Vol. 44 No. SIG 1(HPS 6) IPSJ Transactions on High Performance Computing Systems Jan. 2003

Regular Paper

Optimization Techniques for Parallel Codes of Irregular Scientific Computations

Minyi Guo,[†] Weng-Long Chang[††] and Yi Pan[†††]

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.

1. Introduction

Parallelizing compilers are necessary to allow programs written in standard sequential languages to run efficiently on parallel machines. In order to achieve optimal performance, these compilers must be able to efficiently 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 as well as 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.

Communication overhead influences the performance of parallel programs significantly. According to Hockney's representation, communication overhead can be measured by a linear function of the message length m, that is $T_{comm} = T_s + m \times T_d$, where T_s is the start-up time and T_d is the per-byte messaging time. To achieve good performance, it is necessary to optimize communication by:

- exploiting local computation as much as possible;
- vectorizing and aggregating communication in order to reduce the number of communications; and
- reducing the message length in a communication step.

As the scientists attempt to model and compute more complicated problems, they have to envisage to develop efficient 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-affine 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 fluid 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),4),13),16)], some provide language support such as add irregular facilities to HPF or Fortran 90[15),21),23)], and some works attempt to utilize caches and locality efficiently[5)].

Hwang, et al.[16)] presented a library called CHAOS, which helps user implement irregular programs on distributed memory machines.

† Department of Computer Software, The University of Aizu
†† Department of Information Management, Southern Taiwan University of Technology
††† Department of Computer Science, Georgia State University

The CHAOS library is divided into six phases. They are Data Partitioning, Data Remapping, Iteration Partitioning, Iteration Remapping, Inspector, and Executor phase. The first 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[21]. Also, in their paper Ref. 4), Das, et al. discussed some primitives to support communication optimization of irregular computations on distributed memory architectures. These primitives coordinate interprocessor 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.

Some researchers have focused on improving data locality and cache utilization for sparse matrix and irregular computations. Mellor-Crummey et al. manually applied a geometric partitioning algorithm based on space-filling curves to map multi-dimensional data to memory, showing large improvements in performance[17]. Mitchell, et al.[18] improved locality using bucket sorting to reorder loop iterations in irregular computations. They improved the performance of two NAS applications (CG, and IS) and a medical heart simulation. However, this method works only for computations containing a single irregular access per loop iteration.

Other works investigated to provide efficient runtime and compiler of parallel irregular reductions[11], and efficient implementation of sparse matrix computations (i.e., LU decomposition, multiplication, etc.)[3].

In this paper, we propose some optimization techniques to minimize the communication cost in pre-processing for compiling irregular scientific codes. We first 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 inter-procedure calls.

2. Reducing Communication Cost for Loop Partitioning with Indirection Array Reference

As mentioned above, there are two kinds of irregular loops — data array are indexed through indirection arrays or indexed through nonlinear subscript expressions — that we called indirection array loop or nonlinear loop respectively. In this section, we propose a communication cost reduction technique for loop partitioning with indirection array reference. 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 difficult to test irregular loop-carried dependence since dependence testing methods for linear subscripts are completely disabled), because most of practical irregular scientific applications have this kind of loops.

2.1 Motivation Examples

Consider the irregular loop below, which is a simplified version extracted from ZEUS-2D code[19]:

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(j1(i)) = X(j0(i)) + Y(j2(i))
S2:   X(j3(i)) = X(j2(i)) + Y(j0(i)) + Z(j2(i))
S3:   Y(j0(i)) = Y(j0(i)) + Z(j2(i)) - X(j0(i))
S4:   Y(j3(i)) = Y(j2(i)) - X(j2(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 specifies 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 com-

Table 1 The owner of executing assignments and required communications for Example 1.

Statement	Owner	array elements required communication
S1:	P_1,	X(j0(i)) ($P_0 \longrightarrow P_1$), Y(j2(i)) ($P_2 \longrightarrow P_1$)
S2:	P_3,	X(j2(i)), Z(j2(i)) ($P_2 \longrightarrow P_3$), Y(j0(i)) ($P_0 \longrightarrow P_3$)
S3:	P_0,	Z(j2(i)) ($P_2 \longrightarrow P_0$)
S4:	P_3,	X(j2(i)), Y(j2(i)) ($P_2 \longrightarrow P_3$)

putes rule is applied, the first step is to distribute the loop into four individual loops each of which includes the statement S1, S2, S3, and S4, respectively. Without loss of generality, suppose that the loop would be executed on 4 processors in parallel. If the array elements Y(i) and Z(i) are aligned with X(i) in the initial distribution, clearly Y(j0(i)) (or Z(j0(i))) is also distributed onto the same processor with X(j0(i)). So we can assume that P_0, P_1, P_2, and P_3 own [X(j0(i)), Y(j0(i))], [X(j1(i))], [X(j2(i)),Y(j2(i)), Z(j2(i))], and [X(j3(i)),Y(j3(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 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 difficult to partition the loop iterations according to the owner computers rule. Therefore, in CHAOS library, Ponnusamy, et al.[21),22)] 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:
  X(j0(i)), Y(j0(i)): $P_0 \longrightarrow P_2$
- Export communication after the loop iteration is executed:
  tmp_Yj0: $P_2 \longrightarrow P_0$
  tmp_Xj1: $P_2 \longrightarrow P_1$
  tmp_Xj3, tmp_Yj3: $P_2 \longrightarrow P_3$

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[23)]. However, when we parallelize a fluid dynamics solver ZEUS-2D code by using almost owner computes rule, we find 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:
  X(j2(i)), Y(j2(i)), Z(j2(i)): $P_2 \longrightarrow P_0$
- Export communication after the loop iteration is executed:
  tmp_Xj1: $P_0 \longrightarrow P_1$
  tmp_Xj3, tmp_Yj3: $P_0 \longrightarrow P_3$

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 efficient computes rule for irregular loop partition[9)]. This approach partitions iterations on a particular processor so 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 first investigate for all processors $P_k, 0 \le k \le m$ (m is the number of processors) in which two sets $FanIn(P_k)$ and $FanOut(P_k)$ for each processor P_k are defined. $FanIn(P_k)$ 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.

2.3 Least Communication Loop Iteration Partitioning Algorithm

Iteration partitioning according to least communication computes rule is to assign iterations to the processors such that whole communication (import and export) steps and message length is minimized. Before the loop partitioning algorithms are presented, we give the following definitions.

Definition 1 Let an iteration i_r be partitioned onto processor P_k, set $D_j(i_r, k)$ is defined as all the data array elements which must be sent from P_j to P_k. $|D_j(i_r, k)|$ is the number of data in the set. Similarly, $D'_j(i_r, k)$ is defined as all the data array elements which must be sent back to P_j from P_k.

Definition 2 Let an iteration i_r be partitioned onto processor P_k, set $FanIn(P_k)$ is defined as a set of processors $\hat{P}_1, \hat{P}_2, \ldots, \hat{P}_l$ which have to send data to processor P_k before the iteration is executed, where $\hat{P}_1, \hat{P}_2, \ldots, \hat{P}_l \in \mathcal{P}$ need not assure contiguous processor numbers ($\mathcal{P}$ is a processor set). Each processor $\hat{P}_j, 1 \le j \le l$ has a degree $|\hat{P}_j| = |D_j(i_r, k)|$. If there is no need to import data when executing the iteration on P_k, $FanIn(P_k) = \emptyset$. Similarly, set $FanOut(P_k)$ is defined as a set of processors $\hat{P}_1, \hat{P}_2, \ldots, \hat{P}_{l'}$ which have to receive data from processor P_k after the iteration is executed. Each processor $\hat{P}_j, 1 \le j \le l'$ has a degree $|\hat{P}_j| = |D'_j(i_r, k)|$. If there is no need to export data after executing the iteration on P_k, $FanOut(P_k) = \emptyset$. $|FanIn(P_k)|$ and $FanOut(P_k)$ are the number of processors in $FanIn(P_k)$ and $FanOut(P_k)$ respectively.

Definition 3 The degrees of the set $FanIn(P_k)$ and $FanOut(P_k)$, $deg(FanIn(P_k))$ and $deg(FanOut(P_k))$, are defined as

$deg(FanIn(P_k)) = \sum_{j=1}^{l} |\hat{P}_j| = \sum_{j=1}^{l} |D_j(i_r, k)|$,
$deg(FanOut(P_k)) = \sum_{j=1}^{l'} |D'_j(i_r, k)|$.

From the above definitions, we have the following proposition.

Proposition 1 The least communication computes rule is to partition an iteration to a processor P_k such that

(1) $|FanIn(P_k)| + |FanOut(P_k)| = \min_{P_j \in \mathcal{P}}(|FanIn(P_j)| + |FanOut(P_j)|)$, where $\mathcal{P}$ is the processor set.

(2) If more than one P_k, say $P_{k_1}, P_{k_2}, \ldots, P_{k_l}$ satisfy the above formula, then select a P_{k_j} such that $deg(FanIn(P_{k_j})) + deg(FanOut(P_{k_j})) = \min_{P_j \in \{P_{k_1}, \ldots, P_{k_l}\}}(deg(FanIn(P_j)) + deg(FanOut(P_j)))$.

In the following algorithms, we assume that a loop body is composed of n statements $S_1, S_2, \ldots, S_n$, and each S_i has one left-hand array elements l_i and h right-hand array elements $r_1, r_2, \ldots, r_h$, $D(S_i)$ and $U(S_i)$ represent the define-variable set and use-variable set of statement S_i respectively. We also abbreviate $d \in P$ if a data is distributed onto processor P. The algorithms for computing $D_j(i_r, k)$ are as follows. Computing $D'_j(i_r, k)$ is similar to Algorithm 1 but it needs data dependence analysis in reverse order of the statements $S_1, S_2, \ldots, S_n$.

Algorithm 1 $D_j(i_r, k)$

```
Input: Iteration $i_r$, Processor $P_k$, and iteration block $\{S_1, S_2, \ldots, S_n\}$;
Output: $D_j(i_r, k)$;

$D_j(i_r, k) = \emptyset$;
for $i = 1, n$
    for $r_t (1 \le t \le h)$ in $S_i$
        if $\{r_t\} \cap (D(S_1) \cup \cdots \cup D(S_{i-1})) = \emptyset \wedge$
           $\{r_t\} \cap (U(S_1) \cup \cdots \cup U(S_{i-1})) = \emptyset$
           $\wedge\ r_t \in P_j (j \ne k)$
           // There is no true dependence between
           // $r_t$ and all of the left hand
           // variables of the statements before $S_i$,
           // and there is no input dependence
           // between $r_t$ and all other right hand
           // variables of the statement before $S_i$.
        then
            if $r_t \neg\in D_j(i_r, k)$ then
              $D_j(i_r, k) = D_j(i_r, k) \cup \{r_t\}$
            end if
        end if
    end for
end for
```

The algorithm 2 computes the set *FanIn*. The computation of *FanOut* is similar with *FanIn*. Finally, Algorithm 3 determines the processor on which the loop iteration is partitioned.

Algorithm 2 $FanIn(P_k)$

```
Input: Processor $P_k$, and $D_j(i_r,k)(0 \le j < m)$;
Output: $FanIn(P_k)$;

$FanIn(P_k) = \emptyset$;
for $i = 1, m, \neq k$
    if $D_j(i_r,k) \neq \emptyset$
        for $d_i \in D_j(i_r,k)$
            if $\exists P_\alpha . d_i \in P_\alpha$ then
                if $P_\alpha \neg \in FanIn(P_k)$ then
                    $FanIn(P_k) = FanIn(P_k) \cup \{P_\alpha\}$
                end if
                $|P_\alpha| = |P_\alpha| + 1$
            end if
        end for
    end if
end for
```

Algorithm 3 $Partition(FanIn, FanOut, P_k)$

```
Input: $FanIn(P_j)$ and $FanOut(P_j)$ for all processor $P_j, 0 \le j \le m$;
Output: iteration executing processor $P_k$;

for $j = 1, m$
    get $P_{k_1}, \ldots, P_{k_l}$ such that
    $|FanIn(P_k)| + |FanOut(P_k)| = \min_{P_j \in \mathcal{P}}(|FanIn(P_j)| + FanOut(P_j)|$;
    select a $P_{k_j}$ such that
        $deg(FanIn(P_{k_j})) + deg(Fanout(P_{k_j})) = \min_{P_j \in \{P_{k_1},\ldots,P_{k_l}\}}(deg(FanIn(P_j)) + deg(Fanout(P_j)))$
end for
```

The node (SPMD) program has three parts: pre-execution import communication (gathering phase), irregular loop execution (executing phase), and post-execution export communication (scattering phase). Notice in the executing local iterations phase, the remapping from global index arrays to local index arrays is required. If an iteration i_r is partitioned to processor P_k, the index array elements $ix(i_r), iy(i_r), \cdots$ may not be certainly resident in P_k. Therefore, we need to redistribute all index arrays so that for the local iterations $iter(P_k) = \{i_{k,1}, i_{k,2}, \ldots, i_{k,\alpha_k}\}$ and every index array ix, elements $ix(i_{k,1}), \ldots, ix(i_{k,\alpha_k})$ are local accessible. Because this topic is out of the scope of this paper, we cannot explain the detail here. We have proposed a method for remapping index array and scheduling communication in redistribution procedure[10].

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 difficult 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, and 3 in Section 2.3 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,

```
        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 $N * C_2 * \cdots * C_n$. Although this transformation spends calculation overhead, because all these lower bounds and upper bounds in the nested loop can be determined in the compile-time, it does not influence the execution time of the loop.

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

We only handled the intra-procedural optimization until now. In some irregular scientific codes, an important optimization required is communication preprocessing among procedure calls. In this section, we extend a classical data flow optimization technique — Partial Redundancy Elimination — to an Interprocedural Partial Redundancy Elimination as a basis for performing interprocedural communication optimization. This technique is also used by Ref. 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 **Fig. 1** Initial intraprocedural analysis in Section 2 (see Ref. 9) also) inserts pre-communicating call (including one buffering and one gathering routine) and post-communicating (buffering and scattering routine) call for 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 **Fig. 2**. Here, since the array IA and IC are never modified 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 Interproce-

```
      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
```

Fig. 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
```

Fig. 2 After interprocedural optimized node program.

```
      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)

Fig. 3 Common and incremental buffering original code.

dural 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 Fig. 2.

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

Consider the program shown in **Fig. 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 modified between flow control from the first loop to the second loop. There will be overlap between required communication data elements made in these two loops. Another case is that the data array and indirection array are the same but the loop bound are different (See Fig. 3 (b)). In this case, the first loop can be

```
      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)

Fig. 4 Interprocedural optimized node code using common and incremental buffering.

viewed as a part of the second loop.

We distinguish communication routines into two kinds — common routine and incremental routine — for such situations. A common communication routine takes more than one indirection array, or takes common part of two indirection arrays. In the example ind Fig. 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 first loop.

Figure 4 (a) and (b) show such optimization from the corresponding original programs represented in Fig. 3 (a) and (b) respectively, where *send* $(\&U, common, Pany)$ indicates sending data according to *common* buffering part and that *send* $(\&A, incml, Pany)$ indicates sending according to *incremental* buffering.

5. Experiments and Performance Results

We now present experimental results to show the efficacy 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. Because our proposed techniques are not implemented in a practical compiler yet, our experimental programs are optimized by hand according to the steps and algorithms in Section 2, 3, and 4. 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 (MPICH 1.2.1 on SGI Origin 2000 and MPICH 1.0.9 on CM-5) The Fortran experimental programs are compiled by `f77 -O3` command.

The first experiment is to select a subroutine OLDA from the code TRFD, appearing in Perfect benchmark[20]. The aim of this experiment is to show the performance of our proposed partitioning method used in nonlinear subscript loops. Since the largest number of array reference is same as the owner computes rule, we only compare the performance of our method with owner computes rule. A simplified version of this loop nest is shown in the left side of **Fig. 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 Fig. 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). The curves denoted as *owner compute* and *least comm* respectively represent that we use owner computes rule and our least communication computes rule to partition the loop. The execution time according to owner computes rule includes run-time inspector-executor analysis time[16], whereas the execution time according to least communication computes rule includes run-time communication analysis proposed in this paper. 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 figure shows that our method gets good performance in most cases.

Our another experiment for irregular loop with interprocedural optimization selects an irregular kernel of fluid dynamics code, ZEUS-

```
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
```

Fig. 5 Simplified version of loop nest OLDA from TRFD.

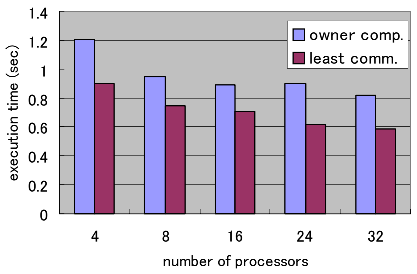

Fig. 6 Execution time of OLDA program on CM-5 for owner computes rule and transformed single loop using least communication computes rule for loop partitioning.

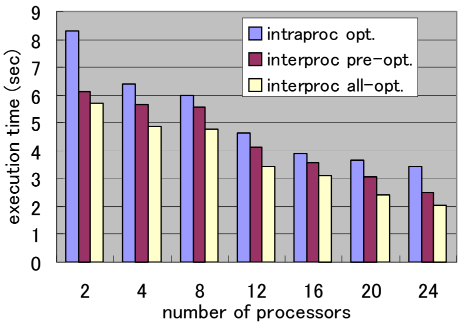

Fig. 7 Effect of interprocedural communication optimization in executing ZEUS-2D irregular loops (Execution time) on SGI Origin 2000.

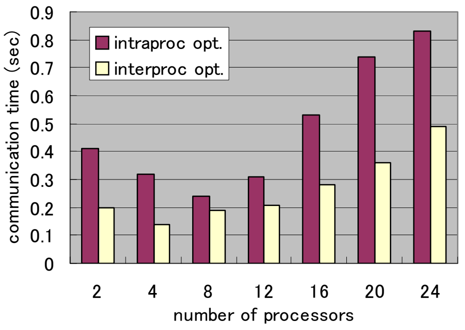

Fig. 8 Communication time for intraprocedural and interprocedural optimization in executing ZEUS-2D irregular loops on SGI Origin 2000.

2D for our study. ZEUS-2D is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for astrophysical radiation magnetohydro dynamics problems[19]. ZEUS-2D solves the equations of ideal (non-resistive), non-relativistic, hydrodynamics, including radiation transport, (frozen-in) magnetic fields, rotation, and self-gravity. Boundary conditions may be specified as reflecting, periodic, inflow, or outflow. The kernel irregular subroutine `emfs` includes some loops whose 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.

In **Fig. 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 figure 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 significant part of the total execution time and the communication optimization makes significant difference in the overall performance of the program.

In **Fig. 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 influences significantly total performance of parallel program.

6. Conclusions

The efficiency of loop partitioning influences performance of parallel program considerably. For automatically parallelizing irregular scientific codes, the owner computes rule is not suitable for partitioning irregular loops. In this paper, we have presented an efficient 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 efficacy of our schemes. In the future, we will examine the optimization techniques to more irregular applications and implement the techniques into a practical parallelizing compiler.

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

References

1) Allen, R. and Kennedy, K.: Optimizing compilers for Modern Architectures, Morgan Kaufmann Publishers (2001).

2) Agrawal, G. and Saltz, J.: Interprocedural compilation of Irregular Applilcations for Distributed memory machines, *Language and Compilers for Parallel Computing*, pp.1–16 (Aug. 1994).

3) Asenjo, R., Gutierrez, E., Lin, Y., Padua, D., Pottengerg, B. and Zapata, E.L.: On the Automatic Parallelization of Sparse and irregular Fortran codes, Technical Report 1512, University of Illinois at Urbana-Champaign, CSRD (Dec. 1996).

4) Das, R., Uysal, M., Saltz, J. and Hwang, Y-S.: Communication optimizations for irregular scientific computations on distributed memory architectures, *Journal of Parallel and Distributed Computing*, Vol.22, No.3, pp.462–479 (Sept. 1994).

5) Ding, C. and Kennedy, K.: Improving cache performance of dynamic applications with computation and data layout transformations, *Proc. SIGPLAN'99 Conference on Programming Language Design and Implementation*, Atlanta, GA (May 1999).

6) Guo, M., Nakata, I. and Yamashita, Y.: Contention-free communication scheduling for array redistribution, *Parallel Computing*, Vol.26, pp.1325–1343 (2000).

7) Guo, M. and Nakata. I.: A framework for efficient array redistribution on distributed memory machines, *The Journal of Supercomputing*, Vol.20, No.3, pp.253–265 (2001).

8) Guo, M., Pan, Y. and Liu, C.: Symbolic Communication Set generation for irregular parallel applications, *The Journal of Supercomputing* (2002).

9) Guo, M., Liu, Z., Liu, C. and Li, L.: Reducing Communication cost for Parallelizing Irregular Scientific Codes, *Proc. 6th International Conference on Applied Parallel Computing*, Finland (June 2002).

10) Guo, M., Chang, W.-L., Li, L. and Pan. Y.: Efficient Loop Partitioning for Parallel Codes of Irregular Scientific Computation, *Proc. International Conference on Algorithms and Architectures for Parallel Processing*, IEEE Computer Society Press, Beijing, China (Oct. 2002).

11) Gutierrez, E., Plata, O. and Zapata. E.L.: On automatic parallelization of irregular reductions on scalable shared memory systems, *Proc. Fifth International Euro-Par Conference*, pp.422–429, Toulouse, France (Aug.–Sep. 1999).

12) Gutierrez, E., Asenjo, R., Plata, O., Plata, and Zapata, E.L.: Zapata. Automatic parallelization of irregular applications, *Parallel Computing*, Vol.26, No.2000, pp.1709–1738 (2000).

13) Han, H. and Tseng, C.-W.: Improving compiler and run-time support for adaptive irregular codes, *Proc. International Conference on Parallel Architectures and Compilation Tech-*

niques, Paris, France (Oct. 1998).
14) Hu, Y., Cox, A. and Zwaenepoel, W.: Improving fine-grained irregular shared-memory benchmarks by data reordering, *Proc. SC'00*, Dallas, TX (Nov. 2000).
15) Hu, Y., Johnsson, S.L. and Teng, S.-H.: High Performance Fortran for highly irregular problems, *Proc. Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV (June 1997).
16) Hwang, Y.-S., Moon, B., Sharma, S.D., Ponnusamy, R., Das, R. and Saltz, J.: Runtime and language support for compiling adaptive irregular programs on distributed memory machines, *Software-Practice and Experience*, Vol.25, No.6, pp.597–621 (1995).
17) Mellor-Crummey, J., Whalley, D. and Kennedy, K.: Improving memory hierarchy performance for irregular applications, *Proc. 1999 ACM International Conference on Supercomputing*, Rhodes, Greece (June 1999).
18) Mitchell, N., Carter, L. and Ferrante, J.: Localizing non-affine array references, *Proc. International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, LA (Oct. 1999).
19) Stone, J.M. and Norman, M.: ZEUS-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions: The hydrodynamic algorithms and tests, *Astrophysical Journal Supplement Series*, Vol.80, pp.753–790 (1992).
20) Berry, M., Chen, D., Koss, P., Kuck, D., Lo, S., Pang, Y., Roloff, R., Sameh, A., Clementi, E., Chin, S., Schneider, D., Fox, G., Messina, P., Walker, D., Hsiung, C., Schwarzmeier, J., Lue, K., Orzag, S., Seidl, F., Johnson, O., Swanson, G., Goodrum, R. and Martin, J.: The PERFECT club benchmarks: Effective performance evaluation of supercomputers, *International Journal of Supercomputing Applications*, Vol.3, No.3, pp.5–40 (1989).
21) Ponnusamy, R., Hwang, Y-S., Das, R., Saltz, J., Choudhary, A. and Fox, G.: Supporting irregular distributions in Fortran D/HPF compilers, Technical Report CS-TR-3268, University of Maryland, Department of Computer Science (1994).
22) Ponnusamy, R., Saltz, J., Choudhary, A., Hwang, S. and Fox, G.: Runtime support and compilation methods for user-specified data distributions, *IEEE Transactions on Parallel and Distributed Systems*, Vol.6, No.8, pp.815–831 (1995).
23) Ujaldon, M., Zapata, E.L., Chapman, B.M. and Zima, H.P.: Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation, *IEEE Transactions on Parallel and Distributed Systems*, Vol.8, No.10, pp.1068–1083 (Oct. 1997).

(Received June 6, 2002)
(Accepted October 2, 2002)

Minyi Guo received his Ph.D. degree in information science from University of Tsukuba, Japan in 1998. From 1998 to 2000, Dr. Guo had been a research fellow of NEC Soft, Ltd., Japan. He is currently an assistant professor at the Department of Computer Software, The University of Aizu, Japan. From 2001 to 2002, he was a visiting professor of Georgia State University, USA. Dr. Guo has served as program committee or organizing committee member for many international conferences, and delivered more than 15 invited talks in USA, Australia, China, and Japan. He has also been invited as a referee by many famous international journal including IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Systems, Man, and Cybernetics, etc. Dr. Guo's research interests include parallel and distributed processing, parallelizing compilers, data parallel languages, data mining, molecular computing and software engineering. He is a member of the ACM, IEEE, IEEE Computer Society, and IEICE. He is listed in Marquis Who's Who in Science and Engineering.

Weng-Long Chang was born in 1966 in Taiwan. He received the B.S. and M.S. degrees in Computer Science and Information Engineering from Feng Chia University and in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, in 1988 and 1994, respectively. He is currently an assistant professor in Department of Information Management, Southern Taiwan University of Technology, Tainan. His research interests include languages and compilers for parallel computing and molecular computing.

Yi Pan received his Ph.D. degree in computer science from the University of Pittsburgh, USA, in 1991. Currently, he is an associate professor in the Department of Computer Science at Georgia State University. His research interests include parallel and distributed computing, optical networks and wireless networks. His pioneer work on computing using reconfigurable optical buses has inspired extensive subsequent work by many researchers. He is a co-inventor of three U.S. patents (pending) and several provisional patents. He has published more than 120 research papers including over 50 journal papers (more than 20 of which have been published in various IEEE journals) and received many awards from agencies such as NSF, AFOSR, JSPS, IISF and Mellon Foundation. Dr. Pan is currently serving as an editor-in-chief or editorial board member for 7 journals including 3 IEEE Transactions. He has also served as a program or general chair for several international conferences and workshops. Dr. Pan has delivered over 40 invited talks, including keynote speeches and colloquium talks, at conferences and universities worldwide. Dr. Pan is an IEEE Distinguished Speaker (2000–2002), a Yamacraw Distinguished Speaker (2002), a Shell Oil Colloquium Speaker (2002), and a senior member of IEEE. He is listed in Men of Achievement, Who's Who in Midwest and Who's Who in America.