Non-Singular Data Transformations: Definition, Validity and Applications

M.F.P. O'Boyle
Department of Computer Science
The University of Edinburgh
Edinburgh EH9 3JZ
United Kingdom
mob@dcs.ed.ac.uk

P.M.W. Knijnenburg
Department of Computer Science
Leiden University
Niels Bohrweg 1, 2333 CA Leiden
the Netherlands
peterk@cs.leidenuniv.nl

Abstract

This paper describes a unifying framework for non-singular data transformations. It shows that a wide class of existing transformations may be expressed in this framework, allowing compound transformations to be performed in one step. Validity conditions for such transformations are developed as is the form of the transformed program and data. Constructive algorithms to generate data transformations for different applications are described and applied to example programs. It is shown that they can have a significant impact on program performance and may be used in situations where traditional loop transformations are inappropriate.

Keywords: data structure transformations, compiler optimizations, spatial locality optimization, data alignment, page replication reduction

1 Introduction

Recent years have seen a great improvement in loop transformation theory. By using an affine representation of loops, several loop transformations have been incorporated into one single framework(1). Banerjee(2) has shown that loop interchange, reversal and skewing can be described as unimodular transformations of the iteration space. Li and Pingali(3) extend the class of transformations to non-singular matrices, thereby incorporating loop scaling. Furthermore, this framework has been extended to statement-wise transformations[4] and integrated with loop alignment[5]. From this treatment, general validity tests have been derived. This enables compound transformations to be applied in one go rather than step by step[6]. Kelly and Pugh[7] extend the class of transformations further and present a unifying framework based on schedules.

Loop transformations, however, are not the only program transformations of interest when optimising programs for parallel execution. In this paper we describe a unifying framework for data transformations, similar in approach to the framework used for loop transformations. Data transformations can be loosely defined as those transformations concerned with the layout, storage and access of array data, rather than reordering the program control flow. Data transformations have a number of applications, including global index reordering, data alignment, data partitioning, access normalisation, false sharing elimination and storage reduction. For instance, Temam *et al.*[8] use copying of data to eliminate cache conflicts. Jeremiassen and Eggers[9] explore the use of several data transformations to reduce false sharing in parallel C programs. Also, the relative alignment of arrays to reduce communication overhead has been examined[10,11,12]. Data transformations can be used in those cases where data dependence prevents loop transformations and where access to other arrays should not be altered. They are unaffected by control-flow structure and may be used in the presence of imperfectly nested loops and arbitrary control-flow graphs. For a good survey of loop and data transformations, see the overview paper by Kulkarni and Stumm[13].

Despite the wide applicability of such transformations, there has been, until recently, little work done to incorporate them into one single framework. This is probably due to the loop oriented approach to parallelisation and dependence testing as developed for shared memory computing. However, data oriented approaches, necessitated by distributed memory computing and NUMAs, highlight the need for a framework concerned with data rather than control structure.

Although there has been little work in developing a framework, it has recently received some attention. Cierniak and Li[14] investigate the effect of row and column-major storage of arrays combined with loop transformations. Many of the points raised here are discussed, such as data layout being independent of (flow) dependences and how loop and data transformations are related. Their work, however, is preliminary in that the only data transformation discussed is *linearisation*, the mapping of n-dimensional arrays to 1-dimensional arrays. However, loop transformations are full rank and dimension preserving. Hence there is an asymmetry in the treatment of loop and data transformations, weakening the unification theme in the paper. The transformations presented in this paper allow the row to column-major reordering described in the afore mentioned paper by Cierniak and Li[14] and describe a much larger set of transformations. Ju and Dietz[15] propose an algorithm for reducing cache coherence overhead by data layout and traditional loop transformations. However, the only data layouts they consider are row-major or column-major storage. Hence, in terms of the present paper, they restrict attention to array permutations. Anderson, Amarasinghe and Lam[16] propose that arrays are viewed as polyhedrons and that strip-mining and unimodular transformations are used for changing array layout to improve memory system performance, though they restrict their attention to permutations. In this paper, however, we provide a general framework for data transformations, encompassing unimodular transformations and show that this wide class of transformations is useful by developing optimisation algorithms that lead beyond simple permutations. Leung and Zahorjan[17] also propose an algebraic framework for data transformations. However, they restrict their attention to unimodular data transformations and improving data locality for uniprocessors. The present paper discusses the more general class of non-singular data transformations, the symmetry between non-singular loop and data transformations and presents algorithms that are also applicable to multiprocessor optimisation.

This paper makes the following contributions:

- it describes a unifying framework for a large class of data transformations.
- it shows how existing transformations may be expressed within the framework.
- it defines the application of data transformations on program structure and code generation.
- it describes the symmetry between non-singular loop and data transformations.
- it provides sample algorithms to constructively generate data transformation and show their applicability in program optimisation.

The paper is organised as follows. In the next section a motivating example is presented. Section 3 defines the notation used throughout the paper, while section 4 defines the form of transformations. Section 5 presents validity criteria and section 6 describes how data and loop transformations can be integrated. Section 7 describes three applications of data transformations and finally, section 8 concludes this paper.

2 Example

In this section, a simple example is used to illustrate the effect of data transformations on programs and why they are useful for compiler optimisation. Stride 1 access of arrays is important in architectures with some form of cache structure as it allows exploitation of spatial locality. Loop interchange is frequently used to ensure this for simple access functions. For general affine accesses, a non-singular loop transformation[3] to first normalise the access is required, followed by loop interchange[16]

Loop transformations that improve spatial locality on one access may adversely affect others and even then may not be valid due to data dependences. The example in column 1 of figure 1 illustrates both these limitations with loop transformations. Fortran stores arrays in a column-major fashion. This is highlighted by the vertical lines within the diagrams of arrays `A` and `B`. The accessed region for each array is shaded grey, with arrows representing the traversal order imposed by the loops. At present there is bad stride access on both arrays `A` and `B`. Loop interchange would improve the access to `A` but is illegal due to the dependence vector $[1, -1]$. Furthermore, while access to `A` would improve, the access stride of `B` would still be poor. Changing column to row-major storage, as described by Cierniak and Li(14) and shown in column 2, would improve stride access to `A` but would again be of no use in improving access to `B`. This can be readily seen in the diagram of column 2.

Non-singular data transformations can be used as they are not restricted by data dependences, and they do not affect other array accesses within that loop nest. Although they must be globally propagated, this impact can be reduced as described in section 6.

Applying the appropriate data transformations gives the new program in column 3. Since arrays in Fortran need to be rectangular, the new `B` array is larger than the original. We have to take the bounding box to generate the new array bounds. The dark region represents unaccessed elements introduced by the need to restrict arrays to rectangular form. The skewed access region is mapped to a square one, and stride 1 access is now possible for access to both arrays. The form of the non-singular data transformation, how it is deduced, applied and its impact on actual machine performance is described later in section 7.1.

3 Notation

In this section we briefly describe the notation used to develop the data transformation framework. It is based on a linear algebraic representation of program constructs.

3.1 Iteration Spaces

The iterators in a Fortran program surrounding any statement can be represented as an $m \times 1$ column vector

$$J = [j_1, j_2, \ldots, j_m]^T$$

where m is the number of enclosing loops or iterators. The loop range or affine bounds of the iterators can be described by a system of inequalities defining the *polyhedron*

$$AJ \leq b \tag{1}$$

where A is a $(\ell \times m)$ integer matrix and b an $(\ell \times 1)$ vector. The integer values taken on by J define the *iteration space* of the iterators.

3.2 Index Space

The data storage of an array `A` can also be viewed as a rectangular polyhedron. We introduce *formal indices* $\mathcal{I}$ for the array to describe the array index space

$$\mathcal{I} = [i_1, i_2, \ldots, i_n]^T$$

where n is the dimension of array `A`. The formal indices have a certain range which describe the size of the array, or *index space*, as follows:

$$\lambda \leq \mathcal{I} \leq \mu \tag{2}$$

where $\lambda = [\lambda_1, \ldots, \lambda_n]^T$ and $\mu = [\mu_1, \ldots, \mu_n]^T$ are $n \times 1$ vectors. Due to the restrictions on the bounds of arrays in Fortran (or other languages), only *constant vectors* are required to describe array bounds.

3.3 Array Accesses

The subscripts in a reference to an array `A` represent a function that maps the values of the iteration space to the index space. If J is the iteration vector, we assume in this paper that these subscripts can be written as an affine mapping that has the following form

$$\mathcal{U}J + u$$

where $\mathcal{U}$ is a $n \times m$ matrix and u is a $n \times 1$ vector.

Thus the access to `B` in figure 1, `B(i+j,j)`, is denoted by

$$\mathcal{U}J + u = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The subscripts in a reference to an array are normally referred to by the *access matrix/vector pair* $(\mathcal{U}, u)$.

4 Data Transformations

This section describes an algebraic framework for data transformations and their application. We discuss the range of transformations contained with the class of non-singular matrices. Data transformations for an array update all access to that array and also change the way the data is stored.

Definition *A* data transformation $(\mathcal{A}, a)$ *for an array* `A` *of dimension* n *consists of a (non-singular)* $n \times n$ *matrix* $\mathcal{A}$*, and a* $n \times 1$ *vector* a

When the shift vector a equals $\vec{0}$, we write $(\mathcal{A}, a)$ simply as $\mathcal{A}$. Given an access matrix/vector pair $(\mathcal{U}, u)$ for an array `A` occurring in the program, application of a data transformation $(\mathcal{A}, a)$ to this access results in a new access $(\mathcal{U}', u')$ given by

$$(\mathcal{U}', u') = (\mathcal{A}\mathcal{U}, \mathcal{A}u + a) \tag{3}$$

$\mathcal{A}$ is a *left-hand* acting transformation, in contrast to loop transformations which are *right-hand* acting. Unlike loop transformations, which are local to a loop nest but affect all contained references, data transformations affect just one array but affect all accesses to the array globally. Thus the transformation $(\mathcal{A}, a)$ must be applied to every occurrence of array `A` throughout the program. In addition, the actual storage of the array must be updated. Thus the index space $\mathcal{I}$ of array `A` must also be updated.

The application of $(\mathcal{A}, a)$ is straightforward: all accesses are updated according to equation (3) and the index space is updated as described in section 4.1 below. The validity of such transformations is discussed in section 5. No change need be made to the loop or control structure of the program.

In the example in figure 1 the following transformations are applied to the arrays in column 1 to give the transformed form shown in column 3.

$$\mathcal{A}_{\mathtt{A}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \text{ and } a_{\mathtt{A}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathcal{A}_{\mathtt{B}} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \text{ and } a_{\mathtt{B}} = \begin{bmatrix} 0 \\ 4 \end{bmatrix} \tag{4}$$

Applying both transformations, we obtain

$$\mathcal{U}'_{\mathtt{A}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and

$$u'_{\mathtt{A}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Hence `A(i,j)` is mapped to `A(j,i)`.

$$\mathcal{U}'_{\mathtt{B}} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and

$$u'_{\mathtt{B}} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

Hence `B(i+j,j)` is mapped to `B(j,i+4)`.

In section 7.1, a polynomial algorithm is presented which determines the correct transformation to obtain stride-1 access.

4.1 Transforming the Index Space

As well as transforming the accesses to an array, the actual storage for that array needs to be modified. The index space is transformed as follows.

Suppose $(\mathcal{A}, a)$ is a data transformation for an array `A` with index space given by

$$\lambda \leq \mathcal{I} \leq \mu \tag{5}$$

Then we need to compute a new index space

$$\lambda' \leq \mathcal{I}' \leq \mu'$$

The new indices $\mathcal{I}'$ are related to the old indices $\mathcal{I}$ as

$$\mathcal{I}' = \mathcal{A}\mathcal{I} + a$$

Hence

$$\mathcal{I} = \mathcal{A}^{-1}\mathcal{I}' - \mathcal{A}^{-1}a \tag{6}$$

Substituting equation 6 in equation (5), we obtain

$$\lambda + \mathcal{A}^{-1}a \leq \mathcal{A}^{-1}\mathcal{I}' \leq \mu + \mathcal{A}^{-1}a$$

since both λ and μ are constant vectors.

If $\mathcal{A}$ is solely a permutation matrix, this is sufficient: $\mathcal{A}^{-1}\mathcal{I}'$ is a permutation of the vector $\mathcal{I}$. Hence we can easily obtain $\lambda' \leq \mathcal{I}' \leq \mu'$. However, in the general case further work is necessary: we must solve for $\mathcal{I}'$ and take the bounding box of the transformed index space, as arrays in Fortran have to be rectangular.

For example, consider the access to array `B` in figure 1. $\mathcal{A}_{\mathtt{B}}$ and $a_{\mathtt{B}}$ are given in equation (4) and

$$\mathcal{A}_{\mathtt{B}}^{-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

This gives the new bounds

$$\begin{array}{rcccl} 1 & \leq & i'_1 & \leq & 4 \\ 5 - i'_1 & \leq & i'_2 & \leq & 12 - i'_1 \end{array}$$

The new index space is not rectangular. We need to project on index i_2 to obtain its minimum and maximum values. This means that we have to substitute the maximum value of i'_1 in the lower bound for i'_2, and the minimum value of i'_1 in the upperbound for i'_2. This amounts to turning the system of inequalities "upside down" and applying Fourier-Motzkin elimination to this new system. Hence we obtain the new bounds `B(1:4,1:11)`.

4.2 Forms of $\mathcal{A}$

Within this paper we restrict our attention to transformations $\mathcal{A}$ that are non-singular. This is analogous to concentrating on non-singular loop transformations. It follows that $\mathcal{A}$ always has an inverse though this inverse is not necessarily integer. For the purposes of showing the usefulness of such transformations, we can subdivide them and illustrate their effect on the program segment in figure 2. Figure 3 summarises the forms of the transformations considered.

Unimodular Unimodular matrices have been extensively studied and their properties will not be expounded here. They encompass *permutation*, *reversal* and *skewing* of the access matrix. They have the property that the convex hull of the transformed space will have the same number of points as the original space. However, as arrays must be rectangular in Fortran, additional storage will be needed to "pad" skewed arrays. Uses of such transformations include global index reordering and restricted forms of data alignment. Column 1 of figure 3 summarises the effect of unimodular transformations.

Integer determinant larger than 1 These matrices are a generalisation of unimodular matrices incorporating *scaling* or *expansion*. One use would be to expand small arrays so that, when partitioned over a large template, the array is evenly distributed across all the processors. Scaling by cache-line size may also help alleviate false sharing. Scaling the first index by a factor of 2 is shown in figure 3.

Rational determinant Rational non-singular transformations allow *compression* of the access and hence of data space. This is useful when space is critical, especially for reducing the sizes of "windows"(19) in caches. Regular sparse accesses can be compressed so as to reduce the overall data allocation for that array. The example in the table shows compression in the first index by a factor of 2.

Shift This is a very simple transformation where the arrays are *shifted* by a constant value. This can prove useful when concerned with aligning data on coherency unit boundaries or to remove problems associated with cache hardware mapping. The effect of a shift of -1 is shown in figure 3.

4.3 Existing Transformations

While there has been much effort to show that loop transformations can be unified, there has been little work with regard to data transformations. Some existing transformations can be considered as special cases of non-singular data transformations.

Global index reordering of a particular array corresponds to a simple permutation data transformation matrix.

Data Partitioning is in fact a rank deficient data transformation matrix[1] whose non-null rows correspond to partitioned array dimensions.

[1] the legality of which is not considered in this paper

Data Alignment reorientates data with respect to a template. Non-singular data transformations cover all the linear alignment directives used in HPF.

All data layout schemes can be described as data transformations. Thus transformations from a wide range of applications can be viewed within one framework.

5 Validity

This section describes when data transformations can be legally applied. Data storage transformations can be considered to be an advanced form of renaming variables. Rather than just renaming a variable, e.g., x becomes y, we rename the index values as well. For example, `A(1,2)` will now become `B(3,4)`. Renaming or changing the symbol of a variable is legal as long as the renaming occurs for all occurrences of the variable and there does not exist a variable with the same name. The legality tests for $(\mathcal{A}, a)$ are similar.

First of all, we must guarantee that all references are updated and secondly, that no clashes between two references occur. For example, if elements `A(1,2)` and `A(3,4)` are both accessed, they should not both be stored at point `B(1,2)`. We must ensure that no values are lost in this new storage scheme. As an additional check we should ensure that all data dependences are preserved. In fact, it will be shown that non-singular data transformations always preserve data dependence. Fortran allows arrays to change dimension throughout a program especially when passing across subroutine boundaries. We assume that such a change in dimension does not occur in the program under consideration.

Let R be the total number of references to some array `A`. Let, for $1 \leq i \leq R$, the ith reference to `A` be given by access matrix $\mathcal{U}_i$ and vector u_i. Let $(\mathcal{A}, a)$ be a data transformation for `A`. All references are updated, that is, for all $1 \leq i \leq R$,

$$\mathcal{U}_i \mapsto \mathcal{A}\mathcal{U}_i \text{ and } u_i \mapsto \mathcal{A}u_i + a$$

The first requirement we impose on the data transformation matrix $\mathcal{A}$ is that after application array index functions map iterators to integer values. That is, for every point J in the iteration space,

$$\mathcal{A}\mathcal{U}_i J + \mathcal{A}u_i + a \in Z^{\mathcal{R}(\mathcal{A})}$$

where $\mathcal{R}(\mathcal{A})$ denotes the number of rows in $\mathcal{A}$.

The second requirement on $\mathcal{A}$ is that all distinct elements referenced in the original program are mapped to distinct points. Let $(\mathcal{U}_i, u_i)$ be an access matrix/vector pair occurring in a loop J_i and let $(\mathcal{U}_k, u_k)$ be an access matrix/vector pair occurring in another loop J_k. Then we require that for every iteration J_i and J_k in these loops,

$$\begin{aligned}&\mathcal{U}_i J_i + u_i = \mathcal{U}_k J_k + u_k \\ &\quad \text{iff} \quad \mathcal{A}\mathcal{U}_i J_i + \mathcal{A}u_i + a = \mathcal{A}\mathcal{U}_k J_k + \mathcal{A}u_k + a\end{aligned}$$

Note that we do not require that every element in the array is mapped to a distinct element, but rather that every element *referenced in the program* is mapped to a distinct element. This distinction is important since we want to consider storage compression for arrays that are only partially filled. Since in this paper we are only considering data transformation matrices that are non-singular and hence bijective, this requirement is automatically fulfilled. In future research we hope to consider singular data transformation matrices also. In this case, this requirement is not trivial and needs some careful consideration. A desirable application of a singular data transformation is given by transforming a high dimensional temporary array into a low dimensional one, where we reuse storage. A second application is given by strip-mining an array, in which case a low dimensional array is transformed into a high dimensional one. In fact, Anderson, Amarasinghe and Lam[16] have shown that strip-mining (a singular data transformation) together with permutation (a non-singular data transformation) can be used for obtaining data layout automatically. This result shows that a general framework of singular data transformations, encompassing non-singular transformations, is important for obtaining efficient code for parallel machines.

Next we consider the validity of non-singular data transformations. First, we state a lemma the proof of which is trivial since $\mathcal{A}$ is bijective.

Lemma *Let* $(\mathcal{A}, a)$ *be a non-singular data transformation for some array* `A`*. Suppose* $(\mathcal{U}_1, u_1)$ *and* $(\mathcal{U}_2, u_2)$ *be two access matrix/vector pairs for* `A` *in some loop. Then there exists a dependence from iteration* J_1 *to iteration* J_2 *caused by these accesses before the application of* $(\mathcal{A}, a)$ *to* `A` *if and only if there exists a dependence from* J_1 *to* J_2 *after the application of* $(\mathcal{A}, a)$ *to* `A`*.*

This Lemma immediately leads to the following proposition.

Proposition *The application of a non-singular data transformation is always valid.*

Note that in case of singular data transformations the statement in the above proposition does not hold. For example, if we want to check that transforming a high dimensional temporary array into a low dimensional one is valid, we need to check that every assignment to an element of the second array is not live anymore before we reassign that element. In general, sophisticated value based data dependence analysis is required for deciding the validity of singular data transformations. In future work we hope to give a precise characterisation of the validity requirements for these data transformations.

6 Unifying Loop and Data Transformations

There has been references to and parallels drawn with loop transformations within this paper. In fact, there is a strong symmetry between data and loop transformations. This can be readily seen if we look at the effect of either transformation from an *access* rather than a loop or data storage point of view.

First, let us recapitulate some of the issues involved. For the sake of clarity, we ignore shifts in the discussion below. Given a loop L and an array A, we denote the iteration space of L by $S(L)$ and the index space of A by $S(\mathtt{A})$. Then an access can be seen as mapping $(\mathcal{U}, u) : S(L) \to S(\mathtt{A})$. If we apply a transformation $\mathcal{T}$ to L we obtain a new loop L', and we can view the transformation as a mapping $\mathcal{T} : S(L) \to S(L')$. On the other hand, if we apply a transformation $\mathcal{A}$ to A we obtain a new array A', and we can view the transformation $\mathcal{A} : S(\mathtt{A}) \to S(\mathtt{A'})$. In order to obtain the bounds for L' and A', we need to use $\mathcal{T}^{-1}$ and $\mathcal{A}^{-1}$, respectively, for expressing the new iterators or indices in terms of the old iterators and indices, respectively. However, for transforming accesses, in case of a loop transformation we need to go from $S(L')$ to $S(L)$, using the inverse of the loop transformation, $\mathcal{T}^{-1}$, and then to $S(\mathtt{A})$, using the access matrix. Hence $\mathcal{T}^{-1}$ is a *right hand operating* transformation. In case of a data transformation, we need to go from $S(L)$ to $S(\mathtt{A})$, using the access matrix, and then to $S(\mathtt{A}')$, using the data transformation $\mathcal{A}$. So $\mathcal{A}$ is a *left hand operating* transformation. Thus a loop and a data transformation together have the following effect upon a data access matrix:

$$\mathcal{U} \mapsto \mathcal{A}\mathcal{U}\mathcal{T}^{-1}$$

Frequently, both transformations can have the same effect on one particular access. In such a situation the choice of a data or loop transformation will depend on legality criteria being satisfied and any side effect on the rest of the program.

One useful outcome of these observations is that the *global* impact of data transformations can frequently be resolved *locally* by loop transformations. Consider the example in figure 1. The initialisation of B is not shown here. Let us assume it is of the form shown in column 1 of figure 4. On propagating the alignment transformation the initialisation code will be transformed into that of column 2. Unfortunately, this code exhibits very poor access stride, so we have shifted the problem from one place to another. However, the access stride can be improved locally by a loop transformation ($\mathcal{T} = \mathcal{U}^{-1}\mathcal{A}\mathcal{U}$ and $t = a$) to give the code in column 3. In this case there are no problems due to data dependence and conflicting access requirements. However, it may not be possible in general to generate a transformation that is both legal and satisfies all access requirements. In such instances, a local algorithm that attempts to find the best loop ordering given a particular data layout and access matrices can be applied. Bik and Knijnenburg[(21)] describe an algorithm that constructs a unimodular transformation that is guaranteed to be legal and maximises spatial locality for a particular loop nest.

Both loop and data transformations can improve spatial locality. This is due to the form of the access matrix required (see section 7.1). For there to be temporal locality, however, the access matrix must be invariant of one or more of the iterators i.e. one of the iterators must not appear in the array subscripts. More formally, the access matrix must be column rank deficient and the null space of the access matrix must correspond to the innermost loop. For this to take place, column reordering of the access matrix is required which can only be achieved by right-hand acting transformations, i.e. loop transformations. As data transformations are left-hand acting and can only reorder the rows of an access matrix, they are incapable of exploiting temporal locality. There are cases, however, where spatial locality can only be achieved using data layout transformations (see section 7.1.1) and therefore loop and data transformations should be considered as complimentary. By considering non-singular loop and data transformations at the same time, we can apply both elementary row and column operations to an access. Further investigation of the relationship between loop and data transformations is beyond the scope of this paper and will be the subject of future work.

7 Applications

In this section the use of data transformations in a number of applications is outlined. Each subsection will describe the optimisation goal and a constructive algorithm to generate the appropriate transformation. Experimental results on the SGI Challenge, Meiko CS-2 and KSR-1 are presented to illustrate the usefulness of data transformations by comparing them to other techniques.

7.1 Spatial Locality

Stride-1 access of arrays is important in architectures with some form of cache structure since this allows exploitation of spatial locality. Traditionally, loop transformations are used to optimize for spatial locality by reordering accesses to a matrix. There has been extensive work in this area(18,19,21,22) which use loop reordering to exploit spatial locality . In this section, we show that the same goal can be reached by reordering array layout using data transformations. Such an approach is useful when loop reordering is not possible either due to legality constraints or, as for instance in the case of the second example program in this section `vptst`, there is only one surrounding loop nest.

Assuming a column-major ordering of arrays, the innermost iterator should ideally access only the first index, if any. Accessing the second or subsequent index should be avoided as it will destroy spatial locality. Assuming that the dimension of the referenced array is n and that the nesting depth of the surrounding loops is m, the desired structure of an access matrix $\mathcal{U}$ for spatial locality is therefore

$$\mathcal{U} = \begin{bmatrix} y & 1 & o_1 \\ Y & o_2 & O \end{bmatrix} \tag{7}$$

where y is a $1 \times \ell$ row matrix, Y an $(n-1) \times \ell$ matrix, o_1 an $1 \times (m-\ell-1)$ null row matrix, o_2 an $(n-1) \times 1$ null column matrix and O a $(n-1) \times (m-\ell-1)$ null matrix. The 1 is the right-most non-null entry in row one and corresponds to the most inner iterator referenced by the first index. For good spatial locality, that iterator must not be referred to by any other index, i.e., there should be just one column entry for that iterator. This implies that o_2 must be null. More than one entry will imply diagonal or skewed access.

For example, assuming an iterator ordering `k,j,i`, then `A(i,j+k)` and `B(i+j,j)` are good accesses, while `C(i,i+j)` is not. This can be seen from their access matrices:

$$\mathcal{U}_{\mathtt{A}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \mathcal{U}_{\mathtt{B}} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \mathcal{U}_{\mathtt{C}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

where only the access matrix of `C` does not have the desired structure. In both `A(i,j+k)` and `B(i+j,j)` the innermost iterator `i` is only referenced in the first index while the access `C(i,i+j)` refers to `i` in both indices.

7.1.1 Algorithm

To achieve the desired structure of the access matrix requires reordering of the matrix by elementary row or column operations which directly correspond to left and right hand transformations, i.e., data and loop transformations. The algorithm below describes a constructive technique for optimising spatial locality using data transformations. It attempts to create a matrix with an anti-diagonal structure using rowwise operations. Such a matrix has the desired property of equation (7).

Algorithm

1. Let T be the reverse permutation of identity, where $t_{ij} = 1$ if $j - i = n - 1$, 0 otherwise.
2. Apply rowwise operations $\mathcal{A}$ to reduce $\mathcal{U}T$ to integer row echelon form.
3. Determine lower bounds λ_k and upper bounds μ_k of each index i_k after applying $(\mathcal{A}, 0)$, using Fourier-Motzkin elimination.
4. Let $a_k = -\lambda_k + 1$ for all k, yielding the shift vector a. This shift vector guarantees that all lower bounds become 1, as is the default in Fortran.
5. Apply the data transformation $(\mathcal{A}, a)$ to the array.

To illustrate this algorithm consider the access to `B` in figure 1 and assume that the array `B` is of size $(2n \times n)$. Applying the algorithm, we obtain the initial data transformation

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

This gives the following lower and upper bounds after applying $\mathcal{A}$:

$$1 \le i_1 \le n \quad \text{and} \quad -n + 1 \le i_2 \le 2n - 1$$

The new lower bounds of the array are negative after applying $\mathcal{A}$, so the shift vector a is applied to restore the indices to positive values, where $a = [0, n]^T$. The updated program for $n = 4$ is given in column 3 of figure 1.

The algorithm described above is applicable to more complex references such as `A(i+j+k,i+k,i-j+k)`. We have that `A(i+j+k,i+k,i-j+k)` will be mapped to `A(2*i+k,j,0)`, where the innermost iterator `k` has stride 1 access on `A`. Although such references are unlikely to be as complex in practice, it demonstrates the generality of the above scheme.

Rank Deficient Accesses If data transformations to improve spatial locality on a particular access are restricted to permutations then they can be seen as a formulation of global index reordering or a change from row to column-major storage. In the general case for arbitrary affine accesses they can be seen as the data dual of access normalisation[(18)]. Data transformations can be shown to be strictly more powerful than loop transformations in enhancing spatial locality, as there exists accesses, namely rank deficient accesses, where no loop transformation can improve its spatial locality, but a data transformation can.

Rank deficient accesses, where two rows are dependent, cannot be removed by column operations (loop transformations). This causes a problem if the two dependent rows contain an entry in the innermost iterator column.

For example, consider the access `A(i+j,i+j)`. A loop transformation may map

$$\texttt{A(i+j,i+j)} \mapsto \texttt{A(i',i')}$$

while a data transformation may have the following effect:

$$\texttt{A(i+j,i+j)} \mapsto \texttt{A'(i+j,0)}$$

In this case data transformations can remove diagonal access causing references which cannot be removed by loop transformations. Column/row-major reordering is also unable to remove the poor stride access in rank deficient accesses.

7.1.2 Results

To illustrate the benefit of the above algorithm the three versions of the program shown in figure 1 were executed and the results shown in figure 5 on one processor of an SGI Challenge. Parallel versions could not be evaluated due to the serialising data dependences within the program[2]. Changing from column to row-major layout, as available using the method described by Cierniak and Li[(14)], provides a small improvement on large data sizes as there is improved access to array `A`. The poor stride on `B`, however, prevents significant performance improvement. The data transformation approach, based on the algorithm in section 7.1.1, improves the stride access to both arrays and shows up to a factor of 5 improvement over the existing methods. This, however, is at the cost of an increase in the amount of storage required which may adversely

[2] The figure shows time against varying data sizes for three layouts, rather then temporal performance (1/Time) against processor number which is used in the other (parallel) applications.

affect locality. In general, there will be a trade-off especially in those instances where the working-set just fits in cache.

Although useful as an illustrative example, the program in figure 1 is synthetic and not representive of general programs. In order to show the usefulness of this transformation for larger applications, it was added as a post data partitioning phase in the MARS compiler[23] and applied to the Specfp92 benchmark `vptst`. The original parallelised code ("Original") was compared to the new version ("Data") incorporating the spatial locality algorithm. Both programs were executed on an SGI Origin 2000, with up to 32 processors, and the temporal performance shown in figure 6. The algorithm gives a performance improvement especially for larger values of p. In the original code, there is poor spatial locality, with the innermost `k` loop striding through the second dimension of all arrays. By reordering the array layout, stride-1 access throughout the program is achieved, without the need for any loop reordering and hence there is no trade off between loop nests to consider. If the method described by Cierniak and Li[14] were applied, loop reordering for spatial locality would be first applied followed, if necessary, by data layout modification. While this would improve spatial locality in two of the main 2-dimensional loop nests, it would not improve spatial locality in the remaining five 1-dimensional loop nests. As the number of processors increase, the overhead of poor spatial locality in these loop nests becomes a significant overhead, as shown by the graph in figure 6 labelled ("Loop").

7.2 Data Alignment

The relative storage, or alignment, of array data in distributed memory critically determines the amount of communication overhead. After mapping data to processors, alignment determines which portion of one array will be in the same processor as another part of a second array. Obviously, if this is done 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. This is an extensively studied optimisation where the majority of research has been aimed at finding a globally reasonable array alignment in the presence of conflicting requirements[10,11,12,24]. Bau et al.[25] propose a linear algebraic framework for finding alignment mappings, but do not consider restructuring arrays to obtain an optimal alignment. O'Boyle and Hedayat[12] present an algorithm, based on row echelon form, which generates a data transformation. This algorithm, provably, maximises pair-wise alignment by restructuring array layout.

7.2.1 Algorithm

The algorithm attempts to maximise the number of identical rows of a pair of access matrices. If partitioned along such a row/index, no communications will be incurred in the parallel program. Let $\mathcal{U}$ and $\mathcal{V}$ be the references that we want to align. The data transformation matrix $\mathcal{A}$ is chosen as to maximize the number of perfectly aligned subscripts or rows of $\mathcal{A}\hat{\mathcal{U}}$ and $\hat{\mathcal{V}}$, where $\hat{\mathcal{U}}$ and $\hat{\mathcal{V}}$ denote expanded array accesses padded with rows and columns of zeroes to make them of equal size. In order to determine the form of $\mathcal{A}$ such that the number of identical rows in $\mathcal{A}\hat{\mathcal{U}}$ and $\hat{\mathcal{V}}$ is maximised, it is necessary to find a solution for as many rows $\mathcal{V}_i$ of $\mathcal{V}$ to the following equation:

$$\hat{\mathcal{U}}^T y_i^T = \hat{\mathcal{V}_i}^T \tag{8}$$

We then use the solutions y_i to compose $\mathcal{A}$. To solve for y_i^T, concatenate $\hat{\mathcal{U}}^T$ and $\hat{\mathcal{V}}^T$ and reduce $\hat{\mathcal{U}}^T$ by elementary row operations to row echelon form, whilst simultaneously performing the same operations on $\hat{\mathcal{V}}^T$. This algorithm is detailed below. The algorithm reduces an $\hat{\mathcal{U}}^T$ to row echelon form, while simultaneously carrying the same elementary operations on $\hat{\mathcal{V}}^T$ and the identity matrix I_M. It has been shown[12] that a column y_i^T of the reduced matrix $\hat{\mathcal{V}}^T$ is eligible to form a row of $\mathcal{A}$ if it is perpendicular to the null space of $\mathcal{U}$ and it does not cause $\mathcal{A}$ to be rank deficient, i.e., it is independent of existing rows of $\mathcal{A}$. Such tests are basic operations in linear algebra and are not described further here. If y_i^T is not eligible, a suitable candidate from I_M is chosen.

Algorithm

1. Reduce
$$\left[\begin{array}{ccc} \hat{\mathcal{U}}^T & \hat{\mathcal{V}}^T & \multirow{2}{*}{I_M} \\ O & O & \end{array}\right] \tag{9}$$
to row echelon form.

2. Reorder by elementary row operations such that leading non-zeros lie on the diagonal.

3. For each column i of the reduced $\hat{\mathcal{V}}$ matrix, if this column is perpendicular to the null space and independent, then place this row in row i of $\mathcal{A}$. Otherwise, select an independent row of I_M and place that row in row i of $\mathcal{A}$.

Although in general $\mathcal{A}$ may be a matrix with rational elements, $\mathcal{A}\hat{\mathcal{U}}$ is always integer. Thus, for pairwise alignment, the row echelon form gives the required alignment transformation.

To illustrate this algorithm, consider the example from O'Boyle and Hedayat[12] shown in figure 7(a). In this example, we want to align the references to arrays `B` and `C` with the reference to `A`. Applying the algorithm above gives

$$\mathcal{A}_{\mathtt{B}} = \begin{bmatrix} -1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } a_{\mathtt{B}} = \begin{bmatrix} -3 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathcal{A}_{\mathtt{C}} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \text{ and } a_{\mathtt{C}} = \begin{bmatrix} 0 \\ -n-1 \\ 0 \end{bmatrix}$$

When applied to the array accesses, the new program in figure 7(b) is derived, where all the first indices are aligned. Partitioning along the first index will guarantee that there is no non-local access in the parallel program.

7.2.2 Results

To demonstrate the impact of this transformation, both versions of the example program, partitioned along the first index, were executed on the Meiko CS-2 and the temporal performance shown in figure 8. The low level elan communication library was used, providing highly efficient access to remote memory. In the example program there is $O(\frac{n^3}{p})$ work per processor where p is the number of processors and $O(n^2)$ remote memory access for each processor. From this analysis the overhead of communication will have greatest impact when p is large relative to n, i.e., small granularities. The experimental results reflect this. The application of the data transformation improves temporal performance by up to a factor of 3 in the case of smallest granularity ($n = p = 64$). In implementations with less efficient communication, the overhead can increase by an order of magnitude.

In order to show the usefulness of this transformation for larger applications, it was applied to the Specfp92 benchmark `chotst` using the MARS compiler(23). program were generated, one with global data alignment ("Align"), one without ("Original"), both of which were executed on an SGI Origin 2000 and the temporal performance shown in figure 9. As the number of processors increases, so does the improvement due to data alignment, up to a factor of 2 in the case of $p = 32$. The reason is that data alignment not only reduces communication but can also eliminate cross-processor data dependences(26). By reducing communication and therefore improving locality, we can reduce the amount of synchronisation inserted to honour cross-processor dependences and hence increase parallelism. When data alignment is not applied to `chotst`, one barrier is inserted to cover all the cross-processor dependences. The relative cost of this barrier synchronisation will increase as the number of processors increases as can be seen in figure 9. Finally, data alignment has the overall effect of making common rows of access matrices similar. Thus any loop order that is beneficial for one access, in terms of spatial locality, will tend to beneficial for others. This is indeed the case for `chotst` where no loop reordering is required once data transformations have been applied.

7.3 Page Replication

In this section we will look at page replication in Cache-only Memory Architectures (COMAs). We consider an SPMD execution model with the owner computes rule, where data is partitioned across the processors such that each processor has part of the data resident Note that different parts of a single page may be owned by distinct processors.

Partitioning of arrays along the first index can lead to page replication in SVM systems. Let the dimension of the first array index be n, the number of processors p and the size of the coherence unit[3] c. Difficulties arise in COMAs when $\frac{n}{p} > c$ for 2 or higher dimensional arrays, if partitioned on the first index. This means that every processor is allocated a collection of rows of the array. Data is allocated column-wise to pages and each time an element is referenced, the page must be first allocated in memory by the Operating System before it may be accessed. Partitioning on the first index of a 2-dimensional array implies that potentially every column will be accessed by every processor and thus all pages must be allocated in each processors attraction memory. For large arrays, there will be extensive cache conflict and performance degradation. However, in those cases where parallelism is only available in the first array index, existing loop based approaches cannot alleviate this problem without loss of parallelism.

7.3.1 Algorithm

Consider the program fragment from the ADI example[20] shown in figure 10(a). Here parallelism is available only in the first index. Interchanging loops will not change this but will actually lead to worse stride access. If, however, the storage of all arrays were transposed, leading to a columnwise data partitioning throughout the program, then page replication is avoided. However, this is at the expense of stride access as the innermost iterator now strides through the second array index. This is easily rectified by loop interchange and is shown in figure 10(b).

The algorithm below performs a data storage transformation program wide if partitioning is upon the first index. It takes as input a partitioning matrix $\mathcal{P}$. With each partitioning matrix $\mathcal{P}$ a serialising matrix $\mathcal{S}$ is associated such that

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

Given the formal indices $\mathcal{I}$, $\mathcal{PI}$ and $\mathcal{SI}$ define those indices to be partitioned and serialised, respectively. A full rank $\mathcal{P}$ implies that all the indices will be partitioned; a null matrix means the array is not distributed.

The partitioning matrix $\mathcal{P}$ is checked to see if it partitions the first index and if so, whether there are other indices which could be partitioned instead. This can only be true if $\mathcal{P}$ is not full rank. If successful, the partitioning matrix, all array accesses and data storage are updated. In the algorithm $E_{1,j}$ denotes the identity matrix I with columns 1 and j interchanged. Its application swaps the first row of a matrix with the jth, hence the jth row will now be partitioned rather than the first.

The algorithm can now informally be stated as follows.

[3] Cache line, page or, in the case of the KSR, sub-page.

Algorithm

if $\mathcal{P}$ is not full rank **and** $\mathcal{PI}[1] \neq 0$
then
  Let j be the first null row of $\mathcal{P}$
  Let $\mathcal{A} = E_{1,j}$
  Apply $(\mathcal{A}, 0)$ to all accesses
  Let $\mathcal{P}' = \mathcal{APA}^{-1}$
endif

In the example given in figure 10(a), we have before transformation the following value of $\mathcal{P}$ and access matrix/vector pairs, where we number accesses from left to right.

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

$$(\mathcal{U}_1, u_1) = \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

$$(\mathcal{U}_2, u_2) = \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right)$$

$$(\mathcal{U}_3, u_3) = \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

Given the formal indices $\mathcal{I}$, $\mathcal{PI}$ defines those indices which will be partitioned. In this case, $\mathcal{PI}[1]$, the first element of $\mathcal{PI}$, equals i_1. Hence the first index is to be partitioned. $\mathcal{P}$ is not full rank as one of its rows is null; it is a 2-dimensional matrix of rank 1. If $\mathcal{P}$ were full rank, this would imply that all indices were to be partitioned and there would be no point in reordering indices. In the case of a rank deficient partition matrix, other indices can be chosen to partition.

As there is only one null row in $\mathcal{P}$, $j = 2$ and $\mathcal{A}$ is readily found:

$$\mathcal{A} = E_{1,2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Application of the data transformation $(\mathcal{A}, 0)$ gives the new accesses

$$(\mathcal{U}_1', u_1') = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

$$(\mathcal{U}_2', u_2') = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right)$$

$$(\mathcal{U}_3', u_3') = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

The new index space (storage) for both arrays is

$$\mathcal{I}' = [i'_1, i'_2]^T = [i_2, i_1]^T$$

and

$$\begin{array}{ccccc} 1 & \leq & i'_1 & \leq & 400 \\ 1 & \leq & i'_2 & \leq & 200 \end{array}$$

Finally, the new partitioning matrix $\mathcal{P}'$ is formed where

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

Therefore, the new index to partition on is given by

$$\mathcal{P}'\mathcal{I}' = \begin{bmatrix} 0 \\ i'_2 \end{bmatrix}$$

which is the second index, avoiding the problem of false sharing and page replication[4]. After loop interchange is performed, the program in figure 10(b) is found.

This algorithm was implemented in the MARS compiler(23) after the data partitioning decision is made. Although it may also be necessary to update loop structures using loop interchange in cases where bad stride occurs, it is not necessary to include this in the present transformation, as there already is an existing module in MARS which reorders loop for locality after partitioning and mapping has occurred.

7.3.2 Results

To demonstrate the importance of this algorithm we compared it with the existing MARS compiler and KAP, an existing commercial auto-parallelising compiler. We applied KAP, MARS and a version of the MARS compiler in which this transformation is implemented to the ADI code fragment for the data sizes $n = 256$, 512, 1024 and 2048 and ran the resulting parallel programs on the KSR on 1, 2, 4, 8, 16 and 32 processors. The results are shown in figure 11. Applying the data transformation can give dramatic performance improvements by up to a factor of 50 improvement over the old version of MARS and a factor of up to 60 over KAP. As the data size increase, so does the amount of page replication and the number of cache conflicts in the attraction memory leading to severe performance degradation in the case of $n = 2048$. It is here that the transformation gives the greatest improvement.

8 Conclusion

In this paper we have described the use of non-singular data transformations in optimising program performance. We have described their impact on program

[4]The local node programs after partitioning are not shown here.

structure and developed validity tests for their application. Constructive algorithms have been developed to generate transformations that can significantly improve program performance outperforming existing techniques. Non-singular data transformations unify existing transformations and are strongly connected with non-singular loop transformations. As data transformations are global in nature, they may adversely affect certain sections of the program. By considering the duality between loop and data transformations it has been shown that loop transformations may be constructed to locally reverse the effect of a data transformation. This, however, is not always possible due to, for instance, legality constraints on loop transformations. Future work will consider the combination of loop and data transformations in global optimisation. and extend the class of transformations examined beyond non-singular matrices. Future work will also consider the usefulness of dynamic data layout transformations. They may be useful in instances where there is severe load imbalance or when there is a lack of parallelism due to data layout. Such work, however, will have to consider the trade-off of the overhead of communication against the increased load balance or parallelism.

References

1. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley (1996).

2. U. Banerjee, Unimodular Transformations of Double Loops, *Proc. of 3rd Workshop on Programming Languages and Compilers for Parallel Computing* (August 1990).

3. W. Li and K. Pingali, A Singular Loop Transformation Framework Based on Non-singular Matrices, *Int'l J. of Parallel Programming*, **22**(2):183-205 (1994).

4. P.M.W. Knijnenburg, E. Ayguadé and J. Torres, Multi-transformations of Nested Loops for Parallelizing Compilers, Tech. Rep. 96–14, Leiden University (1996).

5. D. Kulkarni and M. Stumm, CDA Loop Transformations, *Languages, Compilers and Run-Time Systems for Scalable Computers*, Kluwer Academic Press (May 1995).

6. M.E. Wolf and M. Lam, An Algorithmic Approach to Compound Loop Transformations, *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau, D. Gelernter, T. Gross and D. Padua (eds.), Pitman London (1991).

7. W.A. Kelly, W. Pugh, A Framework for Unifying Reordering Transformations, Tech Report CS-TR-3193, Dept CS, Univ. Maryland (May 1993).

8. O.Temam, E.D.Granston and W.Jalby, To Copy or Not To Copy: A compile-time technique for assessing when data copying should be used, *Proc. Supercomputing93*, Portland Oregon (1993).

9. T.E.Jeremiassen and S.J. Eggers, Reducing False Sharing on Shared Memory

Multiprocessors through Compile-Time Data Transformations, *Proc. Principles and Practice of Parallel Programming*, ACM Press (1995).

10. S. Chatterjee, J.R. Gilbert, R. Schreiber and S-H. Teng, Automatic Array Alignment in Data Parallel Programs, *Proc. Principles of Programming Languages*, ACM Press (January 1993).

11. J. Li and M. Chen, Index Domain Alignment: Minimising Cost of Cross-Referencing between Distributed Arrays, *IEEE Proc. of the Third Symposium on the Frontiers of Massively Parallel Computation* (1990).

12. M.F.P. O'Boyle and G.A. Hedayat, Data Alignment: Transformations to Reduce Communication on Distributed Memory Architectures, *Proc. Scalable High Performance Computing Conference*, Williamsburg, IEEE Press (April 1992).

13. D. Kulkarni and M. Stumm, Loop and Data Transformations: A Tutorial, University of Toronto, Tech Rep CSRI-337 (June 1993).

14. M. Cierniak and W. Li, Unifying Data and Control Transformations for Distributed Shared-Memory Machines, *Proc. Programming Language Design and Implementation* (June 1995).

15. Y.-J. Lu and H. Dietz, Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformations, *Proc. Fourth International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 589, Springer Verlag (1992).

16. J.M. Anderson, S.P. Amarasinghe and M.S. Lam, Data and Computation Transformations for Multiprocessors, *Proc. Principles and Practice of Parallel Programming*, ACM Press (1995).

17. S.-T. Leung and J. Zahorjan, Optimizing Data Locality by Array Restructuring, University of Washington, Department of Computer Science and Engineering, Tech Rep 95-09-01 (September 1995).

18. W. Li and K. Pingali, Access Normalisation: Loop Restructuring for NUMA Compilers, ACM Transactions on Computer Systems **11**(4):353-375 (1993).

19. D. Gannon, W. Jalby and K. Gallivan, Strategies for Cache and Local Memory Management by Global Program Transformations, *J. Parallel and Distributed Computing* **5**:587-616 (1988).

20. A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka and C.-W. Tseng, Unified Compilation of Fortran 77D and 90D, *ACM Letters on Programming Languages and Systems*, **2** (1993).

21. A.J.C. Bik and P.M.W. Knijnenburg, Reshaping Access Patterns for Improving Data Locality, *Proc. Sixth Workshop on Compilers for Parallel Computers* (1996).

22. M.E. Wolf and M.S. Lam, A Data Locality Optimizing Algorithm, *Proc. Programming Languages Design and Implementation* (1991).

23. F. Bodin and M.F.P. O'Boyle, A Compiler Strategy for SVM, *Proc. Third*

Workshop on Languages, Compilers and Runtime Systems, New York, Kluwer Press (May 1995).

24. K. Knobe, J.D. Lukas and G. Steele, Data Optimisation: Allocation of Arrays to Reduce Communication on SIMD machines, *J. Parallel and Distributed Computing*, **8**:102-118 (1990).

25. D. Bau, I. Kodukula, V. Kotlyar, K. Pingali and P. Stodghill, Solving Alignment using Elementary Linear Algebra, *Proc. Seventh Annual workshop on Languages and Compilers for Parellel Computing*, Ithaca, New York (1994).

26. M.F.P. O'Boyle, L. Kervella and F. Bodin, Synchronisation Mininimisation in a SPMD Execution Model, *J. Parallel and Distributed Computing* ,**29**:196-210 (1995).

Original	Column ↦ Row Major	Non-Singular Data Transformation

`Real A(4,8), B(8,4)`
`.`
`.`
`Do i = 1,3`
`  Do j = 2,4`
`    A(i,j)=A(i+1,j-1)+B(i+j,j)`
`  Enddo`
`Enddo`
`.`
`.`
	`Real A(4,8), B(8,4)`
	`C Row-major ordering`
	`.`
	`Do i = 1,3`
	`  Do j = 2,4`
	`    A(i,j)=A(i+1,j-1)+B(i+j,j)`
	`  Enddo`
	`Enddo`
	`.`
	`.`
		`Real A(8,4), B(4,11)`
		`.`
		`.`
		`Do i = 1,3`
		`  Do j = 2,4`
		`    A(j,i)=A(j-1,i+1)+B(j,i+4)`
		`  Enddo`
		`Enddo`
		`.`
		`.`

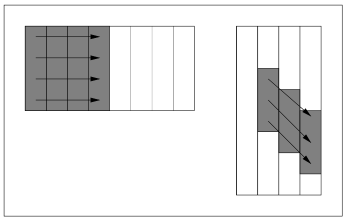
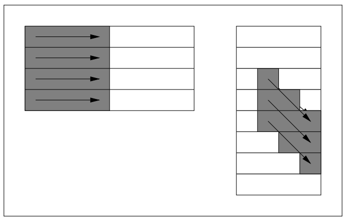
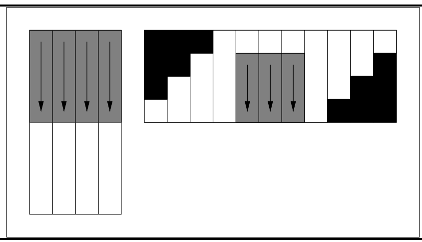
		

Figure 1: Stride Access Patterns

```
Do i = 1,n
 Do j = 2,n
  a(2*i,i+j) = i+j
Do i = 1,n
 Do j = 2,n
  a(2*i,j) = b(i,j)
```

Figure 2: Example

$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
permutation
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(i+j,2*i) = i+j`
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(j,2*i) = b(i,j)`
		$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
		scaling
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,2*i+2*j) = i+j`
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,2*j) = b(i,j)`

$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
reversal
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,-i-j) = i+j`
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,-j) = b(i,j)`
		$\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
		compression
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(i,i+j) = i+j`
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(i,j) = b(i,j)`

$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
skewing
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,3*i+j) = i+j`
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,2*i+j) = b(i,j)`
		$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
		shifting
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,i+j-1) = i+j`
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,j-1) = b(i,j)`

Figure 3: Transformations

Before Data Transformation	After Data Transformation	After Loop Transformation

`Real  B(8,4)`
`.`
`.`
`Do j = 1,4`
`  Do i = 1,8`
`    B(i,j) = i+j`
`  Enddo`
`Enddo`
`.`
`.`
	`Real  B(4,11)`
	`.`
	`.`
	`Do j = 1,4`
	`  Do i = 1,8`
	`    B(j,i-j+4) = i+j`
	`  Enddo`
	`Enddo`
	`.`
	`.`
		`Real B(4,11)`
		`.`
		`.`
		`Do j = 1,11`
		`  Do i = max(1,5-j), min(4,12-j)`
		`    B(i,j) = 2*i +j -4`
		`  Enddo`
		`Enddo`
		`.`
		`.`

Original	$\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}, a = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$	$\mathcal{T} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}, t = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$

Figure 4: Interaction of Loop and Data Transformations

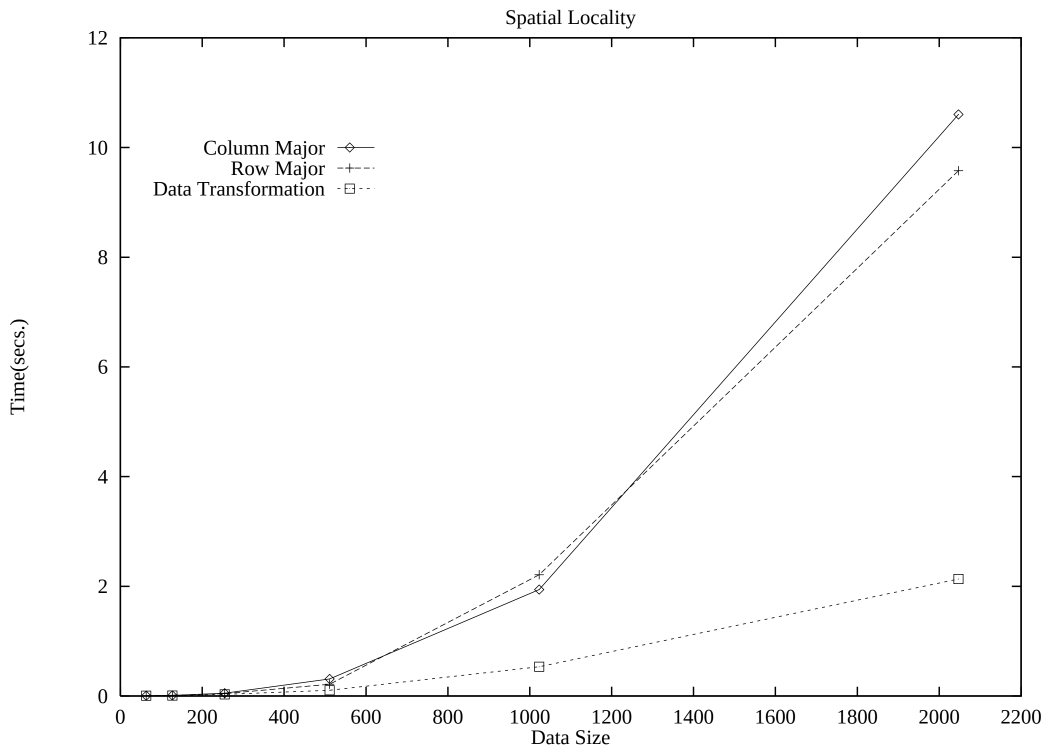


Figure 5: Spatial Locality

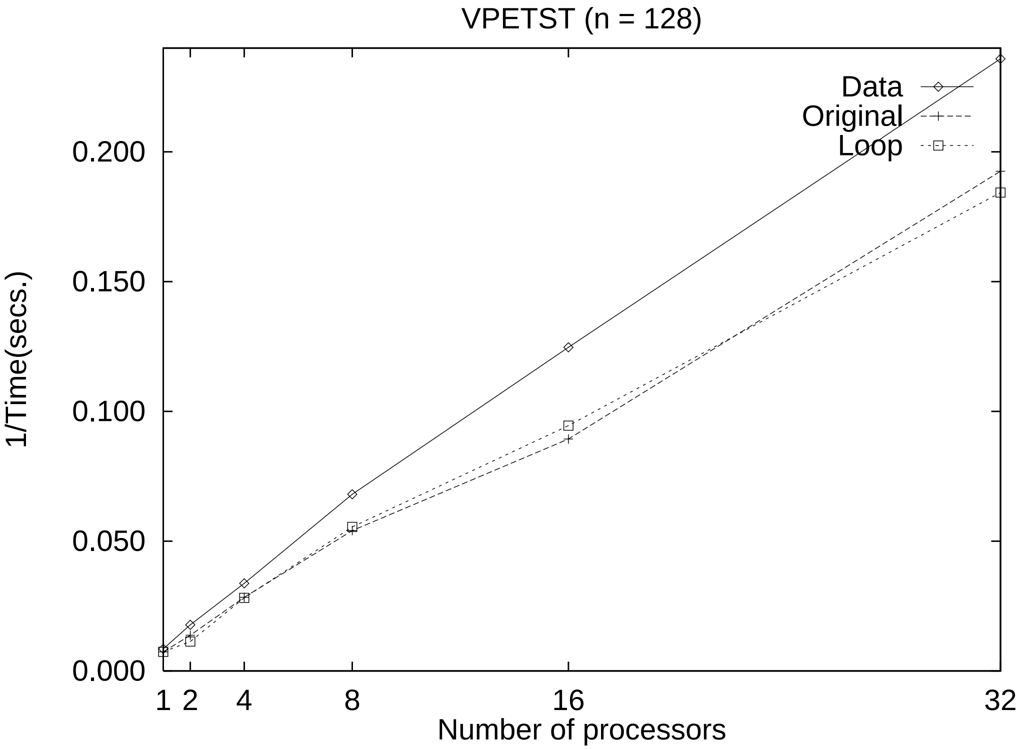


Figure 6: VPTST

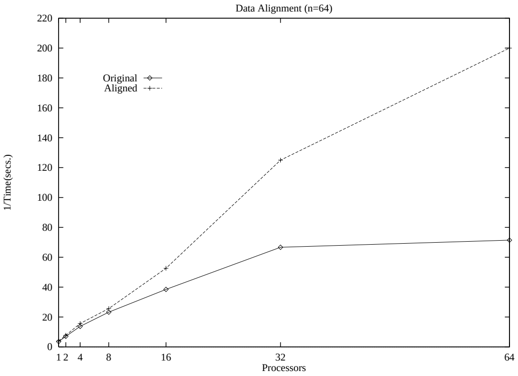
```
Do i = 1,n
 Do j = 1,n
   Do k = 1,n
     A(i,j) = A(i,j) + B(j+3,i+j) * C(k,n-i+1)
   Enddo
 Enddo
Enddo
```

(a) Before Transformation

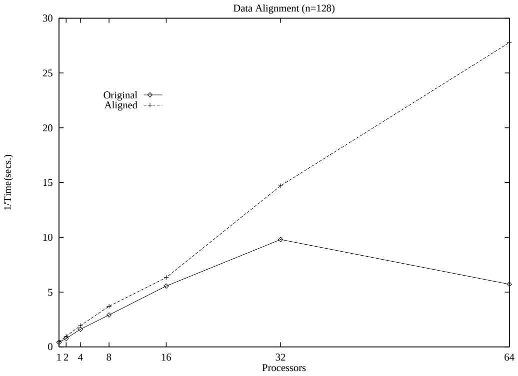
```
Do i = 1,n
 Do j = 1,n
   Do k = 1,n
     A(i,j) = A(i,j) + B(i,j) * C(i,k)
   Enddo
 Enddo
Enddo
```

(b) After Transformation

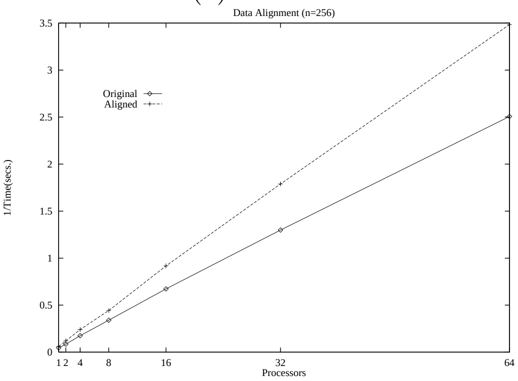
Figure 7: Alignment Example



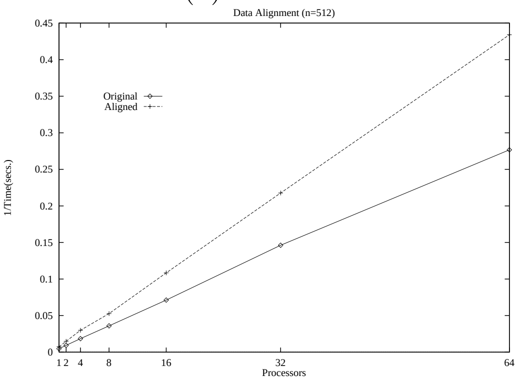
(a) $N = 64$



(b) $N = 128$



(c) $N = 256$



(d) $N = 512$

Figure 8: Data Alignment

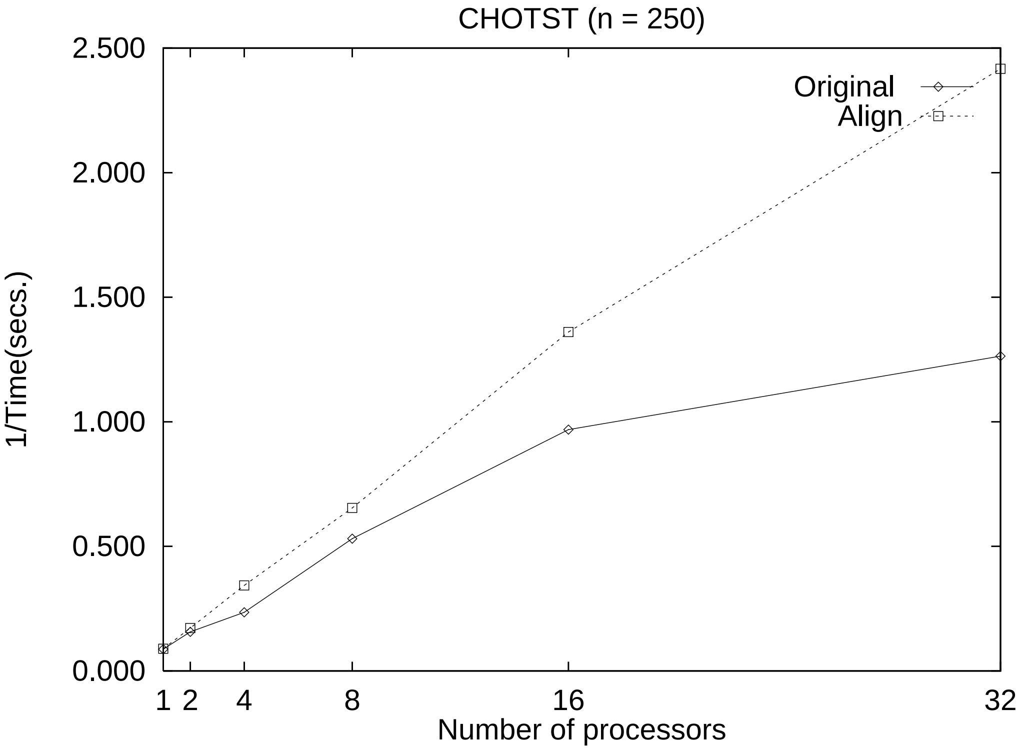


Figure 9: Chotst $N = 250$

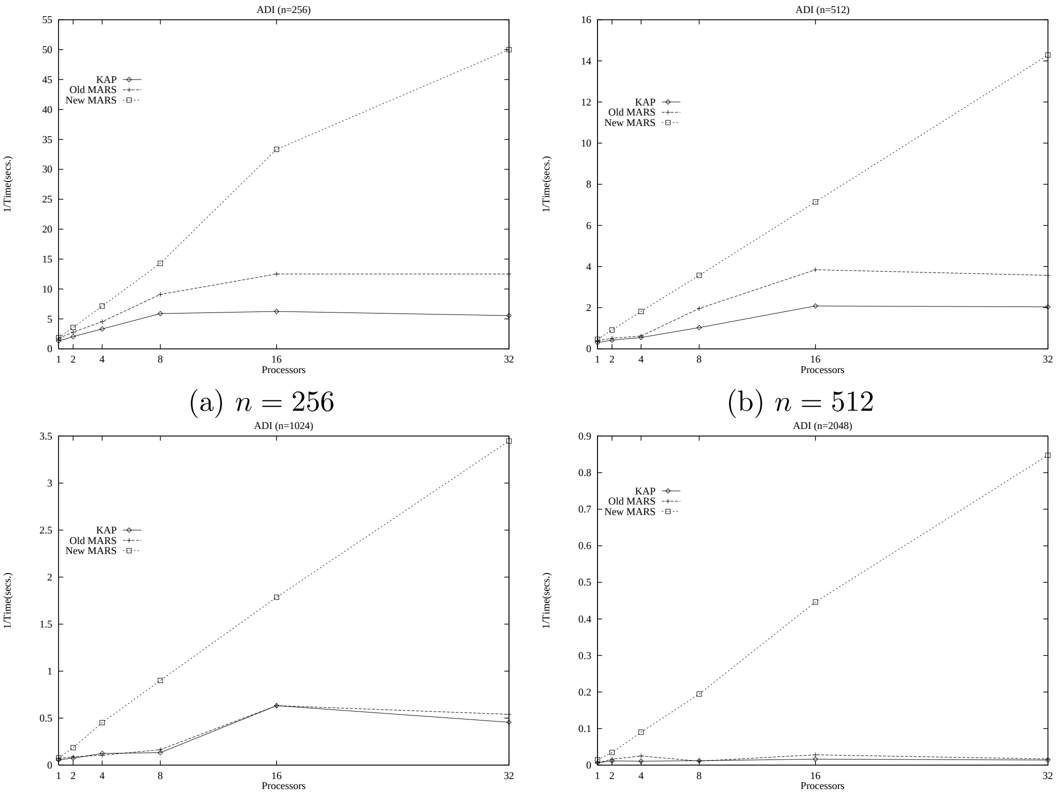
```
Real A(200,400), B(200,400)
Do k = 1,n
  Do i = 1,n
    A(i,k) = A(i,k-1) + B(i,k)
  Enddo
Enddo
```

(a) Before transformation

```
Real A(400,200), B(400,200)
Do i = 1,n
  Do k = 1,n
    A(k,i) = A(k-1,i) + B(k,i)
  Enddo
Enddo
```

(b) After transformation

Figure 10: ADI fragment



(a) $n = 256$ (b) $n = 512$

(c) $n = 1024$ (d) $n = 2048$

Figure 11: ADI fragment