Optimization Techniques for Parallel Codes of Irregular Scientific Computations *

Minyi Guo
Dept. of Computer Software
The University of Aizu
Aizu-Wakamatsu City
Fukushima, 965-8580, Japan
minyi@u-aizu.ac.jp

Weng-Long Chang
Dept. of Info. Management
Southern Taiwan University
of Technology
Tainan, Taiwan, 710, R.O.C.
changwl@csie.ncku.edu.tw

Yi Pan
Dept. of Computer Science
Georgia State University
University Plaza, Atlanta
GA 30303, USA
pan@cs.gsu.edu

Abstract

In this paper, we propose a communication cost reduction computes rule for irregular loop partitioning, called least communication computes rule. For an irregular loop with nonlinear array subscripts, the loop is transformed to a normalized single loop, then we partition the loop iterations to processors on which the minimal communication cost is ensured when executing those iterations. We also give some interprocedural optimization techniques for communication preprocessing when the irregular code has the procedure call. The experimental results show that, in most cases, our approaches achieved better performance than other loop partitioning rules.

Keywords *Parallelizing compilers, Irregular scientific application, Communication optimization, Loop transformation, Loop partitioning, Interprocedural optimization.*

1 Introduction

Parallelizing compilers are necessary to allow programs written in standard sequential languages to run ef ciently on parallel machines. In order to achieve good performance, these compilers must be able to ef ciently generate communication sets for nested loops. Parallelizing compilers that generate code for each processor have to compute the sequence of local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. The distribution of computation in most compilers follows the *owner-computes rule*. That is, a processor performs only those computations (or assignments) for which it owns the left hand side variable. Access to non-local right hand side variables is achieved by inserting sends and receives.

*This research was supported in part by the Grant-in-Aid for Scienti c Research (C)(2) 14580386 and The Japanese Okawa Foundation for Information and Telecommunications under Grant Program 01-12.

Communication overhead in uences the performance of parallel programs signi cantly. According to Hockney's representation, communication overhead can be measured by a linear function of the message length m — $T_{comm} = T_s + mT_d$ — where T_s is the start-up time and T_d is the per-byte messaging time. To achieve good performance, we must optimize communication in following three aspects:

- to exploit local computation as much as possible;
- to vectorize and aggregate communication in order to reduce the number of communications; and
- to reduce the message length in a communication step.

As the scientists attempt to model and compute more complicated problems, they have to envisage to develop ef cient parallel code for sparse and unstructured problems in which array accesses are made through a level of indirection or nonlinear array subscript expressions. This means that the data arrays are indexed either through the values in other arrays, which are called *indirection arrays/index arrays*, or through non-af ne subscripts. The use of indirect/nonlinear indexing causes the data access patterns, i.e. the indices of the data arrays being accessed, to be highly irregular. Such a problem is called *irregular problem*, in which the dependency structure is determined by variable causes known only at runtime. Irregular applications are found in unstructured computational uid dynamic (CFD) solvers, molecular dynamics codes, diagonal or polynomial preconditioned iterative linear solvers, and n-body solvers.

Researchers have demonstrated that the performance of irregular parallel code can be improved by applying a combination of computation and data layout transformations. Some researches focus on providing primitives and libraries for runtime support [2, 10, 3], some provide language support such as add irregular facilities to HPF or Fortran 90 [13, 15], and some works attempt to utilize caches and locality ef ciently [4].

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

Hwang et al. [10] presented a library called CHAOS, which helps user implement irregular programs on distributed memory machines. The CHAOS library provides ef cient runtime primitives for distributing data and computation over processors; it supports index translation mechanisms and provides users high-level mechanisms for optimizing communication. In particular, it provides support for parallelization of adaptive irregular programs where indirection arrays are modi ed during the course of computation. The CHAOS library is divided into six phases. They are Data Partitioning, Data Remapping, Iteration Partitioning, Iteration Remapping, Inspector, and Executor phase. The rst four phases concern mapping data and computations onto processors. The next two steps concern analyzing data access patterns in loops and generating optimized communication calls. The same working group as the above, Ponnusamy et al. extended the CHAOS runtime procedures which are used by a prototype Fortran 90D compiler to make it possible to emulate irregular distribution in HPF by reordering elements of data arrays and renumbering indirection arrays [13]. Also, in their paper [3], Das. et al. discussed some primitives to support communication optimization of irregular computations on distributed memory architectures. These primitives coordinate inter-processor data movement, manage the storage of, and access to, copies of off-processor data, minimize inter-processor communication requirements and support a shared name space.

In this paper, we propose some optimization techniques to minimize the communication cost in pre-processing for compiling irregular scienti c codes. We rst partition irregular loops using a communication cost reduction computes rule, called least communication computes rule. According to these information we partition the loop iteration to a processor on which the minimal communication is ensured when executing that iteration. Then, for irregular nested loops with nonlinear subscript references, we transform the loops to single loops using loop coalescing and symbolic analysis. After transformed to single loops, the loops can be treated as the loops with indirection array. Additionally, using inter-procedural partial redundancy elimination algorithm, we optimize the preprocessing routine and collective communication if the irregular codes include interprocedure calls.

2 Reducing Communication cost for Indirection Array Loop Partitioning

As mentioned above, there are two kinds of irregular loop – data array are indexed through indirection arrays or indexed through nonlinear subscript expressions – we called indirection array loop or nonlinear loop respectively. In this section, we propose a communication cost reduction technique for indirection array loop partitioning. In the following discussion, we assume that the indirection array loop body has only loop-independent dependence, but no loop-carried dependence (it is very dif cult to test irregular loop-carried dependence since dependence testing methods for linear subscripts are completely disabled), because most of practical irregular scienti c applications have this kind of loops.

2.1 Motivation Examples

Consider the irregular loop below, which is a simpli ed version extracted from ZEUS-2D code [11]:

Example 1

```
DO 10 t = 1, time_step
C  Outer loop takes the execution times
C  of irregular loop
    DO 100 i = 1, N
S1: X(j2(i)) = X(j1(i))+Y(j3(i))
S2: X(j4(i)) = X(j3(i))+Y(j1(i))+Z(j3(i))
S3: Y(j1(i)) = Y(j1(i))+Z(j3(i))-X(j1(i))
S4: Y(j4(i)) = Y(j3(i))-X(j3(i))
100 CONTINUE
    ......
10  CONTINUE
```

Generally, in distributed memory compilation, loop iterations are partitioned to processors according to the owner computes rule [1]. This rule speci es that, on a single-statement loop, each iteration will be executed by the processor which owns the left hand side array reference of the assignment for that iteration.

For the loop in Example 1, if owner computes rule is applied, the rst step is to distribute the loop into three individual loops each of which includes the statement `S1`, `S2`, and `S3`, respectively. Without loss of generality, suppose that the loop would be executed on 4 processors in parallel. If the array element `Y(i)` is aligned with `X(i)` in the initial distribution, clearly `Y(j1(i))` is also distributed onto the same processor with `X(j1(i))`. So we can assume that P_0, P_1, P_2, and P_3 own `[X(j1(i)), Y(j1(i))]`, `[X(j2(i))]`, `[X(j3(i)),Y(j3(i)), Z(j3(i))]`, and `[X(j4(i)),Y(j4(i))]`, respectively, for iteration i. Then the iteration i of executing `S1, S2, S3`, and `S4` would be partitioned to processor P_1, P_3, P_0, and P_3, respectively. Thus if any references to array elements on the right-hand side is not owned by the processor executing

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

Statement	Owner	array elements required communication
S1:	P_1,	X(j1(i)) $(P_0 \longrightarrow P_1)$, Y(j3(i)) $(P_2 \longrightarrow P_1)$
S2:	P_3,	X(j3(i)), Z(j3(i)) $(P_2 \longrightarrow P_3)$, Y(j1(i)) $(P_0 \longrightarrow P_3)$
S3:	P_0,	Z(j3(i)) $(P_2 \longrightarrow P_0)$
S3:	P_3,	X(j3(i)), Y(j3(i)) $(P_2 \longrightarrow P_3)$

Table 1. The owner of executing assignments and required communications for the example loop.

the statement (say, an off-processor reference), the array data on the right-hand side would have to be communicated to the owner. Table 1 shows the owner of executing assignments and required communications for the example loop.

However, owner computes rule is often not best suited for irregular codes. This is because use of indirection in accessing left hand side array makes it dif cult to partition the loop iterations according to the owner computers rule. Therefore, in CHAOS library, Ponnusamy et al. [13, 14] proposed a heuristic method for irregular loop partitioning called *almost owner computes rule*, in which an iteration is executed on the processor that is the owner of the largest number of distributed array references in the iteration.

According to almost owner computes rule, this loop iteration would be partitioned to P_2 because it has the majority number of data elements. The communication would be as the follows, where $tmp\_$ means the values obtained at the loop executing owner but need to send back to the array element owners:

- Import communication before the loop iteration is executed:
  1. X(j1(i)), Y(j1(i)): $P_0 \longrightarrow P_2$
- Export communication after the loop iteration is executed:
  1. tmp_Yj1: $P_2 \longrightarrow P_0$
  2. tmp_Xj2: $P_2 \longrightarrow P_1$
  3. tmp_Xj4, tmp_Yj4: $P_2 \longrightarrow P_3$

Obviously the communication cost is reduced as compared to the owner computes rule. Some HPF compilers employ this scheme by using EXECUTE-ON-HOME clause [15]. However, when we parallelize a uid dynamics solver ZEUS-2D code by using almost owner computes rule, we nd that the almost owner computes rule is not optimal manner in minimizing communication cost — either communication steps or elements to be communicated. Another drawback is that it is not straightforward to choose optimal owner if several processors own the same number of array references.

2.2 Efficient Loop Iteration Partitioning

If we consider communication overhead when the iteration is partitioned to P_0, we can obtain the communication pattern as follows:

- Import communication before the loop iteration is executed:
  1. X(j3(i)), Y(j3(i)), Z(j3(i)): $P_2 \longrightarrow P_0$
- Export communication after the loop iteration is executed:
  1. tmp_Xj2: $P_0 \longrightarrow P_1$
  2. tmp_Xj4, tmp_Yj4: $P_0 \longrightarrow P_3$

Although the number of elements to be communicated is 6, same as the almost owner computes rule, the communication steps are reduced (three times). This improvement is important when the outer sequential time step-loop is large.

Based on the above observation, we propose a more ef cient computes rule for irregular loop partition [8]. This approach partitions iterations on a particular processor such that executing the iteration on that processor ensures

- the communication steps is minimum, and
- the total number of data to be communicated is minimum

In our approach, neither owner computes rule nor almost owner computes rule is used in parallel execution of a loop iteration for irregular computation. A communication cost reduction computes rule, called least communication computes rule, is proposed. For a given irregular loop, we rst investigate for all processors $P_k, 0 \leq k \leq m$ (m is the number of processors) in which two sets $FanIn(P_k)$ and $FanOut(P_k)$ for each processor P_k are de ned. $FanIn(P_k)$ is a set of processors which have to send data to processor P_k before the iteration is executed, and $FanOut(P_k)$ is a set of processors which have to send data from processor P_k after the iteration is executed. According to these knowledge we partition the loop iteration to a processor on which the minimal communication is ensured when executing that iteration. Then, after all iterations are partitioned into various processors. Please refer to [8] for details.

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

23 Node Program

After least communication loop partitioning analysis, we can develop a node program which has three parts: pre-execution import communication (gathering phase), irregular loop execution (executing phase), and post-execution export communication (scattering phase). In the node program, $D_k(j)$ ($D'_k(j)$) is the all data which need send to P_j from P_k (current executing processor), before (after) loop execution. $i\$local$ marks as local loop index. α_k is the number of iterations partitioned onto P_k.

P_k's node program:

// pre-communicating required elements with other processors.
$\forall P_j \in FanIn(P_k)$
  receive data from P_j;
for $j = 0, m-1, \neq k$
  if $P_k \in FanIn(P_j)$ **then**
    buffering data from $D_k(j) = D_k(i_1, j) \cup \cdots \cup D_k(i_{\alpha_k}, j)$;
    send to P_j;
  end if
end for
// executing the local iterations
for $i\$local = 1, \alpha_k$
  $S_1(i\$local); S_2(i\$local); \ldots; S_n(i\$local)$;
end for
// post communicating changed remote elements with other processors.
for $\forall P_j \in FanOut(P_k)$
  buffering data from $D'_j(k) = D'_j(i_1, k) \cup \cdots \cup D_j(i_{\alpha_k}, k)$;
  send changed data to P_j;
end for
for $j = 0, m-1, \neq k$
  if $P_k \in FanOut(P_j)$ **then**
    receive changed data from P_j;
  end if
end for

3 Communication Optimization for Nonlinear Array Loops

Given a perfectly nested irregular loop with nonlinear array subscripts as shown in the following.

```
      DO i1 = X1,Y1,Z1
      ......
      DO in = Xn,Yn,Zn
S:    A[f(i1,i2,...,in)] = F(B[g(i1,i2,...,in)])
      CONTINUE
      ......
      CONTINUE
```

For the sake of simplicity, we will assume that the referenced array A and B have only one dimension. The array access functions (f and g), the loop's lower and upper bounds (X_i, Y_i), and stride (Z_i) may be arbitrary symbolic expressions made up of loop-invariant variables and loop indices of enclosing loops. We will also assume that all loop strides are positive. It is not dif cult to extend our method to handle imperfectly nested loops, negative strides, multidimensional arrays, and loop-variant variables. Furthermore, let the arrays A and B be initially distributed as BLOCK across P processors.

In order to parallelize and partition this nested loop, it has to be transformed to single loop using loop coalescing. The loop transformation can be performed as the following steps:

1. Loop normalization:
   For $j = 1, n$ do
   $DO\ i = X_j, Y_j, Z_j \Longrightarrow DO\ i = 1, YY_j$
   where $YY_j = (Y_j - X_j + Z_j)/Z_j$.

2. Transforming to a single loop (loop coalescing):

```
      DO i1 = 1, YY1
      ......
      DO in = 1, YYn
S:    A[f(i'1,i'2,...,i'n)] = &
          F(B[g(i'1,i'2,...,i'n)])
      CONTINUE
      ......
      CONTINUE

        ⇓

      DO ii = 1, YY
      i'1 = ((ii-1)/(YY2*...*YYn))  &
            *(YY2*...*YYn) + 1
      i'2 = ((ii-1)/(YY3*...*YYn))  &
            *(YY3*...*YYn) + 1
      ......
      i'n = ((ii-1) mod YYn) + 1
S:    A[f(i'1,i'2,...,i'n)] =  &
          F(B[g(i'1,i'2,...,i'n)])
      CONTINUE
```

   where YY = YY1*YY2*...*YYn.

3. Using Algorithm 1, 2, 3, and 4 in [8] to compute the least communication owner for each iteration.

4. Partitioning the loop according to least communication computes rule.

However, The above steps can only be applied if the bounds of loops are loop invariant. For the bounds are expression including loop variables or indices, such as the following loop,

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

```
      DO i1 = 1, N
      DO i2 = 1, i1
      IA = i1*(i1-1)/2 + i2
      IB = i2*(i2-1)/2 + i1
S:    A[IA] = F(B[IB])
      CONTINUE
      CONTINUE
```

the above steps cannot be used, because after loop coalescing there are loop variables in loop bounds. Here, we propose a symbolic sum computation algorithm for determining the constant loop bounds.

Suppose that the nested loop has the loop bounds as

$DO\ i_1 = 1, N$

$DO\ i_2 = L_2(i_1), U_2(i_1)$

$\cdots$

$DO\ i_n = L_n(i_1, \ldots, i_{n-1}), U_n(i_1, \ldots, i_{n-1})$

We can count the number of integer solutions between the bounds by using the following formulas.

$$C_2 = \sum_{i_1=1}^{N} (U_2(i_1) - L_2(i_1) + 1)$$

$$C_3 = \sum_{i_1=1}^{N} \sum_{i_2=L_2(i_1)}^{U_2(i_1)} (U_3(i_1, i_2) - L_3(i_1, i_2) + 1)$$

$\cdots$

$$C_n = \sum_{i_1=1}^{N} \cdots \sum_{i_{n-1}=L_{n-1}(i_1,\ldots,i_{n-1})}^{U_{n-1}(i_1,\ldots,i_{n-1})} (U_n(i_1, i_2, \ldots, i_{n-1}) - L_n(i_1, i_2, \ldots, i_{n-1}) + 1)$$

Thus, the bounds of single loop is from 1 to $N * C_2 * \cdots * C_n$.

Coming back to the above example, because total number of iterations of the inner loop (i_2) is $\sum_{i_1=1}^{N} i_1 = N * (N+1)/2$, after transformation, the upper bound of the single loop is $N^2 * (N+1)/2$.

4 Inter-procedural Communication Optimization for Irregular Loops

In some irregular scienti c codes, an important optimization required is communication preprocessing among procedure calls. In this section, we extend a classical data ow optimization technique – Partial Redundancy Elimination – to an Interprocedural Partial Redundancy Elimination as a basis for performing interprocedural communication optimization [2]. Partial Redundancy Elimination encompasses traditional optimizations like loop invariant code motion and redundant computation elimination.

Consider the example program presented in the left side of Figure 1. Initial intraprocedural analysis in Section 2 (see [8] also) inserts pre-communicating call (including one buffering and one gathering routine) and post-communicating (buffering and scattering routine) call for

```
PROGRAM
REAL A(n), B(n), C(n), D(n)
INTEGER IA(n), IB(n), IC(n)
……
DO 10 I = 1, 20
CALL SUB1(A,B,C,IA,IC)
CALL SUB2(A, C, IA, IC)
……
10 CONTINUE
END

SUBROUTINE SUB1(U,V,W,X,Y)
……
DO 100 I = 1, n
W(Y(I)) = W(Y(I)) + U(X(I))
100 CONTINUE
……
END

SUBROUTINE SUB2(A, C, IX, IY)
……
DO 200 I = 1, n
C(IY(I)) = C(IY(I)) + A(IX(I))
200 CONTINUE
……
END
```

```
(NODE) PROGRAM
(same as the left)

SUBROUTINE SUB1(U,V,W,X,Y)
recv(&U, &W, Pany)
buffering (X,Y)
send(&U, Pany)
……
DO 100 I$local = 1, n$local
W(Y(I$local)) = W(Y(I$local)) +U(X(I$local))
100 CONTINUE
……
Buffering(Y)
send(&W, Pany)
recv(&W, Pany)
END

SUBROUTINE SUB2(A, C, IX, IY)
recv(&A, &C, Pany)
buffering (IX,IY)
send(&A, Pany)
……
DO 200 I$local = 1, n$local
C(IY(I$local)) = C(IY(I$local)) +A(IX(I$local))
200 CONTINUE
……
Buffering(IY)
send(&C, Pany)
recv(&C, Pany)
END
```

Figure 1. Original code and its intraprocedural compiled node program.

```
PROGRAM
REAL A(n), B(n), C(n), D(n)
INTEGER IA(n), IB(n), IC(n)
……
recv(&A, &C, Pany)
buffering (IA,IC)
DO 10 I = 1, 20
CALL SUB1(A,B,C,IA,IC)
CALL SUB2(A, C, IA, IC)
……
10 CONTINUE
END

SUBROUTINE SUB1(U,V,W,X,Y)
send(&U, Pany)
……
DO 100 I$local = 1, n$local
W(Y(I$local)) = W(Y(I$local)) + U(X(I$local))
100 CONTINUE
……
END

SUBROUTINE SUB2(A, C, IX, IY)
……
DO 200 I$local = 1, n$local
C(IY(I$local)) = C(IY(I$local)) + A(IX(I$local))
200 CONTINUE
……
send(&C, Pany)
recv(&C, Pany)
END
```

Figure 2. After interprocedural Optimized node program.

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

```
      PROGRAM
      REAL A(n), B(n), C(n), D(n)
      INTEGER IA(n), IB(n), IC(n)
      ……
      DO 10 I = 1, 20
      CALL SUB1(A, B, IA)
      CALL SUB2(A, C, IB)
      ……
10    CONTINUE
      END

      SUBROUTINE SUB1(U,V, X)
      ……
      DO 100 I = 1, n
      V(I) = … U(X(I)) …
100   CONTINUE
      ……
      END

      SUBROUTINE SUB2(A, C, IB)
      ……
      DO 200 I = 1, n
      C(I) = C(I) + A(IB(I))
200   CONTINUE
      ……
      END
```

(a)

```
      PROGRAM
      REAL A(n), B(n), C(n), D(n)
      INTEGER IA(n), IB(n), IC(n)
      ……
      DO 10 I = 1, 20
      CALL SUB1(A, B, IA)
      CALL SUB2(A, C, IA)
      ……
10    CONTINUE
      END

      SUBROUTINE SUB1(U,V, X)
      ……
      DO 100 I = 1, m
C     m < n
      V(I) = … U(X(I)) …
100   CONTINUE
      ……
      END

      SUBROUTINE SUB2(A, C, IA)
      ……
      DO 200 I = 1, n
      C(I) = C(I) + A(IA(I))
200   CONTINUE
      ……
      END
```

(b)

Figure 3. Common and incremental buffering original code.

```
      PROGRAM
      REAL A(n), B(n), C(n), D(n)
      INTEGER IA(n), IB(n), IC(n)
      ……
      recv(&A, Pany)
      common = buffering (IA)
      incml = buffering(IA, IB)
      DO 10 I = 1, 20
      CALL SUB1(A, B, IA)
      CALL SUB2(A, C, IB)
      ……
10    CONTINUE
      END

      SUBROUTINE SUB1(U,V, X)
      ……
      send(&U, common, Pany)
      DO 100 I local = 1, n$local
      V(I$local) = … U(X(I$local)) …
100   CONTINUE
      ……
      END

      SUBROUTINE SUB2(A, C, IB)
      ……
      send(&A, incml, Pany)
      DO 200 I$local = 1, n$local
      C(I$local) = C(I$local) + A(IB(I$local))
200   CONTINUE
      ……
      END
```

(a)

```
      PROGRAM
      REAL A(n), B(n), C(n), D(n)
      INTEGER IA(n), IB(n), IC(n)
      ……
      recv(&A, Pany)
      common = buffering (IA(I), I < m)
      incml = buffering(IA(I), m<=I<n)
      DO 10 I = 1, 20
      CALL SUB1(A, B, IA)
      CALL SUB2(A, C, IA)
      ……
10    CONTINUE
      END

      SUBROUTINE SUB1(U,V, X)
      ……
      send(&U, common, Pany)
      DO 100 I$local = 1, m$local
C     m < n
      V(I$local) = … U(X(I$local)) …
100   CONTINUE
      ……
      END

      SUBROUTINE SUB2(A, C, IA)
      ……
      send(&A, incml, Pany)
      DO 200 I$local = 1, n$local
      C(I$local) = C(I$local) + A(IA(I$local))
200   CONTINUE
      ……
      END
```

(b)

Figure 4. Interprocedural Optimized node code using common and incremental buffering.

each of the two data parallel loops in two subroutines (the right side of Fig. 1). After interprocedural analysis, the compiled node program is shown in Figure 2. Here, since the array IA and IC are never modi ed inside the time step loop in the main procedure, the schedules $buffering(X, Y)$ and $buffering(IX, IY)$ are loop invariants and can be hoisted outside the loop.

Further, it can be deduced that the computation of $buffering(X, Y)$ and $buffering(IX, IY)$ are equivalent. So only $buffering(X, Y)$ needs to be computed and the gather routine in $SUB2$ can use $buffering(X, Y)$ instead of $buffering(IX, IY)$. The gather for array IA in subroutine $SUB2$ is redundant because of the gather of array A in $SUB1$. Thus, we hoist the common partial subexpression as $buffering(IA, IC)$.

Similarly, We also can apply Interprocedral Partial Redundancy Elimination analysis to post-communicating call. Further, $buffering(Y)$ is included in $buffering(IA, IC)$, and it can be eliminated. The result is shown in Figure 2.

In the above example, data arrays and index arrays are the same in loop bodies of two subroutines. While some communication statement may not be redundant, there may be some other communication statement, which may be gathering at least a subset of the values gathered in this statement.

Consider the program shown in Figure 3(a). The same data array A is accessed using an indirection array IA in $SUB1$ and using another indirection array IB in $SUB2$. Further, none of the indirection arrays or the data array A is modi ed between ow control from rst loop to the second loop. There will be at least some overlap between required communication data elements made in these two loops. Another case is that the data array and indirection array is the same but the loop bound is different(See Figure 3(b)). In this case, the rst loop can be viewed as a part of the second loop.

We divide two kinds of communication routines for such situations. A common communication routine takes more than one indirection array, or takes common part of two indirection arrays. In the example in Figure 3(a), a common communication routine will take in arrays IA and IB producing a single buffering. Incremental preprocessing routine will take in indirection array IA and IB, and will determine the off-processor references made uniquely through indirection array IB and not through indirection array IA. While executing the second loop, communication using an incremental schedule can be done, to gather only the data elements which were not gathered during the rst loop.

In Figure 4(a) and (b), we show this optimization for Figure 3(a) and (b) respectively, where $send(\&U, common, Pany)$ represents data elements sending according to $common$ buffering part while

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

```
mrsij0 = 0
DO mrs = 0, (N*N+N)/2-1
    mrsij = mrsij0
    DO mi = 0, N-1
       DO mj = 0, mi-1
S1:       mrsij = mrsij + 1
S2:       xrsij(mrsij) = xij(mj)
       ENDDO
    ENDDO
    mrsij0 = mrsij0+(N*N+N)/2
ENDDO
```

⇒

```
DO mrs = 0, (N*N+N)/2-1
   DO mi = 0, N-1
      DO mj = 0, mi-1
S1:      mrsij = (mi*mi+mi+     &
           mrs*(N*N+N))/2+mj+1
S2:      xrsij(mrsij) = xij(mj)
      ENDDO
   ENDDO
ENDDO
```

Figure 5. Simpli ed version of loop nest OLDA from TRFD

$send(\&A, incml, Pany)$ represents sending according to $incremental$ buffering.

5 Experiments and Performance Results

We now present experimental results to show the ef cacy of the methods presented so far. We measure the difference made by using owner computes rule, and our least communication computes rule in an experimental program. Another experiment is examined for the difference of with or without interprocedural optimization. All the experiments are examined on a 24 node SGI Origin 2000 parallel computer or a 32-node CM-5 parallel computer. The node programs are written in Fortran, using MPI communication library and system call `gettimeofday()` for measuring execution time.

We select a subroutine OLDA from the code TRFD, appearing in Perfect benchmark [12]. A simpli ed version of this loop nest is shown in the left side of Figure 5. After using induction variable substitution to replace the induction variable $mrsij$ at statement S_1, the optimized version is shown in the right side of Figure 5. There is a nonlinear array subscript for $xrsij$ at S_2. To parallelize this loop nest, the communication set generation and loop partitioning optimization must be used.

We assume that initial distribution schemes are BLOCK both for arrays $xrsij$ and xij. Figure 6 shows the total loop execution times on CM-5 when $N = 16$ (with global array size 18632) $owner\ compute$ and $least\ comm$ respectively represent that we use owner computes rule and our least communication computes rule to partition the loop. We observe that as the number of nodes increases, the execution time is not so much improved because each processor has to communicate with increasing number of nodes. The g- ure shows that our method gets good performance in most cases.

Our another experiment for irregular loop with interprocdeural optimization selects an irregular kernel of uid dynamics code, ZEUS-2D for our study. ZEUS-2D is a computational uid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for astrophysical radiation magnetohydro dynamics problems[11]. ZEUS-2D solves the equations of ideal (non-resistive), non-relativistic, hydrodynamics, including radiation transport, (frozen-in) magnetic elds, rotation, and self-gravity. Boundary conditions may be speci ed as re ecting, periodic, in ow, or out ow. The kernel irregular subroutine `emfs` includes some loops in which the loop body invokes four subroutines `X1INTFC, X1INTZC, X2INTFC`, and `X2INTZC`, each of which includes irregular loops similar with the motivation example in Section 2. We specify the geometry as Cartesian XY, the grid as uniformly spaced zones 800 by 2, and extend the irregular loop iterations to 1000.

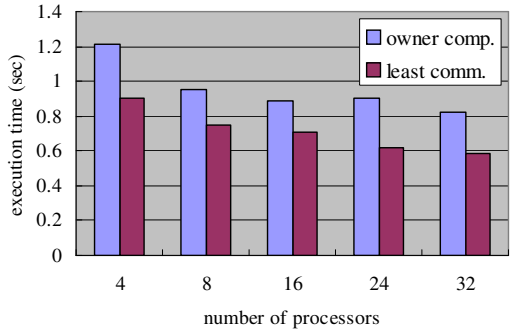

Figure 6. Execution time of OLDA program on CM-5 for owner computes rule and transformed single loop using least communication computes rule for loop partitioning.

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

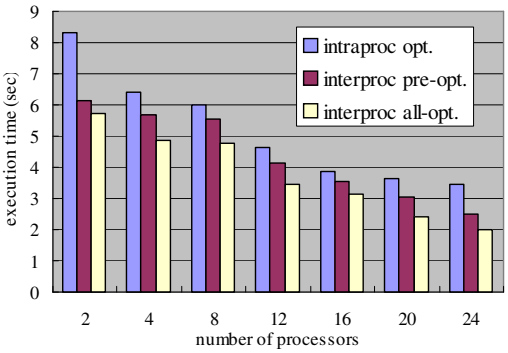

Figure 7. Effect of interprocedural communication optimization in executing ZEUS-2D irregular loops (Execution time) on SGI Origin 2000.

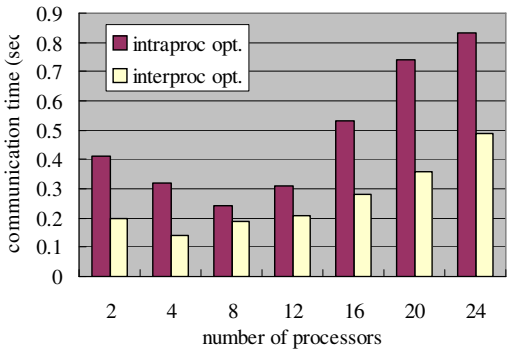

Figure 8. Communication time for intraprocedural and interprocedural optimization in executing ZEUS-2D irregular loops on SGI Origin 2000.

In Figure 7, we show the performance difference of `emfc` obtained by using three kinds of the version: only intraprocedural optimization, with interprocedural pre-communicating optimization, and with all interprocedural communication optimization. Performance of the different versions of the code is measured on SGI Origin 2000 from 2 to 24 processors. The curves marked with *intraproc opt., interproc pre-opt.*, and *interproc all-opt.*. are the versions of the code which the communication statements using intraprocedural optimization, interprocedural pre-communicating optimization and all interprocedural communication optimization. The gure shows that interprocedural communication optimization gets good performance in all cases. When the same data is distributed over a larger number of processors, the communication time becomes a signi cant part of the total execution time and the communication optimization makes signi cant difference in the overall performance of the program.

In Figure 8, we further study the impact of different versions on communication statements. Only the communication time is shown for the various versions of the code. Communication optimizations in our method include gather and scatter before and after loop execution, and common-incremental buffering. When the number of processors is large, our method can lead to substantial improvement in the performance of the code, because the communication time in uences signi cantly total performance of parallel program.

6 Conclusions

The ef ciency of loop partitioning in uences performance of parallel program considerably. For automatically parallelizing irregular scienti c codes, the owner computes rule is not suitable for partitioning irregular loops. In this paper, we have presented an ef cient loop partitioning approach to reduce communication cost for a kind of irregular loop with nonlinear array subscripts. In our approach, runtime preprocessing is used to determine the communication required between the processors. We have developed the algorithms for performing these communication optimization. We have also presented how interprocedural communication optimization can be achieved. We have done a preliminary implementation of the schemes presented in this paper. The experimental results demonstrate ef cacy of our schemes.

References

[1] R. Allen and K. Kennedy. Optimizing compilers for Modern Architectures. Morgan Kaufmann Publishers, 2001.

[2] G. Agrawal and J. Saltz. Interprocedural compilation of Irregular Applilcations for Distributed memory machines. *Language and Compilers for Parallel Computing*, pp. 1-16, August 1994.

[3] R. Das, M. Uysal, J. Saltz, and Y-S. Hwang. Communication optimizations for irregular scienti c computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[4] C. Ding and K. Kennedy. Improcing cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May, 1999.

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 IEEE

[5] M. Guo, I, Nakata, and Y. Yamashita. Contention-free communication scheduling for array redistribution. *Parallel Computing*, 26(2000), pp. 1325-1343, 2000.

[6] M. Guo and I. Nakata. A framework for ef cient array redistribution on distributed memory machines. *The Journal of Supercomputing*, Vol. 20, No. 3, pp. 253-265, 2001.

[7] M. Guo, Y. Pan, and C. Liu. Symbolic Communication Set generation for irregular parallel applications. To appear in *The Journal of Supercomputing*, 2002.

[8] M. Guo, Z. Liu, C. Liu, L. Li. Reducing Communication cost for Parallelizing Irregular Scienti c Codes. In *Proceedings of The 6th International Conference on Applied Parallel Computing*, Finland, June 2002.

[9] E. Gutierrez, R. Asenjo, O. Plata, and E.L. Zapata. Automatic parallelization of irregular applications. *Parallel Computing*, 26(2000), pp. 1709-1738, 2000.

[10] Y.-S Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software-Practivce and Experience*, Vol.25(6), pp. 597-621,1995.

[11] J.M. Stone and M. Norman. ZEUS-2D: A radiation magnetohydrodynamics code for astrophysical ows in two space dimensions: The hydrodynamic algorithms and tests. *Astrophysical Journal Supplement Series*, Vol. 80, pp. 753-790, 1992.

[12] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, pp. 3(3):5-40, 1989.

[13] R. Ponnusamy, Y-S. Hwang, R. Das, J. Saltz, A. Choudhary, G. Fox. Supporting irregular distributions in Fortran D/HPF compilers. Technical report CS-TR-3268, University of Maryland, Department of Computer Science, 1994

[14] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang, and G. Fox. Runtime support and compilation methods for user-speci ed data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8), pp. 815-831, 1995.

[15] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H.P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*. 8(10), Oct. 1997. pp. 1068 -1083.

Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'02)
1530-2016/02 $17.00 © 2002 **IEEE**