Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Non-local Data Reuse in Data Parallel Compiler*

Hyun-Gyoo Yook† Sung-Soon Park‡ Mi-Soon Koo† Myong-Soon Park†

† Department of Computer Science
Korea University
Seoul, Korea

‡ Department of Computer Science
Anyang University
Anyang, Korea

{hyun, pss, kms, myongsp}@cslab1.korea.ac.kr

Abstract

Data parallel language is treated as a promising paradigm to describe a parallel program. In most of data parallel languge, non-local data is disposable. Every time there needs non-local data, communication should be done. However some non-local data are used more than twice before they are redefined. To eliminate redundant communication for them, they must be reused. CHAOS and some communication optimization try to reuse that but their efforts are limited because of the difficulty of coherence between orignal data and copied one.

We propose a new non-local data reuse method. Our method uses data flow dependence to know the redefinition time of non-local data. Our method is composed of two stages. In the first stage, it detects redefinition time and reusable time of non-local data, and in the second stage, it transforms original code to reuse non-local data. We solve the coherence problem and reuse more non-local data. We make an experiment to know the effect of our method in networked workstation environment.

1 Introduction

Nowadays, when increasing the speed of processors become more and more difficult, more and more computer experts admit that the future of high performance computing belongs to parallel computer, such as distributed memory computer. Distributed memory computer offers very high levels of flexibility, scalability and performance. However, in distributed memory environment, programmers are forced to consider the details of computations in each processor and communications between them.

A data parallel language is introduced to make up for the deficiencies of distributed memory computer. In the data parallel language, a programmer doesn't need to consider the details of parallel computation and communication. The only one thing he should do is to specify a distribution pattern of data. It makes parallel program more robust and sometimes more powerful[1, 6].

*This research was supported by SERI under contact No. 96-12-D00450

A compiler for data parallel language performs computation partitioning and communication insertion. According to owner computes rule, which is generally used in computation partitioning, the assignment statement is executed by the owner of LHS(Left-Hand Side). So communications are essential for non-local data in RHS(Right-Hand Side)[5]. Because we know that data reuse is common in scientific programs, non-local data must be reused in distributed memory computer.

For example, the data distribution and access pattern of following HPF [1] program, is Figure 1.

```
      PROGRAM F_PREFIX
      REAL A(100), B(100)
CHPF$ PROCESSOR P(4)
CHPF$ ALIGN B WITH A
CHPF$ DISTRIBUTE A(BLOCK) ONTO P
      DO 100 i=1,N
      DO 100 j=1,i-1
         A(i) = f(B(i-j))
100   CONTINUE
      END
```

According to this pattern, each processor needs not only its own data but neighborhood's data. For A(26), processor 1 needs the data B(25) to B(1) from processor 0 and for A(27), B(26) to B(1) are necessary. B(26) is saved in processor 1, but B(25) to B(1) is not in processor 1 because they were discarded after the using for A(26). As a result, processor 1 communicates for 25 times in each non-local data. To reduce this burden, processor 1 should have saved the non-local data when it received them. If processor 1 saved the non-local data for future reuse, the number of communication is reduced to 1 for each of them.

However, most data parallel compiler has not considered the reuse of non-local data. Only message coalescing and CHAOS try to reuse them, but they can't solve coherence problem between original data and copied data. So their capability is weak. We propose a new non-local data reuse method. Our method consider the flow data dependence to solve the coherence problem. Our method detects the time that non-local data is used for the first and the time that the non-local data is reused. When non-local data is used

[1]HPF is a data parallel version of Fortran90[6]

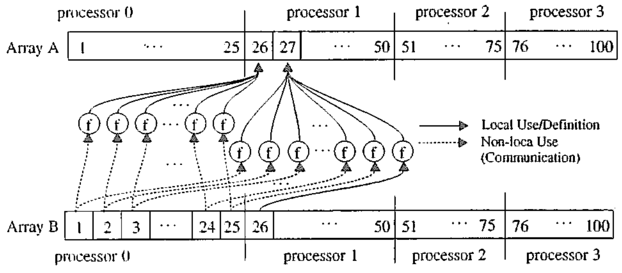

Figure 1: Data distribution and access pattern of F_PREFIX

for the first time, it must be fetched from remote processor and saved in local memory. After that, it can be reused.

The rest of this paper is organized as follows. Section 2 describes the non-local reuse method in message coalescing and CHAOS. Section 3 describes our non-local data reuse method. This section is composed of two subsections. One analyzes given loop and detects iteration spaces for reuse and the other generates target loop. Section 4 shows experimental results in networked workstation environment. And section 5 concludes this paper.

2 Related Works

In this chapter, we explain the reusing way of message coalescing and CHAOS Message coalescing, one of communication optimizations, try to reuse non-local data by deleting redundant communication in a loop. CHAOS propose more systematic method to reuse non-local data in irregular problem.

2.1 Message Coalescing

Message coalescing is one of communication optimizations in data parallel language. Message coalescing eliminate redundant message in a loop and as a result reuse non-local data[5]. To apply message coalescing, all messages in a loop is placed before the loop by message vectorization. For example, in the following loop,

```
    DO 100 i = 1, 90
       t1 = RECEIVE(B(i))
       t2 = RECEIVE(B(i+10))
       A(i) = f(t1, t2,...)
100 CONTINUE
```

message vectorization extracts all communications before loop, like this.

```
    DO 50 i = 1, 90
       t1(i) = RECEIVE(B(i))
       t2(i) = RECEIVE(B(i+10))
50  CONTINUE
    DO 100 i = 1, 90
       A(i) = f(t1(i), t2(i),...)
100 CONTINUE
```

In this loops, the data from B(11) to B(90) are received for two times. Message coalescing eliminate these redundant communiactions by colapsing the two RECEIVE statements in one. The loops after applying message coalescing is this.

```
    DO 50 i = 1, 100
       t(i) = RECEIVE(B(i))
50  CONTINUE
    DO 100 i = 1, 90
       A(i) = f(t(i), t(i+10),...)
100 CONTINUE
```

But message coalescing cannot reuse the data which is defined in a loop because it performs all communication before the loop.

2.2 CHAOS

CHAOS is runtime support system for irregular problem. In irregular problems, the communication pattern depends on the input data. So it is not possible to predict at compile time what data must be prefetched. The lack of information is dealt with by transforming the original parallel loop into two constructs called an inspector and an executor [4].

During program execution, the inspector examines the data references made by a processor, and calculates what non-local data needs to be fetched. The executor then use the information from the inspector to implement the actual computation. For the redundant non-local data, the inspector makes a schedule not to fetch it again and the executor get it from local memory instead of fetching from other processor. CHAOS uses a hash table to store non-local data.

However, there are several limitations in CHAOS. First, CHAOS also cannot handle the data which is defined and used in a loop, because as message coalescing the executor must fetch all the non-local data before the loop. Second, CHAOS uses hash table to temporarily store non-local data, which may be irregular patterned. But it is inefficient in regular problems, because it takes more time to access and needs more memory space than sequential memory.

3 Non-local Data Reuse based on Flow Data Dependence

We found that the redundant use information in the program is extracted from flow data dependences. For example, in the following program,

```
1  A   = ...
2  ... = f(A,...)
3  ... = g(A,...)
4  A   = ...
5  ... = h(A,...)
```

there are three flow data dependences as Figure 2. $\delta_{i,j}$means the flow dependence from statement i to statement j. Because the uses of A in statement 2 and statement 3 depend on the definition of statement 1 by $\delta_{1,2}$ and $\delta_{1,3}$, they use same data. But the use in statement 5 is different. It depend on the statement 4. If the statement 1, 4 are executed in processor 1 and the others are executed in processor 2, only **use** in statement 3 is reusable. From this example, we can find that the first **use** in data dependences which have same definition use new data and the other **uses** use same data with first **use**. In loop, the statement corresponds to iteration space.

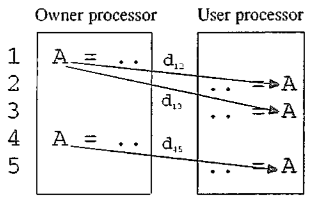

Figure 2: Example for Non-local Data Reuse

In loop, Our method find *a non-local first use iteration space (FUIS)*, in which non-local data must be fetched from remote processor and *a non-local reuse iteration space (RUIS)* in which non-local data is reused, with the information of flow data dependence, data distribution, and execution order. In the first part of this section introduce the algorithm which detects these iteration spaces. In the second part, these iteration spaces are used in code generation. Unlike message coalescing or CHAOS, we don't move communications before loop. We reshape the given loop to isolate these iteration spaces. In this process, we try to minimize the overhead of guard.

3.1 Iteration Space Detection Algorithm

This chapter explains an algorithm which detects three iteration spaces. First one is FUIS in which non-local data is fetched from remote processor, another one is RUIS in which non-local data is reused, and the last one is RIS, which include FUIS and RUIS and can be expressed in lower bound, upper bound and stride. The algorithm is based on data dependence information.

Distance vector and direction vector are generally used to represent data dependence in loop but they are insufficient. Distance vector cannot represent many cases and the direction vector is too ambiguous. For example, there are loop-carried data dependences in the following loop. The data which are defined in iterations 2, 4, 6, 8,and 10 are used in iterations 6, 7, 8, 9, and 10, respectively.

```
      DO i = 1, 10
S1       A(i) = A(2*i - 10)
      ENDDO
```

The distances from **def** to **use** are not constant, so distance vector cannot represents these dependences. On the other hand there is no information about distance in direction vector, '(<)'. Kelly proposed dependence mapping, which represents dependence as mapping between iteration spaces like this[3].

$$\delta^f_{1,1} : \{[i] \rightarrow [\frac{1}{2} * i + 5] | 2 \leq i \leq 10, i mod 2 = 0\}$$

$\delta^f_{p,q}$ means flow data dependence from statement p to statement q. $[i]$ is the iteration space of **def** and $[2 * i - 4]$ is the iteration space of **use**. Dependence mapping is difficult to handle but it can represent more dependences and contains more information.

The algorithm in Figure 3 detectes FUIS, RUIS, and FIS. The algorithm is based on following information

Definition 1. IS(d, u) is an iteration space of **use** u which uses the data defined in **def** d. This information is obtained from flow data dependence from d to u.

Definition 2. LIS(u, np) is a local iteration space that the statement contain u was executed in processor np.

Definition 3. NLUIS(u, np) is an iteration space in which u use non-local data.

For example, in the following loop,

```
     DO i = 1, 100
1       B(i) = B(i-5)
2       A(i) = B(2*i -32)
     ENDDO
```

there are 1 **def** and 2 **uses**. The data dependences between them are like these :

$$\delta^f_{1,1} \quad : \quad \{[i] \rightarrow [i-5] | 1 \leq i \leq 100\}$$
$$\delta^f_{1,2} \quad : \quad \{[i] \rightarrow [\frac{1}{2} \times i + 16] | 1 \leq i \leq 32, i \; mod \; 2 \; = \; 0\}$$

These two dependeces have same **def**, so the **uses** of the dependences can use same data. The iteration spaces of each **use**, $IS(u, d)$, are these :

$$\begin{aligned} IS(1,1) &= \{i | 6 \leq i \leq 105\} \\ IS(1,2) &= \{i | 17 \leq i \leq 32\} \end{aligned}$$

$IS(u, d)$ corresponds to the iteration space of the range in dependence mapping.

To get the $LIS(u, np)$ and $NLUIS(u, np)$, data distribution information is necessary. Figure 4 is the distribution information of array A and B. According

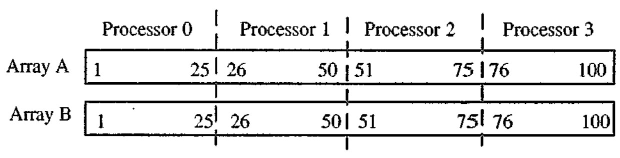

Figure 4: Data Distribution

to this information, process 1 has A(26) to A(50) and B(26) to B(50). If computation is partitioned by owner-computes rule, the iteration spaces which process 1 executes are these :

$$\begin{aligned} LIS(1,1) &= \{i | 26 \leq i \leq 50\} \\ LIS(2,1) &= \{i | 26 \leq i \leq 50\} \end{aligned}$$

The iteration spaces in which each **use** exceeds the data boundary allocated in process 1, $NLUIS(u, 1)$,

```
Algorithm: FindISs
1. Detect data dependences
2. Detect data distribution pattern
3. For all $d$ which define a data $a$, DO
    (a) For all $u$ which use the data $a$, DO
        i. $NLIS(d,u,np) = IS(d,u) \cap LIS(u,np) \cap NLIS(u,np)$
        ii. $NLD(d,u,np) = Is2Ds(NLIS(d,u,np), index\ of\ u)$
        iii. For all $u'(\neq u)$ which use the data $a$, DO
            A. $FIS(u) = FIS(u) \cap iteration\ space\ in\ which\ u\ executes\ before\ u'$
            B. $FIS(u') = FIS(u') \cap iteration\ space\ in\ which\ u'\ executes\ before\ u$
    (b) For all $u$ which use the data $a$, DO
        i. For all $u'$ $(\neq u)$ which use the data $a$, DO
            A. $FUIS(d,u,np) = FUIS(d,u,np) \cup (Ds2Is(NLD(d,u,np)$
               $\cap NLD(d,u',np), index\ of\ u) \cap FIS(u))$
            B. $FUIS(d,u',np) = FUIS(d,u',np) \cup (Ds2Is(NLD(d,u',np)$
               $\cap NLD(d,u,np), index\ of\ u') \cap FIS(u'))$
            C. $RUIS(d,u,np) = RUIS(d,u,np) \cup (Ds2Is(NLD(d,u,np)$
               $\cap NLD(d,u',np), index\ of\ u) - FIS(u))$
            D. $RUIS(d,u',np) = RUIS(d,u',np) \cup (Ds2Is(NLD(d,u',np)$
               $\cap NLD(d,u,np), index\ of\ u') - FIS(u'))$
            E. $RIS(d,u,np) = RIS(d,u,np) \uplus Ds2Is(NLD(d,u,np)$
               $\cap NLD(d,u',np), index\ of\ u)$
            F. $RIS(d,u',np) = RIS(d,u',np) \uplus Ds2Is(NLD(d,u',np)$
               $\cap NLD(d,u,np), index\ of\ u')$
```

Figure 3: Algorithm for detecting FUIS, RUIS, and RIS

are these :

$$\begin{aligned} NLUIS(1,1) &= \{i \mid i-5 \leq 25 \vee i-5 \geq 51\} \\ &= \{i \mid i \leq 30 \vee i \geq 56\} \\ NLUIS(2,1) &= \{i \mid 2\times i-32 \leq 25 \vee 2\times i-32 \geq 51\} \\ &= \{i \mid i \leq 28 \vee i \geq 41\} \end{aligned}$$

Based on these information, the algorithm computes followings.

Definition 4 NLIS(d, u, np) is Non-Local data use Iteration Space in which the statement of u access non-local data when it executed in the processor *np*.

Definition 5 NLD(d, u, np) is a set of Non-Local Data (generally array section) which is used in $NLIS(u)$.

Definition 6 FIS(u) is an iteration space in which u uses non-local data before other uses

In $NLID(d,u,np)$, the use u accesses non-local data which is defined in def d and is executed in processor *np*. In other word, $NLIS(d,u,np)$ is intersection of $IS(d,u)$, $LIS(u,np)$, and $NLUIS(u,np)$. $NLIS(d,u,np)$s of each use in process 1 are these :

$$\begin{aligned} NLIS(1,1,1) &= \{i \mid 6 \leq i \leq 105 \wedge 26 \leq i \leq 50 \wedge \\ &\quad (i \leq 30 \vee i \geq 56)\} \\ &= \{i \mid 26 \leq i \leq 30\} \end{aligned}$$

$$\begin{aligned} NLIS(2,1,1) &= \{i \mid 17 \leq i \leq 32 \wedge 26 \leq i \leq 50 \wedge \\ &\quad (i \leq 28 \vee i \geq 41)\} \\ &= \{i \mid 26 \leq i \leq 28\} \end{aligned}$$

$NLD(d,u,np)$ is a non-local data set which is used in $NLIS(u,d)$. To convert given iteration space to array section, function $Is2Ds()$ is used. $NLD(u,d)$s correspond to $NLIS(u,d)$s are these :

$$\begin{aligned} NLD(1,1) &= \{i \mid 21 \leq i \leq 25\} \\ NLD(2,1) &= \{20, 22, 24\} \end{aligned}$$

To reuse the non-local data, it should be received and saved at first use. So we need the execution ordering information of each use. $FIS(u,d)$ is the iteration space in which u uses a non-local data before other use. $FIS(u,d)$s of each use are like these :

$$\begin{aligned} FIS(1,1) &= \{i \mid i \leq 27\} \\ FIS(2,1) &= \{i \mid i \geq 28\} \end{aligned}$$

In the last stage of this algorithm, calculate following information.

Definition 7. FUIS(d, u, np) is non-local data First Use Iteration Space, in which the non-local data is used for the first time, so communication is inevitable.

Definition 8. RUIS(d, u, np) is non-local data ReUse Iteration Space, in which the non-local data is aleady saved in local memory, so communication is not necessary.

Definition 9. RIS(d, u, np) is non-local data Redundant usable Iteration Space. It is minimal iteration space which contains FUIS and RUIS and is represented in (lower bound, upper bound, stride).

Non-local data which is used by u in $FUIS(d,u,np)$, is first using and is used again after $FUIS(d,u,np)$. The itersaction of $NLD(d,u,np)$ and $NLD(d,u',np)$ is data which is used in u and u' and easily converted to iteration space in u. The intersaction between this iteration space and $FIS(u)$ is $FUIS(d,u,np)$. We can calculate general $FUIS(d,u,np)$ by following formula.

$$\begin{aligned} FUIS(d,u,np) &= \cup_{u'\ in\ use}(Ds2Is(NLD(d,u,np)\cap \\ &\quad NLD(d,u',np), index\ of\ u)\cap FIS(u)) \end{aligned}$$

In this formula, $Ds2Is()$ is a function which converts data to iteration space in which it is used.

To get $RUIS(d,u,np)$, we compute the data set which is used more than once, and computes the iteration space of that data set, and then subtract $FIS(u)$ from that iteration space. The general formula for $RUIS(d,u,np)$ is this.

$$\begin{aligned} RUIS(d,u,np) &= \cup_{u'\ in\ use}(Ds2Is(NLD(d,u,np)\cap \\ &\quad NLD(d,u',np), index\ of\ u) - FIS(u)) \end{aligned}$$

Loop iteration space is represented with (upper bound, lower bound, stride), but unfortunately $FUIS(d,u,np)$ and $RUIS(d,u,np)$ are not represented by the triplet. So guard must be inserted to identify $FUIS$ and $RUIS$. Evaluating guard in each iteration is heavy burden. To avoid guard, we introduce another iteration space, $RIS(d,u,np)$, which is minimal set represented with the triplet but sufficiently large to contain $FUIS(d,u,np)$ and $RUIS(d,u,np)$. $RIS(d,u,np)$ can be calculated following formula.

$$\begin{aligned} RIS(d,u,np) &= \uplus_{u'\ in\ use}(Ds2Is(NLD(d,u,np)\cap \\ &\quad NLD(d,u',np), index\ of\ u)) \end{aligned}$$

In the formular, we use new operator $\uplus$ which is minimal set represented with the triplet but sufficiently large to contain the given two iteration space. The results of this stage are these :

$$\begin{aligned} FUIS(1,1,1) &= \{27\}, \\ RUIS(1,1,1) &= \{29\}, \\ RIS(1,1,1) &= (27,29,2), \\ FUIS(1,2,1) &= \{28\}, \\ RUIS(1,2,1) &= \{27\}, \\ RIS(1,2,1) &= (27,28,1) \end{aligned}$$

Computing these information for all processors at compile time is burdensome and is obstacle to guarantee scalability. So at compile time these information is computed with processor number (np) and the process number is evaluated at run-time. Actual $RUIS(1,1,np)$ is

$$\begin{aligned} RUIS(1,1,np) = \{i| \quad & ((i \geq np \times 25 + 1) \\ \wedge \quad & (i \leq np \times 25 + 5)) \\ \wedge \quad & ((i \geq 2(np \times 25 + 1) - 27) \\ \wedge \quad & (i \leq 2(np \times 25 + 5) - 27)) \\ \wedge \quad & (i \leq 35) \\ \wedge \quad & (i\ mod\ 2 = 0) \\ \wedge \quad & (i \geq 28) \\ \wedge \quad & ((i \leq np \times 25 + 5) \\ \vee \quad & (i \geq (np+1) \times 25 + 6)) \end{aligned}$$

3.2 Code Generation

This section explains code generation method. Generated code by our method reuses the data which is defined in the same loop, and determines its iteration spaces at run time to reduce the overhead at compile time.

In data parallel language, each processor executes its own part of SPMD code. There are two way to inform each processor which is his code. One is using guard like 'if-statement' and the other is adjusting iteration space with processor number. Using guard is easy to use but burdensome because all processor should evaluate the comparison part of guard through the entire iteration space. On the other hand, adjusting iteration space is more efficient because each processor executes only his own iteration space.

Our code generation strategy is simple. It distringuishes FUIS and RUIS from original loop and then allows communication only in FUIS. The data received in FUIS is saved in local memory and reused in RUIS. The only way to distinguish FUIS and RUIS from original loop is using guard because they are not represented with a triplet, (lower bound, upper bound, stride). To reduce the overhead of guard, we use RIS. RIS contains RUIS and FUIS and is represented with the triplet. Instead of the guard, we distinguish RIS with adjusting loop iteration space and the guard is used only in RIS. The code generation method is following :

1. If the given iteration space was exceed the RIS, split the loop at upper and lower boundaries of RIS.

2. If the stride s of RIS is more than 2, then unroll the loop for s times.

3. RIS is the first statement among the unrolled statements. Use guard in RIS to distinguish FUIS and RUIS.

In the following loop,

```
    DO 100 i=1,100
       stmt
100 CONTINUE
```

if FUIS is $\{i|26,30,33,42,49\}$, RUIS is $\{i|30,34,40,45,50\}$, and RIS is (26, 50, 5), then the iteration space of given loop is exceed RIS. So the loop was splited at the 26 and 50 by first rule like this.

```
     DO 80 i=1,25
        stmt
80   CONTINUE
     DO 90 i=26,50
        stmt
90   CONTINUE
     DO 100 i=51,100
        stmt
100  CONTINUE
```

As second rule, unroll the middle loop, then

```
     DO 80 i=1,25
        stmt
80   CONTINUE
     DO 90 i=26,50,5
        stmt-1
        stmt-2
        stmt-3
        stmt-4
        stmt-5
90   CONTINUE
     DO 100 i=50,100
        stmt
100  CONTINUE
```

In this code, RIS is only stmt-1. Use guard at stmt-1 to distinguish FUIS and RUIS like this.

```
     DO 80 i=1,25
        stmt
80   CONTINUE
     DO 90 i=26,50,5
     IF (i is in FUIS) THEN
        stmt-1-1
     ELSEIF (i is in RUIS) THEN
        stmt-1-2
     ELSE
        stmt-1-3
     END
        stmt-2
        stmt-3
        stmt-4
        stmt-5
90   CONTINUE
     DO 100 i=50,100
        stmt
100  CONTINUE
```

And then non-local data is received and saved in FUIS and reused in RUIS.

As we said in section 3.1, iteration space information has processor number and the number is evaluated at run time in each processor, so the iteration space of generated code is determined at run time. Because compiler doesn't know the exact iteration space, it should gernates flexible loop of which iteration spaces are variable. If RIS is (rlb, rub, rs) and given loop is

```
     DO 100 i=lb,ub,s
        A(i) = f(B(i),...)
100  CONTINUE
```

then generated code is

```
     DO 80 i=lb,rub-s,s
        T = Receive(B(i))
        A(i) = f(T,...)
80   CONTINUE
     DO 90 i=rlb, rub, LCM(s,rs)
        IF (i is in FUIS) THEN
           TEMP(i/rs+1)=Receive(B(i))
           A(i) = f(TEMP(i/rs+1),...)
        ELSEIF (i is in RUIS) THEN
           A(i) = f(TEMP(i/rs+1),...)
        ELSE
           T = Receive(B(i))
           A(i) = f(T,...)
     END
     DO 90 j=i+s,i+LCM(s,rs)-1,s
        T = Receive(B(j))
        A(j) = f(T,...)
90   CONTINUE
     DO 100 i=rub+s,ub,s
        T = Receive(B(i))
        A(i) = f(T,...)
100  CONTINUE
```

In this program, LCM(a, b) is the least common multiple of a and b.

Until now, we propose improved non-local data reuse method. Our method reuse non-local data which is defined in the same loop by solving coherence problem between non-local data and temperarily saved data, reduce the burden of compiler and secure the scalability with flexible iteration space, AND minimize the use of guard.

4 Experimental Results

We examine Equation of State Fragment, Matrix Multiplication (MXM), and Image Filtering Program to evaluate the effect of our method. Experiment is performed in networked workstation environment by using MPI as communication primitives. We use 4 workstations, one is Sun Spark 2 and the others are Sun Spark 10. The parallel programs which are used in examination are made by hand and anyother optimization is not applied.

The HPF code of MXM is

```
      PROGRAM MXM
      REAL A(N,N), B(N,N), C(N,N)
CHPF$ PROCESSOR P(4)
CHPF$ ALIGN WITH A :: B,C
CHPF$ DISTRIBUTE A(BLOCK, *) ONTO P
      DO 100 i=1,N
      DO 100 j=1,N
      DO 100 k=1,N
         A(i,j)= A(i,j)+B(i,k)*C(k,j)
100   CONTINUE
      END
```

This program uses 4 processors and distributes arrays onto them in (BLOCK, *) type. To calculate A(i,j), *i*th raw of B and *j*th column of C are necessary. Owner of A(i,j) does not have 3/4 of *j*th column of C. It receives them from other processors. As a whole, each processor receives non-local data for $((M-1)/M) \times N^3$ times (M is number of processors). But *j*th column of C is also used for A(i+1,j), so the number of communication can be reduced to $(M-1)/M \times N^2$ and each non-local data is reused for N times. Table 1 is number of communication and number of reused non-local data in each processor.

	Number of Communication	Number of Reused non-local data
Don't reuse	$\frac{M-1}{M}N^3$	0
Reuse	$\frac{M-1}{M}N^2$	$\frac{M-1}{M}N \times (N-1)$

Table 1: Number of communication and reused non-local data in each processor

Because the array C is defined before the loop, CHAOS as well as our method reuse the non-local data. We examined three case, case 1 does not reuse, case 2 performs reusing with CHAOS's method and case 3 uses our method. In Figure 5, our method shows same execution time with CHAOS method. They take extremely small execution time compare to case 1 and the gap increases according to array size.

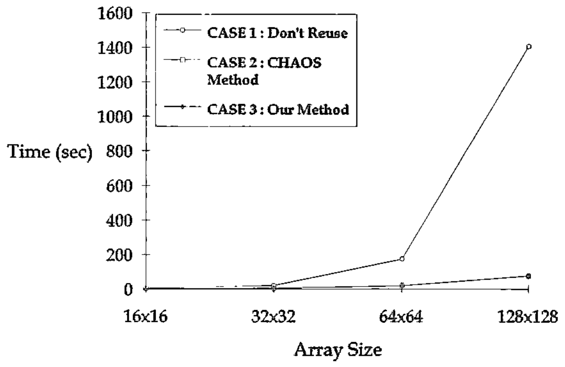

Figure 5: Execution Time Graph of MXM

The HPF code of EOSF is

```
        PROGRAM EOSF
        REAL U(N), X(N), Y(N), Z(N)
CHPF$   PROCESSOR P(4)
CHPF$   ALIGN WITH X :: U, Y, Z
CHPF$   DISTRIBUTE X(BLOCK) ONTO P
        DO 100 k=1,N
          X(k) = U(k)+R*(Z(k)+R*Y(k)) +
     &            T*(U(k+3)+R*(U(k+2)+R*U(k+1))) +
     &            T*(U(k+6)+Q*(U(k+5)+Q*U(k+4))))
100     CONTINUE
        END
```

Each array is distributed onto 4 processor and distribution type is BLOCK. Each processor performs 15 communication for 6 non-local data. So 9 communication can be reduced. The requested non-local data is defined before the loop. So CHAOS method as well as our method can reuse them. But CHAOS uses hash table to save non-local data, it takes a little overhead so, as shown in Figure 6, our method takes less time than CHAOS.

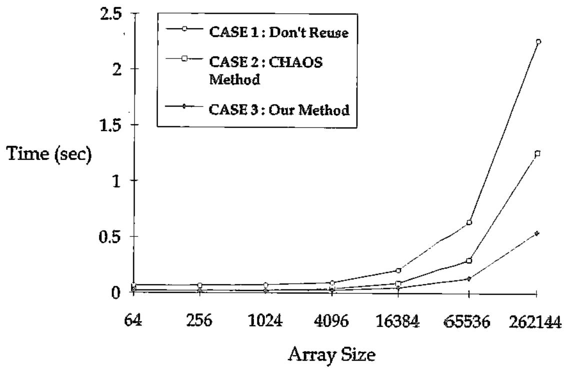

Figure 6: Excution Time Graph of EOSF

Image filtering is used to smooth out an image to hide jaggies. It replaces each pixel with the average or some function of itself and its neighbors[2]. The HPF code of image filtering is

```
        PROGRAM Filter
        REAL A(N,N)
CHPF$   PROCESSOR P(4)
CHPF$   ALIGN WITH A :: B,C
CHPF$   DISTRIBUTE A(BLOCK, *) ONTO P
        DO 100 i=1,N
        DO 100 j=1,N
          A(i,j) =
            0.08*( A(i-1,j-1)+A(i-1,j)+A(i-1,j+1)+
     &             A(i ,j-1)+ A(i ,j+1)+
     &             A(i+1,j-1)+A(i+1,j)+A(i+1,j+1))+
     &      0.36*  A(i,j)
100     CONTINUE
        END
```

As MXM, array A is saved for 4 processor and distribution type is (BLOCK,*). To get A(i,j), 9 elements around A(i,j) include itself are necessary. So at each boundary, there needs 3 non-local data as Figure 7(a). But one of them, gray one in Figure 7(b), is aleady stored in local memory when previouse line was calculated. If it was reused, 1/3 communications are reduced. But CHAOS cannot reuse them because they are defined in the same loop. Figure 8 shows CHAOS is similar to case 1. Our method reuses them, so the execution time of our method is less than any other methods.

5 Conclusion

Many data parallel language do not reuse non-local data. Only CHAOS and message coalescing attempt

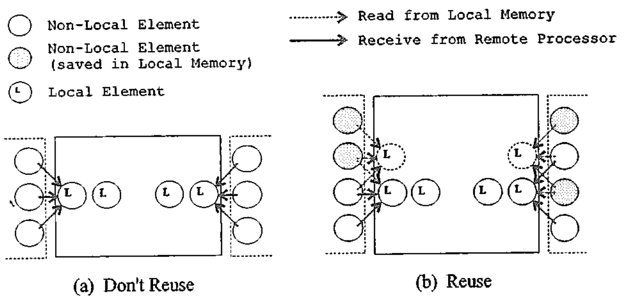

Figure 7: Non-local data access Patterns of Image Filtering Program

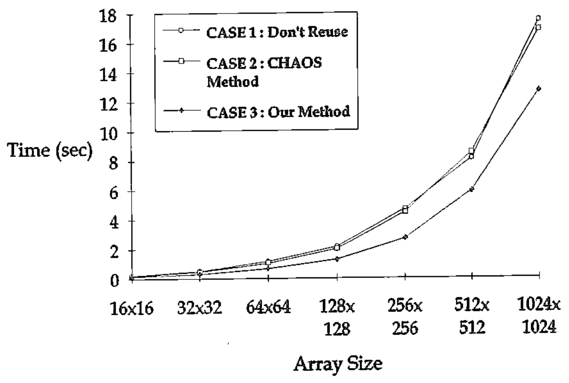

Figure 8: Execution Times of Image Filtering Program

to reuse them. However they cannot reuse the data which is defined and used in the same loop. We propose the method to reuse such data. Our method is based on flow data dependence. We found that if there are more than two flow data dependences which have same definition, the uses except the first use can reuse the data which is received in first use. And we made an experiment to show the effectiveness of our method.

However our method still has several problems especially in complex loop. First, if there are many definitions and uses in the loop, the generated code will be very complex. Second, if there are d definitions and $f(d)$ uses in each d, the complexity of our method is $O(d \times f(d)!)$.

As a future work, we plan to apply this method to the optimizing process of our under constructing HPF compiler, PPTran [7].

References

[1] Brian, J. N. Wylie, Michael G. Norman, Lyndon J. Clarke, "High Performance Fortran: A Perspective," University of Edinburgh, May 1992.

[2] J. D. Foley, A. V. Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics - principles and practice*, Addison Wesley, 1992.

[3] Wayne Kelly, William Pugh, *A framework for unifying reordering transformations*, Technical report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[4] J. Saltz and et. al. *A mannual for the CHAOS runtime library*, Technical report, University of Maryland, 1993.

[5] Chau-Wen Tseng, *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*, Ph.D thesis, Dept. of Computer Science, Rice University. January 1993.

[6] High Performance Fortran Forum, *High Performance Fortran Language Speficication, Version 1.1*, Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, Tex. 1994.

[7] Taegeun Kim, Kyeongdeok Moon, Jungkwon Kim, Yearback Yoo, Myongsoon Park, Sungsoon Park, "A Parallel Program Translator from HPF to PVM," *Electronic Proceeding of HPC-ASIA'95*, 1995.