Advanced Program Restructuring for High-Performance Computers 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

Department of Computer Science,
University of Illinois at Urbana-Champaign,
1304 W. Springfield Avenue,
Urbana, IL 61801, USA
polaris@csrd.uiuc.edu
http://polaris.csrd.uiuc.edu

Abstract

Multiprocessor computers are rapidly becoming the norm. Parallel workstations are widely available today, and it is likely that most PCs in the near future will contain multiple processors. To accommodate these changes, some classes of applications must be developed in explicitly parallel form. Yet, in order to avoid a substantial increase in software development costs, compilers that translate conventional programs into efficient parallel form clearly will be necessary. In the ideal case, multiprocessor parallelism should be as transparent to programmers as instruction level parallelism is to programmers of today's superscalar machines. However, compiling for multiprocessors is substantially more complex than compiling for functional unit parallelism, in part because successful parallelization often requires a very accurate analysis of long sections of code. This article discusses recent experience at Illinois on the automatic parallelization of scientific codes using the Polaris restructurer. We also present several new analysis techniques that we have developed in recent years based on an extensive analysis of the characteristics of real Fortran codes. These techniques, based on both static and dynamic parallelization strategies, have been incorporated into the Polaris restructurer. Preliminary results on parallel workstations are encouraging, and once the implementation of the new techniques is complete, we expect that Polaris will be able to obtain good speedups for many real scientific codes on parallel workstations.

Keywords: Compilers, Program Restructuring, Automatic Parallelization, Parallel Computing.

1 Introduction

Parallel computing is an enabling technology for many areas of science and engineering. Parallel computing is also our best investment to continue achieving performance gains as the limits of semiconductor technology are approached. Whereas the design of parallel computers is reasonably well-understood, the programming of these machines remains in its infancy. As a consequence, programming parallel computers is substantially more difficult than programming conventional uniprocessors. In fact, it is commonly said that the development of effective parallel programming techniques is the main challenge faced by high-performance computing today.

Unless this challenge is met, the acceptance of parallel computers will remain slow, thereby hampering progress in computational science and engineering. An appealing strategy to meet this challenge is to develop *parallelizers*. That is, compilers that translate conventional sequential programs into parallel form. Such compilers would enable the execution of existing programs on new parallel machines, thus allowing a seamless transition into parallel computing. Also, with the support of parallelizers, new programs could be developed in the familiar sequential paradigm, thus liberating the programmer from the complexities of explicit, machine-oriented parallel programming.

In this article, we present an overview of Polaris, an experimental parallelizer for shared-memory multiprocessors and scalable machines with a global address space, such as the Cray T3D. For a broader overview of compiler techniques for parallel computers and a more extensive list of references, the reader is referred to [2].

2 Techniques Implemented in Polaris

The development of the most important techniques implemented in Polaris ensued from a study of the effectiveness of commercial Fortran *parallelizers* [4]. In that study, the Perfect Benchmarks, a collection of conventional Fortran programs representing the typical workload of high-performance computers, were compiled for the Alliant FX/80, an eight-processor multiprocessor popular in the late 1980s. For each program, we computed the *speedup* as a measure of the quality of the parallelization. By speedup, we mean the ratio of a program's sequential execution time to the execution time of the automatically parallelized version. Although the speedups obtained were

satisfactory in a few cases, the Alliant Fortran compiler failed to deliver any significant speedup for the majority of the programs.

The main reason for this failure was the inability of the compiler to parallelize some of the most important loops in the Perfect Benchmarks. In retrospect, this was not surprising because the parallelization module of the Alliant compiler was originally developed for vectorization, then retro-fitted to parallelization. This was a common practice in the late 1980s when commercial multiprocessors were still relatively new, and the only software available was for vectorization. Vectorizers [1, 7] focus primarily on innermost loops in their search for vector operations. While multiprocessor compilers can take advantage of the availability of multiple control units by parallelizing outer loops. Furthermore, outer loop parallelization is often needed to amortize the start-up overhead typical of today's multiprocessors.

Our study showed that extending the four most important analysis and transformation techniques traditionally used for vectorization led to significant increases in speedup. Next, we briefly discuss each of these four techniques and the extensions we found necessary to obtain good multiprocessor speedups.

2.1 Dependence Analysis

A loop can be transformed into parallel form if it contains no *cross-iteration dependences*. That is, the loop must not have two iterations that access the same memory location, if at least one of the accesses changes its value.

Dependence analysis techniques proceed by analyzing in turn, every pair of references to the same array within a loop to check whether their subscript expressions might produce the same value in two different iterations. To guarantee correct code, dependence analysis techniques must assume cross-iteration dependences 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
```

Studying the subscript expressions is necessary to determine if cross-iteration dependences are possible between different executions of statement R, between the statements R and S, and between R and T. Notice that there cannot be a cross-iteration dependence between S and T because there is no write to array A in either statement.

One of the simplest dependence analysis techniques is the *Equality Test*. This test determines that there are no cross-iteration dependences between two array references whenever the subscripts are identical *linear* expressions. In the previous loop, the Equality Test will determine that there are no cross-iteration dependences between two executions of statement R, nor between R and S. However, it will leave open the possibility of a cross-iteration dependence between R and T.

Another simple test is the *GCD Test*, which checks whether an equation involving two linear subscript expressions has an integer solution. Thus, to analyze the potential dependence between R and T, the GCD Test considers the equation $2i = 2i' + 1$. Since there are no integer solutions for i and i' in this equation, R and T will never access the same element of A.

The GCD Test determines that there are no integer solutions to the equation when the greatest common divisor of the coefficients does not divide the independent term. Thus, the previous equation has no integer solution because $GCD(2, 2) = 2$ does not divide 1. This shows that there are no dependences between R and T. Notice that in the loop above, the GCD Test cannot disprove the two potential dependences which were disproved by the Equality Test. Therefore, both tests are necessary to obtain accurate results. In practice, parallelizing compilers apply a variety of these dependence tests in sequence.

Like the GCD and Equality tests, most dependence tests only work on linear expressions. However, we found several cases of nonlinear subscript expressions in the Perfect Benchmarks. As discussed below, some nonlinear subscripts were generated by our induction variable substitution transformations, and the rest were part of the

original program. To handle these, we developed the *Range Test* [3]. Our implementation of the Range Test makes use of sophisticated computer algebra capabilities as well as data range information extracted from the whole program.

Another powerful dependence test is the *Omega Test* [9], which is capable of detecting parallelism in many, but not all, cases that the Range Test can analyze accurately.

2.2 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
```

Temporary variables cause cross-iteration dependences because they are read and written on different loop iterations. However, when a temporary variable is assigned before it is read within each iteration, it is possible to use a different memory location for the variable in each loop iteration. This eliminates the dependence.

When translating for multiprocessors, it is sufficient to give such a variable a different location for each processor, by declaring the variable *private* to the loop.

For vectorization, it is often sufficient to identify loop private scalars. However, we found many cases in the Perfect Benchmarks where arrays are used as temporaries within multiply-nested loops. Identifying private arrays eliminates many apparent dependences in outermost loops.

Arrays may be privatized under conditions analogous to those for scalars - each element of an array which is read in an iteration must have already been assigned in that iteration.

In Polaris, we have implemented a privatization technique to deal with both scalars and arrays [11]. This algorithm had to be substantially more complex than the traditional scalar privatization algorithm 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.

2.3 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
```

The presence of induction variables presents a parallelization problem for two reasons. First, induction variables are read and written on every iteration and, thus, are the source of cross-iteration dependences. Second, a subscript expression involving induction variables cannot be directly analyzed by most dependence tests because the variables are not loop indices, nor do they have constant values.

To avoid these problems, all occurrences of an induction variable must be transformed by the compiler. They are replaced by an expression involving the loop index. For example, in the previous loop, the array reference could be replaced by `U(2*I)`. Once that is done, the induction statement (and its dependence) can be removed.

Many current 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 different levels of nesting within a single loop. Techniques that produce closed form expressions in terms of several loop indices for these cases have been implemented in Polaris [8]. For example, in the loop:

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

all occurrences of J after S in the outer loop would be replaced by I + I*(I-1)/2, and those after T within the inner loop by I + I*(I-1)/2 + K. Notice that, as mentioned above, these expressions are nonlinear and, although unanalyzable by a traditional dependence test, the Range Test applied by Polaris can handle them. Another approach to induction variable recognition, also inspired by our earlier study, is presented in [12].

2.4 Reduction Substitution

A reduction operates across one or more dimensions of an array to produce a *reduction variable* in the form of a lower dimensional array or a scalar. A reduction causes cross-iteration dependences in the loop that evaluates it. Within the loop, if a reduction variable is accessed only by the statements performing the modification and the reduction operation is associative, it can be parallelized. For example, in the loop

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

Q is accessed only in S. If we assume that floating-point addition is associative then the loop can be transformed into a parallel form in which each processor computes the sum of a different section of A. The partial sums are added at the end of the loop to obtain the final value of Q. As is well-known, floating point operations are not associative, but in many cases assuming that they are does not change significantly the result of the program.

Vectorizers considered only simple reductions, as in our example. However, more complex patterns occur in many programs. For example, the reduction variable could be an array, there could be multiple reduction statements in a loop, and the subscripts of reduction arrays can be array elements themselves. Advanced techniques to handle cases like these have been implemented in Polaris[8][6].

3 Effectiveness of Polaris Techniques

In addition to the four techniques presented in the preceding section, Polaris applies *Autoinlining* [5] and *Interprocedural Value Propagation (IPVP).* Autoinlining replaces a call to a subroutine with the code for that subroutine, when heuristics deem it profitable. Interprocedural Value Propagation finds instances in which an expression representing the value of an integer variable in one subroutine is valid upon entry to a different subroutine. When a routine is called from more than one call site with different values for a given variable, IPVP clones the subroutine, and uses a different value in each clone.

As can be seen below, these two transformations, although not as powerful as a comprehensive interprocedural analysis algorithm, have improved the accuracy of program analysis in several cases.

Figure 1 presents a speedup comparison between Polaris and SGI's parallelizer PFA. PFA, like the Alliant parallelizer, was developed by an independent software house. Both parallelizers were originally developed as vectorizers.

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

- `arc2d`, `bdna`, `flo52`, `mdg`, `ocean`, and `trfd` from the Perfect Benchmark suite;
- `applu`, `appsp`, `hydro2d`, `su2cor`, `swim`, `tfft2`, `tomcatv`, and `wave5` from the SPEC CFP95 Benchmark suite; and,
- `cmhog` and `cloud3d` from the National Center for Supercomputing Applications (NCSA).

The first two suites are well-known. The third source is a collection of programs currently being used in scientific research at NCSA. The programs in the NCSA collection are of moderate size, containing approximately

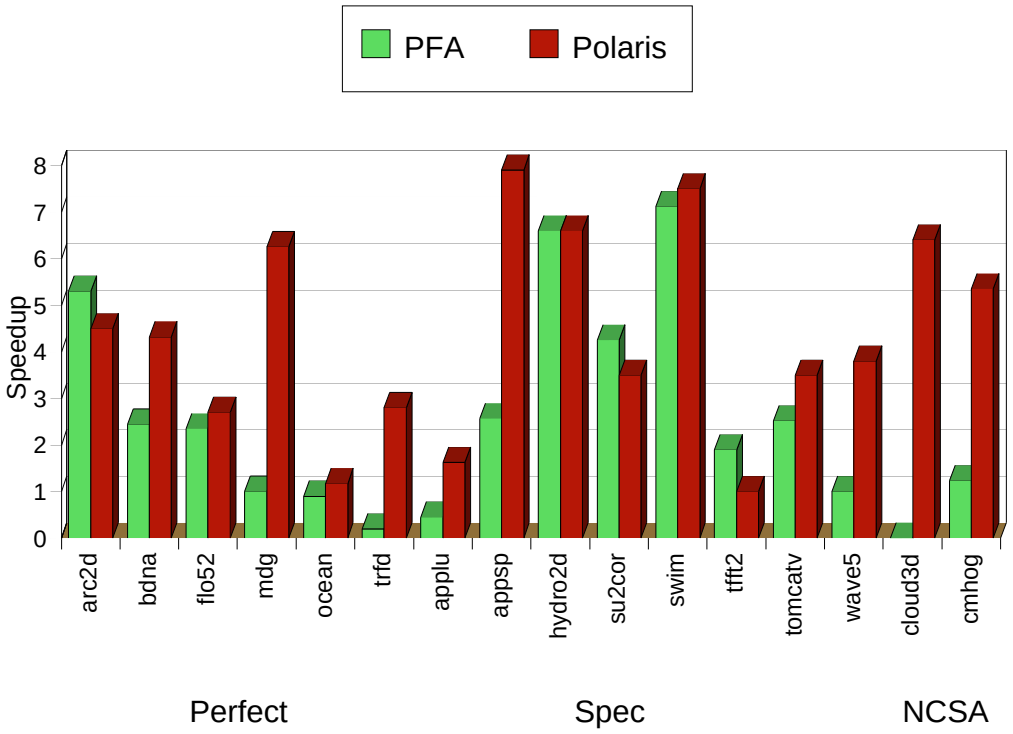

Figure 1: Speedup Comparison Between the SGI PFA Compiler and Polaris

10,000 lines each.

The programs were executed (in real-time mode for timing accuracy) on eight processors of an SGI Challenge with 150 MHz R4400 processors, located at NCSA. Figure 1 shows that Polaris delivers, in many cases, substantially better speedups than PFA.

In three of the sixteen programs, PFA produces better speedups than Polaris. The reason is that PFA uses an elaborate code generation strategy that includes loop transformations, such as loop interchanging, unrolling, and fusion, which, when applied to the right loops, improve performance by decreasing overhead, enhancing locality, and facilitating the detection of instruction-level parallelism. However, the elaborate strategy has a negative effect for the program `tomcatv`, for which PFA detects as much parallelism as does Polaris, yet the Polaris-transformed program ran faster.

To evaluate the effectiveness of Autoinlining, IPVP, Array Privatization, Range Test, Multiply-Nested Induction Substitution, and Advanced Reduction Substitution, we transformed each program six times, with a different technique turned off each time, and then compared those with the program compiled with all techniques enabled.

These six transformations contain all that is new in Polaris with respect to PFA and the Alliant parallelizer. However, PFA includes some of the capabilities in these six transformations. For example, the dependence analysis modules of PFA and most commercial parallelizers, although not as powerful as the Range Test, are substantially more powerful than the Equality and GCD tests. Also, PFA can substitute multiply-nested induction variables but only if the bounds of inner loops do not contain indices of outer loops. A detailed comparison of the capabilities of Polaris and commercial parallelizers is beyond the scope of this article.

Figure 2 presents the results of the six experiments conducted for each code. The height of the bar at (P, T) represents, in logarithmic scale, the percentage of the total number of loops in program P which become serialized by disabling technique T.

From Figure 2, it can be seen that all techniques enable parallelization of loops from many programs in the collection. Although the inspiration for the techniques came from analyzing programs in only the Perfect Benchmarks, they seem equally useful for programs outside the Perfect Benchmarks.

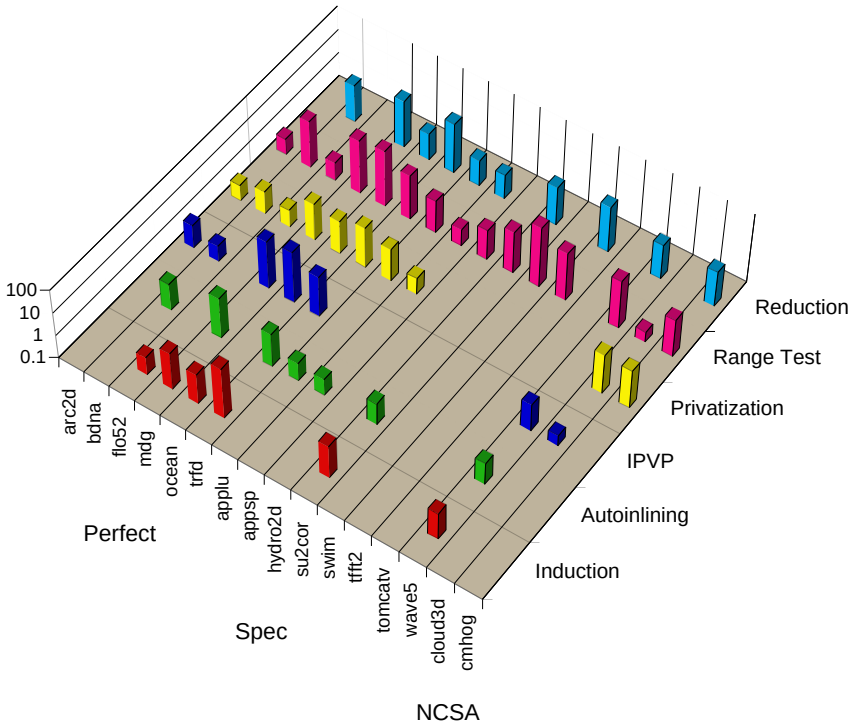

Figure 2: Percentage of Loops Serialized when Disabling Certain Techniques

4 The Future: More Parallelism, Better Performance

Polaris has detected much, but not all, of the parallelism available in our set of benchmark codes. A careful analysis of the loops in our benchmark codes which Polaris could parallelize, but does not, shows several areas where improvement is possible.

First, we need a true interprocedural framework for analysis. However, due to the large amount of information required by our analysis algorithms, a traditional global algorithm would be too inefficient. We are, therefore, focusing on a demand-driven strategy that would allow high accuracy without significantly increasing the analysis time.

Second, improving our analysis techniques for dependence and privatization is important. When compile time analysis fails, analysis code needs to be generated for use at run-time. This will allow Polaris to parallelize even loops with access patterns determined by input values [10].

Third, Polaris needs to take into account additional program patterns, such as more complicated forms for

induction and reduction variables, associative recurrences, multiple exit loops, and loops containing I/O statements.

We also must improve the efficiency of Polaris in order to allow us to compile very large Fortran programs. Although Polaris can routinely compile programs with 10,000 lines, we hope to eventually be able to compile programs 10 or 100 times larger.

5 Conclusions

The techniques implemented in the first version of Polaris have proven quite effective on a variety of programs. This verifies the results of our study involving the Alliant machine. Even so, we find that further progress is necessary. A wide range of techniques is necessary to detect all the parallelism in real programs.

Perhaps the most important achievement of our work in the Polaris project has been the demonstration that substantial progress in compiling conventional languages is possible. There have been, and still are, many who do not believe it is possible to develop compilers capable of generating effective parallel code for a wide range of real programs. We disagree, and believe that, at least for Fortran and other similar languages, effective parallelizers will be available within the next decade. Our experimental results and careful hand analysis of real codes strongly support this opinion.

6 Acknowledgments

The research described in this article is supported by Army contracts #DABT63-92-C-0033 and #DABT63-95-C-0097. This work is not necessarily representative of the positions or policies of the Army or the Government.

References

[1] J. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):221–243, February 1993.

[3] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94, Washington D.C.*, pages 528–537, November 1994.

[4] Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, and David Padua. Restructuring Fortran Programs for Cedar. *Concurrency: Practice and Experience*, 5(7):553–573, October 1993.

[5] John Robert Grout. Inline Expansion for the Polaris Research Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995. CSRD Technical Report 1431.

[6] Jee Ku. The Design of and Efficient and Portable Interface Between a Parallelizinbg Compiler and Its Target Machine. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995. CSRD Technical Report 1500.

[7] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe. The Structure of an Advanced Vectorizer for Pipelined Processors. *Proc. of COMPSAC 80, The 4th Int'l. Computer Software and Applications Conf.*, pages 709–715, Oct., 1980.

[8] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 1995 ACM International Conference on Supercomputing.*, pages 444–448, July 1995.

[9] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13, November 1991.

[10] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 218–232, June 1995.

[11] Peng Tu and David Padua. Automatic Array Privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.

[12] Michael Wolfe. Beyond induction variables. In *Proc. ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 162–174, 1992.