Parallel Programming and Performance Evaluation with The URSA Tool Family*

Insung Park Michael Voss Brian Armstrong Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University

Abstract

This paper contributes to the solution of several open problems with parallel programming tools and their integration with performance evaluation environments. First, we propose interactive compilation scenarios instead of the usual black-box-oriented use of compiler tools. In such scenarios, information gathered by the compiler and the compiler's reasoning are presented to the user in meaningful ways and on-demand. Second, a tight integration of compilation and performance analysis tools is advocated. Many of the existing, advanced instruments for gathering performance results are being used in the presented environment and their results are combined in integrated views with compiler information and data from other tools. Initial instruments that assist users in "data mining" this information are presented and the need for much stronger facilities is explained.

The URSA Family provides two tools addressing these issues. URSA MINOR supports a group of users at a specific site, such as a research or development project. URSA MAJOR complements this tool by making available the gathered results to the user community at large via the World-wide Web.

This paper presents objectives, functionality, experience, and next development steps of the URSA tool family. Two case studies are presented that illustrate the use of the tools for developing and studying parallel applications and for evaluating parallelizing compilers.

1 Introduction

Interactive use of parallelizing compilers. Many programming tools exist that assist the user in the challenging task of developing well-performing parallel programs. Parallelizing compilers are one important class of such tools [BDE+96, 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. One disadvantage of this scenario is that 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, although the compiler detects a value-specific data dependence, the user may know that in every reasonable program input the values are such that the dependence does not occur. In other cases, users may know that the array sections accessed in different loop iterations 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 that ensures parallel execution. Because of these reasons, manual code modification in addition to automatic parallelization is often necessary to achieve good performance.

*This work was supported in part by Purdue University, U. S. Army contract #DABT63-92-C-0033, and an NSF CAREER award. This work is not necessarily representative of the positions or policies of the U. S. Army or the Government.

Integrated compilation and performance evaluation. During the process of compiling a parallel program and measuring its performance, a considerable amount of information is gathered. For example, timing information becomes available from various program runs, structural information of the program is gathered from the code documentation, and compilers offer a large amount of program analysis information. Finding parallelism starts from looking through this information and locating potentially parallel sections of code. Improving parallel performance is the immediate next step. Decisions are made based on timing results and their relationship to program characteristics. The bookkeeping effort accompanying this procedure is often overwhelming. Tools that assist this process are important.

In this paper, we introduce an on-going tool project that supports a scenario of user-plus-compiler parallelization. The tool helps a programmer understand the structure of a program, identify parallelism, and compare performance results of different program variants. The tool, Ursa Minor [PVAE97], 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. Using the tool, the programmer comes to an understanding of the compilation process, the characteristics of the given program, its performance results, and the relationships of these data. It is the basis for enhancing the performance of an existing parallel program as well as for beginning to parallelize a serial program.

The presented tool is closely related to the Polaris compiler infrastructure [BDE+96]. Polaris, as a compiler, includes advanced program analysis and transformation techniques, such as 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 about the program structure and its potential parallelism. Polaris plays a major role in generating the data files used as input to Ursa Minor. 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. Section 3 gives an overview and discusses its functionality. Section 4 presents the Ursa Major tool [PE98], a web-based tool built upon Ursa Minor that was designed for distribution and evaluation of experimental results with various parallel applications. Section 5 then shows two case studies of Ursa Minor in use. 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 using parallelizing compilers and performance analysis tools. In order to assist them in identifying and exploiting 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 or she feels the need to concentrate on a specific portion of the program. The tool complements and integrates capabilities provided by 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 an interactive scenario. 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. To this end interactive capabilities

are provided to view program information gathered by the compiler and relate it to information provided by other programming tools.

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. In the sense that the tool provides users with advice to improve performance, Ursa Minor has a similar objective to that of VTune[Int97], which is an advanced tool for single-processor systems.

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). The tool makes use of information gathered by other facilities, such as the Polaris compiler and its performance analysis libraries, which themselves are portable to many platforms. In addition, Ursa Minor is 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 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 new type of information can be understood without significant modifications.

3 Description of Ursa Minor

In this section, we provide an overview of Ursa Minor [PVAE97] and describe its functionality. We will discuss how our design objectives were realized in the concrete 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 illustrates interaction between Ursa Minor and the various data files.

Ursa Minor 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 [Eig93]. Maximum parallelism estimates are supplied by the Max/P tool [Pet93, KE97]. Information about which loops are serial or parallel is provided by the actual Polaris compiler. The Ursa Minor tool includes a subroutine and loop nest 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,

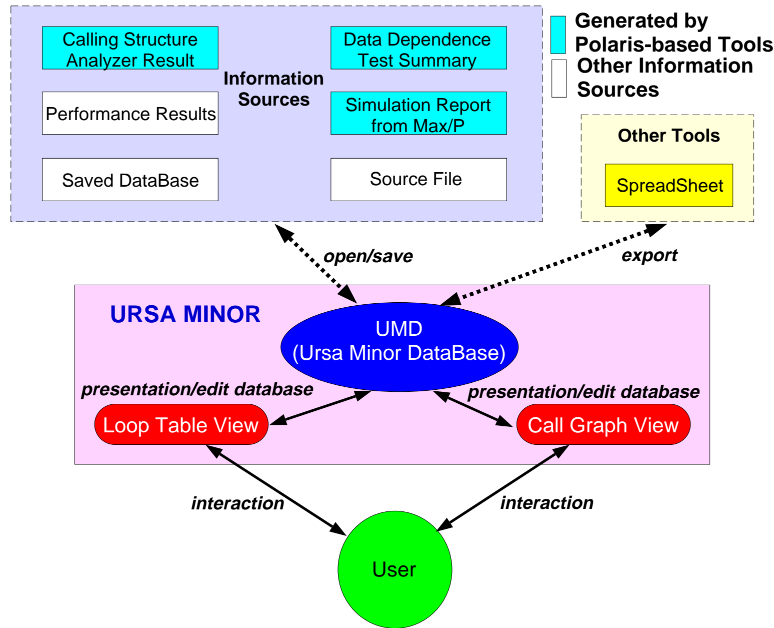

Figure 1: Components of the Ursa Minor tool and their interactions.

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.

Internally, Ursa Minor stores information in Ursa Minor/Major Databases (UMD). A UMD is a storage unit that holds the collective information about a program and its execution results in a certain system environment. This database is organized as a text file, which can optionally be inspected with an editor and printed. Furthermore, the information can be saved in a format that can be read by commercial spreadsheets, providing a richer set of data manipulation functions 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 provide a good environment for prototyping user interfaces, which enable us to focus on the design of the tool functionality. Furthermore, Java, with its network support, makes a useful language for realizing another goal of this project: making available the gathered program, compilation, and performance results to remote users. This goal has been realized in the Ursa Major tool, which is discussed in Section 4. In the next section, we examine the functionality of Ursa Minor more closely.

3.2 Functionality

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

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 nest 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,

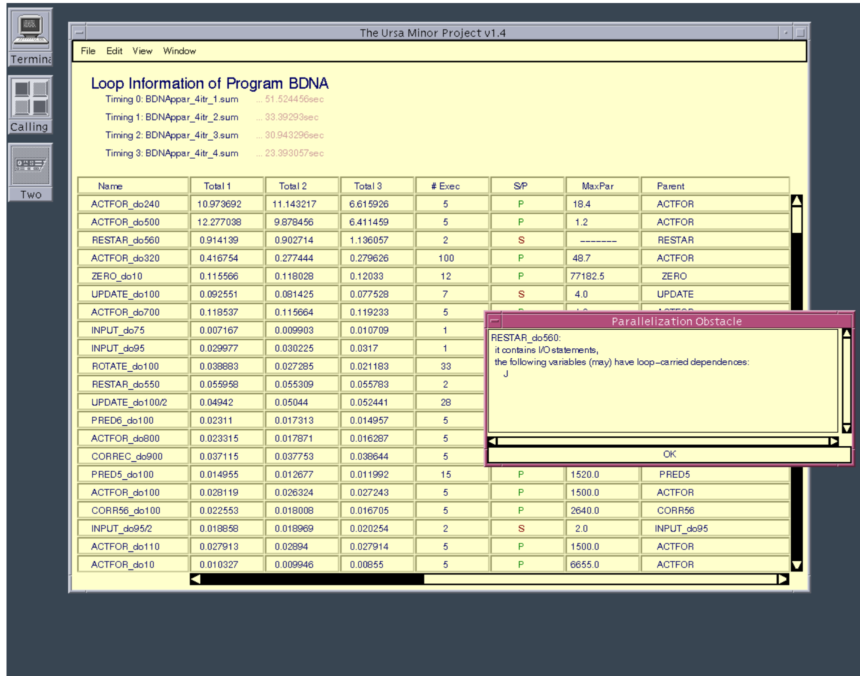

Figure 2: Loop Table View of the Ursa Minor tool.

the user has clicked on loop `RESTAR_do560` to see the reason inhibiting parallelization.

Whenever new information from other tools becomes available, the user can add columns in this view. Also, a user can rearrange columns, delete columns, sort the entries alphabetically or based on the execution time. By specifying a reference column, speedups can be calculated on-demand. In our program tuning projects, an Ursa Minor loop table is usually present all the time. After each program run, the newly collected timing information is included as an additional column in the loop table. In this way, performance differences can be inspected immediately for each individual loop as well as for the overall program. Effects of program modifications on other program sections become obvious as well. The modification may change the relative importance of the loops, so that sorting them by their newest execution time yields a new most-time-consuming loop on which the programmer can focus next.

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. For example, parallel loops are represented by green rectangles, and serial loops by red rectangles. 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 helps to understand 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 spreadsheet 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, (a third line indicating "linear speedup" for reference).

4 Ursa Major: Web-based evaluation of parallel applications

Ursa Major [PE98] is an extension of the Ursa Minor tool. Because we chose Java as an imple-

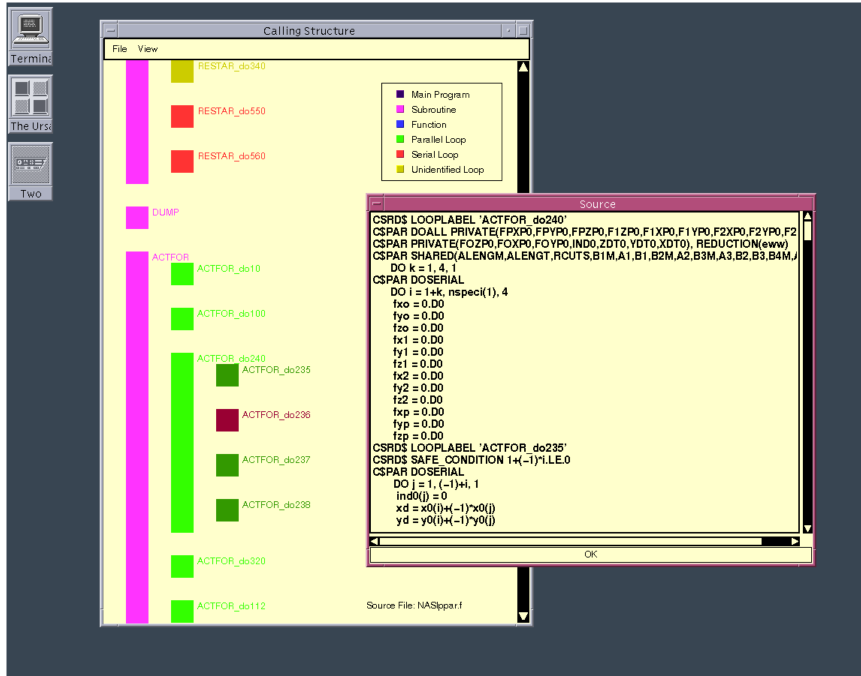

Figure 3: Annotated Call Graph View of the Ursa Minor tool.

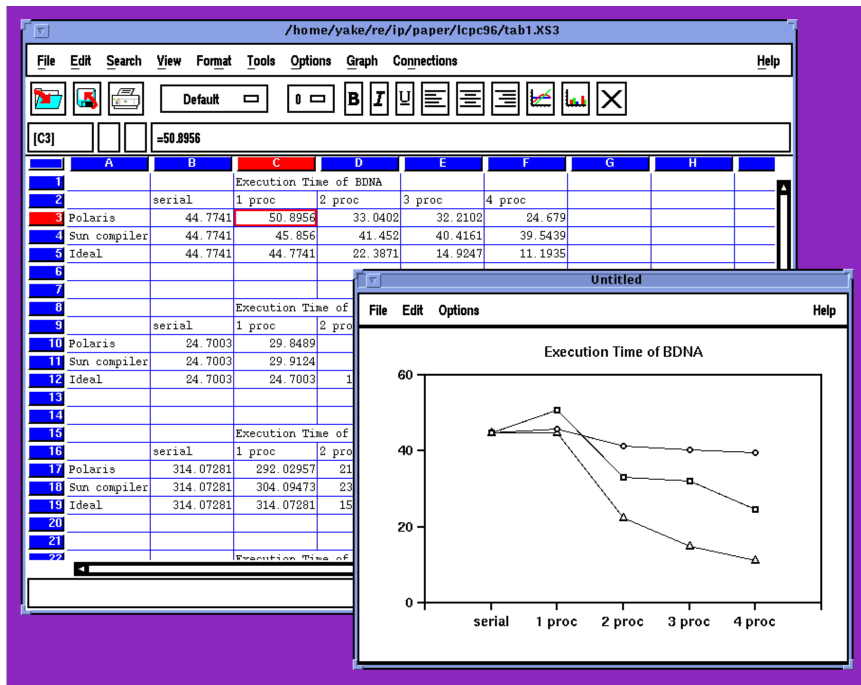

Figure 4: Spread-Sheet View of the Ursa Minor tool.

mentation language, it was natural to combine our tool capabilities with the rapidly advancing internet technology and, in this way, allow users at remote sites to access our experimental data.

In extending the use of our tool to a world-wide audience we are addressing several new issues:

First, affordable multiprocessor workstations and PCs are currently leading to a substantial increase in non-expert users and programmers. However, there are no established programming methodologies that could guide these users in exploiting the new machines. Ursa Major provides a methodology of "learning by example" to both local and remote users. New users see a variety of sample programs, their serial and parallel source code, performance improvements resulting from compilation or source code modifications, etc. The tool helps relate all these pieces of information, so that, for example, one can identify precisely the effect of a source code change on the performance for both the modified code section and the overall program.

Second, a core need for advancing the state of the art of computer systems is performance evaluation and the comparison of results with those obtained by others. To this end, many test applications have been made publicly available for study and benchmarking by both researchers and industry. Although a large body of measurements obtained from these programs can be found in the literature and on public data repositories, it is usually extremely difficult to combine them into a form meaningful for new purposes. In part this is because data are not readily available (i.e., they have to be extracted from several papers) and they have to undergo substantial re-categorizations and transformations. In addressing this issue, the Ursa Major project is creating a comprehensive database of information. From the beginning, the abstraction of performance and program information into a form that answers the questions of the observer was one of our goals. However, this issue becomes drastically more complex as we consider large data repositories organized into a multitude of dimensions. The internet technology and its combination with high-performance computing tools opens this new realm of questions and opportunities, which we are beginning to explore with Ursa Major.

4.1 Description of Ursa Major

Ursa Major is a web-based tool capable of presenting the Ursa Minor/Major database to a remote user. Figure 5 shows an overall view of the interactions between Ursa Major, a user, and the Ursa Major repository (UMR), which will be discussed in the next section. Ursa Major is available at http://www.ecn.purdue.edu/~ipark/UM/index.html.

Ursa Minor's facilities for manipulating databases and for creating graphical user interfaces are basic building blocks for Ursa Major. Java class inheritance was utilized extensively for developing Ursa Major's modules from these components. In addition, new modules were created for the tool's networking features and for organizing the data into a repository that is easy to access from remote Web sites. The latter includes the definition of naming schemes with which information can be found intuitively and can easily be related to other information.

Since it is based on Ursa Minor, Ursa Major offers the same basic functionality. One difference is the access to the repository. Remote Java applications cannot access disk files directly. They have to retrieve data in the form of Web documents. This is due to Java security restrictions. Users may choose the UMDs of their interest by examining the descriptions provided for the available UMDs. UMDs are then retrieved by their URL. Once a UMD is displayed, users may perform the same tasks as they do with Ursa Minor, except that they cannot save files on the local disk. The look and feel of the Ursa Major tool is almost identical with those of Ursa Minor, but Ursa Major is embedded in a web page through Java Applet and is invoked by clicking a button in the web page.

4.2 Ursa Major Repository (UMR)

During the process of compiling a parallel program and measuring its performance, a considerable amount of information is gathered. Several such efforts are ongoing in our group, hence the UMR is

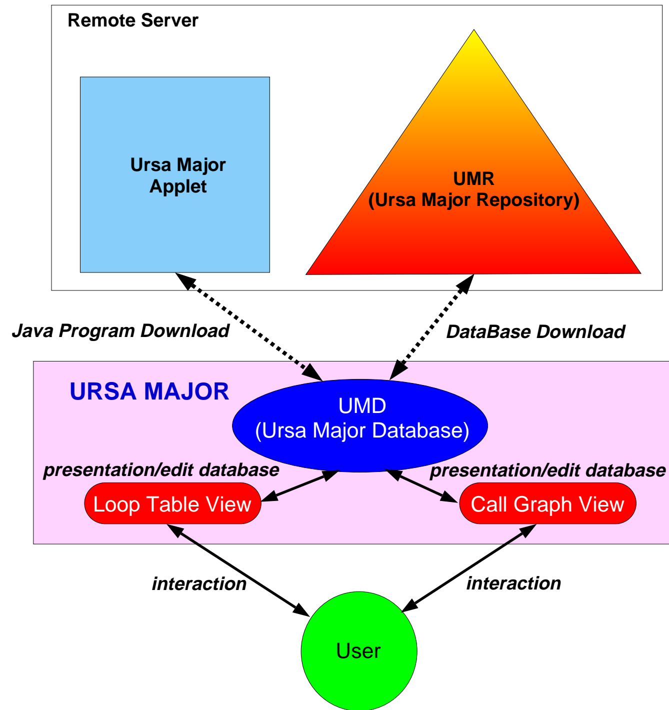

Figure 5: Interaction provided by the Ursa Major tool.

continuously being extended. It currently contains several benchmark suites that have been studied at Purdue University, including SPEC and Perfect benchmarks.

The specific data includes structural program information, results of program analysis, simulation reports, as well as the timing information of various program runs. Finding parallelism starts from looking through this information and locating potentially parallel sections of code. Several tools and methodologies are being used to gather and organize such data [VGGJ+89, EM93].

One issue in designing the repository was to define storage schemes that makes it easy for users to find information entered by other users. To this end, the repository structure consists of extensions on file and directory names indicating data such as the program names, platforms, compilers, optimization, and parallel languages. To be flexible, these extensions are not hard-coded. Instead, they are described in a configuration file that is read by Ursa Major at the start of a session.

4.3 Experiences with Ursa Major

We present early experiences with using the Ursa Major tool and with its implementation. We have used the tool in our research team, on multiple workstation platforms and also PCs connected through modems at home. Our team includes researchers at two universities, so that realistic remote accesses were involved. Based on these experiences we can picture scenarios of how the different user communities can best take advantage of the tool and what challenges need to be addressed to make it even more useful in the future.

Ursa Major targets several audiences. They include novice parallel programmers, advanced programmers, and researchers interested in performance evaluation and benchmarking. Obviously these categories can overlap. For beginners, the tool supports a methodology of “learning by example”. New programmers start by getting the general feel for the repository. This is best done starting with the call graph view and clicking on several nodes in this graph to inspect the source programs. To get more

insights about an individual program the user now can step through the most time-consuming loops and compare serial and parallel program versions. Ursa Major supports this by providing the loop table view. Source code corresponding to serial and the parallel variant can be opened. The loop table also shows timings of the two variants giving the user a first view of the speedups obtained by each loop. The tool can compute and display these speedup numbers as an option. Comparing these program variants gives the new user a first idea of how programs need to be transformed to run in parallel and what performance improvement can be obtained.

The advanced programmer may benefit from this tool by exploiting the features allowing the inspection of the reasons why certain parallel loops or program sections perform well or poorly in more detail and why a code section is not parallel. In this way, users may identify the bottleneck and possible improvements by combining the perspectives from both performance evaluation and compiler analysis results.

Ursa Major further serves the research community in general by making available the large amount of information kept in the Ursa Major repository and facilitating access to this information in various dimensions. Even within our research group the availability of the repository enabled many different studies, such as architectural comparisons, comparisons of different compilers, different programs, different subroutines and loops within a program, and scalability studies over numbers of processors and data set sizes. Increasing the support for inspecting our database from these various angles is an important ongoing effort.

5 Case Studies

5.1 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. One concern with instrumentation is that the associated overhead 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 loops had an instrumentation overhead of more than 0.1% of the loop execution time. We chose 0.1% as the cutoff to ensure that the instrumented timing measurements still reflected the program performance with high accuracy. Removing the instrumentation from these 114 loops, reduced the total execution time of the program by 46%. Ursa Minor currently provides the average execution times for computing the overhead. 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 serial. The relative importance of these loops in the serial version can be seen in Figure 6.

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 close relative of the new OpenMP industry standard [OMP97]. Browsing through the performance results displayed by Ursa Minor it was seen that on 4 processors, the KAP/Pro directive language exhibited superior performance.

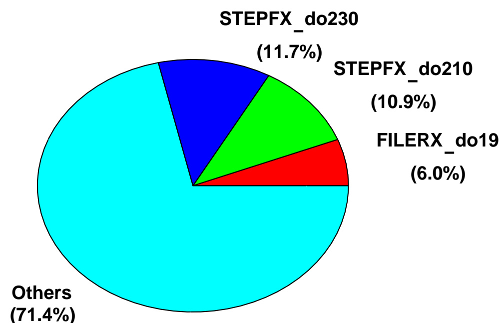

Figure 6: Percentage of execution time spent in major loops of ARC2D.

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 7.

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 [Vos97].

A further analysis of the serial source, the Polaris translated versions, and their graphical loop structure representation, showed that the two most significant loops STEPFX_do210 and 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 apply further optimizations. Figure 8 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.

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 6 through 8 can be generated by exporting the Ursa Minor/Major database to the Xess3 spreadsheet and using its graphing functions.

The full results of this study, performed on 8 benchmark programs across 4 multiprocessor architectures, can be interactively explored through the Ursa Major web page. Performance measurements obtained on a 4 processor SPARCstation 20, a 6 processor UltraSPARC Enterprise, a 16 processor Silicon Graphics Power Challenge and a 32 processors Origin 2000 have been made available as UMDs at

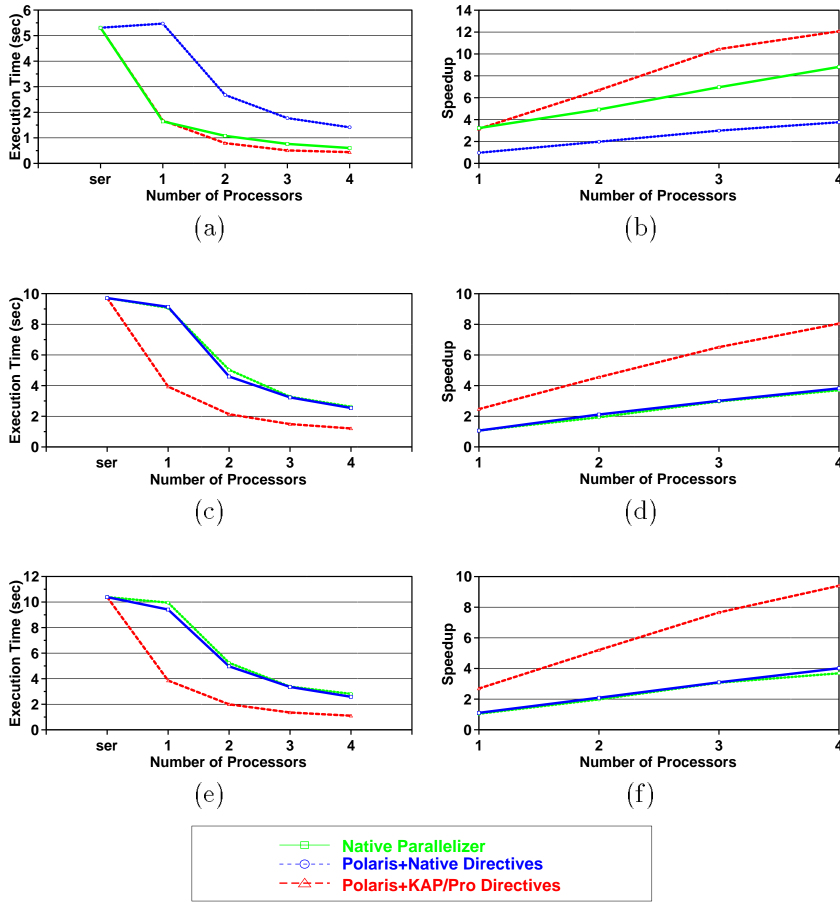

Figure 7: Loop performance of ARC2D on an UltraSPARC: (a) Execution time of FILERX_do19, (b) Speedup of FILERX_do19, (c) Execution time of STEPFX_do210, (d) Speedup of STEPFX_do210, (e) Execution time of STEPFX_do230 and (f) Speedup of STEPFX_do230.

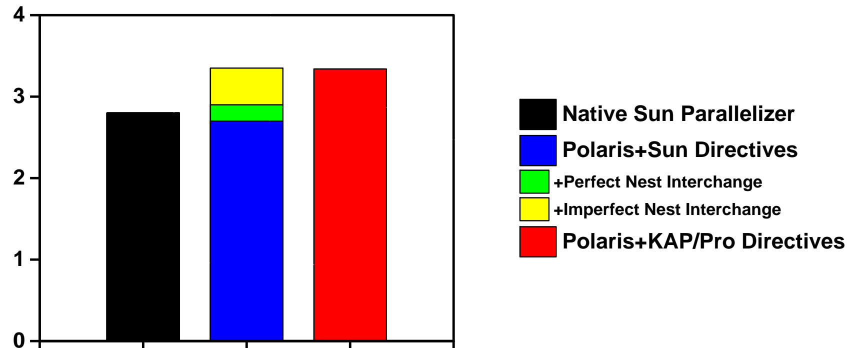

Figure 8: Performance of ARC2D on 4 Processors of UltraSPARC.

that site. For a detailed description of these results refer to [Vos97].

5.2 Experiment with the Seismic Application Suite

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], a seismic activity simulation program consisting of 20,000 lines of Fortran code. The Seismic Benchmark Suite contains a deep hierarchy of nested subroutines and loops. Our goal is to understand how the computational complexity of the overall application suite scales with the number of processors and with the input data space. Here, we will briefly describe how the Ursa Minor tool can be of help in the process of characterizing a large application.

To characterize an application's execution time we sum the times contributed by each loop. A loop's execution time, exclusive of any inner-loops, is estimated by obtaining an expression for the number of iterations the loop will execute and combining this expression with the average time per iteration of the loop from actual measurements. In order to do this we use the loop table view in Ursa Minor which provides average loop execution times as well as a loop's parent in the calling structure. With codes as large as the Seismic Suite the simple task of locating the beginning and ending of loops becomes cumbersome and prone to human errors. Ursa Minor greatly simplifies this task and provides a visual description of the loop nest hierarchy with its call graph view.

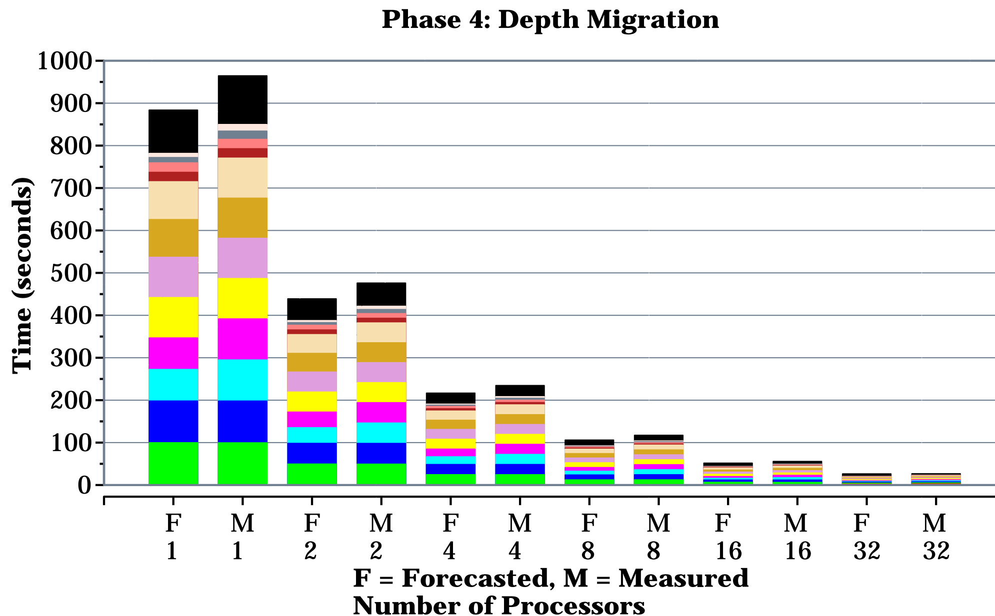

Figure 9: Actual measurements of loop execution times were compared with predicted times to determine the accuracy of the model on a loop-by-loop basis. The separate colors represent the loops of this seismic phase. The actual measurements were gathered using a 32-processor node of an SGI/Cray Origin2000 of NCSA at the University of Illinois.

After we characterized the code's performance, we used Ursa Minor to determine the accuracy of our characterization and locate the points needing refinement. Figure 9 compares actual measurements with our predicted times for one seismic processing phase (called *depth migration*) as the number of processors increases from 1 to 32. Ursa Minor aided in gathering the data from both the measurements and our model so that each loop's performance could be analyzed individually. Loops that scaled differently from the measured timings were easy to find. Our model could then be modified for more accurate

predictions. We used this process to test our model's scalability when the data size increased as well as when the number of processors increased.

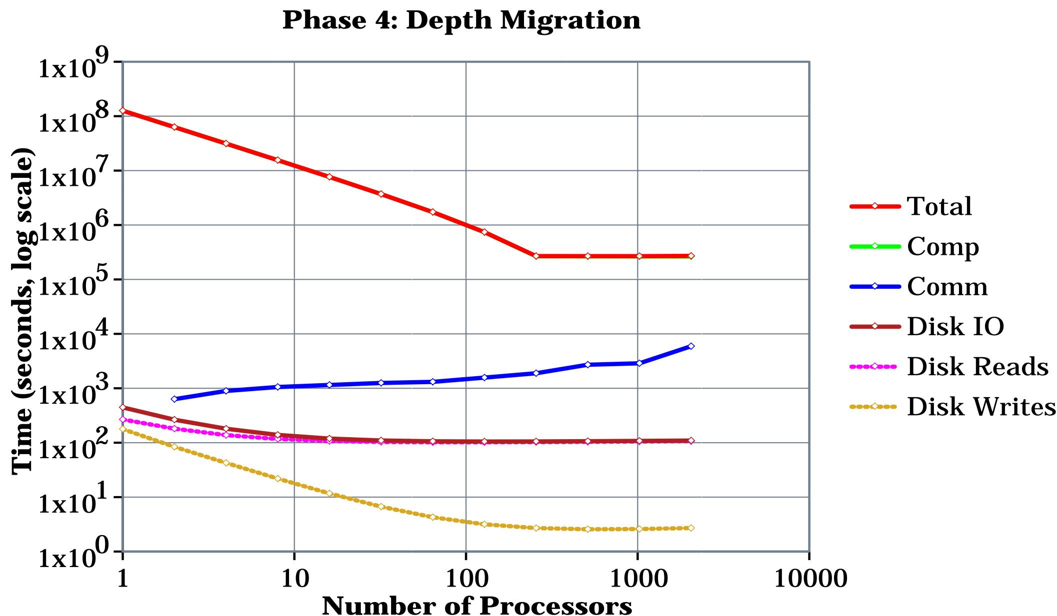

Figure 10: Forecasted performance of the Seismic Suite as the machine size is scaled up. The curves divide the total execution time into computation, communication, and disk IO times. The total time is dominated by the computation time (because of this the curves "Total" and "Comp" overlap).

The final goal of our characterizing process was extrapolating the Seismic Suite's performance to machines larger (more processors) than we currently have available and to input data sizes appropriate for such large machines. Databases of predicted execution times were exported from Ursa Minor for importing into an Xess3 spreadsheet, in which we produced graphs visually depicting the scalability of the application. Figure 10 shows extrapolation results for one seismic processing phase, again depth migration, as the number of processors is increased from 1 to 2,048 processors. The dataset is one which would use 3 terabytes of disk space.

Another objective of the Seismic Benchmark case study was to produce a well-performing loop-parallel program. As originally written, the Seismic Benchmark is a message-passing code. We investigated how well a loop-parallel version of the program would perform using Polaris as a starting point. Ursa Minor calculated the speedup of our loop-parallel program for each loop, flagging the loops with speedups below 1. These loops were then investigated further to improve their automatic parallelization by Polaris. If no improvements could be made, we forced a loop to execute serially so that it would not incur any parallel execution overhead.

The data from the Seismic Benchmark case study is currently available to outside users through the use of Ursa Major. Measurements were gathered using the SGI/Cray Origin 2000 at NCSA.

6 Conclusion

We have presented an on-going project that provides tools and methodologies for parallel program development and performance evaluation. Ursa Minor and Ursa Major support user models of "parallel programming by examples" for beginners and interactive compilation and performance tuning for experts. They also serve as a program and benchmark database for computing systems research. The

tools integrate information available from performance analysis tools, compilers, simulators, and source programs to a degree not provided by previous tools. Ursa Major can be executed on the World-Wide Web, from where a growing repository of information can be viewed.

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

Several enhancements are planned next. New categories of information will be integrated into the tools and their user views. For example, we will include improved compiler explanations why certain optimizations were or were not performed. This enables the programmer to input missing data to the compiler or to perform certain transformations by hand. Another important goal is the support for user methodologies. As a long-term goal we envision facilities that allow one to query the information repository directly for suggested improvements of programs, compilers, or architectures. Better support for the tool's Web response is another ongoing effort. As we have only begun to explore the potential offered by the new Internet technology, continuous feedback from its user community will help improve the tool's service to a world-wide audience.

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.

[AO88] J. Ambras and V. O'Day. MicroScope: A Knowledge-Based Programming Environment. *IEEE Software*, pages 50–58, May 1988.

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

[BDE+96] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[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.

[BST86] G. Bruno, P. Spiller, and I. Tota. AISPE: An Advanced, Industrial Software-Production Environment . *Proceedings of Computer Software and Applications Conf.*, pages 94–99, 1986.

[Eig93] Rudolf Eigenmann. Toward a Methodology of Optimizing Programs for High-Performance Computers. *Conference Proceedings, ICS'93, Tokyo, Japan*, pages 27–36, July 20-22, 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.

[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.

[KT87] J. H. Kuo and H. C. Tu. Prototyping a Software Information Base for Software-Engineeri ng Environments. *Proceedings of Computer Software and Applications Conf.*, pages 38–44, 1987.

[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.

[OMP97] OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Technical report, OpenMP, October 1997.

[PE98] Insung Park and Rudolf Eigenmann. URSA MAJOR: Exploring Web technology for design and evaluation of high-performance systems. In *Proc. of the International Conference on High Performance Computing and Networking*, April 1998.

[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.

[PVAE97] Insung Park, Michael J. Voss, Brian Armstrong, and Rudolf Eigenmann. Interactive compilation and performance analysis with Ursa Minor. In *Workshop of Languages and Compilers for Parallel Computing*, August 97.

[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.

[VGGJ+89] Jr. Vincent Guarna, Dennis Gannon, David Jablonowski, Allen Malony, and Yogesh Gaur. Faust: An Integrated Environment for the Development of Parallel Programs. *IEEE Software*, pages 20–27, July 1989.

[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 97.