Parallel Programming with Polaris

William Blume
Ramon Doallo
Rudolf Eigenmann
John Grout
Jay Hoeflinger
Thomas Lawrence
Jaejin Lee
David Padua
Yunheung Paek
Bill Pottenger
Lawrence Rauchwerger
Peng Tu
University of Illinois

Steady improvements in computer hardware in the past 40 years have led to dramatic increases in program speed. As we reach the technological limits of hardware improvement, we must rely on multiple processors to improve programming speed.

Parallel programming tools are limited, making effective parallel programming difficult and cumbersome. Compilers that translate conventional sequential programs into parallel form would liberate programmers from the complexities of explicit, machine-oriented parallel programming. Polaris, an experimental translator of conventional Fortran programs that target machines such as the Cray T3D, is the first step toward this goal.

POLARIS TECHNIQUES

The most important techniques implemented in Polaris resulted from a study of the effectiveness of commercial Fortran parallelizers.[1] We compiled the Perfect Benchmarks, a collection of conventional Fortran programs representing the typical workload of high-performance computers, for the Alliant FX/80, an eight-processor multiprocessor popular in the late 1980s.

For each program, we measured the quality of the parallelization by computing the speedup—the ratio of a program's sequential execution time to the execution time of the automatically parallelized version. With a few exceptions, the Alliant Fortran compiler failed to deliver any significant speedup for the majority of the programs.

The compiler failed to produce a speedup because it could not parallelize some of the most important loops in the Perfect Benchmarks. Programmers originally developed the parallelization module of the Alliant compiler for vectorization, then retrofitted it for parallelization. Vectorizers[2] focus primarily on innermost loops, while multiprocessor compilers focus on parallelizing outer loops.

Our study showed that extending the four most important analysis and transformation techniques traditionally used for vectorization leads to significant increases in speedup.

Dependence analysis

A loop can be transformed into parallel form if it contains no *cross-iteration dependencies*: The loop must not have two iterations that access the same memory location if either iteration changes the location's value. Dependence analysis techniques analyze every pair of references to the same array within a loop to check if their subscript expressions might produce the same value in two different iterations. To guarantee correct code,

0018-9162/96/$5.00 © 1996 IEEE

dependence analysis techniques must assume cross-iteration dependencies when they are unable to accurately analyze the subscripts.

We illustrate these ideas using the loop

```
   DO I = 1,N
R:   A(2*I) = ...
S:   ... = A(2*I)
T:   ... = A(2*I+1)
   END DO
```

The *equality test* determines the absence of cross-iteration dependence whenever the subscripts of two array references are identical, linear functions of the loop index. In the previous loop, the equality test will determine that cross-iteration dependence exists neither between two executions of statement R nor between R and S. (There cannot be a cross-iteration dependence between S and T because there is no assignment to array A in either statement.)

The *GCD test* equates the linear subscript expressions of two statements to see if they have an integer solution. If they don't, no cross-iteration dependency exists. To analyze the potential dependence between R and T, the GCD test considers the equation

$$2i = 2i' + 1$$

No integer solutions to the equation exist when the greatest common divisor of the coefficients (2, in this case) does not divide the independent term, 1.

Parallelizing compilers, including Polaris, apply a variety of these dependence tests in sequence, but most dependence tests only work on linear expressions. We found nonlinear subscript expressions (some generated by our transformations and the rest part of the original program) in the Perfect Benchmarks. To handle these, we expanded on earlier dependence tests and developed the *range test*,[3] which deals with nonconstant coefficients. The range test uses computer algebra and data range information extracted from the structure of the program to test for cross-iteration dependencies. The test successfully analyzes a number of complex access patterns that arise in real codes.

For example, in the fragment

```
DO I=1,N
 DO K=1,M
  A(M*I + K)=...
   =A(M*I + M)
 END DO
END DO
```

accesses to A are M elements apart in consecutive iterations of the I loop. The range test determines that there are no cross-iteration dependencies by noticing that all the elements read or written in each iteration fit within this M element-long separation.

Privatization

Temporary variables are often used inside loops to carry values between statements.

An example is variable T in the loop

```
DO I = 1,N
 T = A(I) +1
 B(I) = T**2
 C(I) = T + 1
END DO
```

This code contains a cross-iteration dependency because every iteration reads and writes the same temporary location, T. However, every iteration assigns to T before reading it, so we could replace the single copy of T with one copy for each iteration. The code would still produce the same result, but the cross-iteration dependence would be eliminated. We call such replaced temporary variables *loop private*.

Vectorizers identify only loop-private scalars. However, we found many cases in the Perfect Benchmarks where the temporary variables within multiply-nested loops were arrays. Identifying private arrays eliminated many apparent dependencies in outermost loops.

Polaris is an experimental translator of conventional Fortran programs that target machines such as the Cray T3D.

In Polaris, we implemented a privatization technique to deal with both scalars and arrays.[4] This algorithm proved to be substantially more complex than the traditional scalar privatization algorithm used for vectorization because the reading and writing of a single array may occur at multiple points within the loop and because the subscript expressions involved may be arbitrarily complex.

Our technique analyzes the subscript expressions of all array references, finding cases where all elements of an array are assigned on every iteration before they are used. Where this happens, privatization can occur.

Induction variable substitution

Induction variables have integer values and are incremented by constants on every loop iteration. An example is variable J in the loop

```
J = 0
DO I=1,N
   J =
   J + 2
   U(J)= ...
END DO
```

Induction variables present parallelization problems for two reasons: First, the compiler reads and writes induction variables on every iteration, making them a source of cross-iteration dependencies. Second, dependence tests cannot directly analyze a subscript expression involving induction variables unless their variation is expressed in terms of the loop indices.

The compiler must transform each occurrence of an induction variable, replacing it with an expression involving the loop index. For example, in the previous loop, the array reference could be replaced by U(2*I).

Many compilers transform only induction variables that can be expressed in terms of a single loop index. However, in the multiply-nested loops of real programs, induction variables can be incremented at several levels of nesting within a single loop. In Polaris[5] we have implemented tech-

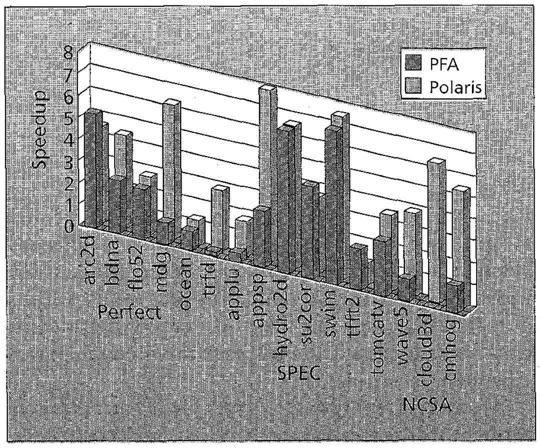

Figure 1. Speedup comparison between Silicon Graphics Power Fortran Analyzer and Polaris.

niques that produce closed form expressions in terms of several loop indices for these cases. For example, in

```
   J = 0
   DO I=1,N
     ...
S:   J = J + 1
     ...
     DO K = 1, I
     ...
T:      J = J + 1...
     END DO
   END DO
```

`I + I*(I-1)/2` may replace all occurrences of the induction variable after `S` in the outer loop, and `I + I*(I-1)/2 + K` may replace those after `T` within the inner loop. Once again, a traditional dependence test cannot analyze these nonlinear expressions, but the range test applied by Polaris can handle them.[5]

Reduction substitution

A programmer may reduce the information from an array by summing it in reduction variables. When this occurs, the compiler sees that pattern and realizes that it can parallelize the loop. Reduction variables change incrementally with each iteration (usually by added floating-point values), which causes cross-iteration dependencies in the loop. Only if the statements performing the increment access the reduction variable within the loop and the reduction operation is associative (may be grouped in any order), the loop can be parallelized. For example:

```
   DO I=1,N
     ...
     S:A(K(I))=A(K(I)) + B(I)...
   END DO
```

Polaris could place statement S within a *critical section* of the loop. This guarantees no interference between accesses to the same array element because only one processor at a time can enter a critical section—all other processors are locked out. Or it could create a copy of array A in every processor cooperating in the parallel execution of the loop, perform partial sums in the copies of A, then add the partial sums to form the final version of A.

Polaris takes into account the number of iterations, the overhead of critical section locks, and the size of array A when deciding which strategy to use.

Vectorizers consider only simple reductions, but more complex patterns occur in many programs. For example, the reduction variable could be an array, a loop could have multiple reduction statements, and the subscripts of reduction arrays could be array elements themselves. We incorporated advanced techniques into Polaris to handle such cases.

In addition to the four analysis techniques already discussed, Polaris applies *autoinlining*[6] and *interprocedural value propagation.* Autoinlining replaces a call to a subroutine with the code for that subroutine, when heuristics deem it profitable. This process places the code for the subroutines directly in the calling routine at the calling site, giving Polaris a chance to analyze it. IPVP finds cases in which a subroutine parameter takes on different values at different call sites. In these cases, Polaris makes a different copy of the subroutine—a *clone*—for each different value. Consequently, the compiler knows the value of the parameter inside a given clone, which can enable code optimizations.

POLARIS EFFECTIVENESS

Figure 1 shows the speedup comparison between Polaris and Silicon Graphics's Power Fortran Analyzer. As with the Alliant parallelizer, engineers originally developed the Power Fortran Analyzer as a vectorizer.

The 16 benchmark programs used for the analysis come from three different sources:

- arc2d, bdna, flo52, mdg, ocean, and trfd from the Perfect Benchmarks suite;
- applu, appsp, hydro2d, su2cor, swim, tfft2, tomcatv, and wave5 from the SPECfp95 Benchmark suite; and
- cmhog and cloud3d from the National Center for Supercomputing Applications collection of moderate- size programs (approximately 10,000 lines each) used in scientific research at NCSA.

We executed the programs (in real-time mode for timing accuracy) on eight processors of an SGI Challenge with 150-MHz R4400 processors at NCSA. Figure 1 shows that Polaris delivered substantially better speedups than the Power Fortran Analyzer in many cases. The Power Fortran Analyzer did produce better speedups than Polaris in three programs because it uses an elaborate code-generation strategy that includes loop transformations, such as loop interchanging, unrolling, and fusion. Applying these transformations to the right loops improves performance by decreasing overhead, enhancing locality, and facilitating the detection of instruction-level parallelism. However, the elaborate strategy decreases speedup for other programs.

To evaluate the effectiveness of autoinlining, IPVP, array privatization, range test, multiply-nested induction substitution, and advanced reduction substitution, we compiled each program six times, turning off a different technique each time. Then we compared those results with the program compiled with all techniques enabled.

The six compilations contain all that is new in Polaris with respect to the Alliant parallelizer, though Power Fortran Analyzer includes some of these capabilities. For example, Power Fortran Analyzer can substitute multiply-nested induction variables if the bounds of inner loops do not contain indices of outer loops.

Figure 2 presents the results of the six experiments conducted for each code. The height of the bar at the intersection (P,T) represents, in logarithmic scale, the percentage of the total number of loops in program P which became serialized by disabling technique T. The analysis of programs in the Perfect Benchmarks inspired these techniques, but Figure 2 shows that these techniques enable parallelization of loops from other programs in the collection, too.

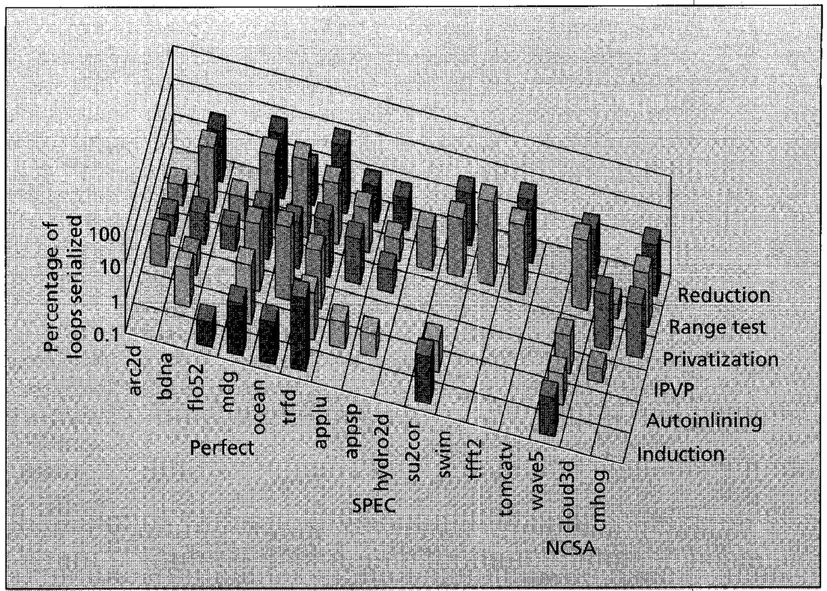

Figure 2. Percentage of loops serialized when disabling certain techniques.

POLARIS DETECTED MUCH OF THE PARALLELISM available in our set of benchmark codes. A careful analysis of the remaining loops that Polaris could parallelize highlights areas for improvement.

First, we need a true interprocedural framework for analysis. Our analysis algorithm requires a large amount of information, making a traditional global algorithm too inefficient. Consequently, we are focusing on a highly accurate demand-driven strategy (Polaris would do an analysis only when it is neccessary) that doesn't significantly increase the analysis time.

Second, we must improve our analysis techniques for dependence and privatization. If we generate analysis code for use at runtime, then we can use it when compile-time analysis fails. This will allow Polaris to parallelize loops with access patterns determined by values assigned during runtime.[7]

Third, Polaris must account for additional program patterns, such as more complicated forms for induction and reduction variables, associative recurrences, multiple exit loops, and loops containing I/O statements.

Finally, we must improve the efficiency of Polaris so we can compile very large Fortran programs. Although Polaris can routinely compile programs with 15,000 lines, we hope to eventually be able to compile programs 10 or 100 times larger.

Our Polaris project demonstrates that substantial progress in compiling conventional languages is possible. Based on our experimental results and hand analysis of real codes, we believe effective parallelizers for Fortran and similar languages will be available within the next decade. ▮

Acknowledgments

The research described here was supported by Army contracts DABT63-92-C-0033 and DABT63-95-C-0097.

References

1. R. Eigenmann et al., "Restructuring Fortran Programs for Cedar," *Concurrecy: Practice and Experience*, Oct. 1993, pp. 553-537.
2. D.J. Kuck et al., "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proc. Compsac 80*, IEEE CS Press, Los Alamitos, Calif., 1980, pp. 709-715.
3. W. Blume and R. Eigenmann, "The Range Test: A Dependence Test for Symbolic, Nonlinear Expressions," *Proc. Supercomputing '94*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 528-537.
4. P. Tu and D. Padua, "Automatic Array Privatization," *Lecture Notes in Computer Science*, Springer Verlag, 1993, pp. 500-521.
5. B. Pottenger and R. Eigenmann, "Idiom Recognition in the Polaris Parallelizing Compiler," *Proc. Ninth Int'l Conf. Supercomputing*, ACM Press, New York, 1995, pp. 444-448.
6. J.R. Grout, "Inline Expansion for the Polaris Research Compiler," Master's thesis, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, May 1995.
7. L. Rauchwerger and D. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *Proc. SIGPlan '95*, ACM Press, New York, 1995, pp. 218-232.

William Blume *is a software engineer in the Computer Languages Lab at Hewlett-Packard. His research interests are performance analysis and optimizing and parallelizing compilers. He received a PhD in computer science at the University of Illinois at Urbana-Champaign. He is a member of the IEEE Computer Society and ACM.*

Ramon Doallo *is an associate professor in the Department of Electronics and Systems at the Universidade Da Coruna, Spain. His research interests include parallelizing compilers, parallel sparse algorithms design, and effectiveness of cache utilization. He received his Licenciatura and PhD, both in physics, from the Universidad de Santiago de Compostela, Spain.*

Rudolf Eigenmann *is an assistant professor at the School of Electrical and Computer Engineering at Purdue University. His research interests include optimizing compilers, characteristics of computational applications, and performance evaluation for high-performance computer architectures; http://www.ece.purdue.edu/~eigenmann.*

John Grout *is a PhD student at the University of Illinois. He implemented the inline expansion component of the Polaris research compiler and is interested in demand-driven intraprocedural and interprocedural analysis techniques for parallelizing compilers. Grout received a BS in computer science from Worcester Polytechnic Institute and an MS in computer science from the University of Illinois at Urbana-Champaign. He is a student member of IEEE.*

Jay Hoeflinger *is a member of the Compiler Group at the Center for Supercomputing Research and Development at the University of Illinois. He worked on the Cedar Fortran project and now works on the Polaris project. He received a BS and an MS in computer science from the University of Illinois at Urbana-Champaign.*

Thomas Lawrence *is a software development engineer at Microsoft. His research interests include detection and exploitation of parallelism at runtime. He received a BA in computer science from the University of Wisconsin, Madison, and an MS in computer science from the University of Illinois, Urbana-Champaign.*

Jaejin Lee *is a PhD student in computer science at the University of Illinois and a research assistant at the Center for Supercomputing Research and Development. His research interests include optimizing explicitly parallel programs, parallelizing and optimizing compilers, parallel architectures, programming language theory, formal methods of code verification, functional programming, and lambda calculus. He received a BS in physics from Seoul National University, Korea, and an MS in computer science from Stanford University.*

David Padua *is a professor of computer science at the University of Illinois. His research interests include parallelizing compilers, software development tools, and parallel computer organization. Padua received a Licentiate in computer science from the Universidad Central de Venezuela and a PhD in computer science from the University of Illinois at Urbana-Champaign.*

Yunheung Paek *is a PhD student in computer science at the University of Illinois. His research interest is parallelizing compilers. He received a BS and an MS in computer engineering from Seoul National University.*

Bill Pottenger *is a visiting software engineer with the Polaris project at the University of Illinois. He is also a PhD student in computer science at the Center for Supercomputing Research and Development. He received a BS in computer science from the University of Alaska and an MS in computer science from the University of Illinois at Urbana-Champaign.*

Lawrence Rauchwerger *is an assistant professor in the Department of Computer Science at Texas A&M University. He received a Diploma Engineer from the Polytechnic Institute of Bucharest, Romania, an MS in electrical engineering from Stanford University, and a PhD in computer science from the University of Illinois. He is a member of the IEEE Computer Society and ACM.*

Peng Tu *is a member of the technical staff at Silicon Graphics. His research interests include program transformation for concurrency and locality, symbolic evaluation, global optimization, and loop nest optimization. He received a BS and an MS in computer science from Shanghai Jiao Tong University, China, and a PhD in computer science from University of Illinois.*

Contact the authors at the Polaris Project, Univ. of Illinois, 1304 W. Springfield Ave., Urbana, IL 61801; polaris@csrd.uiuc.edu.