Compiling for the New Generation of High-Performance SMPs *

Insung Park Michael J. Voss Rudolf Eigenmann

November 8, 1996

Abstract

Shared-Memory Parallel computers (SMPs) have become a major force in the market of parallel high-performance computing. Parallelizing compilers have the potential to exploit SMPs efficiently while supporting the familiar sequential programming model. In recent work we have demonstrated that Polaris is one of the most powerful translators, approaching this goal. Although shared memory machines provide one of the easier models for parallel programming, the lack of standardization for expressing parallelism on these machines makes it difficult to write efficient portable code. In this paper we will report on a new effort to retarget the Polaris compiler at a range of new SMP machines through a portable directive language, Guide^{TM}, in an attempt to provide a solution to this problem. We will discuss issues in compiling with this language and the performance obtained on two machines based on a number of significant application programs.

1 Introduction

Shared-memory multiprocessors (SMPs) have recently regained much interest. For example, it is possible to buy a multiprocessor workstation at only little additional cost over a uniprocessor system. It is likely that such multiprocessor systems will become very widely used in the near future. Shared memory machines provide one of the easier conceptual models for parallel programming. Although this makes programming these machines relatively straight forward, there has been a lack of standardization of expressing parallelism. In moving from one machine of this type to another, it is often necessary to learn a new set of vendor-specific parallel language constructs in order to program on the new architecture [EPV96, VPE96]. Furthermore, even if an existing program is successfully ported, users may not get the level of performance they expect. Because of the increasing variety of SMP architectures currently available on the market, this problem represents a still substantial hurdle that we need to take, before parallel computing with these machines can become ubiquitous.

One way to address this problem is to use a parallelizing compiler, which would free the user from the need to invest time in learning architecture-specific parallel languages [VPE96]. The burden, then, is on compilers which face the challenge of translating into a vendor-specific target language (this can be either an explicit parallel language or an internal representation). However, this creates the need for a unique set of language constructs to be generated for each machine that a code may be run on. Creating such architecture-specific back-ends to a compiler adds complexity and costs time. Time, that may otherwise be available for developing techniques that apply to a range of architectures.

In this paper we begin to explore one possible solution to these problems. Kuck and Associates (KAI) of Champaign, Illinois has developed a parallel programming model for shared memory machines, based on the results of two previous working groups, the Parallel Computing Forum, and the ANSI subcommittee, X3H5. They have implemented their model with the Guide^{TM} parallel language and its translator. The Fortran version of Guide, which is used in this study, is a Fortran 77 (F77) plus

*This work was supported in part by U. S. Army contract #DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the U. S. Army or the Government.

parallel directive language. Its associated translator converts code expressed in this dialect into F77 plus vendor-specific library calls, which are available for all of the popular SMP architectures in use today. This frees programmers from machine-specific syntax, allowing portable code to be written by users, and generated by parallelizing compilers. Thus, in this paper we discuss the performance of automatically parallelized programs with Guide directives, as compared to the performance of the same codes automatically parallelized by vendor-specific compilers. In cases where there is a significant difference in the observed behavior, possible explanations are discussed.

The Guide language can be used as both an explicit parallel user language and a target language of a parallelizing source-to-source translator. Since our main interest lies in automatic parallelization, we chose this as our means of obtaining experimental codes. In order to automatically generate Guide constructs, we added a back-end (referred to as Guide back-end henceforth) to the Polaris parallelizing compiler. Polaris is a restructuring compiler originally developed at the University of Illinois at Urbana-Champaign, that uses many advanced techniques to recognize, enhance, and exploit loop-level parallelism. An overview of the Polaris compiler can be found in [BDE+96].

To determine the efficiency of portable code expressed in the Guide parallel language, two shared-memory machines were then chosen for experimentation. These machines are a four-processor Sun SPARC workstation, and a twelve-processor SGI Challenge machine. The performance of code expressed in the portable Guide language is compared with that automatically parallelized by the Sun SPARC20 native compiler and the SGI Power Fortran Accelerator.

Section 2 will give some details of the history and syntax of the Guide programming language. Section 3 gives an overview of the Polaris compiler and describes the modifications that we have made so that it generates optimized code for Guide. Next, in Section 4, we briefly outline the architectures of the machines on which our experiments were performed. Following that, Section 5 presents measurements of parallel programs run on our machines and a discussion of their performance. Section 6 outlines our ongoing research. This section also includes subsection 6.3, which gives our view on the features that could have been useful and evaluates desirable directions for directive languages for shared memory machine as well as subsection 6.4, which introduces tools used to analyze and extract data from the experimental results. Finally, the conclusions of the paper are presented in Section 7.

2 Portable Language for Shared-Address-Space Machines

2.1 A Brief History of Portable Parallel Languages

In 1987 the Parallel Computing Forum (PCF), an informal industry group, was formed with the goal of standardizing DO loop parallelism in Fortran 77. This group was active from April 1987 through February 1990, with their final report being published in 1989. Although PCF was dissolved, an ANSI authorized subcommittee, X3H5, was chartered for building a language-independent model for parallelism in high-level programming languages, as well as the bindings of this model into Fortran and C. This subcommittee used the PCF final report as a foundation for their model. The group began their work in February of 1990; however by May of 1994 interest was lost, and the proposed standards abandoned. Nevertheless, their results yielded a completed language-independent model, an almost complete binding for Fortran 90, and a preliminary draft for ANSI C. [KA]

Kuck and Associates (KAI) continued to work on the partially completed documents of the ANSI X3H5 committee, and expanded them when required. This resulted in the Guide programming system, illustrated in Figure 1, which takes Fortran 77 with the proposed ANSI X3H5 directives as input, and generates F77 code with vendor-specific thread calls as output. These threads are managed by calls to the Guide support libraries.

Before describing Guide in more detail, two related efforts shall be noted, that also attempt to facilitate portable parallel programs: the High-Performance Fortran (HPF) [For93] and the PVM/MPI [BDG+93, For94] efforts. Both PVM and MPI provide message passing functions for programs that run on parallel processors with separate memories. Several factors have contributed to the fact that programming in these message passing packages has become widely applied. These factors include the standardization achieved by PVM and MPI, the fact that, for some time, message passing has been the only feasible way to program distributed-memory machines, and the fact that such programs can be ported to shared-address-space machines as well. On the negative side, message-passing programming can be tedious because of the task of partitioning a program and data space across processors and bookkeeping loop indices, array subscripts and conditional operations on partition boundaries.

The HPF effort was initiated with the goal of simplifying this very task. It provides a shared-address model to the user and generates the message passing program through intelligent translators. The user specifies data distributions of arrays, but does not need to get involved in array subscript bookkeeping operations. However, although HPF supports a shared-address-space model, its emphasis is on distributed-memory/message passing machines. For example, it lacks concepts of loop-private arrays, which are among the most important language elements for programming SMP machines. On the other hand, HPF's pioneer work in data distribution concepts has generated knowledge that may, in the future, benefit the languages focused on in this paper. Currently, the Guide language does not provide data distribution or other latency-hiding facilities, which may become important when we deal with true non-uniform memory access machines.

2.2 An Overview of the Guide Programming System

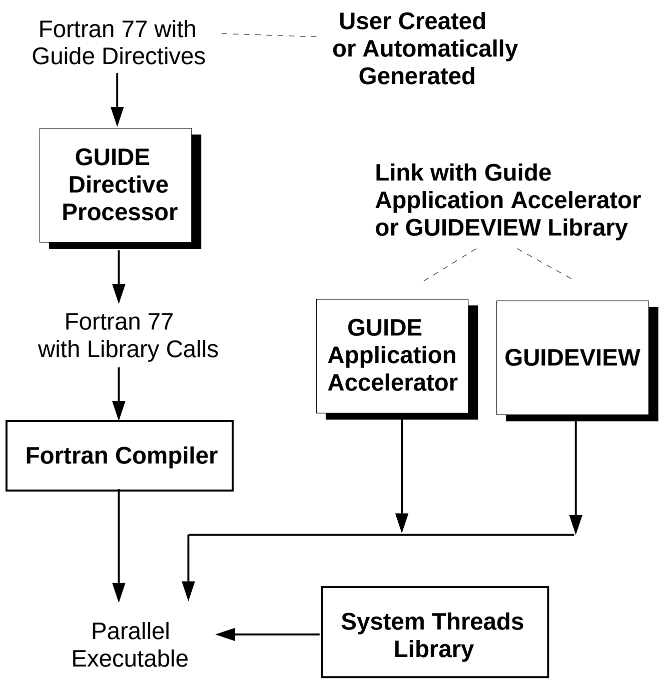

Fortran 77 with Guide Directives
User Created or Automatically Generated
GUIDE Directive Processor
Link with Guide Application Accelerator or GUIDEVIEW Library
Fortran 77 with Library Calls
GUIDE Application Accelerator
GUIDEVIEW
Fortran Compiler
Parallel Executable
System Threads Library

Figure 1: The Guide Programming System

The input to the Guide Directive Translator, pictured in Figure 1, is standard sequential Fortran 77, with Guide directives used to express parallelism. The Guide language is a relatively large directive language. A detailed description of the Guide programming model and syntax can be found in [Kuc96]. For our compiler interface with the Polaris translator we have used only a small subset of the directives.

A parallel section of code begins with the **C$PAR PARALLEL** directive, and ends with the **C$PAR END PARALLEL** directive. Statements between these directives will be executed on each of the processors. At the beginning of a parallel section, each variable contained within, must be classified as being shared by all processors, or private to each. This is done by including the variable within the argument list of either a **SHARED()** or **LOCAL()** directive.

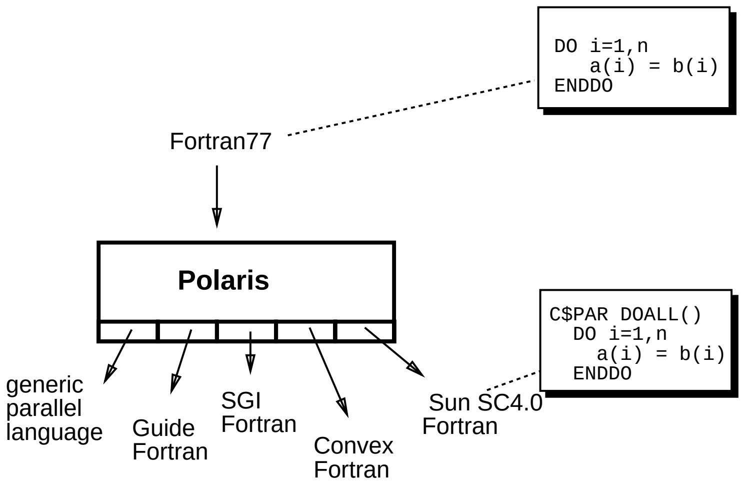
Within a parallel section, a parallel do statement is enclosed within **C$PAR PDO** and **C$PAR END PDO** directives. These loops will then have their iteration space divided among the available processors. The Guide language also provides for scalar reductions, which are statements within a loop of the form sum = sum + a(i), where i is the loop index. Any scalar reduction variables must be identified at the beginning of a parallel do statement by including them in a **REDUCTION()** directive. Array reductions are similar patterns, however, the reduction variable is an array with a loop-variant subscript. Array reductions are not supported by the directive language. Because of this, Polaris transforms them into fully parallel loops, as we will describe.

Separating parallel regions from the parallel do sections, allow for pre- and postambles, sections of code executed by each processor participating in the parallel execution once at the beginning and at the end of the loop. These sections were used for initializing private variables, performing reduction operation, etc.

3 The Polaris Parallelizing Compiler

3.1 Polaris overview

Polaris is a parallelizing compiler, originally developed at the University of Illinois. As illustrated in Figure 3.1, the compiler takes a Fortran77 program as input, transforms it so that it can run efficiently on a parallel computer, and outputs this program version in one of several possible parallel Fortran dialects.



```
DO i=1,n
   a(i) = b(i)
ENDDO
```

```
C$PAR DOALL()
  DO i=1,n
    a(i) = b(i)
  ENDDO
```

Figure 2: Overview of the Polaris parallelizing compiler

The input language includes several directives, which allow the user of Polaris to specify parallelism explicitly in the source program. The output language of Polaris is typically in the form of Fortran77 plus parallel directives as well. For example, the *generic parallel language* includes the directives “CSRD$ PARALLEL” and “CSRD$ PRIVATE a,b”, specifying that the iterations of the subsequent loop shall be executed concurrently and that the variables `a` and `b` shall be declared “private to the current loop”, respectively. Figure 3.1 shows several other output languages that Polaris can generate, such as the directive language available on the SGI Challenge machine series, the Sun SC4.0 Fortran directive language, and the Guide directives introduced above.

Polaris performs its transformations in several *compilation passes*. In addition to many commonly known passes, Polaris includes advanced capabilities for array privatization, symbolic and nonlinear data dependence testing, idiom recognition, interprocedural analysis, and symbolic program analysis. An extensive set of optional switches allow the user and the developer of Polaris to experiment with the tool in a flexible way. An overview of the Polaris transformations is given in [BDE+96].

The implementation of Polaris consists of some 200,000 lines of C^{++} code. A basic infrastructure provides a hierarchy of C^{++} classes that the developers of the individual compilation passes can use for manipulating and analyzing the input program. This infrastructure is described in [FHP+94]. The Polaris Developer's Document [Hoe96] gives a more thorough introduction for compiler writers.

Polaris is a general infrastructure for analyzing and manipulating Fortran programs. The use of this infrastructure as a parallelizing source-to-source restructurer is the main application. Another application is that of a program instrumentation tool. Currently, Polaris can instrument programs for gathering loop-by-loop profiles, iteration count informations, and for counting data references. We made use of these facilities in order to obtain the results described in this paper. It is also possible to use this infrastructure to obtain other useful informations on the program under consideration. Other tools based on Polaris is discussed in Section 6.4.

3.2 Polaris modifications and enhancements

The Guide Programming System is targeted at shared-memory architectures, which is one of the classes of machines for which Polaris has been developed. Because of this, the initial movement of the compiler to this language was fairly simple. Most of the additions to the code were localized to the final *output pass*, where the generic directives used in the internal representation are mapped to vendor-specific directives. The output pass is the last one in a series of compilation passes. It generates the Fortran-plus-directive output language from the internal representation.

Polaris internally provides to its output pass generic directives that distinguish between serial and parallel loops, and shared and private variables. In our new Guide output pass, these generic directives can be mapped directly to their corresponding Guide directives. The next section describes some of the important transformations conducted by our new Guide back-end.

Array Reduction and Privatization

Both reduction parallelization and array privatization are among the most important transformations of a parallelizing compiler [BDE+96, TP93, PE95]. As we have mentioned above, Polaris is capable of transforming array reductions into fully parallel loops. Figure 3 gives an example array reduction in its serial and parallelized forms. Often, the number and indices of the privatizable array elements cannot be determined at compile time, which is a frequently encountered situation in an array reduction operation (array `A` in our example) and an array privatization. In the case of array reduction, one solution is to provide a local copy of the array to each processor, perform the accumulation operations of this array in the now fully parallel loop, and then recombine the local arrays after the loop is complete.

Array privatization likewise requires creation of local copies of an array for each processor. In cases that warrant privatization, an array is used as temporary storage for a given iteration. Data dependencies that may exist are a matter of storage reuse, and are not true flow dependencies [TP93]. Private arrays can be expressed in the Guide directive language by listing the arrays on the LOCAL list. Arrays whose size is not known at compile time cannot be declared private in this way. Instead, they can be expanded by a dimension equal to the number of processors and accessed by the processor identification as an index, as shown in Figure 4. Dynamic allocation of shared arrays is supported in most parallel Fortran dialects.

As shown in these examples, `my_proc_id()` may be called frequently within the resulting parallel codes. This may cause too much run-time overhead. Since Guide allows for preambles to parallel loops, Guide back-end places a single call and stores the value in a local variable in a preamble, and replaces

```
DIMENSION A(M)
DO I=1,N
 A(B(I)) = A(B(I)) + X
ENDDO
```

(a) Serial Array Reduction

```
DIMENSION A(M),Aloc(M,number_of_processors)
PARALLEL DO I=1,M
 Aloc(I,my_proc_id()) = 0
ENDDO

PARALLEL DO I=1,N
 Aloc(B(I),my_proc_id()) =
  Aloc(B(I),my_proc_id()) + X
ENDDO

PARALLEL DO I=1,M
 DO J=1,number_of_processors
   A(I)=A(I)+Aloc(I,J)
 ENDDO
ENDO
```

(b) Parallel Array Reduction

Figure 3: Array Reduction operation

```
DO J=1,M
 DO I=1,N
   A(I) = ...
 ENDDO
 ...
 DO I=1,N
   ... = A(I)
 ENDDO
ENDDO
```

(a) Serial

```
PARALLEL DO J=1,M
 DO I=1,N
   A(I,my_proc_id())= ...
 ENDDO
 ...
 DO I=1,N
   ... = A(I,my_proc_id())
 ENDDO
ENDO
```

(b) Expanded array

Figure 4: Array privatization through *expansion*

subsequent calls within parallel code section with that variable. This can amortize the cost over the total execution time. We have used this transformation in both cases where the processor_id would be needed: parallel loops with reduction operations and private arrays. Figure 5 shows array expansion transformation using preambles with actual Guide directives.

4 The Machines Used for Experimentation

The Sun SPARC 20 Multiprocessor architecture

Figure 6 shows the Sun SPARC20 workstation used in our experiments. It consists of four 100 MHz hyperSPARC processors that share a single global memory space [Wp96]. Each processor has its own 256-KB external cache and an 8-KB on-chip instruction cache. Both write-through with no write allocate and copy-back with write allocate caching schemes are provided. In the case of multiprocessing, the external cache uses the copy-back scheme, maintaining cache coherency through a high performance snoop mechanism. The copy-back scheme is preferred as it writes to main memory less often than a write through scheme, and therefore saves memory bandwidth for interprocessor communications through the shared global memory.

```
PARALLEL DO J=1,M
   ... A(x,my_proc_id()) ...
ENDO
```

(a) With Function

```
C$PAR PARALLEL
 proc = mpptid() + 1
C$PAR PDO
 DO J=1,M
    ... A(x,proc) ...
 ENDO
C$PAR END PDO
C$PAR END PARALLEL
```

(b) Using Preamble

Figure 5: Identifying the current processor

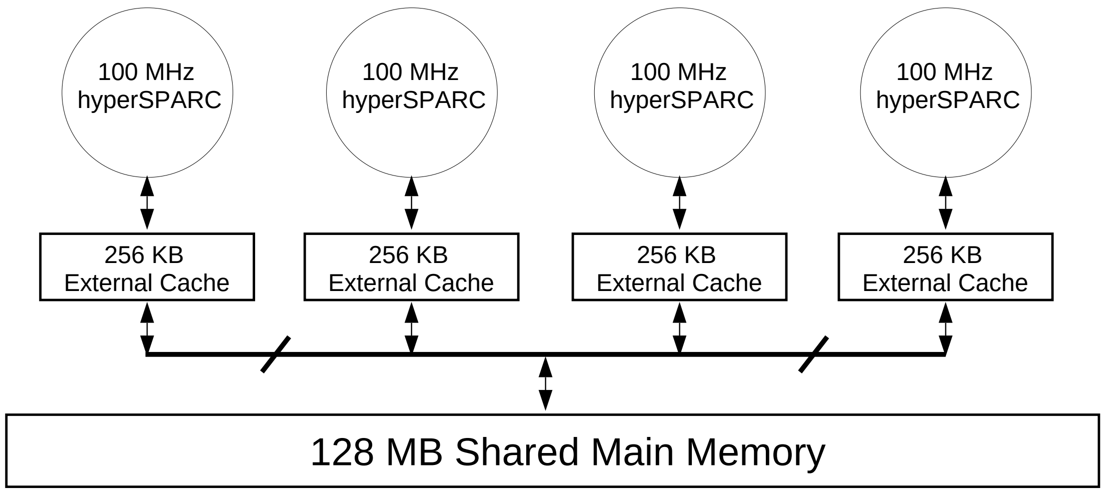


Figure 6: The Sun SPARCstation20 architecture

The SGI Challenge Multiprocessor architecture

A twelve-processor Silicon Graphics Challenge machine was our second target. Even though it has twelve processors, we only used up to four processors for the comparison with the Sun workstation. SGI challenge is a shared memory machine with basically the same block diagram as in Figure 6. It is based on the 150 MHz MIPS superscalar RISC R4400 chip with peak performance of 75 MFLOPs [fSA]. Each processor is equipped with separate 16-KB direct-mapped on-chip instruction cache and data cache as well as 4-MB four-way set associative external cache. It has 1 GB of memory, and the communication between processors is done by a fast shared-bus interconnect with bandwidth of 1.2 gigabyte per second. The basic schematics for SGI Challenge is the same as Figure 6 except the changes in parameters and the number of processors. As with the SPARC20, it is a bus-based machine which uses a snoop cache-coherence scheme; however, unlike the Sun workstation, it always uses an Illinois Protocol write-invalidate scheme [PP84].

5 Measurements

Our measurements will help quantify the following issues:

Performance of Guide directives: The main objective of this paper is automatic parallelization through a portable directive language for SMPs. Good performance of Guide is the basis of our research.

Performance of Polaris versus commercial parallelizers: There are several commercial parallelizing compilers available on the market. We will compare the performance of these parallelizers with Polaris.

Possible improvements for better performance: Another important goal of our effort is to make suggestions to software and hardware vendors on improving their language, compiler, and architecture.

The measurements are presented in two parts: the experiments with the Sun workstation and those with the SGI Challenge machine.

5.1 The Experiment

In our experiments, we have studied several programs in three forms; the serial version, the version produced by Polaris expressing parallelism with the Guide-specific directives, and the version parallelized by the vendor-specific parallelizing compiler. We usedtheS recently released version 2.1 of Guide. The codes we used are MDG, ARC2D, and FLO52Q from the Perfect Club Benchmarks, and APPSP and EMBAR from the NAS kernels.

The serial version is the original version instrumented with timing functions. In the first step of our experiments, we measured the execution time of the serial loops. We inserted the instrumentation functions at the beginning and end of each loop and ran the resulting code to generate the timing profile. In order to minimize perturbation, we then eliminated the instrumentation functions inserted for either insignificant loops or the loops that execute a large number of times. We did this based on the profile obtained previously. We ran the resulting program to obtain the final profile, which then served as the basis for performance comparisons.

The Fortran SC4.0 compiler on the Sun workstation also includes a capability to do automatic parallelization. We will compare the performance of programs parallelized by Polaris with the performance of this compiler. Likewise, SGI machines have a native parallelizing compiler, Power Fortran Accelerator (PFA). When we use these commercial parallelizing compilers, we used the partially instrumented serial programs described above as the input, so that we can have timing information on the loops of interest. The two versions of code generated by these compilers will then show us the differences in automatically generating parallel programs with vendor specific parallelizer and with Polaris in the portable Guide directives.

To ensure reliable timing data, we usually ran the programs several times under the same environment and checked the range of execution time before we took the measurement. During our program runs, we did not enforce a single-user mode on either machine. However we made sure that the machine load was low during the experiments. In this way we could measure the performance that an ordinary user would see without taking special precautions.

5.2 Results on the SPARC20

Overall Performance Results

This section presents the experimental results obtained through various runs of the Perfect Benchmark programs ARC2D, FLO52Q, and MDG, and NAS kernels APPSP and EMBAR on the SUN SPARC20 workstation.

Figure 7 shows the overall timing of the different parallel versions of the Perfect Benchmark programs. Labels on the x axis indicate the number of processors used to run the program. The graphs show the timings of the programs parallelized with Polaris+Guide back-end with solid lines and those parallelized with the Sun parallelizing compiler with dotted lines, respectively. For comparison, the "ideal" curve

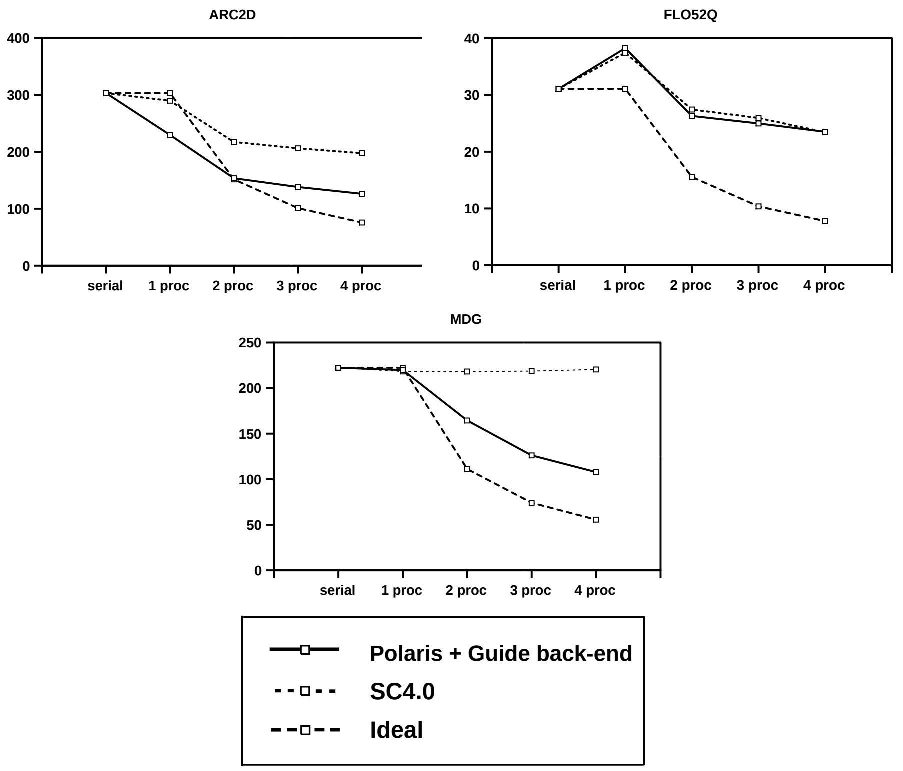


Figure 7: Execution time of different parallel versions of Perfect benchmarks on SPARC20

shows $\frac{serial\_execution\_time}{number\_of\_processors}$, and they are presented in dashed lines. The overall timing of the three NAS kernel programs is shown in Figure 8 in the same format.

Figure 9 shows the speedups of the 1 through 4-processor parallel versions of the programs, relative to the serial execution. The gray bars represent the speedups of the versions generated by Polaris Guide back-end and the black bars represent those of the versions generated through SC4.0 automatic parallelization.

The obtained 4-processor speedups range from 0.9 to 2.5. Parallel versions created by Polaris mostly outperformed those by the Sun parallelizer. In the cases with FLO52Q and EMBAR the overhead of the Polaris version between the serial and the 1-processor parallel variant, is visibly high. We refer to this as the *parallelization overhead*, which it is briefly discussed in Section 6.

In all cases, the speedup curves are almost linear, although we hardly reached the ideal speedup. Some programs such as FLO52Q have a considerable portion of code that could not be parallelized due to I/O. Also, the overall parallelization overheads may result in a degradation of performance, especially when there are many parallel loops with small execution time. We will discuss these results in more detail by looking at the performance of individual loops.

Performance results of individual loops

The performance figures of individual loops give us more insight into the behavior of the programs and the machine architecture. We measured the total execution time of each loop accumulated over repetitive runs in the program as well as the average execution time.

The overall execution time of MDG shows a good speedup compared to others, and the reason is clear in Figure 10. The execution time of MDG is dominated by one loop, INTERF_do1000. The 4-processor speedup of this loop shows a speedup of 2.26. The second largest loop, POTENG_do2000,is

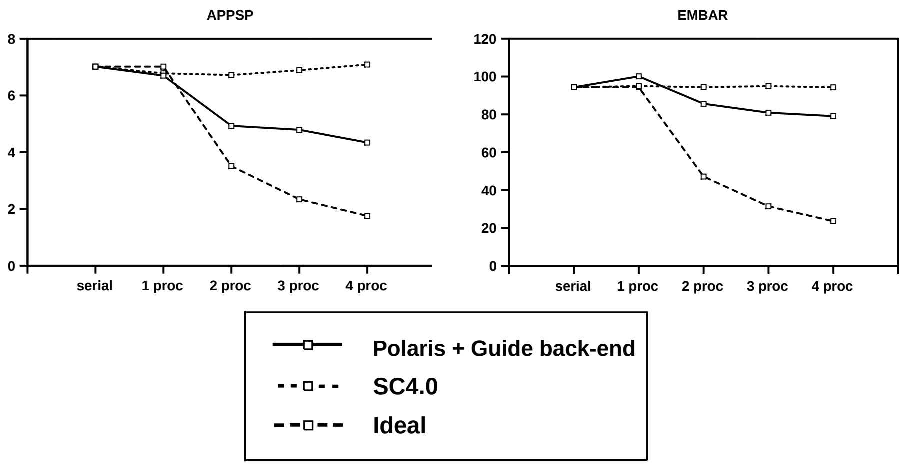


Figure 8: Execution time of different parallel versions of NAS kernels on SPARC20

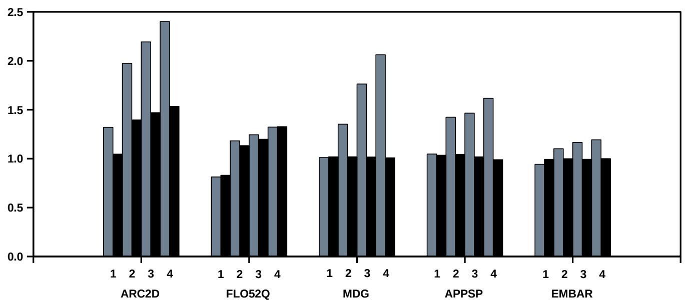


Figure 9: Speedups of the test programs relative to the serial execution on SPARC20. (gray: Polaris Guide back-end; black: SC4.0)

also parallelized with a speedup of 2.37, taking part in the high performance. On the other hand, SC4.0 was not able to parallelize these two loops, resulting in no speedup on multiple processors.

Another code that we studied is ARC2D. Figure 11 shows the parallel execution times of a few of the most time-consuming loops. Almost all of these loops are easy for compilers to parallelize. In fact, in studies with previous parallelizing compilers, we have found that all loops of this code were almost fully parallelized [BE92]. The Individual loop speedups are very high compared to the serial execution, but a close examination reveals that two loops, STEPFX_do210 and STEPFX_do230, run two to three times faster on one processor than those in the serial version. Compared to the parallel program run on one processor the speedups of the loops on two, three, and four processors are significantly lower. We attribute this "negative parallelization overhead" to the loop interchange transformation, which is applied by the back-end compiler to the parallel code but not to the serial code. It will be discussed further is Section 6.1.

FLO52 is another program with plenty of parallelism. Similarly to ARC2D, previous compilers have been successful in parallelizing this code. As can be seen in Figure 12, major loops were all parallelized with good speedups, except CPLOT_do30, which is a loop that handles file I/O. Several loops showed differences in the optimizations performed by the two compilers, however the overall difference in execution time is insignificant.

One of the two NAS kernel programs we considered, APPSP, executes for a short amount of time.

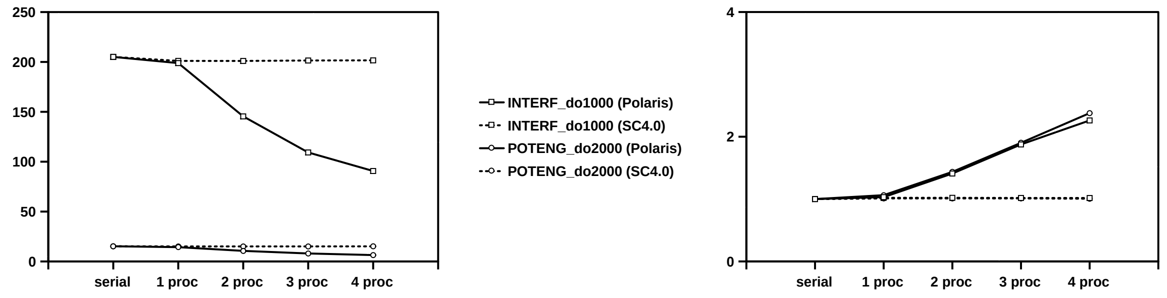


Figure 10: Execution times and speedups of individual loops in MDG on SPARC20

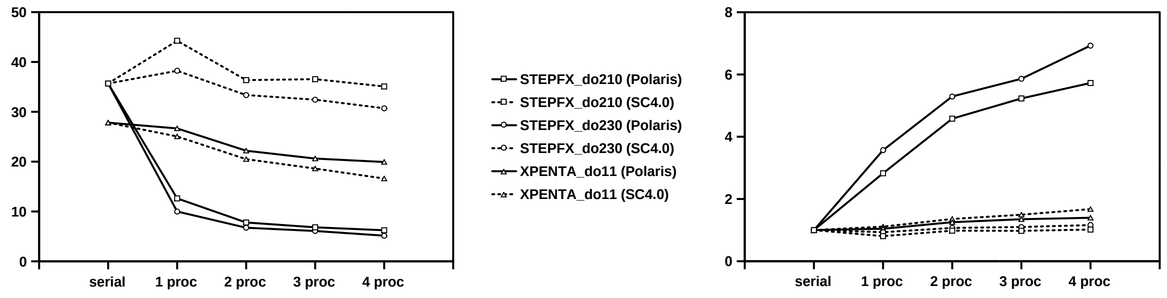


Figure 11: Execution times and speedups of individual loops in ARC2D on SPARC20

Nevertheless, the speedup characteristics are much the same. All three loops shown in Figure 13 are parallelized by Polaris, but SC4.0 parallelized several inner loops with these three loops, resulting in rather worse performance.

In the case of EMBAR, Polaris was not successful in parallelizing the most time-consuming loop, VRANLC_do120, due to data dependences. This seriously limits the overall speedup, although the second-most time-consuming loop, EMBAR_do140, was parallelized using the array reduction technique, yielding a speedup of 1.3. None of these loops were parallelized by SC4.0, so there was no speedup, as shown in Figure 14.

The results presented so far indicate that the usage of Guide as a portable directive language along with the Polaris parallelizing compiler is an efficient way to achieve high performance through automatic

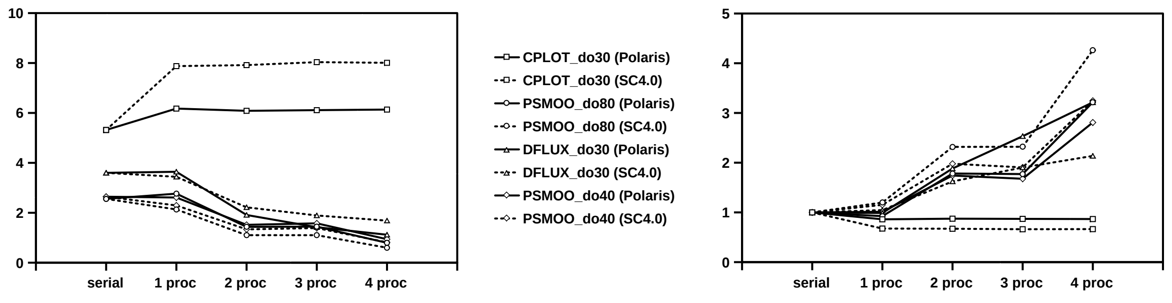


Figure 12: Execution times and speedups of individual loops in FLO52Q on SPARC20

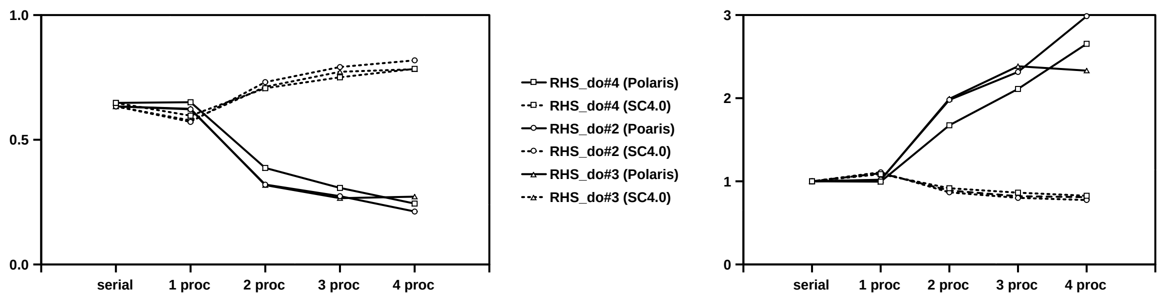


Figure 13: Execution times and speedups of individual loops in APPSP on SPARC20

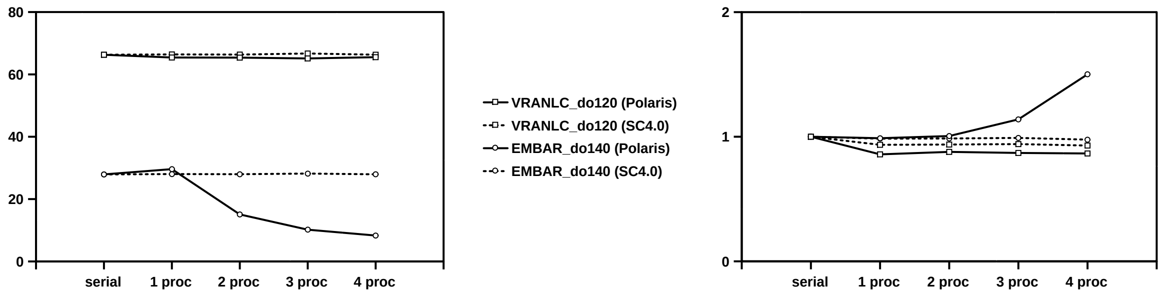


Figure 14: Execution times and speedups of individual loops in EMBAR on SPARC20

parallelization on the Sun workstation. However, we have also identified the performance of the memory and the I/O system to be serious bottlenecks [EPV96]. In Section 6.2 we will discuss this further.

5.3 Results on the SGI Challenge

Overall Performance Results

On the SGI Challenge machine we measured the execution time of the programs on one through four processors – the same as for the SPARC20 machine. The overall execution time of the Perfect Benchmark programs MDG, ARC2D, and FLO52Q is shown in Figure 15. In these graphs, dotted lines are the results from PFA, a native parallelizer on the SGI challenge machine, whereas solid lines represent the Polaris+Guide measurements.

The programs execute faster on the SGI Challenge machine than on the Sun workstation, but the improvement is not uniform over all programs. Also, ARC2D parallelized by PFA is faster than the Polaris version. Figure 16 presents the execution time of NAS kernels APPSP and EMBAR.

The charts show curves similar to those for the Sun workstation. However, the speedups on four processors are slightly lower. It should be noted that the effect of multiuser mode had more impact on our experiments on the SGI machine than on the Sun workstation. In fact our measurement, plotted in Figure 17, show significantly lower performance than results obtained previously in quasi-single-user mode [BDE+96].

Performance results of individual loops

In this section, we will examine the performance of the test programs in more detail. We consider the same loops that are discussed in Section 5.2.

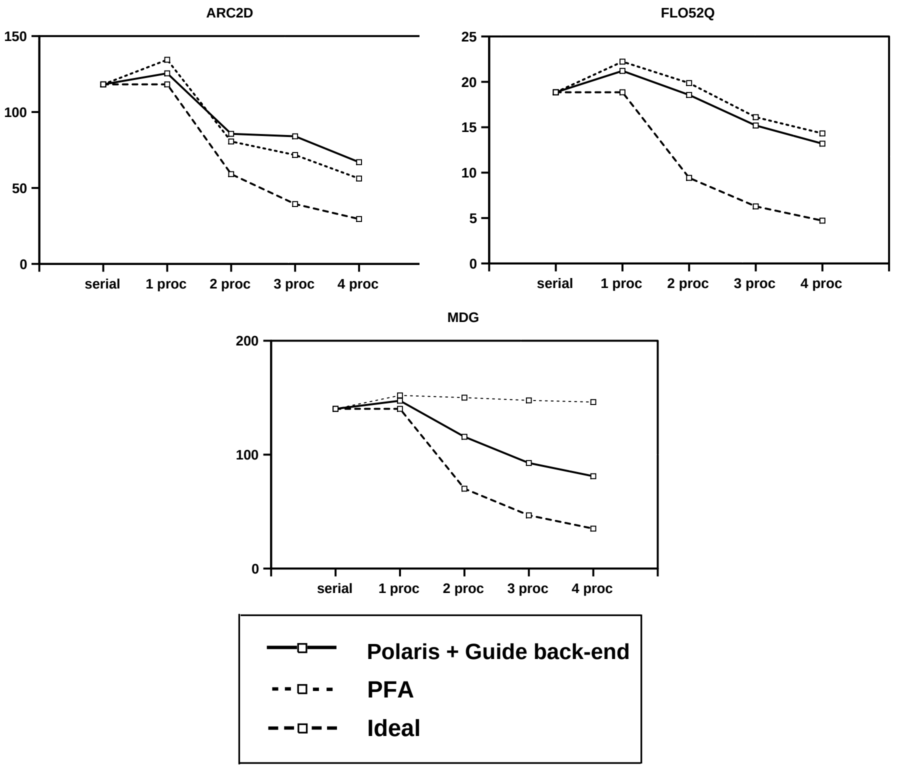


Figure 15: Execution time of different parallel versions of Perfect benchmarks on SGI Challenge

Figure 18 shows similar curves as Figure 10. The speedups on both machines are also almost identical. None of these loops were parallelized by PFA, which is obvious from the figure. The speedups of these loops on four processors on both machines are almost identical.

The performance of three loops in ARC2D is shown in Figure 19. The effect of the loop interchange are still present with STEPFX_do210 and STEPFX_do230, although they are not as pronounced as in Figure 11. This time, the PFA version benefits more from loop interchange. Also, they show speedup curves. All three loops were parallelized by both Polaris and PFA.

Two things are worth noting about the performance of FLO52Q on the SGI Challenge. First, the effect ofthe I/O loop is not dominant. On SPARC20, CPLOT_do30 is the most time-consuming loop, taking more than five seconds to execute, but here it is insignificant. As mentioned earlier, this shows that the I/O system on the Sun workstation could be improved significantly. On the other hand, the speedups of the other three loops on SPARC20 ranges from 2.8 to 3.2 on four processors, but on the Challenge machine they stay within the range of 1.4 to 2.7 on four processors. In this case, the Sun workstation shows better scalability for the chosen configuration and environment.

All three loops from APPSP shown in Figure 21 are parallelized by both Polaris and PFA. APPSP is parallelized very successfully. The situation is different with EMBAR. The most time-consuming loop, VRANLC_do120, remains serial in both cases. Polaris parallelized EMBAR_do140, but the speedup is less than two on four processors. In fact, like SC4.0, PFA was not able to parallelize any loops, leaving EMBAR as a serial program.

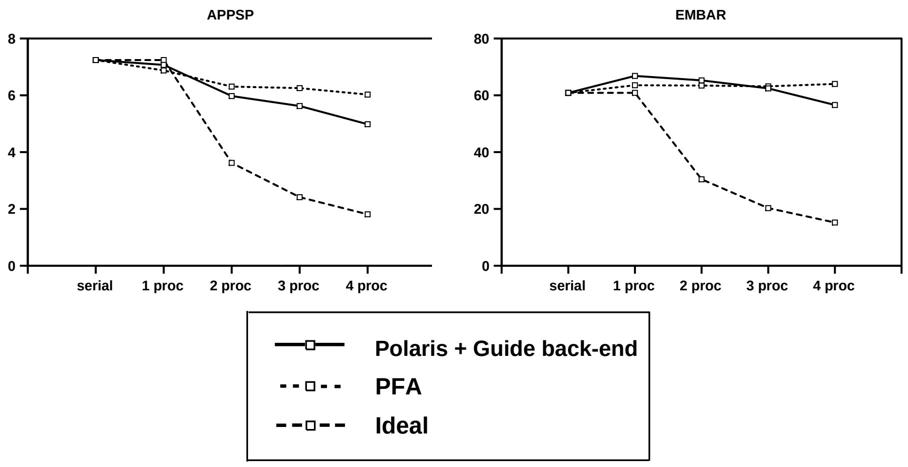


Figure 16: Execution time of different parallel versions of NAS kernels on SGI Challenge

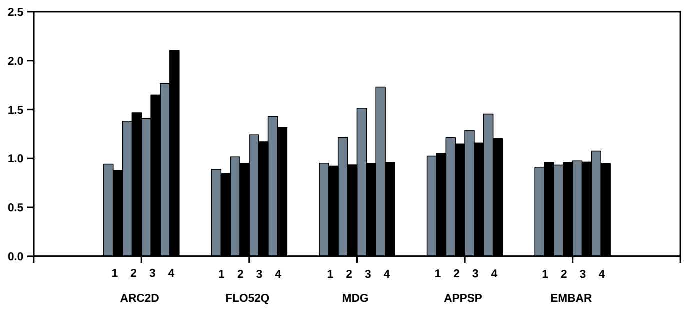


Figure 17: Speedups of the test programs relative to the serial execution on SGI Challenge. (gray: Polaris Guide back-end; black: PFA)

6 Ongoing Research

6.1 Improved code generation in the parallelizing compiler

As described earlier, we attribute to the loop interchange transformation the improvement of the parallel version of ARC2D on one processor over the serial version. The strategy chosen by the back-end compiler (SC4.0) for applying this transformation is unclear. It seems preferable to make decisions on what performance improving transformation to apply in the front-end, parallelizing compiler, where advanced program analysis and performance modelling capabilities are available.

Loop interchanging is a well-known technique [BENP93], but not yet applied by Polaris. The development of loop interchange strategies is an ongoing project. Here we report on measurements that have quantified the effects of the transformation. We have compiled the parallel version of ARC2D without applying loop interchange and compared its performance with the program with manually interchanged loops. The transformation was applicable in two distinct cases. The first case was to increase locality of reference by creating a stride of 1 in array accesses. This was applied to 7 loops in the program ARC2D: RHSC_do400, COEF24_do20, STEPFX_do210, STEPFX_do230, FILERX_do16, FILERX_do19 and FILERX_do20 with great success.

The second benefit from this manipulation is increased parallelism by exchanging an inner parallel loop with an outer serial loop, when such an exchange is legal. This increases granularity and allows

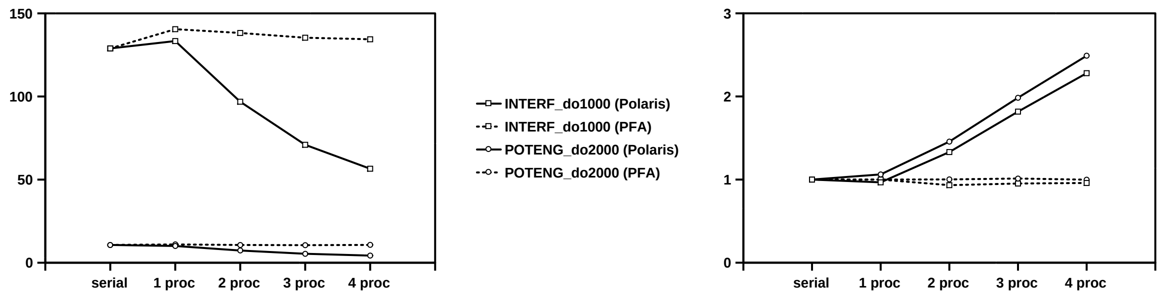


Figure 18: Execution times and speedups of individual loops in MDG on SGI Challenge

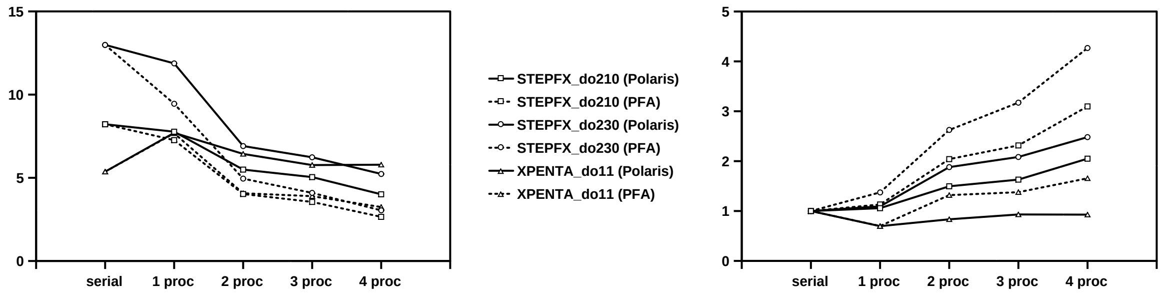


Figure 19: Execution times and speedups of individual loops in ARC2D on SGI Challenge

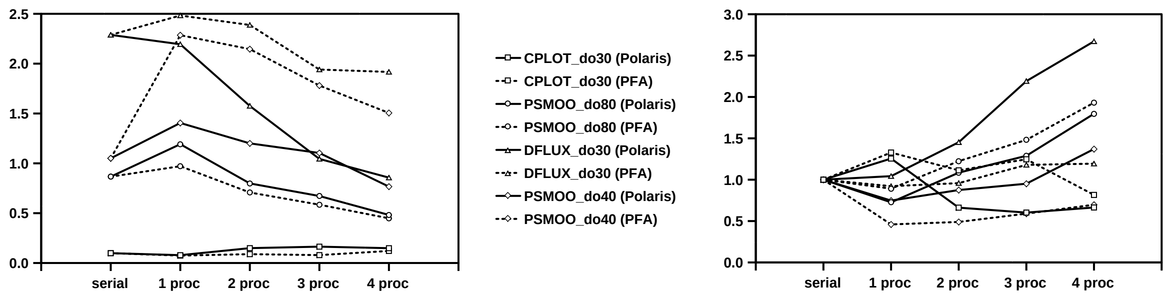


Figure 20: Execution times and speedups of individual loops in FLO52 on SGI Challenge

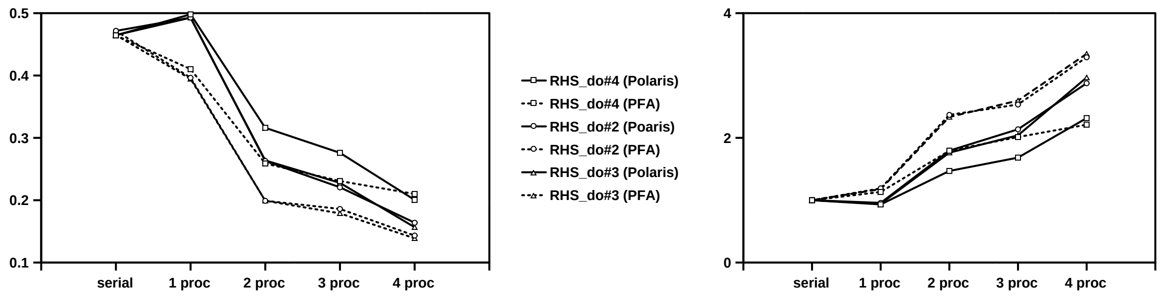


Figure 21: Execution times and speedups of individual loops in APPSP on SGI Challenge

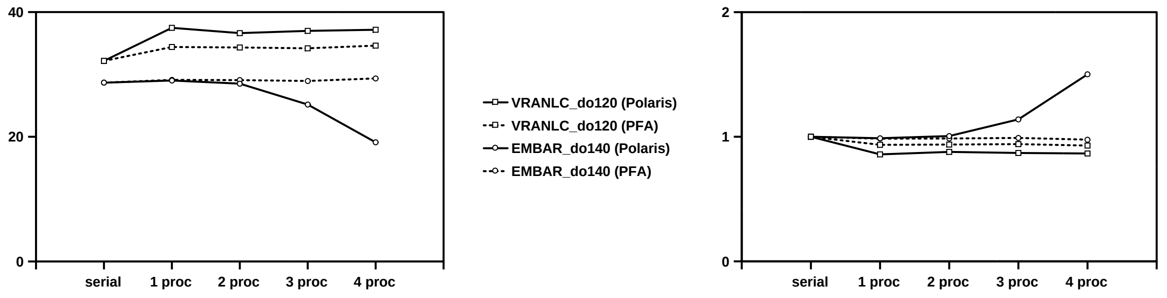


Figure 22: Execution times and speedups of individual loops in EMBAR on SGI Challenge

more work to be done in parallel, reducing the total startup overhead of repeatedly starting the inner loop. This transformation was applied to 3 loop nests in ARC2D: XPENT2_do3, XPENTA_do3, and XPENTA_do4. Each of these loops showed significant improvement when interchanged.

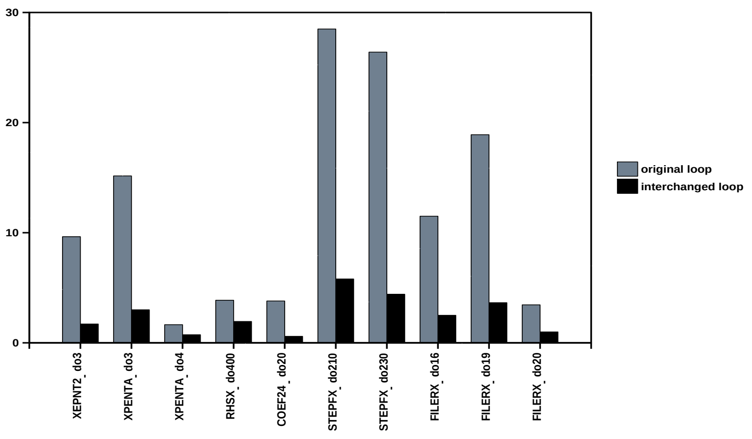


Figure 23: Execution Times of Interchanged Loops in ARC2D

Figure 23 presents the change in execution times for each of these modified loops. The loops STEPFX_do210 and STEPFX_do230, which were the two most time-consuming loops in the benchmark, greatly reduce their execution time due to the creation of stride 1 accesses. Their 4-processor execution times decreased by the factors 4.88 and 5.84, respectively. Both loops have an inner loop, which when interchanged enhances the spatial locality that each cache can exploit. This speedup is also present in the 1-processor execution, since it is a result of improving the cache behavior of each individual processor.

In XPENT2_do3 and XPENTA_do3 the effect of interchanging for the sake of increased granularity can been seen. Reducing the startup overhead and increasing the total work done in parallel allows these loops to reduce their parallel execution time by factors 5.77 and 5.26, respectively.

After interchanging these 10 loops, the overall execution time of the code dropped from 192 seconds to 97 seconds, a speedup of 2. This increased the program speedup with respect to the original, serial version significantly, from 1.6 to 3.2. We are currently implementing a Polaris pass that determines where loop interchanging is profitable and then applies the transformation.

6.2 Improved performance modeling

Understanding the detailed performance behavior is important for identifying applicable transformations. To this end, we have used a simple parallel loop model. It includes the notions of a *parallelization overhead* and a *spreading overhead*, both of which prevent a parallel loop from speeding up by the number of processors. Improving the understanding of these overheads is another important effort. In the following subsections, the analysis for the Sun architecture is given, although a similar analysis can also be done for the SGI.

The parallelization overhead represents the difference between the serial program and the 1-processor parallel program performance. It is mainly caused by the parallel loop startup latency, code inserted by transformations that may need to be performed for parallelizing the code, and the usually lesser degree of scalar optimizations that can be applied to the parallel code.

We have measured the loop startup latency on the SPARC20 to be typically 150 microseconds. For loops that execute serially in less than 0.5 milliseconds, parallelization is usually not profitable. Our measurements show that parallelization overheads for the significant loops in the program range up to 20%. For ARC2D we have seen apparent negative parallelization overheads. That is, the 1-processor parallel version runs faster than the serial loop. We attribute this effect to advanced optimizations that are applied to the parallel code only, although they could benefit the serial code as well. An example is the loop interchange transformation, as seen for the ARC2D code.

The spreading overhead limits the speedup of the parallel program when increasing the number of processors. The primary factor on our machine is the performance of the memory system. In order to model this effect, we have studied the amount of references to global and private variables. Private variables are touched by one processor only and they are likely to reside in the cache. In contrast, shared references are less likely to exploit the cache well because they will be referenced by different processors over the course of the program execution. Hence, the ratio between the amount of private and shared data can be an indicator of the expected degradation caused by memory traffic.

This simple model would explain some of the performance differences in our programs. For example, the loops is MDG have a large amount of private data, whereas in ARC2D, a significant portion of the references are shared. In order to validate this model more quantitatively we counted the number of private and shared references for the major parallel loops in ARC2D, and we plotted the ratio of private to total references, as shown in Figure 24.

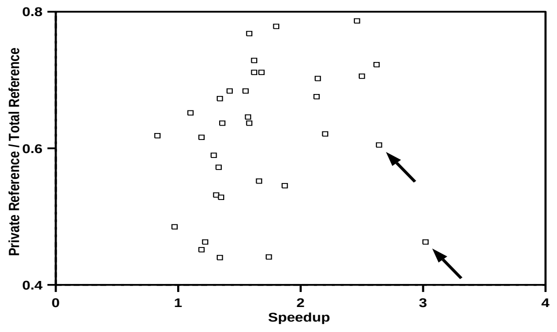


Figure 24: Private/Total References versus Speedup

A trend toward a higher speedup is noticeable as the ratio of private references to total references increases. However there are also a few points that deviate from this trend. We have looked at the source code of these loops (indicated by arrows in Figure 24) and have found that they contain code patterns that lead to good cache locality of shared variables. For example, the loop with the largest speedup of 3.0, GRADCO0_do4, shown in Figure 25, accesses many global variables, but the coef array is read-only shared and so they are more likely to be cached.

```
C$PAR PARALLEL LOCAL(K,J), SHARED(JLOW,JUP,KMAX,COEF,JMINU,JPLUS,PRSS)
C$PAR PDO
CSRD$ LOOPLABEL 'GRADCO0_do4'
      DO j = jlow, jup, 1
CSRD$ LOOPLABEL 'GRADCO0_do4/2'
       DO k = 2, (-1)+kmax, 1
        prss(j, k) = MAX(coef(jminu(j), k), coef(jplus(jplus(j)), k), co
     *ef(jplus(j), k), coef(j, k))
       ENDDO
      ENDDO
C$PAR END PDO
C$PAR END PARALLEL
```

Figure 25: The GRADCO0_do4 Loop Nest

The other outlying point in Figure 24 represents CALCPS_do1, Figure 26. This loop demonstrates the effect of a stride of 1 in enhancing cache performance. CALCPS_do1 reads from the read-only q array often, and since the inner loop iterates on j, successive iterations on a given processor access this array, and others with a stride of 1. This greatly enhances the locality of reference exploited by the cache. In fact, in the previous example, GRADCO0_do4 does not access the coef array with a stride of 1, however if the j and k loops are interchanged this is corrected, and the execution time is reduced by 16%. The results of similar manual modifications on the most time consuming loops of ARC2D are presented in the next section.

```
C$PAR PARALLEL LOCAL(J,K), SHARED(KMAX,JMAX,GAMI,Q,PRESS,GAMMA,SNDSP)
C$PAR PDO
CSRD$ LOOPLABEL 'CALCPS_do1
      DO k = 1, kmax, 1
CSRD$ LOOPLABEL 'CALCPS_do1/2'
       DO j = 1, jmax, 1
        press(j, k) = gami*(q(j, k, 4)+(-1)*((0.5*(q(j, k, 2)**2+q(j, k,
     * 3)**2))/q(j, k, 1)))
        sndsp(j, k) = SQRT((press(j, k)*gamma)/q(j, k, 1))
       ENDDO
      ENDDO
C$PAR END PDO
C$PAR END PARALLEL
```

Figure 26: The CALCPS_do1 Loop Nest

We can conclude that program sections that contain many private references or cacheable global references can take advantage of the 4 processors in our machine, whereas for other loops the available memory bandwidth is a serious performance limitation.

The data presented in Figure 24 is only a starting point. Understanding the cache and memory effects that influence the performance more quantitatively is critical, as it will help us identify not only compiler transformations but necessary architectural improvements. Both are important ongoing projects.

6.3 Suggested Improvements to the Guide Directive Language

The parallelism that Polaris detected in our programs could be expressed adequately in Guide. Nevertheless the language interface between Polaris and the backend compiler could be improved. The following suggested additions to the parallel directive language would help reduce the complexity of Polaris' interface pass. They would also help to express parallel programs more concisely for users who wish to do so explicitly. Our findings match those of other experiments on different machines [PE95].

Processor identification: The availability of an efficient function to query the current processor number is important. This number is used in many array subscript expressions. Therefore obtaining it should be not significantly more expensive than accessing a register. This processor identification could be provided through an intrinsic function, which is the case on some other machines.

Array reductions: The parallelization of array reductions is among the most important compiler transformations. The Guide directives allow only for scalar reductions. In array reductions the range of an array being accumulated into is often unknown at compile time. Because of this, dynamically-sized private arrays need to be available as a basis for implementing array reductions.

Dynamically-sized private arrays: The size of arrays or array sections that need to be privatized may not be a compile-time constant. The current directive language does not accept such arrays on private lists of parallel loops. Because of this, Polaris had to expand these arrays, allocate them in shared space, and use the processor number as an index. Placing expanded private arrays in shared space is likely to have a negative impact on the locality because it obscures the fact that the array references do not need to be coherent.

Processor-private data: These are private variables whose lifetime is longer than the duration of a single loop. The availability of such variables could further increase the efficiency of the generated code. For example, expensive functions for processor identification could be called at the beginning of the program and be kept in private storage. Furthermore, in other studies [PE95] we have seen the need for processor-private data for expressing data structures that are placed in private storage for the duration of the program. Guide provides one means for this: a Fortran common block attribute that places the common data in processor-private space. The same attribute for non-commonblock data is desirable.

6.4 Performance Analysis Tools

Tools for understanding the performance of a parallel program are important complements of a parallelizing compiler. The tools that we used in our study will be equally useful for a somewhat knowledgeable programmer who may take advantage of the capabilities of an optimizing compiler at first, but then wishes to understand and tune the program performance. Also, researchers who would like to add to the existing automatic parallelization techniques will find these tools useful in identifying more possible parallel loops.

Our tools facilitated the collection, manipulation, and visualization of information from various sources. These are

- Loop-by-loop timing information. We have instrumented our programs with Polaris and collected timing information using a trace library developed in previous projects [EM93].
- Loop iteration count information was collected in the same way. The Polaris instrumentation capability can insert calls to a statistics library, which collects and summarizes the number of iterations for each loop as the program executes.
- Reference count information was needed to find the private and global memory reference ratios. For this purpose we have developed a Polaris pass that counts data references in each loop body. Iteration count information is then used to compute the ratios for larger program sections.

We used a spreadsheet for manipulating and visualizing this information. The tool allowed us to combine the various sources of data, compute statistics, build ratios, and create graphic displays in a variety of forms. For example, the graphs presented in this paper were generated with these tools.

Polaris, as a general infrastructure for analyzing and manipulating programs, provides a variety of facilities for extracting other useful informations from programs. We are currently developing tools based on these facilities that can help better understand the program structure for performance tuning and enable more applicable parallelization techniques.

- Calling structure information. In identifying possible parallel section of a program, one needs to know which loops to focus on. Nested subroutines, functions, and inner loops make this task not so trivial. We have developed a facility that systematically displays this information.
- Range propagation information helps identify privatizable arrays. This information proves to be useful in many cases in which a programmer uses a large block of memory to simulate dynamic memory allocation.

The information provided by these facilities can also be incorporated into the spreadsheet mentioned above. A user, then, can combine the information that he or she needs from this data set. To develop a tool that will enable convenient gathering and displaying of these informations is the object of another one of our ongoing projects.

7 Conclusion

In four out of five programs, the Polaris parallelizing compiler was able to recognize all time-consuming loops as parallel. This means that in many cases, the user of parallel machines does not need to deal with the issue of searching for parallelism in ordinary scientific and engineering programs. Compilers can express this parallelism in the Guide language, for which a portable implementation across many SMP platforms exists. This directive languages includes most of the feature we needed in our experiments to express parallelism directly on multiprocessors from both Sun and SGI.

However, there remains substantial work to make parallel processing on SMPs widely-usable. First, the state of the art of commercial compilers is significantly below the capabilities demonstrated in the Polaris project. Substantial technology transfer will need to happen in order to achieve the successful automatic parallelization that we were able to prove.

Second, although the identification of parallelism can be automated to a high degree, there is still significant room for improving the parallel performance of the programs. It would not be correct to assume that SMP machines are easy targets for executing parallel programs. In our measurements we have found that on both the SPARC20 and the SGI Challenge machine, fully parallel programs can show speedups of less than two on four processors in low-load multi-user mode. We have identified the memory systems and the I/O systems to contribute significantly to this performance limitation. Hence, better performance modeling leading to a better understanding of the performance behavior remains not only an issue for complex, distributed-memory machines, but also for shared memory multiprocessors.

References

[BDE$^+$96] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Advanced program restructuring for high-performance computers with polaris. *IEEE Computer*, December 1996.

[BDG$^+$93] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpole. PVM: Experiences, current status and future direction. In *Proceedings of Supercomputing '93*, page 765, 1993.

[BE92] William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions of Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[BENP93] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[EM93] Rudolf Eigenmann and Patrick McClaughry. Practical Tools for Optimizing Parallel Programs. *Presented at the 1993 SCS Multiconference, Arlington, VA*, March 27 - April 1, 1993.

[EPV96] Rudolf Eigenmann, Insung Park, and Michael J. Voss. Are parallel workstations the right target for parallelizing compilers? In *Processdings of the Ninth Workshop on Languages and Compilers for Parallel Computers*, August 96.

[FHP$^+$94] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–586, October 1994.

[For93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical report, Rice University, Houston Texas, May 1993.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, Tennessee, May 1994.

[fSA] National Center for Supercomputing Application. *NCSA POWER CHALLENGEarray User Guide: Overview*. Web document. http://www.ncsa.uiuc.edu/Pubs/UserGuides/Power/PCABook2_8.html.

[Hoe96] Jay Hoeflinger. Polaris developer's document. Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, 1996. http://www.csrd.uiuc.edu/polaris/polaris_developer/polaris_developer.html.

[KA] Kuck and Inc. Associates. *Control Structure Based Parallelism.* Web document. http://www.kai.com/hints/csbparallelism.html.

[Kuc96] Kuck and Associates, Inc. *Guide^{TM} Reference Manual, Version 2.1*, September 1996.

[PE95] Bill Pottenger and Rudolf Eigenmann. Targeting a Shared-Address-Space version of the seismic benchmark Seis1.1. Technical Report 1456, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., September 1995.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 444–448, 95.

[PP84] M. Papamarcos and J. Patel. A low overhead coherent solution for multiprocessors with private cache memories. In *Proc. of the 11th International Symposium on Computer Architecture*, 1984.

[TP93] Peng Tu and David Padua. Automatic Array Privatization. In Utpal BanerjeeDavid GelernterAlex NicolauDavid Padua, editor, *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.

[VPE96] Michael J. Voss, Insung Park, and Rudolf Eigenmann. On the machine-independent target language for parallelizing compilers. In *Processdings of the Sixth Workshop on Compilers for Parallel Computers (CPC'96), Aachen, Germany*, December 96.

[Wp96] SPARCstation 20 Series with SuperSPARC and SuperSPARC-II Processors. Sun Microsystems, Inc., 1996. http://www.sun.com:80/products-n-solutions/hw/wstns/jtf_ss20.html.