Implementing Flexible Computation Rules with Subexpression-level Loop Transformations

Dattatraya Kulkarni*, Michael Stumm*, and Ronald C. Unrau**

*Department of Computer Science and
Department of Electrical & Computer Engineering
University of Toronto, Toronto, Canada, M5S 1A4
Email: kulki@cs.toronto.edu

**Parallel Compiler Development
IBM Toronto Laboratory
Toronto, Canada, M3C 1V7

Abstract. Computation Decomposition and Alignment (CDA) is a new loop transformation framework that extends the linear loop transformation framework and the more recently proposed Computation Alignment frameworks by linearly transforming computations at the granularity of subexpressions. It can be applied to achieve a number of optimization objectives, including the removal of data alignment constraints, the elimination of ownership tests, the reduction of cache conflicts, and improvements in data access locality.
In this paper we show how CDA can be used to effectively implement flexible computation rules with the objective of minimizing communication and, whenever possible, eliminating intrinsics that test whether computations need to be executed or not. We describe CDA, show how it can be used to implement flexible computation rules, and present an algorithm for deriving appropriate CDA transformations.

1 Introduction

In a SPMD framework such as HPF [7], data alignments and distributions are usually specified by the user or suggested by some automatic tool such as PARADIGM [8]. Given the data alignments and distributions, the compiler then maps computations to processors using a computation rule. The choice of computation rule can have a significant impact on performance, since it affects the amount of communication generated and the number of intrinsics needed in the code.

Traditionally, computation rules have been *fixed* in that they do not take alignments and distributions of all references into account. For example, the owner-computes rule is a fixed rule and is used almost exclusively. It maps a statement instance to the processor which owns the *lhs* (left hand side) data element of the statement [7], even if it would be more efficient to compute the statement on another processor, and it always maps a statement instance in its entirety. Fixed computation rules provide a general schema for computation mapping and hence simplify code generation, especially the insertion of communication code.

In contrast, *flexible* computation rules take into account the location of all the data needed for the computation and the cost of communication when deciding where a computation is to be executed [5]. The granularity of the computation being mapped is usually at the subexpression level. It is therefore possible to achieve optimal or near optimal computation mappings. However, the code generation is much more complex. Flexible computation rules are also important on shared memory multiprocessors. For example, it is important to minimize remote memory accesses on shared memory multiprocessors with non-uniform access times. Moreover, the location of computations can significantly affect cache locality and interference patterns.

In this paper, we show how the recently proposed CDA loop transformation framework [13, 14] can be used to efficiently implement a flexible computation rule called *P-Computes*. Section 2 briefly reviews the most important related work. Section 3 introduces the P-Computes rule. We show by example how P-Computes can be implemented with CDA in Section 4 and present the formal algorithms in Section 5.

2 Related Work

There has been much work in developing techniques that improve the performance of SPMD code. Linear loop transformation is a general technique developed in 1990 that changes the execution order of the iterations [4, 15, 17, 23] and can be used, for example, to reduce communication overhead by moving communications to the outer loop levels [17, 23]. However, linear loop transformations are limited in their optimization capabilities, since they leave iterations unchanged as they map the original iteration space onto a new one.

Over the last three years, Computation Alignment (CA) frameworks have been proposed that extend the capabilities of linear transformations [9, 12, 21] by transforming loops at the granularity of statements.[1] By applying a separate transformation to each statement in the loop body they change the execution order of each statement, thus effectively changing the constitution of the iterations. CA transformations have been applied to improve SPMD code in a variety of ways [12, 22]. For example, CA may be used to align the statements in the loop body so that all lhs data elements accessed in an iteration are located on the same processor in the hope of eliminating the need for ownership tests.

Computation Decomposition and Alignment (CDA) [13] is a generalization of CA and goes a step further in that it can transform computations of granularity smaller than a statement. Instead of transforming the statements as written by the programmer, CDA first partitions the original statements into finer statements and then aligns the statements at this finer granularity. This creates

[1] The origins of CA can be traced to loop alignment [1, 19], which is a special case of CA.

additional opportunities for optimization. In later sections we show how CDA can be used to implement flexible computation rules.

A number of optimization techniques have been proposed that reduce communication by transforming data. Bala and Ferrante [3] proposed the insertion of XDP directives that explicitly move data to transfer ownerships. XDP can thus be used to implement variations of owner-computes rule. In this context, dynamic data alignment [11] can be considered as a structured form of implicit data movement. Earlier approaches resorted to static solutions by deriving best data alignments [16] and distributions [2, 8] considering global constraints.

Chatterjee et al. [5] developed algorithms that derive communication optimal flexible computation rules for a class of expressions. They take the machine topology into account to find optimal mappings of subexpressions onto processors in polynomial time.

3 P-Computes Rules

The P-Computes operator, $\otimes$, can be used to specify flexible computation rules. A P-Computes rule, $\otimes_p$ $(expr)$, specifies that the expression $expr$ is to be executed on processor p. Generally, p and $expr$ will be functions of the enclosing loop iterators. If $expr$ does not assign value to a lhs, then the result of executing $\otimes_p(expr)$ is to be sent to the processor designated by the enclosing P-Computes operator.

The specification of p in $\otimes_p(expr)$ can be either *direct* or *indirect*. If the processor is specified as a direct function of the iterators, then the P-Computes rule is direct. Otherwise, if the processor is specified in terms of the location of array elements, then the rule is indirect. The indirect specification can be achieved through an intrinsic similar to `iown(e)` used by owner-computes that evaluates to *true* on the processor that owns data element `e`. Here we use a variant, `owner(e)`, that returns the processor that has data element `e`. Hence, `iown(e)` is equivalent to the conditional (`myid` = `owner(e)`).

The $\otimes$ operator is very general and can be used to express a variety of computation rules. For example:

$$\otimes_{\mathtt{owner}(\mathtt{e}_1)}(\mathtt{e}_1 = \mathtt{e}_2 + \mathtt{e}_3)$$
$$\otimes_{\mathtt{f}(\mathtt{I})}(\mathtt{e}_1 = \otimes_{\mathtt{g}(\mathtt{I})}(\mathtt{e}_2 + \otimes_{\mathtt{h}(\mathtt{I})}(\mathtt{e}_3 + \mathtt{e}_4)))$$
$$\otimes_{\mathtt{owner}(\mathtt{A}(\mathtt{I}))}(\mathtt{e}_1 = \otimes_{\mathtt{owner}(\mathtt{B}(\mathtt{I}))}(\mathtt{e}_2 + \mathtt{e}_3) + \otimes_{\mathtt{owner}(\mathtt{C}(\mathtt{I}))}(\mathtt{e}_4 + \mathtt{e}_5))$$

are all valid P-Computes rules, where the `e`'s are subexpressions. The first example above is equivalent to the application of the owner-computes rule. The second example specifies the processors directly as a function of the loop iterators, whereas the mapping is indirect in the last example.

To compare the difference between a P-Computes rule and the owner-computes, consider the following loop:

```
for i = 1,n
   for j = 1,n
      S1 : ⊗owner(A(i,j)) (A(i,j) =
                  ⊗owner(A(i−1,j)) (A(i − 1,j) + A(i − 1,j − 1) + B(i − 1,j))
                  +B(i,j + 1) + A(i,j − 1))
      S2 : ⊗owner(B(i,j−1)) (B(i,j − 1) = A(i,j − 1) + B(i,j))
   end for
end for
```

Assume that each iteration is mapped onto a different processor and that $\mathtt{B(i,j)}$ is aligned to $\mathtt{A(i,j)}$, perhaps due to constraints from a previous loop. If the owner-computes rule were applied directly, then six non-local accesses would be necessary. In contrast, the specified P-Computes rule requires five non-local accesses, one of which is due to sending the result from $\otimes_{\mathtt{owner(A(i-1,j))}}$ in statement $\mathtt{S_1}$. Thus, the P-Computes rule can reduce communication by taking the location of data into account. As we will see in Section 4, CDA transformations can be applied to further reduce the number of communications to three.

4 CDA Implementation of P-Computes Rules

To implement a flexible computation rule specified with P-Computes operators, we transform the loop in a three stage process. The first two stages correspond to CDA, as described in [13]. In the first stage, the statements in the loop are decomposed so that statements can be assigned to processors in their entirety. This may require the introduction of new temporary arrays. Second, the (possibly new) set of statements are linearly transformed so as to eliminate (or reduce) the need for intrinsics. Finally, in a third stage, the newly introduced temporary arrays are data aligned to the existing arrays so that the given computation rule can be specified relative to the ownership of the lhs temporaries. In this section we give an overview of how these techniques can be applied, using the example of Section 3.

4.1 Computation Decomposition

Computation Decomposition is the first stage of our method. It decomposes the loop body into statements so that each statement can be mapped in its entirety to a processor. The P-Computes rules specify how the loop is to be decomposed. In a first step, each P-Computes rule that contains more than one statement is split, so that an equivalent P-Computes rule is applied to each statement separately. In a second step, those statements containing more than one P-Computes rule are split into multiple statements so that each new statement has a single $\otimes$ operator. If a statement must be split, then the expression of the embedded P-Computes rule will be elevated to the status of a statement and a temporary array is introduced to accumulate the results of the expression and to pass the result to the statement corresponding to the enclosing $\otimes$ operators. The temporary variables are typically chosen as arrays in order to reduce the number of dependences introduced by the decomposition, allowing for more freedom in the subsequent search for alignments.

Considering the example of Section 3, statement $\mathtt{S_1}$ is decomposed into statements $\mathtt{S_{1.1}}$ and $\mathtt{S_{1.2}}$. $\mathtt{S_{1.1}}$ corresponds to the expression of the P-Computes rule embedded in statement $\mathtt{S_1}$. The result of evaluating that expression is assigned to the temporary `t` so that it can be passed to the remainder of $\mathtt{S_1}$, namely $\mathtt{S_{1.2}}$. The loop body after computation decomposition becomes:

$$
\begin{array}{l}
\mathtt{for\ i = 1,n} \\
\quad \mathtt{for\ j = 1,n} \\
\quad\quad \mathtt{S_{1.1}}: \otimes_{\mathtt{owner(A(i-1,j))}} (\mathtt{t(i,j) = A(i-1,j) + A(i-1,j-1) + B(i-1,j)}) \\
\quad\quad \mathtt{S_{1.2}}: \otimes_{\mathtt{owner(A(i,j))}} (\mathtt{A(i,j) = t(i,j) + B(i,j+1) + A(i,j-1)}) \\
\quad\quad\quad \mathtt{S_2}: \otimes_{\mathtt{owner(B(i,j-1))}} (\mathtt{B(i,j-1) = A(i,j-1) + B(i,j)}) \\
\quad \mathtt{end\ for} \\
\mathtt{end\ for}
\end{array}
$$

There are three things worth noting in this loop. First, each new statement has a single P-Computes rule. Second, each statement is mapped to a different processor, so it is necessary to evaluate intrinsics, resulting in considerable overhead. Stage 2 will attempt to eliminate the need for the intrinsics. Finally, note that $\mathtt{S_{1.1}}$ as is does not use owner-computes. Stage 3 of our process will translate the existing P-Computes rule to an owner-computes rule.

If the programmer does not explicitly specify the P-Computes rule (as in the above example), then the compiler will have to derive it automatically. A part of that process is to decide which subexpressions are to be elevated to the status of statements. The other part is deciding which processors those subexpressions should be mapped onto.

4.2 Computation Alignment

The computation decomposition of Section 4.1 produces a new loop body that can have more statements than the original. We can now employ CA to separately transform each statement of the new loop body in attempting to eliminate the need for intrinsics [12, 22]. Intuitively, the mapping causes a relative movement of the statement instances across iterations. The idea is to move the computations so that those that are mapped to the same processor belong to the same iteration.

Just as there is an iteration space for the loop body, there is a computation space for a statement `S`, $CS(S)$, which is an integer space representing all execution instances of `S` in the loop. CA applies a separate linear transformation to each computation space. That is, if the decomposition produces a loop body with statements $\mathtt{S_1}, \ldots, \mathtt{S_K}$, which have computation spaces $CS(S_1), \ldots, CS(S_K)$, then we can separately transform these computation spaces with linear transformations $T_1, \ldots, T_K$, respectively. Before the alignment, an iteration $(i_1, \ldots, i_n)$

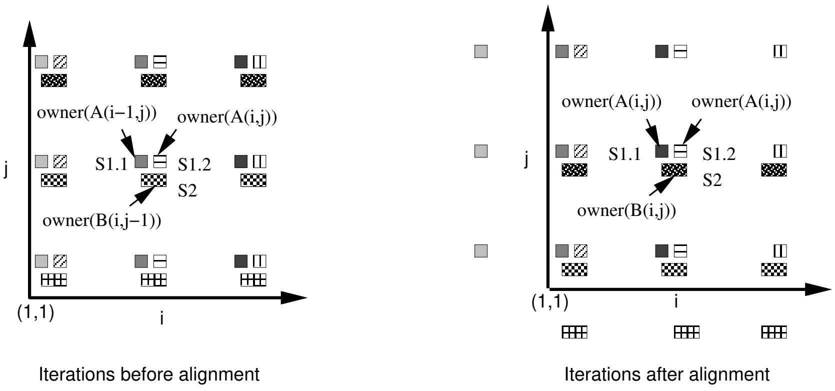

Fig. 1. Movement of computations in the computation space.

consists of computations $\{(i_1,\ldots,i_n;S_1),\ldots,(i_1,\ldots,i_n;S_K)\}$, where $(i_1,\ldots,i_n;S_j)$ is the execution instance of statement $\mathtt{S_j}$ in iteration $(i_1,\ldots,i_n)$. After the alignment, iteration $(i_1,\ldots,i_n)$ consists of computations $\{(T_1^{-1}\cdot(i_1,\ldots,i_n);S_1),\ldots,(T_K^{-1}\cdot(i_1,\ldots,i_n);S_K)\}$.

This type of computation movement at the statement level can be used to redefine the iterations so that all (most) computations in an iteration belong to the same processor. The basic idea in choosing an alignment can be illustrated with a simple example. Suppose we want to align statement $\mathtt{S_1}$ below to statement $\mathtt{S_K}$, where $\mathtt{S_K}$ assigns to the original lhs array, say $\mathtt{A(i,j)}$.

$$\mathtt{S_1} : \otimes_{\mathtt{owner(B(i-c1,j-c2)}} (\mathtt{t_1(i,j)} = \ldots)$$
$$\mathtt{S_K} : \otimes_{\mathtt{owner(A(i,j))}} (\mathtt{A(i,j)} = \mathtt{t_1(i,j)} + \ldots)$$

Also assume that $\mathtt{A(i,j)}$ and $\mathtt{B(i-a1,j-a2)}$ are collocated due to a prior data alignment. We can align $\mathtt{S_1}$ to $\mathtt{S_K}$ by applying a transformation that shifts the $\mathtt{S_1}$ computations by $(\mathtt{c1-a1,c2-a2})$ relative to the computations of $\mathtt{S_K}$. Doing so modifies the statements to become:

$$\mathtt{S_1} : \otimes_{\mathtt{owner(B(i-a1,j-a2)}} (\mathtt{t_1(i+c1-a1,j+c2-a2)} = \ldots)$$
$$\mathtt{S_K} : \otimes_{\mathtt{owner(A(i,j))}} (\mathtt{A(i,j)} = \mathtt{t_1(i,j)} + \ldots)$$

and both statements are now to be executed by the same processor.[2] We can align the statements of the decomposed loop of Section 4.1 in a similar way to eliminate the need for intrinsics. The $\mathtt{S_{1.1}}$ computations are moved along the i direction to bring $(i+1,j;S_{1.1})$ to iteration (i,j). Similarly, the $\mathtt{S_2}$ computations are moved along the j direction so that $(i,j+1;S_2)$ is now executed in iteration (i,j). Figure 1 shows these alignments. The resulting CDA transformed loop is:

[2] The alignment is legal when all dependences between $\mathtt{S_1}$ and $\mathtt{S_2}$ remain positive.

```
for i = 0,n
   for j = 0,n
      S1.2 : (i > 0 ∧ j > 0)
               ⊗owner(A(i,j)) (A(i,j) = t(i,j) + B(i,j + 1) + A(i,j − 1))
      S2 :   (i > 0 ∧ j < n)
               ⊗owner(B(i,j)) (B(i,j) = A(i,j) + B(i,j + 1))
      S1.1 : (i < n ∧ j > 0)
               ⊗owner(A(i,j)) (t(i + 1,j) = A(i,j) + A(i,j − 1) + B(i,j))
   end for
end for
```

and all statement instances in an iteration are now mapped to the same processor so that the intrinsics can be eliminated altogether.

Such a computation alignment changes the references and the dependences in the loop, as well as the loop bounds. If computation space $CS(S)$ is transformed by T, then reference matrix R of each reference r in $\mathtt{S}$ is changed to RT^{-1}. We represent data flow constraints in the loop with *dependence relations* [20], and we keep the exact dependence information between each pair of read and write [6, 18]. Consider a read reference r in statement $\mathtt{S_r}$ flow dependent on a write reference w of statement $\mathtt{S_w}$. The dependence relation $w[d_{wr} \cdot \mathtt{I}] \rightarrow r[\mathtt{I}]$ between the references is changed to $w[d_{wr} \cdot T_w \cdot T_r^{-1} \cdot \mathtt{I}] \rightarrow r[\mathtt{I}]$, when T_w is applied to $CS(S_w)$ and T_r is applied to $CS(S_r)$. The alignment is legal if all new dependence relations are positive.

The new loop bounds are obtained by projecting all computation spaces onto an integer space that becomes the iteration space of the aligned loop. Because each statement can potentially be transformed by a different linear transformation, the new iteration space can be non-convex. There are two basic strategies that can be pursued to generate code. First, it is possible to take the convex hull of the new iteration space and then generate a *perfect* nest that traverses this hull. This is the strategy chosen to generate the above loop, but requires the insertion of *guards* that disable the execution of statements where necessary. A second, alternative strategy is to generate an *imperfect* nest that has no guards. Guard-free code is usually desirable for better performance, but a perfect loop may be desirable in some cases, for instance to avoid non-vector communications or to avoid loop overheads. Algorithms to generate code employing both strategies can be found in the literature [9, 10, 12, 21, 22].

4.3 Aligning the Temporary Arrays

In the third stage, each temporary array is data aligned to the array used by the $\otimes$ operator to specify the processor onto which the computations are to be mapped. The idea is to collocate the lhs reference of a statement (assuming it is to a temporary) and the array reference in the $\otimes$ operator. This allows the P-Computes rule to be interpreted as the owner-computes rule. In our running example, $\mathtt{t(i+1,j)}$ is data aligned to $\mathtt{A(i,j)}$, so that $\mathtt{owner(A(i,j))}$ is equivalent to $\texttt{owner}(\texttt{t}(\texttt{i}+1,\texttt{j}))$. Hence, we implement a flexible computation rule, but can retain the simplicity of owner-computes for code generation.

4.4 Properties of CDA Transformed Loops

Notice that the transformed loop implements the specified P-Computes rule. For instance, subexpression $\texttt{A}(\texttt{i}-1,\texttt{j})+\texttt{A}(\texttt{i}-1,\texttt{j}-1)+\texttt{B}(\texttt{i}-1,\texttt{j})$ is computed on the owner of $\texttt{A}(\texttt{i}-1,\texttt{j})$. The transformation has separated out the complex computation rule into its constituent simpler rules, which happen to be owner-computes here. Moreover, the computation alignment has made the loop efficient by moving the computations relative to each other so that all computations in an iteration are to be computed by the same processor, namely $\texttt{owner}(\texttt{A}(\texttt{i},\texttt{j}))$. The intrinsics can now be eliminated altogether by changing the loop strides. The end result is a loop that requires only three non-local accesses, compared to six in the original (assuming again that each iteration is mapped to a different processor).

One drawback of our approach is that the computation alignment may change the loop independent dependences on the temporaries to become loop carried dependences. This can reduce the degree of available parallelism in the loop.

5 Algorithms

In this section we outline specific algorithms that implement a given P-Computes rule under the assumption that there is one statement in the original loop body. The algorithms correspond to the three stages discussed in Section 4. They can be easily extended to handle the case where the original loop body has multiple statements. We assume that the references are affine functions of iteration vector $\texttt{I}$ and are represented by a reference matrix. The affine function $\texttt{f}$ in a reference $\texttt{A}(\texttt{f}(\texttt{I}))$ is used to denote the reference matrix as well. Data alignments also have a matrix representation, and data alignment of $\texttt{d}_\texttt{j}$ on an array changes each reference r to the array to be $\texttt{d}_\texttt{j} r$.

Algorithm *comp-decomp* decomposes the statement so that each P-Computes operator maps a separate statement. In each iteration of the algorithm, each innermost $\otimes$-subexpression is rewritten as a full statement with a new temporary array element as its lhs, and the $\otimes$-subexpression is replaced by the corresponding reference to the temporary.

Algorithm : *comp-decomp*
Decompose statement $\texttt{S}:\ \otimes_{\texttt{owner}\,(\texttt{A}(\texttt{f}(\texttt{I})))}\,(\texttt{lhs}=\texttt{rhs})$
begin
    $\texttt{i} \leftarrow 1$
    while rhs of $\texttt{S}$ has a P-Computes operator
        Choose an innermost $\otimes_{\texttt{owner}\,(\texttt{B}(\texttt{g}(\texttt{I})))}\,(\texttt{expr})$ in $\texttt{S}$
        $\texttt{t}_\texttt{i} \Leftarrow$ new temporary array
        generate statement $\texttt{S}_\texttt{i}:\ \otimes_{\texttt{owner}\,(\texttt{B}(\texttt{g}(\texttt{I})))}\,(\texttt{t}_\texttt{i}(\texttt{I})=\texttt{expr})$

```
        replace ⊗owner (B(g(I))) (expr) in rhs by t_i(I)
        i ← i + 1
    end while
    K ← i
    generate S_K : ⊗owner (A(f(I))) (lhs = rhs)
end
```

The second stage uses algorithm *comp-align* in attempting to find a computation alignment that aligns the K statements generated by the decomposition stage so that all statement instances in an iteration are mapped to the same processor. The search space of all legal computation alignments is large, and therefore it is not possible to exhaustively search this space in reasonable time. For this reason, we apply a heuristic.

We know that a statement $\mathtt{S_j}$ with P-Computes operator $\otimes_{\mathtt{owner}\,(\mathtt{A_j}(\mathtt{f_j}(\mathtt{I})))}$ can be aligned to statement $\mathtt{S_i}$ with operator $\otimes_{\mathtt{owner}\,(\mathtt{A_i}(\mathtt{f_i}(\mathtt{I})))}$ by applying a transformation $\mathtt{T_j} = \mathtt{f_i}^{-1}\mathtt{d_j}\mathtt{f_j}$ to $\mathtt{S_j}$, where array $\mathtt{A_j}$ is data aligned to array $\mathtt{A_i}$ by transformation $\mathtt{d_j}$. Since there are K statements in the loop, we can construct K computation alignments, $\alpha = \{\alpha_1, \ldots, \alpha_K\}$, with computation alignment α_i aligning each statement of the loop to statement $\mathtt{S_i}$. Some of these computation alignments may be illegal. If there are legal alignments, then the algorithm will choose the one that results in a loop with the minimal communication overhead. If there are no legal alignments, then the algorithm attempts to make an illegal computation alignment legal by discarding individual statement alignments that violate dependences. The algorithm then selects the computation alignment with the fewest discarded statement alignments.

Algorithm : *comp-align*
Given: statements $\mathtt{S_1}; \ldots; \mathtt{S_K}$,
where $\mathtt{S_i}$ has the P-Computes operator $\otimes_{\mathtt{owner}\,(\mathtt{A}(\mathtt{f_i}(\mathtt{I})))}$. Statements $\mathtt{S_1}; \ldots; \mathtt{S_{K-1}}$ have temporary variables on the lhs. $\mathtt{S_K}$ has the same lhs as the original statement.

`begin`

`Step 1`: Construct a set of K computation alignments, α. Each alignment $\alpha_i \in \alpha$ contains K statement alignments, $\alpha_i = \{\mathtt{T_1}, \ldots, \mathtt{T_K}\}$, such that $\mathtt{T_j}$ aligns S_j to S_i. Hence, $\mathtt{T_j} = \mathtt{f_i}^{-1}\mathtt{d_j}\mathtt{f_j}$, where array $\mathtt{A_j}$ is data aligned to $\mathtt{A_i}$ by $\mathtt{d_j}$.

`Step 2`: If α contains no legal computation alignments, then go to Step 3. Otherwise, choose alignment $\alpha_i \in \alpha$ that (1) is legal and (2) results in a transformed loop with lowest communication overhead. Return α_i.

`Step 3`: If none of the K computation alignments in α are legal, then modify each $\alpha_k \in \alpha$ to make it legal:

(i) `While` the alignment α_k is illegal due to a violated flow dependence on a temporary

Choose such a dependence
Assume it is from S_i to S_j
(that is the lhs of S_i is accessed in S_j and $T_i \not\leq T_j$)[3]
Set $T_i \leftarrow T_j$

(*ii*) **While** the alignment α_k is illegal due to a violated dependence on the lhs of S_K
Choose such a dependence
If it is a flow dependence from S_K to S_i
then set $T_i \leftarrow T_j$ where T_j is chosen such that $T_i < T_j$
and there is no other T_r with $T_i < T_r < T_j$
else (it is an anti-dependence from S_i to S_K)
then set $T_i \leftarrow T_j$ where T_j is chosen such that $T_i > T_j$
and there is no other T_r with $T_i > T_r > T_j$

Return the computation alignment with the fewest discarded statement alignments.

end

Finally, algorithm *data-align-temp* aligns the temporaries to existing arrays so that P-Computes rules can be replaced by simple owner-computes rules.

Algorithm : *data-align-temp*
Data Align the temporary arrays introduced in Stage 1.
The alignment chosen in stage two is $\alpha_k = \{T_1, \ldots, T_K\}$.
begin
for each statement $S_i : \otimes_{A_i(f_i(I)T_i^{-1})} (t_i(T_i^{-1}I) = \ldots)$
data align t_i to A_i by f_i
end for
end
As discussed before, this data alignment makes it possible to convert the P-Computes rule for the loop into the familiar owner-computes rule, assuming the P-Computes rule for statement S_K is $\otimes_{\mathtt{owner}\ (A_K(f_K(I)))}$.

6 Concluding Remarks

We have shown how P-Computes, a representative flexible computation rule, can be effectively implemented by using CDA loop transformations. The implementation eliminates intrinsics whenever possible. Since the P-Computes rule is finally translated into the familiar owner-computes rule, the code generation for a given P-Computes is simpler than if the rule were implemented directly.

We have assumed that the P-Computes rule was specified by the programmer so as to minimize communication. Ideally, a compiler should be able to automatically derive the most appropriate P-Computes rules. However, the derivation of

[3] $T_i \leq T_j$ denotes that for all iterations I, T_iI is lexicographically less than or equal to T_jI.

an optimal P-Computes is still an open problem. We believe that it is possible to produce near optimal computation rules by first deriving CDA transformations that minimize the number of distinct references, and then employing existing algorithms that derive flexible computation rules [5]. For example, previous work by Chatterjee et al. shows that subexpressions can be optimally mapped to processors in polynomial time [5]. However, their results have to be extended if CDA transformations are taken into account. If each data element in the statement

$\mathtt{A(i,j)} = \mathtt{A(i-1,j)} + \mathtt{A(i-1,j-1)} + \mathtt{A(i,j-1)} + \mathtt{A(i,j)}$

is mapped to a different processor, then Chatterjee's algorithm would map the execution of this statement to the processor that owns $\mathtt{A(i,j)}$, resulting in three remote accesses. However, the statement can be CDA transformed to:

$\mathtt{t(i+1,j)} = \mathtt{A(i,j)} + \mathtt{A(i,j-1)}$

$\mathtt{A(i,j)} = \mathtt{t(i,j)} + \mathtt{A(i,j-1)} + \mathtt{A(i,j)}$

With the temporary appropriately data aligned, the original statement is now effectively executed on two different processors, namely those that own $\mathtt{A(i-1,j)}$ and $\mathtt{A(i,j)}$. The execution of the transformed statements require only two remote accesses.

CDA is a general subexpression-level transformation framework which we applied here only for SPMD code optimization. CDA can be used in several other optimization contexts, for example to remove data alignment constraints, improve locality, eliminate cache conflicts, or reduce register pressure [13, 14].

Our current work includes the development of efficient algorithms that derive P-Computes rules and an analysis of their complexity. We are also working on more efficient algorithms to eliminate intrinsics that take intermediate alignments into account while constructing partial alignments.

References

1. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, West Germany, January 1987.
2. J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28, June 1993.
3. V. Bala, J. Ferrante, and L. Carter. Explicit data placement (xdp): A methodology for explicit compile-time representation and optimization of data movement. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 28, pages 139–149, San Diego, CA, July 1993.
4. Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.
5. S. Chatterjee, J.R. Gilbert, , R. Schreiber, and S. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM Transactions on Programming Languages and Systems*, 17(1):123–156, January 1995.
6. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

7. HPF Forum. HPF: High performance fortran language specification. Technical report, HPF Forum, 1993.
8. M. Gupta. Automatic data partitioning on distributed memory multicomputers. Technical report, Dept of computer Science, University of Illinois at Urbana Champaign, 1992.
9. W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126, University of Maryland, 1992.
10. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report UMIACS-TR-94-87, University of Maryland, 1994.
11. K. Knobe, J.D. Lucas, and W.J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers, Vienna*, pages 394–404, 1992.
12. D. Kulkarni and M. Stumm. Computational alignment: A new, unified program transformation for local and global optimization. Technical Report CSRI-292, Computer Systems Research Institute, University of Toronto, January 1994.
13. D. Kulkarni and M. Stumm. CDA loop transformations. In *Proceedings of Third workshop on languages, compilers and run-time systems for scalable computers*, Troy, NY, May 1995.
14. D. Kulkarni, M. Stumm, R. Unrau, and W. Li. A generalized theory of linear loop transformations. Technical Report CSRI-317, Computer Systems Research Institute, University of Toronto, December 1994.
15. K.G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, July 1992.
16. J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of parallel and distributed computing*, 13:213–221, 1991.
17. W. Li and K. Pingali. A singular loop transformation framework based on nonsingular matrices. In *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.
18. D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. *SIGPLAN Notices*, 26(6):1–14, 1991.
19. D. Padua. Multiprocessors: Discussion of some theoretical and practical problems. PhD thesis, University of Illinois, Urbana-Champaign, 1979.
20. W. Pugh. Uniform techniques for loop optimization. In *International Conference on Supercomputing*, pages 341–352, Cologne, Germany, 1991.
21. J. Torres and E. Ayguade. Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of 1993 International Conference on Supercomputing*, Tokyo, Japan, July 1993.
22. J. Torres, E. Ayguade, J. Labarta, and M. Valero. Align and distribute-based linear loop transformations. In *Proceedings of Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.
23. M.E. Wolf and M.S. Lam. An algorithmic approach to compound loop transformation. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.