Ursa Major: Exploring Web Technology for Design and Evaluation of High-Performance Systems

Insung Park Rudolf Eigenmann*

School of Electrical and Computer Engineering
Purdue University
November 13, 1997

Abstract

This paper explores opportunities and issues in the design and use of Web-based tools for the development and evaluation of high-performance computer systems and applications. Our discussion is based on a concrete tool, called Ursa Major, which addresses two specific problems with the design of parallel systems. First it deals with the fact that the growing popularity of multiprocessor workstations and high-performance PCs is leading to a substantial increase in non-expert users and programmers of this machine class. Such users need new programming paradigms - perhaps most importantly, they need good examples to learn from. Ursa Major serves these needs through a Web-accessible database of small and large programs, which are characterized extensively from several viewpoints. They include application properties, compiler-oriented views, performance characteristics, machine attributes, and user comments.

This publicly available repository serves a second purpose. It allows the study and comparison of experimental results from diverse sources. In this way, it addresses needs of benchmarkers who want to compare machine performance and of scientists who want to evaluate and compare new research prototypes and ideas.

A tool that has served similar needs within our research group has been described in previous reports. While this tool (called Ursa Minor), is successful in facilitating research and development within a project, Ursa Major extends the user community to a much wider audience: its Java implementation allows the operation across the internet. In developing the tool we have gained experiences in writing large Java applications, we have seen strength and limitations of the language and its environment, and - most importantly - we have gained experiences in using such a tool and its underlying technology for the design and evaluation of parallel machines and applications. This paper will share these experiences and present our vision of how the new internet technology and its combination with high-performance computing systems can be put to maximal use.

1 Introduction

Parallel high-performance machines are becoming mainstream technology. Today, affordable multiprocessor workstations and PCs are attracting an increasing number of users. Harnessing these parallel machines with user-oriented software systems is more important than ever before. One approach is to keep parallelism transparent. "Parallelizing compilers" attempt to do this [BEH$^+$94, HAA$^+$96]. They allow programs to be written in standard languages and then fit them automatically onto the parallel hardware. Indeed, this technology has become significantly more attractive than in the context of previous-generation supercomputers. This is due to recent progress in automatic parallelization, but

*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.

also because keeping a machine run at 100% efficiency all the time is no longer the overriding objective of a computer operation.

New parallel programming interfaces are needed more than ever. In addition to using this interface of "transparent parallelism," general programmers are also becoming more aware of parallel programming. In part this is because new programming languages, such as Java, provide multi-threading functionality. Furthermore, the new parallel language standard for multiprocessors, OpenMP [OMP97], promises an attractive interface for those programmers who find interest and value in exploiting parallelism explicitly. A programmer who is to develop a parallel program has to face a number of challenging questions: What parallel language is suitable for the given environment? What are the known techniques for parallelizing this program? What information is available for the program at hand? How much speedup can be expected from this program? What are the limitations for the parallelization of this program? It usually takes substantial experience to find the answers to such questions. Most general programmers do not have the time and resources to acquire this experience.

Many tools can gather raw program and performance information and present it to the user, which is a starting point for answering the questions above. However, in addition to providing raw information, advanced tools must help filter and abstract the potentially very large amount of data. Tools have to support a methodology of developing programs for parallel high-performance machines. The methodology will allow the user to comprehend the large data space and derive steps towards his or her objective.

Information exchange is essential for performance comparisons The tool presented here targets a second audience: the computer systems research community. 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 such programs can be found in the literature and on public data repositories, it is usually very difficult to combine them into a form meaningful for new purposes. In part this is because data are not readily available (e.g., they have to be extracted from several papers), but also because they may have to undergo substantial re-categorizations and transformations for their interpretation.

For example, one important way of understanding and improving application, compiler and machine behavior is to inspect high-performance applications and their timings on a loop-by-loop basis. However, program characteristics, compilation reports, timings numbers, cache statistics, and machine overheads are not usually available at this level.

The URSA MAJOR approach. URSA MAJOR addresses the described issues by providing an instrument with which application, machine, and performance information can be obtained from various sources and can be displayed at any viewer attached to the World-Wide Web. It provides a repository for this information and supports methodologies for its abstraction and comprehension. The methodologies support several goal functions. Industrial benchmarkers may be interested in "one single number" for machine comparisons; programmers may be interested in transformations that can improve the performance of an application program; computer architects may want to compare their cache measurements with those obtained by their peers. URSA MAJOR provides hooks for such methodologies and it includes instruments for the underlying data mining task.

The data repository has been constructed from the results gathered in various research projects. Currently it consists of characteristics of a number of programs, the results of compiler analyses of these programs, their performance numbers on diverse architectures, and the data generated in several simulator runs.

URSA MAJOR has its origin in the URSA MINOR tool [PVAE97], which was designed with the original goal of combining compiler information with performance results and of assisting the development process of parallel programs. Because we chose Java as an implementation language it was natural to combine these resources with the rapidly advancing Internet technology and, in this way, allow users at remote sites to access our experimental data. New goals appeared within reach. These are to create a comprehensive database of information for classes of users that go beyond our original parallel programmer community. 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 in a multitude of dimensions. The internet technology and its combination with high-performance computing tools open this new realm of questions and opportunities. This paper presents initial experiences that help us take advantage of the new potential.

Related work Use of the Internet for research in parallel computing has been made in several efforts. PUNCH (Purdue University Network Computing Hub), which enables distributed execution of programs, is one example [KLF96]. It enables users at remote sites to execute programs available at the Hub. It does not include the goal of presenting to the user information that leads to the understanding of high-performance computer systems and their improvements, which is one of our objectives.

Several Web tools offer performance numbers of various benchmarks [LaR93, Uni97]. Typically, the presented data are timing numbers such as overall program performance or specific timings of communication in message passing systems. Extensive characteristics of the measured programs are usually not part of the online databases. The user will have to obtain information from separate sources, which is often necessary for interpreting the numbers. Furthermore, these repositories do not provide information gathered by other tools, such as compilers or simulators, and consequently they do not support the comparison or the combined presentation of performance aspects and program characteristics.

Organization of the paper. Section 2 introduces the Ursa Major tool and discusses its functionality. Section 3 briefly introduces Ursa Minor [PVAE97], the tool on which Ursa Major is based. Section 4 describes the data repository. Section 5 discusses the experience we gained through this effort and gives our vision for the future of Ursa Major and for internet technology and high-performance systems in general. Section 6 concludes the paper.

2 Description of Ursa Major

2.1 Overview

The Ursa Major project has its root in the Ursa Minor tool [PVAE97], discussed in the next section. One objective of Ursa Major is to assist parallel programmers by effectively providing information on the characteristics and performance results of various programs. These data have been gathered through several high performance computing research efforts at Purdue University [Vos97, AE97, KMA+96]. While Ursa Minor operates in a local environment, the new tool is able to tranfer data from the Ursa Major repository (UMR) constructed on our server to remote sites. It comes with interactive features that allow selective views and combinations of the data. Figure 1 shows an overall view of the interaction between Ursa Major, a user, and UMR. Ursa Major is available at http://www.ecn.purdue.edu/~ipark/UM/UrsaMajor.html.

The information currently available to users includes

- basic program information, such as source codes and the subroutine/loop calling structure,
- results of program analysis tools, such as data dependence test summaries, parallelism profiles, variable value ranges, loop iteration counts, global/local data access ratios,
- performance results, overall, per subroutine, and loop-by-loop,

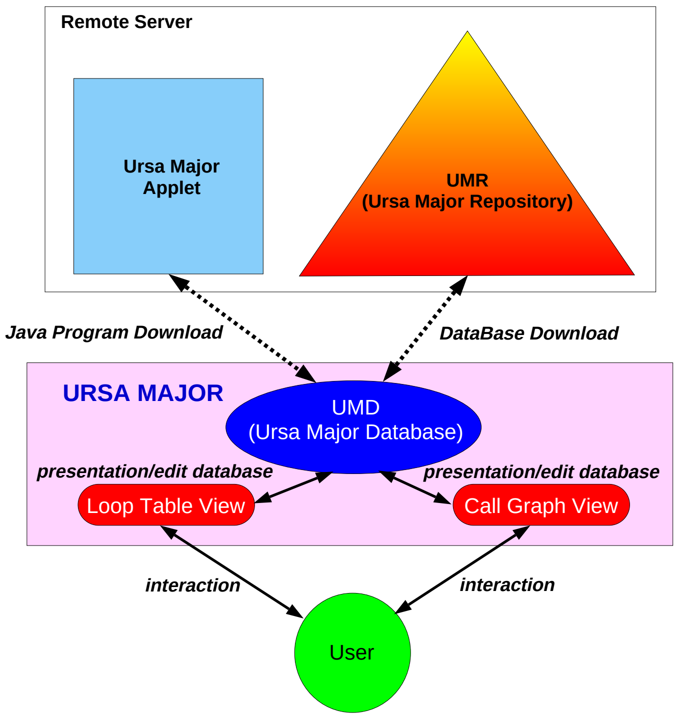


Figure 1: Interaction provided by the Ursa Major Tool.

- simulation results, such as the execution on an idealized dataflow machine (which gives an upper bound on the available parallelism in the programs)

The UMR is organized in a hierarchical file structure that contains information corresponding to likely user requests. The process of gathering data from various sources is discussed in Section 3.

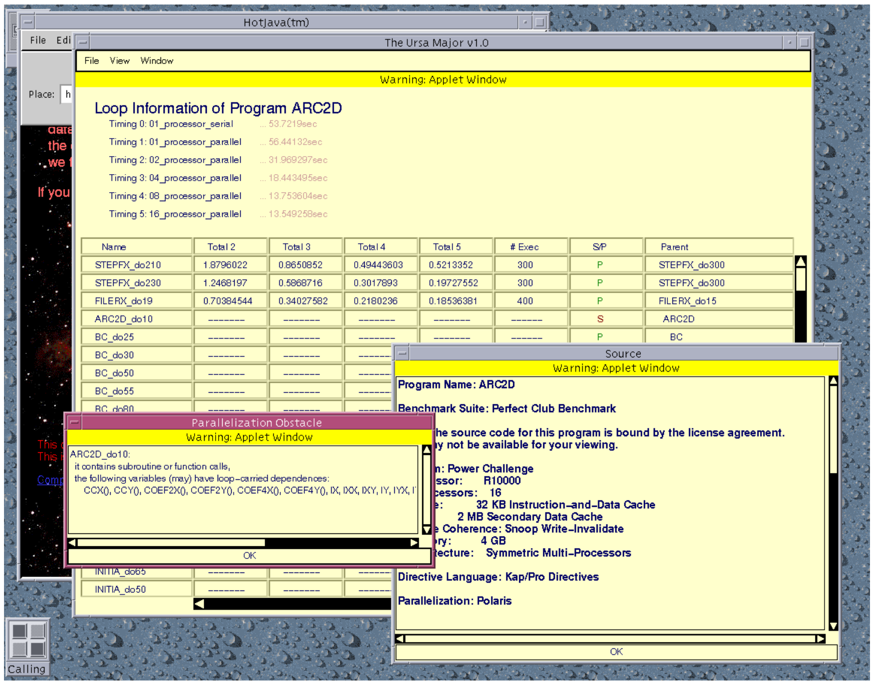
Ursa Major is closely related to the Polaris compiler infrastructure [BEH$^+$94]. Polaris, as a compiler, includes advanced program analysis and transformation techniques for array privatization, symbolic and nonlinear data dependence testing, idiom recognition, interprocedural analysis, and symbolic 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.

The Ursa Major tool is written in 8,000 lines of Java code. Thus, any platform on which a Java-compatible Web browser is available can be used to run Ursa Major. It uses the basic Java language with standard Application Programming Interfaces (APIs), which enhances the portability of the tool. The Java windowing toolkits and utilities enabled us to focus on the practicality of the tool. In the next section, we examine more closely the functionality of Ursa Major.

2.2 Functionality

In this section, we describe the interaction between Ursa Major and a user. Thetool is embedded in a Web page using the Java applet mechanism. The tool is invoked by clicking a button in the Web page.

An Ursa Major Database (UMD) is a storage unit that holds the collective information about a program and its execution results in a certain environment. The Ursa Major repository (UMR) is a hierarchical file organization that contain a collection of UMDs. A user is asked to choose a UMD



Loop Information of Program ARC2D

Timing 0: 01_processor_serial ... 53.7219sec
Timing 1: 01_processor_parallel ... 56.44132sec
Timing 2: 02_processor_parallel ... 31.969297sec
Timing 3: 04_processor_parallel ... 18.443495sec
Timing 4: 08_processor_parallel ... 13.753604sec
Timing 5: 16_processor_parallel ... 13.549258sec

Name	Total 2	Total 3	Total 4	Total 5	# Exec	S/P	Parent
STEPFX_do210	1.8796022	0.8650852	0.49443603	0.5213352	300	P	STEPFX_do300
STEPFX_do230	1.2468197	0.5868716	0.3017893	0.19727552	300	P	STEPFX_do300
FILERX_do19	0.70384544	0.34027582	0.2180236	0.18536381	400	P	FILERX_do15
ARC2D_do10	-------	-------	-------	-------	------	S	ARC2D
BC_do25	-------	-------	-------	-------	------	P	BC
BC_do30	-------	-------	------				
BC_do50	-------	-------	------				
BC_do55	-------	-------	------				
BC_do80	-------	-------	------				
INITIA_do65	-------	-------	------				
INITIA_do50	-------	-------	------				

Parallelization Obstacle

Warning: Applet Window

ARC2D_do10:
it contains subroutine or function calls,
the following variables (may) have loop-carried dependences:
CCX(), CCY(), COEF2X(), COEF2Y(), COEF4X(), COEF4Y(), IX, IXX, IXY, IY, IYX, I

OK

Source

Warning: Applet Window

Program Name: ARC2D

Benchmark Suite: Perfect Club Benchmark

he source code for this program is bound by the license agreement.
y not be available for your viewing.

n: Power Challenge
ssor: R10000
cessors: 16
e: 32 KB Instruction-and-Data Cache
2 MB Secondary Data Cache
e Coherence: Snoop Write-Invalidate
ry: 4 GB
tecture: Symmetric Multi-Processors

Directive Language: Kap/Pro Directives

Parallelization: Polaris

OK

Figure 2: Loop Table View of the Ursa Major Tool.

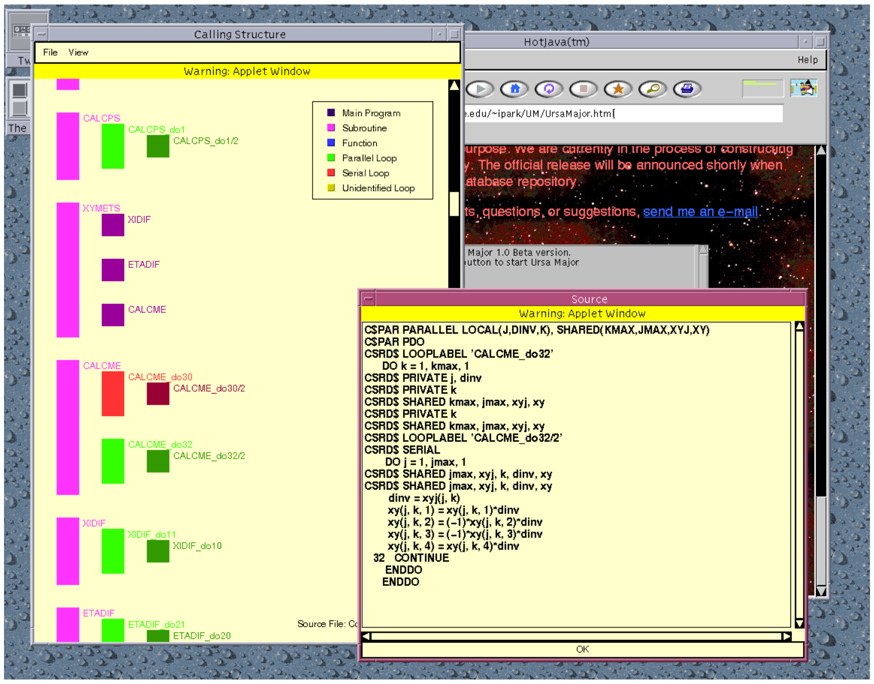
within the list of short descriptions of programs and their runtime environments. A textual description is available for each of the databases. It contains information about the program, the architecture that it ran on, the parallel language used, applied input sets, compilers and their options, the descriptions of optimizations resulting in the performance, and so on. This description is usually provided by the researcher who performed the experiment. It can serve as a starting point for inspecting a new program.

The user interacts with Ursa Major by choosing menu items and mouse-clicking, with pop-up dialog windows being used for occasional entry of parameters and for the display of small information items. The main display of the tool includes two windows: A loop information table and a call graph. The Ursa Major tool can hold many UMDs at one time, allowing users easy comparison.

Figure 2 shows the table, each line being the name of an individual loop. Currently, the table displays 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 the parallelizing compiler. If it is serial, the reason given by the compiler can be displayed on mouse-clicking. In Figure 2, the user is viewing the database of program ARC2D and its performance on a SGI Power Challenge machine. The user has clicked on loop `ARC2D_do10` to see the reason inhibiting parallelization. Also, the user opened a window displaying the textual description of this database.

A user can rearrange or hide columns to highlight certain information. The entries can be sorted alphabetically or based on execution time. By choosing a reference column, speedup can be calculated on-demand. Also, the percentage of each loop's execution time relative to the overall program, can be displayed through menu selection. This helps one determine the importance of each loop in the program.

Another view provides the calling structure of a given program, which includes subroutine, function, and loop nest information as shown in Figure 3. Each color-coded 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 `CALCME_do32` in program ARC2D in this way. Zoom-in/out functions of the call structure are available. An important role of this display is to help understand the nesting structure, which is important for tasks such as finding loops to interchange, and finding outer subroutines and loops for coarse-grain parallelzation.



```
C$PAR PARALLEL LOCAL(J,DINV,K), SHARED(KMAX,JMAX,XYJ,XY)
C$PAR PDO
CSRD$ LOOPLABEL 'CALCME_do32'
    DO k = 1, kmax, 1
CSRD$ PRIVATE j, dinv
CSRD$ PRIVATE k
CSRD$ SHARED kmax, jmax, xyj, xy
CSRD$ PRIVATE k
CSRD$ SHARED kmax, jmax, xyj, xy
CSRD$ LOOPLABEL 'CALCME_do32/2'
CSRD$ SERIAL
     DO j = 1, jmax, 1
CSRD$ SHARED jmax, xyj, k, dinv, xy
CSRD$ SHARED jmax, xyj, k, dinv, xy
      dinv = xyj(j, k)
      xy(j, k, 1) = xy(j, k, 1)*dinv
      xy(j, k, 2) = (-1)*xy(j, k, 2)*dinv
      xy(j, k, 3) = (-1)*xy(j, k, 3)*dinv
      xy(j, k, 4) = xy(j, k, 4)*dinv
  32  CONTINUE
     ENDDO
    ENDDO
```

Figure 3: Annotated Call Graph View of the Ursa Major Tool.

3 The Ursa Minor project: the Foundation of Ursa Major

Ursa Minor is the basis for the tool project presented in this paper. The difference between the two projects is that Ursa Minor attempts to provide advanced programmers with a parallel programming environment, whereas Ursa Major was designed to distribute our experiences and results to a more general audience.

One objective of the Ursa Minor project is to promote a parallel programming methodology by providing an integrated browser for programs, compilation, and performance data. Ursa Minor helps organize data (source files, itermediate files, results) in the process of developing, compiling, and running a program. It also helps identify potential performance bottlenecks and improvements. Ursa Minor combines static program information provided by compilers with performance results of program runs. In doing so it incorporates capabilities of parallel programming tools pioneered by tools such as Pablo [Ree94], Paradyne [MCC+95], PTOPP [EM93], Polaris [BEH+94] and SUIF [HAA+96].

In a typical scenario, the programmer uses the information provided by Ursa Minor as a starting point for analyzing and tuning the performance of a program. From this information he or she derives and applies further program optimizations. In this process, the tool provides feedback (in the form of updated program analysis results and performance factors) and facilitates the gathering and display of new results.

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.

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. For more information about the Ursa Minor tool, the reader is referred to [PVAE97].

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

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 are being used to gather and organize such data [VGGGJ88, EM93, BST86, AO88, KT87].

One issue in designing the repository was to define storage schemes that makes is easy for users to find information entered by other users. To this end, the repository structure consists of extentions 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.

5 Experiences using Ursa Major

We present early experiences using Ursa Major. We have taken advantage of the tool in our research team, on multiple platforms, including wokstations, high-performance servers and 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.

Supporting Ursa Major's Users

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". Typically we tell new programmers to exploit the tool doing the following steps:

- Get a general feel for the programs in 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. The call graph view includes both subroutines and loops and provides color-coded information about which program sections will run in parallel.
- 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 diplayed in Figure 2. Source code corresponding to the 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.

There is no clear cut between the beginner and the advanced programmer. Even the expert may want to start with the described steps and learn from parallel programs developed by others. In addition, the advanced user will

- inspect the reasons why certain parallel loops or program sections perform well or badly in more detail. For example, small average loop execution times may cause large parallel loop startup overheads, and large numbers of global data references may cause poor cache locality – both limiting speedups. This analysis can be done either by inspecting the raw data or by relying on suggestions given by the tool. In a simple case the tool can flag "bad situations", such as speedups less than one. More advanced options will derive a set of optimization factors that characterize the performance and that give clues for possible improvements. In further scenarios the tool would filter and abstract the raw information, giving the user suggestions towards his or her objective. Developing such suggestion methods is an important ongoing research area.
- understand reasons why a code section is not parallel. Ursa Major can present the explanations given by the parallelizing compiler. Together with the serial and parallel program the user can come to a good understanding of the kind of transformations that make a program parallel. Also one can find the limitations of automatic transformations, and what has to be done to perform them by hand. An additional aid in this process is the column "Maximum Parallelism" which gives the user a clue about the theoretically obtainable degree of parallelism, against which the success can be measured.
- include the program under development into the Ursa Major tool and comparing it with available programs. Potential improvements can be indentified in this way. Additional information is available that helps determine further improvements. For example, information on accessed array sections in a loop is important for finding which arrays contain data dependences and which arrays are privatizable. Information about inner and outer loops helps identify candidate loops for interchanging (improving cache locality) and for increasing the grain size of the parallel computation (reducing startup overheads). Again, the loop table turned out to be most useful for providing this type of information.

Ursa Major further serves the research community in general by making available the large about 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. Enhancing the support for inspecting our database from these various angles is another important ongoing effort.

The loop table view has turned out to be crucial in the use of Ursa Major. Most information items are associated with a loop in some manner and, hence, can adequately be identified by the user via the loop table. Loops can represent the innermost computation of a code as well as the program as a whole (e,g., the outermost time-stepping loop of an application). Nevertheless there are data sets that need a different representation. Examples are the time line of subroutine calls in a program and the trace file generated by a simulator. Including such information is a future goal.

Experience with Java

In the process of developing a relatively large Java application, we found both advantages and disadvantages of this language. The simple design and ease of debugging significantly increased productivity throughout the development. The support for graphical user interfaces at the basic language level increased the portability and readibility. The networking features played a key role in developing our application. One issue that made our design difficult was the young age of Java, with several upgrades happening within a single year, accompanied by the instability inherent to early releases. Because our application is large (8000 lines of Java) the load time at a remote site can be significant. We expect this situation to improve as network bandwidths increase world-wide. Finally, despite the full language support for relatively high level graphical functions, a significant portion of our codes had to be written for the user interfaces. As visual Java programming modules are emerging, we expect that a significant portion of this code can be built from libraries in the future. Overall, the Ursa Major project proved the value of Java as a general purpose language for major applications and for network applications in particular.

Experience with Ursa Major as a Web Tool

In Ursa Major the operation across platforms and across distant nodes is made easy. This is thanks to the underlying language and the internet server technology. We need no longer develop tools that are designed specifically for serving internet requests, such as netlib and internet database servers [LaR93, Uni97] (which are notable pioneers of this technology). The facilities are now part of programming environments. Java is an example. Ursa Major just takes advantage of this fact. As it does so, it also emphasises an important question raised about the new internet technology: how to deal with the large amount of available data. Structuring, filtering, and abstracting the new "world-encompassing database" has become the new Grand Challenge. For our tool this means, specifically, to find a methodology for interpreting the program, compiler, and performance information that can guide the user in achieving the desired improvement of the application, the compiler, or the underlying machine. Achieving this goal will take many additional steps such as the one presented in this paper.

6 Conclusion

We have presented an on-going project that provides tools and methodologies for parallel program development and performance evaluation. Ursa Major supports a user model of "parallel programming by examples" and it serves as a program and benchmark database for high performance computing. It integrates 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.

Ursa Major 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 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 tool and its 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, February 97.

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

[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.

[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.

[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.

[KE97] Seon-Wook Kim and Rudolf Eigenmann. *Max/P: detecting the maximum parallelism in a Fortran program*, 1997. Manual ECE-HPCLab-97201.

[KLF96] Nirav H. Kapadia, Mark S. Lundstrom, and Jose A. B. Fortes. A network-based simulation laboratory for collaborative research and technology transfer. In *Semiconductor Research Corporation's TECHCON '96*, September 1996.

[KMA+96] M. Kandaswamyi, Z. Ben Miled, B. Armstrong, S. Kim, V. Taylor, R. Eigenmann, and J. Fortes. The Design of a Hierarchical Processor-and-Memory Architecture for High Performance Computing. *Proceedings of the Workshop on the Petaflops Frontier at the Sixth Symposium on the Frontiers of Massively Parallel Computationssing*, Oct., 1996.

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

[LaR93] B. LaRose. The Development and Implementation of a Performance Database Server. Technical Report CS-93-195, Univ. of Tennessee, August 1993.

[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.

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

[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.

[Uni97] The University of Southampton. *GRAPHICAL BENCHMARK INFORMATION SERVICE (GBIS)*, 1997. http://www.ccg.ecs.soton.ac.uk/gbis/papiani-new-gbis.html.

[VGGGJ88] Jr. Vincent Guarna, Dennis Gannon, Yogesh Gaur, and David Jablonowski. Faust: An Environment for Programming Scientific Applications. *Proceedings of Supercomputing '88*, pages 3–10, November 1988.

[Vos97] Michael J. Voss. Portable loop-level parallelism for shared memory multiprocessor architectures. Master's thesis, 97. (in preparation).