Generating Realignment-Based Communication for HPF Programs

Tsunehiko Kamachi, Kazuhiro Kusano, Kenji Suehiro, Yoshiki Seo
C&C Research Laboratories, NEC Corporation
4-1-1 Miyazaki, Miyamae-ku, Kawasaki, Kanagawa 216 JAPAN
{kamachi, kusano, suehiro, seo}@csl.cl.nec.co.jp

Masanori Tamura, Shoichi Sakon
1st Computers Software Division, NEC Corporation
1-10 Nissincho, Fuchu, Tokyo 183 JAPAN
{tmr, sakon}@lang1 bs1 fc.nec.co.jp

Abstract

This paper presents methods for generating communication on compiling HPF programs for distributed-memory machines. We introduce the concept of an iteration template corresponding to an iteration space. Our HPF compiler performs the loop iteration mapping through the two-level mapping of the iteration template in the same way as the data mapping is performed in HPF. Making use of this unified mapping model of the data and the loops, communication for nonlocal accesses is handled based on data-realignment between the user-declared alignment and the optimal alignment, which ensures that only local accesses occur inside the loop. This strategy results in effective means of dealing with communication for arrays with undefined mapping, a simple manner for generating communication, and high portability of the HPF compiler. Experimental results on the NEC Cenju-3 distributed-memory machine demonstrate the effectiveness of our approach: the execution time of the compiler-generated program was within 10% of that of the hand-parallelized program.

1. Introduction

Over the past several years, a great deal of effort has been devoted to developing data parallel languages and their compilers[8, 11, 12]. Based on these efforts, High Performance Fortran (HPF)[6] was proposed and is expected to become the first standard data parallel language. HPF allows programmers to explicitly express data parallelism by a set of compiler directives. An HPF compiler has to determine how computation should be assigned to processors, and generate appropriate communications based on the data mapping specified by the directives.

This paper focuses on generating communications for nonlocal accesses during regular loop computation in the compilation process of HPF programs. We present an efficient method to generate appropriate communication, making use of an unified mapping model of data and loops. In our HPF compiler, both the mapping of the data and the loops are represented by the two-level mapping model of alignment and distribution provided by HPF. The basic idea is that required communication can be regarded as remapping to be performed before or after the loop execution.

This method is flexible enough to deal with communication whether the data mapping is defined or not at compile-time, while existing compilers for distributed-memory machines[1, 2, 3, 4, 11] can not handle communication efficiently in the case where the data mapping is undefined. Furthermore, the portability of the HPF compiler is enhanced, because communication patterns based on the remapping result in high-level interfaces of collective communication primitives.

In generating communication, our HPF compiler first derives the optimal data mapping, which ensures that only local accesses occur inside the loop, from a comparison between the mappings of the data and the loops, and then regards the communication as the remapping between the optimal mapping and the user-declared mapping specified by the HPF directives. As a matter of fact, the optimal mapping can be defined by the alignment, regardless of the distribution, and the remapping therefore results in the realignment changing the alignment relationship. The realignment patterns are then specialized for several regular remapping patterns by means of distribution information if it is defined at compile-time. As compared with the realignment patterns, the regular remapping patterns are expected to reduce run-time overhead of the communication primitives corresponding to the patterns by exploiting their regularity.
2. Mapping and Communication Model

2.1. Data and Iteration Mapping Model

HPF employs a set of directives to allow users to advise the compiler about how array elements should be mapped onto distributed memories. The data mapping is accomplished through the two-level mapping of alignment and distribution onto abstract processors defined by users.

In our HPF compiler, both the data mapping and the iteration mapping are handled under the two-level mapping model as shown in Fig. 1. To perform the iteration mapping, we introduce the concept of an iteration template, which can be thought of as a virtual array with the same shape of a loop iteration space. The iteration mapping is accomplished by aligning or distributing the iteration template. Consider the following example:

\[
\begin{align*}
&\text{DO I=1,M} \\
&\text{DO J=1,N} \\
&\quad A(I,J) = B(I,J)
\end{align*}
\]

The outer loop has M iterations and the inner loop has N iterations; therefore, the iteration space for the double-nested DO loop has the size \(M \times N\). In the above example, the iteration template \(L\) of the DO loop can be expressed as \(L(M,N)\) like an array. Our HPF compiler attempts to align the iteration template \(L\) with some array which appears in the loop to accomplish the iteration mapping based on the owner computes rule[11]. \(L\) is aligned with \(A\) with the statement “ALIGN \(L(I,J)\) WITH \(A(I,J)\),” and \(L\) is then distributed in the same manner of \(A\). As a result, the iteration and the element of \(A\) accessed by the iteration are mapped onto the same processor.

Utilizing this unified mapping model, the HPF compiler can deal with both the data and the iteration mapping under the same framework, and easily generate communication necessary for the loop execution.

2.2. Communication Model

Our HPF compiler regards communication that is required for the regular loop computation as a data remapping to be carried out between the user-declared mapping and the optimal mapping, which ensures that only local accesses occur during the loop execution. Both the point where the remapping is to be performed and the remapping order depend on whether the target array appears on the lhs (left-hand side) or the rhs (right-hand side) of the statements inside the loop as follows:

- **lhs**: The array should be remapped from the user-declared mapping into the optimal mapping after the loop execution.
- **rhs**: The array should be remapped from the optimal mapping into the user-declared mapping after the loop execution.

The optimal data mapping is the one in which both an iteration and an array element accessed by the iteration are mapped onto the same element of the array (or the template) with which they are ultimately aligned. It is possible to ensure that elements in different arrays are mapped onto the same processor’s memory by only their alignment relationship, regardless of the distribution. Therefore, it can be said that finding communication patterns is equivalent to deriving alignment patterns of the optimal data mappings.

Consider the following example:

\[
!\text{HPF$ ALIGN B(I) WITH A(I) DO I = 1, N-1} \\
\quad A(I) = B(I+1)
\]

When the owner computes rule is applied in the iteration mapping, the optimal mapping of \(B\) is defined as the alignment of “ALIGN \(B(I)\) WITH \(A(I-1)\).” That is, the required communication for \(B\) is equivalent to the realignment that changes the alignment of “ALIGN \(B(I)\) WITH \(A(I)\)” into “ALIGN \(B(I)\) WITH \(A(I-1)\)” in advance of the loop execution. This communication model provides the following advantages:

1. The HPF compiler can find required communication patterns even if the mapping of the target array is undefined at compile-time.
2. Communication primitives used for the loop execution can share the major functions with run-time primitives for the remapping directives in HPF.
3. The derived communication patterns result in high-level interfaces of collective communication primitives; consequently, the portability of the HPF compiler is enhanced.

In particular, the capability of finding communication for arrays with undefined mapping is very significant, since there are many cases in which the mapping of the array can not be determined at compile-time. For example, if a dummy argument has the INHERIT attribute, the mapping of the dummy argument is undefined at compile-time.

3. Representation of Mapping Information

Since our HPF compiler deals with the iteration mapping in the same way as the data mapping, and generates communication in terms of the realignment patterns, it is important to develop a data structure representing the two-level mapping features efficiently.

We provide the Data Mapping Descriptor (DMD), an integrated representation of alignment and distribution characteristics. The DMD is capable of simply representing three types of mapping: a distributed mapping, a collapsed mapping, and a replicated mapping[6]. Mapping information is also maintained in run-time DMDs during the execution of the program (see subsection 5.3.). Table1 lists information kept in DMDs.

<table>
<thead>
<tr>
<th>kind</th>
<th>member</th>
<th>information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obj-Info</td>
<td>ultimate-target</td>
<td>ultimate align-target of an array</td>
</tr>
<tr>
<td>proc</td>
<td>dist-target</td>
<td>distribution target of the ultimate-target</td>
</tr>
<tr>
<td>Dim-Info of ith dimension</td>
<td>align-dim</td>
<td>aligned dimension on the ultimate-target</td>
</tr>
<tr>
<td></td>
<td>align-range</td>
<td>aligned range on the ultimate-target</td>
</tr>
<tr>
<td></td>
<td>proc-dim</td>
<td>mapped dimension on the proc</td>
</tr>
<tr>
<td></td>
<td>dist-type</td>
<td>type of distribution</td>
</tr>
<tr>
<td></td>
<td>dist-size</td>
<td>size of distributed block</td>
</tr>
<tr>
<td></td>
<td>replication-type</td>
<td>type of replication</td>
</tr>
</tbody>
</table>

The DMD consists of Obj-Info and Dim-Info. Two kinds of symbolic information are stored in the Obj-Info: ultimate-target and proc. The ultimate-target is an array or a template with which the array is ultimately aligned. The proc is the name of the abstract processor arrangement on which the ultimate-target is distributed.

Mapping information of each dimension is stored in the Dim-Info including two kinds of information: alignment information of align-dim and align-range, and distribution information of proc-dim, dist-type, dist-size, and replication-type. The aligned range on the ultimate-target is represented by a regular section, that is, align-range with three parameters: l, u, and s. They stand for lower, upper, and stride on the ultimate-target. Each item in the Obj-Info and the Dim-Info of the ith dimension is referred to as DM D(member) and DM Di(member), respectively, throughout this paper. Consider the following example:

```
DIMENSION A(M), B(M+1), C(2*M+2)
!HPF$ ALIGN A(I) WITH B(I+1)
!HPF$ ALIGN B(I) WITH C(2*I)
```

A, B, and C are ultimately aligned with C (C is aligned with itself). The DMD for A represents that the first dimension of A is aligned to the first dimension of C by align-range [4:2*(M+1):2] as follows:

```
DM D(ultimate-target) = C
DM D1(align-range) = [L:U:S] = [4:2*(M+1):2]
```

The distributed mapping and the collapsed mapping are identified by the align-dim and the proc-dim. If HPF directives specify a replicated mapping, the replication-type discriminates the type of replication.

The mapping characteristics of the iteration template is also summarized in Loop Mapping Descriptors (LMDs) created for a group of perfectly-nested loops. The LMD is almost the same as the DMD except that it has a loop induction variable, the value range of the induction variable, and the size of the iteration space in each Dim-Info.

4. Identification of Communication Patterns

Given the DMD, the LMD, and data access information, our HPF compiler identifies communication patterns through the following steps:

1. Derive the optimal alignment and decide the realignment order between the user-declared alignment and the compiler-specified optimal alignment.

2. Apply the special remapping pattern using distribution information, if the information is defined at compile-time.

Data access information is the collection of information for each dimension and is composed of an access range, an induction variable, and a subscript expression. The access range is expressed using a regular section descriptor[5].

4.1. Deriving Optimal Alignment

In HPF, the alignment characteristics of an array are defined by the following three items presented in the DMD:

1. ultimate-target: an array or a template with which the array is ultimately aligned.

2. align-dim: aligned dimension on the ultimate-target.
3. **align-range**: aligned range on the *ultimate-target*.

Taking account of these items, an alignment relationship which does not induce remote data accesses during the loop execution must satisfy the following conditions:

1. The array should be aligned with the *ultimate-target* that the LMD has.

2. Each dimension of the array should be aligned to the same *align-dim* that the Dim-Info of the LMD, associated with the dimension, is aligned to.

3. The array should be aligned by the *align-range* which ensures that an iteration and an element of the array accessed by the iteration are aligned to the same element of the *ultimate-target*.

If we let \( DMD_{\text{opt}} \) be the data mapping descriptor of the optimal mapping, the above conditions are represented as follows (\( g(i) \) gives the dimension of the array associated with the \( i \)th Dim-Info of the LMD):

1. \( DMD_{\text{opt}}(\text{ultimate-target}) = LMD(\text{ultimate-target}) \)
2. \( DMD_{\text{opt}}(g(i)) = LMD_{i}(\text{align-dim}) \)
3. \( DMD_{\text{opt}}(g(i)) = [l_{\text{opt}}, u_{\text{opt}}, s_{\text{opt}}] \)

The parameters of the optimal *align-range* i.e., \([l_{\text{opt}}, u_{\text{opt}}, s_{\text{opt}}] \), can be obtained from the following calculation. Let \( \mathbf{A} \) be a one-dimensional array aligned by the *align-range* \([l:u:s] \). Then the aligned position on the *ultimate-target* of \( \mathbf{A} : [\mathbf{l}] \) is given by the following function \( F(LB \text{ and } UB) \) are the lower and upper bounds of \( \mathbf{A} \):

\[
F(i) = (i - LB) \times s + l \quad \{ i \mid LB \leq i \leq UB \}
\]

\[ LMD_{i}(\text{align-range}) \text{ expressed as } [l_{p}, u_{p}, s_{p}], \text{ the access range } [l_{ac}, u_{ac}, s_{ac}], \text{ and the optimal } \text{align-range} \text{ should satisfy the following relationships:}
\]

\[
l_{tp} = F(l_{ac}) = (l_{ac} - LB) \times s_{\text{opt}} + l_{\text{opt}},
\]

\[
u_{tp} = F(u_{ac}) = (u_{ac} - LB) \times s_{\text{opt}} + l_{\text{opt}},
\]

\[
s_{tp} = s_{ac} \times s_{\text{opt}}.
\]

Consequently, the optimal *align-range* is derived from the following expressions:

\[
s_{\text{opt}} = s_{tp} / s_{ac} \quad (1)
\]

\[
l_{\text{opt}} = l_{tp} - (l_{ac} - LB) \times s_{\text{opt}} \quad (2)
\]

\[
u_{\text{opt}} = (UB - LB) \times s_{\text{opt}} + l_{\text{opt}} \quad (3)
\]

Consider the example presented in Fig. 2-(a). We assume in this example that the iteration template \( \mathbf{L} \) is ultimately aligned with \( \mathbf{A} \) by the *align-range* \([2:16:2] \) as shown in Fig. 2-(b). In this case, the access range of \( \mathbf{B} \) is \([1:15:2] \) and the user-declared *align-range* is \([1:16:1] \); therefore, \( \mathbf{B} \) needs realignment from \( \mathbf{A} \) to \( \mathbf{L} \) in advance of the loop execution by the optimal *align-range* \([2:17:1] \) calculated by the expressions (1)-(3) as illustrated in Fig. 2-(c).

### 4.2 Applying Special Remapping Patterns

After the realignment patterns are derived, the HPF compiler attempts to change them into special remapping patterns, utilizing distribution information in DMDs when it is available. As compared with the realignment patterns, the special remapping patterns realize more efficient communications. Furthermore, it is possible to apply optimizations to reduce communication overhead by exploiting regularity of the patterns. Currently, our HPF compiler recognizes the following special remapping patterns:
- **alignment shift**: shifting the align-range when distribution information of the optimal DMD is identical to that of the original DMD.

- **replicate**: changing a distributed mapping into a replicated mapping which makes a copy of an array along a specific dimension.

- **redistribute**: changing distribution when only distribution information is different between the optimal DMD and the original DMD.

- **gather**: gathering distributed elements on specific processors. This is a kind of the replicate pattern.

- **scatter**: scattering array elements replicated on specific processors over processors which own them.

## 5. Communication Placement

### 5.1 Communication Primitives

Currently, two versions of the communication primitives, corresponding to the remapping patterns described in the preceding section, are implemented, that is, they support both MPI[10] and the NEC Cenju-3 native communication primitives[9]. The following examples demonstrate how calls to the primitives are generated for three major communication primitives: the LREALIGN, the SHIFT, and the REPLICATE primitives.

The HPF compiler passes the access range for each dimension through the array argument range to the primitives so that remapping is performed within the access range. The compiler also passes the runtime managed DMDs (referred to as DMD_of_XXX in the examples) so that the primitives can obtain data mapping information related to the array.

**LREALIGN(Loop REALIGN)** Figure 3 shows an example of a compiler-generated program. The LREALIGN primitive corresponding to the general realignment pattern is used for communication of \( B \), because the mapping of \( B \) in the example is unknown at compile-time. The HPF compiler sets the optimal alignment pattern in the array argument opt_mapping. The LOOP_BOUND calculates boundaries of the parallelized loop for each processor according to the mapping information of the iteration template.

**SHIFT** There are two types of shift primitives: the PRE_SHIFT, for arrays appearing on the rhs, is placed before a loop and the POST_SHIFT, for arrays on the lhs, is placed after a loop as follows:

```fortran
SUBROUTINE FOO( B )
REAL A(100,100), B(100,100)
!HPF$ INHERIT B
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)
  call LREALIGN(DMD_of_B,opt_mapping,range)
  call LOOP_BOUND(DMD_of_A,start1,end1,..)
  call LOOP_BOUND(DMD_of_A,start2,end2,..)
  DO I = start1, end1
    DO J = start2, end2
      A(I,J) = B(J,I)
    END DO
  END DO
SUBROUTINE FOO( B )
```

Both primitives transfer array elements to the neighbor processors across the distribution boundaries. In the alignment shift pattern, however, there are some cases where the dist-size determines whether data movement is occurred or not (shown in Fig. 4). If the dist-size is not constant at compile-time, the SHIFT takes care of the communications.

**Figure 3. Example of LREALIGN.**

**REPLICATE** This primitive, corresponding to the replicate pattern, performs the multicast operation among processors belonging to a specific processor array dimension with limited the range specified in range.

```fortran
SUBROUTINE FOO( B )
REAL A(100,100), B(100,100)
!HPF$ INHERIT B
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)
  call LREALIGN(DMD_of_B,opt_mapping,range)
  call LOOP_BOUND(DMD_of_A,start1,end1,..)
  call LOOP_BOUND(DMD_of_A,start2,end2,..)
  DO I = start1, end1
    DO J = start2, end2
      A(I,J) = B(J,I)
    END DO
  END DO
```

**Figure 4. Alignment shift communication.**

### 5.2 Multidimensional Communication

If an array has a multidimensional distribution, a single reference of the array may require communication in each distributed dimension, such as SHIFT or REPLICATE. Our HPF compiler deals with this multidimensional communication by generating calls to the communication primitives for each dimension. This strategy makes it possible to avoid implementing communication primitives for each
multidimensional communication pattern. The following is the example of the two-dimensional SHIFT.

\[
\text{!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: A,B} \\
\text{DO } I = 1+p,N \\
\text{DO } J = 1+q,N \\
\text{A(I,J) = B(I-p,J-q)}
\]

The reference of \( B \) requires shift communication for both the first and the second dimensions. In this case, dimension-wise communication is insufficient to complete the whole communication in a straightforward manner for the darkly shaded area shown in Fig. 5-(a). In our implementation, however, it is possible to accomplish the entire communication by using only the SHIFT primitive as illustrated in Fig. 5-(b). This implementation results in a small extra data movement, because a single slantwise communication is divided into two dimension-wise steps, but the total number of communication steps is nonetheless reduced. The total performance is therefore improved over that obtained by the straightforward communication because the startup cost in most distributed-memory machines is much greater than the cost of this small data movement.

Let \( N \) be the number of distributed dimensions requiring shift communication and \( T \) be the number of communication steps, \( T = N \) in the multidimensional communication, but \( T = 2^N - 1 \) in the straightforward communication. This indicates that as \( N \) is increased, the multidimensional SHIFT results in increasingly better performance. Our HPF compiler places two calls to the PRE_SHIFT primitive in dimensional order to realize the two-dimensional SHIFT.

Figure 5. Multidimensional SHIFT\((N=2)\).

5.3. Run-time Support

The communication patterns identified by our HPF compiler indicate data movement among abstract processors. Each communication primitive maps abstract processors onto physical processors using mapping information kept in run-time DMDs, and manages data movement over physical processors. The run-time DMDs containing data mapping information have the same format as compile-time DMDs described in Section 3. The run-time DMDs are initialized using compile-time DMDs information, and are then maintained throughout the execution of the program. The run-time DMDs are passed through subroutine boundaries using an extended argument. At the entry point of the subroutine, remapping from a DMD for an actual argument into a DMD for a dummy argument is performed if there are differences between two DMDs. If required, remapping is also performed at the exit point in reverse order.

6. Optimizations

Several compiler optimizations to reduce communication overhead have been introduced in previous works[1, 7]. A set of optimizations are also implemented in both our HPF compiler and the communication primitives.

6.1. Compile-time Optimizations

Message coalescing The compiler takes the union of access ranges of different references to the same array. This optimization avoids redundant communication for the same element and reduces communication overhead by coalescing individual accesses into a contiguous access.

Message aggregation Noncontiguous array elements to be transferred between the same pair of processors are aggregated. Two kinds of message aggregation, namely, self aggregation for the same array access and group aggregation for different array accesses[1], are implemented. Both are performed in the communication primitives, but group aggregation needs the compiler’s support. The compiler discovers a set of arrays which can be communicated in the same primitive call, that is, they have the same mapping and the same access pattern, using the DMD and access information. This set is passed to the communication primitives.

Eliminating unnecessary communication If scalar variables are updated inside the parallelized loop, each processor may have different values at the end of the loop. Because scalar variables are replicated over all processors in our implementation, the processor that has the ultimately updated value should broadcast the value in order to maintain coherence of the variable. However, this communication can be eliminated if the assignment statement is loop-invariant or the variable is dead at the end of the loop.

6.2. Optimizations in the Primitives

Applying special remapping pattern The HPF compiler generates the LREALIGN primitive call if the mapping of
the array is undefined. However, the \texttt{LREALIGN} primitive applies the special remapping patterns in the same way as the compile-time analysis, by comparing the run-time DMD and the compiler-specified DMD information passed through the argument \texttt{opt\_mapping} (See Section 5.), and then calls the communication primitives of the patterns.

\textbf{Message aggregation} All the communication primitives employ message aggregation. The primitives first perform self aggregation, then apply group aggregation if there are multiple arrays to be aggregated.

\section{7. Evaluation Results}

This section reports the results of experimentally evaluating the performance of the communication primitives and three benchmark programs. The communication primitives used in this experiment are the MPI version. All results were measured on the NEC Cenju-3 with 64 PEs (processing elements) connected via a multistage interconnection network[9]. Each PE consists of a 75MHz VR4400 RISC microprocessor and 64MB of memory.

\subsection{7.1. Performance of Communication Primitives}

We measured the performance of the \texttt{SHIFT} primitive, which is one of the most important primitives because it is observed frequently in regular loop computation.

\textbf{Hand-written vs. \texttt{SHIFT} Primitive} Figure 6-(a) shows the results of comparing the performance of hand-written shift communication and that of the \texttt{SHIFT} primitive using various sizes of the \texttt{NxN} array declared by \texttt{real*4}. The first dimension of the array was distributed by \texttt{BLOCK}, and was shifted with the shift-width of 1; therefore, N elements were moved between two PEs. Although the \texttt{SHIFT} primitive refers to run-time managed DMDs in order to obtain data mapping information and carries out error checking, it provided acceptable performance compared with the hand-written version.

\textbf{Effectiveness of Aggregation} Figure 6-(b) and Fig. 6-(c) show the results of self aggregation and group aggregation, respectively, applied to the \texttt{SHIFT} primitive. Self aggregation was applied under the same conditions in the preceding experiment, and group aggregation was performed for three arrays which have the \texttt{BLOCK} distribution in the second dimension. The results include the \texttt{memcpy} overhead necessary for aggregation. Even with the overhead, both aggregation, especially self aggregation, achieve good performance compared with results of the \texttt{SHIFT} primitive with no aggregation applied (referred to in the Fig. 6 as “Non-Aggregation”).

\textbf{Multidimensional \texttt{SHIFT}} We also compared the performance of the straightforward shift communication with that of the multi-dimensional \texttt{SHIFT} described in subsection 5.2. This experiment was carried out under the case that “p=q=1” in Fig. 5. The result shown in Fig. 6-(d) tells that the multidimensional \texttt{SHIFT} is, as expected, superior to the straightforward shift communication.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{shift.png}
\caption{Performance evaluation of \texttt{SHIFT}.}
\end{figure}

\subsection{7.2. Benchmark Results}

To illustrate the effect of our approach, we used the three benchmarks: the weather-prediction program \texttt{shallow} from NCAR, the mesh generation program \texttt{tomcatv} from the SPEC, and the partial differential equation program \texttt{pde1} from the GENESIS benchmark suite.

\textbf{Comparison with Hand-parallelized Code} Using \texttt{shallow}, we compared the performance of the hand-parallelized code with that of the parallelized code generated by our HPF compiler. As shown in Fig. 7-(a), the compiler-generated code achieves good performance close to hand-parallelized code: the execution time of the compiler-generated code was within 10% that of the hand-parallelized code.

\textbf{Evaluation of Optimizations} Figure 7-(b) and (c) illustrate effectiveness of message aggregation in the \texttt{SHIFT} primitive, when self aggregation and group aggregation were applied to \texttt{pde1} and \texttt{tomcatv}, respectively. In \texttt{pde1}, self aggregation results in a 1.2 times speedup as compared to using the primitive without the optimization. In \texttt{tomcatv}, there were two arrays which could be aggregated in the same the \texttt{SHIFT} primitive call. Compared with the result of \texttt{pde1}, the optimization has less effect for \texttt{tomcatv}. However, we can expect to achieve high performance using
8. Conclusions

This paper has presented the methodology of communication generation needed for regular loop computation incorporated into our HPF compiler. We introduced the concept of the iteration template to carry out the iteration mapping by the two-level mapping of the iteration template in the same way that data mapping is performed in HPF. Our HPF compiler generates realignment-based communication, taking advantage of this unified mapping model of the data and the iteration template. The major advantage of this strategy is that the HPF compiler can deal with communication efficiently even when the distribution is undefined at compile-time. The results of evaluating our approach experimentally show its effectiveness.

9 Acknowledgements

We would like to thank H. Katayama and T. Nakata for their helpful suggestions. We are also grateful to Y. Hayashi, Y. Suzuki, Y. Watanabe, and Y. Shiroto for many valuable discussions and their contribution to the development of the HPF compiler.

References


