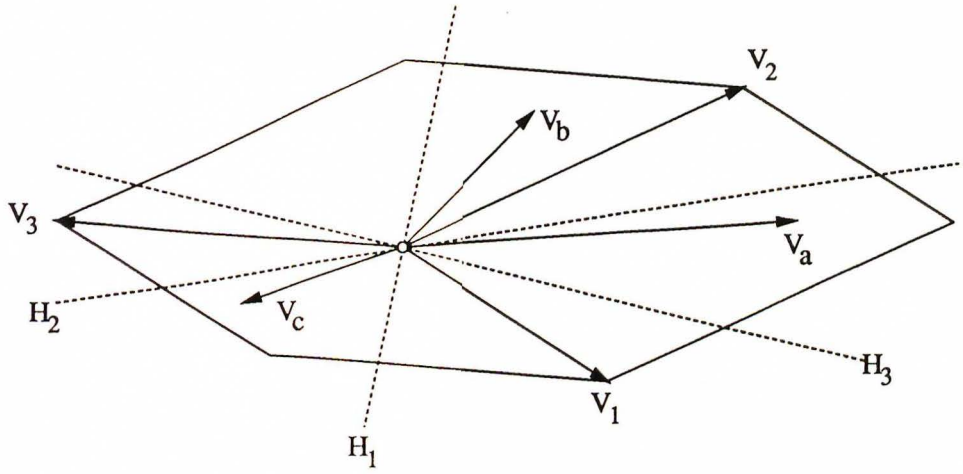


Figure 12: Dividing 2-dimensional Space into Half Spaces

for a 2-dimensional half space may be extended to the full space. The process is so simple that it need not be repeated here, but an example will be given to show how the extended theorem might work for a 2-dimensional space. In Figure 12, any vector inside the $cone(v_1, v_2)$, e.g. v_a, can be represented as a linear combination of v_1 and v_2 that are in a half space divided by line H_1. Similarly, v_b inside the $cone(v_2, v_3)$ (v_c inside $cone(v_3, v_1)$) can be represented as a linear combination of v_2 and v_3 (v_3 and v_1) that are in a half space divided by line H_2 (H_3.)

From the process of the construction of $R_{b,t}$, it is clearly seen that $R_{b,t}$ are mutually disjoint except at the boundary vectors when coefficient α_b or α_t is 0 or 1. In addition, any linear combination of vectors $v_1, \ldots, v_n$ falls into one of $R_{b,t}$. This implies that the images of $\binom{n}{2}$ faces cover the whole image of the domain under H, since $v_k = (u_k - l_k)h_k$ is an image of $(u_k - l_k)e_k$ under H. When all vectors $v_1, \ldots, v_n$ are properly ordered as described in Definition 3, the faces are

$$\eta_1 \times \cdots \times \eta_{b-1} \times D_b \times \eta_{b+1} \times \cdots \times \eta_{t-1} \times D_t \times \eta_{t+1} \times \cdots \times \eta_n, \quad \text{for } 1 \leq b < t \leq n$$

$$\text{where } \eta_j = \begin{cases} u_j & \text{if } v_j \in cone(v_b, v_t) \\ l_j & \text{otherwise} \end{cases}$$

The images of faces, $H(\eta_1 \times \cdots \times \eta_{b-1} \times D_b \times \eta_{b+1} \times \cdots \times \eta_{t-1} \times D_t \times \eta_{t+1} \times \cdots \times \eta_n) = R_{b,t}$, for $b < t$ are mutually disjoint except at the boundaries. The images of faces are called *tessellations* for a 2-dimensional space, and the process of constructing tessellations is the *tessellation process.*

As a final example with the case of a 2-dimensional array, consider the nested loop of Figure 8. From the code,

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix},$$

$$v_1 = (u_1 - l_1)h_1 = \begin{pmatrix} 5 \\ 0 \end{pmatrix}, \quad v_2 = (u_2 - l_2)h_2 = \begin{pmatrix} 6 \\ 6 \end{pmatrix}, \quad v_3 = (u_3 - l_3)h_3 = \begin{pmatrix} 0 \\ 7 \end{pmatrix}$$

According to the reordering scheme described in Definition 3, the vectors should be reordered as

$$(v_1, v_2, v_3).$$

In this example code, since all vectors are in the same half space, the theorems for half space can be applied.

$$v_{inside}^{(1,2)} = 0, \quad v_{inside}^{(2,3)} = 0, \quad v_{inside}^{(1,3)} = v_2,$$

and the faces for tessellations are

$$D_1 \times D_2 \times l_3, \quad l_1 \times D_2 \times D_3, \quad D_1 \times u_2 \times D_3.$$

So the figure of the tessellations looks like Figure 8 (a).

With the tessellation process, we can locate all the array elements referenced in a nested loop using the concept of $v_{inside}^{(b,t)}$.

Chapter 4

Contiguity

In parallel computer systems, processors that run parallel processes interact with each other through shared memory or by passing messages via the network that connects them. Since each processor has the ability to run a process while other processors are concurrently running other processes, the sharing of the data becomes inevitable. If the system provides shared memory, then the data to be shared are allocated there and then fetched from there by all processors. The data thus fetched move through the switching network or data bus. If there is no shared memory, then the data are usually allocated on each processor's local memory.

Since the capacities of the switching network and data bus bandwidth are limited, both of these system components may not function well when the data traffic approaches their capacity limits. Heavy traffic between shared memory and processors or between processors may cause the data transfer to be delayed. To reduce the resulting performance degradation, several remedies have been suggested. One of them is to allocate data to the local memories of those processors that are most likely to access the data. With this method, however, the task of data allocation places a heavy burden on the programmer, making parallel programming difficult. Another method is to develop a compiler capable of transforming nested loops so as to reduce remote data accesses and increase local data accesses. This method alleviates the burden of data allocation, and developing it is one of the main concerns of this thesis.

Most scientific programs handle arrays of large size in nested loops, which are

```
for i0 = 1,10
   for i1 = 0,6
      for i2 = 0,8
         A = X(i0, i1, i2)
      endfor
   endfor
endforall
```

(a) Given code

```
forall i0 = 1,10
   for i1 = 0,6
      for i2 = 0,8
         A = X(i0, i1, i2)
      endfor
   endfor
endforall
```

(b) Parallel code

Figure 13: Parallelization

usually shared. Once dependence analysis determines that a nested loop is parallelizable, then the nested loop will result in a parallelized form as in Figure 13 (a) and (b). In the figure, there are ten parallel processes. Those parallel processes are assigned to processors and run concurrently. Because all processors frequently need to access the data in the shared array X, the required data must be transferred from either the shared or distributed memories through the network or the data bus. If a data element is referenced several times by one processor, it may be efficient to keep that element in local memory. Even if each data element is accessed only once, if a processor accesses all data in a block (contiguous area), we may be able to utilize a fast *block copy* to take them to local memory before they are used for actual computation. Block copy, a feature found in many parallel computer systems, is a method of transferring data from one contiguous block in memory to another at high speed.

In this chapter, we develop an algorithm in order to decide whether the data accessed in a nested loop form contiguous blocks. Without loss of generality, it is assumed throughout this thesis that arrays are allocated in row major order.

4.1 q-contiguity

In this section, we introduce q-contiguity to help determine whether all the referenced array elements form contiguous blocks. In order for the concept to be applied to any

array allocation scheme, whether row major or column major, q is prefixed. For example, Figure 14 (a) is 1-contiguous and 2-contiguous; (b) is 2-contiguous but not 1-contiguous. In (b), the referenced elements on a row are contiguous in the memory with the row major allocation scheme, but not contiguous with the column major scheme. For a language whose array allocation scheme is row major, like C, if the referenced elements satisfy the condition for d-contiguity (recall that d is the dimension of the array referenced in a nested loop), then they form contiguous blocks in memory. Conversely, if the array allocation scheme is column major, like Fortran, then the condition for 1-contiguity should be satisfied for contiguous blocks.

Definition 5 *In the nested loop of Figure 5, let* $1 \leq q \leq d$ *and*

$$\vec{R}_q = (r_1, \ldots, r_{q-1}, *, r_{q+1}, \ldots, r_d)^T$$

is a d-tuple of integers with an undefined value at position q where

$$\min_{\vec{l} \leq \vec{i} \leq \vec{u}} (g_k(\vec{i})) \leq r_k \leq \max_{\vec{l} \leq \vec{i} \leq \vec{u}} (g_k(\vec{i})), \quad \textit{for } k = 1, \ldots, d,$$

$$\textit{where } g_k(\vec{i}) = \sum_{j=1}^{n} c_{kj} i_j.$$

Let $S_{\vec{R}_q}$, m_q *and* M_q *be*

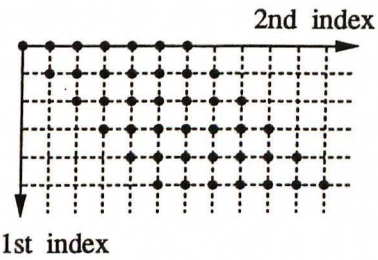
$$S_{\vec{R}_q} = \{\vec{i} \mid g_k(\vec{i}) = r_k, \quad \textit{for } k = 1, \ldots, d, \ k \neq q\}, \tag{1}$$

$$m_q = \min_{\vec{i} \in S_{\vec{R}_q}} (g_q(\vec{i})), \qquad M_q = \max_{\vec{i} \in S_{\vec{R}_q}} (g_q(\vec{i})). \tag{2}$$

Then the footprint is said to be q-**contiguous**, *if*

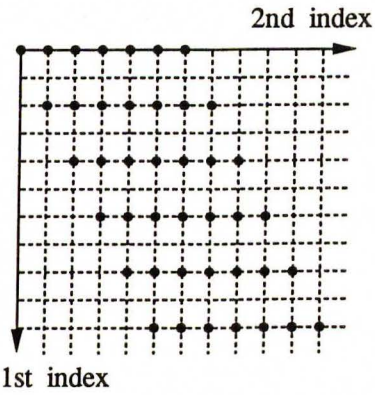
$$\exists\, \vec{i} \in S_{\vec{R}_q} \quad \textit{such that} \quad g_q(\vec{i}) = r_q, \quad \textit{for } m_q \leq r_q \leq M_q.$$

```
for i1 = 0,6
   for i2 = 0,5
      X(i2, i1 + i2)
   endfor
endfor
```



(a) 1-contiguous and 2-contiguous

```
for i1 = 0,6
   for i2 = 0,5
      X(2i2, i1 + i2)
   endfor
endfor
```



(b) Not 1-contiguous, but 2-contiguous

Figure 14: q-contiguous Footprints

In this chapter, algorithms will be presented that determine if the footprint of an array by a nested loop given in the form of Figure 5 is q-contiguous. The results can be applied for any arbitrary value of q, $1 \leq q \leq d$. Here, the algorithms are developed for $q = d$ because the row major allocation scheme is assumed. Relevant algorithms for other values of q can be easily obtained, based on the algorithms of the case $q = d$.

4.2 Contiguity When Nesting Depth is Equal to Array Dimension

Let's consider the simplest case where the nesting depth and the dimension of the array variable are the same, i.e., $n = d$, and the mapping matrix, H, for the index

function is non-singular. If it is singular, either the case can be reduced to the lower dimensional case when some indices of the index function are constant, or the footprint is not contiguous because the simultaneous linear equation (3) does not have a solution. These possibilities will be discussed in detail for 1- and 2-dimensional arrays in the appropriate sections.

In a nested loop of depth d with a d-dimensional array reference whose index function is

$$(c_{11}i_1 + \cdots + c_{1d}i_d, \ldots, c_{d1}i_1 + \cdots + c_{dd}i_d),$$

we will develop the condition for the footprint to be d-contiguous. The mapping function, H, is

$$H = \begin{pmatrix} c_{11} & \cdots & c_{1d} \\ \vdots & \ddots & \vdots \\ c_{d1} & \cdots & c_{dd} \end{pmatrix}.$$

The following theorem states the necessary and sufficient condition for the footprint to be d-contiguous.

Theorem 6 *In a nested loop of depth d with a d-dimensional array reference, let the mapping function, H, be non-singular. Then the footprint is d-contiguous if and only if*

$$\left|\frac{\hat{M}_{dk}}{M}\right| \leq u_k - l_k \quad \textit{are integers, for } k = 1, \ldots, d, \tag{3}$$

where

$$M = \det(H), \qquad \hat{M}_{dk} = \det(\hat{H}_{dk}),$$

$$\hat{H}_{dk} = \begin{pmatrix} c_{11} & \cdots & c_{1,k-1} & c_{1,k+1} & \cdots & c_{1d} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ c_{d-1,1} & \cdots & c_{d-1,k-1} & c_{d-1,k+1} & \cdots & c_{d-1,d} \end{pmatrix}.$$

Proof Suppose that the footprint is d-contiguous. Then there exist two different

vectors of loop variables, $\vec{i} \neq \vec{i'}$, so that the two coordinates

$$(\sum_{j=1}^{d} c_{1j} i_j, \ldots, \sum_{j=1}^{d} c_{dj} i_j) \text{ and } (\sum_{j=1}^{d} c_{1j} i'_j, \ldots, \sum_{j=1}^{d} c_{dj} i'_j)$$

are adjacent along the direction of the x_d-axis in the range; that is,

$$(\sum_{j=1}^{d} c_{1j} i_j, \ldots, \sum_{j=1}^{d} c_{dj} i_j) - (\sum_{j=1}^{d} c_{1j} i'_j, \ldots, \sum_{j=1}^{d} c_{dj} i'_j) = (0, \ldots, 0, \pm 1),$$

$$(\sum_{j=1}^{d} c_{1j} (i_j - i'_j), \ldots, \sum_{j=1}^{d} c_{dj} (i_j - i'_j)) = (0, \ldots, 0, \pm 1)$$

has a solution. For simplicity, put $\vec{I} = \vec{i} - \vec{i'}$. Then the problem becomes equivalent to the following simultaneous systems of linear equations:

$$\begin{cases} \sum_{j=1}^{d} c_{1j} I_j & = & 0 \\ \vdots & & \\ \sum_{j=1}^{d} c_{d-1\,j} I_j & = & 0 \\ \sum_{j=1}^{d} c_{dj} I_j & = & \pm 1 \end{cases}$$

Since H is assumed non-singular, according to *Cramer's rule*,

$$I_k = i_k - i'_k = \frac{\pm(-1)^{d-k} \hat{M}_{dk}}{M} \text{ for } k = 1, \ldots, d.$$

Since i_k and i'_k are between l_k and u_k, and integers,

$$\left| \frac{\hat{M}_{dk}}{M} \right| \leq u_k - l_k \text{ are integers, for } k = 1, \ldots, d.$$

Conversely, suppose condition (3) is true. For any $\vec{R}_d = (r_1, \ldots, r_{d-1}, *)$, we can compute m_d and M_d from Equation (2). Let $m_d \leq r_d \leq M_d$. If an element with the

coordinate $(r_1, \ldots, r_{d-1}, r_d)$ is in the footprint, the coordinate of the base-element, $H(\vec{l})$, can be obtained by subtracting some displacement

$$(\sum_{j=1}^{d} c_{1j}t_j, \ldots, \sum_{j=1}^{d} c_{dj}t_j), \quad \text{where } 0 \leq \vec{t} \leq \vec{u} - \vec{l}$$

from the coordinate

$$(r_1, \ldots, r_{d-1}, r_d).$$

Note that the above subtraction tends to move the coordinate $(r_1, \ldots, r_{d-1}, r_d)$ toward the coordinate of the base-element, because $0 \leq \vec{t}$. Therefore,

$$(r_1, \ldots, r_{d-1}, r_d) - (\sum_{j=1}^{d} c_{1j}t_j, \ldots, \sum_{j=1}^{d} c_{dj}t_j) = (\sum_{j=1}^{d} c_{1j}l_j, \ldots, \sum_{j=1}^{d} c_{dj}l_j)$$

$$(r_1, \ldots, r_{d-1}, r_d) = (\sum_{j=1}^{d} c_{1j}(t_j + l_j), \ldots, \sum_{j=1}^{d} c_{dj}(t_j + l_j))$$

should have a solution such that $0 \leq \vec{t} \leq \vec{u} - \vec{l}$. We get the following systems of equations:

$$\begin{cases} \sum_{j=1}^{d} c_{1j}(t_j + l_j) &= r_1 \\ \vdots & \\ \sum_{j=1}^{d} c_{dj}(t_j + l_j) &= r_d \end{cases}$$

Solving the equations yields:

$$\begin{aligned} t_k + l_k &= \frac{1}{M} \begin{vmatrix} c_{11} & \cdots & c_{1,k-1} & r_1 & c_{1,k+1} & \cdots & c_{1d} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{d1} & \cdots & c_{d,k-1} & r_d & c_{d,k+1} & \cdots & c_{dd} \end{vmatrix} \\ &= \frac{(-1)^{k+1}}{M} \left[r_1 \hat{M}_{1k} - + \cdots + (-1)^d r_{d-1} \hat{M}_{d-1,k} + (-1)^{d+1} r_d \hat{M}_{dk} \right]. \end{aligned}$$

Since $r_k = \sum_{j=1}^{d} c_{kj} i_j$, for $k = 1, \ldots, d-1$, if $\vec{i} \in S_{\vec{R}_d}$, the above equation can be

expanded as follows:

$$\begin{aligned}
t_k + l_k &= \frac{(-1)^{k+1}}{M}\left[\hat{M}_{1k}\sum_{j=1}^{d} c_{1j}i_j - + \cdots + (-1)^d \hat{M}_{d-1,k}\sum_{j=1}^{d} c_{d-1,j}i_j \right. \\
&\quad \left. +(-1)^{d+1}\hat{M}_{dk}r_d\right] \\
&= \frac{(-1)^{k+1}}{M}\left[\left\{\left(c_{11}\hat{M}_{1k} - + \cdots + (-1)^{d+1}c_{d1}\hat{M}_{dk}\right)i_1 + (-1)^d c_{d1}\hat{M}_{dk}i_1\right\}\right. \\
&\quad + \cdots + \left\{\left(c_{1d}\hat{M}_{1k} - + \cdots + (-1)^{d+1}c_{dd}\hat{M}_{dk}\right)i_d + (-1)^d c_{dd}\hat{M}_{dk}i_d\right\} \\
&\quad \left. +(-1)^{d+1}\hat{M}_{dk}r_d\right] \\
&= \frac{(-1)^{k+1}}{M}\left[(-1)^{k+1}Mi_k + (-1)^d\hat{M}_{dk}\sum_{j=1}^{d} c_{dj}i_j + (-1)^{d+1}\hat{M}_{dk}r_d\right] \\
&= i_k + (-1)^{k+d}\frac{\hat{M}_{dk}}{M}\left(r_d - \sum_{j=1}^{d} c_{dj}i_j\right) \\
&= i_k + (-1)^{k+d}\frac{\hat{M}_{dk}}{M}\left(r_d - g_d(\vec{i})\right)
\end{aligned}$$

From the assumption, $\left|\frac{\hat{M}_{dk}}{M}\right|$ are integers, for $k = 1, \ldots, d$. So $t_k + l_k$ are integers. As the difference between r_d and $g_d(\vec{i})$ increases by 1, the value of $t_k + l_k$ changes by the amount of $\left|\frac{\hat{M}_{dk}}{M}\right|$. Let us assume that

$$\begin{aligned}
&\vec{i}_m = (i_1^m, \ldots, i_d^m), && \vec{i}_M = (i_1^M, \ldots, i_d^M), \\
&m_d = g_d(\vec{i}_m), && M_d = g_d(\vec{i}_M).
\end{aligned}$$

Then for $\vec{i} = \vec{i}_m$ or $\vec{i}_M$,

$$\begin{aligned}
t_k + l_k &= i_k^M + (-1)^{k+d}\frac{\hat{M}_{dk}}{M}(r_d - M_d), \quad \text{or} \\
t_k + l_k &= i_k^m + (-1)^{k+d}\frac{\hat{M}_{dk}}{M}(r_d - m_d).
\end{aligned}$$

When $r_d = m_d$ or M_d, $t_k + l_k = i_k^m$ or i_k^M. So as r_d moves from m_d to M_d, $t_k + l_k$ varies

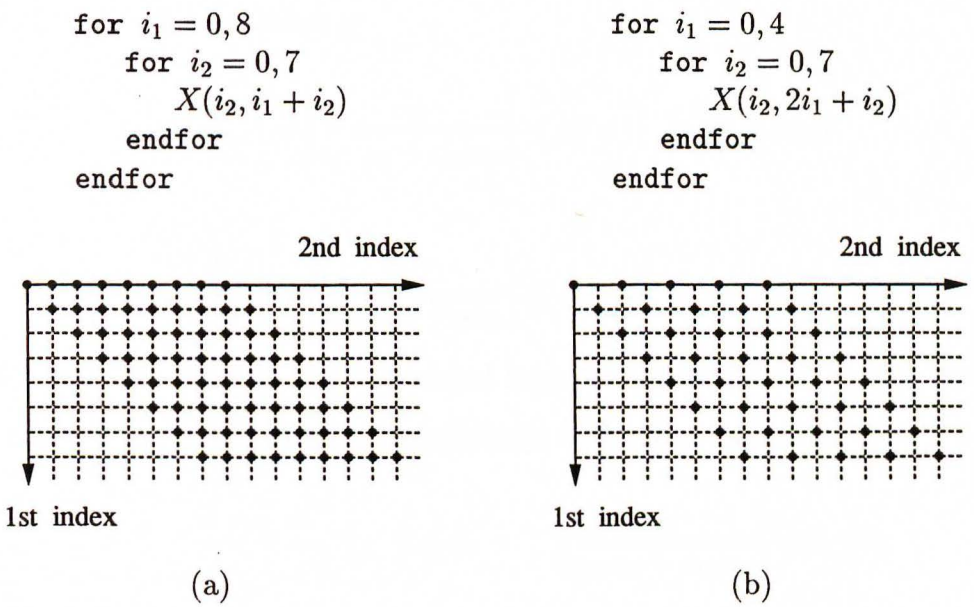


Figure 15: 2-contiguity When Nesting Depth is Equal to Array Dimension

from i_k^m to i_k^M by the step $(-1)^{k+d}\frac{\hat{M}_{dk}}{M}$. Therefore, the value of $t_k + l_k$ is bounded by i_k^m and i_k^M; i.e.,

$$\min(i_k^m, i_k^M) - l_k \leq t_k \leq \max(i_k^m, i_k^M) - l_k.$$

Since $l_k \leq i_k^m, i_k^M \leq u_k$, t_k is positive and less than $u_k - l_k$. So the footprint is d-contiguous.

□

For example, when $d = 1$, the index function is $(c_{11}i_1)$. The condition that the footprint is 1-contiguous is

$$\left|\frac{\hat{M}_{11}}{M}\right| = \left|\frac{1}{c_{11}}\right| \leq u_1 - l_1 \text{ is an integer.}$$

So if $|c_{11}| = 1$ and $1 \leq u_1 - l_1$, the footprint is 1-contiguous.

When $d = 2$, the index function is $(c_{11}i_1 + c_{12}i_2, c_{21}i_1 + c_{22}i_2)$. The condition that

the footprint is 2-contiguous is

$$\left|\frac{\hat{M}_{21}}{M}\right| = \left|\frac{c_{12}}{c_{11}c_{22} - c_{12}c_{21}}\right| \leq u_1 - l_1 \text{ is an integer,}$$

and

$$\left|\frac{\hat{M}_{22}}{M}\right| = \left|\frac{c_{11}}{c_{11}c_{22} - c_{12}c_{21}}\right| \leq u_2 - l_2 \text{ is an integer.}$$

For example, refer to the two nested loops in Figure 15. In (a), the array index function is $(i_2, i_1 + i_2)$. From the coefficients of the loop variables,

$$\left|\frac{c_{12}}{c_{11}c_{22} - c_{12}c_{21}}\right| = 1 \leq 8, \qquad \left|\frac{c_{11}}{c_{11}c_{22} - c_{12}c_{21}}\right| = 0 \leq 7.$$

The above two values are integers, so the footprint is 2-contiguous, as we can see in the illustration.

In (b), the array index function is $(i_2, 2i_1 + i_2)$. From the coefficients of the loop variables,

$$\left|\frac{c_{12}}{c_{11}c_{22} - c_{12}c_{21}}\right| = \frac{1}{2}, \qquad \left|\frac{c_{11}}{c_{11}c_{22} - c_{12}c_{21}}\right| = 0.$$

Since one of the above values is not integer, the footprint is not 2-contiguous, as illustrated in the figure.

Whereas most theorems in this thesis are restricted to the 1- and 2-dimensional arrays, Theorem 6 is applied to an array of any dimension.

4.3 Contiguity When Nesting Depth is Greater than Array Dimension

We need to generalize Theorem 6 for the d-contiguity of the footprint when the nesting depth is greater than the array dimension. When $n > d$, the footprint of a nested

loop is the area swept by the footprint of the nested loop induced by a set of $(n-1)$ loop variables, as the other loop variable changes the value from its lower bound to the upper bound. The following theorem formalizes this property.

Theorem 7 *In a nested loop of depth n with a d-dimensional array reference, where $n > d$, if the footprint of a nested loop is d-contiguous, then there exists a nested loop, induced by a set of $(n-1)$ loop variables, whose footprint is d-contiguous.*

Proof There are $\binom{n}{n-1}$ choices to select $(n-1)$ loop variables from n loop variables. Suppose that the footprint of the nested loop is d-contiguous and none of the footprints of the nested loops induced by any set of $(n-1)$ loop variables is not d-contiguous. For a nested loop induced by $\{i_1, \ldots, i_{n-1}\}$, let the submatrix of H be

$$C = \begin{pmatrix} h_1 & \cdots & h_{n-1} \end{pmatrix}.$$

For any $\vec{R}_n = (r_1, \ldots, r_{n-1}, *)$, we can compute m_n and M_n from Equation (2). Let $m_n \leq r_n \leq M_n$. Because the footprint of the nested loop is d-contiguous, $X(r_1, \ldots, r_{n-1}, r_n)$ is one element of the footprint. For some $t_n \in [l_n, u_n]$, the coordinate

$$\vec{r} = (r_1, \ldots, r_{d-1}, r_d) - t_n(c_{1n}, \ldots, c_{dn})$$

is in the base-space for $\{i_n = l_n\}$. (Note that the base-space for $\{i_n = l_n\}$ is the footprint of the nested loop induced by $\{i_1, \ldots, i_{n-1}\}$.) But since the base-space is assumed not to be d-contiguous, the point $X(\vec{r})$ can not be a referenced point, which is a contradiction. Therefore, at least one set of $(n-1)$ loop variables should induce a nested loop whose footprint is d-contiguous.

□

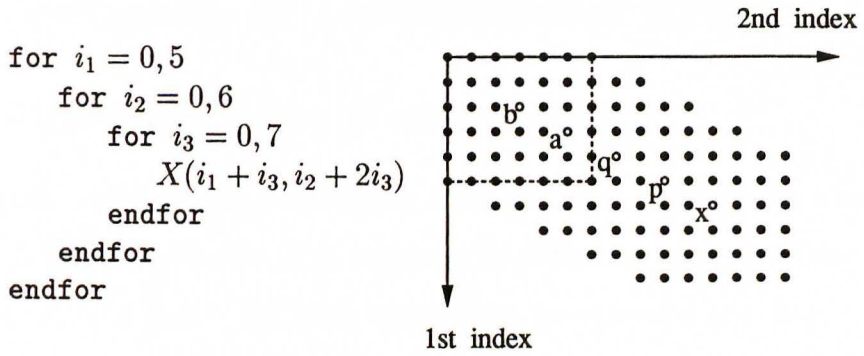


Figure 16: 2-contiguity When Nesting Depth is Greater than Array Dimension

For example, in Figure 16, the index function in matrix form of the nested loop is

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

As we can see in the illustration, the footprint of the nested loop is 2-contiguous. There are three nested loops induced by $\{i_1, i_2\}$, $\{i_2, i_3\}$, and $\{i_3, i_1\}$. The footprint of the nested loop induced by $\{i_1, i_2\}$ or $\{i_2, i_3\}$ is 2-contiguous, but the footprint of the nested loop induced by $\{i_3, i_1\}$ is not. In the illustration, the element x is not in the base-space for $\{i_3 = 0\}$. By subtracting $t_3(1,2)$ from the coordinate of x, we can get the element p when $t_3 = 1$, q when $t_3 = 2$, a when $t_3 = 3$, and b when $t_3 = 4$. The points p and q are not in the base-space for $\{i_3 = 0\}$, but a and b are in that base-space.

Theorem 7 can be applied recursively by reducing the value of the nesting depth by 1 until it reaches d. When the nesting depth is reduced to d, then we can apply Theorem 6 to determine the d-contiguity of the footprint.

In the following sections, we will develop concrete conditions for the footprint to be d-contiguous when $d = 1$ and 2.

4.4 Contiguity of 1-dimensional Array

4.4.1 Contiguity When Nesting Depth is 2

To develop the condition for 1-contiguity when the array dimension is 1 and the nesting depth is 2 in this chapter, it is necessary to define the term *extended tessellation* for a 1-dimensional space. For each tessellation, an extended tessellation can be defined as follows. The extended tessellation of the tessellation OA in Figure 17 (a) is the segment PQ, where P is the next point on the left of O, and Q is the next point on the right of A.

When $d = 1$ and $n = 2$, the index function of the nested loop, in matrix form, is

$$H = \begin{pmatrix} g_1 \end{pmatrix} = \begin{pmatrix} h_1 & h_2 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \end{pmatrix}.$$

If $rank(H) = 0$, all coefficients are zero; i.e., there is only one array element referenced by the nested loop. We assume $rank(H) = 1$ if $d = 1$.

From Theorem 7, there should exist at least one loop induced by one loop variable whose footprint is 1-contiguous. In this section, we will deduce the condition for the footprint of a nested loop to be 1-contiguous, provided that the footprint of a nested loop induced by a loop variables i_1 is 1-contiguous.

Figure 17 (a) shows the footprint of the loop induced by $\{i_1\}$ at $i_2 = l_2$, i.e., the base-space for $\{i_2 = l_2\}$. First, let us assume $c_{11} = 1$. Then, the thick solid line segment OA is the base-space for $\{i_2 = l_2\}$, and is assumed to be 1-contiguous. Let us also assume that the point O is the base-element and its coordinate is $H(l_1, l_2)$, which is the minimum index value, and the coordinate of A is $H(u_1, l_2)$, which is the maximum index value. The points P and Q are $H(l_1, l_2) - 1$ and $H(u_1, l_2) + 1$, respectively. The footprint of the loop induced by $\{i_1\}$ at some value $i_2 \in [l_2, u_2]$ is the footprint to which the base-space, segment OA, is moved by the displacement

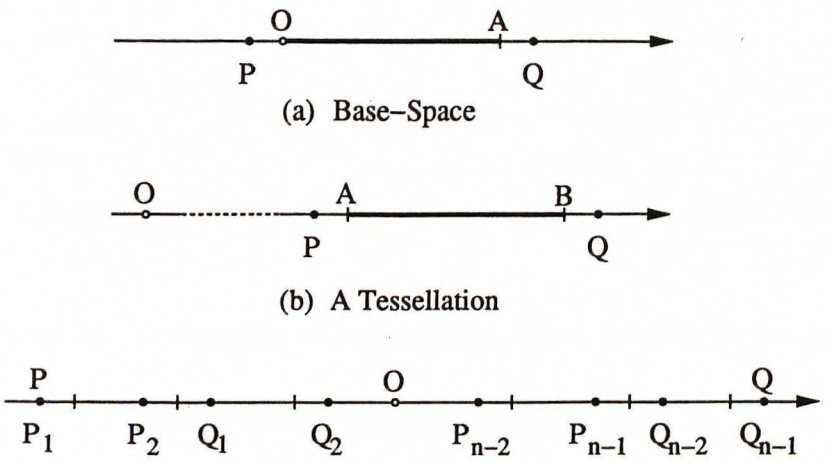


Figure 17: 1-contiguity in a 1-dimensional Space

$c_{12}i_2$ and it is 1-contiguous. As the value i_2 grows, the footprint is thus moved farther from the base-space for $\{i_2 = l_2\}$. In order for the footprint of the given nested loop to be 1-contiguous, either the coordinate $H(l_1, l_2 + 1)$ or $H(u_1, l_2 + 1)$ must be in the extended tessellation, i.e., between P and Q inclusively. So the condition for the footprint to be 1-contiguous is

$$\begin{cases} H(l_1, l_2) - 1 \leq H(l_1, l_2 + 1) \leq H(u_1, l_2) + 1, & \text{or} \\ H(l_1, l_2) - 1 \leq H(u_1, l_2 + 1) \leq H(u_1, l_2) + 1. \end{cases} \tag{4}$$

Using the coefficients, we get the following condition:

$$\begin{cases} c_{11}l_1 + c_{12}l_2 - 1 \leq c_{11}l_1 + c_{12}(l_2 + 1) \leq c_{11}u_1 + c_{12}l_2 + 1, & \text{or} \\ c_{11}l_1 + c_{12}l_2 - 1 \leq c_{11}u_1 + c_{12}(l_2 + 1) \leq c_{11}u_1 + c_{12}l_2 + 1. \end{cases}$$

$$\begin{cases} -1 \leq c_{12} \leq 1 + c_{11}(u_1 - l_1), & \text{or} \\ -1 - c_{11}(u_1 - l_1) \leq c_{12} \leq 1. \end{cases}$$

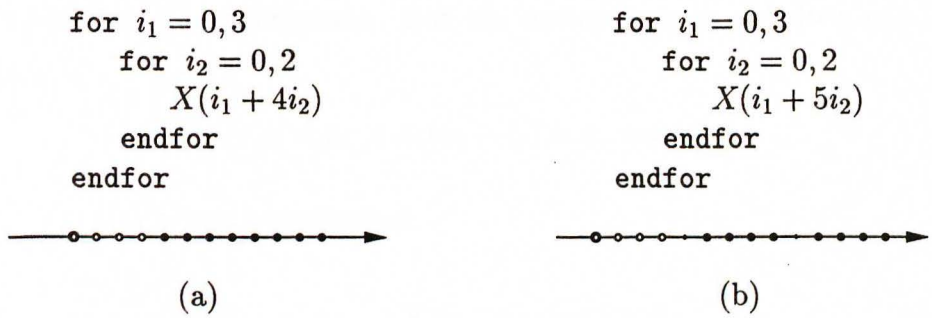


Figure 18: 1-contiguity of 1-dim Array When Nesting Depth is 2

Finally, we get the condition for the footprint to be 1-contiguous

$$|c_{12}| \leq 1 + c_{11}(u_1 - l_1). \tag{5}$$

Similarly, if $c_{11} = -1$, the condition for the footprint to be 1-contiguous is

$$\begin{cases} H(u_1, l_2) - 1 \leq H(l_1, l_2 + 1) \leq H(l_1, l_2) + 1, & \text{or} \\ H(u_1, l_2) - 1 \leq H(u_1, l_2 + 1) \leq H(l_1, l_2) + 1. \end{cases} \tag{6}$$

Here, the corresponding condition to (5) is

$$|c_{12}| \leq 1 - c_{11}(u_1 - l_1). \tag{7}$$

Consequently, from the two conditions (5) and (7), when $d = 1$, $n = d + 1$, and $|c_{11}| = 1$, the condition for the footprint to be 1-contiguous is

$$|c_{12}| \leq 1 + |c_{11}|(u_1 - l_1) = 1 + (u_1 - l_1). \tag{8}$$

For example, in Figure 18, the set of points marked with o is the base-space for $\{i_2 = 0\}$ and is 1-contiguous. The nested loop in (a) satisfies condition (8);

$$|c_{12}| = 4, \quad 1 + (u_1 - l_1) = 4 \implies 4 \leq 4.$$

So the footprint is 1-contiguous. But the nested loop in (b) does not satisfy the condition;

$$|c_{12}| = 5, \quad 1 + (u_1 - l_1) = 4 \implies 5 \not\leq 4.$$

So the footprint is not 1-contiguous.

4.4.2 Contiguity When Nesting Depth is Greater Than 2

When $d = 1$ and $n > 2$, the index function of the nested loop, in matrix form, is

$$H = \begin{pmatrix} g_1 \end{pmatrix} = \begin{pmatrix} h_1 & \cdots & h_n \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1n} \end{pmatrix}.$$

From Theorem 7, if the footprint is 1-contiguous, there should exist a nested loop induced by a set of $(n-1)$ loop variables whose footprint is 1-contiguous. In this section, the condition for the footprint of a nested loop to be 1-contiguous is deduced, provided that the footprint of a nested loop induced by a set of loop variables $\{i_1, \ldots, i_{n-1}\}$ is 1-contiguous.

In the 1-contiguous footprint of a nested loop induced by a set of loop variables $\{i_1, \ldots, i_{n-1}\}$, there exist $(n-1)$ tessellations, $R_1, \ldots, R_{n-1}$ that are disjoint line segments except at the boundary points. In the footprint of the base-space for $\{i_n = l_n\}$, the minimum and maximum values in the index occur, respectively, at

$$\vec{i}_{m0} = (\nu_1, \ldots, \nu_{n-1}, l_n), \quad \text{where } \nu_k = \begin{cases} l_k & \text{if } c_{1k} \geq 0 \\ u_k & \text{if } c_{1k} < 0 \end{cases}$$

$$\vec{i}_{M0} = (\mu_1, \ldots, \mu_{n-1}, l_n), \quad \text{where } \mu_k = \begin{cases} u_k & \text{if } c_{1k} \geq 0 \\ l_k & \text{if } c_{1k} < 0 \end{cases}$$

When $i_n = l_n + 1$, the coordinates $H(\vec{i}_{m0})$ and $H(\vec{i}_{M0})$ in the base-space are moved

to the coordinates $H(\vec{i}_{m1})$ and $H(\vec{i}_{M1})$, where

$$\vec{i}_{m1} = \vec{i}_{m0} + (0, \ldots, 0, 1), \qquad \vec{i}_{M1} = \vec{i}_{M0} + (0, \ldots, 0, 1).$$

In order that the footprint is 1-contiguous, either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ should be inside one of the extended tessellations in the base-space.

As for a tessellation R_k, let us assume $c_{1k} > 0$. Figure 17 (b) shows a tessellation R_k when $c_{1k} > 0$. The point O is the base-element and its coordinate is $H(\vec{l})$. The coordinates of the points A and B are $H(\vec{l}) + v^{(k)}_{inside}$ and $H(\vec{l}) + v^{(k)}_{inside} + (u_k - l_k)h_k$, respectively. Since the index function is linear,

$$\begin{aligned} H(\vec{l}) + v^{(k)}_{inside} &= H(l_1, \ldots, l_n) + \sum_{v_j:\ \text{term of } v^{(k)}_{inside}} v_j \\ &= \sum_{j=1}^{n} l_j h_j + \sum_{v_j:\ \text{term of } v^{(k)}_{inside}} (u_j - l_j)h_j \\ &= \sum_{j=1}^{n} \delta_j h_j = H(\vec{\delta}) \end{aligned}$$

$$\text{where } \vec{\delta} = (\delta_1, \ldots, \delta_n), \qquad \delta_j = \begin{cases} l_j & \text{if } v_j \text{ is not term of } v^{(k)}_{inside} \\ u_j & \text{if } v_j \text{ is term of } v^{(k)}_{inside} \end{cases}$$

The condition for either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ to be inside the extended tessellation R_k, i.e., between P and Q, is

$$\begin{cases} H(\vec{\delta}) - 1 \le H(\vec{i}_{m1}) \le H(\vec{\delta}) + (u_k - l_k)h_k + 1, & \text{or} \\ H(\vec{\delta}) - 1 \le H(\vec{i}_{M1}) \le H(\vec{\delta}) + (u_k - l_k)h_k + 1. \end{cases}$$

Similarly, when $c_{1k} < 0$, the condition is

$$\begin{cases} H(\vec{\delta}) + (u_k - l_k)h_k - 1 \le H(\vec{i}_{m1}) \le H(\vec{\delta}) + 1, & \text{or} \\ H(\vec{\delta}) + (u_k - l_k)h_k - 1 \le H(\vec{i}_{M1}) \le H(\vec{\delta}) + 1. \end{cases}$$

In summary, if there exists any tessellation such that either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ is inside the extended tessellation, the footprint is 1-contiguous. As in Figure 17 (c), either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ is in any interval $[P_k, Q_k]$ for $k = 1, \ldots, n-1$. In that figure, it will be noticed that, as is characteristic of a 1-dimensional space, the union of all extended tessellations makes one big interval PQ. Therefore, we can get the following simple unified condition for 1-contiguous footprint:

$$\begin{cases} H(\vec{i}_{m0}) - 1 \le H(\vec{i}_{m1}) \le H(\vec{i}_{M0}) + 1, & \text{or} \\ H(\vec{i}_{m0}) - 1 \le H(\vec{i}_{M1}) \le H(\vec{i}_{M0}) + 1 \end{cases}$$

$$\begin{cases} -1 \le H(\vec{i}_{m1}) - H(\vec{i}_{m0}) \le 1 + (H(\vec{i}_{M0}) - H(\vec{i}_{m0})), & \text{or} \\ -1 - (H(\vec{i}_{M0}) - H(\vec{i}_{m0})) \le H(\vec{i}_{M1}) - H(\vec{i}_{M0}) \le 1 \end{cases}$$

$$\begin{cases} -1 \le H(0, \ldots, 0, 1) \le 1 + \sum_{j=1}^{n-1} |c_{1j}|(u_j - l_j), & \text{or} \\ -1 - \sum_{j=1}^{n-1} |c_{1j}|(u_j - l_j) \le H(0, \ldots, 0, 1) \le 1 \end{cases}$$

$$|c_{1n}| \le 1 + \sum_{j=1}^{n-1} |c_{1j}|(u_j - l_j). \tag{9}$$

4.5 Contiguity of 2-dimensional Array

4.5.1 Contiguity When Nesting Depth is 3

The extended tessellation of a 2-dimensional space can also be defined in the above manner. The extended tessellation of the tessellation $ABCD$ in Figure 19 (a) is the parallelogram enclosed by the points P_1, P_2, P_3, Q_1, Q_2, and Q_3, where P_1, P_2, and

P_3 are the next points on the left of A, B and C, respectively, and Q_1, Q_2, and Q_3 are the next points on the right of A, D, and C, respectively.

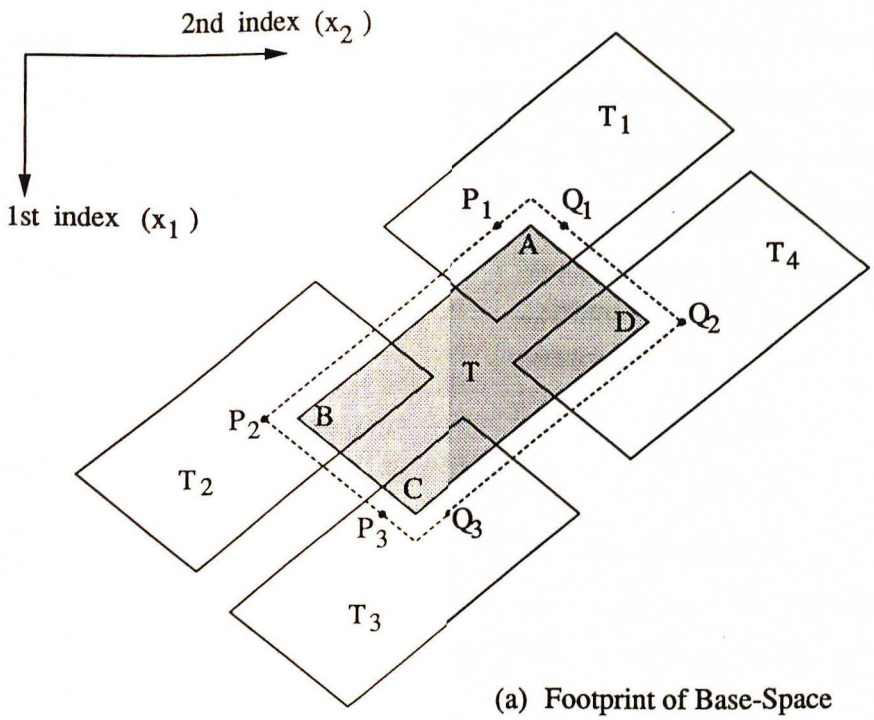
When $d = 2$ and $n = 3$, the index function of the nested loop, in matrix form, is

$$H = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{pmatrix}.$$

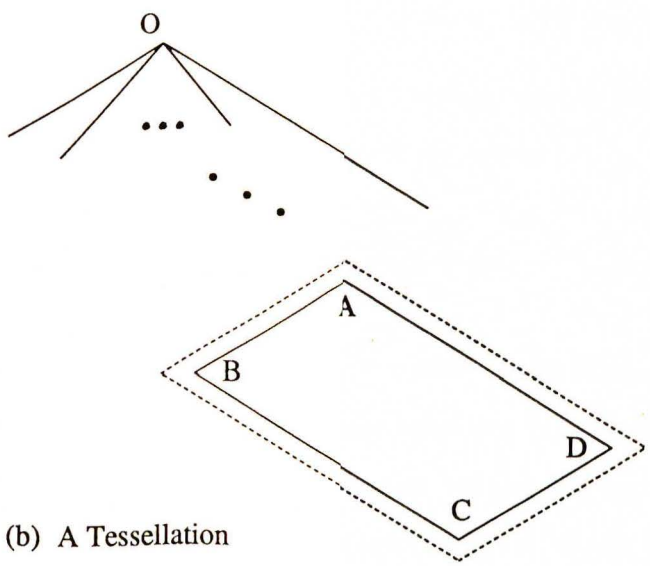
If $rank(H) = 1$, one row of the matrix H is a multiple of the other; that is, $c_{1j} = k_a c_{2j}$ or $c_{2j} = k_b c_{1j}$ for $j \in \{1, 2, 3\}$. If $k_a = 0$ (the first row is zero), this case is reduced to the case where $d = 1$, because only the elements on one row of the array are referenced. If $k_b = 0$ (the second row is zero), only the elements on one column of the array are referenced. Since the row major allocation is assumed, the footprint can never be 2-contiguous when $k_b = 0$. When $c_{2j}/c_{1j} = k_a$ or $1/k_b$, then k_a and k_b are not zero, and all vectors v_1, v_2, and v_3 are on the same straight line with the slope k_a or $1/k_b$. The footprint of this case can never be 2-contiguous, because the line is slanted, not horizontal. Therefore, it is assumed that $rank(H) = 2$ if $d = 2$.

From Theorem 7, there should exist at least one nested loop induced by a set of two loop variables whose footprint is 2-contiguous, in order for the footprint of an array by nested loop of depth 3 to be 2-contiguous. In this section, such a condition will be deduced, provided that the footprint of a nested loop induced by $\{i_1, i_2\}$ is 2-contiguous.

There is only one tessellation in the footprint of a nested loop induced by a set of two loop variables. In Figure 19 (a), the shaded parallelogram T is the base-space for $\{i_3 = l_3\}$ and is assumed to be 2-contiguous. In order for the footprint of the nested loop to be 2-contiguous, the extended parallelogram and the base-space for $\{i_3 = l_3 + 1\}$ must overlap. They may overlap in four possible ways, represented in the figure by T_1, T_2, T_3, and T_4, depending on the coefficients of the loop variables in the index function. Combined with the 2-contiguous parallelogram, T, T_1 and



(a) Footprint of Base-Space



(b) A Tessellation

Figure 19: 2-contiguity in a 2-dimensional Space

T_3 keep the property of 2-contiguity, but T_2 and T_4 do not. From this fact, we notice that if the point A or C of the base-space in the figure is moved inside of the extended parallelogram of T, i.e., the parallelogram $P_1P_2P_3Q_3Q_2Q_1$, the footprint is 2-contiguous, where P_1, P_2, P_3, Q_1, Q_2, and Q_3 are one position left or right from A, B, C, and D accordingly. In general, the point with the minimum or maximum value in the first index should be moved into the inside of the extended tessellation.

In the 2-contiguous base-space for $\{i_3 = l_3\}$, the minimum and maximum values in the first index occur, respectively, at

$$\vec{i}_{m0} = (\nu_1, \nu_2, l_3), \quad \text{where } \nu_k = \begin{cases} l_k & \text{if } c_{1k} > 0 \\ u_k & \text{if } c_{1k} < 0 \\ l_k \text{ or } u_k & \text{if } c_{1k} = 0 \end{cases}$$

$$\vec{i}_{M0} = (\mu_1, \mu_2, l_3), \quad \text{where } \mu_k = \begin{cases} u_k & \text{if } c_{1k} > 0 \\ l_k & \text{if } c_{1k} < 0 \\ u_k \text{ or } l_k & \text{if } c_{1k} = 0 \end{cases}$$

When $i_3 = l_3 + 1$, the coordinates $H(\vec{i}_{m0})$ and $H(\vec{i}_{M0})$ on the base-space are moved to the coordinates $H(\vec{i}_{m1})$ and $H(\vec{i}_{M1})$ respectively, where

$$\vec{i}_{m1} = \vec{i}_{m0} + (0, \ldots, 0, 1), \qquad \vec{i}_{M1} = \vec{i}_{M0} + (0, \ldots, 0, 1).$$

In order for the footprint to be 2-contiguous, the coordinates $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ should be inside the extended tessellation of the base-space for $\{i_3 = l_3\}$.

According to the directions of vectors h_1 and h_2, i.e., the sign of the coefficients c_{11} and c_{12}, the base-element of the base-space can be A, B, C, or D, and we must consider each case. In all cases, it is assumed that the vectors $v_k = (u_k - l_k)h_k$, for k=1 or 2, are ordered by the angles between the x_1-axis and v_k. This is to be

consistent with the assumptions of the tessellation process.

Case 1: When $c_{11} > 0$ and $c_{12} > 0$.

In this case, the base-element is A and $v_1 = \vec{AB}$, $v_2 = \vec{AD}$. The points A and C have the minimum and maximum values in the first index, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (l_1, l_2, l_3) \quad \text{and} \quad \vec{i}_{M0} = \vec{\delta}_u = (u_1, u_2, l_3).$$

Here, $\vec{\delta}_l$ and $\vec{\delta}_u$ are equivalent to $\vec{i}_{m0}$ and $\vec{i}_{M0}$. However, it should be noted well in advance that $\vec{\delta}_l$ and $\vec{\delta}_u$ will be different from $\vec{i}_{m0}$ and $\vec{i}_{M0}$ in the next subsection.

From the figure, we have four equations of lines, P_1P_2, P_2P_3, Q_1Q_2, and Q_2Q_3 as in the following:

$$\begin{cases} c_{11}[x_2 - (g_2(\vec{\delta}_l) - 1)] = c_{21}[x_1 - g_1(\vec{\delta}_l)] & \text{for line } P_1P_2 \\ c_{12}[x_2 - (g_2(\vec{\delta}_u) - 1)] = c_{22}[x_1 - g_1(\vec{\delta}_u)] & \text{for line } P_2P_3 \\ c_{11}[x_2 - (g_2(\vec{\delta}_u) + 1)] = c_{21}[x_1 - g_1(\vec{\delta}_u)] & \text{for line } Q_2Q_3 \\ c_{12}[x_2 - (g_2(\vec{\delta}_l) + 1)] = c_{22}[x_1 - g_1(\vec{\delta}_l)] & \text{for line } Q_1Q_2 \end{cases} \tag{10}$$

In order for the footprint to be 2-contiguous, $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$, which constitutes the minimum or maximum value in the first index in the base-space for $\{i_3 = l_3 + 1\}$, should be inside the extended tessellation, where

$$\vec{i}_{m1} = \vec{i}_{m0} + (0, 0, 1), \qquad \vec{i}_{M1} = \vec{i}_{M0} + (0, 0, 1).$$

From equations (10), the condition for the footprint to be 2-contiguous is

$$c_{11}[x_2 - (g_2(\vec{\delta}_l) - 1)] \geq c_{21}[x_1 - g_1(\vec{\delta}_l)], \quad \text{and} \tag{11}$$

$$c_{12}[x_2 - (g_2(\vec{\delta}_u) - 1)] \geq c_{22}[x_1 - g_1(\vec{\delta}_u)], \quad \text{and} \tag{12}$$

$$c_{11}[x_2 - (g_2(\vec{\delta}_u) + 1)] \leq c_{21}[x_1 - g_1(\vec{\delta}_u)], \quad \text{and} \tag{13}$$

$$c_{12}[x_2 - (g_2(\vec{\delta}_l) + 1)] \leq c_{22}[x_1 - g_1(\vec{\delta}_l)] \tag{14}$$

$$\text{where} \quad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = H(\vec{i}_{m1}) \quad \text{or} \quad H(\vec{i}_{M1}).$$

Solving inequality (11) for $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$,

$$\begin{vmatrix} c_{11} & x_1 - (g_1(\vec{\delta}_l)) \\ c_{21} & x_2 - (g_2(\vec{\delta}_l) - 1) \end{vmatrix} \geq 0$$

$$\begin{vmatrix} c_{11} & x_1 - g_1(\vec{\delta}_l) \\ c_{21} & x_2 - g_2(\vec{\delta}_l) \end{vmatrix} + \begin{vmatrix} c_{11} & 0 \\ c_{21} & 1 \end{vmatrix} \geq 0$$

$$| \; h_1 \quad x - H(\vec{\delta}_l) \; | + c_{11} \geq 0$$

$$| \; h_1 \quad H(\vec{z} - \vec{\delta}_l) \; | + c_{11} \geq 0, \quad \text{where } \vec{z} = \vec{i}_{m1} \text{ or } \vec{i}_{M1}.$$

In a similar way, from inequalities (12), (13), and (14), we may derive the following conditions, where $\vec{z} = \vec{i}_{m1}$ or $\vec{i}_{M1}$,

$$| \; h_2 \quad H(\vec{z} - \vec{\delta}_u) \; | + c_{12} \geq 0,$$

$$| \; h_1 \quad H(\vec{z} - \vec{\delta}_u) \; | - c_{11} \leq 0,$$

$$| \; h_2 \quad H(\vec{z} - \vec{\delta}_l) \; | - c_{12} \leq 0.$$

In summary, the condition for the footprint to be 2-contiguous is

$$\begin{cases} \mid h_1 \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{11} \geq 0, & \text{and} \\ \mid h_2 \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{12} \geq 0, & \text{and} \\ \mid h_1 \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{11} \leq 0, & \text{and} \\ \mid h_2 \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{12} \leq 0 \end{cases} \tag{15}$$

$$\text{where } \vec{z} = \vec{i}_{m1} \text{ or } \vec{i}_{M1}.$$

Case 2: When $c_{11} > 0$ and $c_{12} < 0$.

In this case, the base-element is B and $v_1 = \vec{BC}$, $v_2 = \vec{BA}$. The points A and C have the minimum and maximum values in the first index, respectively. Therefore, the minimum and maximum values in the first index occur, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (l_1, u_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (u_1, l_2, l_3).$$

In a manner similar to case 1, we obtain a condition similar to condition (15)

$$\begin{cases} \mid h_2 \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{12} \leq 0, & \text{and} \\ \mid h_1 \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{11} \geq 0, & \text{and} \\ \mid h_2 \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{12} \leq 0, & \text{and} \\ \mid h_1 \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{11} \geq 0 \end{cases} \tag{16}$$

$$\text{where } \vec{z} = \vec{i}_{m1} \text{ or } \vec{i}_{M1}.$$

Case 3: When $c_{11} < 0$ and $c_{12} < 0$.

In this case, the base-element is C and $v_1 = \vec{CD}$, $v_2 = \vec{CB}$. The points A and C have the minimum and maximum values in the first index, respectively. Therefore,

the minimum and maximum values in the first index occur, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (u_1, u_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (l_1, l_2, l_3).$$

In a manner similar to case 1, we obtain a condition similar to condition (15)

$$\begin{cases} \mid h_1 \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{11} \leq 0, & \text{and} \\ \mid h_2 \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{12} \leq 0, & \text{and} \\ \mid h_1 \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{11} \geq 0, & \text{and} \\ \mid h_2 \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{12} \geq 0 \end{cases} \tag{17}$$

$$\text{where } \vec{z} = \vec{i}_{m1} \text{ or } \vec{i}_{M1}.$$

Case 4: When $c_{11} < 0$ and $c_{12} > 0$.
In this case, the base-element is D and $v_1 = \vec{DA}$, $v_2 = \vec{DC}$. The points A and C have the minimum and maximum values in the first index, respectively. Therefore, the minimum and maximum values in the first index occur, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (u_1, l_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (l_1, u_2, l_3).$$

In a manner similar to case 1, we obtain a condition similar to condition (15)

$$\begin{cases} \mid h_2 \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{12} \geq 0, & \text{and} \\ \mid h_1 \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{11} \leq 0, & \text{and} \\ \mid h_2 \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{12} \geq 0, & \text{and} \\ \mid h_1 \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{11} \leq 0 \end{cases} \tag{18}$$

$$\text{where } \vec{z} = \vec{i}_{m1} \text{ or } \vec{i}_{M1}.$$

Case 5: When $c_{11} = 0$ and $c_{12} > 0$.

In this case, the base-element is A and $v_1 = \vec{AB}$, $v_2 = \vec{AD}$, but AB is parallel to the axis x_2. If A, C, B, or D is moved inside of the extended parallelogram of T, then the footprint is 2-contiguous. Therefore the minimum and maximum values in the first index are regarded as the points A and C, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (l_1, l_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (u_1, u_2, l_3),$$

or as the points B and D, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (u_1, l_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (l_1, u_2, l_3).$$

Hence, the condition of *Case 1* or that of *Case 4* above can be applied to this case.

Case 6: When $c_{11} = 0$ and $c_{12} < 0$.

In this case, the base-element is C and $v_1 = \vec{CD}$, $v_2 = \vec{CB}$, but CD is parallel to the axis x_2. If A, C, B, or D is moved inside of the extended parallelogram of T, then the footprint is 2-contiguous. Therefore the minimum and maximum values in the first index are regarded as the points A and C, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (u_1, u_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (l_1, l_2, l_3),$$

or as the points B and D, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (l_1, u_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (u_1, l_2, l_3).$$

Hence, the condition of *Case 3* or that of *Case 2* above can be applied to this case.

Case 7: When $c_{11} > 0$ and $c_{12} = 0$.

In this case, the base-element is A and $v_1 = \vec{AB}$, $v_2 = \vec{AD}$, but AD is parallel to the

axis x_2. If A, C, B, or D is moved inside of the extended parallelogram of T, then the footprint is 2-contiguous. Therefore the minimum and maximum values in the first index are regarded as the points A and C, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (l_1, l_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (u_1, u_2, l_3),$$

or as the points B and D, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (l_1, u_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (u_1, l_2, l_3).$$

Hence, the condition of *Case 1* or that of *Case 2* above can be applied to this case.

Case 8: When $c_{11} < 0$ and $c_{12} = 0$.
In this case, the base-element is C and $v_1 = \vec{CD}$, $v_2 = \vec{CB}$, but CB is parallel to the axis x_2. If A, C, B, or D is moved inside of the extended parallelogram of T, then the footprint is 2-contiguous. Therefore the minimum and maximum values in the first index are regarded as the points A and C, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (l_1, l_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (u_1, u_2, l_3),$$

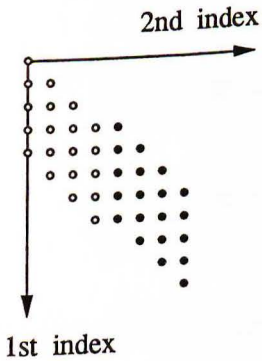
or as the points B and D, respectively, at

$$\vec{i}_{m0} = \vec{\delta}_l = (u_1, l_2, l_3) \text{ and } \vec{i}_{M0} = \vec{\delta}_u = (l_1, u_2, l_3).$$

Hence, the condition of *Case 3* or that of *Case 4* above can be applied to this case.

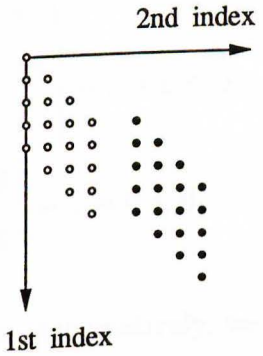
For an example of *Case 1*, see Figure 20. There are two nested loops, of which (a) is 2-contiguous, but (b) is not. Both have a nested loop induced by $\{i_1, i_2\}$ whose footprint is 2-contiguous. The base-space for $\{i_3 = 0\}$, the set of points marked with o in the picture, has the minimum and maximum values in the first index, respectively,

```
for i1 = 0,4
   for i2 = 0,3
      for i3 = 0,1
         X(i1 + i2 + 3i3, i2 + 4i3)
      endfor
   endfor
endfor
```



(a)

```
for i1 = 0,4
   for i2 = 0,3
      for i3 = 0,1
         X(i1 + i2 + 3i3, i2 + 5i3)
      endfor
   endfor
endfor
```



(b)

Figure 20: 2-contiguity of 2-dim Array When Nesting Depth is 3

at

$$\vec{i}_{m0} = \vec{\delta}_l = (0,0,0) \quad \text{and} \quad \vec{i}_{M0} = \vec{\delta}_u = (4,3,0).$$

The base-space for $\{i_3 = 1\}$ has the minimum and maximum values in the first index, respectively, at

$$\vec{i}_{m1} = \vec{i}_{m0} + (0,0,1) = (0,0,1) \quad \text{and} \quad \vec{i}_{M1} = \vec{i}_{M0} + (0,0,1) = (4,3,1).$$

First, we will show that the footprint of the nested loop in (a) is 2-contiguous. If we

apply $\vec{z} = \vec{i}_{m1} = (0,0,1)$ to condition (15), we get the following result:

$$\begin{cases} \left| h_1 \;\; H(\vec{z} - \vec{\delta}_l) \right| + c_{11} = \begin{vmatrix} 1 & 3 \\ 0 & 4 \end{vmatrix} + 1 = 5 \geq 0 \\ \left| h_2 \;\; H(\vec{z} - \vec{\delta}_u) \right| + c_{12} = \begin{vmatrix} 1 & -4 \\ 1 & 1 \end{vmatrix} + 1 = 6 \geq 0 \\ \left| h_1 \;\; H(\vec{z} - \vec{\delta}_u) \right| - c_{11} = \begin{vmatrix} 1 & -4 \\ 0 & 1 \end{vmatrix} - 1 = 0 \leq 0 \\ \left| h_2 \;\; H(\vec{z} - \vec{\delta}_l) \right| - c_{12} = \begin{vmatrix} 1 & 3 \\ 1 & 4 \end{vmatrix} - 1 = 0 \leq 0 \end{cases}$$

The result satisfies condition (15) for 2-contiguity. Alternatively, we will show that the footprint of the nested loop in (b) is not 2-contiguous. The base-space for $\{i_3 = 1\}$ has the minimum and maximum values in the first index, respectively, at

$$\vec{i}_{m1} = \vec{i}_{m0} + (0,0,1) = (0,0,1) \quad \text{and} \quad \vec{i}_{M1} = \vec{i}_{M0} + (0,0,1) = (4,3,1).$$

Applying $\vec{z} = \vec{i}_{m1} = (0,0,1)$ to condition (15), we have,

$$\begin{cases} \left| h_1 \;\; H(\vec{z} - \vec{\delta}_l) \right| + c_{11} = \begin{vmatrix} 1 & 3 \\ 0 & 5 \end{vmatrix} + 1 = 6 \geq 0 \\ \left| h_2 \;\; H(\vec{z} - \vec{\delta}_u) \right| + c_{12} = \begin{vmatrix} 1 & -4 \\ 1 & 2 \end{vmatrix} + 1 = 7 \geq 0 \\ \left| h_1 \;\; H(\vec{z} - \vec{\delta}_u) \right| - c_{11} = \begin{vmatrix} 1 & -4 \\ 0 & 2 \end{vmatrix} - 1 = 1 \not\leq 0 \\ \left| h_2 \;\; H(\vec{z} - \vec{\delta}_l) \right| - c_{12} = \begin{vmatrix} 1 & 3 \\ 1 & 5 \end{vmatrix} - 1 = 1 \not\leq 0 \end{cases}$$

The result does not satisfy condition (15) for 2-contiguity. If we use $\vec{z} = \vec{i}_{M1} =$

$(4, 3, 1)$, we have the following:

$$\begin{cases} \left| h_1 \quad H(\vec{z} - \vec{\delta}_l) \right| + c_{11} = \begin{vmatrix} 1 & 10 \\ 0 & 8 \end{vmatrix} + 1 = 9 \geq 0 \\ \left| h_2 \quad H(\vec{z} - \vec{\delta}_u) \right| + c_{12} = \begin{vmatrix} 1 & 3 \\ 1 & 5 \end{vmatrix} + 1 = 3 \geq 0 \\ \left| h_1 \quad H(\vec{z} - \vec{\delta}_u) \right| - c_{11} = \begin{vmatrix} 1 & 3 \\ 0 & 5 \end{vmatrix} - 1 = 4 \not\leq 0 \\ \left| h_2 \quad H(\vec{z} - \vec{\delta}_l) \right| - c_{12} = \begin{vmatrix} 1 & 10 \\ 1 & 8 \end{vmatrix} - 1 = -3 \leq 0 \end{cases}$$

This result also does not satisfy condition (15) for 2-contiguity.

4.5.2 Contiguity When Nesting Depth is Greater Than 3

When $d = 2$ and $n > 3$, the index function of the nested loop, in matrix form, is

$$H = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} h_1 & \cdots & h_n \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ c_{21} & \cdots & c_{2n} \end{pmatrix}.$$

From Theorem 7, if the footprint is 2-contiguous, there should exist a nested loop induced by a set of $(n-1)$ loop variables whose footprint is 2-contiguous. In this section, the condition for the footprint of a nested loop to be 2-contiguous is deduced, provided that the footprint of a nested loop induced by a set of loop variables $\{i_1, \ldots, i_{n-1}\}$ is 2-contiguous.

In the 2-contiguous footprint of the nested loop induced by a set of loop variables $\{i_1, \ldots, i_{n-1}\}$, there exist $\binom{n-1}{2}$ tessellations, $R_{b,t}$ for $1 \leq b < t \leq n-1$, that are disjoint parallelograms except at the boundaries. In the footprint of the base-space for $(i_n = l_n)$, the minimum and maximum values in the first index occur, respectively,

at

$$\vec{i}_{m0} = (\nu_1, \ldots, \nu_{n-1}, l_n), \quad \text{where } \nu_k = \begin{cases} l_k & \text{if } c_{1k} > 0 \\ u_k & \text{if } c_{1k} < 0 \\ l_k \text{ or } u_k & \text{if } c_{1k} = 0 \end{cases}$$

$$\vec{i}_{M0} = (\mu_1, \ldots, \mu_{n-1}, l_n), \quad \text{where } \mu_k = \begin{cases} u_k & \text{if } c_{1k} > 0 \\ l_k & \text{if } c_{1k} < 0 \\ u_k \text{ or } l_k & \text{if } c_{1k} = 0 \end{cases}$$

When $i_n = l_n + 1$, the coordinates $H(\vec{i}_{m0})$ and $H(\vec{i}_{M0})$ on the base-space are moved to the coordinates $H(\vec{i}_{m1})$ and $H(\vec{i}_{M1})$ respectively, where

$$\vec{i}_{m1} = \vec{i}_{m0} + (0, \ldots, 0, 1), \qquad \vec{i}_{M1} = \vec{i}_{M0} + (0, \ldots, 0, 1).$$

In order for the footprint to be 2-contiguous, the coordinates $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ should be inside one of the extended tessellations on the base-space.

Let us think about a tessellation $R_{b,t}$. In a similar way to the case when $d = 2$ and $n = 3$, we must consider this kind of tessellation according to the shape of the tessellation which is characterized by the sign of coefficients c_{1b} and c_{1t}. However, because the base-element is not part of a tessellation, we must handle it a little differently from our approach when $d = 2$ and $n = 3$.

Case 1: When $c_{1b} > 0$ and $c_{1t} > 0$.

For example, see Figure 19 (b), which shows a tessellation $R_{b,t}$ of this case. The point O is the base-element and its coordinate is $H(\vec{l})$, where $\vec{l} = (l_1, \ldots, l_n)$. The coordinate of the point A, which has the minimum value in the first index, is $H(\vec{l}) + v_{inside}^{(b,t)}$, and the coordinate of the point C, which has the maximum value in the first index, is

$H(\vec{l}) + v_{inside}^{(b,t)} + v_b + v_t$.

$$\begin{aligned}
H(\vec{l}) + v_{inside}^{(b,t)} &= H(l_1, \ldots, l_n) + \sum_{v_j:\ \text{term of } v_{inside}^{(b,t)}} v_j \\
&= \sum_{j=1}^{n} l_j h_j + \sum_{v_j:\ \text{term of } v_{inside}^{(b,t)}} (u_j - l_j) h_j \\
&= \sum_{j=1}^{n} \delta_j h_j \ = \ H(\vec{\delta}_o) \\
\text{where } \vec{\delta}_o &= (\delta_1, \ldots, \delta_n), \\
\delta_j &= \begin{cases} l_j & \text{if } v_j \text{ is not term of } v_{inside}^{(b,t)} \\ u_j & \text{if } v_j \text{ is term of } v_{inside}^{(b,t)} \end{cases} \\
H(\vec{l}) + v_{inside}^{(b,t)} + v_b + v_t &= H(\vec{\delta}_o) + (u_b - l_b)h_b + (u_t - l_t)h_t \ = \ H(\vec{\delta}_u) \\
\text{where } \vec{\delta}_u &= \vec{\delta}_o + (0, \ldots, 0, (u_b - l_b), 0, \ldots, 0, (u_t - l_t), 0, \ldots, 0).
\end{aligned}$$

So the minimum and maximum values in the first index occur, respectively, at

$$\begin{aligned}
\vec{\delta}_l &= \vec{\delta}_o \ \text{ and} \\
\vec{\delta}_u &= \vec{\delta}_o + (0, \ldots, 0, (u_b - l_b), 0, \ldots, 0, (u_t - l_t), 0, \ldots, 0).
\end{aligned}$$

The condition for either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ to be inside the extended tessellation of $R_{b,t}$ is, from condition (15),

$$\begin{cases}
\mid h_b \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{1b} \geq 0 \\
\mid h_t \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{1t} \geq 0 \\
\mid h_b \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{1b} \leq 0 \\
\mid h_t \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{1t} \leq 0
\end{cases} \tag{19}$$

$$\text{where } \vec{z} = \vec{i}_{m1} \text{ or } \vec{i}_{M1}.$$

Case 2: When $c_{1b} > 0$ and $c_{1t} < 0$.

In this case, the tessellation has the minimum and maximum values in the first index, respectively,

$$H(\vec{l}) + v_{inside}^{(b,t)} + v_t \quad \text{and} \quad H(\vec{l}) + v_{inside}^{(b,t)} + v_b.$$

Therefore, the minimum and maximum values in the first index occur, respectively, at

$$\begin{aligned} \vec{\delta}_l &= \vec{\delta}_o + (0, \ldots, 0, (u_t - l_t), 0, \ldots, 0) \quad \text{and} \\ \vec{\delta}_u &= \vec{\delta}_o + (0, \ldots, 0, (u_b - l_b), 0, \ldots, 0). \end{aligned}$$

The condition for either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ to be inside the extended tessellation of $R_{b,t}$ is, from condition (16),

$$\begin{cases} \mid h_t \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{1t} \leq 0 \\ \mid h_b \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{1b} \geq 0 \\ \mid h_t \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{1t} \leq 0 \\ \mid h_b \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{1b} \geq 0 \end{cases} \tag{20}$$

$$\text{where} \quad \vec{z} = \vec{i}_{m1} \quad \text{or} \quad \vec{i}_{M1}.$$

Case 3: When $c_{1b} < 0$ and $c_{1t} < 0$.

In this case, the tessellation has the minimum and maximum values in the first index, respectively,

$$H(\vec{l}) + v_{inside}^{(b,t)} + v_b + v_t \quad \text{and} \quad H(\vec{l}) + v_{inside}^{(b,t)}.$$

Therefore, the minimum and maximum values in the first index occur, respectively,

at

$$\begin{aligned}\vec{\delta}_l &= \vec{\delta}_o + (0,\ldots,0,(u_b - l_b),0,\ldots,0,(u_t - l_t),0,\ldots,0) \quad \text{and} \\ \vec{\delta}_u &= \vec{\delta}_o.\end{aligned}$$

The condition for either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ to be inside the extended tessellation of $R_{b,t}$ is, from condition (17),

$$\begin{cases} \mid h_b \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{1b} \leq 0 \\ \mid h_t \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{1t} \leq 0 \\ \mid h_b \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{1b} \geq 0 \\ \mid h_t \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{1t} \geq 0 \end{cases} \tag{21}$$

$$\text{where} \quad \vec{z} = \vec{i}_{m1} \quad \text{or} \quad \vec{i}_{M1}.$$

Case 4: When $c_{1b} < 0$ and $c_{1t} > 0$.

In this case, the tessellation has the minimum and maximum values in the first index, respectively,

$$H(\vec{l}) + v_{inside}^{(b,t)} + v_b \quad \text{and} \quad H(\vec{l}) + v_{inside}^{(b,t)} + v_t.$$

Therefore, the minimum and maximum values in the first index occur, respectively, at

$$\begin{aligned}\vec{\delta}_l &= \vec{\delta}_o + (0,\ldots,0,(u_b - l_b),0,\ldots,0) \quad \text{and} \\ \vec{\delta}_u &= \vec{\delta}_o + (0,\ldots,0,(u_t - l_t),0,\ldots,0).\end{aligned}$$

The condition for either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ to be inside the extended

tessellation of $R_{b,t}$ is, from condition (18),

$$\begin{cases} \mid h_t \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{1t} \geq 0 \\ \mid h_b \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{1b} \leq 0 \\ \mid h_t \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{1t} \geq 0 \\ \mid h_b \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{1b} \leq 0 \end{cases} \tag{22}$$

$$\text{where } \vec{z} = \vec{i}_{m1} \text{ or } \vec{i}_{M1}.$$

Case 5: When $c_{11} = 0$ and $c_{12} > 0$.

As we saw previously when $d = 2$ and $n = 3$, the condition for *Case 1* or *Case 4* above can be applied to this case.

Case 6: When $c_{11} = 0$ and $c_{12} < 0$.

The condition for *Case 3* or *Case 2* above can be applied to this case.

Case 7: When $c_{11} > 0$ and $c_{12} = 0$.

The condition for *Case 1* or *Case 2* above can be applied to this case.

Case 8: When $c_{11} < 0$ and $c_{12} = 0$.

The condition for *Case 3* or *Case 4* above can be applied to this case.

In summary, if there exists any tessellation such that either the coordinate $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ is inside one of the extended tessellations on the base-space for $(i_n = l_n)$, then the footprint is 2-contiguous.

For example, see Figure 21. The nesting depth of the nested loop is four. As shown in the diagram, the base-space for $\{i_4 = 0\}$, marked with o in the illustration, is 2-contiguous, and divided into three tessellations. With the terminology defined in Chapter 3,

$$v_1 = 3\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}, \quad v_2 = 5\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 5 \end{pmatrix}, \quad v_3 = 4\begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} -4 \\ 4 \end{pmatrix},$$

```
for i1 = 0,3
   for i2 = 0,5
      for i3 = 0,4
         for i4 = 0,1
            X(i1 − i3 + i4, i2 + i3 + 8i4)
         endfor
      endfor
   endfor
endfor
```

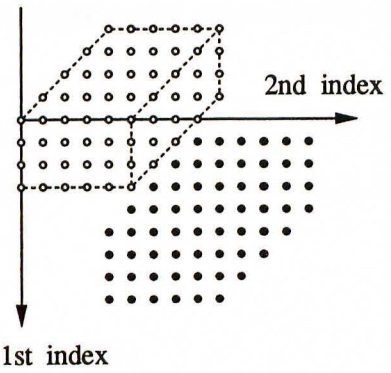


Figure 21: 2-contiguity of 2-dim Array in General Case

$$
\begin{aligned}
R_{1,2} &= \{\alpha_1 v_1 + \alpha_2 v_2 \mid \alpha_1, \alpha_2 \in [0,1]\} \\
R_{2,3} &= \{\alpha_2 v_2 + \alpha_3 v_3 \mid \alpha_2, \alpha_3 \in [0,1]\} \\
R_{1,3} &= \{\alpha_1 v_1 + \alpha_3 v_3 + v_{inside}^{(1,3)} \mid \alpha_1, \alpha_3 \in [0,1]\} \\
&= \{\alpha_1 v_1 + \alpha_3 v_3 + v_2 \mid \alpha_1, \alpha_3 \in [0,1]\}
\end{aligned}
$$

The base-space has the minimum and maximum values in the first index, respectively, at

$$\vec{i}_{m0} = (l_1, l_2, u_3, l_4) = (0,0,4,0) \quad \text{and} \quad \vec{i}_{M0} = (u_1, u_2, l_3, l_4) = (3,5,0,0).$$

If the footprint of the nested loop is 2-contiguous, the point $H(\vec{i}_{m1})$ or $H(\vec{i}_{M1})$ is inside one of the extended tesselleations, where

$$\vec{i}_{m1} = \vec{i}_{m0} + (0,0,0,1) = (0,0,4,1), \qquad \vec{i}_{M1} = \vec{i}_{M0} + (0,0,0,1) = (3,5,0,1).$$

In the diagram, the point $H(\vec{i}_{m1})$ is clearly inside the extended tessellation of $R_{1,3}$. The tessellation is spanned by v_1 and v_3. Because $c_{11} > 0$ and $c_{13} < 0$, the tessellation falls into *Case 2*, and the corresponding value of $\vec{\delta}_o$ is

$$\vec{\delta}_o = (l_1, u_2, l_3, l_4) = (0,5,0,0).$$

The minimum and maximum values in the first index occur, respectively, at

$$\vec{\delta}_l = \vec{\delta}_o + (0,\ldots,0,(u_t - l_t),0,\ldots,0) = (0,5,0,0) + (0,0,4,0) = (0,5,4,0) \text{ and}$$
$$\vec{\delta}_u = \vec{\delta}_o + (0,\ldots,0,(u_b - l_b),0,\ldots,0) = (0,5,0,0) + (3,0,0,0) = (3,5,0,0).$$

Applying $\vec{z} = \vec{i}_{m1} = (0,0,4,1)$ to condition (20), we have,

$$\begin{cases} \mid h_t \quad H(\vec{z} - \vec{\delta}_l) \mid + c_{1t} = \begin{vmatrix} -1 & 1 \\ 1 & 3 \end{vmatrix} = -5 \leq 0 \\ \mid h_b \quad H(\vec{z} - \vec{\delta}_u) \mid + c_{1b} = \begin{vmatrix} 1 & -6 \\ 0 & 7 \end{vmatrix} = 8 \geq 0 \\ \mid h_t \quad H(\vec{z} - \vec{\delta}_u) \mid - c_{1t} = \begin{vmatrix} -1 & -6 \\ 1 & 7 \end{vmatrix} = 0 \leq 0 \\ \mid h_b \quad H(\vec{z} - \vec{\delta}_l) \mid - c_{1b} = \begin{vmatrix} 1 & 1 \\ 0 & 3 \end{vmatrix} = 2 \geq 0 \end{cases}$$

This result satisfies condition (20) for the point $H(\vec{i}_{m1})$ to be inside the extended tessellation $R_{1,3}$, thus determining that the footprint is 2-contiguous.

Chapter 5

Localization of Contiguously Accessed Data

5.1 Loop Transformations

In the previous chapter, conditions for the footprint of a nested loop to be d-contiguous when $d = 1$ or 2 were developed. Based on those conditions, in this chapter algorithms are developed to decide the proper location for a block copy statement to be inserted. The template of a parallelized nested loop used for this section is given in Figure 22.

The first step to check if the footprint is d-contiguous is to look into the index function, and search for a set of loop variables $\{i_{k_1}, \ldots, i_{k_d}\}$ with the coefficients satisfying the conditions described in Theorem 6. If any such set exists, we set aside the loop variables. (If not, the footprint is not 2-contiguous.) Next, we are to look

```
      forall i_0 = l_0, u_0
I_1 :   for i_1 = l_1, u_1
 ⋮        ...
I_n :       for i_n = l_n, u_n
                X(c_11 i_1 + ··· + c_1n i_n, ..., c_d1 i_1 + ··· + c_dn i_n)
            endfor
          ...
      endfor
    endforall
```

Figure 22: Parallelized Nested Loop Code

into the index function for a loop variable $i_{k_{d+1}}$ with coefficients satisfying condition (9) if $d = 1$, or one of conditions (15), (16), (17), and (18) if $d = 2$. We repeat this process until the searching process fails or all eligible loop variables are selected. When the process stops, if there is no remaining loop variables unselected, the footprint is considered d-contiguous. If so, a block copy statement can be inserted before the loop statement I_1.

Even if the footprint is not d-contiguous, we may still use block copy to localize data for a set of some loops, that is, when the footprint of the nested loop induced by a set of loop variables is d-contiguous and their loops are inside the other loops. To select the loop variables of the most innermost loops we can reach, we therefore need to select the innermost loop variable whenever possible. When the process terminates, we then divide the selected loop variables into two sets,

$$G_1 = \{i_{k_1}, \ldots, i_{k_s}\} \quad \text{and} \quad G_2 = \{i_{k_{s+1}}, \ldots, i_{k_t}\},$$

so that G_1 and G_2 satisfy the following criteria:

- $i_n \in G_1$.
- $k_j > k_l$, for $j \in \{1, \ldots, s\}, l \in \{s+1, \ldots, t\}$.
- Subscripts of elements in G_1 are consecutive integers, i.e. $\{i_{k_1}, \ldots, i_{k_s}\} = \{i_{n-s+1}, \ldots, i_n\}$.
- The number of elements in G_1 is maximized.

The above criteria are required to make sure that the loop variables in G_1 are from the innermost loops. If they are not from the innermost loops, the data cannot be localized by using a block copy. But, if there does exist such a set of loop variables to satisfy the above criteria, let $G_1 = \{i_j, \ldots, i_n\}$. During the execution of the nested

```
forall i_0 = l_0, u_0
   dimension Xtmp(···)
loop
   for i_1 = l_1, u_1
      ···
         for i_{j-1} = l_{j-1}, u_{j-1}
           block copy from X to Xtmp
           for i_j = l_j, u_j
             ···
                for i_n = l_n, u_n
                   Xtmp(c_11 i_1 + ··· + c_1n i_n, ..., c_d1 i_1 + ··· + c_dn i_n)
                endfor
             ···
           endfor
         endfor
      ···
   endfor
endforall
```

Figure 23: Data Localization by Block Copy

loop for loop variables in the set G_1, the values of the loop variables $i_1, \ldots, i_{j-1}$ are fixed, and the referenced data are d-contiguous. Therefore, a block copy statement can be inserted before the loop statement I_j, as shown in Figure 23.

5.1.1 Localization of a 1-dimensional Array

When a 1-dimensional array is referenced in a nested loop, and $G_1 = \{i_j, \ldots, i_n\}$, the footprint of the nested loop induced by those loop variables is 1-contiguous. Since the array is 1-dimensional, the 1-contiguous data form a single block. To localize the data by block copy, the lower and upper bounds of the block, l and u, can be computed as follows:

$$l = g_1(\vec{i}_m), \quad u = g_1(\vec{i}_M)$$

$$\text{where } \vec{i}_m = (i_1, \ldots, i_{j-1}, \nu_j, \ldots, \nu_n), \quad \nu_k = \begin{cases} l_k & \text{if } c_{1k} \geq 0 \\ u_k & \text{if } c_{1k} < 0 \end{cases}$$

$$\vec{i}_M = (i_1, \ldots, i_{j-1}, \mu_j, \ldots, \mu_n), \quad \mu_k = \begin{cases} u_k & \text{if } c_{1k} \geq 0 \\ l_k & \text{if } c_{1k} < 0 \end{cases}$$

For example, consider Figure 24. In the first step of the process to select a loop variable with the coefficient of absolute value 1, select i_4 rather than i_1 because the loop for i_4 is inside the loop for i_1. In the second step, there are three possible choices, i_1, i_3, and i_5. To satisfy condition (9), choose i_5, i_3, and i_1, in that order. In the next step, since there are no more loop variables satisfying condition (9), stop the process. Because the loop variable i_2 is not selected, the footprint is not 1-contiguous. So far, the loop variables are selected in the order i_4, i_5, i_3, and i_1. According to the above criteria, we then divide the selected loop variables into two sets,

$$G_1 = \{i_4, i_5, i_3\} \text{ and } G_2 = \{i_1\}.$$

The lower and upper bound of the 1-contiguous block are

$$\begin{aligned} l &= g_1(i_1, i_2, u_3, l_4, l_5) \\ &= i_1 + 14i_2 + (-2)2 + (1)0 + (2)0 = i_1 + 14i_2 - 4 \quad \text{and} \\ u &= g_1(i_1, i_2, l_3, u_4, u_5) \\ &= i_1 + 14i_2 + (-2)0 + (1)2 + (2)2 = i_1 + 14i_2 + 6. \end{aligned}$$

With this result, the array can be localized by inserting a block copy statement just after the loop statement for i_2, as shown in Figure 24 (b).

```
forall i0 = 0,9
   for i1 = 0,2
      for i2 = 0,2
         for i3 = 0,2
            for i4 = 0,2
               for i5 = 0,2
                  X(i1 + 14i2 − 2i3 + i4 + 2i5)
               endfor
            endfor
         endfor
      endfor
   endfor
endforall
```

(a) Given parallelized nested loop

```
forall i0 = 0,9
   dimension Xtmp(···)
loop
   for i1 = 0,2
      for i2 = 0,2
         l = i1 + 14i2 − 4
         u = i1 + 14i2 + 6
         bcopy(&X(l), &Xtmp(l), (u − l + 1))
         for i3 = 0,2
            for i4 = 0,2
               for i5 = 0,2
                  Xtmp(i1 + 14i2 − 2i3 + i4 + 2i5)
               endfor
            endfor
         endfor
      endfor
   endfor
endforall
```

(b) Localization by block copy

Figure 24: Localization of a 1-dimensional Array

5.1.2 Localization of a 2-dimensional Array

When a 2-dimensional array is referenced in a nested loop, and $G_1 = \{i_j, \ldots, i_n\}$, the footprint of the nested loop induced by the set G_1 is 2-contiguous. Since the array is 2-dimensional, there are several rows in the footprint, each of which has a single 2-contiguous block. To localize the data by a block copy, we first have to know the minimum and maximum index values of each row of the footprint. Similarly to the case of the 1-dimensional array, they can be computed as follows:

$$row_l = g_1(\vec{i}_m), \quad row_u = g_1(\vec{i}_M)$$

$$\text{where} \quad \vec{i}_m = (i_1, \ldots, i_{j-1}, \nu_j, \ldots, \nu_n), \quad \nu_k = \begin{cases} l_k & \text{if } c_{1k} > 0 \\ u_k & \text{if } c_{1k} < 0 \\ l_k \text{ or } u_k & \text{if } c_{1k} = 0 \end{cases}$$

$$\vec{i}_M = (i_1, \ldots, i_{j-1}, \mu_j, \ldots, \mu_n), \quad \mu_k = \begin{cases} u_k & \text{if } c_{1k} > 0 \\ l_k & \text{if } c_{1k} < 0 \\ u_k \text{ or } l_k & \text{if } c_{1k} = 0 \end{cases}$$

For each row of the index r between row_l and row_u, the lower and upper bounds of the 2-contiguous data block should be pre-determined if we want to effect the block copy. To find the lower and upper bounds, l and u, of the 2-contiguous data block on the row r, we derive the following integer linear programming problem:

$$\text{Solve :} \quad \begin{cases} l = \min(c_{21}i_1 + \cdots + c_{2j}i_j + \cdots + c_{2n}i_n) \\ u = \max(c_{21}i_1 + \cdots + c_{2j}i_j + \cdots + c_{2n}i_n) \end{cases}$$

$$\text{subject to :} \quad \begin{cases} c_{11}i_1 + \cdots + c_{1j}i_j + \cdots + c_{1n}i_n = r, \quad \text{for } row_l \le r \le row_u \\ l_k \le i_k \le u_k, \quad \text{for } k = j, \ldots, n \end{cases}$$

Because the value of the loop variables $i_1, \ldots, i_{j-1}$ are fixed inside the loop statement I_{j-1}, the expressions of those terms are equivalent to the constant terms, where

$$C_1 = c_{11}i_1 + \cdots + c_{1,j-1}i_{j-1}, \quad C_2 = c_{21}i_1 + \cdots + c_{2,j-1}i_{j-1}.$$

Then, the above integer linear programming problem can be rewritten into:

$$\text{Solve :} \quad \begin{cases} l = C_2 + \min(c_{2j}i_j + \cdots + c_{2n}i_n) \\ u = C_2 + \max(c_{2j}i_j + \cdots + c_{2n}i_n) \end{cases}$$

$$\text{subject to :} \quad \begin{cases} c_{1j}i_j + \cdots + c_{1n}i_n = r - C_1, \quad \text{for } row_l \leq r \leq row_u \\ l_k \leq i_k \leq u_k, \quad \text{for } k = j, \ldots, n \end{cases}$$

For example, see Figure 25. In the first step of the process to select a set of two loop variables with the coefficients to satisfying the condition described in Theorem 6, i_4 and i_2 are selected. In the second step, i_3, whose coefficient satisfies condition (15), is selected. In the next step, since there are no more loop variables that satisfy any of conditions (15), (16), (17), and (18), the process terminates. Because there is a loop variable i_1 which is not selected, the footprint is not 2-contiguous. So far, the loop variables i_4, i_2, and i_3 are selected in that order. According to the criteria, we divide the selected loop variables into two sets,

$$G_1 = \{i_4, i_2, i_3\} \quad \text{and} \quad G_2 = \phi.$$

The minimum and maximum index values of rows that have a 2-contiguous data block are

$$\begin{aligned} row_l &= g_1(i_1, l_2, l_3, l_4) = i_1 + (1)0 + (1)0 + (1)0 = i_1 \quad \text{and} \\ row_u &= g_1(i_1, u_2, u_3, u_4) = i_1 + (1)2 + (1)2 + (1)2 = i_1 + 6. \end{aligned}$$

```
forall i0 = 0,9
   for i1 = 0,2
      for i2 = 0,2
         for i3 = 0,2
            for i4 = 0,2
               X(i1 + i2 + i3 + i4, 5i1 + i2)
            endfor
         endfor
      endfor
   endfor
endforall
```

(a) Given parallelized nested loop

```
forall i0 = 0,9
   dimension Xtmp(···)
loop
   for i1 = 0,2
      rowl = i1
      rowu = i1 + 6
      for r = rowl, rowu
         get_block_bound(r, &l, &u)
         if (l ≤ u) bcopy(&X(r, l), &Xtmp(r, l), (u − l + 1))
      endfor
      for i2 = 0,2
         for i3 = 0,2
            for i4 = 0,2
               Xtmp(i1 + i2 + i3 + i4, 5i1 + i2)
            endfor
         endfor
      endfor
   endfor
endforall
```

(b) Localization by block copy

Figure 25: Localization of a 2-dimensional Array

```
forall i0 = 0,9
   dimension Xtmp(···)
loop
   for i2 = 0,2
      l = 14i2 − 2
      u = 14i2 + 8
      bcopy(&X(l), &Xtmp(l), (u − l + 1))
      for i1 = 0,2
         for i3 = 0,2
            for i4 = 0,2
               for i5 = 0,2
                  Xtmp(i1 + 14i2 − 2i3 + i4 + 2i5)
               endfor
            endfor
         endfor
      endfor
   endfor
endforall
```

Figure 26: Improved Localization

With this result, the array can be localized by inserting a block copy statement just after the loop statement for i_1, as shown in Figure 25 (b).

5.2 Loop Interchange

So far, a block copy statement is inserted before a loop statement only when the footprint of a nested loop induced by loops enclosed by that loop is d-contiguous. However, there may be some other loop variables whose footprint is d-contiguous but whose loop statements do not occur in the innermost loops. In such cases, the block copy method cannot be used to localize array data. Still, the same method can be used, provided only that interchanging loops to move the loops whose footprints are d-contiguous to the innermost loops does not violate the data dependence rule for

parallel execution [68], which requires that the data dependence should be forward directed in the execution sequence.

For example, reconsider Figure 24. In the process of selecting loop variables to satisfy the 1-contiguous condition, i_4, i_5, i_3, and i_1 are selected. By contrast, the loop of i_1 is not selected and not inside the loop of i_2. Even though the loop of i_1 can contribute to form a larger 1-contiguous block, it is excluded from the set G_1. Nevertheless, if it can be shown that there is no dependence conflict resulting from interchanging the loops of i_1 and i_2, localization can be done more effectively by doing so, as shown in Figure 26.

Chapter 6

Localization of Non-contiguously Accessed Data

In this chapter, we describe how to localize data that are not contiguously accessed in a nested loop as in Figure 5. When the data referenced in a nested loop is not contiguous, we can not utilize a fast block copy. Instead, we have to copy elements word-by-word if we do intend to localize them.

In a nested loop with an array reference, let r be the *rank* of the mapping function; i.e., $rank(H) = r$. When the nesting depth of a nested loop is equal to the rank of the index function for the array reference, i.e., $n = r$, the number of accesses to the array is equal to that of the referenced elements; thus, we cannot expect performance improvement by data localization, because the localization time for copying word-by-word is same as the time for accessing to the array during computation. If the nesting depth is greater than the rank of the index function, i.e., $n > r$, the total number of accesses to the array is much greater than the number of the referenced array elements. In such a case, therefore, if we can accurately take into account the overall system overhead, which depends on the relative access time to remote and local memory, then copying elements word-by-word from remote to local memory and computing with local data may result in better system performance. Therefore, throughout this chapter, we assume that $n > r$.

6.1 The Thickened Face

6.1.1 Concept of Thickened Face

This subsection is mostly based on the work of Gallivan, Jalby, and Gannon [24]. The main concept of *thickened face* found in this paper will be summarized here to give a theoretical basis to our extension.

In order to localize all the elements of an array, say X, we first declare a temporary array $Xtmp$ in local memory. The simplest implementation would be to declare $Xtmp$ as having the same shape as that of the original array X so that we may use the same index functions.

The problem is to copy the exact set of elements referenced in the nested loop into the temporary array as efficiently as possible. One way to do this is to get faces in the domain, i.e., iteration space, and to copy the images of the faces, i.e., tessellations, to the corresponding area of the temporary local array $Xtmp$.

The main problem is to characterize the geometrical structure of the referenced array elements, $H(D)$, by considering the mapping of faces, F, of the domain, D, in Z^n. Let the set of r-dimensional faces generated by r iteration variables on the iteration space be defined by

$$\mathcal{F} = \{F \mid F = \prod_{j=1}^{r} D_{k_j}, \text{ where } k_j \in \{1, \ldots, n\}\}.$$

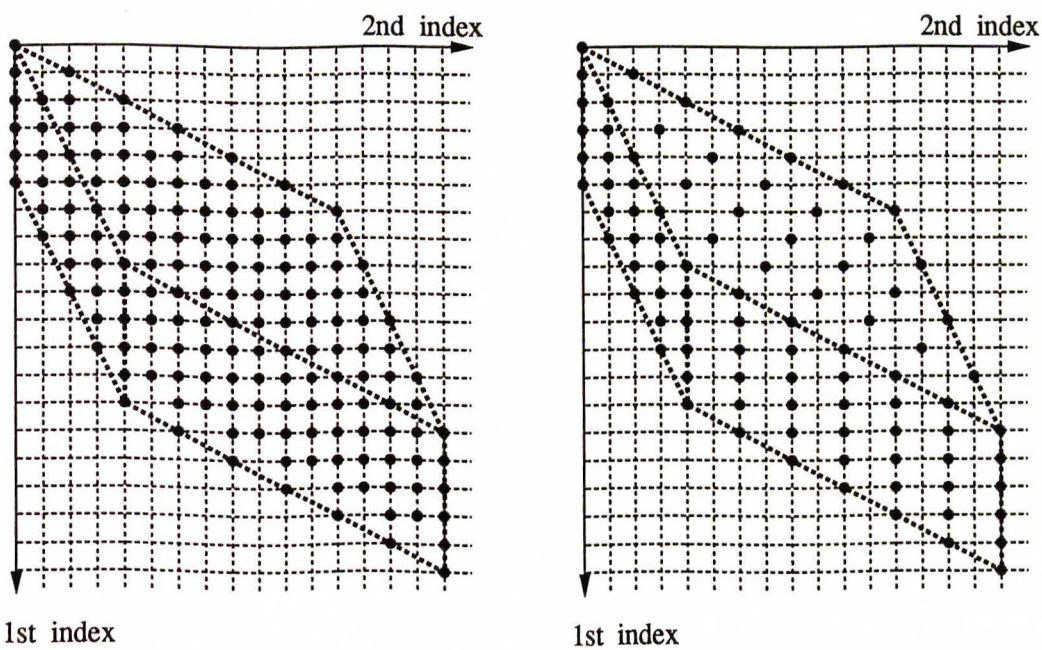
Then,

$$H(D) = \bigcup_{F \in \mathcal{F}} H(F).$$

In the domain D, there are many faces, but as we saw in Chapter 3, we can select $\binom{n}{r}$ faces so that the whole range can be covered with the images of those selected faces, the tessellations, that are mutually disjoint except at the boundaries.

```
for i1 = 0,5
   for i2 = 0,4
      for i3 = 0,6
         X(i1 + 2i2 + i3, i2 + 2i3)
      endfor
   endfor
endfor
```

(a) Nested Loop



(b) Footprint (c) Images of Faces

Figure 27: Footprint and Tessellations

If the domain is real, the image of a selected face will cover the corresponding tessellation completely. But since the domain of the nested loop, i.e., the iteration space, is integer, the image of a selected face may not refer to all the elements in the corresponding tessellation. For example, consider Figure 27. The nested loop is shown in (a), and the footprint is illustrated in (b). In the iteration space, there are $\binom{3}{2} = 3$ faces:

- F_1 : '$[0,5] \times [0,4]$ including $(0,0,0)$'
- F_2 : '$[0,4] \times [0,6]$ including $(0,0,0)$'
- F_3 : '$[0,5] \times [0,6]$ including $(0,4,0)$'

The faces F_1, F_2, and F_3 are the iteration spaces for i_1 and i_2 at $(0,0,0)$, for i_2 and i_3 at $(0,0,0)$, and for i_1 and i_3 at $(0,4,0)$, respectively. The images of the three faces are illustrated in (c). From (b) and (c), we know that the images of the three faces do not refer to all the elements of the footprint. While the image of the face F_1 is the same as the elements in the tessellation for the face, the images of F_2 and F_3 do not refer to all the elements in the corresponding tessellations of the footprint.

The points, which are in the footprint but not included in the image of the face, may be determined with the effect of other iteration variables that are not used to form the face. For example, reconsider Figure 27. In the tessellation for the face F_2, to cover all the elements in that tessellation of the footprint with the image of the face F_2, all the images of the face F_2 at $(0,0,0)$, at $(1,0,0)$, and at $(2,0,0)$ are required. In this case, we need images of three clone faces of F_2 along the i_1-axis. Thus the effect of i_1 to the face F_2 is 3 so that the image of F_2 can cover all the referenced elements in the tessellation. Similarly, in the tessellation for the face F_3, the images of the face F_3 at $(0,4,0)$ and at $(0,3,0)$ are required to cover all the elements in the tessellation for F_3 of the footprint. Thus the effect of i_2 to F_3 is 2.

The effect of each iteration variable to a face F is called the *thickness* of a face F along the axis of the iteration variable; F is extended along each axis by the amount of its effect and forms a new face, called the *thickened face* and denoted by $thk(F)$, that is a higher dimensional space than $r = rank(H)$.

The thickness and thickened face are constructed as follows. Let $e_1, \ldots, e_r$ be the basis to form a face F. Since $rank(H) = r$, and the vector space we are working with is on the integer domain, for every $t > r$, we have, for some positive integer α_t,

$$\alpha_t h_t \in span(h_1, \ldots, h_r). \tag{23}$$

The thickness of F along e_t is the smallest positive integer α_t satisfying Equation (23). The thickened face of F depends upon the vertex that F includes. Let us assume that F includes the vertex $(0, \ldots, 0, \eta_{r+1}, \ldots, \eta_n)$, where $\eta_j = l_j$ or u_j, for $j = r+1, n$. Define T_t by

$$T_t = \begin{cases} [l_t, \min(l_t + \alpha_t - 1, u_t)] & \text{if } \eta_t = l_t \\ [\max(l_t, u_t - \alpha_t + 1), u_t] & \text{if } \eta_t = u_t \end{cases}$$

Then the thickened face of F is

$$thk(F) = F \times \prod_{j=r+1}^{n} T_j.$$

With this construction, we can get the following formula:

$$H(D) = \bigcup_{F \in \mathcal{F}} H(thk(F)).$$

6.1.2 Thickness of Face

In this section, we explain how to compute the thickness of a face, which is an iteration space for $i_{k_1}, \ldots, i_{k_r}$, along the direction of a vector e_t, where $t \notin \{k_1, \ldots, k_r\} \subseteq$

$\{1, \ldots, n\}$.

Lemma 1 *Let $a_0, \ldots, a_r$ be integers. Then the smallest positive integer α satisfying*

$$\frac{a_j}{a_0} \cdot \alpha \quad \textit{are integers, for } j = 1, \cdots, r$$

is

$$\alpha = \frac{a_0}{\gcd(a_0, a_1, \ldots, a_r)}.$$

Proof Let $\{p_1, \ldots, p_n\}$ be a set of prime numbers which are factors of $a_0, \ldots, a_r$. Then the prime factorization of a_j can be denoted by

$$a_j = p_1^{\lambda_{j1}} \cdots p_n^{\lambda_{jn}}, \quad \text{where } \lambda_{j1}, \ldots, \lambda_{jn} \text{ are non-negative integers.}$$

For any value of j,

$$\frac{a_j}{a_0} = \frac{p_1^{\lambda_{j1}} \cdots p_n^{\lambda_{jn}}}{p_1^{\lambda_{01}} \cdots p_n^{\lambda_{0n}}}.$$

In order to get integer value for $\frac{a_j}{a_0} \cdot \alpha$,

$$\alpha = p_1^{\max\{\lambda_{01} - \lambda_{j1}, 0\}} \cdots p_n^{\max\{\lambda_{0n} - \lambda_{jn}, 0\}},$$

$$\alpha = p_1^{\lambda_{01} - \min\{\lambda_{j1}, \lambda_{01}\}} \cdots p_n^{\lambda_{0n} - \min\{\lambda_{jn}, \lambda_{0n}\}},$$

$$\alpha = \frac{a_0}{p_1^{\min\{\lambda_{j1}, \lambda_{01}\}} \cdots p_n^{\min\{\lambda_{jn}, \lambda_{0n}\}}}.$$

The above should be true for all $j = 1, \ldots, r$. Thus,

$$\alpha = \frac{a_0}{p_1^{\min_{j=0}^{j=r} \lambda_{j1}} \cdots p_n^{\min_{j=0}^{j=r} \lambda_{jn}}} = \frac{a_0}{\gcd(a_0, a_1, \ldots, a_r)}.$$

□

Let $\{e_1, \ldots, e_r\}$ be a set of basis vectors to form a face F. Then $V_F = \{h_1, \ldots, h_r\}$ is a set of the basis vectors to produce the image of a face F in an r-dimensional space. The image of a face generated by the iteration of loop variables $i_1, \ldots, i_r$, i.e., the footprint of the nested loop induced by those loop variables, is the linear combination of $h_1, \ldots, h_r$ with the integer coefficients in the range of $[l_1, u_1], \ldots, [l_r, u_r]$ respectively. The following theorem provides an efficient method to compute the thickness of a face F along the direction of e_t, where $t > r$.

Theorem 8 *Let $rank(H) = r$, $E_F = \{e_1, \ldots, e_r\}$ be a set of basis vectors to form a face F, $V_F = \{h_1, \ldots, h_r\}$, $C = (h_1, \ldots, h_r)$, and $e_t \notin E_F$. Then the thickness, α_t, of the face F along the direction of e_t is*

$$\alpha_t = \begin{cases} \frac{N_{0t}}{\gcd(N_{0t}, N_{1t}, \ldots, N_{rt})} & \text{if } N_{0t} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

where

$$N_{kt} = |\det(\tilde{C}_{kt})|, \qquad \tilde{C}_{kt} = \begin{pmatrix} h_1 & \ldots & h_{k-1} & h_t & h_{k+1} & \ldots & h_r \end{pmatrix}.$$

Proof The thickness of a face F along the direction of e_t is the smallest positive integer α_t such that

$$\alpha_t h_t \in span(h_1, \ldots, h_r) = \gamma_1 h_1 + \cdots + \gamma_r h_r, \quad \text{where } \gamma_1, \ldots, \gamma_r \text{ are integers,}$$

that is, in matrix form,

$$\begin{pmatrix} h_1 & \cdots & h_r \end{pmatrix} \begin{pmatrix} \gamma_1 \\ \vdots \\ \gamma_r \end{pmatrix} = \alpha_t h_t, \qquad (\text{i.e., } C \begin{pmatrix} \gamma_1 \\ \vdots \\ \gamma_r \end{pmatrix} = \alpha_t h_t). \tag{24}$$

Recall that d is the dimension of the array variable that we are interested in. The matrix C is a $(d \times d)$ square matrix if $r = d$, but a $(d \times r)$ rectangular matrix if $r < d$. If $r = d$, the answer is a straightforward one. In the case $r < d$, however, we have to think differently. If the r columns selected are linearly independent, we can select r linearly independent rows, and the remaining $(d - r)$ rows are generated by a linear combination of the selected r rows, so the square matrix of selected r rows is regarded as C. But if the r columns are not linearly independent, selecting any combination of r rows gives a singular matrix, so C is a singular matrix. From now on, let C be a $(r \times r)$ square matrix in any case.

If the matrix C is not singular (i.e., $N_{0t} \neq 0$), then we get the following equation, using *Cramer's rule*,

$$\gamma_k = \frac{N_{kt}}{N_{0t}} \cdot \alpha_t, \quad \text{for } k = 1, \ldots, r.$$

Since γ_k are integers, from Lemma 1,

$$\alpha_t = \frac{N_{0t}}{\gcd(N_{0t}, N_{1t}, \ldots, N_{rt})}. \tag{25}$$

If the matrix C is singular, then Equation (24) can be reduced to

$$0 = \alpha_t \cdot f(c_{1t}, \ldots, c_{rt}).$$

So α_t should be 0.

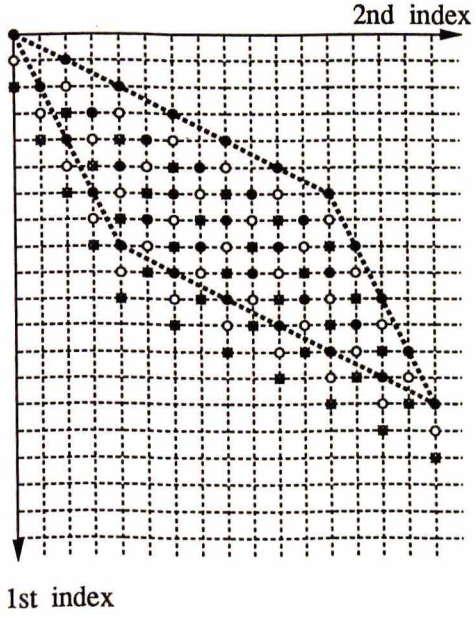
□

For example, consider the nested loop of Figure 27 (a). If a face F generated by the iteration i_2 and i_3 is being considered, then

$$C = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix},$$

```
for i1 = 0,2      or      for i1 = 3,5
   for i2 = 0,4
      for i3 = 0,6
         Xtmp(i1 + 2i2 + i3, i2 + 2i3) = X(i1 + 2i2 + i3, i2 + 2i3)
      endfor
   endfor
endfor
```

(a) Code to Copy Image of Thickened Face



(b) Image of Thickened Face

Figure 28: Image of Thickened Face

$$N_{01} = 3, \qquad N_{21} = 2, \qquad N_{31} = 1,$$

$$\alpha_1 = 3.$$

The thickened face is

$$thk(F) = \begin{cases} [0,2] \times [0,4] \times [0,6] & \text{if } F \text{ include } (0,0,0) \\ [3,5] \times [0,4] \times [0,6] & \text{if } F \text{ include } (5,0,0) \end{cases}$$

So the image of $thk(F)$, $H(thk(F))$, can be obtained from the code of Figure 28 (a), and diagram (b) shows the image of the thickened face, where the black circles, white circles, and white circles with cross represent the elements of the image of the face F when $i_1 = 0$, $i_1 = 1$, and $i_1 = 2$, respectively. Unlike Figure 27 (b), the image of the thickened face covers all the elements in the tessellation.

As we can see in Theorem 8, when the vectors forming a face are not linearly independent, the thicknesses of the face along any vectors are zero. In that case, we do not have to generate codes to localize the elements.

Corollary 1 *If* $N_{0t} = 0$, *then* $\alpha_t = 0$, *for* $t = r+1, \ldots, n$.

Corollary 2 *If* $N_{0t} \neq 0$, *then* $\alpha_t \neq 0$, *for* $t = r+1, \ldots, n$.

Corollary 3 *If* $N_{0t} = 1$, *then* $\alpha_t = 1$, *for* $t = r+1, \ldots, n$.

The proofs of these Corollaries are clear from Equation (25), because the denominator is the greatest common divisor of some numbers, of which one is N_{0t}, and the numerator is N_{0t}.

6.1.3 Optimizing Thicknesses of Faces

A thickness of a face along vector e_t for a loop variable i_t corresponds to the number of iterations of the loop variable i_t. So a smaller thickness is definitely preferable.

An optimization process may well be required to reduce the values of the thicknesses. Here, we will consider such optimization methods.

The thickened face of the construction described so far does not take into consideration the thicknesses of a face along those vectors that have been considered previously. Once we get a thickness along a vector, however, we can consider the effect of the thickness of that vector on the thickness along another vector. By selecting vectors in the proper order in the process of applying thicknesses of a face, we can reduce the thicknesses of a face along the vectors to be applied later.

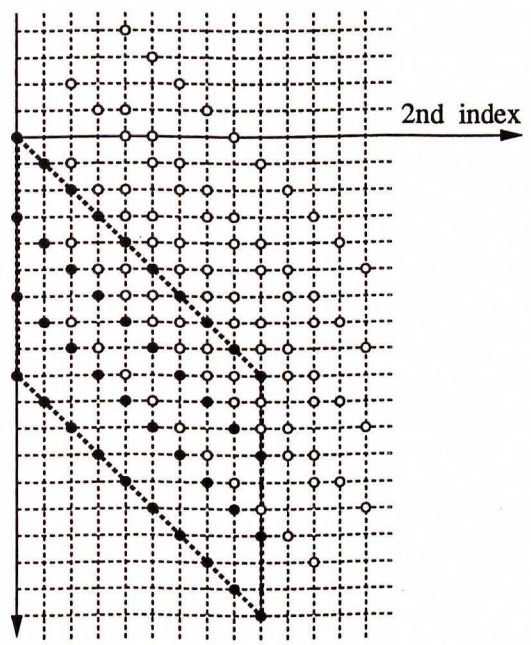
If the tessellation for a face is a smaller dimensional space than that of the face, i.e., the images of the basis vectors that form the face are not linearly independent, then the thickness of any direction is zero; thus, we do not have to compute any thickness for these cases. Otherwise, the thicknesses of the face are not zero, which follows from Corollary 2. From among these non-zero thicknesses, therefore, we need to select the smallest thickness, and use that thickness along the vector to optimize thicknesses along other directions.

When there happen to be more than one vector with the same thickness, which is the smallest, we need another criterion to choose one of them. Since the purpose of thickness is to fill the inside of the tessellation for a face, with the actual elements referenced in a nested loop, when the thickness is applied, the vector that produces a denser image is better. The criterion to decide whether it produces a denser image is the value of the *Euclidean norm* of the coefficient vector h_t for the corresponding loop variable i_t. Because the norm is the distance between points, the smaller the norm, the denser the image. For example, in Figure 29 a sample nested loop is given in (a). The mapping matrix H is

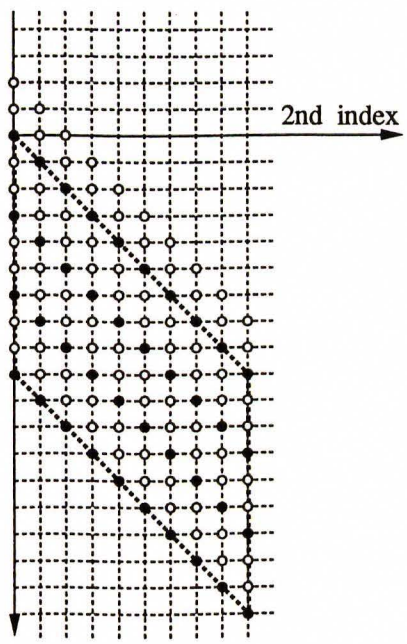
$$H = \begin{pmatrix} 3 & 1 & -2 & -1 \\ 0 & 1 & 2 & 0 \end{pmatrix}.$$

```
for i1 = 0,3
   for i2 = 0,9
      for i3 = 0,9
         for i4 = 0,9
            X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
         endfor
      endfor
   endfor
endfor
```

(a) Nested Loop



(b) Image of Thickened Face along e_3



(c) Image of Thickened Face along e_4

Figure 29: Effect of Applying a Thickness to a Face

When a face F formed by the basis vectors $\{e_1, e_2\}$, the thicknesses of the face along e_3 and e_4 are all 3; i.e., $\alpha_3 = \alpha_4 = 3$. Here the new criterion can be applied. From the definition of the Euclidean norm,

$$\|h_3\| = |(-2)^2 + 2^2|^{\frac{1}{2}} = 2\sqrt{2}, \qquad \|h_4\| = |(-1)^2 + 0^2|^{\frac{1}{2}} = 1.$$

Therefore, the thickness along e_4 is first selected. The effect of applying the thickness of a face is shown in Figure 29 (b) and (c). In the diagram, the parallelogram with dashed lines is the tessellation of a face, the set of black circles is the image of the face, and the set of white circles is the image of the face thickened along a direction, which is e_3 in figure (b) and e_4 in (c). It is clear that by applying the thickness along e_4 the inside of the tessellation is covered completely, and it is not necessary to apply the thickness along e_3; but if the thickness along e_3 is applied first, then there are many uncovered elements in the tessellation and it is still necessary to apply the thickness along e_4.

So far, we have made two levels of criteria by which to choose a thickness along a vector among many thicknesses for a face:

1. Choose vectors with smallest thickness.

2. Choose a vector e_t when $\|h_t\|$ is minimum.

With the selected thickness along a vector, we can optimize the other thicknesses. For instance, let $\{e_1, \ldots, e_r, e_s, e_t, e_w\}$ be a set of unit vectors for loop variables $i_1, \ldots, i_r, i_s, i_t$ and i_w, and $E_F = \{e_1, \ldots, e_r\}$ be a set of basis vectors of a face F. We need to compute thicknesses α_s, α_t, and α_w along vectors $e_s, e_t, e_w \notin E_F$. First, compute α_s, α_t, and α_w and select one of them with the criteria stated above. Let α_s be the first one selected as the smallest. After a thickened face of the face F is formed along e_s direction, we can interpret that thickened face in the following $(r+1)$ ways:

- Thickened face of the face formed by the basis vectors $\{e_s, e_2, \ldots, e_r\}$ along e_1 direction.
- ...
- Thickened face of the face formed by the basis vectors $\{e_1, \ldots, e_{r-1}, e_s\}$ along e_r direction.

From the first interpretation, we can compute the thickness of the face formed by $\{e_s, e_2, \ldots, e_r\}$ along e_t. From all interpretations, we can get r α_t values, but all zero values should be discarded, because the thicknesses of the face are not zero. From among those non-zero α_t values computed at this step and the original α_t value computed at the first step, select the one with the smallest value as the new thickness of the face F along e_t. Similarly, we can compute r additional α_w values and select one of them as α_w value at this step. In this way, we can get better thickness of a face along all vectors not in the set E_F.

In the next optimization step, select one value of α_t and α_w according to the selection criteria. Let α_t be the second one selected as the smallest. Then the face F is thickened along e_s and e_t. The thickened face of F along e_s and e_t can be interpreted as one of the following $\binom{r}{2}$ thickened faces:

- – Thickened face of the face formed by the basis vectors $\{e_s, e_t, e_3, \ldots, e_r\}$ along e_1 and e_2 directions.
  - ...
  - Thickened face of the face formed by the basis vectors $\{e_s, e_2, \ldots, e_{r-1}, e_t\}$ along e_1 and e_r directions.
- ...
- – Thickened face of the face formed by the basis vectors $\{e_1, \ldots, e_{r-2}, e_s, e_t\}$ along e_{r-1} and e_r directions.

From the above interpretations, we can get additional $\binom{r}{2}$ thicknesses α_w values. From among those non-zero α_w values computed at this step and the current α_w value, select the one with the smallest value as the new thickness of the face F along the e_w. In this second optimization step, we can improve thickness values for the face along all vectors not in the set E_F. This optimization is carried out further.

Figure 30 shows the algorithm to compute optimal thicknesses. The inputs to the algorithm are H, the index function in matrix form, and $\mathcal{F}$, the set of r-dimensional faces that are properly selected according to the tessellation process. $\mathcal{F}$ is modified to keep the faces whose thicknesses are to be computed. The outputs are the thicknesses of faces returned through a 2-dimensional array α whose first subscript refers to a face and whose second subscript refers to the direction of the thickness. The variable U is the set of all unit vectors for iteration variables in the nested loop. E_F is the set of unit vectors forming the face F. S_F is the set of vectors along whose directions the thicknesses of the face F is to be computed. T_F is the set of vectors along which the thicknesses are fixed and thus are not to be improved further. The value of n_F is the number of vectors in the set T_F but does not exceed $r = rank(H)$, and n_F vectors from the set E_F replace n_F vectors from the set T_F.

In the first large scoped `for` loop, thicknesses are computed for every face along every vector not in the set E_F; all the faces are removed from the set $\mathcal{F}$ either if the vectors in E_F are not linearly independent, i.e., $N = 0$, or if all the thicknesses are one, i.e., $N = 1$. From Corollary 1, if $N = 0$, all the thicknesses are 0, and from Corollary 3, if $N = 1$, all the thicknesses are 1, which cannot be reduced further. Once we get all thicknesses, select the one vector with the smallest thickness (in the case that there happen to be more than one vector with the same smallest thickness, then select the one with the smallest *norm* of corresponding column vector of H) and remove the vector from S_F and add it to T_F so that the thickness of the face along the vector is fixed.

```
Procedure Optimal_Thickness(H, F, α);
   /* Input :  H : (d × n) mapping function array */
   /* Input :  F : set of (n choose r) faces */
   /* Output :  α : ((n choose r) × n) array */
   /* where r = rank(H) */
U = {e_1, ..., e_n}; /* set of unit vectors of iteration space */
for (every face F ∈ F) {
   E_F = set of basis vectors forming face F;
   S_F = U − E_F;
   if (N == 0) { /* N is defined in Theorem 8 */
      for (every e_t ∈ S_F) α(F,t) = 0;
      F = F − {F};
   } else {
      if (N == 1) {
         for (every e_t ∈ S_F) α(F,t) = 1;
         F = F − {F};
      } else {
         for (every e_t ∈ S_F) Compute α(F,t);
         Select one index s such that
                ||h_s|| = min{ ||h_s'|| | α(F,s') = min{α(F,t)|e_t ∈ S_F}};
         S_F = S_F − {e_s};  T_F = {e_s};  n_F = 1;
} } }
while (S_F ≠ φ, for any F ∈ F) {
   for (every face F ∈ F) {
      for (every e_t ∈ S_F) {
         n_F = min(n_F, r);    β = ∞;
         for (every P_F, where P_F : set of n_F elements in E_F)
            for (every Q_F, where Q_F : set of n_F elements in T_F) {
               E'_F' = (E_F − P_F) ∪ Q_F; /* basis for face F' */
               if (α(F',t) > 0) β = min(β, α(F',t));
            }
         α(F,t) = min(α(F,t), β);
      }
      Select one index s such that
             ||h_s|| = min{ ||h_s'|| | α(F,s') = min{α(F,t)|e_t ∈ S_F}};
      S_F = S_F − {e_s};  T_F = T_F ∪ {e_s};  n_F = n_F + 1;
} }
```

Figure 30: Algorithm to Compute Optimal Thicknesses

```
forall i0 = 1, np
   for i1 = 0, 9
      for i2 = 0, 9
         for i3 = 0, 9
            for i4 = 0, 9
               X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
            endfor
         endfor
      endfor
   endfor
endforall
```

Figure 31: Nested Loop to be Localized

The `while` loop runs while there are any faces with a non-empty S_F set, because if S_F is not empty, then there are still some vectors whose thicknesses are not yet optimized. On every iteration, one vector for a face is selected as an optimal thickness, removed from S_F, and added to T_F. Thus S_F finally becomes empty. In the `for (every` $e_t \in S_F$`)` loop, an improved thickness α_t of the face F along the e_t direction is set to the smallest among the thicknesses of the faces, which are formed by the unit vectors in E_F by replacing n_F vectors with n_F vectors in T_F, along the direction of e_t. The optimal thickness of a face along a direction of a vector, whose thickness is not yet optimized, is chosen in the `for (every face` $F \in \mathcal{F}$`)` loop by applying the two levels of criteria: choose vectors with the smallest thickness, and choose a vector with the minimum norm of column vector.

6.2 Loop Transformation

Using the theories developed so far, we can localize non-contiguously accessed data efficiently. In this section, we explain how to transform the given nested loop into one with data localization and how the transformed code affects program performance. For example, consider the nested loop shown in Figure 31.

The array variable X cannot be localized by a fast block copy as described in Chapter 5 because the footprint is not 2-contiguous. Each processor executes the nested `for` loop in parallel with other processors. For each processor, the total number of elements referenced is 1324, but the number of accesses to the array X is 10000. Since the number of accesses to the array is 7.6 times greater than the number of elements accessed, if we make a copy of an element into local memory, then further access to the element can be carried out very rapidly within local memory.

Let us apply the procedure developed so far. The index function in matrix form, H, is

$$H = \begin{pmatrix} 3 & 1 & -2 & -1 \\ 0 & 1 & 2 & 0 \end{pmatrix}.$$

The column vectors are properly ordered as described in the tessellation process, and $rank(H) = 2$. Therefore, The faces stated in the tessellation process are 2-dimensional. There are 6 faces by selecting 2 vectors out of 4:

- F_1 : formed by basis $\{e_1, e_2\}$, $[0,9] \times [0,9]$ including $(0,0,0,0)$
- F_2 : formed by basis $\{e_2, e_3\}$, $[0,9] \times [0,9]$ including $(0,0,0,0)$
- F_3 : formed by basis $\{e_3, e_4\}$, $[0,9] \times [0,9]$ including $(0,0,0,0)$
- F_4 : formed by basis $\{e_1, e_3\}$, $[0,9] \times [0,9]$ including $(0,9,0,0)$
- F_5 : formed by basis $\{e_2, e_4\}$, $[0,9] \times [0,9]$ including $(0,0,9,0)$
- F_6 : formed by basis $\{e_1, e_4\}$, $[0,9] \times [0,9]$ including $(0,9,9,0)$

The unoptimized and optimized thicknesses corresponding to the faces are computed as in Table 1. From the table, we can construct the thickened faces. When thicknesses are unoptimized, the corresponding thickened faces are

- $thk(F_1) = [0,9] \times [0,9] \times [0,2] \times [0,2]$

face	unit vectors	thickness							
		unoptimized				optimized			
		α_1	α_2	α_3	α_4	α_1	α_2	α_3	α_4
F_1	e_1, e_2			3	3			1	3
F_2	e_2, e_3	4			4	1			4
F_3	e_3, e_4	1	2			1	2		
F_4	e_1, e_3		6		3		2		3
F_5	e_2, e_4	1		1		1		1	
F_6	e_1, e_4		0	0			0	0	

Table 1: Computed Thicknesses of Faces

- $thk(F_2) = [0,3] \times [0,9] \times [0,9] \times [0,3]$
- $thk(F_3) = [0,0] \times [0,1] \times [0,9] \times [0,9]$
- $thk(F_4) = [0,9] \times [4,9] \times [0,9] \times [0,2]$
- $thk(F_5) = [0,0] \times [0,9] \times [9,9] \times [0,9]$
- $thk(F_6) = [0,9] \times \phi \times \phi \times [0,9]$

When thicknesses are optimized, the corresponding thickened faces are

- $thk(F_1) = [0,9] \times [0,9] \times [0,0] \times [0,2]$
- $thk(F_2) = [0,0] \times [0,9] \times [0,9] \times [0,3]$
- $thk(F_3) = [0,0] \times [0,1] \times [0,9] \times [0,9]$
- $thk(F_4) = [0,9] \times [8,9] \times [0,9] \times [0,2]$
- $thk(F_5) = [0,0] \times [0,9] \times [9,9] \times [0,9]$
- $thk(F_6) = [0,9] \times \phi \times \phi \times [0,9]$

```
for i1 = 0,9
   for i2 = 0,9
      for i3 = 0,2
         for i4 = 0,2
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
endfor endfor endfor endfor
for i1 = 0,3
   for i2 = 0,9
      for i3 = 0,9
         for i4 = 0,3
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
endfor endfor endfor endfor
i1 = 0
   for i2 = 0,1
      for i3 = 0,9
         for i4 = 0,9
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
   endfor endfor endfor
for i1 = 0,9
   for i2 = 4,9
      for i3 = 0,9
         for i4 = 0,2
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
endfor endfor endfor endfor
i1 = 0
   for i2 = 0,9
      i3 = 9
         for i4 = 0,9
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
   endfor  endfor
```

Figure 32: Unoptimized Data Localization

```
for i1 = 0,9
   for i2 = 0,9
      i3 = 0
         for i4 = 0,2
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
endfor endfor  endfor
i1 = 0
   for i2 = 0,9
      for i3 = 0,9
         for i4 = 0,3
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
   endfor endfor endfor
i1 = 0
   for i2 = 0,1
      for i3 = 0,9
         for i4 = 0,9
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
   endfor endfor endfor
for i1 = 0,9
   for i2 = 8,9
      for i3 = 0,9
         for i4 = 0,2
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
endfor endfor endfor endfor
i1 = 0
   for i2 = 0,9
      i3 = 9
         for i4 = 0,9
            Xtmp(3i1 + i2 − 2i3 − i4, i2 + 2i3) = X(3i1 + i2 − 2i3 − i4, i2 + 2i3)
   endfor  endfor
```

Figure 33: Optimized Data Localization

The loop transformations for data localization are based on the thickened faces. The nested loop given in Figure 31 is transformed with the data localization as shown in Figure 32 and Figure 33. The code in Figure 32 is based on the unoptimized thickness and the one in Figure 33 on the optimized thickness. The number of copy operations of the unoptimized code is 4600 with 247% overhead, but that of the optimized code is 1600 with 21% overhead. This overhead comes from around the boundaries of tessellations when thicknesses are applied along several directions.

6.3 Optimizing Performance

For a given nested loop with an array reference, we can calculate the total number of accesses to the array. If we apply the tessellation process to make a local copy, we can calculate the number of copy operations thus required. Because remote access time is usually much greater than local access time (for example, it is about eleven times greater for Butterfly) if the number of accesses is much greater than the number of local copy operations, then we can get big performance improvement by making a local copy; but if there is not a big difference between the number of accesses and local copy operations, making a local copy might degrade the performance because of the overhead. The following analysis does not take into consideration any adverse effects caused by network traffic saturation, memory hot spot contention, and so on.

Let the access time be l machine cycles for local data, and r machine cycles for remote data. Assume that there are P processors executing in parallel, the number of data access for each processor is N, and data are distributed evenly over all memory modules. Then the probabilities to access local and remote data are $\frac{N}{P} \cdot \frac{1}{N} = \frac{1}{P}$ and $\frac{N*(P-1)}{P} \cdot \frac{1}{N} = \frac{P-1}{P}$ respectively.

When the data are not localized, the time for each processor to access all data is

$$(l \cdot \frac{1}{P} + r \cdot \frac{P-1}{P}) \cdot N. \tag{26}$$

When all data are localized by using the tessellation process, if the calculated number of copy operations to localize all required data is C, the time for each processor to copy all required data is

$$(l \cdot \frac{1}{P} + r \cdot \frac{P-1}{P}) \cdot C.$$

Once all the data needed for execution are copied into local memory, all the data accesses during computation occurs locally. So the total time to reference data is $l \cdot N$. The sum of the time to make a local copy and to access it locally is

$$(l \cdot \frac{1}{P} + r \cdot \frac{P-1}{P}) \cdot C + l \cdot N. \tag{27}$$

From the two equations (26) and (27), the localization effort is worthwhile only when the following inequality holds:

$$(l \cdot \frac{1}{P} + r \cdot \frac{P-1}{P}) \cdot N > (l \cdot \frac{1}{P} + r \cdot \frac{P-1}{P}) \cdot C + l \cdot N.$$

Expressing the above equation in terms of the ratio of the remote access time to the local access time and the ratio of the number of copy operations to the total number of references, we get

$$\frac{C}{N} < 1 - \frac{1}{\frac{r}{l}(1 - \frac{1}{P}) + \frac{1}{P}}. \tag{28}$$

The right hand side is the upper bound of $\frac{C}{N}$ where the localization may improve the performance. When the number of processors, P, is fixed, the value $\frac{r}{l}$ affects the condition of the equation. As the ratio $\frac{r}{l}$ approaches to 1, i.e., the remote access time approaches to the local access time, the right hand side of Equation (28) approaches

No. of Procs	r/l					
	2	4	8	11	16	32
2	0.33	0.60	0.78	0.83	0.88	0.94
4	0.43	0.69	0.84	0.88	0.92	0.96
8	0.47	0.72	0.86	0.90	0.93	0.96
16	0.48	0.74	0.87	0.90	0.93	0.97
32	0.49	0.74	0.87	0.91	0.94	0.97

Table 2: Upper Bound of $\frac{C}{N}$ for Performance Improvement

to 0, and as the ratio $\frac{r}{l}$ increases, the right hand side of Equation (28) approaches to 1. Therefore, if the remote access time is much larger than the local access time, then data localization is efficient in most cases. However, if there is not a big difference between the remote and the local access time, data localization is efficient only for the small value of the ratio $\frac{C}{N}$.

In Table 2, the upper bound of $\frac{C}{N}$ for the better performance are computed for various combinations of the number of processors and the ratio of the remote to local access time. As expected, the upper bound gets larger as the ratio $\frac{r}{l}$ increases. In the case of the Butterfly GP1000 parallel machine, the remote read and write accesses take 7 μ second and 4 μ second, and the local read and write accesses take 0.53 μ second and 0.38 μ second [8]. Therefore, the approximate ratio $\frac{r}{l}$ is 11. From the table, we can see that using data localization may be more efficient than the original program, when the ratio $\frac{C}{N}$ is less than 0.83.

Chapter 7

Experiments

In the previous chapters, we have considered data localization techniques to improve the execution performance of nested loops. In this chapter, we present the experimental results of the localization algorithms run on a parallel machine, a Butterfly GP1000. (See the Appendix A for its detailed description.) Since the GP1000 is a hybrid class machine, it has properties both of the shared memory and distributed memory systems.

The GP1000 machine can be equipped with up to 256 processors. The machine used for this test has 28 processors, 3 for the public cluster, 1 for I/O process, and 24 for users. At any given time, a user can secure as many processors as he wants, up to 24. The set of processors allocated to a user is called a cluster. Each processor has a local memory of 4M bytes, and the combined local memory modules of processors form the shared memory of the system. The shared data array is uniformly distributed to the local memory of each processor in the user cluster to avoid memory hot spots.

To verify the effect of the program transformation, we have run two programs, the original and the transformed one, on 1 through 24 processors. For convenience, we will name the original program $CONT_U$ and the transformed program $CONT_L$.

7.1 Contiguous Data Access

A simple matrix multiplication program $CONT_U$, with a contiguous data access pattern in a nested loop as shown in Figure 34, is chosen for this demonstration. When

```
forall i = 0, n − 1
   for j = 0, n − 1
      c(i, j) = 0
      for k = 0, n − 1
         c(i, j) = c(i, j) + a(i, k) ∗ b(k, j)
      endfor
   endfor
endforall
```

Figure 34: Matrix Multiplication (Contiguous) — $CONT_U$

the program runs on P processors, n parallel processes, each of which is to execute the `forall` loop body for a value of i between 0 and $n-1$, are generated, and P processors fetch, and activate, the parallel processes until there is no process waiting to be run.

Let us apply the techniques developed in Chapter 5 to the program $CONT_U$. In the k-loop, the array a is 2-contiguous, b is 1-contiguous, and only one element of c is referenced. Thus, the row elements of the array a are the only ones to be localized outside the k-loop. In the j-loop, array a is still 2-contiguous, b is 1- and 2-contiguous, and c is 2-contiguous. The row elements of a and c, as well as all the data elements of b, may be localized outside the j-loop. Array a can be localized outside both the k-loop and j-loop. For better results, it is important to have as much of the outer loop as possible localized in order to avoid duplicated local copies. Array b can be localized outside the j-loop. However, it can also be localized outside the i-loop, since b is in the special situation that the index function does not depend on the `forall` loop variable i. In other words, array b may be copied into all local memories of processors in the user cluster. Also, the reference modes are such that a and b are referenced in read mode and c in read/write mode. Obviously, a and b should be copied before they are referenced. For c, the process is a little tricky. Based on the dependence analysis, c is initialized, modified, and keep the final results of the loop: namely, c is independent of the original values. Unlike the others, therefore,

```
forall i = 0, n − 1
   dimension la(0:n-1,0:n-1),lb(0:n-1,0:n-1),lc(0:n-1,0:n-1)
   for k = 0, n − 1
      bcopy(&b(k, 0), &lb(k, 0), n)
   endfor
loop
   bcopy(&a(i, 0), &la(i, 0), n)
   for j = 0, n − 1
      lc(i, j) = 0
      for k = 0, n − 1
         lc(i, j) = lc(i, j) + la(i, k) ∗ lb(k, j)
      endfor
   endfor
   bcopy(&lc(i, 0), &c(i, 0), n)
endforall
```

Figure 35: Data Localization of $CONT_U$ — $CONT_L$

it does not have to be copied into the local memory first. The computation can be carried out using data in the local memory and the results should be copied from the local array to the original array. The transformed program with data localization, $CONT_L$, is shown in Figure 35.

In $CONT_L$, all of the arrays are localized by means of the fast block copy, and all computations are carried out with those local variables. The number of remote memory accesses is $O(n^3)$ in $CONT_U$, and $O(n^2)$ in $CONT_L$. Moreover, the remote memory accesses in $CONT_L$ are done by a block copy. The remote data access time, which adversely affects the program performance, is significantly reduced in $CONT_L$ by reducing the number of remote memory accesses and using a fast block copy.

Both programs $CONT_U$ and $CONT_L$ were run for n=32, 64, 128, and 256. In Figure 38 and 39, graphs (a), (c), (e), and (f) represent the execution time in the real time clock ticks of Butterfly machine, and graphs (b), (d), (f), and (h) represent the speed-ups on varying number of processors. The speed-up $S(n, P)$ on P processors

for matrix size n is defined by

$$S(n,P) = \begin{cases} \frac{E_U(n,1)}{E_U(n,P)} & \text{for } CONT_U \\ \frac{E_U(n,1)}{E_L(n,P)} & \text{for } CONT_L \end{cases}$$

where $E_U(n,P)$ is the execution time of $CONT_U$, and $E_L(n,P)$ is that of $CONT_L$, on P processors for a matrix of size n. In these graphs, solid lines with squares represent the behavior of $CONT_U$ and dotted lines with circles that of $CONT_L$. For all matrices of any size, the speed-ups of $CONT_U$ on any number of processors are below 2, but those of $CONT_L$ increase almost linearly. Thus, the bigger the size of the matrix, the greater the increasing rate. When the matrix size is 256, the speed-up of about 22 on 24 processors is almost perfect. Such improved performance may be explained by the fact that the computation complexity is $O(n^3)$ while the number of remote data accesses is $O(n^2)$. The computation time dominates the remote data access time as the matrix size gets bigger.

7.2 Non-Contiguous Data Access

As a program to test the algorithm for non-contiguously accessed data, the modified matrix multiplication program, named $NCONT_U$, is chosen here as shown in Figure 36.

Let us apply the techniques of Chapter 6 first to the nested loop of loop variables j and k in $NCONT_U$, for they are all inside the `forall` loop of the loop variable i. The index functions of the array references to a, b and c in matrix representation are, respectively,

$$H^a = \begin{pmatrix} 0 & 0 \\ 0 & 2 \end{pmatrix}, \qquad H^b = \begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix}, \qquad H^c = \begin{pmatrix} 0 & 0 \\ 2 & 0 \end{pmatrix}.$$

```
forall i = 0, n − 1
   for j = 0, n/2 − 1
      c(i, j) = 0
      for k = 0, n/2 − 1
         c(i, 2j) = c(i, 2j) + a(i, 2k) ∗ b(2k, 2j)
      endfor
   endfor
endforall
```

Figure 36: Matrix Multiplication (Non-Contiguous) — $NCONT_U$

The first column of each matrix denotes the loop variable j, and the second column k. The first row of each matrix is for the first index of each array variable, and the second row the second index. As for the matrix H^a for the array variable a, its rank is 1. Therefore, we can choose 1 column that is not zero. There is only one column that is not zero, which is the second column for loop k. The thickness of the face $\{e_k\}$ along the vector e_j is 1, and that of the face $\{e_j\}$ is 0. Thus, the resulting localization code for a is

```
for k = 0, n/2 − 1
   la(i, 2k) = a(i, 2k)
endfor
```

Because the array a is used in read mode, it should be localized before the computation. The array c can be localized in the similar way to the array a. However, c is used in write mode. Thus, the resulting localization code for c is

```
for j = 0, n/2 − 1
   c(i, 2j) = lc(i, 2j)
endfor
```

Finally, the rank of H^b is 2. Because the rank and the size of the array are the same, it is not necessary to compute the thickness pertaining to b. Thus, the resulting localization code for b is

```
forall i = 0, n − 1
   dimension la(0:n-1,0:n-1),lb(0:n-1,0:n-1),lc(0:n-1,0:n-1)
   for k = 0, n/2 − 1
      for j = 0, n/2 − 1
         lb(2k, 2j) = b(2k, 2j)
      endfor
   endfor
loop
   for k = 0, n/2 − 1
      la(i, 2k) = a(i, 2k)
   endfor
   for j = 0, n/2 − 1
      lc(i, 2j) = 0
      for k = 0, n/2 − 1
         lc(i, 2j) = lc(i, 2j) + la(i, 2k) * lb(2k, 2j)
      endfor
   endfor
   for j = 0, n/2 − 1
      c(i, 2j) = lc(i, 2j)
   endfor
endforall
```

Figure 37: Data Localization of $NCONT_U$ — $NCONT_L$

```
for k = 0, n/2 − 1
   for j = 0, n/2 − 1
      lb(2k, 2j) = b(2k, 2j)
   endfor
endfor
```

In fact, all the localization codes are to be inside the `forall` loop, but the localization code for b can be moved outside the `forall` loop, since the index of the array reference to b is independent of the `forall` loop variable i. The transformed program with data localization, $NCONT_L$, is shown in Figure 37.

In $NCONT_L$, all of the arrays are localized through copying referenced elements word-by-word, and all computations are carried out using those local variables.

Unlike the localization process of $CONT_L$, the fast block copy cannot be used.

However, the number of remote memory accesses is still $O(n^3)$ in $NCONT_U$, and $O(n^2)$ in $NCONT_L$, but the localization cannot be done as efficiently as $CONT_L$ which utilizes the fast block copy.

Both programs $NCONT_U$ and $NCONT_L$ were also run for n=32, 64, 128 and 256. In Figure 40 and 41, graphs (a), (c), (e), and (f) represent the execution time in the real time clock ticks of the Butterfly machine, and graphs (b), (d), (f), and (h) represent the speed-ups on a different number of processors. The speed-up $S(n, P)$ on P processors for a matrix of size n is defined, as in the case of the contiguous data access case, to be

$$S(n,P) = \begin{cases} \frac{E_U(n,1)}{E_U(n,P)} & \text{for } NCONT_U \\ \frac{E_U(n,1)}{E_L(n,P)} & \text{for } NCONT_L \end{cases}$$

where $E_U(n, P)$ is the execution time of $NCONT_U$ and $E_L(n, P)$ is that of $NCONT_L$, on P processors for a matrix of size n. In the graphs, solid lines with squares represent the behavior of $NCONT_U$ and dotted lines with circles that of $NCONT_L$. For all matrices of any size, the maximum speed-ups of $NCONT_U$ are about 2 which is a little higher than those of the $CONT_U$. This improved performance results from having fewer remote data accesses than $CONT_U$. (The number of remote accesses of $NCONT_U$ is one quarter of that of $CONT_U$.) The reduction in remote accesses can reduce the delay caused by network traffic. The speed-ups of $NCONT_L$ increase almost linearly, and the linear property becomes more apparent as the matrix size gets bigger, because the computation complexity $O(n^3)$ dominates the remote access complexity $O(n^2)$. However, the speed-ups are less evident than those of the $CONT_L$, because a fast block copy cannot be used for data localization.

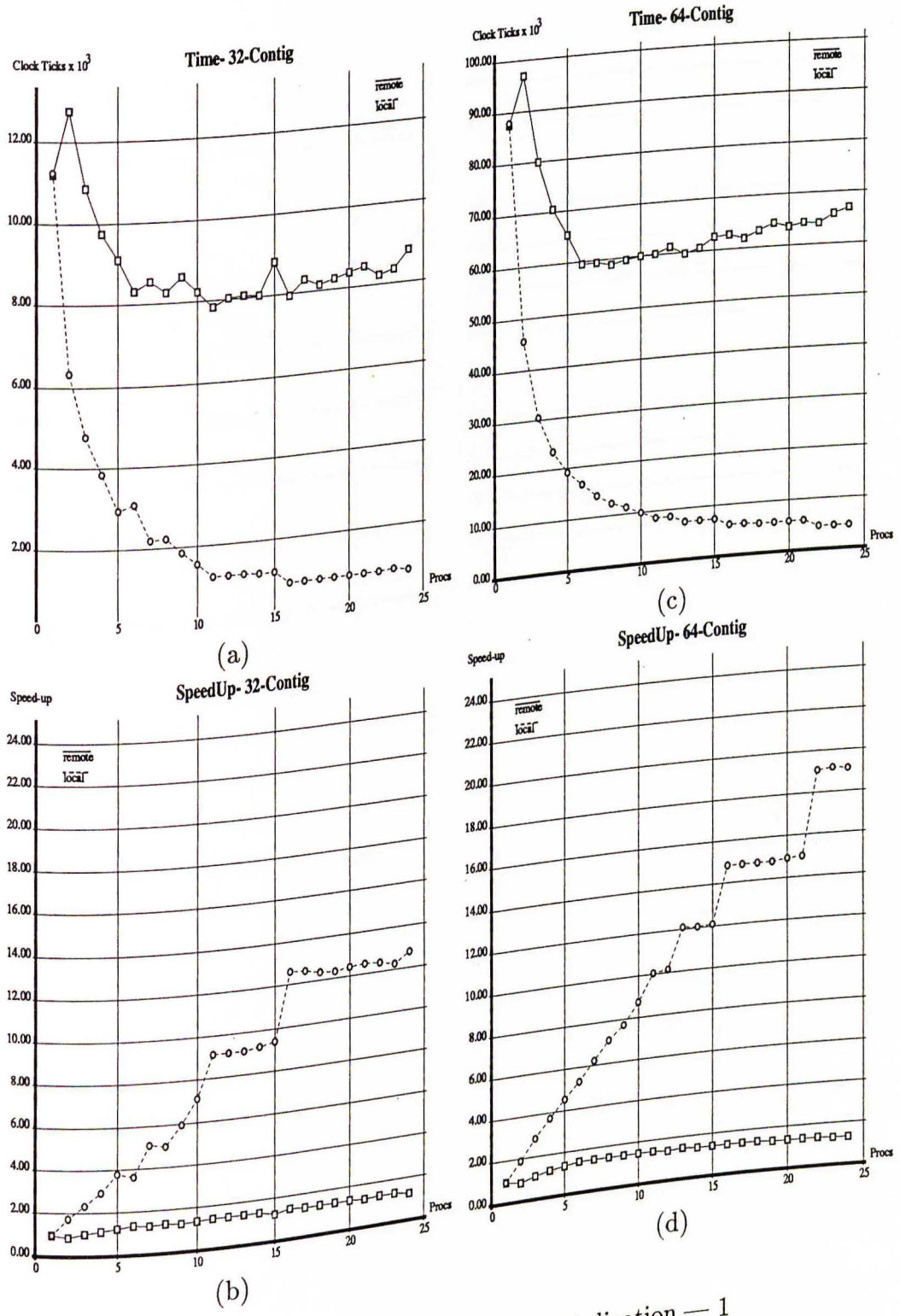


Figure 38: Contiguous Matrix Multiplication — 1

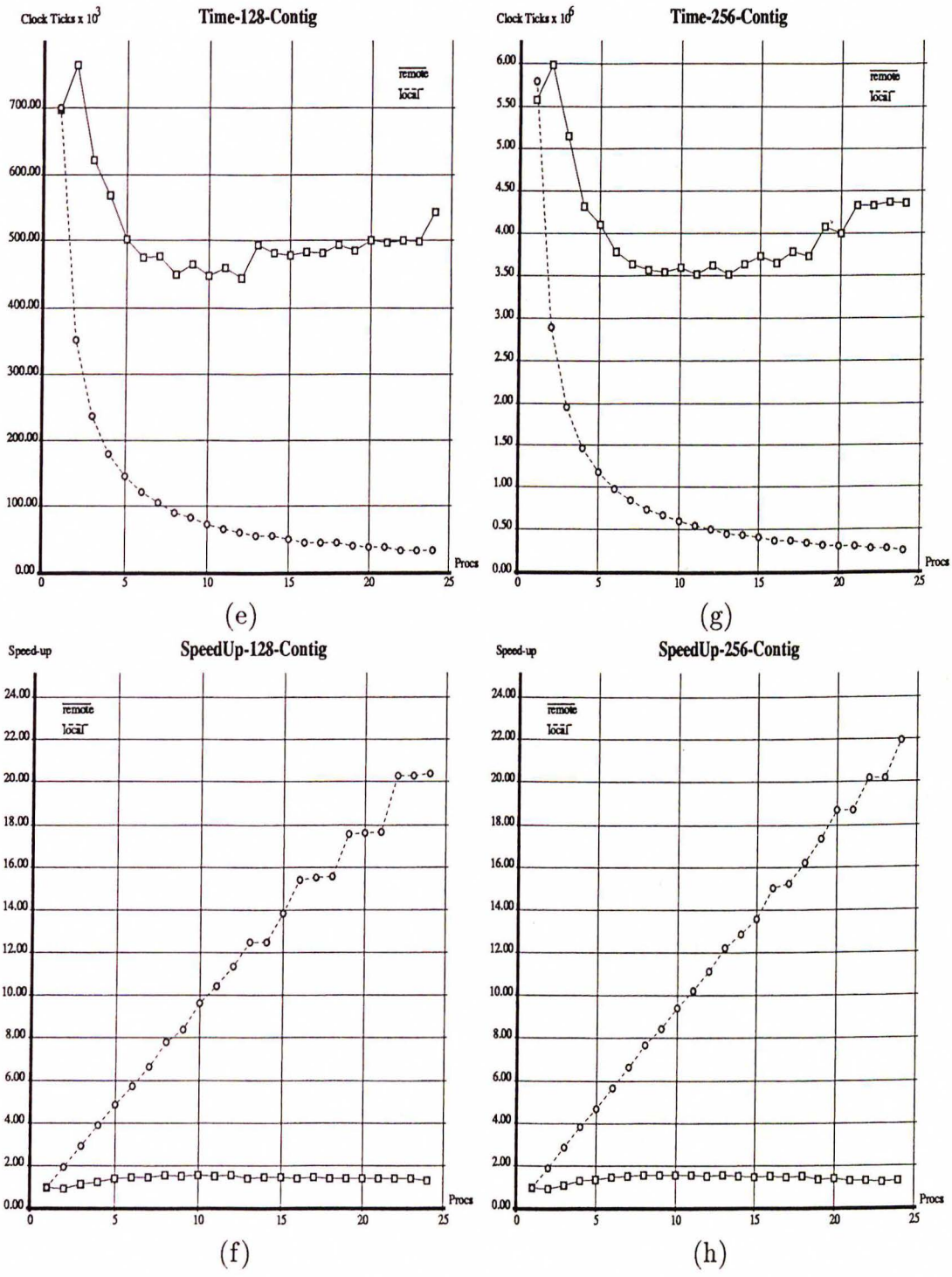


Figure 39: Contiguous Matrix Multiplication — 2

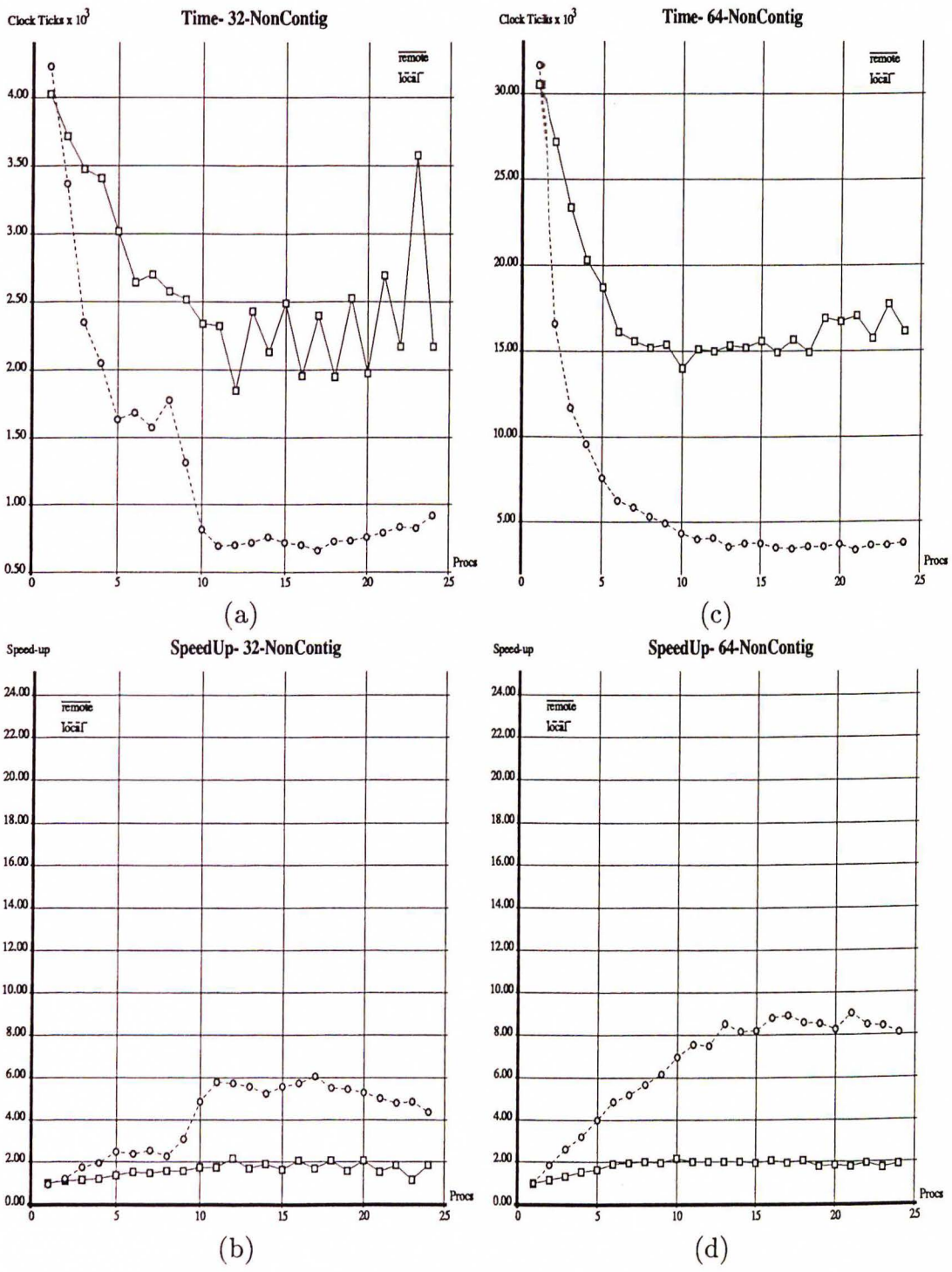


Figure 40: Non-contiguous Matrix Multiplication — 1

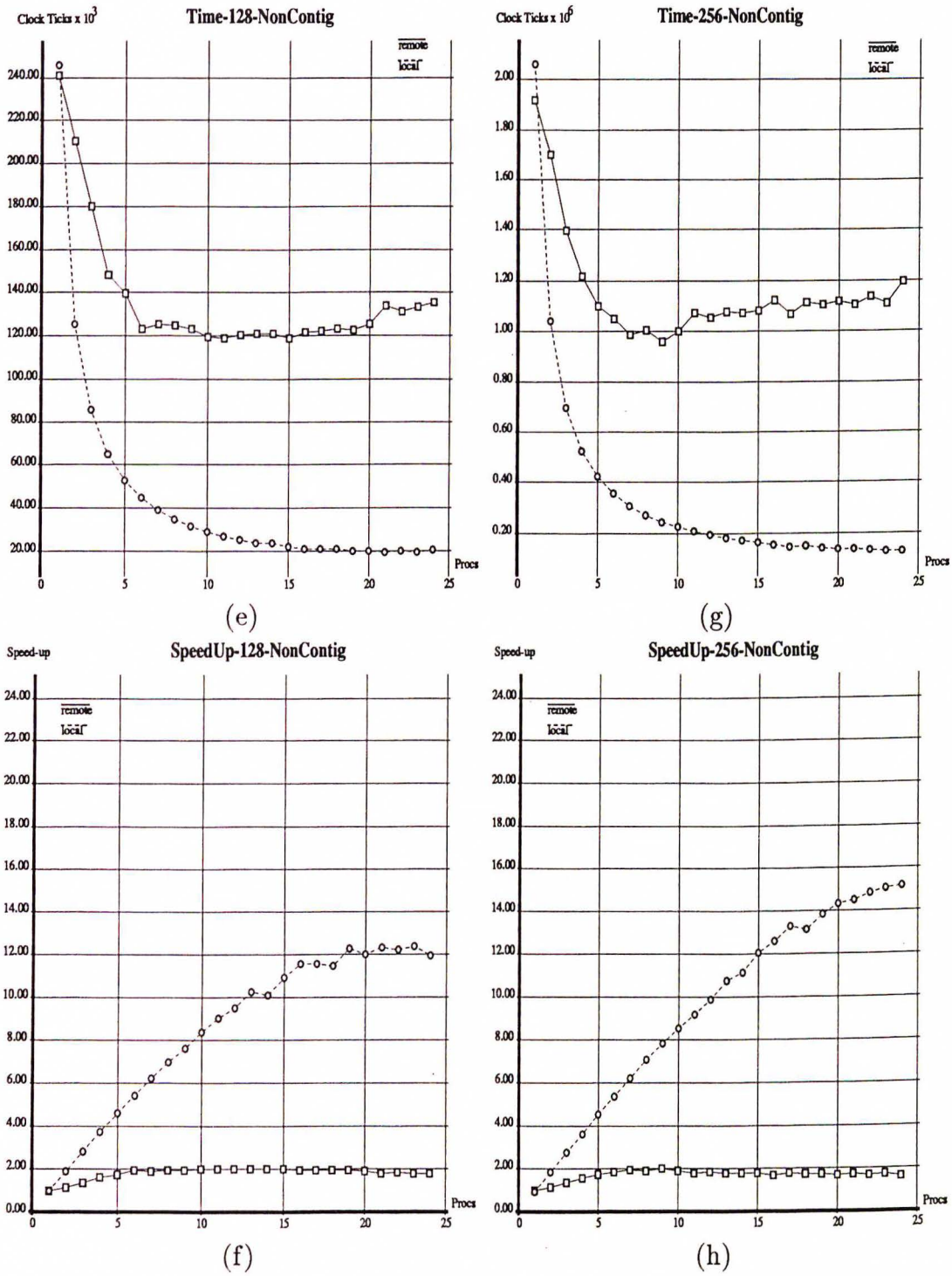


Figure 41: Non-contiguous Matrix Multiplication — 2

Chapter 8

Conclusions

In this thesis we have introduced code transformations that influence the management of local memory in MIMD parallel computers. These transformations localize, preferably without redundancy, the array elements referenced by inner loops whose outermost loop is executed in parallel. The set of these elements may be identified by a combination of the coefficients and the lower and upper bounds of the loop variables in the index functions. If it is determined that they happen to form contiguous blocks in the memory, they may be efficiently localized by means of the fast block transfer. If not, they may nevertheless be localized word-by-word, provided that such a tedious process still promises to be more efficient.

A method, called the *tessellation process*, was introduced for 1- and 2-dimensional arrays in Chapter 3; by this method, we were able to analyze the pattern of the array elements referenced in a nested loop. Once a d-dimensional array variable is referred to inside a nested loop of depth n, the process helps divide the referenced elements into $\binom{n}{d}$ subsets which are called their *tessellations*. A tessellation is an image of the face of a d-dimensional subspace of the iteration space under the index function. Each tessellation corresponds to a set of d loop variables and is characterized as the area spanned by the loop variables with the coefficients between the lower and upper bounds of the corresponding loop variables.

In Chapter 4, the tessellation process was applied to decide whether the array elements thus referred to actually form contiguous blocks in the memory, relying on

a specific array allocation scheme. If the array reference pattern is determined to be contiguous, the localization of the remote data may then be carried out rapidly. For a 2-dimensional array, a linear integer programming problem was derived from the index function and loop bounds, so as to find the starting and ending locations of the contiguous blocks. The localization of the contiguous data was discussed in Chapter 5.

In Chapter 6, simple algorithms to compute the thicknesses of the faces of the iteration space were considered. Further, the thicknesses were optimized to reduce the percentage of redundancy in copy operations between the remote and local memory from several hundred to a few tens of percent. Then, the nested loops were transformed to localize the non-contiguously accessed array, using the information from the tessellation process and the calculated thicknesses of faces.

Finally, in Chapter 7, a matrix multiplication program, which satisfies certain assumptions, was chosen to demonstrate how much the algorithms improve program performance on a parallel machine. The experimental results show that the data localization algorithms greatly improve program performance on a parallel machine equipped with local memory.

These achievements aside, this thesis suggests quite a few shortcomings that need to be dealt with in the future, in order for the algorithms to be generalized. For one, we made the initial assumption that the loop bounds are independent of the other loop variables. In addition, we assumed that all the references to an array have the same index functions, and further simplified the index functions to the exclusion of any constant terms. Of course, even with these assumptions and simplifications, we may handle a limited number of cases effectively and often get very good results, as we have shown in Chapter 7. However, the range of cases might be much broader if we try to apply these algorithms to a number of practical application programs. In fact, there are many examples in linear algebra where many loop structures easily

outgrow the simplifying assumptions prescribed in this thesis.

Therefore, we have to address the issue of constants in those cases where the variables are referenced with different index functions. Suppose that a 1-dimensional array is referenced twice in a loop with the index functions, $(2i)$ and $(2i+1)$, where i is the loop variable. Then, while the footprint is in fact contiguous, it will instead be determined by the algorithms described in this thesis to be non-contiguous, because the constant terms have not been sufficiently considered.

When we try to program those triangular matrices found in many problems in linear algebra, our own experience tells us that the matrices are to be handled by the nested loops, where the loop bounds of one variable often depend on other loop variables. Therefore, further research needs to be done for such cases.

For these reasons, it is important that the tessellation process be generalized to high-dimensional cases. So far, we have been able to achieve some tangible results, using only the 1-dimensional and 2-dimensional models. If we succeed in generalizing the tessellation process, however, we can offer a consistent explanation for all kinds of high-dimensional cases. Given that this may not be feasible to do so in the near future, if we could extend the process at least to 3-dimensional cases, we would still be able to cover a great number of practical problems.

Bibliography

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Allen, J. R., and Kennedy, K. A parallel programming environment. *IEEE Software 2*, 4 (July 1985), 21–29.

[3] Allen, J. R., and Kennedy, K. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems 9*, 4 (Oct. 1987), 491–542.

[4] Alliant Computer Systems Corporation. *CONCENTRIX C Handbook.* Acton, MA, Aug. 1986.

[5] Alliant Computer Systems Corporation. *FX/Fortran Language Manual.* Acton, MA, Jan. 1986.

[6] Alliant Computer Systems Corporation. *FX/Series Architecture Manual.* Littleton, MA, Jan. 1986.

[7] Ametek Computer Research Division. *Concurrent Processing on Hypercube*, Feb. 1987.

[8] BBN Advanced Computers Inc. *The Butterfly GP1000 Switch Tutorial.* Cambridge, MA.

[9] BBN Advanced Computers Inc. *Programming in Fortran with the Uniform System.* Cambridge, MA.

[10] BBN ADVANCED COMPUTERS INC. *Programming in C with the Uniform System.* Cambridge, MA, 1988.

[11] BBN ADVANCED COMPUTERS INC. *Inside the GP1000.* Cambridge, MA, May 1989.

[12] BOUKNIGHT, W., DENENBERG, S., MCINTYRE, D., RANDALL, J., SAMEH, A., AND SLOTNICK, D. The Illiac IV system. *Proceedings of the IEEE 60*, 4 (Apr. 1972), 369–388.

[13] BRANTLEY, W., MCAULIFFE, K., AND WEISS, J. RP3 processor-memory element. In *Proceedings of the 1985 International Conference on Parallel Processing, University Park, PA* (Aug. 1985), IEEE, IEEE Computer Society, pp. 782–789.

[14] BREWER, O., DONGARRA, J., AND SORENSEN, D. Tools to aid in the analysis of memory access patterns for Fortran programs. *Parallel Computing 9* (1988/1989), 25–35.

[15] COFFMAN, E. G., AND DENNING, P. J. *Operating Systems Theory.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[16] DAVIES, J., HUSON, C., MACKE, T., LEASURE, B., AND WOLFE, M. The KAP/S-1: An advanced source-to-source vectorizer for the S-1 Mark II a supercomputer. In *Proceedings of the 1986 International Conference on Parallel Processing, St. Charles, IL* (Aug. 1986), IEEE, IEEE Computer Society Press, pp. 833–835.

[17] DEKKER, E. The Cray-2 architecture. In *Parallel Computing 89*, D. Evans, G. Joubert, and F. Peters, Eds. Elsevier Science Publishers B. V., 1990, pp. 575–580.

[18] DENNING, P. J. The working set model for program behavior. *Communications of the ACM 11*, 5 (May 1968), 323–333.

[19] DONGARRA, J. J., AND SORENSEN, D. C. SCHEDULE: Tools for developing and analyzing parallel Fortran programs. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds. The MIT Press, 1987, ch. 15, pp. 363–394.

[20] ETA SYSTEMS. *ETA-10 System Overview: Introduction*, Feb. 1986.

[21] FERNBACH, S., Ed. *Supercomputers: Class VI Systems, Hardware and Software.* North-Holland, 1986.

[22] FERRANTE, M. Cyberplus and Map V interprocessor communications for parallel and array processor systems. In *Multiprocessors and Array Processors*, Karplus, Ed. Simulation Councils, Inc., San Diego, CA, Jan. 1987, pp. 45–54.

[23] FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers C-21*, 9 (Sept. 1972), 11–23.

[24] GALLIVAN, K., JALBY, W., AND GANNON, D. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of International Conference on Supercomputing, St. Malo, France* (July 1988), pp. 238–253.

[25] GANNON, D., JALBY, W., AND GALLIVAN, K. Strategies for cache and local memory management by global program transformation. *Parallel and Distributed Computing 5*, 5 (Oct. 1988), 587–616.

[26] GEHANI, N. *Ada: Concurrent Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[27] GEHANI, N., AND ROOME, W. Concurrent C. *Software - Practice and Experience 16*, 9 (Sept. 1986), 821–844.

[28] GELERNTER, D., CARRIERO, N., CHANDRAN, S., AND CHANG, S. Parallel programming in Linda. In *Proceedings of the 1985 International Conference on Parallel Processing, University Park, PA* (Aug. 1985), D. Degroot, Ed., IEEE, IEEE Computer Society Press, pp. 255–263.

[29] GILMORE, P. A. The massively parallel processor. Tech. Rep. GER-17272, Goodyear Aerospace Corporation, Akron, OH, May 1985.

[30] GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The NYU ultracomputer — designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers C-32*, 2 (Feb. 1983), 175–189.

[31] GRAHAM, J., AND RATTNER, J. Expert computation on the iPSC concurrent computer. In *Multiprocessors and Array Processors*, Karplus, Ed. Simulation Councils, Inc., San Diego, CA, Jan. 1987, pp. 167–176.

[32] GUZZI, M. Cedar Fortran reference manual. Tech. Rep. 601, Center for Supercomputer Research and Development, University of Illinois, Urbana, Illinois., Nov. 1986.

[33] HANSEN, P. The programming language concurrent Pascal. *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975), 199–206.

[34] HAWKINSON, S. The FPS T series: A parallel vector super computer. In *Multiprocessors and Array Processors*, Karplus, Ed. Simulation Councils, Inc., San Diego, CA, Jan. 1987, pp. 147–156.

[35] HILLIS, W. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[36] HOARE, C. Monitors: An operating system structuring concept. *Communications of the ACM 17*, 10 (Oct. 1974), 549–557.

[37] HOARE, C. Communicating sequential processes. *Communications of the ACM 21*, 8 (Aug. 1978), 667–677.

[38] HUDAK, D. E., AND ABRAHAM, S. G. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the ACM International Conference on Supercomputing* (1990), ACM, ACM Press, pp. 187–200.

[39] HUSON, C., MACKE, T., DAVIES, J., WOLFE, M., AND LEASURE, B. The KAP/205: An advanced source-to-source vectorizer for the Cyber 205 supercomputer. In *Proceedings of the 1986 International Conference on Parallel Processing, St. Charles, IL* (Aug. 1986), IEEE, IEEE Computer Society Press, pp. 827–832.

[40] HWANG, K. Advanced parallel processing and supercomputer architectures. In *Proceedings of the IEEE* (Oct. 1987), vol. 75 of *10*.

[41] HWANG, K., AND BRIGGS, F. A. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[42] JORDAN, H. F. The Force. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds. The MIT Press, 1987, ch. 16, pp. 395–436.

[43] KALLSTROM, M., AND THAKKAR, S. S. Programming three parallel computers. *IEEE Software 5*, 1 (Jan. 1988), 11–22.

[44] KARP, A. H. Programming for parallelism. *IEEE Computer 20*, 5 (May 1987), 43–57.

[45] KARP, A. H., AND BABB, R. G. A comparison of 12 parallel Fortran dialects. *IEEE Software* (Sept. 1988), 52–67.

[46] KERNIGHAN, B., AND RITCHIE, D. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[47] KUCK, D. J. *The Structure of Computers and Computations*, vol. 1. John Wiley and Sons, New York, 1978.

[48] KUCK, D. J., DAVIDSON, E. S., LAWRIE, D. H., AND SAMEH, A. H. Parallel supercomputing today and the Cedar approach. *Science 23I* (Feb. 1986), 967–231.

[49] KUCK, D. J., KUHN, R., PADUA, D., LEASURE, B., AND WOLFE, M. Dependence graphs and compiler optimizations. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages, Williamsburg, VA* (Jan. 1981), ACM, pp. 207–218.

[50] LAWRIE, D. Access and alignment of data in an array processor. *IEEE Transactions on Computers C-24*, 12 (Dec. 1975), 1145–1155.

[51] LEE, R. On hot spot contention. *ACM SIGARCH 13*, 5 (Dec. 1985), 15–20.

[52] LEWIS, T. Issues in parallel programming:why aren't we having fun yet? *Supercomputing Review* (July 1990), 34–37.

[53] LOVEMAN, D. Program improvement by source-to-source translation. *Journal of the ACM 20*, 1 (Jan. 1977), 121–145.

[54] LUSK, E. L., AND OVERBEEK, R. A. A Minimalist approach to portable, parallel programming. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds. The MIT Press, 1987, ch. 14, pp. 351–362.

[55] MAY, D. Occam. *ACM SIGPLAN Notices 18*, 4 (Apr. 1984), 69–79.

[56] MEHROTRA, P., AND VAN ROSENDALE, J. The BLAZE language: A parallel language for scientific programming. Tech. Rep. 85-29, ICASE, NASA Langley Research Center, Hampton, VA, May 1985.

[57] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Communications of the ACM 29*, 12 (Dec. 1986), 1184–1201.

[58] PALMER, J. The NCUBE family of parallel supercomputers. In *Multiprocessors and Array Processors*, Karplus, Ed. Simulation Councils, Inc., San Diego, CA, Jan. 1987, pp. 177–187.

[59] PFISTER, G. F., BRANTLEY, W., GEORGE, D., HARVEY, S., KLEINFELDER, W., MCAULIFFE, K., MELTON, E., NORTON, V., AND WEISS, J. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing, University Park, PA* (Aug. 1985), IEEE, IEEE Computer Society, pp. 764–771.

[60] PFISTER, G. F., AND NORTON, V. A. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers C-34*, 10 (Oct. 1985), 943–948.

[61] POLYCHRONOPOULOS, C. D. Parallel programming trends: Specifying and exploiting parallelism. csrd.

[62] POLYCHRONOPOULOS, C. D. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Aug. 1986.

[63] SMITH, A. J. Cache memories. *Computing Surveys 14*, 3 (Sept. 1982), 473–530.

[64] SMITH, B. Parallel computing forum. In *Proceedings of the IFIP WG 2.5 Working Conference* (Amsterdam, Netherlands, Aug. 1988), M. Wright, Ed., North-Holland, p. 235.

[65] SNIR, M. Communication complexity in parallel computations. Presentation at Indiana University on October 6, 1989.

[66] THOMPSON, J. R. The Cray-1, the Cray X-MP, the Cray-2 and beyond: The supercomputers of Cray research. In *Supercomputers Class VI Systems, Hardware and Software*, S. Frenbach, Ed. North-Holand, 1986, pp. 69–81.

[67] TUCKER, S. The IBM 3090 system: An overview. *IBM Systems Journal 25*, 1 (Jan. 1986), 4–19.

[68] WOLF, M. E., AND LAM, M. S. An algorithm to generate sequential and parallel code with improved data locality. Computer Systems Laboratory, Stanford University.

[69] WOLFE, M. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Ill, Oct. 1982.

[70] WOLFE, M. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming 15*, 4 (Aug. 1986), 279–294.

[71] WOLFE, M. Iteration space tiling for memory hierarchies. In *Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Programming, Philadelphia, PA* (Dec. 1987), G. Rodrigue, Ed., Society for Industrial and Applied Mathematics, pp. 357–361.

[72] WOLFE, M. More iteration space tiling. In *Proceedings Supercomputing '89, Reno, Nevada* (Nov. 1989), ACM, ACM Press, pp. 655–664.

[73] WOLFE, M., AND BANERJEE, U. Data dependence and its application to parallel processing. *International Journal of Parallel Programming 16*, 2 (Apr. 1987), 137–178.

Appendix A

Butterfly Architecture

The GP1000, a BBN Butterfly parallel machine, is of the MIMD type. It composed of up to 256 *Processor Nodes (PN)*, which are tightly-coupled by an interconnection network, the *Butterfly Switch*. Figure 42 illustrates the basic architecture of the GP1000.

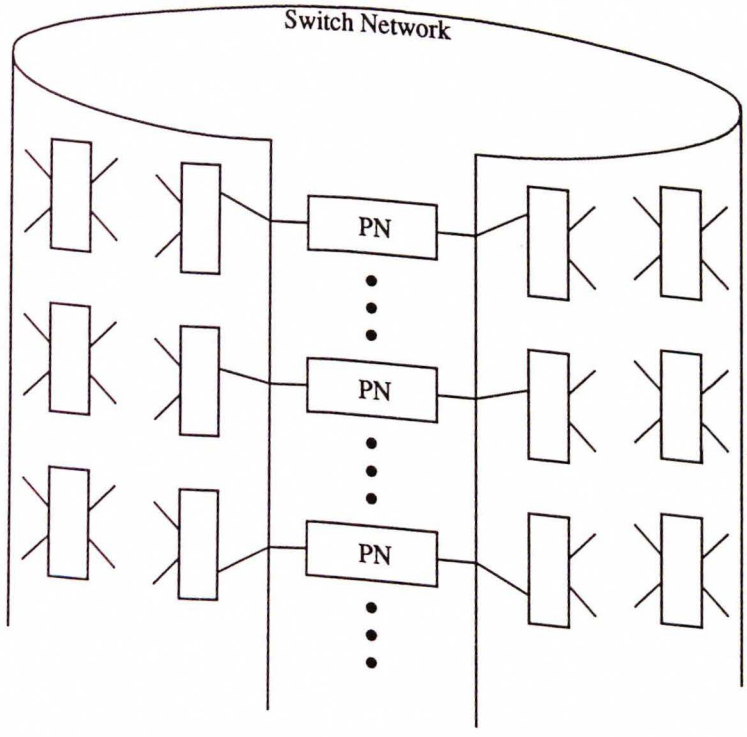


Figure 42: BBN Butterfly Architecture

All PNs are identical in that they are connected with a switch network in the same manner and that any set of PNs can run an application program with the same result. Each PN has a processor and 4M bytes of memory. All memory modules of

the PNs form a shared memory of the machine. For a processor on a PN, its own memory module is considered as local to itself, while all others are external. Any processor can access data in local memory as well as external memory via the switch network. The access time to local data is much faster than that to external data.

A.1 Processor Node (PN)

Each PN consists of an *MC68020* microprocessor, an *MC68881* floating-point co-processor, 4M bytes of *memory*, an *MC68851* memory management unit and an AM2901 bitslice processor, which is a very important co-processor called the *Processor Node Controller (PNC)*.

The Butterfly machine adopts the paged virtual memory system whose page size is 8K bytes. The virtual addresses are transparent to all application programs. The MC68020 and MC68881, which handle instructions, use virtual addresses, and all other components that handle data use physical addresses. The MC68881 receives 32-bit virtual addresses that are provided by the MC68020/MC68881 and translates them into physical addresses, using a page table stored in its address translation cache. A physical address consists of an 8-bit PN number and a 24-bit local address of 4M bytes memory module of that PN. The PNC receives physical addresses from the MC68851, compares the 8-bit PN number and its own PN number. If the two numbers are identical, the physical address is viewed as a local address; otherwise, it is viewed as an external address. In the former case, the PNC performs read or write operation on its local memory. In the latter, it creates and sends a packet requesting other PNs to perform read or write operation. If the requested operation is read, the PNC waits for the reply. After the PNC gets the reply, it then passes the data to the MC68020/MC68881. The access time to a byte, word, or long-word in local memory is 0.53 μ sec for a read operation and 0.38 μ sec for a write operation. The

access time for a byte, word, or long-word in remote memory is 7 μ sec for a read operation and 4 μ sec for a write operation. The memory bandwidth capacity is 102M Bytes/sec.

The PNC controls all resources in each PN, and performs operations to provide the parallel processing capability that the MC68020 does not have. Those functions are to control all memory references, which enables it to perform atomic arithmetic/logical operations, to regulate all communication transactions, to maintain a 32-bit realtime clock with 62.5 μ sec resolution, and so on.

The PNC is in charge of communication with other PNs. It has a switch output port and input port, both of which are connected to the switch network. To prevent deadlock, each port has two buffers: a request buffer and an acknowledgement buffer.

The request buffer of the input port accepts messages that require a PN to send out a reply in any form, such as the incoming requests from other PNs to read the data in the PN. Its acknowledgement buffer accepts reply messages from other PNs or simple control messages, such as incoming requests from other PNs to write the data into the memory of the PN. Similarly, the request buffer and the acknowledgement buffer of the output port store outgoing messages that require a reply and do not require a reply, respectively. These two ports perform their functions independently. Even though one port may be full of messages that have not yet been processed, the other port can still function. When a message comes in to a PNC and the input port buffer that it is supposed to enter is full, then that message is rejected. The bandwidth capacity of the PNC switch port is 32M bits/sec.

The size of a message may be fixed (as in byte, word, or long-word) or of variable length in the case of a block transfer. The PNC creates a packet for every message that is to be sent to other PNs. Every packet has an 80-bit message head describing source and destination nodes, message type and message length, and so on.

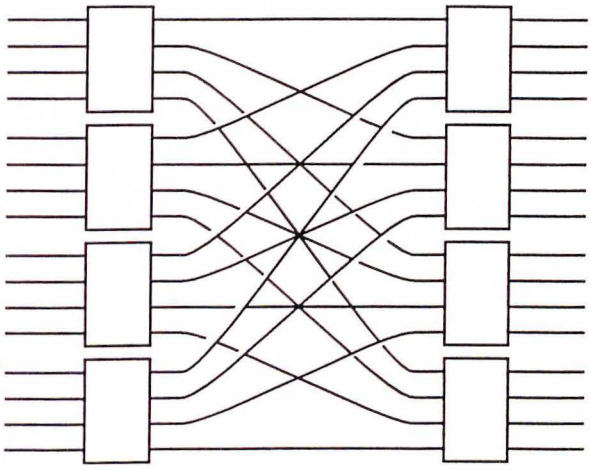


Figure 43: Switch Card

A.2 Switch Network

The basic element of a switch network is a 4-input and 4-output crossbar style switching node. A switch card is a circuit board with 8 switching nodes in two columns, and it functions as a 16-input and 16-output switch network. The connection between the two columns is of a *shuffle-exchange* network type, and since it is fixed on the circuit board, it cannot be customized. Thus, one switch card is the basic installation unit of a switch network. As shown in Figure 43, there is one and only one path from any input line to any output line. The system with 1 through 16 PNs uses only one switch card.

For the system with 17 to 64 PNs, even though $3(=\log_4 64)$-stage network is theoretically sufficient, a 4-stage network is required, because the basic installation unit is a 16-input and 16-output switch card. As an example of a 4-stage network, Figure 44 shows the switch network configuration of the Butterfly presently installed at Indiana University; this is the minimum configuration of a 4-stage network. As we can see in the diagram, there are four paths for each pair of source and destination PNs. These four paths are used as alternate paths when one path is not available

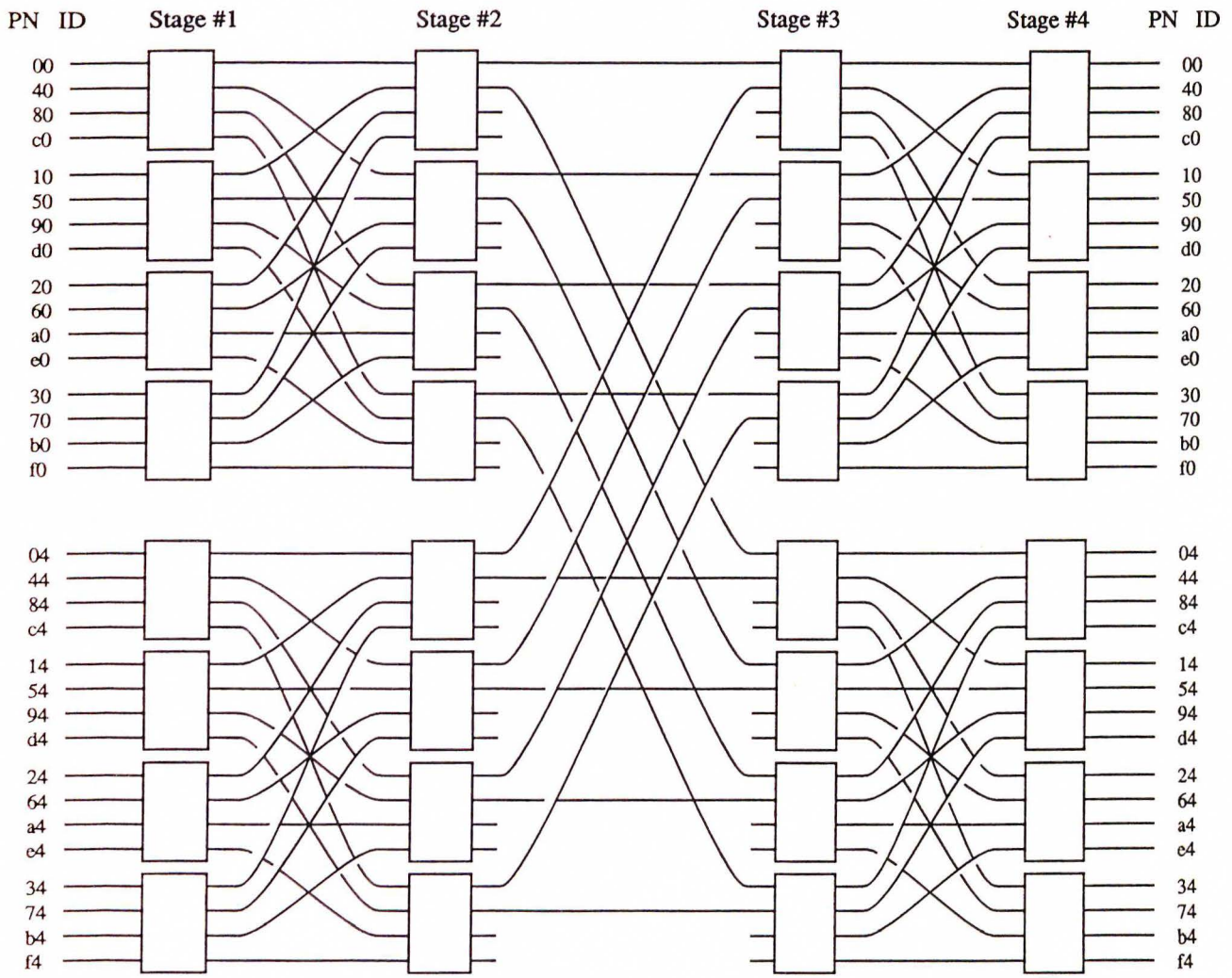


Figure 44: Switch Network of System with 32 PNs

because it is occupied by a message. The number of available alternate paths can be set at system boot time.

A larger system with more than 32 PNs extends the switch network by installing two additional switch cards vertically for every additional 16 PNs: one for the left two columns and the other for the right two columns. The second and the third stage of the network are connected via shuffle-exchange. The number of alternate paths is reduced as the number of PNs increases because the number of stages of the network is fixed at four, while the number of PNs that use the paths increases. There are four paths for 17 to 64 PNs, two paths for 65 to 128 PNs and one path for 129 to 256 PNs.

The bandwidth capacity of each path is 32M bits/sec, the bandwidth of the PNC port. Every path can function at the same time. Thus, the system with 32 PNs has a maximum switch bandwidth of 1024M bits/sec.

A message from one PN to another goes through the path set by the PNC of the sender PN. As a message is transmitted, if it encounters a conflict at any switching node on the path, it retreats to the PNC that sent the message; then the PNC sends it again using an alternate path, thus improving the performance of the switch network. Message *conflict* occurs when two messages arrive at the same switching node and try to exit through the same output port.

A.3 Memory System

The memory system of the GP1000 is a two level hierarchy with virtual memory and real memory of 4M bytes per PN. An address space of virtual memory is 4G bytes with the page size of 8K bytes. A real memory address consists of an 8-bit PN number and a 24-bit local address of the 4M bytes memory module of that PN. Memory space is initially allocated on the virtual memory and when a location is referenced, the page containing the location is loaded into real memory, and the correspondences between

virtual addresses and real addresses are kept in the page table cache of the MC68851 memory management unit.

The address space is divided into two parts, the *process private* area and the *shared* area. Each PN has its own process private area. The process private area can be accessed only by the PN that owns it. It is further divided into a *text* area, *heap* area, and *stack* area. A program code is loaded into the text area of all PNs. When parallel execution is initiated, each PN participating in the parallel execution runs the program segment stored in its own memory. The heap area is used for the C global variables and for the space allocated dynamically by `malloc()` or `calloc()`. The stack area is used for the C local variables of procedures. The shared area is used for the space allocated by `UsAlloc()` family procedures, and can be shared between PNs by using `Share()` procedure.

The procedure `Share()` takes the location of the variable declared as the C global variable, and makes a copy of the value of the location onto the locations with the same variable name as the argument of each PN at the time of the initiation of the parallel execution. If the value of the location used in the procedure `Share()` is not an address, then the value is copied.

Using Figure 45, which refers to Uniform System (US) library procedures, we will briefly explain the memory management scheme.

The variables a, b, and n are allocated to the heap area of all PNs. The contents of n, a, and b are 10, a pointer to a location in the shared area, and a pointer to a location in the private area, respectively. `GenTaskForEachProc()` is a procedure that creates one process for each PN, and the created processes execute the procedure `Worker()` in parallel. Before all PNs start the parallel execution, contents of all of the declared variables that are shared by `Share()` procedure are copied into the corresponding locations of all PNs. The procedure `Worker()` is executed by all PNs at the same time. No conceivable problem arises in accessing the variables n and a,

```
#include <us.h>
int *a, *b, n;

Worker(dum)
int dum;
{
    a[1] = n;
    b[2] = n;
}

main()
{
    InitializeUs();
    n = 10; Share(&n);
    a = (int*)UsAlloc(n * sizeof(int)); Share(&a);
    b = (int*)malloc(n * sizeof(int)); Share(&b);
    GenTaskForEachProc(Worker, 0);
}
```

Figure 45: Uniform System C Program

because n of each PN has the value 10, and a points to a location in the shared area. However, accessing $b[2]$ by all PNs, except the one PN that has the memory to which the array b is allocated, will cause an address error, because the array b is allocated on the heap area, the private area that cannot be shared between PNs.

A.4 Block Transfer

The Butterfly machine supports a very efficient operation for transferring blocks of data from one PN to another. The block transfer operation is implemented by the PNC microcode. Once the path has been set up for block transfer, the path is held for the block transfer until it finishes, and it is performed at the full 32M bits/sec bandwidth of a path through the switch network. Because of the initial set-up time, a long message is preferred to a short one.

While the PN is engaged in the block transfer operation as a source or destination, 75 % of total memory bandwidth is used for the block transfer, and 25 % is used for local processes. Thus the performance of source and destination PNs is significantly reduced.

The maximum data size for block transfer is 64K bytes. If the message is more than 256 bytes, then it is split into blocks of 256 bytes and sent one by one.

The block transfer time is 72 μ sec for a 256-byte block, and 8 μ sec + $\frac{1}{4}$ μ sec/byte for a block of less than 256 bytes.

Curriculum Vitae

Mann-Ho Lee was born on September 4, 1952 in Chonju, Korea. He attended Seoul National University in Korea, obtaining a Bachelor of Engineering degree in Applied Mathematics in 1975. He received a Master of Science degree in Computer Science at the Korea Advanced Institute of Science and Technology in Seoul in 1977. He worked for a research institute for defense development from 1977 to 1980, and taught at Chungnam National University at Tajon, Korea as a faculty member of the Computer Science Department from 1980 to 1984. He entered Indiana University at Bloomington in 1984 as a graduate student in the Computer Science Department. He worked as an associate instructor and research assistant during his stay in Bloomington and completed Doctor of Philosophy degree in 1991.