A Technique to Eliminate Redundant Inter-Processor Communication on Parallelizing Compiler TINPAR

Atsushi KUBOTA[1]*, Shogo TATSUMI[1], Toshihiko TANAKA[1],
Masahiro GOSHIMA[1], Shin-ichiro MORI[1],
Hiroshi NAKASHIMA[2], and Shinji TOMITA[1]

[1] Department of Information Science, Kyoto University
[2] Department of Information and Computer Sciences,
Toyohashi University of Technology

Abstract. Optimizing inter-processor(PE) communication is crucial for parallelizing compilers for message-passing parallel machines to achieve high performance. In this paper, we propose a technique to eliminate redundant inter-PE messages. This technique utilizes a data-flow analysis to find a definition point that corresponds to a use point where the definition and the use are occurred in different PEs. If several read accesses occurred in the same PE use the data defined at the same definition point in another PE, redundant inter-PE messages are eliminated as follows: only one inter-PE communication is performed for the earliest read access and the previously received data are used for the following read accesses. In order to guarantee the consistency of the data, a valid flag and a sent flag are provided for each chunk of received data. The control of these flags is equivalent to the coherence control by the self invalidation on a compiler aided cache coherence scheme.

keywords: parallelizing compiler, message-passing multiprocessor, dependence analysis, data-flow analysis, message coalescing

1 Introduction

In our research project, we are developing a parallelizing compiler TINPAR(TINy PARallelizer)[3] for message-passing multiprocessors. TINPAR accepts scientific programs with data parallelism as many HPF[1] compilers. TINPAR accepts sequential programs with data distribution directives written in an imperative language such as FORTRAN and produce an SPMD code for message-passing multiprocessors. Each statement is scheduled at compile time with the owner computes rule.

For such compilers, optimizing inter-processor(PE) communication is crucial to achieve good performance. There are two major optimization techniques:

– reducing communication costs and

* Research Fellow of the Japan Society for the Promotion of Science

– hiding communication latency by overlapping communication and computation.

Reducing communication costs is achieved by

– reducing the number of invocation of communication and
– eliminating unnecessary communication.

The former technique is achieved by packing elements of the same array to be transferred to the same PE into one message (*message vectorization*) and elements from the different arrays to be transferred to the same PE into one message (*message aggregation*). In our earlier work, we have already implemented these techniques for TINPAR.

This paper presents a technique called *message coalescing* that eliminates unnecessary communication and its implementation for TINPAR.

In the remainder of this paper, we show requirements for a program to apply message coalescing in Section 2. In Section 3, we describe a data-flow analysis to obtain the def-use chain of data, which is necessary for message coalescing. In Section 4, we discuss an implementation issue of message coalescing using the data-flow analysis. Finally in Section 5, we show concluding remarks.

2 Message Coalescing

2.1 Message Coalescing

Message coalescing[2] is a technique to reduce the volume of messages; when an element e of an array in the remote memory of a PE s is read by another PE r and the element e is read by the PE r multiple times, the transfer of the value of the element e can be performed just once instead of multiple transfers. The received value is kept in the local memory of the PE r and reads of the value of the remote element e are performed by supplying the value kept in the local memory.

Fig. 1 is an example program of message coalescing shown in [2]. When the arrays `A` and `B` are distributed among several PEs, read accesses to an element of the array `B` in the right hand side of the assignment statement may require inter-PE communication. In this case, the element used as `B(i+2)` in the l'th iteration is used again as `B(i+1)` in the $l+1$'th iteration. In this way, when the same data is used multiple times, the transfer of the data is performed just once and send/receive library routines are not called on the following uses.

2.2 Conditions to Apply Message Coalescing

Hiranandani, et al[2] did not show conditions for programs required to apply message coalescing clearly. We therefore clarify the required condition for message coalescing.

To apply message coalescing, it must be examined, for each read(*use*) of an element, which statement writes(*defines*) the value of the element. If the

```
do i=1,n
  A(i) = B(i+1) + B(i+2)
enddo
```

⇓ Parallelized with block-wise distribution of arrays `A,B`

```
/* P:the number of PEs, p:PE ID */
send B(1..2) to p-1
recv tmp_B(1..2) from p+1
for l_i=1,n/P-2
  A(l_i) = B(l_i+1) + B(l_i+2)
endfor
A(n/P-1) = B(n/P) + tmp_B(1)
A(n/P) = tmp_B(1) + tmp_B(2)
```

Fig. 1. Message coalescing

statement is executed multiple times in a loop, the iteration in which the value is defined must be determined. We call these statements and iterations in the program in which the values of data are defined and used as the *definition point* and the *use point*. For data which is defined and used repeatedly in the program, it is required to examine the sequence of definition and use points. Let d be a definition point in this sequence and d' be the next definition point. The value to be read at all use points r_1, r_2, $\ldots$ r_n put between d and d' is the value updated at the definition point d. At the use point r_1, the data is received and stored in a buffer area on the receiver PE then it is read from this buffer area, while at the other use point r_2, $\ldots$ r_n, the data is simply read from the buffer area.

3 Data-Flow Analysis

We adopt a tree structure called LWT(Last Write Tree)[4] to represent the precise data-flow of programs. Because many data elements are defined and used in a loop nest, we focus on the analysis of definition and use points in a loop nest.

3.1 LWTs

In this section, we show how to construct LWTs for a loop nest. We use an example code shown in Fig. 2.

Let (i^r, j^r) be the iteration of a use point and (i^w, j^w) be the iteration of a definition point. Then we get the definition point for each use of elements of the array `a` as follows.

- If $j^r = 1$, the value to be used is defined before entering the loop. We use $\perp$ to represent this situation.

```
for i = 1, n do
  for j = 1, n do
    ... = a(i, j-1)
    a(i, j) = ...
  endfor
endfor
```

Fig. 2. A loop with a pair of definition and use points

– If $2 \leq j^r \leq n$, the value to be used is defined in the iteration $(i^r, j^r - 1)$. This data dependence is carried by the inner loop.

The LWT for this loop nest is shown in Fig. 3. For a pair of a use point and a definition point that defines the value to be used, the iteration of the definition point is expressed using the iteration of the use point (i^r, j^r). Dependence levels in the loop nest are also shown in LWTs.

Dependence level is the nest level of the most outer loop which carries data dependence. In the example shown in Fig. 3, the dependence level is 2, which is the nest level of the inner loop.

If the dependence is carried by the outer loop, the dependence level is 1. If the definition point is located outside the loop(that is the case $\perp$), the dependence level is 0. If there is no loop carried dependence, the dependence level is defined as ∞.

In general, for a n-deep loop nest, the LWT is constructed as follows.

Let $\boldsymbol{i^w} = (i_1^w, i_2^w \ldots, i_n^w)$ be an iteration of a definition point in a n-deep loop nest and $\boldsymbol{i^r} = (i_1^r, i_2^r \ldots, i_n^r)$ be an iteration of a use point.

Let F_r and F_w be subscripts of use and definition of array elements in the loop. If there is a data dependence, there exist a use iteration $\boldsymbol{i^r}$ and a definition iteration $\boldsymbol{i^w}$ such that $F_r \boldsymbol{i^r} = F_w \boldsymbol{i^w}$.

The LWT for this loop is constructed as follows:

1. Express $\boldsymbol{i^w}$ using $\boldsymbol{i^r}$. If F_w is invertible, $\boldsymbol{i^w} = F_w^{-1} F_r \boldsymbol{i^r}$.
2. Restrict the range of the use iteration variable $\boldsymbol{i^r}$ according to the loop bounds constraints of the iteration variable $\boldsymbol{i^w}$. For a use point $\boldsymbol{i^r}$ that does not satisfy this restriction, the corresponding definition point is $\perp$.
3. For each loop nest level which satisfies $1 \leq l \leq n$, select one from the following steps according to the relationship between i_l^w and i_l^r.
   – If $i_l^w > i_l^r$, the definition point is lexicographically after the use point. The definition point is $\perp$.
   – If $i_l^w < i_l^r$, there is a loop carried dependence. The definition point is $\boldsymbol{i^w} = F_w^{-1} \boldsymbol{i^r}$.
   – If $i_l^w = i_l^r$, go down to the nest $l + 1$ and repeat the selection from these steps.
4. If $i_l^w = i_l^r$ is satisfied for all l nest levels, the data dependence is determined by the execution order of the definition and the use in the loop body of

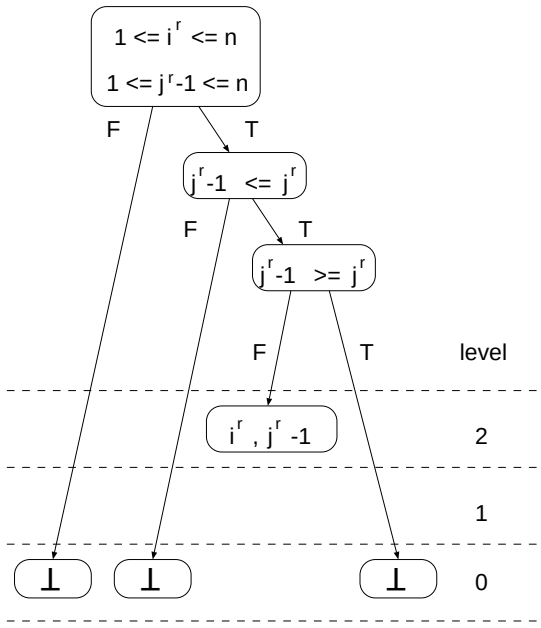

Fig. 3. LWT

the inner most nest. If the definition point precedes the use point, there is a loop independent dependence and the definition point is expressed using the iteration $\boldsymbol{i^w}$. If the use points precedes the definition point, there is no dependence and the definition point is $\perp$.

In the example shown in Fig. 2, there is a relationship between a use point and a corresponding definition point as $(i^w, j^w) = (i^r, j^r - 1)$. This relationship is calculated as follows.

Firstly, restrict the range of the use iteration according to the loop bounds of the definition point. Substituting the replationship $i^w = i^r$ and $j^w = j^r - 1$ shown above to the loop bounds conditions $1 \leq i^w \leq n$ and $1 \leq j^w \leq n$, we get two inequalities $1 \leq i^r \leq n$ and $1 \leq j^r - 1 \leq n$. Rearranging the latter inequality gives $2 \leq j^r \leq n + 1$. For $j^r = 1$, the corresponding definition point is located out of the loop($\perp$). This means that the value defined before entering the loop is read at the use point $j^r = 1$.

Then the outer loop is examined and the two nodes corresponding to $i^w \leq i^r$ and $i^w \geq i^r$ is created in the LWT. Because these conditions are always true, these nodes are not added in Fig. 3.

For the inner loop nest, the condition $j^w \leq j^r - 1 \leq j^r$ is always true while the condition $j^w = j^r - 1 \geq j^r$ is always false. Thus the iteration $(i^r, j^r - 1)$ is the definition point for the use point (i^r, j^r). Because the dependence between the definition point and the use point is carried by the inner loop (level 2), the node

$(i^r, j^r - 1)$ which represents the definition point is added in the level 2 row in the LWT.

3.2 Merging Multiple LWTs

In our method, a use point u which requires communication is selected and then its corresponding definition point d to the use point can be obtained by tracing the edges of the LWT. The sequence of the def-use chain is constructed by finding all use points corresponding to the definition point d and put these define and use points in the lexicographically order of the iterations. Arranging the LWTs corresponding to these use points in the def-use chain in the lexicographical order, the sequence of the LWTs can be constructed. Then the def-use chain of each nest level is constructed by tracing the same nest level of the sequence of the LWTs.

4 Implementation Issue

In this section, we show an optimization technique at compile-time and how it works at run-time to realize message coalescing.

Depending on whether the nest level of a definition point is equal to the nest level of a use point or not, we use different techniques to realize message coalescing. If the both nest levels are equal to each other, a dynamic message coalescing technique using run-time libraries is applied. We call this technique as *message caching*. If the nest levels are different, that is, the definition point is out of the loop in which the use point located, a static message coalescing is performed by reordering the communication code and making temporary array variables to keep the received data.

4.1 Message Caching

Basic Principle Data elements previously received by PE r are kept in the temporary buffer(called *receive buffer* for short) maintained by the run-time library routines for communication in PE r. For a receive request for a data element in PE r, if the valid value of the data element r is kept in the receive buffer, this valid value is supplied from the receive buffer. Decisions of the duration of the value kept in the receive buffer and the way to ensure the validness of the value of the maintained data element are the implementation dependent issues.

Management of Receive Buffer The information about "which data is used multiple times" and "where the definition point and the last use point of the data are" is obtained by the examination of the LWT. With this information, allocation and release of an area in the receive buffer and the consistency control for the data which is defined multiple times are performed.

In order to ensure the consistency, a *valid flag* is provided to indicate the validness of the value of the data kept in the receive buffer. A *sent flag* is also

provided in the sender PE to indicate whether the data is sent after the value of the data is defined.

Because communication is performed with synchronized send/receive messages, the sent flag that corresponds to the valid flag of the receive buffer must be provided in the sender PE.

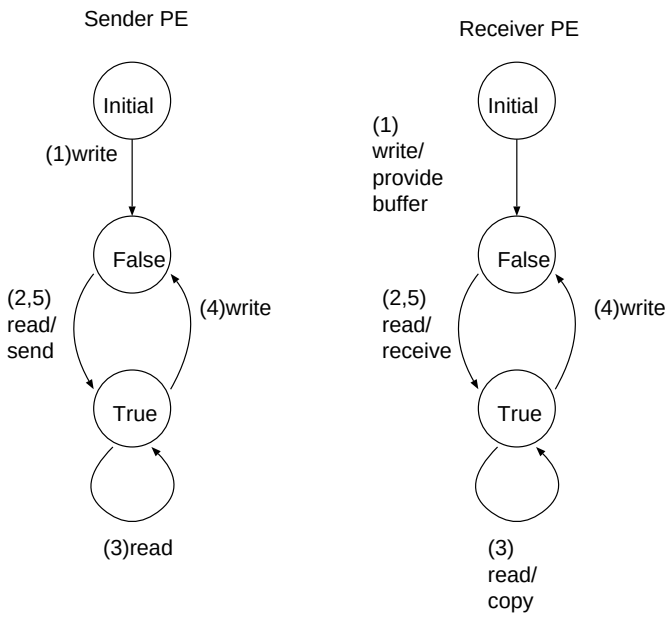

Fig. 4. State diagram of control of receive buffer

The consisitency of the data kept in the receive buffer is maintained as shown in Fig. 4. The details is shown as follows.

1. When the data to be sent is defined in the sender PE for the first time, the sent flag turns to false. In the receiver PE, area is allocated in the receive buffer and the valid flag turns to false.
2. When the data is used in the receiver PE, the data is transferred from the sender PE and the sent flag turns to true. In the receiver PE, the data is received, and the valid flag turns to true. The received data is saved into the receive buffer and the value to be used is supplied from the receive buffer.
3. While the data is not defined in the sender PE, the data is supplied from the receive buffer for the use of the data in the receiver PE.
4. When the data is defined in the sender PE again, the sent flag turns to false. In the receiver PE, the valid flag corresponding to the redefined data turns to false and the area in the receive buffer is released.

5. When the data is used in the receiver PE again, the updated value of the data is transferred from the sender PE and the sent flag is turned to true. In the receiver PE, the data is received and the valid flag is turned to true. The received data is saved into the receive buffer and the value to be used is supplied from the receive buffer.

The receive buffer can be regarded as a cache area controlled by software. When the value of the data is updated, this cache area is invalidated without control signal by making use of the the result of the analysis at compile-time. The control of these flags is equivalent to the coherence control by the self invalidation on a compiler aided cache coherence scheme.

Optimization The area in the receive buffer and the both valid and sent flags corresponding to that area are deleted when they become unnecessary. This deleting point in the execution is the last use point examined by the data-flow analysis. After the use at this point, any use and definition to the data is not performed. It is therefore guaranteed that the area of the receive buffer and the both valid and sent flags can be deleted. This prevents the excessive use of the memory for the receive buffer.

If there exist some control statements such as `if` statements in a loop and the data-flow is not precisely analyzed, it is enough for optimization at compile-time to find a relationship between a use points and a definition point in the loop. At run-time, the consistency of the data is guaranteed by turning the both sent and valid flags to false at the definition points and turning these flags to true at the use points.

If the valid and the sent flags are prepared for each element of the array, the send/receive status of the element can be presented accurately by these flags. However, this array of flags amounts to huge tables and the cost to define and use of element in the table may not be negligible.

To reduce the size of table, the buffer area and the flags are provided for a part of the array represented by a rectangular region. All elements in this rectangle must be defined and used in succession so that the definition points and the use points can be regarded as one definition point and one use point respectively.

Each dimension of the rectangular region in the array is represented by a tuple $[l : u : s : b]$ where l, u, s, b represent lower bound, upper bound, step size and block size respectively.

If the size of the rectangle maintained together is changed at run-time, the rectangle is divided into smaller rectangles and flags are provided for each rectangles.

4.2 Static Message Coalescing

The static message coalescing is applied if the definition point is located out of the loop. The code to receive data is moved out from the loop and the received values are saved in a temporary array variable. In the loop nest, the use of the

data is replaced by the use of the temporary variable. If the data transferred can be vectorized, a temporary array variable is provided.

This code replacement technique can be applied when the data reference accompanied with communication can be separated into communication and reading of the value of the variable. Thus a temporary array variable for the data to be received can be provided at compile time. This leads to reduce the overhead of allocation and release of buffer area managed by the run-time library routines.

```
for i = 1, n do
  for j = 1, n do
    for k = 1, n do
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    endfor
  endfor
endfor
```

⇓ Insert send/recv codes

```
for l_i = 1, n/P do /* P: Number of PE */
  for j = 1, n do
    for k = 1, n do
      send b(k,j) to owner(a(i, j))
      recv tmp_b = b(k,j)
      c(l_i,j) = c(l_i,j) + a(l_i,k) * tmp_b
    endfor
  endfor
endfor
```

⇓ Move send/recv codes

```
for pe=1, P (pe ≠ p) do
  send b(n(p-1)/P+1..n(p-1)P,1..n) to pe
  recv tmp_b(n(pe-1)/P+1..n(pe-1),1..n) from pe
endfor
for l_i = 1, n/P do
  for j = 1, n do
    for k = 1, n do
      c(l_i,j) = c(l_i,j) + a(l_i,k) * tmp_b(k,j)
    endfor
  endfor
endfor
```

Fig. 5. Moving send/recv codes in the matrix multiply program

In the matrix multiply program shown in Fig. 5, all the arrays `a`, `b`, `c` are divided in the first dimension and distributed to PEs. Then elements of the array

`b` excluding the locally located ones must be used accompanied with communication. Because all values of the array `b` are not updated in the loop nest, the communication code can be moved out from the loop.

5 Conclusion

In this paper, we have shown the technique for parallelizing compilers to reduce the redundant communication of data. The data-flow analysis to detect the redundant communication and the dynamic/static message coalescing technique to reuse the transferred data are described.

In the current implementation, the static message coalescing can be performed when there is no definition point in the loop nest. We got parallelized program of problem size 128×128 shown in Fig. 5 and measured the execution time of it on Fujitsu AP1000 multiprocessor. We achieved the 4 times speedup compared to the program parallelized without message coalescing.

In the future work, we will examine more detailed data-flow analysis across loop nests. In addition, we will evaluate the effectiveness of the techniques we proposed by incorporating them into the parallelizing compiler TINPAR.

Acknowledgments

We would like to thank Fujitsu Laboratory Ltd. for offering their AP1000. We also wish to thank all members in Tomita Laboratory of Kyoto University for their contribution through daily discussion. This work is supported in part by the Grant-in-Aid #00093007 from the Ministy of Education, Science, Sports and Culture.

References

1. High Performance Fortran Forum. High Performance Fortran language specification (ver. 2.0). Technical report, January 1997.
2. Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluating compiler optimizations for fortran D. *J. Parallel and Distributed Computing*, 21:27–45, 1994.
3. Atsushi Kubota, Ikuo Miyoshi, Koji Maeyama, Shin ya Goto, Shin ichiro Mori, Hiroshi Nakashima, and Shinji Tomita. TINPAR: A parallelizing compiler for message-passing multiprocessors. In *Proc. of Int'l Symposium on Parallel and Distributed Supercomputing*, pages 214–223, September 1995.
4. Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *20'th Annual ACM Symposium on Principles of Programming Languages*, pages 2–15, January 1993.