To appear in the Proceedings of the 5th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), May 2000.

Pipelining Wavefront Computations: Experiences and Performance*

E Christopher Lewis and Lawrence Snyder

University of Washington
Department of Computer Science and Engineering
Box 352350, Seattle, WA 98195-2350 USA
{*echris,snyder*}*@cs.washington.edu*

Abstract. Wavefront computations are common in scientific applications. Although it is well understood how wavefronts are pipelined for parallel execution, the question remains: How are they best presented to the compiler for the effective generation of pipelined code? We address this question through a quantitative and qualitative study of three approaches to expressing pipelining: programmer implemented via message passing, compiler discovered via automatic parallelization, and programmer defined via explicit parallel language features for pipelining. This work is the first assessment of the efficacy of these approaches in solving wavefront computations, and in the process, we reveal surprising characteristics of commercial compilers. We also demonstrate that a parallel language-level solution simplifies development and consistently performs well.

1 Introduction

Wavefront computations are characterized by a data dependent flow of computation across a data space, as in Fig. 1. Wavefronts are common, and the scientific applications in which they appear demand parallel execution [9, 14]. Although the dependences they contain imply serialization, it is well known that wavefront computations admit efficient, parallel implementation via pipelining [5, 15], *i.e.*, processors only partially compute local data before sending data to dependent neighbors. The question of how wavefront computations are best presented to the compiler for the effective generation of pipelined parallel code remains open.

In this paper, we address this question in a study of three approaches for expressing wavefront computations: programmer implemented via message passing, compiler discovered via automatic parallelization, and programmer defined via explicit parallel language features for pipelining. The general parallel programming implications of these approaches are well known, but not in the context of wavefront computations. They are all potentially efficient, but each has a downside. Message passing programming requires considerable time to develop, debug, and tune. The benefits of automatic parallelization are only realized when a program is written in terms that the compiler is able to parallelize. And high-level parallel languages only benefit those who are willing to learn them. We assess these issues in the context of pipelining wavefront computations.

* This research was supported in part by DARPA Grant F30602-97-1-0152

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i-1,j))/2.0
   enddo
enddo
```

(a) WF/1D/VERT

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i,j-1))/2.0
   enddo
enddo
```

(b) WF/1D/HOR

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i-1,j)+
                a(i,j-1))/3.0
   enddo
enddo
```

(c) WF/2D

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i-1,j))/2.0
   enddo
enddo
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i,j-1))/2.0
   enddo
enddo
```

(d) WF/1D/BOTH

Fig. 1. Wavefront kernel computations.

We evaluate the three approaches by developing each of the four wavefront kernels in Fig. 1 on two dissimilar parallel machines (IBM SP-2 and Cray T3E). The kernels are representative of a large class of wavefronts (*e.g.*, those in SWEEP3D, SIMPLE, and Tomcatv), and they are sufficiently simple that they allow us to focus on the first order implications of their parallelization. We use the Message Passing Interface (MPI) [12] as an illustration of message passing, High Performance Fortran (HPF) [7][1] of automatic parallelization, and the ZPL parallel programming language [13] of language-level representation.

This work is the first quantitative and qualitative assessment of developing parallel wavefront computations by the three approaches. Furthermore, we compare the development experience and performance of these approaches via a common set of kernels. The evidence we gather both confirms widely held beliefs about these representations and challenges conventional wisdom. We find that a language-level representation is both simple to develop and consistently efficient. In addition, our study reveals surprising characteristics of commercial HPF compilers.

This paper is organized as follows. The next section describes the representations that we consider. Sect. 3 relates our experiences parallelizing wavefront computations, and Sect. 4 presents performance data for each representation. The final section gives conclusions.

[1] HPF is not strictly an automatically parallelized language, but it lacks intrinsic or annotational support for pipelining, relegating pipelining to an automatic parallelization/optimization task.

2 Representations: MPI, HPF, and ZPL

This section summarizes the three representations we consider. MPI and HPF are well known, so we only address ZPL in any detail.

Although often efficient, message passing—in this case MPI—programs are laborious to develop, for the programmer must manage every detail of parallel implementation. This is illustrated by the 626 line kernel of the ASCI SWEEP3D benchmark [1], only 179 lines of which are fundamental to the computation. The remainder manage the complexities of implementing pipelining via message passing. Furthermore, by obscuring the true logic of a program, complexity hinders maintenance and modification. Conceptually small changes may result in substantially different implementations.

An HPF program is a sequential Fortran 77/90 program annotated by the programmer to guide data distribution (via `DISTRIBUTE`) and parallelization (via `INDEPENDENT`) decisions [7]. The HPF standard does not include annotations to identify computations that may be pipelined, but Gupta *et al*. indicate that the IBM xlHPF compiler for the IBM SP-2 automatically recognizes and optimizes them [6]. Some forms of task-level pipelining are supported by HPF2, but no commercial compilers support the new standard. Furthermore, a representation of this form would look more like an MPI code, thus sacrificing the benefits of HPF.

ZPL is a data-parallel array programming language [13].[2] It supports all the usual scalar data types (*e.g.*, `integer` and `float`), operators (*e.g.*, math and logical), and control structures (*e.g.*, `if` and `while`). As an array language, it also offers array data types and operators. ZPL is distinguished from other array languages by its use of *regions* [3]. A region represents an index set and may precede a statement, specifying the extent of the array references within its dynamic scope. For example, the following Fortran 90 (slice-based) and ZPL (region-based) array statements are equivalent.

```
F90: a(n/2:n,n/2:n) = b(n/2:n,n/2:n) + c(n/2:n,n/2:n)
ZPL: [n/2..n,n/2..n] a = b + c;
```

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 enclosing region's indices. ZPL provides a number of array operators (*e.g.*, shifts, reductions, parallel prefix, broadcasts, and permutations), but this discussion requires only the shift operator, represented by the `@` symbol. It shifts the indices of the enclosing 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 ZPL statement performs a four point stencil computation. Let the directions `north`, `south`, `west`, and `east` represent the programmer defined vectors $(-1,0)$, $(1,0)$, $(0,-1)$, and $(0,1)$, respectively.

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

In a scalar language, wavefront computations are implemented by loop nests that contain non-lexically forward loop carried true data dependences. Traditional array language semantics do not allow the programmer to express such a dependence at the array

[2] This ZPL summary is sufficient for this discussion. Details appear in the literature [13, 4].

```
[2..m,2..n] a := (a+a'@north)/2.0;
```

(a)

```
[2..m,2..n] a := (a+a'@west)/2.0;
```

(b)

```
[2..m,2..n] a := (a+a'@west
                  +a'@north)/3.0;
```

(c)

```
[2..m,2..n] a := (a+a'@north)/2.0;
[2..m,2..n] a := (a+a'@west)/2.0;
```

(d)

Fig. 2. ZPL wavefront kernels corresponding to those in Fig. 1.

level, so ZPL provides the *prime* operator for this purpose. Primed array references refer to values written in previous iterations of the loop nest that implements it.[3] For example, the ZPL statements in Fig. 2 are semantically equivalent to the corresponding loop nests in Fig 1. The prime operator permits the array-level representation of arbitrary loop-carried flow data dependences, but here we only describe its use in wavefronts.

ZPL may further be distinguished from other parallel languages by its what-you-see-is-what-you-get (WYSIWYG) performance model [2]. The language shields programmers from most of the tedious details of parallel programming, yet the parallel implications of a code are readily apparent in the source text. Naturally, the prime operator—indicating that pipelined parallelism is available—supports this model [4].

3 Parallelization Experiences: MPI, HPF, and ZPL

In this section we describe our experiences writing and tuning wavefront computations.

3.1 MPI

A message passing implementation of pipelining is conceptually simple, but it is surprisingly complex in practice. As an illustration, the WF/2D kernel is 40 lines long, which is large when compared to the single loop nest that is represents. In addition, its development, debugging, and performance tuning consumed three hours, a long time for such a trivial computation. Naturally, with message passing even moderate computations will be slow to develop and lengthy.

Furthermore, despite the conceptual similarity between the four kernels, the four MPI implementations differ in significant ways, such as location of communication, allocation of ghost cells, and indexing. The structure of each code is closely tied to the distribution of data and the dependences that define the wavefront, so there is little code reuse between the four implementations. In addition, we are faced with the problem of finding the best tile size (*i.e.*, the granularity of the pipeline). In order to contain development time, we forgo a dynamic scheme [10] in favor of direct experimentation for each kernel on each machine. Naturally, the results will not extend to other machines and different problem sizes.

[3] This discussion excludes the mechanism for enlarging the scope of the primed reference.

3.2 HPF

HPF programmers need not manage per processor details and explicit communication. Nevertheless, they direct the compiler's parallelization via annotations. HPF lacks annotations to identify wavefront computations, so the compiler is solely responsible for recognizing and optimizing them from their scalar representations [8, 11]. We consider the Portland Group, Inc. PGHPF and IBM xlHPF compilers separately, below.

PGHPF. The HPF compiler from Portland Group, Inc. (PGHPF) does not perform pipelining. We determine this by examining the intermediate message passing Fortran code produced by the `-Mftn` compiler flag. The performance data will confirm this.

PGHPF strictly obeys the `INDEPENDENT` annotations, redistributing arrays before and after loop nests so that all the annotation specified parallelism is exploited. An implication of this is that parallel loops exploit parallelism—at the cost of data redistribution—even when the user specified data distribution precludes parallelism. In this way PGHPF extracts some parallelism from two of the kernels—as we will see in the next section—but it is not competitive with a pipelined implementation.

Another implication of strictly respecting annotations is that they must be placed very carefully. If an `INDEPENDENT` annotation is placed on the inner loop in WF/1D/HOR, the compiler will redistribute the array inside the `j` loop, resulting in performance three orders of magnitude worse than that of the loop nest in WF/1D/VERT. The programmer may interchange the two loops, making the outer loop `INDEPENDENT`, but the resulting array traversal will have poor cache performance.

While the loop nests in WF/1D/VERT, HOR, and BOTH can use redistribution to exploit parallelism, that in WF/2D can not, for it contains dependence in both dimensions. Only pipelining will extract parallelism from this code. We found that because the loop contains no `INDEPENDENT` annotations, every array element read is potentially transmitted in the inner loop. It appears that only the source and destination processors of each scalar communication block while the communication takes place, thus other processors are permitted to compute ahead, limited only by data dependences. This realizes a crude form of fine grain pipelining when arrays happen to be traversed in the right way. Despite this, the inner loop communication prevent this code from being competitive with a true pipelined implementation.

XLHPF. A published report indicates that IBM xlHPF performs pipelining [6]. The compiler does not provide an option for viewing the intermediate message passing code and the parallelization summary excludes this information, so we experimentally confirm that the compiler does indeed perform pipelining. Specifically, we observe that an HPF wavefront computation has single node performance comparable to the equivalent Fortran 77 program and that it achieves speedup beyond this for multiple processors.

Unlike PGHPF, xlHPF only exploits parallelism on `INDEPENDENT` loops that iterate over a distributed dimension. This fact and the pipelining optimization result in good parallel performance for all of the kernels. Despite this, we find that the pipelining optimization fails on even modestly more complex wavefront. For example, loops that iterate from high to low indices or contain non-perfectly nested loops are not pipelined. Certainly, they could be. But the lesson is that when optimizing arbitrary code, certain cases or idioms may easily be over looked. Conversely, a language-level solution makes explicit both the semantic and performance implications of a computation.

3.3 ZPL

The ZPL representations of the kernels are trivial to express (Fig. 2). It is apparent to both the programmer and the compiler how parallelism may be derived from them, and tuning was unnecessary.

4 Performance

We gather performance data on two parallel machines: a 272 processor Cray T3E-900 (450MHz DEC Alpha 21164 nodes) and 192 processor IBM SP-2 (160MHz Power2 Super Chip nodes). We use a number of compilers in this evaluation: on the T3E, we use the Cray CF90 Version 3.2.0.1 and Portland Group, Inc. PGHPF v2.4-4; on the SP-2, we use IBM xlf Fortran v4.1.0.6, IBM xlHPF v1.4, IBM xlc v3.1.4.0, and PGHPF v2.1. On both machines we use the University of Washington zc v1.15 ZPL compiler [16]. All compilers are used with the highest optimization level that guarantees the preservation of semantics.

We study four different representations: C+MPI, ZPL, xlHPF, PGHPF. The C+MPI code is a well tuned pipelined message-passing program. It represents the best that can be achieved on these machines using the C programming language. Because the xlHPF compiler is only available on the SP-2, we do not have results for it on the T3E.

For all the experiments, the `a` array is distributed across a dimension that gives rise to a loop carried dependence (*e.g.*, the first dimension in WF/1D/VERT) so as to isolate the impact of pipelining. Although it appears that WF/1D/VERT and WF/1D/HOR do not require pipelining (*i.e.*, there exists a distribution that permits complete parallel execution), these kernels may appear in a context that requires a different distribution.

We find that the single processor execution times for C+MPI and ZPL—which generates C code—are comparable to that of a sequential C program. Similarly, on a single processor, xlHPF and PGHPF match sequential Fortran. On the T3E, sequential C code typically executes in twice the time of comparable Fortran codes, while on the SP-2 this ratio varies with character of the kernel. Such disparities between C and Fortran implementations of the same computation are common. In any case, that single processor execution times match sequential languages (within each language domain) indicate observed scaling behavior is relative to an efficient baseline.

All the performance data is summarized by the graphs in Fig. 3. These graphs depict performance (*i.e.*, inverse execution time), so higher bars indicate faster execution. Furthermore, the performance is scaled relative to C+MPI. First, observe that the ZPL performance keeps pace with that of C+MPI. This indicates that ZPL is both performing as well and scaling as well as the hand coded program. At times the ZPL code even surpasses C+MPI, because it performs low level optimizations for more efficient array access. Consider the PGHPF performance. It is competitive on a single processor for the WF/1D/VERT and WF/1D/BOTH kernels, but it quickly trails off as the number of processors increases. This is because, PGHPF redistributes the data to achieve parallelism, which does not scale. The SP-2 exaggerates this effect, because its high communication costs outweigh the benefits of redistribution. Furthermore, for WF/1D/HOR and WF/2D significant communication appears in the inner loop, resulting in abysmal performance (the bars are not even visible!).

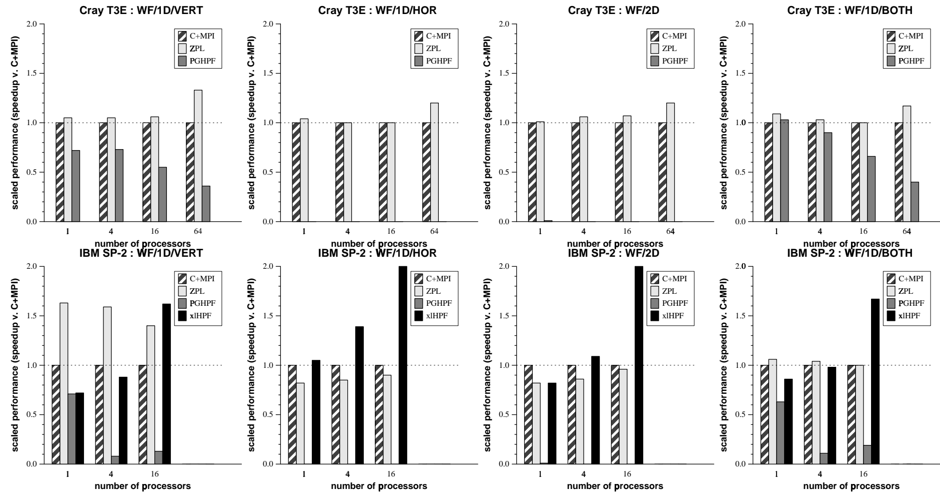

Fig. 3. Performance summary. Kernel names are from Fig. 1. Note that all PGHPF bars are present, but they are very small for WF/1D/HOR and WF/2D.

XlHPF is competitive with the C+MPI and ZPL, because it performs pipelining. The single processor bars highlight disparities in local computation performance. ZPL performs considerably better than any of the others for WF/1D/VERT. We hypothesize that the dependences in this kernel thwart proper array access optimization by the xl optimizer (used by both the Fortran and C compilers). The ZPL code does not suffer from this, because its compiler generates direct pointer references rather than using C arrays. When the C+MPI code is modified in this way, its performance matches ZPL. Conversely, ZPL is worse for WF/1D/HOR. Again, we believe this is an optimization issue. When the ZPL code is modified to use C arrays rather than pointer manipulation, it matches HPF. The summary is that when we ignore the differences that arise from using C versus Fortran, the C+MPI, xlHPF, and ZPL kernel performance are comparable. Nevertheless, as stated in the previous section, we found a number of wavefronts that even the xlHPF compiler failed to optimize.

5 Conclusion

We have evaluated the experience and performance of expressing wavefront computations by three different approaches: programmer implemented via message passing, compiler discovered via automatic parallelization, and programmer defined via explicit parallel language features for pipelining. Our study reveals that in developing wavefronts, each approach can produce an efficient solution, but at a cost. The message passing codes took considerably longer to develop and debug than the other approaches. The HPF codes did not reliably perform well. Although one compiler produced efficient code, the other was three orders of magnitude worse. Even the better compiler failed to pipeline some very simple cases. We find that the language-level approach embod-

ied in ZPL simplifies program development and results in good performance that is consistently achieved.

Acknowledgements. This research was supported in part by a grant of HPC time from the Arctic Region Supercomputing Center.

References

1. Accelerated Strategic Computing Initiative. ASCI SWEEP3D homepage. *http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/sweep3d_readme.html*.
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, 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.
4. Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Language support for pipelining wavefront computations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1999.
5. Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, pages 836–844, 1986.
6. Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF compiler for the IBM SP2. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference (CD-ROM)*, 1995.
7. High Performance Fortran Forum. *HPF 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. 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.
10. David K. Lowenthal and Michael James. Run-time selection of block size in pipelined parallel programs. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 82–7, 1999.
11. Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *ACM SIGPLAN PLDI '89*, pages 69–80, June 1989.
12. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 2nd edition, 1998.
13. Lawrence Snyder. *The ZPL Programmer's Guide*. The MIT Press, 1999.
14. 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.
15. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.
16. ZPL Project. ZPL project homepage. *http:/www.cs.washington.edu/research/zpl*.