A General Stencil Compilation Strategy for Distributed-Memory Machines

Gerald Roth
Steve Carr
John Mellor-Crummey
Ken Kennedy

CRPC-TR96652-S
June 1996

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

A General Stencil Compilation Strategy for Distributed-Memory Machines

Gerald Roth — Dept of Comp Sci, Rice University, Houston, TX 77005

Steve Carr — Dept of Comp Sci, Michigan Tech, Houghton, MI 49931

John Mellor-Crummey — Dept of Comp Sci, Rice University, Houston, TX 77005

Ken Kennedy — Dept of Comp Sci, Rice University, Houston, TX 77005

Abstract

For many Fortran 90 programs performing dense matrix computations, the main computational portion of the program belongs to a class of kernels known as stencils. This paper describes a strategy for optimizing such stencil computations for execution on distributed-memory multiprocessors. The optimizations presented target the overhead of data movement that occurs between processors, within the local memory of the processors, and between the memory and registers of the processors. We focus on the application of this strategy on distributed-memory architectures, although it is more broadly applicable.

1 Introduction

High-Performance Fortran (HPF)[17], an extension of Fortran 90, has attracted considerable attention as a promising language for writing portable parallel programs. HPF offers a simple programming model shielding programmers from the intricacies of concurrent programming and managing distributed data. Programmers express data parallelism using Fortran 90 array operations and use data layout directives to direct partitioning of the data and computation among the processors of a parallel machine.

For HPF to gain acceptance as a vehicle for parallel scientific programming, it must achieve high performance on problems for which it is well suited. To achieve high performance on a distributed-memory parallel machine, an HPF compiler must do a superb job of translating Fortran 90 data-parallel stencil computations which are common to many dense matrix algorithms. In this paper, we focus on the problem of optimizing stencil computations, no matter how they are instantiated by the programmer, for execution on distributed-memory architectures.

In the next section we briefly discuss stencil computations and their execution cost on distributed-memory machines. In Section 3 we give an overview of our compilation strategy, and then address the individual optimizations in detail in Sections 4, 5, and 6. In Section 7 we compare this strategy with other known efforts.

2 Stencil Computations

In this section we introduce stencil computations and discuss their execution cost on distributed-memory machines.

2.1 Stencils

A *stencil* is a stylized matrix computation in which a group of neighboring data elements are combined to calculate a new value. They are typically combined in the form of a sum of products. Consider the following the Fortran 90 array assignment statement that is commonly referred to as a 5-point stencil:

```
 DST(2:N-1,2:N-1) = C1 * SRC(1:N-2,2:N-1)
&                 + C2 * SRC(2:N-1,1:N-2)
&                 + C3 * SRC(2:N-1,2:N-1)
&                 + C4 * SRC(3:N ,2:N-1)
&                 + C5 * SRC(2:N-1,3:N )
```

in which SRC and DST are arrays, and C1–C5 are either scalars or arrays. Each interior element of the result array DST is computed from the corresponding element of the source array SRC and the neighboring elements of SRC on the North, West, South, and East.

A 9-point stencil that computes all grid elements by exploiting the CSHIFT [1] intrinsic might be specified like this:

[1] The CSHIFT intrinsic takes three arguments: the array to be shifted, the amount of the shift, and the dimension to shifted.

```
 DST = C1 * CSHIFT(CSHIFT(SRC,-1,1),-1,2)
&    + C2 * CSHIFT(SRC,-1,1)
&    + C3 * CSHIFT(CSHIFT(SRC,-1,1),+1,2)
&    + C4 * CSHIFT(SRC,-1,2)
&    + C5 * SRC
&    + C6 * CSHIFT(SRC,+1,2)
&    + C7 * CSHIFT(CSHIFT(SRC,+1,1),-1,2)
&    + C8 * CSHIFT(SRC,+1,1)
&    + C9 * CSHIFT(CSHIFT(SRC,+1,1),+1,2)
```

In the two previous examples the stencil was specified as a single array assignment statement but this need not always be the case. Let's consider again the 9-point stencil above. If the programmer attempted to optimize the program by hand, or if the stencil was preprocessed by an optimization phase performing common subexpression elimination, we might be presented with the following (taken from Problem 9 of the Purdue Set [23]):

```
 TMP1 = CSHIFT(SRC,-1,1)
 TMP2 = CSHIFT(SRC,+1,1)
 DST = C1 * CSHIFT(TMP1,-1,2)
&    + C2 * TMP1
&    + C3 * CSHIFT(TMP1,+1,2)
&    + C4 * CSHIFT(SRC,-1,2)
&    + C5 * SRC
&    + C6 * CSHIFT(SRC,+1,2)
&    + C7 * CSHIFT(TMP2,-1,2)
&    + C8 * TMP2
&    + C9 * CSHIFT(TMP2,+1,2)
```

When encountering a set of statements such as these, we would like to be able to produce code equivalent to that produced for the single-statement stencil. Thus we have designed our optimizer to handle the most general form which has several distinguishing characteristics:

- CSHIFT intrinsics and temporary arrays have been inserted to perform data movement needed for operations on array sections that have different processor mappings.
- Each CSHIFT intrinsic occurs as a singleton operation on the right-hand side of an array assignment statement and is only applied to whole arrays.
- The expression that actually computes the stencil operates on operands that are perfectly aligned, and thus no communication operations are required.

All stencil and stencil-like computations can be translated into this general form by factoring expressions and introducing temporary arrays. In fact, this is the intermediate form used by several distributed-memory compilers [21, 24]. For example, given the 5-point stencil computation presented above, the CM Fortran compiler would translate it into the following statement sequence[2]:

```
 ALLOCATE TMP1, TMP2, TMP3, TMP4
 TMP1 = CSHIFT(SRC,SHIFT=-1,DIM=1)
 TMP2 = CSHIFT(SRC,SHIFT=-1,DIM=2)
 TMP3 = CSHIFT(SRC,SHIFT=+1,DIM=1)
 TMP4 = CSHIFT(SRC,SHIFT=+1,DIM=2)
 DST(2:N-1,2:N-1) = C1 * TMP1(2:N-1,2:N-1)
&                 + C2 * TMP2(2:N-1,2:N-1)
&                 + C3 * SRC (2:N-1,2:N-1)
&                 + C4 * TMP3(2:N-1,2:N-1)
&                 + C5 * TMP4(2:N-1,2:N-1)
 DEALLOCATE TMP1, TMP2, TMP3, TMP4
```

For the rest of this paper we assume that all stencil computations have been put into this form, and that all arrays are distributed in a BLOCK fashion. And although we concentrate on stencils expressed using the CSHIFT intrinsic, our techniques can be generalized to handle the EOSHIFT intrinsic as well.

2.2 Stencil Execution Costs

The cost of a stencil computation on a distributed-memory machine can be analyzed by breaking it down into its two major components: the set of CSHIFT operations and the calculation of the sum of products.

When a CSHIFT operation is performed on a distributed array two major actions take place:

1. Data elements that must be shifted across processing element (*PE*) boundaries are sent to the neighboring PE. This is the interprocessor component of the shift. The dashed lines in Figure 1 represent this data movement for arrays distributed in a BLOCK fashion.
2. Data elements that stay within the memory of the PE must be copied to the appropriate locations in the destination array. This is the intraprocessor component of the shift. The solid lines in Figure 1 represent this data movement.

Assuming a BLOCK distribution and that each PE contains a 2D subgrid of size $(g \times g)$, a shift amount of d, $d < g$, consists of an interprocessor move of d columns (of size g), and an intraprocessor move of

[2]For this reason most CM Fortran programmers use CSHIFTs explicitly in their stencil computations since array-syntax stencils produced the same CSHIFT intrinsic calls but then had the additional overhead of the vector masking operations required for handling the array subsections [25].

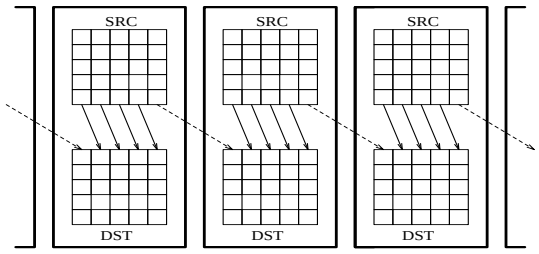

Figure 1: `DST = CSHIFT(SRC,SHIFT=-1,DIM=2)`

$g - d$ columns. The cost of such a shift operation is described by the following model [14]:

$$T_{\text{shift}} = g\,(g-d)\,t_{\text{onpe}} + C_{\text{onpe}} + g\;d\,t_{\text{offpe}} + C_{\text{offpe}} \quad (1)$$

where t_{onpe} and t_{offpe} represent the time to perform an intraprocessor and interprocessor copy respectively, and C_{onpe} and C_{offpe} represent the startup time (or latency) for each type of copy. Different models are required for the cases $d = g$ and $d > g$. CYCLIC(K) distributions also require a different model which include a parameterization for the blocking factor.

For most common stencil computations, the shift amount d is small compared to g. For such cases Equation (1) is $O(g^2\ t_{\text{onpe}})$ and the execution time T_{shift} is dominated by the cost of the intraprocessor copies, even when $t_{\text{onpe}} \ll t_{\text{offpe}}$.

The sum of products is calculated within a loop nest. The loop nest is a result of scalarization [28], where the array constructs are replaced by serial code, and the generation of SPMD code [11], where the computation is partitioned and loop bounds are reduced. This loop nest is the *subgrid loop nest* which will be executed on each PE of the parallel machine such that the PE only computes the data which it owns. At this point specific array elements are referenced rather than full arrays or array sections. If the stencil is not computed over the entire matrix, or if the subgid size is not known at compile time, each PE must compute its subgrid loop bounds. The subgrid loop for the 5-point stencil presented at the end of the previous subsection would look like this (where $\$MYPID$ returns the processor id number (zero based) for the given dimension, G is the extent of the local subgrid, and N is the extent of the original array):

```
LB1=MAX((MYPID(1)*G)+1,2)-(MYPID(1)*G)
UB1=MIN((MYPID(1)+1)*G,N-1)-(MYPID(1)*G)
LB2=MAX((MYPID(2)*G)+1,2)-(MYPID(2)*G)
UB2=MIN((MYPID(2)+1)*G,N-1)-(MYPID(2)*G)
DO J = LB2, UB2
  DO I = LB1, UB1
      DST(I,J) = C1 * TMP1(I,J)
&              + C2 * TMP2(I,J)
&              + C3 * SRC (I,J)
&              + C4 * TMP3(I,J)
&              + C5 * TMP4(I,J)
   ENDDO
 ENDDO
```

Calculating the execution cost of such a loop nest is usually accomplished by totalling the number of floating point operations in the loop, dividing that number by the rate the target machine can execute those flops, and then multiplying by the total number of iterations. Unfortunately, due to the large number of array references found in such a loop, this metric is insufficient. To better measure the performance of subgrid loops in relation to their memory accesses we will use the notion of balance as defined by Callahan, *et al.* [6].

The *machine balance* (β_M) for a particular machine is defined to be the relationship between the rate at which memory can be accessed compared to the rate that floating-point operations can be executed:

$$\beta_M = \frac{max\ words/cycle}{max\ flops/cycle}$$

The *loop balance* (β_L) for a given loop is defined as:

$$\beta_L = \frac{number\ of\ memory\ references}{number\ of\ flops}$$

If $\beta_L = \beta_M$ then the loop is balanced for the target machine and will run well. If $\beta_L < \beta_M$ then data can be supplied faster than it can be processed, and the loop is said to be *compute bound*. In this case the machine will be running at its peak computational rate. If $\beta_L > \beta_M$ then data can be processed faster then it can be supplied, and there will exist idle computational cycles. Such a loop is *memory bound*. For many advanced architectures which offer efficient multiply-add operations, the value of β_L for loops generated from stencils will often be larger than β_M, resulting in memory bound loops. The value of β_L will also be significantly increased if the stencil is specified with array-valued coefficients rather than with scalar values, thus exacerbating the problem.

3 Compilation Strategy

In this section we give an overview of our compilation strategy. We then present the details of this strategy in the subsequent sections.

Assuming that the program containing the stencil has been put into the form presented at the end of

Section 2.1, we begin by optimizing the CSHIFT operations. We apply two separate optimization phases: one addressing the intraprocessor data movement and the other handling the interprocessor data movement.

Intraprocessor data movement is optimized by completely eliminating it when possible. This can safely be done whenever we can determine that the source array (SRC) and the destination array (DST) of the CSHIFT can share the same memory locations. When this determination has been made, we can transform the program to perform only the interprocessor data movement and rewrite references to DST to refer to SRC with indexing adjusted by the shift amount. We call such a destination array an *offset array*.

Once the intraprocessor data movement has been eliminated, we analyze the resulting interprocessor data movement to eliminate redundant and partially redundant movement. The resulting program will require only a single communication operation across each edge [3] of the stencil. This optimization will produce only four communication operations for the 9-point stencil example presented earlier, even though its original specification required twelve CSHIFT intrinsics.

Finally, after scalarization has produced a subgrid loop nest, we optimize it by applying a set of loop transformations designed to improve the performance of memory-bound programs. These transformations include unroll-and-jam, which addresses memory references, and loop permutation, which addresses cache references. Each of these optimize the program by exploiting reuse of data values.

4 Eliminating Intraprocessor Copying

As mentioned in the previous section, the intraprocessor copying of data can be eliminated if we can determine that the source array and the destination array of the CSHIFT operation can share the same memory locations. If this is the case only the interprocessor data movement needs to occur. We exploit *overlap areas* [16] to receive the data that is copied between processors. After this has been accomplished, appropriate references to the destination array can be rewritten to refer to the source array with indexing offset by the shift amount. We call such a destination array an *offset array* [19].

The principal challenge then is to determine when the source and destination arrays can share storage. We have established a set of criteria to determine when it is safe and profitable to create an offset array. Given an assignment statement `DST = CSHIFT(SRC,SHIFT,DIM)` within our intermediate representation, the array DST may be treated as an offset array if the following criteria can be verified for this statement at compile time:

(1) The source array SRC is not modified while this definition of DST is *live*. **(2)** The destination array DST is not partially modified[4] while SRC is *live*. **(3)** The SRC array and the DST array are distributed in the same BLOCK (or CYCLIC(K)) fashion and are aligned with one another. **(4)** The values SHIFT and DIM are compile-time scalar constants. **(5)** The amount of interprocessor data must fit within the bounds placed on the size of the overlap areas. **(6)** For each use of DST that is reached by the given definition, all the definitions of DST that reach that use are identical offset arrays of the same source array SRC.

From the work on copy elimination in functional and higher-order programming languages [26], we know that the first two criteria are necessary and sufficient conditions for when the two objects can share the same storage. However, the sharing of storage may not always be profitable. To insure profitability, we added the remaining criteria. These efficiency criteria may be relaxed if we are willing to generate multiple versions of code for statements that use the array DST, and then select the appropriate version depending upon run-time conditions. However, due to the drawbacks of multiple versions of code, in particular code growth, we consider these additional criteria as important.

Once we have determined that the destination array of the assignment statement `DST = CSHIFT(SRC,SHIFT,DIM)` may be an offset array, we perform the following transformations on the code. These transformations take advantage of the data that may be shared between the source array SRC and destination array DST and move only the required data between the PEs.

First we replace the shift operation with a call to a routine that moves the interprocessor data into the appropriate overlap area: `CALL OFFSET_SHIFT(SRC,SHIFT,DIM)`. We then replace all uses of the array DST, that are reached from this definition, with a use of the array SRC. The newly created

[3] An edge is determined by the dimension and direction specified in a shift operation.

[4] Any partial modification will require a copy of the shifted array SRC and so we simply go ahead and make the copy at the point of the shift. Any full modification of DST which *kills* the whole array does not require the copy of SRC and thus DST may still be treated as an offset array up to the point of the killing definition.

references to SRC carry along special annotations representing the values of SHIFT and DIM. Finally, when creating subgrid loops during the scalarization phase, we alter the subscript indices used for the offset arrays. The array subscript used for the offset reference to SRC is identical to the subscript that would have been generated for DST with the exception that the DIM-th dimension has been incremented by the SHIFT amount.

It is possible that offset arrays are themselves used in other shift operations. If these shift operations also meet all of the criteria to be an offset array then the above transformations can again be applied. We call such arrays *multiple-offset arrays*. If one dimension is shifted multiple times the annotations for the SHIFT amounts are simply added together.

The algorithm that we have devised for verifying the stated criteria and for performing the above transformations is based upon the *static single assignment* (SSA) intermediate representation [13]. The algorithm, after validating the use of an offset array at a shift operation, transforms the program and propagates that information in an optimistic manner. The propagation continues until there are no more references to transform or one of the criteria have been violated. When a criterion has been violated, it may be necessary to insert an array copy statement into the program to maintain its original semantics. The inserted copy statement performs the intraprocessor data movement that was avoided with the OFFSET_SHIFT. The details of the algorithm are presented elsewhere [19].

It is important to note that due to the algorithm's optimistic nature, it is able to employ offset arrays in many difficult situations. In particular, it can determine when offset arrays can be exploited even when their definition and uses are separated by program control flow. This allows our stencil compilation strategy to eliminate the intraprocessor data movement in situations that other strategies would not even consider.

5 Reducing Interprocessor Movement

After eliminating the intraprocessor data movement via our offset array optimization, we now focus our attention on the interprocessor data movement that occurs during the calls to OFFSET_SHIFT. Due to the nature of offset arrays we are presented with many opportunities to eliminate redundant and partially redundant data movement.

Before discussing our strategy, we need to extend the definition of our OFFSET_SHIFT routine. We add an optional fourth argument that takes a regular section descriptor (RSD) [3]. The RSD is used to specify which data elements in the overlap areas of other dimensions are to be transferred along with the specified subgrid elements. This extension allows us to include "corner" elements that are a part of multi-offset arrays. The RSD will contain a null specifier "$*$" for the dimension being shifted. The default RSD would contain the range $1 : N$ for all other dimensions. Here's an example of an OFFSET_SHIFT along the second dimension that carries along the data in the overlap area from the top of the column but not the overlap area from the bottom: `OFFSET_SHIFT(SRC,+1,2,[0:N,*])`.

There are two key observations that will allow us to find and eliminate redundant interprocessor data movement. First, shift operations, including OFFSET_SHIFT, are commutative:

```
CSHIFT(CSHIFT(SRC,+1,1),-1,2) ≡
CSHIFT(CSHIFT(SRC,-1,2),+1,1)
```

And secondly, since all OFFSET_SHIFTs move data into the overlap areas of the subgrids, a shift of a large amount in a given direction and dimension may subsume all shifts of smaller amounts in the same direction and dimension. Or more formally, an OFFSET_SHIFT of amount i in dimension k is redundant if there exists an OFFSET_SHIFT of amount j in dimension k such that $|j| > |i|$ and $sign(j) = sign(i)$. Given these two points, we proceed to eliminate redundant data movement in the following manner.

First we reorder the statements within basic blocks so that OFFSET_SHIFT calls are grouped into maximal sets; *i.e.*, as many calls as possible are made adjacent. This is accomplished by applying our *context partitioning* optimization [20]. From this point on we then restrict our focus to the individual groups of OFFSET_SHIFT calls.

Next we use the commutative property to rewrite all the shifts for multi-offset arrays such that OFFSET_SHIFTs for the lower dimensions occur first and are used as input to the OFFSET_SHIFTs for higher dimensions. We then reorder all the calls to OFFSET_SHIFT, sorting them by the shifted dimension.

We now scan the OFFSET_SHIFTs for the first dimension and keep only the largest shift amount in each direction. All others can be eliminated as redundant.

Next we process the OFFSET_SHIFTs for each higher dimension in ascending order by performing the following three actions. First we scan the OFFSET_SHIFTs for the given dimension to determine the largest shift

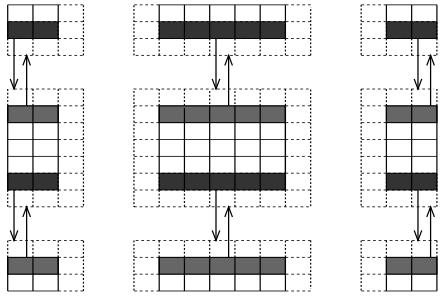

Figure 2: First half of 9-point stencil communication

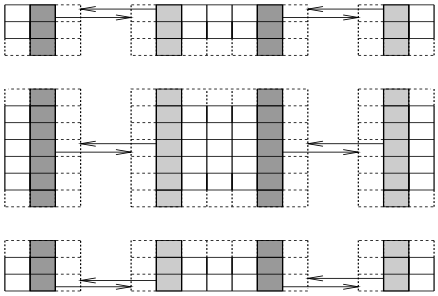

Figure 3: Second half of 9-point stencil communication

amount in each direction. Secondly, we look for source arrays that are already offset arrays, indicating a multi-offset array. For these, we use the annotations associated with the source array to create an RSD to be used as the fourth argument in the call to OFFSET_SHIFT. As with shift amounts, larger RSDs subsume smaller RSDs. Finally, we generate a single OFFSET_SHIFT in each direction, using the largest shift amount and including the RSD as needed – all other OFFSET_SHIFTs for that dimension can be eliminated.

This eliminates all communication for an offset array, except for a single message in each direction of each dimension. The number of messages is thus minimized.

As an example, consider again the 9-point stencil computation presented in Section 2.1. The original stencil specification required twelve CSHIFT intrinsics. After applying the above transformations, only the following four calls are required:

```
CALL OFFSET_SHIFT(SRC,-1,1)
CALL OFFSET_SHIFT(SRC,+1,1)
CALL OFFSET_SHIFT(SRC,-1,2,[0:N+1,*])
CALL OFFSET_SHIFT(SRC,+1,2,[0:N+1,*])
```

Figures 2 and 3 display the data movement that results from these calls. The figures contain a 5×5 subgrid (solid lines) surrounded by its overlap area (dashed lines). Portions of the adjacent subgrids are also shown. Figure 2 depicts the data movement specified by the first two calls. The data movement of the last two calls is shown in Figure 3. Notice how the last two calls pick up data from the overlap areas that were filled in by the first two calls, and thus they populate all overlap area elements needed for the subsequent computation.

6 Optimizing the Computation

Once the communication optimization has been completed, we must optimize the performance of the stencil on each node. Our strategy involves the following compiler optimizations to improve data locality:

1. Improve the order of memory accesses through loop permutation [9].
2. Improve loop balance through *unroll-and-jam* and *scalar replacement* [8, 7].

Note that strip-mine-and-interchange can be included here [27]. We have omitted it because of its relative instability and the large amount of cache reuse that already exists in stencil computations [22, 12]. In the rest of this section we give an overview of loop permutation, unroll-and-jam and scalar replacement.

6.1 Loop Permutation

Not all loops exhibit good cache locality, resulting in idle computational cycles while waiting for main memory to return data. For example, in the loop,

```
      DO 10 I = 1, N
        DO 10 J = 1, N
 10       B(I,J) = A(I,J) + A(I+1,J)
```

references to successive elements of `B` and `A` are a long distance apart in number of memory accesses (this assumes Fortran's column-major storage). Most likely, current cache architectures would not be able to capture the potential cache-line reuse available because of the volume of data accessed between reuse points. With each reference to `B(I,J)` and `A(I+1,J)` being a cache miss, the loop would spend a majority of its time waiting on main memory. However, if we interchange the `I`- and `J`-loops to get

```
      DO 10 J = 1, N
        DO 10 I = 1, N
10        B(I,J) = A(I,J)+ A(I+1,J)
```

the references to successive elements of `B(I,J)` and `A(I+1,J)` immediately follow one another. In this case, we have attained locality of reference for `B(I,J)` and `A(I+1,J)` by moving reuse points closer together. The result will be fewer idle cycles waiting on main memory. For a more complete discussion of loop permutation see Wolf and Lam [27], Kennedy and McKinley [18] and Carr, *et al.* [9].

6.2 Scalar Replacement

Even with better cache performance through loop permutation, a loop may still not perform as well as possible. If a loop is memory bound, then its balance must be lowered. Balance can be lowered by reducing the number of memory references in a loop by replacing references to arrays with sequences of scalar variables. In the code shown below,

```
      DO 10 J = 1, N
        DO 10 I = 1, N
10        B(I,J) = A(I,J)+ A(I+1,J)
```

the value accessed by `A(I,J)` is defined on the previous iteration of the loop by `A(I+1,J)` on all but the first iteration. Using this knowledge, the flow of values between the references can be expressed with scalar temporaries as follows.

```
      DO 10 J = 1, N
        T1 = A(1,J)
        DO 10 I = 1, N
          T0 = A(I+1,J)
          B(I,J) = T0 + T1
10        T1 = T0
```

Since the values held in scalar quantities will probably be in registers, the load of `A(I,J)` has been removed, resulting in a reduction in the memory cycle requirements of the loop (the register copy, `T1 = T0`, can be removed by unrolling `I`) [10]. This transformation is called *scalar replacement* and is described in detail elsewhere [7].

6.3 Unroll-And-Jam

Unroll-and-jam is a transformation that can be used in conjunction with scalar replacement to improve the performance of many memory-bound loops [1, 2, 6]. The transformation unrolls an outer loop and then jams the resulting inner loops back together. Using unroll-and-jam, more computation can be introduced into an innermost loop body without a proportional increase in memory references. For example, the loop:

```
      DO 10 J = 1, 2*N
        DO 10 I = 1, N
10        B(I,J) = A(I,J)+ A(I,J+1)
```

after unroll-and-jam of `I` by a factor of 1 becomes:

```
      DO 10 J = 1, 2*N, 2
        DO 10 I = 1, N
          B(I,J) = A(I,J)+ A(I,J+1)
10        B(I,J+1) = A(I,J+1)+ A(I,J+2)
```

In the original loop, one floating-point operation and three memory references are left after scalar replacement, giving a balance of 3. After applying unroll-and-jam, two floating-point operations and five memory references exist in the loop, giving a balance of 2.5 (the second reference to `A(I,J+1)` can be scalar replaced). If the original loop were memory bound, the unroll-and-jammed loop would perform better as it has a lower balance.

Carr and Kennedy describe an automatic method for applying unroll-and-jam [8]. Their method computes the unroll amount for a loop that best balances the nest with respect to a target architecture while limiting register pressure. For a detailed discussion of this method, see [8].

7 Related Work

One of the first major efforts to specifically address stencil compilation for a distributed-memory machine was the stencil compiler for the CM-2 [4, 5]. Like our strategy, they eliminated the intraprocessor data movement. They also optimized the interprocessor data movement by exploiting the CM-2's polyshift communication [15]. The final computation was performed by hand-optimized library microcode that took advantage of different loop transformations.

However, the CM-2 stencil compiler had many limitations. It could only handle single-statement stencils. The stencil had to be specified using the CSHIFT intrinsic; no array-syntax stencils would be accepted. Since the compiler relied upon pattern matching, the stencil had to be in a very specific form: a sum of terms, each of which is a coefficient multiplying a shift expression. No variations were possible. And finally, the programmer had to recognize the stencil computation, extract

it from the program and place it in its own subroutine to be compiled by the stencil compiler.

Our compilation scheme handles a strict superset of patterns handled by the CM-2 stencil compiler. Our strategy will optimize single-statement stencils, multi-statement stencils, CSHIFT intrinsic stencils, and array-syntax stencils all equally well. And since our optimizations are made to be included into an HPF compiler, our optimizations will benefit those computations that only slightly resemble stencils.

There are also some other commercially available compilers that can handle certain stylized, single-statement stencils. The MasPar Fortran compiler will avoid the intraprocessor data movement for single-statement stencils written using array notation. This is accomplished by scalarizing the Fortran 90 expression (avoiding the generation of CSHIFTs) and then using dependence analysis to find loop-carried dependences which indicate interprocessor data movement. Only the interprocessor data is moved, and no local copying is required. However, the compiler still performs all the data movement for single-statement stencils written using SHIFT intrinsics. This strategy is shared by many HPF/Fortran 90 compilers that really only want to handle scalarized code. As with the CM-2 stencil compiler, our methodology is a strict superset of this strategy.

In general, there have been several different methods for handling specific subclasses of stencil computations. In this paper, we have presented a strategy that encompasses all of them and more.

8 Conclusion

In this paper, we have presented a general compilation scheme for compiling HPF stencil computations for distributed-memory architectures. The strategy optimizes such computations by eliminating unnecessary intraprocessor data movement resulting from CSHIFT intrinsics, eliminating redundant interprocessor data movement, and optimizing memory accesses via loop-level transformations. The optimizations are general enough to be included in a general-purpose HPF/Fortran 90 compiler as they will benefit many computations, not just those that fit a stencil pattern.

References

[1] A. Aiken and A. Nicolau. Loop quantization: An analysis and algorithm. Technical Report 87-821, Dept. of Computer Science, Cornell University, March 1987.

[2] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[3] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[4] R. G. Brickner, W. George, S. L. Johnsson, and A. Ruttenberg. A stencil compiler for the Connection Machine models CM-2/200. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[5] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[6] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[7] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[8] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[9] Steve Carr, Kathryn McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 252–262, Santa Clara, California, 1994.

[10] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6:45–57, January 1981.

[11] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C.-W. Tseng. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[12] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization. *SIGPLAN Notices*, 30(6):279–280, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[13] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[14] R. Fatoohi. Performance analysis of four SIMD machines. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[15] W. George, R. Brickner, and S. L. Johnsson. Polyshift communications software for the Connection Machine systems CM-2 and CM-200. *Scientific Programming*, 3(1):83, Spring 1994.

[16] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[17] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[18] K. Kennedy and K. McKinley. Optimizing for parallelism and memory hierarchy. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 323–334, Washington, DC, July 1992.

[19] K. Kennedy, J. Mellor-Crummey, and G. Roth. Optimizing Fortran 90 shift operations on distributed-memory multicomputers. In *Languages and Compilers for Parallel Computing, Eighth International Workshop*, Columbus, OH, August 1995. Springer-Verlag.

[20] K. Kennedy and G. Roth. Context optimization for SIMD execution. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[21] K. Knobe, J. Lukas, and M. Weiss. Optimization techniques for SIMD Fortran compilers. *Concurrency: Practice and Experience*, 5(7):527–552, October 1993.

[22] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 63–74, Santa Clara, California, 1991.

[23] J. R. Rice and J. Jing. Problems to test parallel and vector languages. Technical Report CSD-TR-1016, Dept. of Computer Science, Purdue University, 1990.

[24] G. Sabot. A compiler for a massively parallel distributed memory MIMD computer. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[25] G. Sabot. Optimized CM Fortran compiler for the Connection Machine computer. In *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*, Kauai, HI, January 1992.

[26] J. T. Schwartz. Optimization of very high level languages – I. Value transmission and its corollaries. *Computer Languages*, 1(2):161–194, 1975.

[27] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

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