Message-Passing Generator for HPF on CRAY T3D

Kyeongdeok Moon, Nanjoo Ban, Teageun Kim, Joongkwon Kim, and *Yearback Yoo*, Supercomputer Center, KIST/SERI, UhEun-Dong 1, Yoosung-Gu, Taejon, Korea

ABSTRACT : *Writing a parallel program using message-passing primitives is time-consuming and error-prone. Therefore, many researchers perform to develop data parallel programming model which is easy and portable. In compiling a data parallel program in distributed memory machine, effective detection of non-local data reference and insertion of message-passing primitives are important to program performance. Thus, we are developing the message-passing generator for HPF program on CRAY T3D. It fetches all non-local data needed by each processor, and puts them into processor's local memory using PVM message-passing primitives.*

1. Introduction

Solving a large-scale problem in scientific applications have increased the need for high performance computing which requires more enhanced technology in parallel processing than in high speed computer architecture. In the beginning, writing parallel program using the message-passing primitives is the only way to exploit parallelism, but it is time-consuming and error-prone. Therefore, many researchers have sought to provide an efficient machine-independent programming model over the past decade. Initial efforts concentrated on automatic parallelization. However, after more than five years of research, most researchers admitted that fully automatic techniques were not successful in all cases and believed that data parallel programming model in which a programmer provide an additional information of parallelism to a program should be effective and portable[Phi93][Tse93].

Many researches, such as Fortran D[Tse93], High Performance Fortran(HPF)[HPF93], and Data Parallel C, are performed to consider extensions to the existing languages, such as Fortran and C, that allow to specify parallelism. HPF, de facto standard of data parallel language, is extension of Fortran 90. The idea behind HPF is to develop the minimal set of extensions to Fortran 90 supporting data parallel programming model. The data parallel programming model is defined as single thread, global name space and loosely synchronous parallel computation. The purpose of HPF is to provide software tools, such as compiler, that produce very efficient code for distributed memory systems[HPF93].

In distributed memory system, such as CRAY T3D, data parallel program is written in global address space, then compiler transforms it into Single Program Multiple Data(SPMD) parallel program with its own local address to be run concurrently on processors. Compiler distributes data on each processor using information of user directive, and computation using "Owner Computes Rule". Therefore, non-local data reference is required in some cases. This causes compiler to insert message-passing primitives.

We develop a Parallel Programming TRANslator (PPTran) which is a tool that transforms data parallel programs written in HPF to parallel program with explicit Parallel Virtual Machine(PVM) message-passing primitives[Moo95]. In compiling data parallel program for a distributed memory machine, efficient detection of non-local data reference and insertion of message-passing primitives are important to program performance. In this paper, we describe the message-passing generator for HPF program on CRAY T3D. It fetches all non-local data needed by each processor, and puts them into processor's local memory using PVM message-passing calls.

2. Overview of PPTran

PPTran is a tool that transforms data parallel program written in HPF to SPMD parallel program written in Fortran 77 and PVM message-passing primitives. Fig. 1 shows the structure and compilation module of PPTran. PPTran consists of four modules : Program Analyzer, Partitioner, Message Generator and Code Generator[Moo95].

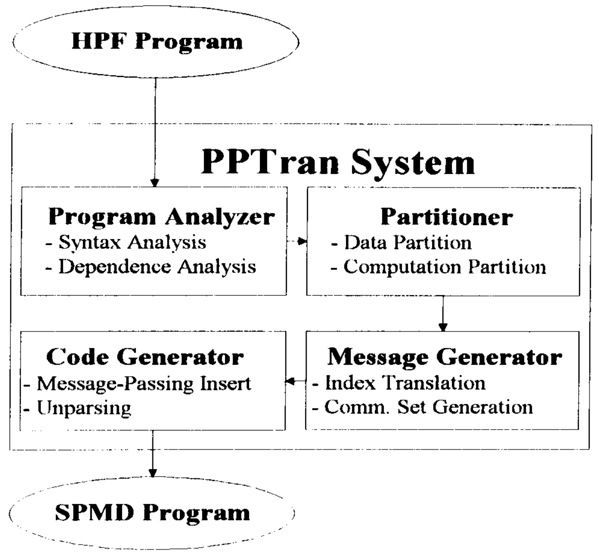

Figure 1. Structure of PPTran

Program analyzer performs syntax analysis and dependence detection from HPF program. Through syntax analysis, PPTran builds the Abstract Syntax Tree(AST) and symbol table using lex and yacc. Dependence of program represents the parallelism of program and consists of two types : data dependence and control dependence.

In order to exploit the parallelism, and improve the performance on distributed memory system, PPTran must distribute the data and computation to increase the data locality. Data are partitioned onto each processor using information of ALIGN and DISTRIBUTE directives which are analyzed by Program Analyzer. The current PPTran only supports static distribution. Given the data distribution, PPTran partitions the computation, such as array expression, and forall, onto each processor using "Owner Computes Rule". Thus, a processor only executes those iterations for which the left-hand side(lhs) of an assignment statement is local to p.

HPF supports a programming model based on single address space. Message Generator performs the index translation and communication set generation. PPTran must distribute the data to each processor that has its own local address, so each processor has local address space and global address space. Thus, PPTran is needed to translate global address to local address and processor number, and local address to global address. Given the data and computation partition, the computation performed on a processor may require data which reside on other processor. Thus, PPTran is required to determine the communication sets which are non-local elements needed by processors.

Finally, PPTran performs the SPMD code generation. Code Generator inserts the PVM message-passing primitives to the AST and unparses the AST to produce the SPMD parallel program.

3. Message-Passing Generator

In HPF, array statements, do loops and forall statements are used to express data parallelism. Given the data and computation partition, a processor executes only those iteration for which the lhs of an assignment statement is local to p. If a processor contains right hand side(rhs) in an assignment whose lhs is not local to processor p, non-local data reference is occurred. This forces the compiler to detect non-local data reference and to insert message-passing primitives effectively[Gup94] [Sti93].

We represent the array using slice, (*lower bound* : *upper bound* : *stride*). On a processor, both indices of array, owned by it and used for computation by it, can be expressed using slice[Gup94][Sti93]. Thus, a processor can detect non-local data reference using intersection of above two slices. The intersection of two slices, $S_i = (l_i{:}u_i{:}s_i)$ and $S_j = (l_j{:}u_j{:}s_j)$, is eq (1) :

$$S_i \cap S_j = (\max(l_i : l_j) : \min(u_i : u_j) : (s_i * s_j) / g) \qquad (1)$$

In eq (1), stride of intersection is the least common divisor(lcm) of s_i and s_j, and g is the greatest common divisor(gcm) of s_i and s_j.

Let array $A(l_1{:}u_1{:}s_1)$ and $B(l_2{:}u_2{:}s_2)$ be distributed on P processors, where P is the total number of processors. Consider the array statement :

$$A(l_a : u_a : s_a) = F(B(l_b : u_b : s_b))$$

To execute the above statement on each processor concurrently, following 3 steps are performed[Sti93].

1. Send those elements of B that owned by processor S but required by processor D
2. Receive non-local elements of B required to update a local element of $A(l_a{:}u_a{:}s_a)$ on processor S but owned by processor D
3. Execute the computation on each processor

Let array $A(l_1 : u_1 : s_1)$ and $B(l_2 : u_2 : s_2)$ be block distributed onto P processors. When executing the above array statement, $A(l_a{+}i{*}s_a)$ and $B(l_b{+}i{*}s_b)$ are referred to correspondingly on each processor. So, $A(l_a{+}i{*}s_a)$ and $B(l_b{+}i{*}s_b)$ must be located on the same processor's local memory before execution. Therefore, the compiler must compute the SendSet that a processor S must send to a processor D and the RecvSet that the processor D must receive from processor S.

To determine the SendSet from sending processor S to receiving processor D, OWND(A) and OWNS(B) must be computed :

$$OWND(A) = (l_1 + j : u_1 : s_1), 0 \leq j \leq n_1 \quad (2)$$
$$OWNS(B) = (l_2 + i : u_2 : s_2), 0 \leq i \leq n_2 \quad (3)$$

OWND(A) is the set of array A owned by processor D, and OWNS(B) is the set of array B owned by processor S. In eq (2) and (3), n_1 and n_2 are consecutive block size allocated to processor D and S, respectively.

The SendSet from the processor S to D can be computed by obtaining index set of array A that are owned by processor D and involved in the computation, SD, and index set of array B that are owned by processor S and involved in the computation on processor D, SS. The index set of array A owned by processor D involved in computation is in eq (4), and the index set of array B owned by processor S involved in computation is in eq (5).

$$SD = OWND(A) \cap (l_a : u_a : s_a) \quad (4)$$
$$SS = OWNS(B) \cap (l_b : u_b : s_b) \quad (5)$$

Since SD is the index set of array A on processor D, we need to obtain the corresponding index set of array B in the same sequence on processor S. If $l_a + i * s_a \in SD$ and $l_b + i * s_b \in SS$ for any integer i, then S sends $B(l_b + i * s_b)$ to D which uses it with $A(l_a + i * s_a)$ to execute computation. Following mapping function in eq (6) generates the index set of array A that are owned by processor D and referenced in the computation.

$$Map(y) = \frac{y - l_a}{s_a} s_b + l_b \quad (6)$$

In eq (6), y is index of array B which is involved in computation. $A(l_a + i * s_a)$ corresponds to $B(l_b + i * s_b)$ $= B(Map(l_a + i * s_a))$. Therefore if $Map(l_a + i * s_a) = (l_b + i * s_b) \in Map(SD)$ for any integer i, then processor S sends $B(l_b + i * s_b)$ to D. Thus the intersection of these two sets, SS and Map(SD), is the SendSet.

$$SendSet = SS \cap Map(SD) \quad (7)$$

```
Sending Algorithm
{
   $SendSet = 0$;
   For $j = 0$ to $n_2 - 1$
      For $i = 0$ to $n_1 - 1$
         $SendSet = SendSet \cap (SS \cap Map(SD))$
      Endfor
   Endfor
}
```

Figure 2. Algorithm for sending processor

The algorithm for computing the indices of the array elements to send is in Fig. 2.

To determine the *RecvSet* from processor S to processor D, the indices of array B that processor D receives from processor S is following :

$$RecvSet = SD \cap Map^{-1}(SS) \quad (8)$$

In eq (8), for any integer i, if $Map^{-1}(l_b + i * s_b) = (l_a + i * s_a) \in Map^{-1}(SS)$, then processor D receives $B(l_b + i * s_b)$ from processor S. The algorithm for computing the indices of the array elements to send is in Fig. 3.

```
Receiving Algorithm
{
   $RecvSet = 0$;
   For $i = 0$ to $n_1 - 1$
      For $j = 0$ to $n_2 - 1$
         $RecvSet = RecvSet \cap (SD \cap Map^{-1}(SS))$
      Endfor
   Endfor
}
```

Figure 3. Algorithm for receiving processor

After processor S and D finish the sending and receiving algorithm, processor D has element of Array A and B needed to execute computation. Thus processor D can execute the computation.

4. Examples

We show the example of four-point relaxation program. Fig. 4 is the HPF program. The array A and B are distributed onto 4 processors specified in the directives. A processor p needs to determine elements of array B owned by processor q but needed by p.

```
        Real a(99,99), b(99,99)
!HPF$   DISTRIBUTE (:, BLOCK) :: a, b

        do j=2, 98
           do i=2, 98
              a(i,j) = 0.25*(b(i-1,j) + b(i+1,j)
                       +b(i,j-1) + b(i,j+1))
```

Figure 4. Four-point relaxation HPF program

In Fig. 5, we show the code for sending and receiving algorithm. In Fig. 5, i1 and i2 are index of first computation used by a processor, and j1 and j2 are index

```
p=mypid()
i1=ceil(max(p*25,0)/1)
j1=floor(min(24+p*25,97)/1)
do q=(1+i1*1)/25, (1+j1*1)/25
   SendSet = 0
   i2=ceil(max(q*25,0)/1)
   j2=floor(min(23+q*25,97)/1)
   do r=max(i1,i2)*1-p*25, min(j1,j2)*1-p*25,1
      $SendSet = SendSet \cap A(0{:}99, r)$
   enddo
   send SendSet from processor q to processor p
enddo
i2=ceil(max(p*25,0)/1)
j2=floor(min(23+p*25,97)/1)
do q=i2*1/25, j2*1/25
   RecvSet = 0
   i1=ceil(max(q*25,0)/1)
   j1=floor(min(24+q*25,97)/1)
  do r=1+max(i1,i2)*1-p*25, 1+min(j1,j2)*1-p*25,1
      $RecvSet = RecvSet \cap A(0{:}99, r)$
  enddo
  receive RecvSet from processor q to processor p
enddo

i1=ceil(max(2+p*25,0)/1)
j1=floor(min(22+p*25,97)/1)
do q=(1+i1*1)/25, (1+j1*1)/25
   SendSet = 0
   i2=ceil(max(1+q*25,0)/1)
   j2=floor(min(23+q*25,97)/1)
  do r=2+max(i1,i2)*1-p*25, 2+min(j1,j2)*1-p*25,1
      $SendSet = SendSet \cap A(0{:}99, r)$
  enddo
  send SendSet from processor q to processor p
enddo
i2=ceil(max(1+p*25,0)/1)
j2=floor(min(23+p*25,97)/1)
do q=(2+i2*1)/25, (2+j2*1)/25
   RecvSet = 0
   i1=ceil(max(2+q*25,0)/1)
   j1=floor(min(22+q*25,97)/1)
  do r=1+max(i1,i2)*1-p*25, 1+min(j1,j2)*1-p*25,1
     $RecvSet = RecvSet \cap A(0{:}99, r)$
  enddo
  receive RecvSet from processor q to processor p
enddo
```

Figure 5. Example of send/receive algorithm

of last computation used by a processor p and q respectively. In Fig. 5, if the processor number, p, is greater than 0, it sends B(1:98, 0) to processor whose pid is p-1, and receives B(0:98, 0) from processor whose pid is p-1. If the processor is less than 3, it sends B(1:98, 24) to processor whose pid is p+1, and receives B(0:98, 24) receives from processor whose pid is p+1.

5. Conclusions

Data parallel language, such as Fortran D and HPF, makes it much easier for scientist to program for distributed memory system. A programmer develops a parallel program in a global address space, and then the compiler translates the global address space into local address space for each processor's local memory and insert message-passing primitives. Therefore, many users develop the efficient parallel program with only minimal user effort.

In this paper, we describe the algorithm to determine the communication set owned by a processor but used by other processor. We will implement this algorithm in PPTran which is a compiler for HPF on CRAY T3D. PPTran is a source-to-source translator. Therefore, PPTran translates data parallel program written in HPF into SPMD program using PVM message-passing primitives on CRAY T3D.

References

[Gup94] S. K. S. Gupta, S. D. Kaushik, C. H. Hiang, and P. Sadayappan, *On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machine*, Technical Report 19, Ohio State University, 1994.

[HPF93] High Performance Fortran Forum, *High Performance Fortran Language Specification Version 1.0*, Tech. Report CRPC-TR92225, Rice University, 1993.

[Moo95] K. D. Moon, T. G. Kim, N. J. Ban, J. K. Kim and Y. B. Yoo, *A Study on HPF Parallel Programming Translator Using PVM*, Proc. of Parallel Processing System, Vol. 6, No. 1, May 1995.

[Phi94] Philip J. H., *The Impact of High Performance Fortran*, IEEE Parallel & Distributed Technology, Vol. 2. No. 3, Fall 1994.

[Sti94] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross, *Generating Communication for Array Statements : Design, Implementation, and Evaluation,* Journal of Parallel and Distributed Computing, Vol. 21, No. 1, pp. 150-159, 1994.

[Tse93] C. Tseng, *An Optimizing Fortran D Compiler for MIMD Distributed Memory Machine*, Ph.D. Thesis, Dept. of Computer Science, Rice University, 1993.