Department of Computer Science
University of Manchester
Manchester M13 9PL, England

Technical Report Series
UMCS–93–1–6

Michael F.P. O'Boyle

Program and Data Transformations for Efficient Execution on Distributed Memory Architectures

Program and Data Transformations for Efficient Execution on Distributed Memory Architectures[1]

Michael F.P. O'Boyle

Department of Computer Science
University of Manchester
Oxford Rd., Manchester, U.K.
`mob@cs.man.ac.uk`

January 1992

Abstract

This report is concerned with the efficient execution of array computation on Distributed Memory Architectures by applying compiler-directed program and data transformations. By translating a sub-set of a single-assignment language, Sisal, into a linear algebraic framework it is possible to transform a program so as to reduce load imbalance and non-local memory access. A new test is presented which allows the construction of transformations to reduce load imbalance. By a new expression of data alignment, transformations to reduce non-local access are derived. A new pre-fetching procedure, which prevents redundant non-local accesses, is presented and forms the basis of a new data partitioning methodology. By applying these transformations in a straightforward manner to some well known scientific programs, it is shown that this approach is competitive with hand-crafted methods.

Preface

The author graduated from Aston University in 1987 with an upper second B.Sc.(Hons.) in Computational Mathematics and Computer Science with Artificial Intelligence. After working in Industry for a year, he joined the Mapping Project at the University of Manchester in 1988 to pursue his postgraduate studies. In May 1990 he was awarded an M.Sc. by research in Computer Science. He continued to work towards his Ph.D which was awarded in July 1992. This technical report is an amended version of the thesis. He is currently employed as a SERC Postdoctoral Research Fellow at the University of Manchester.

Acknowledgements

I would like to thank my supervisor John Gurd for his guidance and patience over the last two years.

Thanks a lot to Gholam Hedayat for his ideas and enthusiasm. Much of this work came out of our joint discussion, over the past two years. Thanks also for reading endless drafts of this thesis and helping with the proofs.

Thanks a lot to Matt for reading too many drafts, enduring the maths and taming the commas.

Thanks a lot to Stewart for also reading drafts of this thesis.

Thanks also to my Room 2.126 companions Allison and Alfred.

Lots and lots of thanks to **Lizzy**.

Special thanks to my Mum, Dad and sister Kim.

Thanks a lot to my friends and climbing companions who have kept me sane: Annie, Adam, Andrew, Cath, Deidre, Ian, John, Kevin, Judy, Louise, Jane, Mark, Richard, Simon, Stewart, Mark and Vince.

Glossary

a	alignment vector
b	integer constraint vector
e_k	*k*th row of Identity
$\overleftarrow{j}$	outer strip-mined iterator
$\overrightarrow{j}$	inner strip-mined iterator
l	nested loop lower bound vector
na_z	non-local access in processor z
p	number of processors
u	nested loop upper bound vector
z	processor number
A	integer constraint matrix
$E_{k,j}$	identity matrix with rows k and j interchanged
H	hit function
I	identity matrix
J	iterator vector
$\hat{J}$	outermost iteration vector
$\check{J}$	invariant iteration vector
$\tilde{J}$	scalar expanded iteration vector
L	nested loop lower bound matrix
N_x	number of rows of matrix x
O	zero matrix
P	processor space
Q	computation set
$S_{\mathcal{X},\mathcal{Y}}$	hyperplane where $\mathcal{X} = \mathcal{Y}$
T	linear transformation matrix
U	nested loop upper bound matrix
$\mathcal{A}$	alignment matrix
$\mathcal{C}_v$	subscript matrix for c-occurrence of variable v
$\mathcal{E}$	nested loop extra constraint matrix

$\mathcal{J}$	iterator type vector
$\mathcal{M}$	innermost iteration vector
$\mathcal{N}(\mathcal{X})$	null space of $\mathcal{X}$
$\mathcal{P}$	projection matrix
$\mathcal{R}(\mathcal{X})$	range space of $\mathcal{X}$
$\mathcal{S}$	serialising matrix
$\mathcal{T}$	interchange matrix
$\mathcal{U}_{r.v}$	subscript matrix for u-occurrence of variable v
$\mathcal{U}_{v}$	set of all $(\mathcal{U}_{r.v}, \nu_{r.v})$ matrix-vector pairs of variable v
$\mathcal{U}$	union of all $\mathcal{U}_v$ sets
$\mathcal{V}$	occurrence matrix
$\mathcal{X}$	occurrence matrix
$\overline{\mathcal{X}}$	partitioned occurrence matrix
$\underline{\mathcal{X}}$	serialised occurrence matrix
$\mathcal{Y}$	occurrence matrix
$\lvert x \rvert$	cardinality of variable x
$\lvert\lvert [x_1, \ldots, x_m]^T \rvert\rvert = \prod_{i=1}^{m}(\max x_i - \min x_i + 1)$	size of vector x
λ	lower bound index vector
ε	nested loop extra constraint vector
$\nu_{r.v}$	subscript vector for u-occurrence of variable v
π	transformation
σco_z	load imbalance in processor z
θ_v	number of u-occurrences of variable v
υ	upper bound index vector
ζ	one to many transformation
Γ	multiple hit function
$Latt(A.b)$	lattice of points denoting iteration space
#	interleave function
##	inverse of interleave function

Superscripts on matrices or vectors indicate dimension rather than exponentiation, unless otherwise stated.

Chapter 1

Introduction

The exploitation of parallelism to achieve faster computation has been a subject of research for many years [HOCK88]. As the physical limit of sequential processor performance is approached, more attention is focused on designing parallel hardware, where more than one task may be performed simultaneously.

Using parallelism to achieve increased performance has two major difficulties. Firstly, the discovery of appropriate parallelism, if it exists, in the input language will require a certain amount of program analysis [WOLF89]. Secondly once the parallel activities have been identified, they must be organised so as to efficiently utilise any parallel hardware.

Initial attempts at exploiting parallelism were relatively modest. In the CDC6600 [THOR70], up to 10 operations could be performed in parallel, using multiple function units. Multiple function units are still used today to provide parallelism, a good example being the superscalar Intel i860 [MARG90]. A more ambitious approach has been the the construction of computers with multiple processors, such that the parallel work is spread throughout the machine [HOCK88].

The first section of this chapter gives a brief description of existing parallel machines and provides motivation for research into compiling for distributed memory architectures (DMAs). The second section briefly describes background material specific to compiling for DMAs. It first describes the form of parallelism, data parallelism, investigated in this thesis and then gives an outline of some of the languages used for programming DMAs and a method for the implementation of such languages. The third section covers the main implementation issues in compiling for DMAs in more detail, which include parallelism detection, load balancing, communication overhead, scalability and memory coherence. The fourth section gives a brief summary of existing related work which consists of shared memory program transformations, existing DMA implementations and automated translation schemes. This chapter is concluded with an outline of the remainder of the thesis.

1.1 Overview

1.1.1 Parallel Machines

There have been many attempts to classify computer architectures [SHOR73] and [KUCK77], so only a brief overview, based on Flynn's taxonomy [FLYN72], will be given. Flynn classifies architectures by the number of instruction and data streams they possess. The two parallel classifications, which describe present day computers, are SIMD (single instruction multiple data) and MIMD(multiple instruction multiple data).

In SIMD architectures, one instruction is applied to multiple data elements. This group includes vector processors[HOCK88] such as the CRAY-1 [RUSS78], which has special instructions for handling vectors, and array processors such as the DAP [REDD73] and CM-1 [HILL85]. Array processors have one control unit and multiple processors which apply the same instruction to their own local data which operates in a lock-step manner.

The MIMD classification includes a diverse range of computers including reduction machines, dataflow machines and shared memory and distributed memory multiprocessors. Reduction machines such as GRIP [CLAC86] and Flagship [WATS88], employ graph reduction, where computation operates in a demand driven manner upon a set of program functions while dataflow machines, by contrast, execute in a data-driven manner. A discussion of the relative merits of these architectures is beyond the scope of this thesis; for a discussion of dataflow refer to [GURD85] and for graph reduction refer to [WATS88]

At present there are many commercially available shared memory multiprocessors e.g. Sequent Balance [SEQU87] and the Alliant [ALLI90]. These machines consist of von Neumann processors which may have a small amount of local memory, or cache, but all share the same physical main memory to which they are all connected by an interconnection network or bus. While these machines have been very successful there is an architectural limit on the performance available. As the number of processors is increased, the processor to memory bandwidth becomes saturated and therefore these architectures are not scalable.

Several commercially available distributed memory architectures are in existence. All have the advantage of scalability but are generally more difficult to program than shared memory architectures. Typical machines of this class are the Intel hypercube [INTE90], the Meiko Computing Surface [MEIK87] and the WARP [ANNA86]. Such machines consist of several processor-memory pairs that are interconnected by a network. Each processor is von Neumann in design having its own program, data and program counter. As the number of processors increases so does the processor to memory bandwidth, or interconnect, which avoids the architectural bottleneck of shared memory machines. As there is no physically shared memory, memory access time is non-uniform so data that resides in a processor's own memory will take less time to access than data that is remote. However it is precisely this lack of shared memory and non-uniform access time that has made programming and compiling for distributed memory architectures so difficult.

1.1.2 Motivation

While there has been a steady increase in the power of parallel machines, the software to exploit them has lagged behind. When the CRAY-1 was first launched in the mid 1970s, the compiler's vectorising ability was poor and today if a programmer wishes to exploit the parallelism of, say, the Intel Hypercube [INTE90], much of the work must be done by hand. This thesis is concerned with the compilation of a high level language to produce efficient implementations on DMAs.

The languages used to program DMAs have been, traditionally, imperative with message passing constructs added.

In such languages, it is the programmer's responsibility to decompose the program and data into processes and data partitions. This error-prone and time-consuming procedure must then be repeated for each new target machine.

The major barrier to wide acceptance of distributed memory computing is the primitive state of compiler technology. If DMAs are to be successful then the programming effort must be reduced. This thesis describes compilation transformations which allow a programmer to write a program, in a high level language, without regard to the target architecture such that, after compilation, the program may be executed efficiently on DMAs.

This thesis shows that it is possible, for a large class of problems, to achieve the automatic mapping of programs written in an array orientated language to a parallel distributed memory architecture which is competitive with hand-crafted methods. This mapping is based upon analysis and transformations so as to minimise execution time. By using a simple architectural model, it is possible to compare the results acquired by using the transformations developed in this thesis with existing hand-coded methods.

1.2 Compiling for DMAs

1.2.1 Data Parallelism

A large amount of research into compilers for distributed memory architectures has been concerned with the exploitation of data parallelism where the elements of an array may be evaluated in parallel. This approach has been widely used in hand-written implementations [FOX86] and has been shown to be very effective . There is a strong relationship between data parallelism and loop parallelisation which is often used when compiling for shared memory machines [PADU86]. Arrays are manipulated by loops and, conversely loops create arrays and so the parallel evaluation of one implies the parallel evaluation of the other. This duality allows transformations, which have been originally developed for shared memory machines, to be used, in a modified manner, in compiling for DMAs.

Although this approach is by definition limited and exploits program parallelism to a much lesser degree than, say, dataflow [GURD85], there exist a very large number of scientific programs where program parallelism of this nature is much greater than machine parallelism and therefore this approach is more than adequate.

1.2.2 Languages

This section presents a brief summary of languages that have been used for programming DMAs. The summary is restricted to languages which contain the array data structure, allowing the expression of data parallelism These programming languages can be split in to four main groups.

1. explicit process / explicit distributed memory e.g. occam, csp, cstools
2. explicit process / implicit distributed memory e.g. MultiLisp
3. implicit process / explicit distributed memory e.g. FORTRAN D, Kali
4. implicit process / implicit distributed memory e.g. FORTRAN 90, Crystal

This classification is based on just two criteria which are thought to be relevant in compiling for DMAs. For a more comprehensive survey of languages see [SARK89].

The first group illustrates one extreme of the programmer/compiler trade-off. Implementations using these languages have produced good performance as the compiler has little analysis or work to perform. However this is at the expense of large programmer effort. Programming in occam requires the programmer to express any parallelism within the problem as a group of communicating sequential processes, each with its own local data. After determining the message passing between processes, the processes have to be scheduled to processors. New problems now arise, such as non-determinacy and deadlock, which have to be considered along-side performance issues.

The second group include languages such as MultiLisp [HALS86]. Although several implementations of this class of languages exist on shared memory machines, no well known implementations are available on DMAs. This is probably due to their explicit shared memory model, which is difficult to map to a distributed memory space.

The majority of recent research has been focused on the third group of languages. The user is required to determine the distribution of array data in a particular program by the embedding of pragmas. Although explicit process decomposition is not always necessary, usually the marking of loops that may be executed in parallel is required. In section 1.4.2 there is a review of such languages and their implementations.

With languages in the final category, the programmer simply writes a program without regard to machine architecture, let alone how the data and computation are to be sub-divided and scheduled for DMAs. The primary concern is writing a program with enough inherent parallelism. Informally this is governed by the algorithm used and the data dependencies within the program.

This thesis is concerned with the compilation of SISAL [MCGR85], a language which belongs to the fourth group. In chapter 2, the important features of SISAL are detailed however the advantages of writing in a high-level architecture-independent language as opposed to a machine specific one are obvious. The cost, however, is that the compiler must now perform the necessary analysis to map the program to the particular architecture in such a manner as to be competitive with hand-crafted methods. Until recently this seemed to be very difficult for DMAs. This thesis proposes a methodology for efficient mapping of SISAL, an architecture independent language, to DMAs.

SISAL is a first-order functional language supporting array structures, where computation is performed by expression evaluation. As SISAL has no notion of process, it is the responsibility of the compiler to map the program to a particular parallel architecture and not the programmer.

SISAL has been chosen not only from the point of view of ease of programming but also because problems encountered whilst compiling imperative languages for parallel machines, such as aliasing, are absent due to the single assignment semantics. In fact in [WOLF89], it is stated that having a very high level language makes things easier for an optimising compiler, as it has more freedom in implementation.

1.2.3 SPMD Computation

Once any data parallelism has been detected by the compiler, a method to exploit this parallelism on a DMA must be used. The method of evaluation has invariably been single process multiple data, SPMD [DARE88], where the array data is distributed across the processors so that each processor works upon a separate section of the array in parallel. Essentially each process runs the same program but operates on different data which are independent portions of an array. Each processor has one process and its own, statically assigned, local data. In this thesis, two useful methods of exploiting parallelism are considered.

- **Creation Parallelism** Each process accesses any data required for the calculation of its local portion of a distributed array. After accessing the necessary data, each local array is calculated in parallel.
- **Reduction Parallelism** Each process first performs computation upon its own local data in parallel which is then accessed by another processor.

Creation parallelism can be viewed as accessing data before computation is performed at the site where the write will take place. In contrast, reduction parallelism implies that every processor performs part of the total computation on its own local data in parallel before its results are accessed by the processor which will perform the write. Some systems such as [GERN89] only consider creation parallelism, while others rely on reduction parallelism, e.g. [TSEN89].

1.3 Implementation Issues

1.3.1 Parallelism and Overheads

There has been a large amount of research into the detection of program parallelism and in the review section 1.4.1 an overview of such work is given. The identification of data parallelism requires the detection of arrays whose elements may be evaluated in parallel. The easiest way to achieve this is to ensure that there is no data dependency between two elements of the same array which may be manifested as a loop carried data dependency. In Sisal, the **for** loop guarantees that there are no such dependencies and for this reason it forms the basis of exploitation of parallelism in this thesis. All arrays that are generated by a **for** loop may be distributed across the DMA's processors and evaluated in parallel. A pre-processor using parallelising transforms as described in [LU90] and [PUGH91], would be required to make the results of this thesis applicable to existing imperative languages.

In shared memory machines, it is usual to partition the program graph into processes. This may be viewed as a procedure, usually performed at compile time, where the graph is chopped into a number of sub graphs which are called processes. These processes are then scheduled to physical processors either at compile or run-time. However, for DMAs it has been more usual to partition and schedule the data (array) to physical processors at compile time.

A completely static, compile time, approach to the mapping of data and computation to processors is used in this thesis. This is not only to reduce the complexity of the problem but also to demonstrate that such an approach can be successful. Run-time methods have been particularly popular for balancing the work load [SARG86] but, in chapter 3, it is shown that it is possible to determine at compile time what the work load would be given a particular partitioning. While such analysis, in general, cannot work in the case of compile time unknowns, there are many programs [SHEN90], where this is not the case. At the heart of this static approach is the fact that a compiler knows more about the program, than a run-time system can. Armed with this knowledge, it can transform the program into a form which will run efficiently without incurring any run-time overheads. To aid this static approach, dynamic arrays and recursion are not addressed in this thesis and it is assumed that array sizes are known at compile time. Finally on this point, Sarkar [SARK89], in his concluding chapter, compares static and dynamic approaches to compiling Sisal for a shared memory multi-processor. In all but one of his examples, the compile time method gave better results.

Using a static SPMD model of computation implies a coarse-grain approach, as there is only one process per processor. The advantage of this scheme is that no context-switching overhead is incurred. This, however, prevents the hiding of memory latency by switching out a process that has made a non-local access and executing one which is ready to run. By making a memory access ahead of time it is possible, by pre-fetching, to provide latency tolerance even with a coarse granularity. Latency was considered a critical issue in [ARVI87] but it is not considered in this thesis as it has been extensively covered in Rogers' thesis [ROGE91] in the context of compiling for DMAs.

Once the program parallelism has been identified, it is the task of the compiler to transform the program so as to fully utilise the available machine parallelism and to reduce any overheads. In this thesis it is assumed that the problem sizes are much larger than the number of processors and that the amount of program data parallelism is greater than the available machine parallelism. Therefore, as it is relatively easy to utilise the available machine parallelism, then the compiler must focus its attention on finding a mapping that minimises overhead. The major overheads are load imbalance and non-local access. Load imbalance occurs when work is unevenly distributed across the processors so that some processes take longer to execute than others. A compiler should employ a scheme whereby the work load is as evenly distributed as is possible. The other major overhead, non-local access, occurs when there exists a data dependence between two items of data on separate processors. Non-local access, often referred to as communication overhead, should be minimised by a compiler for DMAs. However these objectives conflict and generally there exists a trade-off between them.

1.3.2 Parallelism v Non-Local Access v Load Balance

By distributing the computation and data over several processors, the execution time may be reduced. As the number of processors utilised increases, each processor will perform less work and hence the execution time should decrease. However as the number of processors increases, the amount of data local to a processor will decrease and hence the number of non-local accesses will increase. This will manifest itself as communication overhead and tends to increase the execution time. This is the well known trade-off between locality and parallelism [PEYT86]. It is reasonable to adopt the strategy where, firstly, a compiler must use all the processors and, secondly, determine mappings to reduce non-local access. It is possible that for some programs, a more efficient implementation will take place if not all the processors are used. This is not considered in this thesis.

Less immediately apparent is the trade-off between load balancing and non-local access. One method of balancing the work load is to randomly distribute the data and, hence, computation across the processors. This can have the effect of decreasing the overhead due to load imbalance but it also destroys any spatial and temporal locality within the program. This will have the effect of increasing the number of non-local accesses. It is not clear how to reconcile this conflict as it will depend on the particular program concerned and the relative cost of communication and computation for a particular machine.

1.3.3 Distributed Memory

One of the major reasons for investigating DMAs is their potential scalable performance and if this potential is to be realised, all implementations should also be scalable. The major impact on compilation is that array data should not be replicated across the processor. If replication is used then as the problem size and number of processors grow linearly, the amount of memory needed grows quadratically.

Within this thesis a global address space is assumed. Machines such as EDS [HAYW90] and KSR support this at the hardware level. In other machines, such as the Meiko and Intel Hypercube where each processor has its own local address space, each memory reference must be translated. Successful schemes for message based machines have been described by [RUHL90] and [ROGE91]. Implementing a global address space on message passing machines was one of the first issues tackled by researchers in this area [CALL88].

One of the benefits of considering a single-assignment language is that a data item may only be written to once. In imperative languages such as FORTRAN or Pascal, a variable may be overwritten many times. This is a real problem in parallel machines as the most recent value of a variable has to be maintained. This problem of *memory coherence* has normally been solved by the use of expensive hardware mechanisms. The Sequent Symmetry, for instance, has a snoopy-bus which monitors the memory addresses to see if a local copy of a variable has been updated. In [LI89], several methods for ensuring memory coherence in a distributed memory machine are discussed. SISAL does not require any memory coherence mechanisms to ensure a correct implementation.

1.4 Review of Related Work

The first part of this review section surveys the transformations currently in use. Most were originally developed for vector and shared memory multiprocessors but are beginning to be used in the context of DMAs. The second part addresses compiling for DMAs and describes systems where the user has added some pragmas directing the compiler how to map the data and computation to the processors. The compiler then applies simple transformations or mappings to produce a local program for each processor.

The third part describes those few systems where an attempt to automate the whole process with no user intervention is attempted. In these systems much analysis is required before appropriate transformations may be applied.

1.4.1 Analysis and Transformations

There has been much work published on program restructuring to discover parallelism. The major constraint is data dependence. Determining whether two occurrences of a particular array reference the same data element and hence form a data dependence is non-trivial. Over the years, successively more accurate tests have been derived which is important as all program transformations based upon loop restructuring rely upon dependency analysis. A transform is legal only if data dependency is preserved. In [WOLF89] a good survey of classical techniques is given. In [GOFF91] the application of such analysis in a mature compiler is described and recently, increasingly more sophisticated analysis has become available [LI90b], [MAYD91] and [LU90]. The relationship between data dependency and restructuring is described in [WOLF90b]. A particularly interesting technique for analysing data dependence, by describing access regions, is given in [BALA89]. This data access summary, rather than giving a yes/no decision of existence, can be used for purposes other than determining data dependency such as determining a caching strategy.

Traditionally, program transformation research has largely focused upon two areas, parallelism detection and memory management. Some of the earliest work to uncover parallelism can be found in [LAMP74]. A good survey of such transforms is given in [ALLE86], [PADU86] and [WOLF91] which describe loop interchange, reversal, etc. Recently tools have been designed to allow the user to apply program transformations in an interactive manner to determine the parallel forms [WOLF90d]. These transformations are described using a functional notation. However, recently, some researchers have looked to unimodular matrix transformations as a more elegant approach [DOWL90], [BANJ90], but there exist legal loop transformations which are not unimodular and thus this approach is limited.

A major difficulty in applying transformations is determining in which order to apply them. In [WHIT90] and

[WOLF90c], this issue is discussed, but in general the relative merit of each transformation depends on the target architecture involved and in which phase of the compiler the transformations can take place.

Because of the problems of ordering in *ad hoc* approaches, there have been recent moves towards a unification of transformations. In both [LU90] and [PUGH91] a methodology to determine the maximum parallelism within a particular loop structure is given. Rather than applying transformations in a piece-wise fashion, a schedule is determined which maximises parallelism. The major drawback of this scheme is its computational complexity but by restricting the number of schedules considered, it is possible to determine the best form for many reasonable programs.

Although many transformations have been aimed at detecting parallelism, there has been a large amount of research in finding transforms to use local memory or cache more effectively. In [CARR89], the use of strip-mining and loop interchange is discussed so as to maximise accesses to local cache rather than main memory in a hierarchical memory. In this paper, blocking methods, as studied by numerical analysts, are used to minimise communication to main memory. Although a uni-processor is assumed, it is apparent that the same arguments apply to multi-processor systems.

In [GANN88] the concept of a reference window is introduced. By transforming the program such that local data is used by all that need it, efficiency may be improved. In [WOLF91a], further work in this area is detailed and an algorithm to transform a restricted set of programs to increase locality is given. In [LAM91] there are presented experiments which verify the usefulness of such an approach. As well as increasing access to local memory, transformations to hide memory latency for non-local memory have been addressed. In [GORN90] and [CALL91], prefetching of data before it is required helps to hide latency.

1.4.2 Pragma based DMA Implementations

The state of research in compiling for DMAs is much less advanced than for shared memory machines. Firstly those loops that may be performed in parallel are denoted as such by the programmer using special constructs such as DOALL. Once the compiler knows which regions may be executed in parallel, it has to decide how to decompose the data over the processor. The decomposition of data is often crucial to the performance of a program on DMAs. It eventually determines the amount of communication required, the work distribution and hence load balancing. This is directed by the programmer and can be broken into two parts, alignment and distribution. Alignment is concerned with the relative orientation of arrays stored in distributed memory and is largely independent of machine considerations. If two arrays should be stored transposed relative to one another, this is independent of the number of processors available or the underlying interconnection network. The distribution of arrays is concerned with the partitioning of arrays into rows, columns, blocks etc. and the mapping to processors, whether it be in a folded or interleaved manner.

Once the data distribution has been given, the compiler has to determine what computation is to be performed on each processor. Either this is given by the programmer or it may be derived by some form of loop elimination. Once the data and computation for each processor is known, then the compiler inserts the necessary sends and receives. This insertion of message passing code has been the main focus of attention in the schemes outlined below.

The first significant description of this approach is given in [CALL88]. In this paper the authors derive an efficient message passing program from a sequential shared memory program annotated with directions on how elements of shared arrays are decomposed and distributed to processors. In particular all communications are managed by the compiler. The user defines how the data is to be broken up or decomposed. Such decompositions are restricted to rows or blocks etc. which are mapped to a virtual array. The user then defines how the virtual array is be distributed or mapped to an array of virtual processors. They allow folded or wrapped mappings. Optimisations for compiling such programs are given which include strip-mining, loop interchange and loop elimination. They concentrate on the issue of inserting message passing code and describe a technique where redundant messages can be removed. The major difficulty with this paper is its applicability to problems other than the matrix multiplication example given. There is no systematic description of why, how or when a transformation should take place. Nevertheless this approach spawned several projects, wherein the mapping of data and computation is directed by pragmas.

The programmer usually defines the partitioning of an array and its allocation, usually with respect to virtual processors. Some systems allow alignment constructs, while for others each distribution has to be given. Such systems include Pandore [ANDR90a] [ANDR90b], Oxygen [RUHL90], Kali [KOEL90],[KOEL91], AL [TSEN89] and Superb [GERN89], [GERN91]. In the following brief descriptions an indication of the form of pragmas required will be given. Some are at a more developed stage than others.

No description of the Pandore implementation is given in [ANDR90a] and an excessive amount of user pragmas are needed, some of which seem redundant. Here the user must define parallel loops and indicate that an array is to be distributed. The user also has to define how an array is to be partitioned, and how it is to be mapped on to a virtual processor domain. The Oxygen project is more comprehensive. Here pragmas are added to FORTRAN and a working system is available. But once again the programmer must insert a wide range of pragmas as in Pandore. Its main drawback is the run-time determination of communication, therefore the programmer is required to insert some communication primitives in order to give an efficient implementation. Much of this can be discovered at compile time, reducing overhead, and they are presently looking at such a scheme.

Kali is a functional language annotated with partitioning and virtual processor directives. They have recently concentrated on generating efficient code in the prescence of compile time unknown dependencies. They store such patterns at compile time if they will be repeated rather than recomputing them. Saltz et. al. take a similar approach [SALT90]. In the DINO [ROSI90] project, a similar approach to Pandore is taken. A particular feature of DINO is that by annotating assignments with a “#”, the programmer indicates that non-local data will be required.

The AL compiler for the WARP is more sophisticated in that it requires less user pragmas to perform an effective mapping. The user does not have to define the data over a virtual processor space or define the exact data mapping. Instead those arrays to be distributed are marked by a key word and the compiler determines whether they should be folded or interleaved. However the class of loops and array references are restrictive and the programmer must add some information on reference patterns in the form of data relation pragmas.

Other implementations include [BABE90] and the Booster language [PAAL90] . In the latter the data partitioning is specified by a functional description which is separate from the imperative program. In Booster, it is intended that the user experiment with different data mappings. This idea has been extended in [BALA91] where the environment tries to determine the suitability of such a mapping.

One of the most comprehensive and advanced approaches is given in [ROGE91], and [PING90] where the whole compilation process is described. Once the data partition has been described by the user, an optimising compiler maps the program to the Intel Hypercube to give a performance comparable with hand written methods. What is interesting in this approach is that compile time transformations are used wherever possible. In the case where this is not possible, an efficient run-time mechanism is used.

Recently an attempt to provide a consistent set of pragmas for FORTRAN on DMAs has been made [FOX91]. The pragmas include alignment, data partitioning and parallel loops.

1.4.3 Automated Translation

Not all research has been based upon pragmas. For a restricted set of programs with nearest neighbour data dependencies, known as stencil problems, a mapping known as tiling has been used. Papers using this method include [RAMA90] and [HUDA90]. Here the optimal size of the tiles for data distribution is considered using a surface area to volume argument. The tiling transformation is placed in a more general context in [ANCO91] where arbitrary data dependence are allowed.

The unimodular transformations described by [BANJ90] are proposed as a method for compiling for DMAs in [KULK91]. They describe the effect of transformations on certain important machine characteristics such as load balancing and communication. However they only consider double loops and no strategy is given as how to apply them for compilation.

An interesting approach is given in [RIBA90]. Here a restricted set of programs is mapped to the systolic WARP machine. By using a matrix representation, he is able to detect a parallelism enhancing ordering and hence computation and data distribution. The compiler performs very well for the examples given. No user intervention was required. The main limitation is that general while loops and non-constant data dependencies cannot be implemented in this approach. Automatic data distribution so as to minimise non-local access for SIMD architectures is described in [WEIS91] and [KNOB90]. Many of the techniques are applicable to an SPMD model. Unfortunately the issue of load balancing is not addressed and the class of programs is once again quite restrictive. Only array references which have one iterator accessing any one index are considered. Additionally each occurrence of a particular iterator in an array reference has to have the same coefficient.

The methods described in [DENN89] and [IKUD90] do not require user pragmas. However they are only exploiting certain very restricted program structures. For instance in [IKUD90] only 4 very specific program patterns are considered for parallelisation. Thus while a wider class of programs can be accepted than in [RIBA90], only a relatively small number of programs can be compiled, for any advantage. In addition they lack any analysis on which decisions are made.

By far the most advanced project is the Crystal project at Yale [CHEN88], [LI90]and [YANG91]. They consider all issues involved in mapping Crystal to DMAs. Throughout this thesis relevant comparisons will be made where appropriate. [GUPT90] based their alignment strategy upon the work at Yale. The Crystal project is the most ambitious in that it encompasses all the issues involved in compiling for DMAs. They intend to fully automate all aspects of the compilation process.

One of the major focuses in the Crystal project is reducing communication overhead. In [LI90], a polynomial algorithm is given to determine the relative alignment of all the arrays in a program. Although this is an NP-complete problem, the approximating algorithm gives good performance. Once a data partition has been determined, the necessary sends and receives can be planted [LI90a]. In trying to determine the best data layout, the compiler searches through a variety of schemes suggested by the array reference patterns. For each layout, a communication cost is calculated. Less emphasis is placed upon load balancing.

1.5 Outline of Thesis

In chapter 2 the computation set notation used throughout this thesis, based upon linear algebra, is introduced. The basic translation from Sisal to computation sets to an imperative language is outlined. A basic architectural model, with associated metrics, is defined so as to allow later evaluation of the compilation scheme.

Chapter 3 investigates load balancing. By describing the amount of work per processor for a particular partition, it is possible to determine whether there exists a transformation to achieve even distribution of work and what that transformation is.

Before the data and computation are mapped to the processors it is necessary to determine the relative alignment of the arrays. This will crucially determine the amount of non-local access and is the subject of chapter 4. Transformations to improve locality are described.

In chapter 5, the folded and interleaved mapping of data and computation to the processors is first described. Pre-fetching transformations to reduce redundant non-local access are then given. Finally a method to determine the data partition that reduces non-local access is presented.

In chapter 6 all these techniques are ordered and applied to eight well known problems. The implementations are evaluated using the metrics developed in chapter 2 and compared with hand written implementations.

Finally in chapter 7, the work is critically evaluated and suggestions for further work are outlined.

Chapter 2

Notation

This chapter describes the language Sisal and a restricted form which is used throughout this thesis. Each Sisal program is expressed in a computation set notation, which is amenable to analysis and transformation. The target language, a description of the translation scheme used and the constraints upon program transformation are outlined. Finally a simple machine model is presented, allowing later evaluation of the compilation scheme.

The translation scheme for Sisal can be summarised as follows:

$$P_0 \overset{\tau_0}{\mapsto} P_1 \overset{\pi_1}{\mapsto} P_2 \overset{\zeta}{\mapsto} \begin{bmatrix} \underline{P_3} \\ \underline{P_3} \\ \vdots \\ \underline{P_3} \end{bmatrix} \overset{\pi_2}{\mapsto} \begin{bmatrix} \underline{P_4} \\ \underline{P_4} \\ \vdots \\ \underline{P_4} \end{bmatrix} \overset{\tau_1}{\mapsto} \begin{bmatrix} \underline{P_5} \\ \underline{P_5} \\ \vdots \\ \underline{P_5} \end{bmatrix} \tag{2.1}$$

P_0 is the original Sisal program and $\underline{P_5}$ the local imperative program for a particular processor. τ_0 represents the translation of Sisal into a computation set representation P_1, as described in section 2.2. π_1 represents any pre-partitioning transformation which transforms P_1 into a new computation set P_2. The ζ transformation maps P_2 into p separate computation sets $[\underline{P_3}, \ldots, \underline{P_3}]^T$, one per processor. The underline symbol implies that the particular representation is **local** to a particular processor. π_2 represents any post-partitioning transformation which is applied to each local computation set to give the new local computation sets $[\underline{P_4}, \ldots, \underline{P_4}]^T$. Finally τ_1 represents the translation of the computation sets into the local imperative programs $[\underline{P_5}, \ldots, \underline{P_5}]^T$ written in the imperative language described in section 2.3 The π and ζ transformations form the body of this thesis. In this chapter, however, we are concerned with τ_0 and τ_1 which translate Sisal, first, into a sequence of computation sets and then into the imperative language. The proof of correctness of such translation schemes is important but beyond the scope of this thesis. Instead an intuitive description is given.

2.1 Sisal

Sisal is a strongly-typed single-assignment language where a name has only one value associated with it. It can be considered to be a first-order functional language, where each function is evaluated without side-effect. Its functional nature provides referential transparency and computation should be thought of as proceeding by expression evaluation rather than state-transition. As values are bound to names, rather than memory locations, there is no aliasing.

Sisal has the usual primitive data types: *integer*, *real*, *double*, *boolean* and *character*. The aggregate data-types are *array*,*record*, *union* and *stream*. The *array* data structure is one-dimensional; multi-dimensional arrays are represented as arrays of arrays. Each component of an array may itself be an array and, since they may be of different lengths, jagged arrays are allowed. A stream is similar to an array in that it, too, is one-dimensional; the major difference is that access is restricted, so that only the *head* of a stream may be accessed at a time. Arrays are defined as strict data-structures, while streams are intended to be implemented non-strictly. Each element of a stream is available as soon as it is produced, and streams are often used in producer-consumer type applications.

2.1.1 Definition before use

As in many languages, such as Pascal, an object must be declared before it is used. If a function is to be used in a mutually recursive manner, then it is necessary to declare it using a **forward** function definition. Sisal, however, does not allow recursive or implicit definitions of value names. One consequence of this is that array data structures may not be defined in terms of themselves and are, therefore, strict data structures. In other words, an array element cannot be accessed until all of the array has been created. This simplifies the implementation of array structures [FEO90B]. In addition, every element of an array is defined once only. Thus, if an array is distributed, each element is written just once and, hence, there will be no memory coherence problems.

2.1.2 Constructs

Sisal has the usual language constructs including functions, conditionals, and iteration. Computation proceeds by evaluation, so just as a function returns a value, so do the **if** and **for** constructs. Sisal programs can use recursion to express repetitive computation but, unlike most functional languages, Sisal also supports iteration.

Sisal has two iterative constructs, the **for initial** and **for** loops. The former allows cross-iteration data dependencies while the latter does not. The **for initial** construct consists of an *initialisation section*, a *loop body*, a *termination test* and a *return clause*. Cross-iteration data-dependency is restricted to values of the previous iteration only, which may be accessed using the **old** prefix. For example consider the **for initial** loop in figure 2.1. The loop body could be written in FORTRAN as $A(I) = I * A(I-1) + b$. Dependencies such as $A(I) = ..A(I+1)..$, $A(I) = ..A(I-J)..$ are not permissible in Sisal.

The **for** construct consists of a *range generator*, a *loop body* and a *return clause*. The range generator defines the values a loop or *iterator* ranges over. These may be combined by the **dot** or **cross** product operator which correspond to the dot and cross product in linear algebra.

The **for** construct is less general than the **for initial** loop, in that the number of iterations must be known on entering the loop and no reference to the previous value of a variable may be made. Finally, Sisal supports seven return operators:**value of**, **array of**, **stream of**, **catenate**, **sum**, **product**, **least** and **greatest**. More details on the structure of both **for** loops may be found in [MCGR85].

```
  d := for initial
         i:= 1;              -- Initialisation
         a:= 0;              -- Section
       while (i <=n)          -- Termination Test
       repeat
         i := old i + 1;      -- Loop
         a := (i * old a) + b  -- Body
       returns value of        -- Returns
       product a              -- Clause
       end for
```

Figure 2.1: A **for initial** Loop

```
e := for i in 1,n cross j in 1,i -- Range Generator
       a:= c[i,j]              -- Loop Body
     returns value of sum a     -- Returns Clause
     end for
```

Figure 2.2: A **for** Loop

2.1.3 Restricted Sisal

The aim of this thesis is to automatically detect and exploit parallelism within a Sisal program suitable for a DMA using an SPMD model of computation. To this end, we are primarily concerned with programs involving the array data structure. Thus, no consideration is given to the implementation of the *record*, *union* and *stream* data types. All functions are assumed to be inlined to simplify analysis. This is clearly impossible for programs with recursive forms and, therefore, attention is limited to non-recursive programs.

With respect to arrays, the following restrictions are imposed: firstly, all arrays must be rectangular and, secondly, all array name instances, including those pre-fixed by the **old** keyword, within a particular lexical scope must have a fixed size associated with them which can be determined at compile-time. This greatly aids data partitioning and allows a simple policy of memory allocation so as to maintain scalability.

Sisal has a rich set of *array creation* operators. In this thesis the only *array create* operation considered is the *array gather*. In figure 2.3 a generic *array gather* is shown, where *J* is a set of iterators and *x* any set of variables. All the remaining *array creators* can be expressed in terms of the *array gather* operator. For example consider the array creations in figures 2.4 to 2.6. Figure 2.4 shows an array, *a*, being created whose elements are the same as array *b* except that the 7th element is replaced by the value *n*. This can be re-written using an *array gather* as shown in figure 2.7.

```
for J
returns array of x
end for
```

Figure 2.3: Generic Array Gather

```
a := b[7:n]
```

Figure 2.4: Replace Operator

```
a := array_fill(1,n,0.0)
```

Figure 2.5: Array Fill

Figure 2.5 shows the creation of an *n* element array where every element is set to 0.0. This can be re-written using an *array gather* as shown in figure 2.8.

Figure 2.6 shows the creation of a new array *a* by the concatenation of two *n*-element arrays, *b* and *c*. This can be re-written using an *array gather* as shown in figure 2.9. Finally a restriction is imposed upon the format of the **for initial** loop, it may only have **value of** as its **return** argument.

At first these conditions may seem overly restrictive, but they are introduced only in order to focus attention upon the main issues. In Sisal it is possible to express the same problem in many different ways, the consideration of which would be distracting.

2.2 Computation Sets

In this section the form of a *computation set*, *array occurrence*, *iteration space* and *index domain* are defined. This notation is based upon the formalism developed in [RIBA90]. A program consists of a sequence of computation sets where each computation set describes the creation of an array variable. An array occurrence describes how an array is accessed, while the iteration space determines the work to be performed. The index domain describes the size and form of an array, and is required when partitioning data across the processors. In this thesis those programs with *affine* loop and occurrence structures are of primary interest. An affine function is one that can be represented as a matrix/vector pair. The focus of attention upon affine forms is justified by an empirical study [SHEN90], where 80% of the program structures were found to be affine. However, whenever non-affine constructs are found it is important that they can be represented otherwise translation is impossible. Where appropriate, procedures for handling non-affine structures will be explained.

Definition 1 *A program consists of a sequence of computation sets* $(Q_1, \ldots, Q_\eta)$ *where* η *is the number of computation sets and* $Q = (A, J, \mathcal{J}, b, \mathcal{S})$ *is a computation set.*

Each of the terms contained within this definition will be described in the subsequent sections.

```
a := b||c
```

Figure 2.6: Array Concatenate

```
a:=for i in 1,n
   returns array of
     if (i = 7)
     then n
     else b[i]
     end if
   end for
```

Figure 2.7: Array Gather

```
a:=for i in 1,n
   returns array of
     0.0
   end for
```

Figure 2.8: Array Gather

```
a:=for i in 1,2*n
   returns array of
     if (i<=n)
     then b[i]
     else c[i-n]
     end if
   end for
```

Figure 2.9: Array Gather

```
a:= for i in 1,n  cross j in 1,n
   returns array of
      b[i,j] * b[j,i+3]
   end for
```

Figure 2.10: A Sisal Program

2.2.1 Array Occurrence

In this section the subscripts of each of the array occurrences are of interest. Consider the Sisal fragment in figure 2.10 where the sub-scripts are of the form:

$$a[i,j],\ \ b[i,j],\ \ b[j,i+3] \tag{2.2}$$

The subscripts of the two occurrences of b are obvious but the subscripts of a are inferred from the iterators in the array gather, i and j. These subscripts can be written in matrix form:

$$a[i,j] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} b[i,j] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

$$b[j,i+3] = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

This representation of an array's sub-scripts is called an *occurrence matrix/vector pair*. There are two distinct types of occurrences matrix.

Definition 2 *A* **c-occurrence** *is the occurrence matrix of the array being created where each element of this array is given a value. Due to the strictness of array structures in Sisal it is not possible to create or write to part of an array structure.*

Definition 3 *A* **u-occurrence** *is the occurrence matrix of an array referenced or used in an array creation. It is possible to use or read part of an array structure. There is no restriction on the number of u-occurrences of an array.*

Definition 4 *A* **segment** *is a fragment of a program which contains no iterators and has only one c-occurrence.*

Let θ_v be the number of distinct u-occurrences of a variable v in a segment.

Let $(\mathcal{U}_{r.v}, v_{r.v}), r \in 1, \ldots, \theta_v$ be the matrix/vector pair corresponding to the rth distinct u-occurrence of a variable v.

Let $\mathcal{U}_v = \{(\mathcal{U}_{r.v}, v_{r.v}), \forall r \in 1, \ldots, \theta_v\}$ be the set of all u-occurrences of variable v in a particular segment.

Let Y be the set of array variables in a segment.

Let $\mathcal{U} = \cup_{v \in Y} \mathcal{U}_v$ be the set of all u-occurrences in a segment

Let $\mathcal{C}_v$ be the c-occurrence matrix in a segment.

For a segment within m nested loops, each occurrence of a variable v is represented by a $N_v \times m$ matrix and $N_v \times 1$ vector pair where N_v is the number of array dimensions of variable v.

Definition 5 *A segment, $\mathcal{S}$, is defined as the tuple $(F, \mathcal{C}, \mathcal{U})$ where F is a parse tree of the segment.*

In Sisal the c-occurrence can always be represented in matrix form. However it is possible for u-occurrences to have non-affine form, i.e. they cannot be represented by a matrix-vector pair. For example, the indirection in the array occurrence, $d[a[i]]$, cannot be represented. In such cases a special entry, $[\perp]$, will be made in $\mathcal{U}_d$, signifying that the particular array occurrence cannot be represented in matrix form. No information will be lost, as the occurrence will remain in the parse tree F.

2.2.2 Iteration Space

The loops in a Sisal program surrounding a segment can be represented as an $m \times 1$ vector where m is the number of loops or iterators.

Let

$$J = [j_1, j_2, \ldots, j_m]^T \tag{2.3}$$

be the *iterators* surrounding a segment. In the example 2.10, J is as follows:

$$J = \begin{bmatrix} i \\ j \end{bmatrix} \tag{2.4}$$

In this thesis three types of loops are identified.

Definition 6 *The* **forall** *form corresponds to the Sisal* **for** *construct, where each iteration may be evaluated in parallel.*

Definition 7 *The* **foriter** *form corresponds to the Sisal* **for initial** *construct, where the number of iterations to be performed can be calculated on entry of the loop. In particular there exists one loop induction variable which is incremented by one on each iteration and the termination test is an affine function of the enclosing iterators.*

Definition 8 *The* **while** *form corresponds to all other Sisal* **for initial** *constructs. In particular it corresponds to loops where the number of iterations depends on the loop body.*

To ensure legal program transformations, it is necessary to record the type of each loop. It is also important to record information about the *return clause*. In the restricted form of Sisal, **for initial** loops may only have a

```
a:= for i in 1,n  cross j in n-i,2*n+i
   returns array of
     for k in 3*j+i+6,4*i-7
     returns value of sum
       b[i,k]
     end for
   end for
```

Figure 2.11: A Sisal Program

returns value of clause. **for** loops may have **returns array of**, or a reduction operator such as **returns value of sum**. For the purposes of compiling for parallelism, the three terms **value**, **array** and **reduction** will be recorded.

Let $\mathcal{J}$ be the $m \times 1$ vector describing the form of each of the iterators and its *return clause*. Each entry in $\mathcal{J}$ will be a tuple of the form (itype,rtype) where itype = (**forall | foriter | while**) and rtype = (**value | array | reduction**)

Affine bounds of the iterators are described by the following set of inequalities which can be determined from the Sisal program :

$$LJ \geq l \tag{2.5}$$

$$UJ \leq u \tag{2.6}$$

where L and U are $(m \times m)$ lower unit triangular matrices l and u are $(m \times 1)$ vectors. The range of values taken on by J define the **iteration space** of a segment. This set of inequalities can be re-written as:

$$\left[\begin{array}{c} -L \\ \hline U \end{array}\right] J \leq \left[\begin{array}{c} -l \\ \hline u \end{array}\right] \tag{2.7}$$

or as

$$AJ \leq b \tag{2.8}$$

where $A = [-L, U]^T, b = [-l, u]^T$.

Definition 9 *Given A, an integer $(l \times m)$ matrix and b an integer $(l \times 1)$ vector, where m is the number of iterators then* **Latt(A.b)** *is the set of points $p \in Z^m$ such that $p \in poly(A.b)$ where $poly(A.b)$ is the m-dimensional polytope given by the set of inequalities* **AJ ≤ b**. *Thus the lattice is a subset of a polytope [SCHR86] that have integer coordinates.*

The convex lattice of integer points corresponds directly to the *iteration space*. More general, i.e non-affine, *iteration spaces* cannot be represented in this way. Once again, in such cases a special entry, $[\perp]$, will be made in A, signifying that the particular iteration space cannot be represented in matrix form. For example consider the Sisal program in figure 2.11 It is possible to represent the loop bounds as follows:

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ -1 & -1 & 0 \\ 1 & 3 & -1 \\ \hline 1 & 0 & 0 \\ -1 & 1 & 0 \\ -4 & 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \\ k \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -n \\ -6 \\ \hline n \\ 2 * n \\ -7 \end{array}\right] \tag{2.9}$$

In general all affine bounds can be represented in this form.

```
a:= for i in 1,n
      returns array of
        for k in 1,n returns value of sum
          if i>k
          then  b[i,k] * c[k]
          else  0
          end if
        end for
      end for
```

Figure 2.12: A Sisal Program

2.2.3 Incorporating Conditionals in Polytopes

In the program in figure 2.12, both branches of the **if** expression can be expressed as a condition that may be added to a polytope.

$$i > k \Leftrightarrow -i + k \leq -1 \tag{2.10}$$

$$i \leq k \Leftrightarrow i - k \leq 0 \tag{2.11}$$

In this example there are two polytopes describing the calculation of two different regions of the same array, *a*. Each polytope will have a different segment associated with it corresponding to the two expressions in the two **if** branches. This process of splitting the calculation of array, *a*, is similar to loop distribution [WOLF89]. The polytope corresponding to the first branch is:

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline -1 & 1 \end{array}\right] \left[\begin{array}{c} i \\ k \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ \hline n \\ n \\ \hline -1 \end{array}\right] \tag{2.12}$$

while the second is given by:

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline 1 & -1 \end{array}\right] \left[\begin{array}{c} i \\ k \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ \hline n \\ n \\ \hline 0 \end{array}\right] \tag{2.13}$$

The polytopes will only describe unique regions if the branches of the **if** expression are mutually exclusive. For a discussion of loop distribution in the presence of conditionals, see [KENN90A]. For those cases where the conditionals cannot be expressed as a constraint matrix/ vector pair, the special entry [⊥] is used.

Definition 10 *The* **canonical form** *of a polytope of a segment is*

$$\left[\begin{array}{c} -L \\ \hline U \\ \hline \mathcal{E} \end{array}\right] J \leq \left[\begin{array}{c} -l \\ \hline u \\ \hline \varepsilon \end{array}\right] \tag{2.14}$$

where $\mathcal{E}, \varepsilon$ are a matrix/vector pair consisting of extra constraints due to loop body conditionals.

```
a:= for i in 1,17  cross j in 1,53
    returns array of
      for k in 1, i cross l in 1,32-j
      returns value of sum
        b[i+l,j+k] * b[j+k,i+3]
      end for
    end for
```

Figure 2.13: A Sisal Program

2.2.4 Index Domain

To allow data partitioning it is necessary to record the size and dimensionality of each array. Each array, v, has the following form:

$$[i_1, i_2, \ldots, i_{N_v}]^T = \mathcal{I}^v \tag{2.15}$$

where N_v is the number of array dimensions, i_x is the xth index of the array v, and $\mathcal{I}^v$ is the vector containing all the indices of that array. The indices have a certain range which describe the size of the array as follows:

$$\lambda \leq \mathcal{I} \leq \upsilon \tag{2.16}$$

where $\lambda = [\lambda_1, \ldots, \lambda_{N_v}]^T$ and $\upsilon = [\upsilon_1, \ldots, \upsilon_{N_v}]^T$ are $N_v \times 1$ vectors. As $\mathcal{I}$ is restricted to rectangular forms, only constant vectors are required to describe its bounds. Each array, in general, has a different range of indices and thus has a different $\mathcal{I}^v$.

The upper and lower bounds of each array are defined by the range of array iterators enclosing its creation occurrence. That is, the range of the iterators which are used to create the c-occurrence of an array determine its size.

The range of the array, a, in figure 2.13 is given by:

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \leq \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \leq \begin{bmatrix} 17 \\ 53 \end{bmatrix} \tag{2.17}$$

Within an array computation only one array is created, so only one index vector, $\mathcal{I}$, is defined corresponding to the c-occurrence. The sizes of the u-occurrence arrays involved are determined by their own c-occurrences. In the case of arrays that are input parameters to the program it is assumed that their sizes are given at compile time.

2.2.5 Local Notation

When the many computation sets are mapped to several processors, each processor will have its own local iteration and data spaces. In this thesis, the convention is to represent local objects by underlining them. For example the polytope enclosing the local iteration space is $\underline{A}\,\underline{J} \leq \underline{b}$, and the local index domain of an array, v, is $\underline{\mathcal{I}^v}$.

```
for initial
 i := 1;
 a := init-a;
 b := init-b;
while (i<=n)
repeat
  i := old i +1;
  a := for j in 1,n
       returns array of
         for k in 1,old i
         returns value of sum old a [k]*c[k,j]
         end for
       end for;
  b := for l in 1,2*n
       returns array of old b[i] * d[l]
       end for;
returns value of a
        value of b
end for
```

Figure 2.14: A Sisal Program

2.2.6 Sisal $\mapsto$ Computation Sets

The basic translation scheme, τ_0, creates a sequence of computation sets and index vectors from a Sisal program as described in section 2.2 defining the computation set notation. For example consider the Sisal program in figure 2.14.

This program is represented by (Q_1, Q_2) where $Q_1 = (A_1, J_1, \mathcal{J}_1, b_1, \mathcal{S}_1)$ and $Q_2 = (A_2, J_2, \mathcal{J}_2, b_2, \mathcal{S}_2)$. The polytope $A_1 J_1 \leq b_1$ is

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \\ k \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ -1 \\ \hline n \\ n \\ 0 \end{array}\right] \tag{2.18}$$

and $\mathcal{J}_1$=[(**forinit**, **value**), (**forall**, **array**), (**forall**, **reduction**)]. $\mathcal{S}_1 = (F, \mathcal{C}, \mathcal{U})$ where F is the segment parse tree and $\mathcal{C}_a$ = [0,1,0] and $\mathcal{U}_{olda}$= [0,0,1], $\mathcal{U}_c$= $\left[\begin{array}{ccc} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right]$

Similarly for Q_2:

The polytope $A_2 J_2 \leq b_2$ is

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ l \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ \hline n \\ 2 * n \end{array}\right] \tag{2.19}$$

```
alo = 7
ahi = 15
A[(alo..ahi)]:real
```

Figure 2.15: Array Declaration

and $\mathcal{J}_2$=[(**forinit**, **value**), (**forall**, **array**)]. $\mathcal{S}_2 = (F, \mathcal{C}, \mathcal{U})$ where F is the segment parse tree and $\mathcal{C}_b = [0,1]$, $\mathcal{U}_{oldb} = [1,0]$, $\mathcal{U}_d = [0,1]$

The index domains for the two c-occurrences are:

$$\begin{bmatrix} 1 \end{bmatrix} \leq \mathcal{I}^a \leq \begin{bmatrix} n \end{bmatrix} \tag{2.20}$$

and

$$\begin{bmatrix} 1 \end{bmatrix} \leq \mathcal{I}^b \leq \begin{bmatrix} 2 * n \end{bmatrix} \tag{2.21}$$

2.3 Imperative Language

The target architecture consists of multiple von Neumann memory processor pairs and it is appropriate for the target language of the Sisal compiler to be imperative. Each processor will have its own imperative program which can be compiled by the machine's native compiler.

The language described here closely mimics the restricted Sisal form. The major difference is that Sisal is functional while the semantics of this form are imperative. A complete WBNF syntax description of the imperative language used in this thesis is given in appendix A. The important features of the imperative language are described in the following sub-sections.

2.3.1 Data types

The same scalar types as found in Sisal are present in this intermediate form. The only aggregate data type allowed is the multiple dimensional array whose elements are scalars. All variables are declared locally but may be accessed globally. The arrays are static and declared at the beginning of the program and can have any constant lower bound. The declaration in figure 2.15 is typical.

2.3.2 Constructs

As the intermediate language is imperative, the most basic construct is assignment, denoted, as usual, by :=. The usual constructs of repetitive and conditional evaluation are provided by the **IF**, **FOR**, **FORITER** and **WHILE** constructs. The **FOR** construct corresponds to the **for** loop in Sisal where there are no cross-iteration data dependencies and the number of iterations is known on entering the loop. The **FORITER** construct corresponds to the Sisal **for initial** loop where the number of iterations is known on entering the loop body. As a sequential implementation of the **for initial** is assumed, a barrier synchronisation occurs at the end of each iteration so as to ensure that the processors do not become out of step. The **WHILE** construct corresponds to all the remaining

```
a[(1..n),(1..n)]:real
b[(1..n),(1..n)]:real
c[(1..n),(1..n)]:real
 FOR i = 1 TO n
   FOR j = 1 TO n
     c[i,j] := 0
     FOR k = 1 TO n
       c[i,j] := c[i,j] + Get(a[i,k]) * Get(b[k,j])
     END FOR
   END FOR
 END FOR
```

Figure 2.16: An Imperative Program

forms of the Sisal **for initial** loop. Again, as it is implemented sequentially, a barrier synchronisation takes place after each iteration.

Access to potentially non-local array elements is made via the **Get** function which takes a u-occurrence as its argument. If the data to be accessed is local then a simple memory transfer takes place. If however the data is non-local, then communication with another processor will take place.

An example of this imperative language is the matrix multiplication program in figure 2.16.

2.3.3 Matrix Notation $\mapsto$ Imperative Language

The translation of matrix notation into the imperative form is described in two parts: firstly, the translation of a particular computation set into the imperative language and, secondly, the translation of a sequence of computation sets into the imperative language.

Given $Q = (A, J, \mathcal{J}, b, \mathcal{S})$ and $\mathcal{S} = (F, C, U)$. the basic translation scheme is to produce a nest of appropriate loops given by the J and $\mathcal{J}$ vectors with an assignment as the loop body whose form is governed by $\mathcal{S}$.

Essentially each iterator is printed out, depending on its type, with its loop bounds determined by A, b. The $\mathcal{E}, \varepsilon$ entries in the polytope $AJ \leq b$ are represented as **IF** statements. These are placed immediately after the iterators they reference. An algorithm to perform this translation is given in appendix B. In the simple case of addition, the loop body will be as in 2.22. However, in general, the exact form will depend upon the operands in F.

$$v[\mathcal{C}J]{:=}\mathbf{Get}(v_1[\mathcal{U}_1 J + v_1]) + \cdots + \mathbf{Get}(v_{\theta_v}[\mathcal{U}_{\theta_v} J + v_{\theta_v}]) \tag{2.22}$$

All u-occurrences are preceded by the **Get** operator as they are potentially distributed. Sisal requires a c-occurrence of an array to occur only once. In addition every element is defined once only. Because of this, the translation scheme must guarantee that all c-occurrence writes are local.

All array references to the old value of array v will be referenced as *oldv*. Thus those arrays which are defined within a **for initial** loop have two copies in the imperative form. At the end of each **for initial** loop the contents are swapped. This is a very conservative implementation. In [CANN89A] a sophisticated copy avoidance method is given.

```
FOR i = j TO j+n
  FOR j = 1 TO n
    .....
  END FOR
END FOR
```

Figure 2.17: An Illegal Imperative Program

```
a:= for i in 1,15
    returns array of b[i]
    end for
```

Figure 2.18: A Sisal Program

It is now possible to determine restrictions upon transformations for each computation set.

- The form of the $AJ \leq b$ polytope must represent a legal program in the imperative language.

  Essentially this means that one iterator's loop bounds cannot make reference to an iterator deeper in the nest. This implies that $AJ \leq b$ must be in **canonical form**. For example the following polytope is prohibited

$$\left[\begin{array}{rr} -1 & 1 \\ 0 & -1 \\ \hline 1 & -1 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] \leq \left[\begin{array}{r} 0 \\ -1 \\ \hline n \\ n \end{array}\right] \qquad (2.23)$$

  as the corresponding program in figure 2.17 is illegal.

- The size of an array must, obviously, remain the same. Consider the program in figure 2.18 with

$$\left[\ 1\ \right] \leq \left[\ i_1\ \right] \leq \left[\ 15\ \right] \qquad (2.24)$$

  clearly shrinking the array to give the following is illegal

$$\left[\ 1\ \right] \leq \left[\ i_1\ \right] \leq \left[\ 10\ \right] \qquad (2.25)$$

  because it will give the imperative program in figure 2.19

- In order to preserve data dependence, no transformation is allowed on any sequential iterator. This restriction will result in the preservation of the sequential ordering. This is a conservative approach since data dependence may only require that part of an iteration is performed before the next one commences.

```
FOR i = 1 TO 10
  a[i] := Get(b[i])
END FOR
```

Figure 2.19: An Imperative Program

```
a:= for i in  2,7 cross j in 2, 7
    returns array of b[i,j]
    end for
```

Figure 2.20: A Sisal Program

```
FOR i = 2 TO 7
  FOR j = 2 TO 7
    a[i,i] := Get(b[i,i])
  END FOR
END FOR
```

Figure 2.21: An Imperative Program

- The occurrence matrices must still reference the same number of points. For example consider the Sisal program in 2.20 with has the occurrence matrices:

$$\mathcal{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathcal{U} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.26}$$

  If the occurrence matrices are transformed into

$$\mathcal{C} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \mathcal{U} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \tag{2.27}$$

  then the imperative program in figure 2.21 results, which defines only the diagonal elements of a, rather than the whole array.

When translating multiple computation sets into imperative form it is essential that data dependence is preserved. The simplest method is to ensure that each **foriter** and **while** loop is neither distributed nor transformed in any manner. An algorithm to translate multiple computation sets with this restriction is given in appendix B.

The major consequence of this requirement is that any two computation sets sharing a common sequential iterator at, say, loop nest depth $k, 1 \leq k \leq m$ must have all the previous iterators in the same order with the same loop bounds so that the two computation sets may be merged.

To illustrate this point consider the program in figure 2.22, where arrays a and b have the same iteration space.

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \left[\begin{array}{c} -1 \\ -1 \\ -1 \\ \hline n \\ n \\ n \end{array}\right] \tag{2.28}$$

If it was decided to interchange the iterators i and j in the polytope of b to give the new polytope:

```
c,d := for i in 1,n cross j in 1,n
      returns array of
        for initial
          k := 1;
          a := init-a;
          b := init-b;
        while (k<=n)
        repeat
          k := old k + 1;
          a := old a + (3 * old b);
          b := a + (2 * old a) + (3 * old b);
        returns value of a
                value of b
        end for
      end for
```

Figure 2.22: A Sisal Program

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \begin{bmatrix} j \\ i \\ k \end{bmatrix} \leq \left[\begin{array}{c} -1 \\ -1 \\ -1 \\ \hline n \\ n \\ n \end{array}\right] \tag{2.29}$$

then, on translating to the imperative language, it would be impossible to fuse the loops and thus the program in figure 2.23 would result, which clearly does not preserve data dependence.

If the interchange had not taken place then the legal program in figure 2.24 would be given.

Thus when more than one computation set shares a nested sequential iterator, the analysis and transformations must be performed simultaneously with respect to the computation sets involved.

With the aforementioned conditions, transformations upon each computation set can proceed largely independently. Only the condition regarding a shared sequential loop restricts this.

2.3.4 Creation and Reduction Parallelism

As mentioned in the introductory chapter, two forms of parallelism are exploited, namely creation and reduction parallelism. Translating for creation parallelism is essentially the method described in the previous sub-section. Translating for reduction parallelism requires a small modification.

Consider the Sisal program in figure 2.25 which has the following polytope, occurrence matrices and index domains:

```
FOR i = 1 TO n
  FOR j = 1 TO n
    FORITER k = 1 TO n
      a := olda+(3* oldb)
    END FOR
    c[i,j] := a
  END FOR
END FOR
FOR j = 1 TO n
  FOR i = 1 TO n
    FORITER k = 1 TO n
      b := a+ (2* olda) + (3* oldb)
    END FOR
    d[i,j] := b
  END FOR
END FOR
```

Figure 2.23: An Imperative Program

```
FOR i = 1 TO n
  FOR j = 1 TO n
    FORITER k = 1 TO n
      a := olda+ (3* oldb)
      b := a+ (2* old a) + 3* old b
    END FOR
    c[i,j] := a
    d[i,j] := b
  END FOR
END FOR
```

Figure 2.24: An Imperative Program

```
a:= for i in 1,64
    returns array of
      for k in 1, 64
      returns value of sum
        b[i,k]
      end for
    end for
```

Figure 2.25: A Sisal Program

```
sum : real
a[(17..32)] : real
b[(17..32),(1..64)] :real
FOR i = 17 TO 32
  sum := 0
  FOR k = 1 TO 64
    sum := sum + Get(b[i,k])
  END FOR
  a[i] := sum
END FOR
```

Figure 2.26: An Imperative Program

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ k \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ \hline 64 \\ 64 \end{array}\right] \tag{2.30}$$

$$\mathcal{C}_a = \left[\begin{array}{cc} 1 & 0 \end{array}\right] \mathcal{U}_{1.b} = \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right] \tag{2.31}$$

$$\left[\ 1\ \right] \leq \left[\ i_1^a\ \right] \leq \left[\ 64\ \right] \tag{2.32}$$

$$\left[\begin{array}{c} 1 \\ 1 \end{array}\right] \leq \left[\begin{array}{c} i_1^b \\ i_2^b \end{array}\right] \leq \left[\begin{array}{c} 64 \\ 64 \end{array}\right] \tag{2.33}$$

Consider the case when there are 4 processors and we are interested in the the local iteration space and index domain of processor 2. In compiling for creation parallelism, the *i* loop will be parallelised which implies that *b* is to be partitioned along the first index. This gives the following local values:

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \underline{i} \\ \underline{k} \end{array}\right] \leq \left[\begin{array}{r} -17 \\ -1 \\ \hline 32 \\ 64 \end{array}\right] \tag{2.34}$$

$$\left[\ 17\ \right] \leq \left[\ \underline{i_1^a}\ \right] \leq \left[\ 32\ \right] \tag{2.35}$$

$$\left[\begin{array}{r} 17 \\ 1 \end{array}\right] \leq \left[\begin{array}{c} \underline{i_1^b} \\ \underline{i_2^b} \end{array}\right] \leq \left[\begin{array}{c} 32 \\ 64 \end{array}\right] \tag{2.36}$$

this results in the local program given in figure 2.26

The method by which the particular local polytope and index domain are formed is not the concern here. The important feature is that the processor accesses potentially non-local data which is used to assign values to the **local** portion of the distributed array.

```
sum : real
psum[(2..2)]:real
a[(17..32)] : real
b[(1..64),(17..32)] :real
FOR i = 1 TO 64
  psum[2] := 0
  FOR k = 17 TO 32
    psum[2] := psum[2] + b[i,k]
  END FOR
  IF (i>=17) AND (i<=32)
  THEN sum:= 0
       FOR x = 1 TO  4
         sum := sum + Get(psum[x])
       END FOR
       a[i] := sum
  END IF
  Sync
END FOR
```

Figure 2.27: An Imperative Program

Exploiting reduction parallelism implies parallelising the *k* loop and partitioning the *b* array on the second index. This gives the following local iteration space and index domain:

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \underline{i} \\ \underline{k} \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -17 \\ \hline 64 \\ 32 \end{array}\right] \tag{2.37}$$

$$\left[\ 17\ \right] \leq \left[\ \underline{i^a_1}\ \right] \leq \left[\ 32\ \right] \tag{2.38}$$

$$\left[\begin{array}{c} 1 \\ 17 \end{array}\right] \leq \left[\begin{array}{c} \underline{i^b_1} \\ \underline{i^b_2} \end{array}\right] \leq \left[\begin{array}{c} 64 \\ 32 \end{array}\right] \tag{2.39}$$

In translating into the imperative language the program, shown in figure 2.27, is derived.

Several points are worth noting. Firstly no **Get** operator prefixes the u-occurrence of $b[i,k]$. This is because, in reduction parallelism, all work is performed upon local data. The **IF** statement is required as only part of the time will the accumulated value, calculated locally, be assigned to local data. When a local element is to be created, the partial sums calculated by each of the other 3 processors are accessed and hence the need for the **Get** operator around the u-occurrence of *psum*[*x*]. Finally the barrier synchronisation **Sync** is required so that the correct value of *psum* is accessed. This construct is only used in compiling for reduction parallelism.

Compiling for reduction parallelism is slightly more involved in the translation to imperative form stage. An algorithm to perform this additional work is given in appendix B.

2.4 Architectural model

A simple model of a DMA is used in this thesis. Firstly they are defined as being homogeneous i.e. each processor is identical and has an equal amount of memory. Non-local access on DMAs, such as the Intel hypercube, for example, is two orders of magnitude more expensive than local access. The time per hop across the processors is a small fraction of the total time to transfer the data and thus it is reasonable to model such DMAs as two-tier memory hierarchies [ROGE90]. Therefore memory access is either local or non-local where there is no consideration of the distance to non-local memory. No interconnection topology is assumed, except that there exists a path from any one processor to all others and that multi- dimensional rectangular grids may be embedded.

It is important to develop a cost model based on this architectural model to later determine the efficiency of a transformation. The measures introduced in this section broadly describe the cost of computation and communication for a DMA. These measures assume a static, one process per processor implementation and, thus, the costs of process creation or spawning, process migration and context switching are not present.

The execution time will, in general, be a function of the problem size and the number of processors available.
Let Problem size = n, Number of processors =p
The following metrics are largely based on [TSEN89].

2.4.1 Computation

The first metric is concerned with the amount of "useful work" and is simply the amount of computation performed. This is defined as the number of operations performed including all integer and real arithmetic. The time to execute a program in parallel is assumed to be the number of operations performed on the critical path.

Let the total number of operations, if the program were executed upon one processor, be Op, then Op can be split $Op = S + Co$ where Co is the time spent in potentially parallel activity e.g. **for** loops, and S is the remainder. The best execution time would be $S + \frac{Co}{p}$ but this may not be achievable due to load imbalance.

2.4.2 Load Balance

Let the average number of parallel operations per processor: $\bar{co} = \frac{1}{p}Co$. The load balance of a processor is defined as the difference between the number of computations performed by that processor and the average. If the modulus of this value is used, then the load imbalance for processor z is $\sigma co_z = |co_z - \bar{co}|, z \in 1, \ldots, p$

2.4.3 Communication

For most non-trivial applications, communication is inevitable when a program is implemented on a DMA, so references to memory are distinguished by whether or not the accessed memory is local to the processor. Communication takes place due to non-local access to memory. Let the number of non-local accesses in processor z be: $na_z, z \in 1, \ldots, p$

It is assumed that the maximum values of σco_z and na_z will dominate the overhead in execution time. Therefore techniques are developed in chapters 3 to 5 to reduce these values. In chapter 6 the maximum load imbalance and non-local access will be calculated for several transformed programs. These values will help evaluate the compilation strategy used.

2.5 Summary

In this chapter, the source and target languages for compilation have been presented. A computation set notation has been developed which is suitable for analysis and transformation. An outline of a scheme for translating one program form into another has been given, which included a description of some transformation restrictions. Finally a simple machine model has been presented so as to allow later evaluation of the compilation scheme presented in this thesis.

Chapter 3

Load Balancing

This chapter is concerned with the efficient mapping of array computation to processors. The major overhead associated with the mapping of computation is load imbalance. An optimising compiler must find a mapping such that this overhead is minimised.

The first section describes load balancing in general and describes an algorithm to determine how to partition the computation. In general the algorithm is very expensive and may not find the best load balancing partition. In the second section, perfect load balancing is defined as an invariancy condition. A transformation to reveal such invariancy for conditional-free polytopes is given. Section two is based upon the paper [OBOY92].

In the subsequent section, the analysis is extended to include **if** expressions, where, by reordering the iterators, it is possible that they may be removed so as to allow invariant analysis. For those computations where no invariant partitions can be found, interleaving is discussed. Finally a summary of the major points is given.

3.1 General Load Balancing

3.1.1 Identification of Parallelism

Compilation should minimise parallel time by utilising machine parallelism and reducing overhead. The first stage of compilation is therefore to identify and match program parallelism to machine parallelism. Machine parallelism is defined simply as the number of processors *p*. As the underlying model of computation used in this thesis has one process per processor, it is necessary to identify and divide an array computation into *p* sub-computations.

Each iteration of a Sisal **for** loop may be executed independently and is thus ideal for parallel execution. In this thesis only **for** loops are considered for exploitation of parallelism. **for initial** loops are considered to require sequential execution. Of course this is a very conservative view of program parallelism, in particular, it is possible (though possibly bad programming style) to write a **for initial** loop describing completely parallel computation. Compiling for parallelism is the task of selecting one or more iterators which are partitioned into groups and scheduled across the processors. In effect each processor performs a sub-set of the loop iterations. This has, in the past, been called loop elimination [CALL88].

```
a := for  i in 1,100 cross j in 1,100
    returns array of
      for  k  in 1,i
      returns value of sum b[i,k]+c [k,j]
      end for
    end for
```

Figure 3.1: Nested Computation

To motivate the rest of this chapter consider the example in figure 3.1. If the *i* loop is partitioned and statically scheduled across 10 processors, such that the first processor receives the first 10 iterations, the second processor the next 10 etc., then the first processor will perform 5,500 iterations, the last processor will perform 95,500, with the average being 50,500. If, however, the *j* loop were chosen all processors would perform 50,500 iterations. If it is assumed that the time to execute such a program is dominated by the processor performing the most iterations, then clearly partitioning with respect to *j* is preferable.

3.1.2 Optimisation

Associated with each computation set, Q, there is a polytope, $AJ \leq b$, describing the iteration space. In general this polytope is of the form:

$$\begin{bmatrix} -L \\ U \\ \mathcal{E} \end{bmatrix} J^m \leq \begin{bmatrix} -l \\ u \\ \varepsilon \end{bmatrix} \tag{3.1}$$

where L, U, l, u are derived from the loop bounds and $\mathcal{E}, \varepsilon$ are conditionals.

The polytope, $AJ \leq b$, encloses a lattice of points, $Latt(A.b)$, known as the iteration space. It is assumed that the amount of work associated with each lattice point is constant. If there are conditionals which are a function of the iterators present, these will be incorporated in the polytope which will restrict the number of lattice points, but the amount of work per point remains the same. However if the conditionals are not a function of the iterators, i.e. data dependent, then it is not possible to determine at compile time their effect. Thus it is a reasonable assumption to ignore their effect by considering the amount of computation associated with each point to be constant throughout the lattice. A load balanced mapping is one where the number of lattice points is the same in each processors. This is subject to the scheduling constraint that partitioning may take place only with respect to **for** loops.

Given that m is the number of iterators or the dimension of the iteration space, let r be the number of parallel iterators. It is necessary to partition the sub-lattice enclosed within $A^r J^r \leq b^r$, which is a projection along the serial iterators [SCHR86], into p sub-spaces. In general $r \neq m$ as J^m contains serial **for initial** loops, which are not candidates for partitioning.

The points in the sub-lattice $Latt(A^r.b^r)$ are those that may be evaluated in parallel. Let q be such a point, $q \in Latt(A^r.b^r)$. Each point q has a lattice $Latt(A^{m-r}.b^{m-r})$ associated with it. In other words each point that may be evaluated in parallel has computation points associated with it, which must be evaluated sequentially. Let $|q|_x$ be the number of points assigned to a processor x. We can state load balancing in this case to be find a mapping,

π, which maps each parallel lattice point to a processor x :

$$\pi : q \mapsto x \ \forall q \in \ Latt(A^m.b^m), x \in \ 1, \dots, p \tag{3.2}$$

such that

$$|q|_x = |q|_y, \forall x, y \in \ 1, \dots, p \tag{3.3}$$

In general this is not achievable and must be expressed as an optimisation problem where 3.3 is replaced by

$$Minimise(\max_x(|q|_x) - \frac{\sum_{x=1}^{p} |q|_x}{p}) \tag{3.4}$$

3.1.3 Volume of Polytope

For each iterator $j \in \ J$, if the number of computation points can be determined, then the best choice of iterator for partitioning is the one where the number of computation points varies the least with respect to the iterator.

One expensive method based on this observation is to partition the iteration space into a number of sub-spaces say s. The lattice $Latt(A.b)$ will also be partitioned into a similar number of sub-lattices. This is made possible by forming s instances of the polytope, where the iterator(s) to be partitioned have the added constraint that they must range over a sub-range(s). Partitioning a lattice into s sub-lattices is realised by intersecting the iteration space with $s - 1$ parallel hyperplanes. The choice of hyperplane is, in general, arbitrary and can be any linear combination of the iterators. In practice some constraint must be placed on the number of partitions investigated to make the scheme computationally feasible. If the hyperplanes are restricted to being orthogonal with respect to the iterators, then the number of choices reduces to r or $2^r - 1$ if the lattice is to be partitioned by more than one iterator.

It is then necessary to determine the number of integer points in each of the sub-polytopes where the maximum value is recorded as this will dominate the load imbalance. This is repeated for the other partitions where the partition having the smallest maximum is the best candidate for load balancing.

It is important that finding the number of points in a sub-polytope is relatively easy. There exists an $O(m^{19})$ algorithm [DYER91] based on random walking for general convex bodies. It determines the volume of euclidian space rather than the number of integer points. This approximation is acceptable for our purposes, however the probabilistic nature of the algorithm and its high polynomial term are prohibitive. Other methods to calculate the volume of polytopes have been found [LAWR91]. However in the general case they are #P hard, though the average complexity for polytopes with integral points may be polynomial [STAN86]. At present the complexity of determining the best partitioning is $O((2^r - 1) \times s \times m^{19})$, where s is the number of partitions.

Ideally the value of s should be exactly p, the number of processors. If however this is too expensive then a smaller value of s may be used i.e. when $s >> 2^r - 1$ and $s >> m^{19}$.

Although this procedure can be used, it is computationally expensive. Additionally, it may be that there exists a non-orthogonal partition which gives a better load balanced implementation. Throughout the remainder of this chapter, a method is developed which determines the existence and form of a transformation which will convert a computation set into a load balanced form. Initially the analysis is developed for polytopes with no conditionals, i.e. $\mathcal{E} = 0, \varepsilon = 0$ in 3.1. Once this has been established, additional criteria for the general case are developed.

3.2 Conditional- Free Polytopes

Transforming for load balancing is based upon the following observation: **The iterator that neither makes reference to any other iterator in its loop bounds, nor is referenced by any other, may be partitioned to give perfect load balance**. A computation set is defined as being in perfect load balanced form when each iteration of a particular iterator involves exactly the same amount of computation. The following section formalises this idea and provides a mechanism to transform loops accordingly.

Consider the lattice $Latt(A.b)$. We seek a method of partitioning this polytope into p subsections such that the number of points scheduled to each is equal. Such a scheme provides perfect load balance. If we consider just orthogonal partitions of the polytope (partitions that are perpendicular to an iterator axis), then we seek the parallel iterator that may possess this property. In general we seek an iterator where the number of points associated with that iterator is **not** a function of that iterator, i.e. it is invariant.

Each iterator $j \in J^m$ has a lattice $Latt(A^{m-1}.b^{m-1})$ of computation points associated with it. We need to find a parallel iterator, $j_b \in J^r$, where r is the number of parallel iterators, such that the number of points in its associated lattice is invariant of such an iterator.

For the sub-set of programs that possess affine loops, it is possible to determine whether a lattice may be partitioned in a load balanced manner. Firstly a representation is introduced which describes nested affine loops for which the criteria for perfect load balancing can be formally stated. Secondly a method is introduced which discovers if a particular iterator can be transformed into a perfect load balanced iterator. This transformation is equivalent to a change in basis for the iteration space. An extension to this method follows whereby partitioning with respect to several iterators can be determined. Finally the need for reordering the iterators for code generation is explained and a method to achieve this for load balanced iterators is described.

We first formally present the invariance condition necessary for load balancing in terms of the loop structure and provide an example to illustrate this.

Given the system of inequalities described by the loop bounds:

$$LJ \geq l \tag{3.5}$$

$$UJ \leq u \tag{3.6}$$

Definition 11 *Let e_b^T be the bth row of the identity matrix, then the* **invariance condition** *is defined as:*

$$e_b^T L = e_b^T U = e_b^T \tag{3.7}$$

$$Le_b = Ue_b = e_b \tag{3.8}$$

The significance of the invariance condition is that the iterator j_b satisfying 3.7 and 3.8 is invariant if it does not make reference to other iterators nor is it referenced by any other iterator. In general both L and U will be of the form:

$$\left[\begin{array}{c|c|c} L_f & 0 & 0 \\ \hline y_L & 1 & 0 \\ \hline A_L & x_L & L_g \end{array}\right], \left[\begin{array}{c|c|c} U_f & 0 & 0 \\ \hline y_U & 1 & 0 \\ \hline A_U & x_U & U_g \end{array}\right] \tag{3.9}$$

```
a:= for i in 1,n cross j in 1,n
    returns array of
      for k in 2*i, 3*i
      returns value of sum b[i,i]-c[k-j]
      end for
    end for
```

Figure 3.2: A Sisal Program

where L_f and U_f are $(b-1)\times(b-1)$ lower unit diagonal triangular matrices, L_g and U_g are $(m-b)\times(m-b)$ lower unit diagonal triangular matrices, y_L and y_U are $1\times(b-1)$ vectors, x_L and x_U are $(m-b)\times 1$ vectors and A_L and A_U are arbitrary integer $(m-b)\times(b-1)$ matrices.

As stated previously a candidate iterator for partitioning is one which does not refer to the bounds of any other loop. Therefore for an iterator $j_b \in J^m$ to satisfy 3.7 and 3.8 we require the following:

$$y_L = y_U = 0 \tag{3.10}$$

and

$$x_L = x_U = 0 \tag{3.11}$$

Therefore it is necessary to inspect each row in L, U to see if these conditions are satisfied. In general for a given set of loops this will not be true.

To illustrate these points, consider the computation in figure 3.2. The range of each of the iterators is represented by two matrix inequalities where each row corresponds to a unique iterator, and each matrix corresponds to the lower and upper bounds of the loop respectively :

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \tag{3.12}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \begin{bmatrix} n \\ n \\ 0 \end{bmatrix} \tag{3.13}$$

Here conditions 3.10 and 3.11 hold only for the second loop j.

However it is possible that none of the iterators have this form but may be transformed (with corresponding adjustments to the array occurrences) to a load balanced form. This is the subject of the next section.

3.2.1 Transformations

Legal transformations include any reordering of the computation that maintains the data dependency of the original program. By restricting this reordering to **forall** loops which contain no cross-iteration dependencies, all data dependencies are preserved. Additionally after transformation the polytope representation of the iteration space should be in the canonical form so as to allow translation to the imperative language described in chapter 2. Finally it is assumed that if the following transformations are performed on a computation set which has a common nested serial iterator, then the analysis and transformations are performed simultaneously.

3.2.2 Change of Basis

Given an L, U and a form L^b, U^b, where the iterator j_b is in a load balanced form satisfying 3.10 and 3.11, find a transformation, π, such that:

$$\pi : L \mapsto L^b \tag{3.14}$$

$$\pi : U \mapsto U^b \tag{3.15}$$

We look at a restricted set of unimodular transformations [BANJ90] which satisfy 3.14 and 3.15 by post multiplication by a unit lower triangular matrix T, which changes the basis of $J \mapsto TJ_b$. The system of inequalities described by the loop bounds remains unchanged by this transformation.

$$LJ \geq l \tag{3.16}$$

$$UJ \leq u \tag{3.17}$$

$$L(TT^{-1})J \geq l \tag{3.18}$$

$$U(TT^{-1})J \leq u \tag{3.19}$$

$$(LT)(T^{-1}J) \geq l \tag{3.20}$$

$$(UT)(T^{-1}J) \leq u \tag{3.21}$$

If in addition an integer matrix T exists such that $LT = L^b$, $UT = U^b$, with J_b as the new iterators, then 3.20 and 3.21 may be written:

$$L^b J_b \geq l \tag{3.22}$$

$$U^b J_b \leq u \tag{3.23}$$

3.2.3 Existence Condition

In this section the necessary and sufficient conditions for the existence of a unimodular unit lower triangular matrix T is addressed.

Necessity: Assume there is a T such that $LT = L^b, UT = U^b$. The forms of L, U, T are as follows:

$$\left[\begin{array}{c|c|c} L_f & 0 & 0 \\ \hline y_L & 1 & 0 \\ \hline A_L & x_L & L_g \end{array}\right], \left[\begin{array}{c|c|c} U_f & 0 & 0 \\ \hline y_U & 1 & 0 \\ \hline A_U & x_U & U_g \end{array}\right], \left[\begin{array}{c|c|c} T_f & 0 & 0 \\ \hline y_T & 1 & 0 \\ \hline A_T & x_T & T_g \end{array}\right] \tag{3.24}$$

To satisfy the invariance condition, we require L^b and U^b to be of the following form:

$$\left[\begin{array}{c|c|c} L^b_f & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline A^b_L & 0 & L^b_g \end{array}\right] \left[\begin{array}{c|c|c} U^b_f & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline A^b_U & 0 & U^b_g \end{array}\right] \tag{3.25}$$

Condition 3.10 implies that:

$$y_L T_f + y_T = 0 \tag{3.26}$$

$$y_U T_f + y_T = 0 \tag{3.27}$$

Therefore

$$(y_U - y_L) T_f = 0 \tag{3.28}$$

As $T_f \neq 0$ then

$$y_U = y_L \tag{3.29}$$

This is the first condition for existence of T. Condition 3.11 implies that:

$$L_g x_T = -x_L \tag{3.30}$$

$$U_g x_T = -x_U \tag{3.31}$$

Thus the solutions for x_T given by 3.30 and 3.31 must be consistent. Equations 3.30 and 3.31 may be written:

$$\left(\begin{bmatrix} 1 & 0 \\ x_L & L_g \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ x_U & U_g \end{bmatrix} \right) \begin{bmatrix} 1 \\ x_T \end{bmatrix} = 0 \tag{3.32}$$

$$\begin{bmatrix} 1 & 0 \\ x_L & L_g \end{bmatrix} \begin{bmatrix} 1 \\ x_T \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3.33}$$

Together these form the second and third condition for existence of T. Clearly 3.29, 3.32 and 3.33 must hold if a transformation is to be determined. This establishes the necessity.

Sufficiency: Assume 3.29 holds, and there is x_T satisfying 3.32 and 3.33 then:

$$T = \left[\begin{array}{c|c|c} I_{b-1} & 0 & 0 \\ \hline -y_L & 1 & 0 \\ \hline 0 & x_T & I_{m-b} \end{array} \right] \tag{3.34}$$

is the desired unimodular transformation corresponding to iterator j_b.□

3.2.4 Algorithm 1

The following algorithm determines whether a transformation T exists and, if it does, finds it. In addition, the relationship between the new iteration space and the old one is determined, so that the relevant array occurrences may be altered accordingly.

for each $j_b \in J^m$

1. Check $y_L = y_U$. If not terminate.
2. Choose an arbitrary lower unit diagonal T_f e.g. unity.
3. Calculate $y_T = -y_L T_f$
4. Solve $\left(\begin{bmatrix} 1 & 0 \\ x_L & L_g \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ x_U & U_g \end{bmatrix} \right) \begin{bmatrix} 1 \\ x_T \end{bmatrix} = 0$
5. Check if consistent. If not terminate
6. Solve $\begin{bmatrix} 1 & 0 \\ x_L & L_g \end{bmatrix} \begin{bmatrix} 1 \\ x_T \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
7. Check steps 4 and 6 are consistent. If not terminate

```
a := for  i in 1,n
     returns array of
       for j in n+i-1,2*n+i+1 cross k in 1+2*i+2*j,n+3*i+2*j
       returns value of sum c[i,k]*d[i-j,k]
       end for
     end for
```

Figure 3.3: A Sisal Program

8. Choose an arbitrary lower unit diagonal T_g e.g. unity.
9. Choose an arbitrary matrix A_T e.g. the null matrix.
10. Construct T
11. Calculate $L^b = LT, U^b = UT$
12. For each $j \in J$ in each array occurrence substitute $J = TJ_b$

The complexity of this algorithm is dominated by steps 4 and 6. Thus the upper bound complexity is $O(m^2)$. If this process is repeated for all the iterators then the upper bound complexity is $O(m^3)$. To illustrate this algorithm, consider the program in figure 3.3. In its present form it does not satisfy the invariance condition. The upper and lower bounds for each of the iterators are as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & -2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 1 \\ n-1 \\ 1 \end{bmatrix} \tag{3.35}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -3 & -2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \begin{bmatrix} n \\ 2n+1 \\ n \end{bmatrix} \tag{3.36}$$

By applying algorithm 1, it is possible to determine if this program may be transformed into an invariant form. Test the first iterator i:

1. Not Applicable.
2. Not Applicable.
3. Not Applicable.
4. Find x_T where
   $$\left(\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & -2 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -3 & -2 & 1 \end{bmatrix} \right) \begin{bmatrix} 1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$
   This implies $1 = 0$. A contradiction and thus loop i is rejected.

Now try the second iterator j i.e. $b = 2$.

1. $y_U = -1$ and $y_L = -1$ therefore $y_U = y_L$ is satisfied.
2. $T_f = 1$
3. $y_T = -(-1)1 = 1$

```
FOR i = 1 TO n
  a[i] := 0
  FOR j = n-1 TO 2*n+1
    FOR  k = 1+4*i TO  n+5*i
      a[i] := a[i]+ (Get(c[i,k+2*j])*Get(d[-j,k+2*j]))
    END FOR
  END FOR
END FOR
```

Figure 3.4: An Imperative Program

4. Find x_T where
$$\left(\begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\right)\begin{bmatrix} 1 \\ x \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$
This is true for all x.
5. No contradiction
6. Find x_T where
$$\begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ x \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$
This implies x=2. No inconsistency
7. 5 and 6 give consistent results for x
8. $T_g = 1$
9. $A_T = 0$
10. $T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}$
11. $L^b =$
$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & -2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix}$$
$U^b =$
$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -3 & -2 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -5 & 0 & 1 \end{bmatrix}$$
12. $\begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}\begin{bmatrix} i' \\ j' \\ k' \end{bmatrix} = \begin{bmatrix} i' \\ j' + i' \\ k' + 2j' \end{bmatrix}$

If the same procedure is applied to iterator k, $b = 3$, it is seen that it fails on the first step. $y_L = [-2, -2], y_U = [-3, -2], y_U \neq y_L$. So loop k is not a candidate for partitioning for load balancing.

As only loop j is in invariant form we derive the program given in figure 3.4. For the sake of clarity, the initialisation of variables is not shown in the examples in the remainder of this thesis.

Note that the constant terms in the loop are unaffected by the transformation. The array references are adjusted in accordance with step 12. No other loop depends on the new j loop, therefore parallelising and partitioning with respect to this loop will give a load balanced implementation.

3.2.5 Multiple Iterators

If each of the r parallel loops are successful candidates for load balancing, then there are potentially $2^r - 1$ permutations which may be used to partition the iteration space. To determine whether a combination of iterators can partition the iteration space in a load balanced manner, a modification of algorithm 1 is required. In this section we propose a new algorithm to construct a new transformation T'_s which has the combined effect of transforming each load balanced loop $j = i_1, \dots, i_s$ where $s \le r$ is the number of invariant loops and $i_1 < \cdots < i_s$ are the values of loops to be load balanced.

Let T_{i_k} be the individual transformation on a particular iterator k as given by algorithm 1. The first theorem below shows that given the transformation $T'_{\ell-1}$ that makes invariant all loops $i_1 < i_2 < .. < i_{\ell-1}$ and the transformations T_{i_ℓ} determined by Algorithm 1 for iterator i_ℓ alone, T'_ℓ can be constructed to include the new iterator i_ℓ . Two lemmas are required to prove this theorem. The first is a technical condition to aid the proof, the second ensures that the form of the transformation is legal.

T'_s is defined by a corollary to the main theorem and provides the basis for a simple algorithm to simultaneously make invariant all load balanced loops.

Lemma 1 *Given $T'_{\ell-1}$ such that $LT'_{\ell-1}, UT'_{\ell-1}$ is jointly invariant for $j \in i_1, \dots, i_{\ell-1}$ i.e.*

$$LT'_{\ell-1}e_{i_k} = UT'_{\ell-1}e_{i_k} = e_{i_k}\forall k \in 1, \dots, \ell-1 \tag{3.37}$$

$$e^T_{i_k}LT'_{\ell-1} = e^T_{i_k}UT'_{\ell-1} = e^T_{i_k}\forall k \in 1, \dots, \ell-1 \tag{3.38}$$

then $T'_{\ell-1}$ can be chosen so that:

$$T'_{\ell-1}e_{i_k} = e_{i_k}\forall k \ge \ell \tag{3.39}$$

Proof of Lemma 1 *This follows immediately by observing that the invariance conditions 3.37,3.38 impose constraints only on elements in the $m \times (\ell-1)$ sub-matrix of $T'_{\ell-1}$ and hence the $(m-\ell+1)\times(m-\ell+1)$ right hand corner sub-matrix of $T'_{\ell-1}$ can be set to identity without violating 3.37 and 3.38.* □

Theorem 1 *If T'_ℓ is defined as:*

$$T'_\ell = T'_{\ell-1} - e_{i_\ell}e^T_{i_\ell}LT'_{\ell-1} + T_{i_\ell}e_{i_\ell}e^T_{i_\ell} \tag{3.40}$$

the T'_ℓ is the transformation that satisfies the invariance condition for all load balanced loops $j \in i_1, \dots, i_\ell$ i.e.

$$LT'_\ell e_{i_k} = UT'_\ell e_{i_k} = e_{i_k}\forall k \in 1, \dots, \ell \tag{3.41}$$

$$e^T_{i_k}LT'_\ell = e^T_{i_k}UT'_\ell = e^T_{i_k}\forall k \in 1, \dots, \ell \tag{3.42}$$

The proof requires the following preliminary lemma.

Lemma 2 *T'_ℓ given in 3.40 is unit lower triangular.*

Only an outline of the proof is presented

Proof of Lemma 2 *Substitute for T'_ℓ from 3.40 and use 3.37 and 3.38 to show*

$$e_i^T T'_\ell e_j = 0 \; \forall i < j \tag{3.43}$$

and

$$e_i^T T'_\ell e_i = 1 \tag{3.44}$$

Proof of Theorem 1 *It is sufficient to show 3.41 and 3.42 for LT'_ℓ.*

step1. Show 3.41 and 3.42 is true $\forall k \in 1, \ldots, \ell - 1$ [Proof Omitted]

step2. Show 3.41 and 3.42 is true for $k = \ell$

$$LT'_\ell e_{i_\ell} = L(T'_{\ell-1} - e_{i_\ell} e_{i_\ell}^T LT'_{\ell-1} + T_{i_\ell} e_{i_\ell} e_{i_\ell}^T) e_{i_\ell} \tag{3.45}$$

By Lemma 1 $T'_{\ell-1} e_{i_\ell} = e_{i_\ell}$

$$LT'_\ell e_{i_\ell} = Le_{i_\ell} - Le_{i_\ell} e_{i_\ell}^T Le_{i_\ell} + LT_{i_\ell} e_{i_\ell} \tag{3.46}$$

As $e_{i_\ell}^T Le_{i_\ell} = 1$ and $LT_{i_\ell} e_{i_\ell} = e_{i_\ell}$ by invariance of i_ℓ for LT_{i_ℓ} then:

$$LT'_\ell e_{i_\ell} = Le_{i_\ell} - Le_{i_\ell} + e_{i_\ell} = e_{i_\ell} \tag{3.47}$$

Similarly:

$$e_{i_\ell}^T LT'_\ell = e_{i_\ell}^T LT'_{\ell-1} - e_{i_\ell}^T Le_{i_\ell} e_{i_\ell}^T LT'_{\ell-1} + e_{i_\ell}^T LT'_{i_\ell} e_{i_\ell} e_{i_\ell}^T \tag{3.48}$$

As $e_{i_\ell}^T LT_{i_\ell} = e_{i_\ell}^T$ and simplifying gives

$$e_{i_\ell}^T LT'_\ell = e_{i_\ell}^T e_{i_\ell} e_{i_\ell}^T = e_{i_\ell}^T \tag{3.49}$$

which is the invariance condition and thus T'_ℓ is the required transformation. □

It has been shown that T'_ℓ is the transformation that load balances ℓ iterators provided $T'_{\ell-1}$ is given. The following corollary constructs the transformation T'_s that load balances all s iterators.

Corollary 1 $T'_s = T'_{s-1} - e_{i_s} e_{i_s}^T LT'_{s-1} + T_{i_s} e_{i_s} e_{i_s}^T$ *is the transformation for joint invariance of $j \in i_1, \ldots, i_s$*

Proof of Corollary 1 *set $T'_1 = T_{i_1}$ and recursively apply theorem 1 for $k = 2, \ldots, s$*□

3.2.6 Algorithm 2

It is now possible to give a simple algorithm that uses the result of theorem 1 to construct a transformation that transforms all load balanceable loops into invariant form.

1. Apply Algorithm 1 to give the invariant iterators $j \in i_1, \ldots, i_s$ and the canonical transformations $T_{i_1}, \ldots, T_{i_s}$. If $s \leq 1$ Stop.
2. Set $T'_1 = T_{i_1}$

```
a := for i in  1,n
    returns array of
      for  j in -i+1,n-i
      cross k in  -1-i-j ,-j+3
      cross l in 1-i-2*j-k , 2*n-2*j-k-i
      cross m in 2*j+6*i+k-l-1, i-k-l+n
      returns value of sum
        2*(b[i,k] * c[k,l] + d[m,m-1])/b[j,j]
      end for
    end for
```

Figure 3.5: A Sisal Program

3. For $k \in 2, \dots, s$

$$T'_k = T'_{k-1} - e_{i_k} e^T_{i_k} L T'_{k-1} + T_{i_k} e_{i_k} e^T_{i_k} \tag{3.50}$$

Complexity: The extra cost of computing the transformation T'_s, step 3 of Algorithm 2 has an upper bound less than $O(s.{i_s}^2) \leq O(m^3)$. As Algorithm 1 has an upper bound complexity of $O(m^3)$ this new algorithm does not alter the overall complexity of the scheme. To illustrate this algorithm, consider the following slightly contrived example in figure 3.5. This example has been chosen to show that load balancing of parallel affine loops is non-trivial in more complex cases, such as when more than one iterator is a candidate for load balancing.

This loop nest has the following upper and lower bound matrices on J^5:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 1 & 0 \\ -6 & -2 & -1 & 1 & 1 \end{bmatrix} \quad l = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{3.51}$$

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 1 & 0 \\ -1 & 0 & 1 & 1 & 1 \end{bmatrix} \quad u = \begin{bmatrix} n \\ n \\ 3 \\ 2n \\ n \end{bmatrix} \tag{3.52}$$

On applying algorithm 1, j_2 and j_4 are the only candidates for load balancing, s=2. The corresponding transformation matrices for both iterators are as follows:

$$T_{j_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 & 1 \end{bmatrix} \tag{3.53}$$

$$T_{j_4} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -1 & -2 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.54}$$

To find the transformation $T_s', s = 2$ for joint invariance of j_2, j_4, apply step 2 of Algorithm 2.

$$T_1' = T_{j_2} \tag{3.55}$$

$$T_2' = T_1' - e_{j_4} e_{j_4}^T L T_1' + T_{j_4} e_{j_4} e_{j_4}^T \tag{3.56}$$

This gives

$$T_2' = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 1 & -1 & -1 & 1 & 0 \\ 0 & 2 & 0 & -1 & 1 \end{bmatrix} \tag{3.57}$$

Applying T_2' to L and U gives:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ -3 & 0 & -2 & 0 & 1 \end{bmatrix} \quad l = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{3.58}$$

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad u = \begin{bmatrix} n \\ n \\ 3 \\ 2n \\ n \end{bmatrix} \tag{3.59}$$

Note that rows and columns 2 and 4 are in invariant form for both upper and lower bounds. The array occurrences must be expressed with respect to the new iteration basis:

$$\begin{bmatrix} i \\ j \\ k \\ l \\ m \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 1 & -1 & -1 & 1 & 0 \\ 0 & 2 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \\ k' \\ l' \\ m' \end{bmatrix} \tag{3.60}$$

$$\begin{bmatrix} i \\ j \\ k \\ l \\ m \end{bmatrix} = \begin{bmatrix} i' \\ j' - i' \\ k' - j' \\ l' - k' - j' + i' \\ m' - l' + 2j' \end{bmatrix} \tag{3.61}$$

This give the transformed program of figure 3.6.

Although loops j and l are in load balanced form, they are not outermost. The next section describes a method whereby the loops may be always re-ordered so that they are outermost.

3.2.7 Reordering Iterators

Having determined which iterator(s) are to be used to partition the computation lattice, it may be desirable to have these iterators as far out as possible in the loop structure. In this section it is shown that load balanced iterators

```
FOR i = 1 TO n
  FOR j = 1 TO n
    FOR k = -1 TO i+3
      FOR l = 1 TO 2*n
        FOR m = 3*i+2*k -1 TO n
          a[i] := a[i] +2*(Get(b[i,k-j])*Get(c[k-j,l-k-j+i])
               + Get(d[m-l+2*j,m-l+2*j-1]))/ Get(b[j-i,j-i])
        END FOR
      END FOR
    END FOR
  END FOR
END FOR
```

Figure 3.6: An Imperative Program

can be moved to the outermost nest by a sequence of unimodular transformations, whilst preserving the affine structure of the loop.

Theorem 2 shows that one invariant iterator can be moved one nest level up by a unimodular transformation. By extending this result it is possible to move multiple iterators to the outermost nest by a succession of these unimodular transformations. To prove this theorem some preliminary definitions and lemmas are first required.

Let $E_{i,k}$ be the permutation of identity with row i and k interchanged. $E_{i,k}$ is unimodular, and $E_{i,k}{}^{-1} = E_{i,k}$. Interchange of iterator j_i with j_k can be represented as:

$$J' = E_{i,k}J \quad j_i, j_k \in J \tag{3.62}$$

Let L, U be in load balanced form with iterator $j_i \in J$ invariant. The following are defined:

$$L' = E_{i-1,i}LE_{i-1,i} \tag{3.63}$$

$$U' = E_{i-1,i}UE_{i-1,i} \tag{3.64}$$

$$J' = E_{i-1,i}J \tag{3.65}$$

$$l' = E_{i-1,i}l \tag{3.66}$$

$$u' = E_{i-1,i}u \tag{3.67}$$

Lemma 3 *L', U' are unit lower triangular.*

The importance of Lemma 3 is in establishing that the unimodular transformation described in 3.63 and 3.64 preserve the affine structure of the loop.

Proof of Lemma 3 *L, U are in load balanced form and j_i is the invariant iterator. Therefore*

$$e_i^T L e_{i-1} = e_i^T U e_{i-1} = 0 \tag{3.68}$$

As loop interchange is restricted to neighbouring iterators of j_i, in this case j_{i-1}, the only possible non-zeros in the upper triangular part of L' and U' are the $(i-1, i)$ elements. From 3.63 and 3.64 it can be seen that

$$e_{i-1}^T L' e_i = e_i^T L e_{i-1} = 0 \tag{3.69}$$

$$e_{i-1}^T U' e_i = e_i^T U e_{i-1} = 0 \tag{3.70}$$

The effect of the transformations in 3.63 and 3.64 on the diagonal elements of L and U is to interchange the (i, i) *and* $(i-1, i-1)$ *elements, which are both equal to one. This establishes that* L' *and* U' *are unit lower triangular.*

Lemma 4 *The iteration spaces represented by*

$$LJ \geq l \tag{3.71}$$

$$UJ \leq u \tag{3.72}$$

and

$$L'J' \geq l' \tag{3.73}$$

$$U'J' \leq u' \tag{3.74}$$

are equivalent.

Proof of Lemma 4 *By Lemma 3, 3.73 and 3.74 represent a legal affine loop. It remains to show the equivalence of system of inequalities 3.71, 3.72 and 3.73, 3.74.* $E_{i-1,i}E_{i-1,i} = I$ *in 3.75 and 3.76 which preserves the system of inequalities.*

$$LE_{i-1,i}E_{i-1,i}J \geq l \tag{3.75}$$

$$UE_{i-1,i}E_{i-1,i}J \leq u \tag{3.76}$$

Now we substitute from 3.65:

$$LE_{i-1,i}J' \geq l \tag{3.77}$$

$$UE_{i-1,i}J' \leq u \tag{3.78}$$

Now multiply both sides of inequalities in 3.77 and 3.78 by $E_{i-1,i}$. This amounts to reordering the inequalities, thus preserves the iteration space. Substitute from 3.63 to 3.67 to give:

$$L'J' \geq l' \tag{3.79}$$

$$U'J' \leq u' \square \tag{3.80}$$

Theorem 2 *Let* L' *,*U' *be defined as in 3.63 and 3.64, then* $j_{i-1} \in J'$ *is an invariant iterator for* L', U'*.i.e.*

$$L'e_{i-1} = U'e_{i-1} = e_{i-1} \tag{3.81}$$

and

$$e_{i-1}^T L' = e_{i-1}^T U' = e_{i-1}^T \tag{3.82}$$

Proof of Theorem 2 *It suffices to show 3.81 and 3.82 for* L'*. By assumption,* j_i *is an invariant iterator for* L, U*. Thus*

$$Le_i = e_i \tag{3.83}$$

$$e_i^T L = e_i \tag{3.84}$$

By definition

$$E_{i-1,i}e_{i-1} = e_i \tag{3.85}$$

$$e_{i-1}^T E_{i-1,i} = e_i^T \tag{3.86}$$

Substitute 3.85 in 3.83:

$$LE_{i-1,i}e_{i-1} = E_{i-1,i}e_{i-1} \tag{3.87}$$

Multiply both sides of 3.87 by $E_{i-1,i}$ and substitute from 3.63:

$$L^{'}e_{i-1} = e_{i-1} \tag{3.88}$$

Similarly, substitute 3.86 in 3.84:

$$e_{i-1}^{T}E_{i-1,i}L = e_{i-1}^{T}E_{i-1,i} \tag{3.89}$$

Multiply both sides of 3.89 by $E_{i-1,i}$ and substitute from 3.63:

$$e_{i-1}^{T}L^{'} = e_{i-1}^{T} \tag{3.90}$$

which is the invariant condition and thus j_{i-1} is an invariant iterator. □

Theorem 2 shows that the new transformed iterators have any one load balanced loop one loop nest further out than before.

Observation 1 *Similarly,it can be shown that any load balanced iterator j_i can be moved one loop nest further in by applying the permutation transformation $J^{'} = E_{i,i+1}J$. Thus two neighbouring iterators that are both load balanced will remain so upon interchange*

Given the set of iterators J_B where iterators $j = i_1, i_2, \ldots, i_s$ are the values of the iterators in load balanced form for

$$L^B J_B \geq l \tag{3.91}$$

$$U^B J_B \leq u \tag{3.92}$$

It is necessary to find a unimodular transformation E such that:

$$J_o = EJ_B \tag{3.93}$$

and J_o is the iteration vector with the first s iterators load balanced. Let E^i be the transformation that moves a particular iterator j_i to the outermost scope. It is defined thus:

$$E^i = E_{1,2} \times E_{2,3} \times \cdots \times E_{i-1,i} \tag{3.94}$$

It should be noted that in general

$$E^i \not\equiv E_{1,i} \tag{3.95}$$

E is now defined as:

$$E = E^{i_s} \times E^{i_{s-1}} \times \cdots \times E^{i_1} \tag{3.96}$$

E is unimodular as it is the product of unimodular transformations. Let

$$L^o = EL^B E^{-1} \tag{3.97}$$

$$U^o = EU^B E^{-1} \tag{3.98}$$

$$l^o = El \tag{3.99}$$

$$u^o = Eu \tag{3.100}$$

L^o, U^o, l^o, u^o are in the canonical form. This is shown by repeated application of lemma 3. The set of inequalities given by:

$$L^o J_o \geq l^o \tag{3.101}$$

$$U^o J_o \leq u^o \tag{3.102}$$

are equivalent to 3.91 and 3.92. This can be shown by observing that:

$$EL^B(E^{-1}E)J_B \geq El \tag{3.103}$$

$$EU^B(E^{-1}E)J_B \leq Eu \tag{3.104}$$

$$(EL^B E^{-1})(EJ_B) \geq El \tag{3.105}$$

$$(EU^B E^{-1})(EJ_B) \leq Eu \tag{3.106}$$

Substituting from 3.97 to 3.100 gives the required form of 3.101 and 3.102.

Finally, by the repeated application of theorem 2 and using observation 1, it can be shown that the first s iterators of J_o are load balanced for 3.101 and 3.102.

This analysis gives the following algorithm to reorder the iterators.

3.2.8 Algorithm 3

```
1. For $\ell \in 1, \ldots, s$
2. For $k \in i_\ell$ to 2 step -1
3. Interchange rows $k$ and $k-1$ of $U, L, l, u$
4. Interchange columns $k$ and $k-1$ of $U, L$, rows $k$ and $k-1$ of J
5. End For
6. End For
```

To illustrate this algorithm consider the matrix form of the program given in figure 3.4 after transforming to invariant form.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 1 \\ n-1 \\ 1 \end{bmatrix} \tag{3.107}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -5 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \begin{bmatrix} n \\ 2n+1 \\ n \end{bmatrix} \tag{3.108}$$

There is only one loop j to move out, i.e. $s = 1, i = 2, j_i = j_2$ On interchanging rows we have

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ -4 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} n-1 \\ 1 \\ 1 \end{bmatrix} \tag{3.109}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ -5 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \begin{bmatrix} 2n+1 \\ n \\ n \end{bmatrix} \tag{3.110}$$

```
FOR j = n-1 TO 2*n+1
  FOR i = 1 TO n
    FOR k = 1+4*i TO  n+5*i
      a[i] := a[i] + (Get(c[i,k+2*j])*Get(d[-j,k+2*j]))
    END FOR
  END FOR
END FOR
```

Figure 3.7: An Imperative Program

Interchanging columns of U, L and rows of J gives

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -4 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \\ k \end{bmatrix} \geq \begin{bmatrix} n-1 \\ 1 \\ 1 \end{bmatrix} \tag{3.111}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -5 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \\ k \end{bmatrix} \leq \begin{bmatrix} 2n+1 \\ n \\ n \end{bmatrix} \tag{3.112}$$

which finally gives the program in figure 3.7.

This program has a load balanced loop which is outermost. Each iteration of j will have exactly the same amount of work to perform and all that is now required is for the $n+3$ iterations to be divided amongst the processors.

3.3 General Polytopes

The previous section has described a method for determining a load balanced form (if it exists) of nested affine loops with no IF expressions. This section addresses a wider class of computation set which includes conditional evaluation. It is shown that a sub-set of the polytopes can be transformed into a condition free form which is then amenable to the analysis developed in the previous section. In order to remove conditionals, iterator reordering is employed which necessitates the addition and removal of constraints. The condition for successful translation to a condition free polytope is presented. An interesting product of this approach is that it includes all the loop interchange results described by Wolfe [WOLF91] who uses a functional approach. Finally, the translation of a polytope into upper and lower bound constraint matrices is defined, whereupon it may be tested for load balancing purposes.

3.3.1 Polytope Form

The **canonical** form of a bounded affine polytope

$$AJ \leq b \tag{3.113}$$

is defined as:

```
a := for i in 1,n cross j in 1,n
     returns array of
       if (i <= j)
       then  b[i]*b[j]
       else  0
       end if
     end for
```

Figure 3.8: A Sisal Program

$$\begin{bmatrix} -L \\ \hline U \\ \hline \mathcal{E} \end{bmatrix} J \leq \begin{bmatrix} -l \\ \hline u \\ \hline \varepsilon \end{bmatrix} \tag{3.114}$$

where A is a $\ell \times m$ integer matrix, b a $\ell \times 1$ vector and $\mathcal{E}, \varepsilon$ are a matrix/vector pair consisting of extra constraints due to loop body conditionals.

To transform this polytope so as to describe an affine form, it is required that the additional constraints be removed. The method applied in this section is to reorder the iterators and in the process of removing redundant constraints, examine if $\mathcal{E} = 0, \varepsilon = 0$, i.e. there are no conditionals. It is possible to determine the conditions under which such a constraint may be removed.

The ability to reorder the iterators of a general polytope is of greater applicability than searching for a load balanced form. In chapter 5, it is used in a pre-fetching transformation.

3.3.2 Reordering Iterators

In this section, the reordering of iterators is used to form a new polytope. However in general such reordering introduces additional constraints. Rules are derived whereby spurious conditions may be safely removed. By a combination of these techniques, it is possible to transform some affine loops into a representation without any conditionals, which is then amenable to perfect load balance analysis. Consider the program in figure 3.8, this has the following polytope form describing the computation $b[i] * b[j]$:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \\ \hline n \\ n \\ \hline 0 \end{bmatrix} \tag{3.115}$$

which is in the canonical form described in 3.114. Before iterators i and j may be interchanged, it is necessary to re-formulate equations 3.63 to 3.67 for polytopes. To interchange an iterator at position r with the preceding iterator to give a new polytope we have:

$$\begin{bmatrix} -L' \\ U' \\ \mathcal{E}' \end{bmatrix} = \begin{bmatrix} E_{r-1,r} & 0 & 0 \\ 0 & E_{r-1,r} & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} -L \\ U \\ \mathcal{E} \end{bmatrix} E_{r-1,r} \tag{3.116}$$

```
FOR j = 1 TO n
  FOR i = 1 TO n
    IF (i <= j)
    THEN  a[i,j] := Get(b[i]) * Get(b[j])
    ELSE a[i,j] := 0
    END IF
  END FOR
END FOR
```

Figure 3.9: An Imperative Program

```
a := for i in 1,n
     returns array of
       for j in i,n
       returns value of sum
         b[i]*b[j]
       end for
     end for
```

Figure 3.10: A Sisal Program

$$J' = E_{r-1,r}J \tag{3.117}$$

$$\begin{bmatrix} -l' \\ u' \\ \varepsilon' \end{bmatrix} = \begin{bmatrix} E_{r-1,r} & 0 & 0 \\ 0 & E_{r-1,r} & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} -l \\ u \\ \varepsilon \end{bmatrix} \tag{3.118}$$

Where I is the $(\ell - 2m) \times (\ell - 2m)$ Identity matrix. On applying this to 3.115 we have

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline -1 & 1 \end{array}\right] \begin{bmatrix} j \\ i \end{bmatrix} \leq \left[\begin{array}{c} -1 \\ -1 \\ \hline n \\ n \\ \hline 0 \end{array}\right] \tag{3.119}$$

which gives the imperative program in figure 3.9.

However unlike load balanced iterators in the previous section, it is not always possible to simply interchange iterators in this way. For instance consider the Sisal program in figure 3.10 and its associated polytope.

$$\left[\begin{array}{cc} -1 & 0 \\ 1 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \begin{bmatrix} i \\ j \end{bmatrix} \leq \left[\begin{array}{c} -1 \\ 0 \\ \hline n \\ n \end{array}\right] \tag{3.120}$$

After column and row interchanging we have the following form:

$$\left[\begin{array}{cc} -1 & 1 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} j \\ i \end{array}\right] \leq \left[\begin{array}{c} 0 \\ -1 \\ \hline n \\ n \end{array}\right] \tag{3.121}$$

Unfortunately this is not in the canonical form as the first row does not have a 0 as its second element to restore the lower triangularity. This is the subject of the next section.

3.3.3 Constraint Addition

Constraints are added to maintain the canonical form of the polytope. The canonical form, in general, is destroyed after iterator interchange which introduces non-zeros above the two diagonals. By replacing a constraint which violates the canonical form with an appropriate one and moving the non-canonical constraint to the $\mathcal{E}, \varepsilon$ portion of the polytope, it is possible to maintain the correct form.

This substitution involves the creation of a new constraint. A new constraint that can always be legally added is one which is redundant, i.e. a positive linear combination of 2 or more existing constraints.

Let

$$B = \left[\begin{array}{ccc} E_{r-1,r} & 0 & 0 \\ 0 & E_{r-1,r} & 0 \\ 0 & 0 & I \end{array}\right] AE_{r-1,r} \tag{3.122}$$

$$c = \left[\begin{array}{ccc} E_{r-1,r} & 0 & 0 \\ 0 & E_{r-1,r} & 0 \\ 0 & 0 & I \end{array}\right] b \tag{3.123}$$

After row and column interchange, it is possible that non-zero terms will appear in the elements $B_{r-1,r}, B_{m+r-1,m+r}$. In the following discussion the case that a non-zero may have appeared in $B_{r-1,r}$ is examined. The argument is trivially extended for the $B_{m+r-1,m+r}$ case.

Let $\alpha = B_{r-1,r}$ be the potential above diagonal non-zero.

Let d be a $1 \times m$ row vector to be known as the desired row constraint such that

$$d_y = \begin{cases} x_y & y < r-1 \\ 1 & y = r-1 \\ 0 & y > r-1 \end{cases} \tag{3.124}$$

where x_y is any integer. In other words d is of the same general form as row $r-1$ except that α is set to zero. Substituting this row d in B is only legal if a positive linear combination of two or more other constraint rows can form such a row, i.e. it is redundant and describes precisely the same polytope. In fact it can be shown that d is the positive combination of two rows of B such that:

$$d = \begin{cases} B_{r-1} + \alpha B_r & \alpha > 0 \\ B_{r-1} - \alpha B_{m+r} & \alpha < 0 \end{cases} \tag{3.125}$$

In the case $\alpha = 0$ no constraint need be added. The corresponding right hand constraint value f is easily derived,

$$f = \begin{cases} c_{r-1} + \alpha c_r & \alpha > 0 \\ c_{r-1} - \alpha c_{m+r} & \alpha < 0 \end{cases} \tag{3.126}$$

The new constraint matrix $\ell + 1 \times m$ matrix C, in canonical form, is defined as

$$C_{x,y} = \begin{cases} B_{x,y} & x \neq r-1 \wedge x \neq \ell + 1 \\ d_y & x = r-1 \\ B_{r-1,y} & x = \ell + 1 \end{cases} \tag{3.127}$$

In other words C is the same as B except that the row containing the extra non-zero element is moved to the $\mathcal{E}$ region of the polytope and a legal constraint row replaces it. The new constraint vector g is similarly constructed.

$$g_y = \begin{cases} c_y & y \neq r-1 \wedge y \neq \ell + 1 \\ f & y = r-1 \\ c_{r-1} & y = \ell + 1 \end{cases} \tag{3.128}$$

As an example consider the polytope 3.121. The first row is $[-1, 1]$ when $[-1, 0]$ is needed. The combination of row 1 + row 2 is the necessary combination. This gives $f = -1$. On replacing row 1 by $[-1, 0]$, the first element of the constraint vector by -1 and moving the the old row constraint to the $\mathcal{E}$ region, we have the following polytope which is in the canonical form.

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline -1 & 1 \end{array} \right] \left[\begin{array}{c} j \\ i \end{array} \right] \leq \left[\begin{array}{c} -1 \\ -1 \\ \hline n \\ n \\ \hline 0 \end{array} \right] \tag{3.129}$$

3.3.4 Removing Redundant Conditions

In this section it is assumed that the polytope is in canonical form at all times, i.e. the procedure described in the previous section is applied after iterator interchange. The section describes a procedure to find redundant conditions which can be formed or duplicated by a linear combination of the others.

Firstly Ξ, an $(\ell \times \ell)$ elimination matrix, is created where each $\Xi_{x,y}$ entry represents whether a positive linear combination of the row constraints x and y of the polytope would legally replace an existing one. The elements of this matrix contain the value null ($\perp$) if no such combination exists and the value z if it does exist, where z is the redundant row condition in the polytope. As it is symmetric, only the lower triangular portion need be calculated .

This matrix is calculated by starting with the first condition adding a linear combination of each of the remaining constraints one at a time to see if it duplicates another constraint. If this is so then it is recorded in the elimination matrix. This is repeated for all the constraints. In other words, does there exist an z such that:

$$\left[\begin{array}{cc} r_1 & r_2 \end{array} \right] \left[\begin{array}{c} c_x \\ c_y \end{array} \right] = \left[\begin{array}{c} c_z \end{array} \right] \forall x \in 1, \ldots, \ell, y \in x+1, \ldots, \ell \tag{3.130}$$

where r_1 and r_2 are positive integers, c_x, c_y and c_z are rows of the canonically formed C constraint matrix. and

$$\left[\begin{array}{cc} r_1 & r_2 \end{array} \right] \left[\begin{array}{c} g_x \\ g_y \end{array} \right] = \left[\begin{array}{c} g_z \end{array} \right] \forall x \in 1, \ldots, \ell, y \in x+1, \ldots, \ell \tag{3.131}$$

where g_x, g_y and g_z are rows of the canonically formed c constraint vector. This is equivalent to solving for r_1, r_2 in an overdetermined system of equations. An additional requirement is that one of the two constraints must replace the redundant constraint whilst still preserving the canonical form of the polytope. If it cannot, then the redundant condition must be maintained. Let *Nz*() be a function that returns the number of right-hand zero elements in a row vector of a constraint matrix and *Right*() the value of the right most non-zero term. Thus assuming 3.130 and 3.131 are satisfied then:

$$\Xi_{x,y} = z, if(Nz(c_x) = Nz(c_z)) \wedge (Right(c_x) = Right(c_z)) \wedge (x > 2m) \tag{3.132}$$

$$\Xi_{x,y} = z, if(Nz(c_y) = Nz(c_z)) \wedge (Right(c_y) = Right(c_z)) \wedge (y > 2m) \tag{3.133}$$

$$\Xi_{x,y} = \perp, otherwise \tag{3.134}$$

In other words it is only a valid entry if it not only satisfies equations 3.130 and 3.131 but there is also a row that may replace it with the same number of right hand zeros and the same sign of the diagonal term as the one that is redundant. This is to preserve lower triangularity in accordance with the canonical form. If these conditions do not hold then the constraint must be retained.

When all combinations of rows have been examined the following system of equations will be formed

$$V\gamma = \delta \tag{3.135}$$

where γ is a $(m \times 1)$ vector variable corresponding to each row of the constraint matrix C and δ a $(s \times 1)$ vector variable whose s elements represent the s redundant constraints in C. V is an $(s \times m)$ integer matrix defining the linear combination of redundant rows. This can be re-expressed as:

$$W\gamma = 0 \tag{3.136}$$

A negative entry in a row of W implies that the corresponding condition in C is redundant. Furthermore the positive entries in that same row of W, if combined, make the negative entry redundant. Before a redundant condition can be removed, all references to it must be adjusted. For example if row 3 of C is a redundant condition
row 3 = row 4 + row 5
and so is row 6
row 6 = row 3 + row 9
Then, if row 3 is removed then its left hand side is substituted in row 6 to give:
row 6 = row 4 + row 5 + row 9

Once all possible conditions of C have been eliminated, each condition corresponding to a remaining negative entry in W may be replaced by any other condition corresponding to a positive entry as long as the new condition preserves lower triangularity. Thus if the following condition is left.
row 4 = row 2 + row 7
row 4 can only be removed if either row 2 or row 7 can be placed in the fourth row of C without ruining the canonical form of the polytope If two rows to be eliminated both require the same condition, then either one is chosen arbitrarily. So if
row 3 = row 2 + row 8
row 7 = row 4 + row 8
and only row 8 can replace row 3 or row 7 but not both, then a decision is made arbitrarily. Returning to 3.129, this has the following elimination matrix:

$$\begin{bmatrix} \perp & \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp & \perp \\ \perp & \perp & \perp & \perp & \perp \\ \perp & \perp & 4 & \perp & \perp \end{bmatrix} \tag{3.137}$$

```
FOR j =  1 TO n
  FOR i = 1 TO j
    a[j] :=  a[j] + Get(b[i])* Get(b[j])
  END FOR
END FOR
```

Figure 3.11: An Imperative Program

```
a := for i in 1,n
     returns array of
       for j in 1,n cross k in 1,n
       returns value of sum
         if (i>=j) & (i>=k) & (j >=k)
         then b[i,k]*c[k,j]
         else 0
         end if
       end for
     end for
```

Figure 3.12: A Sisal Program

There is just one duplicated constraint:
row 3 + row 5 = row 4
Row 4 can be replaced by row 5 maintaining the canonical form. Although row 1 is a combination of row 4 and row 2 and strictly redundant, it cannot be removed as it would destroy the canonical form. As there is only one row to be removed it is not necessary to create the system of equations $W\gamma = 0$. In the two examples following the definition of algorithm, its use will be demonstrated. After removing the spurious condition we have the following polytope form and the corresponding imperative program in figure 3.11 which has maintained its affine form after reordering, with loops *i* and *j* interchanged.

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ -1 & 1 \end{array}\right] \left[\begin{array}{c} j \\ i \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ \hline n \\ 0 \end{array}\right] \qquad (3.138)$$

3.3.5 Example

This fourth algorithm is given appendix c, to illustrate the use of this algorithm consider the Sisal program in figure 3.12. To illustrate the elimination phase we will first remove any spurious constraints (**if** conditions) from the program as it stands i.e. from step 9 onwards. This is followed by an loop interchange of *i* and *j* obeying all steps of the algorithm.

Program 3.12 has the following polytope form

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \le \begin{bmatrix} -1 \\ -1 \\ -1 \\ \hline n \\ n \\ n \\ \hline 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.139}$$

which is of the form $CJ \le g$. Following the steps up to 12 gives:
12.
row 5 = row 4 + row 7
row 6 = row 4 + row 8
row 6 = row 5 + row 9
row 8 = row 7 + row 9
giving $W\gamma = 0$i.e.
13.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} \gamma_1 \\ \vdots \\ \gamma_9 \end{bmatrix} = 0 \tag{3.140}$$

14. $W_{1,5} = -1$
15. $W_{3,5} = 1$
16. Add W_1 to W_3, remove W_1 remove C_5, giving a W:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & 1 \end{bmatrix} \tag{3.141}$$

Repeating the above gives: Remove C_8 and

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 1 \end{bmatrix} \tag{3.142}$$

20. Replace C_6 by C_9 and f_6 by f_9
This then leaves the following polytope

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \le \begin{bmatrix} -1 \\ -1 \\ -1 \\ \hline n \\ 0 \\ 0 \end{bmatrix} \tag{3.143}$$

```
FOR j = 1 TO n
  FOR i = j TO n
    FOR k = 1 TO j
      a[i] := a[i] + Get(b[i,k]) * Get(c[k,j])
    END FOR
  END FOR
END FOR
```

Figure 3.13: An Imperative Program

```
a := for i in 1,2*n-1
     returns  array of
       for j in i-n+1 ,i
       returns  array of
         if (j >=1) & (j<= n)
         then
              b[i]*c[j]
         else 0 end if
       end for
     end for
```

Figure 3.14: A Sisal Program

If algorithm 3 is now repeated from the beginning such that iterators *i* and *j* are interchanged then the imperative program 3.13 can be derived.

Notice that the **if** expression has been removed at the expense of variable loop bounds on *j* and *i*. To show the usefulness of such a procedure for load balancing consider the program in figure 3.14 where the existence of a load balanced partition is not obvious. The polytope, corresponding to the first branch of the **if** expression, is as follows:

$$\begin{bmatrix} -1 & 0 \\ 1 & -1 \\ \hline 1 & 0 \\ -1 & 1 \\ \hline 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \le \begin{bmatrix} -1 \\ n-1 \\ \hline 2*n-1 \\ 0 \\ \hline -1 \\ n \end{bmatrix} \tag{3.144}$$

At present the computation is not in a load balanced form. However on loop interchange the polytope is of the following form:

$$\begin{bmatrix} -1 & 0 \\ 1 & -1 \\ \hline 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \le \begin{bmatrix} -1 \\ -1 \\ \hline n \\ n \end{bmatrix} \tag{3.145}$$

Since $\mathcal{E} = 0, \varepsilon = 0$, load balancing analysis can be performed which gives after transformation the program in figure 3.15.

```
FOR j = 1 TO n
  FOR  i = 1 TO n
    a[i+j,j] := Get(b[i+j])* Get(c[j])
  END FOR
END FOR
```

Figure 3.15: An Imperative Program

This program may be partitioned along iterator i or j to give a load balanced implementation. This example illustrates the use of iterator interchange. In the original Sisal program it is far from obvious that a perfectly load balanced partitioning can be found. After re-ordering, whenever $\mathcal{E} = 0, \varepsilon = 0$, the polytope is in a form to apply invariancy analysis.

3.4 Interleaving

Interleaving is unlike the previous transformations described in this chapter in that it does not alter the order of the indices or their ranges. It reorders the **values** that the iterator "passes" through. For example if an iterator j passes through the values $1, 2, 3, 4, 5, 6, 7, 8$, then a particular interleave function gives the following order $1, 5, 2, 6, 3, 7, 4, 8$. The form of the interleave function is given in chapter 5, section 5.1.3.

A very good analysis of the properties of the interleave or scatter decomposition is given in [NICO90] where they show that in general it improves the distribution of work. When no other transformation can be found, it is a useful method to improve load balancing.

Interleaving performs badly in the presence of periodic computation sets if the period coincides with the number of processors. As all polytopes are convex, this is not a problem. A useful heuristic is that, once the best combination of iterators has been determined, assuming no perfectly balanced form, then the interleave transformation may be applied to each iterator when mapping the iterators to the processors. This is dealt with more fully in chapter 5.

3.5 Summary

In this chapter the issue of load balancing has been addressed. After describing it as an optimisation problem, a method was presented to determine whether there exists a partitioning of the iterators which gives perfect load balancing. This was initially developed for conditional free polytopes and later extended to include **if** statements. In order to reveal invariancy in general polytopes, a method whereby iterators may be reordered was given In the case where no load balanced form exists, interleaving may be used.

Chapter 4

Alignment

This chapter is concerned with the relative alignment of arrays. The alignment and subsequent data partitioning will determine the amount of non-local access exhibited by a program.

The first section describes the alignment problem and derives two measures of alignment. The second section develops a procedure whereby alignment can be enhanced when compiling for creation parallelism. The third section extends the alignment transformation to compiling for reduction parallelism and in the fourth section, the effect of a local alignment transformation on the whole program is defined. In the fifth section, the interaction between alignment and partitioning is described and, finally, a summary concludes this chapter.

4.1 Alignment

4.1.1 Identification of Partitions

Compilation should minimise parallel time by firstly utilising machine parallelism and secondly reducing overhead. When mapping array computation to the processors, it is necessary to evenly distribute the array data. Ideally, this should be done in a manner which reduces non-local access. As there are many ways to map any one array on to an array of processors, this is, in general, a complex problem.

To simplify this process it is convenient to break it into two sub-problems [LI90] and [FOX91]. All arrays are first mapped to one common array which is then partitioned across the processors. This chapter covers the first stage, mapping the array data to a common array, known as alignment, which is concerned with the relative orientation of arrays appearing in a computation set .

Alignment determines which portions of two arrays will be in the same processor for a particular data partition. Obviously if they are aligned in such a manner that both portions are involved in the same computation, then this will reduce the number of non-local accesses that would have been incurred if they were in separate processors. Alignment is concerned with the relative allocation of arrays so as to maximise local accesses.

In this chapter, the question of aligning the arrays is focused upon those occurring in a particular computation set. Because there are many computation sets in a program, and as a computation set has just one array creation

```
a := for i in 1,n
    returns array of
      b[i+1]
    end for
```

Figure 4.1: A Sisal Program

associated with it, the analysis is correspondingly local in nature. In [LI90], the global determination of alignment is shown to be NP-complete so a heuristic method which gives good performance was used. This heuristic could also be used with the local alignment transformations developed in this chapter. In [LI90] however only simple aligning functions are considered. This chapter describes alignment as the intersection of a hyperplane with the iteration space and considers a greater class of alignment transformations within a linear algebraic framework. Thus the contributions of this chapter are the formal description of alignment and the introduction of new framework which includes previous results from the Crystal project [LI90]and the Compass compiler [KNOB90] and introduces new alignment transformations.

Only static allocation schemes are addressed, so the issue of redistribution of data is not addressed. By disregarding the issue of data redistribution, it is not necessary to investigate the effect of reordering the computation sets (whilst preserving data dependence) to exploit the new data allocation. In any allocation scheme, however, once a particular alignment of arrays has been decided, the effect of such an alignment must be propagated throughout the program to preserve meaning. This is dealt with in the section 4 of this chapter.

4.1.2 Example

To illustrate the effect of alignment consider the array occurrences of the program in figure 4.1. The *i*th element of a makes reference to $i+1$th element of array b:

$$a[i] = b[i+1] \tag{4.1}$$

If each $a[i], b[i+1]$ were stored in the same processor, then no non-local access would be required. In this case if b were to be allocated such that it were shifted one place to the left, relative to a, then no matter how the iterator i was partitioned, and hence how a and b were partitioned, they would both reside in the same processor. The occurrence of iterators in each array reference exactly determines the alignment required. To align b with a, a function π which shifts b is needed

$$\pi : b[i+1] \mapsto b^{'}[i] \tag{4.2}$$

or, more usually,

$$\pi : b[i] \mapsto b^{'}[i-1] \tag{4.3}$$

This idea can easily be extended to higher dimension data spaces. Consider the Sisal fragment in figure 4.2 where:

$$a[i,j] = b[j,i] \tag{4.4}$$

Here for 0 non-local access, it is necessary, to store a and b transposed relative to each other which gives the following alignment function:

$$\pi : b[i,j] \mapsto b^{'}[j,i] \tag{4.5}$$

```
a := for i in 1,n cross j in 1,n
     returns array of
       b[j,i]
     end for
```

Figure 4.2: A Sisal Program

```
a := for i in 1,n cross j in 1,n
     returns array of
       for k in 1,n
       returns value of sum
            b[i,k]
       end for
     end for
```

Figure 4.3: A Sisal Program

4.1.3 Hyperplanes of Alignment

In this section a formal description of alignment is presented. A computation set, Q, consists of, amongst other things, a computation lattice, $Latt(A.b)$, and a set of occurrence matrices, $(\mathcal{C}, \mathcal{U})$.

Let v be any array variable with a $(N_v \times m)$ occurrence matrix, $\mathcal{X}$, and $(N_v \times 1)$ vector x pair and w be another array variable with a $(N_w \times m)$ occurrence matrix $\mathcal{Y}$ and $(N_w \times 1)$ vector y pair where v and w are referenced in the same computation set Q . The occurrences $(\mathcal{X}, x), (\mathcal{Y}, y)$ define how v, w are referenced.

If we assume a common embedding of each array within the lattice $Latt(A.b)$, then an element of array w is aligned with an element of array v if they are referenced and embedded at the same computation point in the lattice.

Definition 12 *An alignment between an element of array w and v with occurrences $(\mathcal{X}, x)$ and $(\mathcal{Y}, y)$, occurs at J_1 iff $\mathcal{X}J_1 + x = \mathcal{Y}J_1 + y$ where J_1 is a value of J and $J_1 \in Latt(A.b)$.*

In general there is more than one point of alignment between two arrays. The system of equations

$$\mathcal{X}J_1 + x = \mathcal{Y}J_1 + y, J_1 \in Z^m \tag{4.6}$$

form a hyperplane denoted by $S_{\mathcal{X},\mathcal{Y}}$. The intersection, $S_{\mathcal{X},\mathcal{Y}} \cap Latt(A.b)$, defines all the points in the iteration space where v and w are aligned.

For example, consider the Sisal program in figure 4.3. Alignment occurs between arrays a and b when $i = i$ and $j = k$. The first condition always holds, whilst there is just one solution to the second condition for each value of j. These conditions define a 2 dimensional plane in the three dimensional iteration space spanned by $[i, j, k]^T$.

The object of compiling for alignment can now be stated as finding a transformation (T_i, t_i) where $S_{\mathcal{X},T_i\mathcal{Y}}$ is defined as

$$S_{\mathcal{X},T_i\mathcal{Y}} = \{J \in Z^m, \mathcal{X}J + x = T_i\mathcal{Y}J + t_i + y\} \tag{4.7}$$

such that $|S_{\mathcal{X},T_i\mathcal{Y}} \cap Latt(A.b)|$ is maximised.

```
a := for i in 1,n cross j in 1,n
    returns array of
      b[j]
    end for
```

Figure 4.4: A Sisal Program

Even for pairwise alignment the above optimization problem, in general, involves enumerating the polytope, $S_{\mathcal{X},T_i\mathcal{Y}} \cap Latt(A.b)$, for each candidate, (T_i, t_i), which can be an expensive task. To maximise alignment would require the determination of the volume of the polytope in equation 4.8

$$\begin{bmatrix} A \\ \mathcal{X} - T_i\mathcal{Y} \\ T_i\mathcal{Y} - \mathcal{X} \end{bmatrix} J^m \leq \begin{bmatrix} b \\ t_i + y - x \\ x - t_i - y \end{bmatrix} \tag{4.8}$$

for each transformation (T_i, t_i). Determining the volume of a polytope has been described in section 3.1.3 and at present the best algorithm has a complexity of $O(|T|m^{19})$ where $|T|$ is number of transformations considered. Even if the m^{19} term is decreased, there are potentially an infinite number of transformations to consider. The approach used in this thesis is to use an approximate metric of alignment, the hit function, which leads to the derivation of transformations to improve alignment

4.1.4 Hit Function

Before the new metric is introduced, it is necessary to look at the problem of aligning arrays with a differing number of dimensions. In general there are difficulties if $N_v > N_w$ or $N_w < N_v$, i.e. the number of occurrences in one matrix is greater than the other. For example, consider the program in figure 4.4 which has the following occurrence matrices:

$$\mathcal{X} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathcal{Y} = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{4.9}$$

Whilst it is desirable that b is aligned with the second dimension of a, there is no way to represent this in a 1×2 matrix; one solution is to use a 2×2 matrix.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.10}$$

where a zero row corresponds to a null-occurrence. By adding an extra row of zeros, it is possible to convey in this example, that b is to be aligned with a on the second dimension. To allow alignment between arrays of different dimensionality, for the remainder of this chapter all array occurrences $\mathcal{X}$ are defined as being of size $M \times m$ where $M = \max(m, N_v, N_w)$ and:

$$\begin{array}{ccccc} \hat{\mathcal{X}}^T & = & [\mathcal{X}^T, & & O] \\ M \times m & & N_v \times m & & M - N_v \times m \end{array} \tag{4.11}$$

$\hat{\mathcal{X}}$ will be used throughout the remainder of the chapter and will be referred to as $\mathcal{X}$.

The function used to determine the alignment of arrays is called the hit function. Essentially it compares each sub-script of the aligning array for equality. The value of the hit function is the number of equal rows.

```
a := for i in 1,n
     returns array of
       for k in 1,i
       returns value of sum
         b[k]
       end for
     end for
```

Figure 4.5: A Sisal Program

```
a := for i in 1, n
     returns array of
       for k in 1,i-1
       returns value of sum
         b[k]
       end for
     end for
```

Figure 4.6: A Sisal Program

Definition 13 *The hit function H is defined as:*

$$H_{\mathcal{X},\mathcal{Y}} = \sum_{k=1}^{M} \delta_{\mathcal{X}_k,\mathcal{Y}_k} \times \delta'_{x_k,y_k} \tag{4.12}$$

$$\delta_{e,f} = \begin{cases} 1 & e = f \wedge e \neq 0 \\ 0 & otherwise \end{cases} \quad \delta'_{e,f} = \begin{cases} 1 & e = f \\ 0 & otherwise \end{cases} \tag{4.13}$$

For example consider the following array occurrences.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_v + \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix}_v , \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_w + \begin{bmatrix} 1 \\ 4 \\ 0 \end{bmatrix}_w \tag{4.14}$$

Here $H = 1$, which is the number of perfectly aligned sub-scripts of v and w, and is equal to the dimension of the hyperplane of alignment, $dim(S_{\mathcal{X},\mathcal{Y}})$. This approximation does not take into consideration the intersection of the hyperplane with the iteration space and it is possible that two arrays may be aligned in a region beyond the iteration space. For example, consider the two Sisal programs in figures 4.5 and 4.6. Both of these programs have a value of $H = 1$. In the first program the alignment line, $i = k$, falls within the triangular iteration space along the diagonal. However in the second program k ranges from 1 to $i - 1$ and thus k can never be equal to i. In general, perfect alignment occurs when

$$\mathcal{X} = \mathcal{Y}, x = y \tag{4.15}$$

For perfect alignment we seek a transformation $(\mathcal{A}, a)$ such that:

$$\mathcal{A}\mathcal{Y} = \mathcal{X} \tag{4.16}$$

$$a + y = x \tag{4.17}$$

Equation 4.17 can be trivially solved by rearrangement, while 4.16 may only be solved by finding $\mathcal{X}\mathcal{Y}^{-1}$, or solving by Gaussian elimination, as long as $\mathcal{Y}$ is non-singular. As the equation 4.17 always has a solution the remaining

```
a := for i in 1,n cross j in 1,n
     returns array of
       b[i+j,i-j+1]
     end for
```

Figure 4.7: A Sisal Program

```
a := for i in 1,n cross j in 1,n cross k in 1,n
     returns array of
       b[i+j+k,i+k,i+k]
     end for
```

Figure 4.8: A Sisal Program

discussion of alignment transformations will largely focus on $\mathcal{A}$. To illustrate transforming for perfect alignment, consider the program in figure 4.7. In this program the arrays a and b have the following array occurrences:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_a \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}_b \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}_b \tag{4.18}$$

Here $\mathcal{U}_{1.b}$, the occurrence matrix of array b, is non-singular and thus there exists a transformation such that the arrays may be perfectly aligned.

$$\mathcal{A} \times \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, a + \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4.19}$$

i.e.

$$\mathcal{A} = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, a = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{4.20}$$

However, this is not always the case. Consider the program in figure 4.8 which has the following occurrence matrices:

$$\mathcal{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathcal{U} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{4.21}$$

There is no solution to $\mathcal{AU} = \mathcal{C}$, due to the singularity of $\mathcal{U}$.

A method to align u-occurrences with a c-occurrence matrix is presented in the next section which determines an alignment function even in the case of singular matrices.

4.2 Creation Alignment

This section investigates the alignment of each u-occurrence of a computation set with the set's c-occurrence which is appropriate when compiling for creation parallelism. An important property of the c-occurrence matrix is that each of its rows are independent. This is due to the language definition of Sisal which does not allow part of an array to be created.

Although it is not possible, in general, to calculate a perfect alignment even for 2 arrays of the same dimension referencing the same iterators, it is possible in some cases to determine the best possible alignment. This requires the finding of an $\mathcal{A}$ where:

$$\mathcal{A}\mathcal{U} = \mathcal{V} \tag{4.22}$$

such that $H_{\mathcal{C},\mathcal{V}}$ is maximised. As the upper bound on H is given in theorem 6, it is possible to determine optimal alignment in some cases.

4.2.1 Legal Alignment Functions

At this point it is important to define the restrictions on the form of $\mathcal{V}$ which will define the class of legal transformations, $\mathcal{A}$. The matrix, $\mathcal{U}$, references a number of points, determined by the iteration space, which correspond to array elements. Any matrix $\mathcal{V}$ must reference the same points, the same number of times as $\mathcal{U}$. This implies, amongst other things, that $rank(\mathcal{U}) = rank(\mathcal{V})$. Intuitively this means that references can not be made to new iterators, nor old references dropped. In addition the constraint that $\mathcal{V}$ be of size $M \times m$ is imposed. In other words $\mathcal{V}$ should have the same rank as $\mathcal{U}$, same size as $\mathcal{U}$ and, in addition, have the same number of zero rows as $\mathcal{U}$.

A legal transformation, $\mathcal{A}$, is one which preserves the rank of $\mathcal{U}$. In general if $\mathcal{U}$ is rank deficient so may be $\mathcal{A}$. However in a more general context if there exists another occurrence of the same array variable in the program then it must also be premultiplied by the alignment transformation $\mathcal{A}$. If this occurrence is not rank deficient then multiplying by a rank deficient $\mathcal{A}$ will reduce the rank of that occurrence, which is illegal. Therefore $\mathcal{A}$ is restricted to forms which always preserve the rank of any matrix, $\mathcal{U}$. Such a matrix is one which has full rank. As $\mathcal{A}$ is square, $M \times M$, then it follows that $\mathcal{A}$ is always invertible We require a $\mathcal{V}$ where as many rows are equal to $\mathcal{C}$ as possible. Each row, i, of $\mathcal{C}$ is the ith row of the identity matrix, where the ith element is one. For alignment we require that:

$$\delta_{\mathcal{V}_i,c_i} = 1, \forall i \in 1, \ldots, M \tag{4.23}$$

or

$$\mathcal{V}_i = c_i \forall i \in 1, \ldots, M \tag{4.24}$$

thus

$$\mathcal{A}_i\mathcal{U} = c_i \forall i \in 1, \ldots, M \tag{4.25}$$

However, this may be true for few or no values of i. As $\mathcal{A}$ is $M \times M$ and the number of values of i for which there is a solution to 4.25 is, say, k then there remains $M - k$ rows of $\mathcal{A}$ which must be independent of each of the k solutions so as to maintain the full rank of $\mathcal{A}$. To maximise alignment in the sense of H, it is necessary to find as many solutions, y_i, to

$$y_i\mathcal{U} = c_i \tag{4.26}$$

where y_i is the ith row of the matrix $\mathcal{A}$. In the following sections we consider the issues of existence, uniqueness and independence which are based on well known results from linear algebra.

4.2.2 Existence, Independence and Uniqueness

$$y_i\mathcal{U} = c_i \tag{4.27}$$

can be written in the more familiar form

$$\mathcal{U}^T y_i^T = c_i^T \tag{4.28}$$

where y^T, c_i^T are column vectors.

Theorem 3 *It is possible to align a row of a u-occurrence matrix $\mathcal{U}$ with a c-occurrence iff c_i^T is perpendicular to the null space of $\mathcal{U}$ i.e. $c_i^T \perp \mathcal{N}(\mathcal{U})$ where $\mathcal{N}(\mathcal{U})$ are the iterators not referenced by $\mathcal{U}$.*

Proof of Theorem 3 *The equation*

$$\mathcal{U}^T x^T = c_i^T \tag{4.29}$$

is of the form

$$Ax = b \tag{4.30}$$

where A is an $(m \times M)$ integer matrix, b is an $(M \times 1)$ integer column vector and x is the $(M \times 1)$ unknown vector. This equation has a solution iff b is in the range space of A [NOBL88]i.e.

$$b \in \mathcal{R}(A) \tag{4.31}$$

From linear algebra [SCHR86]

$$\Re^m = \mathcal{R}(A) \oplus \mathcal{N}(A^T) \tag{4.32}$$

In other words, the m dimensional real space is a direct sum of the range space of A and the null space of A^T. Thus

$$\forall z \in \Re^m \exists u \in \mathcal{R}(A), v \in \mathcal{N}(A^T), u \perp v \ \ s.t. \ \ z = u + v \tag{4.33}$$

Thus

$$b \in \mathcal{R}(A) \Leftrightarrow b \perp \mathcal{N}(A^T) \tag{4.34}$$

but

$$\mathcal{N}(A^T) = \mathcal{N}((\mathcal{U}^T)^T) = \mathcal{N}(\mathcal{U}) \tag{4.35}$$

and $b = c_i^T$ hence

$$c_i^T \perp \mathcal{N}(\mathcal{U})\square \tag{4.36}$$

It is important that each solution to 4.25 is independent, this is easily shown.

Theorem 4 *Each solution row of $\mathcal{A}$ in $\mathcal{A}\mathcal{U} = \mathcal{C}$ is independent.*

Proof of Theorem 4 *Assume two solutions of 4.25 are a_x and a_y and they are dependent thus:*

$$\exists \alpha, \beta \mid \alpha \neq 0, \beta \neq 0, \alpha a_x + \beta a_y = 0 \tag{4.37}$$

and

$$\alpha a_x \mathcal{U} = \alpha c_x^T \tag{4.38}$$

$$\beta a_y \mathcal{U} = \beta c_y^T \tag{4.39}$$

then

$$(\alpha a_x + \beta a_y)\mathcal{U} = 0 = \alpha c_x^T + \beta c_y^T \tag{4.40}$$

But

$$c_x = e_x, c_y = e_y \tag{4.41}$$

and

$$\alpha e_x^T + \beta e_y^T \neq 0 \tag{4.42}$$

as these are independent rows of Identity, thus the assumption that a_x, a_y are dependent is false. □

It follows that equation 4.25 may be solved independently for each value of i. The remaining rows of $\mathcal{A}$ that are not solutions have to be constructed in a way such that they are independent and maintain the full rank of $\mathcal{A}$.

Theorem 5 *The solution to $\mathcal{U}^T y_i^T = c_i^T$ is unique iff $dim\mathcal{N}(\mathcal{U}^T) = 0$*

Proof of Theorem 5 *Let $a_x^T \neq a_y^T$ be two solutions to $\mathcal{U}^T y_i^T = c_i^T$. Then*

$$\mathcal{U}^T a_x^T = c_i^T \tag{4.43}$$

$$\mathcal{U}^T a_y^T = c_i^T \tag{4.44}$$

Hence

$$\mathcal{U}^T a_x^T - \mathcal{U}^T a_y^T = 0 \tag{4.45}$$

$$\mathcal{U}^T (a_x^T - a_y^T) = 0 \tag{4.46}$$

By assumption

$$(a_x - a_y)^T \neq 0 \tag{4.47}$$

Therefore

$$dim(\mathcal{N}(\mathcal{U}^T)) \neq 0 \tag{4.48}$$

A contradiction. Conversely let $b^T \neq 0 \in \mathcal{N}(\mathcal{U}^T)$ and a_x^T be a solution, then $a_x^T + b^T$ is also a solution and therefore non-unique $\Box$

Theorem 6 *The maximum value of $H_{\mathcal{AU},\mathcal{C}}$ is $\min(r, d)$ where $r = rank(\mathcal{U})$ and $d = rank(\mathcal{C})$*

Proof of Theorem 6 *By definition, the maximum value of H is the number of solutions to 4.25 There at most $d = rank\, \mathcal{C}$ columns of $\mathcal{C}$ to provide solutions.*

From theorem 3 y_i is a solution if $c_i^T \in \mathcal{R}(U^T)$. Now $dim(\mathcal{R}(\mathcal{U}^T)) = r$,so there are at most r linearly independent solutions.

Hence there are at most $\min(r, d)$ *solutions to 4.25* $\Box$

4.2.3 Row Echelon Form

In order to determine the form of $\mathcal{A}$, such that $H_{\mathcal{AU},\mathcal{C}}$ is maximised, then it is important, as has been stated previously, to find a solution for as many rows of $\mathcal{C}$ to the following equation:

$$y_i \mathcal{U} = c_i \tag{4.49}$$

Transposing gives

$$\mathcal{U}^T y_i^T = c_i^T \tag{4.50}$$

To solve for y, concatenate $\mathcal{U}^T$ and c_i^T and reduce by a row operation $\mathcal{U}^T$ to row echelon form, whilst simultaneously performing the same operations on c_i^T. After reducing to row echelon form, the rows are re-ordered such that the leading element on non-zero rows is the diagonal. Equation 4.50 is re-written as:

```
a := for i in 1,n cross j in 1,n cross k in 1,n
     returns array of
       for l in i,k
       returns value of sum
         b[l+k,k-2*l+3,2*j-6,l+i] * c[l+k,k+l,2*j-6,l+k+i]
       end for
     end for
```

Figure 4.9: A Sisal Program

$$\begin{array}{c} m \\ M-m \\ \\ \end{array} \left[\begin{array}{c|c} \mathcal{U}^T & c_i^T \\ \hline O & O \end{array}\right] \atop \begin{array}{cc} M & 1 \end{array} \tag{4.51}$$

Reducing by any legal row operation, gives the unique row-echelon form:

$$\begin{array}{c|cc|c} & r & M-r & 1 \\ \hline r & Re & * & c_i^1 \\ m-r & O & O & c_i^2 \\ M-m & O & O & O \end{array} \tag{4.52}$$

Here Re and $*$ are non-zero sub-matrices. By theorem 3 there is a solution iff $c_i^T \perp \mathcal{N}(\mathcal{U})$. This is equivalent to $c_i^2 = 0$ which provides a simple existence test. If a solution exists then it forms a row of $\mathcal{A}$.

By theorem 6 there are, at most, $\min(r, d)$ solutions and the remaining rows of $\mathcal{A}$ must be filled. It is important that these additional rows be independent so that $\mathcal{A}$ is full rank. One method of achieving this, is to concatenate the whole $\mathcal{C}$ matrix to $\mathcal{U}^T$. After reducing to row echelon form there will be r solutions and $N_\mathcal{C} - r$ independent columns which are not solutions, but can form independent rows of $\mathcal{A}$. There is a problem if $M > N_\mathcal{C}$ as the remaining rows of $\mathcal{A}$ have to be filled. By further augmenting the I_M matrix and reducing to row echelon form, the following augmented matrix is formed.

$$\begin{array}{c|cc|cc|c|c} & r & M-r & r & N_\mathcal{C}-r & N_\mathcal{C} & M-N_\mathcal{C} \\ \hline r & Re & * & X^1 & Y^1 & W^1 & O \\ m-r & O & O & O & Y^2 & W^2 & O \\ M-m & O & O & O & O & O & I \end{array} \tag{4.53}$$

$N_\mathcal{C}$ rows of $\mathcal{A}$ are available from the row-echelon form of $\mathcal{C}$. The remainder come from the last section of the augmented matrix. As the W region never gives independent rows of $\mathcal{A}$ it can be omitted. It is now possible to define an algorithm to find a legal $\mathcal{A}$ which is detailed in appendix c.

4.2.4 Example

To illustrate this procedure, consider the program in figure 4.9 which has the following occurrence matrices/vectors.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}_a, \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -2 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}_b \begin{bmatrix} 0 \\ 3 \\ -6 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}_c \begin{bmatrix} 0 \\ 0 \\ -6 \\ 0 \end{bmatrix} \tag{4.54}$$

where $M = 4$.

As long as the u-occurrence matrices refer to different arrays, they may be compared pair-wise with the c-occurrence matrix. Thus on aligning b with a we have:

$$\left[\begin{array}{cccc|cccc|c} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & -2 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{4.55}$$

On reducing to row-echelon form we have:

$$\left[\begin{array}{cccc|cccc|c} 1 & 0 & 0 & 0 & -\frac{1}{3} & 0 & \frac{2}{3} & 0 & \frac{1}{3} \\ 0 & 1 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & -\frac{1}{3} \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}\right] \tag{4.56}$$

Columns 5,6,7 are solutions and the ninth column is independent and forms the fourth row of $\mathcal{A}$. As long as the last row chosen is independent, it is arbitrary as the maximum number of alignment solutions is limited here by the rank of the c-occurrence matrix i.e. 3.

$$\begin{bmatrix} -\frac{1}{3} & \frac{1}{3} & 0 & 1 \\ 0 & 0 & \frac{1}{2} & 0 \\ \frac{2}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{3} & -\frac{1}{3} & 0 & 0 \end{bmatrix} \tag{4.57}$$

Which gives the following occurrence matrix $\mathcal{V}$:

$$\mathcal{A} \times \mathcal{U} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.58}$$

the new vector $a + u$ is trivially calculated and is $[0, 0, 0, 0]^T$.

Repeating this procedure for array c we have:

$$\left[\begin{array}{cccc|cccc|c} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{4.59}$$

On reducing to row-echelon form and reordering so that leading elements are on the diagonal:

```
a := for i in 1,n cross j in 1,n cross k in 1,n
     returns array of
       for l in i,k
       returns value of sum
         b[i,j,k,l] * c[i,j,k+l,k+l]
       end for
     end for
```

Figure 4.10: A Sisal Program

$$\left[\begin{array}{cccc|cccc|c} 1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}\right] \tag{4.60}$$

Both columns 5 and 7 of the augmented matrix are legal solutions for rows of $\mathcal{A}$. Column 6 although a legal candidate for a row of $\mathcal{A}$ is not a solution as it is not perpendicular to the null space of $\mathcal{U}^T$. This can be seen as the second row of the row echelon form of $\mathcal{U}^T$ is zero, but there is a non-zero element in this position in column 6. Again row four of $\mathcal{A}$ is arbitrary and the final column is a suitable choice.

$$\mathcal{A} = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 & 0 \\ 2 & -1 & 0 & 0 \end{bmatrix} \tag{4.61}$$

Which gives the following occurrence matrix, $\mathcal{V}$:

$$\mathcal{A} \times \mathcal{U} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{4.62}$$

Therefore the program in figure 4.9 may be rewritten as the program in figure 4.10. Before the alignment transformations, $H_{a,b} = 0, H_{a,c} = 0$ and after aligning $H_{a,b} = 3, H_{a,c} = 2$ showing that $\mathcal{A}$ has improved alignment.

4.2.5 Multiple u-occurrences

If an array, v, has more than one u-occurrence, $\theta_v > 1$, then there may be a conflict in alignment between each occurrence. If H is evaluated for each alignment function, then the best function is the one for which H has the greatest value. Consider the program fragment in figure 4.11: Here if b_{i+10} is aligned with a_i then $\Sigma H_{\mathcal{C}\mathcal{A}\mathcal{U}} = 3$, as opposed to 1 if b_i is aligned with a_i. This may be generalised as follows: The number of occurrences of a particular array x is θ_x. Let $r \in 1, \ldots, \theta_x$. The following strategy is employed to determine the alignment function. Construct

$$\mathcal{A}^r \forall r \in 1, \ldots, \theta_x \tag{4.63}$$

```
a := for i in 1,n cross j in 1,n
    returns array of
        b[i,j] + b[i+10,j] +b[i+10,j-1] +b[i+10,j+1]
    end for
```

Figure 4.11: A Sisal Program

```
a := for i in 1,n cross j in 1,n
    returns array of
      for k in 1,n
      returns value of sum
        b[i,k] * c[k,j]
      end for
    end for
```

Figure 4.12: A Sisal Program

Let

$$\Gamma(r) = \sum_{i=1}^{\theta_x} H_{\mathcal{A}^r \mathcal{U}_x, \mathcal{C}} \forall r \tag{4.64}$$

The alignment matrix chosen will be $\mathcal{A}^r$ where Γ is greatest.

4.3 Reduction Alignment

The forgoing analysis has been based on alignment of arrays when compiling for creation parallelism, A different approach is required when compiling for parallelism associated with reduction iterators. Different iterations of the reduction iterator are scheduled on different processors. One of the arrays involved in the reduction makes access to the other u-occurrence arrays involved in the computation. The reduction operations take place, whereupon the resulting value is accessed by the process calculating the relevant portion of the c-occurrence array. Partitioning of data for reduction parallelism always involves non-local access. Intuitively it is because work is being performed in parallel on several different processors, the result of which is accumulated by the creating process. It can also be demonstrated by looking at the occurrence matrices. A reduction iterator reference **only** occurs on the right hand side of a definition and thus can never be aligned with respect to the left hand, c-occurrence, side.

To illustrate these points consider the program in figure 4.12. Here the reduction iterator is *k*, which is recognised syntactically by the “returns value of sum” operator. On inspection it can be seen that *k* only appears on the right hand side of the definition.

$$a[i,j] = b[i,k]c[k,j] \tag{4.65}$$

Aligning for reduction necessitates the choice of a u-occurrence with which to align to. In this case, the array *b* is arbitrarily chosen rather than *c* which would give a different set of alignments. In general, both *b* and *c* should be considered as one may be preferable in a more global context.

On aligning with array *b*, *a* retains its present orientation, while *c* is stored in a transposed manner. It is worth noting that aligning for creation parallelism would give a different alignment.

```
a := for i in 1,n cross j in 1,n
    returns array of
      for k in 1,n
      returns value of sum
        b[k+j,i]*c[i+j,i-j]
      end for
    end for
```

Figure 4.13: A Sisal Program

4.3.1 Row Echelon

Once again the method for determining alignment transformations is based upon the row echelon form of the augmented matrix. The major difference is that, in general, the array to be aligned with will be rank deficient. This implies that some rows of its occurrence matrix may not be independent and thus theorem 4 no longer applies. Thus each row of $\mathcal{A}$ must be checked for independence. There are a variety of ways of achieving this but the method used in the algorithm in appendix c is to reduce a copy of the partially filled $\mathcal{A}$ to row echelon form. If the copy has full rank then the row just added is independent.

4.3.2 Example

To illustrate aligning for reduction parallelism consider the Sisal program in figure 4.13. In this example it is assumed that both arrays *a* and *c* are to be aligned with array *b* as reduction parallelism is to be exploited. They have the following occurrence matrices:

$$\mathcal{C}_a : \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathcal{U}_b : \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathcal{U}_c : \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.66}$$

Augmenting the transpose of the creation matrix and Identity to that of the reduction matrix gives:

$$\left[\begin{array}{ccc|ccc|ccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{4.67}$$

Only the fifth column is a solution and forms the second row of $\mathcal{A}$. The remaining rows are picked from the Identity region to give:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.68}$$

Repeating the same process with *c* and *b* and by augmenting the transpose of the u- occurrence matrix and identity matrix gives:

```
a := for j in 1,n cross i in 1,n
    returns array of
      for k in 1,n
      returns value of sum
          b[k+j,i]*c[j,i]
      end for
    end for
```

Figure 4.14: A Sisal Program

$$\left[\begin{array}{ccc|ccc|ccc} 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{4.69}$$

Reducing to row echelon form:

$$\left[\begin{array}{ccc|ccc|ccc} 1 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & -\frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 1 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{4.70}$$

The fourth and fifth columns are independent solutions. The remaining independent row of $\mathcal{A}$ comes from the final column of the reduced matrix.

$$\begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.71}$$

Before the alignment transformations, $H_{b,a} = 0, H_{b,c} = 0$, after aligning $H_{b,a} = 1, H_{b,c} = 1$, thus improving alignment and giving the program in figure 4.14.

4.4 Alignment Propagation

Once a new alignment has been chosen for an array, v, the effect must be propagated throughout the program to maintain meaning. To illustrate this point, consider the program in figure 4.15. When aligning b to a for the creation of a, b will be shifted by one. This effect will have to be propagated to the creation of b as is illustrated in the imperative translation in figure 4.16.

However the c-occurrence of b is no longer a simple sub-matrix of identity. This makes the partitioning of data in general more difficult. In [LI91], c-occurrences are assumed to be sub-matrices of identity to aid partitioning. Restoring b to its previous form will affect the loop body $2(j+1)$ to give the program in figure 4.17. This idea can now be formalised.

Given an occurrence say, $\mathcal{U}_{1.v}$ and the new occurrence of alignment $\mathcal{U}'_{1.v} = \mathcal{A}\mathcal{U}_{1.v}$ then, if there is an occurrence

```
for initial
   k := 1;
   a := x;
   b := y;
while (k <= n)
repeat
   k := old k +1
   b:= for j in 1,n
      returns array of 2*(j+1)
      end for;
   a:= for i in 1,n
      returns array of
        if (i=1)
        then 0
        else old b[i+1]
        end if
      end for
   returns value of a
end for
```

Figure 4.15: A Sisal Program

```
FORITER k = 1 TO n
  FOR j = 1 TO n
    b[j-1] := 2*(j+1)
  END FOR
  a[1] := 0
  FOR i = 2 TO n
    a[i]:=  oldb[i]
  END FOR
END FORITER
```

Figure 4.16: An Imperative Program

```
FORITER k = 1 TO n
  FOR j = 1 TO n
    b[j] := 2*((j+1) mod n +1)
  END FOR
  a[1] := 0
  FOR i = 2 TO n
    a[i]:=  oldb[i]
  END FOR
END FORITER
```

Figure 4.17: An Imperative Program

```
a:= for i in 1,2*n
   returns array of d[i]
   end for;
b:= for i in 1 ,n
   returns array of  a[2*i]
   end for;
c:= for i in 1 ,n
   returns array of  a[2*i-1]
   end for;
```

Figure 4.18: A Sisal Fragment

```
FOR i = 1 TO 2*n
  a[i] := d[i]
END FOR
FOR i = 1 TO n
  b[i] := a[i]
END FOR
FOR i = 1 TO n
  c[i] := a[i-1]
END FOR
```

Figure 4.19: An Imperative Program

of that array in any computation set, transform it to the new alignment i.e

$$\forall Q \forall x \in 1, \ldots, \theta_v, \mathcal{U}'_{x.v} = \mathcal{A}\mathcal{U}_{x.v}, \mathcal{C}'_v = \mathcal{A}\mathcal{C}_v \tag{4.72}$$

To restore the identity matrix of each c-occurrence, left multiply by $\mathcal{A}^{-1}$:

$$\forall Q \mathcal{C}'_v \neq I \rightarrow \mathcal{C}''_v = \mathcal{A}^{-1}\mathcal{C}'_v \tag{4.73}$$

where $\mathcal{C}''_v = \mathcal{C}_v$ and is thus returned to its original form. All references to iterators which are not array occurrences must also be adjusted. Let $f(J)$ be a usage of the iterators J in F, the parse tree of a computation set, then the new usage is $f'(J)$ where

$$f'(J) = f(\mathcal{A}^{-1}J) \tag{4.74}$$

and the value of the iterators $\mathcal{A}^{-1}J$ are restricted to the values of J in the lattice $Latt(A.b)$. This procedure can be used for an arbitrary number of re-alignments. However, at present, a problem occurs if the matrix, $\mathcal{A}^{-1}$, is non-unimodular and there is more than one reference to v with a different occurrence matrix.

Consider the Sisal fragment in figure 4.18. If a is to be aligned with b and the effect propagated, as described above, then this will result in the imperative program shown in figure 4.19. Before alignment propagation, arrays b and c referred to alternate elements of a but alignment propagation has compressed the reference to a with the consequence that, now, b and c refer to overlapping regions which no longer preserves meaning. At present alignment is, therefore, restricted to unimodular alignment transforms, if there is more than one different u-occurrence of a particular array. Although this is restrictive, in [SHEN90] it is shown that a large proportion of scientific programs comply with this, and it seems likely that in the near future this restriction may be relaxed.

```
a:= for i in 1,n cross j in 1,n
    returns array of
        b[i,i]
      end for
    end for
```

Figure 4.20: A Sisal Program

4.5 Data Partition

Once the relative alignment between arrays is determined, by the previous phase, the arrays must be mapped to processor space. At this stage only orthogonal partitions of the arrays are considered. Thus if an array has N dimensions, it may be partitioned in 2^{N-1} different ways. If possible, the partitioning of data should be along those dimensions of the array that may be evaluated in parallel. In the next chapter, the mapping of data to processors, so as to determine the local data and computation space is defined and a method based upon the volume of access is developed. In this section, however, the influence of alignment on the partitioning process is described. In particular, the transformations required to partition an array along one dimension and serialise the remainder are presented.

4.5.1 Aligned Form

If two arrays are aligned with respect to a particular index, then no matter how those individual array elements are partitioned, any reference between the two arrays with respect to this index will always be local. Non-aligned indices do not have this property and thus partitioning with respect to these indices should be avoided. If an array has N dimensions and say k of them are aligned, then there are k dimensions the array may be partitioned along, and $N-k$ that should be *serialised*. As the size of any one dimension of an array is assumed to be greater than the number of processors i.e. $n >> p$, it is reasonable to consider only one dimension to partition along at this stage. In chapter 5, the volume of access analysis considers a greater variety of data partitions. Therefore, within this section, of the k parallel dimensions, only one will be selected to partition the array across the processors and the remaining $N-1$ will be serialised.

Consider the Sisal program in figure 4.20. This program has the following array occurrences:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \tag{4.75}$$

The first index of b is aligned with a as $\delta_{i,i} = 1$, but the second index is not $\delta_{j,i} = 0$.

Partitioning an array into k dimensions is the act of projecting an $M \times m$ occurrence matrix so that there are k non-zero rows in the new occurrence matrix, where each of the k rows corresponds to that dimension of the array being distributed over the p processors. Thus the data partitioning/scheduling function ζ is defined as follows:

$$\zeta : \mathcal{X} \mapsto \overline{\mathcal{X}} \tag{4.76}$$

where $\mathcal{X} = \mathcal{C}$ or $\mathcal{U}$ and $\overline{\mathcal{X}} = \overline{\mathcal{C}}$ or $\overline{\mathcal{U}}$ and $\mathcal{X}$ is the partitioned matrix and $\mathcal{X}$ and $\overline{\mathcal{X}}$ are both $M \times m$ matrices.

Conversely a serialising function ρ can be defined which describes those indices that are not to be partitioned. i.e.

$$\rho : \mathcal{X} \mapsto \underline{\mathcal{X}} \tag{4.77}$$

where $\underline{\mathcal{X}}$ is a $M \times m$ serialised occurrence matrix with at least k zero rows.

Hence

$$\zeta(X) + \rho(X) = X \tag{4.78}$$

So the definition of either ζ or ρ automatically defines the other function. The particular form of these transformations is chosen to be $M \times M$ integer matrices $\mathcal{P}$ and $\mathcal{S}$ respectively, i.e.

$$\mathcal{P} \times \mathcal{X} = \overline{\mathcal{X}} \tag{4.79}$$

$$\mathcal{S} \times \mathcal{X} = \underline{\mathcal{X}} \tag{4.80}$$

where

$$\mathcal{P} + \mathcal{S} = I \tag{4.81}$$

Unlike the alignment matrix $\mathcal{A}$ these transformation matrices are formed to reduce the rank of the occurrence matrices and thus are singular.

The rows of $\mathcal{P}$ are either rows of identity or rows of the null matrix. After pre-multiplying the occurrence matrix, the new partitioned occurrence matrix has non-zero and zero rows. Non-zero rows are those that are to be partitioned.

In the previous program, figure 4.20, $\mathcal{P}$ is chosen to be of the following form:

$$\mathcal{P} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.82}$$

which gives the following reduced occurrence matrices:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.83}$$

If $H_{\mathcal{C},\mathcal{U}}$ is determined for non-zero rows it is found to be 1. If $\mathcal{P}$ was chosen to be

$$\mathcal{P} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{4.84}$$

this would give the following reduced occurrence matrices:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \tag{4.85}$$

Here $H_{\mathcal{C},\mathcal{U}}$ is 0. This observation gives a simple method of deciding which index to partition along and hence a method for constructing $\mathcal{P}$. If two rows of the arrays to be aligned are equal, then let the corresponding row of $\mathcal{P}$ be the relevant row of Identity, otherwise set it equal to zero.

$$\mathcal{P}_i = \begin{cases} e_i^T & \mathcal{C}_i = \mathcal{U}_i \\ 0 & \mathcal{C}_i \neq \mathcal{U}_i \end{cases} \quad \forall i \in 1, \ldots, M \tag{4.86}$$

Perfect data partitioning occurs when all non-zero rows of the reduced occurrence matrices are equal, whereupon $H_{\mathcal{PC},\mathcal{PU}} = k$. At present only the case $k = 1$ is considered, i.e. only one dimension to partition along is sought, and thus a suitable method of determining $\mathcal{P}$ is as follows :

$$\mathcal{P}_i = \begin{cases} e_i^T & \mathcal{C}_i = \mathcal{U}_i \wedge \mathcal{P}_k \neq e_k^T \forall k \in 1, \ldots, i-1 \\ e_i^T & \mathcal{P}_k = 0 \forall k \in 1, \ldots, N-1 \\ 0 & \mathit{otherwise} \end{cases} \quad \forall i \in 1, \ldots, M \tag{4.87}$$

```
a:= for i in 1,n
    returns array of
      for k in 1,n
      returns value of sum
        b[i] * c[i,k]
      end for
    end for
```

Figure 4.21: A Sisal Program

```
a:= for i in 1,n cross j in 1,n
    returns array of
        b[i] * c[j]
    end for
```

Figure 4.22: A Sisal Program

4.5.2 Arrays of Differing Dimensions

Problems occur when the arrays to be partitioned have a different number of dimensions. If two arrays which reference each other in a computation have a different number of dimensions, then the approach taken here is that they may not be partitioned by more than the lowest dimensioned array's number of dimensions. Consider the example in figure 4.21. As *a* and *b* are both one-dimensional, *c* can only be partitioned by rows or columns, not both. The motivation for this constraint is that it ensures that data is evenly distributed across the processors.

To illustrate this point, consider the program in figure 4.22 with the associated occurrence matrices:

$$a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, b = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, c = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.88}$$

Two equally optimal forms of $\mathcal{P}$ can be found using 4.81 and they are:

$$\mathcal{P} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \mathcal{P} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.89}$$

After applying both partitioning matrices the two possible forms of the reduced occurrence matrices are found:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.90}$$

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.91}$$

In both cases there is a problem as in each case there is a matrix with no non-zero entries after partitioning. This implies that the array is not partitioned across the processors and hence resides solely in one processor. Due to the scalability constraint, this is illegal. Because of this there must always be an entry in the partitioned row of the reduced occurrence matrix , which may be any other non-zero row of the occurrence matrix. By applying this principle to the present example we have:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{4.92}$$

```
a := for i in 1,n cross j in 1,n
     returns array of
       for k in 1, n
       returns value of sum b[k,j]
       end for
     end for
```

Figure 4.23: A Sisal Program

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.93}$$

If the total value of H is calculated for both partitions, they are shown to be equal, and in this case either partitioning is equivalent but more importantly the data is evenly distributed.

In the case of different sized arrays, $\mathcal{P}$ is determined as before. For those arrays that are of a smaller dimension than the creation array, a re-ordering of their occurrence matrix is required. Let the re-ordered matrix be $\mathcal{T}$ which is defined

$$\mathcal{T}_i = \begin{cases} e_i^T & U_i \neq 0 \wedge i \neq t \\ e_k^T & U_i = 0 \wedge i = t \wedge i \neq k \quad \forall i \in 1, \ldots, M \\ e_t^T & i = k \end{cases} \tag{4.94}$$

where t is the dimension which $\mathcal{P}$ has been selected to partition on i.e $\mathcal{P}_t = e_t^T$ and k is the row swapped with row t if row t is zero. Data partitioning can now be defined as finding a $\mathcal{P}$, $\mathcal{S}$ and $\mathcal{T}$ such that:

$$\mathcal{P} \times \mathcal{T} \times \mathcal{X} = \overline{\mathcal{X}} \tag{4.95}$$

$$\mathcal{S} \times \mathcal{T} \times \mathcal{X} = \underline{\mathcal{X}} \tag{4.96}$$

where

$$\mathcal{P} + \mathcal{S} = I \tag{4.97}$$

The matrix $\mathcal{T}$ is also useful in ensuring that there are no later problems in data partitioning due to padding. To illustrate this, consider the program in figure 4.23. The occurrence matrices of a and b are respectively:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{4.98}$$

and in padded form

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.99}$$

After aligning the arrays will be of the following form

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.100}$$

If it is decided to partition along the first index then b is completely serialised. Having an n^2 sized array resident on a processor is not desirable. Applying the $\mathcal{T}$ transform gives the following occurrence matrices which ensures even data distribution when partitioned along the first index.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.101}$$

The necessity of this final transform is because a 2-dimensional object has been embedded in a three dimensional iteration space. A projection along an iterator forming this space does not necessarily partition the two dimensional sub-space of the array. This was the case for the array b, where the transformation $\mathcal{T}$ ensured that the array was orthogonal to the partitioning direction.

4.6 Summary

In this chapter, a formal description of alignment has been given. By considering the arrays as embedded in the iteration space, it has been possible to describe alignment as a hyperplane whose size can be measured. By taking an approximate metric H, alignment transformations based on the row echelon transformation have been developed for creation and reduction alignment. How local alignment affects the remainder of the program has been addressed. Finally a method to allow later data partitioning has been described.

Chapter 5

Partitioning and Pre-Fetching

This chapter describes the mapping of data and computation to processors and the pre-fetching of data to reduce non-local access. The first section describes how data is scheduled across the processors and, once this has been performed, how the local iteration space may be determined.

Once the data has been mapped to the processors, it is desirable to minimise any non-local access by adopting a compiler directed caching policy. An invariancy analysis is presented in the second section to determine any opportunity for data re-use. Pre-fetching transformations are described which take advantage of any re-use and can be used in the presence of general affine loop and array occurrences.

In chapters 3 and 4, the criteria for partitioning of data was either based upon load balancing, or alignment. In the third section a new method is introduced which attempts to determine the best data partition given that pre-fetching transformations are available. As this method depends on such transformations, it is presented after the pre-fetching section. Finally a summary concludes this chapter

5.1 Indices, Iterators and Processors

In this section the mapping of data and computation to processors is examined. Initially the mapping of just one array dimension to several processors is derived. This is extended to multiple dimensions which may be wrapped. Once the local data space for each processor has been determined, a method to calculate the local iteration space is shown.

5.1.1 Mapping Indices

Each array, v, in a computation set has the following index domain:

$$[i_1, i_2, \ldots, i_{N_v}]^T = \mathcal{I} \tag{5.1}$$

where each i_x is the xth index of the array, and $\mathcal{I}$ is the vector containing all the indices of that array. Each index corresponds to a non-zero row of the $\mathcal{C}$ occurrence matrix of array v. Each non-zero row will have one reference to an iterator whose bounds will determine the range of that index. The index range is defined by the inequalities:

$$\lambda \le \mathcal{I} \le \upsilon \tag{5.2}$$

This set of inequalities can be reordered to give:

$$\begin{bmatrix} -I \\ I \end{bmatrix} \mathcal{I} \le \begin{bmatrix} -\lambda \\ \upsilon \end{bmatrix} \tag{5.3}$$

where λ, υ are $N_v \times 1$ vectors. Only constant vectors are required to describe the bounds of $\mathcal{I}$ because only rectangular arrays are considered in this thesis. Each array, in general, has a different range of indices and thus has a different $\mathcal{I}$.

If the array is to be partitioned along just one index then that index, say i_x, must be divided into p sub-ranges. Once a particular partition, has been chosen, as defined in section 4.5, then the indices corresponding to $\mathcal{PC}$ will be defined over a sub-range. The remaining serialised ones, corresponding to $\mathcal{SC}$, will be unaffected.

This section is concerned with the function, ζ, which maps the array indices to the various processors. In the case of partitioning just one index, it has the following form:

$$\zeta : \mathcal{I} \mapsto \underline{\mathcal{I}_1} \times \underline{\mathcal{I}_2} \times \cdots \times \underline{\mathcal{I}_p} \tag{5.4}$$

For some processor, $z \in 1, \ldots, p$, the bounds on the indices $\underline{\mathcal{I}_z}$ describe the elements of an array, a, local to processor z. The indices local to a processor z consist of d separate sequences of index values. $\underline{\mathcal{I}_z} = \underline{\mathcal{I}_z^1} \times \underline{\mathcal{I}_z^2} \times \cdots \times \underline{\mathcal{I}_z^d}$, where d is the number of times an array is wrapped around the processors. In the majority of cases d is equal to one. The kth sequence $\underline{\mathcal{I}_z^k}$, $1 \le k \le d$ is of the following form:

$$\left[\begin{array}{c} -I_N \\ I_N \\ \hline -e_x \\ e_x \end{array}\right] \begin{bmatrix} \underline{i_1^k} \\ \vdots \\ \underline{i_x^k} \\ \vdots \\ \underline{i_N^k} \end{bmatrix} \le \left[\begin{array}{r} -\lambda \\ \upsilon \\ \hline -[(z-1) \times b + k \times r + 1] \\ z \times b + k \times r \end{array}\right] \tag{5.5}$$

where e_x is the xth row of the identity matrix, b is the amount of continuous data per processor in index x, $r = b \times p$ and $d = \lceil \frac{\upsilon_x - \lambda_x + 1}{r} \rceil$ is the number of sequences or wrap arounds.

For example, let an array, $a[(1..64)]$, be partitioned across $p = 4$ processors in a wrapped manner, such that the number of continuous data elements in any processor is $b = 8$. Now, as $r = p \times b, 32 = 4 \times 8$, this implies that the data has to be wrapped around twice, $d = \lceil \frac{64}{32} \rceil$. Therefore processor $z = 1$ will have the elements $a[(1..8), (33..40)]$ local whilst, processor $z = 3$ will have the elements $a[(17..24), (49..56)]$ local etc.

It is often necessary to partition an array by more than one dimension. If the array is to be partitioned by q indices where

$$1 \le q \le N_v \tag{5.6}$$

then the processor space, P, has to be rearranged as follows:

$$P \mapsto P_1 \times P_2 \times \cdots \times P_q \tag{5.7}$$

As there are p processors we have

$$(p_1 \times p_2 \times \cdots \times p_q) = p \tag{5.8}$$

```
c := for i in 1, 200 cross j in 1,300
     returns array of  i +j
     end for;
a := for i in 1, 100 cross j in 1,100
     returns array of  c[i,j]
     end for;
```

Figure 5.1: A Sisal Fragment

where p_i is the number of processors in the ith dimension , $1 \leq i \leq q$. Each dimension of the processor space corresponds to a particular dimension of the array. Let the array dimension associated with processor dimension s be χ_s. Thus the array dimensions to be partitioned are given by:

$$\chi_1, \chi_2, \ldots, \chi_q \tag{5.9}$$

Thus 5.3 can be reformed to define the value of the **local** indices in some processor z. The range of the local indices in equation 5.3 is of the form given in 5.10.

$$\begin{bmatrix} -I \\ I \end{bmatrix} \underline{\mathcal{I}_z^k} \leq \begin{bmatrix} -\underline{\lambda_z} \\ \underline{\upsilon_z} \end{bmatrix} \tag{5.10}$$

$$\begin{bmatrix} -I_N \\ I_N \\ \hline -e_{\chi_1} \\ \vdots \\ -e_{\chi_q} \\ \hline e_{\chi_1} \\ \vdots \\ e_{\chi_q} \end{bmatrix} \begin{bmatrix} \underline{i_1^k} \\ \underline{i_2^k} \\ \vdots \\ \underline{i_{\chi_1}^k} \\ \vdots \\ \underline{i_{\chi_q}^k} \\ \vdots \\ \underline{i_N^k} \end{bmatrix} \leq \begin{bmatrix} -\lambda \\ \upsilon \\ \hline -[(z-1)\times b_{\chi_1} + k\times r_{\chi_1} + 1] \\ \vdots \\ -[(z-1)\times b_{\chi_q} + k\times r_{\chi_1} + 1] \\ \hline z\times b_{\chi_1} + k\times r_{\chi_1} \\ \vdots \\ z\times b_{\chi_q} + k\times r_{\chi_q} \end{bmatrix} \tag{5.11}$$

The form of z, d and k have to be altered to accommodate the partitioning by more than one index.

$$z = (z_1, z_2, \ldots, z_q) \tag{5.12}$$

$$d = (d_1, d_2, \ldots, d_q) \tag{5.13}$$

$$k = (k_1, k_2, \ldots, k_q) \tag{5.14}$$

To illustrate the partitioning of data consider the Sisal fragment described in figure 5.1, where the index ranges for both arrays, a and c, are to be be determined. Array a has the following bounds.

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix}_a \leq \begin{bmatrix} -1 \\ -1 \\ \hline 100 \\ 100 \end{bmatrix} \tag{5.15}$$

Which translates to the imperative array declaration in 5.16:

$$a[(1..100),(1..100)] \tag{5.16}$$

Similarly the array bounds for c are :

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} i_1 \\ i_2 \end{array}\right]_c \leq \left[\begin{array}{c} -1 \\ -1 \\ \hline 200 \\ 300 \end{array}\right] \tag{5.17}$$

In other words the array c has the following size:

$$c[(1..200),(1..300)] \tag{5.18}$$

Let the number of processors be $p = 16$ and let the array be partitioned along two dimensions. In other words the array is to be partitioned into blocks giving $q = 2$, $\chi_1 = 1$ and $\chi_2 = 2$. For convenience let $p_1 = p_2 = \sqrt{p} = 4$. In this example a is to be partitioned in a folded manner with no wrap around of data, so the amount of data in each processor is $\frac{100}{4} = 25$ per row, and $\frac{100}{4} = 25$ per column. This can be represented in the general case by:

$$b_{\chi_i} = \frac{(\upsilon_{\chi_i} - \lambda_{\chi_i} + 1)}{p_i} \forall i \in 1,\ldots,q \tag{5.19}$$

After the removal of redundant constraints, consider the the index values of array a on, say, processor $z = (3,2)$:

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \underline{i_{\chi_1}} \\ \underline{i_{\chi_2}} \end{array}\right]_a \leq \left[\begin{array}{c} -51 \\ -26 \\ \hline 75 \\ 50 \end{array}\right] \tag{5.20}$$

In other words the array a has the following size on this processor:

$$a[(51..75),(26..50)] \tag{5.21}$$

The amount of continuous data per processor in each dimension $p_1 = 1,\ldots,4$ and $p_2 = 1,\ldots,4$ is 25. Thus $b_{\chi_1} = 25$, $b_{\chi_2} = 25$ $r_{\chi_1} = 4 \times 25 = 100$ and $r_{\chi_2} = 4 \times 25 = 100$. At the stage of mapping data to processors, it is assumed that the arrays have been aligned so each element $c[i,j]$ should be in the same processor as $a[i,j]$. As c is of a greater size than a, c will be mapped in a wrapped manner.

$$d_{\chi_1} = \lceil \frac{200}{100} \rceil = 2 \tag{5.22}$$

$$d_{\chi_2} = \lceil \frac{300}{100} \rceil = 3 \tag{5.23}$$

Therefore $k_1 = 1,\ldots,2$, $k_2 = 1,\ldots,3$ and array c will be wrapped around twice in one direction and three times in the other. After removing the redundant constraints of the polytope, the index values of array c are:

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \underline{i_{\chi_1}} \\ \underline{i_{\chi_2}} \end{array}\right]_c \leq \left[\begin{array}{c} -51 \\ -26 \\ \hline 75 \\ 50 \end{array}\right], \tag{5.24}$$

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \underline{i_{\chi_1}} \\ \underline{i_{\chi_2}} \end{array}\right]_c \leq \left[\begin{array}{c} -51 \\ -126 \\ \hline 75 \\ 150 \end{array}\right], \ldots \tag{5.25}$$

$$\ldots \left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \underline{i_{\chi_1}} \\ \underline{i_{\chi_2}} \end{array}\right]_c \leq \left[\begin{array}{c} -151 \\ -226 \\ \hline 175 \\ 250 \end{array}\right] \tag{5.26}$$

i.e.

c[(51..75 , 26..50) , (51..75 , 126..150) , (51..75 , 226..250) , (151..175 , 26..50) , (151..175 , 126..150), (151..175 , 226..250)]

The mapping results in the elements a[(51..75),(26..50)] and c[(51 .. 75) , (26 .. 50)] being present in the same processor which preserves alignment. The mapping of data to processors is always done with respect to the smallest range as distributing the data with respect to a largest array implies either non-even distribution of data, or non-alignment of the smaller array. Hence mapping with respect to the smallest array is employed, despite the inconvenience of wrapped mappings and discontinuous index values

5.1.2 Mapping Iterators

Once the indices of a particular processor have been determined, it is necessary to determine the local iteration space for each processor where all writes are local[1] The iteration space is partitioned in a similar manner to that of the index space, namely:

$$\zeta : J \mapsto \underline{J_1} \times \underline{J_2} \times \cdots \times \underline{J_p} \tag{5.27}$$

The **local** iterators of a processor z are denoted by $\underline{J_z}$ which are defined over as the lattice points of the local iteration space:

$$A\underline{J_z} \leq \underline{b} \tag{5.28}$$

This polytope is derived from:

$$\left[\begin{array}{c} -L \\ \hline U \\ \hline \mathcal{E} \\ \hline -\mathcal{C} \\ \hline \mathcal{C} \end{array}\right] \left[\begin{array}{c} \underline{j_1} \\ \underline{j_2} \\ \vdots \\ \underline{j_m} \end{array}\right] \leq \left[\begin{array}{c} -l \\ \hline u \\ \hline \varepsilon \\ \hline -\underline{\lambda_z} \\ \hline \underline{\upsilon_z} \end{array}\right] \tag{5.29}$$

Adding the index bounds of the local array to the polytope guarantees local writes. As many of the entries in the polytope will prove to be redundant, they should be removed, using the techniques described in chapter 3. As an example, consider the program in figure 5.2.

We will assume the data space is to be partitioned by blocks, and the total number of processors is 16, 4 in each dimension. Consider the iteration space of processor $z = (3, 3)$ where the index range of array a is a[(9..12),(9..12)]. The local iteration space is given by:

[1]This is true for creation parallelism, however, when translating for reduction parallelism, code is inserted to ensure that writes are local (see appendix B). The only modification required is that $\mathcal{C}$ be replaced by the reduction array $\mathcal{U}_{1.v}$ in 5.29.

```
a: = for i in 1,16 cross j in 1,16
     returns array of
       for k in 1,i
       returns value of sum
         if (j>=k)
         then b[i,k] * b[j,k]
         else 0
         end if
       end for
     end for
```

Figure 5.2: A Sisal Program

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ \hline 0 & -1 & 1 \\ \hline -1 & 0 & 0 \\ 0 & -1 & 0 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] \left[\begin{array}{c} \underline{j_1} \\ \vdots \\ \underline{j_3} \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ -1 \\ \hline 16 \\ 16 \\ 0 \\ \hline 0 \\ \hline -9 \\ -9 \\ \hline 12 \\ 12 \end{array}\right] \tag{5.30}$$

By removing redundant conditions of the polytope, as described in chapter 3, the following polytope is found:

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ \hline 0 & -1 & 1 \end{array}\right] \left[\begin{array}{c} \underline{j_1} \\ \vdots \\ \underline{j_3} \end{array}\right] \leq \left[\begin{array}{c} -9 \\ -9 \\ -1 \\ \hline 12 \\ 12 \\ 0 \\ \hline 0 \end{array}\right] \tag{5.31}$$

which corresponds to the program in figure 5.3. This can be extended to determine the iteration space of wrapped data, by calculating the separate iteration spaces of the *d* wrapped sections of an array.

5.1.3 Interleaving

One method to improve the load balancing of a system is to interleave the work load. In chapter 3 the interleaving of iterators was presented for this purpose. This section describes how interleaving affects data distribution and the iteration space.

Interleaving an iterator, using the interleave function #, implies that every reference to that iterator is interleaved.

```
FOR i =  9 TO 12
   FOR j =  9 TO 12
     FOR  k = 1 TO i
       IF (k<=j)
       THEN a[i,j] := a[i,j] +  Get (b[i,k])
                            *  Get (b[j,k])
       END IF
     END FOR
   END FOR
 END FOR
```

Figure 5.3: An Imperative Program

```
a: = for i in 1,16
     returns array of
       for j in 1,16
       returns value of sum
         if (j<=i)
         then b[i] * b[j] +i
         else 0
         end if
       end for
     end for
```

Figure 5.4: A Sisal Program

This includes array occurrences, conditionals and general usage in expressions. For example consider the program in figure 5.4, on interleaving the *i* iterator, the program in figure 5.5 is formed.

This example illustrates two potential problems

1. The iteration space is no longer convex and hence cannot be represented as a lattice of points contained within a polytope
2. The c-occurrence is interleaved and it is not obvious whether the data to be written to is local.

```
FOR i =  1 TO 16
   FOR j =  1 TO 16
    IF (j <=#i)
    THEN a[#i] := a[#i] + Get ( b[#i])
                * Get ( b[j]) + #i
    END IF
   END FOR
END FOR
```

Figure 5.5: An Imperative Program

```
a: = for i in 1,16
     returns array of
       for j in 1,i
       returns value of sum
         b[i] * b[j] + i
       end for
     end for
```

Figure 5.6: A Sisal Program

Fortunately both these potential problems can be solved.

1. In general the iteration space is not convex, but each **local** iteration space after partitioning is in fact convex, and can be represented as a polytope. The interleave function is defined as

$$\#\underline{j} = z + p(\underline{j} - \underline{jlo}) \tag{5.32}$$

   where $\underline{j}$ is the local interleaved iterator, z is the processor number, p is the total number of processors and $\underline{jlo}$ is the lower bound of the iterator .

2. The c-occurrence matrix can be restored to its original form by applying the inverse of #, defined as ##, to the interleaved **indices** of the c-occurrence and all aligned arrays, In [ROGE91] the existence and a method for calculating the inverse interleave function ## is shown.

Rectangular loop bounds are relatively easy to interleave, but not all iteration spaces are rectangular. However it is possible to express all iterations spaces as rectangular ones which are cut by hyperplanes, where the hyperplanes correspond to **if** conditions.

For example consider the program in 5.6 where the i iterator is to be interleaved. In its present form, it is not obvious how to perform interleaving.

This program has the following polytope:

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ -1 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ \hline 16 \\ 0 \end{array}\right] \tag{5.33}$$

This can be rewritten as

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline -1 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ \hline 16 \\ 16 \\ \hline 0 \end{array}\right] \tag{5.34}$$

Moving from the polytope in 5.34 to 5.33 has been the concern of constraint removal in chapter 3, section 3.3.4, however, in this example the opposite is required. The necessary techniques required to add constraints have also been described in chapter 3, section 3.3.3, where it was used for loop interchange. If we consider the local program on the second of four processors, then figure 5.7 describes the program after interleaving one of the iterators and one of the indices.

```
  FOR i =  5 TO 8
    FOR j =  1 TO 16
      IF (j <= #i)
      THEN
         a[i] := a[i] +  Get (b[i])
                     *  Get (b[##j]) + #i
      END IF
    END FOR
 END FOR
```

Figure 5.7: An Imperative Program

Interleaving an array's indices has similar consequences to those of the alignment transformations, in that the effect has to be propagated throughout the remainder of the program. Such a procedure is described in chapter 4, section 4.4.

5.2 Pre-Fetching

This section is concerned with the minimising the amount of non-local access, once alignment and data partitioning have taken place. Essentially if a non-local data element is to be accessed more than once, then it is preferable to store it locally after the first access. This is achieved by pre-fetching, where the data item is accessed before it is needed and stored in a local temporary variable. The criteria for data re-use and the transformations required to achieve it, are described within the following sub-sections. Other researchers have been working in this area, most notably M.E. Wolf and M. Lam [WOLF91]. One of the contributions of this thesis is that a methodology for re-use is developed which allows the application of re-use transformations to a wide range of programs. Specifically, the transformations can be applied to general affine loops in the presence of affine conditionals and array occurrences.

If reduction parallelism is exploited, there is no need for pre-fetching transformations as all the data required for the local computation will be local. Pre-fetching is, therefore, only relevant in compiling for creation parallelism.

Initially, an invariancy condition is derived which determines if pre-fetching is worth while, whereupon a transformation based upon loop interchange is used. In the case of multiple array accesses, it is generally necessary to scalar expand one or more array references, but this is limited by the availability of memory and the need to provide a scalable implementation. These points are covered in the next two sub-sections. In the final sub-section, strip-mining is introduced as a means of maximising the amount of pre-fetching but maintaining the scalability constraint.

5.2.1 Invariance

To motivate the analysis and the remainder of this section, consider the program in figure 5.8. If the array to be created is partitioned by j, 'column-wise', this will result in the program described in figure 5.9, where *jlo* and *jhi* are the local lower and upper bounds of j which have been determined by the mapping transformation.

The amount of array b that is accessed by the reference $b[j,k]$ is $|j| \times |k|$ where $|j|$ and $|k|$ are the number of iteration

```
a := for i in 1,n cross j in 1,n
     returns array of
       for k in 1,n
       returns value of sum
         b[j,k]
       end for
     end for
```

Figure 5.8: A Sisal Program

```
FOR i = 1 TO n
  FOR j = jlo TO  jhi
    FOR k = 1 TO n
       a[i,j] := a[i,j] + Get( b[j,k])
    END FOR
  END FOR
END FOR
```

Figure 5.9: An Imperative Program

points of j and k respectively. The **volume of access** is

$$|jlo, \dots, jhi| \times n = \frac{n^2}{p} \tag{5.35}$$

However, the number of times b is, at present, accessed is defined by the number of loop iterations performed:

$$n \times |jlo, \dots, jhi| \times n = \frac{n^3}{p} \tag{5.36}$$

There are an $O(n)$ more non-local accesses than is necessary. To achieve the value determined by the volume of access calculation, the program has to be re-ordered to give the program in figure 5.10, where *tempb* is a local variable. In this example the (potentially non local) array, b, is referenced the minimum number of times with the remaining references being made to a local variable. This reordering is based upon the analysis of data re-use and general pre-fetching transformations. Given an $(N_v \times m)$ u-occurrence $\mathcal{U}_{1.v}$ and the iteration vector J^m, then if the rank of $\mathcal{U}$ is d, $d < m$, there are $|\underline{J^{m-d}}|$ points of reuse. The program in figure 5.8 has the occurrence matrix:

```
FOR k = 1 TO n
  FOR j := jlo TO  jhi
    tempb:= Get(b[j,k])
    FOR i = 1 TO n
       a[i,j] := a[i,j] + tempb
    END FOR
  END FOR
END FOR
```

Figure 5.10: An Imperative Program

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \tag{5.37}$$

There is no reference to iterator i in the u-occurrence matrix of b, which implies that the value of $\mathcal{U}_{1.b}J^m$ remains constant for all values of i. In other words $\mathcal{U}_{1.b}$ is **invariant** of the i iterator.

Definition 14 *The null space of $\mathcal{U}_{1.v}$, $\mathcal{N}(\mathcal{U}_{1.v})$, are the iterators which $\mathcal{U}_{1.v}$ is invariant of.*

In chapter 4, section 4.2.2 it was shown that a u-occurrence matrix, $\mathcal{U}_{1.v}$ could only be aligned with a row of the c-occurrence matrix, c_j, if $c_j \perp \mathcal{N}(\mathcal{U}_{1.v})$. Therefore, either an array occurrence may be aligned or it is a candidate for re-use.

Before pre-fetching transformations are applied there is one overall restriction, that is, no reference may occur before its definition. One important example of this is that if an array is defined within a **foriter** loop the reference to it may not be moved outside the loop.

The number of points referenced by a u-occurrence of an array v is $|\mathcal{U}_{1.v}\underline{J^m}|$, where $\underline{J^m}$ is the local iterator vector. If the rank of $\mathcal{U}_{1.v}$ is d and $m > d$, then it is desirable to reduce the iterators enclosing an array reference from $\underline{J^m}$ to $\underline{J^d}$ where $\underline{J^d}$ are the iterators accessed, as this will reduce the total potential non-local accesses.

It is necessary to consider the form of a computation set before the pre-fetching transformation can be formally stated. Q is defined in chapter 2, section 2.2, as a computation set whose elements are the iteration space, array occurrences and a parse tree containing the operations to be performed. $\underline{Q}$ is now defined as the local computation set which is identical to Q , except for its iteration space.

In the equation 5.38 the transformation has the form $\underline{Q} \mapsto (.., (\underline{Q_1}, \underline{Q}'))$, which is equivalent to creating a new computation set whose body contains two further computation sets. Transformations of this complexity have not been previously introduced, but are necessary here. In effect an array creation is split into two parts, the first accesses non-local data and stores it in a local temporary, while the second accesses that temporary. These two parts correspond to the ordered pair $(\underline{Q_1}, \underline{Q}')$. There is a, transformation introduced, data-dependency between the two computation sets, which is preserved by nesting them within another computation set. The overall transformation is:

$$\pi : \underline{Q} \mapsto (A, \hat{\underline{J}}, b, (\underline{Q_1}, \underline{Q}')) \tag{5.38}$$

where

$$\underline{Q_1} = \underline{Q}(\underline{J^m} \mapsto \emptyset, tempv := v[\mathcal{U}_{1.v}\underline{J^m}]) \tag{5.39}$$

$$\underline{Q}' = \underline{Q}(\underline{J^m} \mapsto \check{\underline{J}}, v[\mathcal{U}_{1.v}\underline{J^m}] \mapsto tempv) \tag{5.40}$$

and

$$\hat{\underline{J}} = \underline{J} - \mathcal{N}(\mathcal{U}) \tag{5.41}$$

$$\check{\underline{J}} = \mathcal{N}(\mathcal{U}) \tag{5.42}$$

tempv is a temporary variable introduced to store the prefetched value. The mapping $v[\mathcal{U}_{1.v}\underline{J^m}] \mapsto tempv$ describe the substitution of an array reference by a local temporary. $\hat{\underline{J}}$ is the d dimensional vector of the d iterators that $\mathcal{U}_{1.v}$ makes reference to, and conversely $\check{\underline{J}}$ are the $m - d$ iterators that $\mathcal{U}_{1.v}$ does not make reference to. Intuitively $\hat{\underline{J}}$ should be the outer most loops and $\check{\underline{J}}$ the inner most ones in the transformed program where $\emptyset$ represents the empty iterator vector.

```
a := for i in 1,n cross j in 1,n
    returns array of
      for k in 2*i+j, 3*i-j
      returns value of sum
        if k >= n then b[k,2*k]
        else 0 end if
      end for
    end for
```

Figure 5.11: A Sisal Program

The determination of this transformation 5.38 is relatively easy if $\mathcal{N}(\mathcal{U}_{1.v})$ is aligned with respect to the iteration space. Therefore in this section we concentrate on u-occurrences that contain only one iterator variable per reference such as $a[i,k,j+3]$, $a[2*i,j,j]$ as opposed to more complex occurrences such as $a[i,j+k]$. General affine occurrences will be discussed in section 5.2.5. To determine 5.38, the main task is to perform the reordering transformation:

$$\underline{J} \mapsto \begin{bmatrix} \hat{\underline{J}} \\ \check{\underline{J}} \end{bmatrix} \tag{5.43}$$

This transformation requires $O(m)$ interchanges of iterators which has been covered in chapter 3, section 3.2.7. To illustrate the usefulness of this technique, consider the program in figure 5.11. If we concentrate on the polytope involved in the access of array b, then it has the following iteration space:

$$\left[\begin{array}{rrr} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 2 & 1 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 1 & 1 \\ \hline 0 & 0 & -1 \end{array}\right] \begin{bmatrix} i \\ j \\ k \end{bmatrix} \le \left[\begin{array}{r} -1 \\ -1 \\ 0 \\ \hline n \\ n \\ 0 \\ \hline -n \end{array}\right] \tag{5.44}$$

Now

$$\check{\underline{J}} = \mathcal{N}(\mathcal{U}_{1.b}) = [i,j]^T \tag{5.45}$$

and

$$\hat{\underline{J}} = \underline{J} - \mathcal{N}(\mathcal{U}_{1.b}) = [k]^T \tag{5.46}$$

Section 5.1.3, dealing with interleaving, described a method whereby general iteration spaces can be re-expressed as rectangular ones restricted by conditionals. On interchanging the iterators by methods described in chapter 3, we have:

$$\left[\begin{array}{rrr} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 1 & 2 & 1 \\ 1 & -3 & 1 \end{array}\right] \begin{bmatrix} k \\ i \\ j \end{bmatrix} \le \left[\begin{array}{r} -n \\ -1 \\ -1 \\ \hline 3n-1 \\ n \\ n \\ \hline 0 \\ 0 \end{array}\right] \tag{5.47}$$

which on applying 5.38 gives the program in figure 5.12. This program illustrates the power of pre-fetching

```
FOR  k = n TO 3*n-1
  tempb := Get(b[k,2*k])
  FOR i = 1 TO n
    FOR j = 1 TO n
      IF ((k+j)<= -2*i) AND ((k+j)<=3*i)
      THEN   a[i,j] := a[i,j] + tempb
      END IF
    END FOR
  END FOR
END FOR
```

Figure 5.12: An Imperative Program

transformations. By combining the re-use analysis and general loop interchange, the number of potentially non-local accesses has been significantly reduced and works in the presence of general affine loop limits and conditional evaluation.

A computation set usually has more than one u-occurrence each with a different $\underline{\check{J}}$. Let $\underline{\check{J}_{r.v}}$ be the local invariant iterators of the rth u-occurrence $r \in 1, \ldots, \theta_v$ of array v. The iterators common to each $\underline{\check{J}_{r.v}}$ will be the iterators that all the u-occurrences will be invariant of. It is convenient to describe each of the vectors $\underline{\check{J}_{r.v}}$ in set notation as the common iterators naturally corresponds to set intersection. Two functions, *vtos, stov* are required to translate a vector into a set and a set into a vector:

$$vtos : [j_1, \ldots, j_m]^T \mapsto \{j_1, \ldots, j_m\} \tag{5.48}$$

$$stov : \{j_1, \ldots, j_m\} \mapsto [j_1, \ldots, j_m]^T \tag{5.49}$$

For the sake of convenience, curly brackets, $\{\}$ surrounding a vector, denote a set representation of that vector, i.e. the function *vtos* has been applied. For example, $\{J^m\} = \{j_1, \ldots, j_m\}$ where $J^m = [j_1, \ldots, j_m]^T$. Given $\{J^m\}$ it is possible to determine J^m by applying the natural ordering function *stov*.

The intersection of each of the $\{\underline{\check{J}_{r.v}}\}$ will be the iterators that all of the u-occurrences are invariant of. The size of this intersection may be much smaller than the $\{\underline{\check{J}_{r.v}}\}$ of each u-occurrences and an opportunity to exploit invariance may be lost.

The common invariant iterators are defined by the intersection of each $\{\underline{\check{J}_{r.v}}\}$ which is $\{\underline{\check{J}}\} = \bigcap_{r=1}^{\theta_v}\{\underline{\check{J}_{r.v}}\}, \forall v$.

The difference $\{\underline{\check{J}_{r.v}}\} - \{\underline{\check{J}}\}$ is the shortfall in exploitation of invariance of $\mathcal{U}_{r.v}$ and will lead to excessive non-local access. To illustrate this point, consider the well known matrix multiplication program shown in figure 5.13. After a straightforward translation the form described in figure 5.14 is derived. Array b is invariant of iterator j, but array c is invariant of iterator i, so there are **no common invariant iterators**.

$$\underline{\check{J}_b} = j, \underline{\check{J}_c} = k, \{\underline{\check{J}}\} = \{j\} \cap \{k\} = \{\emptyset\} \tag{5.50}$$

so there is a shortfall in the exploitation of invariancy:

$$\{\underline{\check{J}_b}\} - \{\underline{\check{J}}\} = \{j\} \tag{5.51}$$

$$\{\underline{\check{J}_c}\} - \{\underline{\check{J}}\} = \{k\} \tag{5.52}$$

By reordering the iterators, the invariancy of either b or c may be realised but not both. This is not really satisfactory, as the remaining u-occurrence will dominate the non-local access cost.

```
a:= for i in 1,n cross j in 1,n
    returns array of
      for k in 1,n
      returns value of sum
        b[i,k]*c[k,j]
      end for
    end for
```

Figure 5.13: A Sisal Program

```
FOR  i= 1 TO n
  FOR j =  1 TO n
    FOR k = 1 TO n
      a[i,j] :=  a[i,j] + Get(b[i,k]) * Get(c[k,j])
    END FOR
  END FOR
END FOR
```

Figure 5.14: An Imperative Program

5.2.2 Scalar Expansion

One approach to improve exploitation of invariant access, is to pre-fetch data which is not invariant of the relevant iterator. The temporary introduced will be scalar expanded [PADU86] by this action and will be the same size as the range of the iterator. This scalar expansion is appropriate if it allows the realisation of the invariancy of other u-occurrences.

Let $\mathcal{M}$ be the iterators that are going to be innermost. Each u-occurrence will be prefetched with respect to $\mathcal{M}$ and may require some scalar expansion. If $\underline{\check{J}_{r.v}}$ is the invariant iterators of the *r*th occurrence of *v* then prefetching, which may include scalar expansion, is only appropriate if it exploits, or reveals, some invariancy of access. Pre-fetching of $\mathcal{U}_{r.v}$ is worth while if:

$$\{\mathcal{M}\} \cap \{\underline{\check{J}_{r.v}}\} \neq \{\emptyset\} \tag{5.53}$$

Given $\mathcal{M}$ let $\{\underline{\tilde{J}_{r.v}}\}$ be defined as

$$\{\underline{\tilde{J}_{r.v}}\} = \{\mathcal{M}\} - \{\underline{\check{J}}\} \ \forall r \in 1, \ldots, \theta_v \forall v \tag{5.54}$$

In other words $\underline{\tilde{J}_{r.v}}$ are the iterators which any introduced temporary for the *r*th u-occurrence of *v* will be scalar expanded by. The size of any temporary introduced by scalar expansion will be:

$$||\mathcal{U}_{r.v}\underline{\tilde{J}_{r.v}})||, \tag{5.55}$$

Pre-fetching with scalar expansion can be described as the following transformation :

$$\pi : \underline{Q} \mapsto (A, \underline{\hat{J}}, b, (\underline{Q_{1.v}}, \ldots, \underline{Q_{\theta_v.v}}) \forall v, \underline{Q'}) \tag{5.56}$$

$$\underline{Q_{r.v}} = \underline{Q}(\underline{J^m} \mapsto \underline{\tilde{J}_{r.v}}, temprv[\mathcal{U}_{r.v}\underline{J^m}] := v[\mathcal{U}_{r.v}\underline{J^m}) \forall r \in 1, \ldots, \theta_v \forall v \tag{5.57}$$

$$\underline{Q'} = \underline{Q}(\underline{J^m} \mapsto \mathcal{M}, v[\mathcal{C}\underline{J^m}] \mapsto temprv \ \forall r \in 1, \ldots, \theta_v \forall v) \tag{5.58}$$

where

$$\{\underline{\hat{J}}\} = \{\underline{J^m}\} - \{\mathcal{M}\} \tag{5.59}$$

```
FOR k = 1 TO n
  FOR i =  ilo TO ihi
    tempb[i] := Get(b[i,k])
  END FOR
  FOR j =  jlo TO jhi
    tempc[j] := Get(c[k,j])
  END FOR
  FOR i =  ilo TO ihi
    FOR j =  jlo TO jhi
      a[i,j] := a[i,j] +  tempb[i] * tempc[j]
    END FOR
  END FOR
END FOR
```

Figure 5.15: An Imperative Program

Maximum invariancy occurs when:

$$\{\mathcal{M}\} = \bigcup_{r=1}^{\theta_v} \{\underline{\check{J}_{r.v}}\} \forall v \tag{5.60}$$

In other words, the maximum amount of re-use of data occurs when all invariant iterator are placed innermost.

This is equivalent to saying if a u-occurrence is sufficiently rank deficient, so as to imply re-use, then the transformation will exploit this invariancy. As stated previously, $\{\mathcal{M}\} \cap \{\underline{\check{J}_{r.v}}\}$ will be the iterators that $\mathcal{U}_{r.v}$ will be invariant of for a particular $\mathcal{M}$. To illustrate this point consider the matrix multiplication program in figure 5.13. The invariant iterators of arrays b and c are

$$\underline{\check{J}_b} = j, \underline{\check{J}_c} = i, \tag{5.61}$$

Using equation 5.60, $\mathcal{M}$ is

$$\{\mathcal{M}\} = \{j\} \cup \{i\} = \{i,j\} \tag{5.62}$$

In other words, the maximum re-use of data is achieved by pre-fetching with respect to i and j. The temporaries associated with a and b will be scalar expanded when passing through non-invariant loops.

$$\{\underline{J}\} - \{\mathcal{M}\} = \{k\}, \{\mathcal{M}\} - \{\underline{\check{J}_b}\} = \{i\}, \{\mathcal{M}\} - \{\underline{\check{J}_c}\} = \{j\} \tag{5.63}$$

So the outer most iterator is k, the reference to array b is scalar expanded by i and the reference to c is scalar expanded by j. Applying the new transformation and subsequent translation to imperative form, gives the program in figure 5.15. In this program the potential non-local accesses have been reduced by $O(\frac{n}{\sqrt{p}})$. However it has incurred the cost of two scalar expanded temporaries, both of which are of size $(\frac{n}{\sqrt{p}})$, as the size of the temporaries depends on the size of the loops involved in the scalar expansion.

At this point it may be asked why all array u-occurrences are not scalar expanded to the outermost lexical level. Aside from the constraint that a u-occurrence may not be moved past its definition, the amount by which an occurrence may be expanded is limited by the scalability constraint. The determination of the scalability constraint is the subject of the next sub-section.

5.2.3 Scalability Constraint

In order to maintain scalability, a limit must be placed upon the size of the temporaries introduced by scalar expansion. The size of temporaries has to be the same order of magnitude in size, or less, as the data being

```
FOR i = ilo TO ihi
  FOR j = jlo TO jhi
    FOR k = 1 TO n
      FOR l = 1 TO n
        a[i,j] := a[i,j] + Get(b[i,k])
                         * Get(c[l,j])
      END FOR
    END FOR
  END FOR
END FOR
```

Figure 5.16: An Imperative Program

locally created. For the sake of simplicity, the amount of data allowed to be pre-fetched is restricted to the size of the the local data being created. Any constant multiple is satisfactory and will depend on the actual amount of memory available in a particular implementation.

If $||\underline{\mathcal{I}^{\mathcal{C}}}||$ is the amount of local data per processor being created then the constraint on the amount of scalar expansion by $\underline{\tilde{J}_{r.v}}$ is given by:

$$||\mathcal{U}_{r.v}\underline{\tilde{J}_{r.v}}|| \leq ||\underline{\mathcal{I}^{\mathcal{C}}}||, \forall r \in 1, \ldots, \theta_v, \forall v \tag{5.64}$$

To illustrate the effect of this constraint on the exploitation of invariance, consider the program in figure 5.16. Array a has been partitioned by rows and columns and therefore both the iterators i and j are partitioned. Assuming a square grid of p processors, both the ranges of i and j are $\frac{n}{\sqrt{p}}$. The invariant iterators are simply calculated:

$$\underline{\check{J}_b} = [j, l]^T \;\; \underline{\check{J}_c} = [i, k]^T \;\; \{\underline{\check{J}_b}\} \cap \{\underline{\check{J}_c}\} = \{\emptyset\} \tag{5.65}$$

If $\mathcal{M}$ is defined so as to maximise invariancy then

$$\{\mathcal{M}\} = \{\underline{\check{J}_b}\} \cup \{\underline{\check{J}_c}\} = \{i, j, k, l\} \tag{5.66}$$

and

$$\underline{\tilde{J}_b} = [i, k]^T \;\; \underline{\tilde{J}_c} = [j, l]^T \tag{5.67}$$

where the temporaries will be of the following size:

$$||\mathcal{U}_b\underline{\tilde{J}_b}|| = \frac{n^2}{\sqrt{p}} \;\; ||\mathcal{U}_c\underline{\tilde{J}_b}|| = \frac{n^2}{\sqrt{p}} \tag{5.68}$$

But

$$||\mathcal{U}_b\underline{\tilde{J}_b}|| > \frac{n^2}{p}, ||\mathcal{U}_c\underline{\tilde{J}_c}|| > \frac{n^2}{p} \tag{5.69}$$

Here scalar expansion with respect to $\underline{\check{J}_b}$ and $\underline{\check{J}_c}$ invalidates the scalability constraint. Ideally $\mathcal{M}$ should be chosen so that 5.64 holds at equality. If $\{\mathcal{M}\}$ is chosen to be a sub-set of $\{\underline{\check{J}_b}\} \cup \{\underline{\check{J}_c}\}$ then the constraint 5.64 will be satisfied, but will not hold at equality, and an opportunity to exploit invariance may be lost. The solution is presented in the following section.

5.2.4 Strip-Mining

By breaking an iterator into two sub-iterators, which cover the same number of lattice points, it is possible to exploit invariancy which would otherwise be lost. The range of one of the iterators is chosen so that it is possible

```
a:= for i in 1,16 cross j in 1,16
    returns array of
        b[i,j]
    end for
```

Figure 5.17: A Sisal Program

to scalar expand a temporary, with respect to the iterator, without breaking the scalability constraint. The act of splitting an iterator into two or sub-iterators is known as strip-mining.

On strip-mining an iterator, the new loop bounds must be found and all references to the old iterator updated. Initially, strip-mining in a rectangular iteration space is described which is easily extended to the general affine case. To motivate the formal analysis of strip-mining, consider the program in figure 5.17. This has the following iteration space:

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ \hline 16 \\ 16 \end{array}\right] \tag{5.70}$$

If the j loop is strip-mined to give two new iterators $\overleftarrow{j}, \overrightarrow{j}$, then the new polytope is given by:

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \left[\begin{array}{c} i \\ \overleftarrow{j} \\ \overrightarrow{j} \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ -1 \\ \hline 16 \\ 4 \\ 4 \end{array}\right] \tag{5.71}$$

Instead of the j loop ranging from 1 to 16, the two new iterators iterate over a smaller range but cover the same number of points $4 \times 4 = 16$ In general if an iterator is of the following form:

$$l \leq j \leq u \tag{5.72}$$

then the corresponding strip-mined iterators will be:

$$1 \leq \overleftarrow{j} \leq x \tag{5.73}$$

$$l \leq \overrightarrow{j} \leq \frac{(u-l+1)}{x} + (l-1) \tag{5.74}$$

where x is the strip-mining 'width' which divides the range of j, in this example $x = 4$. After the strip-mining of the j loop the u-occurrences must be updated. Before strip-mining the array reference to b was $b[i, j]$, and after strip-mining it will be $b[i, 4(\overleftarrow{j}-1)+\overrightarrow{j}]$ or $b[i, 4\overleftarrow{j}-4+\overrightarrow{j}]$ on expansion. In general each reference to j must be replaced by $\frac{(u-l+1)}{x}(\overleftarrow{j}-1)+\overrightarrow{j}$.

After strip-mining the j loop and updating the references to j, the program in figure 5.18 is given. Strip-mining can now be expressed more formally. The local iteration space is of the general form:

```
FOR i = 1 TO 16
  FOR j1 = 1 TO 4
    FOR j2 = 1 TO 4
      a[i,4*j1-4+j2] := Get (b [i,4*j1- 4+j2])
    END FOR
  END FOR
END FOR
```

Figure 5.18: An Imperative Program

$$\begin{bmatrix} \underline{-L} \\ \underline{U} \\ \underline{\mathcal{E}} \end{bmatrix} \underline{J^m} \le \begin{bmatrix} \underline{-l} \\ \underline{u} \\ \underline{\varepsilon} \end{bmatrix} \tag{5.75}$$

If the iteration space is rectangular then $L = U = I_m, \mathcal{E} = 0$ and the strip-mining transformation, π, on an iterator, j, has the following effect on the iteration space:

$$\pi : AJ^m \le b \mapsto A^{'}J^{m+1} \le b^{'} \tag{5.76}$$

where $A^{'}$ is a $(2(m+1)\times m)$ integer matrix, $b^{'}$ a $(2(m+1)\times 1)$ integer vector and $J^{'}$ is the new iterator vector. This transformation can be expressed as

$$\begin{bmatrix} -L \\ U \end{bmatrix} J^m \le \begin{bmatrix} -l \\ u \end{bmatrix} \mapsto \begin{bmatrix} -L^{'} \\ U^{'} \end{bmatrix} J^{m+1} \le \begin{bmatrix} -l^{'} \\ u^{'} \end{bmatrix} \tag{5.77}$$

where $L^{'}, U^{'}, l^{'}, u^{'}$ represent the new loop bounds and

$$L^{'}_r = \begin{cases} \begin{bmatrix} L_{r,1..j} & 0 & L_{r,j+1..m} \end{bmatrix} & r \le j \\ \begin{bmatrix} O_{r,1..j} & 1 & O_{r,j+1..m} \end{bmatrix} & r = j+1 \\ \begin{bmatrix} L_{r-1,1..j} & 0 & L_{r-1,j+1..m} \end{bmatrix} & r > j+1 \end{cases} \quad \forall r \in 1,\dots,m+1 \tag{5.78}$$

$$U^{'}_r = \begin{cases} \begin{bmatrix} U_{r,1..j} & 0 & U_{r,j+1..m} \end{bmatrix} & r \le j \\ \begin{bmatrix} O_{r,1..j} & 1 & O_{r,j+1..m} \end{bmatrix} & r = j+1 \\ \begin{bmatrix} U_{r-1,1..j} & 0 & U_{r-1,j+1..m} \end{bmatrix} & r > j+1 \end{cases} \quad \forall r \in 1,\dots,m+1 \tag{5.79}$$

$$l^{'}_r = \begin{cases} l_r & r < j \\ 1 & r = j \\ l_{r-1} & r > j \end{cases} \quad \forall r \in 1,\dots,m+1 \tag{5.80}$$

$$u^{'}_r = \begin{cases} u_r & r < j \\ x & r = j \\ \frac{u_j - l_j + 1}{x} + l_j - 1 & r = j+1 \\ u_{r-1} & r > j+1 \end{cases} \quad \forall r \in 1,\dots,m+1 \tag{5.81}$$

and

$$J^{'}_r = \begin{cases} J_r & r < j \\ \overleftarrow{j} & r = j \\ \overrightarrow{j} & r = j+1 \\ J_{r-1} & r > j+1 \end{cases} \quad \forall r \in 1,\dots,m+1 \tag{5.82}$$

where the iterator j has been strip-mined to form two sub-iterators:

$$j \mapsto \overleftarrow{j} \times \overrightarrow{j} \tag{5.83}$$

```
a:= for i in 1,16 cross j in 2*i+1, i+8
    returns array  of
      b[i,j]
    end for
```

Figure 5.19: A Sisal Program

and x is the upper bound of the $\overleftarrow{j}$ iterator. To extend strip-mining for non-rectangular regions, and to adjust array occurrences, all usages of j must be updated using the following relationship

$$j = \frac{(u - l + 1)}{x}(\overleftarrow{j} - 1) + \overrightarrow{j} \tag{5.84}$$

To illustrate the effect of strip-mining in non-rectangular regions consider the program in figure 5.19.

The iteration space is:

$$\left[\begin{array}{rr} -1 & 0 \\ 2 & -1 \\ \hline 1 & 0 \\ -1 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ \hline 16 \\ 8 \end{array}\right] \tag{5.85}$$

which can be re-ordered to give:

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \\ \hline 2 & -1 \\ -1 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ \hline 16 \\ 24 \\ \hline 1 \\ 8 \end{array}\right] \tag{5.86}$$

If the strip-mining width is chosen to be 4 i.e. $x = 4$ then we have:

$$\left[\begin{array}{rrr} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 2 & -4 & -1 \\ -1 & 4 & 1 \end{array}\right] \left[\begin{array}{c} i \\ \overleftarrow{j} \\ \overrightarrow{j} \end{array}\right] \leq \left[\begin{array}{r} -1 \\ -1 \\ -1 \\ \hline 16 \\ 4 \\ 2 \\ \hline 5 \\ 12 \end{array}\right] \tag{5.87}$$

After adjusting the array occurrences the program and replacing $\overleftarrow{j}$ by j1 and $\overrightarrow{j}$ by j2, the program in figure 5.20 is derived.

This example illustrates that strip-mining can be performed in general iteration spaces. Strip-mining was introduced to allow pre-fetching without violating the scalability constraint. An algorithm to perform pre-fetching transformations so as to exploit invariancy using scalar expansion and strip-mining is given in appendix c. It is based upon the following observations.

```
FOR i = 1 TO 16
  FOR j1 = 1 TO 4
    FOR j2 = 1 TO 2
      IF ((2*i-4*j1-j2) <= 5)
      AND ((4*j1 +j2-i) <= 12)
      THEN
      a[i,4*j1- 4+j2] := Get(b [i,4*j1-4+j2] )
      END IF
    END FOR
  END FOR
END FOR
```

Figure 5.20: An Imperative Program

1. the scalability constraint must be observed
2. an iterator which is expanded upon should realise invariancy in as many other u-occurrences as possible
3. the choice of iterators should be fair so that each u-occurrence is expanded uniformly so as to avoid one u-occurrence dominating the non-local access.
4. scalar expansion should be with respect to the smallest iterator
5. strip-mine when further expansion by steps 1,2,3,4 is not possible. The choice of iterator should be based on 1,2,3,4

These points can now be applied to the program of figure 5.16 which motivated this sub-section. After expanding by iterators i and j, the iterators k and l are chosen to be strip-mined, where x is chosen such that it satisfies the scalability constraint which finally gives the program in figure 5.21.

This program minimises the non-local access for a given data partition but it is unlikely that such a program would be generated by hand and thus demonstrates the usefulness of the pre-fetching technique.

5.2.5 Affine Occurrences

Previously the case where one of the array sub-scripts makes reference to more than one iterator was ignored i.e. $a[i-j]$ The conditions whereby an occurrence of this form may be prefetched remain the same. However an additional result is required in order to exploit invariancy. To motivate the remainder of this section consider the two programs in figures: 5.22 and 5.23.

In the first program rank($\mathcal{U}_{1.b}$) = 2 = dim(J^2) and thus there is no opportunity to exploit invariance by prefetching. The second program has rank($\mathcal{U}_b$) = 2 < 3 =dim(J^3) and thus there is an opportunity to exploit invariance by prefetching. However the null space of $\mathcal{U}_{1.b}$ in the second program, $\mathcal{N}(\mathcal{U}_{1.b})$, is $j-i$, which is not an iterator but a linear combination of two iterators. Thus it is not possible to prefetch an occurrence of this nature without incurring some scalar expansion. To demonstrate this point consider the program in figure 5.23 which is transformed to give the local program for the first of sixteen processors, as shown in figure 5.24, after partitioning and pre-fetching. Though legal this translation has had no useful effect. However it can be seen that the accesses made by tempb are “almost invariant” of the i iterator, with only one element changing per iteration. If the basis of $\underline{\tilde{J}}$ is changed so that it is orthogonal to $\mathcal{U}_{1.b}$ then the new program in figure 5.25 is formed. The change of

```
FOR k1 = 1 TO p1
  FOR l1 = 1 TO p2
    FOR k2 = 1 TO  n/p1
      FOR i= ilo TO ihi
        tempb[i, k2] := Get(b[i,(k1-1)*(n/p1) + k2])
      END FOR
    END FOR
    FOR l2 = 1 TO  n/p2
      FOR j = jlo TO jhi
        tempc[l2,j] := Get(c[(l1-1)*(n/p2)+l2,j])
      END FOR
    END FOR
    FOR k2 = 1 TO  n/p1
      FOR l2 = 1 TO  n/p2
        FOR i = ilo TO ihi
          FOR j = jlo TO jhi
            a[i,j] := a[i,j]+ tempb[i, k2]
                             *tempc[l2, j]
          END FOR
        END FOR
      END FOR
    END FOR
  END FOR
END FOR
```

Figure 5.21: An Imperative Program

```
a :=  for i in 1,16 cross  j in 1,16
      returns array of b[i,i+j]
      end for
```

Figure 5.22: A Sisal Program

```
a :=  for i in 1,16 cross  j in 1,16
      returns array of
        for k in 1,16
        returns value of sum
           b[k,i+j]
        end for
      end for
```

Figure 5.23: A Sisal Program

```
FOR k = 1 TO 16
  FOR i = 1 TO 4
    FOR j = 1 TO 4
      tempb[i+j] := Get(b[k,i+j])
    END FOR
  END FOR
  FOR i = 1 TO 4
    FOR j = 1 TO 4
      a[i,j] := tempb[i+j]
    END FOR
  END FOR
END FOR
```

Figure 5.24: An Imperative Program

```
FOR k = 1 TO 16
  FOR i = 1 TO 4
    FOR j = i+1 TO i+4
      tempb[j] := Get(b[k,j])
    END FOR
  END FOR
  FOR i = 1 TO 4
    FOR j = 1 TO 4
      a[i,j] := tempb[i+j]
    END FOR
  END FOR
END FOR
```

Figure 5.25: An Imperative Program

basis transformation has been described in section 3.2.2 where it was used to reveal invariancy of work in the iteration space. It is used here to reveal invariancy of access. Re-ordering the polytope associated with the local imperative program will give the program in 5.26 which now can be finally be written as in figure 5.27. The size of the temporary remains the same but there are less accesses to *b*. The strategy for pre-fetching general affine occurrences is easily summarised:

1. Only consider pre-fetching if the u-occurrence matrix is rank deficient
2. Perform pre-fetching transformations so as to derive $\underline{\tilde{J}}$
3. Change the basis of $\underline{\tilde{J}}$
4. Perform pre-fetching transformations on the new $\underline{\tilde{J}'}$

Although this is a more complex procedure than was required for orthogonal occurrences, it allows the exploitation of invariancy of access, by pre-fetching, in the presence of general affine occurrences.

```
FOR k = 1 TO 16
  FOR i = 1 TO 4
    FOR j = 2 TO 8
      IF (j>= i+1) AND (j <= i+4)
      THEN
        tempb[j] := Get(b[k,j])
      END IF
    END FOR
  END FOR
  FOR i = 1 TO 4
    FOR j = 1 TO 4
      a[i,j] := tempb[i+j]
    END FOR
  END FOR
END FOR
```

Figure 5.26: An Imperative Program

```
FOR k = 1 TO 16
  FOR j = 2 TO 8
    tempb[j] := Get(b[k,j])
  END FOR
  FOR i = 1 TO 4
    FOR j = 1 TO 4
      a[i,j] := tempb[i+j]
    END FOR
  END FOR
END FOR
```

Figure 5.27: An Imperative Program

```
a := for i in 1,16 cross j in 1,16
    returns array of
      for k in 1,16
      returns value of sum
        b[j,k]
      end for
    end for
```

Figure 5.28: A Sisal Program

5.3 Partitioning

The first section described how to map the elements of an array to the processors and how to determine the local iteration space. A method of choosing the data partition based on alignment, was described in chapter 4. In this section, a method based on volume of access is explored. The volume of access corresponds to the size of the region of the data accessed rather than the number of times the data is accessed. It assumes that a particular alignment has been determined and it is the role of this strategy to determine which dimension the arrays are to be partitioned across so as to minimise non-local access. Alignment can be considered as the action of orienting arrays so as to maximise local access, whilst data partitioning, based on the volume of access, is the process of partitioning so as to minimise non-local access.

This partitioning scheme is placed after the mapping and pre-fetching sections as it crucially depends on pre-fetching to remove redundant non-local accesses. Essentially it searches through several possible data partitions and determines what amount of non-local data will be accessed for that partition, based on the assumption that once a data item has been accessed it remains local as long as it is required.

It is possible to precisely define the amount of data required for a particular computation by examining the array occurrences and determining how the shape of the local iteration space affects that access pattern.

The amount of data accessed by a program on processor z of an array v, when compiling for creation parallelism, is the number of points described by $\mathcal{U}_{1.v}\underline{J_z}$ with respect to the local iteration space $A_z\underline{J_z} \leq b_z$. The amount of non-local access is the total access minus that which is local and it is the purpose of data partitioning to minimise this amount.

For example consider the Sisal program in figure 5.28. If it is decided to partition by the first dimension of the aligned arrays a and b, then the volume of access to array b will be:

$$||\mathcal{U}_b\underline{J_2}|| = (16-1+1)\times(16-1+1) = 256 \tag{5.88}$$

for the second processor, $z = 2$, of two processors, $p = 2$.

If, instead, it is decided to partition by the second dimension of the aligned arrays, the volume of access will be:

$$||\mathcal{U}_b\underline{J_2}|| = (16-9+1)\times(16-1+1) = 128 \tag{5.89}$$

Thus a makes reference to 256 elements of b in the first case and 128 elements in the second case. The same is true for the local program on the first processor, $z = 1$. Clearly for this program, partitioning by the second index is preferable. Choosing the second index to partition along, has the effect of reducing the local range of the j iterator. As j appears in the u-occurrence matrix, the reduction in the range of j implies that there will be a smaller volume of access to b. The i iterator will have a smaller range if a is partitioned by the first index, but as i does not occur in the u-occurrence matrix, it will not reduce the volume of access to b.

```
a := for i in 1,n cross j in 1,n
    returns array of
      for k in 1,n
      returns value of sum
        b[k,i] * c[j,j]
      end for
    end for
```

Figure 5.29: A Sisal Program

Now $\underline{\mathcal{I}_z}$ are the local elements of an array v in processor z, so partitioning should minimise:

$$||\mathcal{U}_{1.v}\underline{J_z} - \underline{\mathcal{I}_z}|| \tag{5.90}$$

The amount of access is in general a function of the processor z, so the maximum value over the processors should be minimised. This should be carried out for all the u-occurrences in that particular computation set. Therefore partitioning should minimise:

$$\sum_{v=1}^{|v|}\sum_{r=1}^{\theta_v}\max_{z\in P}(||\mathcal{U}_{r.v}\underline{J_z} - \underline{\mathcal{I}_z}^v||) \tag{5.91}$$

where $|v|$ is the number of arrays in a computation set.

If there are N indices along which the data may be partitioned, then there are $2^N - 1$ possible data partitions available. In general the calculation of $||\mathcal{U}_{r.v}\underline{J_z}||$ is non-trivial. It is usually a function of the processor z, and depends on the shape of the iteration lattice. To determine the best partition would take $O(2^N - 1 \times p \times |\theta_v| \times m^{19})$ operations using Dyer's algorithm [DYER91]. This may be acceptable, if the m^{19} term can be reduced, as the calculation only has to be performed once for the array to which the others are aligned. However if the number of processors is large, then even performing this once may be too expensive. If this is the case then an approximating algorithm (given in appendix c) is used. This algorithm is a heuristic approximation to 5.91 and is based upon the following observations

1. If an iterator is referenced by a u-occurrence, then partitioning with respect to that iterator will reduce the volume of access to that u-occurrence. Therefore partitioning the c-occurrence by all those iterators that are common to the c-occurrence and any u-occurrence will be beneficial.
2. The volume of access is related to the rank of the u-occurrence matrices. The higher the rank, the greater the number of different data points accessed. For example in the program in figure 5.29, there are n^2 accesses to b but only n accesses to c. Therefore partitioning is only with respect to the highest rank occurrence matrices as they will dominate the non-local access cost.

This heuristic works surprisingly well and may be used when the volume of access calculation is considered too expensive.

To demonstrate the volume of access partitioning strategy, consider the program in figure 5.30 which has the following array occurrences:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_a \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}_b \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}_c \tag{5.92}$$

```
a := for i in 1,n cross j in 1,n
     returns array of
       for k in 1,n
       returns array of
         b[i,k] *c[k,j]
        end for
     end for
```

Figure 5.30: A Sisal Program

There are three ($2^2 - 1 = 3$) possible partitions which are described using the notation developed in chapter 4.

1. $\mathcal{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathcal{T}_c = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, "rows"
2. $\mathcal{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathcal{T}_b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ "columns"
3. $\mathcal{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \mathcal{T}_b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \mathcal{T}_c = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$ "blocks"

If the volume of access is calculated in each of these cases, assuming that $p_1 = p_2 = \sqrt{p}$

1. $||\mathcal{U}_b(\underline{J_z})|| = \frac{n}{p} \times n \; ||\mathcal{U}_c(\underline{J_z})|| = n \times n$
2. $||\mathcal{U}_b(\underline{J_z})|| = n \times n \; ||\mathcal{U}_c(\underline{J_z})|| = \frac{n}{p} \times n$
3. $||\mathcal{U}_b(\underline{J_z})|| = \frac{n}{\sqrt{p}} \times n \; ||\mathcal{U}_c(\underline{J_z})|| = \frac{n}{\sqrt{p}} \times n$

the local access is

1. $||\mathcal{I}_z^b|| = \frac{n}{p} \times n \; ||\mathcal{I}_z^c|| = n \times \frac{n}{p}$
2. $||\mathcal{I}_z^b|| = n \times \frac{n}{p} \; ||\mathcal{I}_z^c|| = \frac{n}{p} \times n$
3. $||\mathcal{I}_z^b|| = \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \; ||\mathcal{I}_z^c|| = \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

Thus the total non-local access for each partition is:

1. $n(n - \frac{n}{p}) = O(n^2)$
2. $n(n - \frac{n}{p}) = O(n^2)$

3. $2\frac{n^2}{\sqrt{p}}(1 - \frac{1}{\sqrt{p}}) = O(\frac{n^2}{\sqrt{p}})$

Clearly in this example partitioning by "blocks" is preferable. The same result would have been given by the heuristic.

1. Partition by all the indices common to the c-occurrence and aligned u-occurrences. Iterator i is common to arrays a and b, iterator j is common to arrays a and c, so partition by i and j.
2. Both u-occurrences are full rank and equally important

5.4 Summary

This chapter has developed a method to evenly distribute data across processors whilst maintaining alignment for arrays of differing sizes. Having determined the local data, it is possible to calculate the local iteration space even in the presence of interleaved data and computation. Once data partitioning has taken place, prefetching transformations to reduce non-local accesses have been presented which can be used in the presence of general iteration spaces, affine occurrences and interleaving. Finally a data partitioning method to reduce non-local access has been developed. If it is assumed that pre-fetching transformations are used then a simple approximation algorithm can be used to give efficient partitioning.

Chapter 6

Evaluation

This chapter applies the transformation techniques developed in chapters 3,4 and 5 to several well known problems. In general the goals of load balancing, alignment and data partitioning may conflict so a heuristic is developed, in the first section, where these goals are prioritised based upon some simple assumptions.

The programs have been selected so as to illustrate a range of program characteristics that are either common or are perceived to be difficult for compilers. These include non-rectangular iteration spaces, non-orthogonal array occurrences, array indirection, sequential computation and general while loops.

The remaining eight sections describe the application of the heuristic to the selected Sisal programs. With the aid of Mathematica [WOLF88], it is possible to determine the load imbalance and non-local access using the machine model and metrics given in chapter 2. A comparison with well known hand tuned methods is given where appropriate.

In the conclusion it is noted that the automatically generated solutions are competitive with hand written implementations.

6.1 Heuristic

Throughout this thesis the various issues involved in mapping array computation to distributed memory architectures have been treated separately but, when compiling a particular program, a strategy based on the relative importance of each issue is required. Finding program parallelism so as to fully exploit machine parallelism is the most important task. Creation parallelism is preferred over reduction parallelism as it does not require synchronisation after completion of each local reduction and generally requires less non-local access.

Load balancing, in this heuristic, is considered to be more important than reducing non-local access as non-local access may be masked by an implementation which hides latency in memory access [ROGE91].

Alignment is required before data partitioning takes place and is a complementary process. Data partitioning will be restricted in order to allow as even a distribution of data as is possible. Prefetching will take place subject to the restriction that it does not violate the scalability constraint.

Thus, the crucial decisions on how to partition the data and computation are based on the following relative ordering:

Parallelism > Load Imbalance > Non-Local Access

A different ordering based on different architectural and implementation assumptions will necessarily lead to a different heuristic from the one presented below:

1. Determine the array computation of the program which dominates the computation cost. This implies finding the largest polytope of computation within the program.
2. Within the specific polytope determine the iterators where computation may be performed in parallel
3. If any of the selected iterators are referenced by the creation occurrence matrix of that polytope, then restrict the iterators of interest to them.
4. Determine the set of perfectly load balanced iterators and transform the program. If none are perfectly balanced, then determine the most load balanced one(s) and restrict later partitioning to just one of them.
5. Align with the $\mathcal{C}$ occurrence matrix if compiling for creation parallelism, otherwise arbitrarily choose a $\mathcal{U}$ occurrence containing a reference to the reduction iterator.
6. Perform volume of access analysis to determine which of the remaining iterators to partition upon.
7. Given the iterators upon which the data and computation is to be partitioned, determine the form of parallelism and data partitioning for all the other aligned arrays.
8. Partition and map the data space and calculate the local iteration space for each polytope and processor.
9. Perform prefetching on each local program as appropriate.

These transformations within this scheme can be summarised as follows

$$P_0 \overset{\tau_0}{\longmapsto} P_1 \overset{\pi_1}{\longmapsto} P_2 \overset{\pi_2}{\longmapsto} P_3 \overset{\zeta}{\longmapsto} \begin{bmatrix} \underline{P_4} \\ \underline{P_4} \\ \vdots \\ \underline{P_4} \end{bmatrix} \overset{\pi_3}{\longmapsto} \begin{bmatrix} \underline{P_5} \\ \underline{P_5} \\ \vdots \\ \underline{P_5} \end{bmatrix} \overset{\tau_1}{\longmapsto} \begin{bmatrix} \underline{P_6} \\ \underline{P_6} \\ \vdots \\ \underline{P_6} \end{bmatrix} \tag{6.1}$$

P_0 represents any Sisal program and $[\underline{P_6}, \ldots, \underline{P_6}]$ is an array of imperative programs, one for each processor. τ_0 and τ_1 represent the translation of Sisal to a computation set representation, and the translation of the computation sets into the imperative language respectively. π_1 is any load balancing transformation, π_2 is the alignment transformation and is program wide, ζ is the mapping of data and computation to give p local programs and it may include the interleaving transformation. Finally π_3 is the prefetching transformation performed on each local program.

While one particular array computation is chosen as the basis for data partitioning decisions, its effect, via alignment, is program wide. Within this thesis the global effect of such transformations has not been studied. It is precisely for this reason that the above heuristic has been devised. An improved heuristic would consider the global impact of each transformation.

```
c:= for i in 1,n cross j in 1,n
      returns array of
        for k in 1,n
        returns value of sum
          a[i,k]*b[k,j]
        end for
      end for
```

Figure 6.1: Matrix Multiplication

6.2 Matrix Multiplication

Matrix multiplication is a very well known program and a compiler must perform at least reasonably well upon this problem to be credible. The program, figure 6.1, has just one array computation, the calculation of array c. There are three parallel iterators defining a rectangular iteration space where each of the three two-dimensional arrays makes reference to two iterators.

Applying the heuristic:

All three iterators in the program are parallel and are suitable candidates for partitioning of data and iteration space.
Any combination of the three iterators can partition the iteration space to give perfect load balance.
Creation parallelism is preferred, so the parallel iterators to consider are restricted to i and j.
Aligning for creation parallelism leaves the arrays in their present alignment.
To minimise the volume of access, partitioning should be with respect to both i and j.
Arrays c, b and a are partitioned by i and j in a folded manner. As i and j are perfectly load balanced no interleaving transformations are required. The local iteration spaces are determined.
Access to arrays a and b may be prefetched to exploit invariancy such that there are no multiple accesses to the same item of non-local data.

By applying the necessary program transformations, the generic program for a processor $z = (z_1, z_2)$ is derived as shown in figure 6.2 where ilo = $(z_1 - 1) \times \frac{n}{p_1} + 1$, ihi = $z_1 \times \frac{n}{p_1}$, jlo = $(z_2 - 1) \times \frac{n}{p_2} + 1$, jhi = $z_2 \times \frac{n}{p_2}$.

If it is assumed that the processors are arranged in the form of a square mesh, i.e $p_1 = p_2 = \sqrt{p}$, then as the non-local access is the same across all the processors, the maximum value is $2\frac{n^2}{\sqrt{p}}(1 - \frac{1}{\sqrt{p}})$. Load imbalance = 0 and thus the maximum amount of computation in any one processor is $2\frac{n^3}{p}$. In this example the local bounds on the first and second indices of arrays A, B, C are equal to the ranges of the iterators i and j respectively.

This program gives the best known performance for distributed memory machines. It is equivalent to the blocking technique used by numerical analysts [GOLU89]. In future examples, in order to aid readability, the declaration and initialisation of variables will not be shown, nor will the calculation of the bounds of the local iterators.

```
c[(c1lo..c1hi),(c2lo..c2hi)]:real
a[(a1lo..a1hi),(a2lo..a2hi)]:real
b[(b1lo..b1hi),(b2lo..b2hi)]:real
tempa[(tempalo..tempahi)]:real
tempb[(tempblo..tempbhi)]:real
FOR k = 1 TO n
  FOR i = ilo TO ihi
    tempa[i] := Get (a[i,k])
  END FOR
  FOR j = jlo TO jhi
    tempb[j] := Get (b[k,j])
  END FOR
  FOR i = ilo TO ihi
    FOR j = jlo TO jhi
      c[i,j] := c[i,j] + tempa[i]*tempb[j]
    END FOR
  END FOR
END FOR
```

Figure 6.2: An Imperative Program

```
c:= for i in 1,n cross j in 1,n
    returns array of
      for k in j,n
      returns value of sum
        a[i,k]*b[k,j]
      end for
    end for
```

Figure 6.3: Square × Triangle

6.3 Square × Triangle

This problem is very similar to matrix multiplication described in the previous section 6.2, except that the reduction iterator, *k*, has a variable lower bound so the iteration space resembles a “wedge”. This program, figure 6.3, performs roughly half the amount of work of full matrix multiplication and the compiler’s decisions should reflect this.

Applying the heuristic:

All three iterators in the program are parallel and are suitable candidates for partitioning of data and iteration space.
Only iterator *i* can partition the iteration space to give perfect load balance.
Iterator *i* occurs in the creation matrix and therefore partitioning is for creation parallelism.
Aligning for creation parallelism leaves the arrays in their present alignment.
As only iterator *i* is to be considered, there is no choice as to how the data space should be partitioned.
Arrays *c*, *b* and *a* are partitioned by *i* in a folded manner. As *i* is perfectly load balanced no interleaving transformations are required. The local iteration spaces are determined.

```
FOR k = 1 TO n
  FOR i = ilo TO ihi
    tempa[i] := Get (a[i,k])
  END FOR
  FOR j = 1 TO k
    tempb[j] := Get (b[k,j])
  END FOR
  FOR i = ilo TO ihi
    FOR j = 1 TO k
      c[i,j] := c[i,j] + tempa[i]*tempb[j]
    END FOR
  END FOR
END FOR
```

Figure 6.4: An Imperative Program

Both arrays *a* and *b* may be prefetched to exploit invariancy such that there are no redundant non-local accesses.

This gives the generic program in figure 6.4 for a processor *z*.

The maximum non-local access occurs in the first processor $z = 1$. Here the non-local access is $\frac{n^2}{2p^2}(p^2-1)+\frac{n}{2p}(p-1)$, load imbalance = 0 and therefore the maximum amount of computation in any one processor is $\frac{n^2(n+1)}{2}$.

While load balance is ideal, there is more non-local access than if a "blocking" method were employed. However in that case load balancing would suffer and thus a tradeoff between load balancing and minimising non-local access may be seen. The method suggested by [GOLU89], that of interleaved columns, has a slightly reduced maximum non-local access of $\frac{n^2}{2p^2}(p-1)^2 + \frac{n}{2p}(p-1)$ but has much worse load imbalance, $n^2\frac{(p-1)}{p}$, and thus the maximum computation to be performed by any one processor would be $\frac{n^2(n+1)}{2} + n^2\frac{(p-1)}{p}$.

6.4 Symmetric Matrix Multiplication

This problem, as shown in figure 6.5, like the previous example, has a "wedge" like iteration space but this is due to the ranges of *i* and *j*. The amount of work performed is the same as the previous program, shown in figure 6.3. In Sisal all elements of an array must be defined and as the restricted form used in this thesis restricts arrays to being rectangular, the upper triangular portion of the array must be set to zero.

Applying the heuristic :

There are two polytopes of computation in this program corresponding to the two branches of the if condition. The first branch has a larger polytope than the second and therefore transformation criteria should be based on this polytope.
All three iterators of this polytope are parallel and are suitable candidates for partitioning of data and iteration space.
Only iterator *k* can partition the iteration space to give perfect load balance but it is a reduction iterator. Either of the remaining iterators *i* or *j* can be selected. As neither of them is perfectly load balanced, only one will be

```
c:= for i in 1,n cross j in 1,n
    returns array of
      if i >= j
      then
        for k in 1,n
        returns value of sum
          a[i,k]*a[j,k]
        end for
      else
        0.0d0
      end if
    end for
```

Figure 6.5: Symmetric Matrix Multiplication

selected for partitioning. This is based on the assumption that it is better to partition on one unbalanced iterator than to partition by two or more unbalanced iterators.
Both iterators i and j occur in the creation matrix and thus partitioning is for creation parallelism.
Aligning for creation parallelism leaves the arrays in their present orientation.
As only one of the iterators i and j is to be considered the data space should be partitioned with respect to the iterator i in order to minimise the volume of access.
Arrays a and c are partitioned by i in an interleaved manner as i is not perfectly load balanced.
Both occurrences of array a may be prefetched to exploit invariancy such that there are no redundant non-local accesses.

This gives the generic program shown in figure 6.6 for a processor z.

The maximum amount of computation takes place in the last processor and is $\frac{n^3}{p}+n^2$ whilst the average is $\frac{n^3}{p}+\frac{n^2}{p}$ thus the imbalance is $n^2(1-\frac{1}{p})$. The maximum non-local access is $n^2(1-\frac{1}{p})$. Although this program is related to the rank-k update implemented in the BLAS3 kernels [DONG90], there are no well-known implementations of this program. If a blocking method were used the maximum non-local access would be reduced to that of matrix multiplication 6.2 but the maximum load imbalance would be $O(\frac{n^3}{p})$. If interleaved columns as suggested by [GOLU89] for the previous example, 6.3, is employed, the load imbalance would remain the same but the non-local access would increase. Therefore the implementation chosen is a reasonable one.

6.5 Livermore Loop 6

Unlike the previous three programs which are basic linear algebra "kernels", the following is a well known benchmark based upon segments of programs commonly used at Lawrence Livermore National Laboratories.

This program, figure 6.7, has several interesting features. It has an outer sequential iterative loop, two array creations, a non-orthogonal array access $B[j, j-oldi]$ and different sized arrays.

The update of *WOut* could be rewritten as

```
FOR k = 1 TO n
  FOR  i = ilo TO ihi
    tempa1[i] := Get (a[i,k])
  END FOR
  FOR j = 1 TO n
    tempa2[j] := Get (a[##j,k])
  END FOR
  FOR i = ilo TO ihi
    FOR j = 1,n
      IF (j<=#i)
      THEN
        c[i,j] := c[i,j] +
                  tempa1[i]*tempa2[j]
      ELSE
        c[i,j] := 0
      END IF
    END FOR
  END FOR
END FOR
```

Figure 6.6: An Imperative Program

```
WOut:=old WOut[i:W[i]]
```

However, this construct has not been used as it is not symmetric with respect to different dimensions of an array i.e. it is row biased. The cost of updating the *WOut* array is not considered, as no floating point operations are performed and, by using a sensible updating implementation based upon Cann's work [CANN89A], this assignment would not be evaluated.

Applying the heuristic:

The biggest polytope of computation is the first branch of the first conditional.
Within this polytope the only parallel iterator to partition the iteration and data space is j.
As j is the only iterator to be considered, it should be interleaved as it is not perfectly load balanced.
Iterator j occurs in the creation matrix thus partitioning should be for creation parallelism.
Arrays *oldW, WOut, WIn* are trivially aligned with W as they are all one dimensional. The first dimension of B is aligned with that of W.
After interleaved partitioning of iterator j, prefetching reduces the number of non-local accesses to $oldW[oldi]$.
Applying the necessary transformations, gives the program in figure 6.8.

The maximum load imbalance occurs in the last processor and is $\frac{n}{2}$. This gives the maximum amount of computation performed in any one processor to be $\frac{n^2}{2p} - \frac{n}{2p} + \frac{n}{2}$. The maximum non-local access is small, $n - \frac{n}{p}$, and, again, occurs in the last processor.

There are no well known hand written implementations of this program for DMAs as it is intended as a benchmark for computer systems. However this implementation will be competitive with most implementations. A possible improvement is to find a better load balancing mapping than interleaving, such as the reflection mapping described in [GERA89A]. This would reduce the load imbalance slightly. It is unlikely that the amount of non-local access can be significantly reduced due to the dependency $W[j] := oldW[oldi]$. Each $W[j]$ will have to make reference to

```
function main(n:integer; B:TwoDim;WIn:OneDim; returns OneDim)
  for initial
      i := 1;
      W := WIn;
      WOut:= Win;
  while i < n repeat
      i := old i + 1;
      W := for j in 1,n returns array of
             if (j> old i) then
              old W[j] + B[j, j - old i] * old W[old i]
             else old W[j]
             end if
           end for;
      WOut:= for j in 1,n
             returns array of
               if (j= old i)
               then W[j]
               else old WOut[j]
               end if
             end for;
  returns value  of WOut
  end for
end function
```

Figure 6.7: Livermore Loop 6

```
FORITER i = 1 TO n
  tempw := Get(oldW[##i])
  FOR j = jlo TO jhi
    IF (i <=#j)
    THEN
      W[j] := W[j] + Get(B[j,#j-i])*tempw
    ELSE
      W[j] := Get(oldW[j])
    END IF
  END FOR
  FOR j = jlo TO jhi
    IF (#j=i)
    THEN
      WOut[j] := Get(W[j])
    ELSE
      WOut[j] := Get(oldWOut[j])
    END IF
  END FOR
END FOR
```

Figure 6.8: An Imperative Program

```
function main(n:integer; xIn :OneDim; bIn :OneDim; aIn:TwoDim;
             returns array[integer])
 for initial
   i := 1;
   x := xIn;
   b := bIn;
   a := aIn;
 repeat
   i := old i + 1;
   x := for j in 1,n
        returns array of
          if j=i
          then (b[i]- for k in 1 ,i-1
                      returns value of sum
                        (a[i,k]* old x[k])
                      end for) /a[i,i]
                else
                      old x[j]
                end if
        end for;
 until (i >n)
 returns value of x
 end for
end function
```

Figure 6.9: Triangular Back Substitution

the same element, *oldW*[*oldi*], and as *W* must be distributed across the processors, it is not possible to have *W*[*j*] and *oldW*[*oldi*] in the same processor for all values of *j*.

6.6 Triangular Back Substitution

This is a well known linear algebra program. It has, in the past, been a bottle-neck to parallel implementations of linear system solvers [HEAT88]. This is due to the relatively high communication overhead.

This program, figure 6.9, again has an outer iterative loop with the only parallelism available being with respect to the *k* iterator. Although *j* belongs to a **for** loop, computation only takes place when $i = j$. As *i* is a sequential iterator, this implies that any work associated with *j* will also be sequential. This is discovered when examining the polytope in the load balancing analysis phase. Unfortunately iterator *k* forms a triangular iteration space and is a reduction iterator so this is a difficult problem to implement efficiently, and should be a good test of a compiler.

Applying the heuristic:

The polytope of interest is the first branch of the if condition.
Only the *k* iterator is parallel, partitioning with respect to *j* as described above,implies purely sequential computation.
As *k* is the only candidate iterator for partitioning, it should be interleaved as it is not perfectly load balanced.

```
FORITER i = 1 TO n
  tempa := Get(a[i,##i])
  tempb := Get(b[i])
  FOR j = 1 TO n
    IF (#j=i)
    THEN
      FOR  k = klo TO khi
        IF (#k <= i-1)
        THEN
          psum[z] := psum[z] +a[i,k]*oldx[k]
        ELSE
          psum[z]:= 0
        END IF
      END FOR
      IF  (#j >= xlo) AND (#j <= xhi)
      THEN
        FOR y = 1 TO p
          sum := sum + Get(psum[y])
        END FOR
        x[j] := (tempb -sum)/tempa
      END IF
      Sync
    ELSE
      x[j] := Get(oldx[j])
    END IF
    Sync
  END FOR
END FOR
```

Figure 6.10: An Imperative Program

The iterator *k* is not in the creation matrix so partitioning should be for reduction parallelism.
Array *a* is arbitrarily selected to align with for reduction parallelism. Arrays *x*, *oldx* and *b* are aligned with *a* on the second dimension.
The data is partitioned in an interleaved manner on a one dimensional processor grid.

The maximum time taken by one processor executing the translated program, figure 6.10, occurs in the last processor and is $\frac{n^2}{p} + (np + \frac{5n}{2} - \frac{2n}{p})$. The parenthetical terms are the deviations from linear speedup. The third and fourth terms are due to load imbalance, whilst the dominant overhead term, np, is due to the implementation of summing the partial sums involved in the reduction parallelism. As the number of processors grows large, relative to the size of the problem, this term will eventually dominate. This reduction term may be reduced to $n \log p$ if a binary spanning tree was embedded in a higher dimensional processor grid, with each node processor adding its result to the previous accumulations before passing it on. Such implementation details are beyond the scope of this thesis.

The maximum amount of non-local access occurs in the first processor and is $np + n$. Again the np term is due to the implementation of the accumulation of partial sums.

The program derived is similar to the fan-in algorithm given in [HEAT88]. In this paper improvements to this

```
 define main
 global Sqrt(x: double_real returns double_real)

function main(n :integer;A:TwoDim;L:TwoDim  returns TwoDim)
 for initial
   col := 1;
   diag := double_real(1.0);
 while (col <= n)
 repeat
   col := old col + 1;
   diag := Sqrt(A[ old col,old col]- for k in 1,old col-1
                returns value of sum
                  (old L[old col,k]*old L[old col,k])
                end for);
   L := for i in 1,n cross j in 1,n
        returns array of
          if (j = old col) & (i>j)
          then
              (A[i,j]- for k in 1 ,j-1
                       returns value of sum
                         (old L[i,k]* old L[j,k])
                       end for) / diag
          else if (j = old col) & (i=j)
               then diag
               else old L[i,j]
               end if
          end if
        end for;
 returns value of L
 end for
end function
```

Figure 6.11: Cholesky Factorisation

scheme are based upon improving performance by carefully overlapping communication and computation and vectorisation of messages. As the amount of non-local access is of the same order of magnitude as the computation such modifications become increasingly important. However, although these techniques are beyond the scope of this thesis, the form presented here can be transformed into their efficient cyclic form.

6.7 Cholesky Factorisation

Cholesky factorisation is a well known linear algebra program which factorises a symmetric positive definite matrix into two triangular matrices i.e. $A \mapsto LL^T$. The program, figure 6.11, is interesting in that it has four iterators, the outermost of which is a sequential loop. There are two major computations, one of which involves the calculation of a scalar. Both reduction and creation parallelism are present.

Applying the heuristic:

The biggest polytope is the first branch of the if condition when defining L.
Only partitioning with respect to i or k will give parallel execution.
Although iterator k is better load balanced than i, it is a reduction iterator and therefore i is chosen. As i is not perfectly load balanced, it should be interleaved.
Iterator i belongs to the creation occurrence matrix and therefore partitioning is for creation parallelism.
Aligning with respect to L does not change the relative alignment of the arrays. Although the allocation of scalars has been previously addressed, the method employed by the AL compiler [TSEN89] amongst others, is used. As the scalar *diag* is a function of the outer **foriter** loop, its allocation is a function of this iterator.
As there is only one iterator with which to partition, no useful analysis is available from looking at the volume of access.
As i is not perfectly load balanced, partitioning is by interleaved rows.
There is much opportunity to prefetch the data.

The maximum amount of work $\frac{n(n+p)(2n+p+9)}{12p}$ takes place in the final processor when executing the translated program shown in figure 6.12. The average work is $\frac{n(n+1)(n+5)}{6p}$ giving an overhead figure of $\frac{n(p-1)(3n+p+10)}{12p}$. The maximum non-local access is again in the last processor $\frac{n^2}{2}+\frac{n}{2}-\frac{n(n+p)}{2p}$.

The maximum amount of work for an alternative column interleaved implementation as suggested by [GERA89A] is $\frac{n}{12p}(12+9n+2n^2+9p+3np-p^2)$ and therefore has a reduced overhead of $\frac{np}{6}$ with the same maximum non-local access. Although both methods share the same highest order term $\frac{n^2}{4p}$, as the number of processors, p, tends towards n, then the $\frac{np}{6}$ term will begin to dominate. Thus while the implementation given by the transformation scheme is reasonable, there is a better implementation available as far as load imbalance is concerned. If the avoidance of reduction parallelism in the heuristic were dropped, then an interleaved column implementation would have resulted. However, the amount of non-local access is an order of magnitude greater in the column scheme. This would be difficult to mask in a latency tolerant implementation as the amount of communication is of the same order of magnitude $O(\frac{n^3}{p})$ as computation. Therefore, overall, the implementation given by the transformation scheme is acceptable.

6.8 Livermore Loop 14

This is another program, figure 6.13, used extensively in benchmarking computer systems. The most interesting characteristic is that it contains indirection whereby the data dependencies are run-time dependent. There are also two distinct phases, one highly parallel the second almost completely sequential, and is therefore an interesting task for a compiler.

Applying the heuristic:

The largest polytope is the first **for** loop.
There is only one iterator, i, to partition the iteration and data space.
The only candidate iterator, i, is perfectly load balanced.
All the array computations in this polytope are of equal importance but only the occurrence of array j can be expressed in matrix form and is therefore chosen to align with respect to. The iterator i occurs in the array occurrence of j and thus alignment is for creation parallelism. It is impossible to determine the relative alignment of the remaining arrays as they have indirection occurrences. By default they are aligned on the first dimension of j without any alignment transformations.
Iterator i can only be partitioned in one dimension and does not require interleaving as it is perfectly load balanced.

```
FORITER col = 1 TO n
  IF (col>=ilo) AND (col<=ihi)
  THEN
    FOR k = 1 TO col- 1
      temps1 := temps1 +
             Get(oldL[##col,k])*Get(oldL[##col,k])
    END FOR
    diag  := sqrt(Get(A[##col,col])-temps1)
  END IF
  FOR j =col TO col
    FOR k = 1 TO j-1
      templ1[j,k] := Get(L[##j,k])
    END FOR
  END FOR
  FOR i = ilo TO ihi
    FOR k = 1 TO col-1
      templ12[i,k] := Get(L[##i,k])
    END FOR
  END FOR
  FOR i = ilo TO ihi
    FOR j = 1 TO n
      IF (j=col) AND (#i>j)
      THEN
        FOR  k = 1 TO j-1
          temps2 := templ1[j,k] *templ2[i,k] +temps2
        END FOR
        L[i,j] := (Get(A[i,j]-temps2)/Get(diag)
      ELSE IF (j=col)  AND (#i = j)
           THEN  L[i,j] := Get(diag)
           ELSE L[i,j] := Get(oldL[i,j])
           END IF
      END IF
    END FOR
  END FOR
END FOR
```

Figure 6.12: An Imperative Program

```
function main(n:integer; FLX:double; DEXIn,EXIn,
            GRD,RHIn : OneDim;
            returns OneDim,OneDim,IntOneDim,IntOneDim,
                   OneDim,OneDim,OneDim,OneDim,OneDim)
  let DEX1,EX1,IR1,IX1,RX1,VX1,XI1,XX1 :=
       for i in 1,n
          j := Trunc(GRD[i]);
          EX := EXIn[j];
          DEX := DEXIn[j];
          XI := Double_Real(j);
          VX := EX - DEX * XI;
          k  := Trunc(VX + FLX);
          IR := MOD2N(k,512) + 1;
          RX := VX + FLX - Double_Real(k);
          XX := VX + FLX - Double_Real(k)
               + Double_Real(IR)
       returns array of DEX array of EX
              array of IR  array of j
              array of RX  array of VX
              array of XI  array of XX
       end for
  in  DEX1,EX1,IR1,IX1,RX1,VX1,XI1,XX1,
     for initial
        i := 0; RH := RHIn
     while i < n repeat
        i := old i + 1;
        RH := for j in 1,n
              returns array of
                if (i=j) then
                old RH[IR1[i]] - RX1[i] + 1.0d0
                elseif (i+1=j)
                then old RH[IR1[i] + 1] + RX1[i]
                else old RH[j]
                end if
              end for
     returns value of RH
     end for
  end let
end function  %Loop14
```

Figure 6.13: Livermore Loop 14

Prefetching of data is impossible due to the presence of indirection.

As there is indirection in the array occurrences, it is impossible to determine the amount of non-local access of any processor since it will be data dependent. The first part of the program, figure 6.14, is perfectly load balanced but the **for initial** loop has to be evaluated serially. This gives an overall parallel time figure of $\frac{15n}{p}+4(n-1)-2(p-1)$

The serial term, $4n-4$, could be reduced if neighbouring elements were placed on separate processors such that the two assignments $i = j$ and $i+1 = j$ may be evaluated in parallel. Interleaving the j iterator would give this property and would reduce the parallel time to $\frac{15n}{p}+2(n-1)$. The effect this would have upon non-local access will be unknown but, in general, interleaving increases its value. As p tends to n the folded value of non-local access tends towards the interleaved value. Unfortunately, there are no well known hand implementations of this program with which it may be compared.

6.9 Jacobi Iteration

Solving partial differential equations using a 5-point grid is a well-known application. Various algorithms are known including SOR, red-black and Jacobi iteration, the last of which is presented here. The most interesting characteristic of this program, figure 6.15, is the outer while loop where the exact number of iterations is run-time dependent.

Applying the heuristic:

The polytopes creating array A and eps are equal. The polytope creating A is chosen arbitrarily.
There are two parallel iterators, i and j, with which to partition the data and iteration space.
Both iterators are perfectly load balanced.
Both iterators occur in the creation matrix of A and, thus, partitioning is for creation parallelism.
All arrays remain aligned as they are. The scalar may be arbitrarily allocated as there is no iteration space enclosing it.
Partitioning to minimise the volume of access suggests that the program should be partitioned by both iterators. Hence array A is partitioned by rows and columns.
There is no opportunity for prefetching

The program shown in figure 6.15 is perfectly load balanced and has a parallel time of $E\times(\frac{7n^2}{p}+p)$ where E is the number of times the program must iterate before it converges. This is unknown at compile time. The only non-scalable component of this implementation is the p term due, once again, to the implementation of accumulation of partial sums. The maximum non-local access of any one processor is $E\times\frac{4n}{\sqrt{p}}+p+1$

The only hand-written implementations with which to improve on this, use an approach where the data space is “tiled” with hexagonals rather than squares i.e. non-orthogonal data partitioning. In [REED87] it is shown that the non-local access may be reduced to $E\times\frac{3n}{\sqrt{p}}+p+3$. However due to the boundary conditions there will be a slight increase in load imbalance and thus it may be argued that the tiling by rectangles implemented here, is preferable.

```
FOR i = ilo TOihi
   j[i] := Trunc(GRD[i])
   IX1[i]:=j[i]
   EX[i] := EXIn[j[i]]
   EX1[i] := EX[i]
   DEX[i] := DEXIn[j[i]]
   DEX1[i] := DEX[i]
   XI[i] := Double_Real(j[i])
   XI1[i]:= XI[i]
   VX[i] := EX[i] - DEX[i] * XI[i]
   VX1[i]:=VX[i]
   k[i]  := Trunc(VX[i] + FLX[i])
   IR[i] := MOD2N(k[i],512) + 1
   IR1[i] := IR[i]
   RX[i] := VX[i]+FLX[i]-Double_Real(k[i])
   RX1[i]:= RX[i]
   XX[i] := VX[i]+FLX[i]-Double_Real(k[i])
               +Double_Real(IR[i])
   XX1[i] := XX[i]
END FOR
FORITER i = 0 TO  n-1
  RH[i] := Get(RHIn[i])
  tempRH1:= Get(oldRH[IR1[i]])
  tempRH2:= Get(oldRH[IR1[i]+1])
  tempRX11:= Get(RX1[i])
  tempRX12:= Get(RX1[i])
  FOR j =  jlo TO jhi
    IF (i=j)
    THEN
      RH[j]:= tempRH1 - tempRX11 + 1.0
    ELSE IF  (i+1=j)
         THEN RH[j] := tempRH2 + tempRX11
         ELSE  RH[j]:=Get(old RH[j])
         END IF
    END IF
  END FOR
END FOR
```

Figure 6.14: An Imperative Program

```
% Jacobi iteration
%
function main(n:integer;tol: double_real;init_eps;
               AIn:TwoDim; returns TwoDim)
  for initial
      A := AIn;
      eps := init_eps
  while eps > tol repeat
      A := for i in 1,n cross  j in 1,n
           returns array of
             if (i>= 2)&(j>=2)&(i<=n-1)&(j<=n-1)
             then  old A[i-1,j] + old A[i+1,j]
              + old A [i,j-1] + old A[i,j+1]
              + old A[i,j]
             else old A[i,j]
             end if
           end for;
       eps:= for i in 1,n cross j in 1,n
             returns value of sum
              abs(A[i,j]- old A[i,j])
             end for;
  returns value  of A
  end for
end function
```

Figure 6.15: Jacobi Iteration

```
IF (z = 1)
THEN
   eps := 10000.0d0
END IF
WHILE ( Get(eps) > tol)
  FOR i = ilo TO ihi
    FOR j = jlo TO jhi
      IF (i>= 2)  AND (j>=2) AND (i<=n-1) AND (j<=n-1)
      THEN
        A[i,j] := Get(oldA[i-1,j]) + Get(oldA[i+1,j])
                + Get(oldA[i,j-1])
                + Get(oldA[i,j+1]) + Get(oldA[i,j])
      ELSE
        A[i,j] :=Get(oldA[i,j])
      END IF
    END FOR
  END FOR
  FOR i = ilo TO ihi
    FOR j = jlo TO jhi
      psum[z] := psum[z] +abs(A[i,j]- oldA[i,j])
    END FOR
  END FOR
  IF (z=1)
  THEN
    FOR y = 1 TO p
      sum := sum + Get(psum[y])
    END FOR
    eps :=sum
  END IF
  Sync
END WHILE
```

Figure 6.16: An Imperative Program

6.10 Summary

The eight examples have demonstrated that the transformation scheme based upon the heuristic developed in the first section produces an efficient implementation for DMAs. Overall the test programs implemented in this chapter have justified the use of compiler directed, program transformations to implement array computation on distributed memory architectures.

Chapter 7

Conclusion

This thesis has presented a methodical approach to compiling array computation to distributed memory architecture using program transformations. A summary of the thesis and its contributions is made in the first part of this chapter. Following this summary, a critical review of the overall approach is given in section 2. Finally, in the last section, some recommendations for further work are described.

7.1 Summary

The introductory chapter described the need for research into compiling for DMAs. Such architectures have the promise of delivering great performance but are restricted by the primitive state of compiler technology. Until very recently message passing distributed memory machines and shared memory ones were seen as being very different architectures requiring different compilation strategies. This has certainly been reinforced by the languages used to program them. By considering distributed memory as forming one address space, many of the compilation techniques developed for shared memory machines are available. Most present day compilers for DMAs require significant help from the programmer if they are to produce efficient implementations. The two major overheads in compiling for DMAs were identified as being load imbalance and communication or non-local access.

Chapter 2 described the overall compilation process from Sisal to an imperative language via an intermediate computation set representation. Essentially the process can be described as a sequence of transformations. Describing Sisal in a computation set representation allows the investigation of program transformations to reduce load imbalance and non-local access.

Chapter 3 makes the first major contribution of this thesis, in that it describes load balance as an invariancy condition of the iteration space. Further transformations are described which can convert a sub-class of programs into a load balanced form. As a consequence of these transformations, general loop interchange for nested loop within a polytope representation is developed.

By the careful alignment of arrays it is possible to reduce the amount of non-local access after data partitioning. The main contribution of Chapter 4 is that it describes alignment in terms of hyperplanes. With this insight, generalised alignment transformations are derived which extend previous results in this area.

Both the load balancing and alignment transformations of chapters 3 and 4 take place before the data and computation is mapped to the processor space. The first contribution of chapter 5 is that it provides a systematic method of mapping data and computation to processors even in the presence of interleaving. The second contribution is the presentation of program transformations that allow general pre-fetching of non-local data on a distributed memory machine. This relies on the ability to perform loop interchange and strip-mining in the presence of general loops and interleaving.

Chapters 3,4, and 5 all describe methods to partition the computation and data space depending on whether load imbalance or non-local access is to be minimised. In chapter 6, a simple heuristic was used to determine the relative importance of each of these transformations, where it was shown that the implementations derived by program transformations were comparable to hand coded techniques.

The overall contribution of this thesis is that a compilation strategy has been devised such that program transformations can be used in an ordered manner for specific reasons. In [SARK89] Sarkar states that placing the partitioning stage within the context of a general optimising compiler is a “difficult task”. In this thesis, it has been shown that it is possible for a restricted class of Sisal on DMAs.

7.2 Critique

One of the major criticisms of this work is that it does not address the trade-off in overhead cost between load-imbalance and non-local access. Instead, each problem is solved in isolation, leaving the reconciliation to a heuristic. This can be justified, somewhat, in that the relative cost will depend on the target machine. However given the relative cost of communication and computation, the compiler should be able to detect the relative importance of each. Recent work by the author suggests that it will be possible to determine whether load balancing or non-local access is the dominant cost for a particular part of a program.

DMAs have been considered to be a two-level memory hierarchy, local and non-local. This observation is largely based on the fact that non-local access has a large start-up time. One consequence of this overhead is that multiple non-local accesses should be grouped so that only one start-up cost is incurred. The reason this issue was not addressed is that it forms a large proportion of Rogers’ thesis [ROGE91], where the grouping of accesses was called message vectorisation. It is relatively simple to incorporate message vectorisation into the pre-fetching transformations described in chapter 5.

While load imbalance and communication overhead have been considered, the cost of synchronisation and hiding memory latency have been ignored. Again [ROGE91] has addressed this area. It would be useful to incorporate latency hiding transformations into the pre-fetching scheme.

Most of the analysis has been for the mapping of parallel array computation associated with Sisal’s **for** loop construct. This conservative approach is inappropriate if there are no **for** loops available. Do-across parallelism [LI90], which allows part of a loop iteration to be executed in parallel, may be extracted from **foriter** loops. This approach would require an efficient synchronisation implementation , so the present barrier synchronisation would have to be replaced.

On a more technical level, the alignment propagation algorithm in chapter 4 is too restrictive. Only certain alignment transformations can be propagated through the program. Recent work by the author suggests that this restriction can soon be removed.

One criticism of any program transformation approach to efficient implementation, is that it may work on small examples but will fail when applied to large, “real” programs. The major impediment, at present, is determining the relative importance of each section of a program. In chapter 6, the biggest polytope was considered the portion of the program to focus attention upon. However in larger programs, there may be many sections of equal importance, where data alignment may be a trade-off between the various sections. In general the approach of this thesis has been to focus upon local analysis. Although this is less than ideal, the amount of work needed to understand the behaviour of small problems, justifies this restricted approach before program analysis “in the large” is considered..

7.3 Further Work

The obvious next step is to implement the transformation scheme on a DMA so as to empirically test it. Inevitably other machine characteristics not considered in this thesis will become significant, for instance, in those machines that have a page-based cache system, the ‘interference’ by the hardware will have to be considered.

Some distributed memory architectures such as the EDS[HAYW90] and KSR have a global address space. Those machines lacking such an addressing scheme, will have to have it simulated in the code produced. Either a run-time software layer will have to be introduced or message passing primitives will have to be inserted at compile time. The latter method has been successfully implemented by [CALL88], [TSEN89] and [ROGE91].

Once an implementation is available, then a greater sub-set of Sisal can be considered. Adding the dynamic features of Sisal such as recursion and variable sized arrays, will have a significant impact upon the implementation. Dynamic array allocation will have to be performed such that data is evenly distributed across the machine. An interleaved data allocation method would ensure even distribution, but will have an adverse affect upon locality and thus increase the communication overhead. Recursion could be handled in a strictly von Neumann manner such as is employed by the OSc [CANN89A] Sisal compiler for the Sequent Balance. However there is potentially a large amount of parallelism available in the independent evaluation of recursive functions. This leads on to a more general point of going beyond data parallelism . Certainly only a small proportion of the available program parallelism is exploited at present. As mentioned earlier in this section, do-across parallelism may be explored but it is not obvious how parallelism available in an expression or function evaluation may be integrated into the SPMD framework. One approach might be to relax the static, one process to one processor, implementation and allow the spawning of parallel processes that may migrate around the system. The problem then becomes how to keep data and process together as the compiler no longer has control over data and process allocation. The Flagship project [WATS88] has investigated such a dynamic, medium-grain, graph-reduction approach to the execution of functional programs on a distributed memory machine. It would be interesting to see if any benefit could be gained by the integration of these two approaches.

Although Sisal has provided a useful framework to investigate array computation, it is not a widely used language. If the ideas in this thesis are to have wider significance then they have to be applied to traditional imperative languages, in particular FORTRAN. One method would be to pre-process the imperative language so that it is in a functional form. A more fruitful approach might be to integrate data dependency analysis into the compilation process so as to perform parallelisation transformations before applying the transformation scheme described in this thesis. It is interesting to note that there has been a movement towards single-assignment semantics in the vector notation employed in FORTRAN 90. If this trend continues, then the results of this thesis will become more immediately applicable.

On a more fundamental level, a greater investigation of integer linear algebra would be very useful. In this thesis the use of polytopes has been central. A greater understanding of their properties will bear fruit in compiler analysis and program transformation. In particular an efficient method to determine the number of lattice points within a polytope would be very useful.

In this thesis the array size and the number of processors are assumed to be known at compile time. This restriction can be relaxed, but may require some symbolic analysis as described by [HAGH90].

Although many programs have an affine structure, more work is required for those that do not possess this characteristic. Properties such as invariancy still have meaning in non-affine space, in [KOEL90], for example, data dependencies that are run-time dependent but loop invariant are exploited by storing the dependencies after the first loop iteration. Work within this area will be useful for compiling sparse matrix problems.

Finally, one very interesting area of research is the impact of locality of reference on architectures with different computational models such as dataflow. If increasingly large machines are to be created, then the memory will, at some point, have to become distributed. It would be interesting to investigate how a compiler could influence data distribution and program execution in such an architecture.

Appendix A

Language Definitions

A.1 Restricted Sisal

```
Program        =  ‘‘define’’  idr {‘‘,’’idr}[type-def-part]
                  {‘‘global’’ function-header} function-def
                  {function-def}

function-def    =  ‘‘function’’ function-header
                  [type-def-part]
                  expression ‘‘end function’’

type-def-part   =  type-def ‘‘;’’ {type-def ‘‘;’’}

type-def       =  ‘‘type’’ idr = type-spec

function-header =  idr ‘‘(’’ {decl ‘‘;’’} ‘‘returns’’
                  type-list ‘‘)’’

type-list      =  type-spec {‘‘,’’ type-spec}

type-spec      =  ‘‘boolean’’| ‘‘character’’ | ‘‘double_real’’
                  | ‘‘real’’|
                  ‘‘integer’’ |  ‘‘array’’ ‘‘[’’ idr ‘‘]’’

expression     =  s-expression {‘‘,’’ s-expression}

s-expression    =  primary { bin-op primary}

un-op          = ‘‘+’’| ‘‘-’’ | ‘‘~’’

bin-op         = ‘‘<’’ | ‘‘<=’’ | ‘‘>’’ | ‘‘>=’’ | ‘‘=’’
                | ‘‘~=’’|
                ‘‘+’’ | ‘‘-’’  | ‘‘|’’ | ‘‘*’’  | ‘‘/’’
```

```
               |  ‘‘&’’

primary        = constant | idr | ‘‘(’’expression‘‘)’’ |
               let-in-exp|
               iteration-exp  | ‘‘old’’ idr | un-op primary

array-ref      = primary ‘‘[’’expression‘‘]’’

let-in-exp     = ‘‘let’’ def ‘‘;’’{def ‘‘;’’} ‘‘in’’ expression
                ‘‘end let’’

def            = idr {‘‘,’’ idr} ‘‘:=’’ expression

conditional-exp = ‘‘if’’ expression ‘‘then’’ expression
               {‘‘elseif’’ expression ‘‘then’’ expression}
               ‘‘else’’ expression ‘‘end’’ ‘‘if’’
iteration-exp  = ‘‘for’’ ‘‘initial’’ def iter-term ‘‘returns’’
                forinit-exp ‘‘end’’ ‘‘for’’
               |
               ‘‘for’’ in-exp-list  def ‘‘returns’’
                forall-exp ‘‘end’’ ‘‘for’’

iter-term      = iterator term-test | term-test iterator

iterator       = ‘‘repeat’’ iter-body

term-test      = ‘‘while’’ expression | ‘‘until’’ expression

iter-body      =  def ‘‘;’’{def ‘‘;’’}

in-exp-list    = in-exp {(‘‘dot’’ |  ‘‘cross’’) in-exp}


in-exp         = idr ‘‘in’’ expression

forall-exp     = return-exp {return-exp}

return-exp     = ‘‘value’’ ‘‘of’’ [reduction-op] expression |
               ‘‘array’’ ‘‘of’’ expression

forinit-exp    = ‘‘value’’ ‘‘of’’ expression { ‘‘value’’
               ‘‘of’’ expression}

reduction-op   = ‘‘sum’’ | ‘‘product’’ | ‘‘least’’ | ‘‘greatest’’

constant       = ‘‘false’’ | ‘‘true’’ | integer-num | real-num
               | char-const | char-string-const
```

A.2 Imperative

```
Program   = {const-def} {var-def} stmt {stmt}

const-def = idr [‘‘:’’ type] ‘‘=’’ expr

var-def   = idr [‘‘[’’ ‘‘(’’idr ‘‘..’’ idr{‘‘,’’idr‘‘..’’idr}
            ‘‘)’’‘‘]’’
            {‘‘,’’‘‘[’’ ‘‘(’’idr ‘‘..’’ idr{‘‘,’’idr‘‘..’’idr}
            ‘‘)’’‘‘]’’}‘‘]’’]
            ‘‘:’’ type

stmt      = assign-st| for-st | while-st | foriter |if-st |
            ‘‘Sync’’

assign-st = var ‘‘:=’’ expr

for-st    = ‘‘FOR’’ idr ‘‘=’’ expr ‘‘TO’’ expr stmt {,stmt}
            ‘‘END FOR’’

foriter   = ‘‘FORITER’’ idr ‘‘=’’ expr ‘‘TO’’ expr stmt
             {,stmt} ‘‘END FOR’’

while-st  = ‘‘WHILE’’   expr  stmt {,stmt} ‘‘END WHILE ’’

if-st     = ‘‘IF’’ expr ‘‘THEN’’ stmt{stmt} [‘‘ELSE’’
            stmt{stmt}] ‘‘END IF’’


expr      = constant | var | unop expr | [‘‘Get’’]
            ‘‘(’’expr‘‘)’’| expr op expr

unop      = ‘‘-’’|’’+’’|‘‘#’’|‘‘##’’|‘‘NOT’’

var       = idr [‘‘[’’expr {‘‘,’’expr} ‘‘]’’]

op        = ‘‘-’’|‘‘+’’|‘‘/’’| ‘‘*’’ | ‘‘>’’ | ‘‘>=’’
            | ‘‘<=’’ | ‘‘<’’ | ‘‘=’’ | ‘‘<>’’
            | ‘‘AND’’ | ‘‘OR’’

type      = integer | real | double |char |bool
```

Appendix B

Translation Algorithms

B.1 Translation of Polytopes into Loops

```
1. Loop(i) =
2. If (i <=m) then
3. If (J.desc[i] <> **while** AND A[i] <> $\perp$ )
4. then Print (J.desc[i] J[i] **=**  -A[i] J - b[i] **TO** A[2i]J+b[2i])
5. else Print(**WHILE**, Getcond (i,$F$)) endif
6. doif(i,1)
7. Print(**END FOR**)
8. end if
```

```
1. doif(i,j) =
2. conditional := FALSE
3. if ( j +2m <= $\ell$)
4. then if (A[j+2m, i+1..m] =0) AND (A[j+2m, i] <> 0)
5. Print (**IF** A[j+2m]J **<=** b[j+2m] **THEN**)
6. conditional := TRUE
7. end if
8. doif (i,j+1)
9. end if
10. if (i < m) then Loop (i+1) end if
11. if (i= m) then Printbody($\mathcal{S}$) end if
```

```
12. if (conditional = TRUE) then Print (END IF)
13. end
```

Loop(1)

B.2 Loop Merging

```
 1. Merge(down,across return next) =
 2. i := down, j := across, k := down
 3. PrintLoop (Q[i].J[j])
 4. If (NumLoop[i]=j) then Print(Q[i].S) return k
 5. else i := Merge(i,j+1)
 6. while (NumLoop[i+1] >= j) AND Same(L[i+1,j],L[i,j]) do
 7. i := i+1, k := k+1, i := Merge(i,j+1), end while
 8. end If
 9. PrintEndLoop
10. return k
```

Merge(1,1)

B.3 Translating for Reduction Parallelism

```
1. Print (IF( $\underline{J}$ >= $\underline{\lambda_v}$ ) AND ( $\underline{\upsilon_v}$ >= $\underline{J}$))
2. Print ( THEN)
3. Print ( r := identity(reduction-op))
4. Print ( FOR x := 1 , p)
5. Print ( r := reduction-op( r,:= Get ( pr [x])))
6. Print ( END FOR)
7. Print (v [ $\mathcal{C}_v J_v$ ] := r)
8. Print (END IF)
9. Print (Sync)
```

where r is a local scalar, p is a distributed array, x is an iterator and p is the number of processors. The reduction operator, reduction-op, and its corresponding identity value is given by the following table:

Reduction Operator	Identity
sum	0
product	1
least	Maxint
greatest	Minint

Appendix C

Algorithms

C.1 Loop Interchange

1. Let the two iterators to be interchanged be j_r and j_{r-1}
2. Let $B = \begin{bmatrix} E_{i-1,i} & 0 \\ 0 & E_{i-1,i} \end{bmatrix} A$
3. Let $J' = E_{r-1,r}J$
4. Let $c = \begin{bmatrix} E_{i-1,i} & 0 \\ 0 & E_{i-1,i} \end{bmatrix} b$
5. If $B_{r-1,r} = 0$ Goto 12
6. $d = \begin{cases} B_{r-1} + B_{r-1,r}B_r & B_{r-1,r} > 0 \\ B_{r-1} - B_{r-1,r}B_{m+r} & B_{r-1,r} < 0 \end{cases}$
7. $f = \begin{cases} c_{r-1} + B_{r-1,r}c_r & B_{r-1,r} > 0 \\ c_{r-1} - B_{r-1,r}c_{m+r} & B_{r-1,r} < 0 \end{cases}$
8. $C_{x,y} = \begin{cases} B_{x,y} & x \neq r-1 \\ d_y & x = r-1 \\ B_{r-1,y} & x = \ell+1 \end{cases} \quad \forall x \in 1,\ldots,\ell+1, y \in 1,\ldots,m$
9. $g_y = \begin{cases} c_y & y \neq r-1 \\ f & y = r-1 \\ c_{r-1} & y = \ell+1 \end{cases}$
10. $\ell := \ell + 1$
11. $d = \begin{cases} B_{m+r-1} + B_{m+r-1,r}B_{m+r} & B_{m+r-1,r} > 0 \\ B_{m+r-1} - B_{m+r-1,r}B_r & B_{m+r-1,r} < 0 \end{cases}$
12. If $B_{r-1,r} = 0 \wedge B_{m+r-1,m+r} = 0$ Terminate.
13. $f = \begin{cases} c_{m+r-1} + B_{m+r-1,r}c_{m+r} & B_{m+r-1,r} > 0 \\ c_{m+r-1} - B_{m+r-1,r}c_r & B_{m+r-1,r} < 0 \end{cases}$
14. $C_{x,y} = \begin{cases} B_{x,y} & x \neq m+r-1 \\ d_y & x = m+r-1 \\ B_{r-1,y} & x = \ell+1 \end{cases} \quad \forall x \in 1,\ldots,\ell+1, y \in 1,\ldots,m$
15. $g_y = \begin{cases} c_y & y \neq m+r-1 \\ f & y = m+r-1 \\ c_{r-1} & y = \ell+1 \end{cases} \quad \forall y \in 1,\ldots,m$

```
16. For $x \in 1,\dots,\ell, y \in 1,\dots,x, z \in 1,\dots,\ell$
17. Solve $r_1, r_2 \begin{bmatrix} r_1 & r_2 \end{bmatrix} \begin{bmatrix} C_x \\ C_y \end{bmatrix} = \begin{bmatrix} C_z \end{bmatrix}$.
    and
    $\begin{bmatrix} r_1 & r_2 \end{bmatrix} \begin{bmatrix} f_x \\ f_y \end{bmatrix} = \begin{bmatrix} f_z \end{bmatrix}$.
18. If a solution $\wedge(Nz(C_x) = Nz(C_z) \wedge Right(C_x) = Right(C_z)) \wedge x > 2m \vee (Nz(C_y) = Nz(C_z) \wedge Right(C_y) = Right(C_z) \wedge y > 2m$
    then $\Xi_{x,y} = z$ Else $\Xi_{x,y} = \perp$ end for
19. Form $V\gamma = \delta, W = V - \delta$
20. For $y \in 1,\dots,\ell, x \in 1,\dots,r$
21. If $W_{x,y} = -1$ Then For $z \in 1 \dots r$
22. If $W_{z,y} = 1$ Then $W'_{z,y} = W_{x,y} + Wz,y$, Remove $W_x, C_y, f_y$ end end if end if end for
23. For $x \in 1,\dots,\ell$
24. Find only $-1$ element in $W_x$ at $W_{x,a}$
25. Find first 1 in $W_x$ at $W_{x,b}$
26. Replace $C_a$ by $C_b$
27. end for end for
```

Complexity: $O(\ell^3)$

C.2 Alignment

C.2.1 Creation Alignment

```
1. Reduce
   $$\left[\begin{array}{c|c|c} \mathcal{U}^T & \mathcal{C}^T & O \\ \hline O & O & I \end{array}\right] \quad (C.1)$$
   to row echelon form F
2. solution[1..M] = 0
3. $\forall c \in M+1,\dots,M+N_{\mathcal{C}}$
4. if (F[rank($\mathcal{U}^T$)..M,c] $\not\equiv$ 0)
5. then solution[$c-M$]= c
6. else solution[$c-M$]= -1
7. next := M+$N_{\mathcal{C}}$+1
8. $\forall r \in 1,\dots,M$
9. if (solution[r] = -1) then solution[r] = next, next++
10. $\forall c \in 1,\dots,M$
11. if F[c,c] $\not\equiv$ 1 then
12. $\forall r \in 1,\dots,M$ if F[r,c] = 1 then swap (F[r],F[c])
```

```
13. $\forall r \in 1, \dots, M$ $\mathcal{A}$[r,1..M] = F[1..M,solution[r]]
```

Complexity: $O(M^3)$

C.2.2 Reduction Alignment

```
1. Reduce
```

$$\left[\begin{array}{c|c|c} \mathcal{U}^T & \mathcal{R}^T & \\ \hline O & O & \end{array} \; I \right] \tag{C.2}$$

```
   to row echelon form F
2. solution[1..M] = 0
3. $\forall c \in M+1, \dots, M+N_{\mathcal{C}}$
4. if (F[rank($\mathcal{U}^T$)..M,c] $\neq 0 \wedge$ independent(F[1..M,c],$\mathcal{A}$)
5. then solution[$c-M$]= c
6. else solution[$c-M$]= -1
7. next := M+$N_{\mathcal{C}}$+1
8. $\forall r \in 1, \dots, M$
9. if (solution[r] = -1) $\wedge$ independent(F[1..M,next],$\mathcal{A}$) then solution[r] = next, next++
10. $\forall c \in 1, \dots, M$
11. if F[c,c] $\neq$ 1 then
12. $\forall r \in 1, \dots, M$ if F[r,c] = 1 then swap (F[r],F[c])
13. $\forall r \in 1, \dots, M$ $\mathcal{A}$[r,1..M] = F[1..M,solution[r]]
```

```
1. independent( newrow, $\mathcal{A}$)
2. $\forall r \in 1, \dots, M$
3. if solution(r) >0 then i:i+1, B[i] := $\mathcal{A}[r]$
4. $N_B = i$
5. Reduce B to row echelon
6. rankB := 0
7. $\forall k \in 1, \dots, N_B$
8. if B[1..M] $\neq$ 0 then rankB := rankB +1
9. B[$N_B$] := newrow
10. Reduce B to row echelon
11. ranknewB := 0
```

```
12. $\forall k \in 1, \dots, N_B$
13. if B[1..M] $\neq 0$ then ranknewB := ranknewB +1
14. if ranknewB = rankB then independent := false else independent := true
```

Complexity: $O(M^4)$

C.3 Pre-Fetching

```
 1. $\underline{\check{J}} = \bigcap_{r=1}^{\theta} \underline{\check{J}_r}$
 2. if $\underline{\check{J}} \neq \emptyset$
 3. then apply interchange transforms(chapter 3) such that $\underline{J} \mapsto [\underline{\hat{J}}, \underline{\check{J}}]$
 4. Set $\underline{J} = \underline{\hat{J}}$
 5. Set $M = \underline{\check{J}}$
    end if
 6. Unexpanded =$\bigcap_{r=1}^{theta} \mathcal{U}_r$
 7. UnexIterators = $\bigcap_{r=1}^{\theta} \underline{\check{J}_r}$
 8. $ExJ = \emptyset$
 9. $RemJ = \underline{J}$
10. WHILE $|Latt(AExJ \leq b)| \leq |\mathcal{I}_C|,\ \forall r \in 1, \dots, \theta \bigwedge \exists j \in \underline{\check{J}} | |Latt(A(ExJ \cup j) \leq b)| \leq |\mathcal{I}_C|,\ \forall r \in 1, \dots, \theta$
11. REPEAT
12. Set $j$ = the most common smallest iterator appearing in all the $\mathcal{U}$s occurring in $RemJ$
13. If $|Latt(A(ExJ \cup j) \leq b)| \leq \underline{\mathcal{I}_{\mathcal{C}}} \forall r | j \in \underline{J_r}$
14. then $M := M + j$
15. $\underline{J} := \underline{J} - j$
16. $ExJ := ExJ \cup j$
17. $RemJ := RemJ - \bigcap_k \underline{\check{J}_k}$ where $j \in \underline{J_r}$
18. else $RemJ := RemJ - j$
19. UNTIL RemJ = $\emptyset$
20. $RemJ := \underline{J}$
21. END DO
22. $RemJ = \underline{J}$
23. $StJ = RemJ$
24. WHILE $StJ \neq \emptyset$
```

```
25. REPEAT
26. Set $j$ = the most common smallest iterator appearing in all the $\mathcal{U}$s occurring in $RemJ$
27. Strip mine $j = j1 \times j2$
28. If $|Latt(A(ExJ \cup j2) \leq b)| \leq \underline{\mathcal{I}_{\mathcal{C}}} \forall r | j \in \underline{J_r}$
29. then $M := M + j2$
30. $\underline{J} := \underline{J} - j2 \cup j1$
31. $ExJ := ExJ \cup j2$
32. $RemJ := RemJ - \bigcap_k \underline{\check{J}_k}$ where $j \in \underline{J_r}$
33. else $RemJ := RemJ - j$end if
34. $StJ := StJ - j$
35. UNTIL RemJ = $\emptyset$
36. $RemJ := \underline{J}$
37. END DO
```

Complexity: $O(m^4 \times |\theta_v|)$

C.4 Volume of Access

```
1. highest :=0
2. FOR v = 1 to $|v|$
3. FOR r = 1 to $\theta_v$
4. if rank($\mathcal{U}_{1.v}$ ) > highest then comparelist = $\mathcal{U}_{1.v}$
5. else if rank($\mathcal{U}_{1.v}$ ) = highest
6. then comparelist = comparelist $\cup \mathcal{U}_{1.v}$
7. end if
8. END FOR
9. END FOR
10. $\forall \mathcal{U} \in$ comparelist
11. FOR i = 1 to $N_{\mathcal{C}}$
12. FOR j = 1 to $N_{\mathcal{U}}$
13. if $\mathcal{C}[i] = \mathcal{U}[j]$
14. then part-iterators := part-iterators + J[i]
15. end if
```

```
16. END FOR
17. END FOR
```

Complexity: $O(N_{\mathcal{C}} \times N_{\mathcal{U}} \times |\theta_v|)$

Bibliography

[ALLE86] Allen F. and et al., *"Compiling for Parallelism"*, Technical Report IBM York Heights, March 1986.

[ALLI90] Alliant Computer Systems Corporation,*"FX-C-2800 Programmer's Handbook"*, Alliant March 1990.

[ANDR90a] André F., Pazat J-L. and Thomas H., *"Pandore: A System to Manage Data Distribution"*, International Conference of SuperComputing, pp 380-389 Amsterdam 1990.

[ANDR90b] André F., Pazat J-L. and Thomas H., *"Data Distribution in Pandore"*, The 5th Distributed Memory Computing Conference 5, pp 1115-1121, Charleston, April 1990.

[ANCO91] Ancourt C. and Irigion F. *"Scanning Polyhedra with DO Loops"*, 3rd ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming , April 1991.

[ANNA86] Annaratone M., Armould E., Gross T., Kung H.T., Lam M.M., Menzilcioglu O., Sarocky K. and Webb J.A., *"Warp Architecture and Implementation"*, Proccedings of the 13th Annual International Symposium on Computer Architecture , IEEE/ACM, pp 346 - 356, June 1986.

[ARVI87] Arvind and Iannucci R.A.,*"Two Fundamental Issues in Multiprocessing"*, Proccedings of the 4th International DFVLR Seminar on Foundations of Engineering Sciences, Bonn, Federal Republic of Germany, June 1987.

[BABE90] Babe .,*"Hypertasking: Automatic Data Parallel Domain Decomposition on the Intel Parallel Supercomputer"*, Technical Report CS/E90-006 Department of Computer Science and Engineering Oregon Graduate Institute, May 1990.

[BALA89] Balasundaram V. and Kennedy K., *"A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations"*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,pp 41-53, 1989.

[BALA91] Balasundaram V., Fox G., Kennedy K. and Kremer U. *"A Static Estimator to Guide Data Partitioning Decisions"*, Proceedings of the Third ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, April 1991.

[BANJ90] Banerjee U. *"Unimodular Transformations of Double Loops"*, CSRD Rpt. No. 1036, CSRD, University of Illinois, Urbana-Champaign, August 1990.

[BOHM90] Böhm A.P.W. Grit, D.H. Oldehoeft R.R., Cann D.C., and Feo J.T. *"SISAL Reference Manual Language Version 2.0"*, Technical Report Computer Science Dept. Colorado State University 1990.

[CALL88] Callahan D. and Kennedy K., *"Compiling Programs for Memory Multiprocessors"*, The Journal of Supercomputing 2,pp151-169 1988.

[CALL90] Callahan D., Carr S. and Kennedy K.,*“Improving Register Allocation for Subscripted Variables”*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation White Plains New York, 1990.

[CALL91] Callahan D., Kennedy K. Porterfield A. *“Software Prefetching”*, Proceedings of the ACM Conference on Architectural Support Programming Languages and Operating Systems, 1991.

[CANN87] Cann D.C.,*“Sisal Multiprocessing Support”*, Technical Report UCID-21115, Lawrence Livermore National Laboratory, July 1987.

[CANN89A] Cann D., *“Compilation Techniques for High Performance Application Computation”*, Technical Report CS-89-108, Department of Computer Science, Colorado State University, 1989.

[CANN89B] Cann D. and Oldehoeft R.R., *“High Performance Applicative Computing”*, Technical Report CS-89-104, Department of Computer Science, Colorado State University, 1989.

[CARR89] Carr S. and Kennedy K., *“Blocking Linear Algebra Codes for Memory Hierarchies”*, Proceedings of the Fourth Siam Conference on Parallel Processing for Scientific Computing , Chicago, Illinois December 1989.

[CHA88] Cha H., *“Performance Maximisation Strategies for a class of Parallel Distributed Programs”*, MSc. Thesis, Department of Computer Science, University of Manchester, 1988.

[CHEN88] Chen M., *“Compiling Parallel Programs by Optimizing Performance”*, Journal of Supercomputing Vol. 2, pp 171-207, 1988.

[CLAC86] Clack C.D. and Peyton Jones S.L., *“The Four-Stroke Reduction Engine”*, ACM Conference on Lisp and Functional Languages, Boston, August 1986.

[DARE88] Darema D., George D.A., Norton V.A. and Pfister G.F., *“ A Single Program Multiple Data Computation Model for Epex/Fortran”*, Parallel Computing No. 7 pp 11 -24, 1988.

[DENN89] Dennis J.B.,, *“Mapping Programs for Data Parallel Execution on the Connection Machine”*, Technical Report Research Institute for Advanced Computer Science, November 1989.

[DONG90] Dongarra J.J., Du Croz J., Hammarling S., and Duff I.,*“A Set of Level 3 Basic Linear Algebra Subprograms”*, ACM Transactions on Mathematical Software, Vol. 16, No. 1, pp 1-17, March 1990.

[DOWL90] Dowling M.L..,*“Optimal Code Parallelization using Unimodular Transformations”*, Parallel Computing Vol. 16, pp 157-171, 1990

[DYER91] Dyer M., Frieze A. and Kannan R., *“A Random Polynomial-Time Algorithm for Approximating the Volume of Convex Bodies”*, Journal of the ACM Vol. 38,No. 1 pp 1-17, January 1991.

[FEO90A] Feo J.T., Cann D.C. and Oldehoeft R.R., *“Partitioning of Regular Computation Multiprocessor Systems”*, Journal of Parallel and Distributed Computing, Vol. 9, pp 312-317, 1990.

[FEO90B] Feo J.T., Cann D.C. and Oldehoeft R.R., *“A Report on the Sisal Language Project”*, Journal of Parallel and Distributed Computing, Vol. 10, pp 349-366, 1990.

[FOX86] Fox G., Johnson M. and Lyzenga, Otto S.,Salmon J and Walker D., *“Solving Problems on Concurrent Processors Volume 1”*, Prentice-Hall, Englewood Cliffs NJ,1986.

[FOX91] Fox G., Hiranandani S., Kennedy K., Koelbel C., Kremer U., Tseng C-W. and Wu M-Y., *“FORTRAN D Language Specification”*, Rice COMP TR90-141, Department of Computer Science, Rice University, February 1991.

[FLYN72] Flynn M.J., *“Some Computer Organisations and their Effectiveness”*, IEEE Transactions on Computers Vol. 21 pp 948-960, 1972.

[GAO90] Gao G.R.,*"Exploiting Fine-Grain Parallelism on Dataflow Architectures"*, Parallel Computing Vol. 13, pp 309-320, 1990

[GANN88] Gannon D. and Jalby W., *"Strategies for Cache and Local Memory Management by Global Program Transformation"*, Journal of Parallel and Distributed Computing , pp87 -616 1988.

[GERA89A] Gerasoulis A. and Nelken I., *"Scheduling Linear Algebra Parallel Algorithms on MIMD Architectures"*, Proceedings of the Fourth SIAM conference on Parallel Processing for Scientific Computing , pp68-95 1989.

[GERA89B] Gerasoulis A., Venugopal S. and Yang T., *"Clustering Task Graphs for Message Passing Architectures"*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation White Plains New York, 1990.

[GERN89] Gerndt M., *"Array Distribution in SUPERB"*, Proceedings of the ACM International Conference of SuperComputing, pp164-174 1989.

[GERN91] Gerndt M., "*Work Distribution in Parallel Programs for Distributed Memory Multiprocessors"*, Proceedings of the International Conference of SuperComputing, June 1991.

[GOFF91] Goff G., Kennedy K. and Tseng S-W., *"Practical Dependence Testing"*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, June 1991.

[GOLU89] Golub G.B. and Van Loan C.F *"Matrix Computations"*, 2nd Edition, The John Hopkins University Press, Maryland, 1989.

[GORN90] Gornish G., Granston E.D. and Viedenbaum A.V., *"Compiler-directed Prefetching in Multiprocessors with Memory Hierarchies"*, Proceedings of the International Conference of Supercomputing, pp354-368 Amsterdam 1990.

[GUPT90] Gupta M. and Banerjee P.,"Automatic Data Partitioning on Distributed Memory Multiprocessors", UILU-ENG-90-2248,CRHC-90-14, Center for Reliable and High-Performance Computing, University of Illinois, October 1990.

[GURD85] Gurd J.R., Kirkham C.C. and Watson I., *"The Manchester Prototype Dataflow Computer"*, Communications of the ACM, Vol. 28, no.1, pp 34-523, January 1985.

[GROS89] Gross T.. and Sussman A., *"Mapping a Single-Assignment Language onto the Warp Systolic Array"*, Technical Report Department of Computer Science Carnegie Mellon Univerity 1989.

[HAGH90] Haghighat M.R. ,*"Symbolic Dependence Analysis for High Performance Parallelizing Compilers"*, MS Thesis,CSRD Rpt No 995, Center for Supercomputing Research and Development, University of Illinois, May 1990.

[HAYW90] Hayworth G., Leunig S., Hammer C. and Reeve M.,*" The European Declarative System, Database, and Languages"* IEEE Micro, December 1990.

[HALS86] Halstead R.H., *" Parallel Symbolic Computing"*, IEEE Computer Vol. 19, No. 8 pp35-43, August 1986.

[HEAT88] Heath M. T. and Romine C.H., *"Parallel Solution of Triangular Systems of Distributed-Memory Multiprocessors"*, Scientific and Statistical Computing, May 1988.

[HILL85] Hillis W.D., *"The Connection Machine"*, Cambridge, MIT 1985.

[HOCK88] Hockney R.W. and Jesshope C.R., *"Parallel Computers 2*, IOP Publishing Ltd., Bristol, 1988.

[HUDA91] Hudak D. and Abraham S., *"Beyond Loop Partitioning: Data Assignment and Overlap to Reduce Communication Overhead"* , International Conference on Supercomputing, June 1991.

[HUDA90] Hudak D. and Abraham S., *"Compiler Techniques for Data Partitioning of Sequentially Iterated Loops"* , Proceedings of ACM International Conference on Supercomputing, June 1990.

[IKUD90] K Ikudome,Fox G.C., Kolawa A. and Flower J.W., *"An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers "*,The 5th Distributed Memory Computing Conference 5, pp 1105-1114, Charleston, April 1990.

[INTE90] Intel Corporation, *"iPSC/2 and iPSC/860 User's Guide"*, Intel, Jan 1990.

[KENN90A] Kennedy K and McKinley K.S., *"Loop Distribution with Arbitrary Control Flow"*, IEEE/ACM Supercomputing '90, New York, November 1990.

[KENN90] Kennedy K and Zima H.P., *"Virtual Shared Memory for Distributed Memory Machines"*, Parallel Software Support Tools, IBM Europe Institute, Austria, 1990.

[KNOB90] Knobe K., Lukas J.D., and Steele G.L., *"Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines"*, Journal of Parallel and Distributed Computing 8, pp102-118 1990.

[KOEL90] Koelbel C. and Mehrotra P. and Rosendale J.V., *"Supporting Shared Data Structures on Distributed Memory Architectures"*, 2nd ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming, pp177-186, Seattle 1990.

[KOEL91] Koelbel C. and Mehrotra P., *" Programming Data Parallel Algorithms on Distributed Memory Machines Using Kali"*, 5th International Conference on Supercomputing, pp414-423, June 1991.

[KUCK77] Kuck D.J., *"A Survey of Parallel Machine Organisation and Programming "*, ACM Computing Surveys, Vol. 9. No. 1, pp29-59, 1977.

[KULK91] Kulkarni D., Kumar K.G., Basu A. and Paulraj A., *"Loop Partitioning for Distributed Memory Multiprocessors as Unimodular Transformations"*, International Conference on Supercomputing, June 1991.

[LAM91] Lam M.S., Rothberg E.E. and Wolf M.E. *"The Cache Performance and Optimizations of Blocked Algorithms"*, Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.

[LAMP74] Lamport L., *"The Parallel Execution of DO loops"*, Communications of the ACM, Vol. 17, No. 2, February 1974.

[LAWR91] Lawrence J., *"Polytope Volume Computation"*, Mathematics of Computation, Vol. 57, No. 195, pp 259-271, July 1991.

[LEE88] Lee C., *"Experience of Implementing Applicative Parallelism on CRAY-XMP"*, Proceedings Conpar88, Stream B, pp 19-25, 1988

[LI90] Li J. and Chen M., *"Index Domain Alignment: Minimising Cost of Cross-Referencing between Distributed Arrays"*, IEEE Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, October 1990.

[LI91] Li J. and Chen M., *"Compiling Communication Efficient Programs for Massively Parallel Machines"*, IEEE Transactions on Parallel and Distributed Systems Vol. 2, No. 3, July 1991.

[LI89] Li K. and Hudak P., *"Memory Coherence in Shared Virtual Memory Systems"*, ACM Transactions on Computer Systems Vol. 7, No.4, pp 321-359, 1989.

[LI90a] Li J. and Chen M., *"Generating Explicit Communication from Shared-Memory Program References "*, IEEE/ACM Supercomputing '90, New York, November 1990.

[LI90b] Li Z. and Yew P-C. and Zhu C-Q., *"An Efficient Data Dependence Analysis for Parallelizing Compilers "*, IEEE Transactions on Parallel and Distributed Systems Vol. 1 No .1 , Jan 1990.

[LI91] Li Z., *"Compiler Algorithm for Event Variable Synchronization "*, CSRD Report 1082, 1991.

[LU90] Lu L-C., *"A Unified Framework for Systematic Loop Transformations"*, 3rd ACM Symposium on the Principles and Practice of Parallel Programming, April 1991.

[LU90a] Lu L-C. and Chen M.C., *"Subdomain Dependence Test for Massive Parallelism"*, IEEE Proceedings of Supercomputing '90, Los Alamitos, CA. 1990.

[MARG90] Margulis N. "i860 Microprocessor Architecture, Intel/Osborne-McGraw Hill, 1990.

[MAYD91] Maydan D.E., Hennesey J.L. and Lam, M.S. *"Efficient and Exact Data Dependence Analysis"*, Proc. of ACM SIGPLAN June 1991.

[MEIK87] Meiko, *" Meiko Computing Surface Hardware Manual"*, Bristol, 1987.

[MUND86] Mundie D.A. and Fisher D.A., *" Parallel Programming in Ada"*, IEEE Computer Vol. 19, No. 8 pp20-25, August 1986.

[MCGR85] McGraw J.R., Skedzielewski S.K., Allan S., Grit D., Oldehoeft R., Glauert J.R.W., Dobes I. and Hohensee P., *"SISAL - Streams and Iteration in a Single-Assignment Language"*, Language Reference Manual, Version 1.2, Lawrence Livermore National Laboratory, California, January 1985.

[NICO90] Nicol D.M and Reynolds P.F.,*"Optimal Remapping of Data Parallel Computations"*, IEEE Transactions on Computers Vol. 39, No. 2, pp206-219, February 1 990.

[NICO90a] Nicol D.M and Saltz J.H.,*"An Analysis of Scatter Decomposition"*, IEEE Transactions on Computers Vol. 39, No. 11, pp1337-1345 , November 1990.

[NOBL88] Noble B. and Daniel J.W.,*"Applied Linear Algebra"*, Prentice-Hall, NJ 1988.

[OBOY92] O'Boyle M.F.P. and Hedayat G., *Load Balancing of Parallel Affine Loops by Unimodular Transformations*, to appear at the European Workshop on Parallel Computing, Barcelona 1992.

[OLDE85] Oldehoeft R.R, Allan S.J.,, *"Adaptive Exact-Fit Storage Management "*, Communications of the ACM Vol.28, pp506-511, May 1985.

[OLDE88] Oldehoeft R.R, Cann D.C., *"Applicative Parallelism on a Shared-Memory Multiprocessor"*, IEEE Software 1988

[PAAL90] Paalvast E.M.,van Gemund A.J. and Sips H.J., *"A Method for Parallel Program Generation with an Application to the Booster Language"*, International Conference of SuperComputing, pp 457-469,Amsterdam 1990.

[PADU86] Padua D.A. and Wolfe M.J., *"Advanced Compiler Optimizations for Supercomputers"*, Communications of the ACM Vol.29 No.12 pp 1184-1201, December 1986.

[PARB87] Parberry I., *"Parallel Complexity Theory"*, Pitman, London 1987.

[PEYT86] Peyton-Jones S., *The Implementation of Functional Programming Languages*, Prentice - Hall 1986.

[PING90] Pingali K. and Rogers A., *"Compiler Parallelization of SIMPLE for a Distributed Memory Machine"*, Technical Report TR 90-1084, Department of Computer Science, Cornell University, January 1990.

[POLY87] Polychronopoulos C.D., and Kuck D.J., *"Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers"*, IEEE Transactions on Computers, Vol. C-36, No. 12, pp. 1425-1439, December 1987.

[PUGH91] Pugh W., *“Uniform Techniques for Loop Optimisation”*, International Conference on Supercomputing, June 1991.

[RAMA90] Ramanujan J. and Sadyappan P., *“Nested Loop Tiling for Distributed Memory Machines”*, The 5th Distributed Memory Computing Conference 5, pp 1115-1121, Charleston, April 1990.

[RANE87] Ranelletti J.E., *“Graph Transformation Algorithms for Array Memory Optimization in Application Languages”*, Ph.D. thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[REDD73] Reddaway S.F., *“DAP - A Distributed Array Processor”*, IEEE/ACM 1st Annual Symposium on Computer Architecture, Florida, 1973.

[REED87] Reed D.A., Adams L.M. and Patrick M.L. *“Stencils and Problem Partitionings: Their Influence on the Performance of Multiple Processor Systems ”*, IEEE Transactions on Computers Vol c-36 No. 7, July 1987.

[RIBA90] Ribas H.B, *“Automatic Generation of Systolic Programs from Nested Loops”*, Ph.D. thesis, Carnegie-Mellon University, Computer Science Department, June 1990.

[ROGE90] Rogers A. and Pingali K., *“Process Decomposition through Locality of Reference”*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,pp 69-80, 1989.

[ROGE91] Rogers A., *“Compiling for Locality of Reference”*, Technical Report TR 91-1195, Department of Computer Science, Cornell University, March 1991.

[ROSI90] Rosing M., Schnabel R. and Weaver R. *The DINO parallel programming language*, Technical Report CU-CS-457-90, University of Colorado at Boulder, April 1990.

[RUHL90] Rühl R. and Annaratone M., *“Parallelization of FORTRAN Code on Distributed-memory Parallel Processors”*, Proceedings of the International Conference of SuperComputing, pp342-353 Amsterdam 1990.

[RUSS78] Russell R. M., *“The CRAY-1 Computer System ”*, Communication of the ACM, Vol. 21, pp63-72 1978.

[SALT90] Saltz J., Crowley K., Mirchandaney R. and Berryman *“Run-time Scheduling and Execution of Loops on Message Passing Machines”*, Journal of Parallel and Distributed Computing, Vol.8 pp303-312, 1990.

[SARK88] Sarkar V., *“An Automatically Partitioning Compiler for SISAL”*, Proceedings Conpar88, Stream B, pp 26-33, 1988

[SARK89] Sarkar V., *“Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors”*, Research Monographs in Parallel and Distributed Computing, The MIT Press, Cambridge, Massachusetts, 1989.

[SARG86] Sargeant J., *“Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines”* Tech. Rep. UMCS-86-11-5, Dept. of Computer Science, University of Manchester, 1986.

[SEQU87] Sequent, *“Balance Technical Summary”*, Sequent Computer Inc., Beverton, Oregon, 1987.

[SCHR86] Schrijver A., *“Theory of Linear and Integer Programming”*, Wiley, Chichester 1986

[SHEN90] Shen Z., Li Z. and Yew P-C,*“An Empirical Study of Fortran Programs Parallelizing Compilers”*, IEEE Transactions on Parallel and Distributed Systems Vol. 1 No. 3, July 1990.

[SHOR73] Shore J.E.,*“Second Thoughts on Parallel Processing”*, Computer Electrical Engineering, Vol. 1 pp 95-109, 1973.

[SHIE88] Shield D.T., *"Translating SISAL into occam"*, (M.Sc. Thesis), Department of Computer Science, University of Manchester, 1988.

[SKED85] Skedzielewski S.K. and Glauert J.R.W., *"IF1 - An Intermediate Form for Applicative Languages"*, Reference Manual M-170, Lawrence Livermore National Laboratory, California, January 1985.

[STAN86] Stanley R.P., *" Enumerative Combinatorics Volume 1"*, Wadsworth and Brooks, 1986.

[STUA91] Stuart J.L. and Weaver J.R. *"Matrices that Commute with a Permutation Matrix"*, Linear Algebra and its Applications, No. 150 pp 255-265, 1991.

[TANG90] Tang P., Yew P-C. and Zhu C-Q., *"Compiler Techniques for Data Synchronization on Nested Parallel Loops"*, Proceedings of the International Conference on Supercomputing, 1990, Vol. 1, pp177-186, May 1990.

[THOR70] Thorton J.E., *"Design of a Computer, the Control Data 6600*, Scott, Foresman and Co., Illinois, 1970.

[TSEN89] Tseng P-S., *"A Parallelizing Compiler for Distributed Memory Parallel Computers"*, Ph.D. Thesis, Technical Report CMU-CS-89-148, Computer Science Department, Carnegie Mellon University, May 1989.

[TSEN90] Tseng P-S., *"Compiling Programs for a Linear Systolic Array"*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation White Plains New York, 1990.

[VELD90] Van de Velde E.F.,*"Data Redistribution and Concurrency"*, Parallel Computing Vol. 16, pp 125-138,1990.

[WATS88] Watson I. et al., *"Flagship: A Parallel Architecture for Declarative Programming"*, Proceedings of the 15th Annual Symposium on Computer Architectures, pp. 124-130, 1988.

[WEIS91] Weiss M., *" Strip Mining on SIMD Architectures"*, International Conference on Supercomputing, June 1991.

[WEND89] Wendleborn A., *"The Development and Efficient Execution of SISAL programs"*, Proceedings of the Eilat Workshop on Dataflow Computation, Bic L. and Gaudiot J.L., May 1989.

[WHIT90] Whitfield D. and Soffa M.L., *"An Approach to Ordering Optimizing Transformations"*, Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming, pp177-186, Seattle 1990.

[WOLF91a] Wolf M.E. and Lam M.,*"A Data Locality Optimizing Algorithm"*, ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1991.

[WOLF89] Wolfe M.,*"Optimising Supercompilers for Supercomputers"*, Research Monograph in Parallel and Distributed Computing, Pitman, London, 1989.

[WOLF90a] Wolfe M.,*"Massive Parallelism through Program Restructuring"*, Technical Report CS/E90-009 Department of Computer Science and Engineering Oregon Graduate Institute, June 1990.

[WOLF90b] Wolfe M.,*"Data Dependence and Program Restructuring"*, Technical Report CS/E90-007 Department of Computer Science and Engineering Oregon Graduate Institute, May 1990.

[WOLF90c] Wolfe M.,*"Serial vs Parallel Optimisations"*, Technical Report CS/E90-010 Department of Computer Science and Engineering Oregon Graduate Institute, July 1990.

[WOLF90d] Wolfe M.,*"A Loop Restructuring Research Tool"*, Technical Report CS/E90-010 Department of Computer Science and Engineering Oregon Graduate Institute, August 1990.

[WOLF90e] Wolfe M. and Tseng C-W,.“*The Power Test for Summarising Data Access and its Use in Parallelism Enhancing Transformations’*, Technical Report CS/E90-010 Department of Computer Science and Engineering Oregon Graduate Institute, August 1990.

[WOLF91] Wolfe M.,“*Data Dependence and Program Restructuring*”, The Journal of Supercomputing Vol.4 No. 4, January 1991.

[WOLF88] Wolfram S.,“*Mathematica: A System for doing Mathematics by Computer* ”, Addison-Wesley, Reading, Mass., 1988.

[YANG91] Yang A. and Choo Y-L., “*Parallel-Program Transformations Using a Metalanguage*”, Third ACM SIGPLAN Syposium on Principles and Practice of Parallel Programming. pp 11-20, April 1991.

[ZIMA88] Zima H.P. Bast H,J. and Gerndt H.M, “*A Tool for Semi-Automatic MIMD/SIMD parallelization*”, Parallel Computing 6, pp1 - 18 1988.