Efficient Parallelisation using Combined Loop and Data Transformations

M.F.P. O'Boyle
Institute for Computing Systems Architecture
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 attempts to minimise parallelisation overhead on distributed shared memory machines, such as the SGi Origin 2000, by the combination of non-singular loop and data transformations. We show that conflicting requirements on a loop transformation may be resolved by using a data transformation and vice-versa. We develop optimisation criteria for locality, synchronisation and communication and show that neither loop nor data transformations can be solely used for efficient parallelisation. This leads to the development of a novel global optimisation heuristic which is applied to 3 SPEC kernels where it is shown to outperform techniques solely based on loop or data transformations and to give significant improvement over an existing state-of-the-art commercial auto-paralleliser.

1 Introduction

Effective utilisation of distributed shared memory multiprocessors relies on the efficient mapping of program parallelism to machine parallelism. Due to the increasing relative cost of memory latency, exploiting the memory hierarchy is essential. Otherwise, any gains made by discovering parallelism can easily be outweighed by overheads such as remote memory access. Parallelising compilers need to minimise any introduced synchronisation and inter-processor communication overhead as well as maximising temporal and spatial locality within a program.

Loop transformations for parallelism and locality [15, 21] have been extensively studied in the context of shared memory parallel machines. Although frequently successful, they suffer from the fact that the analysis and transformations are inevitably local, since the unit of consideration is a loop nest rather than the entire program. Furthermore, they are restricted in their application by data dependences. Conversely, data transformations, such as alignment and partitioning, have received much attention in distributed memory compilation [6, 9, 12]. As data layout has program wide impact, these techniques have, by necessity, been more global in their consideration. They are unaffected by data dependences but there has been, until recently, difficulty in applying data transformations to reshaped arrays across procedure boundaries (in [19] a solution to this problem is presented). Though potentially determining good overall layouts, data transformations are unable to remedy any introduced poor code localised within a section of the program.

The central problem with both these approaches is that of balancing conflicting requirements throughout the program. More specifically, if one part of a program, be it a loop or an array access, requires a particular transformation but another part requires a completely distinct transformation, how do we determine transformations that trade-off such requirements to give a globally acceptable result? In this paper we show that conflicting requirements on a loop transformation may be resolved by using a data transformation and vice-versa. Such a combination of loop and data transformations so as to get the best from both approaches has recently received some attention. In particular, Cierniak and Li [7] and Kandemir et al. [10, 11] have combined non-singular loop transformation with data transformations to improve spatial locality. The asymmetry between loop and data transformations in their representation, however, prevents the direct combination of loop and data transformations.

This paper develops a compiler heuristic to minimise parallelisation overhead. It explores how conflicting requirements on a loop transformation may be resolved by using a data transformation and vice-versa. This is achieved by treating loop and data transformations in a unified manner. We develop specific optimisation criteria for spatial and temporal locality, and communication and synchronisation reduction. We define those instances where both loop and data transformations can be used or when only one is applicable. We examine how different optimisation criteria can lead to conflict and where the use of complementary transformations may overcome this. This is followed by an

```
L1: Do j = 1,8
     Do i = 1 ,8
      B(i,j) = i + j
     Enddo
    Enddo
L2:
    Do k = 1, 8
     Do j = 1,8
      Do i = j+1,8
       A(i,i+j) = A(i,i+j)
       + B(i-j,i)+ C(k,i)
      Enddo
     Enddo
    Enddo
```

Figure 1. Serial Code

overall optimisation heuristic that trades-off costs in cases where conflicts are irreconcilable. This approach has been implemented in the MARS compiler [5] and we show that it outperforms existing approaches on three SPEC kernels.

The next section presents a motivating example outlining some of the main ideas developed later in the paper. Section 3 briefly outlines the notation used and provides a description of non-singular loop and data transformations and when they may be used interchangeably. Section 4 examines optimising transformations and criteria for when they may be applied. Section 5 presents a global optimising heuristic which is followed in section 6 by experimental results showing the applicability of our scheme. Section 7 reviews related work which is followed in section 8 by some brief concluding remarks.

2 Example

In this section, we examine a simple example to illustrate some of the main points described in this paper. Consider the program in Figure 1. If this code is mapped to a four processor parallel machine by row partitioning and the owner computes-rule, we have the local node code in row 1, column 1 of Figure 2 where the terms `lo` and `hi` refer to the local array bounds. The impact of subsequent transformations to this program is shown in row 1, columns 2 to 4 whereas the diagrams in rows 2 to 5 give a graphical representation. Here, each box represents a particular array, with dotted lines showing how the data is partitioned across the 4 processors[1] and we will focus our attention on the behaviour of processor 3. The dark shaded regions refer to local data accessed by the programs. Lighter shading corresponds to remote memory access, incurring possible synchronisation. The direction of data access is shown by an arrow.

Due to column-major layout in Fortran, array B in the first loop has good stride access (row 2, column 1). However, in the second loop, all arrays have poor spatial locality. Furthermore, in Loop 2 there is remote memory access to arrays B and C, shown by the light shaded regions. There is consequently a cross-processor flow dependence on array B from Loop 1 to 2 resulting in the inserted barrier.

If we first apply a data transformation that realigns data so as to minimise communication and synchronisation we obtain the new code in row 1, column 2. Now there is no remote memory accesses and the cross-processor data dependence has been eliminated and thus the barrier synchronisation has been removed. However, there is still poor spatial locality with respect to arrays A and B in Loop 2 and the spatial locality in Loop 1 has been destroyed.

Further data transformations can be applied to arrays A, B and C, giving the code in column 3. Here, spatial locality has been improved in Loop 2 without introducing synchronisation or communication overhead. However, the spatial locality in Loop 1 is still just as poor. In fact, it is impossible to find a data layout that will improve spatial locality to array B in both loop nests. If, however, a loop transformation is applied then we have the code in column 4, which has the same properties as column 3 with the addition of good spatial locality in Loop 1. Neither loop nor data transformations alone could produce such a program: a combined approach was necessary. In subsequent sections, we will detail when each transformation can be used before developing a compiler heuristic that exploits the benefits of both approaches and performs trade-offs where necessary.

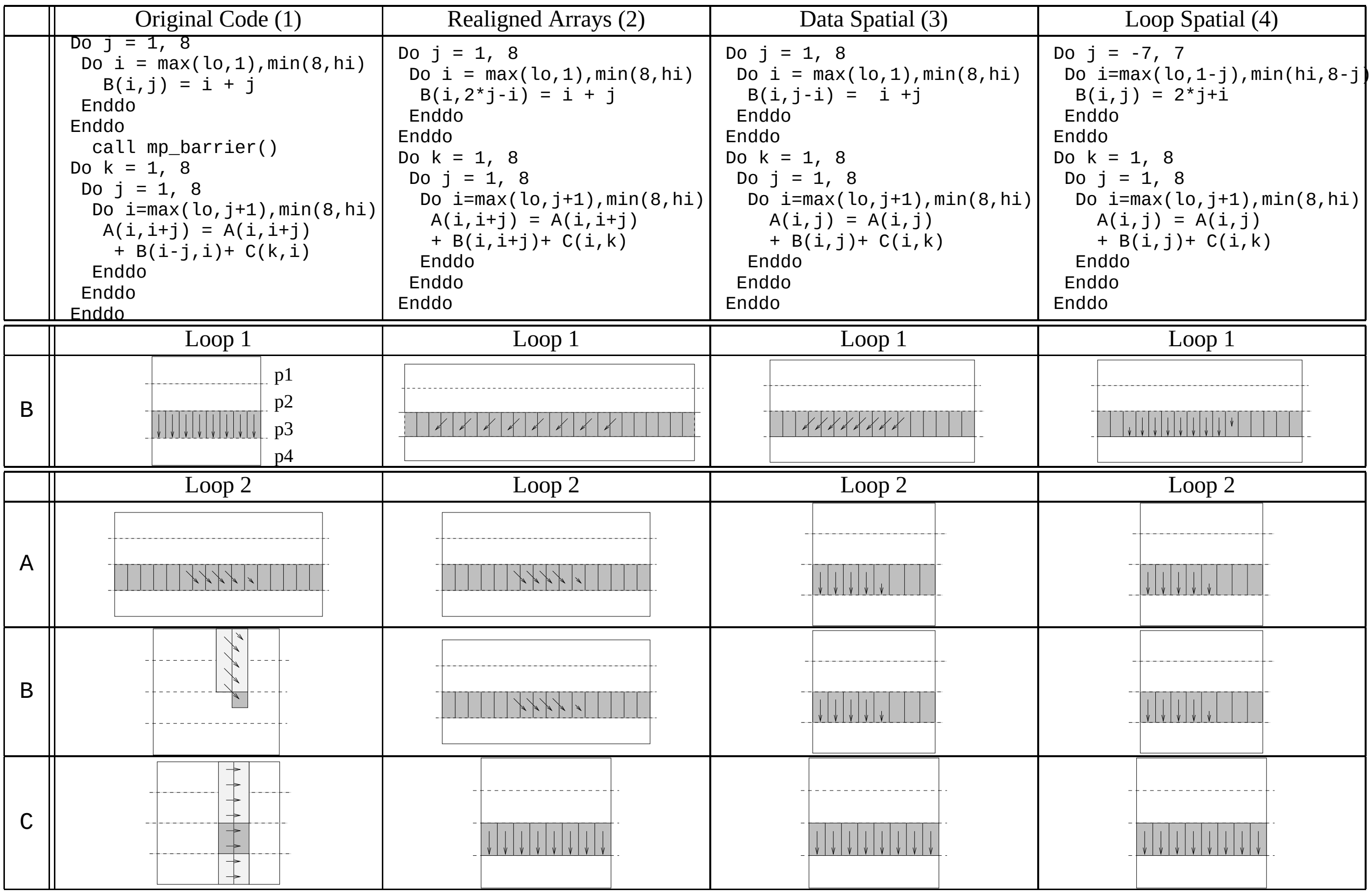
3 Transformations

In this section, we briefly introduce the algebraic notation used to describe transformations and their properties.

Spaces The loop indices or *iterators* 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. The loop ranges can be described by a system of inequalities defining the *polyhedron* or *iteration space* $\mathbf{B}J \le \mathbf{b}$. where $\mathbf{B}$ is a $(\ell \times M)$ integer matrix and $\mathbf{b}$ a $(\ell \times 1)$ vector for some ℓ. The data storage of an array A can also be viewed as a polyhedron. We introduce *formal indices* $\mathcal{I} = [i_1, i_2, \ldots, i_N]^T$, where N is the dimension of array A, to describe the *array index space*. This space is given by the polyhedron $\mathbf{A}\mathcal{I} \le \mathbf{a}$, where $\mathbf{A}$ is a $(2N \times N)$ integer matrix and $\mathbf{a}$ a $(2N \times 1)$ vector. We assume that the subscripts in a reference to an array A can be written as $\mathcal{U}J + u$, where $\mathcal{U}$ is a $(N \times M)$ integer matrix and u is a $(N \times 1)$ vector.

Data Transformations A linear data transformation is applied to the index space of a particular array and to *all* accesses to that array throughout the program and is therefore *global* in nature. A data transformation $\mathcal{A}$ maps an index vector $\mathcal{I}$ to a new index vector $\mathcal{I}' = \mathcal{A}\mathcal{I}$. Each array access $\mathcal{U}$ for an array A must be globally updated to $\mathcal{U}' =$

[1]In SGI terms, we can consider part of each array as having it's home node determined by C$distribute(block,*)



Original Code (1)

```
Do j = 1, 8
 Do i = max(lo,1),min(8,hi)
   B(i,j) = i + j
 Enddo
Enddo
  call mp_barrier()
Do k = 1, 8
 Do j = 1, 8
  Do i=max(lo,j+1),min(8,hi)
   A(i,i+j) = A(i,i+j)
    + B(i-j,i)+ C(k,i)
  Enddo
 Enddo
Enddo
```

Realigned Arrays (2)

```
Do j = 1, 8
 Do i = max(lo,1),min(8,hi)
  B(i,2*j-i) = i + j
 Enddo
Enddo
Do k = 1, 8
 Do j = 1, 8
  Do i=max(lo,j+1),min(8,hi)
   A(i,i+j) = A(i,i+j)
   + B(i,i+j)+ C(i,k)
  Enddo
 Enddo
Enddo
```

Data Spatial (3)

```
Do j = 1, 8
 Do i = max(lo,1),min(8,hi)
  B(i,j-i) =  i +j
 Enddo
Enddo
Do k = 1, 8
 Do j = 1, 8
  Do i=max(lo,j+1),min(8,hi)
    A(i,j) = A(i,j)
    + B(i,j)+ C(i,k)
  Enddo
 Enddo
Enddo
```

Loop Spatial (4)

```
Do j = -7, 7
 Do i=max(lo,1-j),min(hi,8-j)
  B(i,j) = 2*j+i
 Enddo
Enddo
Do k = 1, 8
 Do j = 1, 8
  Do i=max(lo,j+1),min(8,hi)
    A(i,j) = A(i,j)
    + B(i,j)+ C(i,k)
  Enddo
 Enddo
Enddo
```

Figure 2. Partitioned and Transformed Code

$\mathcal{AU}$. Data transformations are therefore *left-hand* transformations when applied to array access functions.

Loop Transformations A linear loop transformation T maps an iteration vector J to a new iteration vector $J' = TJ$. Each access $\mathcal{U}$ within the loop nest must be updated to $\mathcal{U}' = \mathcal{U}T^{-1}$. Thus, loop transformations are *right-hand* transformations when applied to array accesses and are *local* in nature.

Complementary Transformations Frequently, if we wish to transform an array access $\mathcal{U}$ into a more desirable form $\mathcal{U}'$, we may be able to *either* apply a data transformation $\mathcal{A}$ *or* a loop transformation T to give the desired effect, depending on legality and optimisation criteria being satisfied and any side effects on the rest of the program. It is also possible to reverse the effect of a data transformation by a loop transformation and vice-versa. Consider the application of data transformation $\mathcal{A}$ on access $\mathcal{U}$. If we then apply a loop transformation $T^{-1} = \mathcal{U}^{-1}\mathcal{A}^{-1}\mathcal{U}$ to the updated access $\mathcal{U}'$, we recover the original access: $\mathcal{U}'T^{-1} = \mathcal{U}$. Similarly for a loop transformation T, we can reverse its impact by using a data transformation $\mathcal{A} = \mathcal{U}T\mathcal{U}^{-1}$. Thus the *global* impact of data transformations can frequently be resolved *locally* by loop transformations. Conversely, data transformation can recover the structure of a particular access without affecting other accesses within the loop nest.

These complementary transformations depend on inverses. Although an access matrix $\mathcal{U}$ is often singular, in practise it is straightforward to generate a pseudo-inverse $\mathcal{U}^\dagger$ such that $\mathcal{U} \times \mathcal{U}^\dagger = I$. For example, consider the access to A in Loop 2 in Figure 2 row 1, column 1.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}_{\mathcal{U}_A} \times \begin{bmatrix} 0 & 0 \\ -1 & 1 \\ 1 & 0 \end{bmatrix}_{\mathcal{U}_A^\dagger} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We cannot, however, guarantee that the resulting constructed transformation actually reverses the effect of the original. The practical implication is that for certain optimisations and certain access matrices, only one form of transformation may be used.

4 Properties

This section examines four optimisation criteria in terms of access matrix structure, allowing both loop and data transformations to be considered. It develops existence criteria for each optimisation and examines whether or not loop and data transformations may be applied.

4.1 Communication and Synchronisation

References requiring access to remote memory incur inter-processor communication and potentially synchronisation, to cover cross-processor dependences. This section describes how transformations can reduce both of these overheads. We assume a data centric approach to parallelisation, achieved by partitioning data across the processor space and scheduling work according to the owner-computes rule [5].

We wish to find a data transformation $\mathcal{A}$ that reduces communication. Consider two references $\mathcal{U}_{\mathtt{A}}$ and $\mathcal{V}_{\mathtt{B}}$. If the ith index of the reference $\mathcal{V}_{\mathtt{B}}$ is to be aligned to $\mathcal{U}_{\mathtt{A}}$, we have to apply a data transformation $\mathcal{A}$ to `B` and the ith row of $\mathcal{A}$, denoted by y, should obey the following equation:

$$y\mathcal{V} = \mathcal{U}_i \quad \text{or} \quad \mathcal{V}^T y^T = \mathcal{U}_i^T \tag{1}$$

where $\mathcal{U}_i$ is the ith row of $\mathcal{U}$. This immediately leads to the following theorem that says that two arrays can be aligned on a particular dimension as long as one of the arrays is not-invariant of a particular (combination of) iterators appearing in the corresponding subscript of the other array.

Theorem 1 *It is possible to align a row of a matrix $\mathcal{V}$ with another access matrix $\mathcal{U}$ iff u_i^T is perpendicular to the kernel, or null space, of $\mathcal{V}$.*

This theorem [2] enables us to determine whether there exists a data transformation to align arrays to eliminate inter-processor communication. In section 5, this property will be used as part of the global optimisation heuristic.

To deal with synchronisation, we use the following two theorems [17]. The first is based on the owner-computes model which implies that all write accesses to an array are to local data. The second implies that if we align two or more array references, we reduce the chance of synchronisation associated with them.

Theorem 2 *Output data dependences never require synchronisation.*

Theorem 3 *If a read array reference is partitioned on aligned indices with the write array reference in a particular statement, no data dependences associated with that read array reference require synchronisation.*

[2]Due to space restrictions, proofs are omitted.

Once we have computed the minimum number of cross-processor dependences, we need to know where to place synchronisation points so as to preserve program semantics at minimal cost [5].

4.2 Spatial and Temporal Locality

This section examines uni-processor locality which must be exploited if the gains of parallelisation are not to be lost due to poor use of the memory hierarchy.

4.2.1 Spatial Locality

Assuming a column-major ordering of arrays, the innermost iterator should, ideally, access only the first index of an array, if any, for good spatial locality. Hence we say that an access matrix $\mathcal{U}$ has *good spatial locality* if $\mathcal{U} = \begin{bmatrix} Y & \mathbf{1} & O \end{bmatrix}$ where O is an optional null sub-matrix, $\mathbf{1} = [1, 0, \ldots, 0]^T$ and Y is an arbitrary sub-matrix. The 1 corresponds to the innermost iterator referenced by the first index. For good spatial locality, this iterator must not be referred to by any other index. The following theorems define when loop and data transformations can be used to improve spatial locality, depending on the access matrix structure.

Theorem 4 *It is always possible to construct a data transformation $\mathcal{A}$, which transforms an access $\mathcal{U}$ into a form with good spatial locality.*

Theorem 5 *It is possible to construct a loop transformation T which transforms an access $\mathcal{U}$ of the form $\mathcal{U}^T = [X^T O]^T$ into a form with good spatial locality iff $rank(X) = N$, where N is the number of rows of X.*

One immediate consequence of the above two theorems is that data transformations are strictly more powerful in improving spatial locality than loop transformations.

4.2.2 Temporal Locality

Temporal locality occurs when the same value is used more than once within a calculation. If a reference is *invariant* of the innermost iterator, then temporal locality is exploited [4, 8, 21]. Hence $\mathcal{U}$ has *good temporal locality* if $\mathcal{U} = \begin{bmatrix} Y & O \end{bmatrix}$ where Y is an arbitrary sub-matrix and O is a null sub-matrix. We present two theorems which describe when loop and data transformations can derive such a form.

Theorem 6 *Data transformations cannot improve or affect temporal locality.*

Theorem 7 *A loop transformation T to transform an access $\mathcal{U}$ into a form with good temporal locality exists iff $rank(\mathcal{U}) < M$, where M is the number of enclosing iterators.*

When finding a loop transformation to enhance locality, the optimisation criteria for temporal and spatial locality are complementary, allowing direct construction of loop transformations that improves both spatial and temporal locality (see section 5.4).

5 Combining Data and Loop Transformations for Optimisation

This section examines how loop and data transformations interact with respect to different optimisation criteria. It also develops a heuristic that attempts to construct the best set of transformations to trade off any potentially conflicting requirements.

5.1 Global Optimisation

We wish to find a data transformation $\mathcal{A}$ and a loop transformation T such that $\mathcal{A}\mathcal{U}T^{-1}$ has the "best" structure for all accesses $\mathcal{U}$. This requirement forms a system of equations with a quadratic number of unknowns making direct solution impossible. Due to their global nature, any data transformation $\mathcal{A}$ must consider all references to the array throughout the program and there will be a trade-off when different transformations are required in different program locations. For loop transformations difficulties arise if different accesses within a loop nest require different loop nest orderings. Finally, the notion of "best" is compile-time undecidable. Instead, we propose a heuristic that attempts to minimise overheads. We prioritise those sections of the program that are likely to be executed most frequently and those overheads that are most expensive. Thus, we focus on the deepest loop nests and consider synchronisation and communication before intra-processor locality since a barrier synchronisation, for instance, is an order of magnitude more expensive than a L1 cache miss. We still, however, want good uni-processor code and therefore, after parallelisation, we optimise so that there will be as few accesses as possible to expensive levels of the memory hierarchy. We first apply data transformations as long as they do not affect inter-processor synchronisation, since they do not affect temporal locality (Theorem 6) nor the ability of loop transformations to exploit temporal locality. Moreover, data transformations will almost inevitably require adjustments to loop nest structure [19]. Hence we apply loop optimisations after data transformations.

5.2 Global Data Layout Optimisation

Determining data layout in the context of alignment for message-passing architectures has received a large amount of interest [3, 12]. These approaches rely on a restricted subset of transformations and are not directly applicable in the present context. Instead, we use a technique based on determining the best transformation for each array reference pair and use a metric to help trade-off conflicting requirements.

1. Reduce the following matrix to row echelon form
$$\left[\begin{array}{ccc} \hat{\mathcal{V}}^T & \hat{\mathcal{U}}^T & \multirow{2}{*}{I_M} \\ O & O & \end{array}\right]$$
2. Reorder by elementary row operations such that leading non-zeros lie on the diagonal.
3. For each column i of the reduced $\hat{\mathcal{U}}$ matrix, if this column is independent, place this row in row i of $\mathcal{A}$. Otherwise, select an independent row of I_M

Figure 3. Data Layout Algorithm

Alignment metric The more corresponding rows of array access matrices $\mathcal{U}$ and $\mathcal{V}$ are equal, the more aligned the arrays are. The alignment evaluation function H therefore is simply the number of equal rows.

$$H(\mathcal{U},\mathcal{V}) = \sum_{k=1}^{M} \delta_{\mathcal{U}_k,\mathcal{V}_k} \text{ where } \delta_{e,f} = \begin{cases} 1 & e = f \wedge e \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

For example, for arrays A and B in figure 2, $H(\mathcal{U}_{\mathtt{A}},\mathcal{U}_{\mathtt{B}}) = 0$ in the program shown in column 1, but $H(\mathcal{U}_{\mathtt{A}},\mathcal{U}_{\mathtt{B}}) = 2$ in the program shown in column 2.

Construction of a Data Transformation The algorithm is shown in Figure 3, where the data transformation $\mathcal{A}$ is chosen in order to maximise the number of perfectly aligned subscripts or rows of $\hat{\mathcal{U}}$ and $\mathcal{A}\hat{\mathcal{V}}$, 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 $\mathcal{A}$, it is necessary to find a solution for as many rows to equation (1). The algorithm reduces an $\hat{\mathcal{V}}^T$ to row echelon form, while simultaneously carrying the same elementary operations on $\hat{\mathcal{U}}^T$ and the identity matrix I_M.

Global Data Alignment Algorithm Based on the previous section, we now develop a global algorithm, shown in Figure 4, to determine the best layout for all arrays. The most significant regions are first selected based on loop nest depth or profiling. Once we have the global data layout, we can use theorems 6 and 7 to mark any dependences guaranteed to be local as a pre-processing step for barrier synchronisation placement [5].

```
1. Order arrays in terms of importance
2. For each pair of arrays A and B, determine within each
   statement $s$ the data transformation $\mathcal{A}_s$ that maximises
   $H(\mathcal{U}_{\mathtt{A}}, \mathcal{A}_s\mathcal{V}_{\mathtt{B}})$ for all accesses $\mathcal{U}_{\mathtt{A}}$ and $\mathcal{V}_{\mathtt{B}}$ in $s$.
3. For each pair A and B, determine $s$ such that
   $\sum_k H(\mathcal{U}^k_{\mathtt{A}}, \mathcal{A}_s\mathcal{V}^k_{\mathtt{B}})$ is maximised for all accesses
   $\mathcal{U}^k_{\mathtt{A}}$ and $\mathcal{V}^k_{\mathtt{B}}$ to A and B, respectively. Set $\mathcal{A}_{\mathtt{A},\mathtt{B}} = \mathcal{A}_s$.
4. For $\mathtt{A} = 1, \ldots, \mathtt{NumArrays}$, propagate alignment:
     For $\mathtt{B} = \mathtt{A} + 1, \ldots, \mathtt{NumArrays}$
       For $\mathtt{C} = \mathtt{B} + 1, \ldots, \mathtt{NumArrays}$
         $\mathcal{A}_{\mathtt{B},\mathtt{C}} := \mathcal{A}_{\mathtt{A},\mathtt{B}} \times \mathcal{A}_{\mathtt{B},\mathtt{C}}$
```

Figure 4. Global Alignment Algorithm

```
1. Let $T$ be given by $t_{ij} = 1$ if $i + j = N + 1$, and 0
   otherwise.
2. Reduce $\mathcal{U}T$ to integer row echelon form by rowwise
   operations $\mathcal{B}$.
3. For $i \in 1, \ldots, N$
     If row $P_i$ is null, set $\mathcal{A}_i = \mathcal{B}_i$.
     Otherwise, set $\mathcal{A}_i$ to the $i$th basis vector.
```

Figure 5. Data Transformation for Spatial Locality

5.3 Data Transformations for Spatial Locality

We now consider data layout transformations to improve spatial locality. These data transformations will not affect temporal locality (Theorem 6), but it is important that indices that have been aligned by the previous stage are not "de-aligned" as this may possibly introduce cross-processor dependences. Again we use a metric to help trade-off conflicting requirements.

Spatial locality metric The spatial locality metric S evaluates the improvement of a layout transformation $\mathcal{A}$ for an array by assigning 1 if the access matrix is in the correct format (see section 4.2.1) and 0 otherwise.

$$S_{\mathcal{U}} = \begin{cases} 1 & \mathcal{U}_{1,k} = 1 \text{ and } \forall i \in 2, \ldots, N : \mathcal{U}_{i,k} = 0 \\ 0 & otherwise \end{cases}$$

where k is the first non-zero column starting from the rightmost column of $\mathcal{U}$.

Constructing a Spatial Locality Data Transformation The algorithm in Figure 5 attempts to create an access matrix

```
1. For each array A and each reference $r \in 1, \ldots, R_{\mathtt{A}}$ to
   array A, determine best legal $\mathcal{A}^r_{\mathtt{A}}$
2. For each A, determine $\mathcal{A}_{\mathtt{A}} = \max_{r=1}^{R_{\mathtt{A}}} \sum_{k=1}^{R_{\mathtt{A}}} S_{\mathcal{A}^r_{\mathtt{A}}\mathcal{U}^k_{\mathtt{A}}}$
```

Figure 6. Global Spatial Locality Algorithm

with good spatial locality. It is an extension of the algorithm described in [18] by restricting attention to those indices of the array that are guaranteed to be non-partitioned. We introduce an $N \times N$ partition matrix $\mathcal{P}$, where each row is either the corresponding row of the identity matrix that signifies that the corresponding row of the array under consideration is to be partitioned along that dimension, or null (no partitioning).

Global Spatial Locality Algorithm - Data The global algorithm uses the preferred data layout transformations of each reference to an array and selects the best transformation based on the evaluation function S. By construction parallelism, synchronisation and communication are unaffected.

5.4 Loop Transformations for Spatial and Temporal Locality

In this section we show how to construct a valid loop transformation that increases the locality in a loop nest. We take into consideration multiple accesses of possibly different arrays and then show how the set of all dependences $\mathcal{D}$ can be used to construct a legal transformation.

Construction of a Transformation Given an access matrix $\mathcal{U}$, the array is traversed along lines with direction equal to the last column of $\mathcal{U}$. A loop transformation T changes this access direction to a new direction given by the last column of $\mathcal{U}T^{-1}$ which equals $\mathcal{U}\vec{c}$ if we denote the last column of T^{-1} by $\vec{c}$. Hence, if the transformed access should have a column wise direction for, say, a 2 dimensional array, then $[1\ 0]^T$ and $\mathcal{U}\vec{c}$ should be linearly dependent. If we can find $\vec{c}$ with this property, then we can construct T^{-1} and hence T by a completion method for unimodular transformations [2]. Note that in case $\mathcal{U}\vec{c} = \vec{0}$ *temporal locality* occurs.

First, we observe that a vector $\vec{a}$ is linearly dependent on $[1\ 0]^T$ iff $[0\ 1]\ \vec{a} = \vec{0}$. Hence we need to find $\vec{c}$ such that $[0\ 1]\ \mathcal{U}\,\vec{c} = \vec{0}$. If we have a collection $\{\mathcal{U}_i \mid 1 \leq i \leq m\}$ of accesses, then all access patterns are given simultaneous column wise access if $\vec{c}$ satisfies $\mathcal{S}\vec{c} = \vec{0}$ for the following *objective matrix* $\mathcal{S}$:

1. Construct the set $\tilde{\mathcal{D}} = \{\vec{d} \in \mathcal{D} \mid T\vec{d} \equiv (0,\ldots,0,a)^T\}$.

2. Select $z = -1$ if $U\vec{d} \prec \vec{0}$ for any $\vec{d} \in \tilde{\mathcal{D}}$, and $z = 1$ otherwise.

3. Construct Y. Let $\tilde{T}$ consist of the first $M-1$ rows of T. Determine whether there exists a vector $\vec{y}$ with $\gcd(y_1,\ldots,y_{M-1}) = 1$ such that $\vec{y}\cdot\tilde{T}\vec{d} > 0$ holds for all $\vec{d} \in \mathcal{D} - \tilde{\mathcal{D}}$. The construction of Y fails if no such y exists. Otherwise, we use y as the first row of Y and use a completion method.

Figure 7. Construct Valid V

$$\mathcal{S} = \begin{bmatrix} 0\,1 & & \\ & \ddots & \\ & & 0\,1 \end{bmatrix} \begin{bmatrix} \mathcal{U}_1 \\ \vdots \\ \mathcal{U}_m \end{bmatrix} \qquad (2)$$

We use integer echelon reduction to compute the kernel of $\mathcal{S}$ [2]. We choose a vector $\vec{c}$ for the last column of T^{-1} from the set of basis vectors of this kernel and use a completion method [2] to construct an $M \times M$ unimodular matrix T. If $\mathcal{S}$ is non-singular then the reshaping method fails: the only element in its kernel is $\vec{0}$.

The loop transformation T is not necessarily legal. However, the following V is still a unimodular matrix for which the last column of the inverse is $\pm\vec{c}$ and hence is a transformation that we can use:

$$V = \begin{bmatrix} & & & 0 \\ & Y & & \vdots \\ & & & 0 \\ 0 & \ldots & 0 & \pm 1 \end{bmatrix} T \qquad (3)$$

Consequently, if we can construct a valid V then we are done. Otherwise, another $\vec{c}$ is tried until either this construction is successful, or none are left. In the latter case, the reshaping method fails.

Global Spatial and Temporal Locality Algorithm - Loop Summarising, the following is an algorithm to construct a valid loop transformation to increase both temporal and spatial locality.

1. Construct the objective matrix $\mathcal{S}$ and hence T.

2. Construct V (see Figure 7).

3. If the reshaping method fails, construct a new $\mathcal{S}$ using less access matrices and try again.

6 Results

In order to validate the approach described in this paper, we ran a number of experiments on an SGi Origin 2000. Three SPECfp92 kernels, `vpetst, btrtst` and `chotst`, were selected to test our compiler algorithm against alternative techniques. Four different approaches were tested and their performance plotted against the number of processors p. Firstly, each kernel was parallelised by our compiler MARS [5], where the combined algorithm developed in this paper was compared against loop only and data only approaches. Secondly, to give a broader comparison, PFA, a commercial loop-orientated parallelising compiler that makes use of limited data layout transformations such as global index reordering, was also evaluated. The results of these experiments can be seen in Figures 8, 9 and 10.

VPETST The combined algorithm gives a performance improvement over the other approaches especially for larger values of p. By reordering the array layout, stride-1 access throughout the program is achieved, without the need for any loop reordering. For this reason, the programs derived from the Data only and Combined approaches are identical. In the case of Loop only transformations, the lack of data transformations has relatively little impact, as data alignment is already ideal, so there is no unnecessary communication/synchronisation. Applying loop transformations improves spatial locality in two of the main 2-dimensional loop nests, but it does 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 8 labelled ‘Loop’. Finally, the performance of PFA is approximately 50% slower on 32 processors than the combined approach and 20% slower than the loop based approach. The later difference is primarily due to additional synchronisation overhead when using loop-based parallelisation, as the communication and locality properties are the same as ‘Loop’.

BTRTST Once again the combined approach has the best overall performance. Using purely data transformations achieves similar results except for large p where the inability to improve spatial locality within one of the loop nests increases in significance. The solely loop based approach performs significantly worse than either the combined or data approaches. Here, there is poor data alignment which results in communication and inserted barrier synchronisation. PFA improves on the loop only approach by merging loops and decreasing synchronisation overhead. The combined approach is, however, a factor 2 faster than PFA for large p.

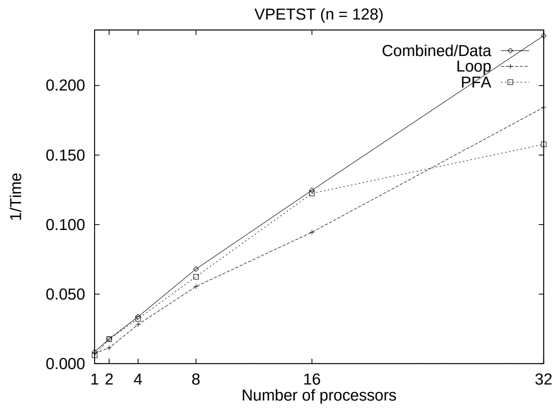


Figure 8. VPETST

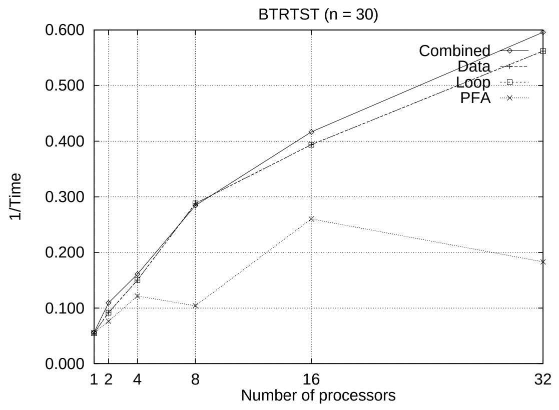


Figure 9. BTRTST

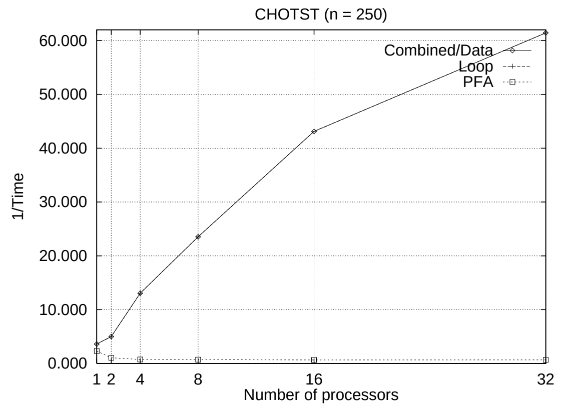


Figure 10. CHOTST

CHOTST This program shows the importance of using data transformations in parallelisation. Their usage eliminates communication and synchronisation and improves spatial locality without the need for further loop transformations, hence the combined and data only approaches give the same results. Both the loop only approach and PFA suffer from poor stride and excessive synchronisation. In the case of PFA there are two orders of magnitude more synchronisation than in the combined case. This is compounded by poor spatial locality in one of the deepest loop nests so that the parallelised code never runs faster than the sequential version. The combined approach is a factor of 80 times faster than PFA on 32 processors.

7 Related Work

There is a very large body of work concerned with improving parallelism and locality using program transformations. In [21], unimodular loop transformations are used to improve parallelism and locality, which is extended to the non-singular case in [15]. They restrict themselves to loop transformations and do not consider cases where there are conflicting access requirements. In such instances, [16] proposes a simple heuristic, limited to just loop permutations, which is extended to the unimodular case in [4]. These approaches are limited in that they do not consider array layout transformations.

Other researchers have considered data transformations, primarily with respect to data alignment and partitioning. In [3, 12] approaches based on graph theory, integer programming and linear algebra are explored. Most of this work, however, considers alignment to be part of a mapping process rather than a program transformation and thus, parallels with loop transformation are absent. In [20], alignment as a program level transformation is first presented while [14] uses a similar representation for uni-processor spatial locality. In [7], data linearisation transformations are considered as a means to change array layout, while data permutation and strip-mining transformations are considered in [1]. In [18], we developed a new framework describing non-singular data transformations equivalent in standing to loop transformations and described how they may be used to improve program performance. Again, these approaches are limited as they restrict themselves to just data transformations.

There has been recent work considering the combination of loop and data transformations in improving program performance. Using a hyperplane formulation of data transformations and non-singular loop transformations, an algorithm, which considers a restricted set of loop and data transformations, is proposed as a means of improving locality [7]. In [10, 11], a similar formulation is used, but considers a wider class of transformations. These approaches, however, have an asymmetric treatment of loop and data transformations and do not explicitly consider multi-processor issues such as communication and synchronisation overhead. Instead, reliance is placed on the fact that exploiting inner locality may give outer loops which may be parallelised, an argument similar in spirit to [21]. This paper, however, uses a unified representation of loop and data transformations and explicitly considers multi-processor issues and how these interact with locality.

8 Conclusion

This paper has shown how both non-singular loop and data transformations may be used for the same optimisation goals and has defined when they may be used interchangeably. It has developed optimisation criteria for locality, synchronisation and communication in such a manner that allows both loop and data transformations to be used. It has further shown that neither loop nor data transformations can be solely used for efficient parallelisation. This paper has developed a global parallelisation heuristic and has shown that combining non-singular loop and data transformations significantly outperforms existing techniques.

Due to the use of a data partitioning based parallelisation implementation, data alignment transformations have been shown to have significant impact. It remains to be seen if loop alignment will prove as useful in loop-based affinity scheduling. Future work will concentrate on integrating the optimisation approach described in this paper with our previous work on rank-modifying transformations, allowing iteration and data space tiling to be incorporated into one uniform optimisation approach. Future work will also investigate other formulations of the overall optimisation algorithm and examine the interaction of optimisation phases.

References

[1] J.M. Anderson S.P. Amarasinghe and M.S. Lam. **Data and Computation Transformations for Multiprocessors**, Proc. PPoPP, ACM Press, 1995.

[2] U. Banerjee, **Loop Transformations for Restructuring Compilers: The Foundations**, Kluwer Academic Publishers, Boston, 1993.

[3] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali and P. Stodghill, **Solving Alignment using Elementary Linear Algebra**, Proc. Languages and Compilers for Parallel Computing, 1994.

[4] A.J.C. Bik and P.M.W. Knijnenburg, **Reshaping Access Patterns for Improving Data Locality**, Proc. 6th Workshop on Compilers for Parallel Computers, 1996.

[5] F. Bodin and M.F.P. O'Boyle. **A Compiler Strategy for SVM** Proc. 3rd Workshop on Languages, Compilers and Runtime Systems, Kluwer Press, 1995.

[6] S. Chatterjee, J.R. Gilbert, R. Schreiber and S-H. Teng, **Automatic Array Alignment in Data Parallel Programs**, Proc. POPL, ACM Press, 1993.

[7] M. Cierniak and W. Li. **Unifying Data and Control Transformations for Distributed Shared-Memory Machines**, Proc. PLDI, ACM Press, 1995.

[8] D. Gannon, W.Jalby and K. Gallivan, **Strategies for Cache and Local Memory Management by Global Program Transformation**, J. of Parallel and Distributed Computing 5(5), 1988.

[9] J. Garcia, E. Ayguade and J. Labarta, **A Novel Approach Towards Automatic Data Distribution**, Proc. Automatic Data Layout and Performance Predictors, 1995.

[10] M. Kandemir, J. Ramanujam and A. Choudhary, **A Compiler Algorithm for Optimizing Locality in Loop Nests**, Proc. ICS, ACM Press, 1997

[11] M. Kandemir, A. Choudhary, J. Ramanujam and P.Banerjee, **A Matrix Approach to the Global Locality Optimization Problem**, Proc. PACT, IEEE Press, 1998.

[12] K. Kennedy and U. Kremer. **Automatic Data Layout for High Performance Fortran**, Proc. Supercomputing, 1995.

[13] I. Kodukla, N. Ahmed and K.Pingali, **Data-centric multilevel blocking**, Proc. PLDI, 1997.

[14] S.-T. Leung and J. Zahorjan, **Optimizing Data Locality by Array Restructuring**, U. of Washington, Dept. of Comp. Sci. and Eng., Tech. Rep. 95-09-01.

[15] W. Li and K. Pingali. **A Singular Loop Transformation Framework Based on Non-singular Matrices**, Proc. Languages and Compilers for Parallelism, 1992.

[16] K. McKinley, S Carr. and C-W Tseng, **Improving Data Locality with Loop Transformations**, ACM TOPLAS, 1996.

[17] O'Boyle M.F.P. and Bodin F., **Compiler Reduction of Synchronisation in Shared Virtual Memory Systems**, Proc. ICS, ACM Press, 1995.

[18] M.F.P. O'Boyle and P.M.W. Knijnenburg, **Non-Singular Data Transformations: Definition, Validity and Applications**, *accepted for publication in* the International Journal of Parallel Programming.

[19] M.F.P O'Boyle and P.M.W. Knijnenburg, **Integrating Loop and Data Transformations for Global Optimisation**, Proc. PACT, IEEE Press, 1998.

[20] M.F.P. O'Boyle and G.A. Hedayat, **Data Alignment: Transformation to Reduce Communication on Distribu ted Memory Architectures**, Proc. Scalable High Performance Computing Conference, IEEE Press, 1992.

[21] M.E. Wolf and M. Lam. **A Loop Transformation Theory and An Algorithm to Maximise Parallelism**, in *Advances in Languages and Compilers for Parallel Processing*, IEEE Trans. on Parallel and Distributed Systems 2(4), 1991.