P-Computes : A Flexible Computation Rule and Its Implementation by Explicit Computation Movement

Dattatraya Kulkarni and Michael Stumm

Department of Computer Science

Department of Electrical and Computer Engineering

University of Toronto, Toronto M5S 1A4

Canada

July 19, 1994

Abstract

We present a loop transformation framework that operates at a granularity much smaller than the existing transformation frameworks. *Computation decomposition* divides a loop body into subcomputations down to the level of subexpressions; and *computation alignment* linearly transforms them separately and independently. Together they redefine iterations and reorganize them in the iteration space. This explicit computation movement is used to implement a new, flexible computation rule, *P-computes*, which takes into account the locations of the data and thus optimizes SPMD code. We present polynomial time algorithms that derive optimal P-computes rules for given data distributions.

1 Introduction

In a framework such as HPF [6], the data distributions are usually specified by the user or suggested by some automatic tool such as PARADIGM [7]. A computation rule then decides where each statement instance is to be executed, effectively deriving a computation partition for the given distribution. Traditionally, computation rules have been *fixed* in that they do not take the distributions and alignments of all references into account. This generally necessitates the introduction of additional data alignments, if code is to be generated that performs well, i.e. with few or no intrinsics, minimal communication, and good load balance. The *owner-computes rule* [8, 6] is an example of such a fixed rule and is used almost exclusively. This rule is optimal only when the lhs and the rhs references are either all on the same processor (or when they are all on different processors, in which case there is no room for optimization).

We believe that *flexible* computation rules that take distributions and alignments into account are necessary for optimizing SPMD code, especially when explicit data movements occur through redistribution or XDP [3] directives or implicit data movements occur as in dynamic data alignments [10]. This paper presents a method that decomposes the computations of a nested loop into smaller computations so that they can be moved individually to the appropriate iterations so as to optimize performance. The result is an implementation of a new, flexible computation rule $\bigotimes_p$, called *P-computes*.

Traditional linear loop transformation theory [22, 4, 11, 15] assumes that the loop body is an indivisible unit of computation, but we believe that the loop body is too coarse a unit to manipulate for the purpose of optimizing SPMD code. Instead, we identify computational units at a much finer granularity and align them relative to one another in the computation space. A loop body can be decomposed into smaller units in two steps. First, each subexpression on the rhs of a statement can be considered a distinct subcomputation and be represented by a separate statement; the original statement is thus transformed into a sequence of statements that together compute the same result as the original statement. Second, each statement in the loop body (including those corresponding the subexpressions) can be considered a separate entity. A different linear transformation can be applied to each of these statements, allowing us to align them relative to one another.

The notion of applying a different transformation to each statement has origins in loop alignment [17, 1]. Kelly and Pugh [9] and independently Kulkarni and Stumm [12] generalized linear transformations so as to include arbitrary alignments of statements. The amount of communication in SPMD code depends on the location of data. Bala and Ferrante [3] proposed the insertion of XDP directives that explicitly move the data to transfer ownerships. Dynamic data alignment [10] can be considered in this context as a structured form of implicit data movement. Earlier approaches resorted to static solutions by deriving best data alignments [14] and distributions [7, 2] considering global constraints. Kulkarni and Stumm [12] developed the notion of computation movement as a *dual* of data alignment and provide an algorithm to generate *guard-free* aligned code. Torres et al. [21] employed loop alignment to account for data alignments.

We introduce computation decomposition and alignment in Section 2. We present the P-computes rule in Section 3 and discuss its implementation by computation decomposition and alignment. In Section 4, we discuss some of the effects computation decomposition and alignment has on the loop structure. In Section 5, we present algorithms to derive optimal non-nested P-computes rules for given data distributions.[1] Finding an optimal *direct* P-computes is polynomial in the size of the loop and the number of processors ($O(PN^n)$) but more expensive than the loop itself, whereas deriving an optimal *indirect* P-computes is less expensive ($O(r(N/P)^n)$). Lastly, we present an efficient algorithm ($O(nrn_D v^{r^2})$) to derive good computation decomposition and alignment when block distributions are used.

2 Computation Decomposition and Alignment

In this section, we introduce the concepts of computation decomposition, where the computation corresponding to a loop body is decomposed into smaller computations, and computation alignment, where these computations are aligned relative to one another. The semantics of an assignment statement only dictates *which* array elements are accessed, but not *when*. An array element can be referenced in any iteration, as long as it has the correct value needed. Therefore, the subcomputations of a statement resulting from a decomposition can be executed in earlier iterations by some computation alignment.[2] Computation decomposition and alignment thus redefine the iterations themselves, besides reorganizing these new iterations in the iteration space. Together they form a transformation framework that operates at a much finer granularity than existing frameworks.

[1] Parameters introduced later: dimension of the array (loop) n which is usually not more than 4, the maximum iterations in a dimension N, the number of references in the loop r, the maximum offset in the references v, the number of processors P, and the number of computation decompositions (a polynomial in of r) n_D.

[2] We could view this as a generalization of prefetching – just as elements are accessed before they are needed to overlap latency, subcomputations are performed before they are needed.

2.1 Computation Decomposition

Computation decomposition divides a loop body into a sequence of statements and the computation of each statement into a sequence of several smaller subcomputations corresponding to subexpressions. Recurrence for execution on systolic arrays are decomposed in a similar way [19].

As an example, one possible decomposition of the statement S_1:

```
for  i = 1,n
  for  j = 1,n
    S1 :  a(i,j)  =  a(i-1,j)  +  a(i,j-1)  +  a(i-1,j-1)  +  a(i-1,j+1)
  end for
end for
```

results in subcomputation S_{11} and S_{12}:

```
for  i = 1,n
  for  j = 1,n
    S11 :  t(i,j)  =  a(i-1,j)  +  a(i-1,j-1)
    S12 :  a(i,j)  =  t(i,j)  +  a(i,j-1)  +  a(i-1,j+1)
  end for
end for
```

where t is a temporary array. Temporary arrays have the same dimension as that of the lhs of the original statement. The only new dependence introduced by computation decomposition is a loop independent dependence on the temporary. The decomposed loop clearly has more computation to do, but the idea is that the additional cost will be more than offset by the reduction in communication as a result of a subsequent computation alignment. The additional storage required by temporaries is not of much concern in large systems.

Most grid computations in numeric applications involve statements having several rhs array references. Without loss of generality, we assume that the expression is an addition (subtraction, multiplication) of several elements.[3] For a statement with r right hand side references there are

$$ {}^{r}C_{m_1} \times {}^{r-m_1}C_{m_2} \times \ldots {}^{(r-\sum_{j=1}^{j=i-1} m_j)}C_{m_i} \ldots \times {}^{(r-\sum_{j=1}^{j=k-1} m_j)}C_{m_k} \tag{1} $$

[3] Whenever there are several kinds of operators on the right hand side, or when there are procedure calls, one has to adhere to operator precedence constraints and any side effects of the procedures.

decompositions, for each choice of m_i satisfying $1 \leq m_i \leq r - \sum_{j=1}^{j=i-1} m_j$ and $r = \sum_{j=1}^{j=k} m_j$. We denote the total number of decompositions by n_D, and the expression ${}^r\mathtt{C}_m$ denotes the number of ways to choose a subset of m items from a set of r items. Clearly $k \leq r$. The decomposition introduces $k-1$ temporary arrays, $\mathtt{t_1} \ldots \mathtt{t_{k-1}}$. It is assumed here that all temporary arrays are referenced in the k^{th} subcomputation which contains the assignment to the lhs of the original statement; that is, above Equation does not include the additional combinatorics relating to the temporary variables.

2.2 Introduction to Computation Alignment

Computation alignment [12, 9] is a generalization of linear loop transformation in that each statement in a loop body can be transformed independently and differently. Computation alignment redefines the iterations, thus moving individual subcomputations across processors through later mappings from iterations to processors.

Just as a loop body has an iteration space, each statement in the loop body has a corresponding computation space. Dependences are then relations between computations of the same or different computation spaces. A computation alignment of a loop is a set of linear transformations, one for each computation space. A transformation on a computation space affects only the dependences between the computations in that space and the other computation spaces and leaves all other dependences unchanged. Determining the legality of a particular alignment has become feasible only after the recent advances in deriving the dependence information [16, 5, 18].

Traditional linear loop transformation and loop alignment are just special cases of computation alignment. The extra degree of freedom provided by computation alignment results in several advantages. *i*) Computation alignment can be applied to both perfect and imperfect nestings [12, 9]. *ii*) It can be used to improve variable reuse by making the reference patterns in the loop as similar as possible. *iii*) Some data alignment requirements can be achieved through computation alignment instead of explicit data alignments, thus avoiding expensive data realignments [12]. We expect local optimization algorithms to employ both data and computation alignments to minimize (and possibly eliminate) ownership tests. *iv*) Computation alignment can be used to selectively transform references and thus make the structure of a loop better match a given set of data distributions. *v*) Computation alignment can be used for global optimization, since a loop can potentially be transformed to be similar to another loop in the way it accesses the data [12].

The union of the transformed computation spaces determines the bounds of the transformed loop. In general, the union can be non-convex and the loop bounds are typically generated from closest convex approximation to the union. In this case, *guards* are used to prevent the execution of the additional computations thus introduced. As an alternative, it is possible to generate loops that are guard-free, but have a more complicated structure to eliminate the runtime overhead of guards [12].

Figure 1 contains an example of how computation alignment is used to eliminate ownership tests.

3 Explicit Computation Movement

Alignment transformations on a statement's subcomputations explicitly move the subcomputations in the computation space. The execution of the original statement may thus be distributed across possibly several iterations. The alignments of the subcomputations therefore collectively specify a complex computation rule for the statement. For instance, a possible alignment for the decomposition of the previous section is:

$$\begin{array}{ll} \mathtt{S}_{12}: & \mathtt{a(i,j)} \;=\; \mathtt{t(i,j)} \;+\; \mathtt{a(i,j-1)} \;+\; \mathtt{a(i-1,j+1)} \\ \mathtt{S}_{11}: & \mathtt{t(i+1,j)} \;=\; \mathtt{a(i,j)} \;+\; \mathtt{a(i,j-1)} \end{array}$$

Suppose that we collocate $\mathtt{t(i+1,j)}$ and $\mathtt{a(i,j)}$. If the owner-computes rule is used on this aligned loop, then the original computation is performed partly on the processor with $\mathtt{a(i,j)}$ and partly on the processor with $\mathtt{a(i-1,j)}$. This alignment changes the dependence structure from {(1,1), (1,-1), (1,0), (0,1)} to {(1,-1), (1,0), (0,1)}. The decomposed form with temporary arrays can usually be optimized to eliminate the intrinsics.

Computation decomposition and alignment thus in effect realize a new computation rule with the aid of owner-computes. For that matter, we can use any fixed rule along with suitable decompositions and alignments to implement a variety of computation rules.

We introduce the P-computes operator $\bigotimes$ capable of expressing a range of computation rules. $\bigotimes_p (expr)$ specifies that the expression $expr$ for the current iterator values should be executed on processor p. The result is sent to the processor designated by the enclosing P-computes operator. To simplify the presentation of this paper, however, we restrict ourselves to non-nested $\bigotimes$ operators.[4] The processor p in $\bigotimes_p$ can be designated

[4] That is, in an rhs P-computes $\bigotimes_p (expr)$, $expr$ itself does not contain $\bigotimes$ operators

either *directly* as a mapping from iterators to the processors,[5] or *indirectly* in terms of the distribution functions of the arrays. The decomposition and alignment of $\mathtt{S}_{11}$ and $\mathtt{S}_{12}$ above is equivalent to the following indirect P-computes rule:

$$\begin{aligned}\mathtt{S}_1: \bigotimes_{\mathtt{iown\ a(i,j)}}: \mathtt{a(i,j)} \;&=\; \bigotimes_{\mathtt{iown\ a(i-1,j)}} (\mathtt{a(i-1,j)} \;+\; \mathtt{a(i-1,j-1)}) \;+ \\ &\quad\; \bigotimes_{\mathtt{iown\ a(i,j)}} (\mathtt{a(i,j-1)} \;+\; \mathtt{a(i-1,j+1)})\end{aligned}$$

where $\mathtt{iown\ a(i,j)}$ is the classical ownership intrinsic [8, 6, 3] that evaluates to true on the processor that owns data element $\mathtt{a(i,j)}$.

To compare P-computes with owner-computes rule, consider the following computation, tailored to accentuate the difference.

$$\mathtt{a(i,j)} = \mathtt{b(i,j)} + \mathtt{c(i,j)} + \mathtt{d(i,j)}$$

Suppose $\mathtt{a(i,j)}$ is mapped onto processor $(i \; div \; P)$, $\mathtt{b(i,j)}$ and $\mathtt{c(i,j)}$ onto $(i \; mod \; P)$, and $\mathtt{d(i,j)}$ onto $(i \; mod \; P) + 1$. Applying the owner-computes rule to this computation requires $3(n - P)$ communications, assuming that the arrays of size $n \times n$. In contrast, the P-computes rule below

$$\bigotimes_{\mathtt{iown\ a(i,j)}} \mathtt{a(i,j)} = \bigotimes_{\mathtt{iown\ b(i,j)}} (\mathtt{b(i,j)} + \mathtt{c(i,j)} + \mathtt{d(i,j)})$$

requires only n communications.

4 The Effects of Computation Decomposition and Alignment

In this section, we discuss some of the effects of computation decomposition and alignment on the loop structure. Since the references and dependences determine the amount parallelism available and communication required, computation decomposition and alignment can have significant impact on performance.

[5] Clearly, the function should be complete, where every computation is mapped onto some processor.

To simplify the discussion in this paper, we assume references with indices of the form $i \pm c$, where i is an iterator and occurs only once in a reference, and c is some constant. This allows the references to be represented by offset vectors, or "stencils" just like constant dependences.[6] We also assume that all memory related dependences are eliminated by some renaming technique.

4.1 Dependences and References

Computation decomposition groups the references and dependences of a statement into subsets, each corresponding to a subcomputation. An alignment transformation on a subcomputation transforms the references and dependences of only one subset relative to the others.

Suppose a statement $\mathtt{S_1}$ is decomposed into $\mathtt{S_{11}}$ and $\mathtt{S_{12}}$, and correspondingly the dependences D and the references R in $\mathtt{S_1}$ are partitioned into $\{D_1, D_2\}$ and $\{R_1, R_2\}$ respectively. Shifting $\mathtt{S_{11}}$ by $\mathtt{t}$ results in new dependences D' and references R' where

$$D' = \{\mathtt{d} - \mathtt{t} \mid \mathtt{d} \in D_1\} \cup D_2 \cup \{\mathtt{t}\}$$

$$R' = \{\mathtt{r} + \mathtt{t} \mid \mathtt{r} \in R_1\} \cup R_2 \cup \{\mathtt{t}, \mathtt{0}\}$$

The new dependence $\mathtt{t}$ is on the temporary array introduced by the decomposition, and the new reference offsets $\mathtt{t}$ and $\mathtt{0}$ are due to references to the temporary array in $\mathtt{S_{11}}$ and $\mathtt{S_{12}}$ respectively. Note that when there is more than one statement in the loop body, D includes self dependences in $\mathtt{S_1}$ as well as the dependences between $\mathtt{S_1}$ and other statements.

4.2 Parallelism

The maximum do-across parallelism in a loop is a constant irrespective of any applied linear transformation. The amount of do-across parallelism that can be exploited, however, is determined by the magnitude of the components in the dependence that is internalized[7] by the linear transformation. For example, a linear transformation internalizing dependence (a, b) in a 2-dimensional loop with constant bounds of N will have a do-across parallelism of $P = (\mid a \mid + \mid b \mid) * N$ [11]. Computation decomposition and alignment will

[6] The particular array involved is implicit.

[7] The dependence that is transformed to have no dependence at the outermost level.

change the original dependence structure itself, and thus can be used to increase the maximum and realizable do-across parallelism in the loop. The number of do-all parallel levels, however, can not be changed by any transformation, because all linear transformations preserve the rank of the dependence matrix.[8]

4.3 Communication Optimization in Full-rank Loops

When the rank of the dependence matrix is equal to the dimension of the loop, one usually applies a data distribution that minimizes communication because it is not possible to obtain do-all parallelism in the outer loop [13]. Since the choice of distribution depends on the reference offset vectors, computation decomposition and alignment can increase the number of distribution alternatives available by changing the offset vectors. On the other hand, if a distribution is given, then decomposition and alignment can reduce the communication requirements by changing the reference offset vectors. We discuss this further in the next section.

4.4 Intrinsics Elimination

A major source of overhead in SPMD code is the cost of evaluating the intrinsics in the computation rule. While it is difficult to eliminate intrinsics with arbitrary distributions and computation rules, computation alignment is effective in eliminating intrinsics when all arrays in the original loop have the same distribution. The basic idea is to align the subcomputations such that each is mapped onto the same processor. The general procedure to align the subcomputations $sc_1, \ldots, sc_{k-1}$ to sc_k which entails the assignment to $\mathtt{a(i,j)}$, the lhs of the original statement is as follows. Without loss of generality, suppose $\bigotimes_{\mathtt{iown}\ \mathtt{b(i-c1,j-c2)}}$ is the computation rule applied to sc_1.

$$\mathtt{sc_1}: \bigotimes_{\mathtt{iown}\ \mathtt{b(i-c1,j-c2)}} \mathtt{t_1(i,j)} = \ldots + \mathtt{b(i-c1,j-c2)}$$

$$\vdots$$

$$\mathtt{sc_k}: \bigotimes_{\mathtt{iown}\ \mathtt{a(i,j)}} \mathtt{a(i,j)} = \ldots$$

In general, since there may exist an alignment of $\mathtt{a}$ to $\mathtt{b}$, assume $\mathtt{a(i,j)}$ and $\mathtt{b(i-\alpha_1,j-\alpha_2)}$ are collocated. Clearly, α_1 and α_2 will be zero when they are not aligned.[9] We align sc_1 to sc_k shifting sc_1 by

[8] A linear transformation that increases the amount of outer loop parallelism only exposes the existent parallelism at the outer level.

[9] The only difference when the computation rule for sc_1 is $\bigotimes_{\mathtt{iown}\ \mathtt{a(i-c1,j-c2)}}$ is that there are no alignments and $\mathtt{(c1,c2)}$

$(\mathtt{c1}-\alpha_1, \mathtt{c2}-\alpha_2)$. The alignment is legal when $(\mathtt{c1}-\alpha_1, \mathtt{c2}-\alpha_2)$ is positive and all other dependences between sc_k and sc_1 remain positive. We follow this by aligning the temporary array $\mathtt{t}_1$ to $\mathtt{a}$ so that $\mathtt{t}_1(\mathtt{c1}-\alpha_1, \mathtt{c2}-\alpha_2)$ and $\mathtt{a}(\mathtt{i},\mathtt{j})$ are collocated.

$$\mathtt{sc}_1: \ \bigotimes_{\mathtt{iown}\ \mathtt{b}(\mathtt{i}-\alpha_1, \mathtt{j}-\alpha_2)} \ \mathtt{t}_1(\mathtt{i}+\mathtt{c1}-\alpha_1, \mathtt{j}+\mathtt{c2}-\alpha_2) = \ldots \ + \ \mathtt{b}(\mathtt{i}-\alpha_1, \mathtt{j}-\alpha_2)$$

$$\vdots$$

$$\mathtt{sc}_\mathtt{k}: \ \bigotimes_{\mathtt{iown}\ \mathtt{a}(\mathtt{i},\mathtt{j})} \ \mathtt{a}(\mathtt{i},\mathtt{j}) = \ldots$$

The P-computes rules of both sc_1 and sc_k in a given iteration now map to the same processor. Similar steps are taken for $sc_2, \ldots sc_{k-1}$. The entire iteration can thus be mapped to a single processor so that the intrinsics need not be evaluated, and therefore the computation partition can be expressed in the loop bounds themselves. The procedure eliminates the intrinsic for every computation that can be aligned legally. It is guaranteed to do so when $(c1, c2)$ are the smallest of the offset vectors in the subcomputation to be aligned.

5 Deriving Optimal P-computes Rule

It is an optimization problem to derive a P-computes rule for a statement that results in minimal communication. In this section, we present algorithms to derive optimal direct and indirect P-computes rules.

5.1 Optimal Direct P-Computes Rule

Deriving an optimal direct P-computes rule given P processors and data distributions for the referenced arrays entails finding a function that maps the subcomputations in the loop to processors so as to minimize communication. The decision as to where a given subcomputation is to be executed most efficiently is independent of the location of the other subcomputations. Therefore, a simple exhaustive search will find the optimal mapping.

Figure 2 contains a simple algorithm, A1, that exhaustively determines the best processor on which to execute each subcomputation. $Cost(ref, \mathtt{I}, p)$ in step 7 is the cost of accessing element $ref(\mathtt{I})$ from processor p, assuming $\mathtt{I}$ is an iteration. The result of the algorithm is a table mapping each computation onto the best

is a dependence, therefore lexicographically positive.

processor. The algorithm is polynomial in the size of the array and the number of processors, for any given data distributions. Step 3 has $n_D = \mid C_D \mid$ iterations, as given by equation 1 which is a small number since there are few references in a statement. Similarly, step 4 has few iterations. Step 5 is the most expensive and has N^n iterations. Step 6 has P iterations. The complexity of A1 is thus $O(PN^n)$, where n is usually not more than 4 in practice. The proof of optimality is straight forward for non-nested P-computes operators and is omitted here.

The algorithm demonstrates that the problem is polynomial. It requires that N be a compile time constant. Although the mapping is derived only once at compile time, the complexity is higher than that of the original loop itself. At run-time, each processor uses the table produced by the algorithm to look up whether it must execute a particular computation or not. This table look up for each computation my incur substantial run-time overhead. As an alternative, the user may be able to provide closed formulae for direct mappings.

5.2 Optimal Indirect P-Computes Rule

Specifying the P-computes rule indirectly has the advantage that the run-time tables used for the data distributions can be used for the computation rule as well. It is more intuitive for the programmer to think of a computation rule in terms of distribution functions, at the time she provides the data distributions. Algorithm A2 in Figure 3 provides steps that replace steps 5, 6 and 7 in A1 to find an optimal indirect P-computes rule for a loop. $Cost(r_i, own(r_l))$ in step 7 is the cost of accessing element r_i from processor $own(r_l)$.

A2 derives the optimal P-computes rule when the "sample" iteration space $\mathcal{I}'$ chosen in step 5 is the same as as the iteration space $\mathcal{I}$ for the loop. The complexity of the algorithm can be reduced significantly by observing that the iteration space is uniform. Therefore, we can still obtain good solutions by choosing $\mathcal{I}'$'s that are much smaller subsets of $\mathcal{I}$. $\mathcal{I}'$ should, however, be representative in that it covers all n dimensions, and it should have more iterations than the largest number of data elements mapped to a single processor. The number of data elements on a processor is usually $O((N/P)^n)$.[10] Note that p in $\bigotimes_p(c)$ has to be the owner of one of the references in c. Therefore step 6 will iterate only a small number of times. The complexity of A2 is thus $O(r(N/P)^n)$, where r is the number of references.

[10] Assume that the arrays have have dimension n as well.

5.3 Specialization for Blocked Distributions

In grid computations that have several references ib tge rhs of the statements, the dependence matrix often will have full rank, so there will be no parallel do-all loops. Communication is then minimized by distributing the data in a blocked fashion so communication is necessary only at the block boundaries. In this case, it is not necessary to analyze the entire iteration space to estimate the communication cost. We present an algorithm that generates a good P-computes rule that analyzes communication at just the block boundaries.

Suppose the block distribution of an n-dimensional array is specified by $F = (\mathtt{f_1 f_2 \ldots f_s})^T$, and $(A_1, A_2, \ldots, A_s)$, where $\mathtt{f_i}$ is the normal to the i^{th} hyperplane of the defining data partition, and A_i is the number of integer points on it.[11] The cost of communications is:

$$C = 2\left(\sum_{f_i \in F} A_i \sum_{\mathtt{d} \in D} \mathtt{f_i} \cdot \mathtt{d}\right) \tag{2}$$

where D is the set of all dependences (or equivalently offset vectors, since the references are in $i \pm c$ form). When two arrays have the same dependence $\mathtt{d}$, then D contains two instances of $\mathtt{d}$. Equation 2 is a generalization of the equation presented in [20] for arbitrary dimensions. Note that this is only a measure of communication volume as it does not account for possible vectorizations and coalescing.[12]

Step 3-7 in Figure 4 replace those in algorithm A1. The references in each of the subcomputations determines the alignments that can be applied, and $\mathcal{A}$ in step 5 is the space of these alignments. The size of $\mathcal{A}$ is $O(v^{rk})$ where v is the maximum offset over all dimensions of all references. Step 7 estimates the amount of communication along all the boundaries, give a candidate alignment $\mathtt{a}$ of the subcomputations. This step is $O(sr)$, which is usually a small number. Therefore, the overall complexity of the algorithm is $O(nrn_D v^{r^2})$, since the maximum of k is the number of references r, and s is a small multiple of n, the number of dimensions of the loop (array).

[11] One usually specifies the sizes of the block along various hyperplanes, and A_i's are derived from them.

[12] For instance, when the distribution is square blocks of size b, receiving $\mathtt{a(i-1, j-1)}$ and $\mathtt{a(i-1, j)}$ is counted as $2b$ communications, whereas the same can be achieved in $b+1$ communications or even one communication vector if possible.

6 Summary and Future Work

We introduced computation decomposition and alignment that provide a transformation framework at a granularity that is finer than those of existing techniques. The P-computes rule is a flexible computation rule which takes into account the distributions of the data and hence can be used to generate SPMD code with much less communication when compared to fixed computation rules. Computation alignment is also shown to eliminate ownership intrinsics in most cases. The presented algorithms that generate optimal *direct* and *indirect* non-nested P-computes rules for given distributions demonstrate that the problem is polynomial. We are currently improving the algorithms and extending them for nested P-computes rules.

Algorithm A1 of this paper outputs direct P-computes rule in the form of a table. We are working on recognizing patterns in distribution functions so that closed form functions for good P-computes rules can be derived. We are also working on lowering the sizes of the temporary arrays and making them private. The alignment functions are more complicated and difficult to enumerate when the references are arbitrary linear functions. We are extending the algorithms to handle such references.

References

[1] Randy Allen, David Callahan, and Ken 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] P. Feautrier. Dataflow analysis of array and scalar references. volume 20, 1991.

[6] HPF Forum. HPF: High performance fortran language specification. Technical report, HPF Forum, 1993.

[7] M. Gupta. Automatic data partitioning on distributed memory multicomputers. Technical report, Dept of computer Science, University of Illinois at Urbana Champaign, 1992.

[8] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the fortran d programming system. Technical Report CRPC-TR91121, Dept of computer Science, Rice University, 1991.

[9] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126, University of Maryland, 1992.

[10] 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.

[11] D. Kulkarni, K.G. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[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, 1994.

[13] K.G. Kumar, D. Kulkarni, and A. Basu. Generalized unimodular loop transformations for distributed memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, Chicago, MI, July 1991.

[14] 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.

[15] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[16] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 1–14, Toronto, Ontario, Canada, 1991.

[17] D. Padua. Multiprocessors: Discussion of some theoretical and practical problems. Phd thesis, University of Illinois, Urbana-Champaign, 1979.

[18] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. Technical Report CS-TR-3196, University of Maryland, 1993.

[19] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice Hall, 1991.

[20] J. Ramanujam and P. Sadayappan. Compile time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distributed Systems*, 2(4):472–482, October 1991.

[21] 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.

[22] 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.

Example: Consider the following loop that references two arrays `A` and `B`, that have been previously aligned such that `A(i,j)` and `B(i-1,j)` are collocated. Since they are aligned, they will be distributed the same way.

```
for  i  =  0,n
  for  j  =  0,n
    S1 :   A(i,j)  =  ...
    S2 :   B(i,j+i)  =  ...
  end for
end for
```

The following transformation applied to S_2's computation space aligns the S_2 computations to the S_1 computations so that a loop partitioning can be accomplished without any ownership tests or data movements.

$$f_c = \begin{bmatrix} 1 & 0 & -1 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The aligned loop becomes:

```
for  i  =  0,n+1
  for  j  =  i,n+i
    (i ≤ n, j ≤ n) :  S1 :   A(i,j)  =  ...
    (i ≥ 1, j ≥ i) :  S2 :   B(i-1,j)  =  ...
  end for
end for
```

The guards are shown to simplify the code but can be eliminated.

Figure 1: An Example Computation Alignment

Algorithm : A1

input: processors $P_s = \{1, \ldots, P\}$
$D_A : A \rightarrow P$ for each array **A**
$S = \{S_1, \ldots, S_m\}$ statements in the loop
output: P-computes rule table for S
begin
1. **for** each $S_i \in S$ **do**
2. $C_D \leftarrow \{c_{d1}, \ldots, c_{dq}\}$ decompositions for S. /* $q = n_D$ the number of decompositions */
$cost(C_D) \leftarrow \infty$
$best\_decomp \leftarrow c_{d1}$
3. **for** each $c_{dj} \in C_D$ **do**
$cost(c_{dj}) = 0$
4. **for** each subcomputation $sc_l \in c_{dj}$ **do**
$cost(sc_l) = 0$
5. **for** each $\mathtt{I} \in \mathcal{I}$ do
$best\_proc(sc_l) \leftarrow 1$
$cost(sc_l, \mathtt{I}) = \infty$
6. **for** each $p \in P_s$ **do**
7. $cost(sc_l, \mathtt{I}, p) \leftarrow \sum_{ref \in sc_l} cost(ref, \mathtt{I}, p)$ /* use D_A for the ref */
if $(cost(sc_l, \mathtt{I}, p) < cost(sc_l, I))$ **then**
$best\_proc(sc_l, \mathtt{I}) \leftarrow p$
$cost(sc_l, \mathtt{I}) = cost(sc_l, \mathtt{I}, p)$
end if
end for
$cost(sc_l) = cost(sc_l, \mathtt{I}) + cost(sc_l)$
end for
$cost(c_{dj}) \leftarrow cost(c_{dj}) + cost(sc_l) + 1$
end for
if $(cost(c_{dj}) < cost(C_D))$ **then**
$cost(C_D) = cost(c_{dj})$
$best\_decomp = c_{dj}$
end if
end for
end for
end

Figure 2: A1 – Algorithm for Optimal Direct P-computes

Algorithm : A2

$R = \{r_1, \ldots, r_t\}$ references in sc_l
$arr(r_l)$ array associated with r_l
$own(r_l)$ processor to which $D_{arr(r_l)}$ maps the element

5. **for** each $\mathtt{I} \in \mathcal{I}'$ do
    $best\_proc(sc_l) \leftarrow 1$
    $cost(sc_l, \mathtt{I}) = \infty$
6.   **for** each $r_l \in R$ **do**
7. $cost(sc_l, \mathtt{I}, own(r_l)) \leftarrow \sum_{r_i \in R} cost(r_i, own(r_l))$
        if $(cost(sc_l, \mathtt{I}, own(r_l)) < cost(sc_l, I))$ **then**
            $best\_proc(sc_l, \mathtt{I}) \leftarrow own(r_l)$
            $cost(sc_l, \mathtt{I}) = cost(sc_l, \mathtt{I}, own(r_l))$
        end if
    end for
    $cost(sc_l) = cost(sc_l, \mathtt{I}) + cost(sc_l)$
end for

Figure 3: A2 – Algorithm for Optimal Indirect P-computes

Algorithm : A3

3. **for** each $c_{dj} \in C_D$ **do**
4. $best\_align(c_{dj}) \leftarrow \mathtt{0}$
   $cost(c_{dj}) = 2\left(\sum_{f_i \in F} A_i \sum_{\mathtt{d} \in D} \mathtt{f_i} \cdot \mathtt{d}\right)$
5. **for** each $\mathtt{a} = (a1, \ldots, a_k) \in \mathcal{A}$ do /* $k =| c_{dj} |$ */
6. $D' \leftarrow$ new dependences after aligning by $\mathtt{a}$
7. $cost(c_{dj}, \mathtt{a}) = 2\left(\sum_{f_i \in F} A_i \sum_{\mathtt{d} \in D'} \mathtt{f_i} \cdot \mathtt{d}\right)$
   if $(cost(c_{dj}, \mathtt{a}) < cost(c_{dj}))$ **then**
   $best\_align(c_{dj}) \leftarrow \mathtt{a}$
   $cost(c_{dj}) = cost(c_{dj}, \mathtt{a})$
   end if
   end for
   end for

Figure 4: A3 – Algorithm for Blocked Distributions