Interactive Compilation and Performance Analysis with URSA MINOR

Insung Park Michael Voss Brian Armstrong Rudolf Eigenmann[*]

Purdue University, School of Electrical and Computer Engineering

Abstract. This paper proposes solutions to two important problems with parallel programming environments that were not previously addressed. The first issue is that current compilers are typically black-box tools with which the user has little interaction. Information gathered by the compiler, although potentially very meaningful for the user, is often inaccessible or hard to decipher. Second, compilation and performance analysis tools are not well integrated. While there are many advanced instruments for gathering and browsing performance results of a program, it is difficult to relate this information to the source program, to the applied program transformations, and to the compiler's reasoning.
The URSA MINOR tool addresses these issues. The tool is designed to help understand the structure of a program and the information gathered by a compiler in an interactive way. It facilitates the comparison of performance results under different environments and the identification of potential parallelism, and it provides a repository for this information. URSA MINOR is built using the Polaris compiler infrastructure. We present case studies that show how programmers can use the tool to find additional parallelism in a compiler-optimized program and to characterize the performance of parallel applications. The tools are currently being used in several projects to develop and study parallel applications and to evaluate parallelizing compilers. These efforts provide feedback for improving the URSA MINOR tool.

1 Introduction

Developing well-performing parallel programs is a challenging task. Many programming tools exist that assist the user in this task. Parallelizing compilers are one important class of such tools [BEH+94, HAA+96]. The apparent advantage of using a parallelizing compiler is that the conversion of a given serial program into parallel form is done mechanically by the tool. However, the compiler may have insufficient knowledge or limited capabilities to parallelize a program optimally. In some cases it would be easy for the user to make up for these shortcomings. For example, the compiler may detect a value-specific data-dependence, whereas the user would know that in every reasonable program input, the values

[*] This work was supported in part by Purdue University and 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.

are such that the dependence does not occur. In other cases, users may know that the accessed array sections in loop iteration do not overlap. Furthermore, certain program transformations may make a substantial performance difference, but are applicable to very few programs, and hence not built into a compiler's repertoire. If a user can find the reason why a loop was not parallelized automatically, a small modification may be applied to ensure parallel execution. Because of these reasons, manual code modification in addition to automatic parallelization is often necessary to achieve good performance.

During the process of compiling a parallel program and measuring its performance, a considerable amount of information is gathered. This includes all the data that a user can deduce from the results of compilation and simulation, such as the timing information of various program runs and structural information of the program. Finding parallelism starts from looking through this information and locating potentially parallel sections of code. However, the bookkeeping effort accompanying this procedure is often overwhelming.

In this paper, we introduce an on-going tool project, supporting a scenario of user-plus-compiler parallelization. The tool is designed to help understand the structure of a program, compare performance results under different environments, identify potential parallelism, and store the information for later use. The tool, Ursa Minor, gathers information along the course of compiling and running a program and presents it in a format that is easy to look up and comprehend. This leads to an understanding of the compilation process, the characteristics of the given program, its performance results, and the relationships of this data. Based on this information, a user may choose to work on enhancing the performance of an already parallelized program or start modifying the serial program.

The tool presented here is closely related to the Polaris compiler infrastructure [BEH+94]. Polaris, as a compiler, includes advanced capabilities for array privatization, symbolic and nonlinear data dependence testing, idiom recognition, interprocedural analysis, and symbolic program analysis. Polaris also represents a general infrastructure for analyzing and manipulating Fortran programs, which can provide useful information regarding the program structure and its potential parallelism. Polaris plays a major role in generating the data files used as input to our tool. Examples of such files are loop parallelization summaries, data-dependence information, and loop/subroutine call graphs. Polaris also instruments programs for timing measurements and maximum parallelism detection.

Section 2 presents our objectives in developing Ursa Minor, and Section 3 gives an overview and discusses its functionality. Section 4 then presents a case study of Ursa Minor in-use, followed by a brief discussion of future developments and improvements of the tool in Section 5. Section 6 concludes the paper.

2 Objectives of Ursa Minor

The intended users of the Ursa Minor tool are parallel programmers that have some experience with parallelizing compilers and performance analysis tools. In order to assist them in their effort to find parallelism, the tool pursues the following objectives:

Integrated Browsers for Program, Compilation, and Performance Data : The Ursa Minor tool collects and facilitates the use of program, compilation, and performance data. The information needs to be presented in a format that conveys high-level as well as detailed descriptions of a program. In this way, a user can start from an overall view of the program and inspect the details whenever he/she feels the need to concentrate on a specific portion of the program. The tool complements and integrates capabilities provided by other tools, such as the Pablo [Ree94], Paradyn [MCC$^+$95], and PTOPP [EM93] performance analysis environments.

Interactive Compilers : The current, predominantly black-box use of parallelizing compilers needs to be changed into interactive scenarios. This goes beyond interactive pass invocation as pioneered by tools such as Start/Pat [ASM89] and Parascope [BKK$^+$89]. The ultimate goal of the Ursa Minor project is to provide a comprehensive environment that encompasses the process of writing, compiling, running, and improving parallel programs. Enhancing parallelizing compilers with an interactive capability to provide information available throughout the program development process, contributes to this goal.

These objectives distinguish our approach from related efforts, such as the Polaris, Pablo and Paradyn projects, which provide advanced facilities for optimizing and instrumenting programs, gathering performance data, and visualizing this information. The Ursa Minor environment provides aids for the user to understand the gathered performance data, and to reason about the information in an interactive way, which goes significantly beyond the level available in these related projects. In the sense that the tool provides users with advice to improve performance, Ursa Minor has a similar objective to that of VTune[Int97], with one important difference being the targeted (in our case parallel) architectures.

In addition to the main objectives, we observe the following design rules to make our tool more useful and easily accessible:

Portability: For disseminating a new tool to the user community, it is important that it be easy to install on new platforms. We approach this goal by implementing Ursa Minor in the target-independent Java language, and by using only widely-available Application Programming Interfaces (APIs). No complicated makefiles and platform-dependent features are included. The tool makes use of information gathered by other tools, such as the Polaris compiler and its performance analysis libraries. The portability of these facilities is provided for many platforms. In addition, Ursa Minor will be built to be flexible in the data format it can read, such that it can adapt to the tools (compilers and performance analyzers) available on the local platform.

Leveraging off of existing tools: We consider using other available tools to augment the features of URSA MINOR that we regarded as "not original but nice to have". For instance, there are spreadsheets capable of rich graphical presentation of data. By allowing the information to be understood by one of these spreadsheets, we can take advantage of its features to create charts, while focusing on the new functionality of URSA MINOR.

Expandability : The main function of the URSA MINOR tool is information gathering and browsing. Hence, whenever we obtain new types of information about the given program we should be able to see it through the tool with minimal modifications. We can also enable the tool to read a generic data file, so that a new type of information, if it is in the specified format, can be understood by the tool without any modification.

3 Description of URSA MINOR

In this section, we give an overview of the URSA MINOR tool and describe its functionality. We also discuss how our design objectives were realized in the tool.

3.1 Overview

The URSA MINOR project provides tools that assist parallel programmers in effectively writing and tuning codes. It provides users with information available from various sources in a comprehensible way. These sources include tools such as compilers, profilers, and simulators. It interacts with users through a graphical interface, which can provide selective views and combinations of the data. Figure 1 shows an overall view of the interaction between URSA MINOR and various data files.

The URSA MINOR tool collects and combines information from various sources. Timing information is gathered from instrumented program runs. The tool performing this instrumentation is a Polaris-based utility, not discussed further in this paper. Maximum parallelism estimates are supplied by the MAX/P tool [Pet93, KE97]. Information on which loops are serial or parallel is provided by the actual Polaris compiler. The URSA MINOR tool includes a subroutine and loop calling structure analyzer, also implemented using the Polaris infrastructure.

In the current implementation, these information sources are available in files that need to be created explicitly by the user before URSA MINOR can read and combine them. Once they exist, several tool options are provided to read from the various original files, add to the existing information incrementally, store the entire database, or read from a previously saved database. In future releases we plan to automate the process of creating the information sources by, for example, invoking the compiler on-demand.

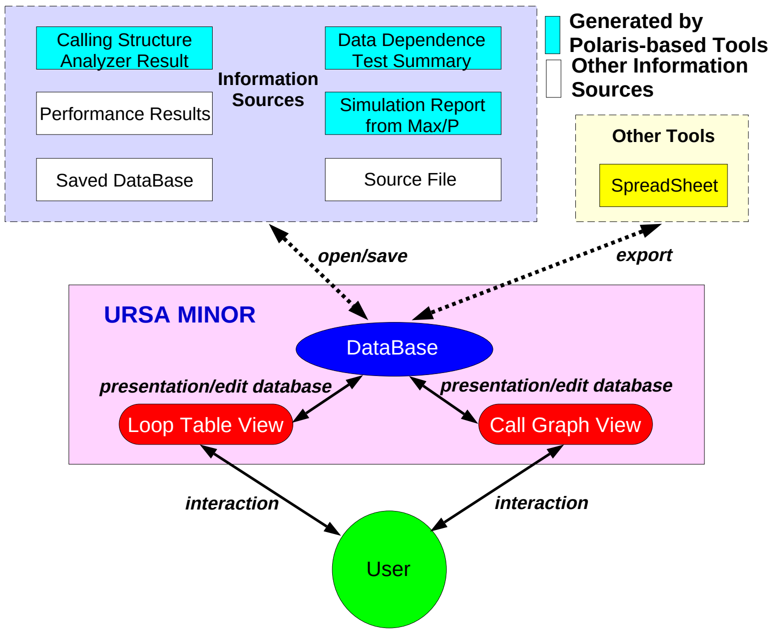

Calling Structure Analyzer Result
Data Dependence Test Summary
Information Sources
Performance Results
Simulation Report from Max/P
Saved DataBase
Source File
Generated by Polaris-based Tools
Other Information Sources
Other Tools
SpreadSheet
open/save
export
URSA MINOR
DataBase
presentation/edit database
presentation/edit database
Loop Table View
Call Graph View
interaction
interaction
User

Fig. 1. Interaction provided by the Ursa Minor Tool.

Internally, Ursa Minor stores information in a "database", which is formatted as text with appropriate annotations. This database in itself can serve as a browsable source of information. Furthermore, the information can be saved in a format that commercial spreadsheets can read from, allowing a richer set of data manipulations and graphical representations.

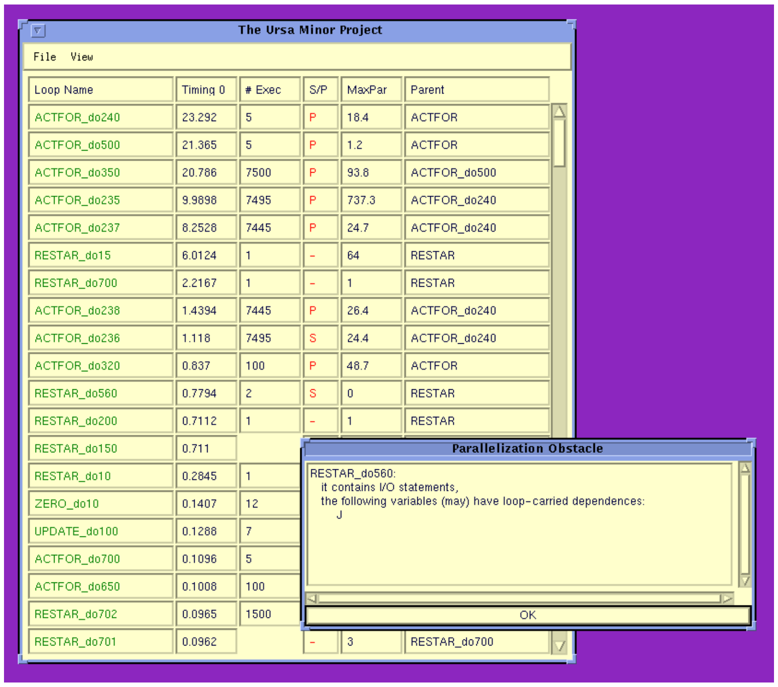
The Ursa Minor tool is written in Java. Thus, any platform on which the Java runtime environment is available can be used to run the tool. It uses the basic Java language with standard APIs, which enhances the portability of the tool. Object orientation in Java allows a relatively easy addition of new types of data to the database. The windowing toolkits and utilities provided a good environment for prototyping user interfaces, which enabled us to focus on the design of the tool functionality.

One of our related projects is to make publicly available a database of program characteristics and performance data for a collection of program and benchmark suites. Java, with its network support, makes a useful language for this project. By enabling the use of the Ursa Minor database via the World-Wide Web, users in remote sites can have a better understanding of the programs they are examining and compare the results of their experiments.

In the next section, we examine more closely the functionality of Ursa Minor.

3.2 Functionality

The Ursa Minor tool presents information to the users through two display windows: A loop information table and a call graph. The user interacts with Ursa Minor by choosing menu items or mouse-clicking.



Loop Name	Timing 0	# Exec	S/P	MaxPar	Parent
ACTFOR_do240	23.292	5	P	18.4	ACTFOR
ACTFOR_do500	21.365	5	P	1.2	ACTFOR
ACTFOR_do350	20.786	7500	P	93.8	ACTFOR_do500
ACTFOR_do235	9.9898	7495	P	737.3	ACTFOR_do240
ACTFOR_do237	8.2528	7445	P	24.7	ACTFOR_do240
RESTAR_do15	6.0124	1	-	64	RESTAR
RESTAR_do700	2.2167	1	-	1	RESTAR
ACTFOR_do238	1.4394	7445	P	26.4	ACTFOR_do240
ACTFOR_do236	1.118	7495	S	24.4	ACTFOR_do240
ACTFOR_do320	0.837	100	P	48.7	ACTFOR
RESTAR_do560	0.7794	2	S	0	RESTAR
RESTAR_do200	0.7112	1	-	1	RESTAR
RESTAR_do150	0.711				
RESTAR_do10	0.2845	1			
ZERO_do10	0.1407	12			
UPDATE_do100	0.1288	7			
ACTFOR_do700	0.1096	5			
ACTFOR_do650	0.1008	100			
RESTAR_do702	0.0965	1500			
RESTAR_do701	0.0962		-	3	RESTAR_do700

```
RESTAR_do560:
  it contains I/O statements,
  the following variables (may) have loop-carried dependences:
      J
```

Fig. 2. Loop Table View of the Ursa Minor Tool.

Figure 2 shows the loop table view, each line displaying information for an individual loop. Currently, the table displays information such as timing results from various program runs, the number of invocations of each loop, the parent in the calling structure, and the maximum degree of parallelism provided by Max/P [Pet93, KE97]. It also indicates whether a loop is serial or parallel as detected by Polaris. If it is serial, the reason given by the compiler can be displayed on mouse-clicking. In Figure 2, the user has clicked on loop `RESTAR_do560` to see the reason inhibiting parallelization.

Whenever additional information is available, more columns can be added. Also, a user can rearrange columns, delete columns, sort the entries alphabetically or based on the execution time. By specifying the column displaying serial timing information, speedup can be calculated on-demand.

Another view of Ursa Minor provides the calling structure of a given program, which includes subroutine, function, and loop nest information as shown in Figure 3. Each rectangle represents either a subroutine, function, or loop. Clicking one of these will display the corresponding source code. In Figure 3 the user is inspecting the loop `ACTFOR_do240` in this way. If one wants a wider view of the program structure, the user can zoom in and out. This display enables a better understanding of the program structure for tasks such as interchanging loops or finding outer or inner candidate parallel loops.

Ursa Minor can save the database in a format that generic spreadsheet programs can understand. In Figure 4 we have read this form into the commercial xess3 spread-sheet program. This allows one to exploit the many options and graphical representations of this tool. In Figure 4 the user has chosen an execution time graph for the program `BDNA`, comparing the performance of Polaris with the compiler from Sun Microsystems, (the third line indicating "linear speedup" for reference).

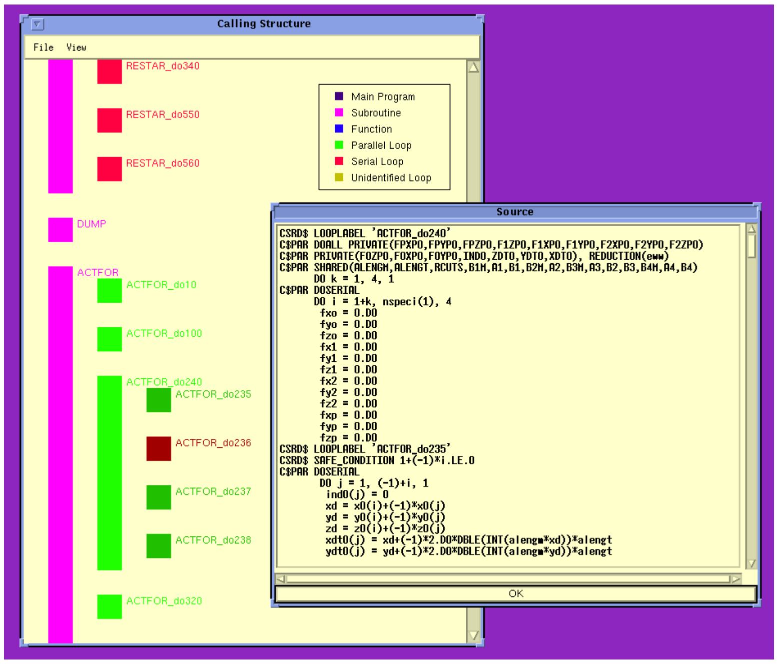
4 Case Studies

4.1 Experiments with the ARC2D Application

In a current study, we are comparing parallel directive languages for their suitability as a portable compiler output representation [Vos97]. In doing so, we have expressed the parallelism in several benchmark codes with various directive sets. If the performance results of these codes are significantly different, Ursa Minor is used in the search for explanations of these differences. An example of such a search performed on the Perfect Benchmarks code `ARC2D`, is presented here as our first case study.

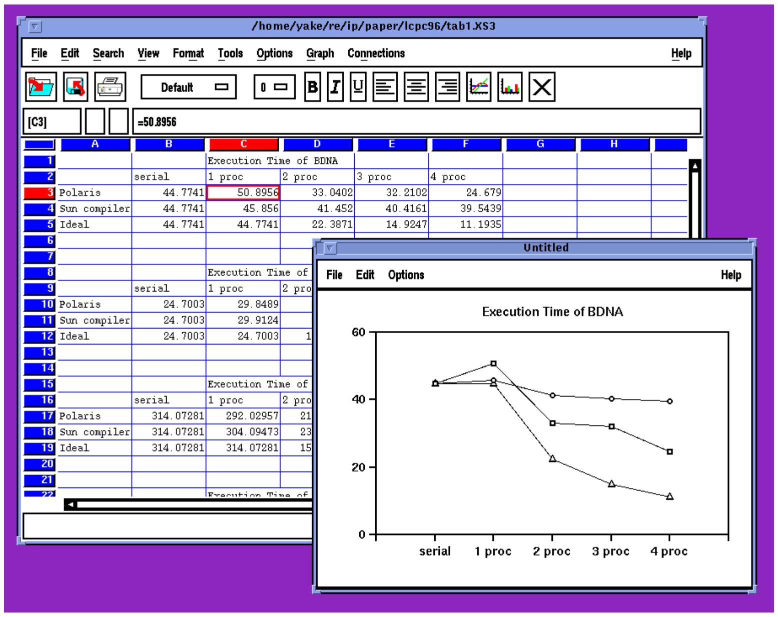
First, as a base-line measurement, a loop by loop profile of the serial version of the code executed on a 4 processor UltraSPARC workstation was done. The results of this instrumentation was then gathered by Ursa Minor and transformed into a form which is readable by commercial spreadsheet packages such as Excel and Xess3. A major concern with instrumentation is that the overhead associated with it will noticeably impact the measured performance. Using the number of times each loop is executed, as well as the execution time measured by the instrumentation, it is easily determined when such perturbation occurs. In ARC2D, 114 of the 149 instrumented loops had significant overhead associated with their instrumentation, this being over 0.1% of the execution time of the loop. Removing the instrumentation from these 114 loops, reduced the total execution time of the program by over 46%. Ursa Minor currently provides the average execution times necessary for this computation. In future releases of the tool, this computation will be fully automated.

Additionally, the most time-consuming loops were identified in the serial code. The Polaris-parallelized versions of these loops were used to compare the performance of several parallel directive languages. The major loops in ARC2D parallelized by Polaris are FILERX_do19, STEPFX_do210 and STEPFX_do230. The identification of these loops was straightforward given that Ursa Minor presented the execution times of each loop as well as annotated it as parallel or



```
CSRD$ LOOPLABEL 'ACTFOR_do240'
C$PAR DOALL PRIVATE(FPXP0,FPYP0,FPZP0,F1ZP0,F1XP0,F1YP0,F2XP0,F2YP0,F2ZP0)
C$PAR PRIVATE(F0ZP0,F0XP0,F0YP0,IND0,ZDT0,YDT0,XDT0), REDUCTION(eww)
C$PAR SHARED(ALENGM,ALENGT,RCUTS,B1M,A1,B1,B2M,A2,B3M,A3,B2,B3,B4M,A4,B4)
      DO k = 1, 4, 1
C$PAR DOSERIAL
      DO i = 1+k, nspeci(1), 4
       fxo = 0.D0
       fyo = 0.D0
       fzo = 0.D0
       fx1 = 0.D0
       fy1 = 0.D0
       fz1 = 0.D0
       fx2 = 0.D0
       fy2 = 0.D0
       fz2 = 0.D0
       fxp = 0.D0
       fyp = 0.D0
       fzp = 0.D0
CSRD$ LOOPLABEL 'ACTFOR_do235'
CSRD$ SAFE_CONDITION 1+(-1)*i.LE.0
C$PAR DOSERIAL
       DO j = 1, (-1)+i, 1
        ind0(j) = 0
        xd = x0(i)+(-1)*x0(j)
        yd = y0(i)+(-1)*y0(j)
        zd = z0(i)+(-1)*z0(j)
        xdt0(j) = xd+(-1)*2.D0*DBLE(INT(alengm*xd))*alengt
        ydt0(j) = yd+(-1)*2.D0*DBLE(INT(alengm*yd))*alengt
```

Fig. 3. Annotated Call Graph View of the Ursa Minor Tool.


/home/yake/re/ip/paper/lcpc96/tab1.XS3
File Edit Search View Format Tools Options Graph Connections Help
Default
0
B I U
Untitled
File Edit Options Help
Execution Time of BDNA
60
40
20
0
serial
1 proc
2 proc
3 proc
4 proc

[C3] =50.8956

	A	B	C	D	E	F	G	H
1			Execution Time of BDNA					
2		serial	1 proc	2 proc	3 proc	4 proc		
3	Polaris	44.7741	50.8956	33.0402	32.2102	24.679		
4	Sun compiler	44.7741	45.856	41.452	40.4161	39.5439		
5	Ideal	44.7741	44.7741	22.3871	14.9247	11.1935		
6								
7								
8			Execution Time of					
9		serial	1 proc	2 pro				
10	Polaris	24.7003	29.8489					
11	Sun compiler	24.7003	29.9124					
12	Ideal	24.7003	24.7003	1				
13								
14								
15			Execution Time of					
16		serial	1 proc	2 pro				
17	Polaris	314.07281	292.02957	21				
18	Sun compiler	314.07281	304.09473	23				
19	Ideal	314.07281	314.07281	15				
20								
21								
22			Execution Time of					

Fig. 4. Spread-Sheet View of the Ursa Minor Tool.

serial. The relative importance of these loops in the serial version can be seen in Figure 5.

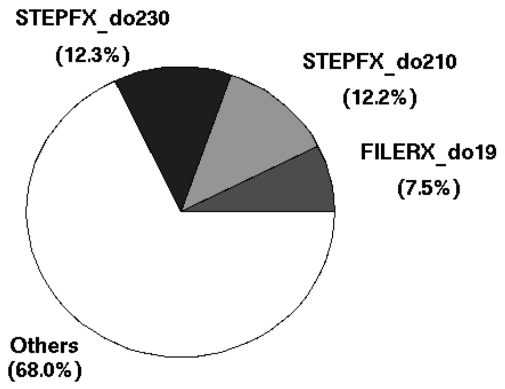


Fig. 5. Percentage of Execution Time Spent in Major Loops of ARC2D.

The parallelism found by Polaris was expressed in two forms. One using the native Sun SPARC dialect and the other using the portable KAP/Pro directive set [Kuc88], a descendant of the proposed ANSI X3H5 standard. Browsing through the performance results displayed by Ursa Minor it was seen that on 4 processors, the KAP/Pro directive language exhibited superior performance. Furthermore, by adding the loop-by-loop profile of ARC2D, as parallelized by the Sun native compiler, an interesting phenomenon was discovered.

A significant "negative overhead" existed for many of the loops in the KAP/Pro version when comparing the 1 processor parallel execution to the execution of the untransformed code. Apparently, sequential optimizations were performed in the KAP/Pro version which were not performed in the serial version. Interestingly, this same optimization was often performed in the loops found to be parallel by the native compiler, but not in the Polaris version which used the Sun SPARC directives. The performance of the three major loops is shown in Figure 6.

Using the source code browsing capabilities, a side-by-side comparison of the loop nests uncovered the reason. Loop interchanging was being applied to many of the loop nests in the KAP/Pro directive version by the back-end compiler. The use of the Sun SPARC directives inhibited this transformation. Loop interchanging was not disabled when parallelizing the code with the native Sun parallelizing compiler; however it was applied less frequently. For a more detailed discussion of this phenomenon and others uncovered during the analysis of ARC2D, please refer to /citeVoss97.

A further analysis of the serial source code and the Polaris translated versions, as well as the program structures shown by the graphical loop structure representation, showed that the two most significant loops STEPFX_do210 and

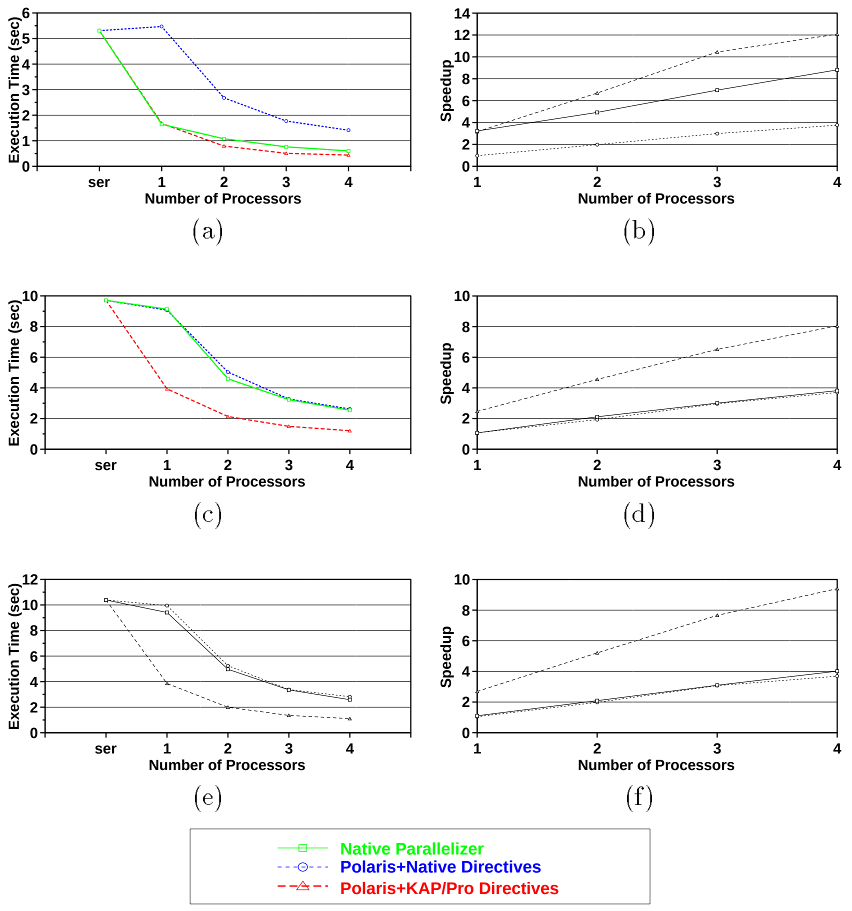


Fig. 6. Loop Performance of ARC2D on an UltraSPARC: (a) Execution Time of FILERX_do19, (b) Speedup of FILERX_do19, (c) Execution Time of STEPFX_do210, (d) Speedup of STEPFX_do210, (e) Execution Time of STEPFX_do230 and (f) Speedup of STEPFX_do230.

STEPFX_do230 were imperfectly nested in the original source, but were transformed into a perfect nest by Polaris. The application of *forward substitution* and *deadcode elimination* by Polaris created perfectly nested loops, which the back-end compiler was then able to interchange. Therefore, although the native Sun parallelizing compiler was able to identify the same amount of parallelism as Polaris, it did not create perfectly nested loops, and therefore did not facilitate further optimization. Figure 7 shows the performance of the three parallel versions of ARC2D executed on 4 processors of the UltraSPARC. This figure also shows the performance that would be obtained in the Sun SPARC directive version if the interchanging had been done.

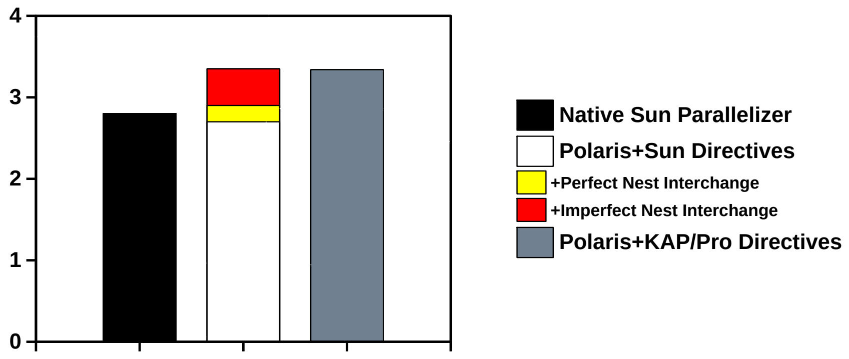


Fig. 7. Performance of ARC2D on 4 Processors of UltraSPARC.

Ursa Minor allowed the characteristics responsible for the performance differences in `ARC2D` to be quickly identified. The often tedious task of tabularizing profiling results was performed automatically and the identification of the parallel loops in this table was made obvious. The nesting structure of the loops was a major factor in the performance of this code, and Ursa Minor's graphical display of the loop structure was a significant aid in quickly identifying this phenomenon. A detailed study of the several versions of the source code for each loop nest was often necessary, and a side-by-side comparison was easily performed with the browsing facilities. The graphs presented in Figures 5 through 7 can be generated by exporting the Ursa Minor database to the Xess3 spreadsheet and using its graphing functions.

4.2 Experiment with the Seismic Application

As the second case study, we introduce another project that characterizes and analyzes large-scope industrial applications [AE97]. One of the programs we considered was the Seismic Benchmark Suite [MH93]. It is a seismic activity simulation program consisting of 20,000 lines of Fortran code. Here, we briefly describe how the Ursa Minor tool can be of help in the process of analyzing a large application.

The Seismic Benchmark Suite contains a deep hierarchy of nested subroutines and loops. Our goal is to understand the computational complexity of the overall application suite and how this complexity changes with increasing numbers of available processors. In order to do this we not only need the measurements of loop execution times but also a formulation of the time spent in each loop as a function of the number of processors and the input data size. The Ursa Minor tool aids in analyzing both the structure of the loop-hierarchy as well as the individual times each loop contributes to the application suite.

The execution time for a loop can be formulated by averaging the time spent in a single iteration and multiplying the average by the number of iterations. If the number of iterations that a loop executes is known, then an estimation of the total loop execution time is the number of iterations times the average execution time per iteration. The Ursa Minor tool displays the average execution time for an iteration of each loop in an easily accessible table.

However, a loop's execution time may consist of a number of inner loops. Knowing the loop hierarchy, provided visually with Ursa Minor, the major components of the total execution time become clear. A representation of the call-graph as well as the hierarchy of loop nestings coupled with the actual times spent in each loop is necessary in finding which sections of the code are the dominant ones.

Another objective of the Seismic case study was to produce a well-performing program, using Polaris as a starting point. We did this by parallelizing the program with Polaris and then improving its performance by hand. Ursa Minor assisted this manual process by providing the loop-by-loop table, from which we could calculate the speedup and efficiency for each loop. Loops which had a speedup below 1 were flagged. These loops were investigated further concerning how to improve their automatic parallelization by Polaris. If no improvements could be made, these loops were forced to execute serially so that they would not incur any parallel execution overhead. In doing so, we used the feature that Ursa Minor can place information in a format importable by a spreadsheet application, with which we then performed operations such as computing "speedup columns" and creating diverse graphical displays.

5 Future Development

Ursa Minor is an ongoing tool project. We will continue to add new program, compilation and performance information to our tool. This means that we search for new sources of data as well as extracting more from existing ones. Polaris, for example, maintains a large amount of information needed for parallelization, such as the value ranges that variables assume, array sections being accessed, data dependences, applicable transformations, performance estimations. If the tool can obtain such data, it can significantly improve the user's understanding of the program under consideration. Also, we are planning to add more display features whenever the newly obtained data call for a different type of presentation for effective communication with the users.

Another goal is to integrate parallelizing compilers and performance analysis facilities more tightly into our tool. In this way we will create an environment that facilitates the entire process of developing parallel programs. Users will be able to selectively apply parallelization techniques by interacting with the tool, or to query available information for a specific piece of code while writing or compiling a program. Following the design objective of leveraging off of existing tools, we will continuously search for available facilities to incorporate into the tool. An important complementing objective is to provide methodologies for interpreting performance data. Such methodologies will guide the user in identifying improvements to the application program and the underlying computer system.

As mentioned in Section 3.1, we are planning to create a database of all the program characteristics and performance data and make it available through the Internet. While it requires minor modifications to the Ursa Minor tool, creating such a repository would take considerable effort in collecting and organizing the data. We are currently developing such an organization, which will encompass characteristics of several program and benchmarks suites, performance results on diverse machines, and facilities to visualize and reason about this information. This database will reduce redundant work that is often necessary for researchers to familiarize themselves with new programs. Also, making available the basis for performance comparison allows active communication among optimization researchers.

An important factor in developing an interactive program is the feedback from its users. In order to be able to serve a growing user community, we will include a way to configure Ursa Minor so that it can adapt to the users' preferences.

6 Conclusions

We have presented an on-going tool project for parallel programming. The Ursa Minor tool fills two substantial voids in the programmer's toolbox. It facilitates interactive access to the information gathered by parallelizing compilers and supports integrated views of this information with that provided by performance analysis tools. We expect this tool to grow and become an overall programming environment for parallel applications. We have shown how users can benefit from simply putting information together in one place. The tool is portable and easy to install, using the Java language. It also provides a basis for developing a database and browser of program characteristics for distribution via the Web.

Ursa Minor is evolving in a need-driven way. Its developers are involved in projects such as the characterization and analysis of real applications and the development of parallelizing compilers. Tool capabilities needed in these efforts will be integrated in Ursa Minor. Keeping close together the tool design projects and application characterization efforts will ensure the practicality of our tool in the future.

References

[AE97] Brian Armstrong and Rudolf Eigenmann. Performance forecasting: Characterization of applications on current and future architectures. Technical Report ECE-HPCLab-97202, Purdue University, School of Electrical and

Computer, Engineering, High-Performance Computing Laboratory, February 97.

[ASM89] Bill Appelbe, Kevin Smith, and Charles McDowell. Start/Pat: A Parallel-Programming Toolkit. *IEEE Software*, 6(4):29–38, July 1989.

[BEH$^+$94] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994.

[BKK$^+$89] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The ParaScope editor: An interactive parallel programming tool. In *International Conference on Supercomputing*, pages 540–550, 1989.

[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.

[HAA$^+$96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, pages 84–89, December 1996.

[Int97] Intel. *VTune: Visual Tuning Environment*, 1997. http://developer.intel.com/design/perftool/vtune/index.htm.

[KE97] Seon-Wook Kim and Rudolf Eigenmann. *Max/P: detecting the maximum parallelism in a Fortran program.* Purdue University, School of Electrical and Computer, Engineering, High-Performance Computing Laboratory, 1997. Manual ECE-HPCLab-97201.

[Kuc88] Kuck & Associates, Inc., Champaign, Illinois. *KAP User's Guide*, 1988.

[MCC$^+$95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), November 1995.

[MH93] C. C. Mosher and S. Hassanzadeh. ARCO seismic processing performance evaluation suite, user's guide. Technical report, ARCO, Plano, TX., 1993.

[Pet93] Paul Marx Petersen. *Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1993.

[Ree94] Daniel A. Reed. Experimental performance analysis of parallel systems: Techniques and open problems. In *Proc. of the 7th Int' Conf on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 25–51, 1994.

[Vos97] Michael J. Voss. Portable loop-level parallelism for shared memory multiprocessor architectures. Master's thesis, School of Electrical and Computer Engineering, Purdue University, October, 1997.

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