Automatic Array Partitioning and Distributed-Array Compilation for Efficient Communication

Hung-Yu Tseng

CENG 98-16

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213 740-4484)
July 1998

Abstract

Automatic Array Partitioning and Distributed-Array Compilation for Efficient Communication

Hung-Yu Tseng

Advisor: Jean-Luc Gaudiot

To obtain high performance on distributed-memory machines, one of the key factors is to minimize the communication overhead. The communication overhead can be reduced by two major approaches: the communication latency-reduction technique and the communication latency-hiding technique. These approaches have resulted to the active researches on the automatic array partitioning, and on the distributed array compilation for data parallel languages, such as High Performance Fortran. This research will demonstrate that our Smith-Normal-From lattice algebra successfully serves as a foundation to develop frameworks for both approaches on the communication optimization.

Automatic array partitioning: A general solution of communication-free partitioning is derived for arrays in a forall loop. The derivation is based on Smith Normal Form decomposition to the matrix which characterizes the array references in a forall loop. When communication-free partitioning is not possible, we derive the partitioning equations which locate all remote data to one processor. Thus, at most one block-communication is required for each processor to get the remote data during computation

Distributed Array compilation: In HPF array distribution which may involve alignment and cyclic(m)-distribution, the enumeration of communication set exhibits a regular pattern which can be modeled as an integer lattice. Our approach derives a parametric solution for such integer lattice by using the Smith-Normal-Form analysis. Based on this integer lattice, the SPMD codes are then constructed.

Contents

List Of Figures

Chapter 1

Introduction

Getting high performance on a distributed-memory machine requires not only finding parallelism in the program but also minimizing the communication overhead.

Writing parallel programs for distributed-memory machines has historically been one of the difficulties with parallel machines. The work of writing a parallel program requires programmers to keep track of enormous information unrelated to the actual computations; and it was very likely not portable to other machines. Thus, it is a need to seek a better way to write portable programs that would perform well on a variety of parallel machines. Consequently, many research projects have been aimed at providing programming languages that allow programmers to express their algorithms by using a global name space. This has motivated the design of Data-Parallel languages like HPF (High Performance Fortran)[50], which allows the programmer to write sequential or shared-memory parallel programs that are annotated with directives specifying how arrays should be distributed. The shared-memory programming paradigm keeps programmers away from the low-level details of the target distributed-memory machines, and from programming explicit processes or explicit inter-processes communication. Therefore, it results to the research on the **distributed-array compilation** to compile data-parallel languages for the distributed-memory parallel computers.

The performance of parallel execution in most message-passing parallel machines is significantly affected by remote data accesses, since the communication latency can be several orders of magnitude higher than the computation costs. In scientific applications, arrays are usually the largest data structures involved in the most time consuming computations. In order to reduce communications, therefore, arrays

must be appropriately partitioned and distributed to processors for efficient parallel operations. This has motivated the research on the **automatic array partitioning** from the program analysis.

1.1 Automatic Array Partitioning

Automatic data partitioning methods to generate communication-free partitionings of arrays focuses on partitioning the arrays in nested loops under non-duplicating criteria maintaining the load balance. Some approaches have focused on analyzing the DOALL loops and partition the loop iterations along with corresponding data to achieve a communication free partitioning [40]. The work also identifies the conditions under which a communication free partition of the loop and the data referenced, is impossible. Ramanujam and Sadayappan [40] have used hyperplanes to partition arrays and have derived a sufficient condition for communication-free partitioning. Li and Chen [36] have presented a method which aligns the indexes of arrays to reduce memory traffic. Basically, an alignment function (permutation) is used to permute the indexes in arrays. The analysis is based on a graph algorithm which incurs NP-complex costs. Some researchers have addressed the problem of data and code partition in a unified framework and developed techniques based on the generalized loop transformations [5]. This method uses tiling to maximize parallelism and for converting more expensive re-organizing communication to the pipelined one. The work deals with the cases of both DOALL as well as DOACROSS loops and presents a mathematical framework to perform decomposition. It also proves that the problem of dynamic decomposition is NP-hard. The problems of data alignment to minimize communication cost has been proved to be NP-complete in [49]. They also described a polynomial time approximation algorithm to solve the problem. Orienting computation to align with communication is another approach which has been attempted in [46]. They use loop interchange and reversal transformations to orient the computation. For some cases, Mace [38] has proved that finding an optimal data storage pattern was NP-complete in its solution-space graph. The concept of Alignment Graphs (AG) has been proposed along with a trapezoid data distribution and alignment algorithm to perform data decomposition preserving load balance [56]. D'Hollander [12] has used the integer programming method to partition

the computation into independent subsets for a Do-loop with constant dependence. Affine methods [6, 37, 51] use affine equations to partition the computation and the arrays in a Do-loop at the same time.

1.2 Distributed Array Compilation

In contrast, some data-parallel language systems, like HPF [22] or Fortran D [31], provide directives for programmers to specify how data are distributed. For a number of important numerical applications which are regular in nature, these approaches produce a good program partition using the standard data distributions provided by the compiler. One of the primary merits of such approaches is that they save a lot of effort on the compiler's part in analyzing the program to decide about the data distribution [33, 44]. The programmer is provided a familiar uniform logical address space and specifies the data distribution by directives. The compiler then exploits these directives to allocate arrays in the local memories, to assign computations to processors and to move data between processors when required. The compilation techniques also carry out a number of optimizations to eliminate or vectorize the communication in the partitioned program to minimize the program execution time. Static analysis aims to improve performance over run-time resolution [10] which includes a lot of pure overhead in form of guards and tests. Many static compilation schemes have been considered; they differ in important points such as interleaving computation and communication as in [33], or having identical management of local and non-local data such as in [17]. However, they all use three basic sets: Compute(s) is the part of the index set which is local to processor s; Send(s) (resp Received(s)) is the part of a distributed array that has to be sent (resp received) by processor s when owner computes rule is applied. The central problem of static analysis is to define these sets at compile-time, and in an efficient form.

Static Analysis of data-parallel programs, for the generation of distributed code, has been proposed by many authors, for instance [42, 32, 17, 43, 53]. To be amenable to static analysis, the references must be affine functions of the parallel loop indices, a reference being an access or alignment function, and the loop bounds must be defined by affine inequalities. These assumptions are the weakest possible. Under these assumptions, deriving efficient closed forms of the previous sets for the most

general block-cyclic distribution is an open problem. [17] gives a general compiling scheme under the weakest assumptions, but provides closed forms only when indices are independent: for instance T[j,i], but not T[2i+j,i-j]. [42] uses a finite state machine approach, allowing optimal memory utilization, but restricts references to array sections and uses integer divides. [43] solves the same problem with a virtualization method. Other special cases have been solved, for unit strides in [53], for one-dimensional arrays in [32] and [26]. Two major costs have to be considered for the code generation scheme: the computing cost, and the communication cost. The computing cost is all the overhead required to compute local indices, and, when a communication occurs, to compute the parameters of the communication, the destination processors and the local addresses. As pointed out by [32], naive resolution leads to a symbolic form involving integer divides for each forwarded data, which may be as inefficient as run-time resolution. The communication cost depends on the volume and number of data to send to a remote processor, cannot be modified, because it is fixed by the placement function (e.g. ALIGN and DISTRIBUTE directives). At the code generation level, optimization is only directed towards the number of communications, by aggregating all data that are to be sent to the same processor. Although this may seem a very specialized problem, the overwhelming part of startup in message cost makes this optimization a major component of performance, as shown in [53]. The enumeration of the local sets and the enumeration of the communication sets are both important issues in distributed array compilation of data parallel languages. Both enumerations may not be straightforward when the distributed arrays are aligned to each other and block-cyclic distributed, which can result in heavy compilation-time and/or run-time overheads. The overheads can be reduced when the enumerations are able to be expressed in simple closed forms.

In order to construct the communication sets, one single pass of local array with destination evaluation is usually required, which also results heavy run-time overheads.

1.3 Our Approaches

This research will demonstrate that our Smith-Normal-From lattice algebra successfully serves as a foundation to develop frameworks for both approaches on the communication optimization.

Automatic array partitioning: A general solution of communication-free partitioning is derived for arrays in a forall loop. The derivation is based on Smith Normal Form decomposition to the matrix which characterizes the array references in a forall loop. When communication-free partitioning is not possible, we derive the partitioning equations which locate all remote data to one processor. Thus, at most one block-communication is required for each processor to get the remote data during computation

Distributed Array compilation: Optimizing communication is a key issue to generating efficient SPMD codes in compiling distributed arrays on data parallel languages, such as High Performance Fortran. In HPF array distribution which may involve alignment and cyclic(m)-distribution, the enumeration of communication set exhibits a regular pattern which can be modeled as an integer lattice.

Many techniques of the communication set enumeration have been proposed. Unlike other works on this problem, our approach derives a parametric solution for such integer lattice by using the Smith-Normal-Form analysis. We also present our algorithm for the SPMD code generation. In our approach, we first derive a single equation which expresses at once both message packing in the sending processor and message unpacking in the receiving processor. From this equation, a particular sequence can be derived so that a message can be constructed and referenced by the same sequence on both processors. Based on this sequence derivation, the SPMD codes for the message packing, the message unpacking, and the iteration of the computation are then constructed.

1.4 Contribution

As compared to the previous efforts on the automatic array partitioning and on the distributed-array compilation, our approach have exhibits following advantages based on our novel analytical methods:

- Communication-free array partitioning:

  A general method which will provide all communication-free partitionings is derived for arrays in a DoAll loop.

- One block-communication partitioning:

  We derive the partitioning equations which locate all remote data to one processor.

- Distributed array compilation:

  In the case of the general data distribution in cyclic(m) with strided alignment, the major advantage of our method is that we are able to symbolically derive the basis vectors of the lattice which models the enumeration of communication set. Based on the symbolic basis vectors, our algorithm of the SPMD-code generation requires only a 4×3 matrix triangular factorization as the *inspector*-like run-time codes. When the parameters are known values, the SPMD code can be completely constructed without any *inspector*-like run-time codes. It avoids the run-time code for the evaluation of the destination in performing message packing. It does not require the run-time code for pre-calculating the unpacking information in performing message unpacking. Our method provides an integrated solution for both message packing and message unpacking problems without incurring such run-time overhead.

1.5 Overview

We close this chapter by providing an overview of this thesis. Our research focuses on two areas: the automatic array partitioning and the distributed-array compilation. Our approach is based on the lattice algebra, which has elegant foundation in mathematics. In next chapter, general concepts related to the distributed array parallel computer are presented. We will also discuss some of the related works. In chapter 3, we will present and discuss all the lattice algebra related in this research. In chapter 4, our array partitioning scheme is presented, which includes the communication-free partitioning and the block-communication partitioning. In chapter 5, our distributed-array compilation scheme is presented, which includes the

analytic method for local addresses enumeration, the communication enumeration method, and our SPMD code-generation scheme. The conclusion can be found in chapter 6.

Chapter 2

Background

In this chapter, general concepts related to this thesis are presented, such as data-parallel languages, SPMD execution models, and data distributed directives. The related researches are also discussed here.

2.1 Single Program Multiple Data (SPMD) Model

It is widely recognized that scaleable parallel machines can continue to increase the computing power available to scientists. However, programming such machines is also known to be cumbersome. To obtain the performance, programmers must write explicit parallel programs and solve many machine-dependent details. When a new architecture arrives, the programs are required to be changed as the source codes are machine dependent. The scientists are then discouraged from utilizing the machines. To enable more efficient usage of the parallel machines, the programs should be easily design-able, portable, and maintained without sacrificing efficiency. The conventional vector computers have shown a successful machine-independent example. Its success is contributed to the automatic vectorizing compilers which can take high level languages, such as Fortran or C, and convert them to run efficiently on any vector machines by handling the machine dependent details. The high-level programs not only allow easily for maintenance and portability, but also enables scientists to concentrate on their algorithms instead of the low-level machine issues.

In the distributed-memory machines, the compiler systems usually produce target codes based on the SPMD model. The Single Program Multiple Data (SPMD) model provides a scheme for each processor to run the same program but execute different code dependent on its processor ID and the data stored in its local memory. The execution is based on the paradigm of **owner-compute-rule** where all updatings of the data are assigned to be performed in the processor owning the data.

2.2 Data Parallel Language

Data Parallel languages provide a Shared Memory Programming Model which can be executed on SIMD or MIMD computers, distributed or shared memory computers. Data parallel paradigm provides the most attractive machine independent model for programmers as it reflects problem and not machine. Its disadvantage is that hard to build compilers, especially the compiler for the distributed memory machines.

Some parallel languages try to make the problems of detecting parallelism, distributing data, and choosing granularity easier by making these operations more explicit in the program. To date, data-parallel programming has been the most successful, and most of its ideas have been adopted in recent versions of Fortran. Data placement is an essential part of a data-parallel algorithm, since the mapping of data to processors determines the locality of data references. Identifying the best distribution of the various data structures operated on by a data-parallel program is a global optimization problem and not generally tractable. Hence, data-parallel languages often provide the programmer with the ability to define how data structures are to be distributed. Although parallel computers have been commercially available for some time, their use has been mostly limited to academic and research institutions. This is mainly due to the lack of software tools available to develop parallel programs. Accordingly, there has been significant research in developing parallelizing compilers for Fortran codes. Most notable examples include Parafrase at the University of Illinois [16] and PFC at Rice University [3]. In this approach, the compiler takes a sequential Fortran 77 program as input, applies a set of transformation rules, and produces a parallelized code for the target machine. A sequential language, such as Fortran 77, hides a problem parallelism in sequential loops and in other sequential constructs. This makes the potential parallelism more difficult

to detect. Therefore, compiling a sequential program into a parallel program is not a natural approach. An alternative approach is to use a source programming language that can naturally represent an application using parallel constructs. Fortran 90 or HPF is such a language. The extensions may include the forall statement and compiler directives for data partitioning, such as decomposition, alignment, and distribution. A Fortran 90D/HPF parallel compiler exploits only the parallelism expressed in these parallel constructs. Fortran 90D/HPF compiler does not attempt to parallelize other constructs, such as do loops and while loops, since they are naturally sequential. Developing a compiler under this assumption becomes much easier. Also users can reliably understand where parallelism will be exploited.

2.2.1 DataParallel C

Dataparallel C [24][23] is a variant of the C* programming language, designed by Thinking Machine Corporation for its Connection Machines processor array. Data parallel C extends C to provide the programmer with access to a parallel virtual machine. It supports a variety of standard domain decomposition primitives, and it also allows the programmer to specify a custom mapping of data to the distributed memories of the hypercube. This compiler generates code suitable for execution on both the nCUBE 3200 and the Intel iPSC/2.

2.2.2 Fortran D

Fortran D is a research project to explore HPF compiler techniques, e.g. I/O. Initial experience with Fortran D showed that early HPF compilers were too machine specific; that they worked well with stencil programs but had to do better on linear algebra codes and pipelined codes. Preliminary experience with Rice's Fortran D compiler showed its very early state (it compiled only BLOCK decompositions, for example)

2.2.3 SUPERB and Vienna Fortran

The compilation system SUPERB (University of Vienna) [19] takes a sequential Fortran program and a specification of the desired data distribution. SUPERB then

converts the code to an equivalent program to run on a distributed memory machine by inserting the communication required and optimizing communications where possible. The user is able to specify arbitrary block distributions and the compiler performs dependence analysis to guide interactive program transformations. Recently, SUPERB has been adapted for a new language called Vienna Fortran 90. Vienna Fortran 90 is a language extension of Fortran 90 which enables the user to write programs for distributed memory machines using only global references. It is similar to Fortran 90D/HPF language. However, Vienna Fortran does not provide a data decomposition, but does support alignment and distribution directives.

2.2.4 High Performance Fortran (HPF)

High Performance Fortran Forum, an informal group of people from academia, industry and national labs, led by Ken Kennedy, developed a language called HPF (High Performance Fortran) [18][2]. HPF language combines the full Fortran 90 language with special user annotations dealing with data distribution It is expected that HPF will be the standard programming language for computationally intensive applications on many types of machines, such as traditional vector processors and newer massively parallel MIMD and SIMD multiprocessors. Companies that have already committed to developing compilers and/or supporting HPF include Intel, TMC, Portland Group(PGI), DEC, IBM, and others.

High Performance Fortran (HPF)[13] is a language definition agreed upon in 1993, and being widely adopted by systems suppliers as a mechanism for users to exploit parallel computation through the data-parallel programming model.

HPF evolved from the experimental Fortran-D system [3] as a collection of extensions to the Fortran 90 language standard [15]. We do not discuss the details of the HPF language here as they are well documented elsewhere [14], but simply note that the central tenet of HPF and data-parallel programming is that program data is distributed amongst the processors' memories in such a way that the "owner computes" rule allows the maximum computation to communications ratio. Language constructs and embedded compiler directives allow the programmer to express to the compiler additional information about how to produce code that maps well to

the available parallel or distributed architecture and thus runs fast and can make full use of the larger (distributed) memory.

2.3 Data Distribution Directives

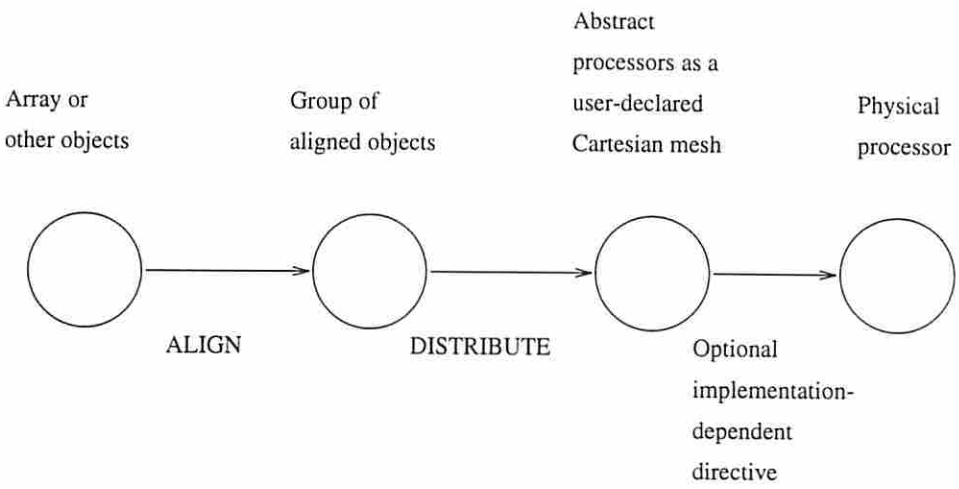
Data distribution can be done in two steps which separate machine independent problem parallelism from machine dependent details. The first step is to determine the best alignment among different arrays. To reduce unnecessary data movement, distributed arrays should be aligned with each other in a fashion that is usually determined by the underlying computation structure. The alignment of arrays depends on the program and is often machine-independent. The second step is to determine how arrays should be distributed to the underlying hardware and is therefore machine dependent. The objective of array distribution is to balance the computation load for each processor and to minimize the communication between processors. Array distribution is largely dependent on hardware, such as the number of processors, communication mechanisms, and interconnection topologies. Many data parallel languages, like Fortran D and HPF, provide users with annotation facilities for data partitioning. The annotation facilities take the form of compiler directives.

2.3.1 Data Alignment and Distribution Directives in HPF

Note: The content in this section is mostly originated from HPF documents, which can be obtained from the web-site: http://www.erc.msstate.edu/hpf. The description in this section is not intended to cover all the details of the HPF directives, however, it should describe the essentials of how data objects are specified to be mapped into processor memories by the directives.

HPF data alignment and distributions directives allow the programmer to advise the compiler how to assign array elements to processor memories. The model is that there is a two-level mapping of array elements to memory regions, referred to as "abstract processors." Array elements are first *aligned* relative to one another; this group of arrays is then *distributed* onto a rectilinear arrangement of abstract processors. (The implementation then uses the same or smaller number of physical

processors to implement these abstract processors.) The following diagram illustrates the model:



The basic concept is that every array is created with *some* alignment to an *array*, which in turn has *some* distribution onto some arrangement of abstract processors. If the specification statements contain explicit specification directives specifying the alignment of an array A with respect to array B, then the distribution of A will be dictated by the distribution of B; otherwise, the distribution of A itself may be specified explicitly. In either case, any such explicit declarative information is used when the array is created. If array A is aligned with array B, which in turn is already aligned to array C, this is regarded as an alignment of A with C directly, with B serving only as an intermediary at the time of specification. The alignment relationships form a tree with everything ultimately aligned to the array at the root of the tree; however, the tree is always immediately "collapsed" so that every object is related directly to the root. Every array which is the root of an alignment tree has an associated *template* or index space. Typically, this template has the same rank and size in each dimension as the array associated with it. The *distribution* step of the HPF model technically applies to the template of an array. Distribution partitions the template among a set of abstract processors according to a given pattern. The combination of alignment (from arrays to templates) and distribution (from template to processors) thus determines the relationship of an array to the processors; this relationship is referred as the *mapping* of the array.

Once an array has been created, it can be remapped by realigning it or redistributing an array to which it is ultimately aligned; but communication may be required in moving the data around. Sometimes it is desirable to consider a large index space with which several smaller arrays are to be aligned, but not to declare any array that spans the entire index space. HPF allows one to declare a TEMPLATE, which is like an array whose elements have no content and therefore occupy no storage; it is merely an abstract index space that can be distributed and with which arrays may be aligned.

2.3.2 DISTRIBUTE Directives

The DISTRIBUTED directive specifies a mapping of data objects to abstract processors in a processor arrangement. For example,

```
      REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
```

specifies that array A should be distributed across some set of abstract processors. If there are 50 processors, the directive implies that the array should be divided into groups of 200 elements, with A(1:200) mapped to the first processor, etc..

The block size may be specified explicitly:

```
      REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK(256))
```

There must be at least $\lceil 10000/256 \rceil = 40$ abstract processors if the directive is to be satisfied. The 40th processor will contain a partial block of only 16 elements.

HPF also provides a cyclic distribution format:

```
      REAL B(52)
!HPF$ DISTRIBUTE B(CYCLIC)
```

If there are 4 abstract processors, the first processor will contain $B(1:49:4)$.

Distributions may be specified independently for each dimension of a multidimensional array:

```
      INTEGER C(8,8), G(19,19)
!HPF$ DISTRIBUTE C(BLOCK, BLOCK)
!HPF$ DISTRIBUTE G(CYCLIC,*)
```

The array C will be carved up into contiguous rectangular patches, which will be distributed onto a two-dimensional arrangement of abstract processors. The "*" specifies that array C is not to be distributed along its second axis; thus an entire row is to be distributed as one object.

```
!HPF$ PROCESSORS P(16)
INTEGER C(100)
!HPF$ DISTRIBUTE C(BLOCK) ONTO P
```

Distributing the array C BLOCK, which in this case would mean the same as BLOCK(7).

2.3.3 PROCESSORS Directives

The PROCESSORS directive declares one or more rectilinear processor arrangements, specifying for each one its name, its rank, and the extent in each dimension.

Examples:

```
!HPF$ PROCESSORS P(10)
!HPF$ PROCESSORS Q(12,12)
!HPF$ PROCESSORS R(1972:1997, -20:17)
```

2.3.4 ALIGN Directives

The ALIGN directive is used to specify that certain data objects are to be mapped in the same way as certain other data objects. The ALIGN directive is designed to make it particularly easy to specify explicit mappings for all the elements of an array at once.

Examples:

!HPF$ ALIGN A(I) with B(2*I+1)

"I" is called align-dummy. The subscript is intended to be a linear function of align-dummy.

!HPF$ ALIGN A(:,*,K) with B(:,K+4)

":" can be replaced by a new dummy variable, and that every "*" can be replaced by an unused dummy variable. The directive above may be transformed into its equivalent:

!HPF$ ALIGN A(I,J,K) with B(I,K+4)

An asterisk "*" as an align-subscript indicates a replicated representation. Each element of the alignee is aligned with every position along that axis of the align-target.

!HPF$ ALIGN A(I) with B(I,*)

means that a copy of A is aligned with every column of B.

Note: The *MDMH* method can derive a set of good ALIGN and cyclic DISTRIBUTE directives automatically.

2.3.5 TEMPLATE Directives

A template is simply an abstract space of indexed positions; it can be considered as an "array of nothings" (as compared to an array of integers", say). A template may be used as an abstract *align-target* that may then be distributed.

The TEMPLATE directive declares one or more templates, specifying for each the name, the rank, and the extent in each dimension.

Examples:

!HPF$ TEMPLATE A(10)

!HPF$ TEMPLATE B(100,100)

2.4 Related Works on Automatic Array Partitioning

In this section, we will discuss some of the related works on automatic array partitioning.

2.4.1 Hyperplane Method

Ramanujam and Sadayappan have proposed in [40] the hyperplane method. In the hyperplane method, a two dimensional array, for example, can be distributed by applying the following equation:

$$PE\# = c = \alpha i + \beta j \tag{2.1}$$

where i and j are the array indexes of the elements.

Example 2.4.1 *Mapping a* 10×10 *array to PEs by the following hyperplane equation.*

$$PE\# = c = 1 \cdot i + 3 \cdot j \tag{2.2}$$

■

We have following partitioning result:

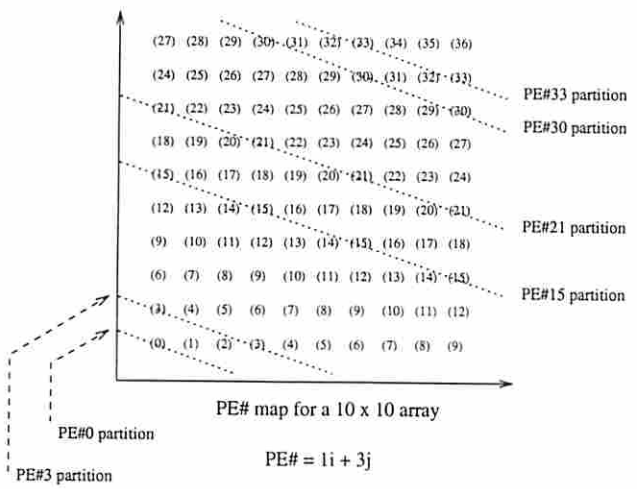


Figure 2.4.1

Figure 2.4.1 shows the PE# for each element in the array by equation (2.2). Note that:

1. There is a total 36 PEs.
2. PE#3, for example, contains the set of elements $\{(0,1),(3,0)\}$ which are mapped to value 3 by equation (2.2).

■

The hyperplane method is aimed at finding at least one communication-free array partitioning. The following shows an example in more generic case.

Example 2.4.2 [1]

1. for i = 1 to N
2. for j = 1 to N
3. A[i,j] := B[i',j'] + B[i",j"]
4. end for
5. end for

where i',j',i" and j" are linear combinations of i and j, and have the generic forms as described by equation (2.3).

$$\begin{aligned} i' &= a_{11}i + a_{12}j + a_{10} \\ j' &= a_{21}i + a_{22}j + a_{20} \\ i'' &= b_{11}i + b_{12}j + b_{10} \\ j'' &= b_{21}i + b_{22}j + b_{20} \end{aligned} \tag{2.3}$$

Let $c = \alpha i + \beta j$ *be the generic distribution equation for array A, and* $c' = \alpha' i + \beta' j$ *be the generic distribution equation for array B.*

Can the partitions induced by the equations for A and B incur zero-communication in computation?

■

[1] The domain of the loop indexes needs to reduce to a set where all references will not go out of bounds.

From the hyperplane method, a matrix equation is derived for each reference of array B. There are two references to array B in the example, so a system of two matrix equations (2.4) are derived.

$$\begin{bmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ -a_{10} & -a_{20} & 1 \end{bmatrix} \begin{bmatrix} \alpha' \\ \beta' \\ c' \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ c \end{bmatrix}$$
$$\begin{bmatrix} b_{11} & b_{21} & 0 \\ b_{12} & b_{22} & 0 \\ -b_{10} & -b_{20} & 1 \end{bmatrix} \begin{bmatrix} \alpha' \\ \beta' \\ c' \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ c \end{bmatrix} \tag{2.4}$$

In the hyperplane method, it has been shown that communication-free partitioning exists if there is a non-zero solution for the parameters α, β, α' and β'. ∎

Advantage: The hyperplane method provides a mathematical formula to verify the existence of communication-free partitioning in its solution space - the hyperplanes with different values of α and β.

Disadvantages:

1. When communication-free partitioning cannot be found, no alternative partitioning is provided.

2. The load may be unbalanced. Example 2.4.3 will show such an imbalance. Figure 2.4.1 also shows the imbalance that PE#3 and PE#21, for example, contain different number of elements.

Example 2.4.3

1. For $i_1 = 1$ *to* N
2. *For* $i_2 = 1$ *to* N
3. $B[i_1, i_2] := A[i_1, i_2] + A[i_1 + 3, i_2 + 4]$
4. *End For*
5. *End For*

Assume array size is 15×15.

1. Equation 2.5 is used for array partitioning.

$$PE\# = j = 4 \cdot i_1 - 3 \cdot i_2 \tag{2.5}$$

2. *PE[-1], for example, contains*
   $\Im_{p[-1]} = \{(2,3),(5,7),(8,11)\}$

3. *PE[0], for example, contains*
   $\Im_{p[0]} = \{(0,0),(3,4),(6,8),(9,12)\}$

4. *PE[56], for example, contains* $\Im_{p[56]} = \{(14,0)\}$

We can see that each PE contains a different number of array elements. The load in the PEs is unbalanced.

2.4.2 Domain Alignment Method

Li and Chen have proposed in [36] the domain alignment method which focuses on the relationships among arrays. Basically, permutations on indexes for each array are performed so that relevant array references can be aligned to the same position in a common virtual array. This minimizes the cost of data movement from array references.

In the domain alignment method, the permutations of array indexes are modeled as a graph problem with NP complexity. A heuristic algorithm is also presented in [36].

Advantage: The method models all arrays in a program into a single graph. It is different from other schemes that focus only on forall-loops.

Disadvantages:

1. The model is NP complete, a heuristic solution is required.

2. The model focuses on inter-array relationships. It does not address the relationship among intra-array partitions of each array.

2.4.3 Linear Skewing Scheme

Budnik and Kuck have proposed the linear skewing scheme [2] [7], which maps a two-dimensional array across the M memory banks of a vector computer by using equation (2.6).

$$m(i,j) = u \cdot i + v \cdot j \pmod{M} \tag{2.6}$$

where i and j are the array indexes, M is the number of memory banks, u and v are parameters, and $m(i,j)$ is the assigned memory bank number.

In [48], [35], and [55], certain properties on the values u,v and M are discussed to illustrate how conflict-free access to linear vectors can be achieved.

Definition 2.4.1 *A linear vector V with an access pattern $[x,y]$ is defined as the set of elements (i,j) such that* $V[x,y] = \{(i,j) | i = x \cdot k + x_0, j = y \cdot k + y_0; 0 \leq k < M\}$

where (x_0, y_0) is the starting array element and array element (i,j) should not exceed the size of the array.

Example 2.4.4 *A linear vector V with an access pattern $[1,2]$, starting at $(0,0)$ would contain the following set of array elements:*

$V[1,2] = \{(0,0),(1,2),(2,4),(3,6),(4,7),\cdots\}$

Theorem 2.4.1 *Let an array A be stored across M memory banks by the linear skewing scheme.* [3] *A linear vector V with an access pattern $[x,y]$ can be conflict-free accessed* [4] *if:*

$$\gcd(u \cdot x + v \cdot y, M) = 1 \tag{2.7}$$

Example 2.4.5 *Mapping a 5×5 matrix across 5 memory banks by the equation:* $m(i,j) = 1 \cdot i + 1 \cdot j \pmod{5}$

---

[2] The linear skewing scheme was originally designed for vector computers (single processor with multiple memory banks). However, we found that its mapping equation is also useful for the array distribution problem in distributed memory machines.

[3] Equation (2.6) is used to map the index of an array element to a memory bank.

[4] From equation (2.7), a prime number of M would have a better chance of being a coprime to the value of $(u \cdot x + v \cdot y)$.

Array indexes

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

Memory module number

$$\xrightarrow{1i+1j \pmod 5} \begin{bmatrix} 4 & 0 & 1 & 2 & 3 \\ 3 & 4 & 0 & 1 & 2 \\ 2 & 3 & 4 & 0 & 1 \\ 1 & 2 & 3 & 4 & 0 \\ 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

We can verify that the following three patterns can be accessed without conflict:[5]

1. Row-wise access (Linear vectors with pattern $[0,1]$) since $\gcd(1\cdot 0+1\cdot 1,5)=1$.
2. Column-wise access (Linear vectors with pattern $[1,0]$) since $\gcd(1\cdot 1+1\cdot 0,5)=1$.
3. Diagonals-wise access (Linear vectors with pattern $[1,1]$) since $\gcd(1\cdot 1+1\cdot 1,5)=1$.

Comment: The linear skewing scheme can be applied in distributing array elements with a certain pattern $[x,y]$ to independent memory banks. It may also result in the assignment of those elements to the same processor.

Table I describes various points between the linear skewing scheme and the hyperplane method.

[5]Theorem 2.4.1 is used to check if suitable parameters can be found for accessing certain vectors without conflict. $u=1$ and $v=1$ are chosen in the example. $M=5$, since there are 5 memory banks.

	Linear skewing scheme	Hyperplane method
Target Machines	Vector computers	Distributed memory computers
Target Data	A set of elements which forms a certain known linear pattern in an array	The arrays in a for-loop computation
Action	Allocate the set of array elements to different memory banks	Find a particular set of array elements and allocate them to the same PE
Objective	Access the set of data without conflict from memory banks	The particular set of array elements contained in a PE should incur no communication for the for-loop computation
Mapping equation	$m\# = u \cdot i + v \cdot j \pmod{M}$	$PE\# = \alpha_1 \cdot i_1 + \cdots + \alpha_n \cdot i_n$
Criterion	$\gcd(u \cdot i + v \cdot j, M) = 1$	A non-zero solution for a set of derived matrix equations

Table I.

2.4.4 The Index-set Labeling Method

D'Hollander [12] has used the integer programming method to decompose computation in a uniform dependence loop. The loop iterations are labeled so that they are divided into subsets with the same labels. This method decomposes loop indices (not arrays) for executing on shared memory machines. However, in the distributed memory machines, it should also allow the partitioning of arrays the same way that loop iterations are labeled.

Optimal independent subsets for uniformed dependence loop can be found by this method if index wrap-around is not considered. (Index wrap-around: given a

reference $A[i+6]$ to an array $A[0..9]$ (size 10), the reference will become $A[3]$ for $i = 6$, since $A[12]$ is out of the bounds of the array A, and is wrapped-around.)

2.4.5 Affine Method

Anderson [6], Lim [37] and Bau [51] use affine equations as the decomposition equations. The maximum independent subsets for multiple loops with array references in linear combination of loop indices can be found in the solution space of affine mappings.

The following theorem (2.4.2) is derived in [6], which is used to determine the parameters of the affine equations.

The mapping (partitioning) functions:

Definition 2.4.2 *For each index $\vec{a}$ of an m-dimensional array, the data decomposition of the array onto an n-dimensional processor array is a function: $\vec{d}(\vec{a}) = D\vec{a} + \vec{\delta}$*

where D is an $n \times m$ parameter matrix and $\vec{\delta}$ is a parameter vector.

■

Definition 2.4.3 *For each iteration $\vec{i}$ with a nested loop of depth l, the computation decomposition of the nested loop onto an n-dimensional processor array is a function: $\vec{c}(\vec{i}) = C\vec{i} + \vec{\gamma}$*

where C is an $n \times l$ parameter matrix and $\vec{\gamma}$ is a parameter vector.

■

Since only linear combinations of loop indices are considered, the array references can be modeled as affine functions: $\vec{f}(\vec{i}) = F\vec{i} + \vec{k}$

where F is a parameter matrix, $\vec{k}$ is a parameter vector, $\vec{i}$ represents the loop indices.

Theorem 2.4.2 *[6]*

Let the computation decomposition for nested loop j be $\vec{c_j}$ and the data decomposition for array x be $\vec{d_x}$. Let $\vec{f_{xj}}$ be an array reference function for array x in nested loop j. For all iterations $\vec{i}$, the elements of the array will be local to the processor that references those elements if and only if

$$D_x(\vec{f_{xj}}(\vec{i})) + \vec{\delta_x} = C_x(\vec{i}) + \vec{\gamma_j} \tag{2.8}$$

■

In [51], the elementary linear algebra is used to solve the problem of finding $C, \delta, D,$ and γ. It demonstrates an integrated and simpler approach to get the independent communication-free subsets than the methods in [6] and [37].

2.5 Related Works on Distributed-Array Compilation

A naive implementation for supporting the distributed arrays can be highly inefficient. For example,

1. ForAll i=0 to 4000, j= 0 to 4000
2. $\quad A(i+3, 2*j) = i+j$
3. end for

Without using the Compute_Set, the naive implementation would use the guarded statement which results in inefficiency due to the heavy run-time ownership testings in each processor:

1. ForAll i=0 to 4000, j=0 to 4000
2. $\quad$ if (Owner($A(i+3, 2*j)$) == me) $A(i+3, 2*j) = i+j$
3. end for

However, if the Compute_Set can be derived, An advanced compiler would generate execution codes without the heavy run-time ownership testings in each processor:

1. ForAll i in Compute_Set(my_PE_id)
2. $\quad A(i+3, 2*j) = i+j$
3. end for

The challenge: How to derive the Compute_Set based on data distribution directives.

The efficiency of message-passing parallel computers is significantly determined by the number of messages. The implementation of data-parallel array statements in

recent language systems all generate at most one message to each remote processor for the referenced array.

As an example, the message construction for above example could be implemented as:

1. ForAll i=0 to 40, j=0 to 40
2. if (owner($B(i+j, i+5)$ == me)
3. append $B(i+j, i+5)$ to the message for owner($A(i+3, 2*j)$);
4. end for

The above code also involves heavy run-time testing.

The challenge: How to derive the communication sets based on data distribution directives.

2.5.1 Hermit-Normal-Form Method

The set of local enumeration forms a lattice in the r-axis and c-axis space for a general alignment and $cyclic(n)$ data distribution, which can be seen, for example, in figure 5.3. Equation (5.36) that expresses the relations of variables c, r, and i_g can be called a implicit lattice equation. The Hermit-form method is developed to convert an implicit lattice equation into an explicit lattice equation as of the form $\vec{x} = x_0 + M \cdot \vec{k}, \vec{k} \in Z^n$, which can then be used to enumerate the local set.

In the following, the Hermit-form method for local set enumeration is described.

First, equation (5.36) can be reorganized into the following matrix form:

$$\begin{bmatrix} 1 & -n_p \cdot m & a_{co} \end{bmatrix} \cdot \begin{bmatrix} c \\ r \\ i_g \end{bmatrix} = b_{co} - m \cdot p \tag{2.9}$$

Based on that given any $n \times m$ integer matrix F, there are an $n \times m$ lower triangular matrix H and an $m \times m$ unimodular (determinant $= \pm 1$) matrix Q, such that $H = F \cdot Q$. Since $|Q| = \pm 1$, Q^{-1} is also an integer matrix such that $F = H \cdot Q^{-1}$.

Let $F = \begin{bmatrix} 1 & n_p \cdot m & -a_{co} \end{bmatrix}$, $\vec{x} = \begin{bmatrix} c \\ r \\ i_g \end{bmatrix}$, and $f_0 = b_{co} - m \cdot p$, equation (5.37) is rewritten as

$$F \cdot \vec{x} = f_0 \tag{2.10}$$

Applying the Hermit-form decomposition to F, we get $H = F \cdot Q$ with

$$H = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \quad Q = \begin{bmatrix} 1 & -n_p \cdot m & a_{co} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, equation (5.37) can be rewritten as:

$$F \cdot \vec{x} = F \cdot Q \cdot Q^{-1} \cdot \vec{x} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \cdot \vec{v} = f_0$$

where $Q^{-1}\vec{x} = \vec{v} = \begin{bmatrix} f_0 \\ v_1 \\ v_2 \end{bmatrix}$ with $v_1, v_2 \in Z$.

Hence

$$\begin{aligned} \vec{x} &= Q \cdot \vec{v} \\ &= \begin{bmatrix} f_0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -n_p \cdot m & a_{co} \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \\ &= \vec{x_0} + F' \cdot \vec{v'} \end{aligned} \tag{2.11}$$

Applying Hermit-form decomposition to F', we have $H' = F' \cdot Q'$ with

$$H' = \begin{bmatrix} -g & 0 \\ \mu & \frac{a_{co}}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix}, Q' = \begin{bmatrix} \mu & \frac{a_{co}}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix} \tag{2.12}$$

where $g = gcd(n_p \cdot m, a_{co})$ and $g = n_p \cdot m \cdot \mu - a_{co} \cdot \omega$ (the Bezout identity).

Equation (2.11) becomes:

$$\vec{x} = \vec{x_0} + F'\vec{v'} = \vec{x_0} + H' \cdot Q'^{-1}\vec{v'} = \vec{x_0} + H' \cdot \vec{u} \tag{2.13}$$

where $\vec{u} = Q'^{-1}\vec{v'} \in Z^2$.

Hence, we have the explicit lattice equation (2.11) for the implicit equation (5.36) or equation (5.37).

Using bounds in (5.33) and equation (2.13), the following constraints are derived:

$$\begin{bmatrix} g & 0 \\ -g & 0 \\ -\omega & -\frac{n_p \cdot m}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \leq \begin{bmatrix} f_0 \\ f_0 + m - 1 \\ 0 \\ n_g - 1 \end{bmatrix} \tag{2.14}$$

A nested loop for enumerating the set $\{i_g\}$ can then be constructed based on (2.14). It results to the same nested loop as loop 5.1.3.

The above derivation can also begin with the following equation (2.15), where $\vec{x}$ is in r, c, i_g order.

$$\begin{bmatrix} 1 & a_{co} & -n_p \cdot m \end{bmatrix} \cdot \begin{bmatrix} r \\ c \\ i_g \end{bmatrix} = b_{co} - m \cdot p \tag{2.15}$$

By using the similar derivation, we will also obtain equation (2.14) and result in the same nested loop 5.1.3.

In Hermit-form method[8], a compressed storage scheme is also presented. This scheme use rectangular storage space in $\vec{u'}$ indices, where

$$\vec{u'} = \begin{bmatrix} -1 & 0 \\ \frac{\mu \cdot g}{a_{co}} & 1 \end{bmatrix} \cdot \vec{u}$$

, where $\frac{\mu \cdot g}{a_{co}}$ is a fraction number. It also results to the same rowwise storage as in figure 5.4 for example 5.1.2.

2.5.2 Finite State Method

The FSM method is based on that the enumeration is always repeating by some pattern. The pattern can be modeled by a state machine (at most m states) that describes the steps (Δr and Δc) to next row number r and column number c.

By enumerating the first $m+1$ sequences of local array set $\{i_a\}$, and using $i_{temp} = r * m + c$, a state machine ($T1$) is derived to specify the difference (Δi_{temp}) for two successive local array enumeration (i_a, i_a').

By enumerating the first $m+1$ sequences of local iteration set $\{i_g\}$, another state machine ($T2$) is derived to specify the difference (Δi_{temp}) of referred array element i_a ($i_a = \alpha \cdot i_g + \beta$) for two successive local iteration enumeration (i_g, i_g').

The FSM method uses the local storage scheme ($i_{local} = k$) where local array elements ($\{(i_a)_k, k = 0 \ldots$ local_element_number-1 $\}$) are stored in local memory consecutively. By using both state machine tables ($T1$, and $T2$), a state machine ($T3$) can then be derived to specify the step (Δk) to next referred array address (i_a) by next local iteration enumeration (i_g).

Based on the state machine table ($T3$), it results to the following nested loop:

Code 2.5.1

```
1.  for p = 0 to n_p − 1 do
2.      T3 = FSM_for_(p)
3.      l = FSM_table_length(p)
4.      i_local = starting_k(p)
5.      last = ending_k(p)
6.      j = 0
7.      while (i_local ≤ last) do
8.          A(i_local) = ....
9.          i_local = i_local + T3[j]
10.         j = (j + 1) (mod l)
11.     enddo
12. enddo
```

where $starting\_k(p)$ and $ending\_k(p)$ can be derived by equation (5.33) and (5.36).

It should be noted that the enumeration sequence ($\{i_g\}$) generated by FSM is identical to the columnwise method. The FSM storage scheme is also shown in figure 2.1 for example 5.1.2.

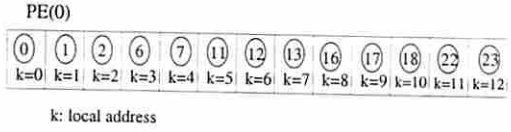


Figure 2.1: FSM Storage scheme.

Chapter 3

Algebra Foundation of Our Research

In this chapter, we discuss all the lattice algebra related in this research which includes the concepts of the unimodular matrix, of the Hermit Normal Forms, of the Smith Normal Forms, and of the periodic skewing equations.

3.1 Unimodular Matrix and Hermit Normal Form

A unimodular matrix is a square, integer matrix where the absolute value of the determinant is 1. For example:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}$$

The following theorem is needed in our strategy in automatic array compilation.

Theorem 3.1.1

Let $a_1, a_2, \ldots, a_n$ be integers, and let $g_n = gcd(a_1, a_2, \ldots, a_n$ be their greatest common divisor. Then there is a matrix D with first row, (or first column) and determinant g.

■

Base on above theorem, in the case of $g_n = 1$, an unimodular matrix can be constructed by given values of first row, or given values of first column.

The following theorem is needed in our strategy in both automatic array compilation and the automatic array partitioning.

Theorem 3.1.2 *(Hermit Normal Form)*

Every matrix A of Z^n is right equivalent ($A = H \cdot T$ and T is unimodular) to a lower triangular matrix H with each diagonal element (unique positive integer) being the maximum entry for each row.

Let A be a $m \times (m+t)$ matrix. Then, its Hermit Normal Form can be taken as $[HO]$, where O is a $(m \times t)$ block of zeros and the $m \times m$ matrix H is in the form as described above.

■

When perform transformation for nested loop in both distributed array compilation, the basic idea in the calculation of exact loop bounds will be based on the Hermit Normal Form of the transformation matrix.

3.2 Lattice and Smith Normal Form

The Smith Form decomposition of a matrix has been useful in many aspects of m-D signal processing including the design of multidimensional multirate filter banks and the design of m-D DFT and FFT algorithms. Furthermore, the Smith Form decomposition has been used in other areas of electrical engineering including linear control theory.

The following theorem 2.4.2 and theorem 3.3.1 with the following definitions and theorems will also be used in developing our scheme in the automatic array partitioning and the distributed array compilation.

Definition 3.2.1 *(Lattice)*

Let $\vec{v_1}, \vec{v_2}, \cdots, \vec{v_n}$ be a set of integer vectors ($\vec{v_i} \in Z^m$), the set of all points (linear combination of the vectors) $\vec{x} = k_1\vec{v_1} + k_2\vec{v_2} + \cdots + k_n\vec{v_n}$, where $k_i \in Z$, is called the lattice generated by the set of vectors ($\{\vec{v_1}, \vec{v_2}, \cdots, \vec{v_n}\}$).

Notation Lattice(A): *Let A be an $m \times n$ matrix with $\vec{v_i}$ as its i-th column, for $1 \leq i \leq n$, the lattice generated by the set of $\vec{v_i}$s (the columns of A) can be denoted as $Lattice(A) = \{\vec{x} | \vec{x} = A\vec{k}$ and $\vec{k} \in Z^n\}$.*

Theorem 3.2.1

Let A be an $m \times n$ integer matrix, then $Lattice(AU) = Lattice(A)$, if U is a $n \times n$ unimodular matrix.

■

Theorem 3.2.2 *(Smith normal form)[39]*

Let A, B be integral matrices. We say that B is equivalent to A if $B = U \cdot A \cdot V$ for some unimodular matrices U and V.

Every integer matrix A is equivalent to a diagonal matrix (its Smith normal form)

$$S = S(A) = diag(s_1, s_2, \ldots, s_r, 0, 0, \ldots, 0)$$

where r is the rank of A, $s_1, s_2, \ldots, s_r$ are nonzero elements and $s_i | s_{i+1}, 1 \leq i \leq r-1$. ($a|b$ means that $\exists k \in Z \quad a = b \cdot k$)

■

3.3 Periodic Skewing Equation

The theory of two-dimensional periodic skewing scheme was initialized by Kuck[34] and extended by Shapiro[48], and Wijshoff and van Leeuwen[55]. We will discuss the foundation of periodic skewing theorem here.

We will first define the periodic skewing equation and show geometrically how partitioning can for arrays by periodic skewing mapping equations.

Definition 3.3.1 *(Periodic Skewing Mapping)*

A periodic skewing mapping is a function $\vec{d}(\vec{a})$ which maps an array element onto an n-dimensional processor array where $\vec{a}$ is the index of an array element by using following periodic skewing equation.

$$\vec{d}(\vec{a}) = (D \cdot \vec{a}) \ mod \ \vec{p}_A \tag{3.1}$$

where D is an $n \times m$ linear transformation matrix, $\vec{p}_A$ is a constant vector associated with each array A, and mod is defined as $l_i = q_i \pmod{p_i}$, for $i = 1, \cdots, n$, if $\vec{l}, \vec{q}$, and $\vec{p}$ are $(n \times 1)$ vectors.

The earlier works that adopts periodic skewing equation [35] [55] [7] [48] have presented the following theorem:

Definition 3.3.2 *(Table and Periodic) ([55])*

Let G be a finite set (e.g., a finite Z-module) with $|G| = M$. A **table** *t (for G) is any bijective mapping from G into $\{0,...,M\text{-}1\}$.*

Let $s : Z^d \rightarrow \{0,..., M\text{-}1\}$ be a skewing scheme using M memory banks. The scheme s is called **periodic** *if and only if there exists a d-dimensional lattice L^d, and a (surjective) homomorphism $\alpha : Z^d \rightarrow Z^d/L^d$ with $Ker(\alpha) = L^d$, and a table for Z^d/L^d such that $s = t$.*

Theorem 3.3.1 *([55])*

Let s be a periodic skewing scheme and L^d its underlying lattice. There exists a homomorphism $\alpha : Z^d \rightarrow B_{L^d}$ and a table t of B_{L^d} such that $s = t \cdot \alpha$, and α is given by an expression of the type $\alpha(i_1, \cdots, i_d) = (L_1(\bar{i} \bmod s_1), \cdots, L_1(\bar{i} \bmod s_d))$ where $L_k(\bar{i}) \equiv \lambda_{k,1} i_1 + \cdots + \lambda_{k,d} i_d$ is an integer linear form for $1 \leq k \leq d$, and B_{L^d} is the set of points $\{(x_1, \cdots, x_d) | 0 \leq x_1 \leq s_1 - 1, 0 \leq x_2 \leq s_2 - 1, \cdots 0 \leq x_d \leq s_d - 1\}$.

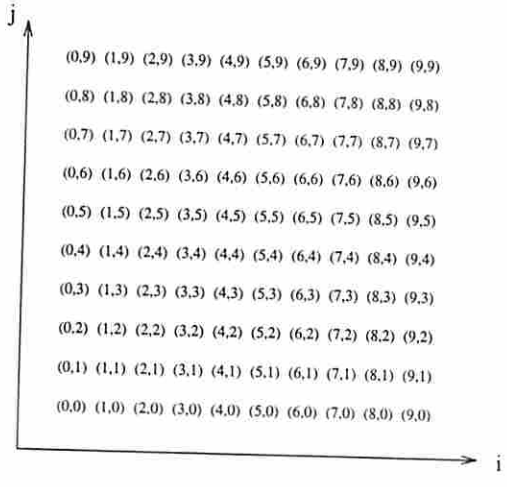
The homomorphism α in the theorem 3.3.1 is actually the periodic skewing mapping of the definition 3.3.1. Incidentally, it should be noted that the index-set labeling method and the periodic skewing scheme can be related together by theorem 3.3.1 with the periodic skewing mapping:

1. For a given lattice L, there exists a periodic skewing mapping m with L as its underlying lattice.

   The index-set labeling method [12] has presented an algorithm that derives the m from a given lattice L.

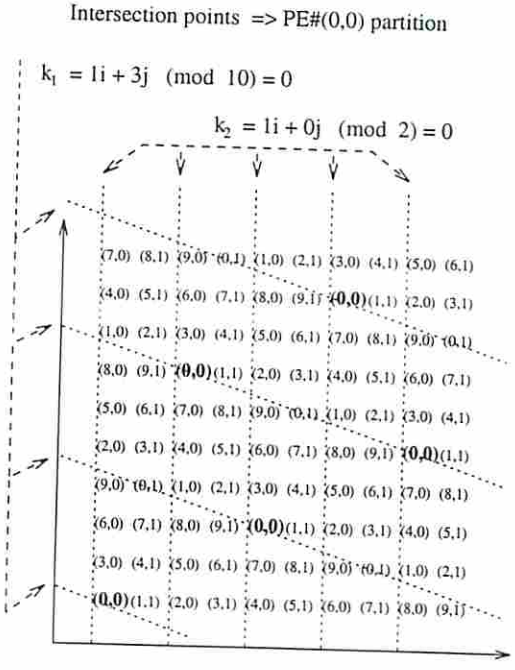
2. Conversely, there exists an underlying L for a given periodic skewing mapping m.

   The periodic skewing scheme [35] [55] [7] is an application where a given m generates the lattice L.



The indexes of a 10 × 10 array.

Figure 3.3.1



PE# map for a 10 × 10 array.

Figure 3.3.2

The following example will describe how the periodic skewing equation will allow partitioning of the data and will show that the data in one partition form a lattice graph.

Example 3.3.1 *Mapping a* 10×10 *array to PEs by the following periodic skewing equation.*

$$PE\#(k_1, k_2) = \begin{cases} k_1 = 1 \cdot i + 3 \cdot j & (\text{mod } 10) \\ k_2 = 1 \cdot i + 0 \cdot j & (\text{mod } 2) \end{cases} \tag{3.2}$$

■

The values of k_1 and k_2 can be directly read from figure 3.3.2. Figure 3.3.2 shows the PE# for each element in a 10×10 array whose index space is shown in figure 3.3.1. The PE# is directly derived by equation (3.2).

1. The two modulos in equation (3.2), 10 and 2, imply that the size of the virtual processor array is 10×2.

2. PE#(0,0), for example, contains the set of elements:

   $\{(0,0),(2,6),(4,2),(6,8),(8,4)\}$ that are located at the intersections of the lines described by the following two equations.

$$\begin{aligned} k_1 &= 1 \cdot i + 3 \cdot j \pmod{10} &= 0 \\ k_2 &= 1 \cdot i + 0 \cdot j \pmod{2} &= 0 \end{aligned} \tag{3.3}$$

Chapter 4

Our Array Partitioning Based on Program Analysis

The performance of parallel execution in most message-passing parallel machines is significantly affected by remote data accesses since the communication latency can be several orders of magnitude higher than the computation costs. In scientific applications, arrays are usually the largest data structures involved in the most time consuming computations. Therefore, arrays must be appropriately partitioned and distributed to processors for efficient parallel operations.

Some language systems, like HPF [22] or Fortran D [31], provide directives for programmers to specify how data are distributed. In contrast, many research projects focus on automatic array partitioning. Li and Chen [36] have presented a method which aligns the indexes of arrays to reduce memory traffic. Basically, an alignment function (permutation) is used to permute the indexes in arrays. The analysis is based on a graph algorithm which incurs NP-complex costs. Ramanujam and Sadayappan [40] have used hyperplanes to partition arrays and have derived a sufficient condition for communication-free partitioning. For some cases, Mace [38] has proved that finding an optimal data storage pattern was NP-complete in its solution-space graph. D'Hollander [12] has used the integer programming method to partition the computation into independent subsets for a Do-loop with constant dependence. Affine methods [6, 37, 51] use affine equations to partition the computation and the arrays in a Do-loop at the same time. No method, however, has derived one equation to describe all the possible communication-free partitionings for arrays in a DoAll loop. Neither has it been possible to derive the partitioning

equations for one block-communication when communication-free is not possible. In this research, we will introduce two novel analytical methods of array partitioning:

(1) Communication-free array partitioning: A general method which will provide communication-free partitioning is derived for arrays in a DoAll loop. The derivation is based on the Smith Normal Form decomposition of the matrix which characterizes the array references in a DoAll loop.

(2) One block-communication partitioning: When communication-free partitioning is not possible, we derive the partitioning equations which locate all remote data to one processor. Thus, at most one block-communication is required for each processor to get the remote data during computation.

We will first recall some basic concepts of (1) loop partitioning for parallel execution in a Do-loop and (2) communication-reduction partitioning for arrays in a forall loop.

4.1 Loop Partitioning for Parallel Execution

The following sample Do-loop is used to explain the basic concepts:

1. Do I,J
2. $\quad A[I, J] = A[I+2, J+8] + A[I+4, J+10]$
3. EndDo

The loop index (I,J) can be represented as a vector $\vec{i} = [I, J]^T$, and the array references can also be represented in vector form: $\vec{r}_1 = [I, J]^T$, $\vec{r}_2 = [I+2, J+8]^T$, and $\vec{r}_3 = [I+4, J+10]^T$.

Parallel Loop partitioning consists in dividing the loop-index set $\{\vec{i}\}$ into independent subsets such that the executions of iterations are totally independent among the subsets.

Usually a mapping equation is used to describe how the loop indices are partitioned. The mapping equation ($\vec{p} = m(\vec{i})$) maps the loop index $\vec{i}$ to the processor index $\vec{p}$.

It can be seen that there are dependencies among loop indices (loop iterations). The dependency can be described by the dependence vectors: $\vec{d}_1 = \vec{r}_2 - \vec{r}_1 = [2, 8]^T$, and $\vec{d}_2 = \vec{r}_3 - \vec{r}_1 = [4, 10]^T$. The do-loop is characterized by the dependence matrix D, which consists of the dependence vectors as its column vectors ($D = [\vec{d}_1 \ \ \vec{d}_2]$).

Two iterations $\vec{i_1}$ and $\vec{i_2}$ are said to have dependency [12] if there exists a linear combination of dependence vectors such that $\vec{i_2} = \vec{i_1} + D \cdot \vec{\lambda}$. $\vec{\lambda} \in Z^2$.

4.2 Array Partitioning for Communication Reduction

Although all iterations in a DoAll loop can be executed in parallel, if the arrays are not properly distributed across the processors, additional communication may be introduced. In the example that follows, we assume that array B and array A have the same distribution. According to the owner-compute rule, an iteration (I, J) is executed on the processor which owns $B[I, J]$ and which also owns $A[I, J]$. If this processor does not also own $A[I+2, J+8]$, $A[I+4, J+10]$, or $A[I+2, J+2]$, remote accesses are then required.

Example 4.2.1

1. *DoAll I,J*
2. $\quad B[I, J] = A[I, J] + A[I + 2, J + 8] + A[I + 4, J + 10] + A[I + 2, J + 2]$
3. *EndDo*

The loop index (I,J) can be represented as a vector $\vec{i} = [I, J]^T$, and the four array references to the array A can also be represented in vector forms: $\vec{r_1} = [I, J]^T$, $\vec{r_2} = [I+2, J+8]^T$, $\vec{r_3} = [I+4, J+10]^T$, $\vec{r_4} = [I+2, J+2]^T$. The relationship between the array references can be described by the distance vectors. Choose one array reference as a reference ($\vec{r_1}$, as example), we then have the following three distance vectors: $\vec{d_1} = \vec{r_2} - \vec{r_1} = [2, 8]^T$, $\vec{d_2} = \vec{r_3} - \vec{r_1} = [4, 10]^T$ and $\vec{d_3} = \vec{r_4} - \vec{r_1} = [2, 2]^T$.

Thus, the PE $(\vec{p})$ that performs iteration $\vec{i}$ will refer $A[\vec{i}]$, $A[\vec{i} + \vec{d_1}]$, $A[\vec{i} + \vec{d_2}]$, and $A[\vec{i} + \vec{d_3}]$. To avoid communication, the four array elements $(G_{(i)})$ must be allocated in PE $\vec{p}$. During iteration $\vec{j}$ $(\vec{j} = \vec{i} + \vec{d_1})$ performed by PE $\vec{p'}$, $A[\vec{i} + \vec{d_1}]$, $A[\vec{i} + 2 * \vec{d_1}]$, $A[\vec{i} + \vec{d_1} + \vec{d_2}]$, and $A[\vec{i} + \vec{d_1} + \vec{d_3}]$ are referenced. They $(G_{(i+d_1)})$ all should be allocated in PE $\vec{p'}$ to avoid communication. Since $A[\vec{i} + \vec{d_1}]$ need to be in both PE $\vec{p}$ and PE $\vec{p'}$, all the array references $(G_{(i)} \cup G_{(i+d_1)})$ during iterations $\vec{i}$ and iteration $\vec{j}$ are then required to be in the same processor $(\vec{p} = \vec{p'})$ to avoid communication. During iteration $\vec{i} + \vec{d_2}$, the four reference array elements $(G_{(i+d_2)})$ are also connected

to the group $(G_{(i)} \cup G_{(i+d_1)})$ by $A[\vec{i} + \vec{d}_2]$. The group of connected array elements becomes $G_{(i)} \cup G_{(i+d_1)} \cup G_{(i+d_2)}$. This group of connected array elements can be expanded by any array element in the group as a connected element such that a set of array elements (dependence set) is constructed:

$$\{A[\vec{i'}] \mid \vec{i'} = \vec{i} + \lambda_1\vec{d}_1 + \lambda_2\vec{d}_2 + \lambda_3\vec{d}_3, \quad \lambda_1, \lambda_2, \textit{ and } \lambda_3 \in Z\} \tag{4.1}$$

The references to array A can be characterized by a distance matrix D which consists of the distance vectors of array A as its column vectors ($D = [\vec{d}_1 \ \ \vec{d}_2 \ \ \vec{d}_3]$). Then according to the dependence set, two array A elements with indices $\vec{i}_1$ and $\vec{i}_2$ must be in the same processor to avoid communication if $\exists\vec{\lambda} \in Z^n$, $\vec{i}_2 = \vec{i}_1 + D \cdot \vec{\lambda}$ ($n = 3$ in this example).

Communication-free array partitioning aims at dividing the array-index set $\{\vec{i}\}$ of an array into as many independent subsets as possible such that array elements in other subsets are not required during computation. The array partitioning is achieved by providing a mapping equation which maps the array index $\vec{i}$ to the processor index $\vec{p}$ and results into a number of subsets identified by $\vec{p}$.

In the next two sections, we will introduce our two techniques for static array partitioning. We assume that (1) There is only one copy of an array element among all processors; (2) Both array index and processor index are starting with 0; (3) All updates to an array element are performed by the processor which owns the element ("owner-compute" rule).

4.3 Communication-Free Array Partitioning

The method developed in this section is aimed at deriving a general partitioning equation which describes all possible communication-free array partitionings for a DoAll loop.

4.3.1 Derivation of Communication-Free Equation

The references to an array in a DoAll loop can be characterized by the distance vectors described in section 5.5. We assume here that the distance vectors $\vec{d}$ are constant vectors, and hence the distance matrix D is a constant matrix. The set of dependent array elements in $\{A[\vec{i}] \mid \vec{i} = \vec{i'} + D \cdot \vec{\lambda},\ \vec{\lambda} \in Z^n\}$ (See equation (4.1)) forms a lattice $\vec{L}$. For a given lattice L, there exists a periodic skewing equation with L as its underlying lattice ([55], theorem 3.8). Thus, the following general periodic skewing equation will be used as our partitioning equation where different values of matrix M and $\vec{s}$ in equation (4.2) represent different forms of lattices [55]. Our goal is then to determine the matrix M and $\vec{s}$ according to the dependence equation $\vec{i} = \vec{i'} + D \cdot \vec{\lambda}$.

$$\vec{p} = m(\vec{i}) = M \cdot \vec{i} \pmod{\vec{s}} \tag{4.2}$$

where $\vec{p}$ is the PE index, $\vec{i}$ is the array index. The operator *mod* denotes the modulo operation: $\vec{a}$ and $\vec{b} \in Z^n$, $\vec{a}\ mod\ \vec{b} \equiv \vec{r}$ where $a_i \equiv r_i\ mod\ b_i$ for $i = 1 \ldots n$.

The above periodic skewing equation can also be written as

$$\vec{p} = M \cdot \vec{i} + S \cdot \vec{t},\ \ for\ \ \vec{t} \in Z^n \tag{4.3}$$

where S is a diagonal matrix with the diagonal elements $s_{ii} = s_i$, and s_i are the elements of the vector $\vec{s}$ in (4.2).

Extracting $\vec{i}$ from (4.3), we obtain [55]:

$$\vec{i} = N \cdot \vec{p} + H \cdot \vec{k},\ \ \vec{k} \in Z^n \tag{4.4}$$

There exist the following relationships among the matrices M, S, N, and H, such that equations (4.3) and (4.4) can be both satisfied.

$$M = T \cdot U,\quad H = U^{-1} \cdot S,\quad M \cdot N \equiv I \pmod{\vec{s}} \tag{4.5}$$

where U is a unimodular matrix (determinant = +1, or -1), I is the unit matrix, and T is a parameter matrix to be determined.

In order to have communication-free partitioning, all the dependent array elements must be located in the same processor. As described in section 5.5, the set of all dependent array elements can be modeled as those whose indices obey the following relationship:

$$\vec{i} = \vec{i'} + D \cdot \vec{\lambda} \tag{4.6}$$

where $\vec{\lambda}$ is any integer vector, and $\vec{i'}$ is any array index in the set of the dependent array elements.

According to the theorem of the Smith Normal Form (Theorem 5.6.1), the distance matrix D can be decomposed as $D = U^{-1} \cdot S \cdot V^{-1}$;

Thus, equation (4.6) becomes

$$\vec{i} = \vec{i'} + (U^{-1} \cdot S \cdot V^{-1}) \cdot \vec{\lambda} \; = \; \vec{i'} + (U^{-1} \cdot S) \cdot \vec{k} \tag{4.7}$$

where $\vec{k} = V^{-1} \cdot \vec{\lambda}$. Since $\vec{\lambda} \in Z^n$ and V^{-1} is a unimodular matrix, $\vec{k} \in Z^n$

According to equations (4.3) and (4.5), the corresponding partitioning equation for (4.6) has the form:

$$\vec{p} = m(\vec{i}) = T \cdot U \cdot \vec{i} + S \cdot \vec{t} \tag{4.8}$$

Substituting $\vec{i}$ by the expression given in (4.7), the processor index becomes:

$$\vec{p} = T \cdot U \cdot (\vec{i'} + (U^{-1} \cdot S) \cdot \vec{k}) + S \cdot \vec{t} \; = \; (T \cdot U \cdot \vec{i'}) + (T \cdot S \cdot \vec{k} + S \cdot \vec{t})$$

In order to have communication-free partitioning, all the dependent elements must be located in one processor, which means that $\vec{p}$ needs to be a constant vector. Therefore, $\vec{p}$ is independent of $\vec{k}$ and $\vec{t}$, which requires $T \cdot S \cdot \vec{k} + S \cdot \vec{t} = \vec{0}$, or

$$T = S \cdot X \cdot S^{-1}, \quad X \in Z^{n \times n} \tag{4.9}$$

Let us return now to equation (4.5), and more particular to the property $M \cdot N \equiv I \pmod{\vec{s}}$:

$M \cdot N \equiv I \pmod{\vec{s}}$ is equivalent to $[M\ S] \cdot \begin{bmatrix} N \\ K \end{bmatrix} = I$, where N and K can be any integer matrices. There exist such N and K, if the Smith Normal Form of $[M\ S]$ is $[I\ O]$, where O is a zero matrix ([39], p37).

When, in general, SNF of $[M\ S] = [W\ O]$, and $W = diag(w_1, w_2, \ldots, w_j, 0, 0, \ldots, 0)$ where j is the rank of W, the partitioning equation (4.2) allows the partitioning of an array into $\frac{s_1 \times s_2 \times \cdots \times s_r}{w_1 \times w_2 \times \cdots \times w_j}$ subgroups. In the case of $[M\ S]$ is $[I\ O]$, the array is then partitioned into $s_1 \times s_2 \times \cdots \times s_r$ subgroups, which is maximum.

We now have the following fundamental property, which can be utilized by the compiler to derive a communication-free array partitioning.

Property 4.3.1

Given a distance matrix D for an array A in a DoAll loop, the communication-free partitioning of the array A can be described by the equation $\vec{p} = m(\vec{i}) = (T \cdot U \cdot \vec{i}) \pmod{\vec{s}}$, where $T = S \cdot X \cdot S^{-1}$, and X is any integer matrix, U and S are obtained from the Smith Normal Form decomposition of D ($D = U^{-1} \cdot S \cdot V^{-1}$), and $\vec{s}$ is formed by collecting the diagonal elements s_{ii} of the matrix S. Moreover, if the SNF of $[M\ S] = [I\ O]$ (where $M = T \cdot U$), the number of independent subgroups is maximum. ∎

When $X = I$ (Unit matrix), the partitioning equation becomes $\vec{p} = U \cdot \vec{i} \pmod{\vec{s}}$ which always divides the array into maximum subgroups.

4.3.2 An Example

The array A in example 4.2.1 is used as an example to demonstrate our communication-free partitioning approach:

The distance matrix of array A in example 4.2.1 is $D = \begin{bmatrix} 2 & 4 & 2 \\ 8 & 10 & 2 \end{bmatrix}$, and its Smith Normal Form decomposition is:

$$S = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 6 & 0 \end{bmatrix} = U \cdot D \cdot V = \begin{bmatrix} 1 & 0 \\ -4 & 1 \end{bmatrix} \cdot D \cdot \begin{bmatrix} 1 & 2 & 1 \\ 0 & -1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

According to property 4.3.1, we have

$$\vec{s} = \begin{bmatrix} 2 \\ 6 \end{bmatrix}$$

$$T = S \cdot X \cdot S^{-1} = \begin{bmatrix} 2 & 0 \\ 0 & 6 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{6} \end{bmatrix} = \begin{bmatrix} x_{11} & \frac{1}{3}x_{12} \\ 3x_{21} & x_{22} \end{bmatrix} \tag{4.10}$$

where x_{11}, x_{12}, x_{21}, and $x_{22} \in Z$

Thus, the communication-free partitioning equation can be described by $\vec{p} = m(\vec{i}) = (T \cdot \begin{bmatrix} 1 & 0 \\ -4 & 1 \end{bmatrix} \cdot \vec{i}) \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$, where T is an integer matrix described by equation (4.10).

For a different value of matrix T, we would have a different equation for the communication-free partitioning as, for example, shown below:

If we choose $T = \begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix}$, we would have the following communication-free partitioning equation:

$$\vec{p} = m(\vec{i}) = (T \cdot U \cdot \vec{i}) \pmod{\vec{s}} = \begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \vec{i} \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}} \tag{4.11}$$

If $T = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$, we would have the following communication-free partitioning equation:

$$\vec{p} = \begin{bmatrix} -7 & 2 \\ -4 & 1 \end{bmatrix} \vec{i} \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$

If $T = \begin{bmatrix} 0 & 1 \\ 3 & 4 \end{bmatrix}$, we would have the following communication-free partitioning equation:

$$\vec{p} = \begin{bmatrix} -4 & 1 \\ -13 & 4 \end{bmatrix} \vec{i} \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$

Note that the above three partitioning equations satisfy that the SNF of $[M\ S] = [I\ O]$, thus all of them result in a maximum number of communication-free subgroups.

Verification of the Communication-Free property:

The property of communication-free partitioning can be verified as follows:

The references to the array A are $\vec{r}_1 = [I, J]^T$, $\vec{r}_2 = [I+2, J+8]^T$, $\vec{r}_3 = [I+4, J+10]^T$, and $\vec{r}_4 = [I+2, J+2]^T$.

According to the partitioning equation, equation (4.11) as an example, the three elements would be located in processors

$$\vec{p}_1 = \begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$

$$\begin{aligned} \vec{p}_2 &= \begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} I+2 \\ J+8 \end{bmatrix} \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}} \\ &= \left(\begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} 2 \\ 6 \end{bmatrix} \right) \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}} \end{aligned}$$

$$\vec{p}_3 = \begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} I+4 \\ J+10 \end{bmatrix} \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$
$$= \left(\begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} -2 \\ 6 \end{bmatrix}\right) \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$

$$\vec{p}_4 = \begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} I+2 \\ J+2 \end{bmatrix} \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$
$$= \left(\begin{bmatrix} -3 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} + \begin{bmatrix} -4 \\ 0 \end{bmatrix}\right) \pmod{\begin{bmatrix} 2 \\ 6 \end{bmatrix}}$$

It can be seen that $\vec{p}_1 = \vec{p}_2 = \vec{p}_3 = \vec{p}_4$, which verifies that the three array references are located in the same processor. Thus, no communication is required during the computation.

4.4 Block-Communication Partitioning

In the previous section, we derived one equation to describe all the possible communication free partitionings for constant distance vectors where the set of the dependent array elements can be expressed in a lattice. When the set of dependent array elements (lattice) is equal to the whole array, the problem of communication-free partitioning becomes trivial as that the whole array should be allocated in one processor to avoid communication during execution. (This would obviously bring parallelism down to nothing.)

When a distance vector includes loop indices as its elements, the set of the dependent array elements usually covers the whole array. It is because a new value of distance vector is created for a new loop iteration (loop indices). In this section, we will derive the partitioning equation for the case of non-constant distance vectors. The equation derived can evenly divide an array into subgroups where the set of the remote data needed for a subgroup will be located in at most one other group, such that at most one block-communication is needed during computation.

4.4.1 Derivation of Block-Communication Equation

The periodic skewing equation ($\vec{p} = m(\vec{i}) = M \cdot \vec{i} \pmod{\vec{s}}$) is also used as the partitioning equation. The approach is to have the processor mapping ($\vec{p} = m(\vec{i})$) of all the remote data to have the same value. such that all remote data are located in one processor. The partitioning equation derived will evenly divide an array into subgroups due to the property of modular operation and the SNF of $[M\ S] = [I\ O]$ as described in previous section.

The derivation is illustrated on the following DoAll loop:

1. DoAll I, J
2. $\quad B[I, J] = A[a_{11}I + a_{12}J + a_{10}, a_{21}I + a_{22}J + a_{20}]$
3. $\quad\quad + A[b_{11}I + b_{12}J + b_{10}, b_{21}I + b_{22}J + b_{20}]$
4. EndDo

where all a_{jk} and b_{jk} are constants.

The references to arrays are assumed to be linear combinations of the loop indices. They can be modeled as:

$$\vec{r}(\vec{i}) = \begin{bmatrix} a_{11}I + a_{12}J + a_{10} \\ a_{21}I + a_{22}J + a_{20} \end{bmatrix} = F\vec{i} + \vec{f}$$

where the parameter matrix $F = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, and the parameter vector $\vec{f} = \begin{bmatrix} a_{10} \\ a_{20} \end{bmatrix}$

The DoAll loop can then be rewritten as

1. DoAll $\vec{i}$
2. $\quad B[\vec{i}] = A[\vec{r}_a(\vec{i})] + A[\vec{r}_b(\vec{i})]$
3. EndDo

As described in section 5.5. there is one distance vector for array A in the loop:

$$\vec{d} = \vec{r_b} - \vec{r_a} = F_d \cdot \vec{i} + \vec{f}_d$$

Applying a partitioning equation to the distance vector, we have $\vec{p}_d = m(\vec{d})$, which represents the distance between the processor locations for the two array

elements ($A[\vec{r}_a]$ and $A[\vec{r}_b]$). In order to have all remote array elements of array A located in one processor, the value of $m(\vec{d})$ must be a constant vector for each processor. In other words, $m(\vec{d})$ depends only on the processor id.

We have

$$m(\vec{d}) = M \cdot (F_d\vec{i} + \vec{f}_d) \pmod{\vec{s}}$$

Let $\vec{t_1} = M \cdot \vec{f}_d \pmod{\vec{s}}$. $M = T \cdot U$. and substitute $\vec{i}$ by equation (4.4):

$$m(\vec{d}) = T \cdot U \cdot F_d \cdot (N \cdot \vec{p} + H \cdot \vec{k}) \pmod{\vec{s}} + \vec{t_1}$$

Let $\vec{t_2}(\vec{p}) = T \cdot U \cdot F_d \cdot N \cdot \vec{p} \pmod{\vec{s}}$:

$$m(\vec{d}) = T \cdot U \cdot F_d \cdot H \cdot \vec{k} \pmod{\vec{s}} + \vec{t_2}(\vec{p}) + \vec{t_1}$$

To have $m(\vec{d})$ depend only on $\vec{p}$, it is required that

$$(T \cdot U \cdot F_d \cdot H) \pmod{\vec{s}} \equiv \vec{0}$$

and hence

$$(T \cdot U \cdot F_d \cdot H) = S \cdot X'$$

where X' is any integer matrix, S is a diagonal matrix ($diag(s_1, s_2, \ldots)$) where $s_i | s_{i+1}$ (Smith Normal Form), and s_i are the elements of the vector $\vec{s}$.

Since $H = U^{-1} \cdot S$ as described in section 4.3.1, we have

$$T \cdot U \cdot F_d \cdot U^{-1} = S \cdot X' \cdot S^{-1}$$

where X' can be any integer matrices, and S^{-1} is a diagonal matrix with $\frac{1}{s_i}$ as its diagonal elements.

From the above derivation and Property 4.3.1, we now have the following property:

Property 4.4.1

Given the references to an array with a distance vector ($\vec{d} = F_d \cdot \vec{i} + \vec{f_d}$), the array can be partitioned such that all the remote references are located in one processor by using the periodic skewing equation $\vec{p} = m(\vec{i}) = (T \cdot U \cdot \vec{i}) \pmod{\vec{s}}$ with a value of T that satisfies (1) SNF of $[M\ S] = [I\ O]$, (2) $T = S \cdot X \cdot S^{-1}$, and (3) $T \cdot U \cdot F_d \cdot U^{-1} = S \cdot X' \cdot S^{-1}$, where X and X' can be any integer matrices.

4.4.2 An Example

The following DoAll loop is used to illustrate how the block-communication partitioning can be derived.

Example 4.4.1

Assume array size: $A[0..999, 0..999]$

1. *DoAll* $I = 0$ *to* $499, J = I$ *to* $999 - I$
2. $\quad B[I, J] = A[I, J] + A[I + J, J - I]$
3. *EndDo*

There are two references to the array A: $\vec{r_a} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \vec{i}$ $\quad \vec{r_b} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \vec{i}$

Thus, we have one distance vector $\vec{d}$ of array A: $\vec{d} = \vec{r_b} - \vec{r_a} = F_d \cdot \vec{i} + \vec{f_d} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{i} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

According to the property 4.4.1, the block-communication partitioning equation has the form:

$$\vec{p} = m(\vec{i}) = (T \cdot U \cdot \vec{i}) \pmod{\vec{s}}$$

Choose any SNF matrix S and any unimodular matrix U:

$$S = \begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix} = \begin{bmatrix} 5 & 0 \\ 0 & 10 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

The relation ($T \cdot U \cdot F_d \cdot U^{-1} = S \cdot X' \cdot S^{-1}$) as described in property 4.4.1 becomes

$$\begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 0 \\ 0 & 10 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \begin{bmatrix} \frac{1}{5} & 0 \\ 0 & \frac{1}{10} \end{bmatrix}$$

Hence $\begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \begin{bmatrix} -1 & 1 \\ -2 & 1 \end{bmatrix} = \begin{bmatrix} x_{11} & \frac{1}{2}x_{12} \\ 2x_{21} & x_{22} \end{bmatrix}$

where $t_{11}, \cdots, t_{22}, x_{11}, \cdots, x_{22}$ can be any integer.

If we choose $T = \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 2 & 1 \end{bmatrix}$ (which yields $X' = \begin{bmatrix} -2 & 4 \\ -2 & 1 \end{bmatrix}$, and $X = \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix}$ for $T = S \cdot X \cdot S^{-1}$), we obtain a block-communication partitioning equation [1]

$$\vec{p} = m(\vec{i}) = (T \cdot U \cdot \vec{i}) \pmod{\vec{s}} = \begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix} \cdot \vec{i} \pmod{\begin{bmatrix} 5 \\ 10 \end{bmatrix}} \tag{4.12}$$

Verification of the one block-communication property:

The set of array elements in processor $\vec{p}$ by partitioning equation (4.12) need to be known first. The derivation can be found in [54], equation (4.13) shows the result: ($A[\vec{i}]$ represents $A[I, J]$)

$$\vec{i} = \begin{bmatrix} -2 & 0 \\ 6 & 1 \end{bmatrix} \cdot \vec{p} + \begin{bmatrix} 5 & 0 \\ -15 & -10 \end{bmatrix} \cdot \vec{k}, \quad \vec{k} \in Z^2 \tag{4.13}$$

[1] After the partitioning equation for array A has been determined, the partitioning equation for array B can be derived by the alignment relation between $B[i]$ and any one of the references $(A[\vec{r}_a(\vec{i})], A[\vec{r}_b(\vec{i})])$. The partitioning equation for array B has the form: $\vec{p} = m_b(\vec{i}) = (M' \cdot \vec{i} + \vec{f}) \pmod{\vec{s'}}$. Equation (m_b) is more general than the periodic skewing equation for the extra constant vector $\vec{f}$. It should be noted that if equation (m_b) is used as the mapping equation for array A instead of equation (4.2), $\vec{f}$ will become any integer vector. If $\vec{f} = \vec{0}$, equation (m_b) would then reduce to equation (4.2).

The two references to array A in example 4.4.1 are (1) $\vec{r}_1 = \vec{i}$ and (2) $\vec{r}_2 = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \cdot \vec{i}$.

Applying the partitioning equation (4.12) to both references, $\vec{r}_1$ and $\vec{r}_2$ are located in processor:

$$\vec{p}_1 = \begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix} \cdot \vec{i} \pmod{\begin{bmatrix} 5 \\ 10 \end{bmatrix}}$$

$$\vec{p}_2 = \begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \cdot \vec{i} \pmod{\begin{bmatrix} 5 \\ 10 \end{bmatrix}}$$

Substituting $\vec{i}$ by the expression given in (4.13), we have:

$$\vec{p}_1 = (\begin{bmatrix} -4 & 0 \\ 0 & 1 \end{bmatrix} \cdot \vec{p} + \begin{bmatrix} 10 & 0 \\ 0 & -10 \end{bmatrix} \cdot \vec{k}) \pmod{\begin{bmatrix} 5 \\ 10 \end{bmatrix}} = \vec{p} \quad \text{as it should be.}$$

$$\vec{p}_2 = \begin{bmatrix} 3 & 2 \\ 0 & 4 \end{bmatrix} \cdot \vec{p} \pmod{\begin{bmatrix} 5 \\ 10 \end{bmatrix}}$$ which depends only on the processor id ($\vec{p}$).

Therefore, we have $A[\vec{r}_1]$ in processor $\vec{p}$ and $A[\vec{r}_2]$ in processor $\vec{p}_2$ which is independent to the loop index. In other words, all the $A[\vec{r}_2]$ required by processor $\vec{p}$ are located in processor $\vec{p}_2$, such that at most one block-communication is needed during computation.

4.4.3 Comparison with Other Partitioning Methods

The analytical result of skewing partitioning is compared to the usual BLOCK distribution and the CYCLIC distribution of the array. The amount of remote accesses and the number of block-communication required for the three partitioning methods are calculated and compared.

Figure 4.1 shows the amount of remote array elements required and the number of block-communications required across the processors for example 4.4.1. Assume that the size of array A is [1000, 1000]. The total number of processors is $5 \times 10 = 50$ (PE number $= 10 * p_1 + p_2$ for processor $\vec{p} = [p_1 \quad p_2]^T$). It is assumed that the remote data are aggregated for communication. Even though there are many references to the array elements in one unique remote PE, only one block-communication will be counted. Thus, the number of block-communication required for a given PE will be the same as the number of remote PEs which contain the

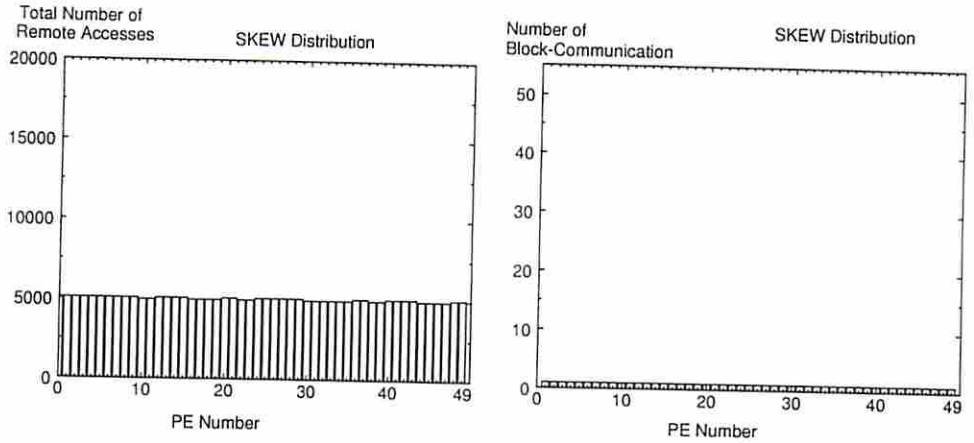


Figure 4.1: Our Skewing Partitioning

data needed during computation. Figure 4.1 also shows that at most one block-communication is required for each PE by the skewing partitioning.

Figure 4.2 and figure 4.3 show the amount of remote references and the number of block-communication for the usual BLOCK and the CYCLIC array partitioning methods, which distribute array A of size [1000, 1000] to an array of 50 PEs along the first dimension of array A for the same example 4.4.1.

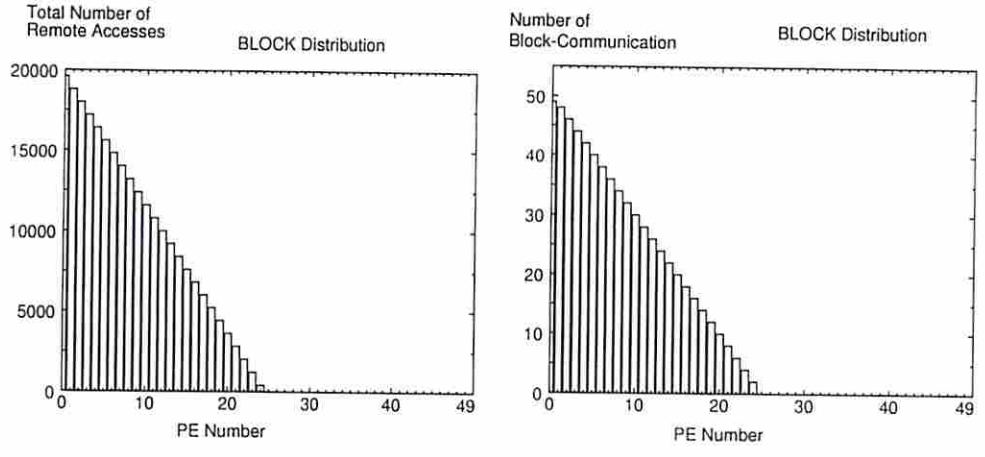


Figure 4.2: A[BLOCK,*] Partitioning

For the BLOCK distribution case, it can be seen that half of the processors have no computation load, and that the amount of the block-communication of the other processors is quite large.

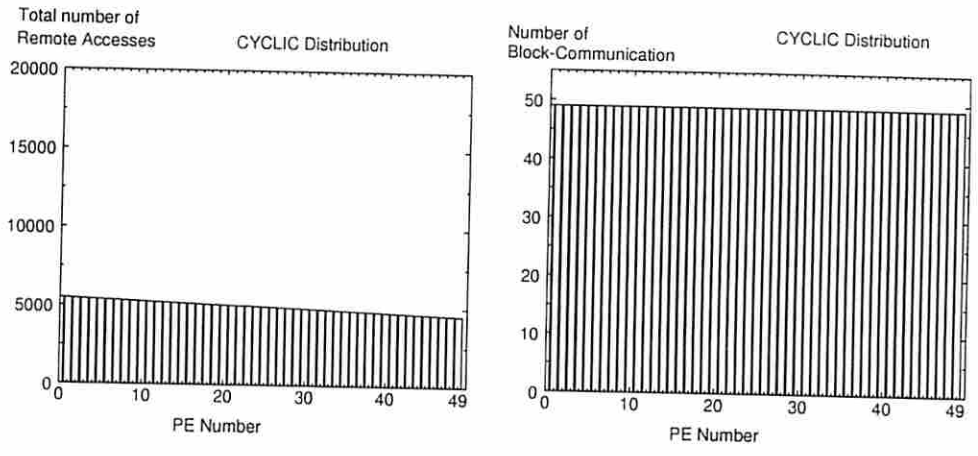


Figure 4.3: A[CYCLIC,*] Partitioning

For the CYCLIC distribution case, it can be seen that the amount of the block-communication is even higher than the amount required in the case of the BLOCK distribution.

We have developed two techniques of array partitioning based on the periodic skewing equation. We have derived the equation which describes all possible communication-free partitioning for arrays in a DoAll loop, and the equation which describes the mapping equation such that all remote data can be located in one processor. The methods were developed on a single DoAll-loop. However, it is also possible to adapt our method to multiple DoAll-loops by using the technique developed in the affine methods [51], where all dependence vectors of multiple loops are constructed into one single matrix form.

We have shown by an example that the load balance and one block-communication for each PE can be achieved by our method. The advantages of the periodic skewing equation have been shown for distributing arrays for vector computers. We have also demonstrated that the periodic skewing equation can be used to distribute arrays for communication-free and one block-communication for message-passing parallel

machines. Periodic skewing equations are of particular interest because they have an elegant foundation in the mathematical theory of lattices [55].

Chapter 5

Our Distributed-Array Compilation

Enumerations of local sets and communication sets are important issues in distributed array compilation of data parallel languages such as High Performance Fortran. The enumerations may not be straightforward when the distributed arrays are aligned to each other and block-cyclic distributed, which can result in heavy compilation-time and/or run-time overheads. The overheads can be reduced when the enumerations are able to be expressed in simple closed forms. In order to construct the communication sets, one single pass of local array with destination evaluation is usually required, which also results heavy run-time overheads. Again, the overheads can be reduced when the communication sets are able to be expressed in simple closed forms. In this paper, we show that the closed-from expressions of the local-set enumerations and of the communication-set can be elegantly derived by using the Smith Normal Form. Based on the SNF method, we also develop our corresponding SPMD code for a FORALL loop, which shows the same level of run-time complexity to that of the local set enumeration.

5.1 Storage and Local Enumeration

Some recent languages [11] provide user directives for distributed arrays and many research results were presented which focused on the enumeration of the local array elements [9, 19, 43, 47, 32, 33, 27, 24, 26, 15, 42, 45, 52, 53, 30].

The following example demonstrates how a local array is stored in local memory by PREPARE project method[30].

Example 5.1.1

```
1.      REAL A[39] /* 0.. 38 */
2.  !PROCESSORS P(4) /* 0..3 */
3.  !ALIGN A[I] with T[3*I+7]
4.  !DIST T[cyclic(4)] ONTO P
```

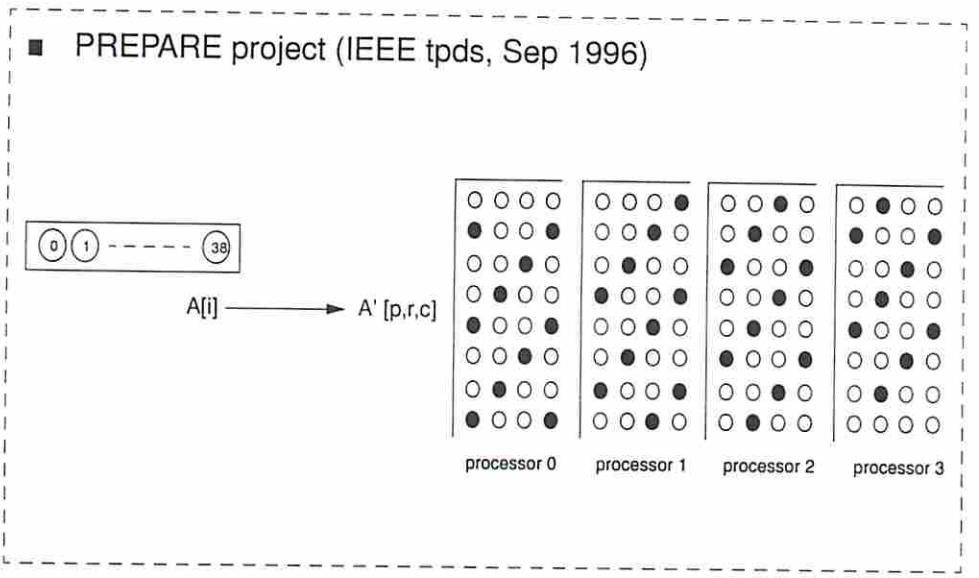


Figure 5.1: The storage of the local array

In this example, an array P of four processors is declared. The array A with 39 elements is to be distributed to the array of processor P according to the user directives. The result is shown in figure 5.1, where a 4×8 two dimensional array is used to store the local array in each processor. In figure 5.1, the black dots represent the local array elements (Local_Set) and the white dots represent the holes which contains no data. It can be seen that many elements in the 4×8 array space is wasted. However, according to the arrangement of the local array elements, the corresponding index function may be simplified.

The following shows its corresponding index conversion from the global index i to the local index $[r, c]$ of the processor p:

$$\begin{aligned} p &= \lfloor \tfrac{3 \times i + 7}{4} \rfloor \pmod 4 \\ r &= \lfloor \tfrac{3 \times i + 7}{4 \times 4} \rfloor \\ c &= (3 \times i + 7) \pmod 4 \end{aligned} \tag{5.1}$$

Local array elements are determined by the user directives, which usually contain the information of the data alignment and the data distribution. Studies of the local array storage can also be seen in [9, 42, 26]. Gerndt [20] considered non-aligned block distribution. In Paalvast [13] and Koelbel [33], cyclic(1) distribution was studied. In Paalvast [14], cyclic(m) was studied. The data alignment has been studied in recent publications [43, 26, 53, 30]. Most researches have restrictions on the alignments. In [45, 52], the FSM method is used. Germain and Delaplace [19] have proposed a scheme to examine the cyclic(1) distribution and have derived the closed forms for the communication sets.

The power of scalable parallel machines are hindered by the difficulties of parallel programming. This has motivated the design of data-parallel languages such as High Performance Fortran. HPF provides a single address space, FORALL statements, and data distribution directives such that users can program distributed memory parallel computers in a way as program familiar sequential machines. However, this also shifts the burden onto compilers that compilers are then responsible to partition computations and to determine communications for processors.

In order to generate distributed array codes for FORALL statements based on the data distribution and on the owner-computes rule where each processor performs the computations that modify values of the array elements it owns, the following aspects are required to be determined:

- Local array enumeration ($Local\_Array\_Set$). How do we enumerate the local elements of an array in each processor?
- Local storage scheme. How do we store the local elements of an array in local memory?

- Local set enumeration($Local\_Set$). What set of the local elements are to be computed by a given FORALL statement?

- Communication set. What is the set of array elements that required to be received from or sent to other processors during computation?

The data distribution directives used in the data-parallel languages usually are, for example in HPF, the $cyclic(n)$ distribution statement and the alignment statement.

A $cyclic(n)$ distribution statement specifies that the array elements are distributed n elements at a time across processors. (Figure 5.2 shows an example of $cyclic(2)$ distribution). An alignment statement specifies how the elements in two arrays are related. For example, HPF allows every element i of an array to be aligned (located on the same processor) with element $a \cdot i + b$ of another array, where a and b are arbitrary integers. By using these two types of statements, the mapping between array elements and processors can be specified. This array-processor mapping for each array can be modeled in two steps: (1) an array is first aligned to an template array by an alignment statement, and (2) the template array is distributed to a processor array by an distribution statement.

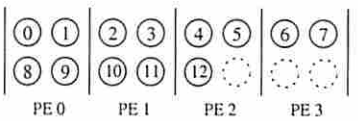


Figure 5.2: An one dimensional array of 13 elements is cyclic(2) distributed across four processors

Consider, as an example, the following HPF-like program, where array A is split into processor array P by alignment statement and distribute statement through template array T.

Example 5.1.2

```
1.  real A(0:26)
2.  ! processors P(0:1)
3.  ! template T(0:9999)
4.  ! distribute T(cyclic(8)) onto P
```

5. *! align A(k) with T(3*k)*

6. *FORALL (i=0:13) A(2*i) = ...*

In example 5.1.2, an array of two processors is declared by statement-2. Statement-3 declares a template array with 10000 elements, and the template array is to be cyclic(8) distributed across the processor array by statement-4. Statement-5 specifies the index relationship between array A and the template array. During, for example, $i = 3$ of the FORALL statement, $A(6)$ is calculated. Since $A(6)$ is aligned to $T(18)$ by statement-5, and $T(18)$ is distributed to $P(0)$ by statement-4, then the iteration $i = 3$ is to be executed by $P(0)$, and $A(6)$ should be distributed to $P(0)$. In other words, we say that $A(6)$ element should be in the $Local\_Set$ of processor $P(0)$, and $i = 3$ should be in the $Compute\_Set$ of $P(0)$.

Figure 5.3 shows which part of array A elements in example 5.1.2 should be stored in processor $P(0)$ and $P(1)$. The set of template T elements distributed to both processors are organized into a two-dimensional grid by r-axis, and c-axis. Sine every 8 elements of T are distributed at a time across the processor array, each row of the grid contains 8 elements. To be exact, the position (r, c) in PE(0) stands for template element $T(16r + c)$, and the position (r, c) in PE(1) stands for template element $T(16r + 8 + c)$. Figure 5.3 also shows the elements of array A distributed to both processors, which can be visualized and be represented as the circles in the figure. The location of the array A elements are determined according to their alignment relation to template T. The global loop indices $\{i\}$ are also shown under the circles if the corresponding elements of array A are to be calculated during these loop iterations.

The set of array A elements distributed to PE(0) or PE(1) can then be visualized in figure 5.3, which is the array elements represented by all the circles in PE(0) or PE(1). The set of array A elements ($Local\_Set$) to be calculated by PE(0) or PE(1) can also be shown by all the circles with iteration numbers under the circles. Since the iterations of FORALL loop can be executed in arbitrary order, the enumeration sequence of the local set can be arbitrary, which yields to different possibilities, for example:

- Rowwise enumeration.

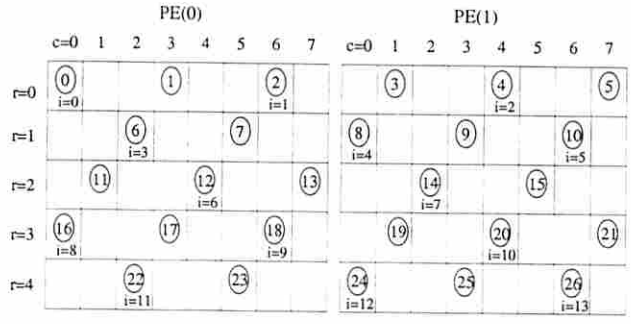


Figure 5.3: Array distribution for Example 5.1.2

The array elements to be computed in example 5.1.2 will be in the sequence as $\{A(0), A(2), A(6), A(12), A(16), A(18), A(22)\}$ in $PE(0)$.

- Columnwise enumeration.

  The array elements to be computed in example 5.1.2 will be in the sequence as $\{A(0), A(16), A(6), A(22), A(12), A(2), A(18)\}$ in $PE(0)$.

- Other enumeration methods.

  It is also possible to have sequences which is not rowwise nor columnwise, for example, $\{A(0), A(16), A(6), A(22), A(12), A(2), A(18)\}$.

The rowwise and columnwise enumerations are two of the standard ones, which have been explored in, for instances, [8] [42] [30]. We will review them in section 5.5.

The node code to be executed on each processor typically is in Single Program Multiple Data style. In SPMD, each processor executes on the same program but different code depending on the data stored in its local memory, which is depending on the processor ID.

The following code shows a possible transformed SPMD program in global address space for example 5.1.2:

Code 5.1.1

1. $p = processor\_ID;$
2. $for\ i = Compute\_Set(p)\ do$
3. $\quad g = 2 * i$

```
4.      A(g)= ....
5.  enddo
```

where the $Compute\_Set(p)$ is the set of loop indices that should be executed in $PE(p)$.

It should be noted that the array indices and the loop iteration indices discussed so far are all in the global space. In the SPMD codes, however, all the indices will be generated and referenced by using the local address space. For example, if the $r-c$ grids in figure 5.3 are used as the local storage scheme for array A in both PE(0) and PE(1), the rowwise enumeration of the local set of PE(0) in local (r, c) address will be in the sequence as $\{A(0,0), A(0,6), A(1,2), A(2,4), A(3,0), A(3,6), A(4,2)\}$. It shows that the local addresses generated also depend on what storage scheme that is used. As the result, the $r-c$ grids in figure 5.3 may not be a good storage scheme since some of the space are wasted. This inefficiency can be improved by using a compressed storage scheme, which will be discussed and shown as in figure 5.4.

It turns out that the above program can be typically compiled into a nested loop as follows:

Code 5.1.2

```
1.  p = processor_ID;
2.  for j0 = j0_min(p) to j0_max(p) do
3.      for j1 = j1_min(p) to j1_max(p) do
4.          l_local = f_local(j0, j1)
5.          A(l_local)= ....
6.      enddo
7.  enddo
```

where l_{local} is the address in local memory which may be (r, c) pair, or others, depending on what storage scheme is used for local array. And the set of $\{l_{local}\}$ represents the $Local\_Set$ in local address space. Noted that the global loop index i in loop 5.1.1 is replaced by the local loop indices (local variables) j_0 and j_1.

Also noted that the $Local\_Set_{[X]}$ is set of the local elements of array X to be computed by a given FORALL statement, and that the $Local\_Array\_Set_{[X]}$ is the set of local elements of array X in a processor.

When the FORALL statement in example 5.1.2 becomes

```
1.  FORALL (i=0:26) A(i)= ....
```

The enumeration of the local set will become the enumeration of local array set because this FORALL loop calculates all the elements of the whole array A. therefore, each processor will calculate all the elements of the array it owns, which is equivalent to the local array set. Thus, the enumeration of local set can be used to enumerate the local array set as described above.

In the case that the calculation of array elements in a FORALL loop requires the elements of another array, for example,

1. FORALL (i=0:26) A(i)= B(2*i)

When the array B elements needed are not local, communications are then required to bring in the elements. Thus, the compilers are also required to generate efficient codes in determining the following sets:

- $Receive\_Set_{[B]}(p, q)$: The set of array elements of array B which must be received by processor p from processor q.
- $Send\_Set_{[B]}(p, q)$: The set of array elements of array B which must be sent by processor p to processor q.

Assume that the set of array B elements referenced by processor p is known and represented as $Ref_{[B]}(p)$, and the set of array B elements in a processor q is $Local\_Array\_Set_{[B]}(q)$, then the communication sets can be expressed as:

- $Receive\_Set_{[B]}(p, q)$:

  $Receive\_Set = Ref_{[B]}(p) \cap Local\_Array_{[B]}(p)$

- $Send\_Set_{[B]}(p, q)$:

  $Send\_Set_{[B]}(p, q) = Receive\_Set_{[B]}(q, p)$

We will discuss the communication sets in more detail in section 2 and we will present our derivation method in section 4.

In this section, we will discuss the enumeration of the local set, which can also be used for enumerating the local array set and be used to deriving the communication sets.

In example 5.5.1, a one-dimensional array A is declared and is distributed to processor array P where all the parameters are represented in variables. According

to the HPF data distribution and HPF data alignment statements syntax, each dimension of an array will not be couple with another dimension, thus, example 5.5.1 can be serve as an generic case of data distribution in HPF.

Example 5.1.3

1. *real* $A(0:(n_a-1))$
2. *! processor* $P(0:(n_p-1))$
3. *! template* T
4. *! distribute T(cyclic(m)) onto P*
5. *! alignment A(i) with T(a*i+b)*
6. *FORALL* $(i_g = 0:(n_g-1))$ $A(\alpha * i_g + \beta) = ...$

According to the definitions of the *cyclic*(n) and the alignment statements, the relationships among the processor number p, the global loop index i_g, array A index i_a, template T index i_t, row number r, and column number c for the general array distribution can be expressed as

$$i_t = a \cdot i_a + b \tag{5.2}$$

$$c = i_t \pmod{m} \tag{5.3}$$

$$r = \left\lfloor \frac{i_t}{n_p \cdot m} \right\rfloor \tag{5.4}$$

$$p = \left\lfloor \frac{i_t}{m} \right\rfloor \pmod{n_p} \tag{5.5}$$

$$i_t = n_p \cdot m \cdot r + m \cdot p + c \tag{5.6}$$

$$i_a = \alpha \cdot i_g + \beta \tag{5.7}$$

where i_a, p, and c are bounded by

$$\begin{aligned} 0 \leq\ & i_a \leq n_a - 1 \\ 0 \leq\ & p \leq n_p - 1 \\ 0 \leq\ & i_g \leq n_i - 1 \\ 0 \leq\ & c \leq m - 1 \end{aligned} \tag{5.8}$$

Equation (5.33) shows the index bounds of the array A, of the processor P, of the loop iteration i_g, and of the column number c. The column number c and the

row number r construct the $r-c$ grid to represent the array T, which is $cyclic(m)$ distributed to processor array P by statement-4. Equations (5.3), (5.4) and (5.5) describe the relationships among i_t, r, c and p according to the definition of $cyclic(m)$ distribution, which can also be expressed in a single equation (5.32). Equation (5.34) describes the alignment relationship between array A and template array T by statement-5. And equation (5.35) describes the relationship between the loop index i_g and the accessing element of the array A by statement-6.

Based on the relationships (5.34)-(5.33), several methods have been developed to enumerate the local set. We now review some of the methods in the following.

The enumeration of the local set for a given processor index p can be expressed as the set of array elements $\{A(\alpha * i+\beta)|i \in Compute\_Set(p)\}$, where $Compute\_Set(p)$ is the set $\{i\}$ of loop indices to be executed in processor p. Therefore, in the Euclid's-extended-algorithm method (Euclid's method)[30], the enumeration of the local set is derived by first determining the $Compute\_Set(p)$.

Combine equations (5.34), (5.32) and (5.35), the following equation is obtained which expresses the relationship of loop index i_g, column number c, and row number r.

$$a_{co} \cdot i_g = c + n_p \cdot m \cdot r + m \cdot p + b_{co} \tag{5.9}$$

where $a_{co} = a \cdot \alpha$ and $b_{co} = a \cdot \beta + b$.

In order to derive the set of i_g for executing on processor p, a method based on the Euclid's algorithm was developed in [30], which is used to find all $x, y_0, \ldots, y_k \in Z$ that satisfy the equation

$$\alpha \cdot x = \beta_0 \cdot y_0 + \cdots + \beta_k \cdot y_k + v \tag{5.10}$$

for a given set of $\alpha, \beta_0, \ldots, \beta_k, v \in Z$.

This method will introduce variables $j_0, \cdots, j_k \in Z$ and the solution of x can be expressed by a linear relation of $j_0, \cdots, j_k$. For example, when $k = 0$,

$$\alpha \cdot x = \beta \cdot y_0 + v \tag{5.11}$$

The Euclid's algorithm finds the values of $x_0, y_{00}, \Delta x, \Delta y_0$, and $g(= gcd(\alpha, \beta))$ such that all $x, y_0 \in Z$ satisfied (5.11) is expressed by the following linear relationship of $j_0 \in Z$:

$$\begin{cases} \begin{bmatrix} x \\ y_0 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_{00} \end{bmatrix} \cdot (\frac{v}{g}) + \begin{bmatrix} \Delta x \\ \Delta y_0 \end{bmatrix} \cdot j_0 & \text{if } (\frac{v}{g}) \in Z \\ \text{No solution} & \text{otherwise} \end{cases}$$

According to the linear relationships for the variables $x, y_0, \ldots,$ and y_k, if we also apply the constraint to the variables by their bounds, all the integer tuples $(j_0, \cdots, j_k)$ can then be determined.

If the Euclid's method is applied to equation (5.36), and applying the bounds in (5.33), the following constraints are derived:

$$\begin{array}{rlll} 0 \leq & j_2 = p & \leq n_p - 1 & \\ r^{min} \leq & j_1 = r & \leq r^{max} & \\ 0 \leq & j_0 = i_g & \leq n_g - 1 & \\ 0 \leq -n_p \cdot m \cdot j_1 & -m \cdot j_2 + b_{co} & +a_{co} \cdot j_0 = c \leq m - 1 & \end{array} \tag{5.12}$$

where r^{min} and r^{max} are the lower and upper bounds of the row number r. According to equation (5.4), they can be expressed as:

$$\begin{aligned} r^{min} &= \lfloor \tfrac{b_{co}}{n_p \cdot m} \rfloor \\ r^{max} &= \lceil \tfrac{a_{co} \cdot (n_g - 1) + b_{co}}{n_p \cdot m} \rceil \end{aligned} \tag{5.13}$$

Based on (5.12), the first four statements (1-4) in nested loop 5.1.3 that enumerate the *Compute_Set* ($\{i_g\}$) can then be constructed by using any polyhedra scanning algorithm [4] [18]. The polyhedra scanning algorithm is a method to scanning all the elements in a set that is constraint by some inequations, such as in this case of equation (5.12).

Code 5.1.3

1. *for* $p = 0$ *to* $n_p - 1$ *do*

2. *for* $r = r^{min}$ *to* r^{max} *do*
3. *for* $i_g = max(0, \lceil \frac{n_p \cdot m \cdot j_1 + m \cdot j_2 - b_{co}}{a_{co}} \rceil)$ *to*
4. $min(n_g, \lfloor \frac{m-1+n_p \cdot m \cdot j_1 + m \cdot j_2 - b_{co}}{a_{co}} \rfloor)$ *do*
5. $i_{local} = g2l(\alpha * i_g + \beta)$
6. $A(i_{local}) = ...$
7. *enddo*
8. *enddo*
9. *enddo*

This enumeration sequence in loop 5.1.3 is called rowwise enumeration since the row number is enumerated first. The $(\alpha * i_g + \beta)$ in statement-5 represents one array index in the local set in the global address space, and the $g2l()$ maps a global address into a local address. Function $g2l()$ can be arbitrary as long as it is bijective, that different indices i map to different local addresses. However, a good $g2l()$ should have the property that all local array elements are able to be stored in compact and retangular local storage.

Two possible compressed storage schemes are also proposed in [30], which remove the holes in $r - c$ grids and result in the following two $g2l()$ functions, where global address i represents the element in the local set for a processor. Figure 5.4 shows the corresponding compressed storage schemes of $PE(0)$ for example 5.1.2. It should be noted that the storage scheme is orthogonal to the local set enumeration scheme, which means that storage scheme can be arbitrary as long as $g2l()$ is bijective.

- Rowwise storage:

$$g2l_{row}(i) = r \cdot \lfloor \frac{f_{co}(i)}{n_p \cdot m} \rfloor \cdot \lceil \frac{m}{a_{co}} \rceil + \lfloor \frac{f_{co}(i) \pmod m}{a_{co}} \rfloor \qquad (5.14)$$

  where $f_{co}(i) = a_{co} \cdot i + b_{co}$.

- Columnwise storage:

$$g2l_{col}(i) = \lfloor \frac{f_{co}(i)}{n_p \cdot m \cdot \Delta r} \rfloor \cdot \lceil \frac{m}{g} \rceil + \lfloor \frac{f_{co}(i) \pmod m}{g} \rfloor \qquad (5.15)$$

  where $g = gcd(a, n_p \cdot m)$, $\Delta r = a/g$ and a is the value in the alignment statement (5.34).

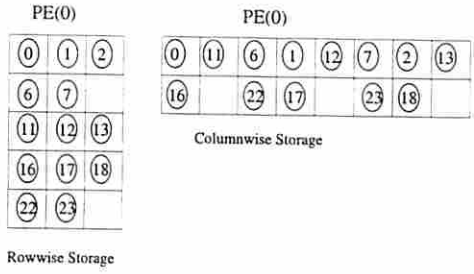


Figure 5.4: Rowwise and Columnwise Storage.

The enumeration sequence can also be arranged as columnwise order, if the Euclid's method is applied to the following equation (5.16). Noted that the sequence of the variables in (5.16) is different to that in (5.36) but they are equivalent equations.

$$a_{co} \cdot i_g = n_p \cdot m \cdot r + c + m \cdot p + b_{co} \tag{5.16}$$

It results to the following constraints:

$$\begin{array}{lll} 0 \leq & j_2 = p & \leq n_p - 1 \\ 0 \leq & b_{co} - m \cdot j_2 + g \cdot j_1 = c & \leq m - 1 \\ 0 \leq & i_0 \cdot j_1 + \Delta i \cdot j_0 = i & \leq n_g - 1 \end{array} \tag{5.17}$$

where $\Delta i = n_p \cdot m/g$, and i_0 is a solution of $a_{co} \cdot i - n_p \cdot m \cdot j_0 = g$.

Based on (5.17), the first four statements (1-4) in nested loop 5.1.4 that enumerate the *Compute_Set* ($\{i_g\}$) can be derived by using any polyhedra scanning algorithm.

Code 5.1.4

1. *for* $p = 0$ *to* $n_p - 1$ *do*
2. *for* $j_1 = \lceil \frac{-b_{co}+m \cdot p}{g} \rceil$ *to* $\lfloor \frac{m-1-b_{co}+m \cdot p}{g} \rfloor$ *do*
3. *for* $j_0 = \lceil \frac{-i_0 \cdot j_1}{\Delta i} \rceil$ *to* $\lfloor \frac{n_g-1-i_0 \cdot j_1}{\Delta i} \rfloor$ *do*
4. $i_g = i_0 \cdot j_1 + \Delta i \cdot j_0$
5. $i_{local} = g2l(\alpha * i_g + \beta)$
6. $A(i_{local}) = ...$

7. *enddo*
8. *enddo*
9. *enddo*

This enumeration sequence in loop 5.1.4 is called columnwise enumeration since the column number is enumerated first.

It should be noted that the local set enumeration can be served as the basis to derive the local array enumeration. When we set $\alpha = 1$, $\beta = 0$, and $n_i = n_a$, both nested loops 5.1.3 and 5.1.4 can be used to enumerate the local array set ($\{i_a\}$) for array A.

5.2 Efficient issues

A naive implementation for supporting the distributed arrays can be highly inefficient. For example,

Example 5.2.1 1. *ForAll i=0 to 4000, j= 0 to 4000*
2. $A(i+3, 2*j) = i+j$
3. *end for*

Without using the Compute_Set, the naive implementation would use the guarded statement which results in inefficiency due to the heavy run-time ownership testings in each processor:

1. ForAll i=0 to 4000, j=0 to 4000
2. if (Owner($A(i+3, 2*j)$) == me) $A(i+3, 2*j) = i+j$
3. end for

However, if the Compute_Set can be derived. An advanced compiler would generate execution codes without the heavy run-time ownership testings in each processor:

1. ForAll i in Compute_Set(my_PE_id)
2. $A(i+3, 2*j) = i+j$
3. end for

The challenge: How to derive the Compute_Set based on data distribution directives.

5.2.1 Efficient Communication

The efficiency of message-passing parallel computers is significantly determined by the number of messages. The implementation of data-parallel array statements in recent language systems all generate at most one message to each remote processor for the referenced array.

As an example, the message construction for example 5.2.1 could be implemented as:

1. ForAll i=0 to 40, j=0 to 40
2. if (owner($B(i+j, i+5)$ == me)
3. append $B(i+j, i+5)$ to the message for owner($A(i+3, 2*j)$);
4. end for

The above code also involves heavy run-time testing.

The challenge: How to derive the communication sets based on data distribution directives.

5.2.2 Efficient Local Storage of Local Array

In order to support the distributed arrays for message-passing parallel computers, compilers are required to determine a way to store the local array. An efficient representation of distributed arrays in the local memory of each processor can reduce storage overheads (holes in the memory which represent no data). Efficient functions for the conversion between the global index and local index of an array can reduce execution overhead.

The challenge: How to efficiently store local arrays in local memory based on data distribution directives.

5.3 Our Smith-Normal-Form Enumeration Method

In this section, we present our method on local set enumeration, which can then be applied to communication set enumeration.

In previous section, the implicit-lattice equation (5.37) is transformed into the explicit-lattice equation (2.13) for variables c, r and i_g by using the HNF decomposition. In order that such transformation can be performed, the Hermit matrix $H = [H_L\ 0]$ of F in (2.10) requires to have the property that H_L is unimodular (integer matrix invertible). Let us call such property as: unimodular Hermit property for matrix F. In the case that H_L is not unimodular, the HNF decomposition cannot be used to transform an implicit-lattice equation

$$F \cdot \vec{x} = f \tag{5.18}$$

into an explicit-lattice equation

$$\vec{x} = \vec{x_0} + H' \cdot \vec{k},\ \ k \in Z^n \tag{5.19}$$

Here, we present our technique in enumerating the local set. It is based on a method which is able to perform the explicit lattice transformation for the case that H_L is non-unimodular.

Our method in explicit-lattice transformation is based on Smith Normal Form (SNF) decomposition to the matrix F in (5.60).

According to the theorem of the Smith Normal Form (Theorem 5.6.1, an integer matrix F can be decomposed into $F = U^{-1} \cdot S \cdot V^{-1}$, where U and V are unimodular matrices, and S is a diagonal matrix.

We want to derive explicit-lattice equation for $\vec{x}$ in equation (5.60). The explicit-lattice equation (5.61) can be seen as the general solution of $\vec{x}$ for (5.60). The explicit transformation is also the same task as to find the general solution of $\vec{x}$ over Z which satisfies (5.60). We now describe the derivation as follows:

Assume F is a $r \times n$ matrix of rank r and $r \leq n$, there are unimodular matrices $U(r \times r)$, and $V(n \times n)$ such that $U \cdot F \cdot V = [S\ 0]$ is in the Smith Normal Form, where $S = diag(s_1, s_2, \ldots, s_r)$, and [0] is a zero matrix.

Let $V^{-1} \cdot \vec{x} = \vec{y}$, $U \cdot \vec{f} = \vec{d}$, (5.60) becomes

$$[S\ 0] \cdot \vec{y} = \vec{d}$$

Then the general solution can be expressed in terms of $(n-r)$ parameters y_i, $(r+1) \leq i \leq n$.

The constraint for $\vec{f}$ to have integer vector $\vec{x}$ can be determined by $[S \ \ 0] \cdot \vec{y} = U \cdot \vec{f}$. Let H_s be the HNF of $U^{-1} \cdot S$ $(= H_s \cdot Q)$, and $\vec{y}_{(1:r)}$ be the vector with the first r components of $\vec{y}$.

The constraint for integer vector $\vec{x}$ can then be expressed as

$$\vec{f} = U^{-1} \cdot [S \ \ 0] \cdot \vec{y} = H_s \cdot \vec{t} \tag{5.20}$$

where $\vec{t} = Q \cdot \vec{y}_{(1:r)}$ and $\vec{t} \in Z^r$.

If $[S]$ = Unit matrix, the constraint becomes $\vec{f} \in Z^n$. It is also the property that when $[S]$ = Unit matrix, the Hermit Normal Form of F will obey the unimodular hermit property.

The following shows an example to illustrate how the explicit-lattice transform is performed:

Example 5.3.1

Finding an explicit-lattice equation for x_1 and x_2 such that

$$\begin{bmatrix} 1 & 3 & 5 & 0 \\ 2 & 4 & 0 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

■

The derivation of the explicit lattice equation can be described by the following 5 steps:

1. Apply SNF decomposition to matrix F:

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \quad U = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}$$

$$V = \begin{bmatrix} 1 & 3 & 10 & -15 \\ 0 & -1 & -5 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \cdot \vec{y} = \vec{d} = U \cdot \vec{f} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix} \cdot \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

Hence

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix} \cdot \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{5.21}$$

and $y_3 \in Z, y_4 \in Z$.

2. The constraint for integer solution of $\vec{x}$:

$$\vec{f} = U^{-1} \cdot [S \;\; 0] \cdot \vec{y} = H_s \cdot \vec{t} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} t_1 \\ 2t_2 \end{bmatrix} \tag{5.22}$$

where $t_1, t_2 \in Z$.

3. Since $\vec{x} = V \cdot \vec{y}$:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & -1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 10 & -15 \\ -5 & 5 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_3 \\ y_4 \end{bmatrix} \tag{5.23}$$

4. From (5.21) and (5.23), x_1, x_2 can be expressed as:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 & \frac{3}{2} \\ 1 & \frac{-1}{2} \end{bmatrix} \cdot \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} + \begin{bmatrix} 10 & -15 \\ -5 & 5 \end{bmatrix} \cdot \begin{bmatrix} y_3 \\ y_4 \end{bmatrix}$$

where y_3 and $y_4 \in Z$

5. The matrix $\begin{bmatrix} 10 & -15 \\ -5 & 5 \end{bmatrix}$ can further transformed to its Hermit Normal Form (Unique lower triangular matrix for every integer matrix):

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 & \frac{3}{2} \\ 1 & \frac{-1}{2} \end{bmatrix} \cdot \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} + \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix} \cdot \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} \tag{5.24}$$

where k_1 and $k_2 \in Z$

Thus, we have the explicit lattice equation (5.24) with constraint (5.22) of $\vec{f}$ for example 5.3.1.

Based on our explicit-lattice transformation, it can be used just like the HNF explicit-lattice transformation in the HNF method to derive the enumeration of local set, and to derive enumerate for local set in the case of dimension-couple alignment.

Since our SNF explicit-lattice transformation releases the limitation of unimodular Hermit property on matrix F in equation 5.60, we can also derive the communication sets, which will be described in section 5.5.2. We are also able to derive the columnwise and rowwise enumeration of local set by applying the techniques in the followings.

5.3.1 Columnwise Enumeration

Equation (5.36) can be reorganized into the following matrix form $F \cdot \vec{x} = f$:

$$\begin{bmatrix} n_p \cdot m & -a_{co} \end{bmatrix} \cdot \begin{bmatrix} r \\ i_g \end{bmatrix} = b_{co} - m \cdot p - c \tag{5.25}$$

Apply SNF decomposition to F:

$$S = [g \;\; 0],\; U = [1],\; V = \begin{bmatrix} \mu & \frac{a_{co}}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix}$$

where $g = gcd(n_p \cdot m,\ a_{co})$ and $g = n_p \cdot m \cdot \mu - a_{co} \cdot \omega$.

Then

$$[g \;\; 0] \cdot \vec{y} = U \cdot \vec{f} = b_{co} - m \cdot p - c$$

Hence the constraint to have integer lattice is:

$$c = b_{co} - m \cdot p - g \cdot y_1,\quad y_1 \in Z \tag{5.26}$$

Since $\vec{x} = V \cdot \vec{y}$, we have

$$\begin{bmatrix} r \\ i_g \end{bmatrix} = \begin{bmatrix} \mu & \frac{a_{co}}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Hence

$$i_g = \omega \cdot y_1 + (\frac{n_p \cdot m}{g}) \cdot y_2,\quad y_2 \in Z \tag{5.27}$$

Thus we have the set of equation (5.26) and (5.27), which is equivalent to the set of (5.17). Therefore, it can result to the same nested loop 5.1.4.

5.3.2 Rowwise Enumeration

Equation (5.36) can also be reorganized into the following matrix form $F \cdot \vec{x} = f$:

$$\begin{bmatrix} 1 & -a_{co} \end{bmatrix} \cdot \begin{bmatrix} c \\ i_g \end{bmatrix} = b_{co} - m \cdot p - n_p \cdot m \cdot r \tag{5.28}$$

Apply SNF decomposition to F:

$$S = [1 \;\; 0],\; U = [1],\; V = \begin{bmatrix} 1 & a_{co} \\ 0 & 1 \end{bmatrix}$$

Then

$$[1 \;\; 0] \cdot \vec{y} = U \cdot \vec{f} = b_{co} - m \cdot p - n_p \cdot m \cdot r$$

Hence, $r \in Z$ (no additional constraint to r).

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} b_{co} - m \cdot p - n_p \cdot m \cdot r \\ y_2 \end{bmatrix}$$

Since $\vec{x} = V \cdot \vec{y}$, we have

$$\begin{bmatrix} c \\ i_g \end{bmatrix} = \begin{bmatrix} 1 & a_{co} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_{co} - m \cdot p - n_p \cdot m \cdot r \\ y_2 \end{bmatrix} \tag{5.29}$$

Equation (5.29) is equivalent to the set of (5.12). And it can result to the same nested loop 5.1.3.

By using the implicit lattice equation (5.25) where $\vec{x} = \begin{bmatrix} r \\ i_g \end{bmatrix}$, we result to the columnwise enumeration of local set. By using the implicit lattice equation (5.28) where $\vec{x} = \begin{bmatrix} c \\ i_g \end{bmatrix}$ we result to the rowwise enumeration of local set. We now examine the other case that $\vec{x} = \begin{bmatrix} c \\ r \end{bmatrix}$ in the following, which will also result to the rowwise enumeration of local set.

Equation (5.36) can also be reorganized into the following matrix form $F \cdot \vec{x} = f$:

$$\begin{bmatrix} 1 & n_p \cdot m \end{bmatrix} \cdot \begin{bmatrix} c \\ r \end{bmatrix} = b_{co} - m \cdot p + a_{co} \cdot i_g \tag{5.30}$$

Apply SNF decomposition to F:

$$S = [1\ \ 0],\ U = [1],\ V = \begin{bmatrix} 1 & -n_p \cdot m \\ 0 & 1 \end{bmatrix}$$

Then

$$[1\ \ 0] \cdot \vec{y} = U \cdot \vec{f} = b_{co} - m \cdot p - a_{co} \cdot i_g$$

Hence, $i_g \in Z$ (no additional constraint to i_g).

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} b_{co} - m \cdot p - a_{co} \cdot i_g \\ y_2 \end{bmatrix}$$

Since $\vec{x} = V \cdot \vec{y}$, we have

$$\begin{bmatrix} c \\ r \end{bmatrix} = \begin{bmatrix} 1 & -n_p \cdot m \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_{co} - m \cdot p - a_{co} \cdot i_g \\ y_2 \end{bmatrix} \tag{5.31}$$

Equation (5.31) is equivalent to the set of (5.12). And it can also result to the same nested loop 5.1.3.

5.4 Our Communication Enumeration and SPMD Scheme

5.5 Basic Concepts

In the following, we will discuss some background on the enumeration of the local set. It can also serve as a basic algorithm for the enumeration of the communication set.

5.5.1 Local Address Enumeration

Example 5.5.1 shows a generic case of data distribution in HPF where a one-dimensional array A is declared and is distributed across processor array P. The rhs of statement 6 is not involved in determining the local set enumeration and is thus not shown.

Example 5.5.1

1. *real* $A(0:(n_a-1))$
2. *! processor* $P(0:(n_p-1))$
3. *! template* $T(0:(n_t-1))$
4. *! distribute* $T(cyclic(m))$ *onto* P
5. *! alignment* $A(i)$ *with* $T(a*i+b)$
6. *FORALL* $(i_g = 0:(n_g-1))$ $A(\alpha * i_g + \beta) = ...$

"$cyclic(m)$" in statement-4 distributes m elements of template T at a time to each processor across processor array P. Template T distributed in each processor can be organized into a two-dimensional $r-c$ grid, where the row number r specifies the r-th round of the m elements distribution to a processor, and the column number c specifies an element in each set of m elements.

The relationships among the processor number p, the template T index i_t, the row number r, and the column number c can be expressed [21]:

$$i_t = n_p \cdot m \cdot r + m \cdot p + c \tag{5.32}$$

The index bounds of the array A, of the processor P, of the loop iteration i_g, and of the column number c can be described by equations (5.33). The template T is an abstract array. It is assumed that n_T is large enough to be aligned with any array A element.

$$\begin{aligned} 0 \leq\ & i_a \leq n_a - 1 \\ 0 \leq\ & p \leq n_p - 1 \\ 0 \leq\ & i_g \leq n_g - 1 \\ 0 \leq\ & c \leq m - 1 \end{aligned} \tag{5.33}$$

The alignment in statement-5 specifies that the element $A(i)$ and the element $T(a * i + b)$ are to be located in the same processor. The relationship between the array A index i_a and the template T index i_t can thus be expressed as:

$$i_t = a \cdot i_a + b \tag{5.34}$$

According to statement-6, we also have:

$$i_a = \alpha \cdot i_g + \beta \tag{5.35}$$

Combining equations (5.32), (5.34) and (5.35), the following equation expresses the relationship between the loop index i_g, the column number c, and the row number r.

$$a_{co} \cdot i_g = c + n_p \cdot m \cdot r + m \cdot p - b_{co} \tag{5.36}$$

where $a_{co} = a \cdot \alpha$ and $b_{co} = a \cdot \beta + b$.

Equation (5.36) can also be reorganized into the following matrix form:

$$\begin{bmatrix} 1 & n_p \cdot m & -a_{co} \end{bmatrix} \cdot \begin{bmatrix} c \\ r \\ i_g \end{bmatrix} = b_{co} - m \cdot p \tag{5.37}$$

For a given alignment and the *cyclic*(m) data distribution, the set of local array elements that each processor must modify forms a lattice [1, 8] in the r-axis and the c-axis space, which means that the set of array elements can be generated by a linear combination of basis vectors of that lattice. Equation (5.36) or (5.37) is an implicit-lattice equation for the values of r, c and i_g. These values are of primary interest to us: they represent the local addresses in the $r - c$ grid and the corresponding loop

index i_g. To enumerate their values, an explicit-lattice equation for r, c and i_g can be derived from (5.37), which has the following form:

$$\begin{bmatrix} c \\ r \\ i_g \end{bmatrix} = \vec{x}_0 + H' \cdot \vec{k}, \quad \vec{k} \in Z^m \tag{5.38}$$

Equation (5.38) can also be seen as the inverse of the implicit-lattice equation (5.37). The Hermite-Form decomposition [8] has been used to derive the inverse form as the following:

$$\begin{bmatrix} c \\ r \\ i_g \end{bmatrix} = \begin{bmatrix} b_{co} - m \cdot p \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -g & 0 \\ \mu & \frac{a_{co}}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix} \cdot \vec{k} \tag{5.39}$$

Based on (5.39) and (5.33), the following constraints can be derived:

$$\begin{array}{lll} 0 \leq & p & \leq n_p - 1 \\ 0 \leq & c = b_{co} - m \cdot p - g \cdot y_1 & \leq m - 1 \\ 0 \leq & i_g = \omega \cdot y_1 + (\frac{n_p \cdot m}{g}) \cdot y_2 & \leq n_g - 1 \end{array} \tag{5.40}$$

where $y_1, y_2 \in Z$

Equation (5.40) can be reorganized in matrix form for each processor p:

$$\begin{bmatrix} g & 0 \\ -g & 0 \\ -\omega & -\frac{n_p \cdot m}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \leq \begin{bmatrix} b_{co} - m \cdot p \\ -b_{co} + m \cdot p + m - 1 \\ 0 \\ n_g - 1 \end{bmatrix} \tag{5.41}$$

According to equation (5.41), all the integer tuples of $\vec{y}$ can be determined by using a polyhedral scanning algorithm, such as [4, 23].

The column vectors of matrix $\begin{bmatrix} -g & 0 \\ \mu & \frac{a_{co}}{g} \\ \omega & \frac{n_p \cdot m}{g} \end{bmatrix}$ in equation (5.38) are the basis vectors of the lattice which models the enumeration set of the local addresses. Some other techniques [30] have been also derived an equivalent set of the basic vectors. In [1], a set of basis vectors in closed-form expression has also been developed when the alignment is restricted to unit stride. In some other methods [29, 42], the lattice have also been modeled by the FSM tables.

Many research projects have focused intensively and provided efficient solutions for the enumeration of local addresses or for message packing [8, 30, 42]. Since message packing is based on enumeration of local addresses, this is where most of the research efforts have focused.

At first, some special cases of the local address enumeration have been studied, such as: Koelbel and Mehrotra [33] have studied the case when arrays are accessed in unit stride. Paalvast et al. [14] have investigated the *cyclic*(n) distribution without alignment, where rowwise and columnwise distribution scheme are discussed. Benkner et al. [41] have also presented a similar technique. Then, the general local address enumeration have been studied in recent publications, such as: Kaushik et al. [47] have studied the block distribution case and cyclic distributions by using processor virtualization technique. Chatterjee et al.[42] developed a finite state machine (FSM) approach to enumerate local elements, which involves solving a set of linear Diophantine equations to determine the enumeration pattern. Kennedy et al. [29] have derived an algorithm which improves over that in [42] by avoiding the additional Diophantine equations in sorting stage. Stichnoth et al. [25] have presented a dual method for array allocation, where blocks are compressed before the cyclic number is handled. Gupta et al. [43] have also presented a similar method, which solve the block distribution case and use processor virtualization to handle cyclic distribution. Ancourt et al. [8] presented a linear algebra framework to solve some fundamental problems of HPF implementation, where the Hermite Normal form is used to derive the lattice pattern of address enumeration. Thirumalai and Ramanujam [1] have derived closed form expressions for the basic vectors of the lattice pattern as a function of the mapping of data. Sips et al. [21] presented and analyzed several local address enumeration and storage compression schemes.

Once we are able to enumerate the local set, the communication set can then be constructed by using one single-pass over local array to calculate the information and packs elements to the corresponding message buffer as shown below:

```
1.  for each local element
2.        Compute its destination d and
3.        pack it to sending_buffer[d]
4.  end for
```

The enumeration of the communication set in HPF, in fact, can also be modeled as an integer lattice. A pattern table, therefore, has been used by Venkatachar et al. [2] to model the lattice of the send-receive index pairs when the alignment is restricted to unit stride. Kennedy et al. [28] have modeled the communication set by an FSM table based on exploiting the repetitive nature of array accesses. In both methods, their distribution parameters are required to be known such that the FSM tables [28] or Pattern tables [2] can be built to construct the communication set. Chatterjee et al. show in [42] that only a single-pass over local array is needed to calculate the information and packs elements using an *address-value* pair for sending. However, it requires an ownership computation and known parameters to construct the FSM tables. Gupta et al. [43] and Stichnoth et al. [25] discuss similar schemes for enumerating the communication sets. Ancourt et al. [8] presented a linear algebra methods to develop the non-local data exchanges, however, their SPMD model cannot interleave local computations and communications.

Unlike other works on this problem, our approach derives a parametric solution for such integer lattice by using the Smith-Normal-Form analysis. We are able to describe the lattice symbolically for communication set when arrays involves non-unit stride alignment and cyclic(m) distribution. We will present our algorithm in section 5.5.2 for the SPMD code generation which can be symbolically resolved at compile-time except one matrix triangular factorization.

In this section, We will now develop our communication enumeration and our SPMD scheme, where the lattice which models communication set can be elegantly derived in a symbolic form.

5.5.2 Analysis of massage-packing and message-unpacking

The following generic FORALL loop in example 5.5.2 will be used to illustrate our scheme and its corresponding SPMD code will be developed. In example 5.5.2, two one-dimensional arrays A and B are declared, and they are distributed to processor array P through templates array T_A and T_B respectively.

Example 5.5.2

1. *real* $A(0:(n_a-1))$, $B(0:(n_b-1))$
2. *! processor* $P(0:(n_p-1))$
3. *! template* T_A, T_B
4. *! distribute* $T_A(cyclic(m_A))$ *onto* P
5. *! distribute* $T_B(cyclic(m_B))$ *onto* P
6. *! alignment* $A(i)$ *with* $T(a_A * i + b_A)$
7. *! alignment* $B(i)$ *with* $T(a_B * i + b_B)$
8. *FORALL* $(i_g = 0:(n_g-1))$
9. $\quad A(\alpha_A * i_g + \beta_A) = B(\alpha_B * i_g + \beta_B)$

According to equation (5.37), the relationship among the processor index p, the loop index i_g, the column number c_A, and the row number r_A for array A can be expressed as:

$$\begin{bmatrix} 1 & n_p \cdot m_A & -a_{co_A} \end{bmatrix} \cdot \begin{bmatrix} c_A \\ r_A \\ i_g \end{bmatrix} = b_{co_A} - m_A \cdot p \tag{5.42}$$

where $a_{co_A} = a_A \cdot \alpha_A$ and $b_{co_A} = a_A \cdot \beta_A + b_A$.

Equation (5.42) is also an implicit-lattice equation for $[c_A\ r_A\ i_g]^T$. The explicit-lattice form (equations 5.39) for $[c_A\ r_A\ i_g]^T$ can be obtained by the SNF explicit-lattice transformation, which results in the following equation ($\vec{x}_A = \vec{x}_{0_A} + H_A \cdot \vec{u}_A$):

$$\begin{bmatrix} c_A \\ r_A \\ i_g \end{bmatrix} = \begin{bmatrix} b_{co_A} - m_A \cdot p \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -g_A & 0 \\ \mu_A & \frac{a_{co_A}}{g_A} \\ \omega_A & \frac{n_p \cdot m_A}{g_A} \end{bmatrix} \cdot \begin{bmatrix} u_{1A} \\ u_{2A} \end{bmatrix} \tag{5.43}$$

Equation (5.43) is an explicit lattice equation in enumerating the local set (r_A and c_A) and the Compute_Set (i_g), which is the set of loop indices assigned to each processor.

Similarly to the derivation of the equation (5.42), the following equation (5.44) for the rhs $B(\alpha_2 * i_g + \beta_2)$ can be obtained. Equation (5.44) represents the set of array B elements ($\{(r_B, c_B)\}$) in the processor q that will be referenced by all the processors $P(0 : n_p - 1)$ (including itself) during the FORALL computation.

$$\begin{bmatrix} 1 & n_p \cdot m_B & -a_{co_B} \end{bmatrix} \cdot \begin{bmatrix} c_B \\ r_B \\ i_g \end{bmatrix} = b_{co_B} - m_B \cdot q \tag{5.44}$$

where $a_{co_B} = a_B \cdot \alpha_B$ and $b_{co_B} = a_B \cdot \beta_B + b_B$.

Equation (5.44) is an implicit-lattice equation for $[c_B\ r_B\ i_g]^T$. The explicit-lattice form for $[c_B\ r_B\ i_g]^T$ can also be obtained by the SNF explicit-lattice transformation, which results in the following equation ($\vec{x}_B = \vec{x}_{0_B} + H_B \cdot \vec{u}_B$):

$$\begin{bmatrix} c_B \\ r_B \\ i_g \end{bmatrix} = \begin{bmatrix} b_{coB} - m_B \cdot q \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -g_B & 0 \\ \mu_B & \frac{a_{co_B}}{g_B} \\ \omega_B & \frac{n_p \cdot m_B}{g_B} \end{bmatrix} \cdot \begin{bmatrix} u_{1B} \\ u_{2B} \end{bmatrix} \tag{5.45}$$

Again, equation (5.45) is the explicit equation to enumerate the set of array B elements ($\{(r_B, c_B)\}$) in the processor q that will be referenced by all the processors $P(0 : n_p - 1)$ (including itself) during the FORALL computation.

In order to pack the message that is sent by processor q to processor p, one may further partition the set described in equation (5.45) for each referencing processor p.

It can be seen that the global loop index i_g exists in both equations (5.43) and (5.45). Equation (5.46) shows the last row in equation (5.43). Equation (5.47) shows the last row in equation (5.45).

$$i_g = \begin{bmatrix} \omega_A & \frac{n_p \cdot m_A}{g_A} \end{bmatrix} \cdot \begin{bmatrix} u_{1A} \\ u_{2A} \end{bmatrix} \tag{5.46}$$

$$i_g = \begin{bmatrix} \omega_B & \frac{n_p \cdot m_B}{g_B} \end{bmatrix} \cdot \begin{bmatrix} u_{1B} \\ u_{2B} \end{bmatrix} \tag{5.47}$$

The intersection of both sets $\{i_g\}$ described by equations (5.46) and (5.47) thus corresponds to the set of global loop index $\{i_g\}$ that will be executed in the processor p, and that will refer array B elements in the processor q. In other words, this intersection expresses the set of array B elements in processor q that are needed by processor p during the corresponding set of loop index $\{i_g\}$. Similarly, the intersection expresses the set of array A elements in processor p that refer the array B elements in processor q during the corresponding set of loop index $\{i_g\}$.

Thus, we combine both equations (5.46) and (5.47) to have the following single matrix equation ($F \cdot \vec{x} = 0$):

$$\begin{bmatrix} \omega_A & \frac{n_p \cdot m_A}{g_A} & -\omega_B & -\frac{n_p \cdot m_B}{g_B} \end{bmatrix} \cdot \begin{bmatrix} u_{1A} \\ u_{2A} \\ u_{1B} \\ u_{2B} \end{bmatrix} = 0 \tag{5.48}$$

Equation (5.48) is also an implicit-lattice equation for $\vec{x}$. To generate the communication sets, the explicit-lattice equation for $\vec{x}$ is required. By applying the SNF explicit-lattice transformation as described in section 5.6, $\vec{x}$ can be expressed in explicit-lattice form as:

$$\vec{x} = \begin{bmatrix} u_{1A} \\ u_{2A} \\ u_{1B} \\ u_{2B} \end{bmatrix} = N \cdot \vec{k}, \quad \vec{k} \in Z^n \tag{5.49}$$

where $\vec{k}$ are constrained by the bounded $\vec{x}$. The bound on $\vec{x}$ is derived based on equation (5.41).

For clarity purpose, we put the symbolic derivation of N for the explicit-lattice equation (5.49) in section 5.6.2. Equation (5.49) can be decomposed into the following two equations, where N_A is the sub-matrix which contains the first two rows of N, and N_B is the sub-matrix which contains the last two rows of N:

$$\vec{u}_A = \begin{bmatrix} u_{1A} \\ u_{2A} \end{bmatrix} = N_A \cdot \vec{k} \tag{5.50}$$

and

$$\vec{u}_B = \begin{bmatrix} u_{1B} \\ u_{2B} \end{bmatrix} = N_B \cdot \vec{k} \tag{5.51}$$

As described in equation (5.41), the constraint on $[u_{1_A} \quad u_{2_A}]^T$ and $[u_{1_B} \quad u_{2_B}]^T$ are:

$$\begin{bmatrix} -b_{co_A} + m_A \cdot p \\ 0 \end{bmatrix} \leq \begin{bmatrix} -g_A & 0 \\ \omega_A & \frac{n_p \cdot m_A}{g_A} \end{bmatrix} \cdot \vec{u}_A \leq \begin{bmatrix} -b_{co_A} + m_A \cdot p + m_A - 1 \\ n_g - 1 \end{bmatrix} \tag{5.52}$$

$$\begin{bmatrix} -b_{co_B} + m_B \cdot p \\ 0 \end{bmatrix} \leq \begin{bmatrix} -g_B & 0 \\ \omega_B & \frac{n_p \cdot m_B}{g_B} \end{bmatrix} \cdot \vec{u}_B \leq \begin{bmatrix} -b_{co_B} + m_B \cdot p + m_B - 1 \\ n_g - 1 \end{bmatrix} \tag{5.53}$$

Combining equations (5.49), (5.52) and (5.53), we have

$$\begin{bmatrix} -b_{co_A} + m_A \cdot p \\ 0 \\ -b_{co_B} + m_B \cdot q \\ 0 \end{bmatrix} \leq \begin{bmatrix} -g_A & 0 & 0 & 0 \\ \omega_A & \frac{n_p \cdot m_A}{g_A} & 0 & 0 \\ 0 & 0 & -g_B & 0 \\ 0 & 0 & \omega_B & \frac{n_p \cdot m_B}{g_B} \end{bmatrix} \cdot N \cdot \vec{k} \leq \begin{bmatrix} -b_{co_A} + m_A \cdot p + m_A - 1 \\ n_g - 1 \\ -b_{co_B} + m_B \cdot q + m_B - 1 \\ n_g - 1 \end{bmatrix} \tag{5.54}$$

Equation (5.54) can be re-written as:

$$\vec{l}_{(p,q)} \leq M \cdot \vec{k} \leq \vec{u}_{(p,q)} \tag{5.55}$$

where

$$\vec{l}_{(p,q)} = \begin{bmatrix} -b_{co_A} + m_A \cdot p \\ 0 \\ -b_{co_B} + m_B \cdot q \\ 0 \end{bmatrix} \quad \vec{u}_{(p,q)} = \begin{bmatrix} -b_{co_A} + m_A \cdot p + m_A - 1 \\ n_g - 1 \\ -b_{co_B} + m_B \cdot q + m_B - 1 \\ n_g - 1 \end{bmatrix}$$

$$M = \begin{bmatrix} -g_A & 0 & 0 & 0 \\ \omega_A & \frac{n_p \cdot m_A}{g_A} & 0 & 0 \\ 0 & 0 & -g_B & 0 \\ 0 & 0 & \omega_B & \frac{n_p \cdot m_B}{g_B} \end{bmatrix} \cdot N = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{bmatrix}$$

where

$$\begin{aligned}
m_{11} &= -g_A(-a \cdot t_2 + \beta_2) \\
m_{12} &= -g_A(-a \cdot t_3 + a' \cdot \beta_3) \\
m_{13} &= -g_A(-a \cdot t_4 + a'' \cdot \beta_4) \\
m_{21} &= -\omega_A(-a \cdot t_2 + \beta_2) + \tfrac{n_p \cdot m_A}{g_A}(-b \cdot t_2 + \alpha_2) \\
m_{22} &= -\omega_A(-a \cdot t_3 + a' \cdot \beta_3) + \tfrac{n_p \cdot m_A}{g_A}(-b \cdot t_3 + b' \cdot \beta_3) \\
m_{23} &= -\omega_A(-a \cdot t_4 + a'' \cdot \beta_4) + \tfrac{n_p \cdot m_A}{g_A}(-b \cdot t_4 + b'' \cdot \beta_4) \\
m_{31} &= -g_B(-c \cdot t_2) \\
m_{32} &= -g_B(-c \cdot t_3 + \alpha_3) \\
m_{33} &= -g_B(-c \cdot t_4 + c'' \cdot \beta_4) \\
m_{41} &= -\omega_B(-c \cdot t_2) + \tfrac{n_p \cdot m_B}{g_B}(-d \cdot t_2) \\
m_{42} &= -\omega_B(-c \cdot t_3 + \alpha_3) + \tfrac{n_p \cdot m_B}{g_B}(-d \cdot t_3) \\
m_{43} &= -\omega_B(-c \cdot t_4 + c'' \cdot \beta_4) + \tfrac{n_p \cdot m_B}{g_B}(-d \cdot t_4 + \alpha_4)
\end{aligned}$$

It should be noted that $M \cdot \vec{k}$ characterizes an integer lattice in space $\vec{k}$ where each column in M represents the basis vector of the lattice of the communication set enumeration. It should be also noted that the matrix M is obtained symbolically without executing the *inspector*-like codes.

According to the theorem that a lattice characterized by a integer matrix M can also be characterized by any matrix $M_L = (M \cdot F)$ supported that the matrix F is unimodular. Therefore, equation (5.54) can also be transformed into equation (5.56)

where the matrix $M_L = M \cdot F$ is a lower triangular matrix and F is an unimodular matrix.

$$\vec{l}_{(p,q)} \leq M_L \cdot \vec{k'} \leq \vec{u}_{(p,q)} \tag{5.56}$$

where

$$M_L = \begin{bmatrix} d_{11} & 0 & 0 \\ d_{21} & d_{22} & 0 \\ d_{31} & d_{32} & d_{33} \\ d_{41} & d_{42} & d_{43} \end{bmatrix}$$

$\vec{k'}$ is an arbitrary integer vector. The following equations show the conversion between $\vec{k}$ and $\vec{k'}$:

$$\begin{aligned} \vec{k'} &= F^{-1} \cdot \vec{k} \\ \vec{k} &= F \cdot \vec{k'} \end{aligned} \tag{5.57}$$

The unimodular matrix F in (5.57) can be derived by applying the Gaussian elimination method to the matrix M. The matrix M is factorized into a triangular matrix by multiplying with a sequence of elementary matrices or permutation matrices such that $M_L = M \cdot F = M \cdot T_1 \cdot T_2 \cdot \cdots \cdot T_n$. Therefore, $F = T_1 \cdot T_2 \cdot \cdots \cdot T_n$, and F is unimodular since all T_i are unimodular.

Based on equation (5.56), the set of $\{\vec{k'}\}$ can be easily enumerated because M_L is a lower triangular matrix:

Code 5.5.1 *Gen_Set*$_{[\vec{k}]}$($M_L, F, \vec{l}_{(p,q)}, \vec{u}_{(p,q)}$):

1. *For* k'_1 *from* L_1 *to* U_1
2. *For* k'_2 *from* L_2 *to* U_2
3. *For* k'_3 *from* L_3 *to* U_3
4. $\vec{k} = F \cdot \vec{k'}$

where

(When $d_{43} = d_{42} = d_{41} = 0$):

$L_1 = \lceil \frac{l_1}{d_{11}} \rceil, U_1 = \lfloor \frac{u_1}{d_{11}} \rfloor$:

$L_2 = \lceil \frac{l_2 - d_{21}k_1}{d_{22}} \rceil, U_2 = \lfloor \frac{u_2 - d_{21}k_1}{d_{22}} \rfloor;$
$L_3 = \lceil \frac{l_3 - d_{31}k_1 - d_{32}k_2}{d_{33}} \rceil, U_3 = \lfloor \frac{u_3 - d_{31}k_1 - d_{32}k_2}{d_{33}} \rfloor;$

(When $d_{43} \neq 0$):

$$L_3 = max(\lceil \frac{l_3 - d_{31}k_1 - d_{32}k_2}{d_{33}} \rceil, \lceil \frac{l_4 - d_{41}k_1 - d_{42}k_2}{d_{43}} \rceil)$$

$$U_3 = min(\lfloor \frac{u_3 - d_{31}k_1 - d_{32}k_2}{d_{33}} \rfloor, \lfloor \frac{u_4 - d_{41}k_1 - d_{42}k_2}{d_{43}} \rfloor)$$

(When $d_{43} = 0, d_{42} \neq 0$):

$$L_2 = max(\lceil \frac{l_2 - d_{21}k_1}{d_{22}} \rceil, \lceil \frac{l_4 - d_{41}k_1}{d_{42}} \rceil)$$

$$U_2 = min(\lfloor \frac{u_2 - d_{21}k_1}{d_{22}} \rfloor, \lfloor \frac{u_4 - d_{41}k_1}{d_{42}} \rfloor)$$

(When $d_{43} = d_{42} = 0, d_{41} \neq 0$):

$$L_1 = max(\lceil \frac{l_1}{d_{11}} \rceil, \lceil \frac{l_4}{d_{41}} \rceil)$$

$$U_1 = min(\lfloor \frac{u_1}{d_{11}} \rfloor, \lfloor \frac{u_4}{d_{41}} \rfloor)$$

$Gen\_Set_{[k]}()$ represents the set of $\vec{k}$ which will be used in processor q to pack the buf_{send} and which will also be used in processor p to reference the message. Once the $Gen\_Set_{[k]}()$ is derived, the corresponding set $\{[u_{1A}\ u_{2A}\ u_{1B}\ u_{2B}]^T\}$ can be expressed by equation (5.49). The set $(\{r_B, c_B\})$ of array B for communication sets buf_{send} can then be expressed by equation (5.45). The corresponding set $(\{r_A, c_A\})$ of the array A can be expressed by equation (5.43). The corresponding set of global loop index $\{i_g\}$ can be expressed by either equation (5.43) or (5.45).

The (r, c) index pair which represents an array element can be replaced by the $\vec{u}_A$ or the $\vec{u}_B$ index. Therefore, the computation of equations (5.43) and (5.45) can be eliminated and the buf_{send} in SPMD code 5.5.1 is expressed as:

$$buf_{send}(q) = \begin{bmatrix} u_{1B} \\ u_{2B} \end{bmatrix} = N_B \cdot \vec{k} \tag{5.58}$$

where $\vec{k} \in Gen\_Set_{[k]}(\ M_L, F, \vec{l}_{(q,p)}, \vec{l}_{(q,p)}\ )$.

The sequence of message packing $(N_B \cdot \vec{k})$ in the sending processor p is generated by $Gen\_Set_{[k]}(\ M_L, F, \vec{l}_{(q,p)}, \vec{l}_{(q,p)}\ )$. In the receiving processor q, the local array A elements with indices $(N_A \cdot \vec{k})$ will access the remote data $(N_B \cdot \vec{k})$ from processor p. The sequence of non-local references to array B in the message is also generated $Gen\_Set_{[k]}(\ M_L, F, \vec{l}_{(q,p)}, \vec{l}_{(q,p)}\ )$. Since both sequences are generated by the same $Gen\_Set_{[k]}(\ M_L, F, \vec{l}_{(q,p)}, \vec{l}_{(q,p)}\ )$, the receiving processor will access the message in the same order as it is packed in the sending processor according to the same $Gen\_Set_{[k]}(\ M_L, F, \vec{l}_{(q,p)}, \vec{l}_{(q,p)}\ )$.

Based on the above derivation, the SPMD code 5.5.1 for the FORALL loop in example 5.5.2 can then be constructed:

Code 5.5.1

```
1.  for p = 0 to n_p - 1 do
2.
3.    /* Symbolic expressions generated */
4.    /* based on equation (5.56) */
5.    M ← symbolic_expression
6.    l_(p,q) ← symbolic_expression
7.    u_(p,q) ← symbolic_expression
8.
9.    /* Apply Gaussian elimination to M  */
10.   /* to obtain M_L, F  */
11.   (M_L, F) ← Gaussian_Elimination(M)
12.
13.   /* Pack&Send Messages to other PEs */
14.    for q ∈ {0 ··· n_p - 1} and q ≠ p do
15.      for k ∈ Gen_Set_[k]( M_L, F, l_(q,p), u_(q,p) )
16.        Append B(N_B · k) to buf_send(q)
17.      end
18.      Send buf_send(q) to processor q if not empty
19.    enddo
20.
```

```
21.     /* Perform local iteration */
22.     for k⃗ ∈ Gen_Set[k]( M_L, F, l⃗(p,p), u⃗(p,p) )
23.         [u⃗_A u⃗_B]^T = N · k⃗
24.         A(u⃗_A) = B(u⃗_B)
25.     end
26.
27.     /* Receive nonlocal data, and */
28.     /* Perform remaining iterations */
29.     while expecting messages
30.         Wait for a message from q
31.         and Store into Buf[q]
32.         s = 0
33.         for k⃗ ∈ Gen_Set[k](M_L, F, l⃗(p,q), u⃗(p,q))
34.             u⃗_A = N_A · k⃗
35.             A(u⃗_A) = Buf[q](s)
36.             s = s + 1
37.         end
38.     end
39.
40. enddo
```

It should be noted that

- For data distribution in cyclic(m) with strided alignment, our algorithm for the SPMD code generation which can be symbolically resolved at compile-time except one 4×3 matrix triangular factorization.

- To keep the message unpacking simple, other methods put all intelligence in the sending side and the messages contain addr-value pairs, which requires run-time resolution for "addr". In contrast, the messages in our method contain only "values" which still keeps unpacking simple.

  Our method keeps the property of the same packing/unpacking order such that the references to the messages are performed by sequentially and directly accessing the message buffers as shown in statements 35 and 36. In order to have the property of the same packing/unpacking order, our method requires only a transformation to the original access pattern, which becomes another access pattern to replace original access pattern by equation (5.50) and equation (5.51).

- Message packing is efficiently performed by statement 15 and 16 without run-time codes for calculation of $owner(A(i))$ and for pre-calculation of *addr-value* pairs in message unpacking.

- The sending of messages can be performed individually for each remote processor as soon as each packing is completed, instead of waiting for all messages for all processors to be packed. This is shown by the fact that sending statement 18 and packing statements 15 to 17 are both inside the same loop.

- The nonlocal iterations can also be performed based on each receiving message, instead of waiting for all messages to be received. This is possible because the nonlocal iterations are split into groups based on the sending processors.

A message is expected to be received from a processor q if the $Gen\_Set_{[k]}(p, q)$ is not empty, which can be easily determined statically and used by statement 29. In order to have a compact rectangular storage, $\vec{u}_A$ and $\vec{u}_B$ can also be transformed into $\vec{u'}_A$ and $\vec{u'}_B$ by equation (5.59) as developed in [8], where $\vec{u'}_A$ and $\vec{u'}_B$ are the indices to the compact rectangular storage area.

$$\vec{u'} = \lfloor \begin{bmatrix} -1 & 0 \\ \frac{\mu \cdot g}{a_{co}} & 1 \end{bmatrix} \cdot \vec{u} \rfloor \tag{5.59}$$

Compared to existing methods [2, 25, 28, 42, 30], the major advantage of our method is that we are able to symbolically derive the basis vectors of the lattice which models the enumeration of communication set. Based on the symbolic basis vectors, our algorithm to generate the SPMD codes requires only a 4×3 matrix triangular factorization at run-time. In the case that arrays involve non-unit stride alignment and cyclic(m) distributed, other methods use *inspector*, and *executor* codes during run-time for building the FSM tables [28], the pattern Tables [2], or for scanning over local elements [25, 42, 30]. Although there is no experimentation, which is not our focus in this paper, our method shows apparent advantage that our *inspector*-like run-time code contains only a 4×3 matrix triangular factorization. This can be seen by statement 11 in our SPMD code 5.5.1.

5.6 Our Smith-Normal-Form Enumeration Method

In this section, we present our Smith-Normal Form technique for the deriving of the communication set enumeration.

Theorem 5.6.1 *(Smith Normal Form)[39]*

Let A and B be integer matrices. We say that B is equivalent to A if $B = U \cdot A \cdot V$ for some unimodular matrices U and V. Every integer matrix A is equivalent to a diagonal matrix (its Smith Normal Form)

$$S = S(A) = diag(s_1, s_2, \ldots, s_r, 0, 0, \ldots, 0)$$

where r is the rank of A, $s_1, s_2, \ldots, s_r$ are nonzero elements, and $s_i | s_{i+1}$, for $1 \leq i \leq (r-1)$
($a|b$ means $b = a \cdot k$, $k \in Z$). ∎

In the following, we illustrate how an implicit-lattice equation (5.60) can be converted into an explicit-lattice equation (5.61) based on the Smith Normal Form.

$$F \cdot \vec{x} = f \tag{5.60}$$

$$\vec{x} = \vec{x_0} + H' \cdot \vec{k}, \quad \vec{k} \in Z^m \tag{5.61}$$

5.6.1 The SNF Explicit-Lattice Transformation

The explicit-lattice equation (5.61) can be seen as the general solution of $\vec{x}$ for (5.60). We now describe the derivation as follows:

Assume F is an $r \times n$ matrix of rank r and $r \leq n$, there are unimodular matrices $U(r \times r)$, and $V(n \times n)$ such that $U \cdot F \cdot V = [S \ \ 0]$ is in the Smith Normal Form, where $S = diag(s_1, s_2, \ldots, s_r)$, and [0] is a zero matrix.

Let $V^{-1} \cdot \vec{x} = \vec{y}$, $U \cdot \vec{f} = \vec{d}$. (5.60) becomes

$$[S \;\; 0] \cdot \vec{y} = \vec{d} \tag{5.62}$$

If $\vec{y} = [y_1, y_2, \cdots, y_n]^T$, $\vec{d} = [d_1, d_2, \cdots, d_r]^T$, then $y_{r+1}, y_{r+2}, \cdots, y_n$ can be any integers, but $y_1, y_2, \cdots, y_r$ must satisfy

$$s_i y_i = d_i \quad , 1 \leq i \leq r \tag{5.63}$$

Provided that (5.63) or (5.62) has an integer solution, the general solution of $\vec{x}$ can be expressed as

$$\vec{x} = V \cdot \vec{y} \tag{5.64}$$

where $y_{r+1}, y_{r+2}, \cdots, y_n \in Z$, $y_1, y_2, \cdots, y_r$ are solutions of equation (5.63).

Let $\vec{y}_{(1:r)} = [y_1, \cdots, y_r]^T$, $\vec{y}_{(r+1:n)} = [y_{r+1}, \cdots, y_n]^T$, V_1 be the matrix that consists of the first r columns of V, V_2 be the matrix that consists of the last $n - r$ columns of V.

Equation (5.64) becomes:

$$\begin{aligned} \vec{x} &= V_1 \cdot \vec{y}_{(1:r)} + V_2 \cdot \vec{y}_{(r+1:n)} \\ &= \vec{x}_0 + H' \cdot \vec{k} \end{aligned} \tag{5.65}$$

where $\vec{x}_0 = V_1 \cdot \vec{y}_{(1:r)}$, $\vec{k} = V_2 \cdot \vec{y}_{(r+1:n)} \in Z^{n-r}$

Thus, we have derived the explicit-lattice equation (5.65) for equation (5.60).

According to $[S \;\; 0] \cdot \vec{y} = \vec{d} = U \cdot \vec{f}$,

$$\vec{f} = U^{-1} \cdot [S \;\; 0] \cdot \vec{y} = U^{-1} \cdot S \cdot \vec{y}_{(1:r)}$$

When we have $\vec{f} = U^{-1} \cdot S \cdot \vec{l}$ where $\vec{l} \in Z^r$, an integer solution of $\vec{y}$ for (5.62) is then possible. $\vec{f} = U^{-1} \cdot S \cdot \vec{l}$ is then the condition for $\vec{x}$ to be an integer vector.

5.6.2 A Special Case of SNF Explicit-Lattice Transformation

We will developed an algorithm to perform the explicit-lattice transformation for $\vec{u}$ $(= [u_1\ u_2\ u_3\ u_4]^T)$ in the following specific form such as equation (5.48):

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = 0 \tag{5.66}$$

Let W be the 1×4 matrix $[w_1\ w_2\ w_3\ w_4]$, $g = gcd(w_1, w_2, w_3, w_4)$. According to the theorem of Smith Normal Form, the 1×4 matrix W can be decomposed into $W = U^{-1} \cdot S \cdot V^{-1}$ or $S = U \cdot W \cdot V$, where $U = [1]$, $S = [g\ 0\ 0\ 0]$ and V is a 4×4 unimodular matrix. By applying the SNF transformation, an explicit-lattice equation for (5.66) can be derived as:

$$\vec{u} = V \cdot \begin{bmatrix} 0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix} \tag{5.67}$$

where k_1, k_2, k_3 are arbitrary integers. We will show how such unimodular matrix V is derived in the followings.

Based on the Euclid's algorithm [30], all following equations can be determined where $gcd(a, b, c, d) = 1$:

$g = a \cdot w_1 + b \cdot w_2 + c \cdot w_3 + d \cdot w_4$

$g_2 = gcd(a, b) = \alpha_2 \cdot a - \beta_2 \cdot b$

$g_3 = gcd(g_2, c) = \alpha_3 \cdot g_2 - \beta_3 \cdot c$

$g = gcd(g_3, d) = \alpha_4 \cdot g_3 - \beta_4 \cdot d$

$a = a' \cdot g_2 = a'' \cdot g_3$

$b = b' \cdot g_2 = b'' \cdot g_3$

$c = c'' \cdot g_3$

According to a theorem of the unimodular matrix (pp.13 [39]), a 4×4 unimodular matrix Q with the first column $= [a\ \ b\ \ c\ \ d]^T$ can be constructed as:

$$Q = \begin{bmatrix} a & \beta_2 & a' \cdot \beta_3 & a'' \cdot \beta_4 \\ b & \alpha_2 & b' \cdot \beta_3 & b'' \cdot \beta_4 \\ c & 0 & \alpha_3 & c'' \cdot \beta_4 \\ d & 0 & 0 & \alpha_4 \end{bmatrix} \tag{5.68}$$

Since $g = gcd(w_1, w_2, w_3, w_4)$, any linear combination of w_1, w_2, w_3, w_4 will be multiple of g:

$W \cdot Q = [\, g \;\; (g \cdot t_2) \;\; (g \cdot t_3) \;\; (g \cdot t_4) \,]$, where

$g \cdot t_2 = w_1 \cdot \beta_2 + w_2 \cdot \alpha_2$

$g \cdot t_3 = w_1 \cdot a' \cdot \beta_3 + w_2 \cdot b' \cdot \beta_3 + w_3 \cdot \alpha_3$

$g \cdot t_4 = w_1 \cdot a'' \cdot \beta_4 + w_2 \cdot b'' \cdot \beta_4 + w_3 \cdot c'' \cdot \beta_4 + w_3 \cdot \alpha_4$

Let

$$E = \begin{bmatrix} 1 & -t2 & -t3 & -t4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We have

$$W \cdot Q \cdot E = [g \;\; 0 \;\; 0 \;\; 0] \tag{5.69}$$

According to SNF decomposition($S = [g \;\; 0 \;\; 0 \;\; 0] = U \cdot W \cdot V = W \cdot V$), and equation (5.69), we now have

$$V = Q \cdot E$$

.

Let M be the 4×3 sub-matrix constructed from the last three columns of V. The explicit-lattice equation (5.67) becomes

$$\vec{u} = V \cdot \begin{bmatrix} 0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix} = N \cdot \vec{k} \tag{5.70}$$

where $\vec{k} = [k_1\ k_2\ k_3]^T \in Z^3$. and

$$N = \begin{bmatrix} -a \cdot t_2 + \beta_2 & -a \cdot t_3 + a' \cdot \beta_3 & -a \cdot t_4 + a'' \cdot \beta_4 \\ -b \cdot t_2 + \alpha_2 & -b \cdot t_3 + b' \cdot \beta_3 & -b \cdot t_4 + b'' \cdot \beta_4 \\ -c \cdot t_2 & -c \cdot t_3 + \alpha_3 & -c \cdot t_4 + c'' \cdot \beta_4 \\ -d \cdot t_2 & -d \cdot t_3 & -d \cdot t_4 + \alpha_4 \end{bmatrix} \tag{5.71}$$

5.6.3 An Example

This example are merely used for showing the conception and correctness of our method. All the calculations can be evaluated at compile time.

Example 5.6.1

1. *real* $A(0:99)$, $B(0:99)$
2. *! processor* $P(0:3)$
3. *! template* T
4. *! distribute* $T(cyclic(10))$ *onto* P
5. *! alignment* $A(k)$ *with* $T(k)$
6. *! alignment* $B(k)$ *with* $T(3*k)$
7. *FORALL* $(i = 0:99)$
8. $\quad A(i) = B(99-i)$

In example 5.6.1, two arrays A, B an array of four processors P, and a template array T are declared. The template array T is cyclic(10)-distributed across the processor array P. Statement-5 and statement-6 specify the alignment relationships between array A, B and the template array T.

The symbolic expressions can be generated at compile-time even if the values are unknown In this specific example, the values of $N, \vec{l}, \vec{u}, M$ can be evaluated at compile time since they are known values:

$$N = \begin{bmatrix} N_A \\ N_B \end{bmatrix} = \begin{bmatrix} -40 & 13 & -40 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \\ -40 & 13 & -39 \end{bmatrix} \quad \vec{l}_{(p,q)} = \begin{bmatrix} 10 \cdot p \\ 0 \\ -297 + 10 \cdot q \\ 0 \end{bmatrix}$$

$$\vec{u}_{(p,q)} = \begin{bmatrix} 9 + 10 \cdot p \\ 99 \\ -288 + 10 \cdot q \\ 99 \end{bmatrix} \quad M = \begin{bmatrix} 40 & -13 & 40 \\ -1600 & 507 & -1560 \\ 0 & -1 & 0 \\ -1600 & 507 & -1560 \end{bmatrix}$$

Triangular factorizing M, we have $(M_L = M \cdot F)$. The factorization of M can also be done at compile time since M is known.

$$M_L = \begin{bmatrix} 1 & 0 & 0 \\ -39 & 40 & 0 \\ -3 & 0 & -40 \\ -39 & 40 & 0 \end{bmatrix} \quad F = \begin{bmatrix} 0 & -1 & 0 \\ 3 & 0 & 40 \\ 1 & 1 & 13 \end{bmatrix}$$

The $Gen\_Set_{[\vec{k}]}()$ is also constructed at compile-time as shown in Code 5.5.1. In the following, we show its equivalent result after substituting the known values:

$Gen\_Set_{[\vec{k}]}(M_L, F, \vec{l}_{(p,q)}, \vec{u}_{(p,q)})$:

1. for k'_1 from $10 * p$ to $9 + 10 * p$
2. for k'_2 from $\frac{39+39*k'_1}{40}$ to $\frac{99+39*k'_1}{40}$
3. for k'_3 from $\frac{327-10*q-3*k'_1}{40}$ to $\frac{297-10*q-3*k'_1}{40}$
4. $\vec{k} = F \cdot \vec{k'}$

According to SPMD code 5.5.1, it results to the following SPMD code derived at compile-time. The values of $(N \cdot F), (N_A \cdot F), (N_B \cdot F)$ in the SPMD code can also be evaluated at compile time, which are shown below in symbolic forms for the clarity purpose.

Code 5.6.1

1. *for* $p = 0$ *to* $n_p - 1$ *do*
2.
3. /* **Pack&Send Messages to other PEs** */
4. *for* $q \in \{0 \cdots n_p - 1\}$ *and* $q \neq p$ *do*
5. for k'_1 from $10 * q$ to $9 + 10 * q$
6. for k'_2 from $\frac{39+39*k'_1}{40}$ to $\frac{99+39*k'_1}{40}$
7. for k'_3 from $\frac{327-10*p-3*k'_1}{40}$ to $\frac{297-10*p-3*k'_1}{40}$
8. Append $B(N_B \cdot F \cdot \vec{k'})$ to $buf_{send}(q)$;
9. *Send* $buf_{send}(q)$ *to processor* q *if not empty*
10. *enddo*
11.
12. /* **Perform local iteration** */
13. for k'_1 from $10 * p$ to $9 + 10 * p$
14. for k'_2 from $\frac{39+39*k'_1}{40}$ to $\frac{99+39*k'_1}{40}$
15. for k'_3 from $\frac{327-10*p-3*k'_1}{40}$ to $\frac{297-10*p-3*k'_1}{40}$
16. $A(N \cdot F \cdot \vec{k'}) = B(N \cdot F \cdot \vec{k'})$;

```
17.
18.    /* Receive nonlocal data, and */
19.    /* Perform remaining iterations */
20.    while expecting messages
21.      Wait for a message from q
22.      and Store into Buf[q]
23.      s = 0
24.       for k'_1 from 10 * p to 9 + 10 * p
25.        for k'_2 from (39+39*k'_1)/40 to (99+39*k'_1)/40
26.         for k'_3 from (327-10*q-3*k'_1)/40 to (297-10*q-3*k'_1)/40
27.              begin
28.              A(N_A · F · k') = Buf[q](s)
29.              s = s + 1
30.              end
31.  enddo
```

s	(k'_1, k'_2, k'_3)	$\vec{u}_A$	$\vec{u}_B$	i	(r_A, c_A)		(r_B, c_B)	
0	(-2,2,7)	(-2,2)	(286,93)	2	(0,2)	A(2)	(7,1)	B(97)
1	(-2,3,7)	(-2,3)	(286,94)	42	(1,2)	A(42)	(4,1)	B(57)
2	(-2,4,7)	(-2,4)	(286,95)	82	(2,2)	A(82)	(1,1)	B(17)
3	(-1,1,7)	(-1,1)	(283,92)	1	(0,1)	A(1)	(7,4)	B(98)
4	(-1,2,7)	(-1,2)	(283,93)	41	(1,1)	A(41)	(4,4)	B(58)
5	(-1,3,7)	(-1,3)	(283,94)	81	(2,1)	A(81)	(1,4)	B(18)
6	(0,0,7)	(0,0)	(280,91)	0	(0,0)	A(0)	(7,7)	B(99)
7	(0,1,7)	(0,1)	(280,92)	40	(1,0)	A(40)	(4,7)	B(59)
8	(0,2,7)	(0,2)	(280,93)	80	(2,0)	A(80)	(1,7)	B(19)

Table 5.1: Packing and Unpacking Sequence for Sending Processor $P(1)$ and Receiving Processor $P(0)$

As an example, assume that the sending processor is P(1) and the receiving processor is P(0). Table 1 shows the corresponding loop index $\vec{k'}$, the local addresses of $\vec{u}_A$, $\vec{u}_B$, the global loop index i, and the local and the global addresses of A and B in the $r-c$ grid. It can be seen that both the sending processor and the receiving processor enumerate identical sets of (k'_1, k'_2, k'_3), and that the receiving processor P(0) can access the nonlocal elements in the receiving buffer $Buf_{[1]}$ directly and sequentially. For this specific case, the message packing for $P(0)$ in the sending processor $P(1)$ and the unpacking of this message in the receiving processor $P(0)$ have the same sequence listed in Table 1.

Chapter 6

Conclusion

To obtain high performance on a distributed-memory machine, the most important factor is to minimize the communication overhead. The communication overhead can be reduced by two different techniques. The first one is the communication reduction technique. This has resulted to the active research on the automatic array partitioning, and the research on the message aggregation used in the distributed-array compilation. The second technique is the communication latency hiding, which is also implemented in most distributed-array compilation systems. This research have demonstrated that our Smith-Normal-From lattice algebra successfully serves as a foundation to develop frameworks for the above three important communication optimizations.

6.1 Distributed Array Compilation

In the topic of the distributed array compilation, we have demonstrated our SNF method for the enumeration of the local set in section 5.6. Further, we have shown our scheme and the corresponding SPMD code for the message packing and the message unpacking in section 5.5.2. Our SNF method exhibits the following significant advantage over existing methods:

- In the case of the general data distribution in cyclic(m) with strided alignment, the major advantage of our method is that we are able to symbolically derive the basis vectors of the lattice which models the enumeration of communication set. Based on the symbolic basis vectors, our algorithm of the SPMD-code

generation requires only a 4×3 matrix triangular factorization as the *inspector*-like run-time codes. When the parameters are known values, the SPMD code can be completely constructed without any *inspector*-like run-time codes.

- It avoids the run-time code for the evaluation of the destination (owner($A(i)$)) in performing message packing. It does not require the run-time code for pre-calculating the unpacking information in performing message unpacking. Our method provides an integrated solution for both message packing and message unpacking problems without incurring such run-time overhead.

- The sending of messages can be performed individually as soon as each packing is completed. A group of nonlocal iterations can be performed as soon as a single message is received.

- In the receiving processors, the message-unpacking is performed by sequential and direct accesses to the message buffers.

6.2 Automatic Array Partitioning

In the topic of the automatic array partitioning, we have developed two techniques of array partitioning based on the periodic skewing equation. We have derived the equation which describes all possible communication-free partitioning for arrays in a DoAll loop, and the equation which describes the mapping equation such that all remote data can be located in one processor. We have shown by an example that the load balance and one block-communication for each PE can be achieved by our method. Periodic skewing equations are of particular interest because they have an elegant foundation in the mathematical theory of lattices described in section 3.

Reference List

[1] A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data parallel programs using closed forms for basic vectors. *Journal of Parallel and Distributed Computing*, 38(2), November 1996.

[2] A. Venkatachar, J. Ramanujam and A. Thirumalai. Communication generation for block-cyclic distributions. *Parallel Processing Letters*, 7(2), June 1997.

[3] J.R. Allen and K.Kennedy. Pfc: A program to convert fortran to parallel form. *Supercomputers: Design and Applications*, pages 186-205, 1984.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. *Symp. on Principles and Pratice of Parallel Programming, Williamsburg, VA*, 1991.

[5] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *Proc of the SIGPLAN '93 Conference on programming language design and implementation*, June 1993.

[6] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *Proc of the SIGPLAN '93 Conference on programming language design and implementation*, June 1993.

[7] P. Budnik and D.J. Kuck. The organization and use of parallel memories. *IEEE Transactions on computers*, December 1975.

[8] C. Ancourt, F. Coelho, F. Irigoin, and R. KerYell. A linear algebra framework for static high performance fortran code distribution. *Scientific Programming*, 6(1), Spring 1997.

[9] C. Ancourt, F. Irigoin, F. Coelho and R. Keryell. A linear algebra framework for static hpf code distribution. *Tech Report A-278-CRI, Ecole des Mines, Paris*, Nov 1995.

[10] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2(2), Oct 1988.

[11] P. Crooks and R.H. Perrott. Language constructs for data partitioning and distribution. *TR report Department of Computer Science*, The Queen's University of Belfast 1993.

[12] Erik H. D'Hollander. Partitioning and labeling of index sets in do loops with constant dependence vectors. *Proceedings of the 1989 international conference on parallel processing*, II, August 1989.

[13] E.M. Paalvast, H Sips, and L. Breebaart. A method for parallel programs generation with an application to the booster language. *Proceedings Int'l Conf on Supercomputing*, June 1990.

[14] E.M. Paalvast, H.J. Sips and A. van Gemund. Automatic parallel program generation and optimazation from data decompositions. *Proceedings Int'l Conf on Parallel Processing*, Aug 1991.

[15] E.S. Daniel, J. Palermo and P. Banerjee. Processor tagged descriptors: A data structure for compiling for distributed-memory multicomputers. *Conference on Parallel Architecture and Compilation Techniques*, Aug, Montreal 1994.

[16] C.D.Polychronopoulos et al. Parafrase-2 : An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *Proc. Int'l Conf. on Parallel Processing*, pages 39-48, August 1989.

[17] F. Irigoin et al. A linear algebra framework for static hpf code distribution. *4th International Workshop on Compilers for Parallel Computers*, 1993.

[18] M. Le Fur. Scanning parametrized polyhedron using fourier-motzkin elimination. *Publiction Interne 858, IRISA*, Sep. 1994.

[19] C. Germain and F. Delaplace. Automatic vectorization of communications for data-parallel programs. *EURO PAR*, June 1995.

[20] M. Gerndit. Array distribution in superb. *Proceedings of Third Int'l Conf on Supercomputing*, June 1989.

[21] H.J. Sips, K. van Reeuwijk and W. Denissen. Analysis of local enumeration and storage schemes in hpf. *Int'l Conf on Supercomputing. ACM press*, 1996.

[22] HPF Specification. *http://www.erc.msstate.edu/hpff/report.html.*

[23] J.-F. Collard and T. Risset. Construction of do loops from systems of affine constraints. *Parallel Processing Letter*, 5(3), 1995.

[24] J.M. Stichnoth, D. O'Hallaron and T.R. Gross. Generating communication for array statements: Design implementation and evaluation. *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[25] J.M. Stichnoth, D. O'Hallaron and T.R. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21, 1994.

[26] J.M. Stichnoth, D. O'Hallaron and T.R. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21, 1994(5).

[27] K. Kennedy, N. Nedeljkovic and A. Sethi. Efficient address generation for block-cyclic distribution. *Proceedings Int'l Conf on Supercomputing*, June 1995.

[28] K. Kennedy, N. Nedeljkovic, and A. Sethi. Communication generation for cyclic(k) distributions. *Languages, Compilers, and Run-Time Systems for Scalable Computers, editors: B. Szymanski and B. Sinharoy*, 1996.

[29] K. Kennedy, N. Nedeljkovic and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. *ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming*, Santa Barbara, CA. July 1995.

[30] K. van Reeuwijk, W. D.H. Sips and E.M. Paalvast. An implementation framework for hpf distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9), Sep 1996.

[31] S. Hiranandani K. Kennedy and C.W. Tseng. Compiler support for machine-independent parallel programming in fortran d. In *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, 1992. Elsevier Science Publishers B.V.

[32] C. Koelbel. Compile-time generation of regular communication pattern. *Supercomputing*, 1991.

[33] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[34] D.J. Kuck. Illiac iv software and application programming. *IEEE Transactions on Computers*, C-17, 1968.

[35] D.H. Lawrie and C.R. Vora. The prime memory system for array access. *IEEE Transactions on computers*, c-31(5), May 1982.

[36] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. *IEEE*, 1990.

[37] A. Lim and M. Lam. Communication-free parallelization via affine transformations. *Proc. 7th workshop on languages and compilers for parallel computing*, 1994.

[38] Mace. *Memory storage pattern in parallel processing*. Kluwer academic, 1987.

[39] M. Newman. *Integral Matrices*. New York Academic Press, 1972.

[40] J. Ramanujam and P. Sadayappan. Compiler-time techniques for data distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[41] S. Bankner, P. Brezany and H. Zima. Processing array statements and procedure interfaces in the prepare high performance fortran compiler. *5th International Conference on Compiler Construction*, April 1994.

[42] S. Chatterjee, J.R. Gilbert, F. Long, R. Schreiber and S.H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1), April 1995.

[43] S. Gupta, S. Kaushik, S. Mufti, S. Sharma, C.H. Huang and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. *Proceedings of the 1993 International Conference on Parallel Processing*, 2, 1993.

[44] S. Hiranandani, K. Kennedy and C.W. Tseng. Compiler support for machine-independent parallel programming in fortran d. In *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, 1992. Elsevier Science Publishers B.V.

[45] S. Hiranandani, K. Kennedy, J. Mellor-Crummey and A. Sethi. Compilation techniques for block-cyclic distributions. *Proceedings of International Conference on Supercomputing*, July 11-15 Manchester 1994.

[46] Sarkar V. and Gao G.R. Optimization of array accesses by collective loop transformations. *ACM Int'l Conf on Supercomputing.*, 1991.

[47] S.D. Kaushik, C.H. Huang, and P. Sadayappan. Compiling array statements for efficient execution on distributed memory machines: Two-level mappings. *Proceedings Eighth Ann. Workshop Languages and Compilers for Parallel Computing*, Aug 1995.

[48] H.D. Shapiro. Theoretical limitations on the efficient use of parallel memories. *IEEE Transactions on computers*, c-27(5), May 1978.

[49] Sinharoy B. and Szymanski B. Data and task alignment in distributed memory architectures. *Journal of Parallel and Distributed Computing*, 21, 1994.

[50] HPF Specification. *http://www.erc.msstate.edu/hpff/report.html.*

[51] D. Bau I. Kodukula V. Kotlyar K. Pingali P. Stodghill. Solving alignment using elementary linear algebra. *Proc. 7th workshop on languages and compilers for parallel computing*, 1994.

[52] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. *Proceedings Eighth Ann. Workshop Languages and Compilers for Parallel Computing*, 1995.

[53] C.-W. Tseng. *An optimizing Fortran D compiler for MIMD Distributed-Memory Machines*. PhD Thesis, Rice University, 1993.

[54] H.-Y. Tseng and J.-L. Gaudiot. A compiler strategy for generating efficient communication based on array distribution directives. *Third International Conference on Computer Science and Informatics*. Research Triangle Park, NC, March 1997.

[55] H.A.G. Wijshoff and J.V. Leeuwen. The structure of periodic storage schemes for parallel memories. *IEEE Transactions on computers*, c-34(6), June 1985.

[56] Xu H. and Ni L. Optimazing data decompositions for data parallel programs. *Proceedings Int'l Conf on Parallel Processing*, Aug 1991.