Are Parallel Workstations the Right Target for Parallelizing Compilers? *

Rudolf Eigenmann Insung Park Michael J. Voss

Purdue University, School of Electrical and Computer Engineering

Abstract. The growing popularity of multiprocessor workstations among general users calls for a more easy-to-understand approach to parallel programming. Providing standard, sequential languages with automatic translation tools would enable a seamless transition from uniprocessors to multiprocessor workstations. In this paper we study the success and limitations of such an approach. To this end, we have retargeted the Polaris parallelizing compiler at a 4-processor Sun SPARCstation 20 and measured the performance of parallel programs. Here, we present the results from six of the Perfect Benchmark programs along with our analysis of the performance and some of the issues brought up during the experiments. Our research will help answer some of the questions that have been posed by both users and manufacturers concerning the practicality and desirable characteristics of parallel programming in a workstation environment.

1 Introduction

From an end-user's point of view, parallelizing compilers are not yet consistently useful tools. There have been significant improvements of these tools over the past few years [BDE+96, HAA+96]. However, they are not yet in a state where users can write programs in standard sequential languages and expect good performance on most applications and parallel machines. To achieve such performance, programmers have to be willing to write in architecture-specific parallel languages. The user community of expensive high-performance computers often includes such expert programmers.

The situation is different on parallel workstations. The user community of these machines is more diverse and includes a majority of non-computer experts, who may not be able to invest the time in learning architecture-specific parallel languages. The burden, then, is on compilers, which face the challenge of translating programs written by a layman programmer into efficient machine code. For scientific and engineering applications we have measured the success rate of parallelizing compilers in meeting this challenge to be about one in two programs [BDE+96]. Therefore, given the fact that workstation manufacturers

* 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.

currently offer multiprocessors at a price that is only little over the base price of a uniprocessor system, we hypothesize that the availability of parallelizing compilers on such machines is of very practical value.

In this paper we test this hypothesis. Although it may seem straightforward to put parallelizing compilers into a toolbox of parallel workstation, there are a number of open questions: What performance can we expect of applications that have been successfully parallelized on other parallel machines? What are the reasons that the performance is the way it is? Do we need additional optimization techniques to compile for multiprocessor workstations? Are there architectural limitations that impede the performance of parallel or parallelized programs on this class of machines, and can we give recommendations to their manufacturers for removing these limitations?

In order to answer such questions, we have retargeted the Polaris parallelizing compiler at a 4-processor Sun SPARCstation 20 and we have measured the performance of parallel programs run on this system. Section 1.1 will give some details of the architecture of our target machine. Section 2 gives an overview of the Polaris compiler and describes the modifications that we have made so that it generates optimized code for the Sun multiprocessor. In Section 3 we present measurements of parallel programs run on our machine along with our analysis. In Section 3.3, we present open issues and ongoing work.

1.1 The Sun SPARC 20 Multiprocessor architecture

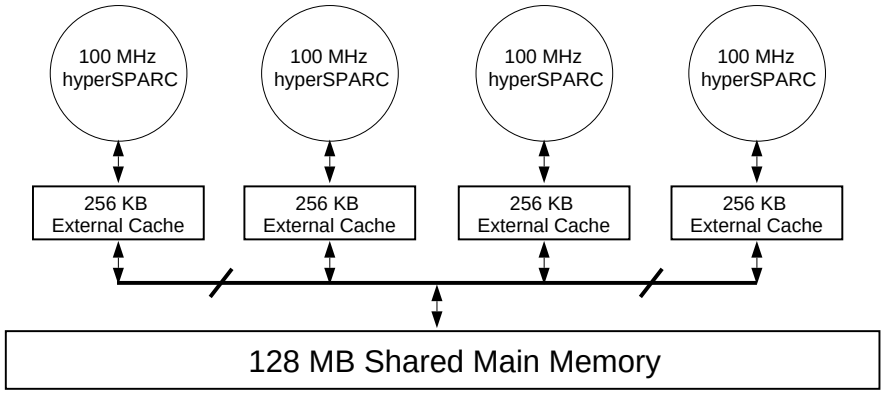

Fig. 1. The Sun SPARCstation20 architecture

Figure 1 shows the Sun Sparc20 workstation used in our experiments. It consists of four 100 MHz hyperSPARC processors that share a single global memory space [Sun96b]. 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. The write-through scheme is

used only for the block copy and fill operations provided in the hyperSPARC instruction set. In all other cases, 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.

1.2 Expressing parallel programs on the SPARC20 MP

The Sun FORTRAN77 Compiler version 4.0 (SC4.0) [Sun96a, GL96] was chosen as the backend compiler for our study. This compiler is an early release of Sun Microsystem's Fortran compiler that supports language features for expressing parallelism in the form of parallel DO loops. All of the information required for specifying such loops are conveyed by two directives, **C$PAR DOSERIAL** and **C$PAR DOALL**.

C$PAR DOSERIAL : The DOSERIAL directive, when placed before a DO loop, specifies that the loop immediately following it is to be executed sequentially. By default, loops labeled with neither DOSERIAL nor DOALL will be serialized, and therefore this directive is optional.

C$PAR DOALL : The DOALL directive, when placed before a DO loop, specifies that the loop immediately following it is to be executed in parallel. In the case of a parallel loop, it is also required to classify variables within the loop as private, shared, or reduction variables. SC4.0 provides three subdirectives for this purpose, **SHARED()**, **PRIVATE()**, and **REDUCTION()**.

The **C$PAR DOALL PRIVATE(***variable_list***)** subdirective specifies variables within the loop body that are to be treated as private to each processor. By default, all scalar variables are assumed to be private.

The subdirective **C$PAR DOALL SHARED(***variable_list***)** specifies which variables within the loop body are to be treated as shared by all processors. By default, all arrays are assumed to be shared. We have also found that shared read-only scalar variables need not be declared as shared.

Finally, the subdirective **C$PAR DOALL REDUCTION(***variable_list***)** specifies which variables are used for reduction operations. The Sun compiler only allows *scalar reductions*, and therefore only scalars are included in the argument list. Scalar reduction operations are statements within a loop of the form `sum = sum + expr`, where `expr` is a loop-variant term and `sum` is the reduction variable. *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.

2 Retargeting Polaris at the SPARC20 Multiprocessor

2.1 Polaris overview

Polaris is a parallelizing compiler, originally developed at the University of Illinois. As illustrated in Figure 2, 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.

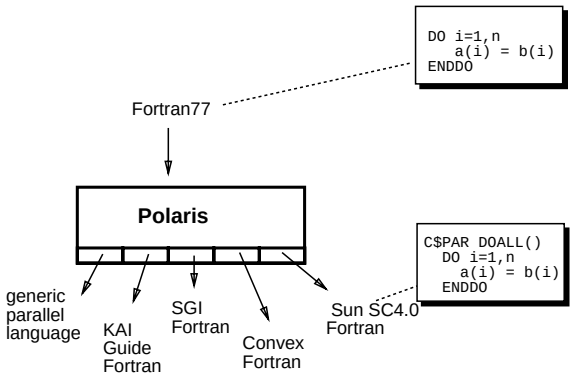

Fig. 2. Overview of the Polaris parallelizing compiler

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 placed "private to the current loop". Figure 2 shows several other output languages that Polaris can generate, such as the directive language available on the SGI Challenge machine series and the Sun SC4.0 Fortran directive language 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 approximately 170,000 lines of C++ code. A basic infrastructure provides a hierarchy of C++ classes that the developers of the individual compilation passes can use for analyzing and manipulating 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 available as 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.

2.2 Polaris modifications and enhancements

The Sun SPARC20 multiprocessor is a shared-memory architecture, 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 platform was fairly simple. Most of the additions to the code were localized to the final *output pass*, since it is here that the generic directives used in the internal representation are mapped to architecture-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, shared and private variables, and reduction and non-reduction variables. These generic directives could be directly mapped to their corresponding Sun directives.

One issue that we had to resolve in the output pass for the Sun machine, was to provide an efficient means of identifying the current processor on which a particular loop iteration is executed. The next section describes why this is necessary. The Sun compiler provides a function that can return the current processor number at runtime. However, Polaris-generated programs read this number once or even several times per loop iteration, and so a subroutine call would cause too much overhead.

Another issue we had to deal with was due to a temporary problem with the Sun backend compiler, which did not allow multiple directive lines. Because of this, we had to shorten the directive lines as much as possible. We did this by leaving out variables from the private and shared lists that had the proper default attributes.

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 transforms 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 array elements involved in an array reduction operation (array `A` in our example) cannot be determined at compile time. In the case of such reductions, Polaris provides a local copy of the array to each

```
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

Fig. 3. Array Reduction operation

processor, performs the accumulation operations of this array in the now fully parallel loop, and then recombines 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 Sun directive language by listing the arrays on the DOALL PRIVATE list. An array whose size is not known at compile time cannot be declared private in this way. Because of this, Polaris expands these arrays by a dimension equal to the number of processors, allocates them in shared space, and uses the processor identification as an index, as shown in Figure 4. Dynamic allocation of shared arrays is supported in the Sun Fortran dialect.

```
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

Fig. 4. Array privatization through *expansion*

As mentioned above, there is no efficient way of getting the value `my_proc_id()` used in these transformations. Polaris resolves this problem by stripmining the loop with 4 (=`number_of_processors`) outer iterations and then using the outer loop index instead of the processor number. This is shown in Figure 5. We have used this transformation in both cases where the processor_id would be needed: parallel loops with reduction operations and private arrays.

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

(a) With Function

```
PARALLEL DO proc = 1,number_of_processors
 DO J=proc,M,number_of_processors
   ... A(x,proc) ...
 ENDO
ENDDO
```

(b) Through Stripmining

Fig. 5. Identifying the current processor

3 Parallel program performance

3.1 The experiment

In our experiments, we have studied several programs in three forms: the serial version, the version produced by Polaris, and the version parallelized by SC4.0. 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. To ensure consistent results, we usually ran the programs several times under the same environment and checked the range of execution time before we obtained the timing measurement.

In some cases we had to hand-modify the Polaris-generated code in order to resolve compatibility problems between Polaris and the back-end compiler. Since the back-end compiler is a preliminary release of Sun's Fortran SC4.0 compiler, we expect these problems to be resolved in the next release.

The Fortran SC4.0 compiler also includes a capability to do automatic parallelization. We will compare the performance of programs parallelized by Polaris with the performance of this compiler. The comparison will be interesting because the SC4.0 compiler is a commercially-available optimizing compiler of the newest generation. Hence, it allows us to compare Polaris with the latest industrial state of technology.

During the experiments, we did not enforce a single-user mode, however, we reduced the load on the machine as much as possible, In this way, we were able to obtain consistent timing while maintaining an ordinary workstation environment.

3.2 Results

Overall Performance Results This section presents the experimental results obtained through various runs of the Perfect Benchmark programs BDNA, TRFD, ARC2D, FLO52, MDG, and OCEAN.

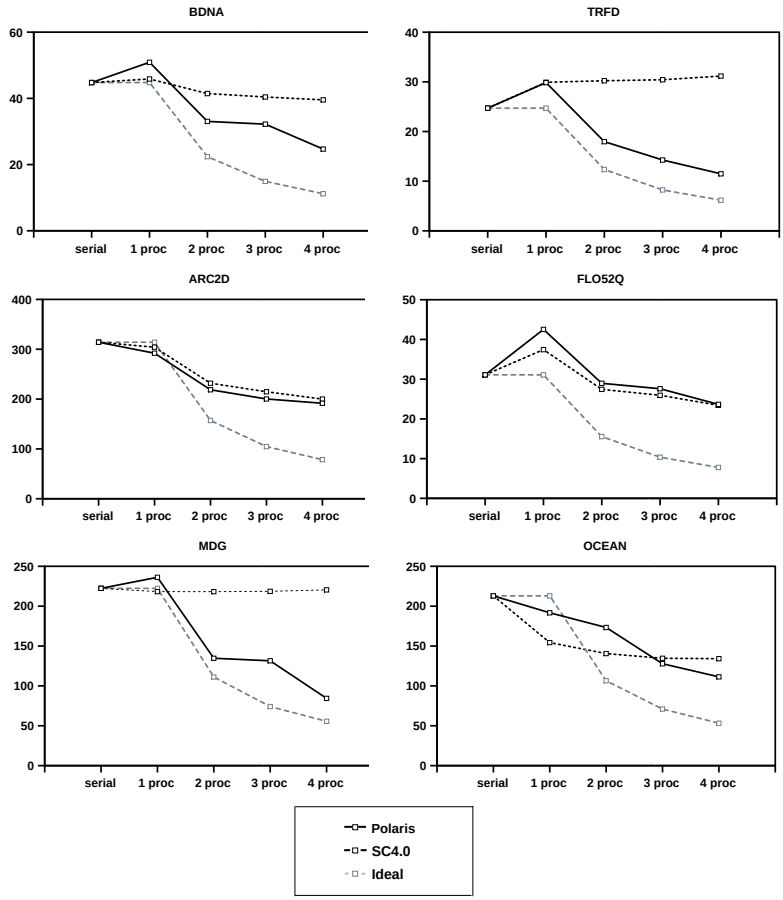

Fig. 6. Execution time of the programs parallelized with Polaris and SC4.0

Figure 6 shows the overall timing of the serial and 4-processor parallel execution of these programs. The curves correspond to programs parallelized with

Polaris and with the Sun parallelizing compiler, respectively. For comparison, the "ideal" curve shows $\frac{serial\_execution\_time}{number\_of\_processors}$.

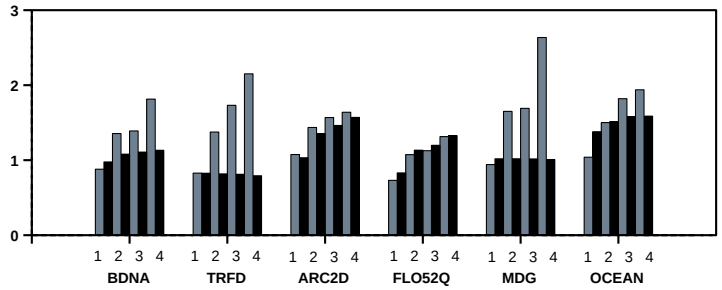

Fig. 7. Speedups of the test programs relative to the serial execution. (gray: Polaris; black: SC4.0)

Figure 7 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 and the black bars represent those of the versions generated through SC4.0 automatic parallelization.

The obtained 4-processor speedups range from 0.8 to 2.6. Parallel versions created by Polaris outperformed those by the Sun parallelizer, although, in the case of BDNA, FLO52, and MDG, the overhead of the Polaris version, when going from the serial to the 1-processor parallel version, is higher. We refer to this as the *parallelization overhead*. The Polaris versions of BDNA and MDG show a relatively small performance increase from 2 to 3 processors. This is due to our method of stripmining, which we will discuss in the next section.

Input/output loops in BDNA take up about 8 seconds of the execution time. These loops are not parallel. Since the total execution time of BDNA is about 45 seconds, Amdahl's law limits the speedup to 45 / (37 / 4 + 8) = 2.6. The number we get from our experimental results is 1.8. TRFD shows similar difference between the expected speedup and the experimental result. In MDG, the second most time-consuming loop, POTENG_do2000, could not be run in parallel due to a limitation of the back-end compiler, mentioned earlier. This restricts the speedup to 3.3 while the measured speedup was 2.6. 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 average execution time. In BDNA, the execution time of the entire program is dominated by three loops:

ACTFOR_do240[2] , ACTFOR_do500, and RESTAR_do15. Figure 8 shows the execution times and speedups of these loops, in the same terms as for Figure 6. RESTAR_do15 handles sequential file I/O and it is not parallelized. Note that this loop becomes dominant in the overall execution time, limiting the overall speedup. In order to parallelize the other two loops, array privatization and parallel reduction transformations had to be performed. Polaris parallelized both ACTFOR_do240 and ACTFOR_do500. In contrast, the Sun SC4.0 optimizer parallelized neither ACTFOR_do240 nor ACTFOR_do500. Only one inner loop within ACTFOR_do240, ACTFOR_do235, was parallelized by the Sun compiler. Hence, as shown in Figure 8, there is only a slight improvement in the execution time of ACTFOR_do240 in the Sun compiler version, and no improvement in the execution time of ACTFOR_do500.

Fig. 8. Execution times and speedups of individual loops in BDNA

Both loops ACTFOR_do240 and ACTFOR_do500 perform reasonably well on our machine. However, there is no increase in performance from 2 to 3 processors. The reason for this is that the loops are stripmined (cf. Fig. 5), with the outer, parallel loop iterating four times. If we use three processors, one processor will have to do two iterations – the same as for two processors.

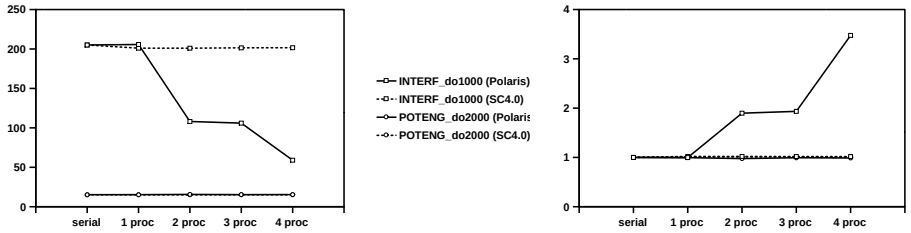

Fig. 9. Execution times and speedups of individual loops in MDG

[2] Our notation means the loop with label 240 in subroutine ACTFOR.

This effect is also visible in the experiment with MDG, as seen in Figure 9. The execution time of MDG is dominated by one loop, INTERF_do1000, which is stripmined in the same way. The 4-processor speedup of this loop shows an excellent 3.47. The loop POTENG_do2000 was not run in parallel because a problem in the back-end compiler, mentioned earlier. Polaris detected both loops as parallel. In contrast, the Sun parallelizer did not detect parallelism in either loop, resulting in no overall speedup.

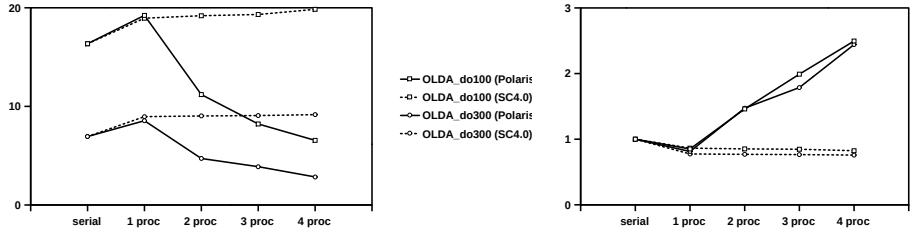

Fig. 10. Execution times and speedups of individual loops in TRFD

In TRFD, there are only two loops that have significant effects on the execution time: OLDA_do100 and OLDA_do300. Parallelization of these loops requires the substitution of generalized induction variables [PE95] and a dependence test on non-linear subscripts [BE94]. They were successfully parallelized by Polaris, but not by the Sun compiler. Instead, the Sun compiler parallelized several loops within these two loops. However, the execution times of these loops is so small that their parallelization is not profitable. The execution times and speedups are plotted in Figure 10.

Another code that we studied is ARC2D. The overall performance of ARC2D is less than that of BDNA, MDG, and TRFD. Moreover, the major loops parallelized by Polaris in BDNA and TRFD show good speedups while the loops in ARC2D, as presented in Figure 11, do not. Individual loop speedups are within the range of 1 to 2 times the serial version. Figure 11 shows the parallel execution times of a few of the most time-consuming loops. Unlike the above three programs, where there are only a few loops dominating the execution time, ARC2D has a large amount of smaller loops that contribute to the overall execution time. 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]. Our performance curves show that fully parallel loops do not necessarily guarantee good speedup. We have identified the performance of the memory to be the primary bottleneckv6 . We have found that loops that access mostly private data can take advantage of the cache and perform well, whereas loops dominated by shared accesses usually perform significantly worse.

FLO52 is another program with plenty of parallelism. As for ARC2D, previ-

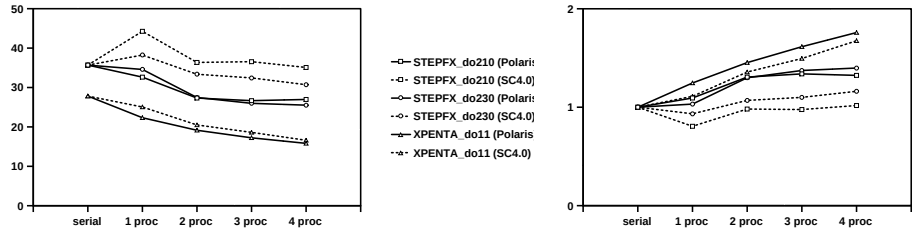

Fig. 11. Execution times and speedups of individual loops in ARC2D

ous compilers have been successful in parallelizing this code. FLO52 is the only program where the Sun compiler matches, and even slightly outperforms Polaris. As can be seen in Figure 12, major loops were all parallelized with speedups between 2.0 and 3.2, 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.

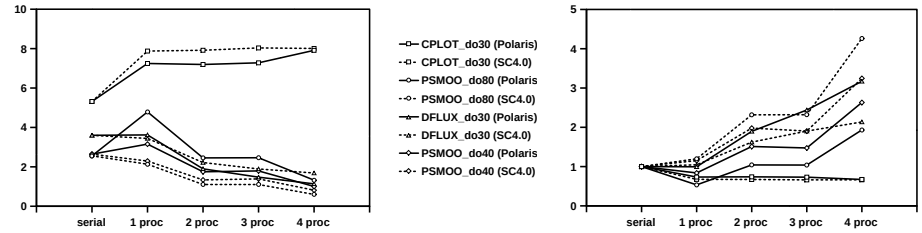

Fig. 12. Execution times and speedups of individual loops in FLO52

Finally, Figure 13 shows three of the most time-consuming loops in OCEAN. Polaris was not able to parallelize the most time-consuming loop, FTRVMT_do109. Instead, it parallelized three inner loops. The Sun compiler did not parallelize either of these loops, but the execution time was reduced by 20 seconds, nevertheless. Such "negative parallelization overheads" can be caused by the back-end compiler applying different scalar optimizations to the serial and parallel code. The same effect can be seen in the other two loops shown. Because of this, the Polaris version does not execute faster than the SC4.0 version until three or more processors are used, as can be seen in Figure 6. Notice also that the speedup for FTRVMT_do109 in the Polaris version is not as good as for the other two loops. However, it turns out that FTRVMT_do109 is parallel and its performance can be improved. A brief discussion on this will be presented in the next section.

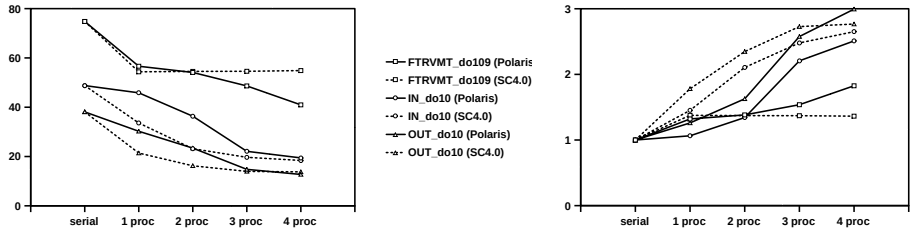

Fig. 13. Execution times and speedups of individual loops in OCEAN

3.3 Ongoing Work

In order to identify additional compiler transformations that could improve performance on our machine, we are currently hand tuning and studying the Polaris-generated codes. The most important such transformation, identified so far, is loop interchanging. Loop interchanging is a well-known technique [BENP93], but not yet applied by Polaris. 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. The second case was to increase parallelism by exchanging an inner parallel loop with an outer serial loop (when such an exchange is legal). This increases granularity and allows more work to be done in parallel, reducing the total startup overhead of repeatedly starting the inner, parallel loop.

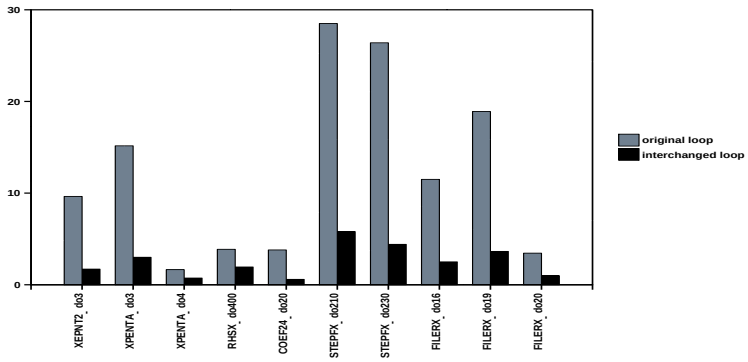

Fig. 14. Execution times of interchanged loops in ARC2D

Figure 14 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 would also be 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.

Another improvement was possible in OCEAN. The most time-consuming loop, FTRVMT_do109, consists of one loop enclosing three inner loops. As mentioned earlier, Polaris parallelized these three loops. We have found that the outer loop can be run in parallel as well, reducing the total execution time of FTRVMT_do109 down to 32 seconds. As the result, the overall speedup of OCEAN increased from 1.91 to 2.07. Recognizing this situations requires enhancements to the symbolic range propagation techniques of Polaris [BE95], which is another ongoing effort.

4 Conclusions

Our results show that compilers can parallelize programs successfully for multiprocessor workstations. Given a machine with balanced resources, we can expect a good return from investing into a small number of workstation CPUs. For many programs, a user can gain substantial performance without having to become involved in the parallel program design. In this sense, parallel processing has become a mature technology that can serve a wide user community and parallel workstations are the right target.

However, in the machine used in our experiments, the necessary balance of resources is not yet achieved. We have identified the performance of the memory and the I/O system to be serious bottlenecks, which restrict the range of well-performing programs to those who have a high degree of locality and insignificant disk traffic.

In the past, parallel programming has not been very popular among workstation users. Because of this, there is a lack of information on how well parallel programs perform, and what languages, compilers, and tools we need. Our study has tried to fill this very void. More such data is necessary. It will enable a field that is perhaps the first really practical application of parallel computing.

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.

[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.

[BE94] 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.

[BE95] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, April 1995.

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

[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.

[GL96] Vinod Grover and Michael Lai. Directives for SC4.0 Fortran and Fortran MP. Technical report, Sun Microsystems, Inc., 1996.

[HAA+96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Getting performance out of multiprocessors with the SUIF compiler. *IEEE Computer*, December 1996.

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

[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.

[Sun96a] Sun Microrsystems, Inc., Mountain View, CA. *FORTRAN 4.0 User's Guide*, 1996.

[Sun96b] Sun Microsystems, Inc. *SPARCstation 20 Series with SuperSPARC and SuperSPARC-II Processors*, 1996. http://www.sun.com:80/products-n-solutions/hw/wstns/jtf_ss20.html.

[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.

This article was processed using the LaTeX macro package with LLNCS style