Massively Parallel Data Optimization

Kathleen Knobe*  Joan D. Lukas†  Guy L. Steele Jr.‡

Abstract

An optimizing compiler for a data parallel programming language can significantly improve program performance on a massively parallel computing system by incorporating new strategies for allocating array elements to processors. We discuss techniques for automatic layout of arrays in a Fortran compiler supporting Fortran 8x array features and targeted to the Connection Machine computer system. Our goal is primarily to minimize the costs of moving data between processors and secondarily to minimize memory usage. Improved array layout may allow communications operations to be eliminated or to be replaced by more specialized communications operations having lower cost. We exhibit and discuss thoroughly a typical example of a code fragment that can be improved by a factor of 2 in memory consumption and a factor of 20 in speed.

1 Introduction

An optimizing compiler for a data parallel programming language can significantly improve program performance on a massively parallel computing system by incorporating new strategies for allocating array elements to processors.

We have designed and implemented a Fortran compiler for the Connection Machine® computer system at COMPASS on behalf of Thinking Machines Corporation. Work on improvements and extensions is currently in progress. The optimizations described here are under design for incorporation into a future version of the compiler.

The language implemented is Fortran 77, extended by Fortran 8x array features, including some features (FORALL and vector-valued subscripts) now listed in the “removed extensions” section of the latest ANSI proposed standard for Fortran [3]. These features suit the data parallel computing style, which associates each processor’s memory.

The Connection Machine system [6,12] supports the data parallel style by providing thousands of hardware processors that can operate on as many data elements simultaneously. A full Connection Machine system includes 65,536 physical processors, each with its own memory. Each processor can perform all of the usual arithmetic and logical operations on integers and floating-point numbers stored within its own memory. Parallel data structures are spread across the processors, with a single element stored in each processor’s memory.

The Connection Machine is accessed through a front end, which provides the programming environment. The front end holds scalar data, and also controls execution of the data parallel program. Program steps involving parallel data are passed over an interface to the Connection Machine, where they are broadcast for simultaneous execution by all the processors.

Interprocessor communication is implemented by a high-speed routing network. In a send operation, each processor contains data to be sent and a pointer to a processor (possibly itself) that is to receive it. In a get operation, each processor contains a pointer to a processor containing data to be copied back to the first processor. A send is faster than a get, but a get allows data from one processor to be copied to many other processors at once.

The Connection Machine Model CM-2 singles out certain patterns of communication for special hardware and microcode support. Cartesian grids of any number of dimensions can be embedded within the boolean hypercube structure used by the router. Array elements that are neighbors along any dimension are allocated to processors that are neighbors within the hypercube structure. As a result, a single-position shift along any axis of such a grid can be performed much faster than the general case of the send instruction. Such operations are called NEWS operations (for North-East-West-South); the simplest one is called simply get-from-news.

Complex operations on Cartesian grids are also directly supported as single Connection Machine instructions. These include spread and scan. A spread operation can take any row of a matrix and copy it into all the other rows. A scan operation takes a combining operation (such as add, max, or logior) and performs a parallel prefix computation [9,7] on each row of a matrix. For example, scan-with-add computes running totals for each row. Both spread and scan generalize to columns instead of rows and to any number of dimensions.

As a rough rule of thumb, if a 32-bit addition (to be performed by all 65,536 processors) takes 1 time unit then

- a 32-bit NEWS transfer takes 4 units
- a spread takes 25 units
- a send takes 80 units
- a get takes 180 units

One unit is approximately 16 microseconds. These figures are only useful rough estimates; actual times depend to some extent on the virtual processor configuration.

Our goal is to minimize the cost of interprocessor communication. Appropriate allocation of data to processors frequently

*Compass, Inc. (formerly Massachusetts Computer Associates, Inc.)
†University of Massachusetts at Boston and Compass, Inc.
‡Thinking Machines Corporation
allows a general communication operation (such as send) to be replaced by a faster one (such as get-from-news) or eliminated altogether.

2 Examples

As an example of the optimization issues addressed here, consider this code fragment:

```plaintext
10 B(1:N) = B(1:N) * S
20 U1(1:N) = B(1:N) / B(1:N)
30 U2(1:N) = B(2:N) + L11(1:N)
40 U3(1:N) = B(3:N) * L11(1:N)
60 U5(1:N) = B(5:N) * L11(1:N)
```

(This fragment is loosely based on a fragment of a vectorized tridiagonal solver subroutine. For emphasis we have written out all subscripts explicitly.)

The simplest and most naive way to arrange the data is to lay out each array with one element per processor, starting at processor zero. See figure 1. This is the "canonical" layout used by the compiler in the absence of other criteria. This places L11 and each U in the same set of processors so they align, but B is spread across the processors in a manner that does not necessarily bear any useful relationship to the other arrays. Alignment of L11 and each U with the sections of B requires motion.

This strategy requires a total of 8 memory slices: one for B, one for each U, one for L11, and a temporary used to hold a moved copy of L11 or B(1:N) in each of statements 30-60. The total communication cost depends on which of two strategies is used for statements 30-60. If L11 is moved to align with B(1:N), and then the result of the multiplication is moved to U1, a total of 9 send operations will be required (1 for statement 20, and 2 each for statements 30-60). If B(1:N) is moved to align with L11 and U1, then only 5 send operations are required in all. The time for motion will thus be either 720 or 480 units.

In summary, the total cost of lines 10-60 for the naive allocation strategy is

- Total time for operations = 6
- Total time for motion = 720 or 400
- Total number of memory slices = 7

An obvious hypothesis is that aligning the one-dimensional arrays with some row of B, say the first, would reduce communication (figure 2). This eliminates one communication step (that between L11 and B(1:N)), but is not the best possible improvement. Observe that each send operation can be replaced by a sequence of NEWS moves, copying values from row to row of B, giving

- Total time for operations = 6
- Total time for motion = 80 or 40
- Total number of memory slices = 7

In this particular case it might be better to lay out the first dimension of the array B as a serial axis rather than a parallel axis; that is, successive elements of B along that dimension will occupy successive locations within the same processor rather than the same location within different processors. See figure 3. In this manner no communication is required for any of the assignment statements, and no temporary locations are required. On the other hand, the multiplication of B by S in line 10 now requires 5 separate computational steps. The cost of lines 10-60 is

- Total time for operations = 10
- Total time for motion = 0
- Total number of memory slices = 10

Another possibility is the layout shown in figure 4. Here B is laid out in parallel and each U is aligned with the appropriate section of B. Copies of L11 are still required, so one extra memory slice is needed for a temporary.

- Total time for operations = 6
- Total time for motion = 40 or 80
- Total number of memory slices = 7
This layout has the same time and space possibilities as in figure 2, with the NEWS costs of the two motion strategies reversed. But this layout leads to an even better one.

Note that the U arrays are now in disjoint processor sets and can share a single slice of memory (figure 5). Then the four multiplications can be collapsed into a single multiplication to be carried out for all of U2, U3, U4, and U5 simultaneously, resulting in

Total time for operations = 3
Total time for motion = 40 or 80
Total number of memory slices = 3

This example illustrates a need to optimize the static layout of user-declared arrays. Sometimes dynamic layout is required, as in the following example.

```fortran
DO 10 J=1,M
  TEMP(J,1:N) = A(J,1:N)
  A(J,1:N) = B(J,1:N)
  B(J,1:N) = TEMP(J,1:N)
10 CONTINUE
```

In this case it is desirable to assign TEMP to one fixed set of locations. Its uses on different iterations do not interact (there is no dependence among them), so TEMP may be allocated to a different set of operations on each iteration. In other words, the compiler can in effect rewrite the code to behave something like this:

```fortran
DO 10 J=1,M
  TEMP(J,1:N) = A(J,1:N)
  A(J,1:N) = B(J,1:N)
  B(J,1:N) = TEMP(J,1:N)
10 CONTINUE
```

## 3 Overview of Solution

Data optimization processing takes place after the Internal Representation (IR) of the source code has been created to expose all source level computations and after this IR has been optimized. IR requiring exact locations, such as for data motion or context setting, has not yet been exposed. The algorithms take machine-specific time and space cost functions as parameters and can therefore be applied to a variety of machines by using different cost functions.

The first phase of data optimization is the Align phase, which determines preferred relative alignments among occurrences of named array sections based on dependences. Resolving conflicts among these preferences indicates a need for data motion. The Inter phase determines the best location for intermediate results, tracking multiple copies of array sections produced by conflict resolution in Align and by Inter's own processing of previously analyzed statements and subexpressions.

When Align is unable to allocate sections according to their preferred relative alignment, motion is required. The Motion phase determines what data is to be moved, which motion instructions are needed and their initial placement in the generated code. A Mini-Vectorizer is then activated to perform vectorization-like transformations on the code introduced by previous phases. The exact allocation of array occurrences is now determined and code is exposed in the IR for context setting and motion. Standard global optimization is then performed on the newly added code.

The major compiler phases that are unique to data optimization are Align, Inter, Motion, and the Mini-Vectorizer. These are discussed in more detail in the remainder of this paper.

## 4 Align

The align phase determines the allocation of occurrences of named arrays and their sections. Align analyzes usage patterns in the source, noticing situations where the best allocation of one array occurrence depends on the allocation of another array occurrence. We call these situations preferences.

An Identity Preference is between corresponding dimensions of a definition and use of the same array and indicates a preference to align identical elements of the array in the same processors.

A Conformance Preference is between corresponding dimensions of occurrences of different arrays that are operated on together and indicates a preference to align corresponding elements of distinct arrays in the same processors.

An Independence Anti-Preference is associated with a single dimension of an array occurrence (not a relationship between them) and indicates a preference to store the distinct elements along that dimension in different processors.

The three characteristics above refer to allocation across processors. There is, of course, a Uniqueness Requirement that each dimension must have a unique effect on the storage of the array elements over the entire memory space of the machine but this effect could well be with respect to the storage within processor memory. Therefore, even with this requirement, the value in a particular dimension may legitimately have no effect on the allocation of an array across processors.
These preferences are discussed below with their implications for the Connection Machine. Some implications for other architectures are presented as well.

Identity Preference: This preference is between corresponding dimensions of a definition and a use of the same array if there is a true dependence between them. Note that the allocation of a definition and use of the same array may be determined independently if there is no true dependence between them. (Also note that output dependences and anti-dependences have no impact on allocation.)

The identity preferences in our example are shown below:

In the assignment to \( U_2 \) there is a conformance preference between the second dimension of \( B \) and the only dimension of \( L_{11} \). If these two dimensions are not allocated identically, motion to align will be required. Restrictions may be associated with each conformance preference. For example, the conformance preference in the statement 30 holds only when the first subscript of \( B \) is 2. Statement 40 has the same preference between the second dimension of \( B \) and \( L_{11} \), but the restriction holds when the first subscript of \( B \) is 3.

On the Connection Machine system, motion will be minimized if dimensions associated by conformance preferences are allocated so that corresponding elements align. On a machine with memory bank contention, a conformance preference means that delays will be minimized if we allocate these two sections or at least the corresponding elements of the sections in distinct memory banks. On a MIMD machine with local memory, a conformance preference means that the two sections or at least the pairs of elements determined by this correspondence must reside in the same processor to minimize data motion.

Independence Anti-Preference: Both the identity preference and the conformance preference attempt to reduce motion. This anti-preference attempts to preserve maximum parallelism. Without this preference, the other preferences can always be resolved by storing large sections or even entire arrays in a single processor. While such a resolution will result in no data motion, it may reduce parallelism and increase memory requirements. (In our example, loss of parallelism occurs if we break the independence anti-preference on the first dimension of \( B \) and store this dimension serially within the processor memory, as shown in figure 3. With this allocation, a single operation on all of \( B \) must be transformed into five operations, one on each row of \( B \).)

The independence anti-preference is a characteristic of a specific dimension occurrence if that dimension contains a potentially parallel subscript, that is if the subscript is not scalar. The independence anti-preference is included so the algorithm will not always decide to minimize motion at the expense of memory and parallelism but will make an explicit choice based on cost by associating a cost with each independence anti-preference.

The relationships described above are represented as arcs in a graph whose nodes are dimensions of occurrences of arrays and array sections. Such a graph can be consistent or in conflict. It is consistent if all the preferences can be satisfied. It is in conflict if preferences contradict each other. In the above example the preference graph contains a conflict. The second dimension of \( B \) must align with \( L_{11} \) where the first subscript of \( B \) is 1 and also where it is 2. If the first dimension of \( B \) is stored across the processors, the conformance preferences cause a conflict. If the first dimension is stored down processor memory, the independence preference causes a conflict. The graph is therefore in conflict.

If the preference graph is consistent, no motion need occur at runtime. Conflict resolution involves either the introduction of motion or, if a dimension is to be stored down the processor memory, additional memory requirements and possibly loss of parallelism. When a conflict is broken, a preference arc is removed from the preference graph. Removal of a conformance or identity arc implies the necessity of data motion. To enable the Motion phase to insert this motion, Align records information about the removed arcs. Removal of an independence arc results in increased storage requirements per processor and possible reduction in parallelism, but no data motion is required.
Align processing begins by building the preference graph for each basic block. If the graph is in conflict, this will involve conflict resolution. Sometimes resolving a conflict can be accomplished in several ways which may differ significantly in their cost. Costs are associated with the arcs in the graph and Align attempts to find the least cost resolution. Costs may include the cost of motion, the cost of memory, and the cost of lost parallelism.

When the block is made consistent, Align processing then proceeds up the control tree, a hierarchical representation of the control flow of the program with basic blocks at the leaves. At each stage in a bottom-up control tree walk, the consistent graphs for the subnodes of a control node are combined by taking their union and then including arcs representing the relationships that cross between the subnodes. When joining control nodes, new identity preferences may be introduced. Conformance preferences arise only within statements and therefore no additional conformance preferences can be introduced by joining the preference graphs of the control nodes on the way up the control tree.

Since the subgraphs (graphs for the children of the control node) are known to be conflict free, conflict resolution is performed incrementally by processing only the new identity preference arcs added between the subgraphs.

Since small local graphs are processed and then incrementally combined, the processing time is significantly faster than that of an equivalent global algorithm. Since it starts at the leaves of the control tree and works outward, the algorithm gives priority to preferences arising in inner loops.

The conflict in our example may be broken in a number of ways. Two are indicated below:

1. Store the first dimension of the B array down the processors so that one copy of L11 can align with all the appropriate sections of B (figure 3). This resolution breaks the independence preference.

2. Store B across the processors and align L11 with B(1,1:N) and each of U2, U3, U4, and U5 with the corresponding section of B (figure 4). This removes the conformance preference between L11 and the sections of B.

Option 1 uses more memory but incurs no loss of parallelism and results in no motion for execution of lines 10-60. This option results in

Total time for operations = 10
Total time for motion = 0
Total number of memory slices = 10

This may be the optimal result. On the other hand, the advantages of this option may well be offset by the necessity of either motion to serialize B from its parallel definition or executing the assignment defining B once for each value of the subscript in dimension 1. In any case, adoption of this strategy will be enlightening for the rest of the paper. Option 2 will be assumed. The effect of this decision will be that for each statement the LHS and the B section are now aligned. Motion will be required to align these with L11.

There are a number of possibilities for this motion. At a minimum, L11 will be moved to align with each LHS at a cost of 1, 2, 3, and 4 NEWS moves, respectively. The total motion time is ten times the unit cost of a NEWS move. At worst, each B section will be moved to align with L11, the operation performed there, and the result moved back to align with the LHS. This would require a total of 20 NEWS moves. This option results in

Total time for operations = 6
Total time for motion = 40 or 80
Total number of memory slices = 7

5 Inter

Once allocation of occurrences of named arrays has been determined, the compiler must deal with the proper allocation of array temporaries arising from operations, that is, it must decide where the operations will be performed. If Align was able to satisfy all preferences, then all operands in a statement as well as the destination of an assignment will have the proper alignment. In this situation the optimal location for each operation will be obvious. In many cases, however, some conflicts will have arisen, causing Align to break preference arcs. If the operands are not aligned, two or more possible locations for temporaries will exist.

The Inter phase, which chooses optimal locations for array temporaries, uses a dynamic programming algorithm over an expression tree. The locations considered are those of the named sections in the same statement. For each temporary, Inter considers each of these locations along with the cost of moving operands to each location to align them for operations. Let M(L1, L2) denote the cost of moving an array from location L1 to location L2. This cost is determined by the Connection Machine instruction(s) needed to effect the move. Then, for each operation @ and each location L, the cost of performing z @ y in L, denoted C(z @ y), is the minimum cost, over all considered pairs of locations L1 and L2, of computing z in L1 and then moving it to L2 added to the cost of computing y in L2 and moving it to L. That is,

\[ C(z @ y, L) = \min_{L1, L2} \left( C(z, L1) + M(L1, L) + C(y, L2) + M(L2, L) \right) \]

For an assignment statement z = y where z is in location Lz, the cost is given by

\[ C(z = y) = \min_{L} \left( C(y, L) + M(L, Lz) \right) \]

The statement level cost equation drives the computation for interior nodes of the right-hand side y.

Note that when an operand is moved (copied) from one location to another, an additional copy of it is available for use in subsequent statements. Inter keeps track of these multiple copies in determining minimum costs. The algorithm for tracking copies uses a mechanism similar to that of value numbering. A copy remains valid until an assignment to any of the elements in the original array section is made. Tracking of multiple copies can be done across control flow as well as within basic blocks.

To return to our example, let us assume that Align has decided to allocate B across the processors and to align L11 with B(1,1:N) in location L1 and each U$i$ with the appropriate section of B in location U$i$. Then statement 20 requires no alignment.
The operation will take place in $L_1$. For statement 30, the cost is given by

$$\min_L \left( C(B(2,1:N) \times L11(1:N), L) + M(L, L_2) \right)$$

The locations to consider for $L$ are $L_1$ (holding $L11$) and $L_2$ (holding $U2$ and $B(2,1:N)$). For $L = L_1$, the cost is

$$C(B(2,1:N) \times L11(1:N), L_1) + M(L_1, L_2)$$

To perform the multiplication in $L_1$ would require moving $B(2,1:N)$ to $L_1$, at a cost of one NEWS move. The total cost of the assignment would then be 2 NEWS moves. For $L = L_2$, the cost of performing the operation in $L$ is one NEWS move and the cost of moving the result to $L_2$ is 0. Thus the total cost is one NEWS move. This option is clearly preferable.

As a result of the choice just made, there will now be two copies of $L11$, the original in $L_1$ and a new one in $L_2$. For statement 40, there are now three possible locations: the two locations holding $L11$, and also $L_3$, which holds $U3$ and $B(3,1:N)$. An analysis similar to the one performed above will determine that the lowest cost will be obtained by performing the operation in $L_3$. In order to do this, $L11$ will be copied to $L_3$. The cost of moving $L11$ from its original location to $L_3$ is 2 NEWS moves while the cost of moving from the copy at $L_2$ is 1 NEWS move. Thus this copy will be used.

As the algorithm continues for the remaining statements, each multiplication will be carried out in $L_1$, with the most recent copy of $L11$ at $L_{i-1}$ being copied at a cost of 1 NEWS move.

At the end of the Align phase, the cost of our example depended on an arbitrary choice of locations for the intermediate operands. Inter assures the best choice and, in fact, does better than this by tracking multiple copies.

Total time for operations = 6
Total time for motion = 16
Total number of memory slices = 7

Although it is not relevant to our example, we describe an additional optimization performed by Inter. The cost of computation for a complex arithmetic expression may sometimes be reduced if reordering of operations is permitted. When such reordering is permitted, a set of adjacent addition and subtraction nodes can be viewed as an $n$-ary addition, with some operands preceded by a unary minus. The same is true for a sequence of multiplication and division operations. Inter will exploit this fact by regrouping operands to minimize the cost of motion among them. Operands in the same location are combined before joining them with other operands. For example, if statement 30 in our example had instead been

$$30 \ U2(1:N) = B(1,1:N) \times B(2,1:N) \times L11$$

then $L11$ would first be multiplied by $B(1,1:N)$ in location $L_1$. The result would then be copied to $L_2$ for multiplication by $B(2,1:N)$ and assignment to $U2$.

6 Motion

After Align and Inter have determined where each occurrence of an array or section will reside, the Motion phase determines which references require motion, the position of the motion code in the IR, the type of motion required and the amount of data to be moved.

Motion is required:

- when Align had an identity or conformance preference that it had to break (breaking an independence anti-preference does not require motion)
- when Inter decides to move a named section
- when Inter decides to move an intermediate result

Motion will attempt to collect motions in a basic block to the same place within the block so that they can be combined to minimize cost. For example, two disjoint sets of values may be more cheaply sent together than separately. Also one motion may totally subsume another. If this is the case, only the larger need be performed. An identity preference may exist because data independence of the subscripts cannot be proved at compile time. In such cases, Motion may insert a scalar test, which will be executed at runtime and possibly save the cost of motion.

In our example, the motions of $L11$ is of the right form to be converted to a spread but it is not moved far enough for conversion to be cost effective. For this example Motion simply inserts motion code after the definition of $L11$, having no additional effect on the costs.

7 Mini-Vectorizer

Upon completion of the previous phases, a number of opportunities for vectorizer-like transformations that further improve performance are uncovered and are handled by this phase. These transformations are with respect to motion code and therefore cannot be performed until allocation and motion have been determined. The following transformations are processed by this phase.

- Motion is associated with a particular loop in a loop nest.
- If loops are interchanged the total cost of the motion may be affected, sometimes significantly.
- If a loop itself is not vectorizable and a series of definitions or uses of individual elements in an array remain, motion between the front end and the Connection Machine is required. The direction of the motion depends on whether the references are uses or definitions. The mini-vectorizer may be able to transform the motion from elemental motion within a loop to array motion outside the loop even if the operations are not vectorizable.
- If all arrays are stored in locations determined by their shapes, the only factor determining when arrays can share a slice of memory is their lifetimes. With the allocation scheme based on preferences, even if two arrays are live at the same time, they can be allocated to the same memory address if they share no processors.
- The same operation performed on distinct sections of the same array or on different arrays that do not share processors can sometimes be converted to a single operation on all sections at once.

The last case above occurs in our example. The four sections of $B$ are being multiplied by different copies of $L11$. In order to convert this to a single multiply, we must prove that
The four locations (set of processors) for the operations are distinct. (They may, however, have array elements in common such as L11.)

No two LHSs may contain the same array element.

The same operation is used on each section.

We allocate a single memory address for all copies of L11, a single memory address for U2, U3, U4, and U5, and perform a single multiplication. This transformation saves seven slices of memory and three multiplications.

The algorithms described above have produced two possibilities for this code sequence, with costs of

Total time for operations = 10
Total time for motion = 0
Total number of memory slices = 10

and

Total time for operations = 3
Total time for motion = 16
Total number of memory slices = 3

Recall that the numbers for the naïve allocation strategy were

Total time for operations = 720 or 400
Total number of memory slices = 7

On this fragment data layout optimization can either

- improve memory usage by a factor of 2 and execution speed by a factor of 20 or more or
- increase memory requirements and time for arithmetic operations while eliminating all data motion.

8 Generalized Common Subexpressions

Generalized common subexpression elimination is an optimization performed by the Inter phase. It is not relevant to our example and therefore was not presented above, however, it can have a significant effect on the cost of motion for for a number of applications and on a variety of machines and is therefore included here.

If a SIMD operation is performed on a section or sections and subsequently the same operation is performed on different sections of the same array(s), the two operations can be combined. This is not exactly a common subexpression because the sections may not be the same, but, except for the context (the set of enabled processors), the operations are, in fact, identical. By adjusting the context, the expressions can be combined. The operations involved may be either local per processor computations or they may involve motion operations. Finding common motion operations is the more interesting (and cost effective) case.

Consider the following assignment:

\[
\]

&

\[
+ A(2:N,1:N-1) + A(1:N-1,1:N-1)
\]

Neglecting boundary conditions, the nature of this assignment may be captured in the following schematic form:

\[
A = A + \text{west}(A) + \text{north}(A) + \text{northwest}(A)
\]

By rewriting \(\text{northwest}(A)\) as \(\text{north}(\text{west}(A))\) and then noting that \(\text{north}\) distributes over +, we get

\[
A = (A + \text{west}(A)) + \text{north}(A + \text{west}(A))
\]

Viewed this way the two occurrences of \(A + \text{west}(A)\) can be thought of as a kind of common subexpression, but our informal notation, because it glosses over the boundary conditions, hides the fact that different sets of values are involved. If we cast this back into correct Fortran (using a temporary variable because Fortran 8x arbitrarily does not permit one to subscript an expression), we find that they are not quite common subexpressions after all:

\[
\text{TEMP} = A(2:N,2:N) + A(1:N-1,2:N)
\]

\[
A(2:N,2:N) = \text{TEMP} + \text{TEMP}(?,?)
\]

This is not exactly a common subexpression, but our informal notation, because it glosses over the boundary conditions, hides the fact that different sets of values are involved. If we cast this back into correct Fortran (using a temporary variable because Fortran 8x arbitrarily does not permit one to subscript an expression), we find that they are not quite common subexpressions after all:

\[
\text{TEMP} = A(2:N,1:N) + A(1:N-1,1:N)
\]

\[
A(2:N,2:N) = \text{TEMP}(1:N,2:N) + \text{TEMP}(1:N,1:N-1)
\]

A solution of this general form will work, but only if the common subexpression is generalized to contain the union of the set of values required by all uses. Thus \(\text{TEMP}\) will contain more values than either of its uses requires:

\[
\text{TEMP} = A(2:N,1:N) + A(1:N-1,1:N)
\]

This is worthwhile when the cost of computing the union of two (or more) sets of values will be cheaper than computing each set separately. This frequently occurs in codes where various shifted sections of a single array are to be combined.

To notice opportunities for this optimization we need to remove attention from the specific elements being operated on and focus instead on the array names of the operands and on their offsets. Specifically, \(A \oplus B\) and \(A' \oplus B'\) are general common subexpressions if

- \(A\) and \(A'\) are sections of some array, and \(B\) and \(B'\) are sections of some array
- the relationship between the corresponding subscripts of \(A\) and \(A'\) is the same as the relationship between the corresponding subscripts of \(B\) and \(B'\)
- flow analysis indicates that the arrays involved have not been altered in the interim

If the operation is a motion operation as opposed to a local computation, then \(B\) and \(B'\) may be viewed as the target set of processors.

9 Related Work

There is a great deal of other literature on compiling Fortran for parallel computers. Much of this work addresses techniques for extracting parallelism from programs that may have been written in a sequential style, for example [2,5,8]. We use some of these parallelizing techniques in our "mini-vectorizer" phase, but regard them as complementary rather than central to our work.

Allen and Kennedy [2] further point out that appropriate program transformations may not only improve opportunities for parallelism but reduce synchronization overhead. This happens not to apply to our particular practical problem, because the SIMD nature of the Connection Machine architecture obviates the need for explicit synchronization, but might well be relevant.
to an adaptation of our data layout techniques to an architecture with asynchronous parallelism.

We pursue questions of data structure transformation rather than program transformation. Our main goal is to determine data layouts that minimize communications overhead in architectures with non-shared memory, once opportunities for parallelism have already been made explicit. The IVTRAN compiler [10,11] addressed similar concerns of array layout; our work differs in considering a richer space of layout possibilities and in handling a larger set of communications primitives spanning a spectrum of functionality/speed tradeoffs.

Crystal [4] also deals with questions of data layout. It accepts and transforms programs in a language based on recursion equations, and targets systolic architectures. Crystal also emphasizes the detection of common subexpressions so that results computed in one processor may be shared with other processors. However, Crystal apparently finds only standard cse's that are costly. The notion of generalized common subexpressions covers more cases.

10 Summary and Conclusions

In an earlier paper [1] on the Connection Machine Fortran compiler, it was demonstrated that the Connection Machine is a natural target for compilation of Fortran 8x array constructs. However, careful attention to allocation of data across processors according to usage patterns is necessary in order to exploit the full benefits of the Connection Machine architecture. Naive allocation strategies give rise to the need for large amounts of expensive motion code in order to align data for parallel operations.

The compilation approach presented in this paper performs analysis of usage patterns and determines the allocation strategy for each occurrence of an array section. The potential performance impact of this compilation technology is measured in orders of magnitude rather than percentages. The problem as well as the solution apply to any SIMD machine with local memory. Aspects of the analysis apply to a variety of architectures where the presence of data in the right place at the right time can significantly improve performance.

References


