The Case for High-Level Parallel Programming in ZPL

BRADFORD L. CHAMBERLAIN, SUNG-EUN CHOI,
E CHRISTOPHER LEWIS, LAWRENCE SNYDER, AND W. DERRICK WEATHERSBY
University of Washington
CALVIN LIN
The University of Texas at Austin

◆ ◆ ◆

Message-passing programs are efficient, but fall short on convenience and portability. ZPL is a high-level language that offers competitive performance and portability, as well as programming conveniences lacking in low-level approaches.

◆

In sequential programming, low-level assembly languages have long given way to languages such as C and Fortran. These higher-level languages offer programming conveniences such as procedures, structured control flow, and richer data types, while still providing good performance. By contrast, other sequential languages that provide even higher levels of abstraction than C and Fortran have not been as widely embraced because they do not perform as well.

The difference among these high-level languages is that the efficient ones map well to the underlying machine, while the inefficient languages are either too far removed from the hardware or have features that disable compile-time optimizations and introduce runtime overheads. An example of the former is Lisp's reliance on recursion and higher-order functions. Examples of the latter are Java's dynamically bound class hierarchies and garbage collection.

The situation in parallel computing is even more severe because performance is more critical. Users have opted for low-level approaches rather than accept the inefficiencies incurred by high-level languages. But high-level parallel languages are not inherently inefficient. Most simply map poorly to the underlying hardware or introduce excessive runtime overhead, or both. Thus, they cannot deliver speed with convenience and portability.

In parallel programming, the low-level approach is represented by message passing; the high-level analog to C and Fortran is ZPL. The analogy with sequential programming is not exact, however, because issues of concurrency and synchronization make message-passing programs even more difficult to write than sequential assembly-language programs. Also, unlike assembly-language programs, there are standards such as the Message Passing Interface (MPI)[1] that provide a degree of portability across parallel machines.

Even with these standards, however, message passing is inadequate for several reasons. First, message passing is too tedious and difficult to use. Second, it undermines portability by exposing the underlying machine to the programmer. Third, message passing embeds assumptions in the source code that obscure the program logic and make the program difficult to modify. Thus, high-level languages are essential.

This article describes the high-level approach embodied by the ZPL programming language. The key to success for high-level languages is to choose language features that simplify programming, accurately expose costs, and can be effectively compiled for different machines. The ZPL language meets these criteria. It runs on a variety of parallel and sequential computers, and provides programming conveniences not found in message pass-

1070-9924/98/$10.00 © 1998 IEEE

ing. Here, we describe the problems with message passing and describe how ZPL simplifies the task of programming for parallel computers—without sacrificing efficiency.

Limitations of message passing

Most parallel programs are written in a sequential language, such as C or Fortran, and use message passing to perform synchronization and communication. Because MPI is an industry standard, programs written with MPI can run on almost any parallel computer. However, even with a standard such as MPI, the low-level nature of message passing causes significant problems.

Low-level reasoning

Message passing forces the programmer to reason about concurrency and synchronization at a low level. Within iterative code, for example, programmers must keep messages from one iteration distinct from those of another. These details are both difficult to write correctly and difficult to optimize. Consider the task of implementing double-buffering to overlap communication with computation. To do this, the programmer must understand a program's data dependences to know which computations are independent of incoming data, where they can safely move the computations, and when to wait for a receive to complete. As we describe below, these are all tasks that a compiler can easily perform for well-designed high-level languages.

Performance portability

A basic tenet of software reuse is to separate interfaces from implementation so that implementations can change without affecting clients of interfaces. Unfortunately, most message-passing interfaces expose much of the implementation to programmers. They do this by providing many ways to transmit data that vary in their timing, synchronization, and completion. For example, some routines buffer the user's data while others receive the data directly into a user-specified location. Such implementation details should be hidden from the interface, but they are exposed to enable significantly improved performance in certain contexts.

By exposing the implementation, the interface indirectly exposes the underlying hardware and its idiosyncrasies. Because the performance of different communication forms varies from machine to machine, a program optimized for one machine might perform poorly on another.[2]

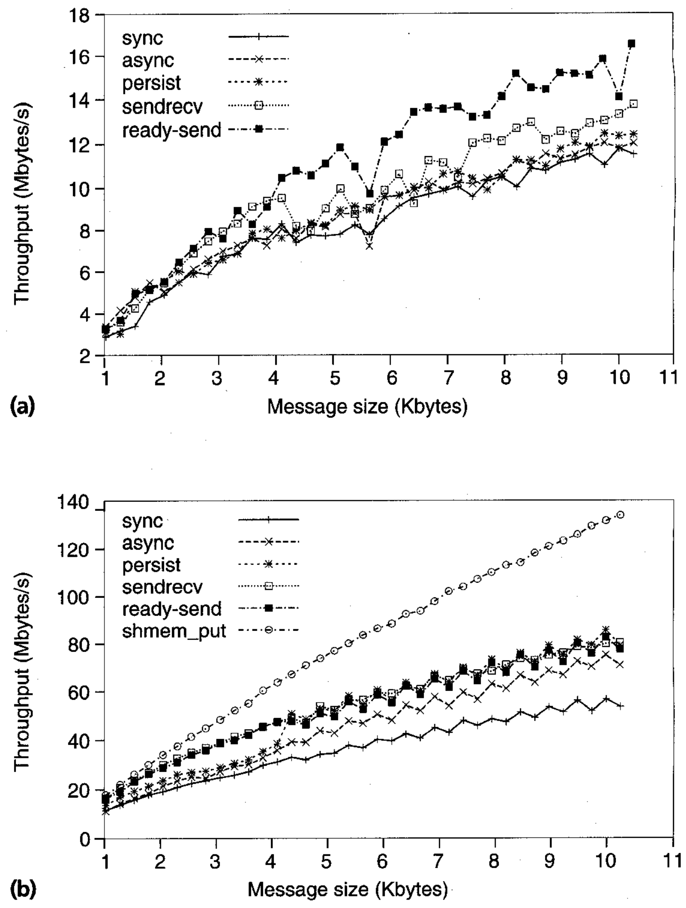


Figure 1. Comparing MPI communication routines. (a) On the IBM SP-2, the `ready-send` mechanism performs best; (b) On the Cray T3E, the `shmem_put` mechanism is the clear winner.

For example, Figure 1 shows the throughput of various MPI point-to-point communication routines on the IBM SP-2 and the Cray T3E. On the IBM, the `ready-send` mechanism is the clear winner for all message sizes; on the Cray, the `shmem_put` mechanism performs best. Consider trying to port to the T3E an MPI program that was originally optimized for the SP-2. While the original SP-2 program might use the `ready-send` mechanism, the tuned T3E program would prefer to use `MPI_Put` (once MPI 2.0 has been implemented). The required translation is an invasive process requiring significant changes, none of which improve the fundamental computational approach or algorithm. Thus, for good performance, message passing requires programmers to change the program as it moves from one machine to

```
program jacobi;

config var n         : integer    = 256;             -- Run-time constants
           delta     : float      = 0.0001;

region     R = [1..n, 1..n];                   -- Declarations

direction  north     = [-1, 0]; south = [1, 0];
           east      = [0, 1]; west = [0,-1];

procedure jacobi();                                  -- Entry point
var A, Temp    : [R] float;
    err        :  float;

             begin
    [R]               A := 0.0;                      -- Initialization
    [north of R]      A := 0.0;                      -- Set boundary conditions
     [east of R]      A := 0.0;
     [west of R]      A := 0.0;
    [south of R]      A := 1.0;
    [R]               repeat                         -- Main body
                        Temp       := (A@north+A@east+A@west+A@south) / 4;
                        err        := max<< fabs(A-Temp);
                        A          := Temp;
                      until err < delta;
              end;
```

Figure 2. The Jacobi iteration in ZPL.

another. The point is that while *some* binding of MPI is appropriate for each machine, *no* binding of MPI is appropriate for all machines.

Modification and maintenance

Message-passing programs are difficult to modify because they embed so many details and assumptions into the source code. The message-passing code is typically scattered throughout the source code and often obscures the basic logic of the algorithm, making the code difficult to understand and debug.[3] In addition, message passing forces the programmer to embed certain assumptions into the source code. For example, a programmer might embed a logical communication topology, such as a binary tree or a mesh. Although such information is often closely linked to the parallel algorithm, there are cases in which it makes sense to change the logical topology.

Long-term prospects

These three problems raise concerns about the future of message passing. First, as new communication mechanisms are introduced for new machines, existing message-passing programs will have to be manually rewritten to exploit their benefits. Even then, the resulting program is likely to be optimized for only a limited number of platforms. Second, imagine what message-passing codes will look like when hybrid machines, such as clusters of bus-based multiprocessors, become more common. In such environments, communication among clusters favors traditional message passing while communication within a cluster favors shared memory `Put` and `Get` operations. Programs that are highly tuned for such architectures might well require a mixture of message passing and `Put` and `Get` operations. Such code will be extremely tedious to optimize manually. If written in a low-level approach, such as MPI or PVM, they will likely be too convoluted to port to other platforms.

A brief introduction to ZPL

ZPL is an array language developed with the goals of portability, good performance, conciseness, and clarity. Whereas languages such as Fortran 77 provide array data types that can only be manipulated one element at a time, ZPL lets programmers operate on entire arrays. ZPL uses *regions* to refer to a collection of array indices. Region operators produce new regions in a structured manner. Together, regions and region operators give programmers a clean mechanism for referring to and operating on sets of array elements, replacing the explicit array indexing found in most languages.

Jacobi iteration in ZPL

As an example, consider the four-point Jacobi iteration on $n \times n$ array A in which each element of the array is to be replaced by the average of its four neighbors. The boundary values are taken to be 0 except at the southern edge, where the boundary values are 1.

Figure 2 shows a ZPL implementation. The configuration variables declared at the top of the program are defined at load time and remain constant thereafter. These variables configure a computation by defining program-specific quantities—such as problem size. Because parallelism is implicit in ZPL's semantics, the number of processors does not appear in the source code. Each configuration variable is given a default value—in the source code—that the programmer can override at the command line or by specifying a configuration file when the program is executed.

Line 6 of the Jacobi iteration shows that region R is declared to be a two dimensional index set containing the indices $\{(i,j) \mid i \in [1..n], j \in [1..n]\}$. This region can now be used in two ways. First, it can be used to declare array variables as shown on line 12. Here, arrays A and Temp are declared. (The ZPL compiler will allocate extra space to hold boundary elements for the array A. Details are provided elsewhere.)[4] Second, regions can specify a statement's domain of computation. For example, line 16 specifies that each element of A in the region R will be set to 0.

Directions are user-defined vectors that can be used in conjunction with region operators such as `of` and `@`. Lines 8 and 9 define four directions—`north`, `south`, `east` and `west`—which are vectors of size 2. Lines 16–20 show the code to initialize the values of A. After the elements of array A described by region R are set, the boundary values are set using four statements, each of which initializes one of the borders. Each of these borders is referred to by an `of` operator, which takes a direction and a region and refers to the adjacent region in the specified direction. For example, as Figure 3 shows, `[north of R]` refers to the region adjacent to R's north boundary, namely, the indices $\{(0,j) \mid j \in [1..n]\}$; thus, line 17 specifies that the elements of array A whose indices are in `[north of R]` will be assigned the value 0. The precise semantics of `of` and other region operators are available in the programmer's guide.[4]

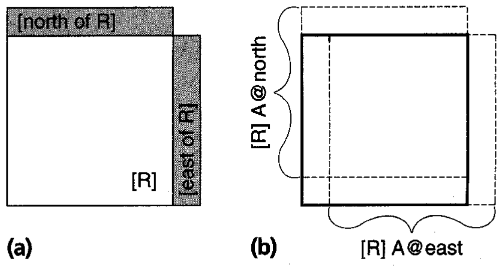


Figure 3. Illustration of (a) regions and (b) the @ operator.

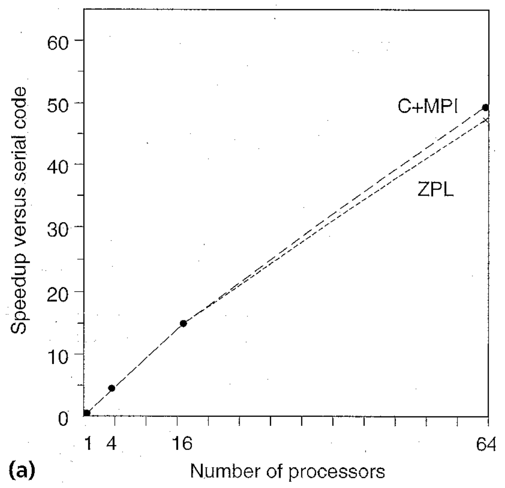
The iteration's main loop consists of a `repeat` statement (lines 21–25), whose region R applies to every statement in the loop body. In line 22, the @ operator is used to replace each element of A with the average of its four neighbors. The @ operator takes an array and a direction and refers to the array elements that are offset by the vector shown in Figure 3. Thus, `A@north` refers to the elements of A whose indices are $\{(i,j) \mid i \in [0..n-1], j \in [1..n]\}$. Line 23 computes the maximum change over all elements using a reduction operator (<<), which leaves the result in a scalar called `err`.

At this point, three of ZPL's features are noteworthy. First, the language has deterministic semantics. Each statement logically completes before the following one executes, and the entire right-hand side of an assignment statement is evaluated before it is assigned to the left-hand side. Second, the ZPL compiler implicitly generates all communication and synchronization. Third, data partitioning is also implicit, with all interacting arrays guaranteed to be aligned in the same manner.

The Jacobi example does not exercise ZPL's more sophisticated features, but it does illustrate the language's basic properties: high-level concepts such as regions support array manipulation; array operations are a natural, implicit expression of data parallelism; and the compiler handles the mechanical details of communication and synchronization.

ZPL and other languages

Several characteristics of ZPL are unique. First, ZPL is based on an underlying abstract machine,



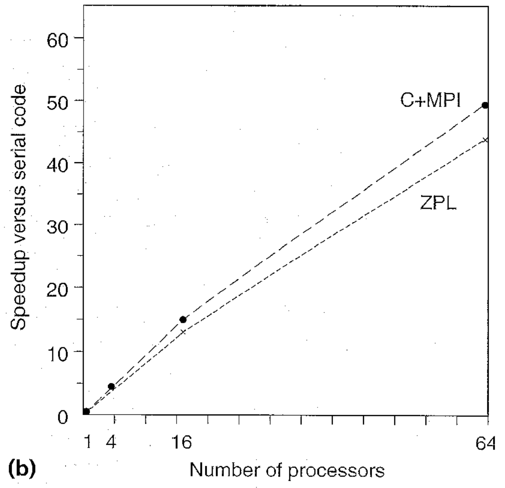


Figure 4. Comparisons of ZPL and C with MPI implementations of the Simple benchmark on (a) the Cray T3E and (b) the IBM SP-2.

the CTA,[5] which represents the essential features of all contemporary parallel computers. By including in ZPL only constructs that execute well on the CTA, ZPL programs should execute well on any parallel computer. Second, ZPL does not inherit baggage from existing sequential languages, which were not designed with parallelism—much less the CTA—in mind. Finally, unlike many parallel languages, ZPL does not hide performance costs from the user. For example, in languages such as High Performance Fortran (HPF), the statement `A(i,j) = B(i,j)` may or may not require communication, depending on how the two arrays are distributed relative to one another. Thus, small changes in data distribution directives can significantly affect performance, which makes it difficult to reason about a program's performance on any parallel machine. In HPF, costs can be similarly hidden at procedure calls, where dynamic redistribution of array operands might be necessary.

ZPL performance

The performance of ZPL programs has been extensively measured and compared against other approaches.[6–12] Here, we highlight some of these results.

Benchmark results

Figure 4 compares the performance of ZPL and C with MPI implementations of the Simple fluid-dynamics benchmark on the Cray T3E and the IBM SP-2. Of the many results, we chose these for two reasons. First, they show that high-level approaches can be as efficient as message passing. Second, Simple is a widely studied fluid-dynamics benchmark intended to be representative of scientific codes.[13]

In his 1997 doctoral dissertation,[7] Ton Ngo did an in-depth analysis of portability and performance of data-parallel languages and their compilers, including three HPF compilers and one ZPL compiler. The results examined performance of basic expressions, dense matrix multiplication, and a subset of the NAS parallel benchmarks suite on an IBM SP-2. Ngo not only measured programs, he analyzed the compilation process to explain why the compilers did what they did.

Two conclusions can be drawn from this in-depth study. First, ZPL was the most consistent in giving the best performance, while the performance of particular HPF codes often varied drastically depending on which compiler was used. Second, ZPL's absolute performance and scaling were good. These results confirm preliminary experiments from 1994. In these tests, an early version of ZPL was compared against an early HPF compiler on a set of eight small benchmark programs. ZPL outperformed HPF on six of the benchmarks,[8] indicating an overall performance advantage.

Application experience

Researchers have used ZPL to produce parallel programs that do not have parallel counterparts in other languages. Examples include a novel hierarchical *N*-body code,[10] two mathematical biology codes,[11] and a large (10,000 lines of ZPL) synchronous circuit simulator.[12] To evaluate performance, ZPL programs were compared on a

```
/* Set up an MPI datatype for column-vectors... */
MPI_Datatype columntype;
MPI_Type_vector(1, Height-2, Height, MPI_FLOAT, &column_type);
MPI_Type_commit(&column_type);

/* MPI_Send calls omitted... */

if (row != Top)
    MPI_Recv(&A[0][1], Width-2, MPI_FLOAT,
             MPI_ANY_SOURCE, North, MPI_COMM_WORLD, &status);

if (col != Right)
    MPI_Recv(&A[1][Width-1], 1, columntype,
             MPI_ANY_SOURCE, East, MPI_COMM_WORLD, &status);

if (row != Bottom)
    MPI_Recv(&A[Height-1][1], Width-2, MPI_FLOAT,
    MPI_ANY_SOURCE, South, MPI_COMM_WORLD, &status);

if (col != Left)
    MPI_Recv(&A[1][0], 1, column_type, column_type,
             MPI_ANY_SOURCE, West, MPI_COMM_WORLD, &status);

/* Calculate average, delta for all points */
delta = 0;
for (i=1; i<Height-1; i++) {
    for (j=1; j<Width-1; j++) {
             average = (A[i-1][j]+A[i][j+1]+
                        A[i+1][j]+A[i][j-1])/4;
             delta = max(delta, fabs(average - A[i][j]));
             B[i][j] = average;
    }
}
/* Find maximum diff */
MPI_Reduce(&delta, &error, 1, MPI_FLOAT;
             MPI_MAX, 0, MPI_COMM_WORLD);
```

(a)

```
[R] begin
   B := (A@north+A@east+A@west+A@south)/4;
   error := max<< fabs(A-B)
end;
```

(b)

Figure 5. The main loop of the Jacobi Iteration in (a) C with MPI and (b) in ZPL. For brevity, the `MPI_Send` calls are omitted from the MPI version.

single processor against sequential implementations, written in either C or Fortran. Speedup (relative to the best sequential results) was then computed to show how the programs performed as the number of processors grew. The performance of ZPL programs scaled well.

How ZPL Helps the Programmer

ZPL programs are more readable, concise, and easier to write than other approaches. ZPL does this in several ways.

Simplified source code

ZPL implicitly specifies communication and synchronization. As Figure 5 shows, this tremendously simplifies the source code. However, communication *is* evident in the source code, because operators such as `@` and `max<<` indicate where communication can be induced. Thus, only the details of communication are hidden from the programmer. Message passing, in contrast, hides too little of the communication; higher level languages such as HPF hide too much.

```
    DO NREL=1,ITER
      WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
!
!   RELAXATION OF THE RED POINTS
!
     U(2:NX-1,2:NY-1,2:NZ-1) =                       &
    &           FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+ &
    & U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+ &
    & U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+ &
    & U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))

     ELSEWHERE
!
!   RELAXATION OF THE BLACK POINTS
!
      U(2:NX-1,2:NY-1,2:NZ-1) =                      &
    &           FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+ &
    & U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+ &
    & U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+ &
    & U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))

     END WHERE
    ENDDO
```

(a)

```
for nrel := 1 to iter do
-- Relaxation of the red points
[I with Red] U := factor*(hsq*F+
                        U@top+U@bot+
                        U@left+U@right+
                        U@front+U@back);
-- Relaxation of the black points
[I without Red] U := factor*(hsq*F+
                        U@top+U@bot+
                        U@left+U@right+
                        U@front+U@back);
end;
```

(b)

Figure 6. Red/black relaxation loop from (a) Fortran90 and (b) ZPL versions of a 3D Poisson solver.

Regions and region operators

ZPL regions elevate the concept of the index set to an entity that can be named and manipulated. As Figure 5 shows, regions and region operators replace explicit looping and array indexing. Thus, the concept of operating on the entire region of an array is made clear. By contrast, explicit indexing is significantly more error prone and forces the programmer to reason about each loop individually. Moreover, by applying regions to entire statements and using expressions such as `A@north` in place of direct array indexing, ZPL removes redundancy and reduces the likelihood of errors.

ZPL's region construct has significant advantages, even when compared to other array languages. Figure 6 shows the inner loop of a 3D Poisson solver using red/black successive overrelaxation in Fortran 90 and in ZPL. The clarity of the ZPL code is striking in comparison to the Fortran. Much of the clarity comes from the region construct. The region `I` in ZPL encapsulates the same information as the Fortran slice `2:NX-1, 2:NY-1, 2:NZ-1,` except that the region applies to all arrays in a statement, while a slice must be specifed for each array reference. Because this "interior" region is used throughout the computation, it is both convenient and conceptually simpler to declare it once and use it symbolically thereafter.

The ZPL code's improved readability reduces the potential for errors. In the ZPL solution, different things look different—in this case using different direction and region names—whereas all of the HPF slices look similar except for the numerous constants, which can easily be mistaken for each other.

Boundary condition specification

ZPL supports common types of boundary conditions, such as constant, periodic, and mirrored boundaries. In Figure 2, we saw how constant boundary conditions were easily defined and how using regions cleanly separated the boundary-condition code from the common case code, making it both easier to understand and modify. By contrast, in message-passing programs, boundary conditions and communication code are often intertwined.

Figure 7 shows how ZPL simplifies the specification of periodic boundary conditions. The example shows Fortran 90 and ZPL code fragments from a finite-difference calculation using the shallow-water equations. This program con-

```
UOLD(M + 1,:N) = UOLD(1,:N)
VOLD(M + 1,:N) = VOLD(1,:N)
POLD(M + 1,:N) = POLD(1,:N)
U(M + 1,:N) = U(1,:N)
V(M + 1,:N) = V(1,:N)
P(M + 1,:N) = P(1,:N)

UOLD(:M,N + 1) = UOLD(:M,1)
VOLD(:M,N + 1) = VOLD(:M,1)
POLD(:M,N + 1) = POLD(:M,1)
U(:M,N + 1) = U(:M,1)
V(:M,N + 1) = V(:M,1)
P(:M,N + 1) = P(:M,1)

UOLD(M + 1,N + 1) = UOLD(1,1)
VOLD(M + 1,N + 1) = VOLD(1,1)
POLD(M + 1,N + 1) = POLD(1,1)
U(M + 1,N + 1) = U(1,1)
V(M + 1,N + 1) = V(1,1)
P(M + 1,N + 1) = P(1,1)
```
(a)

```
[south of I] wrap U, Uold, V, Vold, P, Pold;
[east of I] wrap U, Uold, V, Vold, P, Pold;
[se of I] wrap U, Uold, V, Vold, P, Pold;
```
(b)

Figure 7. Specifying boundaries using (a) Fortran 90 and (b) ZPL code fragments from a finite-difference calculation to predict weather using the shallow-water equations.

tains $m \times n$ arrays with periodic boundaries, so each array is allocated an extra row and column that is kept equal to the row or column on the opposite edge. The ZPL program uses the `wrap` statement to copy items from one side of an array to the other. For example, given `region I = [1..m, 1..n]` and direction `east = [0,1]`, the statement

```
[east of I] wrap U;
-- copy first column into last column
```

assigns to the region `[east of I]` (that is, `[1..m, n + 1]`) the data from the same-sized region on the opposite side of the array, namely, the region `[1..m, 1]`. The programmer's intent is clear from the text, reducing the potential for error. Thus, ZPL raises the level of abstraction by providing a direct solution for periodic boundary conditions. It also directly supports mirrored boundary computations, using the `reflect` statement. Here again, ZPL programmers can define names for indices once and use them symbolically thereafter.

By contrast, each of the first six lines of the Fortran 90 code (Figure 7a) copies the first row of an array to the last row of the same array. The next six lines copy the first column to the last column, and the last six lines copy the upper left corner item to the lower right. The use of explicit array indexing (in this case, slices) obscures the existence periodic boundary conditions.

Sequential semantics

ZPL's deterministic behavior greatly simplifies parallel programming as users no longer need to worry about issues such as explicit communication, race conditions, deadlock, or livelock. Deterministic behavior not only makes it easier to reason about programs, but it lets programmers develop and debug their programs in uniprocessor environments—such as PCs and workstations—and later execute their code for actual data sets on more powerful parallel computers. This scalability is beneficial because PC and workstation environments are typically more familiar, accessible, and available than parallel computers. Furthermore, this approach moves the program-development process off parallel computers so they can be reserved for production runs.

How ZPL Helps the Compiler

If ZPL programs do not perform well, the programming advantages of ZPL are lost. As we discussed above, ZPL offers good performance. In addition, the ZPL compiler can succeed in situations where parallelizing compilers and compilers for other parallel languages face difficulties.

By carefully raising the level of abstraction while remaining faithful to an appropriate abstract machine model, the ZPL language conveys more semantic information to the compiler than is possible with other approaches. The region construct, for example, raises the level of abstraction by explicitly representing an array's index set. Furthermore, region operators such as `of` and `@` provide structured means of modifying regions and produce the same benefits that structured control flow does over `goto`'s: they carry with them semantic information that the compilation process can exploit.

The ZPL compiler can succeed in situations where parallelizing compilers and compilers for other parallel languages face difficulties.

To provide context, we first describe the ZPL compiler's structure. We then explain how ZPL's high-level constructs help the compiler in unique ways.

Compiler structure

Our compiler accepts ZPL source code as input and outputs ANSI C code.[14] The C code is then compiled by native C compilers and linked with ZPL- and machine-specific libraries to produce executables for different hardware. Our ZPL compiler is currently targeted to Unix workstations, networked computers running MPI and PVM, shared-address-space parallel computers such as the Cray T3E and SGI Origin, and distributed-memory parallel computers such as the IBM SP-2 and Intel Paragon. Although the compiler produces a low-level representation of the source program, its internal representation preserves and exploits the high-level nature of the ZPL source program during the machine-independent compilation process.

Simplified analysis

Regions and region operators simplify the analysis required to generate communication. Just as using `@` operators helps programmers identify potential communication, the presence of an `@` indicates to the compiler that communication is necessary. Communication is also required for other operators such as reductions, and these are always immediately apparent from the source code.

Languages that use explicit array indexing must perform sophisticated analysis—such as recognizing linear recurrences of array indices—to detect communication. Where such analysis is imprecise or foiled by procedure-call boundaries, compilers must be conservative and use expensive runtime checks to determine where communication is needed. These limitations apply to automatic parallelization as well as to the compilation of most other parallel languages.

The slices in array languages such as Fortran 90 are not regions. Because they are applied to individual array references, rather than to the entire statement, their analysis can become quite complex and can flummox compilers. Moreover, because Fortran 90 contains Fortran 77 as a subset, Fortran 90 compilers must deal with all of the complexity of the inherited language.

Communication optimizations

ZPL's array language semantics allow the compiler to automatically manipulate entire arrays or slices of arrays. This representation trivializes an optimization known as *message vectorization* in which many small communication operations, typically found in a loop, are bundled together and transmitted as a single large message. This transformation is profitable on most machines because a communication operation's per-message overhead is typically large compared to its per-byte cost. More sophisticated optimizations—such as message pipelining and redundant-communication removal—are also greatly simplified by this high-level representation.[14]

These same transformations can be applied to other languages, but the undisciplined use of the more general array-indexing operation can lead easily to index expressions that cannot be statically analyzed.

Loop nests

As with all array languages, ZPL regions provide many benefits in loop-nest generation in the object C code. Sequential languages such as Fortran force the programmer to specify a complete ordering of loop iterations. This only complicates the compiler's task, as it must find alternate orderings that preserve data dependencies yet enable parallelization. In contrast, ZPL's regions provide exactly the information that is needed: the iteration space. With this information, the compiler can easily construct loop nests for the output C code. These loop nests can be optimized for spatial and temporal locality, and can be fused in the output C code to remove large intermediate arrays.[14]

Efficient indexing is also possible. Rather than compute array offsets for each access to an array element, the compiler can simply move a pointer through the array, requiring only a single addition per iteration. The technical terms for this optimization are *induction variable elimination* and *strength reduction*. In low-level languages such as C, this transformation is typically thwarted by pointer aliasing except in the case of statically declared arrays.

Flexible communication

Yet another advantage of high-level languages is that the compiler can insulate the programmer from machine specifics, while at the same time choosing the communication mechanisms best suited to a given machine.

The ZPL compiler does this by generating generic communication routines that can be mapped to different communication mechanisms on different machines. In particular, the compiler uses the Ironman interface[2] to insert four different routines that together define the legal intervals during which data can be transmitted from one processing element to another. These routines carry no specific implementation, so they unify all forms of communication, including all MPI 2.0 standard routines. The generic Ironman routines are mapped to a particular machine's communcation interfaces by linking in machine-specific implementations. Thus, for example, the routines can be bound to the shared memory `Put` operation on the T3E or MPI's `ready-send` on the IBM SP-2.

Another advantage of this approach is that the compiler-generated C code is not specific to any machine, so retargeting the compiler primarily involves implementing machine-specific communication routines. Because the compiler is easily retargeted, it can be widely available across many platforms.

Low-level parallel programming approaches such as message passing have many inherent problems because they embed too many implementation decisions into the source program. High-level approaches can be successful if they present carefully chosen constructs, such as those in the ZPL data parallel array language. ZPL allows for ease of programming and produces efficient compiled code across different architectures. ◆

Acknowledgments

This research was supported by Darpa Grant E30602-97-1-0152, National Science Foundation Grant CCR-9710284, and a grant of HPC time from the Arctic Region Supercomputing Center. Research was conducted at the Cornell Theory Center, which receives funding from the National Science Foundation and New York State, with additional support from the National Center for Research Resources at the National Institutes of Health, IBM Corporation, and other members of the center's Corporate Partnership Program. We thank Samuel Guyer for his graphing skills.

References

1. MPI Forum, *MPI Standard 2.0*, tech. report; available online at http://www.mcs.anl.gov/mpi/.
2. B.L. Chamberlain, S. Choi, and L. Snyder, "A Compiler Abstraction for Machine-Independent Communication Generation," *Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, Berlin, 1997.
3. D. Notkin et al., "Experiences with Poker," *Proc. ACM Sigplan Symp. Parallel Programming: Experience with Applications, Languages, and Systems*, ACM Press, New York, 1988.
4. L. Snyder, *The ZPL Programmer's Guide*, to be published by MIT Press, Cambridge, Mass., 1998; also available at ftp://ftp.cs.washington.edu/pub/orca/docs/zpl.guide.ps.
5. L. Snyder, "Type Architecture, Shared Memory, and the Corollary of Modest Potential," *Ann. Rev. Computer Science*, Annual Reviews, Inc., Palo Alto, Calif., 1986, pp. 289–318.
6. T. Ngo, L. Snyder, and B. Chamberlain, "Portable Performance of Data Parallel Languages," *Supercomputing '97*, IEEE Computer Society Press, Los Alamitos, Calif., 1997 (published only on CD-ROM, ISBN 0-89791-985-8).
7. T.A. Ngo, *The Role of Performance Models in Parallel Programming and Languages*, PhD thesis, Univ. of Washington, Dept. of Computer Science and Engineering, Seattle, 1997.
8. C. Lin et al., *ZPL vs. HPF: A Comparison of Performance and Programming Style*, Tech. Report 95-11-05, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, 1994.
9. C. Lin and L. Snyder, "Simple Performance Results in ZPL," *Languages and Compilers for Parallel Computing*, K. Pingali et al., eds., Springer-Verlag, Berlin, 1994, pp. 361–375.

10. E.C. Lewis et al., "A Portable Parallel N-Body Solver," *Proc. Seventh SIAM Conf. Parallel Processing Scientific Computing*, D. Bailey et al., eds., Society for Industrial and Applied Mathematics, Philadelphia, 1995, pp. 331–336.
11. M.D. Dikaiakos et al., "The Portable Parallel Implementation of Two Novel Mathematical Biology Algorithms in ZPL," *Ninth Int'l Conf. Supercomputing*, ACM Press, 1995, pp. 365–74.
12. W. Richardson, M. Bailey, and W.H. Sanders, "Using ZPL to Develop a Parallel Chaos Router Simulator," *1996 Winter Simulation Conf.*, SCS Int'l., San Diego, 1996, pp. 809–816.
13. W. Crowley, C.P. Hendrickson, and T.I. Luby, *The Simple Code*, Tech. Report, UCID-17715, Lawrence Livermore Laboratory, Livermore, Calif., 1978.
14. B. Chamberlain et al., "Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines," *Languages and Compilers for Parallel Computing*, D. Sehr, eds., Springer-Verlag, Berlin, 1996, pp. 481–500.

Bradford L. Chamberlain is a PhD student at the University of Washington, Seattle. His areas of interest include compilers, parallel languages, and parallel computing. He received his BS in computer science from Standford University and his MS from the University of Washington. He is a recipient of an Intel Foundation fellowship for the 1997–1998 academic year. Contact him at the Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350.

Sung-Eun Choi is a PhD candidate at the University of Washington, Seattle. Her research interests include parallel-programming language design, compilers, interprocessor communication, high-performance architectures, and scientific computing. She received her BS in computer science from the University of Illinois at Urbana-Champaign and her MS from the University of Washington. Contact her at the Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350.

E Christopher Lewis is working on his PhD at the University of Washington, Seattle. His research interests include compilers, computer architectures, and parallel-programming languages. He received his BS in computer science from Cornell University and his MS from the University of Washington. Contact him at the Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350.

Lawrence Snyder is a professor of computer science and engineering at the University of Washington, Seattle. His research has ranged from the design and development of a 32-bit single-chip CMOS microprocessor, the Quarter Horse, to proofs of the undecidability of properties of programs. He received his BS in mathematics and economics from the University of Iowa and his PhD in computer science from Carnegie Mellon. He is a fellow of the IEEE and ACM. Contact him at the Dept. of Computer Science, University of Washington, Seattle, WA 98195-2350.

W. Derrick Weathersby is a PhD candidate in computer science at the University of Washington, Seattle. His research interests are in languages, compilers, and systems for distributed and parallel computing. He received his BS in mathematics and computer science at the University of Illinois. Contact him at the Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350.

Calvin Lin is an assistant professor in the Dept. of Computer Sciences at the University of Texas at Austin. His research interests are in languages and compilers for portable and efficient parallel programming. He received his BSE from Princeton University and his PhD in computer science from the University of Washington. He is the codesigner of the ZPL programming language. Contact him at the Dept. of Computer Sciences, University of Texas at Austin, Austin, Texas, 78712; lin@cs.utexas.edu.