Array Operation Synthesis to Optimize HPF programs*

Gwan-Hwan Hwang   Jenq Kuen Lee
Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan

Dz-Ching R. Ju
Hewlett-Packard Company, Cupertino, CA 95014, USA

Abstract

An increasing number of programming languages, such as Fortran 90, HPF, and APL, are providing a rich set of intrinsic array functions and array expressions. These constructs which constitute an important part of data parallel languages provide excellent opportunities for compiler optimizations. The synthesis of consecutive array operations or array expressions into a composite access function of the source arrays at compile time has been shown[2] that it can reduce the redundant data movement, temporary storage usage, and loop synchronization overhead on flat shared memory parallel machines with uniform memory accesses. However, it remains open how the synthesis scheme can be incorporated into optimizing HPF programs on distributed memory parallel machines by taking into account communication costs. In this paper, we propose solutions to address this open problem. We first apply the array synthesis scheme (developed earlier by us for Fortran 90 programs) to HPF programs and demonstrate its performance benefits on distributed memory machines. In addition, to prevent a situation we call "synthesis performance anomaly", we derive a cost model and present an optimal solution based on the cost model to guide the array synthesis process on distributed memory machines. We also show that the optimal problem is NP-hard. Therefore, we develop a practical heuristic algorithm for compilers to devise synthesis strategy on distributed memory machines with HPF programs. Experimental results show significant performance improvement over the base codes for HPF code fragments from real applications on a DEC Farm by incorporating our proposed optimizations. The array operation synthesis scheme is demonstrated to be equally effective for programs running on parallel distributed memory machines.

Keywords: HPF Compiler, Array Operation Synthesis, Distributed Memory Machines, Compiler Optimization, Fortran 90 Programs

1 Introduction

An increasing number of programming languages, such as Fortran 90[4], HPF (High Performance Fortran)[1], APL, and possibly C++ with built-in array classes[7], provide a rich set of intrinsic array functions and array constructs. These intrinsic array functions and array expressions operate on the elements of a multi-dimensional array object concurrently without requiring iterative statements. These array operations provide a rich source of data parallelism. Multiple consecutive array operations specify a particular mapping relationship between the source arrays and final target array. A straightforward compilation for these consecutive array functions or array expressions may translate each array operation into a (parallel) nested loop and use a temporary array to pass intermediate results to subsequent array functions. Synthesis of multiple consecutive array functions or array expressions can compose several data access functions into an equivalent composite reference pattern. Thus, the synthesis can improve performance by reducing redundant data movement, temporary storage usage, and parallel loop synchronization overhead.

*The correspondence author is Jenq Kuen Lee. His e-mail address is jklee@cs.nthu.edu.tw and FAX number is 886-35-723694.

Much of earlier work on optimizing array language programs can be traced back to the research done for compiling APL programs. Guibas and Wyatt[12] first proposed a universal selector representation called a stepper to encode the way in which elements of the current function take part in the formation of the final target. The data structure in a stepper consists of four arrays which specify the corresponding axis, start, and stride in the source array and also the size of each target axis. A number of array optimization schemes have since been developed based on the stepper representation. Recently, we develop a functional approach to perform array operation synthesis for Fortran 90 programs on flat shared memory machines with uniform memory accesses[2]. Our scheme is based on the composition of mathematic functions. It starts with deriving a mathematic access function for each array expression or intrinsic array function. We then use a function composition approach to compose those access functions. This composition is semantically equivalent to the synthesis of array functions. Our scheme can handle data accesses between arrays of different numbers of dimensions, masked array expressions, multiple-source array operations, array location operations, array reduction operations, and multiple-clause array functions. As a result, we can handle compositions of extensive Fortran 90 array constructs, such as RESHAPE, SPREAD, EOSHIFT, TRANSPOSE, CSHIFT, and MERGE functions, array sections, array reduction functions, WHERE and ELSEWHERE constructs. None of existing methods can synthesize such an extensive set of array operations.

In spite of the success of the synthesis scheme for Fortran 90 programs on flat shared memory machines, it remains open how the synthesis scheme can be incorporated into optimizing HPF programs on distributed memory parallel machines by taking into account communication costs. In this paper, we propose solutions to address this open issue. Currently, in the process of compiling HPF programs, a compiler partitions the program using the owner-computes rule[18]. An explicit communication is necessary for a remote memory access. Array operation synthesis is able to reduce the number of memory references including loads and stores[3]. However, in distributed memory machine, the memory references include both local references and remote references (communication). A remote memory access usually takes far longer than a local memory access. Although the array operation synthesis can reduce the total number of memory references, a naive application of the synthesis process adopted from flat shared memory parallel machines may actually increase the amount of remote memory loads for programs running on distributed memory machines, which may result in an adverse effect on execution time performance. We call such a behavior "synthesis anomaly". In this paper, we first apply the array synthesis scheme (developed earlier by us for Fortran 90 programs) to HPF programs and demonstrate its potential performance benefits on distributed memory machines. To prevent the synthesis anomaly, we derive a cost model and present an optimal solution based on the cost model to guide the array synthesis process on distributed memory machines. In addition, we also show that the optimal problem is NP-hard. Therefore, we develop a practical heuristic algorithm for compilers to devise synthesis strategy on distributed memory machines with HPF programs. Our strategy is to synthesize the continuous array operations as many as possible. Then, we use the heuristic algorithm to guide the process of code generation. The heuristic algorithm may decide to roll back some of the temporary arrays to reduce synthesis anomaly behaviors. To demonstrate the effects of our proposed algorithms, we have conducted experiments on an 8-node DEC Farm with an HPF compiler. Experimental results show speedups from 1.60 to 8.0 for HPF code fragments from real applications on a DEC Farm by incorporating our proposed optimizations.

The remainder of this paper is organized as follows. Section 2 gives motivating examples to explain the work this paper proposes to solve. For the sake of completeness, we provide an overview of our earlier proposed array operation synthesis scheme in Section 3. In Section 4, we then propose a cost model and an optimal algorithm for array operation synthesis on distributed memory machines. Because the optimal algorithm is NP-hard, a heuristic algorithm is presented in Section 5. Section 6 includes the experimental results and implementation of the synthesis scheme, and Section 7 concludes the paper.

2 Motivating Examples

In this section, we use examples to motivate the need of our optimization mechanism for HPF programs. Our work deals with the array operation synthesis with HPF programs on distributed memory parallel machines. Let us first consider the HPF code fragment below. Parallelism is expressed with array operations including array section move, transpose, reduction, and spread operations. The goal of the compilation is to produce efficient codes on distributed memory parallel machines.

HPF Code Fragment 1

```
          REAL A(N,2*N), B(2*N), C(2*N,2*N)
          REAL D(2*N,N), E(N), F(N), G(N)
!HPF$     TEMPLATE temp(N*N*4,N*N*4)
!HPF$     ALIGN A(i,j) WITH temp(4*i-3,4*j-3)
!HPF$     ALIGN B(i) WITH temp(*,4*i-3)
!HPF$     ALIGN C(i,j) WITH temp(4*j-3,i)
!HPF$     ALIGN D(i,j) WITH temp(4*j-3,4*i-3)
!HPF$     ALIGN G(i) WITH temp(4*i-3,*)
          D=C(:,1:4*N:4)
          E=SUM(A+TRANSPOSE(D),DIM=2)
          F=SUM(SPREAD(B,DIM=2,NCOPIES=N)+D,DIM=1)
          G=B(1:2*N:2)+E+F
```

The code example above is a revised version of the Fortran 90 program first used by Chatterjee et. al[15]. The current technique for compilers to optimize the above code is to first produce a temporary array for each intrinsic array operation and use a temporary array to pass the intermediate result to following array functions. It then tries to find an alignment and a distribution for each array to optimize the completion time. The related work in solving this problem by the above technique can be seen in [5, 9, 10, 13, 14, 15]. Following Chatterjee's scheme (known as one of the best schemes for this type of optimization), we get the optimized code below[15].

HPF Code Fragment 2

```
          REAL A(N,2*N), B(2*N) , C(2*N,2*N)
          REAL D(2*N,N),E(N),F(N),G(N)
          REAL T1(N,2*N),T2(N,2*N),T3(2*N,N),T4(2*N,N)
          REAL T5(N),T6(N)
!HPF$     TEMPLATE temp(N*N*4,N*N*4)
!HPF$     ALIGN A(i,j),T1(i,j),T2(i,j) WITH temp(4*i-3,4*j-3)
!HPF$     ALIGN B(i) WITH temp(*,4*i-3)
!HPF$     ALIGN C(i,j) WITH temp(4*j-3,i)
!HPF$     ALIGN D(i,j),T3(i,j),T4(i,j) WITH temp(4*j-3,4*i-3)
!HPF$     ALIGN E(i),F(i),T5(i),T6(i),G(i) WITH temp(4*i-3,*)
          D=C(:,1:4*N:4)
          T1=TRANSPOSE(D)
          T2=A+T1
          E=SUM(T2,DIM=2)
          T3=SPREAD(B,DIM=2,NCOPIES=N)
          T4=T3+D
          F=SUM(T4,DIM=1)
          T5=B(1:2*N:2)
          T6=T5+E
          G=T6+F
```

However, the scheme above has not considered the optimization opportunities for array operation synthesis. It has been shown in our previous work[2] that the synthesis of consecutive array operations or array expressions into a composite access function of the source arrays at compile time can improve program performance on flat shared memory parallel machines. Our scheme is based on the composition of access functions, which is analogous to a composition of mathematic functions. Our scheme can handle not only data movements of arrays with different numbers of dimensions and with multiple-clause array operations but also masked array expressions and multiple-source array operations. As a result, our scheme[2] can collectively synthesize Fortran 90 RESHAPE, EOSHIFT, MERGE, WHERE constructs, array sections and array reduction operations. Therefore, if we apply our synthesis scheme in [2] originally aimed at Fortran 90 programs directly with Code Fragment 1, we get an optimized code in the following pseudo code format.

HPF Code Fragment 3

```
               REAL A(N,2*N), B(2*N) , C(2*N,2*N)
               REAL D(2*N,N),E(N),F(N),G(N)
        !HPF$  TEMPLATE temp(N*N*4,N*N*4)
        !HPF$  ALIGN A(i,j) WITH temp(4*i-3,4*j-3)
        !HPF$  ALIGN B(i) WITH temp(*,4*i-3)
        !HPF$  ALIGN C(i,j) WITH temp(4*j-3,i)
        !HPF$  ALIGN D(i,j) WITH temp(4*j-3,4*i-3)
        !HPF$  ALIGN G(i) WITH temp(4*i-3,*)
               FORALL(I=1:N)
                    G(I)=B(2*I)+SUM_{J=1:2*N:1}(A(I,J)+C(J,4*I))+ SUM_{J=1:2*N:1}(B(J)+D(J,4*I))
               END FORALL
```

To compare the Code Fragments 3 with 2, we execute the codes by DEC HPF compiler on an 8-node DEC Farm with an FDDI network. We run the two code fragments 100 times. Also, the problem size, N, is set to be 128. The code optimized by our synthesis scheme is almost 5.5 times faster than the code generated by solely using automatic selections of alignments and distributions for arrays. We list the number of loads and stores, and the time for these two codes in the table below.

	number of loads	number of stores	execution time (sec)
Code fragment 2	$16N^2 + 5N$	$10N^2 + 5N$	6.110
Code fragment 3	$8N^2 + 4N$	N	1.103

From the above observation, one may be inclined to conclude that the earlier proposed array operation synthesis is sufficiently applicable to optimizing the array operations for distributed memory parallel machines without any modification. However, if we consider the Code Fragment 4, the communication cost may be increased due to the array operation synthesis.

HPF Code Fragment 4

```
        !HPF$ TEMPLATE TEMP(N,N)
              REAL A(N,N), B(N,N) , C(N,N)
        !HPF$ ALIGN A(i,j) with TEMP(i,j)
        !HPF$ ALIGN B(i,j) with TEMP(i,j)
        !HPF$ ALIGN C(i,j) with TEMP(i,j)
              C=TRANSPOSE( A+B , 1)
```

A compilation without array operation synthesis produces the following code:

HPF Code Fragment 5

```
        !HPF$ TEMPLATE TEMP(N,N)
              REAL A(N,N), B(N,N), C(N,N), T1(N,N)
        !HPF$ ALIGN A(i,j) with TEMP(i,j)
        !HPF$ ALIGN B(i,j) with TEMP(i,j)
        !HPF$ ALIGN C(i,j) with TEMP(i,j)
```

```
!HPF$ ALIGN T1(i,j) with A(i,j)
      T1=A+B
      C=TRANSPOSE(T1)
```

Using the array operation synthesis scheme, the resulted code is as follows:

HPF Code Fragment 6

```
!HPF$ TEMPLATE TEMP(N,N)
      REAL A(N,N), B(N,N), C(N,N)
!HPF$ ALIGN A(i,j) with TEMP(i,j)
!HPF$ ALIGN B(i,j) with TEMP(i,j)
!HPF$ ALIGN C(i,j) with TEMP(i,j)
      FORALL (i=1:N , j=1:N)
         C(i,j)=A(j,i)+B(j,i)
      END FORALL
```

In the Code Fragment 5, due to the fact that arrays A, B, and T1 are all aligned, the only communication is from T1 to C. In the Code Fragment 6, however, elements of arrays A and B will both be sent to the processors which own the corresponding target elements of array C according to the owner-computes rule. The amount of communication of the code after array operation synthesis is two times greater than the code without array operation synthesis. As a result, when we actually execute the programs with the HPF compiler on an 8 node DEC Farm. We run the two code fragments 100 times, and N is set to be 512. The execution time of Code Fragment 6 (with array synthesis) is 3.822 seconds, whereas Code Fragment 5 (without array synthesis) is 3.776 seconds. The synthesis optimization may actually increase the execution time. We call such a behavior, *synthesis anomaly*. In this paper, we will try to solve this problem by deriving a cost model and using the cost model to decide if a temporary array in the target array expression should be eliminated or not.

3 Array Operation Synthesis

Our previously proposed array operations optimization is to synthesize consecutive array operations or array expression into a single composite mathematical function[2, 3]. For the sake of completeness, we present an overview of the array operation synthesis scheme in this section.

3.1 Data Access Functions of Array Operations

We use a *data access function* to specify the mapping between the elements of the source and target arrays of an array operation. For certain array operations, the data accesses can not be represented in a single continuous data access pattern, but instead they can be represented in multiple data access patterns, each with its own array index range. It is a piecewise-defined function. We use a segmentation descriptor as formally defined below to represent an array index range of a data access pattern.

Definition 1 **A segmentation descriptor** is a set. It is recursively defined as follows:

$$
\begin{aligned}
&(1)\phi(\ /f_1(i_1,i_2,\cdots,i_n),\cdots,\ f_m(i_1,i_2,\cdots,i_n)/\ ,\ /l_1\!:\!u_1\!:\!s_1, l_2\!:\!u_2\!:\!s_2,\cdots, l_m\!:\!u_m\!:\!s_m/) = \\
&\quad \{\ (i_1,i_2,\cdots,i_n)\ \mid\ \forall\ k,\ 1\le k\le n,\ l_k\le f_k(i_1,i_2,\cdots,i_n)\le u_k \texttt{ and } \\
&\qquad\qquad f_k(i_1,i_2,\cdots,i_n) = l_k + s_k * I,\ \texttt{where } I \texttt{ is a non-negative integer.}\ \}
\end{aligned}
$$

A regular section specifier has the form, $l_k\!:\!u_k\!:\!s_k$, where the l_k, u_k and s_k indicate the lower bound, upper bound and stride of $f_k(i_1,i_2,\cdots,i_n)$ respectively. We may omit the stride if it is one. We say a segmentation descriptor is **simple** if it is of the above form.

(2)If α_1 and α_2 are two segmentation descriptors, then $\alpha_1 \wedge \alpha_2$ is also a segmentation descriptor but is not simple. $\alpha_1 \wedge \alpha_2 = \{(i_1,i_2, \cdots, i_n) \mid (i_1,i_2, \cdots, i_n) \in \alpha_1$ and $(i_1,i_2, \cdots, i_n) \in \alpha_2 \}$.

A data access pattern illustrates a mapping between the source arrays and the target array within an index range specified by a segmentation descriptor. We give the definition as follows:

Definition 2 A **data access pattern** has the following form.

$$T[i_1, i_2, \cdots, i_n] = \mathcal{F}[S_1(f_1^1(i_1, i_2, \cdots, i_n), \cdots, f_1^{d1}(i_1, i_2, \cdots, i_n)], \cdots S_k[f_k^1(i_1, i_2, \cdots, i_n), \cdots, f_k^{dk}(i_1, i_2, \cdots, i_n)]) \mid \gamma$$

T is the target array which is the destination of the data movement. S_1, $\cdots$, S_k are the source arrays which are the sources of the data movement. f_i^j is an index function. The index functions specify the array subscripts of source arrays at the right-hand side (RHS). $\mathcal{F}$ is a k-ary function called **calculation function**. The data are calculated and then moved to the target array according the calculation function. γ is a segmentation descriptor.

Example of a data access pattern:
B[i,j] = $\mathcal{F}$ (A[j,i+1],T3[i+1,j]) | ϕ(/i,j/,/1:3,1:4/) $\wedge$ ϕ(/j,i+1/,/1:3,1:3/)
$\mathcal{F}$(x,y)=x+y.

Definition 3 Consider an n-dimensional target array T and k source arrays, $S_1, \cdots, S_k$. Let d_i be the number of dimensions of array S_i. A **data access function** has the following form:

$$T[i_1, i_2, \cdots, i_n] = \begin{cases} \mathcal{F}1[S_1(f_1^1(i_1, i_2, \cdots, i_n), \cdots, f_1^{d1}(i_1, i_2, \cdots, i_n)], \cdots S_k[f_k^1(i_1, i_2, \cdots, i_n), \cdots, f_k^{dk}(i_1, i_2, \cdots, i_n)]) \mid \gamma_1 \\ \mathcal{F}2[S_1(h_1^1(i_1, i_2, \cdots, i_n), \cdots, h_1^{d1}(i_1, i_2, \cdots, i_n)], \cdots S_k[h_k^1(i_1, i_2, \cdots, i_n), \cdots, h_k^{dk}(i_1, i_2, \cdots, i_n)]) \mid \gamma_2 \\ \cdots \\ \cdots \end{cases}$$

$\mathcal{F}1$ and $\mathcal{F}2$ are calculation functions.

Assume $\forall\, i$, $1 \le i \le n$, r_i is the size of the i-th dimension of T. In general, a data access function can be translated into the following pseudo code.

```
FORALL i1=1 to r1
 FORALL i2=1 to r2
  .  .  .
  FORALL in=1 to rn
   IF (i1,i2,···,in) ∈ γ1 THEN
      T[i1,i2, ··· ,in]= F1[S1(f1^1(i1,i2,···,in), ···, f1^d1(i1,i2,···,in)], ··· Sk[fk^1(i1,i2,···,in), ···, fk^dk(i1,i2,···,in)] )
   IF (i1,i2,···,in) ∈ γ2 THEN
      T[i1,i2, ··· ,in]= F2[S1(h1^1(i1,i2,···,in), ···, h1^d1(i1,i2,···,in)], ··· Sk[hk^1(i1,i2,···,in), ···, hk^dk(i1,i2,···,in)] )
                      ...
                      ...
```

If a data access function contains only one data access pattern and its corresponding index range spans the whole dimensional space of the target array in the LHS, the segmentation descriptor degenerates to a single true value and can be ignored. If there is only one source array in a data access function, we call this data access function *single-source data access function* and also its corresponding array operation is called a *single-source array operation*. In the case that there are multiple source arrays within a data access function, we call this data access function a *multiple-source data access function* and its corresponding array operation a *multiple-source array operation*, respectively. We also classify the data access functions according to their numbers of data access patterns. If a data access function contains only one data access pattern, we call it a *single-clause data access function*. Similarly, a data access function with multiple data access patterns is called a *multiple-clause data access function*.

The classification is listed in the following table:

	single-clause	multiple-clause
single-source	type 1	type 3
multiple-source	type 2	type 4

Note that the four types of data access are not disjoint sets. The type 1, 2, and 3 data access functions are subsets of type 4 data access function. Following are several examples of data access functions:

Example 1 Assuming that A and B are 4 by 4 matrices, the data access function of B=CSHIFT(A,1,1) is

$$B[i,j] = \begin{cases} A[i-3,j] \mid \phi(\text{/i,j/,/4:4,1:4/}) \\ A[i+1,j] \mid \phi(\text{/i,j/,/1:3,1:4/}) \end{cases}$$

It is a type 3 data access function. That is, CSHIFT is a type 3 array operation.

Example 2 Assume A and B are two-dimensional arrays. The data access function of B=TRANSPOSE(A) is B[i,j]=A[j,i] where f_1(i,j)=j and f_2(i,j)=i. We omit the calculation function because it is an identity function. This is a type 1 data access function. The TRANSPOSE example is a data movement between arrays of equal dimensional space.

Example 3 Function RESHAPE is an example of a data access function between arrays of different dimensional space. For example, the data access function for E=RESHAPE(D,(/2,3/)) is E[i,j]=D[(j-1)*2+i] where f_1(i,j)=(j-1)*2+i. This is a type 1 data access function.

Example 4 The data access function for C(:,:)=A(:,:)+B(:,:) is C[i,j]=$\mathcal{F}$(A[i,j],B[i,j]). This is a type 2 data access function.

Example 5 A more complicated example is the MERGE function in Fortran 90. The data access function for R = MERGE(T, F, M) is R[i,j] = $\mathcal{F}$(T[i,j], F[i,j], M[i,j]) where

$$\mathcal{F}(x,y,z) = \begin{cases} x & \text{if } z = true \\ y & \text{if } z = false \end{cases}$$

This is a type 2 data access function.

Although no array function in Fortran 90 belongs to type 4, the synthesis of multiple array operations may result in a type 4 data access function.

3.2 Synthesis of Array Operations

The synthesis of array operations starts from building the parse tree of the targeted array expression. Figure 1 shows an array expression and its parse tree. The source arrays are in the leaf nodes. The data access function is then derived for each internal node. We use the example in Figure 1 as the running example in this subsection.

For the EOSHIFT function, the data access function is:

$$T1[i,j] = \begin{cases} A[i+1,j] & \mid & \phi(\text{/i,j/,/1:3,1:4/}) \\ 0 & \mid & \phi(\text{/i,j/,/4:4,1:4/}). \end{cases}$$

The access function for the TRANSPOSE function is T2[i,j] = T1[j,i], for the RESHAPE function is T3[i, j] = C[i + j*4 -4], for the "+" operation is T4[i,j]=$\mathcal{F}$1(T2[i,j],T3[i,j]), where $\mathcal{F}$1(x,y)=x+y, and for "*" operation is T5[i,j]=$\mathcal{F}$2(T4[i,j], D[i,j]), where $\mathcal{F}$2(x,y)=x*y. The access function for the CSHIFT function is

$$B[i,j] = \begin{cases} T5[i+1,j] & \mid & \phi(\text{/i,j/,/1:3,1:4/}) \\ T5[i-3,j] & \mid & \phi(\text{/i,j/,/4:4,1:4/}). \end{cases}$$

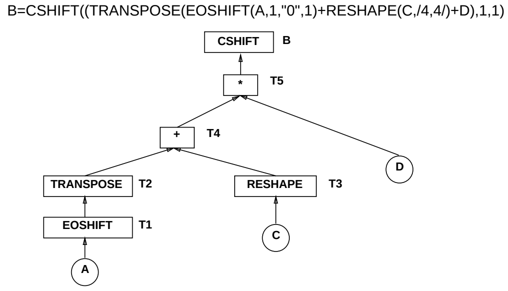

Figure 1: Parse tree of an array expression.

The next step is to synthesize these data access functions. A data access function may contain multiple data access patterns delimited by segmentation descriptors. We first discuss how to synthesize two data access patterns. Then, the synthesis of two data access functions can be divided to a series of syntheses of two data access patterns.

In the case that a data access pattern has no segmentation descriptor, we treat the outcome of its segmentation descriptor as always true. Suppose that we get the following two multiple-source data access patterns:

$$\begin{aligned} \mathtt{S}[i_1,i_2,\cdots,i_m] = \mathcal{F}2(& Q_1[h_{1,1}(i_1,i_2,\cdots,i_m),\cdots,h_{1,x_1}(i_1,i_2,\cdots,i_m)], \\ & Q_2[h_{2,1}(i_1,i_2,\cdots,i_y),\cdots,h_{2,x_2}(i_1,i_2,\cdots,i_m)], \\ & \cdots, \\ & Q_k[h_{k,1}(i_1,i_2,\cdots,i_y),\cdots,h_{k,x_k}(i_1,i_2,\cdots,i_m)]\ ) \mid \beta \end{aligned}$$

$$\mathtt{T}[i_1,i_2,\cdots,i_n] = \mathcal{F}1(\cdots,\ \mathtt{S}[\ f_1(i_1,i_2,\cdots,i_n),f_2(i_1,i_2,\cdots,i_n),\cdots,f_m(i_1,i_2,\cdots,i_n)],\cdots\ ) \mid \gamma$$

where γ and β are segmentation descriptors, and $\beta = \phi(/i_1,i_2,\cdots,i_m/\ ,\ /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$. The S in the RHS of the second equation can be replaced with Q_p (p $= 1 \cdots$ k) with respect to the first equation. It yields the following data access pattern:

$$\begin{aligned} \mathtt{T}[i_1,i_2,\cdots,i_n] = \mathcal{F}1(\cdots,\ \mathcal{F}2(& Q_1[g_{1,1}(i_1,i_2,\cdots,i_n),\cdots,g_{1,x_1}(i_1,i_2,\cdots,i_n)] \\ & ,\cdots, \\ & Q_2[g_{2,1}(i_1,i_2,\cdots,i_n),\cdots,g_{2,x_2}(i_1,i_2,\cdots,i_n)], \\ & \cdots, \\ & Q_k[g_{k,1}(i_1,i_2,\cdots,i_n),\cdots,g_{k,x_k}(i_1,i_2,\cdots,i_n)]),\cdots) \mid \gamma \wedge \beta' \end{aligned}$$

where $g_{a,b}(i_1,i_2,\cdots,i_n) = h_{a,b}(f_1(i_1,i_2,\cdots,i_n),\cdots,f_m(i_1,i_2,\cdots,i_n))$, and
$\beta' = \phi(\ /f_1(i_1,i_2,\cdots,i_n),\cdots,\ f_m(i_1,i_2,\cdots,i_n)\ /,\ /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

For example, if we have the following two data access patterns: T1[i,j] = A[i,j] | ϕ(/i,j/,/1:3,1:3/) and B[i,j] = $\mathcal{F}$(T1[j,i+1], T3[i+1,j]) | ϕ(/i,j/,/1:3,1:4/). Based on the synthesis rule of two data access patterns described above, we can derive the synthesized data access pattern:

B[i,j] = $\mathcal{F}$ (A[j,i+1],T3[i+1,j]) | ϕ(/i,j/,/1:3,1:4/) $\wedge$ ϕ(/j,i+1/,/1:3,1:3/)

We now present how to synthesize two data access functions. For the following two simplified data access functions, (1) is the data access function of array S, and (2) is the data access function of array T.

$$\mathrm{S}[\mathrm{i}_1,\mathrm{i}_2,\cdots,\mathrm{i}_\mathrm{n}] = \begin{cases} \mathcal{G}_1(\cdots,T[\cdots],\cdots) & \mid \gamma_1 \\ \cdots & \\ \mathcal{G}_{k-1}(\cdots,T[\cdots],\cdots) & \mid \gamma_{k-1} \\ \mathcal{G}_k(\cdots,T[\cdots],\cdots) & \mid \gamma_k \\ \mathcal{G}_{k+1}(\cdots,T[\cdots],\cdots) & \mid \gamma_{k+1} \\ \cdots & \\ \mathcal{G}_x(\cdots,T[\cdots],\cdots) & \mid \gamma_x \end{cases} \tag{1}$$

$$\mathrm{T}[\mathrm{i}_1, \mathrm{i}_2, \cdots, \mathrm{i}_\mathrm{p}] = \begin{cases} \mathcal{F}_1(\cdots) & \mid \beta_1 \\ \cdots & \\ \mathcal{F}_q(\cdots) & \mid \beta_q \\ \cdots & \\ \mathcal{F}_y(\cdots) & \mid \beta_y \end{cases} \tag{2}$$

Without loss of generality, we do not specify the index functions. The first and the second data access functions include x and y data access patterns, respectively. Assume that T is a temporary array. T appears in the RHS of the data access patterns of (1). Also, T appears in the LHS of (2). For generality, we assume array T appears in all of the data access patterns of (1). In the synthesis of the two data access functions, we have to perform the synthesis of data access patterns $x * y$ times. Let DAP(p,q) is the data access pattern of combining the p-th and the q-th data access patterns of (1) and (2), respectively. The synthesized data access function should contain all of the DAP(u,v), where 1≤u≤x and 1≤v≤y. This will produce a data access function with $x * y$ data access patterns. The final data access function is (3):

$$\mathrm{S}[\mathrm{i}_1, \mathrm{i}_2, \cdots, \mathrm{i}_\mathrm{n}] = \begin{cases} \mathcal{G}_1(\cdots, \mathcal{F}_1(\cdots), \cdots) & \mid \gamma_1 \wedge \beta'_1 \\ \mathcal{G}_1(\cdots, \mathcal{F}_2(\cdots), \cdots) & \mid \gamma_1 \wedge \beta'_2 \\ \cdots & \\ \mathcal{G}_1(\cdots, \mathcal{F}_y(\cdots), \cdots) & \mid \gamma_1 \wedge \beta'_y \\ \cdots & \\ \mathcal{G}_k(\cdots, \mathcal{F}_1(\cdots), \cdots) & \mid \gamma_k \wedge \beta'_1 \\ \mathcal{G}_k(\cdots, \mathcal{F}_2(\cdots), \cdots) & \mid \gamma_k \wedge \beta'_2 \\ \cdots & \\ \mathcal{G}_k(\cdots, \mathcal{F}_y(\cdots), \cdots) & \mid \gamma_k \wedge \beta'_y \\ \cdots & \\ \mathcal{G}_x(\cdots, \mathcal{F}_1(\cdots), \cdots) & \mid \gamma_x \wedge \beta'_1 \\ \mathcal{G}_x(\cdots, \mathcal{F}_2(\cdots), \cdots) & \mid \gamma_x \wedge \beta'_2 \\ \cdots & \\ \mathcal{G}_x(\cdots, \mathcal{F}_y(\cdots), \cdots) & \mid \gamma_x \wedge \beta'_y \end{cases} \tag{3}$$

The above form describes the process of synthesizing two data access functions. To synthesize an array expression or consecutive array operations, we start with the data access function in the root of the parse tree. For each temporary array at the RHS of the data access function, we substitute it with the data access function of that temporary array. The substitution may produce other temporary arrays in the RHS. This substitution processes is repeated until all the temporary arrays in the RHS are substituted by the source arrays. This substitution process will terminate because there is no cycle in a parse tree.

We use the example in Figure 1 to show how these data access functions are synthesized. We start with the data access function in the root (CSHIFT) of the parse tree:

$$B[i,j] = \begin{cases} T5[i+1,j] & \mid & \phi(\text{/i,j/,/1:3,1:4/}) \\ T5[i-3,j] & \mid & \phi(\text{/i,j/,/4:4,1:4/}) \end{cases}$$

Following is a series of syntheses of two data access functions:

(1) To substitute T5: using T5[i,j]=$\mathcal{F}$2(T4[i,j],D[i,j])

$$\Longrightarrow B[i,j] = \begin{cases} \mathcal{F}2(T4[i+1,j], D[i+1,j]) & \mid & \phi(\text{/i,j/,/1:3,1:4/}) \\ \mathcal{F}2(T4[i-3,j], D[i+1,j]) & \mid & \phi(\text{/i,j/,/4:4,1:4/}) \end{cases}$$

(2) To substitute T4: using T4[i,j]=$\mathcal{F}$1(T2[i,j],T3[i,j])

$$\Longrightarrow B[i,j] = \begin{cases} \mathcal{F}2(\mathcal{F}1(T2[i+1,j], T3[i+1,j]), D[i+1,j]) & \mid & \phi(\text{/i,j/,/1:3,1:4/}) \\ \mathcal{F}2(\mathcal{F}1(T2[i-3,j], T3[i-3,j]), D[i+1,j]) & \mid & \phi(\text{/i,j/,/4:4,1:4/}) \end{cases}$$

(3) To substitute T2: using T2[i,j]=T1[j,i]

$$\Longrightarrow B[i,j] = \begin{cases} \mathcal{F}2(\mathcal{F}1(T1[j,i+1], T3[i+1,j]), D[i+1,j]) & \mid & \phi(\text{/i,j/,/1:3,1:4/}) \\ \mathcal{F}2(\mathcal{F}1(T1[j,i-3], T3[i-3,j]), D[i+1,j]) & \mid & \phi(\text{/i,j/,/4:4,1:4/}) \end{cases}$$

$\mathcal{F}$(x,y)=x+y

(4) To substitute T1: using

$$T1[i,j] = \begin{cases} A[i+1,j] & | \quad \phi(/\text{i,j}/,/1{:}3,1{:}4/) \\ 0 & | \quad \phi(/\text{i,j}/,/4{:}4,1{:}4/). \end{cases}$$

$$\Longrightarrow B[i,j] = \begin{cases} \mathcal{F}2(\mathcal{F}1(A[j+1,i+1],T3[i+1,j]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/1{:}3,1{:}4/) \wedge \phi(/\text{j,i+1}/,/1{:}3,1{:}4/) \\ \mathcal{F}2(\mathcal{F}1(0,T3[i+1,j]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/1{:}3,1{:}4/) \wedge \phi(/\text{j,i+1}/,/4{:}4,1{:}4/) \\ \mathcal{F}2(\mathcal{F}1(A[j+1,i-3],T3[i-3,j]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/4{:}4,1{:}4/) \wedge \phi(/\text{j,i-3}/,/1{:}3,1{:}4/) \\ \mathcal{F}2(\mathcal{F}1(0,T3[i-3,j]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/4{:}4,1{:}4/) \wedge \phi(/\text{j,i-3}/,/4{:}4,1{:}4/) \end{cases}$$

$$\mathcal{F}(\text{x,y})=\text{x+y}$$

(5) To substitute T3: using T3[i,j]=C[i+4*j-4] $\Longrightarrow$

$$\text{B[i,j]} = \begin{cases} \mathcal{F}2(\mathcal{F}1(A[j+1,i+1],C[i+4*j-3]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/1{:}3,1{:}4/) \wedge \phi(/\text{j,i+1}/,/1{:}3,1{:}4/) \\ \mathcal{F}2(\mathcal{F}1(0,C[i+4*j-3]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/1{:}3,1{:}4/) \wedge \phi(/\text{j,i+1}/,/4{:}4,1{:}4/) \\ \mathcal{F}2(\mathcal{F}1(A[j+1,i-3],C[i+4*j-7]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/4{:}4,1{:}4/) \wedge \phi(/\text{j,i-3}/,/1{:}3,1{:}4/) \\ \mathcal{F}2(\mathcal{F}1(0,C[i+4*j-3]),D[i+1,j]) & | \quad \phi(/\text{i,j}/,/4{:}4,1{:}4/) \wedge \phi(/\text{j,i-3}/,/4{:}4,1{:}4/) \end{cases} \tag{4}$$

After having derived the sole synthesized data access function (4), we can use it to generate a nested loop.

For the running example, we use the obtained synthesized data access function to generate the pseudo code below in a rather straightforward manner.

```
FORALL i=1 to 4, j=1 to 4
     if (i,j)∈(φ(/i,j/,/1:3,1:4/) ∧ φ(/j,i+1/,/1:3,1:4/)) THEN B[i,j]=(A[j+1,i+1]+C[i+4*j-3])*D[i+1,j]
     if (i,j)∈(φ(/i,j/,/1:3,1:4/) ∧ φ(/j,i+1/,/4:4,1:4/)) THEN B[i,j]=C[i+4*j-3]*D[i+1,j]
     if (i,j)∈(φ(/i,j/,/4:4,1:4/) ∧ φ(/j,i-3/,/1:3,1:4/)) THEN B[i,j]=(A[j+1,i-3]+C[i+4*j-7])*D[i+1,j]
     if (i,j)∈(φ(/i,j/,/4:4,1:4/) ∧ φ(/j,i-3/,/4:4,1:4/)) THEN B[i,j]=C[i+4*j-7]*D[i+1,j]
```

We have also developed a systematic method to combine predicates of testing index ranges and loop bounds. The optimized code is as follows:

```
FORALL i=1 to 3, j=1 to 3
     B[i,j]=(A[j+1,i+1]+C[i+4*j-3])*D[i+1,j]
FORALL i=1 to 3, j=4 to 4
     B[i,j]=C[i+4*j-3]*D[i+1,j]
FORALL i=4 to 4, j=1 to 3
     B[i,j]=(A[j+1,i-3]+C[i+4*j-7])*D[i+1,j]
FORALL i=4 to 4, j=4 to 4
     B[i,j]=C[i+4*j-7]*D[i+1,j]
```

Because of the space limitation, we only present the essential portion of our work in array operation synthesis. The details of our work can be found in [2, 3].

4 Synthesis of HPF Programs on Distributed Memory Machines

The synthesis scheme shown in the previous section for Fortran 90 programs can be applied directly to synthesize array operations in HPF programs, such as RESHAPE, EOSHIFT, MERGE, WHERE constructs, array sections and array reduction operations. However, as mentioned in Section 2, a naive application of the synthesis scheme to HPF programs may actually increase the execution time of HPF programs due to increased remote memory accesses. In the following discussions, we will first introduce a cost model, and based on this model, we will develop an optimal algorithm to guide the synthesis process to avoid performance anomaly. Note that our work here will focus on the synthesis anomaly due to the execution on distributed memory machines. Previously, in the sequential programs or flat shared memory parallel programs, the synthesis may increase the execution time of programs due to the replication of computation while intermediate arrays are eliminated. This occurs when the array considered to be eliminated is replicated by a **one-to-many array operation**, such as spread or when eliminating temporaries which will be used multiple times. The prevention strategies for the problem of replication is documented in [2]. Throughout this paper, we will assume the replication problem has been taken care of, and will focus only on the issues due to the remote communication costs. In addition, we assume the HPF semantics that all data placement of named arrays are specified by users. The data placement includes alignments and distributions of arrays. We do not consider the case that alignments and distributions of the source and target arrays are not specified, though in such cases, we can always first perform Chatterjee's algorithm[15] before doing synthesis to get approximate alignments and distributions of the source and target arrays.

4.1 Communication Cost Model for Alignment Analysis

The data movement from the source arrays to the target array may cause remote communication across processors. However, if the source arrays and the target array are aligned with respect to their array operation, then there will be no remote communication across processors no matter what kind of distributions the source and the target arrays may have[1]. In the case that the target array and the source arrays are not aligned, the remote communication is always necessary according to the owner-computing rule. The amount of the remote communication depends on the alignment and distribution of the source arrays and the target array. An often-used communication cost model for alignment analysis is to model the communication cost to be the amount of the data moved if the target array and the source array are misaligned. In [6], it defines the abstract machine to be of the same shape as the spatial index domain of the program, that is, each processor owns a element of arrays. Chatterjee et al. modelled the communication cost to be the amount of the misaligned data multiplied by a distance function [15]. Our cost model is also developed based on modelling the communication cost to be the amount of the misaligned data multiplied by a distance function. The cost model is defined below.

Definition 4 Suppose we want to move array $B[\, g_1(i_1, \cdots, i_k), g_2(i_1, \cdots, i_k), \cdots, g_m(i_1, \cdots, i_k)\,]$ to $A[\, f_1(i_1, \cdots, i_k), f_2(i_1, \cdots, i_k), \cdots, f_n(i_1, \cdots, i_k)\,]$ within the index domain defined by the segmentation descriptor $\gamma = \phi(/i_1, \cdots, i_k/, /l_1 : u_1 : s_1, \cdots, l_k : u_k : s_k\ /)$. That is, we want to execute the following section movement.

$$\forall (i_1, \cdots, i_k) \in \gamma,\ A[f_1(i_1, \cdots, i_k), , \cdots, f_n(i_1, \cdots, i_k)] = B[g_1(i_1, \cdots, i_k), , \cdots, g_m(i_1, \cdots, i_k)].$$

We define the communication cost of the above section movement as

$$\sum_{(i_1, \cdots, i_k) \in \gamma} \mathcal{D}(A[f_1(i_1, \cdots, i_k), , \cdots, f_n(i_1, \cdots, i_k)], B[g_1(i_1, \cdots, i_k), , \cdots, g_m(i_1, \cdots, i_k),])$$

where $\mathcal{D}$ is called *distance function.*

The distance function corresponds to the cost per unit data of moving from the source array to the target array. To more precisely estimate the communication cost of data movement between arrays, one should consider both the alignment and the distribution of the source and target arrays to decide the distance function $\mathcal{D}$. For simplicity, we define the distance function in the Definition 5 below. Although the model simplifies real systems, our framework to be presented in Section 4.2, 4.3, and 5 are completely extensible with a more accurate definition of the distance function *calD*.

Definition 5 We define

$\mathcal{D}(A[f_1(i_1, \cdots, i_k), \cdots, f_n(i_1, \cdots, i_k)]\,, B[g_1(i_1, \cdots, i_k), \cdots, g_m(i_1, \cdots, i_k)])$=1 if $A[f_1(i_1, \cdots, i_k), \cdots, f_n(i_1, \cdots, i_k)]$ and $B[g_1(i_1, \cdots, i_k), \cdots, g_m(i_1, \cdots, i_k)]$ are not aligned,

$\mathcal{D}(A[f_1(i_1,\cdots,i_k),\cdots,f_n(i_1,\cdots,i_k)]\,,B[g_1(i_1,\cdots,i_k),\cdots,g_m(i_1,\cdots,i_k)])$=0 if $A[f_1(i_1,\cdots,i_k),\cdots,f_n(i_1,\cdots,i_k)]$ and $B[g_1(i_1,\cdots,i_k),\cdots,g_m(i_1,\cdots,i_k)]$ are aligned.

We present an example here to demonstrate how the cost model can be used. Let us consider again Code Fragment 4, 5, and 6 in Section 2. Be reminded that code fragment 5 is the compiled version without array synthesis. The data access function of T1=A+B in statement 7 is T1[i,j]=A[i,j]+B[i,j]. Since arrays A, B, and T1 are all aligned, there are 20000 real numbers local memory loads without any remote memory references. In statement 8, the data access function is C[i,j]=T1[j,i], but arrays C and T1 are not aligned. The communication cost is $\sum_{(i,j)\in\phi(/i,j/,/1:100,1:100/)}\mathcal{D}(C[i,j],T1[j,i]) = 10000*1 = 10000$. Hence, the execution of the entire array expression has a communication cost of 10000. In contrast to the above result, consider the cost in the synthesized version in the code fragment 6. Using the array operation synthesis scheme, the resulted data access function of C=TRANSPOSE(A+B, 1) is C[i,j]=A[j,i]+B[j,i]. Since C[i,j] is not aligned with A[j,i] and B[j,i], the communication cost for executing the entire array expression is $\sum_{(i,j)\in\phi(/i,j/,/1:100,1:100/)}\mathcal{D}(C[i,j],A[j,i]) + \sum_{(i,j)\in\phi(/i,j/,/1:100,1:100/)}\mathcal{D}(C[i,j],B[j,i]) = 20000$. In the above example, we find that array operation synthesis reduces the amount of memory loads from 30000 to 20000. However, it increases the communication cost from 10000 to 20000 under our cost function.

4.2 Optimal Solution for Executing Array Expression on Distributed Memory Machines

To obtain the optimal solution for executing array expressions under the owner-computes rule model, one may have to synthesize only part of the array operations in the target array expressions. Since each synthesis of two array operations will eliminate one temporary array, the problem turns out to be the decision problem to decide if a temporary array in the target array expression should be eliminated. Furthermore, we must find the alignment and distribution for those temporary arrays which are not eliminated. Assume that a straightforward compilation produces N temporary arrays. The possible cases which we have to consider are 2^N, as we will decide whether each temporary array should be eliminated by the synthesis process or not. The following definition describes the minimum cost in one of the 2^N cases.

Definition 6 The cost function $\mathcal{C}$(U,AE) denoting the minimum cost of executing the parse tree under the encoding set U is defined as follows:

(1)AE is an array expression. The array expression parse tree constructed from AE is $\mathcal{P}$ which consists of N temporary arrays, T_1, T_2, $\cdots$, T_N.

(2) U $\in 2^{\{T_1,T_2,\cdots,T_N\}}$. U is a set which encodes the existence of the temporary arrays from T_1 to T_N. If $T_i \notin$ U, then the temporary array T_i will be eliminated by the array operation synthesis. Otherwise, T_i will be kept.

(3) We use U and $\mathcal{P}$ to construct another parse tree $\mathcal{P}'$. $\mathcal{P}'$ is obtained by using array operation synthesis scheme to eliminate the temporary arrays which are not in U.

(4) The $\mathcal{C}$(U,AE) is defined as the minimum cost of executing $\mathcal{P}'$ among those possible alignments of the kept temporary arrays.

We use U and $\mathcal{P}$ to construct another parse tree $\mathcal{P}'$. For example, in Figure 2, AE is F = CSHIFT(MERGE((A+B) * C, D, E), 1,1), and the parse tree in the left-hand-side of the figure is $\mathcal{P}$. In order to compute the C($\{T_2,T_3\}$,AE), we have to construct $\mathcal{P}'$ according to $\{T_2,T_3\}$. Because only T_1 is not in $\{T_2,T_3\}$, the temporary

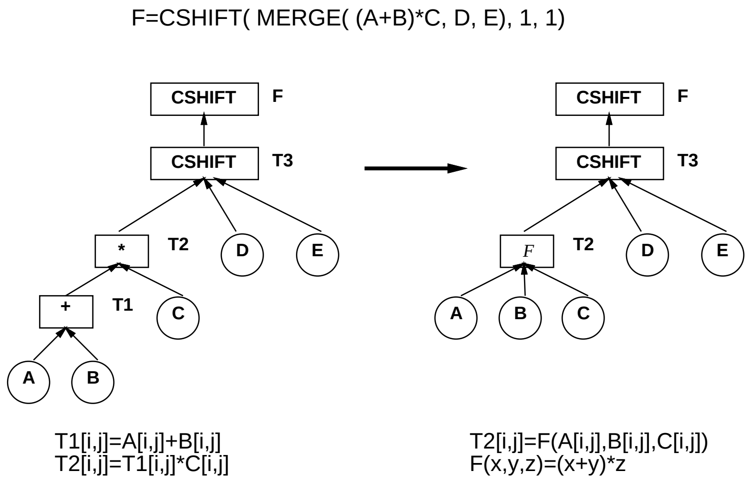

Figure 2: Parse tree for C($\{T_2, T_3\}$,AE)

array T1 is eliminated by the array operation synthesis and the other temporary arrays are kept. By eliminating T1 in the synthesis scheme, we get the parse tree $\mathcal{P}'$ in the right-hand-side of Figure 2.

Note that the cost of executing $\mathcal{P}'$ depends on the alignment and distribution of the temporary arrays left in $\mathcal{P}'$. Each different kind of alignment and distribution of temporary arrays will result in different running time in executing $\mathcal{P}'$. Within all the possible cases for aligning the temporary arrays in $\mathcal{P}'$, the $\mathcal{C}$(U,AE) is defined as the minimum cost of executing $\mathcal{P}'$ among those possible alignments of the kept temporary arrays. Then, the minimum cost for evaluating the target array expression AE can be derived as follows:

$$\Omega = \mathbf{MIN}\{\mathcal{C}(U, AE) \mid U \in 2^{\{T_1, T_2, \cdots, T_N\}}\}$$

Theorem 1 *The complexity of calculating Ω is NP-hard.*

Proof: See Appendix A.

The detail of calculating Ω is presented in Section 4.3.

4.3 Dynamic Programming Algorithm for Optimal Solution

In this section, we devise an algorithm by employing dynamic programming techniques to find the optimal solution. Assume $\mathcal{P}$ is the parse tree of the target array expression AE, $\mathcal{P}$ is with N temporary arrays $T_1, T_2, \cdots, T_N$, and RT is the array corresponding to the root node of $\mathcal{P}$. Let $\mathcal{A}_i$ be the set of possible alignments of temporary arrays $T_i, 1 \le i \le N$.

We first present the recursive algorithm which calculates the minimum communication cost of evaluating a subtree which is rooted at T_s ($T_s \in \{T_1, T_2, \cdots, T_N, RT\}$), with the alignment of T_s selected to be $\tilde{a}$. It is denoted as Min_Comm_Cost(T_s,$\tilde{a}$).

Algorithm 1 : To calculate Min_Comm_Cost(T_s,$\tilde{a}$)

Input:

(1) $T_s \in \{T_1, T_2, \cdots, T_N, RT\}$ is the root node of the subtree which we want to compute its communication cost.

(2) $\tilde{a} \in \mathcal{A}_s$ is a possible alignment of T_s.

Output: Min_Comm_Cost(T_s,$\tilde{a}$)
Begin_of_Algorithm

Derive S=$\{T_i \mid T_i$ is the descendant of $T_s\}$.

Derive P=$2^S - \{V \mid V \in 2^S,\ and\ \exists\ T_a \in V\ and\ T_b \in V,\ where\ T_a\ is\ a\ ancestor\ of\ T_b\}$.

π=∞.

FOR $\Lambda \in$ P DO

- Assume Λ=$\{T_{l_1}, T_{l_2}, \cdots, T_{l_p}\}$.
- Prepare all the data access functions for all the temporary arrays rooted by T_s. We then use T_s as the target source array to perform array operation synthesis. However, any temporary array in Λ is not eliminated by the synthesis process. Let $\mathcal{R}$ be the synthesized result.
- Let $\mathcal{C}_\mathcal{D}(\mathrm{R}, \tilde{a}, a_{l_1}, a_{l_2}, \cdots, a_{l_p})$ be a function which represents the communication cost of evaluating data access function $\mathcal{R}$ based on the owner-computes rule, where the alignment of the T_s is $\tilde{a}$, the alignment of T_{l_j} is a_{l_j}, $a_{l_j} \in \mathcal{A}_{l_j}$, $\mathcal{A}_{l_j}$ is the alignment set of T_{l_j}, $T_{l_j} \in \Lambda$.
- **Temp**= $MIN_{a_{l_1} \in \mathcal{A}_{l_1}, a_{l_2} \in \mathcal{A}_{l_2}, \cdots, a_{l_p} \in \mathcal{A}_{l_p}} \{\mathcal{C}_\mathcal{D}(\mathcal{R}, \tilde{a}, a_{l_1}, a_{l_2}, \cdots, a_{l_p}) + \sum_{i=1}^{p} Min\_Comm\_Cost(T_{l_i}, a_{l_i})\}$
- π=MIN(**Temp**,π).

ENDDO

RETURN π

End_of_Algorithm

The set S in the algorithm above consists of all the temporary arrays of which the predecessor is T_s. The Set P is used to enumerate all the possible combinations of the temporary arrays controlled by T_s. Assume $V \in 2^S$, and T_a is a parent node or grand parent node of T_b. Then if T_a is in V, we no longer need T_b to be in V, as the algorithm will be calculated recursively. We use the following example to illustrate the idea of the algorithm.

Example 6 We use the array expression in Figure 1 of Section 3.2 as a running example to illustrate this algorithm. $\mathcal{P}$ is the parse tree in Figure 1. For computing Min_Comm_Cost(T5,$\tilde{a}$),

(1) S={T1, T2, T3, T4}.

(2) P={ $\emptyset$, $\{T_1\}$, $\{T_2\}$, $\{T_3\}$, $\{T_4\}$, $\{T_1,T_3\}$,$\{T_2,T_3\}$}.

(4) Assume we have Λ={T2}. Based on the running example, $\mathcal{R}$ is obtained by using T5 as the target source array and leaving the data access function related to T2 without synthesis as follows:

$$\mathcal{R}\ :\ T5[i,j] = \mathcal{F}2(\mathcal{F}1(\mathbf{T2}[i,j], C[i+4*j-4]), D[i,j])$$

Then, **Temp** = $MIN_{a' \in \mathcal{A}} \{\mathcal{C}_\mathcal{D}(\mathcal{R}, \tilde{a}, a') + Min\_Comm\_Cost(T_2, a')\}$, where $\mathcal{A}$ is the set of all possible alignments of T2.

Based on Algorithm 1, assume RT is the target array corresponding to the root node of the parse tree and $\tilde{a}$ is its alignment. Then, the minimum communication cost of evaluating the whole array expression is

$$\Omega = Min\_Comm\_Cost(RT, \tilde{a})$$

We have recursively defined the derivation of minimum communication cost. Because this optimization problem is with the property of optimal substructure and overlapping subproblems, we use the dynamic programming technique to reduce the running time. We compute the minimum communication cost in a bottom-up fashion and store the minimum cost of all the subtrees in a table. The optimal solution can then be obtained from the computed information in the table. The dynamic programming algorithm is shown below.

Algorithm 2 : Calculating the Minimum Communication Cost of An Array Expression Using Dynamic Programming.

Input:

(1) AE is the target array expression. $\mathcal{P}$ is the parse tree of AE. $\{T_1, T_2, \cdots, T_N\}$ is the temporary array in $\mathcal{P}$. T_{N+1} is the array corresponding to the root node of $\mathcal{P}$. Let $\mathcal{A}_i$ be the set of possible alignments of temporary arrays $T_i, 1 \leq i \leq N$.

Output:

The minimum communication cost of evaluating AE is stored in π[N+1,*]

Table:

π[1..N+1,1..M]: The algorithm uses this auxiliary table to record the minimum cost of communication in evaluating each temporary array including T_{N+1}. M is the maximal number of the possible alignments of temporary arrays. π[i,$\tilde{a}$] is the minimum communication cost of evaluating T_i, where the alignment of T_i is $\tilde{a}$.

Begin_of_Algorithm

FOR $Ti \in \{T_1, T_2, \cdots, T_N, T_{N+1}\}$ in Postorder of the Parse Tree DO
  IF all the child nodes of Ti are source arrays THEN
    Using owner-computer rule to compute the communication cost of evaluating the array operation corresponding to Ti.
    FOR $\tilde{a} \in \mathcal{A}_i$ DO
      π[i,$\tilde{a}$]=communication cost of evaluating T_i, where the alignment of T_i is $\tilde{a}$.
    ENDDO
  ELSE
    FOR $\tilde{a} \in \mathcal{A}_i$ DO
      Calculating the set S, P, and data access function R as in Algorithm 1.
      π[i,$\tilde{a}$]=∞
      FOR $\Lambda \in$ P DO
        • Assume Λ=$\{T_{l_1}, T_{l_2}, \cdots, T_{l_p}\}$.
        • **Temp**= $MIN_{a_{l_1} \in \mathcal{A}_{l_1}, a_{l_2} \in \mathcal{A}_{l_2}, \cdots, a_{l_p} \in \mathcal{A}_{l_p}} \{\mathcal{C}_{\mathcal{D}}(\mathcal{R}, \tilde{a}, a_{l_1}, a_{l_2}, \cdots, a_{l_p}) + \sum_{i=1}^{p} \pi[l_i, a_{l_i}]\}$
        • π[i,$\tilde{a}$]=MIN(**Temp**,π[i,$\tilde{a}$])
      ENDDO
    ENDDO
  ENDIF
ENDDO

End_of_Algorithm

However, as the problem is NP-hard and the number of all possible alignments of the temporary arrays is also potentially very large. The alignment function can be decomposed into three constituents: axis, strid, and offset. Although there are only finitely many possible axis alignments. However, there is a potential infinite space of strides and offsets[15]. It makes obtaining the optimal solution computationally prohibitive except for problems with small sizes. This motivates us to find a heuristic algorithm. (Note that even for a small problem size, the computational cost can still be very big. Theoretically, one needs to try every possible alignment. But practically, one can try only those alignments or distributions appearing in a parse tree. Or to simplify it further, one can try only those related to source or target arrays.)

5 Heuristic Algorithm to Reduce Synthesis Anomaly

Due to the fact that the optimal algorithm presented in Section 4 is NP-hard, we present a practical and efficient heuristic algorithm in this section to guide the synthesis process to reduce synthesis anomaly on distributed memory machines. Without loss of generality, we assume all the named arrays are aligned to an auxiliary cartesian grid called a template[1]. A template may be used as an abstract align-target that may then be distributed. Templates are particularly useful in the situation where one must align several arrays relative to one another but there is no need to declare a single array that spans the entire index space of interest. Even if users do not specify a global template array, a simple transformation can derive a global template array in a program. The general form of the alignment relation in HPF is:

$$!HPF\$\ ALIGN\ \ A(i_1, i_2, \cdots, i_p)\ \ with\ \ T(\ f_1(i_1, i_2, \cdots, i_p), f_2(i_1, i_2, \cdots, i_p), \cdots, f_q(i_1, i_2, \cdots, i_p)\ ) \tag{5}$$

where $f_k(i_1, i_2, \cdots, i_p)$, k=1 to q, is either a $''*''$ (broadcasting to the whole dimension) or $C_0 * i_r + C_1$, and $1 \leq$ r $\leq$ p, C_0 and C_1 are integers.

If an array A is aligned with a template T in the preceding manner, then we use the following definition to describe each reference of array A.

Definition 7 If A is aligned with T according to (5), a **reference location** of $A(g_1(i_1, i_2, \cdots, i_p),\ g_2(i_1, i_2, \cdots, i_p),\ \cdots,\ g_p(i_1, i_2, \cdots, i_p))$ with respect to template T is:

$$\begin{aligned}\mathtt{T(}\ & \mathtt{f_1(g_1(i_1, i_2, \cdots, i_p), g_2(i_1, i_2, \cdots, i_p), \cdots, gp(i_1, i_2, \cdots, i_p)),}\\ & \mathtt{f_2(g_1(i_1, i_2, \cdots, i_p), g_2(i_1, i_2, \cdots, i_p), \cdots, gp(i_1, i_2, \cdots, i_p)),}\\ & \cdots,\\ & \mathtt{f_q(g_1(i_1, i_2, \cdots, i_p), g_2(i_1, i_2, \cdots, i_p), \cdots, gp(i_1, i_2, \cdots, i_p)\ ))}\end{aligned}$$

For example, arrays A and B are aligned with template T in the following relations:

```
!HPF$ ALIGN A(i,j) WITH T(2*i-5,*,3*j)
!HPF$ ALIGN B(i,j) WITH T(i+3,*,2*j)
```

The reference location of A($2*i$,$2*j$) with respect to template T is T($2*(2*i)-5$,$*$,$3*(2*j)$)=T($4*i-5$,$*$,$6*j$). Similarly, the reference location of B($4*i-8$,$3*j$) with respect to T is T($(4*i-8)+3$,$*$,$2*(3*j)$)=T($4*i-5$,$*$,$6*j$). The reference locations of A($2*i$,$2*j$) and B($4*i-8$,$3*j$) with respect to T are identical.

For a HPF program with consecutive array operations, our optimization strategy is to synthesize continuous array operations whenever possible. We then use our heuristic algorithm to guide the process of code generation. It may rolls back a temporary array if a particular synthesis significantly increases communication costs. Consider the following code fragment.

HPF Code Fragment 7

```
!HPF$    TEMPLATE TEMP(300,300)
         REAL A(100,100), B(100,100) , C(100,100), D(100,100) , E(100,100)
         REAL F(200,100), G(300,100)
!HPF$    ALIGN A(i,j) with TEMP(i,j)
!HPF$    ALIGN B(i,j) with TEMP(i,j)
!HPF$    ALIGN C(i,j) with TEMP(i,j)
!HPF$    ALIGN D(i,j) with TEMP(i,j)
!HPF$    ALIGN E(i,j) with TEMP(i,j)
!HPF$    ALIGN F(i,j) with TEMP(3*i-1,j)
!HPF$    ALIGN G(i,j) with TEMP(2*i,j)
         C(1:100,:)=F(1:200:2,:)
```

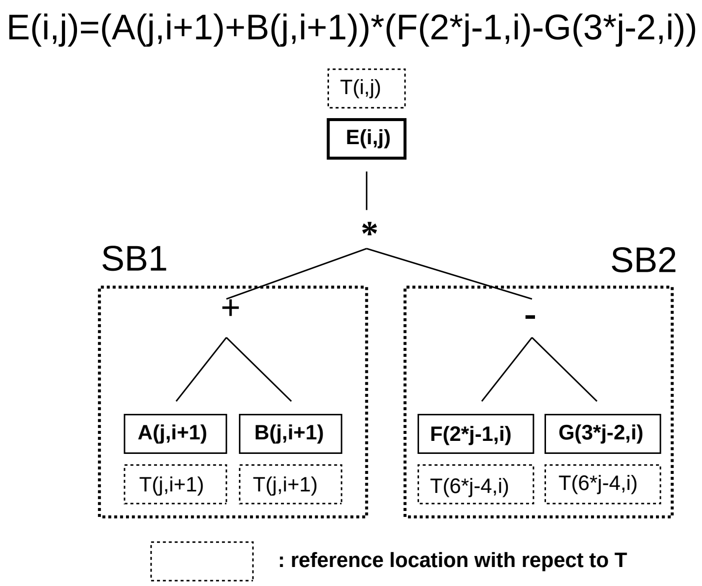

Figure 3: Expression tree of the target data access pattern

```
D(1:100,:)=G(1:300:3,:)
E=CSHIFT( TRANSPOSE(A+B),1,1 ) * (TRANSPOSE(C)+TRANSPOSE(D) )
```

It yields the following data access function to use array operation synthesis scheme to synthesize the last three statements.

$$E[i,j] = \begin{cases} (A[j,i+1]+B[j,i+1])*(F[2*j-1,i]-G[3*j-2,i]) \mid \ \phi(/\mathrm{i,j}/,/1{:}99,1{:}100/) \\ (A[j,i-99]+B[j,i-99])*(F[2*j-1,i]-G[3*j-2,i]) \mid \ \phi(/\mathrm{i,j}/,/99{:}99,1{:}100//) \end{cases}$$

A straightforward code generation would simply use the target data access pattern to generate the loop body of the parallel loop for a synthesized data access function. For example, for the first data access pattern, E[i,j] = (A[j,i+1]+B[j,i+1])∗(F[2∗j-1,i] - G[3∗j-2,i]), a straightforward method would produce a parallel loop nest as follows:

```
FORALL (i=1:99)
  FORALL (j=1:100)
    E(i,j)=(A(j,i+1)+B(j,i+1))*(F(2*j-1,i)-G(3*j-2,i))
  END FORALL
END FORALL
```

However, this code increases the communication cost and causes a synthesis anomaly. Figure 3 shows the expression tree of the data access pattern. The expression A(j, i+1) is actually aligned with B(j,i+1), as they both have the same *reference location* (see Definition 7), T(j,i+1). Similarly, F(2*j-1,i) is aligned with G(3*j-2,i), as both of them have the same reference location T(6*j-4,i). Assuming the owner-computes rule is used, we will be sending elements of arrays A and B (as well as F and G) to the owners of corresponding elements of E. This results in more communication traffics than the code without synthesis, in which only the summation of A and B (as well as the summation of C and D) is sent to the corresponding owners of E elements. Therefore, the code causes a synthesis anomaly. In the following, we propose a heuristic algorithm to guide the process of code generation. The heuristic algorithm may decide to roll back certain temporary arrays to reduce synthesis anomaly behaviors.

Algorithm 3 : A Heuristic Algorithm to Reduce Synthesis Anomaly

Input:

A *data access pattern* with its corresponding index range. The alignment of all source arrays which appear in this data access pattern.

Begin_of_Algorithm

Step 1. Construct the expression tree of the target data access pattern. (Note that the leaf nodes in the expression tree represent the usage of source arrays, root node is the target array location, and internal nodes are operators of array elements.)

Step 2. Derive the reference location with respect to template T of all of the source arrays and target array in the expression tree.

Step 3. Traverse the expression tree to find the maximal subtrees which the number of their leaf nodes is greater than 1 and all their leaf nodes have the same reference location with respect to template T, but this reference location is not the same as the root node's reference location.

Step 4. For each subtree which is found in previous step, we use a separate parallel loop to execute its corresponding subexpression and roll back the synthesis method here to re-generate a temporary array to save the intermediate results of the computation of the subtree.

Step 5. The newly generated temporary array is assigned to align with the subtree which it represents. The generated temporary array is then inserted into the original expression tree to pass the intermediate results.

End_of_Algorithm

We use the following example to illustrate Algorithm 3:

Example 7 We will use the example in Figure 3 as a running example to illustrate the heuristic algorithm.

Step 1. Figure 3 shows the expression tree of the target data access pattern.

Step 2. In Figure 3, we also show the derived reference location of each named array with respect to the template array T. In this case, A(j, i+1) and B(j,i+1) have the same reference location, T(j,i+1). Similarly, F(2*j-1,i) and G(3*j-2,i), have the same reference location T(6*j-4,i).

Step 3. We find two such subtrees, SB1 and SB2. See Figure 3.

Step 4. & *Step 5.* The algorithm generates code for the target data access pattern as follows. The temporary array got rolled back is assigned to align with the reference location of the subtree selected.

```
!HPF$ ALIGN TA1(i,j) WITH T(j,i+1)
!HPF$ ALIGN TA2(i,j) WITH T(6*j-4,i)
      ...
      FORALL (i=1:99)
        FORALL (j=1:100)
          TA1(i,j)=A(j,i+1)+B(j,i+1)
        END FORALL
      END FORALL
      FORALL (i=1:99)
        FORALL (j=1:100)
          TA2(i,j)=F(2*j-1,i)-G(3*j-2,i)
        END FORALL
      END FORALL
      FORALL (i=1:99)
        FORALL (j=1:100)
          E(i,j)=TA1(i,j)*TA2(i,j)
        END FORALL
      END FORALL
```

Time (seconds)	p=1	p=2	p=4	p=8
(1) (PURDUE-Set Prob9)	8.901	8.239	7.293	3.682
(1) Synthesized code	2.930	1.427	0.981	0.526
(1) Improvement	3.038	5.774	7.434	7.000
(2) (Magnetic E.) base code	13.988	7.007	3.511	1.787
(2) Synthesized code	8.472	4.246	2.127	1.075
(2) Improvement	1.651	1.650	1.651	1.662
(3) (Sandia Wave)	4.696	6.636	4.515	2.355
(3) Synthesized code	1.963	0.822	0.594	0.393
(3) Improvement	2.232	8.073	7.601	5.992

Table 1: Performance of the first 3 suites on Digital Alpha Farm with 8 workstations . The performance improvement is measured with the execution time of the base version over the synthesis version on an equal number of processors.

In our heuristic algorithm above, we do not roll back a single-source-node tree, as it will not produce anomaly under normal circumstances. In addition, the communication cost of codes generated from our heuristic algorithm is always better or as good as the straightforward code generation of synthesis. We show the result below.

Theorem 2 *The communication cost of the code generated from algorithm 2 is less than or equal to the communication cost of straightforward code generation.*

Proof: See Appendix A.

6 Experiments and Implementation

Three HPF programs with array operations shown in the Appendix B are used to evaluate the effectiveness of our proposed scheme. They are program fragments extracted from real applications. The execution time shown in Table 1 is obtained by running each program fragment on an 8-node DEC Farm with the DEC HPF compiler. A DEC Farm is a workstation cluster with Alpha workstations connected via FDDI networks. The base code is a straightforward compilation by using the DEC Fortran 90 compiler with HPF extensions. The synthesized codes are obtained by using our synthesis mechanism and incorporating our heuristic algorithm to reduce performance anomaly. The dimension size of the data grid in our test case is 128. The execution times are measured by running the program fragments 100 times. The first test suite is the innermost loop in the Problem 9 of Purdue Set[8]. Most of the programs in the Purdue Set were extracted from large real applications. Problem 9 is to perform a logarithmic transformation $d_i = log(1+d_i)$ for a set of data $d_i, i = 1, \cdots, N$ and compute first four Fourier moments $F_j = \Sigma_{i=1}^{N} d_i * cos(Pi * j/(N+1))$. Our optimizations improve performance by 7 times on 8 processor nodes, as it significantly cut down the amount of remote communications. The speedup is measured by the execution time of the base version over the execution time of the optimized version on the same number of processors. The HPF codes of both the base version and optimized version for the Problem 9 of the Purdue set are in Appendix C. The code fragment in the second suite is from the APULSE routine in the electromagnetic scattering problem[19]. The performance is improved by more than 60 percent regardless of the number of processors. The code fragment in the third suite is extracted from the Sandia Wave application. The performance is improved by nearly six times on eight processors.

For further observing the effect on reducing communication costs, we then focus on Table 2, where simple code fragments are used to explicitly compare the performance effects with or without our heuristic algorithm in reducing

	Number of stores	Number of loads	Communication costs	Execution time (microseconds)
The 4-th suite base code	$3*N^2$	$4*N^2$	$2*N^2$	2901
Synthesized code without Algorithm 3	N^2	$2*N^2$	N^2	1251
Synthesized code with Algorithm 3	N^2	$2*N^2$	N^2	1251
Improvement	3	2	2	2.318
The 5-th suite base code	$2*N^2$	$3*N^2$	N^2	1453
Synthesized code without Algorithm 3	N^2	$2*N^2$	$2*N^2$	1824
Synthesized code with Algorithm 3	$2*N^2$	$3*N^2$	N^2	1452
Improvement	1	1	1	1
The 6-th suite base code	$7*N^2$	$11*N^2$	$3*N^2$	17457
Synthesized code without Algorithm 3	N^2	$5*N^2$	$3*N^2$	13382
Synthesized code with Algorithm 3	$2*N^2$	$6*N^2$	N^2	10267
Improvement	3.5	1.833	3	1.700
The 7-th suite base code	$8*N^2$	$13*N^2$	$6*N^2$	19178
Synthesized code without Algorithm 3	N^2	$6*N^2$	$5*N^2$	12006
Synthesized code with Algorithm 3	$3*N^2$	$8*N^2$	$2*N^2$	7858
Improvement	2.667	1.625	3	2.441
The 8-th suite base code	$4*N^2$	$9*N^2$	N^2	5361
Synthesized code without Algorithm 3	N^2	$6*N^2$	$6*N^2$	5505
Synthesized code with Algorithm 3	$2*N^2$	$7*N^2$	N^2	3994
Improvement	2	1.286	1	1.342

Table 2: Performance of five suites on nCUBE/2. The performance improvement is measured with the execution time of the base version over the synthesis version with Algorithm 3 on sixteen processors.

synthesis anomaly. We amplify these effects by constructing examples which can actually produce synthesis anomaly behaviors. The experiment is done in nCUBE/2 with 16 nodes. In this experiment, we no longer use an HPF compiler, but to generate explicit message passing codes by calculating the sending set and receiving set. In the base code part, the alignments of the temporary arrays are carefully chosen according to Chatterjee's scheme[15] so as to reduce communication costs. There is no synthesis optimization in the base version. In addition, the experiment is conducted in a way that all the factors can be under control, as the DEC HPF compiler is usually like a black box to us. Without our heuristic, a naive application of synthesis scheme results in performance anomaly in the Test Suite 5 and 8. In the Test Suite 6 and 7, the application of our heuristic algorithm can further improves the performance of the synthesized code.

Currently, we have a preliminary implementation of our proposed scheme, called SYNTOOL. It provides a prototype for the experiments in this section. The SYNTOOL takes data access functions as inputs and performs an automatic array operation synthesis to obtain a synthesized data access function as discussed in this paper.

7 Conclusion

An increasing number of programming languages, such as Fortran 90, HPF, and APL, are providing a rich set of intrinsic array functions and array expressions. These constructs which constitute an important part of data parallel languages provide excellent opportunities for compiler optimizations. Previously, we proposed an array operation synthesis scheme to synthesize multiple consecutive array functions or array expressions by composing several data access functions into an equivalent composite reference pattern. Thus, the synthesis can improve performance by reducing redundant data movement, temporary storage usage, and parallel loop synchronization overhead. Performance benefits had been demonstrated for programs running on shared memory multiprocessor machines with uniform memory accesses in earlier work.

In this paper, we propose solutions to an open problem how the synthesis scheme can be incorporated into optimizing HPF programs on distributed memory parallel machines by taking into account communication costs. We first propose an optimal solution for array synthesis on distributed memory machines and show that the optimal problem is NP-hard. Therefore, we develop a practical heuristic algorithm for compilers to devise synthesis strategy on distributed memory machines with HPF programs. Experimental results show speedups from 1.60 to 8.0 for HPF code fragments from real applications on a DEC Farm by incorporating our proposed optimizations. We have demonstrated that it is also profitable in applying the array synthesis optimization to programs running on distributed memory parallel machines.

References

[1] C. Koelbel, D. Loveman, R. Schreiber, G. Steele and M. Zosel. *The High Performance Fortran Handbook*, MIT-press, Cambridge, 1994.

[2] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-ching Ju. *An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs*, Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pp. 112-122, July 1995.

[3] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-ching Ju. *A Functional Approach to Synthesize Fortran 90 Array Operations*, in submission to Journal of Parallel and Distributed Computing. Available on WWW at `http://falcon.cs.nthu.edu.tw/~ghhwang/papers.html`.

[4] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook complete ANSI/ISO reference*, Intertext Publications McGraw-Hill Book Company, 1992.

[5] J. Li and M. Chen. *The data alignment phase in compiling programs for distributed-memory machines*, Journal of parallel and Distributed Computing, Vol. 13, pp. 213-221, Oct. 1991.

[6] J. Li and M. Chen. *Compiling Communication-Efficient Programs for Massively Parallel Machines*, IEEE Tran. On Parallel and Distributed Systems, Vol. 2, No. 3, July 1991.

[7] Jenq Kuen Lee and D. Gannon. *Object-Oriented Parallel Programming: Experiments and Results*, *Proceedings of Supercomputing '91*, New Mexico, November, 1991.

[8] J.R. Rice, J. Jing. *Problems to Test Parallel and Vector Languages*, Purdue University Tech. Rep. CSD-TR-1016, 1990.

[9] J.M. Anderson and M.S. Lam. *Global optimizations for parallelism and locality on scalable parallel machines*, in Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, pp. 112-125, June 1993.

[10] J.R. Gilbert and R. Schreiber. *Optimal expression evaluation for data parallel architectures*, Journal of Parallel and Distributed Computing, Vol.13, pp. 58-64, Sept. 1991.

[11] Fox, G. C., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, G.-W, and Wu, M.-Y. 1990. *Fortran D language specification.* Tech. Rep. Rice COMP TR90-141(Dec.), Dept. of Computer Science, Rice University, Houston, Texas.

[12] L. J. Guibas and D. K. Wyatt. *Compilation and Delayed Evaluation in APL*, Proceeding of the Fifth Annual ACM Symp. on Principles of Programming Languages, pp. 1-8, 1978.

[13] K. Knob, J.D. Lukas, and G.L. Steele. *Data optimization: allocation of arrays to reduce communication on SIMD Machines*, Journal of parallel and Distributed Computing, Vol.2, pp. 102-118, Feb. 1990.

[14] M. Gupta and P. Banerjee. *Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers*, IEEE Transactions on Parallel and Distributed Systems, Vol. 3, pp. 179-193, Mar. 1992.

[15] S. Chatterjee, J.R. Gilbert, R.Schreiber, and S.-H. Teng. *Automatic Array Alignment in Data-Parallel Programs*, Proceedings of the Twentieth Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, January 1993, pp. 16-28.

[16] T. J. Sheffler, R. Schreiber, J.R. Gilbert, and S. Chatterjee. *Aligning Parallel Arrays to Reduce Communication,* Proceedings of Frontiers '95, McLean, VA, February 1995, pp. 324-331.

[17] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to FORTRAN 90*, McGraw-Hill Book Company, 1990.

[18] Zima, H., and Chapman, B. *Compiling for distributed-memory systems.* Proceeding IEEE 81, 2 (Feb.1993), 264-287.

[19] A. Mohamer, G. Fox, G. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, N. and Yeh. *Applications Benchmark Set for Fortran-D and High Performance Fortran*, Technical Report SCCS327, NPAC, Syracuse University.

A Proofs of Theorems

Theorem 1 *The complexity of calculating Ω is NP-hard.*

Proof: The function C(U,AE) in calculating Ω can be easily reduced from the data alignment problem which has been proven to be NP-complete[16, 5]. Let AE be an array expression and the parse tree constructed from AE is with N temporary arrays, the data alignment problem is the same problem of calculating C(U,AE), where U=$\{T_1, T_2, \cdots, T_N\}$. Since the complexity of choosing alignment of temporary arrays is NP-complete, the calculating of Ω is NP-hard. *Q.E.D*

Theorem 2 *The communication cost of the code generated from the above algorithm is less than or equals the communication cost of straightforward code.*

Proof: Assume $\mathcal{E}$ is the expression tree constructed from the target data access pattern. Let $\mathcal{S} \in \mathcal{E}$ and be a maximum subtree which all its leaf nodes have the same reference location with respect to the global template. (Without loss of generality, we assume there is only one such subtree in $\mathcal{E}$.) The segmentation descriptor corresponding to the target data access pattern is γ. Based on the owner-computes rule, the communication cost for computing the subexpression corresponding to $\mathcal{S}$ is:

$$|\mathcal{S}| * \sum_{(i_1,\cdots,i_n)\in\gamma} \mathcal{D}(T[f_1(i_1,\cdots,i_n),\cdots,f_p(i_1,\cdots,i_n)], T[g_1(i_1,\cdots,i_n),\cdots,g_q(i_1,\cdots,i_n)])$$

where $|\mathcal{S}|$ is the number of leaf nodes in $\mathcal{S}$; $T[f_1(i_1,\cdots,i_n)$, $\cdots$, $f_p(i_1,\cdots,i_n)]$ and $T[g_1(i_1,\cdots,i_n)$, $\cdots$, $g_q(i_1,\cdots,i_n)]$ are the reference location of $\mathcal{S}$ and the reference location of the root node of $\mathcal{E}$, respectively.

The algorithm will allocate a temporary array to save the results of $\mathcal{S}$. Since the reference location of this temporary array is the same as all the nodes in $\mathcal{S}$, there will be no communication cost in computing the subexpression corresponding to $\mathcal{S}$. This temporary array has a different reference location with the target array in the root node of $\mathcal{E}$. Based on the owner-computes rule, the communication cost for computing the subexpression corresponding to $\mathcal{S}$ becomes

$$1 * \sum_{(i_1,\cdots,i_n)\in\gamma} \mathcal{D}(T[f_1(i_1,\cdots,i_n),\cdots,f_p(i_1,\cdots,i_n)], T[g_1(i_1,\cdots,i_n),\cdots,g_q(i_1,\cdots,i_n)])$$

Q.E.D

B Code Fragments for Experiments

1.
```
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
      REAL    T(N,N), U(N,N),RIP(N,N),RIN(N,N)
      REAL    B(N,N),PATTERN(N,N),ERRM(N,N)
!HPF$ ALIGN T(i,j) WITH temp(i,j)
!HPF$ ALIGN U(i,j) WITH temp(i,j)
!HPF$ ALIGN RIP(i,j) WITH temp(i,j)
!HPF$ ALIGN RIN(i,j) WITH temp(i,j)
!HPF$ ALIGN B(i,j) WITH temp(i,j)
!HPF$ ALIGN PATTERN(i,j) WITH temp(i,j)
!HPF$ ALIGN ERRM(i,j) WITH temp(i,j)
  RIP=EOSHIFT(U,DIM=1,SHIFT=1)
  RIN=EOSHIFT(U,DIM=1,SHIFT=-1)
  T=U+RIP+RIN+EOSHIFT(U,DIM=2,SHIFT=1)+
*     EOSHIFT(U,DIM=2,SHIFT=-1)+
*     EOSHIFT(RIP,DIM=2,SHIFT=1)+
*     EOSHIFT(RIP,DIM=2,SHIFT=-1)+
*     EOSHIFT(RIN,DIM=2,SHIFT=1)+
*     EOSHIFT(RIN,DIM=2,SHIFT=-1)
  T=T/PATTERN
  WHERE (ABS(U).GT.0.001)
    ERRM=ABS((U-T)/U)
  ELSEWHERE
    ERRM=ABS((U-T)/0.001)
  ENDWHERE
```

2.
```
!HPF$ TEMPLATE temp(N*N,N*N)
!HPF$ DISTRIBUTE temp(CYCLIC,CYCLIC)
      REAL     KIA(N,N),XMA(N,N),YI(N,N),KMM(N),XM(N)
      INTEGER  KM,KI,KII(N)
!HPF$ ALIGN KIA(i,j) WITH temp(i,j)
!HPF$ ALIGN XMA(i,j) WITH temp(i,j)
!HPF$ ALIGN YI(i,j) WITH temp(i,j)
!HPF$ ALIGN KII(i) WITH YI(i,*)
!HPF$ ALIGN XM(j) WITH YI(*,j)
!HPF$ ALIGN KMM(i) WITH XM(i)
 KIA=SPREAD(KII,DIM=2,NCOPIES=KM)
 KJA=SPREAD(KJJ,DIM=1,NCOPIES=KN)
 XM=(KMM+0.5)*HW1/KM*SGN(START1)
 XN=(KNN+0.5)*HW2/KN*SGN(START2)
 XMA=SPREAD(XM,DIM=1,NCOPIES=KI)
 XNA=SPREAD(XN,DIM=2,NCOPIES=KJ)
 WHERE(KIA.EQ.1)
 YI=1./SQRT(HW1)
 ELSEWHERE
 YI=SQRT(2./HW1)*COS(3.141592654*(KIA-1)*XMA/HW1)
 END WHERE
 WHERE(KJA.EQ.1)
 YJ=1./SQRT(HW2)
 ELSEWHERE
 YJ=SQRT(2./HW2)*COS(3.141592654*(KJA-1)*XNA/HW2)
 END WHERE
```

3.
```
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
      REAL     f1(N,N), zxp(N,N), zxm(N,N)
      REAL     zyp(N,N), zym(N,N), fderiv(N,N)
      REAL     fxp(N,N), fxm(N,N), fyp(N,N), fym(N,N)
!HPF$ ALIGN f1(i,j) WITH temp(i,j)
!HPF$ ALIGN zxp(i,j) WITH temp(i,j)
!HPF$ ALIGN zxm(i,j) WITH temp(i,j)
!HPF$ ALIGN zyp(i,j) WITH temp(i,j)
!HPF$ ALIGN zym(i,j) WITH temp(i,j)
!HPF$ ALIGN fderiv(i,j) WITH temp(i,j)
!HPF$ ALIGN fxp(i,j) WITH temp(i,j)
!HPF$ ALIGN fxm(i,j) WITH temp(i,j)
!HPF$ ALIGN fyp(i,j) WITH temp(i,j)
!HPF$ ALIGN fym(i,j) WITH temp(i,j)
 FXP = CSHIFT(F1,1,+1)
 FXM = CSHIFT(F1,1,-1)
 FYP = CSHIFT(F1,2,+1)
 FYM = CSHIFT(F1,2,-1)
 FDERIV = ZXP * (FXP - F1) + ZXM * (FXM - F1) +
         ZYP * (FYP - F1) + ZYM * (FYM - F1)
```

4.
```
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
      REAL     A(N,N), B(N,N), C(N,N)
!HPF$ ALIGN A(i,j) WITH temp(i,j)
!HPF$ ALIGN B(i,j) WITH temp(i,j)
!HPF$ ALIGN C(i,j) WITH temp(i,j)
 C=TRANSPOSE(TRANSPOSE(A,1)+B)
```

5.
```
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
      REAL     A(N,N), B(N,N), C(N,N)
!HPF$ ALIGN A(i,j) WITH temp(i,j)
!HPF$ ALIGN B(i,j) WITH temp(i,j)
!HPF$ ALIGN C(i,j) WITH temp(i,j)
 C=TRANSPOSE( A+B , 1)
```

6.
```
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
      REAL     A(N,N), B(N,N), C(N,N), D(N,N), E(N,N), F(N,N)
!HPF$ ALIGN A(i,j),B(i,j),C(i,j),D(i,j),E(i,j),F(i,j)  WITH temp(i,j)
       F=TRANSPOSE( SIN(TRANSPOSE(A)+TRANSPOSE(B)+C*D-
 E) )
```

7.
```
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
      REAL     A(N,N), B(N,N), C(N,N), D(N,N), E(N,N), F(N,N)
      REAL     G(N,N), H(N,N), I(N,N), J(N,N), K(N,N)
      LOGICAL  LA(N,N)
!HPF$ ALIGN A(i,j),B(i,j),C(i,j),D(i,j),E(i,j),F(i,j)  WITH temp(i,j)
!HPF$ ALIGN G(i,j),H(i,j),IA(i,j),JA(i,j),KA(i,j),LA(i,j)  WITH temp(i,j)
      FORALL (i=1:N,j=1:N)
        A(i,j)=B(N-i+1,j)
      END FORALL
      FORALL (i=1:N,j=1:N)
        C(i,j)=D(N-i+1,j)
      END FORALL
      FORALL (i=1:N,j=1:N)
        E(i,j)=F(N-i+1,j)
      END FORALL
      G=TRANSPOSE(H)
      IA=TRANSPOSE(JA)
      KA=MERGE(A+C+E,G+IA,LA)
```

8.
```
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
    REAL    A(N,N), B(N,N), C(N,N), D(N,N), E(N,N), F(N,N), G(N,N)
!HPF$ ALIGN A(i,j),B(i,j),C(i,j),D(i,j),E(i,j),F(i,j), G(i,j)  WITH temp(i,j)
      G=CSHIFT( (A+B+C+D)*E-F),1,3)
```

C The HPF Code of the First Test Suite

The HPF code of Problem 9 of Purdue set used in experiments is listed.

```
      PARAMETER (N=256,N_ITER=100)
      REAL TARRAY(2),TIME1,TIME2
      INTEGER I,J,ITER
      INTEGER NK,MK
      LOGICAL K
!HPF$ TEMPLATE temp(N,N)
!HPF$ DISTRIBUTE temp(BLOCK,BLOCK)
      REAL     rnum1,rnum2
      REAL     T(N,N), U(N,N),RIP(N,N),RIN(N,N)
      REAL     B(N,N),PATTERN(N,N),ERRM(N,N)
!HPF$ ALIGN T(i,j) WITH temp(i,j)
!HPF$ ALIGN U(i,j) WITH temp(i,j)
!HPF$ ALIGN RIP(i,j) WITH temp(i,j)
!HPF$ ALIGN RIN(i,j) WITH temp(i,j)
!HPF$ ALIGN B(i,j) WITH temp(i,j)
!HPF$ ALIGN PATTERN(i,j) WITH temp(i,j)
!HPF$ ALIGN ERRM(i,j) WITH temp(i,j)

      PRINT *,'————Base code running———'
       NK=N
       MK=N
       PATTERN=4.0
       PATTERN(2:NK-1,2:MK-1)=9.0
       PATTERN(1,2:MK-1)=6.0
       PATTERN(NK,2:MK-1)=6.0
       PATTERN(2:NK-1,1)=6.0
       PATTERN(2:NK-1,MK)=6.0
       U=SPREAD([1:NK]*([1:NK]+1),2,MK) &
       + SPREAD(REAL([1:MK])/([1:MK]+1.0),1,NK)

      TIME1=DTIME(TARRAY)

      DO ITER=1,N_ITER
        RIP=EOSHIFT(U,DIM=1,SHIFT=1)
        RIN=EOSHIFT(U,DIM=1,SHIFT=-1)
        T=U+RIP+RIN+EOSHIFT(U,DIM=2,SHIFT=1)+ &
          EOSHIFT(U,DIM=2,SHIFT=-1)+ &
          EOSHIFT(RIP,DIM=2,SHIFT=1)+ &
          EOSHIFT(RIP,DIM=2,SHIFT=-1)+ &
          EOSHIFT(RIN,DIM=2,SHIFT=1)+ &
          EOSHIFT(RIN,DIM=2,SHIFT=-1)
        T=T/PATTERN
        WHERE (ABS(U).GT.0.001)
              ERRM=ABS((U-T)/U)
        ELSEWHERE
              ERRM=ABS((U-T)/0.001)
        ENDWHERE
      ENDDO
      TIME2=DTIME(TARRAY)

      PRINT *, 'TIME',TIME2,' SECONDS'
      PRINT *, 'USER:',TARRAY(1),' SYSTEM:',TARRAY(2)

      PRINT *,'————Optimized code running ———'
       NK=N
       MK=N
       PATTERN=4.0
       PATTERN(2:NK-1,2:MK-1)=9.0
       PATTERN(1,2:MK-1)=6.0
       PATTERN(NK,2:MK-1)=6.0
       PATTERN(2:NK-1,1)=6.0
       PATTERN(2:NK-1,MK)=6.0
       U=SPREAD([1:NK]*([1:NK]+1),2,MK) &
       + SPREAD(REAL([1:MK])/([1:MK]+1.0),1,NK)

      TIME1=DTIME(TARRAY)
      DO ITER=1,N_ITER
!S1
```

```
       FORALL (I=2:N-1,J=2:N-1)
         T(i,j)= (U(i,j)+U(i+1,j)+U(i-1,j)+ &
                 U(i,j+1)+U(i,j-1)+U(i+1,j+1)+ &
                 U(i+1,j-1)+U(i-1,j+1)+ &
                 U(i-1,j-1) ) / PATTERN(i,j)
         ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                         ABS(U(I,J)-T(I,J))/ 0.001, &
                         ABS(U(I,J)).GT.0.001 )
       END FORALL
!S2
       FORALL (I=N:N,J=2:N-1)
              T(i,j)= (U(i,j)+U(i-1,j)+ &
                      U(i,j+1)+U(i,j-1)+ &
                      U(i-1,j+1)+ &
                      U(i-1,j-1) ) / PATTERN(i,j)
              ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                              ABS(U(I,J)-T(I,J))/ 0.001, &
                              ABS(U(I,J)).GT.0.001 )
       END FORALL
!S3
       FORALL (I=2:N-1,J=N:N)
              T(i,j)= (U(i,j)+U(i+1,j)+U(i-1,j) &
                      +U(i,j-1)+ &
                      U(i+1,j-1)+ &
                      U(i-1,j-1) ) / PATTERN(i,j)
              ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                              ABS(U(I,J)-T(I,J))/ 0.001, &
                              ABS(U(I,J)).GT.0.001 )
       END FORALL
!S4
       FORALL (I=N:N,J=N:N)
              T(i,j)= (U(i,j)+U(i-1,j) &
                      +U(i,j-1)+ &
                      U(i+1,j-1)+ &
                      U(i-1,j-1) ) / PATTERN(i,j)
              ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                              ABS(U(I,J)-T(I,J))/ 0.001, &
                              ABS(U(I,J)).GT.0.001 )
       END FORALL
!S5
       FORALL (I=1:1,J=2:N-1)
              T(i,j)= (U(i,j)+U(i+1,j) &
                      +U(i,j+1)+U(i,j-1)+ &
                      U(i+1,j-1)+ &
                      U(i+1,j+1) ) / PATTERN(i,j)
              ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                              ABS(U(I,J)-T(I,J))/ 0.001, &
                              ABS(U(I,J)).GT.0.001 )
       END FORALL
!S6
       FORALL (I=1:1,J=N:N)
              T(i,j)= (U(i,j)+U(i+1,j) &
                      +U(i,j-1)+ &
                      U(i+1,j-1)+ &
                      U(i+1,j+1) ) / PATTERN(i,j)
              ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                              ABS(U(I,J)-T(I,J))/ 0.001, &
                              ABS(U(I,J)).GT.0.001 )
       END FORALL
!S7
       FORALL (I=2:N-1,J=1:1)
              T(i,j)= (U(i,j)+U(i+1,j)+U(i-1,j) &
                      +U(i,j+1)+ &
                      U(i-1,j+1)+ &
                      U(i+1,j+1) ) / PATTERN(i,j)
              ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                              ABS(U(I,J)-T(I,J))/ 0.001, &
                              ABS(U(I,J)).GT.0.001 )
       END FORALL
!S8
       FORALL (I=N:N,J=1:1)
              T(i,j)= (U(i,j) + U(i-1,j) &
                      +U(i,j+1)+U(i,j-1) + &
                      U(i-1,j+1) ) / PATTERN(i,j)
              ERRM(i,j)=MERGE( ABS(U(I,J)-T(I,J))/T(I,J), &
                              ABS(U(I,J)-T(I,J))/ 0.001, &
                              ABS(U(I,J)).GT.0.001 )
       END FORALL
      ENDDO
      TIME2=DTIME(TARRAY)

      PRINT *, 'TIME',TIME2,' SECONDS'
      PRINT *, 'USER:',TARRAY(1),' SYSTEM:',TARRAY(2)

      STOP
      END
```