CRAFT: A Framework for F90/HPF Compiler Optimizations*

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Taipei, 11529 Taiwan
wuj@iis.sinica.edu.tw

Marina Chen
Computer Science Dept.
Boston University
111 Cummington Street
Chestnut Hill, MA
mcchen@cs.bu.edu

James Cowie
Cooperating Systems Corp.
12 Hollywood Drive
Chestnut Hill, MA
cowie@cooperate.com

Abstract

In this paper, we give an overview of the results of the CRAFT optimizing compiler project (Fortran 90/HPF subset compilers). We start by describing the theoretical framework within which we designed program transformations for the optimization of inter- and intraprocedural data motion, as well as the optimizations for parallel loops; we then describe the implementation of the CRAFT compilers for Thinking Machines' CM-2 and CM-5.

We report results from experiments on the Connection Machine CM-5, the IBM SP-2, and a network of UltraSparc workstations. The results demonstrate that these optimizations can achieve significant object code performance improvement.

1 Introduction

The advance of computer technology has made distributed-memory parallel computers a reality. In recent years a number of parallel machines have been introduced into the market (e.g. the Connection Machines CM-5, Intel Delta Touchstone, Paragon and the iPSC hypercubes, NCUBEs, Cray T3D and T3E, and the IBM SP-2 systems). The largest commercially available parallel machines today can deliver peak rates of over one hundred billion floating point operations per second; this is projected to increase by over an order of magnitude within the next few years. This massive computing power will make this class of machines an attractive choice for solving large problems.

Despite these promises, however, distributed-memory parallel machines have not yet entered the mainstream of computation. The main obstacle is the difficulty of programming them. In recent years, much research effort has been devoted to providing suitable programming tools for these machines. One focus is on the provision of appropriate high-level, data-parallel programming languages (e.g. Fortran 90 [4], CM-Fortran [24], C* [25], Crystal [19], Fortran D [29], Vienna Fortran [14], and High Performance Fortran (HPF) [38]) to ease parallel programming.

In this paper, we study issues critical to achieving high performance for Fortran 90/HPF on distributed-memory parallel systems. We give the theoretical framework within which we designed program trans-

*Support for this work was provided by Cooperating Systems Corporation and Syracuse University under ARPA contract DABT63-91-C-0031, the Army High Performance Computing Research Center at University of Minnesota, and Academia Sinica.

formations for optimizing HPF programs. We also describe the implementation of the CRAFT compilers (Fortran 90/HPF subset compilers) for Thinking Machines' CM-2 and CM-5.

Compiling HPF for Distributed-Memory Parallel Machines

Two main factors in achieving high performance for HPF programs on distributed-memory parallel machines are reducing communication overhead and optimizing code performance at the processor level. While development of optimizing compilers for super-scalar architectures is becoming commonplace in the industry, work on optimization for data movement and node code performance is mostly done in the context of specialized, hand-crafted code written in assembly code, if not microcode, for specific target machines (e.g. TMC's Convolution Compiler for stencil computation [12] and CMSSL (Connection Machine Scientific Software Library) [59] routines for scientific computation).

Performance in HPF is measured by the time spent both moving data into the proper place and configuration for computation, as well as the computation process itself. For most HPC architectures, optimizing data movement is at least as important, if not more so, than the arithmetic actually performed. Movement of distributed data in HPF can occur in two ways: (1) passing of distributed arrays in procedure calls. Note that the actual (the argument supplied by the caller) and the dummy (the formal parameter specification provided by the callee for a given argument) may have different data distribution. (2) assignments on distributed arrays within a procedure body. Again, the LHS (left-hand side) and the RHS (right-hand side) of an assignment statement may have different data distribution. This makes compilation of HPF to efficient target code without excessive data copying a complex task.

Compilation of parallel loops also imposes challenge to an optimizing Fortran 90/HPF compiler. Although in HPF parallel loops can be explicitly given by the user, data partitioning specified by user directives can expose further opportunity for increasing parallelism or reducing communication. To achieve high performance, an optimizing compiler must catch all these opportunities and exploit optimizations through program transformations.

Overview

In this paper, we propose a two-phase transformative framework, as shown by the sketch of a HPF compiler in Figure 1, to tackle these two optimization problems.

First phase (called XFORM-1) is an early, abstract algebraic transformation and runtime support technique for reducing inter-procedural data layout conversion between actual arguments and dummy arguments, as well as intra-procedural data movement caused by array intrinsics within a procedure body. XFORM-1 transformation is done abstractly in the logical, global space defined in the program. It does not require the notion of processor IDs (denoted by P) and local memory offsets (denoted by V).

The second phase (called XFORM-2) performs optimizing transformations for data-parallel loop nests. The loop nests we consider in XFORM-2 transformations are called *iterative spatial loop* nests, which capture a broad class of HPF parallel loops. We have collected a set of optimizations which we have found may improve HPF code performance on distributed-memory parallel systems. The transformations in XFORM-2 are often machine-dependent, and they are aware of the existence of processors and local memory offsets.

In our preliminary experiments, we constructed subset Fortran 90/HPF compilers for Thinking Machines's CM-2 and CM-5 as the testbed for our optimizations. We evaluated the effectiveness of these optimizations using a set of linear algebra applications and PDE solvers. Our experimental results on

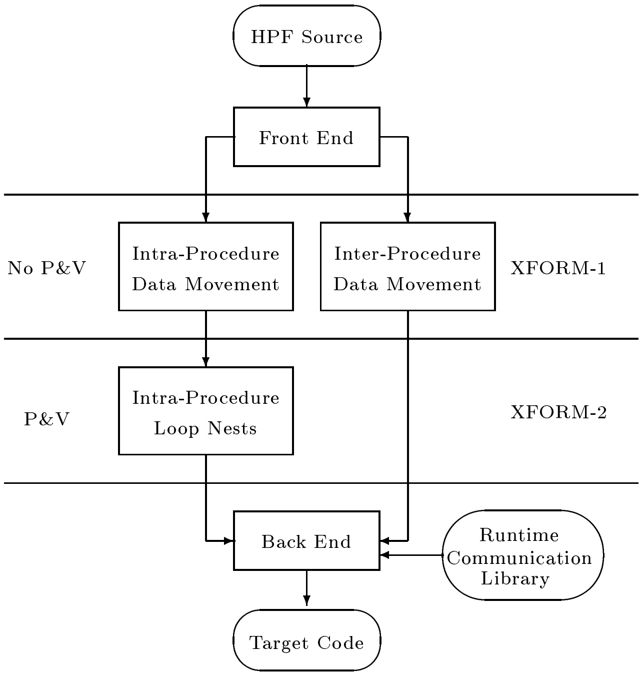


Figure 1: Sketch of a HPF Compiler

the Connection Machine CM-5 are quite encouraging. The performance of some of the optimized native codes is two orders of magnitude better than CM-Fortran compiler (a Fortran90 compiler developed for the Connection Machines) and approaching that of the hand-written CMSSL Library routines.

We have also studied the impact of some of these optimizations on modern parallel machines, including IBM SP-2 systems and UltraSparc workstation clusters, using manual compilation. Our experimental results demonstrated that modern machines can also benefit from these optimizations.

The rest of the paper is organized as follows. Section 2 presents the XFORM-1 algebraic transformation. Section 3 describes the XFORM-2 transformations for parallel loops. Section 4 reports our implementation status of the compilers and the experimental results on the Connection Machine CM-5, the IBM SP-2, and the UltraSparc workstation cluster. Finally, Section 5 reviews some related work and Section 6 concludes.

2 XFORM-1 Transformation

HPF provides a rich set of procedure interface specifications for distributed array arguments. A dummy argument has a data layout that is either explicitly specified via user directives, or implicitly inherited from the caller's actual argument. Similarly, an actual argument may have an explicitly prescriptive data layout, or itself inherits a data layout from yet another calling context. Consider the possibility of extended calling chains ('A' calls 'B' calls 'C', and so forth, each of which may add additional directives to the data layout specification and inherits the rest). This layout effect will be propagated to the inner levels of procedure calls. When the calling chain is long, a systematic approach is desirable to automate the process of optimizing data movement for passing array arguments across procedure boundaries, as well as moving data elements in array assignment statements within a procedure body.

```
ALPHA                        BETA (C,D)
real A(100),B(100)           real C(100),D(100)
distribute T cyclic          distribute T block
align A(i) with T(i)         align C(i) with T(i)
align B(i) with T(i+2)       align D(i) with T(i+1)
call BETA (A,B)              D = EOSHIFT(C,dim=1,shift=1)
                             ...
```

Figure 2: An example for data movement optimization where both the actual and dummy arguments have explicit data layouts

Consider the case where both the actual and dummy arguments of a single-level procedure call have explicit data layouts (Figure 2). Array `A` is cyclically partitioned in procedure `ALPHA`, and should be redistributed using block partition within procedure `BETA`. Layout conversion for array `B` is a complex one due to change of both alignment (changing between offseting two elements and offseting one element) and distribution (changing between cyclic partition and block partition). In the procedure body of `BETA`, the assignment statement shifts array `C` toward the negative direction by one element and assigns the result to array `D` (i.e. `C(i+1)` is assigned to `D(i)`). We call this *logical* data movement defined by the program. Due to the effect of the alignment directives, actual data movement for executing the assignment statement may be different from the logical data movement as it appears. In order to satisfy the directives as given by the user, the compiler must combine all the layout requests that are in force.

Assuming "owner computes" rule for the compilation of data movement, the compiler has two roles. First, the compiler should minimize time spent in moving `A` and `B` to `C` and `D`'s preferred layouts, and moving them back when returning from the call. For instance, the compiler should optimize communication for BLOCK and CYCLIC data redistribution. It should also determine the order in moving data for complex data movement. For instance, layout conversion for passing array `B` to procedure `BETA` involves an offseting alignment change and a conversion from CYCLIC distribution to BLOCK distribution. There are two alternatives in arranging data movement: offsetting realignment under CYCLIC distribution followed by CYCLIC-to-BLOCK redistribution, or CYCLIC-to-BLOCK redistribution followed by offsetting realignment under BLOCK distribution. The latter is more efficient because offsetting realignment requires much less communication under BLOCK distribution. Secondly, the compiler should minimize time spent in moving data array `D` for the execution of the assignment statement. For instance, with compiler optimization, the logical data movement specified by the `EOSHIFT` operation can be turned into local memory accesses as a result of the alignment directives for arrays `C` and `D`.

A simple but naive approach is using general communication or array copying through temporary storage whenever non-canonical data layout is in force. This approach not only causes excessive data copying but also ignores many opportunities for fast communication.

We propose an algebraic transformation called XFORM-1 to reduce intra- and inter-procedural data movement in HPF programs. Figure 3 gives an overview of the optimization. The theoretical framework within which we designed program transformations is algebraic analysis of data movement. We give each of HPF's array operations and data distribution directives an algebraic representation. We then formalize data distribution, intra-procedural and inter-procedural data movement using *communication expressions*. We have developed a *communication algebra* and its associated heuristics to simplify communication expressions, and a hand-coded, optimized runtime communication library to carry out aggregate communication. Fast communication is uncovered by pattern-matching with a set of *communication idioms.*

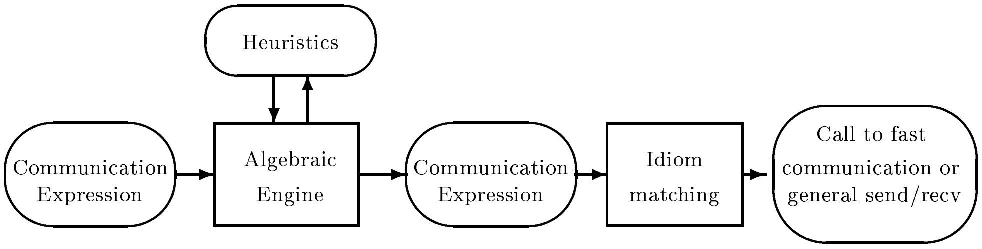


Figure 3: An Overview of XFORM-1 Algebraic Analysis

2.1 Algebraic Representations

The model of data mapping in HPF is that there is a two-level mapping of array elements to logical processors. The user can *align* array elements to a *template*, which is then partitioned and *distributed* onto an array of logical processors. The mapping of logical processors to physical processors is implementation dependent and may be specified by optional physical-mapping directives. We will use the term "*data layout*" as a generic term for the composition of the alignment and distribution (and physical mapping, if given explicitly).

Class	Operator	Domain	Codomain	Definition
ALIGN1	$\texttt{EOSHIFT}(c)$	D	$D+c$	$(i) \mapsto (i+c)$
	$\texttt{CSHIFT}(c)$	D	D	$(i) \mapsto \texttt{lb}(D) + (i - \texttt{lb}(D) + c) \bmod \lvert D\rvert$
	$\texttt{REFLECT}$	D	D	$(i) \mapsto \texttt{lb}(D) + \texttt{ub}(D) - i$
	$\texttt{STRIDE}(a,c)$	D	$aD+c$	$(i) \mapsto (a*i+c)$
ALIGN2	$\texttt{TRANS}^{(n)}(M)$	D	$M(D)$	$(i_1,\ldots,i_n) \mapsto M(i_1,\ldots,i_n)$
	$\texttt{SKEW}(n)(M)$	D	$M(D)$	$(i_1,\ldots,i_n) \mapsto M(i_1,\ldots,i_n)$
	$\texttt{CSKEW}^{(n)}(M)$	D	$\texttt{Mod}(M(D), \lvert D\rvert)$	$(i_1,\ldots,i_n) \mapsto ((M_1 \cdot I) \bmod m_1,$ $\ldots,(M_n \cdot I) \bmod m_n)$ where m_k are the sizes of D_k
ALIGN3	$\texttt{REPLICATE}^{(n)}(V, D_k)$	$D_1 \times \ldots D_{k-1}$ $\times D_{k+1} \ldots \times D_n$	$D_1 \times \ldots \times D_{k-1}$ $\times D_k \times D_{k+1} \ldots \times D_n$	$(i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_n) \mapsto$ $(i_1,\ldots,i_{k-1},\texttt{lb}(D_k):\texttt{ub}(D_k),i_{k+1},\ldots,i_n)$ where $V(k) \neq 0$
	$\texttt{EMBED}^{(m,n)}(M,E)$	D	E	$(i_1,\ldots,i_n) \mapsto M(i_1,\ldots,i_n)$
DIST	$\texttt{BLOCK}(b)$	D	$P \times L$	$(i) \mapsto (i \text{ div } b, i \bmod b)$
	$\texttt{CYCLIC}(p)$	D	$P \times L$	$(i) \mapsto ((i \bmod p, i \text{ div } p)$
	$\texttt{BCYCLIC}(b,p)$	D	$P \times L$	$(i) \mapsto (i \text{ div } b) \bmod p,$ $(i \text{ div } (p*b)) * b + i \bmod b)$
	$\texttt{SEQ}$	D	$[0] \times D$	$(i) \mapsto (0,i)$

Table 1: Algebraic Notations and Definitions of Some Array Intrinsics and Layout Operators, where $\texttt{lb}(D)$ and $\texttt{ub}(D)$ denote the lower and upper bound of interval domain D, $aD+c$ denotes an interval domain of range $[a*\texttt{lb}(D)+c..a*\texttt{ub}(D)+c]$, $M(D)$ constructs a multi-dimensional domain by permuting or skewing domain D according to the integer matrix M

We identify the different algebraic properties of HPF array intrinsics and data layout directives and divide them into several classes, as shown in Table 1. ALIGN1 are the operators whose domain and range are one-dimensional. For instance, both the array intrinsic `EOSHIFT`(A,dim,-c) and the alignment directive ALIGN A(i) with T(i+c) offsets array `A` by distance c, therefore both are denoted by `EOSHIFT`(c). ALIGN2 are multi-dimensional operators whose domain and range have the same dimensionality. Typical examples are the transpose operations, either as `TRANSPOSE` array intrinsics or as "transpose" alignment directives. ALIGN3 operators are for reshaping arrays, replicating arrays, or embedding lower-dimensional arrays into higher-dimensional ones. An ALIGN3 operator has different shapes for its argument and the result array. For instance, both the array intrinsic `SPREAD`(A,dim=2,ncopies=n) and the alignment directive ALIGN A(i) WITH T(i,*) (assuming the size of T at the second dimension is n) replicate n copies of `A` along the second dimension, and therefore, both are denoted by $\texttt{REPLICATE}^{(2)}(\begin{pmatrix} 0 \\ 1 \end{pmatrix}, \texttt{Interval}(1,n))$, where the nonzero element (the second element) in the vector argument indicates that the replication takes place at the second dimension.

The standard distribution directives in HPF are `BLOCK`, `CYCLIC` and generic `BLOCK_CYCLIC` distribution strategies. We collect them in the class DIST. MULTID operations capture multi-dimensional array intrinsics and data layouts that can be formulated as "product" of one-dimensional operators. For instance, a two-dimensional shift operation `CSHIFT(CSHIFT(A,dim=1,shift=-c1),dim=2,shift=-c2)` is denoted by the product of the two ALIGN1 operators `CSHIFT(c1)` and `CSHIFT(c2)`.

2.2 Communication Expressions

We formalize data layout, intra-procedural and inter-procedural data movement as *communication expressions* by functional compositions of the algebraic representations for the array operators and layout operators. In order to formalize the relationship between a data element and a concrete store within a processor, it is necessary to formalize the three stages of data mapping (alignment to template, partitioning template to logical processors, and then mapping logical processors to physical processors). Let α denote the alignment operator which aligns array D to template E, β denote the partition operator which partitions template E into a pair of logical processors L and local index domain M, and γ denote the operator which maps logical processor-memory pairs ($L \times M$) to physical processor-memory pairs ($P \times M$). The following lists the communication expressions we have defined under the "owner computes" rule.

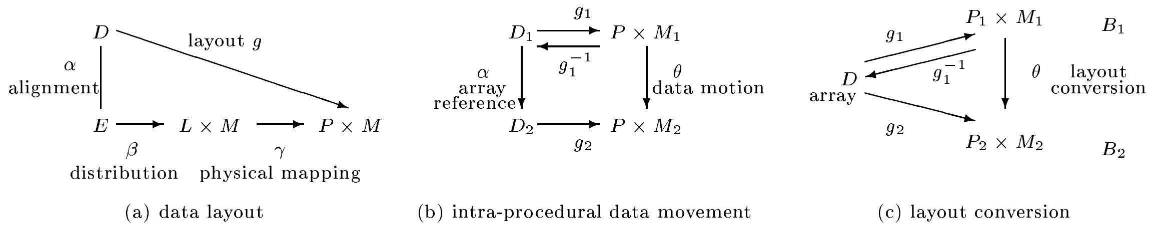


Figure 4: Commuting Diagrams for Data Layout, Intra-Procedural Data Movement, and Inter-Procedural Layout Conversion, where D and E are index domains, L is logical processor domain, P is physical processor domain, and M is local index domain within a processor.

Layout. The *layout* of an array D can be formally defined as a communication expression $g = \gamma \circ \beta \circ \alpha$, as shown in the commuting diagram of Figure 4(a).

Intra-Procedural Data Movement. *Intra-procedural data movement* refers to array references within a given procedure body. Let an array reference from array D_1 (used) to array D_2 (defined) be denoted by α and let their layouts to the machine be g_1 and g_2, respectively. Then the data movement induced by the reference α is given by the communication expression $\theta = g_2 \circ \alpha \circ g_1^{-1}$ as shown in Figure 4(b).

Inter-Procedural Data Movement. The second type of data movement, inter-procedural data movement, also called *layout conversion*, refers to array copying due to change of data layout between actual argument and dummy argument in procedure calls. Let the layout of array D in procedure B_1 and procedure B_2 be g_1 and g_2, respectively. Then the layout conversion of D from B_1 to B_2 is given by the communication expression $\theta = g_2 \circ g_1^{-1}$, as shown in Figure 4(c).

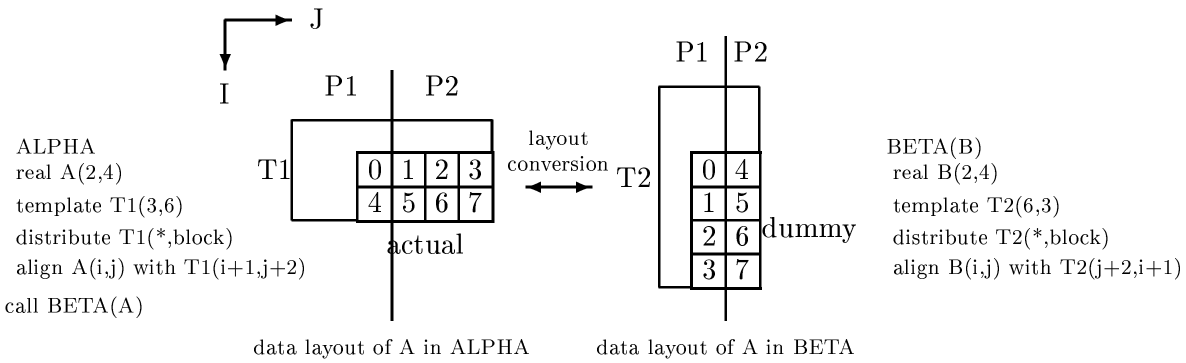
Most of the HPF alignment operators are *reshape morphisms* [20], which essentially are bijective functions defined over index domains. For an operator $f : D \rightarrow E$ which is injective but not bijective, a reshape morphism $f' : D \rightarrow \texttt{image}(D, f)$ can be derived from f where $\texttt{image}(D, f)$ is the image of D under f which is a subset of E (i.e. $f' : D \rightarrow \texttt{image}(D, f) \subset E$). Since f is bijective, let f^{-1} denote the inverse of f.

Example Figure 5 shows a simple communication expression for inter-procedural data movement. Intra-procedural data movement can be constructed similarly according to the commuting diagram in Figure 4(b). In Figure 5, we explicitly indicate the domain D and codomain E of each operator g (written as $g_{D \rightarrow E}$) because now the operators are bound to the index domain of array `A`. Array `A` is aligned with the template `T1`, which is partitioned into columns of blocks (denoted by $\texttt{SEQ}_{D_1+1 \rightarrow D_1+1} \times \texttt{BLOCK}(v)_{D_2+2 \rightarrow P \times V_1}$), by offseting one element at the first dimension and two elements at the second dimension (denoted by $(\texttt{EOSHIFT}(1)_{D_1 \rightarrow D_1+1} \times \texttt{EOSHIFT}(2)_{D_2 \rightarrow D_2+2})$). `T1` is then partitioned into columns of blocks (denoted by $\texttt{SEQ}_{D_1+1 \rightarrow D_1+1} \times \texttt{BLOCK}(v)_{D_2+2 \rightarrow P \times V_1}$). The dummy argument `B` is aligned with template `T2`, which has the same distribution as `T1`, by a transposition followed by an offseting alignment with distance 2 at the first dimension and with distance 1 at the second dimension (denoted by $(\texttt{EOSHIFT}(2)_{D_2 \rightarrow D_2+2} \times \texttt{EOSHIFT}(1)_{D_1 \rightarrow D_1+1}) \circ \texttt{TRANS}^{(2)}\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)_{D_1 \times D_2 \rightarrow D_2 \times D_1}$). The communication expression θ for changing from `A`'s layout to `B`'s is constructed by composing `B`'s layout θ_2 and the inverse of `A`'s layout θ_1 (refer to Figure 4(c)). θ represents the amount of communication in straightforward implementation where the communication is carried out by executing the operators in the expression one by one from right to left. For example, the expression θ in Figure 5 is the composition of five operators (i.e. of length five). Straightforward implementation will require two global address calculations and three data movements. In order to reduce communication, we need to reduce the number of operators in θ.

2.3 Communication Algebra

A generic method for simplifying a communication expression is using functional transformation [66, 11], where the operators are unfolded and reduced one by one following the reduction rules in λ-calculus. Since we only deal with the set of standard HPF data distribution directives(in stead of general functions) in the context of optimizing data movement, we look for a simpler and more efficient solution. We have designed a *communication algebra* to serve this purpose.

By the definition of communication expression, the amount of communication is represented by the number of operators in a *communication expression*; i.e., a communication expression is *simplified* if its operator count is decreased. Our goal is to minimize operator count by reducing away the operators



`index domains:`

$$D_1 = \mathtt{Interval}(1,2),\ D_2 = \mathtt{Interval}(1,4)$$

`actual's layout:`

$$\begin{aligned} g_1 &= (\mathtt{SEQ}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)_{D_2+2\to P\times V_1}) \\ &\circ (\mathtt{EOSHIFT}(1)_{D_1\to D_1+1} \times \mathtt{EOSHIFT}(2)_{D_2\to D_2+2}) \end{aligned}$$

`dummy's layout:`

$$\begin{aligned} g_2 &= (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_2}) \\ &\circ (\mathtt{EOSHIFT}(2)_{D_2\to D_2+2} \times \mathtt{EOSHIFT}(1)_{D_1\to D_1+1}) \circ \mathtt{TRANS}^{(2)}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1\times D_2\to D_2\times D_1} \end{aligned}$$

`Communication expression for changing from actual's layout to dummy's:`

$$\begin{aligned} \theta &= g_2 \circ g_1^{-1} \\ &= (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_2}) \\ &\circ (\mathtt{EOSHIFT}(2)_{D_2\to D_2+2} \times \mathtt{EOSHIFT}(1)_{D_1\to D_1+1}) \circ \mathtt{TRANS}^{(2)}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1\times D_2\to D_2\times D_1} \\ &\circ (\mathtt{EOSHIFT}(1)^{-1}_{D_1\to D_1+1} \times \mathtt{EOSHIFT}(2)^{-1}_{D_2\to D_2+2}) \\ &\circ (\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_1}) \end{aligned}$$

Figure 5: Communication Expressions for Inter-Procedural Layout Conversion. Array `A` defined in procedure `ALPHA` is passed as the actual argument to procedure `BETA`, which has a different layout specification for array `A`.

whose data movements can either be cancelled out with each other or merged into other operators. For example, as we will show in Figure 7, the amount of communication in expression θ of Figure 5 can be reduced to a single array transpose operator by reducing away the two multi-dimensional `EOSHIFT` operators on the left and the right of the `TRANS` operator. Since there are many array operators and layout operators in the context of Fortran90/HPF, a systematic simplification approach is desirable. We have designed a *communication algebra* for this purpose.

The *communication algebra* consists of a set of sub-algebras and bridging sub-algebras. Based on our classification of array operators and layout operators, we design sub-algebras for each class to manipulate the interaction of operators within that class, as well as bridging sub-algebras to manipulate the interaction of operators from different classes. Each sub-algebra contains three kinds of rules: (1) the *inverse* rules that compute the inverse of an operator, (2) the *reduction* rules that reduce two adjacent operators to one or zero new operator, and (3) the *exchange* rules that make two operators adjacent to each other by exchanging with other operators in between, so that they may be reduced later by the *reduction* rules.

Figure 6 lists the algebraic rules in the sub-algebra for ALIGN1 operators. Full description of the communication algebra is presented in [65].

Rule 1 Inverse of ALIGN1 Operators

(1) $\texttt{EOSHIFT}(c)^{-1}_{D\to D+c} = \texttt{EOSHIFT}(-c)_{D+c\to D}$

(2) $\texttt{CSHIFT}(c)^{-1}_{D\to D} = \texttt{CSHIFT}(-c)_{D\to D}$

(3) $\texttt{REFLECT}^{-1}_{D\to D} = \texttt{REFLECT}_{D\to D}$

(4) $\texttt{STRIDE}(a,c)^{-1}_{D\to a*D+c} = \texttt{STRIDE}(\frac{1}{a}, -\lfloor\frac{c}{a}\rfloor)_{a*D+c\to D}$

Rule 2 Reduction of Adjacent ALIGN1 Operators

(1) $\texttt{EOSHIFT}(c1)_{D+c2\to D+c2+c1} \circ \texttt{EOSHIFT}(c2)_{D\to D+c2} = \texttt{EOSHIFT}(c1+c2)_{D\to D+c2+c1}$

(2) $\texttt{CSHIFT}(c1)_{D\to D} \circ \texttt{CSHIFT}(c2)_{D\to D} = \texttt{CSHIFT}(c1+c2)_{D\to D}$

(3) $\texttt{REFLECT}_{D\to D} \circ \texttt{REFLECT}_{D\to D} = \texttt{id}_D$

(4) $\texttt{STRIDE}(a1,c1)_{a2*D+c2\to a1*a2*D+a1*c2+c1} \circ \texttt{STRIDE}(a2,c2)_{D\to a2*D+c2}$
$= \texttt{STRIDE}(a1*a2, a1*c2+c1)_{D\to a1*a2*D+a1*c2+c1}$

(5) $\texttt{STRIDE}(a,b)_{D+c\to a*D+a*c+b} \circ \texttt{EOSHIFT}(c)_{D\to D+c}$
$= \texttt{STRIDE}(a, a*c+b)_{D\to a*D+a*c+b}$

(6) $\texttt{EOSHIFT}(c)_{a*D+b\to a*D+b+c} \circ \texttt{STRIDE}(a,b)_{D\to a*D+b}$
$= \texttt{STRIDE}(a, b+c)_{D\to a*D+b+c}$

Rule 3 Exchange of ALIGN1 Operators

(1) $\texttt{CSHIFT}(c2)_{D+c1\to D+c1} \circ \texttt{EOSHIFT}(c1)_{D\to D+c1}$
$= \texttt{EOSHIFT}(c1)_{D\to D+c1} \circ \texttt{CSHIFT}(c2)_{D\to D}$

(2) $\texttt{CSHIFT}(c)_{D\to D} \circ \texttt{REFLECT}_{D\to D} = \texttt{REFLECT}_{D\to D} \circ \texttt{CSHIFT}(c)_{D\to D}$

(3) $\texttt{EOSHIFT}(c)_{D\to D+c} \circ \texttt{REFLECT}_{D\to D} = \texttt{REFLECT}_{D+c\to D+c} \circ \texttt{EOSHIFT}(c)_{D\to D+c}$

(4) $\texttt{STRIDE}(a,c)_{D\to a*D+c} \circ \texttt{REFLECT}_{D\to D}$
$= \texttt{REFLECT}_{a*D+c\to a*D+c} \circ \texttt{STRIDE}(a,c)_{D\to a*D+c}$

(5) $\texttt{STRIDE}(a,c2)_{D\to a*D+c2} \circ \texttt{CSHIFT}(c1)_{D\to D}$
$= \texttt{CSHIFT}(c1)_{a*D+c2\to a*D+c2} \circ \texttt{STRIDE}(a,c2)_{D\to a*D+c2}$

Figure 6: Algebraic Rules for ALIGN1

Examples

$$
\begin{aligned}
& \texttt{CSHIFT}(2)_{D+1\to D+1} \circ \texttt{EOSHIFT}(1)_{D\to D+1} \circ \texttt{CSHIFT}(2)^{-1}_{D\to D} & \\
= \; & \texttt{CSHIFT}(2)_{D+1\to D+1} \circ \texttt{EOSHIFT}(1)_{D\to D+1} \circ \texttt{CSHIFT}(-2)_{D\to D} & \text{By Rule 1(2)} \\
= \; & \texttt{CSHIFT}(2)_{D+1\to D+1} \circ \texttt{EOSHIFT}(1)_{D\to D+1} \circ \texttt{CSHIFT}(-2)_{D\to D} & \\
= \; & \texttt{EOSHIFT}(1)_{D\to D+1} \circ \texttt{CSHIFT}(2)_{D\to D} \circ \texttt{CSHIFT}(-2)_{D\to D} & \text{By Rule 3(1)} \\
= \; & \texttt{EOSHIFT}(1)_{D\to D+1} & \text{By Rule 2(2)}
\end{aligned}
$$

The compiler simplifies a communication expression by applying a sequence of the algebraic rules. The number of operators in a communication expression (and therefore the complexity of the algebraic simplification) depends on the number of levels in nested procedure calls, whose values may range from small constants to larger numbers depending on the application programs. For efficiency of the compiler, it is desirable to minimize the execution time of the simplification procedure. We use a simple heuristic to solve this problem.

Our current heuristic employs a greedy algorithm which reduces immediately reducible operators as early as possible because that always reduces operator count. The algorithm also avoids infinite looping by adjusting the starting pointer after application of each exchange rule. Note that MULTID operators are denoted by the product of one-dimensional operators (e.g. product of ALIGN1, DIST, etc.). The simplification of the composition of MULTID operators proceeds by simplifying each product terms independently. Details of the simplification algorithm is presented in [65].

2.4 Communication Idiom Matching

A simplified communication expression contains the actual data movement that needs be performed. A naive approach is use general communication for all cases. This approach ignores any opportunity for fast communication. A better approach is to uncover frequently occurring data movement and use specialized, fast communication whenever possible. For instance, the simplified communication expression θ shown previously is a transposition of a two-dimensional matrix which is partitioned one-dimensionally, resulting in so-called *all-to-all personalized communication* [37], in which every processor exchanges distinct data with every other processor. Due to the uniform communication patterns, communication overhead may be reduced by carefully scheduling messages to avoid contention in the network.

Since the advent of massively parallel machines, many researchers (e.g. [30]) have developed specialized communication routines to facilitate direct programming of distributed-memory machines. In building compilers, we might take advantage of these hand-crafted, highly optimized routines which become part of the runtime system for the language. In the Crystal compiler developed at Yale University [44, 45, 46], this approach is used to generate intra-procedural communication. We extend that work further to include those communication routines for converting data layouts between subprograms.

We have collected a set of frequently occurring communication patterns, and extracted the contents of their communication expressions into *communication idioms*. They include most of the array intrinsics and frequently occurring layout conversions such as conversion between `BLOCK` and `CYCLIC` partitioning and conversion between column partition (*,BLOCK) and row partition (BLOCK,*). These idioms may or may not have specialized, fast communication, perhaps microcoded or otherwise hand-optimized, depending on the target machine. A list of communication idioms is shown in Table 2. The optimization procedure simply goes through this list of idioms and pattern matches with the communication expression.

Idioms	Data Movement
$\gamma_1 \circ \gamma_2^{-1}$	change physical mapping
$\times^d(\gamma_{i1} \circ \beta_{i1} \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	change partition
$\gamma_1 \circ \beta_1 \circ \texttt{EOSHIFT}(c) \circ \beta_2^{-1} \circ \gamma_2^{-1}$	end-of-shift
$\gamma_1 \circ \beta_1 \circ \texttt{CSHIFT}(c) \circ \beta_2^{-1} \circ \gamma_2^{-1}$	cyclic shift
$\gamma_1 \circ \beta_1 \circ \texttt{REFLECT} \circ \beta_2^{-1} \circ \gamma_2^{-1}$	reversal permutation
$\times^d(\gamma_{i1} \circ \beta_{i1} \circ \texttt{EOSHIFT}(c_i) \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	multi-dimensional end-of-shift
$\times^d(\gamma_{i1} \circ \beta_{i1} \circ \texttt{CSHIFT}(c_i) \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	multi-dimensional cyclic shift
$\times^d(\gamma_{i1} \circ \beta_{i1} \circ \texttt{REFLECT} \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	multi-dimensional reflection
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \texttt{TRANS}^{(d)}(M^d) \circ \times^d(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	matrix transpose
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \texttt{SKEW}^{(d)}(M^d) \circ \times^d(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	skewing
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \texttt{CSKEW}^{(d)}(M^d) \circ \times^d(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	cyclic skewing
$\times^d(\gamma_i \circ \beta_i) \circ \texttt{REPLICATE}^{(d)}(V^d, D) \circ \times^m(\beta_j^{-1} \circ \gamma_j^{-1})$	replication
$\gamma_1 \circ \beta_1 \circ \texttt{RESHAPE}^{(d,1)}(D, M^d, \texttt{interval}(1,n), I) \circ \times^d(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$	axis combining
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \texttt{RESHAPE}^{(1,d)}(\texttt{interval}(1,n), I, D, M^d) \circ \beta_2^{-1} \circ \gamma_2^{-1}$	axis spliting

Table 2: Communication Idioms, where α denotes alignment operators or array references, β denotes distribution operators, γ denotes physical mapping operators, and $\times^d(a_i \circ b_i)$ denotes $(a_1 \circ b_1) \times \ldots \times (a_d \circ b_d)$.

Example Figure 7 shows the transformation result for the inter-procedural layout conversion given in Figure 5. By exchanging the `TRANS` operator with the two-dimensional `EOSHIFT` operators (using one of the *exchange* rules), the two `EOSHIFT` operators become adjacent and can be cancelled out with each other (using one of the *reduction* rules). The simplified communication expression matches with the idiom for matrix transpose where the matrix is partitioned one-dimensionally. In the transformed program, calls to a communication routine `matrix-transpose-1d-partition` are inserted to move array `A` to the proper layout before calling `BETA` and restore array `A`'s layout after returning from `BETA`.

2.5 Summary

Two major optimization primitives in XFORM-1 are algebraic simplification of communication expressions and idiom matching for fast communication. Most of the communication idioms we have collected are not architecture-specific. They may or may not have specialized, fast communication, depending on the target machine. Specialized implementation of communication routines may not have significant impact on more regular communication architectures. As a result, idiom matching may not be crucial to achieving high performance on this kind of machines. On the other hand, algebraic simplification is a high-level, abstract transformation technique that carries out data movement reduction within the purely logical, global space defined in the program. Any redundant layout conversions between procedure calls and any unnecessary local copying through canonical temporary storage will be reduced away abstractly by algebraic simplification. Consequently, even on a very balanced, regular communication architecture, communication overhead can still be reduced by high-level pattern matching and algebraically simplifying them.

procedure ALPHA in Figure 5

```
source program                      transformed program
real A(2,4)                         real A'(2:3,3:6),TEMP(3:6,2:3)
template T1(3,6)
align A(i,j) with T1(i+1,j+2)
                                    call matrix-transpose-1d-partition(TEMP,A')
call BETA(A)                        call BETA(TEMP)
                                    call matrix-transpose-1d-partition(A',TEMP)
```

Simplification procedure

$$\begin{aligned}
\theta\ &= (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \\
&\circ (\mathtt{EOSHIFT}(2)_{D_2\to D_2+2} \times \mathtt{EOSHIFT}(1)_{D_1\to D_1+1}) \\
&\circ \mathtt{TRANS}^{(2)}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1\times D_2\to D_2\times D_1} \\
&\circ (\mathtt{EOSHIFT}(-1)_{D_1+1\to D_1} \times \mathtt{EOSHIFT}(-2)_{D_2+2\to D_2}) \\
&\circ ((\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2})
\end{aligned}$$

By Rule: exchange of MULTID and ALIGN2

$$\begin{aligned}
&= (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \\
&\circ \mathtt{TRANS}^{(2)}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1+1\times D_2+2\to D_2+2\times D_1+1} \\
&\circ (\mathtt{EOSHIFT}(1)_{D_1\to D_1+1} \times \mathtt{EOSHIFT}(2)_{D_2\to D_2+2}) \\
&\circ (\mathtt{EOSHIFT}(-1)_{D_1+1\to D_1} \times \mathtt{EOSHIFT}(-2)_{D_2+2\to D_2}) \\
&\circ ((\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2})
\end{aligned}$$

By Rule: product composition exchange

$$\begin{aligned}
&= (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \\
&\circ \mathtt{TRANS}^{(2)}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1+1\times D_2+2\to D_2+2\times D_1+1} \\
&\circ ((\mathtt{EOSHIFT}(1)_{D_1\to D_1+1} \circ_1 \mathtt{EOSHIFT}(-1)_{D_1+1\to D_1}) \\
&\times (\mathtt{EOSHIFT}(2)_{D_2\to D_2+2} \circ_1 \mathtt{EOSHIFT}(-2)_{D_2+2\to D_2})) \\
&\circ ((\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2})
\end{aligned}$$

By Rule 2: reduction of ALIGN1 operators

$$\begin{aligned}
&= (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \\
&\circ \mathtt{TRANS}^{(2)}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1+1\times D_2+2\to D_2+2\times D_1+1} \\
&\circ ((\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2})
\end{aligned}$$

Match Idiom: matrix transposition

Figure 7: Algebraic Simplification for Inter-Procedural Layout Conversion

3 XFORM-2 Transformations

In previous section, we have presented the XFORM-1 module for optimizing data motion. Note that XFORM-1 transformations are done without explicitly mentioning the local processor IDs or any details of the local processor memory — that is because manipulations of alignments, array operations, and their communication expressions can be carried out within the purely logical space defined in the program. Also, changing from one distribution to another is a task that's handed over to the runtime system, working at the logical descriptor level only.

The purpose of XFORM-2 optimizations is to further exploit machine-dependent parallelism that otherwise would be difficult to express using HPF's data-parallel language constructs. Since these optimizations are often machine-dependent, it is necessary to partition the global space into local subspaces within processors, making processor IDs and local memory offsets visible to the compiler. We call this the "P&V" style of transformation ("P" for loops over processors, and "V" for loops over local memory, perhaps with vector elements, within processors), after the introduced loops and loop variables for processors and local memory offsets.

The loop nests we consider in XFORM-2 transformations are called *iterative spatial loop* nests, which capture a broad class of HPF parallel loops. We have collected a set of optimizations which we have found may improve HPF code performance on distributed-memory parallel systems: (1) *processor-memory skewing*, which increases processor parallelism via skewing processor loops with local memory loops, (2) *block-cyclic permutation*, which reduces communication via automatically changing data layout for a class of reference patterns that frequently occur in divide-and-conquer algorithms, (3) *interleaved reduction*, that increases processor parallelism and reduces communication for a class of reductions that are carried over loops in which loop-carried dependence may exist, and (4) the familiar *increasing granularity* that increases amount of computation between synchronization via combination of strip-mining local memory loops and interchanging parallel processor loops outward.

In short, XFORM-2 optimizations can help to increase processor parallelism and reduce communication and synchronization in data-parallel loop nests. In this section, we will discuss the motivation for each of these optimizations, the conditions under which they can be applied, and the effect of each optimization.

3.1 Iterative Spatial Loops

The class of loop nests that may profit from XFORM-2 optimizations are called *iterative spatial loop nests*, which was first defined in [22].

In many HPF parallel loop nests, there exists a one-to-one mapping from the iteration space of the loop nest to each of the index domain of the arrays updated in the loop body. We call such loop nest *spatial loop nest*, because parallelism can be achieved by distributing the data and the loop iterations to multiple processors. A *spatial loop* that has loop-carried data dependence is called a *dependent spatial loop*, which implies that the computations have to be sequentialized due to data dependence. A *spatial loop* that does not have loop-carried dependence is a *parallel spatial loop* (or simply *parallel loop*), whose iterations can all be computed in parallel.

For instance, the I loop and the J loop in **Program 1** are *spatial loops* because there is a one-to-one mapping from the iteration space of the two loops to the index domain of array `a`. Loop-carried dependence exists in the `I` loop, which makes the `I` loop a *dependent spatial loop.* The `J` loop carries no dependence, and therefore is a *parallel spatial loop.*

Prog 1

```
real a(100,100)
DO T = 1,10
   DO I = 2,100
      FORALL J = 1:100
         a(I,J) = a(I,J)+a(I-1,J)
   END DO
END DO
```

Loop Nest I (Iterative Spatial Loop Nest)

```
***iterative loops***
DO_T (K_1 = a_1, b_1, c_1)
...
DO_T (K_l = a_l, b_l, c_l)
   ***dependent spatial loops***
   DO_S (I_1 = x_1, y_1, z_1)
   ...
   DO_S (I_m = x_m, y_m, z_m)
      ***parallel loops***
      DOALL_S (I_{m+1} = x_{m+1}, y_{m+1}, z_{m+1}) {
      ...
      DOALL_S (I_{m+n} = x_{m+n}, y_{m+n}, z_{m+n}) {
         A(I_1, ..., I_{m+n}) = τ[B(I_1 + c_1, ..., I_{m+n} + c_{m+n})]
```

Furthermore, in many scientific applications, the same loops may execute several times iteratively. This can be best expressed by enclosing the parallel loop by an outermost sequential loop. The sequential loop updates some array elements more than once and therefore is called an *iterative* loop, which implies that the iteration space of the loop is mapped to time steps, in stead of the index domain of the arrays. For instance, the `T` loop updates array `a` more than once, and thus is an *iterative loop*.

To facilitate XFORM-2 transformations, we denote a *dependent spatial loop* by `DO_S`, a *parallel spatial loop* by `DOALL_S`, and an *iterative* loop by `DO_T`.

Iterative Spatial Loop Nest. An *iterative spatial loop nest* consists of, from outermost level inward, zero or more levels of *iterative loops* (or *temporal loop*) followed by zero or more levels of *dependent spatial loops*, and then followed by one or more levels of *parallel spatial loops*, or simply *parallel loops*, as shown in **Loop Nest I** (which consists of l iterative loops, m dependent spatial loops and n parallel loops, where $l \geq 0, m \geq 0$, and $n \geq 1$).

In **Loop Nest I**, a_i, b_i, c_i and x_j, y_j, z_j are the usual loop lower bound, upper bound and stride. The notation $\tau[b]$ denotes an expression containing b. The offsets c_j in an array index expression must be an expression whose value stays invariant through the entire spatial loop nest. For simplicity, we assume that array indices at the left-hand-side are loop variables. There may be multiple instances of array references in an assignment statement and multiple assignment statements in the loop body, and the right-hand-side array may be either a different array or the same as the left-hand-side. Therefore, *iterative spatial loop nests* capture a broad class of HPF parallel loops, including *forall* loops and *independent* loops.

Loop Nest P (Partitioned Iterative Spatial Loop Nest)

```
***outermost temporal loops***
DO_T (K_1 = a_1, b_1, c_1)
...
DO_T (K_l = a_l, b_l, c_l)
  ***partitioned dependent spatial loops***
  DO_P (I_1 = 0, P_1 - 1)
  DO_V (J_1 = 0, V_1 - 1)
    ...
    DO_P (I_m = 0, P_m - 1)
    DO_V (J_m = 0, V_m - 1)
      ***partitioned parallel spatial loops***
      DOALL_P (I_{m+1} = 0, P_{m+1} - 1)
      DOALL_V (J_{m+1} = 0, V_{m+1} - 1)
        ...
        DOALL_P (I_{m+n} = 0, P_{m+n} - 1)
        DOALL_V (J_{m+n} = 0, V_{m+n} - 1)
          A(I_1, J_1, ..., I_{m+n}, J_{m+n}) = τ[B(Î_1, Ĵ_1, ..., Î_{m+n}, Ĵ_{m+n})]
```

Partitioned Loop Nest. XFORM-2 optimizations are aware of the notion of processors and local memory loops, therefore, it is necessary to partition the *iterative spatial loops.* Once data are partitioned, spatial loop nests also need to be partitioned. A spatial loop is partitioned by splitting the loop into a processor loop (a `P` loop) and a local memory loop (a `V` loop). The index in a processor loop gives the ID of the processor which computes the associated local memory loop. Considering blocked partitioning, a dependent spatial loop (`DO_S` loop) is split into a `DO_P` loop and a `DO_V` loop, meaning that there are dependences between processors, while a parallel loop (`DOALL_S` loop) is split into a `DOALL_P` loop and a `DOALL_V` loop, meaning that all processors can be active concurrently. **Loop Nest P** shows a general partitioned loop nest.

Perfect Parallel Loops. A parallel loop that satisfies the following conditions is a *perfect* parallel loop: (1) all array subscripts in the loop body along its dimension are identity functions of its loop variable, and (2) the bounds of the loop does not depend on the loop variables of the loops between itself and the outer dependent loop. The parallel processor loop `DOALL_P` and the local memory loop `DOALL_V` tiled from a perfect parallel loop are called *perfect processor loop* and *perfect memory loop* respectively.

Next, we present the set of XFORM-2 optimizations: *processor-memory skewing*, *block-cyclic permutation*, *interleaved reduction*, and *increasing granularity.*

3.2 Processor-Memory Skewing

Consider an input loop nest with a dependent spatial loop `DO_S` to be partitioned into non-degenerate processor loop `DO_P`. Such case may arise from a loop nest that inherits the data layout of a previous loop nest where the same array sections were updated in parallel in the previous loop nest and must be now updated sequentially. Consider **Program** 2 in Figure 8. Data dependence exists in the outer loop `DO_S`. Since the array is partitioned into two-dimensional blocks as specified by the `DISTRIBUTE` directive, the two spatial loops will also be partitioned. The outer loop is partitioned into a pair of `DO_P` and `DO_V` loops, and the inner loop into a pair of DOALL_P and DOALL_V loops. The inner loop iterations can be computed in parallel. However, without optimization, execution of outer loop between the processors in the DO_P loop has to be serialized. For this example, our goal is to increase processor parallelism for the execution of the outer loop.

Processor-memory skewing is a technique that may reclaim such wasted processor resources due to inheritance of data layout without paying the cost of data layout conversion. This optimization is an application of *loop skewing* technique [63] in the context of partitioned iterative spatial loop nests. *Processor-memory skewing* skews a sequential processor loop DO_P with respect to a local-memory loop (DOALL_V or DO_V), resulting in a DO_V loop denoting the stages of pipelining and a DOALL_P loop denoting the active processors at each pipeline stage, as shown in Figure 8(a).

Let P be the range of the sequential processor loop DO_P and V be the range of the local memory loop. Processor-memory skewing increases processor parallelism by a factor of $\frac{V*P}{P+V-1}$. When $V \gg P$, processor parallelism is increased by P.

In the following, we present **Procedure processor-memory-skew** which skews a sequential processor loop DO_P and a local memory loop. We assume one level of DO_P loop.

Procedure processor-memory-skew

Input: a partitioned loop nest which contains one DO_P loop.

Output: a new partitioned loop nest if pattern-matching is successful, or else the same input loop nest.

1. Search in the loop nest enclosed by the DO_P loop for a *perfect memory loop* DOALL_V. If no such loop exists, then search for a sequential memory loop DO_V that satisfies the following conditions: (1) no DO_P loop is tiled from the same DO_S loop as the DO_V, that is, the whole DO_S loop is assigned to a single processor, and (2) the DO_V loop can be legally moved next to the DO_P loop by loop interchanging (data dependence analysis [5, 8] can be used to decide the legality). If no such loop exists either, then exit.
2. Move the DOALL_V/DO_V loop outward to be adjacent to the DO_P loop by loop interchanging.
3. Perform loop skewing followed by a loop interchanging between the DO_P loop and the DOALL_V/DO_V loop, resulting in a DO_V loop outside a DOALL_P loop.

Example In **Program 2**, loop-carried data dependence exists in the outer loop, which imposes sequential ordering for the outer loop iterations, as shown by the iterative spatial loop nest (**Loop Nest 1**). The 2D-BLOCK partitioning results in a partitioned loop nest as shown in **Loop Nest 2**. Loop interchanging the DOALL_V(J_2) with the DO_V(I_2) and the DOALL_P(J_1) results in **Loop Nest 3** where the DO_P and DOALL_V loops become adjacent. Finally, **Loop Nest 4** shows the loop structure after *processor-memory skewing* on the DO_P and DOALL_V loops followed by a loop interchange. As shown in Figure 8(a), after processor-memory skewing, in each new DO_V(J_2') loop iteration, multiple processors can start computation in parallel. The pipelined computation among processors in **Loop Nest 4** is illustrated in Figure 8(b).

Adjusting Granularity of Pipelining. The granularity of pipeline parallelism is determined by the amount of computation enclosed by the new DOALL_P loop resulting from processor-memory skewing.

Prog 2

```
      REAL a(m,n),b(m,n)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: a, b
      DO i=2, m
        FORALL j=1,n
          a(i,j) = ... a(i-1,j) ...
        END FORALL
      END DO
```

Loop Nest 1 iterative spatial loops

$\texttt{DO\_S}\ (I = 2, m)\,\{$
  $\texttt{DOALL\_S}\ (J = 1, n)\,\{$
    $a(I, J) = ...a(I-1, J)...$

Loop Nest 2 partitioned loops

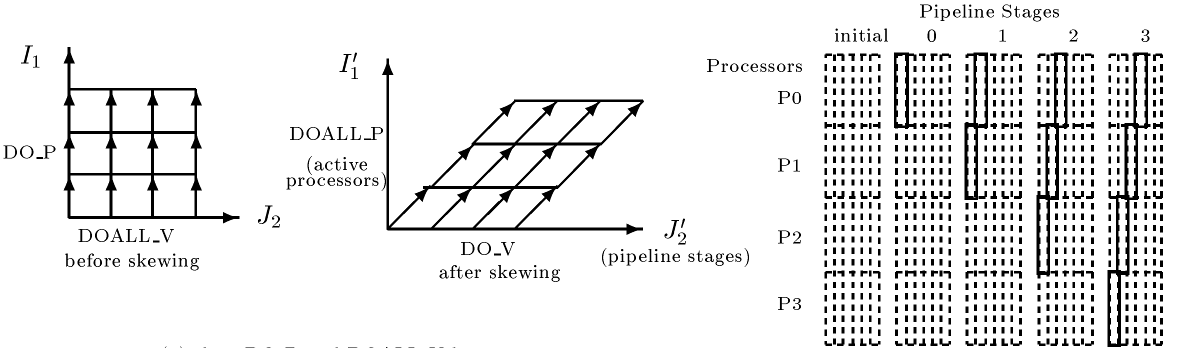
$\texttt{DO\_P}\ (I_1 = 0, p_1 - 1)\,\{$
  $\texttt{DO\_V}\ (I_2 = 0, v_1 - 1)\,\{$
    $\texttt{DOALL\_P}\ (J_1 = 0, p_2 - 1)\,\{$
      $\texttt{DOALL\_V}\ (J_2 = 0, v_2 - 1)\,\{$
        $\hat{a}(I_1, I_2, J_1, J_2) =$
          $...\hat{a}(I_1 + (I_2 - 1)\ \texttt{DIV}\ v_1,$
            $(I_2 - 1)\ \texttt{MOD}\ v_1, J_1, J_2)...$

Loop Nest 3 interchange DOALL_V outward

$\texttt{DO\_P}\ (I_1 = 0, p_1 - 1)\,\{$
  $\texttt{DOALL\_V}\ (J_2 = 0, v_2 - 1)\,\{$
    $\texttt{DO\_V}\ (I_2 = 0, v_1 - 1)\,\{$
      $\texttt{DOALL\_P}\ (J_1 = 0, p_2 - 1)\,\{$
        $\hat{a}(I_1, I_2, J_1, J_2) =$
          $...\hat{a}(I_1 + (I_2 - 1)\ \texttt{DIV}\ v_1, (I_2 - 1)\ \texttt{MOD}\ v_1, J_1, J_2)...$

Loop Nest 4 skew and interchange DO_P and DOALL_V

$\texttt{DO\_V}\ (J_2' = 0, p_1 + v_2 - 2)\,\{$
  $\texttt{DOALL\_P}\ (I_1' = \texttt{max}(0, J_2' - v_2), \texttt{min}(J_2', p_1 - 1))\,\{$
    $\texttt{DO\_V}\ (I_2 = 0, v_1 - 1)\,\{$
      $\texttt{DOALL\_P}\ (J_1 = 0, p_2 - 1)\,\{$
        $\hat{a}(I_1', I_2, J_1, J_2' - I_1') =$
          $...\hat{a}(I_1' + (I_2 - 1)\ \texttt{DIV}\ v_1, (I_2 - 1)\ \texttt{MOD}\ v_1, J_1, J_2' - I_1')...$



(a) skew DO_P and DOALL_V loops

(b) Pipelined computation among processors.

Figure 8: Pipelining via Processor-Memory Skewing

This granularity also determines the amount of communication overhead. That is, increase in pipeline parallelism also cause increase in communication overhead. Based on the observation that the unit communication vs. computation ratios on many modern machines are around two orders of magnitude, to balance parallelism with communication overhead, the CRAFT compiler uses a simple heuristic that works well for many parallel machines. It first increases pipeline parallelism by interchanging the *perfect processor loops* and *perfect memory loops* outside the `DO_P` loop to make pipelining as fine-grain as possible. It then increase pipeline granularity by strip-mining the local memory loop that will be skewed with the `DO_P` loop.

3.3 Increasing Granularity

The granularity of a loop nest is determined by the amount of computation enclosed by the innermost `DOALL_P`; that is, the amount of computation that can be performed without synchronization among processors. The purpose of this transformation is to increase amount of computation between synchronization.

This optimization adopts two existing techniques, *loop interchange* and *strip-mining* [63], each with proper refinement for partitioned iterative spatial loops. *Loop interchange* helps increase granularity by moving parallel processor loops (`DOALL_P` as outward as possible, while *strip-mining* increases amout of computation by grouping local-memory loop iterations into smaller number of chunks.

Increasing granularity by loop interchanging does not cause overhead; it can be performed as long as it does not change the semantics of the original loop nest, while strip-mining is usually used in combination with other optimizations (e.g. coarse-grain pipelining via strip-mining and processor-memory skewing). **Procedure increasing-granularity** increases granularity of loop nest by loop interchange.

Procedure increasing-granularity

Input: a partitioned loop nest.

Output: a new partitioned loop nest with `DOALL_P` loops moved outward maximally.

1. Search for parallel loops (`DOALL_P`, `DOALL_V`); move all `DOALL_P` loops outside the `DOALL_V` loops by loop interchange.
2. Move each *perfect processor loop* (i.e. `DOALL_P` loop tiled from a *perfect* parallel loop) outside all `DO_V`, `DO_P`, `DOALL_L` and `DOALL_V` loops.

Example **Procedure increase-granularity** interchanges outward the perfect processor loop (the inner `DOALL_P` loop) in **Loop Nest 4**, resulting in **Loop Nest 5**. Granularity is increased by a factor of p_2.

Loop Nest 5

$$
\begin{array}{l}
\texttt{DOALL\_P}\ (J_1 = 0, p_2 - 1)\,\{\\
\quad \texttt{DO\_V}\ (J_2' = 0, p_1 + v_2 - 2)\,\{\\
\quad\quad \texttt{DOALL\_P}\ (I_1' = \texttt{max}(0, J_2' - v_2), \texttt{min}(J_2', p_1 - 1))\,\{\\
\quad\quad\quad \texttt{DO\_V}\ (I_2 = 0, v_1 - 1)\,\{\\
\quad\quad\quad\quad \hat{a}(I_1', I_2, J_1, J_2' - I_1') =\\
\quad\quad\quad\quad\quad ...\hat{a}(I_1' + (I_2 - 1)\ \texttt{DIV}\ v_1, (I_2 - 1)\ \texttt{MOD}\ v_1, J_1, J_2' - I_1')...
\end{array}
$$

3.4 Block-Cyclic Permutation

In this subsection, we describe an optimization technique for input programs with communication offset being two to the power of an iterative loop index, e.g. $f(K) = 2^K$, a non-constant-offset shift operation. This type of communication pattern appears frequently in divide-and-conquer algorithms in which a large problem is recursively divided into smaller problems until the problem size is small enough so that the problem can be directly solved; the solutions to the small problems are then recursively combined to form the solution to the large problem. The communication usually occurs between elements of distances recursively doubled (or halved).

We use the Fast Fourier Transform (FFT) as an example to illustrate the optimization. **Loop Nest 7** shows the butterfly stages in FFT, and Figure 9 illustrates the data reference patterns for 8-input FFT.

Assuming both the number of data elements N and the number of processors P are power of two, if data elements are partitioned using `BLOCK` distribution, then in the first $\mathtt{log}P$ stages, each data element will need a remote data element for its computation, while in the final $\mathtt{log}N - \mathtt{log}P$ stages all data references results in local memory accesses, as shown in Figure 9(b). Alternatively, data elements may be partitioned using `CYCLIC` distribution, resulting in inter-processor communication in the first $\mathtt{log}P$ stages and local memory accesses in the rest of the stages, as shown in Figure 9(c).

Since the initial computation of `BLOCK` data distribution and the final computation of the `CYCLIC` data distribution do not require communication, a natural choice is a hybrid layout: `BLOCK` distribution for the first $\mathtt{log}P$ stages and `CYCLIC` distribution for the final $\mathtt{log}N - \mathtt{log}P$ stages. This hybrid layout strategy has been used in several hand-coded FFT routines [26, 59]. The `block-cyclic-permutation` optimization automates this process.

The hybrid layout leads to two data layout conversions that involve all-to-all communication: a `BLOCK`-to-`CYCLIC` conversion right after the $\mathtt{log}P$th stage and its inverse (`CYCLIC`-to-`BLOCK` conversion) right after the last stage to restore the initial `BLOCK` distribution, as shown in Figure 9(d).

In the following, we present **Procedure block-cyclic-permute**, which splits the range of a `DO_T`(K) loop that contains 2^K communication offsets at the dimension of the inner `DOALL_V` loops, and inserts block-cyclic conversion routine calls between splitted loop nests. For simplicity, we assume the 2^K communication offset only occur at one dimension of the inner `DOALL_V` loops, and all arrays in the loop nest have the same data distribution. We also assume the number of processors p is power of two and the number of data elements N is much larger than p (fortunately this is true in most application programs). **Procedure block-cyclic-permute** assumes initial BLOCK distribution for data arrays. The procedure for initial CYCLIC distribution is analogous.

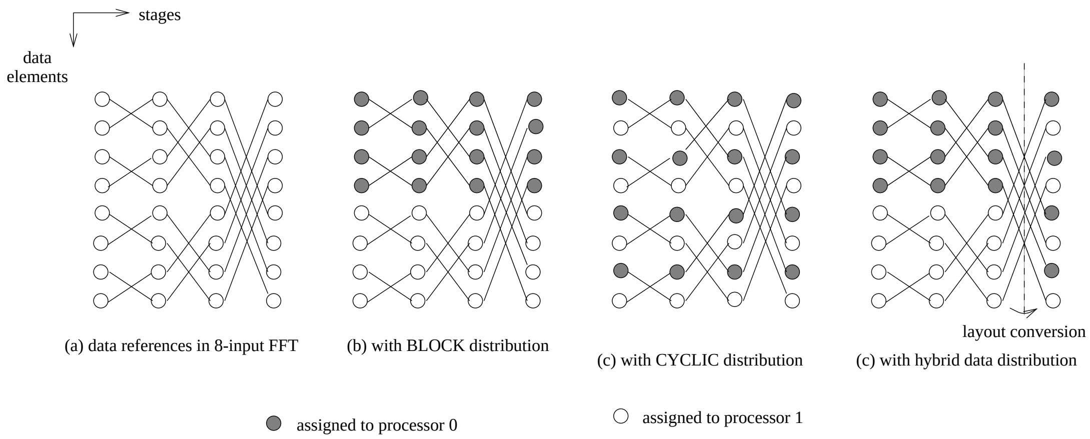
Procedure block-cyclic-permute

Input: a partitioned loop nest

Output: a sequence of partitioned loop nests if pattern-matching is successful, or else the same input loop nest.

1. Check the loop body for the existence of array reference offsets that are two to the power of the `DO_T` loop variable K. If no such reference pattern exist, exit. If there exist another reference pattern whose subscript expression at the same dimension as the 2^K-offset pattern is not the loop variable, exit.

2. Split the range of the `DO_T` loop using the pivot index $\mathtt{log}p$, where p is the number of processors along the dimension where the 2^K communication offsets occur (i.e., if K is in increasing order,



Loop Nest 6 iterative spatial loops

$\texttt{DO\_T}\ (K = 0, \texttt{log}N - 1)\,\{$
$\quad \texttt{DOALL\_S}\ (I = 0, N-1)\,\{$
$\quad\quad \ldots$
$\quad\quad \texttt{IF}\ (\texttt{is\_upper}(I))$
$\quad\quad\quad x(I) = w * x(I) + x(I + 2^K)$
$\quad\quad \texttt{ELSE}$
$\quad\quad\quad x(I) = -w * x(I) + x(I - 2^K)$

Loop Nest 7 partitioned loops

$\texttt{DO\_T}\ (K = 0, \texttt{log}N - 1)\,\{$
$\quad \texttt{DOALL\_P}\ (I_1 = 0, p-1)\,\{$
$\quad\quad \texttt{DOALL\_V}\ (I_2 = 0, v-1)\,\{$
$\quad\quad\quad \texttt{IF}\ (\texttt{is\_upper}(I_1, I_2))$
$\quad\quad\quad x(I_1, I_2) = w * x(I_1, I_2) +$
$\quad\quad\quad\quad x(I_1 + (I_2 + 2^K)\ \texttt{DIV}\ v, (I_2 + 2^K)\ \texttt{MOD}\ v)$
$\quad\quad\quad \texttt{ELSE}$
$\quad\quad\quad x(I_1, I_2) = -w * x(I_1, I_2) +$
$\quad\quad\quad\quad x(I_1 + (I_2 - 2^K)\ \texttt{DIV}\ v, (I_2 - 2^K)\ \texttt{MOD}\ v)$

Loop Nest 8 post-transformed loop nests

$\texttt{!! phase 1: block distribution}$
$\texttt{DO\_T}\ (K = 0, \texttt{log}p - 1)\,\{$
$\quad \texttt{DOALL\_P}\ (I_1 = 0, p-1)\,\{$
$\quad\quad \texttt{DOALL\_V}\ (I_2 = 0, v-1)\,\{$
$\quad\quad\quad \texttt{IF}\ (\texttt{is\_upper}(I_1, I_2))$
$\quad\quad\quad x(I_1, I_2) = w * x(I_1, I_2) +$
$\quad\quad\quad\quad x(I_1 + (I_2 + 2^K)\ \texttt{DIV}\ v, (I_2 + 2^K)\ \texttt{MOD}\ v)$
$\quad\quad\quad \texttt{ELSE}$
$\quad\quad\quad x(I_1, I_2) = -w * x(I_1, I_2) +$
$\quad\quad\quad\quad x(I_1 + (I_2 - 2^K)\ \texttt{DIV}\ v, (I_2 - 2^K)\ \texttt{MOD}\ v)$
$\texttt{block\_to\_cyclic\_conversion}(x)$
$!!\texttt{phase2}: \texttt{ cyclic distribution}$
$\texttt{DO\_T}\ (K = \texttt{log}p, \texttt{log}N - 1)\,\{$
$\quad \texttt{DOALL\_V}\ (I_2 = 0, v-1)\,\{$
$\quad\quad \texttt{DOALL\_P}\ (I_1 = 0, p-1)\,\{$
$\quad\quad\quad \texttt{IF}\ (\texttt{is\_upper}(I_1, I_2))$
$\quad\quad\quad x(I_1, I_2) = w * x(I_1, I_2) +$
$\quad\quad\quad\quad x(I_1, I_2 + (I_1 + 2^K)\ \texttt{DIV}\ p)$
$\quad\quad\quad \texttt{ELSE}$
$\quad\quad\quad x(I_1, I_2) = -w * x(I_1, I_2) +$
$\quad\quad\quad\quad x(I_1, I_2 + (I_1 - 2^K)\ \texttt{DIV}\ p)$
$\texttt{cyclic\_to\_block\_conversion}(x)$

Figure 9: Block-Cyclic Permutation for FFT. By automatic change between Block and Cyclic distribution, data movement within iterations are eliminated, in the expense of layout conversions between the splitted loop nests.

then split the loop into two, one with range $l \leq K < \texttt{log}p$ and the other with range $\texttt{log}p \leq K < u$, where l and u are the lower and upper bounds of the `DO_T` loop, respectively), and replicate the loop body.

3. Let A be the set of arrays in the loop body that are referenced with 2^k offsets. For each array in A and each array in the loop body that is referenced by any array in A, insert one `BLOCK_to_CYCLIC` conversion routine call between the two loop nests and its inverse conversion (`CYCLIC_to_BLOCK`) after the second loop nest.

4. Change the layout for the second new loop nest to CYCLIC distribution and simplify array subscript expressions.

Example Applying *block-cyclic permutation* optimization on **Loop Nest 7** results in **Loop Nest 8**. Data references in the second loop nest of **Loop Nest 8** all become local memory accesses, in the expense of layout conversions outside the two loop nests.

3.5 Interleaved Reduction

Many reductions (for instance, computing the `SUM` of n values) are commonly considered to have both internal associativity and commutativity. We can relax the ordering constraints of these reductions, increasing their available parallelism. For example, to compute `SUM` of n values using p processors, we can divide the n values into p chunks, assign a chunk to each processor, and every processor computes locally the sum of the values assigned to it, independent of other processors, and then at the end sum up their partial results using a global reduction operation.

Parallelization of reductions require language constructs or compiler techniques that can help break data dependences in reductions. One obvious compilation approach is pattern recognition. Either the source language includes explicit reduction operators (e.g. the `SUM` operator in High Performance Fortran), or certain specific loops are recognized as equivalent to known reductions (e.g. the loop for `SUM` reduction as described above). Once such patterns are recognized, hand optimized code for the reductions are emitted in the code generation phase.

Previous work on pattern recognition for reductions had been reported in the Parafrase system [43], the Eave [10] system, and the Fortran D system [60]. Redon and Feautrier proposed an algebraic specification method for recurrences detection [52]. Pinter and Pinter proposed a matching method based on Program Dependence Graphs [49]. Fisher and Ghuloum reported a method for extracting recurrences from loop structures that contain conditional statements [27, 31].

All these existing work only parallelize reduction loops that contain no stores that can overlap any loads within the loop (i.e. loops that only contain dependences caused by the reduction statement). For instance, the SUM loop given above can be parallelized straightforwardly by existing techniques. We call this class of reductions *basic* reductions. Not all reductions are *basic*, however. Consider the loop nest in `Program ??`, which solves a triangular linear system. A reduction `B(J)=B(J)-L(J,I)*x(I)` is carried over the outer loop indexed by `I`. Two kinds of data dependences exist in the outer loop: a loop-carried dependence caused by the reduction statement, and a loop-carried dependence caused by computing the value of `x(I)`, whose value depends on the value of array `B` computed in previous iterations. Existing techniques fail to parallelize this reduction due to the second kind of dependence.

`Interleaved Reduction` is a general technique for parallelizing reductions. The basic idea is to exploit partial parallelism embodied in reduction loops through combination of *data dependence analysis* and *region analysis*. Data dependence analysis identifies the condition that can trigger this optimizing transformation. Region analysis extracts partial parallelism by separating reduction iterations into a sequential region and an order-insensitive region. Parallelism is achieved by interleaving reduction iterations in the order-insensitive region onto multiple processors.

Detailed description of `Interleaved Reduction` can be found in [64]. The compiler transformation is briefly described as follows. The compiler detects reduction patterns using data dependence analysis. If the reduction is a basic reduction, it is parallelized straightforwardly. For non-basic reductions, the compiler performs **region_analysis** to separate the iteration domain into three subdomains: the sequential region, parallel-update region and global-reduction region. Then the compiler splits the loop into three loops corresponding to the three regions, and updates the loop bodies accordingly, and finally it inserts a global reduction operation at the end of the loop corresponding to the global-reduction region.

3.6 Meta-Transformation

The interaction of these optimization primitives poses a challenge to the compiler in finding an appropriate ordering in applying these optimization primitives. For instance, processor-memory skewing may obscure loop structure and therefore may hinder other optimizations that are sensitive to special reference patterns, such as block-cyclic permutation and interleaved reduction. Finding an optimal solution is a NP-complete problem. We have started to study this problem.

Current strategy used in the CRAFT compiler is somewhat ad hoc. Based on the impact on loop nests of each optimization primitive, we have found the following linear ordering of these primitives to be sufficient for catching most chances of optimizations. **Procedure block-cyclic-permute** and **Procedure interleaved-reduction** work on exclusive patterns along one dimension. However, mixed patterns may exist in different dimensions, e.g. a reduction at the first dimension and a butterfly computation at the second dimension. We apply **Procedure block-cyclic-permute** first since **Procedure interleaved-reduction** increase more number of loop nests. Next, we check whether a loop nest contains sequential processor loop `DO_P`. If it does, **Procedure processor-memory-skew** is applied if it improves program performance. Finally, **Procedure increase-granularity** will be applied to each transformed loop nest to increase granularity.

4 Preliminary Experience

In this section, we report the implementation status of this work and some experimental results from a set of benchmark codes demonstrating the effectiveness of algebraic simplification, communication idiom matching for layout conversion, and some XFORM-2 optimizations (processor-memory skewing, block-cyclic permutation, and interleaved reduction).

4.1 The CRAFT Compiler

The two-phase transformative framework is implemented in the context of the CRAFT compilers, as shown in Figure 10. The CRAFT compiler methodology uses a single semantic representation (SemRep) as an intermediate notation between the front end and code generation. SemRep is described by a Standard ML signature. CRAFT compiler front ends produce SemRep, their transformative phases (e.g. XFORM-1 and XFORM-2 transformations) apply black-box pattern-matching transformations over

SemRep, and the back end generates native object code for parallel target machines by partitioning and lowering the transformed SemRep code.

A CRAFT SIMD compiler for CM-5 has been implemented [21]. The front-end and the intermediate modules have been adopted for the CRAFT MIMD compiler. A back-end for CM-5 MIMD mode is under implementation from the SIMD basis.

With manual assistance, the CRAFT MIMD compiler generated message-passing vector codes by translating the final form of the post-transformation program (partitioned SemRep program) to a "nodal" CM-Fortran program (a SPMD programming model supported by CM-Fortran [24]) and inserting communication routine calls as appropriate. The communication routines were implemented using CMMD 3.0 primitives. For scalar code generation, the CRAFT compiler generated message-passing C codes with manual assistance.

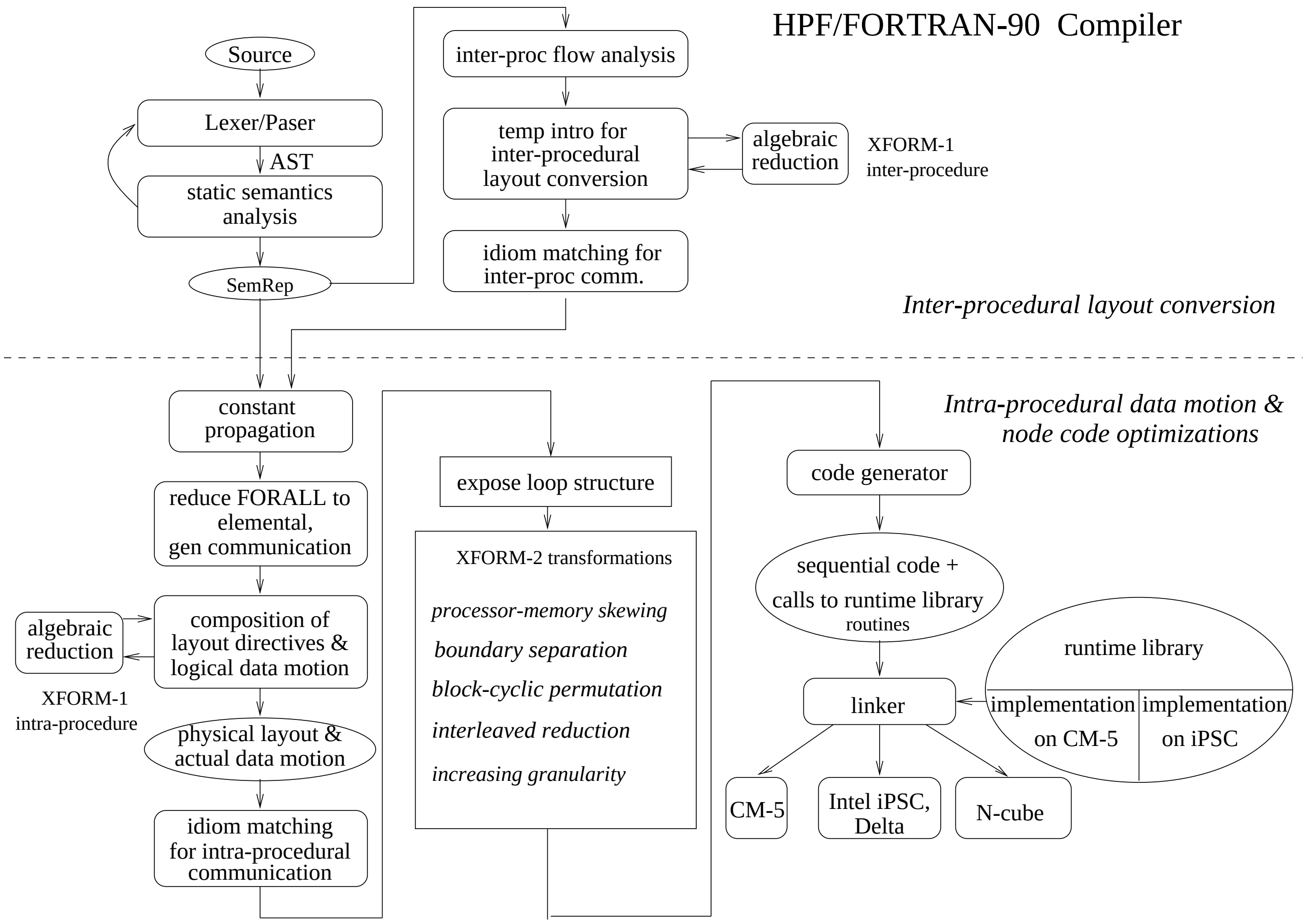


Figure 10: Organization of the CRAFT Compiler

Many of the communication idioms have been implemented as runtime communication routines for the CM-5 using CMMD 3.0 communication primitives, including many of the array intrinsics (e.g. shift, transpose, spread, broadcast, reduction) and a set of data layout conversion routines (e.g. column-to-row, row-to-column, block-to-cyclic, and cyclic-to-block conversions).

4.2 Experimental Results

We evaluated the effectiveness of these optimizations using a set of linear algebra applications (including dense linear solvers using LU decomposition, tridiagonal systems solvers using Gaussian Elimination, Parallel Cyclic Reduction, etc.), PDE solvers using ADI method, and the familiar FFT.

We report our experimental results on the Connection Machines CM-5 located at AHPCRC of Minnesota University. The CM-5 has totally 896 processing nodes (PN), configured as various-sized partitions. Each processing node is a SPARC with four optional vector units that totally can deliver peak rate of 128Mflops [23].

To study the impact of the optimizations on modern machines, we have also hand-compiled some of the benchmark programs for the IBM SP-2 and a UltraSparc workstation cluster. We hand-generated message-passing C codes from the post-transformed SemRep intermediate representation. Experiments were conducted on a 8-node IBM SP-2 and a 4-node UltraSparc cluster located in Academia Sinica.

Each optimization is evaluated by comparing the execution time of the code compiled with the optimization against that of the code without the optimization. The execution time for perfect parallelism, which measures per-processor computation time only, is provided as a basis for comparison.

4.3 XFORM-1 Optimization

Two examples are used to evaluate the effectiveness of XFORM-1 optimization: a PDE solver using Alternating Direction Implicit method (ADI), and the familiar Fast Fourier Transform (using hybrid data layout and layout conversion). Both results show that optimized layout conversion by XFORM-1 transformation can reduce communication time significantly.

ADI. When ADI is applied on a 2D mesh, the solution is obtained iteratively by repeatedly sweeping along the x and y directions alternatively. A natural data distribution strategy for ADI is partitioning the arrays into columns of blocks in x-sweep, re-aligning the arrays by transposing them before performing y-sweep and transposing them back after finishing y-sweep. Without XFORM-1 optimization, CRAFT compiler generates general communication for the two array re-alignments. With XFORM-1 optimization, CRAFT compiler performs algebraic simplification, recognizes `TRANSPOSE` over one-dimensionally partitioned array as a communication idiom, and replaces the two re-alignment directives with calls to specialized fast communication routine associated with the idiom.

FFT. An ideal data layout for FFT is a hybrid (BLOCK,CYCLIC) distribution, as described in Section 3.4. In this benchmark, the FFT loop were split into two, with BLOCK distribution for the first and CYCLIC distribution for the second, given by DISTRIBUTE/REDISTRIBUTE directives. Without XFORM-1 optimization, CRAFT compiler generates general communication for data redistribution. With XFORM-1 optimization, calls to optimized BLOCK_TO_CYCLIC and CYCLIC_TO_BLOCK communication routines were generated for data redistribution.

Figure 11 shows performance of the two benchmarks on 64 processors. Sequential time on single processor is provided as a basis for evaluation. Computation time is almost identical for both optimized

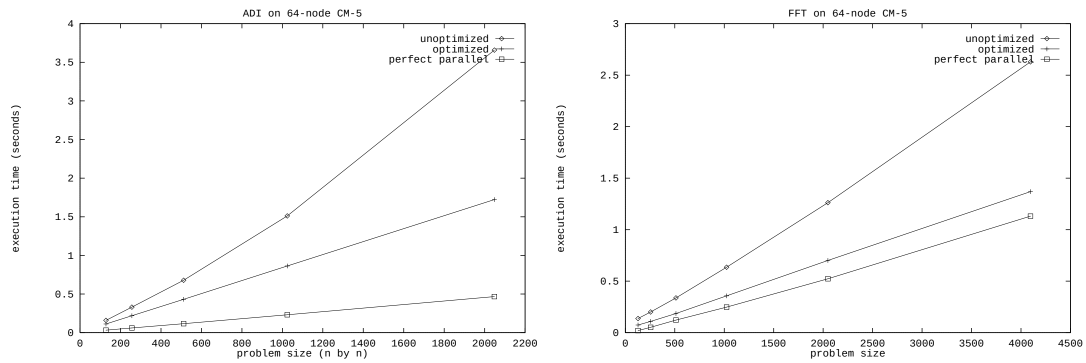


Figure 11: Execution time in seconds on 64 processors (ADI: 10 iterations, double precision floating points, 1d FFT: double precision complex, with block-cyclic hybrid data layout

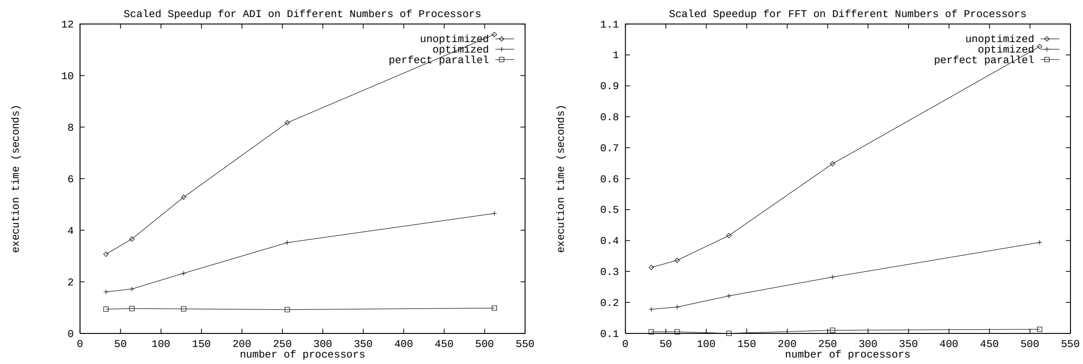


Figure 12: Execution time in seconds on different number of processors with fixed per-processor problem size (ADI: double precision, 128×128 per processor problem size, 1d FFT: double precision complex, $8k$ per processor problem size)

and unoptimized versions. With XFORM-1 optimization, the CRAFT compiler improves communication time of ADI by a factor of 1.6 to 2.5 and the total execution time of ADI by a factor of 1.4 to 2.1. Speedup factors increase with problem sizes. Possible reason is that on CM-5 a long message is sent in patches; larger problem size produces longer messages, and therefore heavier traffic in the network, and, as a result, higher speedup due to XFORM-1 communication optimization. With optimized block-cyclic and cyclic-block layout conversions, the CRAFT compiler reduces communication time of FFT by a factor of 1.4 to 3.5 and the total execution time of FFT by a factor of 1.3 to 1.9. Consistent with the results of ADI, speedup factors in FFT also increase with problem sizes.

We also study the scaled speedups by fixing per-processor problem size and changing number of processors. As shown in Figure 12, when per-processor size is fixed, speedup factors for ADI increase with number of processors (by a factor of 2.15 on 32 processors to a factor of 2.86 on 512 processors), because number of messages increases linearly with machine size; without XFORM-1 optimization, message contention becomes a more serious problem on larger machines. For larger machine sizes, speedup factors for FFT also increase with machine sizes.

4.4 XFORM-2 Optimizations

Processor-Memory Skewing

We use two examples to evaluate the effectiveness of processor-memory skewing(pipelining): a tridiagonal solver using Gaussian Elimination (MGE), and successive over relaxation (SOR). Both results show that processor-memory skewing improves code performance for large problem sizes.

We compare the pipelined version against the non-pipelined version. We also compare their performance with the CM-Fortran compiler (CM-Fortran compiler does not perform pipeline optimization) and the hand-coded CMSSL tridiagonal solver routine using pipelined Gaussian Elimination as the factoring strategy. The sequential execution time on single processor is provided as a basis for comparison. Since in SOR data dependence exists in both dimensions, no good vector codes are generated. We use scalar codes in our comparison.

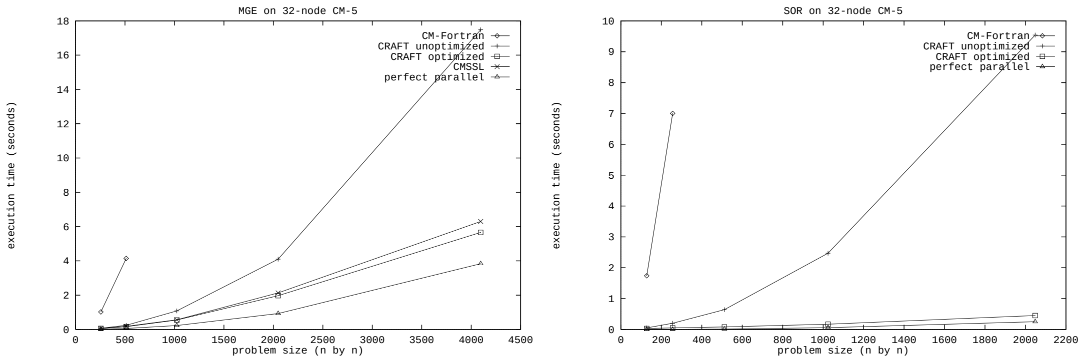


Figure 13: Execution time in seconds on 32-processor CM-5. MGE: double precision, 2D partition with processors configured as 4×8 array and data dependence exists at the first dimension, SOR: double precision, 1D partition

Figure 13 shows the execution time of the two benchmarks on 32-processor CM-5. Pipelining improves performance of MGE by a factor of 1.2 to 3.1 (CRAFT non-pipeline/CRAFT pipeline). With fixed number of processors, the speedup factors increase with problem sizes, but is bound by the number of processors at the first dimension (4, in this example). This is consistent with the estimated speedup factor for processor-memory skewing ($\frac{V*P}{V+P-1}$), where P is number of processors and V the size of local array. The results of SOR are quite consistent with those of the MGE. Speedup factors also increase with problem sizes. The performance of CRAFT optimized MGE is approaching that of the CMSSL hand-coded routine. CM-Fortran compiler does not perform optimization similar to processor-memory skewing. Its virtual processor (VP) compilation model causes excessive communication and data copy in its compiled code.

Figure 14 compares the execution time of the hand-compiled MGE programs on the IBM SP-2 and the UltraSparc workstation cluster. Both results show that *processor-memory skewing* is profitable.

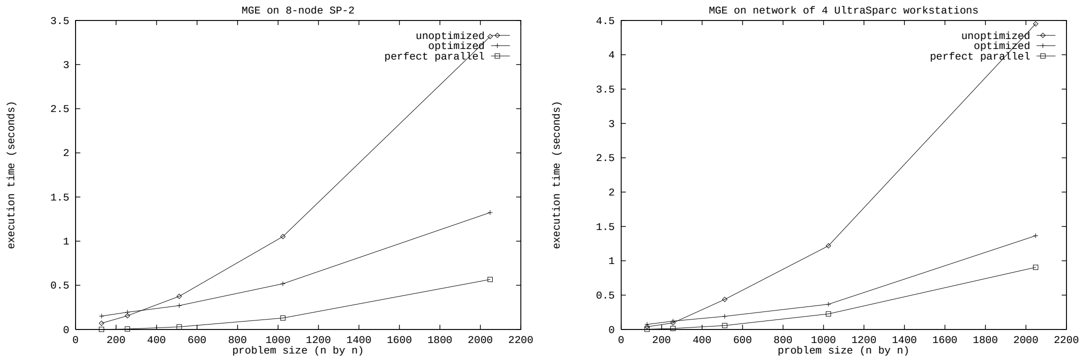


Figure 14: Execution time in seconds for MGE on IBM SP-2 and UltraSparc cluster

Block-Cyclic Permutation

Two divide-and-conquer codes are used to evaluate the effectiveness of block-cyclic permutation: FFT on one-dimensional array (bit reversal excluded) and Parallel Cyclic Reduction (PCR) along single dimension of two-dimensional arrays.

Without this XFORM-2 optimization, the code is compiled using block distribution. Therefore, communications are required for the butterfly iterations in FFT and the reduction iterations in PCR. With XFORM-2 block-cyclic-permutation transformation, communications within iterations are turned into local memory accesses, in the expense of extra data layout conversion between loop nests.

Figure 15 shows the execution time of the two benchmarks on 64-processor CM-5. With block-cyclic permutation, the CRAFT compiler improves code performance by a factor of 3.64 to 5.95 for FFT. When problem size increases, number of butterfly stages and per-processor problem size also increase, which results in longer communication time for the butterfly stages in the unoptimized version. In the optimized version, communication within butterfly stages are all turned into local memory accesses, in the expense of two data layout conversions. Communication time for the two layout conversions also increases with problem size. However, its impact is less significant than that of the butterfly communications. Consequently, total improvement in communication by this XFORM-2 transformation increases with problem sizes. For PCR, this optimization decreases program performance for smaller problem sizes. Possibly due to the fact that more data arrays are involved in layout conversion, which incurs large overhead that dominates program execution time. However, when problem size increases, this optimization also improves program performance by a factor of 1.35 to 1.45.

We also compare the CRAFT hand-compiled codes with the corresponding hand-coded CMSSL routines (FFT_detailed and CMSSL banded solver using parallel cyclic reduction as solution strategy). Both the CMSSL FFT routine and the CMSSL banded solver outperforms the CRAFT compiler. This implies that additional compiler optimization techniques are needed for this class of programs.

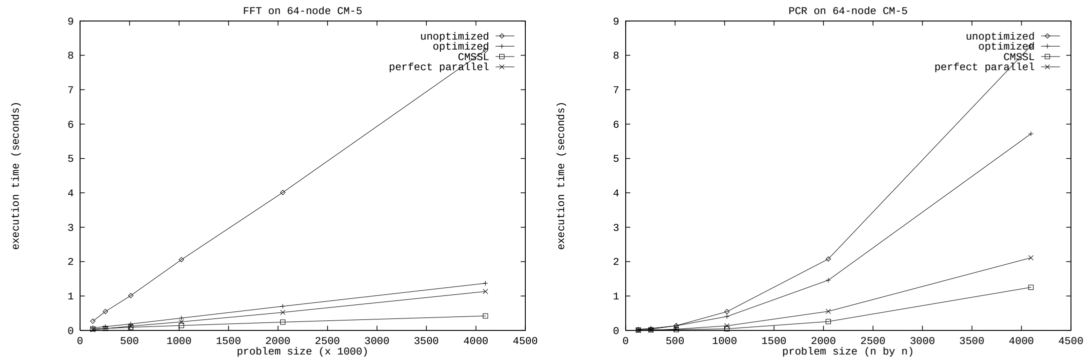


Figure 15: Execution time in seconds on 64-processor CM-5. FFT: double precision complex, with layout conversion. PCR: double precision floating-points, with layout conversion

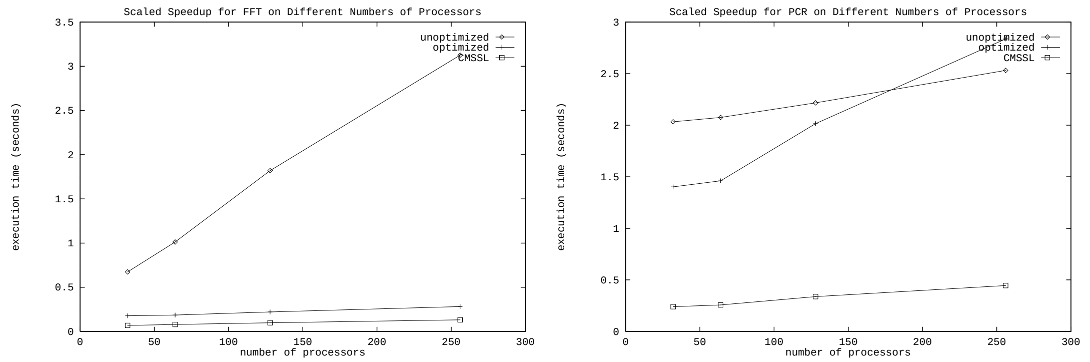


Figure 16: Execution time in seconds on different numbers of processors on CM-5, with fixed per-processor problem size. FFT: double precision complex, $8k$ per-processor, PCR: double precision floating-points, $64k$ per-processor

The scaled speedups for block-cyclic permutation are shown in Figure 16. The speedup factors (Unoptimized/Optimized) for FFT increase with number of processors, possibly because the number of butterfly stages also increase with number of processors, which results in more `CSHIFT` communication within stages. On the contrary, speedup factors for PCR degrade when number of processors increases. This negative effect becomes more significant for larger machine sizes, due to increased layout conversion overhead.

Figure 17 compares the execution time of the hand-compiled FFT programs on the IBM SP-2 and the UltraSparc workstation cluster. Both results show that *block_cyclic permutation* is profitable.

Interleaved Reduction

We use a LU solver to evaluate the effectiveness of interleaved reduction. We compare the performance of the triangular solvers with and without interleaved reduction. We also compare the LU solver using this optimization with CM-Fortran compiler's output code and the hand-coded CMSSL LU solver routine.

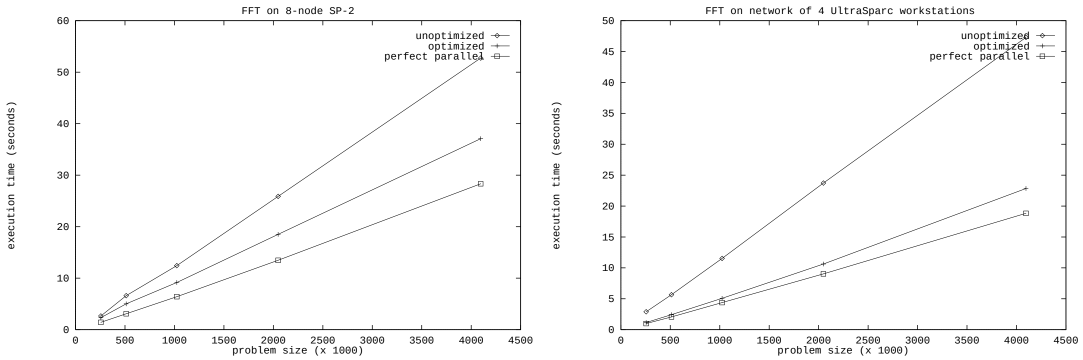


Figure 17: Execution time in seconds for FFT on IBM SP-2 and UltraSparc cluster, with layout conversion

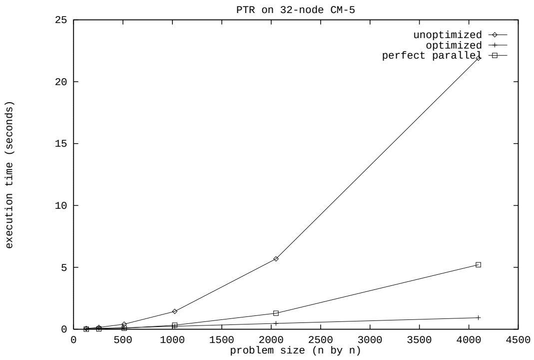


Figure 18: Execution time in seconds for a parallel triangular solver on 32 processors

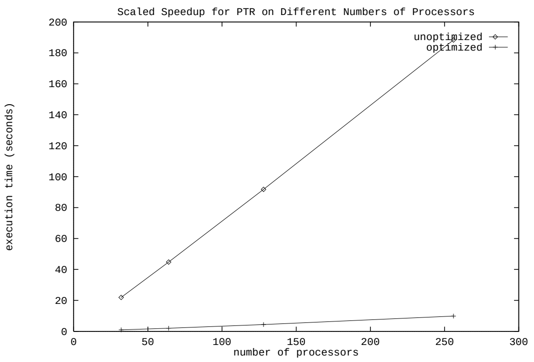


Figure 19: Execution time in seconds for a parallel triangular solver on different number of processors, with fixed per-processor problem size ($512k$)

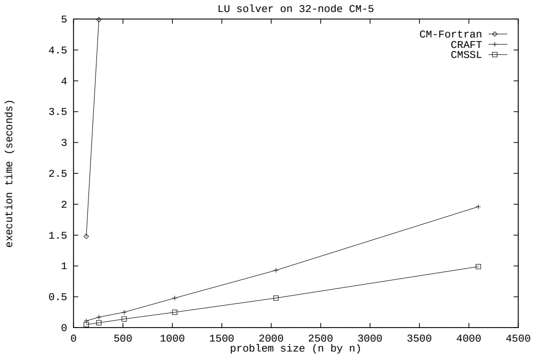


Figure 20: Execution time in seconds for LU solver on 32 processors

Figure 18 and Figure 19 show the effectiveness and scaled speedup of interleaved reduction optimization. When machine size is fixed (Figure 18), execution time of the unoptimized version (cyclic only, without interleaved reduction) varies approximately quadratically with problem size n (this is generally true for larger problem size; for smaller problem size, the execution time does not varies quadratically because the message start-up overhead varies linearly with problem size). With fixed machine size, execution time of the CRAFT optimized version (with interleaved reduction) varies linearly with problem size n until n reaches the threshold value (too large to show experimentally) and after that value the execution time varies quadratically with problem size. Therefore, the speedup of the CRAFT optimized version against the CRAFT unoptimized version increases as the problem size increases. When per-processor problem size is fixed, the speedup factor decreases as number of processors increases. This is because the extra overhead in global reduction operation increases with the number of processors.

Figure 20 shows the performance comparison against CM-Fortran compiler, which does not perform optimization similar to interleaved reduction, and the CMSSL routine, which hard-wire some similar optimization. The CRAFT optimized version runs faster than the CM-Fortran version by two orders of magnitude, and is slower than the CMSSL version within a factor of 2.

4.5 Summary

The results from the two benchmark codes ADI and FFT show that XFORM-1 optimization (algebraic simplification plus idiom matching) can reduce communication time significantly and should be performed. Research has shown that implementing fine-tuned communication routines is crucial to achieving high performance on many distributed-memory parallel machines [9, 50, 51, 55]. We believe that these machines will also profit from XFORM-1 optimization (perhaps with different implementation of the communication idioms).

The results from MGE and SOR demonstrate that processor-memory skewing (with appropriate pipeline granularity) improves code performance independent of problem size and machine size. The results from FFT and PCR show that the tradeoffs of block-cyclic permutation greatly depends on the number of arrays involved in block-cyclic conversion and the number of shift operations in the loop body. In general, the fewer the arrays in layout conversion and the more the shift operations in the loop body, the higher is the effectiveness of this optimization. The results of the lower triangular solver and the LU solver demonstrate that interleaved reduction, although very specific, can improve program performance dramatically, especially for large problem sizes. However, this optimization may not scale up perfectly as number of processors increase, due to increased global reduction overhead on larger machine. In short, XFORM-2 transformations are more machine dependent. Whether they should be performed depends on the loop structure, problem size, machine size and other machine parameters.

5 Related Work

A number of prototype compilers for Fortran90/HPF have been developed in the past few years. A summary of HPF compilers can be found in [1]. We briefly review some of the optimization techniques used in these compilers. The Fortran D compiler [36, 60] performs various optimizations (message vectorization, message pipelining) to reduce communication overhead. It also attempts to increase processor parallelism via pipelining and parallelizing reductions. XFORM-1 and XFORM-2 differ from the Fortran D system in two major aspects. First, the Fortran D compiler only handles a small subset of HPF's data layouts: canonical alignment and one-dimensional data partitioning, while the CRAFT compiler's XFORM-1 transformations are applicable to more general cases. Secondly, the CRAFT compiler's XFORM-2 *interleaved reduction* optimization is applicable to more general reduction loop nests. The Fortran 90D compiler optimizes data movement for subscripted array references in parallel loops using linear index-function transformation and pattern matching for collective communication [11]. By formulating data movement using linear transformations, optimization for non-linear alignments, such as `CSHIFT` and replication, and data redistribution are not possible. Roth [53] designed a set of optimizations for array data movement associated with HPF shift operations. These optimizations do not take the effect of user-provided alignment directives into consideration as we do.

Vienna Fortran and Vienna Fortran-90, based upon the parallelizing system SUPERB, extends Fortran and Fortran-90 by providing alignment and distribution specifications. The Vienna Fortran compiler [13, 67] focus on supporting arbitrary rectilinear block distributions and irregular data distribution through indirect arrays. The SHPF compilation system [48] supports full set of HPF data distribution directives. Similar optimizations are performed to eliminate redundant data movement and data copy. No loop-level optimizations such as XFORM-2 transformations are supported though.

On the part of industry, the CM Fortran compiler uses simple but naive copy-in, copy-out strategy for inter-procedural data movement, and copying via canonical temporary for intra-procedural data movement. The CM Fortran compiler does not perform XFORM-2 type of optimizations. Instead, similar optimizations are hard-wired in the hand-coded CMSSL scientific library routines. DIGITAL High Performance Fortran90 supports full HPF standard alignment and distribution directives. The DIGITAL HPF90 compiler [35] optimizes communication by message vectorization and optimized runtime communication routines. The compiler also performs a set of loop-level optimizations, such as strip minig, loop interchange, and loop fusion. The PGHPF compiler [3] developed by the Portland Group Inc. also supports full HPF. A set of communication optimizations such as overlap shift operations, collective communications, and elimination of redundant data copy, and a set of simple loop transformations such as loop fusion and loop interchange are included in the PGHPF compiler.

Forge90, xHPF, and spf [28] is an interactive parallelization system for MIMD shared and distributed-memory machines developed by Applied Parallel Research. The main focus of Forge90 is on parallelizing sequential Fortran programs and allowing users to fine-tune performance via interactive data distribution specifications and loop transformation invokations. xlHPF [2], developed for IBM, supports a subset of HPF standards. To our knowledge, the xlHPF compiler has not performed advanced compiler optimizations as we do.

The CRAFT compilation framework also relates to other more specific research effort. The technique for generating collective communication, pioneered by the Crystal compiler [46, 47, 44], has great influence on our XFORM-1 optimization. The major differences are: (1) The Crystal compiler finds optimal (or near optimal) data alignment automatically, while the CRAFT compiler's goal is optimizing data movement in the presence of user-provided data layout specifications. (2) The Crystal compiler does not optimize inter-procedural data movement as the CRAFT compiler does.

Array section references also frequently occur in HPF programs. Several approaches have addressed the efficient execution of array statements involving block-cyclically distributed array sections, e.g. Gupta, Kaushik, Huang, and Sadayappan's virtual processor approach [54, 41], Chatterjee et al's finite-state machine approach [15], Stichnoth's [56, 57] array slice analysis, Kennedy et al's [42] and Thirumalai and Ramanujam's [58] integer lattice approach. XFORM-1 transformation does not compete with these work. The XFORM-1 framework adopts similar techniques to implement the communication idioms for array section movement.

There are also work on global optimization for data movement. Gilbert and Schreiber [32, 16] designed a dynamic programming algorithm for optimizing temporary storage use for Fortran 90 array statements. Chatterjee, et al. [17, 18] extended that work to allow loop nests. Ju, Wu, and Carini [40] (and later Hwang, Lee, and Ju [39]) proposed a synthesis scheme for combining consecutive data reference patterns to reduce communication. Another line of work optimize communication using data-flow analysis, e.g. Amarasinghe and Lam's [6], Gong, Gupta and Melhem's [33], and Gupta, Schonberg, and Srinivasan's [34]. The current implementation of CRAFT compiler assumes owner-compute rule for compilation of data movement. In the near future, we will extend our XFORM-1 framework to incorporate global optimization techniques.

Current work in XFORM-2 transformations focused on the application of individual transformations: when it is legal to apply a transformation, and if the transformation directly contributes to certain goal. The heuristic used in combining these optimizations, however, are ad hoc. Two commonly used strategies are: to decide in advance the order in which these transformations should be applied, or to apply all different possible combinations of transformations. Both have been proven to be either inadequate or too expensive. Some work has been done in devising efficient algorithms for combining a set of loop transformations, e.g. Wolf and Lam's [61, 62], and Appelbe and Smith's [7].

6 Conclusions

In this paper, we have described the theoretical and experimental results of the CRAFT compiler project. We studied optimization issues in compiling High Performance Fortran for distributed-memory parallel machines. We presented a two-phase transformative framework: an abstract, algebraic transformation and runtime technique (XFORM-1) for reducing communication overhead, and a set of more machine-dependent transformations (XFORM-2) for increasing parallelism and reducing communications in data-parallel loops.

The advantages we see of the two-phase framework are conceptual cleaness and portability due to (1) the ability to detect and optimize alignment traffic and data layout conversion at the abstract level and leave machine-dependent communication details to the runtime system, and (2) the reduction of complexity in XFORM-2 transformations, because the data alignment problem has been resolved in XFORM-1 phase.

The new XFORM-1 algebraic transformation framework (including algebraic representation of data movement, alignment and distribution, a communication algebra, and runtime layout conversion and communication services) allows a compiler to reduce data movement at the abstract level and leave machine-dependent details to the runtime system. By modeling different stages of data mapping (alignment, distribution, physical mapping) and data movement using *communication expressions* and providing algebraic rules to simplify each stage of data movement, the algebraic framework is conceptually clean and portable to different target architectures.

XFORM-2 transformations consist of a set of new optimizations (e.g. *block-cyclic permutation* and *interleaved reduction*) and a set of refined techniques that are adopted from existing work (e.g. *processor-memory skewing*, *increasing granularity*). We have shown that *processor-memory skewing* and *increasing granularity*) are general-purpose optimizations that are applicable to a broad class of loop nests, while *block-cyclic permutation* and *interleaved reduction* are applicable to programs that involve divide-and-conquer algorithms and reduction opertions, respectively. We have also shown that many of these optimizations are also applicable to modern parallel machines. Finally, we hope that our collection of XFORM-2 transformations may serve as a basis for automated optimizations, which, in commercial systems, have too often been ignored or hard-wired in hand-coded library routines.

Both XFORM-1 and XFORM-2 transformations can be implemented efficiently in an optimizing compiler. They have been implemented and integrated into the CRAFT compiler. The complexity of the algebraic transforms is linear in the number of array references in the program, under HPF's "owner-compute" model. Complexity of most post-partition optimization primitives is polynomial in the number of array references in each loop nest. Since we use a linear ordering in applying these optimizations, overall complexity of XFORM-2 transformations remains polynomial.

Recently, there has been some progress on the part of industry toward applying some simple kinds of layout optimizations in HPF compilers. For example, TMC's CM-Fortran compiler version 2.1 has included some similar optimization techniques for a subset of layout directives (shift/shift combination). We hope that eventually most commercial compiler groups will adopt our two-phase transformative strategy, or something very similar to it, to make sure that users can write HPF code without worrying about compiler blind spots (like "copy in – copy out" calling sequences, or redundant sequences of copies through heap temporaries whenever non-trivial alignments are in force).

References

[1] A survey of hpf compilers and tools. available at *http://www.ac.upc.es/HPFSurvey/indexlist1.html.*

[2] Xl high performance fortran for aix. IBM Inc., 1996.

[3] Pghpf workstation 2.2 release note. the Portland Group Inc., 1997.

[4] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook.* McGraw Hill, 1992.

[5] J. R. Allen. *Dependence Analysis for Subscript Variables and Its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[6] A.P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of ACM SIGPLAN'93 Programming Language Design and Implementation*, Albuquergue, New Mexico, June 1993.

[7] W. Appelbe and K. Smith. Determining transformation sequences for loop parallelization. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, Connecticut, August 1992.

[8] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[9] Shahid H. Bokhari. Multiphase Complete Exchange on A Circuit Switched Hypercube. Technical report, ICASE, NASA Langley Research Center, 1991.

[10] P. Bose. Interactive program improvement via eave. In *Proceedings of the International Conference on Supercomputing*, 1988.

[11] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. Compiling fortran 90/hpf for distributed memory mimd computers. *Journal of Parallel and Distributed Computing*, 1994.

[12] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 145–156, June 1991.

[13] B. Chapman, H. Herbeck, and H.P. Zima. Automatic Support for Data Distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, April 1991.

[14] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Vienna FORTRAN – A Fortran Language Extension for Distributed Memory Multiprocessors. In *High Performance FORTRAN Forum*, Houston, Texas, January 1992.

[15] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of Principles and Practice of Parallel Programming*, pages 149–158, San Diego, CA, May 1993.

[16] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report CSL-92-11, XEROX Palo Alto Research Center, December 1992.

[17] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, 1993.

[18] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal Evaluation of Array Expressions on Massively Parallel Machines'. Technical report, Xerox Corporation, Palo Alto Research Center, December 92.

[19] Marina Chen. A Parallel Language and its Compilation to Multiprocessor Machines. In *19th Annual Symposium on Principles of Programming Languages*, January 1986.

[20] Marina Chen, Young-il Choo, and Jingke Li. Theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers*, chapter 7. ACM Press and Addison-Wesley, 1991.

[21] Marina Chen and James Cowie. prototyping Fortran-90 Compilers for Massively Parallel Machines. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, June 1992.

[22] Marina Chen and Yu Hu. Optimizations for Compiling Iterative Spatial Loops to Massively Parallel Machines. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, 1992.

[23] The Connection Machine CM-5 Technical Summary. Technical report, Thinking Machines Corporation, 1991.

[24] CM Fortran Reference Manual. Thinking Machines Corporation, Cambridge, Massachusetts, July 1991.

[25] C* User's Guide and Programming Guide. Thinking Machines Corporation, Cambridge, Massachusetts, November 1990.

[26] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, and E. Santos. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[27] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.

[28] Forge 90 Distributed Memory Parallelizer: User's Guide. Technical report, Applied Parallel Research, 1992.

[29] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. In *High Performance FORTRAN Forum*, Houston, Texas, January 1992.

[30] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lysenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[31] Anwar M. Ghuloum. *Compiling Recurrent and Irregular Serial Code for High Performance Computer.* PhD thesis, Carnegie-Mellon University, 1996.

[32] J. Gilbert and R. Schreiber. Optimal Expression Evaluation for Data Parallel Architectures. *Journal of Parallel and Distributed Computing*, 13(1), September 1991.

[33] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication in distributed memory systems. In *Proceedings of International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[34] M. Gupta, E. Schonberg, and H. Srinivasan. An unified data-flow framework for optimizing communication. In *Proceedings of 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[35] J. Harris. Compiling High Performance Fortran for Distributed-Memory Systems. *Digital Technical Journal*, 7(3), 1995.

[36] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling fortran d for mimd distributed-memory machines. *Communications of ACM*, 35(8):66–80, August 1992.

[37] Ching-Tien Ho. *Optimal Communication Primitives and Graph Embeddings on Hypercubes.* PhD thesis, Department of Computer Science, Yale University, 1990.

[38] High Performance Fortran Language Specification. Technical report, Rice University, Houston Texas, May 1993.

[39] G.H. Hwang, J.K. Lee, and D.C. Ju. An array synthesis scheme to optimize fortran 90 programs. In *Proceedings of Principles and Practice of Parallel Programming*, April 1995.

[40] D.C. Ju, C.L. Wu, and P. Carin. The synthesis of array functions and its use in parallel computation. In *Proceedings of International Conference on Parallel Processing*, 1992.

[41] S.D. Kaushik, C.H. Huang, and P. Sadayappan. Compiling array statements for efficient execution on distributed memory machines: two-level mappings. In *Proceedings of 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[42] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[43] B. Leasure. The parafrase project's fortran analyzer. Technical Report 85-504, Dept. Computer Science, University of Illinois at Urbana Champaign, 1985.

[44] Jingke Li. *Compiling Crystal for Distributed Memory Machines.* PhD thesis, Dept. of Computer Science, Yale University, 1991.

[45] Jingke Li and Marina Chen. Generating Explicit Communication from Shared-Memory Program References. In *Supercomputing*, pages 865–876, 1990.

[46] Jingke Li and Marina Chen. *Proceedings of the Workshop on Programming Languages and Compilers for Parallel Computing*, chapter Automating the Coordination of Interprocessor Communication. MIT Press, 1990.

[47] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 1991.

[48] John Merlin, Bryan Carpenter, and Tony Hey. shpf: A subset high performance fortran compilation system. Dept. Electronics and Computer Science, University of Southampton, 1996.

[49] S. S. Pinter and R. Y. Pinter. Program optimization and parallelization using idioms. In *Proceedings of Principles of Programming Languages*, 1990.

[50] R. Ponnusamy, R. Thakur, A. Choudhary, and G. Fox. Scheduling Regular and Irregular Communication Patterns on the CM-5. In *Proceedings of Supercomputing '92*, 1992.

[51] T. Ponnusamy, A. Choudhary, and G. Fox. Communication Overhead on CM5: An Experimental Performance Evaluation. In *Proceedings of Frontiers '92*, 1992.

[52] X. Roden and P. Feautrier. Detection of recurrences in sequential programs with loops. In *Lecture Notes in Computer Science, vol. 694*, 1993.

[53] Gerald H. Roth. *Optimizing Fortran90D/HPF for Distributed-Memory Computers.* PhD thesis, Rice University, 1997.

[54] K.S. Gupta S, S.D. Kaushik, C.H. Huang, and P. Sadayappan. Compiling array statements for efficient execution on distributed memory machines: two-level mappings. *Journal of Parallel and Distributed Computing*, (32):155–172, 1996.

[55] T. Schmiermund and S. R. Seidel. A Communication Model for the Intel iPSC/2. Technical report, Michigan Technological University, 1990.

[56] James M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[57] J.M. Stichnoth, D. O'Hallaron, and T.R. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–158, April 1994.

[58] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. In *Proceedings of the International Parallel Processing Symposium*, 1995.

[59] Cmssl for cm fortran (cm-5 edition), version 3.1. Technical report, Thinking Machines Corporation, June 1993.

[60] Chau-Wen Tseng. *An Optimizing Fortran D Compilers for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, 1993.

[61] M. Wolf and M. Lam. Maximizing parallelism via loop transformations. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* UC. Irvine, 1990.

[62] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

[63] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

[64] Jan-Jan Wu. An interleaving transformation for parallelizing reductions for distributed-memory multiprocessors. *The Journal of Supercomputing*, to appear, 1999.

[65] Jan-Jan Wu and Marina Chen. An algebraic machinery for optimizing data motion for hpf. *to appear in Scientific Programming, Vol. 6, No. 4*, 1997.

[66] J. Allan Yang and Young-il Choo. Parallel-program transformation using a metalanguage. In *Proceedings of The Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, Virginia*, pages 11–20, April 1991.

[67] H. Zima and B. Chapman. Compiling for distributed memory systems. In *Proceedings of the IEEE Special Section on Languages and Compilers for Parallel Machines*, pages 264–287, February 1993.