Appears in "Proceedings of the Seventh International Workshop on Languages and Compilers for Parallel Computing," pages 361-375, 1994.

SIMPLE Performance Results in ZPL

Calvin Lin and Lawrence Snyder

University of Washington

Abstract. This paper presents performance results for ZPL programs running on the Kendall Square Research KSR-2 and the Intel Paragon. Because ZPL is a data parallel language based on the Phase Abstractions programming model, these results complement earlier claims that the Phase Abstractions model can lead to portability across MIMD computers. The ZPL language and selected aspects of the compilation strategy are briefly described, and performance results are compared against hand-coded programs.

1 Introduction

In 1991 the authors claimed that programs written in languages founded on the CTA machine model and the Phase Abstractions programming model would be portable across the major families of MIMD parallel computers [6, 8]. The evidence offered to support this claim was the observed performance of the SIMPLE computational fluid dynamics program on five parallel machines that represented the MIMD machines then available: the Sequent Symmetry, BBN Butterfly, Intel iPSC/2, nCUBE/7 and a Transputer array machine; see Figure 1. It was noted that this program achieved at least P/2 speedup on all platforms, and it met or exceeded all published performance for the SIMPLE computation.[2] Though no comparable degree of machine independence had previously been reported, the results did not represent any language or compiler support: The SIMPLE program was written in pseudocode and hand translated to C code that made calls to a message passing library. The problem is that although the hand translation used no exotic analysis and assumed no sophisticated compiler technology, the possibility existed that high level languages and their compilers would be unable to produce the same high quality object code for these disparate computers. This paper addresses this problem by repeating the previous experiment on two modern machines, the Intel Paragon and the Kendall Square Research KSR-2—this time using a high level language called ZPL.

The first goal of this paper is to present evidence that a compiler *can* perform the necessary translations to achieve portability for the SIMPLE program.

[1] This research was supported in part by Office of Naval Research Contract N00014-92-J-1824 and NSF Contract CDA-9211095

[2] Figure 1 compares against results from Hiromoto *et al.*'s hand coded results on the Denelcor HEP [4] and Pingali and Rogers' compiled Id program on the iPSC/2 [11].

Though the evidence is not yet complete—the compiler has only been targeted to two MIMD machines—it is stronger in one sense: The original pseudocode represented a very low level language, so while the pseudocode used powerful concepts from the Phase Abstractions programming model, it did not exploit any high level abstractions such as the data parallel facilities of ZPL. A second goal of this paper is to compare the performance of small ZPL programs with hand coded programs written in C with message passing. These results give an indication of the compiler's effectiveness at producing efficient code. Together, the data presented here supports the claim that the CTA and Phase Abstractions are a conceptual foundation on which portable parallel programs can be written.

This paper is organized as follows. Section 2 provides background by reviewing the Phase Abstractions programming model. Sections 3 and 4 then describe the ZPL language and the ZPL compiler. Our experiments are presented in Section 5, followed by concluding remarks.

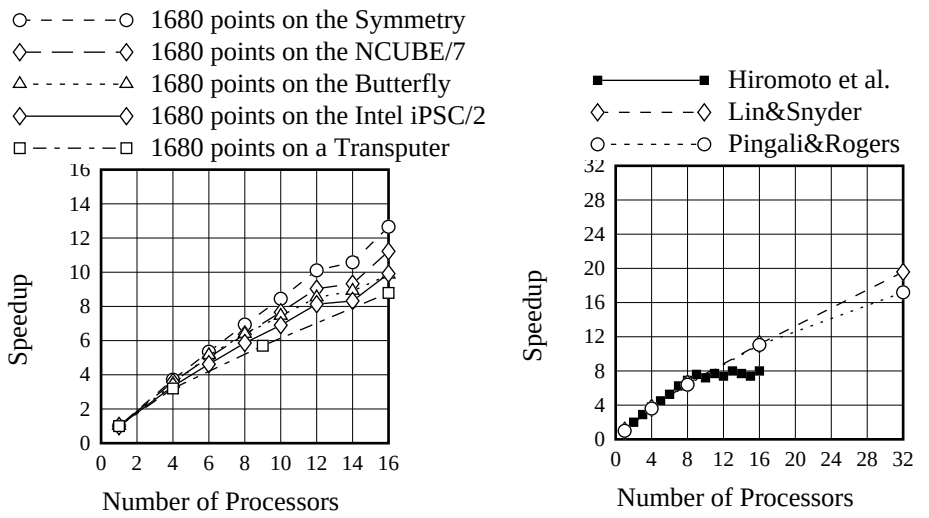

Fig. 1. (a) SIMPLE Speedup on Various Machines (b) SIMPLE with 4096 points

2 Background: Phase Abstractions

The Phase Abstractions programming model serves as the basis for both our earlier results and the current language-based results. This section briefly reviews the salient features of the Phase Abstractions. A more complete description can be found elsewhere [1, 3, 14].

The Phase Abstractions programming model identifies three levels of parallel programming: the X level, the Y level, and the Z level. The Z level specifies a program's overall control logic and the sequential invocation of *phases* to solve the overall problem. Phases are defined at the Y level, which isolates the explicitly parallel aspects of a parallel program. A phase is a parallel algorithm that typically uses a single pattern of communication; examples include parallel implementations of FFT, Gaussian Elimination, and global maximum. Phases are composed of multiple processes, and these are programmed at the X level. X level processes are defined by sequential code that can communicate with other processes through some message passing mechanism.

The Y level deals with explicit parallelism, so new abstractions are provided to help manage this parallelism. In a non-shared memory model, each process consists of local data, local code, and some local interface to the overall communication structure. In the Phase Abstractions model these three components are encapsulated in a *section*, which is a logical unit of concurrency. Each section can be independently assigned to a processor for execution, and the degree and granularity of parallelism is controlled by increasing or decreasing the number of sections.

The notion of *ensembles* is used to partition each phase's data, code and communication structure. A *data ensemble* specifies how global data is partitioned across the different sections. A *code ensemble* assigns a piece of code to each section, and a *port ensemble* defines ports that connect the section with other sections; these ports are used by X level processes when communicating with other processes. A given section typically consists of multiple data ensembles, a single code ensemble, and a single port ensemble[3]: the code specified by the code ensemble manipulates the local data as defined by the data ensembles, and communicates with other processes as defined by the port ensembles. By describing all aspects of a section through the common notion of an ensemble, all aspects of parallelism can scale in a coherent fashion.

3 The ZPL Language

For the sake of performance, low level languages can be built upon the Phase Abstractions, and, in fact, such languages have been proposed [8, 9]: Orca C is a low level non-shared memory language that extends the programming model with modest conveniences. Thus, Orca C is a MIMD language that gives programmers control over the performance-sensitive aspects of their programs. However, for regular data parallel computations, such control is typically not as critical to performance and such a language can be unnecessarily tedious. ZPL is a subset of Orca C that aims to provide conciseness, convenience, and clarity for

[3] In sophisticated computations a section can comprise multiple code ensembles—for example, Kung and Leiserson's systolic matrix multiplication algorithm [5] ideally consists of two processes per processor, one to move data and one to compute new local products—and may comprise multiple port ensembles.

data parallel computations. By addressing a restricted domain the language and compiler design are simplified and good performance can be attained.

ZPL may seem somewhat limited in its expressiveness when compared to other data parallel languages, and these in turn are limited relative to the requirements of full MIMD computation. However, the limitations of ZPL are not a concern because ZPL is a subset of Orca C. Z level programs in Orca C consist of ZPL code mixed with invocations of programmer-defined phases. Thus, computations not easily expressed in ZPL can be written using the full power of Orca C, while those naturally expressible in ZPL can exploit the convenience of ZPL.

3.1 Language Features

As a Z level language, ZPL provides a global view of the computation, so the programmer sees a single address space and all parallelism is implicitly specified. The principal concepts of ZPL are briefly enumerated below:

Regions. The primary mechanism for expressing data parallel computation is through regions. A region is an index set, and can be defined as follows: `R = [1..n, 1..n].` Executing a statement containing array references in the context of a region name causes the statement to be instantiated for and executed with each index value of the region.

Array computations. By using regions, expressions involving arrays can be performed without tedious indexing. For example, the following statement finds for all index values of `R` the difference between the elements of `A` and `B.`

```
[R] C := A-B;
```

The operations are performed elementwise and provide the user with a clear and succinct means of expressing concurrency.

Directions. Directions are used to express offsets and translate index sets. For example, for a two dimensional array, the northern direction can be used to translate a position to the row above it. The northern direction might be defined as follows: `north = [ -1, 0].`

At. The At operator (`@`) is used with directions to provide relative indexing. For example, the averaging of the four nearest neighbors required in the Jacobi iteration is given by the following statement:

```
[R] (A@north+A@east+A@west+A@south)/4;
```

Of. The Of operator uses a direction to define the boundary regions adjacent to some base region. Thus, the "extra" 0^{th} row of `A`, required when shifting array `A` north, is defined and initialized as with one statement: `[north of R] A := 0`.

Wrap and *Reflect.* Special statements, `wrap` and `reflect`, support mirrored and periodic boundary conditions.

Control structures. ZPL includes the usual control constructs, including `if`, `for`, `repeat`, `while`, `exit`, `continue`, and `return`, along with function calls.

Promotion. Scalar values can be used in array expressions in ZPL as if they were arrays of the proper dimension and size. Thus, in the expression `2*A`, the scalar is promoted to conform to the region in which it is executed, and in effect matches the portion of the array operand with which it is executed.

Masks. It is possible to apply operations selectively to the elements of an array according to a boolean condition. For example, `[R with mask]` specifies the indices of `R` for which the value of `mask` is true (non-zero).

Reduce and *Scan.* The usual reduction and scan operators are provided, including sum, product, logicals, bitwise logicals, minimum, and maximum. For example, `max\ A` reduces `A` to its largest element.

ZPL has several other interesting features that, although worthy of mention, are rather involved and not needed in the programs discussed in the experiments below. These are described in the literature [10, 15].

The above summary of ZPL should allow the Jacobi program of Figure 2 to be easily understood. The program begins with declarations, first for the region `R`, then for two array variables, `A` and `Temp`, then for a scalar, `error`, and finally for a set of directions. The next block of code defines and initializes the boundary regions, assigning zeros to the array and all boundaries except the southern boundary, which is set to `1.0`. The body of the program, executed in the context of the region specifier, `[R]`, iterates until convergence: The values of the next iteration are found by averaging the four nearest neighbors at each point. In the statement, `error := max\(abs(Temp - A))`, the absolute difference of the elements of the two iterations is computed by promoting the scalar function, `abs()`, to apply to the elements of its array argument; the maximum reduction is then performed to find the greatest error that will be used to test for termination.

3.2 Parallel Foundation for ZPL

Although its language semantics are implicitly parallel, ZPL's parallel execution is made explicit by exposing the Phase Abstractions as its underlying programming model. This is a necessity because ZPL is a sublanguage of an explicitly parallel language. The parallelism is formulated in terms of the CTA [13], an abstract machine on which the program is logically thought to be executed. Programmers, knowing the general characteristics of the parallel execution, can thus assess the performance implications of alternate program implementations.

Logically, the ZPL program should be thought of as a sequence of phase invocations, with the ZPL compiler producing the X and Y level code that implements these phases. In the parlance of the Phase Abstractions, ZPL's arrays

```
program Jacobi;

config var  N: integer = 100;
direction   east  = [0, 1];
            west  = [0,-1];
            north = [-1,0];
            south = [1, 0];

region      R = [1..N, 1..N];

constant    epsilon: real = .01;
var         A, Temp: [R] real;
            error  : real;

procedure Jacobi();
begin
  [east of R]  A := 0.0;
  [west of R]  A := 0.0;
  [north of R] A := 0.0;
  [south of R] A := 1.0;
  [R]          A := 0.0;

  [R]          repeat
                   Temp := (A@east + A@west + A@north + A@south)/4;
                   error := max\ (abs(Temp-A));
                   A := Temp;
               until error < epsilon;
end;
```

Fig. 2. ZPL Program for the Jacobi Iteration.

are data ensembles that by default are distributed across the processors in a 2D blocked fashion,[4] and scalars are Z level control variables that are replicated across all processors. The code ensemble generated for each of these lightweight phases is the object C code produced by the ZPL compiler for the particular ZPL statement. The port ensemble for each phase, while implicit to the user, is explicit to the compiler: The `@` and `wrap` operators specify connectivity between neighboring sections and the scan and reduce operators specify a machine-specific tree connectivity.

The use of the Phase Abstractions model and the CTA abstract machine is crucial because it provides the compiler with a single view of all MIMD computers. In particular, the model leads to a single compiler that produces identical object code for both the shared address space KSR-2 and the non-shared memory Intel Paragon.[5]

[4] The decomposition can be changed through Orca C's Y level ensemble declarations.

[5] We currently perform no machine-specific optimizations.

4 The ZPL Compiler

We have modified the Parafrase-2 source-to-source translator [12] to compile ZPL programs to SPMD C code with calls to machine-specific communication routines. We rely on each machine's native C compiler to complete the translation to machine code.

Space limitations prevent a full description of the compilation process, but three significant tasks are the insertion of communication, the implementation of array statements, and the representation of ensembles. Communication is explicitly represented in our AST as Send and Receive nodes. In the naive case these communication nodes are inserted wherever the At operator is used. Array statements are implemented by creating *Mloops* in the AST. Mloops represent the region that applies to a given ZPL statement, and Mloops are converted to nested `for` loops in the object code. The use of Mloops differentiates loops derived from array operations from loops defined by users, a distinction that can be useful because of the restricted nature of Mloops. Finally, ensembles are implemented as arrays with additional descriptors that describe their size, shape, and amount of *fluff*, where fluff is a cache used for holding neighboring array values.

The current ZPL compiler performs only a small number of optimizations. One optimization concerns the elimination of redundant communication operations, an effect that is similar to common subexpression elimination. Each local section of an ensemble maintains a cache to hold data from neighboring sections, and these caches are updated only when Def-Use analysis detects that it is necessary. This optimization is explained in more detail in Appendix A.

We currently perform conservative Mloop fusion that reduces the number of Mloops in SIMPLE from about 165 to about 100. More aggressive fusion that looks for optimal loop iteration orders is being explored.

5 Experiments

Our primary result compares the performance of a compiled ZPL implementation of the SIMPLE benchmark against earlier implementations written in C. This result completes earlier claims that the Phase Abstractions model can be used to create portable programs. We then present a second set of results involving the Jacobi iteration. By looking at a small program that is essentially a subset of the SIMPLE computation, this experiment provides a more detailed look at the performance difference between ZPL programs and hand-coded C programs.

Hardware. The Intel Paragon is a distributed memory computer with 18 nodes arranged in a mesh structure. Each node consists of an Intel i860 processor and 16 MB of local memory, and runs the OSF Mach operating system.

The Kendall Square Research KSR-2 has 56 processors connected by two levels of rings. A coherent cache-only memory structure provides a global address space. Each custom processor runs at 40 Mhz and can issue both integer and

floating point instructions on each cycle. Each processor has a 256 KB instruction cache, a 256KB data cache, and 32 MB of local memory.

5.1 SIMPLE Results

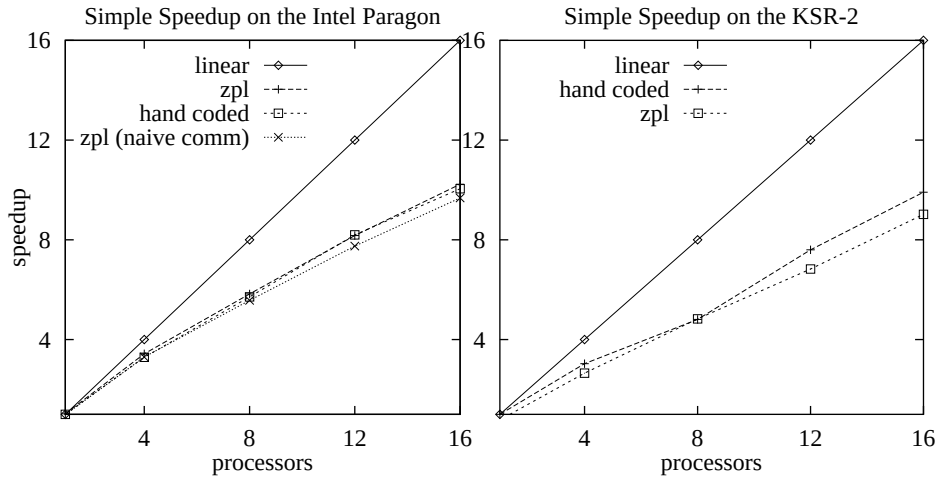

Fig. 3. SIMPLE results.

Figure 3 compares the speedup of SIMPLE written in ZPL against our hand-coded version written in C with messages passing [6]. Ten iterations were run for a problem with 256×256 data points (all C programs were compiled with optimizations at level -O2). The graph on the left shows that on the Paragon the speedup of our ZPL program (the curve labeled "zpl") is very close to that of the hand-coded program. Note that the P=1 time for "hand coded" is a sequential program with no unnecessary overhead for parallelism, and the ZPL program running on one processor was still 1.9% faster. On the Paragon, the better node performance of the ZPL program is due to our use of "walkers" and "bumpers," described below.

At P=4 the ZPL program is 4.1% faster than hand-coded, but as the number of processors increases the hand-coded program does relatively better. For example, at P=16 the ZPL program is 3.9% slower. This performance difference appears to be due to the slightly higher overhead of the ZPL object code, and as P grows the cost of this overhead becomes a larger percent of the execution time. For example, the ZPL compiler produces code for communication between two neighbors that works for any processor and any region. By contrast, the hand-coded program hard codes some of this information, knowing, for example, that

a process on the west edge of the computation has no western neighbors to send messages to.

ZPL's superior performance at P=1 is surprising because the ZPL program has all of the overhead mentioned above. However, a detailed examination of the cost of Mloops versus the cost of hand-coded nested loops shows that our use of walkers and bumpers is a big win on the Paragon: Walkers are pointers that are used to iterate over arrays in Mloops; walkers are advanced at each iteration by bumpers. Of course, the hand-coded program could also use walkers and bumpers, but this would be extremely tedious. Note that if all arrays were declared to reside contiguously in memory, the Paragon's C compiler could use induction variable elimination and strength reduction to achieve better performance than our walkers and bumpers, but this would require that all arrays be statically defined.

The curve labeled "zpl (naive comm)" represents a naive communication insertion strategy where sends and receives are emitted for every statement that contains an @. Figure 3 shows that the performance difference between "naive comm" and "zpl," which uses our Def-Oriented communication insertion algorithm, is considerable.

The graph on the right of Figure 3 shows the performance on the KSR-2. At P=1 the ZPL program is 27.9% slower than hand-coded. For P greater than one the performance of the two programs is very close, with the hand-coded program being about 10% faster. Interestingly, these trends are completely reversed from the Paragon. Further analysis is needed to explain this behavior and an anomaly at P=8, where the ZPL program is slightly faster.

5.2 Jacobi Results

In addition to SIMPLE, we also compare a ZPL implementation of Jacobi against a hand-coded message-passing implementation [7] for a 512×512 problem that converged at 279 iterations. (See Figure 4.) At P=1 the ZPL program is 5.2% slower than hand-coded. For more than one processor the overhead is between 5.4% (P=4) and 1.8% (P=16). The discontinuity at P=8 of the hand-coded curve is an artifact of the program's data partitioning. For a 512×512 problem size, the number of cache misses in the inner loop is roughly halved as we double the number of processors from P=1 to 2 and from 2 to 4. However, after P=8 no further reduction of caches misses occurs because the number of columns per processor remains unchanged. That is, each processor has 128 columns for P=8 (256×128), P=12 (170×128) and P=16 (128×128).

Figure 4 also shows a curve labeled "naive access," which shows the importance of using walkers and bumpers. The "naive access" curve represents a straightforward approach where each array access is computed independently. The high cost of these array accesses is due to the fact that each array may have different amounts of fluff, and many aspects of ZPL arrays—such as their dimension and the size of each element—may not be known at compile time.

Strangely, the ZPL program is faster than hand-coded on the KSR. We suspect the difference is due to data alignment and caching effects, but further

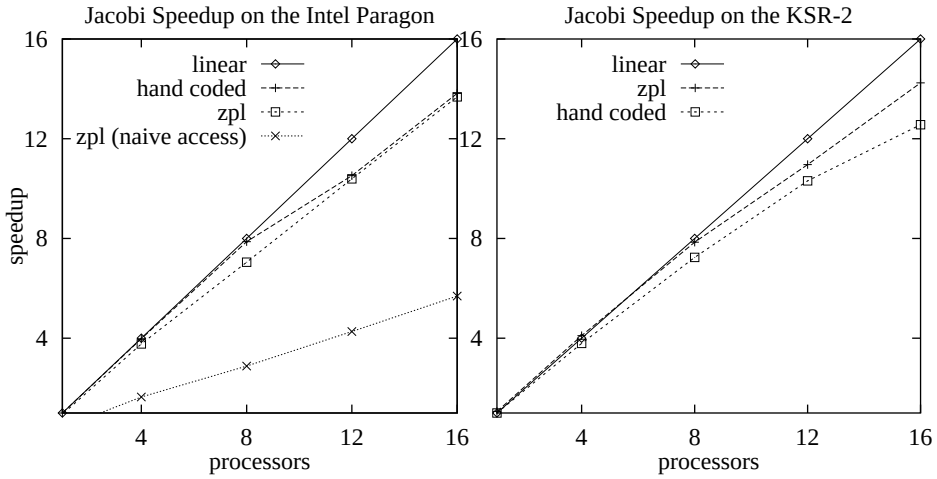

Fig. 4. Jacobi results.

investigation is required. These results contrast with the SIMPLE results where the ZPL program was faster than hand-coded on the Paragon but slower on the KSR. We observe that the Paragon's node compiler is more robust than the KSR's and appears to perform superior optimizations. If we assume that the hand-coded programs do not benefit as much from scalar optimizations, we observe that the Paragon's superior compiler is likely to have little impact on straightforward programs such as Jacobi, and larger impact on more complicated programs such as SIMPLE. Thus, the Paragon's native C compiler improves the ZPL performance of SIMPLE (relative to hand-coded) more than it improves the ZPL performance of Jacobi. On the KSR, however, the node compiler has less effect.

While some of the results in this section have not been fully explained, one conclusion is clear: The ZPL programs are extremely competitive with hand-coded C programs.

6 Conclusion

This paper complements earlier Phase Abstractions portability experiments by showing performance results for a SIMPLE program written in ZPL, a high level language that is based upon the Phase Abstractions. The present data shows good performance results for two very different parallel computers, the Intel Paragon and the Kendall Square Research KSR-2. Additional optimizations are underway that will concentrate on eliminating overhead of high level data parallel operations. For example, array temporaries can be expensive and

should be eliminated, re-used, or converted to scalars whenever possible. Further optimizations to reduce or hide communication latency are also anticipated.

The success of the ZPL compiler can be attributed to two factors. First, the ZPL language was designed to fit within the Phase Abstractions model as part of Orca C, and this design simplifies the compiler's task. ZPL is a Z level language that can focus on well-studied parallel abstractions such as data parallel array operations and reduction operations. Although some language features (not all features were discussed in this paper) cause non-trivial obstacles and interesting implementation problems, the separation from MIMD parallelism is beneficial. For example, communication only comes from structured operations such as `@`, `wrap`, and reductions; this contrasts sharply with other languages that may require communication for arbitrary array accesses. Second, by building on the Phase Abstractions programming model the compiler can use the same concepts that make Phase Abstractions applicable to a wide class of parallel computers: the non-shared memory view of the machine that encourages data locality, the notion of ensembles that parameterizes grain size, and the decomposition of a program into phases that makes ZPL compatible with the other components of Orca C.

Acknowledgments. We thank those who have helped implement ZPL—Ruth Anderson, Bradford Chamberlain, Sung-Eun Choi, George Forman, E Chris Lewis, Kurt Partridge, and W. Derrick Weathersby. We are particularly grateful to E Lewis for his critical role in acquiring the experimental data presented here.

References

1. Gail Alverson, William Griswold, Calvin Lin, David Notkin, and Lawrence Snyder. Abstractions for portable, scalable parallel programming. Technical Report 93–12–09, Department of Computer Science and Engineering, University of Washington, submitted to *IEEE Trans. on Parallel and Distributed Systems*, 1993.
2. S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN'93 Conference on Program Language Design and Implementation*, June 1993.
3. William Griswold, Gail Harrison, David Notkin, and Lawrence Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990. Charleston, South Carolina.
4. R. E. Hiromoto, O. M. Lubeck, and J. Moore. Experiences with the Denelcor HEP. In *Parallel Computing*, pages 1:197–206, 1984.
5. H. T. Kung and C.E. Leiserson. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980. Section 8.3, by C. Mead and L. Conway.
6. Jinling Lee, Calvin Lin, and Lawrence Snyder. Programming SIMPLE for parallel portability. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 84–98. Springer-Verlag, 1992.
7. Calvin Lin and Lawrence Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages II 163–180, 1990.

8. Calvin Lin and Lawrence Snyder. A portable implementation of SIMPLE. *International Journal of Parallel Programming*, 20(5):363–401, 1991.
9. Calvin Lin and Lawrence Snyder. Data ensembles in Orca C. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 112–123. Springer-Verlag, 1993.
10. Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 96–114. Springer-Verlag, 1993.
11. Keshav Pingali and Anne Rogers. Compiler parallelization of SIMPLE for a distributed memory machine. Technical Report 90–1084, Cornell University, 1990.
12. Constantine Polychronopolous, Milind Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 39–48, August 1989.
13. Lawrence Snyder. Type architecture, shared memory and the corollary of modest potential. In *Annual Review of Computer Science*, pages I:289–318, 1986.
14. Lawrence Snyder. Foundations of practical parallel programming languages. In *Proceedings of the Second International Conference of the Austrian Center for Parallel Computation*. Springer-Verlag, 1993.
15. Lawrence Snyder. A ZPL programming guide. Technical report, Department of Computer Science and Engineering, University of Washington, 1994.
16. Reinhard v. Hanxleden and Ken Kennedy. A code placement framework and its application to communication generation. Technical Report CRPC-TR93337, Center for Research on Parallel Computation, Rice University, October 1993.

A Communication Insertion

We employ Def-Use analysis to insert communication nodes into the AST: Send nodes are placed after arrays are modified and Receive nodes are placed before arrays are read. Here we distinguish between standard Defs and Uses that indicate data dependencies and the subset of these that induce communication. Henceforth we restrict our attention to the latter. Communication is induced when a pair of array references have different direction vectors for their Def and their Use. For example, the following pairs of statements induce communication:

```
     A := ...                  /* Def of A */
   ... := A@east;              /* Use of A */

B@east := ...                  /* Def of B */
 ...   := B;                   /* Use of B */
```

The following pairs of statements do not produce communication.

```
     A := ...                  /* Def of A */
   ... := A;                   /* Use of A */

B@east := ...                  /* Def of B */
 ...   := B@east;              /* Use of B */
```

Note that as opposed to standard scalar Def-Use (DU) chains, each of the above DU chains represents the transmission of an entire fluff area of an array whenever the direction vectors for the Def and Use differ.

Multiple direction vectors can map to the same processor direction. For example, the direction vectors `[0,1]` and `[0,2]` both map to the `east` processor direction. To see the importance of using processor directions, consider the following example where a single message consisting of two columns can satisfy both Uses of the variable `A`.

```
  A := ...
... := A@east;        /* insert single Receive before this statement */
... := A@east2;
```

If our algorithm considered only direction vectors, `east` and `east2` would be distinct and require separate communication. Our solution will instead send two columns by inserting a single message before the reference to `A@east`.

Def-Oriented Insertion Algorithm Our initial algorithm uses a "Def-oriented" approach and is shown in Figure 6. The goal of the algorithm is to identify the first DU chain corresponding to each processor direction and insert a Send node and Receive node for this DU chain.

The algorithm traverses the set of DU chains emanating from a single Def node (the *source node*) and maintains a `visited` table (see Figure 5) that has one record per processor direction (there are eight for 2D block decompositions), with the `mark` bits initially set to `0`. Each time a DU chain is traversed, the `mark` bit for the appropriate processor direction is set to 1, the `insertion_point` is set to point to the statement that corresponds to the Use (or end of basic block), and the `size` field is set to the magnitude of the Use's direction vector. Once the `mark` and `insertion_point` fields have been set they are never changed, but if we encounter a variable whose vector length is greater than the value in the table, the `size` field is set to the larger value. When the last of the DU chains for the source node has been traversed, communication nodes are created: A pair of nodes is created for each processor direction whose mark bit is 1. The Send is inserted after the Def and the Receive is inserted at the specified insertion point.

The algorithm is similar for the case that an `@` appears on the left hand side of an assignment, except we then look for *any* Use of the same array on the right hand side. An additional bit is used to differentiate `@`'s that appear on the left hand side from those that appear on the right.

Control flow can complicate the insertion of communication because corresponding Send and Receive operations must either both execute or both not execute. Hence, our algorithm places Send nodes immediately after Defs and places the corresponding Receives as late as possible within the same basic block. Finally, we point out that the above algorithm operates on a per-region basis, so there is a separate `visited` table for each region, and messages are never combined for fluff that corresponds to different regions.

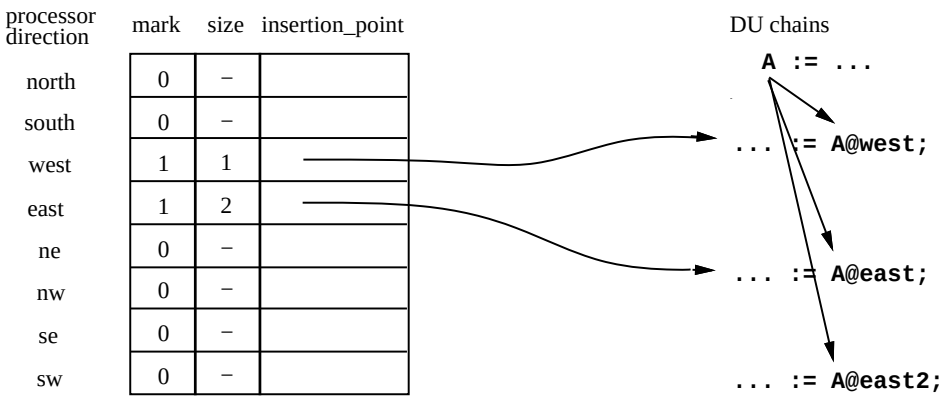

Fig. 5. The Def-Oriented insertion algorithm's `visited` table.

```
for each Def of an array in the dependency graph
{
    Clear the visited table;
    Traverse all DU chains emanating from the current Def;
    {
        if Def has no @
            Look for references to the array with @'s on rhs;
        else Def has an @
            Look for any reference to the array on rhs;
        for each such array reference found
        {
            if mark is 0 for this chain's processor direction
            {
                Set mark to 1 for this processor direction;
                Set insertion_point to statement corresponding to Use;
                Set size according to the direction vector;
            }
            else
            {
                (this processor direction already visited)
                Update size if this Use's direction vector > size;
            }
        }
    }
}
```

Fig. 6. The Def-Oriented communication insertion algorithm.

There are many other possible communication insertion algorithms.[2, 16] A *Use-Oriented* approach would insert Receives immediately before Uses and place Sends as early as possible within the same basic block. This approach may be better than Def-oriented in terms of combining messages for different variables. Much work remains in comparing and developing communication insertion algorithms.

Diagonal Directions Communication along diagonal processor directions results in communication with up to three different processes since a 2D block decomposition is used. For example, a reference to `A@northeast` induces communication with the neighbors to the north, northeast, and east. Thus, we decompose diagonal processor directions into their component pieces (see Figure 7). Notice that each orthogonal component direction—in this case the North and East components—spans an entire dimension of the region. Although `A@northeast` does not require the left element, we include this value (only if we also see `A@north`) to reduce the number of messages. For example, if the North Component did not include the "missing corner," the following code fragment would require four messages: one to send each of the three components of `A@northeast`, and one to satisfy `A@north`.

```
  A := ...
... := A@northeast;          /* send North, East and NE Components */
... := A@north;              /* send North Component */
```

Our implementation, however, sends a single message to update both `A@north` and the North Component of `A@northeast`.

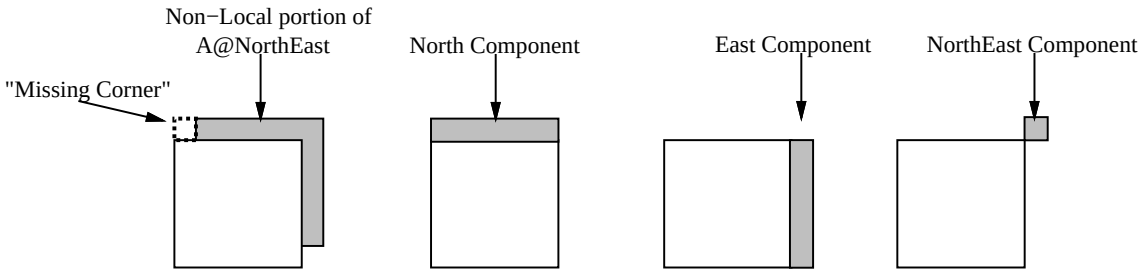

Fig. 7. Diagonal communication is broken into component pieces.

This article was processed using the LaTeX macro package with LLNCS style