Language Support for Pipelining Wavefront Computations*

Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350 USA
{brad,echris,snyder}@cs.washington.edu

Abstract. Wavefront computations, characterized by a data dependent flow of computation across a data space, are receiving increasing attention as an important class of parallel computations. Though sophisticated compiler optimizations can often produce efficient pipelined implementations from sequential representations, we argue that a language-based approach to representing wavefront computations is a more practical technique. A language-based approach is simple for the programmer yet unambiguously parallel. In this paper we introduce simple array language extensions that directly support wavefront computations. We show how a programmer may reason about the extensions' legality and performance; we describe their implementation and give performance data demonstrating the importance of parallelizing these codes.

1 Introduction

Wavefront computations are characterized by a data dependent flow of computation across a data space. Though the dependences imply serialization, wavefront computations admit efficient, parallel implementation via pipelining [6, 21]. Wavefront computations frequently appear in scientific applications, including solvers and dynamic programming codes; and recently, the ASCI (Accelerated Strategic Computing Initiative) SWEEP3D benchmark has received considerable attention as an important wavefront computation [10, 20]. The Fortran 77 and Fortran 90 SPECfp92 Tomcatv code fragments in Figures 1(a) and (b) represent typical wavefront computations.

The growing interest in wavefront computations raises the question of how they may be best expressed and realized on parallel machines. Choosing not to distribute array dimensions across which a wavefront travels is the simplest solution. For example, if only the second dimension of the arrays in the Tomcatv code fragment in Figure 1 is distributed, the entire computation is parallel. This is not a general solution, for other components of the computation may prefer different distributions, perhaps due to surface-to-volume issues or wavefronts traveling along orthogonal dimensions. If we assume that any dimension of each array may be distributed, we must use pipelining to exploit parallelism in wavefront computations.

* This research was supported in part by DARPA Grant F30602-97-1-0152, NSF Grant CCR-9710284 and the Intel Corporation.

```
    DO 100 i = 2 , n-1
       DO 100 j = 2 , n-2
          r=aa(j,i)*d(j-1,i)
          d(j,i)=1.0/(dd(j,i)-aa(j-1,i)*r)
          rx(j,i)=rx(j,i)-rx(j-1,i)*r
          ry(j,i)=ry(j,i)-ry(j-1,i)*r
100 CONTINUE
```

(a)

```
    DO 100 j = 2 , n-2
       r(2:n-1)=aa(j,2:n-1)*d(j-1,2:n-1)
       d(j,2:n-1)=1.0/(dd(j,2:n-1)-aa(j-1,2:n-1)*r(2:n-1))
       rx(j,2:n-1)=rx(j,2:n-1)-rx(j-1,2:n-1)*r(2:n-1)
       ry(j,2:n-1)=ry(j,2:n-1)-ry(j-1,2:n-1)*r(2:n-1)
100 CONTINUE
```

(b)

Fig. 1. Fortran 77 (a) and Fortran 90 (b) wavefront code fragments from SPECfp92 Tomcatv benchmark.

There are three approaches to pipelining wavefront codes: (i) the *explicit* approach requires the programmer to write an explicitly parallel program (*e.g.*, Fortran 77 plus MPI [18]) exploiting pipelining, (ii) the *optimization-based* approach permits the programmer to write a sequential representation of the wavefront computation, from which the compiler produces a pipelined implementation, and (iii) the *language-based* approach provides the programmer with language-level abstractions that permit the unambiguous representation of wavefront computations that are candidates for pipelining.

The explicit approach is potentially the most efficient, because the programmer has complete control over all performance critical details. This control comes at the cost of added complexity that the programmer must manage. For example, the core of the ASCI SWEEP3D benchmark is 626 lines of code, only 179 of which are fundamental to the computation. The remainder are devoted to tiling, buffer management, and communication; different dimensions of the 4-dimensional problem are treated asymmetrically, despite problem-level symmetry, obscuring the true logic of the computation. Furthermore, the explicit approach will probably not exhibit portable performance, because the pipelined computation may be highly tuned to a particular machine.

The optimization-based approach appears to be the simplest for programmers, as they may exploit a familiar sequential representation of the computation. Researchers have described compiler techniques by which pipelined code may be generated from sequential programs [8, 17]. Unfortunately, significant performance will be lost if a compiler does not perform this optimization. As a result, the optimization-based approach does not have portable performance either, because different compilers may or may not implement the optimization. Even if all compilers do perform the transformation, there is a question of how well they perform it. Programmers may have to write code to a particular idiom that a particular compiler recognizes and optimizes, again sacrificing portability. Programmers cannot be certain that pipelined code will be generated, thus they lack complete information to make informed algorithmic decisions. In any case, we are aware of only one commercial High Performance Fortran (HPF) [7, 11] compiler that even attempts to pipeline wavefront codes, and there are many circumstances under which it is unsuccessful [13].

The language-based approach provides a simple representation of the wavefront computation, yet unambiguously identifies the opportunity for pipelining to both the programmer and the compiler. The programmer can be certain that the compiler will generate fully parallel pipelined code. In the best case, all three approaches will result in comparable parallel performance; but the programmer only has guarantees for the explicit and language-based approaches, while the optimization-based approach requires faith in the compiler to perform the transformation. On the other hand, to exploit the language-based approach, programmers must learn new language concepts. In this paper, we introduce two modest extensions to array languages that provide language-level support for pipelining wavefront computations. The extensions have no impact on the rest of the language (*i.e.*, existing programs need not change and do not suffer performance degradation). Though our extension can be applied to any array language, such as Fortran 90 [1] or HPF [7, 11], we describe it in terms of the ZPL parallel array language [19].

This paper is organized as follows. In the next section, we describe our array language extension to support wavefront computations in ZPL. Sections 3 and 4 describe its implementation and parallelization, respectively. Performance data is presented in Section 5, and conclusions and future work are given in the final section.

2 Array Language Support for Wavefront Computation

This section describes array language support for wavefront computations in the context of the ZPL parallel array language [19]. Previous studies demonstrate that the ZPL compiler is competitive with hand-coded C with MPI [3] and that it generally outperforms HPF [14]. The compiler is publicly available for most modern parallel and sequential platforms [22]. The language is in active use by scientists in fields such as astronomy, civil engineering, biological statistics, mathematics, oceanography, and theoretical physics. Section 2.1 gives a very brief summary of the ZPL language, only describing the features of the language immediately relevant to this paper. Section 2.2 introduces our language extension.

2.1 Brief ZPL Language Summary

ZPL is a data parallel array programming language. It supports all the usual scalar data types (*e.g.*, `integer`, `float`, `char`), operators (*e.g.*, math, logical, bit-wise), and control structures (*e.g.*, `if`, `for`, `while`, function calls). As an array language, it also offers array data types and operators. ZPL is distinguished from other array languages by its use of *regions* [4]. A region represents an index set and may precede a statement, specifying the extent of the array references within its dynamic scope. The region is said to *cover* statements of the same rank within this scope. By factoring the indices that participate in the computation into the region, explicit array indexing (*i.e.*, slice notation) is eliminated. For example, the following Fortran 90 (slice-based) and ZPL (region-based) array statements are equivalent.

```
for j := 2 to n-2 do
[j,2..n-1] begin
             r=aa*d@north;
             d=1.0/(dd-aa@north*r);
             rx=rx-rx@north*r;
             ry=ry-ry@north*r;
           end;
end;
```

(a)

```
[2..n-2,2..n-1] scan
                  r=aa*d'@north;
                  d=1.0/(dd-aa@north*r);
                  rx=rx-rx'@north*r;
                  ry=ry-ry'@north*r;
                end;
```

(b)

Fig. 2. ZPL representations of the Tomcatv code fragment from Figure 1. (a) Using an explicit loop to express the wavefront. (b) Using a scan block and the prime operator. Arrays `r`, `aa`, `d`, `dd`, `rx` and `ry` are all $n \times n$.

```
a(n/2:n,n/2:n) = b(n/2:n,n/2:n) + c(n/2:n,n/2:n)       [n/2..n,n/2..n] a = b + c;
```

Regions can be named and used symbolically to further improve readability and conciseness. When all array references in a statement do not refer to exactly the same set of indices, array operators are applied to individual references, selecting elements from the operands according to some function of the covering region's indices. ZPL provides a number of array operators (*e.g.*, shifts, reductions, parallel prefix operations, broadcasts, general permutations), but for this discussion, we will only discuss the shift operator. The shift operator, represented by the `@` symbol, shifts the indices of the covering region by some offset vector, called a *direction*, to determine the indices of its argument array that are involved in the computation. For example, the following Fortran 90 and ZPL statements perform the same four point stencil computation from the Jacobi Iteration. Let the directions `north`, `south`, `west`, and `east` represent the programmer defined vectors $(-1,0)$, $(1,0)$, $(0,-1)$, and $(0,1)$, respectively.

```
a(2:n+1,2:n+1) = (b(1:n,2:n+1)+b(3:n+2,2:n+1)+b(2:n+1,1:n)+b(2:n+1,3:n+2))/4.0
       [2..n+1,2..n+1] a := (b@north + b@south + b@west + b@east) / 4.0;
```

Figure 2(a) contains a ZPL code fragment representing the same computation as the Fortran 90 Tomcatv code fragment in Figure 1(b). The use of regions improves code clarity and compactness. Note that though the scalar variable `r` is promoted to an array in the array codes, we have previously demonstrated compiler techniques by which this overhead may be eliminated via array contraction [12].

2.2 Wavefront Computation in ZPL

Semantics. Array language semantics dictate that the right-hand side of an array statement must be evaluated before the result is assigned to the left-hand side. As a result, the compiler will not generate a loop that carries a non-lexically forward true data dependence (*i.e.*, a dependence from a statement to its self or a preceding statement). For example, the ZPL statement in Figure 3(a) is implemented by the loop nest in 3(b). The compiler determines that the *i*-loop must iterate from high to low indices in order to

```
[2..n,1..n] a := 2 * a@north;
```

(a)

for $i \leftarrow n$ downto 2 **do**
  for $j \leftarrow 1$ to n **do**
    $a_{i,j} \leftarrow 2 * a_{i-1,j}$

(b)

$$a = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{matrix}$$

(c)

```
[2..n,1..n] a := 2 * a'@north;
```

(d)

for $i \leftarrow 2$ to n **do**
  for $j \leftarrow 1$ to n **do**
    $a_{i,j} \leftarrow 2 * a_{i-1,j}$

(e)

$$a = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 \\ 8 & 8 & 8 & 8 & 8 \\ 16 & 16 & 16 & 16 & 16 \end{matrix}$$

(f)

Fig. 3. ZPL array statements (a and d) and the corresponding loop nests (b and e) that implement them. The arrays in (c and f) illustrate the result of the computations if array `a` initially contains all 1s.

ensure that the loop does not carry a true data dependence. If array `a` contains all 1s before the statement in 3(a) executes, it will have the values in Figure 3(c) afterward.

In wavefront computations, the programmer wants the compiler to generate a loop nest with non-lexically forward loop carried true data dependences. We introduce a new operator, called the *prime* operator, that allows a programmer to reference values written in previous iterations of the loop nest that implement the statement containing the primed reference. For example, the ZPL statement in Figure 3(d) is implemented by the loop nest in 3(e). In this case, the compiler must ensure that a loop carried true data dependence exists due to array `a`, thus the *i*-loop iterates from low to high indices. If array `a` contains all 1s before the statement in 3(d) executes, it will have the values in Figure 3(f) afterward. In general, the directions on the primed array references define the orientation of the wavefront.

The prime operator alone cannot represent wavefronts such as the Tomcatv code fragment in Figure 2(a), because it only permits loop carried true dependences from a statement to itself. We introduce a new compound statement, called a *scan block*, to allow multiple statements to participate in a wavefront computation. Primed array references in a scan block refer to values written by any statement in the block, not just the statement that contains it. For example, the ZPL code fragment in Figure 2(b) uses a scan block and the prime operator to realize the computation in Figure 2(a) without an explicit loop. The array reference `d'@north` refers to values from the previous iteration of the loop that iterates over the first dimension. Thus the primed `@north` references imply a wavefront that travels from north to south. Just as in existing array languages, a non-primed reference refers to values written by lexically preceding statements, within or outside the scan block.

The scan blocks we have looked at thus far contain only cardinal directions (*i.e.*, directions in which only one dimension is nonzero, such as north, south, east and west). When non-cardinal directions or combinations of orthogonal cardinal directions appear with primed references, the character of the wavefront is less obvious. Below, we describe how programmers may interpret these cases.

The notation may at first appear awkward. It is important to note, however, that experienced ZPL programmers are already well accustomed to manipulating arrays atomically and shifting them with the `@`-operator. They must only learn the prime operator, which is motivated by mathematical convention where successor values are primed. In the same vein, array languages such as Fortran 90 can be extended to include the prime operator.

Assumptions and Definitions. At this point, we give several assumptions and definitions that will be exploited in the subsequent discussion. We assume that all dimensions of each array may be distributed, and the final decision is deferred until application startup time. In addition, we assume that the dimension(s) along which the wavefront travel, the wavefront dimension(s), are decided at compile time. Define function f, where i and j are integers, as follows.

$$f(i,j) = \begin{cases} 0 \;\text{ if } i = j = 0 \\ \pm \text{ if } ij < 0 \\ + \text{ if } ij \geq 0 \text{ and } (i > 0 \text{ or } j > 0) \\ - \text{ if } ij \geq 0 \text{ and } (i < 0 \text{ or } j < 0) \end{cases}$$

Given two directions, $u = (u_1, \ldots, u_d)$ and $v = (v_1, \ldots, v_d)$, of size d we construct a size d *wavefront summary vector*[1] (WSV), $w = (w_1, \ldots, w_d)$, by letting $w_i = f(u_i, v_i)$ for $1 \leq i \leq d$. In a similar manner, all of the directions that appear with primed array references may be considered to form a single wavefront summary vector. We say that a wavefront summary vector is *simple* if none of its components are $\pm$. For example, $\text{WSV}(\{(-1,0),(-2,0)\}) = (-,0)$, $\text{WSV}(\{(-1,0),(-2,0),(-1,2)\}) = (-,+)$, $\text{WSV}(\{(-1,0),(0,-1)\}) = (-,-)$, and $\text{WSV}(\{(-1,0),(1,-2)\}) = (\pm,-)$. All but the final example are simple.

Legality. There are a number of statically checked legality conditions. (i) Primed arrays in a scan block must also be defined in the block; (ii) the directions on primed references may not over-constrain the wavefront, as discussed below; (iii) all statements in a scan block must have the same rank (*i.e.*, they are implemented by a loop nest of the same depth)—this precludes the inclusion of scalar assignment in a scan block; (iv) all statements in a scan block must be covered by the same region; and (v) parallel operators' operands other than the shift operator may not be primed; this is essential because array operators are pulled out of the scan block during compilation.

An over-constrained scan block is one for which a loop nest can not be created that respects the dependences from the shifted array references. For example, primed `@north` and `@south` references over-constrain the scan block because they imply both north-to-south and south-to-north wavefronts, which are contradictory. In general, the programmer constructs a wavefront summary vector from the directions used with primed array references in order to decided whether the scan block is over-constrained.

[1] The observant reader will recognize many similarities in this presentation to standard data dependence properties and algorithms. We have avoided a technical presentation in order to streamline the programmer's view.

Simple wavefront summary vectors, the common case, are always legal, for a wavefront may travel along any non-zero dimension, always referring to values “behind it.”

Wavefront Dimensions and Parallelism. For performance reasons programmers may wish to determine along which dimensions of the data space wavefronts travel, called *wavefront dimensions*, so that they may understand which dimensions benefit from pipelined parallelism. The dependences do not always fully constrain the orientation of the wavefront, so we give a set of simple rules to be used by the programmer to approximate wavefront dimensions. Again, the programmer examines the wavefront summary vector and considers three cases: (i) the WSV contains at least one 0 entry, (ii) the WSV contains no 0 entries and at least one $\pm$ entry, and (iii) the WSV contains only $+$ and $-$ entries. In case (i), all of the dimensions associated with $+$ or $-$ entries benefit from pipeline parallelism, and all the 0 entry dimensions are completely parallel. In case (ii), all but the $\pm$ entries benefit from pipelined parallelism. In case (iii), all but the leftmost entry benefits from pipelined parallelism. The leftmost entry is arbitrarily selected to minimize the impact of pipelining on cache performance.

Examples. We will use a single code fragment with different direction instantiations to illustrate how programmers may reason about their wavefront computations.

```
a := (a'@d1 + a'@d2)/2.0;
```

Example 1: Let `d1=d2=(-1,0)`. The WSV is $(-,0)$, which is simple, so the wavefront is legal (*i.e.*, not over-constrained). The first dimension is the wavefront dimension, and the second dimension is completely parallel.
Example 2: Let `d1=(-1,0)` and `d2=(0,-1)`. The WSV is $(-,-)$, which is simple, so the wavefront is legal. The wavefront could legally travel along either the first or second dimension, but we have defined it to travel along the second. There will be pipelined parallelism in the second dimension, but the first will be serialized.
Example 3: Let `d1=(-1,0)` and `d2=(1,1)`. The WSV is $(\pm,+)$, which is not simple. Nevertheless the wavefront is legal because there exists a loop nest that respects the dependences. The second dimension is the wavefront dimension.
Example 4: Let `d1=(0,-1)` and `d2=(0,1)`. The wavefront summary vector is $(0,\pm)$, which is not simple. The wavefront is not legal because no loop nest can respect the dependence is the second dimension. The scan block is over-constrained and the compiler will flag it as such.

Summary. We have presented a simple array language extension that permits the expression of loop carried true data dependences, for use in pipelined wavefront computations. It is simple for programmers to reason about the semantics, legality and parallel implications of their wavefront code. In fact, programmers need not actually compute wavefront summary vectors, for they will normally be trivial. For example, only the direction `north=(-1,0)` appears in the Tomcatv code fragment of Figure 2(b). This trivially begets the WSV $(-,0)$, indicating that the second dimension is completely parallel and the first is the wavefront dimension.

Contrast this with an optimization-based approach, where the programmer must be aware of the compiler's optimization strategy in order to reason about a code's potential parallel performance. Without this knowledge, the programmer is ill-equipped to make design decisions. For example, suppose a programmer writes a code that performs both north-south and east-west wavefronts. The programmer may opt to distribute only one dimension and perform a transposition between each north-south and east-west wavefront, eliminating the need for pipelining. This may be much slower than a fully pipelined solution, guaranteed by our language-level approach.

3 Implementation

This section describes our approach to implementing primed array references and scan blocks in the ZPL compiler. Below we describe how loop structure is determined and naive communication is generated.

The ZPL compiler identifies groups of statements that will be implemented as a single loop nest, essentially performing loop fusion. The data dependences (true, anti and output) that exist between these statements determine the structure of the resulting loop nest (which loop iterates over each dimension of the data space and in what direction). The ZPL and scalar code fragment in Figure 3(a) illustrates this. Notice that the *i*-loop iterates from high to low indices in order to preserve the loop carried anti-dependence from the statement to itself.

3.1 Deriving Loop Structure

In previous work we have defined *unconstrained distance vectors* to represent array-level data dependences, and we have presented an algorithm to decide loop structure given a set of intra-loop unconstrained distance vectors [12]. Traditional distance vectors are inappropriate for use in this context because they are derived from loop nests, which are not created until after our transformations have been performed. Because array statements are implemented with a loop nest in which a single loop iterates over the same dimension of all arrays in its body, we can characterize dependences by dimensions of the array rather than dimensions of the iteration space. Unconstrained distance vectors are more abstract than traditional (constrained) distance vectors because they separate loop structure from dependence representation. Though unconstrained distance vectors are not fully general, they can represent any dependence that may appear in a scan block.

The prime operator transforms what an array language would otherwise interpret as an anti-dependence into a true dependence. In order to represent this, the unconstrained distance vectors associated with primed array references are simply negated. The algorithm for finding loop structure is unchanged.

3.2 Communication Generation

Next, we consider naive communication generation for scan blocks. We assume that all parallel operators except shift are pulled out of scan blocks and assigned to temporary

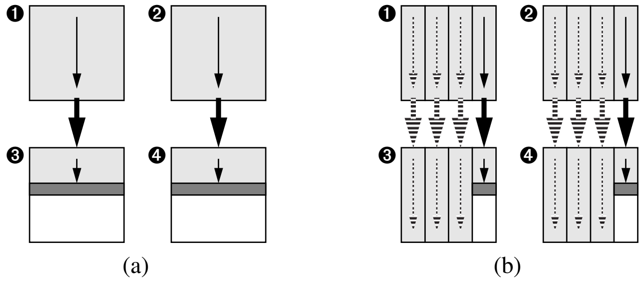


Fig. 4. Illustration of the data movement and parallelism characteristics of wavefront computations with (a) naive and (b) pipelined communication.

arrays. Furthermore, we assume that all arrays in a scan block are aligned and block distributed in each dimension, so communication is only required for the shift operator. This last assumption is the basis of ZPL's *WYSIWYG performance model* [2]. There are obvious extensions for cyclic and block-cyclic distributions. Each processor blocks, waiting to receive all the data it needs to compute its portion of the scan block. When all the data is received by a processor, it computes its portion of the scan block, and sends the necessary data on to the next processor(s) in the wave. The communication has the effect of serializing all the computation along the direction of the wavefront. Figure 4(a) illustrates this interprocessor communication. Though this naive implementation does not exploit any parallelism along the wavefront dimension, the next section describes and analyzes a parallel implementation exploiting pipelining.

4 Parallelization by Pipelining

In the implementation described above, a processor finishes computing on its entire portion of a scan block before data is sent on to later processors in the wavefront computation. As a result, there is no parallelism along the *wavefront dimension*—the dimension along which the wavefront travels. Suppose a north-to-south wavefront computation is performed on an $n \times n$ array block distributed across a 2×2 processor mesh as in Figure 4(a). Processors 3 and 4 must wait for processors 1 and 2 to compute their $\frac{n^2}{4}$ elements before they may proceed. Furthermore, processors 1 and 2 may then need to wait for the others to complete.

Alternatively, the wavefront computation may be *pipelined* in order to exploit parallelism [8, 15, 17, 21]. Specifically, a processor may compute over a *block* of its portion of the data space, transmit some of the data needed by subsequent processors, then continue to execute its next block. The benefit of this approach is that it allows multiple processors to become involved in the computation as soon as possible, greatly improving parallelism. Figure 4(b) illustrates this. Processors 3 and 4 only need to wait long enough for processors 1 and 2 to compute a single block ($\frac{n^2}{4} \times \frac{1}{4} = \frac{n^2}{16}$ elements) each. They can then immediately begin computing blocks of their portions of the scan block.

Smaller blocks increase parallelism at the expense of sending more messages. Several researchers have considered the problem of weighing this tradeoff in order to find the optimal block size. Hiranandani *et al.* developed a model that assumes constant cost communication [9], while Ohta *et al.* model communication as a linear function of message size [16]. In other words, the cost of transmitting a message of n words is given by $\alpha+\beta n$, where α is the message startup cost and β is the per-word communication cost. We present an analysis using this model to demonstrate its improved accuracy versus the constant cost communication model.

Assume that a wavefront is moving along the first dimension of a 2-dimensional $n \times n$ data space, as in the code of Figure 2(b). Let the data space be block distributed across p processors in the first dimension; for simplicity we do not distribute the second dimension. Let α and β be the startup and per-element costs of communication, respectively, and b be the block size. Also we assume that all times are normalized to the cost of computing a single element in the data space. The total computation and communication times of a pipelined implementation are given below.

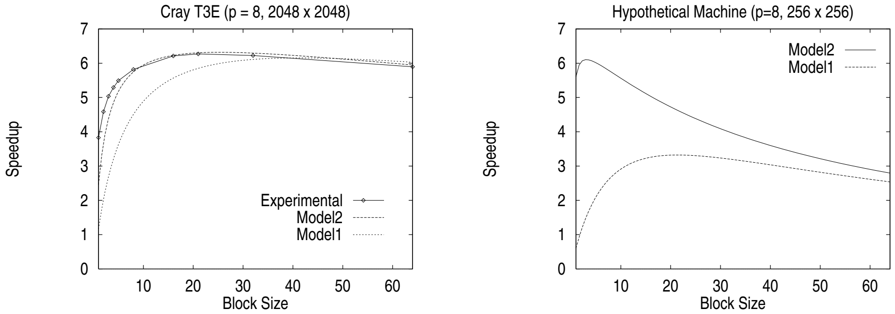
$$T_{\text{comp}}^{\text{pipe}} = \frac{nb}{p}(p-1) + \frac{n^2}{p} \qquad\qquad T_{\text{comm}}^{\text{pipe}} = (\alpha+\beta b)(\frac{n}{b}+p-2)$$

The first term of $T_{\text{comp}}^{\text{pipe}}$ gives the time to compute $p-1$ blocks of size $\frac{nb}{p}$, at which point the last processor may begin computing. The second term gives the time for the last processor to compute on its $\frac{n^2}{p}$ elements. The first factor of $T_{\text{comm}}^{\text{pipe}}$ is the cost of transmitting each b element message. The second factor gives the number of messages on the critical path. A total of $p-1$ messages are required before the last processor has received any data, and then the last processor receives another $\frac{n}{b}-1$ messages. In order to find the value of b that minimizes this time, we differentiate the sum of $T_{\text{comp}}^{\text{pipe}}$ and $T_{\text{comm}}^{\text{pipe}}$ with respect to b, let this equation equal 0 and solve for b.

$$-\frac{\alpha n}{b^2} + \beta(p-2) + \frac{n(p-1)}{p} = 0 \qquad \Longrightarrow \qquad b = \sqrt{\frac{\alpha n p}{(p\beta+n)(p-1)}} \approx \sqrt{\frac{\alpha n}{p\beta+n}} \qquad (1)$$

This approximate equation for b tells us that as α grows, the optimal b grows, because a larger startup communication cost must be offset by a reduction in the number of messages. As β grows, the optimal b decreases, because a larger per-element communication cost decreases the relative per-message startup cost. As p grows, the optimal b decreases, because there are more processor to keep busy. As n grows, the optimal b becomes less sensitive to the relative values of α, β, and p. Equation (1) reduces to the constant communication cost equation of Hiranandani *et al.* when we let $\beta = 0$ (*i.e.*, $b = \sqrt{\alpha}$). This simplified model gives no insight into the relative importance and roles of α, β, n, and p.

In order to assess the value of the added accuracy of modeling communication costs as a function of message size, we compare speedup derived from the two approaches to empirical data from the Cray T3E in Figure 5(a). *Model1* assumes that $\beta = 0$ (*i.e.*, like Hiranandani's model), and *Model2* is the more general model described above. First, observe that *Model2* more closely tracks the observed speedup. *Model1* predicts that $b = 39$ is the optimal block size, while *Model2* predicts $b = 23$, which is in fact better.



(a) (b)

Fig. 5. Model evaluations. *Model1* and *Model2* are identical except that the former assumes that $\beta = 0$, *i.e.*, it ignores the per-element communication cost. (a) Modeled versus experimental speedup due to pipelining of Tomcatv wavefront computation. (b) A demonstration of the value of modeling the per-word cost (β) of communication.

For this particular problem, n is relatively large making speedup not especially sensitive to the exact value of b. If we assume large problem sizes, the assumption that $\beta = 0$ is not an unreasonable one. For smaller problem sizes *Model1* is ineffective because the relative value of α and β becomes increasingly more important. This is particularly a problem on modern machines such as the Cray T3E on which β dominates communication costs. Figure 5(b) illustrates this problem with *Model1*. We use hypothetical values for α and β to illustrate a worst case scenario, so experimental data is not included. In this case, *Model1* does not accurately reflect speedup, and it suggests that the optimal block size is $b = 20$ versus $b = 3$ in *Model2*. We can expect the speedup with a block size of 20 versus 3 to be considerably less. The situation is even worse for larger numbers of processors.

5 Performance Evaluation

In this section we demonstrate the potential performance benefits of providing scan blocks in an array language. We conduct experiments on the Cray T3E and the SGI PowerChallenge using the Tomcatv and SIMPLE [5] benchmarks. For each experiment, we consider the program as a whole, and we consider two components of each that contain a single wavefront computation. Our extensions drastically improve the performance of the two wavefront portions of code in each benchmark, which results in significant overall performance enhancement. We use the following compilers in these experiments: Portland Group, Inc. pghpf version 2.4-4, Cray cf90 version 3.0.2.1, and University of Washington ZPL 1.14.3.

5.1 Cache Benefits

Before we evaluate the parallel performance of wavefront codes, we will examine cache performance implications on a single processor. Consider the code fragment without

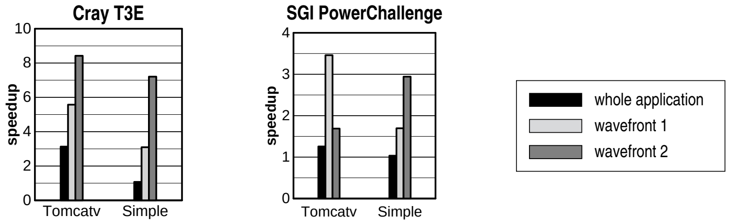


Fig. 6. Potential uniprocessor speedup due to scan blocks from improved cache behavior.

scan blocks in Figure 2(a), and assume the arrays are allocated in column-major-order. For the code to have acceptable cache behavior, the compiler must implement the four statements with a single loop nest and then interchange the compiler generated loop with the user loop. After these transformations, the inner loop iterates over the elements of each column, exploiting spatial locality. Despite the fact that loop interchange is a well understood transformation, there are certain circumstances where it fails. For example, for these codes, at the default optimization level (*-O1*) the pghpf compiler produces Fortran 77 code that the back end compiler is unable to optimize. The Cray Fortran 90 compiler, on the other hand, is successful. The programmer cannot be certain one way or the other.

With this in mind, we experimentally compare the performance of array language codes with and without our language support for wavefront computations. Figure 6 graphs the speedup of the former over the latter. These experiments are on a single node of each machine, so the speedup is entirely due to caching effects. On the Cray T3E, the wavefront computations alone (the two grey bars) speedup by up to $8.5\times$, resulting in an overall speedup (black bars) of $3\times$ for Tomcatv and 7% for SIMPLE. Tomcatv experiences such a large speedup, because the wavefront computations represent a significant portion of the program's total execution time. The SGI PowerChallenge graph has similar characteristics except that the speedups are more modest (up to $4\times$). This is because the PowerChallenge has a much slower processor than the T3E, thus the relative cost of a cache miss is less, so performance is less sensitive to cache behavior than on the T3E.

5.2 Parallelization Benefits

Performance enhancements like those presented below have also been demonstrated for optimization-based approaches to parallelizing wavefront computations [9]. When the optimization is successful, it can produce efficient parallel code. The data below gives a sense for the performance that may be lost if the optimization is not performed, either because its analysis is thwarted by an extraneous dependence or it is not implemented in the compiler. The language-based approach has the advantage that the programmer can be certain that the compiler is aware of the opportunity for pipelining, because the phrasing of the computation is unambiguously parallel. We are aware of no commercial HPF compilers that pipeline wavefront computations.

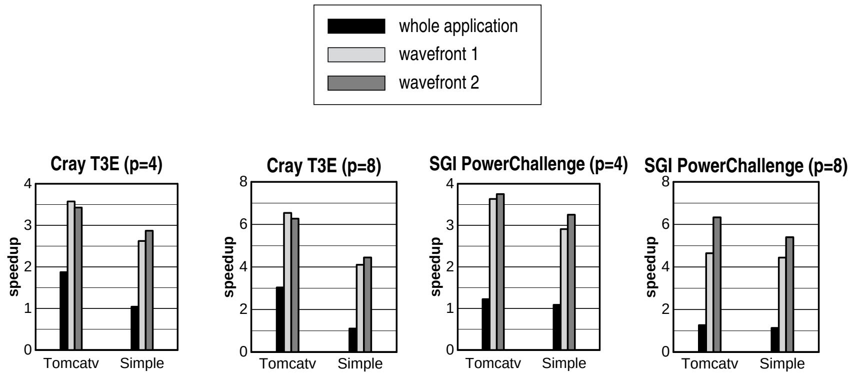


Fig. 7. Speedup of pipelined parallel codes versus nonpipelined codes. All arrays are distributed entirely across the dimension along which the wavefront travels.

We are in the process of implementing the parallelization strategy described in Section 4 in the ZPL compiler. For this study, we have performed the pipelining transformations by hand on Tomcatv and SIMPLE to assess its impact. Figure 7 presents speedup data due to pipelining alone. Speedup is computed against a fully parallel version of the code without pipelining, thus the bars representing whole program speedup (black bars) are speedup beyond an already highly parallel code. The bars representing speedup of the wavefront computations (grey bars) are serial without pipelining, so the baseline in their case does not benefit from parallelism. We would like the grey bars to achieve speedup as close to the number of processors as possible. In all cases the speedup of the wavefront segments approaches the number of processors, and the overall program improvements are large in several cases (up to $3\times$ speedup). The smallest overall performance improvements are still greater than 5 to 8%. Though the absolute speedup improves as the number of processors increases, the efficiency decreases. This is because we have kept the problem size constant, so the relative cost of communication increases with the number of processors.

6 Conclusion

Wavefront computations are becoming increasingly important and complex. Though previous work has developed techniques by which compilers may produce efficient pipelined code from serial source code, the programmer is left to wonder whether the optimization was successful or whether the optimization is even implemented in a particular compiler. In contrast, we have extended array languages to provide language-level support for wavefront codes, thus unambiguously exposing the opportunity for pipelining to both the programmer and the compiler. The extensions do not impact the rest of the language, and they permit programmers to simply reason about their codes' legality and potential performance. We have given a simple analysis that gives insight into the roles of machine parameters, problem size, and processor configuration in determining the optimal block size for pipelining. In addition, we have presented experimental data for the Tomcatv and SIMPLE benchmarks on the Cray T3E and SGI PowerChallenge demonstrating the potential performance lost when pipelining is not performed.

Currently, we are in the process of fully implementing these language extensions in the ZPL compiler. Because the optimal block size is a function of non-static parameters such as problem size and computation cost, we will develop dynamic techniques for calculating it. We will investigate the quality of block size selection using only static and profile information. We will also develop a benchmark suite of wavefront computations in order to evaluate our design and implementation and investigate their properties, such as dynamism of optimal block size.

Acknowledgments. We thank Sung-Eung Choi and Samuel Guyer for their comments on drafts of this paper. This research was supported by a grant of HPC time from the Arctic Region Supercomputing Center.

References

1. Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, New York, NY, 1992.
2. Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *Third IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61, March 1998.
3. Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, and Calvin Lin. The case for high-level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–85, July–September 1998.
4. Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *ACM SIGAPL/SIGPLAN International Conference on Array Programming Languages*, pages 41–49, August 1999.
5. W. Crowley, C. P. Hendrickson, and T. I. Luby. The SIMPLE code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.
6. Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, pages 836–844, 1986.
7. High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*. January 1997.
8. Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing '91*, pages 96–100, Albuquerque, NM, November 1991.
9. Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *International Conference on Supercomputing*, pages 1–14, Washington, DC, July 1992.
10. K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of three-dimensional discrete ordinates equations on a massively parallel machine. *Transactions of the American Nuclear Society*, 65:198–9, 1992.
11. Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., , and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, Massachusetts, 1993.

12. E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.
13. E Christopher Lewis and Lawrence Snyder. Pipelining wavefront computations: Experiences and performance. Submitted for publication, November 1999.
14. 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.
15. Ton A. Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1997.
16. Hiroshi Ohta, Tasuhiko Saito, Masahiro Kainaga, and Hiroyuki Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *International Conference on Supercomputing*, pages 270–9, Barcelona, Spain, 1995.
17. Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.
18. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 2nd edition, 1998.
19. Lawrence Snyder. *The ZPL Programmer's Guide*. The MIT Press, Cambridge, Massachusetts, 1999.
20. David Sundaram-Stukel and Mark K. Vernon. Predictive analysis of a wavefront application using LogGP. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.
21. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.
22. ZPL Project. ZPL project homepage. *http:/www.cs.washington.edu/research/zpl*.