Data Alignment

Guei-Yuan Lueh
School of Computer Science
Carnegie Mellon University

1. Introduction

FX is a data parallel language based on Fortran 77 plus array extensions and other user directives. The FX compiler we develop compiles this language to the iWarp. One of the main goals of the compiler is to distribute data across processors of the iWarp so that each processor executes the same piece of code but on different data segment.

According to the owner-computes-rule, the owner processor must copy all arguments locally for a computation. For example, consider the code A(i) = B(i-1) + C(i). Assume A(i), B(i) and C(i) are stored on the same processor, and A(i) and B(i-1) do not reside on the same processor. To execute the statement, the nonlocal data B(i-1) is moved to the processor that owns A(i) such that all arguments, B(i-1) and C(i), are local to the processor. Runtime communication is required to move nonlocal data. Thus, how well data is distributed across the processors has a great influence on the cost of runtime communication.

There are two phases in the compiler related to determining data distribution, data alignment and data distribution. The data alignment phase maps data that are used together into the same logical processors. The data distribution phase decides which physical processors hold which array elements based on the information produced by the data alignment phase. This report describes the data alignment phase which provides the information of how array slices are aligned with each other to the distribution phase.

In the array extensions, it is allowed to use array slices which are composed of subscript-triplet, $l : u : s$. A slice designates a regular sequence of subscripts. The sequence for $l : u : s$ is l, $l+s$, $l+2s$, ..., $l+ks$, where $l+ks \leq u$ and k is a constant. If s is omitted, it is equivalent to a stride of 1. The data alignment detects if two parallel array slices can be aligned. Two array slices are aligned implies that corresponding elements of the arrays slices are logically stored on the same processor. An elementwise operation on these aligned array slices can be performed without interprocessor communication.

2. Alignment Graph

The algorithm of the data alignment operates on the alignment graph. Nodes of the graph represent array occurrences in an FX program which contain array slices. An array occurrence is the appearance of an array in the program. Arcs of the graph represent preferences of which array slices would like to be aligned. The algorithm tries to merge array slices together based on the arcs of the graph. Similar algorithms have been designed and developed by [1], [2], [3] and [4].

There are two kinds of arcs: conformance and identity. Conformance arcs go from the LHS array occurrences to the RHS array occurrences in the same statement. Identity arcs go from defining array occurrences(LHS) to the use(RHS) of the same arrays that follows. An example for conformance arcs is given in Fig. 1. The arcs indicate preferences of aligning corresponding elements of distinct arrays.

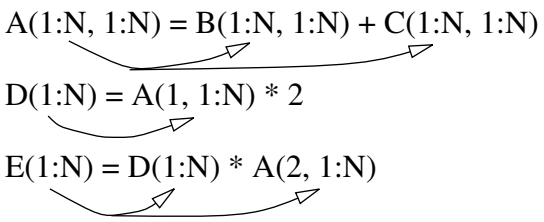


Fig. 1 Conformance arcs

An example for identity arcs is given in Fig. 2. An identity arc between two array occurrences indicates a preference of aligning identical elements of the same array to the same processors.

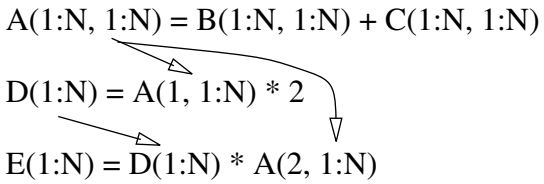


Fig. 2 Identity arcs

For the code in Fig. 3, since the variable x is unknown at compile time, there is no way tell if A(x, 1:N) defines values for the use of A(1, 1:N). Under this circumstance, an identity arc is produced for the two array occurrences. While checking alignment preferences, explained in section 3, those alignment preferences that have the variable x will not be maintained. That is, data movement is required in this case even if the value of x is 1.

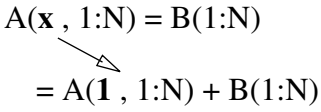


Fig. 3 Unknown variable x at compile time

3. Alignment Merging

From the alignment graph, the data alignment phase tries to find how many alignment preferences can be maintained without introducing runtime communication. To maintain alignment preferences is done via the process of *bundle merging* [4]. The process is called *cell merging* in

[1]. The data alignment phase creates a bundle for each dimension of array occurrences. Then it tries to merge corresponding bundles together based on the preference arcs. Conformance arcs are examined prior to identity arcs. To optimize the common case, identity arcs of the most likely flow are merged first. Bundles are gradually merged into one big merged bundle, if all alignment preferences can be maintained. When an alignment preference can not be satisfied, a conflict arc is introduced to indicate the conflicting alignment request. There are four types of conflicts:

offset conflict: For two array slices, their starting offsets are off by a constant. For instance, assume A(i) resides on the same processor as B(i), then A(**1**:N) and B(**2**:N+1) have *offset* 1 conflict. Data needs to be shifted to satisfy the alignment request.

stride conflict: For two array slices, the stride ratio between the slices differs. For instance, assume A(1:N:**1**) and B(1:N:**1**) are aligned. The stride ratio is **1** between A and B. If we try to merge A(1:N:**1**) and B(1:2N:**2**) (stride ratio is **2**), then a stride conflict exists. In this case, data needs to be compressed or expanded along the dimension.

dimension conflict: The corresponding array slices are not distributed along the same dimensions. For instance, assume array A's first dimension is distributed along array B's first dimension, then A(**1:N**,1) and B(1,**1:N**) have a dimension conflict.

scalar conflict: The scalar array slices of array occurrences are not aligned. For instance, assume A(**1**, 1:N) and B(1:N) are aligned, a scalar conflict exists between A(**2**, 1:N) and B(1:N). In this case, array B needs to be shifted from the processors that own the first row of A to the processors that own the second row of A.

Multiple conflicts may exist between two array occurrences. For instance, assume A(1, 1:N:1) and B(1:N:1) are aligned, then, for A(2, 1:N:1) and B(2:2N+1:2), there are *scalar*, *offset* and *stride* conflicts.

The algorithm of the bundle merging is given below. The code circled by dashed lines is used to handle scalar conflicts. Because a scalar conflict is associated with array occurrences instead of bundles, detecting scalar conflicts is handled separately. Those code is described in

details in section 4.2

```
bundle_merging()
{
    /* merge conformance arcs */
    For each conformance arc between two array occurrences, A and B, do
        For each bundle b1 of array slices of A do
            find the corresponding bundle b2 of array slice of B
            and determine if merging b1 and b2 causes conflicts(offset, stride or dim)
            if conflicts exist then
                For each conflict, a conflict arc is built between b1 and b2.
            else
                b1 and b2 are merged into a merged bundle.

        /* details are described in section 4.2.1 */
        determine scalar conflict between A and B, and also
        determine if A and B can be combined into one group
        The routine for this part of code is called conformance_group_occurrences()

    For each merged bundle which contains more than one bundle
        compute relative mapping positions and stride ratios among bundles of the
        merged bundle. The information is used in merging identity arcs to ensure
        that the relative position and stride ratio between two bundles stay the same
        after merging an identity arc.

    /* merge identity arcs */
    Iterate over loops L from the innermost loop to the outermost loop
            For each identity arc between array occurrences A and A' inside the loop L
            (A and A' refer to the same array)

                /* details are described in section 4.2.1 */
                determine scalar conflict between the two groups that owns A and A'
                The routine for this part of code is called check_identity_conflict().

                For each bundle b1 of A do
                    find b1's corresponding bundle b2 of A'
                    if b1 belongs to a merged bundle
                        B1 = the merged bundle
                    else B1 = b1
                    if b2 belongs to a merged bundle
                        B2 = the merged bundle
                    else B2 = b2
                    determine if merging B1 and B2 causes conflicts(offset, stride or
                                                                    dimension)
                    if conflicts exist then
                        For each conflict, a conflict arc is built between B1 and B2.
                    else
                        B1 and B2 are merged into a merged bundle.

                /* details are described in section 4.2.1 */
                if there is no conflict(dim, scalar, offset or stride) caused by the identity
                arc then
                    call the routine identity_group_occurrences() to merge the two
                    groups.

}
```

An example for the process of merging bundles is given in Fig. 4. The notation A^n_m indi-

```
A( 1:N , 1:N ) = 0                                          ---S1
  A1_1   A2_1
C( 1:N , 1:N ) = A( 1:N , 1:N ) + B( 1:N , 1:N )            ---S2
  C1_1   C2_1        A1_2   A2_2        B1_1   B2_1
D( 1:N , 1:N ) = A( 1:N , 1:N ) + B ( 2:N+1 , 1:N )         ---S3
  D1_1   D2_1        A1_3   A2_3          B1_2   B2_2
```

Fig. 4 Example of bundle merging

cates the bundle of *n* th dimension at *m* th occurrence of array A. The algorithm merges conformance arcs first. In statement S2, the conformance arcs go from occurrence C to occurrence A and occurrence C to occurrence B. For the former arc, the algorithm tries to merge C^1_1 with A^1_2 and C^2_1 with A^2_2. No conflict exists for merging the conformance arc because C and A refer to different arrays. After the process of merging conformance arcs, another graph among bundles is formed (Fig. 5). Each circle indicates a merged bundle. During merging identity arcs, the algorithm tries

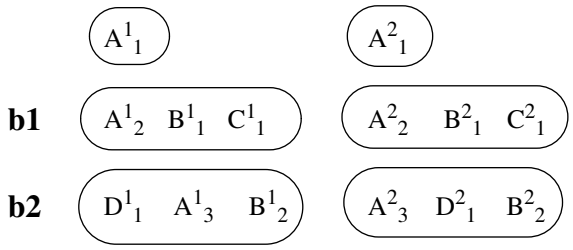


Fig. 5 Merged bundle after merging conformance arcs

to merge bundles **b1** and **b2**. B^1_1 relative to A^1_2 and B^1_2 relative to A^1_3 are off by constant 1 so an offset conflict exists. After merging identity arcs, we get the graph of Fig. 6. Arcs of the graph indicate conflicting alignment requests. From the graph, the later distribution phase knows that array B needs to be shifted before statement S3 is executed.

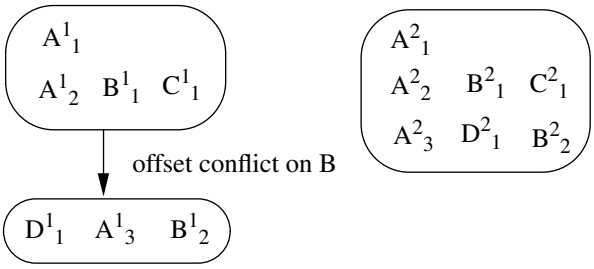


Fig. 6 Merged bundle graph

4. Lazy Alignment Decision

In the data alignment phase, two decisions are made during the process of bundle merging,

data movement and scalar conflict. For the data movement, when a conflict arises, the decision of moving which data/array to satisfy the constraint is made. For a scalar conflict between two different arrays, the decision of which dimensions of one array are distributed along the dimensions of the other is also made. Making the decisions without a good global view may cause unnecessary runtime communication to move data, which is described in sections 4.1 and 4.2. If we make the data movement decision whenever a conflict arises, then the decision we make is insensitive to the bundles that are not yet handled by the bundle merging process, and may result in extra data movements. If we make the decision of choosing dimensions to distribute whenever a scalar conflict needs to be detected, the decision we make is also insensitive to the array occurrences that are yet checked for scalar conflict, and may result in more conflict arcs.

To make good decisions on data movement and scalar conflict, the time of making these decisions is postponed till when it is necessary. More array occurrences and bundles are analyzed during delaying the decisions. This provides us with more accurate information to make better decisions.

4.1. Data Movement

In the example of Fig. 4, when a conflict arises, a decision of which array needs to be moved is made right away[1], e.g. an offset conflict on B. In fact, moving either array A or array B can satisfy the alignment constraint. However, choosing one from them without a global view of array occurrences may result in a poor overall choice. For the code in Fig. 7, like the example in Fig. 4, an offset conflict exists between the two array occurrences B(1:N) and B(2:N+1). Moving array A or array B does not affect the cost of runtime communication. However, if more array occurrences

```
A(:) = ...
B(:) = ...
E(1:N) = A(1:N) + B(1:N)
F(1:N) = A(1:N) + B(2:N+1)
```

Fig. 7 Example of data movement

appear in the code, a good choice can reduce the runtime communication cost. In Fig. 8, array occurrences C and D appear in the code. The choice of moving array B requires to move array C and D as well. On the contrary, moving array A does not require to move any other array. We

```
A(:) = ...
B(:) = ...
E(1:N) = A(1:N) + B(1:N) + C(1:N) + D(1:N)
F(1:N) = A(1:N) + B(2:N+1) + C(2:N+1) + D(2:N+1)
```

Fig. 8 Example of data movement

hope the decision of which arrays need to be moved can be postponed till the distribution phase. At that point, analysis of data alignment is done, so choosing arrays to move without introducing

extra communication cost becomes clear and easy. To achieve this, the representation of conflict arcs is modified as follow.

Let's consider an *offset* conflict. Same algorithm can apply to the others. When the relative position between two arrays, e.g. arrays A and B, differs in two merged bundles, an offset conflict exists between the two bundles. Instead of determining which array causes the conflict, the conflict arc is annotated as "offset conflict caused by array A or B". For the code in Fig. 8, the final merged bundle graph is given in Fig. 9.

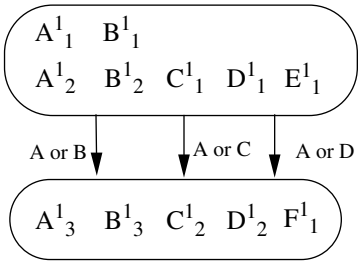


Fig. 9 modified merged bundle graph

There are three offset conflict arcs, "A or B", "A or C" and "A or D". No conflict among arrays B, C and D indicates that corresponding elements of these arrays logically reside on the same processors. After the data alignment phase, the distribution phase then chooses the arrays who appear most frequently in the conflict arcs. In this case, array A is chosen to be moved along the dimension to maintain the alignment request.

Another example is given in Fig. 10. It shows that the approach is always able to make a good choice in terms of communication cost. The relative mapping position between $A^1{}_2$ and $B^1{}_1$ doesn't hold in merged bundle **b2** so an offset conflict is introduced between **b1** and **b2** which is annotated with "A or B". Likewise we can derive offset conflict arcs, "A or C", "A or D", "B or K", "C or K" and "D or K". Since arrays A and K are the most frequent ones in the arcs, the distribution phase will pick up array A and K instead of array B, C and D.

```
A(:) = ...
E(1:N) = A(1:N) + B(1:N) + C(1:N) + D(1:N) + K(1:N)
F(1:N) = A(1:N) + B(2:N+1) + C(2:N+1) + D(2:N+1) + K(1:N)
```

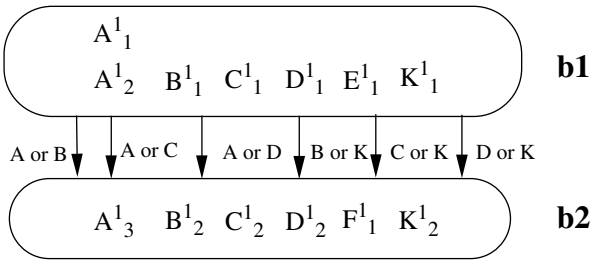


Fig. 10 Example

4.2. Scalar Conflict

While dealing with scalar conflict, we found out that the decision of matching scalar slices also has an impact on the runtime communication cost. For simplicity, all scalar slices of the code below are constants. The same observation also applies to variables. If the first dimension of array B is distributed along the first dimension of A, then there is a scalar conflict. However, if the first dimension of array B is distributed along the second dimension of array A, then the conflict does not exist any more. At the point of merging the conformance arcs, since we only look at array occurrences of the same statement, we don't take into consideration of array occurrences of other statements. Thus, when merging the conformance between arrays A and B of statement S1, we have no idea of which dimension of array A along which the first dimension of array B should be distributed In the Connection Machine compiler[1], one dimension of array A is picked right away to match with the first dimension of array B.

```
A(1, 1, 1:N) = B(1, 1:N)                ---S1
C(1:N) = A(1, 2, 1:N) + B(2, 1:N)       ---S2
```

When scalar variables are in both array occurrences, e.g. A(i,j,1:N) and B(i, 1:N), the dimensions having the same variables are matched. Thus the first dimension of array A is aligned with the first dimension of array B. This is a reasonable good approach if variable **i** is an innermost loop index. In other cases, it may not be a good choice. An example is illustrated later in Fig. 13. Besides, in a case like the code of Fig. 11, if the first dimension of B is distributed along the second dimension of A, no data movement is required inside the loop. Again when merging the conformance between A(i, i, 1:N) and B(i, 1:N), a guess is needed due to lack of a global view about array occurrences of other statements. Here we also try to implement a lazy decision. In other words, we hope to postpone the decision till the time as needed. As postponing the decision, more array occurrences are analyzed, so we may know what a good choice would be. For example, if the decision is delayed till the identity arc between A(i, i, 1:N) and A(i, i - 1, 1:N) is merged, all four array occurrences are available for determining which dimension of A along which the first dimension of B should be distributed.

```
DO i = 1, N
    A( i , i , 1:N) = B( i , 1:N)
      = A( i , i - 1 , 1:N) + B( i - 1 , 1:N)
ENDDO
```

Fig. 11 Scalar variables

4.2.1 Algorithm

There are some notations used in the algorithm:

loc A(x) : The virtual processor on which element A(x) resides, where x is a vector.

dis A(x) : loc A(x1) - loc A(x2) = dis A(x1 - x2)

"Δ" : The symbol "Δ" indicates the dimension that "Δ" associates is ignored.

The process of dealing with scalar conflict is different to that of merging bundles because a scalar conflict is associated with array occurrences instead of array slices. The process combines array occurrences into groups by stepping through the alignment arcs. Like merging bundles, conformance arcs are visited prior to identity arcs. A group is a set of array occurrences which have consistent distributions. Array occurrences with consistent distributions means that no conflict among the bundles of the array occurrences.

The motivation of using *loc* and *dis* is to define the relation of positions of array elements if the arrays are logically distributed. For example, if A(1 , 1:N) logically resides on the processors that owns B(2 , 1:N), we get the equation *"loc* A(1 , Δ) = *loc* B(2 , Δ)". This tells that the second dimension of both arrays is ignored and the first row of A is aligned with the second row of B. If we have another group which have array occurrences A(3 , 1:N) and B(4 , 1:N), we get the equation *"loc* A(3 , Δ) = *loc* B(4 , Δ)". If the relative positions between both array elements derived from the two equations stay the same, it implies that the distribution of the two arrays is still the same. Therefore no scalar conflict exists. The equation "*loc* A(3 , Δ) - *loc* A(1 , Δ)" indicates that the distance between the third row and the first row, which is 2 rows. The distance of "*loc* B(4,Δ) - *loc* B(2 , Δ)" must also be 2 rows if the relative positions between both array elements stay consistent. Hence from the equation "*dis* A(2 , Δ) = *dis* B(2 , Δ)", we can tell if the relation will hold or not.

Assume we only know of A(1,1,1:N) logically resides on the processors that owns B(1,1:N) and have no idea of the first dimension of B is aligned with which dimension of A. However, the equation "*loc* A(1 , 1 , Δ) = *loc* B(1 , Δ)" still holds. Likewise for A(1, 4, 1:N) and B(4 , 1:N), we have the equation "*loc* A(1 , 4 , Δ) = *loc* B(4 , Δ)". From "*loc* A(1 , 4, Δ) - *loc* A(1 , 1 , Δ)", we know the array index of the second dimension is increased by 3. If the first dimension of B is aligned with the second dimension of A, then the increase from loc B(1 , Δ) to B(4 , Δ) must also be 3 to keep the relation of position hold. We can do dimension combinations from the equation "*dis* A(0 , 3 , Δ) = *dis* B(3 , Δ)" to find aligning the first dimension of B with which dimension of A does not cause a scalar conflict. The process of dimension combinations is called matching dimension.

Whenever two groups of array occurrences are considered to be combined, we need to find if a common pair of distinct arrays is in both groups because this affects the consistent distributions of both groups. If a common pair of distinct arrays does exist, then checking scalar conflict between the two arrays is needed. In checking scalar conflict, matching dimensions between the arrays is performed to avoid scalar conflicts. If a scalar conflict is still unavoidable after the process of matching dimensions, no decision of matching dimensions between the arrays is made because there may have a chance to avoid scalar conflicts with other groups. The algorithm is in

Fig. 12.

```
conformance_group_occurrences(occurrence A, occurrence B)
{
    /* First, detect a case of scalar conflict like, A(1:N) = B(1, 1:N)+ B(2, 1:N) */
    From the group that owns A, find if any occurrence B' refers to the same array as B
    if B' exists and the scalar variable of B' and B are not the same then
        return

    detect if any conflicts among bundles of A and B    /* ensure consistent distributions */
    if no conflict exists then
        for each array slice of B do
            find its corresponding slice of A and then put the two slices' dimension into
            their corresponding entry of the matching table of A.
    add B into A's group
}

check_identity_conflict(occurrence A, occurrence B)
{
    /* A and B both refer to the same array */
    for every pair of array occurrences o1 and o2 that are in A's and B's groups respectively
    and refer to the same array but not the array of A
        derive loc equations for (A, o1) and (B, o2).
        use the loc equations to form the dis equation.
        using the dis equation, find a matching dimension between A's and o1's arrays
        without introducing any scalar conflict.
        if a matching dimension is found then
            record the solution of the matching dimension on the matching tables of A
            and B.
        else
            construct a scalar conflict arc.
    if any scalar conflict is constructed then
        return SCALAR_CONFLICT
    else return NO_CONFLICT
}

identity_group_occurrences(occurrence A, occurrence B)
{
    G1 and G2 are the groups that own A and B respectively.
    G1's matching table = union of G1's and G2's matching tables.
    add G2 to G2.
}
```

Fig. 12 Algorithm of handling scalar conflicts

The code below illustrates the process. The routine *conformance_group_occurrences*() is called when every conformance arc is visited. Because no conflict exists in any statement, the routine puts all occurrences of a statement into one group. Let's use Gn to indicate the group of all occurrences of statement Sn.

```
B(: , : , :) = ...                      --- S1
C(: , : , :) = ...                      --- S2
... = B(1 , 1, 1:N) + C(1 , 1:N)        --- S3
... = B(2 , 1, 1:N) + K(2 , 2, 1:N)     --- S4
... = K(1 , 3, 1:N) + C(2, 1:N)         --- S5
```

There are four identity arcs, (S1, S3), (S1, S4), (S2, S4) and (S2, S5), in the code. (S1, S3) means that an identity arc goes from the LHS of statement S1 to the RHS of the same array of statement S3. Checking scalar conflict is not needed for grouping G3 and G4 because no common pair of distinct arrays exists in both G3 and G4. After the identity arcs, (S1, S3), (S1, S4) and (S2, S4), are visited, G1 to G4 are combined into one big group. Hence, to this point, all array occurrences of S1 to S4 are in one group and occurrences K and C of S5 are in another group G5. When these two groups are considered to be grouped, the process of checking scalar conflict is invoked since the pair of arrays K and C is in both groups. The first step is to derive equations from these groups which contain **loc K** and **loc C**. From S3, we can derive

$$\text{loc } B(1\ ,\ 1\ ,\ \Delta) = \text{loc } C(1\ ,\ \Delta) \qquad \text{--- (1)}$$

Similarly from S4 and S5, we can derive

$$\text{loc } B(2\ ,\ 1\ ,\ \Delta) = \text{loc } K(2\ ,\ 2\ ,\ \Delta) \qquad \text{--- (2)}$$

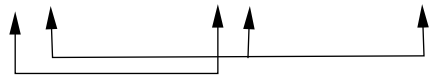
$$\text{loc } K(1\ ,\ 3\ ,\ \Delta) = \text{loc } C(2\ ,\ \Delta) \qquad \text{--- (3)}$$

The reason we use the symbol "Δ" is because those dimensions do not have any effects on scalar conflicts. Therefore, those dimensions are skipped/ignored in the process of matching dimensions among arrays. From (2) - (1), we derive another equation

$$\text{loc } K(2,\ 2,\ \Delta) = \text{dis } B(1,\ 0,\ \Delta) + \text{loc } C(1,\ \Delta) \qquad \text{--- (4)}$$

Then from (4) - (3), we get

$$\text{dis } K(1,\ -1,\ \Delta) = \text{dis } B(1,\ 0,\ \Delta) + \text{dis } C(-1,\ \Delta)$$



For simplicity, we use the notation A^n to indicate the n th dimension of array A. To satisfy the equation, we need to align K^1 with B^1 and also align K^2 with B^2 and C^1. Here we find a distribution for the code which does not have any scalar conflict. The result is then kept in the matching table of the group. The table forces future processes of matching dimension to follow the decision we have made so far. In other words, while finding a combination of dimensions to satisfy the final equation, the relations recorded in the table must also sustain.

Another example for scalar variable slices is given in Fig. 13.Since K is the innermost loop

index, A^3 is aligned with B^3. Using the approach we propose, we derive equations

```
DO I = ...
  DO J = ...
    DO K = ...
      A( I, J, K, 1:N) = B(I, J, K, 1:N)
       = A(I-1, J, K, 1:N) + B(I, J-1, K, 1:N)
    END DO
  END DO
END DO
```

Fig. 13 Example of scalar variables

$$\text{loc } A(I, J, K, \Delta) = \text{loc } B(I, J, K, \Delta) \qquad \text{--- (5)}$$
$$\text{loc } A(I-1, J, K, \Delta) = \text{loc } B(I, J-1, K, \Delta) \qquad \text{--- (6)}$$

From (6) - (5), we get

$$\text{dis } A(-1, 0, 0, \Delta) = \text{dis } B(0, -1, 0, \Delta)$$

Because we predetermine that A^3 is aligned with B^3, the only solution to satisfy the equation is aligning A^1 with B^2 and A^2 with B^1. In this way, no conflict is in the innermost loop. However, for every J iteration, we need to move either array A or B so that A(I, J, K, 1:N) reside in the same processors as B(I, J, K, 1:N). If we determine aligning dimensions of array A and B by matching the same loop index variables[1], a scalar conflict exists for every iteration of K but no data movement is needed for loop J. It is most likely loop K is heavily executed, the former choice, therefore, is better than the latter in most common cases.

The code of the data alignment phase is in appendix.

References

[1] Katherine Knobe, Joan D. Lukas, and Guy L. Steel Jr. “Data optimization: Allocation of arrays to reduce communication on simd machines”. Journal of Parallel and Distributed Computing, pp 102 - 118, 1990

[2] Manish Gupta and Prithviraj Banerjee. “Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers”. IEEE Transactions on Parallel and Distributed Systems, pp 179 - 193, 1992

[3] Jinke Li and Marina Chen. “Index domain alignment: Minimizing cost of cross-reference between distributed arrays”. In Frontiers 90: 3rd Sysposium Frontiers Massively Parallel Computation, 1990.

[4] Skef Wholey. “Automatic Data Mapping for Distributed-Memory Parallel Computers”. Ph.D. Thesis Carnegie Mellon University, 1991.