A New Program Transformation to Minimise Communication in Distributed Memory Architectures

Michael O'Boyle and G.A. Hedayat

Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK

Abstract

One of the major overheads in implementing programs on distributed memory architectures is communication or non-local access. This paper describes a new transformation technique to minimise redundant non-local accesses . Firstly a criteria for determining data re-use is outlined which provides the basis for a new transformation technique based on the Hermite normal form. Once a non-local data item is accessed , it is stored locally and the computation re-ordered so that no further communications are required. This transformation is extended to the case of multiple array accesses where in general scalar expansion is necessary.

1 Introduction

Compilation should minimise parallel time by utilising machine parallelism and reducing overhead. In compiling for distributed memory architectures (DMAs), the first task is to map data and computation to processors so as to allow parallel evaluation. However in doing so it is possible that large overheads due to load imbalance and communication will be incurred.

Reducing communication overhead by making as many accesses local as is possible is often referred to as exploiting locality. There is in general a tension between parallelism and locality. Mapping all data and computation to one processor implies no communication overhead but also no parallelism. The first step to reduce the number of non-local accesses is the alignment of arrays so as to maximise local accesses. For a discussion of Alignment refer to [11], [13].

Once data has been mapped to the processors, it is desirable to minimise any non-local access by adopting a compiler directed caching policy. This paper presents an invariance analysis for a nest of loops in order to determine opportunities for data reuse. Prefetching transformations are then detailed to take advantage of data reuse. In the context of compiling for DMAs, the nest of loops represents the local program on a processor of a DMA. However, the analysis presented, is applicable to compiler directed cache strategies for shared memory multi-processors with a hierarchy of memory.

Data accessed in a computation may not be invariant of all iterators. Prefetching then requires scalar expansion of the temporaries [14]. Thus there is trade off between space, and the advantage gained by prefetching data. This is in general architecture dependent. In this paper we restrict the size of temporaries to the size of arrays being created. This is referred to as the scalability constraint. In order to satisfy the scalability constraint,while prefetching data, strip-mining may be necessary. Strip-mining is beyond the scope of this paper and is treated elsewhere [13].

2 Representation

In this paper we restrict our attention to parallel nested loops as shown in Fig.1. The loop nest represents the *generic local program* on a processor of a DMA. The loop bounds and array references are restricted to the affine form. The number

$$
\begin{array}{l}
\text{DO } j_1 = l_1, v_1 \\
\quad \text{DO } j_2 = L_2(j_1) + l_2, U_2(j_1) + v_2 \\
\qquad \vdots \\
\qquad \text{DO } j_m = L_m(j_1, \ldots, j_{m-1}) + l_m, \\
\qquad\qquad\qquad U_m(j_1, \ldots, j_{m-1}) + v_m \\
\qquad\quad C(g_1(J), \ldots, g_l(J)) = Get(A(f_1(J), \ldots, f_N(J))) \ldots
\end{array}
$$

Fig. 1. Parallel Nested Affine Loops

of loops is m, L_k and U_k are integer linear combinations of the iterators $J = [j_1, j_2, \ldots, j_{k-1}]^T, k \leq m$ and the computation of array C using array $A, \ldots$ make affine references to the enclosing iterators . i.e. g_k, f_k are affine functions of iterators J. All accesses to potentially distributed data are preceded by the *Get* operator and it is assumed that all writes are local. Using the formalism developed in [17], the bounds of iterators may be described by the polytope

$$\begin{bmatrix} -L \\ \hline U \\ \hline \mathcal{E} \end{bmatrix} J \leq \begin{bmatrix} -l \\ \hline v \\ \hline \epsilon \end{bmatrix} \tag{1}$$

where L and U are $(m \times m)$ lower unit triangular integer matrices l and v are $(m \times 1)$ integer vectors, with kth row L_k, U_k, l_k, and v_k respectively. The inequalities described by $\mathcal{E}, \epsilon$ account for any affine loop body conditionals that are present or which may have introduced as a result of loop transformation. Let V denote the iteration space described by polytope in (1), and define

$$F^T(J) = [f_1(J), \ldots, f_N(J)] \quad , G^T(J) = [g_1(J), \ldots, g_l(J)] \tag{2}$$

then for any $J \in V$ $F(J)$ and $G(J)$, since they are affine, can be represented by

$$F(J) = \mathcal{U}_a J + u_a \quad , G(J) = \mathcal{C} J + c \tag{3}$$

where $\mathcal{U}_a$ and $\mathcal{C}$ are $N \times m$ and $l \times m$ integer matrices respectively, and u_a and c are N and l column integer vectors. We refer to the pair $\mathcal{U}_a$, u_a as occurrence matrix, vector pair for array A.

A computation, or a lattice point $J \in V$ references element $A(q_1, \ldots, q_N)$ of array A if

$$\mathcal{U}_a J + u_a = q \tag{4}$$

where $q = [q_1, \ldots, q_N]^T$. Multiple solutions $J \in V$ of (4) indicatesthat there are multiple access to element $A(q)$ of array A.

3 Invariance

In this section reuse of data is described in terms of the invariant subspace of occurrence matrices. The effect of unimodular loop transformations on the integer basis for the invariant subspace is stated in Proposition 1 while a polynomial time procedure to determine the integer basis is outlined in Proposition 2.

To motivate this analysis, consider the Matrix Multiplication program in Fig.2.

```
DO i = 1, n
   DO j = 1, n
      DO k = 1, n
         C(i, j) = C(i, j) + Get(A(i, k)) * Get(B(k, j))
```

Fig. 2. Matrix Multiplication

The set of points in iteration space V that make access to element $A(q_1, q_2)$ of array A are given by the integer solutions of

$$\begin{bmatrix} 1\,0\,0 \\ 0\,0\,1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \tag{5}$$

that are in the iteration space. The solution of (5) is $i = q_1$, $k = q_2$, and j arbitrary. If we further restrict the solutions that are in iteration space, then $1 \leq j \leq n$. Thus in the above program, element $A(q_1, q_2)$ is accessed by n lattice points, or reused n times. Observe that the occurrence of array A in program in Fig.(2) is invariant of iterator j , and that the multiplicity of solution to equation (5) occurs because $\mathcal{U}_a$,the occurrence matrix of A, has a nonempty null space spanned by $[010]^T$. In this section we formalise these ideas. Let

$$S_a^q = \{J \in Z^m : \mathcal{U}_a J + u_a = q\} \tag{6}$$

Then $|S_a^q \cap V|$ is the number of points in iteration space that access element $A(q)$ of array A. If the system of linear equations

$$\mathcal{U}_a J = q - u_a \tag{7}$$

has no integer solutions, then $A(q)$ is not accessed in the computation. If there is an integer solution, but the solution is unique, then $|S_a^q \cap V| \leq 1$ and there is no reuse of the array element. The interesting case is when solution to (7) is non-unique.

Assuming equation(7) has one integral solution J_0, Smith [19] showed that all integral solutions can be represented as $\{J_0 + \sum_{l=1}^{t} c_l * J_l, c_1 \ldots c_t \in Z\}$, where $J_1 \ldots J_t$ are linearly independent fundamental solutions. i.e. An integral basis for the null space of $\mathcal{U}_a$ of dimension t. $J_1 \ldots J_t$ do not depend on q ,the index of array A, and thus can be computed once for all q. In the next section we show that it is desirable to transform the iterators J so that the invariant basis coincides with the iterators that are innermost. The next proposition shows the effect of such a unimodular change of coordinates on the integer basis for the null space of $\mathcal{U}_a$.

Proposition 1. *Let T be a unimodular transformation of J to $J^{'}$.*

$$J^{'} = TJ \tag{8}$$

Then $TJ_1 \ldots TJ_t$ are integer basis for the null space of $\mathcal{U}_a$ in the new coordinate.

Proof. Proof omitted.

As an example consider matrix multiplication. Suppose loops k and j are interchanged. The unimodular transformation T that achieves the loop interchange is

```
DO i = 1, n
   DO k = 1, n
      DO j = 1, n
         C(i, j) = C(i, j) + Get(A(i, k)) * Get(B(k, j))
```

Fig. 3. Matrix Multiplication: Interchange Loop j & k

then given by

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{9}$$

In the new coordinate the occurrence matrix of A is

$$\mathcal{U}_a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{10}$$

and the integer basis for the null space of $\mathcal{U}_a$ is given by

$$J_1^{'} = TJ_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{11}$$

The next proposition gives a polynomial procedure to determine the integer basis for the null space of occurrence $\mathcal{U}_a$.

Proposition 2. *Assume $\mathcal{U}_a$ is of rank r and let $t = m - r$. Then there exists unimodular matrices $X_{N\times N}$ and $Y_{m\times m}$ such that*

$$X\mathcal{U}_aY = \begin{bmatrix} R_{r\times r} & 0_{r\times t} \\ 0_{N-r\times r} & 0_{N-r\times t} \end{bmatrix} \tag{12}$$

where R is the Hermit normal form of $\mathcal{U}_a$. Moreover an integral basis for the null space of $\mathcal{U}_a$ is given by

$$[J_1, \ldots, J_t] = Y \begin{bmatrix} 0_{r\times t} \\ I_{t\times t} \end{bmatrix} \tag{13}$$

and Y can be computed in polynomial time.

Proof. Proof omitted [19].

In the next section transformations to exploit invariance are discussed.

4 Transformation

Invariance of array occurrence can be exploited by prefetching the array element. This requires

- Loop transformations, so that the invariant basis coincides with iterators that are innermost.
- Introduction of local temporaries to hold prefetched data.

In this section, loop transformation and prefetching to exploit the invariance of a single array occurrence is detailed. Proposition 3 characterises the unimodular loop transformation to make invariant basis coincide with iterators that are innermost. Algorithm 1, then outlines the necessary steps to exploit the invariance of a single array occurrence and forms the building block for more complex cases considered in the next section.

To motivate the presentation consider the following examples. In matrix multiplication, the invariant basis for array A coincides with iterator j. The only loop transformation necessary is that of loop interchange to bring iterator j innermost. This is achieved by transformation T given in (9), resulting in the code shown in Fig.(3) Next the value $A(i, k)$ is prefetched and stored in a temporary before entering the j loop. The resulting code is given in Fig.(4)

```
DO i = 1, n
   DO k = 1, n
      temp = Get(A(i, k))
         DO j = 1, n
            C(i, j) = C(i, j) + temp * Get(B(k, j))
```

Fig. 4. Transformed Matrix Multiplication

The next example illustrates the case where the basis for the null space of the occurrence matrix does not coincide with any of the iterators. In order to exploit invariance, a combination of loop skewing, and loop interchange is now required. The polytope representation of the program in Fig.(5) is given by the system of

```
DO i = 1, n
   DO j = 1, n
      DO k = 1, n
         C(i, j) = C(i, j) + Get(A(k, i + j))
```

Fig. 5. Affine Array Occurrence

inequalities

$$\left[\begin{array}{rrr} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \begin{bmatrix} i \\ j \\ k \end{bmatrix} \leq \left[\begin{array}{r} -1 \\ -1 \\ -1 \\ \hline n \\ n \\ n \end{array}\right] \tag{14}$$

In this example, the occurrence matrix of A is

$$\mathcal{U}_a = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{15}$$

The matrix $\mathcal{U}_a$ has rank $r = 2$ and nullity $t = 1$. Its null space is spanned by $J_1 = [-1, 1, 0]^T$ i.e. J_1 is linear combination of iterators i and j. In order to exploit invariance, a transformation T is desired such that in the new coordinate system, the basis for the null space coincides with the new innermost iterator, or $J_1^{'} = [0, 0, 1]^T$. The following proposition characterises the transformation T in terms of the Hermit normal form factorisation of $\mathcal{U}_a$.

Proposition 3. *Let $X\mathcal{U}_aY$ be the Hermit normal form factorisation of $\mathcal{U}_a$. Define*

$$T = Y^{-1} \tag{16}$$

T is unimodular, and in the new coordinate system the integral basis for the new occurrence matrix of A coincides with the innermost iterators.

Proof. That T is unimodular, immediately follows from (16) and unimodularity of Y. By Proposition 1

$$\left[J_1^{'}, \ldots, J_t^{'}\right] = Y^{-1}\left[J_1, \ldots, J_t\right] \tag{17}$$

Substituting (13) in the right hand side of (17)

$$\left[J_1^{'}, \ldots, J_t^{'}\right] = \begin{bmatrix} 0_{r\times t} \\ I_{t\times t} \end{bmatrix} \square \tag{18}$$

Applying Proposition 3 to the example in Fig.(5), $X = I_{N\times N}$ and Y can be easily found as product of sequence of permutation and skewing transformations

$$Y = E_{32}E_{21}E_{23}(-1) = \begin{bmatrix} 0 & 1 & -1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{19}$$

where E_{ij} is the elementary matrix obtained by interchanging the ith and jth column of identity and $E_{ij}(\alpha)$ is the elementary matrix obtained by adding α times the jth column to the ith column of identity. These correspond to loop permutation and skewing respectively [1]. From (16) and (19) the transformation T is given by

$$T = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{20}$$

Thus the new iterators $J^{'}$ are

$$\left[i^{'}, j^{'}, k^{'}\right]^T = [k, i+j, j]^T \tag{21}$$

and the new transformed polytope can be found using the techniques described in [13]

$$\left[\begin{array}{rrr} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 1 & -1 \\ 0 & -1 & 1 \end{array}\right] \begin{bmatrix} i^{'} \\ j^{'} \\ k^{'} \end{bmatrix} \leq \left[\begin{array}{r} -1 \\ -2 \\ -1 \\ \hline n \\ 2n \\ n \\ \hline -1 \\ n \end{array}\right] \tag{22}$$

The transformed code can be derived as given in Fig.(6)

```
DO i' = 1, n
   DO j' = 2, 2n
      DO k' = 1, n
         IF(j' - k' >= 1)&(j' - k' <= n)
            C(j' - k', k') = C(j' - k', k') + Get(A(i', j'))
```

Fig. 6. Affine Array Occurrence: Transformed Program I

It is worth noting that in the transformed program, occurrence of array A is invariant of the innermost iterator $k^{'}$. By introducing temporaries to prefetch $A(i^{'}, j^{'})$ the program in Fig.(7)is obtained. This constitutes an $O(n)$ reduction in access to array A.

The result of this section can formalised giving the following algorithm for exploiting the invariance of a single array.

```
DO i' = 1,n
   DO j' = 2,2n
      temp = Get(A(i',j'))
         DO k' = 1,n
            IF(j' - k' ≥ 1)&(j' - k' ≤ n)
               C(j' - k',k') = C(j' - k',k') + temp
```

Fig. 7. Affine Array Occurrence: Transformed Program II

Algorithm 1

1. *If $\mathcal{U}$ not full row rank, apply elementary row operation to $\mathcal{U}$ to reduce it to*
$$\mathcal{U} \longrightarrow X \begin{bmatrix} \mathcal{U}' \\ 0_{N-r\times m} \end{bmatrix} \tag{23}$$
2. *Apply elementary column operation $Y_1, Y_2, \ldots$ to $\mathcal{U}'$ to reduce it to Hermit normal form and simultaneously elementary row operation $Y_1^{-1}, Y_2^{-1}, \ldots$ to $m \times m$ identity matrix $I_{m\times m}$.*
$$\begin{bmatrix} \mathcal{U}' \\ 0_{N-r\times m} \end{bmatrix} \longrightarrow \begin{bmatrix} R_{r\times r} & 0_{r\times t} \\ 0_{N-r\times r} & 0_{N-r\times t} \end{bmatrix} \tag{24}$$
and
$$I_{m\times m} \longrightarrow Y^{-1} \tag{25}$$
3. *Set $T = Y^{-1}$, $J' = TJ$*
4. *Find the new loop bounds, and transformed program using polytope representation of the nest of loops .*
5. *Introduce temporaries to prefetch before entering the invariant loop .*

For details of steps 4 and 5 of Algorithm 1 see [13]. So far we have focused on the invariance property of a single array occurrence. Even in the simplest case of matrix multiplication, Fig.(2), reference to arrays A and B both have an invariance property. Moreover, j, the invariant iterator for reference to array A is distinct from i, the invariant iterator corresponding to array B. The algorithm in this section allows us to exploit invariance in either A or B but not both. In the next section, scalar expansion is introduced to take advantage of invariance of multiple array occurrence with distinct invariant iterators.

5 Scalar Expansion

For each array occurrence $\mathcal{U}$ the transformation T from Proposition 3, decomposes the iterators into two disjoint sets. The innermost of which are the invariant iterators corresponding to the array occurrence. Define

$$J' = TJ = \begin{bmatrix} \hat{J} \\ \check{J} \end{bmatrix} \tag{26}$$

In the presence of multiple array occurrences, say A and B, because the transformation T_a is in general different from T_b, the innermost invariant iterators $\check{J}_a$ corresponding to array A may be different from $\check{J}_b$ the invariant iterators of array B. For example in matrix multiplication, it is easy to see that

$$\check{J}_a = j \quad \check{J}_b = i \tag{27}$$

indicating that there are no common invariant iterator for arrays A and B, or

$$\check{J}_a \cap \check{J}_b = \emptyset \tag{28}$$

Either one has to forgo exploiting invariance of both arrays, or allow prefetching through iterators that are not invariant by using scalar expansion. We motivate the presentation by looking at matrix multiplication in greater detail. Decompose the iterators $J^{'}$ into two disjoint set, such that the innermost iterators are the union of invariant iterators of arrays A and B.i.e.

$$\check{J} = \check{J}_a \cup \check{J}_b = \begin{bmatrix} i \\ j \end{bmatrix} \tag{29}$$

Therefore

$$J^{'} = \begin{bmatrix} \hat{J} \\ \hline \check{J} \end{bmatrix} = \begin{bmatrix} k \\ \hline i \\ j \end{bmatrix} \tag{30}$$

Transformation T that maps $J \mapsto J^{'}$ is easily determined as the permutation matrix

$$T = E_{21}E_{32} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{31}$$

The new transformed code is given in Fig.(8). The maximum re-use of data is

DO $k = 1, n$
    DO $i = 1, n$
        DO $j = 1, n$
            $C(i, j) = C(i, j) + Get(A(i, k)) * Get(B(k, j))$

Fig. 8. Matrix Multiplication After Loop Interchange

achieved by pre-fetching with respect to both i and j. The temporaries associated with A and B will be scalar expanded when passing through non-invariant loops. The outer most iterator is k, the reference to array A is scalar expanded by $\check{J} - \check{J}_a = i$ and the reference to B is scalar expanded by $\check{J} - \check{J}_b = j$. The introduction of temporaries after scalar expansion gives the program in Fig.(9). The size of a scalar expanded temporary depends on the range of the iterator it is expanded by. In this case $|tempa| = |tempb| = n$ which constitutes an O(n) reduction in access to arrays A and B.

```
DO k = 1, n
   DO i = 1, n
         tempa(i) = Get(A(i, k))
   DO j = 1, n
         tempb(j) = Get(B(k, j))
   DO i = 1, n
   DO j = 1, n
         C(i, j) = C(i, j) + tempa(i) * tempb(j)
```

Fig. 9. Matrix Multiplication After Prefetching

The procedure outlined above can be easily generalised to multiple arrays in a general loop nest when the basis for null spaces of all the occurrence matrices coincide with the iterators. For a detailed treatment see [13]. Here we observe that, in this case, the transformation T is just a permutation of the iterators so that the union of the invariant iterators of all the array occurrences are innermost.

When the basis for the null space of some the array occurrences are an affine function of the iterators, determining the transformation T is more complex, as the following example demonstrates. Consider the loop nest in Fig.(10) The polytope for

```
DO i = 1, n
   DO j = 1, n
      DO k = 1, n
         C(i, j) = C(i, j) + Get(A(j, k)) + Get(B(i + j, k))
```

Fig. 10. Affine Array Occurrence

the program in Fig.(10) is given by the same system of inequalities as in (14). Here the rank of both occurrence matrices $\mathcal{U}_a$ and $\mathcal{U}_b$ is 2, while the number of iterators $m = 3$. Thus each has a one dimensional null space.

$$J_1^a = [1, 0, 0]^T \;\; J_1^b = [-1, 1, 0]^T \tag{32}$$

Moreover the integral basis for the null space of $\mathcal{U}_b$ is an affine function of iterators i and j. The desired transformation T should be such that in the new coordinate system, the basis for the null spaces of $\mathcal{U}_a$ and $\mathcal{U}_b$ are $[0, 1, 0]^T$ and $[0, 0, 1]^T$ respectively. Similar to section 2.3, this can be achieved if we can construct a unimodular matrix Y whose last two columns are J_1^a and J_1^b. In this example this can be achieved by choosing the first column of Y as $[0, 0, 1]^T$. This vector corresponds to the common row of occurrence matrix of array A and B (or a common reference to iterator k by both arrays).

$$Y = \begin{bmatrix} 0 & 1 & -1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{33}$$

Now choose

$$T = Y^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{34}$$

Note that Y and T are identical to their counterparts in the example of Fig.(5). The new iterators are then

$$J^{'} = TJ = [k, i+j, j]^T \tag{35}$$

and the new polytope representation is the same as in (22). The transformed code is given in Fig.(11). From transformed program I, it is seen that the reference to

$$
\begin{array}{l}
\text{DO } i^{'} = 1, n \\
\quad \text{DO } j^{'} = 2, 2n \\
\quad\quad \text{DO } k^{'} = 1, n \\
\quad\quad\quad IF(j^{'} - k^{'} \geq 1)\&(j^{'} - k^{'} \leq n) \\
\quad\quad\quad C(j^{'} - k^{'}, k^{'}) = C(j^{'} - k^{'}, k^{'}) + Get(A(k^{'}, i^{'})) + Get(B(j^{'}, i^{'}))
\end{array}
$$

Fig. 11. Affine Array Occurrence: Transformed Program I

array A is invariant of the second iterator $j^{'}$, and that the reference to array B is invariant of the innermost iterator $k^{'}$. Therefore

$$\check{J} = \check{J}_a \cup \check{J}_b = \begin{bmatrix} j^{'} \\ k^{'} \end{bmatrix} \tag{36}$$

In order, now, to exploit the invariance of both arrays with distinct invariant iterators, scalar expansion is required. The reference to array A is scalar expanded by $\check{J} - \check{J}_a = k^{'}$, and the reference to B is scalar expanded by $\check{J} - \check{J}_b = j^{'}$. The introduction of temporaries after scalar expansion gives the transformed program II in Fig.(12). As in the matrix multiplication example the size of temporaries depends on

$$
\begin{array}{l}
\text{DO } i^{'} = 1, n \\
\quad \text{DO } k^{'} = 1, n \\
\quad\quad tempa(k^{'}) = Get(A(k^{'}, i^{'})) \\
\quad \text{DO } j^{'} = 2, 2n \\
\quad\quad tempb(j^{'}) = Get(B(j^{'}, i^{'})) \\
\quad \text{DO } j^{'} = 2, 2n \\
\quad \text{DO } k^{'} = 1, n \\
\quad\quad IF(j^{'} - k^{'} \geq 1)\&(j^{'} - k^{'} \leq n) \\
\quad\quad C(j^{'} - k^{'}, k^{'}) = C(j^{'} - k^{'}, k^{'}) + tempa(k^{'}) + tempb(j^{'})
\end{array}
$$

Fig. 12. Affine Array Occurrence: Transformed Program II

the range of the iterators. Thus $|tempa| = n$, and $|tempb| = 2n$, an $O(n)$ reduction in access to arrays A and B.

In the general case when the basis for the null space of all array occurrences are an affine function of the iterators, the unimodular transformation T, to align the iterators with all the invariant basis and move them to innermost position, may not exist. The following counterexample demonstrates this point. Consider the program in Fig.(13) The integral basis for the null space of $\mathcal{U}_a$ and $\mathcal{U}_b$ are

DO $i = 1, n$
  DO $j = 1, i-1$
    DO $k = 1, n$
      $C(i,j) = C(i,j) + Get(A(i-j,k)) + Get(B(i+j,k))$

Fig. 13. Multiple Affine Array Occurrences

$$J_1^a = [1,1,0]^T \; J_1^b = [-1,1,0]^T \tag{37}$$

A unimodular transformation T that will align both J_1^a and J_1^b with the transformed iterators at the innermost position exist if and only if, $Y = T^{-1}$ satisfies

$$J_1^a = Y \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, J_1^b = Y \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{38}$$

Let Y_i denote the ith column of Y. Then (38) implies

$$Y = \left[Y_1, J_1^a, J_1^b\right] = \begin{bmatrix} z_1 & 1 & -1 \\ z_2 & 1 & 1 \\ z_3 & 0 & 0 \end{bmatrix} \tag{39}$$

where z_i are arbitrary integers to be determined, so that Y is unimodular. However

$$detY = 2 \times z_3 \neq \pm 1, \forall z_3 \in Z \tag{40}$$

Thus Y, and hence T cannot be unimodular.

In the previous example, even though a single unimodular transformation to align both J_1^a and J_1^b with the iterators does not exist, each may be aligned using Algorithm 1. This suggests that in the general affine case, the introduction of temporaries should precede the loop transformation to reveal invariance. The idea is illustrated by the example program in Fig.(13). From (37)

$$(J_1^a)^\perp \cap (J_1^b)^\perp = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{41}$$

where $(.)^{\perp}$ denotes the orthogonal complement. Equation (41) indicates that iterator k is not invariant of either array. Thus we scalar expand only with respect to iterators i and j. Let

$$J = \begin{bmatrix} \hat{J} \\ \check{J} \end{bmatrix} = \begin{bmatrix} k \\ \hline i \\ j \end{bmatrix} \tag{42}$$

The loops in program in Fig.(13) are interchanged so that the k loop is outermost, and scalar expanded temporaries are introduced. This results in program given in Fig.(14).

```
DO k = 1, n
   DO i = 1, n
   DO j = 1, i - 1
      tempa(i - j) = Get(A(i - j, k))
   DO i = 1, n
   DO j = 1, i - 1
      tempb(i + j) = Get(B(i + j, k))
   DO i = 1, n
   DO j = 1, i - 1
      C(i, j) = C(i, j) + tempa(i - j) + tempb(i + j)
```

Fig. 14. Multiple Affine Array Occurrences: Transformed Program I

Now each of the loop nests involving *tempa* and *tempb* can be treated separately. Using Algorithm 1

$$T_a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \tag{43}$$

resulting in new iterators

$$T_a J \mapsto [k, i^{'}, j^{'}]^T = [k, i - j, j]^T \tag{44}$$

where $A(i-j, k) = A(i^{'}, k)$ is now invariant of the innermost iterator $j^{'} = j$. Similarly for the loop nest creating *tempb*, the transformation T_b is given by

$$T_b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \tag{45}$$

resulting in new iterators

$$T_b J \mapsto [k, i^{''}, j^{''}]^T = [k, i + j, j]^T \tag{46}$$

where $B(i + j, k) = B(i^{''}, k)$ is now invariant of its innermost iterator $j^{''} = j$. The new transformed program is then given in Fig.(15) Observe that $i^{'}$ and $i^{''}$ range over $[1 \ldots n - 1]$ and $[2 \ldots 2n - 1]$ respectively. Thus the size of temporaries are

```
DO k = 1, n
   DO i' = 1, n - 1
      tempa(i') = Get(A(i', k))
   DO i'' = 2, 2n - 1
      tempb(i'') = Get(B(i'', k))
   DO i = 1, n
   DO j = 1, i - 1
      C(i, j) = C(i, j) + tempa(i - j) + tempb(i + j)
```

Fig. 15. Multiple Affine Array Occurrences: Transformed Program II

$|tempa| = n - 1$ and $|tempb| = 2n - 2$. This constitutes an $\mathrm{O}(n)$ reduction in access to arrays A and B.

Scalar expansion trades space for a reduction in non-local access to reusable data. The space requirement depends on the size of the range of the iterator that is scalar expanded by. We impose the following scalability constraint: All temporaries are at most the size of array C being created. i.e.

$$|tempa_i| \leq |C| \ \forall i \tag{47}$$

In the context of DMAs, the scalability constraint limits the scalar expansion allowed. In order to exploit invariance, within the scalability constraint, strip-mining may be required. The discussion of strip-mining is beyond the scope of this paper and is treated elsewhere [13].

6 Related Work

The work presented in this paper forms part of the general mapping of array computation to distributed memory architectures. In [3], the use of strip-mining and loop interchange is discussed so as to maximise accesses to local cache rather than main memory in a hierarchical memory. In this paper, blocking methods, as studied by numerical analysts, are used to minimise communication to main memory. Although a uni-processor is assumed, it is apparent that the same arguments apply to multi-processor systems.

In [6] the concept of a reference window is introduced. By transforming the program such that local data is used by all that need it, efficiency may be improved. In [23], further work in this area is detailed and an algorithm to transform a restricted set of programs to increase locality is given. In [9] experiments are presented which verify the usefulness of such an approach.

The work presented in this paper exploits invariancy of access for a much larger class of problems than tackled by [23].

7 Conclusion

The work presented in this paper is part of a systematic compilation strategy for distributed memory architectures. In [13] load balancing for more general programs including serial loops and conditionals and arbitrary nesting is presented. Matrix transformations to minimise non-local access by alignment, data partitioning and data re-use are also presented where the interaction between load balancing and communication overhead is investigated. In this paper new program transformations which minimises redundant non-local accesses subject to a space constraint is presented. Such compiler transformations are essential in compiling for distributed memory architectures.

References

1. Banerjee U.: Unimodular Transformations of Double Loops, Proc. of 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine CA, August 1990.
2. Callahan D. and Kennedy K.: Compiling Programs for Distributed Memory Multiprocessors, Journal of Supercomputing, Vol. 2 No. 2, pp 151-207, 1988.
3. Carr S. and Kennedy K.:Blocking Linear Algebra Codes for Memory Hierarchies, Proceedings of the Fourth Siam Conference on Parallel Processing for Scientific Computing , Chicago, Illinois December 1989.
4. Dowling, M.: Optimal Code Parallelization using Unimodular Transformations, Parallel Computing Vol. 16, pp 157-171, 1990.
5. Fox G., Hiranandani S., Kennedy K., Koelbel C., Kremer U., Tseng C-W. and Wu M-Y.: FORTRAN D Language Specification, Rice COMP TR90-141, Department of Computer Science, Rice University, February 1991.
6. Gannon D. and Jalby W.: Strategies for Cache and Local Memory Management by Global Program Transformation, Journal of Parallel and Distributed Computing , pp87 -616 1988.
7. Knobe K., Lukas J.D., and Steele G.L.: Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines, Journal of Parallel and Distributed Computing 8, pp102-118 1990.
8. Kulkarni D., Kumar K.G., Basu A., and Paulraj A.: Loop Partitioning for Distributed Memory Multiprocessors as Unimodular Transformations, Proc. of ACM International Conference on Supercomputing, June, 1991.
9. Lam M.S., Rothberg E.E. and Wolf M.E.: The Cache Performance and Optimizations of Blocked Algorithms, Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.
10. Lamport L.: The Parallel Execution of DO loops, CACM Vol. 17 No. 2, Feb 1974.
11. Li J. and Chen M.: Index Domain Alignment: Minimising Cost of Cross-Referencing between Distributed Arrays, IEEE Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, October 1990.
12. Lu L-C.: A Unified Framework for Systematic Loop Transformations, 3rd ACM Symposium on the Principles and Practice of Parallel Programming, April 1991.
13. O'Boyle M.F.P.: Program and Data Transformations for Efficient Execution of Array Computation on Distributed Memory Architectures, PhD thesis, University of Manchester, Dept. of Computer Science, January 1992.
14. Padua D.A. and Wolfe M.J.: Advanced Compiler Optimizations for Supercomputers, Communications of the ACM Vol.29 No.12 pp 1184-1201, December 1986.
15. Pugh W.: Uniform Techniques for Loop Optimisation, International Conference on Supercomputing, June 1991.
16. Ramanujam J. and Sadyappan P.: Tiling of Iteration Spaces for Multicomputers, Proc. of International Conference on Parallel Processing, Vol. 2, pp 179 -186, 1990.

17. Ribas, H.B.: Automatic Generation of Systolic Programs from Nested Loops, Carnegie-Mellon Tech. Rep. CMU-CS-90-143, June 1990.
18. Rogers A.: Compiling for Locality of Reference, Technical Report TR 91-1195, Department of Computer Science, Cornell University, March 1991.
19. Schrijver A.: Theory of Linear and Integer Programming, Wiley, Chichester 1986
20. Wolfe M.: Loop Skewing: The Wavefront Method Revisited, International Journal of Parallel Programming, Vol. 15, No. 4 pp 279-294, August 1986.
21. Wolfe M.: Massive Parallelism through Program Restructuring, 3rd Symposium on the Frontiers of Massively Parallel Computation, pp 407-415, October 1990.
22. Wolf M.E. and Lam M.: An Algorithmic Approach to Compound Loop Transformations, in Advances in Languages and Compilers for Parallel Processing, eds Nicolau A., Gelernter D., Gross T. and Padua D., Pitman London, 1991.
23. Wolf M.E. and Lam M.: A Data Locality Optimizing Algorithm, ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1991.

This article was processed using the LaTeX macro package with LMAMULT style