Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines*

Bradford L. Chamberlain   Sung-Eun Choi   E Christopher Lewis
Calvin Lin†   Lawrence Snyder   W. Derrick Weathersby

University of Washington, Seattle, WA 98195-2350 USA
† University of Texas, Austin, TX 78712 USA

Abstract. This paper describes a new approach to compiling and optimizing array languages for parallel machines. This approach first decomposes array language operations into *factors*, where each factor corresponds to a different communication or computation structure. Optimizations are then achieved by combining, or *joining*, these factors. Because factors preserve high level information about array operations, the analysis necessary to perform these join operations is simpler than that required for scalar programs. In particular, we show how data parallel programs written in the ZPL programming language are compiled and optimized using the *factor-join* approach, and we show that a small number of factors are sufficient to represent ZPL programs.

1 Introduction

Array languages such as Fortran 90 and ZPL introduce compilation issues not encountered in the context of scalar languages such as Fortran 77 or C. Certain problems vanish, others become more complicated, and still others call for techniques not previously available. This paper shows how compilers can exploit this new context. Using the ZPL compiler as an example, we describe a new approach to compiling array languages that is particularly useful when compiling for parallel machines.

To see how an array language can simplify the compilation process, consider the problem of generating explicit interprocessor communication from a scalar language. One challenging problem in compiling scalar Fortran 77 code is performing *message vectorization* [9]—the transmission of multiple values in a single message rather than in separate messages. In scalar languages, the base unit of computation is a single value, and communication is generated per-value, making vectorization an optimization task. But in array languages the unit of computation is a contiguous sub-array, resulting in natural and automatic vectorization, as illustrated in Fig. 1.

To illustrate how an array language's high-level concepts motivate new optimizations, consider ZPL's reduce operators which combine data elements of

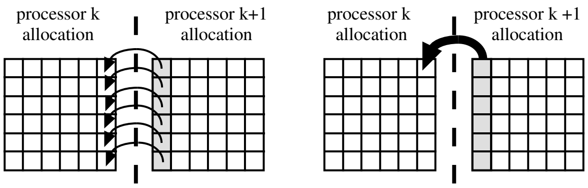
* This research was supported in part by ARPA Grant N00014-92-J-1824.

Appeared in Proceedings of the Workshop on Languages and Compilers for Parallel Computing, 1996.

an array using an associative operator such as plus, logical-and, or minimum. The implementation of reduce requires local computation, a global reduction, and a broadcast. The communication components of consecutive reduces can be merged to yield significant performance improvements in much the same way that message vectorization optimizes access to consecutive scalar values. This optimization is illustrated in Fig. 2. In scalar languages, there is little chance for the compiler to optimize communication in this way. Even if the reduce concept is abstracted to a procedure, it is difficult for a compiler to recognize, much less realize, the *opportunity* to perform the optimization.

```
      do 10 j = 1,n
         do 10 i = 1,m
10    A(i,j) = A(i,j+1)
```

```
[1..m,1..n] A := A@east;
```



(a) Scalar language. (b) Array language.

Fig. 1. Message vectorization in parallelized Fortran 77 and ZPL.

The ZPL compiler achieves these sorts of optimizations with the *factor-join* compilation strategy in which each array operation is decomposed into basic components called *factors*. Each factor describes an elementary computation or data transfer operation, and subsequent analyses manipulate and *join* these factors as optimizations. Though more basic than ZPL array operations, the

(a) ZPL source.

```
minvel := min<<Vel;



maxvel := max<<Vel;
```

(b) Naive code generation.

```
t = DBL_MAX;
for (i=mylow; i<myhi; i++)
    t = min(t,Vel[i]);
Glob_Reduce(<t,min>);
Broadcast(<t,minvel>);
t = -DBL_MAX;
for (i=mylow; i<myhi; i++)
    t = max(t,Vel[i]);
Glob_Reduce(<t,max>);
Broadcast(<t,maxvel>);
```

(c) Optimized code generation.

```
t1 = DBL_MAX;
for (i=mylow; i<myhi; i++)
    t1 = min(t,Vel[i]);


t2 = -DBL_MAX;
for (i=mylow; i<myhi; i++)
    t2 = max(t2,Vel[i]);
Glob_Reduce(<t1,min>,<t2,max>);
Broadcast(<t1, minvel>,<t2, maxvel>);
```

Fig. 2. Combining the communication portions of reductions. Notice the reduce operators in the ZPL source (a): min<< and max<<.

factors preserve the source code's high-level semantics. In contrast to this high level approach the IBM HPF compiler may lose semantic information because it *scalarizes* Fortran 90 array structures early in the compilation process [12].

The ZPL compiler thus employs standard compilation concepts and techniques, but extends them to exploit the language's abstractions and to treat arrays atomically. Our presentation of the ZPL compiler will assume an understanding of scalar compilation and concentrate only on areas of difference. These include the following.

- Internal representations, particularly the AST
- Run-time assumptions about the virtual machine
- Phases of compilation
- The factor-join technique and its resulting optimizations
  - loop fusion and array contraction
  - redundant communication removal
  - communication pipelining and combining

This paper presents the first description of the ZPL compiler's internal workings, and as such it concentrates on compilation strategy rather than performance. Previous work has shown that the generated code's performance is comparable with C using explicit message-passing [21] and is generally superior to the HPF compilers with which it has been compared [19, 22]. ZPL has also been successfully used for scientific and engineering applications [8, 18, 24], and its compiler is available on the Web.[2]

The remainder of this paper is structured as follows. Section 2 briefly introduces basic ZPL language concepts—more complete descriptions are available elsewhere [27, 20]. Section 3 describes runtime assumptions that are used in the compilation process. The compilation process itself is described in Sect. 4, with an emphasis on its structure and use of the factor-join strategy. Section 5 discusses the details of joining, and the final two sections present related work and conclusions.

2 ZPL Language Summary

ZPL is an implicitly parallel array language designed for scientific computations [27]. It is an imperative language, supporting standard data types (integer, float, char, etc.), standard operators (+, -, $*$, etc.), C-like assignment operators (+=, $*$=, etc.), procedures with by-value and by-reference parameters, recursion, a standard set of control constructs (if, for, while, etc.), and C-like I/O.

In addition, ZPL provides a number of abstractions and operations designed to simplify programming while promoting efficiency. *Regions* are a fundamental concept, implicitly specifying the parallelism in a ZPL program. A region is a set of indices and can be declared as follows. (Any text to the right of -- is a comment.)

[2] URL: http://www.cs.washington.edu/research/projects/zpl/

region R = [1..m, 1..n]; -- Declare R={(1,1), (1,2),..., (m,n)}

Regions are used to declare arrays as follows.

var A: [R] float; -- A is an m × n array of floats

Region specifiers prefix statements to define the extent of array operations. A statement whose arrays are of rank r requires a region specifier of rank r, and the array indices for which the statement is executed are the indices in the region specifier. For example, the following statement assigns the value 1 to the elements of A for the indices R = {(1,1), (1,2),..., (m,n)}.

[R] A := 1;

Dynamically scoped region specifiers allow a procedure either to supply region specifiers explicitly in its body or to inherit them from the call site. Thus, procedures can be written in a region-independent fashion and execute over different regions with each call.

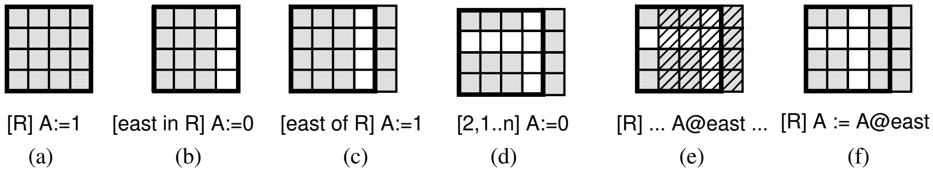


Fig. 3. Sequence of region usage examples. Grey boxes represent the value 1 and white represents 0. The hashed area in (e) represents the elements referred to by A@east.

Regions can be expressed and manipulated in a variety of ways. Figure 3 shows a sequence of operations which use A, R and n as defined above, and east = [0,1]. Array and region names are capitalized, while scalar variables are not. In Fig. 3(a), an array assignment, as just described, sets the R region of A, to 1. Next, an assignment over the region east in R sets the column *in*side the east border to 0. In 3(c), the region east of R causes the implicit allocation of a new column adjacent to, but outside of, the east border and sets it to 1. *Dynamic regions* bind their indices at runtime. The *dynamic region* [2,1..n] in 3(d) specifies that the first n elements of the second row be set to 0. The @-operator translates the specified region, in this case R, by adding the *direction* to all indices in the region, so Fig. 3(e) uses diagonal hashing to indicate the values referenced by A@east. In 3(f), the hashed portion of the array is assigned into A, shifting the array. Notice that the values in the east of R region are unchanged because they are outside of the applied region R.

ZPL supports a full set of reduce and scan (parallel prefix) operators. For example, the following statement *reduces* A to the value of its largest element using the max reduction operator (max<<) and assigns the value to the scalar biggest.

```
biggest := max<<A;    -- Find largest element
```

Other reduction operations include min<<, +<<, ∗<<, and &<<. Scan is similar to reduce, but it produces an array of the same shape and size as its operand, and each element contains the result of the reduction over all lower indexed elements (in row-major order).

```
Biggest_so_far := max||A;   -- Scan finding progressively larger items
```

ZPL also has partial reduces and scans that apply an operator over a subset of the array's dimensions (see Fig. 4). The dimensions in brackets indicate the subset of dimensions to scan across.

1 1 1	1 2 3	1 1 1	1 2 3
1 1 1	1 2 3	2 2 2	4 5 6
1 1 1	1 2 3	3 3 3	7 8 9
A	+||[2]A	+||[1]A	+||A

Fig. 4. Partial scan operation examples.

The concepts introduced up to this point are sufficient for understanding the sample scientific computation shown in Fig. 5. This code takes as input a vector containing the sampled coordinates of an object at various times (SampleT, SampleXPos, SampleYPos). It assumes the object was at the origin at time 0 (lines 18-22) and computes the approximate velocity of the object for each sampled interval (lines 23-28). It then applies reduction operators to determine the object's minimum and maximum velocities (lines 29-30). This program will serve as a running example throughout the paper. A complete listing is provided in Appendix A.

In addition to the above operations, ZPL contains a number of expressive abstractions for array manipulation. Due to space limitations, we only give a brief survey below, but complete information is available elsewhere [27, 20].

– *Shattered control flow* – ZPL has sequential control flow as long as control statements involve only scalars (e.g., if (scalar=1) then...). Control flow can also be specified using arrays, (e.g., [R] if (Array=1) then...), so that each index in the region is given a concurrently-executing thread of control.
– *Flooding* – Arrays can be declared to be *floodable*, causing certain dimensions to be replicated for all indices (e.g., var F:[1..m,∗] float;). The flood operator (>>[R]) can be used to assign rows or columns (indicated by region R) of an array to a floodable array. For example:

```
[1..m,*] F := >>[1..m,4]A;  -- Flood F with column 4 of A
```

```
. . .
direction prev = [-1];
. . .
region R = [1..samplecount];
var SampleT, SampleXPos, SampleYPos : [R] double;
    DeltaT, DeltaXPos, DeltaYPos    : [R] double;
    XVel,YVel                       : [R] double;
    Vel                             : [R] double;
procedure VelocityStats();
          var minvel,maxvel : double;
          . . .
[R]       begin
             . . .
[prev of R]  begin
                SampleT := 0.0;
                SampleXPos := 0.0;
                SampleYPos := 0.0;
             end;
             DeltaT := SampleT - SampleT@prev;
             DeltaXPos := SampleXPos - SampleXPos@prev;
             DeltaYPos := SampleYPos - SampleYPos@prev;
             XVel := DeltaXPos/DeltaT;
             YVel := DeltaYPos/DeltaT;
             Vel := sqrt(XVel*XVel + YVel*YVel);
             minvel := min<<Vel;
             maxvel := max<<Vel;
             . . .
          end;
```

Fig. 5. Excerpt from running example in Appendix A. This ZPL code computes approximate minimum and maximum velocities of a particle from a vector of sampled positions and times.

Since F is a flood array, it has no specific number of columns, and only a single copy of its defining values is stored at each processor. This provides a highly efficient way to refer to substructures of an array. For example, after column 4 of A has been flooded into F (above), the statement [R] A:=A∗F has the effect of multiplying each column of A by column 4.

- *Reflect/Wrap* – Operations are provided to simplify the computation of boundary values. When invoked in the context of an of or in region specifier, reflect and wrap cause the array's values in that region to be filled with those mirrored across the border (reflect) or from the opposite side of the array (wrap).
- *Scalars/Arrays/Indexed Arrays* – Scalars are replicated and redundantly computed. ZPL has two kinds of arrays: parallel arrays (also referred to simply as "arrays") for which indexing is not needed, and indexed arrays for which indexing is required. Parallel arrays are distributed across all processors, while indexed arrays are replicated in the same manner as scalars. Indexed arrays are commonly used as elements of parallel arrays.

This concludes our introduction to ZPL. We note that the language's operators, though very regular and structured, can be combined in non-trivial ways to implement many scientific applications. The language is not ideally suited for certain applications, particularly highly irregular codes. These are handled by ZPL's more general parent language, *Advanced ZPL* [26].

3 The ZPL Runtime

Before describing the ZPL compiler, we state a few assumptions about ZPL's runtime environment. In ZPL, the region is the basis for a program's implied parallelism. In the current implementation, the union of all regions' index sets is block distributed across a two dimensional processor mesh. Each array is allocated based on this block distribution, so all array elements with the same indices are allocated to the same processor.

This assumption leads to the trivial identification of communication—both for the compiler and the user. For example, line 28 of the running example (Vel := sqrt(XVel*XVel + YVel*YVel);) can be executed in parallel because corresponding elements of arrays Vel, XVel, and YVel are known to reside on the same processor. The shifted array reference in Line 23 (DeltaT := SampleT - SampleT@prev;) will require point-to-point communication to transfer non-local values of SampleT to adjacent processors. As a final example, line 29 (minvel := min<<Vel;) computes the minimum-reduction using collective communication involving all processors. This identification of necessary communication is crucial to the compiler's factor-join scheme, as will be seen in Sect. 4.

Although arbitrary alignment is not supported, certain optimizations, such as aligning only interacting arrays, are straightforward extensions. However, in the common case, a single global distribution scheme has proven very effective. The use of a two-dimensional block distribution of higher-dimensional regions was a decision of convenience that results in effective compilation for the common case. Higher-dimensional and alternative (e.g., cyclic, block-cyclic) distributions are a relatively straightforward extension to the existing compiler.

4 Compiler Overview

This section describes how the ZPL compiler transforms ZPL source code into a loosely synchronous SPMD C program that can then be compiled and run on any target machine. The bulk of the work is in compiling array operations into an efficient distributed scalar implementation. Since source-level scalar operations are replicated on each processor, their compilation is straightforward and will receive little attention here.

The ZPL compiler first parses the ZPL source into an abstract syntax tree (AST). The compiler preserves the source program's high-level array operations, rather than *scalarizing* them, to allow the compiler to perform optimizations at the array level. The AST is not transformed into scalar code until the generation of the ANSI C output. Additional AST nodes are introduced during the compilation process, for example to explicitly represent data transfer that is implicitly specified by the source program.

After parsing, the compiler *normalizes* the AST to produce a more uniform AST and to eliminate complex interactions between the different types of array operations. Normalization breaks heterogeneous array statements (i.e., statements containing different varieties of array operations) into a number of simpler array statements by inserting temporary scalars or arrays.

The compiler then performs optimizations using the factor-join strategy. Each normalized statement is decomposed into factors, where each factor represents an elementary array operation involving either local computation (*C-factors*) or interprocessor data transfer (*T-factors*). Because each factor represents a particular communication or computation structure, factors of the same type can always be joined. The joining of the various types of factors is discussed in Sect. 5. Figure 6 summarizes the factorization of the different types of ZPL array statements.

1	S_{array}	$\rightarrow T^*_{pp} \cdot C$
2	S_{wrap}	$\rightarrow T_{pp}$
3	$S_{reflect}$	$\rightarrow T_{pp}$
4	S_{reduce}	$\rightarrow C \cdot T_{gr} \cdot T_{bc}$
5	S_{scan}	$\rightarrow C \cdot T_{gs} \cdot (C \cdot T_{gs})^* \cdot (C \cdot T_{bc} \cdot C)^* \cdot C$
6	S_{flood}	$\rightarrow T_{bc}$

Key
C : Computation
T : Transfer
pp : point-to-point
bc : broadcast
gr : global reduce
gs : global scan
* : zero or more

Fig. 6. Rules for factoring the different types of array statements.

As an example of factorization, the compiler classifies line 28 (Vel:=sqrt(XVel*XVel+YVel*YVel);) of the running example (Fig. 5) as an element-wise array statement (S_{array}) and factors it using rule 1 (Fig. 6). This statement requires no data transfer because no communication-inducing operators are used. Thus, the compiler expands the statement into a single C-factor and no T-factors.

The correspondence between C-factors and local computation simplifies subsequent analysis. Each C-factor is represented by a *multi-loop* (or *m-loop*) node that encapsulates all information needed to generate object code, including the region over which the statement is executed and the code that forms the loop body. The AST node that is generated for this example is shown in Fig. 7(a).

As another example, consider line 23 (Delta:=SampleT-SampleT@prev;) of the running example. This statement is also classified as an array statement but requires communication because it uses the @-operator, so a multi-part T-factor representing point-to-point communication (T_{pp}) is inserted prior to its C-factor. This T-factor is represented in the AST using Send and Receive nodes[3] that

[3] The ZPL compiler actually uses the IRONMAN communication interface which is more hardware independent than a send/receive interface [7]. By using machine-dependent libraries and an unassuming interface, IRONMAN allows the same ZPL object code to exploit each machine's customized interprocessor communication features. This document uses send/receive for simplicity.

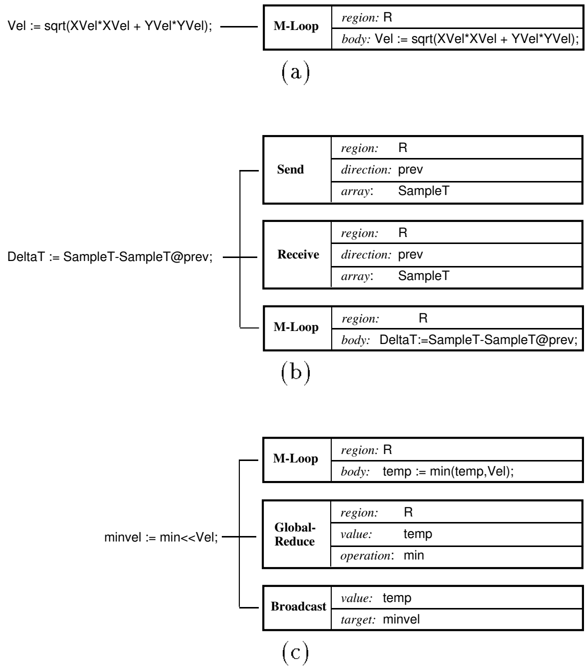


Fig. 7. ZPL source code and the corresponding factored AST. Note that the various node properties (e.g., *region*, *direction*, *body*) are actually pointers to symbol table entries or other parts of the AST.

describe the region, array, and direction of data transfer (Fig. 7(b)).

Some operators are translated into multiple factors. For example, the reduction in line 29 (minvel:=min<<Vel;) of the running example is factored (by rule 4, Fig. 6) into a C-factor that computes the local result for each processor and two T-factors: one to combine the local results into a global result (T_{gr}) and a second to broadcast the global result to all processors (T_{bc}). AST nodes are inserted for each of these factors, as shown in Fig. 7(c).

5 Joining Factors

This section describes how the ZPL compiler performs optimizations by manipulating C-factors and T-factors. The optimization process is simplified because only a small number of C- and T-factors are needed to represent any array operation, and because inter-statement optimizations can take place without having to consider how the many different types of array statements may interact.

5.1 Local Computation: M-Loops

As the only type of C-factor (i.e., the only way to iterate over arrays), m-loops represent the local portions of any array statement, including element-wise assignment, reductions, scans, etc. Thus, optimizing this single factor can yield substantial performance improvement. In the final compilation step, an m-loop is translated into a loop nest that executes on each processor and iterates over that processor's portion of the applicable region.

When a loop nest is generated from an m-loop, the compiler determines the nest depth, determined by the rank of the region, and iteration direction of each generated loop, as constrained by the body of the m-loop. These constraints arise from data dependences and semantic restrictions. Specifically, m-loops introduced by array statements are initially only semantically constrained, i.e., the right-hand side expression is evaulated before the left-hand side. (It is precisely this property that allows these operations to be directly parallelized.) M-loops introduced by reduce and scan operators induce a pseudo dependence due to the accumulation via an associative operator. The joining of factors introduces additional constraints, because additional dependences must be preserved.

Joining M-loops. The joining of m-loop factors has the effect of fusing loops in the object C code. Determining whether two m-loops may be legally joined is similar to the data dependence analysis required to fuse two loop nests [28]: (1) both m-loops must iterate over the same region, and (2) for the joined m-loop there must exist a loop nest that preserves the unjoined data dependences and respects semantic restrictions. This join transformation differs from traditional loop fusion in that the structure of the candidate loop nests is not fixed when the joining decision is made. There are a number of benefits to joining m-loops. Some are traditional, e.g., improved cache locality [6] and reduced loop overhead, and others are unique to the array language context, e.g., joining enables *contraction* of an array to a scalar when the array's definition only reaches uses in the same iteration.

Array Contraction. Array contraction is a well-known technique for scalar languages [28], but it is more important for array languages because the programmer has no control over the structure of the compiler-generated loops. This leads to a potential performance problem since the programmer cannot cache an array value in a *scalar* for later use in the same iteration of a loop, a common technique employed in scalar languages. Instead, the array language programmer must use whole arrays as temporaries (e.g., array Vel in the running example), which waste memory, induce contention in the data cache, and ultimately slow execution of the program. For the programmer, the only alternative to these intermediate arrays is to introduce redundant computation.

Consider the code fragment in Fig. 8(a). Figure 8(b) shows the naive code that is generated when factors are not joined. Notice that arrays XVel, YVel and Vel are used to cache computed values. If the definitions and uses of these variables can be joined into a single m-loop, then scalars can hold these values (Fig. 8(c)), as they are not live outside of the iteration. Since m-loops induced by reductions are no different from m-loops induced by element-wise assignment

```
XVel := DeltaXPos/DeltaT;
YVel := DeltaYPos/DeltaT;
Vel := sqrt(XVel*XVel + YVel*YVel);
minvel := min<<Vel;
maxvel := max<<Vel;
```

(a) ZPL source.

```
for (i=mylow; i<myhi; i++)
    XVel[i] = DeltaXPos[i] / DeltaT[i];
for (i=mylow; i<myhi; i++)
    YVel[i] = DeltaYPos[i] / DeltaT[i];
for (i=mylow; i<myhi; i++)
    Vel[i] = sqrt(XVel[i]*XVel[i]+YVel[i]*YVel[i]);
temp = DBL_MAX;
for (i=mylow; i<myhi; i++)
    temp = min(temp, Vel[i]);
< . . . data transfer code here . . .>
< . . . assignment to minvel . . .>
temp = -DBL_MAX;
for (i=mylow; i<myhi; i++)
    temp = max(temp, Vel[i]);
< . . . data transfer code here . . . >
< . . . assignment to maxvel . . . >
```

(b) Naive loop generation.

```
for (i=mylow; i<myhi; i++) {
    xvel = DeltaXPos[i] / DeltaT[i];
    yvel = DeltaYPos[i] / DeltaT[i];
    Vel[i] = sqrt(xvel*xvel+yvel*yvel);
}
temp = DBL_MAX;
for (i=mylow; i<myhi; i++)
    temp = min(temp, Vel[i]);
< . . . data transfer . . . >
< . . . assignment to minvel . . . >
temp = -DBL_MAX;
for (i=mylow; i<myhi; i++)
    temp = max(temp, Vel[i]);
< . . . data transfer code here . . . >
< . . . assignment to maxvel . . . >
```

(c) Partially optimized loop generation.

```
temp1 = DBL_MAX;
temp2 = -DBL_MAX;
for (i=mylow; i<myhi; i++) {
    xvel = DeltaXPos[i] / DeltaT[i];
    yvel = DeltaYPos[i] / DeltaT[i];
    vel = sqrt(xvel*xvel+yvel*yvel);
    temp1 = min(temp1, vel);
    temp2 = max(temp2, vel);
}
< . . . data transfer . . . >
< . . . assignment to minvel . . . >
< . . . assignment to maxvel . . . >
```

(d) Fully optimized loop generation.

Fig. 8. Effects of joining and contraction on an excerpt from the running example. A number of the array references in the bold statements become scalar references. Arrays DeltaXPos, DeltaYPos, DeltaT in the running example may be similarly contracted.

statements, array Vel may also be contracted as in Fig. 8(d). In fact, for the running example, all but the Sample arrays are eliminated by this array contraction.

We cannot independently join m-loops and perform contraction if we expect to maximize contraction. The compiler therefore joins m-loops with the goal of enabling maximal array contraction. Using a heuristic ordering of the candidate arrays, all m-loops containing a candidate array are joined if the joining enables contraction of the candidate array. This simple greedy strategy produces very high quality code [17].

Our approach not only contracts arrays that a clever scalar language programmer would, it often succeeds in non-obvious cases. There are cases when a group of m-loops may not legally be joined because they over-constrain the resulting loop nest, but simple transformations eliminate the constraints and enable the joining and contraction. The trick of eliminating the constraint is often suitably awkward that programmers are unwilling or unable to do this by hand.

Indexed Arrays. While the use of m-loops nicely encapsulates the local computation that results from array statements, the distinction between m-loops and source-level loops can produce a runtime performance penalty when these two types of loops interact. Consider the ZPL code fragment in Fig. 9(a), which uses a parallel array of indexed arrays. An m-loop will be used to iterate over the parallel array, while a source-level loop iterates over each element of the indexed array (Fig. 9(b)). The problem is that the generated code will exhibit poor cache behavior unless the source-level loop is moved inside the compiler generated loop, as in Fig. 9(c). A source-level loop is a candidate for this transformation when it iterates over an indexed array that is an element of a parallel array. The transformation is performed when all m-loops that contain the involved array may be joined.

```
region R = [1..n];
var A : [R] array [1..m] of integer;
  . . .
[R] for i := 1 to m do
      A[i] := 1;
    end;
```

(a)

```
for (i=1; i<=m; i++)
    for (j=mylow; j<myhi; j++)
        A[j][i] = 1;
```

(b)

```
for (j=mylow; j<myhi; j++)
    for (i=1; i<=m; i++)
        A[j][i] = 1;
```

(c)

Fig. 9. The interaction of source-level loops and compiler generated loops. The ZPL source (a) will naively be compiled into the code in (b). Bringing the source-level loop inside the compiler generated loop (c) will improve cache locality.

5.2 Data Transfer

There are several types of T-factors. Point-to-point T-factors are implied by the `wrap` statement, the `reflect` statement, and the `@` operator, while the remaining T-factors represent collective communication in operations such as scan, reduce and flood. A point-to-point T-factor is multi-part (send and receive), while a single T-factor can represent each variety of collective communication. This section discusses the optimization of data transfer through the manipulation of T-factors.

Data transfer can be optimized in three ways. First, redundant T-factors may be removed. A T-factor is redundant if and only if the data transfer performed by the T-factor is preceded by a T-factor that satisfies the requested data transfer. Next, T-factors involving the same source and destination processors may be combined. Finally, the components of multi-part T-factors may be pushed apart to pipeline and overlap data transfer and computation. Recall that communication in array languages is naturally vectorized, so the compiler does not perform explicit message vectorization. The removal of redundant T-factors and the combining of T-factors are exact instances of the join operation, while the pipelining of T-factors enables additional joins to occur. For convenience, we will refer to each optimization in isolation, though the actual implementation considers all three optimizations simultaneously.

Point-to-Point Communication. Point-to-point T-factors can be optimized by all three techniques. These optimizations require information about the uses and modifications of the array variables being transferred. This information is maintained in the form of def/use-sets on a *per statement* basis. Since arrays are never indexed, the compiler treats them atomically, much like scalars. Unlike languages such as Fortran 77, no index functions or loop bounds information need be examined. Rather, the region and direction indicate the slice of an array to be transferred. The compiler could perform symbolic analysis on the regions to obtain more precise def/use-sets, but this is generally not necessary as most data parallel computations use a small set of regions.

Figure 10 shows a sample code fragment that requires data transfer, along with unoptimized and optimized code generated for it. For simplicity, we again assume that the compiler generates message passing code (send and receive calls). To generate the unoptimized code in Fig. 10(b), the compiler need only generate a library call for each @ induced T-factor. The code generated in Fig. 10(c)-(e) illustrate the three data transfer optimizations performed by the compiler, which are now discussed in turn.

Removing redundant T-factors. If the T-factor due to a statement (Fig. 10, statement 4) is preceded by a T-factor that has already satisfied that data transfer (statement 3) and there are no intervening modifications to the transferred data, the T-factor for the original statement is redundant and can be eliminated (Fig. 10(c)).

Combining T-factors. If several T-factors perform data transfer on different variables in the same direction (statement 3), these T-factors my be combined (see Fig. 10(d)). T-factors from the same (as in this example) or different statements may be combined in this way.

Pipelining T-factors. The send portion of a T-factor may be pushed up to the last statement that defines the variable involved in the data transfer. This overlaps communication and computation. Statement 2 is the most recent modification of A or C before the use of A in statement 3. Therefore the pipelined T-factor can be started immediately after statement 2 (see Fig. 10(e)).

Appendix B (lines 20–23) shows the result of data transfer optimizations in the running example. Notice that the initialization of the identity elements for the reductions have been moved between the send and receive due to local joining operations. Though this is a small amount of computation, in general the separation of the send and receive may be large.

```
1  C := D;
2  A := B;

   . . .

3  E := A@prev+C@prev;


4  F := A@prev;
```

(a) Original ZPL code.

```
<. . . assign C . . .>
<. . . assign A . . .>
      . . .
Send(R,<A,prev>);
Recv(R,<A,prev>);
Send(R,<C,prev>);
Recv(R,<C,prev>);
<. . . assign E . . .>
Send(R,<A,prev>);
Recv(R,<A,prev>);
<. . . assign F . . .>
```

(b) Unoptimized code.

```
<. . . assign C . . .>
<. . . assign A . . .>
      . . .
Send(R,<A,prev>);
Recv(R,<A,prev>);
Send(R,<C,prev>);
Recv(R,<C,prev>);
<. . . assign E . . .>


<. . . assign F . . .>
```

(c) Redundant T-factors removed.

```
<. . . assign C . . .>
<. . . assign A . . .>
      . . .


Send(R,<A,prev>,<C,prev>);
Recv(R,<A,prev>,<C,prev>);
<. . . assign E . . .>


<. . . assign F . . .>
```

(d) Combined T-factors.

```
<. . . assign C . . .>
<. . . assign A . . .>
Send(R,<A,prev>,<C,prev>);

      . . .


Recv(R,<A,prev>,<C,prev>);
<. . . assign E . . .>


<. . . assign F . . .>
```

(e) Pipelined T-factors.

Fig. 10. Example transformations on T-factors.

Collective Communication. Just as in the point-to-point case, collective communication T-factors of the same type are optimized by one or more of the above techniques. Any redundant T-factors may be removed, and all types of collective communication T-factors may be combined with other T-factors of the same type (see Fig. 2). The key to the optimization is that the communication patterns (each factor represents a different pattern) are exposed to the compiler. Library approaches cannot be optimized in this way, for the compiler is unaware of the libraries' contents.

Despite this fixed collective communication interface, ZPL does not lose the advantages of library support. The combined communication compiles to procedure calls in the ZPL runtime library. These procedures are optimized for the particular platform's strengths [7]. For example, on the SP2, the procedure calls are mapped to MPI library routines, while our T3D implementation uses the native SHMEM library routines [2]. In this way, a single copy of optimized code exploits the strengths of all target platform.

Though we have introduced collective communication operations as each producing a single T-factor, we often use multi-part T-factors for these operations. These multi-part factors allow for better communication hiding when implementing non-hardware supported operations (such as column broadcasts or reduces).

6 Related Work

ZPL was designed from first principles to execute efficiently across MIMD computers. There exist a number of compilation efforts that are similar in nature to that of ZPL. Several are summarized below.

The APL language supports the atomic manipulation of and computation on whole arrays [15]. APL was not designed with parallelism in mind, thus it encourages the use of locality insensitive operations. Greenlaw and Snyder demonstrate that the second most common data movement operation in APL (an array subscripted array) is very expensive on parallel machines [10]. Budd describes an APL compiler that decomposes array operations into vector operations for execution on a vector processor [5]. This fine grained approach does not extend to distributed memory or MIMD machines. Ju *et al.* describe a classification and fusion scheme for array language primitives (in APL, Fortran 90, etc.) [16]. Their fusion concept differs from our join transformation in that it only strives to eliminate intermediate storage. In addition, they assume a shared memory model and thus do not consider explicit communication. We optimize computation and communication separately.

NESL is a data-parallel programming language that emphasizes nested parallelism [4]. NESL source code is compiled to an intermediate vector-based code, called Vcode, which is either interpreted or compiled [3]. The Vcode intermediate form is well suited for vector and low-latency shared memory machines but not for distributed memory machines. The primary MIMD compilation effort is in increasing the granularity of parallelism and reducing synchronization overhead.

C* and its descendant Dataparallel C are derivatives of C with support for data parallel programming [25]. The Dataparallel C *domain* and the ZPL *region* both serve as bases for parallelism; they are used to define distributed arrays and to distribute computation. Despite this similarity, the techniques for compiling these languages greatly differ. In particular, the primary Dataparallel C compilation effort is in overcoming inefficiencies due to the sequential nature of the parent language (C) and the SIMD nature of the language itself [13]. As an example of the former, a Dataparallel C program may contain arbitrary C code, which resists static analysis due to pointer arithmetic and weak typing.

High Performance Fortran (HPF) is a language that requires the user specification of parallelism, distribution and alignment via directives in sequential Fortran 77 and Fortran 90 programs [14]. The primary compilation effort is in overcoming the sequential nature of the parent language. Arrays are manipulated at the element level, thus optimizations must be performed to vectorize communication and hoist it from inner loops. HPF and ZPL are similar in the types of parallel operations that they support, though ZPL makes clear to the programmer the execution cost of each operation [22].

Considerable research has been devoted to automatically parallelizing Fortran 77 programs [1, 23, 11]. In contrast to the ZPL approach in which the language was designed to facilitate the recognition and exploitation of parallelism, the primary effort for automatically parallelizing compilers is in recognizing, exposing and efficiently exploiting the parallelism hidden in a sequential program.

Furthermore, optimal sequential and parallel solutions to the same problem often require different algorithms, thus it seems unlikely that a compiler will be able (in general) to transform one to the other.

7 Conclusions

We have argued that when compiling for parallelism, array languages present compilers with new opportunities and challenges for optimization. The array operations of these languages make some standard optimizations (such as message vectorization) disappear, make new opportunities (such as combining reduction operations) appear, and add importance to other optimizations (such as array contraction).

We have described the factor-join approach to compiling array languages and shown how it is used to compile ZPL programs. This approach first decomposes array language constructs into a series of factors, and then joins these factors to perform various optimizations. Each factor represents a unique communication or computation structure, and only a small number of different factors are needed to describe ZPL programs. These factors cleanly separate the treatment of communication and computation. For example, C-factors represent the purely computational aspects of all operations (e.g., element-wise array assignments, reductions, scans, etc.), so optimizing C-factors simultaneously optimizes all inner loops that the compiler generates for array constructs. This factorization also simplifies the movement and joining of the data-transfer factors (T-factors). This approach provides a framework for optimizations that includes redundant communication elimination, message combining, and communication pipelining; and the use of factors abstracts common features of different optimizations. For example, the combining of collective communication operations and the combining of point-to-point communication use the same algorithm applied to different T-factors.

References

1. Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Amy W. Lim. An overview of a compiler for scalable parallel machines. In *Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.
2. Ray Barriuso and Allan Knies. SHMEM user's guide for C. Technical report, Cray Research Inc., June 1994.
3. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.
4. Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, 1993.
5. Timothy A. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

6. Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improved data locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994. San Jose, CA.
7. Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. IRONMAN: An architecture indepedent communication interface for parallel computers. *submitted for publication*, 1996.
8. Marios D. Dikaiakos, Calvin Lin, Daphne Manoussaki, and Diana E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *Ninth International Conference on Supercomputing*, 1995.
9. Michael Gerndt. Updating distributed variables in local computations. *Concurrency–Practice and Experience*, 2(3):171–193, September 1990.
10. R. Greenlaw and L. Snyder. Achieving speedups for APL on an SIMD distributed memory machine. *International Journal of Parallel Programming*, 19(2):111–127, April 1990.
11. Manish Gupta and Prithviraj Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *International Conference on Supercomputing*, July 1993.
12. Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, December 1995.
13. Philip J. Hatcher, Anthony J. Lapadula, Robert R. Jones, Michael J. Quinn, and Ray J. Anderson. A production-quality C* compiler for hypercube multicomputers. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.
14. High Performance Fortran Forum. *High Performance Fortran Specification Version 1.1.* November 1994.
15. Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.
16. Dz-Ching R. Ju, Chaun-Lin Wu, and Paul Carini. The classification, fusion, and parallelization of array language primitives. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1113–1120, October 1994.
17. E Christopher Lewis and Calvin Lin. Array contraction in array languages. Technical report, University of Washington, Department of Computer Science and Engineering, 1996. Forthcoming.
18. E Christopher Lewis, Calvin Lin, Lawrence Snyder, and George Turkiyyah. A portable parallel n-body solver. In D. Bailey, P. Bjorstad, J. Gilbert, M. Mascagni, R. Schreiber, H. Simon, V. Torczon, and L. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 331–336. SIAM, 1995.
19. C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report 95–11–05, Department of Computer Science and Engineering, University of Washington, 1994.
20. Calvin Lin. ZPL language reference manual. Technical Report 94–10–06, Department of Computer Science and Engineering, University of Washington, 1994.
21. Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, pages 361–375. Springer-Verlag, 1994.

22. Ton A. Ngo. *The Effectiveness of Two Data Parallel Languages, HPF and ZPL.* PhD thesis, University of Washington, Department of Computer Science, 1996. In preparation.
23. C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In *Workshop on Languages and Compilers for Parallel Computing*, pages 423–453, 1990.
24. George Wilkey Richardson. Evaluation of a parallel Chaos router simulator. Master's thesis, University of Arizona, Department of Electrical and Computer Engineering, 1995.
25. J.R. Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.
26. Lawrence Snyder. Foundations of practical aprallel programming languages. In Tony Hey and Jeanne Ferrante, editors, *Portability and performance for parallel processing*, pages 1–19, New York, 1994. Wiley.
27. Lawrence Snyder. *The ZPL Programmer's Guide*. May 1996.
28. Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.

A Sample ZPL Source Code

```
/* VelocityStats - compute approximate minimum and
   maximum velocity of particle from sample positions */

program VelocityStats;

direction prev = [-1];

config var samplecount : integer=10;
           datafile    : string="samples.dat";

region R = [1..samplecount];

var SampleT, SampleXPos, SampleYPos : [R] double;    -- samples of location (sorted by t)
    DeltaT, DeltaXPos, DeltaYPos : [R] double;       -- delta from one sample to next
    XVel,YVel : [R] double;                          -- X- and Y-components of velocity
    Vel : [R] double;                                -- velocity


procedure VelocityStats();
           var minvel,maxvel : double; -- min/max velocities
               infile        : file;

[R]        begin
               infile := open(datafile,"r");
               read(infile,SampleT,SampleXPos,SampleYPos);
               close(infile);

[prev of R]    begin
                   SampleT := 0.0;
                   SampleXPos := 0.0;
                   SampleYPos := 0.0;
               end;

               DeltaT := SampleT - SampleT@prev;
               DeltaXPos := SampleXPos - SampleXPos@prev;
               DeltaYPos := SampleYPos - SampleYPos@prev;

               XVel := DeltaXPos/DeltaT;
               YVel := DeltaYPos/DeltaT;

               Vel := sqrt(XVel*XVel + YVel*YVel);

               minvel := min<<Vel;
               maxvel := max<<Vel;

               writeln("Minimum velocity was: ", minvel);
               writeln("Maximum velocity was: ", maxvel);
           end;
```

Appeared in Proceedings of the Workshop on Languages and Compilers for Parallel Computing, 1996.

B Resulting Pseudo-C Code

```
integer samplecount = 10;
char * datafile = “samples.dat”;
region R, prev_of_R;
double *SampleT, *SampleXPos, *SampleYPos;
double deltat, deltaxpos, deltaypos;
double xvel, yvel, vel;

void VelocityStats(void) {
    double minvel, maxvel;
    FILE * infile;
    double temp1,temp2;

    infile = fopen(datafile,“r”);
    FScanParallelArray(infile,<R,SampleT>,<R,SampleXPos>,
                        <R,SampleYPos>);
    fclose(infile);

    for (i = prev_of_R.mylo; i <= prev_of_R.myhi; i++)
        SampleT[i] = 0.0;
    for (i = prev_of_R.mylo; i <= prev_of_R.myhi; i++)
        SampleXPos[i] = 0.0;
    for (i = prev_of_R.mylo; i <= prev_of_R.myhi; i++)
        SampleYPos[i] = 0.0;

    Send(R,<SampleT,prev>,<SampleXPos,prev>,
          <SampleYPos,prev>);
    temp1 = DBL_MAX;
    temp2 = -DBL_MAX;
    Receive(R,<SampleT,prev>,<SampleXPos,prev>,
             <SampleYPos,prev>);

    for (i = R.mylo; i <= R.myhi; i++) {
        deltat = SampleT[i] - SampleT[i-1];
        deltaxpos = SampleXPos[i] - SampleXPos[i-1];
        deltaypos = SampleYPos[i] - SampleYPos[i-1];
        xvel = deltaxpos/deltat;
        yvel = deltaypos/deltat;
        vel = sqrt(xvel*xvel + yvel*yvel);
        temp1 = min(temp1,vel);
        temp2 = max(temp2,vel);
    }
    Glob_Reduce(R,<temp1,min>,<temp2,max>);
    Broadcast(<temp1,minvel>,<temp2,maxvel>);
    printf(“Minimum velocity was: %f\n”,minvel);
    printf(“Maximum velocity was: %f\n”,maxvel);
}

void main(int argc,char * argv[]) {
    DistributeRegions(<R,1,m>,<prev_of_R,0,0>);
    AllocateArrays(<R,SampleT>,<R,SampleXPos>,
                    <R,SampleYPos>);
    VelocityStats();
}
```