A Transformational Approach to Compiling Sisal for Distributed Memory Architectures

Michael O'Boyle and G.A. Hedayat
Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK

ABSTRACT

This paper 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. Three criteria for partitoning are given as well as asystematic method to map the data and computation to the processors. Finally a new pre-fetching procedure, which prevents redundant non-local accesses, is presented

1 INTRODUCTION

Distributed memory architectures (DMAs) hold the promise of scalable performance. By simply adding extra processor/memory pairs it is a possible to solve larger problems or reduce execution time without the memory access bottlenecks of shared memory architectures. However, until recently exploiting the potential performance has been at the expense of great programmer effort. 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 the wide acceptance of distributed memory computing is the primitive state of compiler technology. This paper shows that it is possible 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.

Sisal [MCGR85] is a first-order functional language supporting arrays and iteration. The advantages of writing in a high level language are obvious, the cost, however, is that compilers must now perform the necessary analysis to map the program to the particular architecture in such a manner so as to be competitive with existing methods.

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 a variable may be overwritten many times. This is a real problem in parallel machines as the most recent value of a variable has be maintained. The problem of *memory coherence* has normally been solved by the use of expensive hardware mechanisms. Sisal does not require any memory coherence mechanism to ensure a correct implementation.

The model of computation used is SPMD [DARE88] which exploits data parallelism where the elements of an array mat be evaluated in parallel. This approach, although unusual for Sisal, has been widely used in hand-written implementations [FOX86] and has been shown to be very effective. SPMD forms the basis of many current implementations on DMAs [ANDR90a],[RUHL90],[KOEL91],[ROSI90],[ROGE91],[FOX91].

This paper describes a compilation strategy which consists of a sequence of ordered transformations. Firstly a restricted sub-set of Sisal is translated into an algebraic representation which, after suitable optimising transformations, is translated into local imperative programs, one per processor, which is then compiled by the processor's native compiler.

There has been a large amount of research into the detection of program parallelism e.g. [WOLF89]. 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 paper. 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 paper applicable to existing imperative languages.

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 paper 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 processors 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.

The next section describes the overall transformation scheme and introduces the notation used throughout the remainder of this paper. The third and fourth sections describe transformations to improve load balancing and alignment respectively. The fifth section briefly outlines three criteria for data partitioning and the sixth describes how these partitions are mapped to the processors. In the seventh section , pre-fetching transformations transformations are presented which exploit data re-use. Finally some concluding remarks are made.

2 NOTATION

The compilation of Sisal for a DMA can be considered as a sequence of transformations:

$$P_0 \overset{\tau_0}{\mapsto} P_1 \overset{\pi_1}{\mapsto} P_2 \overset{\pi_2}{\mapsto} P_3 \overset{\zeta}{\mapsto} \begin{bmatrix} \underline{P_4} \\ \underline{P_4} \\ \vdots \\ \underline{P_4} \end{bmatrix} \overset{\pi_3}{\mapsto} \begin{bmatrix} \underline{P_5} \\ \underline{P_5} \\ \vdots \\ \underline{P_5} \end{bmatrix} \overset{\tau_1}{\mapsto} \begin{bmatrix} \underline{P_6} \\ \underline{P_6} \\ \vdots \\ \underline{P_6} \end{bmatrix} \tag{1}$$

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. In the following sub-sections Sisal, the computation set representation and the imperative language used are described. On going work at Manchester will use 'C' as the target imperative language.

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.

2.1 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 paper 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.

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* $Q = (A, J, b, \mathcal{S})$ *is a computation set.*

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

2.2 ARRAY OCCURRENCE

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

$$a[i,j], \;\; b[i,j], \;\; b[j,i+3] \tag{2}$$

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* There are two distinct types of occurrences matrix.

A **c-occurrence** corresponds to a left-hand usage of an array. Due to the strictness of array structures in Sisal it is not possible to create or write to part of an array structure and as Sisal is single-assignment, there is only one c-occurrence matrix $\mathcal{C}_v$ for each array v..

A **u-occurrence** corresponds to a right hand usage of an array. There is no restriction on the number of u-occurrences of an array. Each rth usage of an array v is represented by $\mathcal{U}_{v.r}$,

A **segment** is a fragment of a program which contains no iterators and has only one c-occurrence. 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 and m is the number of nested iterators.

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

2.3 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

$$J = [j_1, j_2, \ldots, j_m]^T \tag{3}$$

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

$$J = \begin{bmatrix} i \\ j \\ k \end{bmatrix} \tag{4}$$

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

$$LJ \geq l \tag{5}$$

$$UJ \leq u \tag{6}$$

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:

$$\left[\begin{array}{c} -L \\ \hline U \end{array}\right] J \leq \left[\begin{array}{c} -l \\ \hline u \end{array}\right] \tag{7}$$

or as

$$AJ \leq b \tag{8}$$

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

Definition 2 *Given A, an $(linteger \times m)$ matrix and b a $(m \times 1)$ integer vector, where m is the number of iterators then* $\mathbf{Latt(A.b)}$ *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* $\mathbf{AJ} \leq \mathbf{b}$. *Thus the lattice is a subset of a polytope that have integer coordinates.*

The lattice of points corresponds directly to the *iteration space.*

For example consider the Sisal program in figure 1 It is possible to represent the loop bounds as follows:

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ -1 & -3 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{array}\right] \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \left[\begin{array}{c} -1 \\ -1 \\ 6 \\ \hline 16 \\ 24 \\ -7 \end{array}\right] \tag{9}$$

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

```
a:= for i in 1,16  cross j in 1,24
    returns array of
      for k in 3*j+i+6,4*i-7
      returns value of sum
        if i>k
        then  b[i,k] * b[j,i+3]
        else  0
        end if
      end for
    end for
```

Figure 1: A Sisal Program

2.4 INCORPORATING CONDITIONALS IN POLYTOPES

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

$$i > k \Leftrightarrow -i + k \leq -1 \tag{10}$$

$$i \leq k \Leftrightarrow i - k \leq 0 \tag{11}$$

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:

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ -1 & -3 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \\ \hline -1 & 0 & 1 \end{array}\right] \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \left[\begin{array}{c} -1 \\ -1 \\ 6 \\ \hline 16 \\ 24 \\ -7 \\ \hline 0 \end{array}\right] \tag{12}$$

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 $[\perp]$ is used.

Definition 3 *The* **canonical form** *of a polytope of a segment is*

$$\left[\begin{array}{c} -L \\ \hline U \\ \hline \mathcal{E} \end{array}\right] J \leq \left[\begin{array}{c} -l \\ \hline u \\ \hline \epsilon \end{array}\right] \tag{13}$$

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

2.5 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:

$$[i_1, i_2, \ldots, i_{N_v}]^T = \mathcal{I}^v \tag{14}$$

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:

$$\lambda \leq \mathcal{I} \leq \upsilon \tag{15}$$

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 **??** is given by:

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \leq \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \leq \begin{bmatrix} 16 \\ 24 \end{bmatrix} \tag{16}$$

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.

When the many computation sets are mapped to several processors, each processor will have its own local iteration and data spaces. In this paper, 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 $\underline{\mathcal{I}^v}$.

2.6 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.

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 is typical.

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.

```
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] := 0.0
     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: An Imperative Program

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.

2.7 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 $Q = (A, J, b, \mathcal{S})$ and $\mathcal{S} = (F, C, U)$. the basic translation scheme is to produce a nest of appropriate loops given by the J and 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 $\mathcal{E}, \epsilon$ entries in the polytope $AJ \leq b$ are represented as **IF** statements. These are placed immediately after the iterators they reference. In the simple case of addition, the loop body will be as in 17. However, in general, the exact form will depend upon the operands in F.

$$v[\mathcal{C}J]{:=}\mathbf{Get}(v_1[\mathcal{U}_1 J + \nu_1]) + \cdots + \mathbf{Get}(v_{\theta_v}[\mathcal{U}_{\theta_v} J + \nu_{\theta_v}]) \tag{17}$$

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 is able to 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.

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 paper 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 paper only **for** loops are considered for exploitation of parallelism. **for initial** loops are considered to require sequential execution. 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 [?].

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. There is a discussion of this point in [OBOY91] 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.

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**. 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. We restrict our attention to condition free-loops i.e. $\mathcal{E} = 0$. In [OBOY91], load balancing in the presence of conditionals is described.

The iterator j_b 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

$$\left[\begin{array}{c|c|c} L_f & 0 & 0 \\ \hline y_L & 1 & 0 \\ \hline A_L & x_L & L_g \end{array}\right] \left[\begin{array}{c|c|c} U_f & 0 & 0 \\ \hline y_U & 1 & 0 \\ \hline A_U & x_U & U_g \end{array}\right] \qquad (18)$$

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 **??** and **??** we require the following:

$$y_L = y_U = 0, x_L = x_U = 0 \qquad (19)$$

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

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.

```
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: A Sisal Program

3.1 TRANSFORMATIONS

Legal transformations include any reordering of the computation that maintains the data dependency of the original program. Additionally after transformation the loop should be in the structure defined by **??** and **??**. More general forms of loop structure are studied in [OBOY91].

In [OBOY92] an algorithm which determines a unimodular transformation T to change the basis of the iteration space so that one or more iterators are in load balanced form is given. This algorithm is not given here but as an example consider the program in figure 5:

In its present form no one iterator satisfies the invariance condition. The upper and lower bounds for each of the iterators are as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -2 & -2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 1 \\ n-1 \\ 1 \end{bmatrix} \qquad (20)$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -3 & -2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \begin{bmatrix} n \\ 2n+1 \\ n \end{bmatrix} \qquad (21)$$

It is possible to determine if this program may be transformed into an invariant form and in this case only the second iterator may be transformed into load balanced. The neccessary transformation T can be derived and is:

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \qquad (22)$$

which gives the imperative program in figure [?]

No other loop depends on the new j loop, therefore partitioning with respect to this loop will give a load balanced implementation.

4 ALIGNMENT

Compilation should minimise parallel time by firstly utilising machine parallelism and secondly reducing overhead.

```
FOR  i = 1 TO n
  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 4: An Imperative Program

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

Figure 5: A Sisal Program

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 section 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.

Consider the Sisal fragment in figure 5 where:

$$a[i,j] = b[j,i] \tag{23}$$

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

$$\pi : b[i,j] \mapsto b^{'}[j,i] \tag{24}$$

In general there is more than one point of alignment between two arrays. Two arrays $v[\mathcal{C}J]$ and $w[\mathcal{U}J+u]$reference each other when

$$\mathcal{C}J = \mathcal{U}J + u, J \in Z^m \tag{25}$$

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

The object of compiling for alignment can now be stated as finding a transformation $A = (\mathcal{A}_i, a_i)$ where $S_{\mathcal{C},\mathcal{U}}$ is defined as

$$S_{\mathcal{C},\mathcal{U}} = \{J \in Z^m, \mathcal{C}J = \mathcal{A}_i\mathcal{U}J + a_i + u\} \tag{26}$$

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

Even for pairwise alignment the above optimization problem, in general, involves enumerating the polytope, $S_{\mathcal{C},\mathcal{A}_i\mathcal{U}} \cap Latt(A.b)$, for each candidate, $(\mathcal{A}_i, a_i)$, which can be an expensive task.

In program **??** if A is chosen as:

$$\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \mathcal{AU} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} \tag{27}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} \tag{28}$$

$S = Z^m$ and $|S \cap Latt(A.b)| = n^2$ which implies that for any data partition there will be n^2 local access and hence no non-local access regardless of any subsequent data partition.

We restrict our attention to class of affine bijective alignment transformations i.e.

$$A : \mathcal{U}J + u \mapsto \mathcal{AU}J + a + u \tag{29}$$

where $\mathcal{A}$ is an $M \times M$ invertible matrix elements and a is an M integer shift vector. $M = \max(N_{\mathcal{C}}, N_{\mathcal{U}}, m)$, so $\mathcal{C}, \mathcal{U}$ are padded with zero rows to make them of equal size.

4.1 MEASURE OF ALIGNMENT

A suboptimal criterion is used to determine the alignment transformation A for a pair of arrays v, w.

$H_{\mathcal{U},\mathcal{C}}$ is defined to be the number of perfectly aligned indices of the corresponding arrays v,w. For example in the assignment

$$v[i, i+j, i+k] = w[j, i+j, j] \tag{30}$$

which can be represented as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \\ k \end{bmatrix} \tag{31}$$

the value of H is one as only the second indices are equal. It can be shown that

$$H_{\mathcal{U},\mathcal{C}} = dim\, S = dim\{J \in Z^m : \mathcal{U}J + u = \mathcal{C}J\} \tag{32}$$

It is the purpose of any applied transform $\mathcal{A}$ to maximise $H_{\mathcal{AU},\mathcal{C}}$. In [OBOY92a] the conditions for existence of such a transformation and the maximum value of H for a given computation are determined.

```
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+3,i+j] * c[k,n-i]
      end for
    end for
```

Figure 6: A Sisal Program

4.2 ROW ECHELON FORM

The optimal shift vector a can always be chosen independently.

$$a = -u \tag{33}$$

$\mathcal{A}$ is chosen to maximize the number of perfectly aligned indices of $\mathcal{AU}$ and $\mathcal{C}$. In order to determine the form of $\mathcal{A}$, such that $H_{\mathcal{AU},\mathcal{C}}$ is maximised $\mathcal{U}^T$ is concatenated with $\mathcal{C}^T$ and reduced by any row operation to row echelon form and then reordered to a give a quasi-diagonal form.

As an example consider the program in figure 6: This has the following occurrence :

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_a \begin{bmatrix} i \\ j \\ k \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}_b \begin{bmatrix} i \\ j \\ k \end{bmatrix}, \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}_c \tag{34}$$

$$\begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ n \\ 0 \end{bmatrix} \tag{35}$$

As long as the occurrence matrices refer to different arrays, they may be compared pair-wise with the common array. Thus on aligning b with a we have and reducing to row-echelon form we have:

$$\left[\begin{array}{ccc|ccc|ccc} 1 & 0 & 0 & -1 & 1 & 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{36}$$

giving

$$\mathcal{A} = \begin{bmatrix} -1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} a = \begin{bmatrix} -3 \\ 0 \\ 0 \end{bmatrix} \tag{37}$$

Which gives the following new occurrence matrix:

$$\mathcal{A} \times \mathcal{U} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} a + u = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{38}$$

This process can be repeated for array c to give the following program:

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

Before the alignment transformations the value of $H_{b,a}$ and $H_{c,a}$ was 0. After the alignment transformations, $H_{b,a} = 2$ and $H_{c,a} = 1$. If the array a is now partitioned along the first index, then the subsequent implementation will have no non-local access and hence no overhead due to communication. This program transformation provides the relative allocation of data so as to achieve maximum data alignment. Such an allocation strategy would not be possible using existing automatic techniques [LI90],[KNOB90].

Once alignment has been determined, all other occurrences of that array variable must be aligned accordingly. In general there may be conflict [KNOB90] so a resolution heurestic must be employed [OBOY91].

5 PARTITIONING

Determining how to partition the data and computation space highlights the trade-off between load imbalance and communication overhead. In [OBOY91], it is shown that there are three different partitioning criteria which can be outlined as follows:

1. Partition the iteration space and hence the data so as to minimise load imbalance.
2. Partition the data and hence computation so as to maximise alignment, i.e. accesses should be to local data.
3. Partition the data and hence computation so as to mininise non-local accesses assuming that caching transformations ar available to exploit re-use of data.

To illustrate the implication of each scheme, consider the program in ?? and the case when there are $p = 10$ processors.

```
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[j,k]
      end for
    end for
```

If the load balancing criteria is followed, then th iteration space should be partitioned by the jth iterator which corresponds to partioning array a by the second index i.e. by columns. As arrays b, c are aligned, they will also be partitioned in the same manner. Each processor will perform exactly 50,500 iterations and will be perfectly load balanced.

Partitioning so as to to maximise alignment would imply partitioning by the first index of a as arrays a, b are perfectly aligned on the first index and thus all accesses to b would now be local. The only non-local accesses will be to c, the maximum number occurring in the last processor and is equal to 85,950. Neither b nor c is aligned with a on the second index; partitioning by the second index increases the maximum non-local access to 99,900.

If, however, pre-fetching/caching transformations are available after partitioning and mapping, then it is preferable to partition by both i and j. The maximum non-local access assuming caching will now reduce to 4,556.

It is precisely because of these trade-offs that many implementations,[refs] leave the partitioning decisions to the programmer. It depends on the relative cost of computation and communication and whether a hardware cache exists or there is compiler transforms to utilise data re-use. In [OBOY92], a compiler heurestic is presented to determine which criteria is more important.

6 MAPPING

6.1 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.

6.2 MAPPING INDICES

Each array, v, in a computation set has the following index domain:

$$[i_1, i_2, \ldots, i_N]^T = \mathcal{I}^v \tag{39}$$

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. 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:

$$\zeta : \mathcal{I} \mapsto \underline{\mathcal{I}_1} \times \underline{\mathcal{I}_2} \times \cdots \times \underline{\mathcal{I}_p} \tag{40}$$

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:

$$\left[\begin{array}{c} -I_N \\ I_N \\ \hline -e_x \\ e_x \end{array}\right] \left[\begin{array}{c} \underline{i_1^k} \\ \vdots \\ \underline{i_x^k} \\ \vdots \\ \underline{i_N^k} \end{array}\right] \le \left[\begin{array}{r} -\lambda \\ v \\ \hline -[(z-1) \times b + k \times r + 1] \\ z \times b + k \times r \end{array}\right] \tag{41}$$

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 $d = \lceil \frac{v_x - \lambda_x + 1}{r} \rceil$ is the number of sequences or wrap arounds.

```
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 7: A Sisal Fragment

Thus **??** can be reformed to define the value of the **local** indices in some processor z. The range of the local indices in equation **??** is of the form given in 42.

$$\left[\begin{array}{r} -I \\ I \end{array}\right] \underline{\mathcal{I}_z^k} \le \left[\begin{array}{r} -\underline{\lambda_z} \\ \underline{v_z} \end{array}\right] \tag{42}$$

and can be easily extended to multiple dimensions.

To illustrate the partitioning of data consider the Sisal fragment described in figure 7, where the index ranges for both arrays, a and c, are to be be determined.

Arrays a and c have the following bounds:

$$a[1 \ldots 100, 1 \ldots 100] c[1 \ldots 200, 1 \ldots 300] \tag{43}$$

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:

$$b_{\chi_i} = \frac{(v_{\chi_i} - \lambda_{\chi_i} + 1)}{p_i} + 1 \forall i \in 1, \ldots, q \tag{44}$$

After the removal of redundant constraints, consider the the index values of array a on, say, processor $z = (3, 2)$:

$$\left[\begin{array}{rr} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c} \underline{i_{\chi_1}} \\ \underline{i_{\chi_2}} \end{array}\right]_a \le \left[\begin{array}{r} -51 \\ -26 \\ \hline 76 \\ 50 \end{array}\right] \tag{45}$$

In other words the array a has the following size on this processor:

$$a[51 \ldots 76, 26 \ldots 50] \tag{46}$$

The amount of continuous data per processor $p_1 = 1, \ldots, 4$ 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.

$$d_{\chi_1} = \lceil \frac{200}{100} \rceil = 2 \tag{47}$$

$$d_{\chi_2} = \lceil \frac{300}{100} \rceil = 3 \tag{48}$$

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:

c[(51..76 , 26..50) , (51..76 , 126..150) , (51..76 , 226..250) , (151..176 , 26..50) , (151..176 , 126..150), (151..176 , 226..250)]

The mapping results in the elements $a[51..76, 26..50]$ and $c[51..76, 26..50]$ 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

6.3 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

The iteration space is partitioned in a similar manner to that of the index space, namely:

$$\zeta : J \mapsto \underline{J_1} \times \underline{J_2} \times \cdots \times \underline{J_p} \tag{49}$$

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:

$$\underline{A} \underline{J_z} \leq \underline{b} \tag{50}$$

This polytope is derived from:

$$\begin{bmatrix} -L \\ \hline U \\ \hline \mathcal{E} \\ \hline -\mathcal{C} \\ \hline \mathcal{C} \end{bmatrix} \begin{bmatrix} \underline{j_1} \\ \underline{j_2} \\ \vdots \\ \underline{j_m} \end{bmatrix} \leq \begin{bmatrix} -l \\ \hline u \\ \hline \epsilon \\ \hline -\lambda \\ \hline v \end{bmatrix} \tag{51}$$

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 [OBOY91]. As an example, consider the program in figure 8.

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:

```
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 8: A Sisal Program

```
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 9: An Imperative Program

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ \hline 0 & -1 & 1 \\ \hline -1 & 0 & 0 \\ 0 & -1 & 0 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \underline{j_1} \\ \vdots \\ \underline{j_3} \end{bmatrix} \leq \begin{bmatrix} -1 \\ -1 \\ -1 \\ \hline 16 \\ 16 \\ 0 \\ \hline 0 \\ \hline -9 \\ -9 \\ \hline 12 \\ 12 \end{bmatrix} \tag{52}$$

By removing redundant conditions of the polytope, the following polytope is found:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ \hline 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} \underline{j_1} \\ \vdots \\ \underline{j_3} \end{bmatrix} \leq \begin{bmatrix} -9 \\ -9 \\ -1 \\ \hline 12 \\ 12 \\ 0 \\ \hline 0 \end{bmatrix} \tag{53}$$

which corresponds to the program in figure 9. 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.

```
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 10: 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 11: A Sisal Program

7 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 outlined within the following sub-sections. Other researchers have been working in this area, most notably M.E. Wolf and M. Lam [WOLF91b].

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.

7.1 INVARIANCE

To motivate the analysis and the remainder of this section, consider the program in figure 11. If the program is partitioned by j, 'column-wise', this will result in the program described in figure **??**, where jlo and jhi are the local lower and upper bounds of j which have been determined by the mapping transformation. To aid presentation the initialisation of variables will not be shown in the remaining examples of this section.

The amount of array b that is accessed by the reference $b[i,k]$ is $|j| \times |k|$ where $|j|$ and $|k|$ are the number of iteration

```
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 12: An Imperative Program

points of j and k respectively. The **volume of access** is

$$||[jlo,\ldots,jhi]|| \times n = \frac{n^2}{p} \tag{54}$$

However, the number of times b is, at present, accessed is defined by the number of loop iterations performed:

$$n \times ||[jlo,\ldots,jhi]|| \times n = \frac{n^3}{p} \tag{55}$$

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 with the introduction of local temporaries

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 $|\mathcal{U}_{1.v}(\underline{J^{m-d}})|$ points of reuse. The program in figure 11 has the occurrence matrix:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \tag{56}$$

There is no reference to iterator i in the u-occurrence matrix of b, which implies that the value of $\mathcal{U}_{1.b}$ remains constant for all values of i. In other words $\mathcal{U}_{1.b}$ is **invariant** of the i iterator.

The null space of $\mathcal{U}_{1.v}(\underline{J^m})$, $\mathcal{N}(\mathcal{U})$, are the iterators which $\mathcal{U}_{1.v}$ is invariant of.

Let $\underline{\hat{J}}$ be the d dimensional vector of the d iterators that $\mathcal{U}_{1.v}$ makes reference to, and conversely $\underline{\check{J}}$ are the $m-d$ iterators that $\mathcal{U}_{1.v}$ does not make reference to. Intuitively $\underline{\hat{J}}$ should be the outer most loops and $\underline{\check{J}}$ the inner most ones in the transformed program.

To determine the transformation, the main task is to perform the reordering transformation:

$$\underline{J} \mapsto \begin{bmatrix} \underline{\hat{J}} \\ \underline{\check{J}} \end{bmatrix} \tag{57}$$

This transformation requires $O(m)$ interchanges of iterators which is described in [OBOY91]. To illustrate the usefulness of this technique, consider the program in figure 15. If we concentrate on the polytope involved in the access of array b, then it has the following iteration space:

```
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
           if k >= n then b[k,2*k]
           else 0 end if
         end for
       end for
```

Figure 13: A Sisal Program

```
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 14: An Imperative Program

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 2 & 1 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 1 & 1 \\ 0 & 0 & -1 \end{array}\right] \left[\begin{array}{c} i \\ j \\ k \end{array}\right] \leq \left[\begin{array}{c} -1 \\ -1 \\ 0 \\ \hline n \\ n \\ 0 \\ \hline -n \end{array}\right] \tag{58}$$

Now

$$\underline{\check{J}} = \mathcal{N}(\mathcal{U}_{1.b}) = [i,j]^T \tag{59}$$

and

$$\underline{\hat{J}} = \underline{J}^3 - \mathcal{N}(\mathcal{U}_{1.b}) = [k]^T \tag{60}$$

This can be expressed as:

$$\left[\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 1 & 2 & 1 \\ 1 & -3 & 1 \end{array}\right] \left[\begin{array}{c} k \\ i \\ j \end{array}\right] \leq \left[\begin{array}{c} -n \\ -1 \\ -1 \\ \hline 3n-1 \\ n \\ n \\ \hline 0 \\ 0 \end{array}\right] \tag{61}$$

which on gives the program in figure 14. This program illustrates the power of pre-fetching 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. For a complete description of the the program transformations required to exploit invariancy see [HEDA92].

```
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] + c[i,k]
         end for
       end for
```

Figure 15: A Sisal Program

```
FOR k = 1 TO 16
  FOR i = 1 TO 4
    tempc[i] := Get(c[i,k])
  END FOR
   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] +tempc[i]
    END FOR
  END FOR
END FOR
```

Figure 16: An Imperative Program

In general exploiting invariance of access requires scalar expansion of temporaries. This is especially true if there are more than one u-occurrence or there are u-occurrences which are not orthogonal to the iteration space. Consider the program in figure **??** which is transformed to give the local program of figure **??** after partitioning and pre-fetching. which now can be finally be written as in figure 16.

The temporaries are scalar expanded but there are $O(n)$ less accesses to b.

The amount of scalar expansion allowed is restricted by the memory space available which is referred to as the scalability constraint. To maximise the amount of invariance/ scalar expansion will often rrequire strip-mining of iterators. In [OBOY91], a general algorithm to perform pre-fetching on arbitrary affine loops/array occurrences in the presence of conditionals and interleaving is given.

8 CONCLUSION

This paper has outlined a new approach to compiling an array-orientated language, Sisal, for DMAs. It used a SPMD model of computation to exploit data parallelism. By expressing each program in a linear algebraic feramework, it is posible to apply optimising transformations to reduce overhead. In [OBOY91] this scheme is extended to include interleaved mappings and is applied successfully to general problems such as Cholesky factorisation and Triangular back substitution and problems including indirection such as Livermore Loop 14. The next stage of the project, is the implementation of this scheme on a DMA. Current work is involved with the Kendal Square Research Machine newly installed at Manchester.

REFERENCES

[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.

[BANE90] Banerjee U.,**Unimodular Transformations of Double Loops**, Proc. of 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine CA, August 1990.

[CANN89A] Cann D., **"Compilation Techniques for High Performance Application Computation"**, Technical Report CS-89-108, Department of Computer Science, Colorado State University, 1989.

[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.

[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.

[HEDA92] Hedayat G.A. and O'Boyle M.F.P. **A New Program Transformation to Minimise Communication in Distributed Memory Architectures**,submitted to PARLE'92, Paris June 1992.

[KENN90A] Kennedy K and McKinley K.S., **"Loop Distribution with Arbitrary Control Flow"**, IEEE/ACM Supercomputing '90, New York, November 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.

[KOEL91] Koelbel C. and Mehrotra P., **" Programming Data Parallel Algorithms on Distributed Menory Machines Using Kali"**, 5th International Conference on Supercomputing, pp414-423, June 1991.

[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.

[LU90] Lu L-C., **"A Unified Framework for Systematic Loop Transformations"**, 3rd ACM Symposium on the Principles and Practice of Parallel Programming, April 1991.

[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.

[OBOY91] O'Boyle M.F.P., **Program and Data Transformations for Efficient Execution of Array Computation on Distributed Memory Architectures**, PhD thesis, University of Manchester, Dept. of Computer Science, to appear December 1991.

[OBOY92] O'Boyle M.F.P. and Hedayat G.A. **Load Balancing of Parallel Affine Loops by Unimodular Transformations**, to appear at the European Workshop on Parallel Computing, Barcelona, March 1992.

[OBOY92a] O'Boyle M.F.P. and Hedayat G.A. **Data Alignment: Transformations to Reduce Communication on Distributed Memory Architectures**,submitted to the Scalable High Performance Computing Conferences, Williamsburg, April 1992.

[PUGH91] Pugh W., **"Uniform Techniques for Loop Optimisation"**, International Conference on Supercomputing, June 1991.

[RIBA90] Ribas H.B, **"Automatic Generation of Systolic Programs from Nested Loops"**, Ph.D. thesis, Carnegie-Mellon University, Computer Science Department, June 1990.

[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.

[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.

[WOLF89] Wolfe M.,**"Optimising Supercompilers for Supercomputers"**, Research Monograph in Parallel and Distributed Computing, Pitman, London, 1989.

[WOLF91a] Wolf M.E. and Lam M.,**"An Algorithmic Approach to Compound Loop Transformations"**, in *Advances in Languages and Compilers for*

Parallel Processing", eds Nicolau A., Gelernter D., Gross T. and Padua D., Pitman London, 1991.

[WOLF91b] Wolf M.E. and Lam M.,**"A Data Locality Optimizing Algorithm"**, ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1991.