Parallel Programming Environment for OpenMP[*]

Insung Park Michael J. Voss Seon Wook Kim Rudolf Eigenmann
1285 EE Bldg., School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285[†]
ipark,mjvoss,seon,eigenman@ecn.purdue.edu

November 4, 2002

Abstract

We present our effort to provide a comprehensive parallel programming environment for the OpenMP parallel directive language. This environment includes a parallel programming methodology for the OpenMP programming model and a set of tools (Ursa Minor and InterPol) that support this methodology. Our toolset provides automated and interactive assistance to parallel programmers in time-consuming tasks of the proposed methodology. The features provided by our tools include performance and program structure visualization, interactive optimization, support for performance modeling, and performance advising for finding and correcting performance problems. The presented evaluation demonstrates that our environment offers significant support in general parallel tuning efforts and that the toolset facilitates many common tasks in OpenMP parallel programming in an efficient manner.

1 Introduction

Today, new affordable multiprocessor workstations and high performance PCs are attracting a large number of users. Many new programmers are inexperienced and demand an easy programming model to harness the power of parallel computing. The recent parallel language standard for shared memory multiprocessor machines, OpenMP [7], promises a simple interface for those programmers who wish to exploit parallelism explicitly. The OpenMP standard resolves a significant portability problem that has been associated with shared memory parallel programming. It is expected to attract an increasing number of programmers and computer vendors in the high performance computing area.

However, there are open issues to be addressed. Perhaps the most serious of all is the lack of a good programming methodology. A programmer who is to develop a parallel program faces a number of challenging questions: 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 answer such questions. Many programmers do not have the time and resources to acquire this experience.

A useful methodology must provide structured guidelines that encompass the whole process of program development, while providing useful tips with which users can navigate through difficult steps. The motivation to develop such a methodology came from our prior research efforts in parallelizing programs for different target architectures [9]. After a great deal of trial and error as novice programmers, we have developed a structured way to a successful optimization of programs. As the number of the programs that we dealt with increased, our general methodology went through several iterations of adjustment and improvement. Finally, we decided to document it so that a wider range of programmers can benefit.

A programming methodology is not useful if it cannot be supported by tools. That is, it is not of much help to programmers to list the programming tasks that need to be performed, if all those tasks must be

[*]This work was supported in part by NSF grants #9703180-CCR, #9872516-EIA, and #9975275-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government.

[†]Current affiliations of authors: I. Park is with Microsoft Corp., M. Voss is with the University of Toronto, S.W. Kim is with Intel Corp.

accomplished manually with only basic utilities. Among these tasks are performance data analysis and management, incremental application of parallelization and optimization techniques, performance measurement and monitoring, and problem identification and devising remedies. Each of them puts a significant burden onto programmers and can be very time-consuming if not facilitated by tools.

There are many tools designed for the purpose of helping programmers accomplish these tasks. However, they usually focus on specific aspects or environments in the program development process and are not based on a common underlying methodology. The goal of our work is to resolve these shortcomings and provide a comprehensive and actively guiding toolset.

We have developed a methodology that has worked well under various environments and a set of tools that address difficult tasks in the OpenMP programming model. Combining our methodology and supporting tools, programmers can now follow a structured approach toward optimizing the performance of parallel programs.

The remainder of the paper is organized as follows. Section 2 presents our proposed programming methodology. Supporting tools are described in Section 3. Section 4 discusses and evaluates the tools. Related work is presented in Section 5. Section 6 concludes the paper.

2 Parallel Programming Methodology

Figure 1 shows the parallelization and optimization steps envisioned by our proposed methodology. The methodology envisions the following tasks when porting an application program to a parallel machine and tuning its performance. We start by profiling the program execution time, usually on a loop-by-loop basis. We do this by instrumenting the program with calls to timer functions or by using hardware counters. The timing profile not only allows us to identify the most important code sections, but also to monitor the program's performance improvements as we convert it from a serial to a parallel program.

The first step of performance optimization is to apply a parallelizing compiler. Ideally, a parallelizing compiler would be able to generate a highly-tuned application that exploits all available parallelism. However in practice [8], this is not always possible. Users may be required to assist in the parallelization process due to both compiler limitations and intrinsic dependences in the target application. If a compiler limitation leads to missed parallelism, a user may simply force parallelization. If parallelism is limited due to an intrinsic dependence, the user may be able to substitute a parallel algorithm. While it may seem that any interaction that a "novice" user can perform should be automatable, being aware of the intent of an application allows users to perform transformations that are still beyond the capabilities of the best research compilers.

While our methodology begins with the application of a parallelizing compiler, if no such tool is available, we can apply program transformations by hand. After the initial parallelization step, whether automatic or manual, we identify time-consuming code sections of the program and optimize their performance using several recipes. More detailed steps and suggestions provided by this methodology can be found in [20].

There are two feedback loops in the Figure 1. The first one reduces excessive overhead introduced by program instrumentation. This overhead not only affects the program's performance. It can also skew the execution profile, so that programmers focus their efforts on the wrong program sections. To measure the overhead one runs the program with and without instrumentation. If necessary, instrumentation can be removed from innermost code sections, or from sections that contribute little to the overall execution time.

The second loop is the actual optimization process, where one applies tuning techniques and evaluates their performance. Tuning steps may need to be modified or even undone if their result is not satisfactory. After each step one identifies the new most time-consuming program section on which to target the next tuning step.

The methodology described above has been empirically devised. It tells programmers "what should be done" in program tuning. The specific tasks in each step are described in detail in [20]. We support this methodology with a set of tools, which answers the question of "how" the tasks can be performed. These tools are described in the next section.

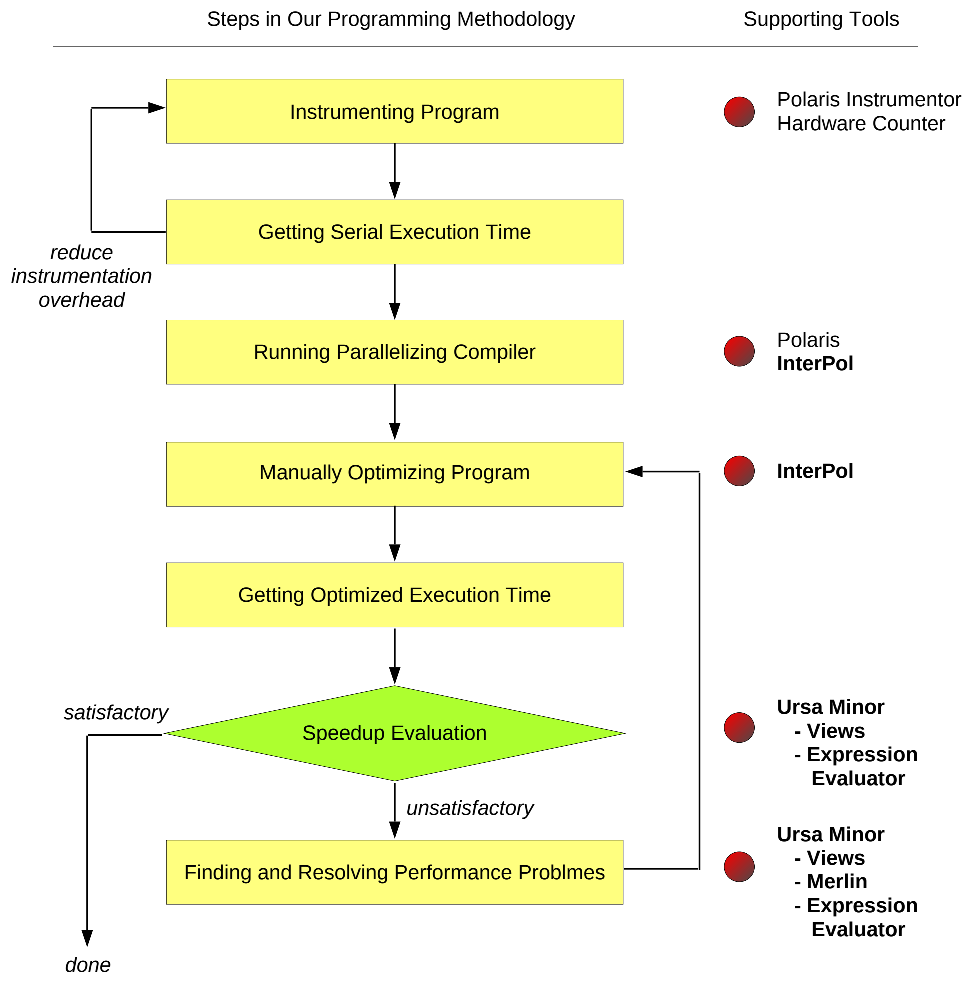


Figure 1: Overview of the proposed performance optimization methodology and supporting tools. Bold-faced tools are described in this paper.

3 Tool Support

Parallel programmers without access to parallel programming tools usually rely on text editors, shells, and compilers. Programmers write codes using text editors and generate an executable with resident compilers. All other tasks such as managing files, examining performance figures, searching for problems and incorporate solutions, is usually achieved without special-purpose tools. However, considerable effort and good intuition are needed in file organization and performance diagnosis. Even with the help of parallelizing compilers, these tasks still remain for the users to deal with. In fact, most users end up writing small helper scripts for many tedious programming tasks. Our new tools step in where these traditional tools have limits. We have set the following design goals.

Consistent support for methodology: This is the main goal of our research. We examine the steps in the methodology and find time-consuming programming chores that call for additional aid. Some tasks are tedious and may be automated. Some require complex analysis, thus assisting utilities are needed. The integration of the methodology and the tool support significantly increase programmers' efficiency and productivity.

Support for performance modeling: In addition to providing raw performance information, advanced tools must help filter and abstract a potentially large amount of data. The ability to flexibly manipulate data and to combine them into high-level performance models, allows users to reason about the performance behavior of a program in a flexible way.

Active guidance system: Tuning programs requires dealing with numerous different instances of code segments. Categorizing these variants and finding the right remedies for performance problems requires experience. We believe that it is possible to address this issue systematically through automated analysis utilities.

Program characteristic visualization and performance evaluation: The task of improving program performance starts with examining performance and program analysis data and finding potential improvements. The ability to browse through this information and visualize it is critical. Tables, graphs, and charts are common ways of expressing a large set of data for easy comprehension.

Integration of program analysis with performance evaluation: Most tools focus on either static program analysis data or performance data. However, good performance only comes from considering both aspects. It is important to identify the relationship between the data from both sides. Without the consideration of performance data, static program optimization can even degrade the performance. Likewise, without the static analysis data, optimization based only on performance data may be marginal.

Interactive and modular compilation: The usual black-box-oriented use of compiler tools have limits in efficiently incorporating users' knowledge of program algorithms and dynamic behavior. Manual code modification in addition to automatic parallelization is often necessary to achieve good performance, and tools should support convenient mechanisms for the incorporation of manual tuning. Another drawback of conventional compilers is their limited support for incremental tuning. The localized effect of parallel directives in the OpenMP programming model allows users to focus on small portions of code for possible improvement in an incremental manner. Hence, the compiler support for incremental tuning is also an important goal in our tool design.

Data Management: In the process of compiling a parallel program and measuring its performance, users experiment with a number of program variants under many different environments. As a result, a considerable amount of information is produced. Managing this information for easy later retrieval and comprehension is a challenge. This is even more for work done in a team, where proper labeling of data and recording of the experimental environment is critical. Therefore, support for experiment data management is an important design goal.

Accessibility: Although the importance of advanced tools for all software development is evident, many available tools remain unused. A major reason is that the process of searching for tools with needed capabilities, downloading and installing them on locally available platforms and resources is very time-consuming. In order to evaluate and find an appropriate tool, this process may need to be repeated many times. Using today's network computing technology, tool accessibility can be greatly enhanced.

Configurability: In order to satisfy a wide community of users, tools must allow individuals to set preferences. By having configurability as one of our design goals, many users' preferences can be incorporated into the tool usage without writing special-purpose utilities.

Flexibility: Flexibility is an important characteristic of general tools. We have seen many situations where users wished to incorporate new types of performance data into their tools. Advanced tools must be open to the type of data that can be included and presented.

Scalability: Tools must work not only for small demonstrations, but also for the large, realistic field use. Scalability can be with respect to several parameters. Our primary concern is that tools be able to handle large science and engineering applications of 100,000 lines of code. In our OpenMP environment we will envision target machines of typically not more than 32 processors. Hence, tool scalability to massively parallel systems is not a primary concern.

In the following sections, we introduce two tools developed based on these goals. Ursa Minor is a performance evaluation tool designed to assist users in parallel performance evaluation. InterPol is an interactive tool for automatic and manual program transformations. While these tools do not directly interface with each other, they provide complementing functionality.

Ursa Minor and InterPol are closely related to the Polaris compiler [4], a source-to-source restructurer, developed at the University of Illinois and Purdue University. Polaris automatically finds parallelism and inserts appropriate parallel directives into the input programs. It includes advanced capabilities for array privatization, symbolic and nonlinear data dependence testing, idiom recognition, interprocedural analysis, and symbolic program analysis. In addition, the current Polaris tool is able to generate OpenMP parallel directives [7] and apply locality optimization techniques such as loop interchange and tiling. Polaris also serves as an instrumentation tool.

We have integrated these tools into a Web-executable programming environment, referred to as the Parallel Programming Hub. It provides "anytime, anywhere" access to our tools via a network computing infrastructure developed in a related project [11]. The Parallel Programming Hub is available at http://punch.ecn.purdue.edu/Netcare/parHub.html.

3.1 Ursa Minor: Performance Evaluation Tool

The Ursa Minor tool assists parallel programmers in the performance evaluation and tuning process [21]. It presents users with information available from various sources in a comprehensive way. These sources include tools such as compilers, profilers, hardware counters, and simulators. It interacts with users through a graphical interface, which can provide selective views and combinations of the data. Ursa Minor combines the performance and static analysis data in integrated views. It provides facilities for performance modeling of the gathered data, and it includes a performance advisor that automates the process of finding performance problems and remedies. Ursa Minor supports the OpenMP parallel programming model.

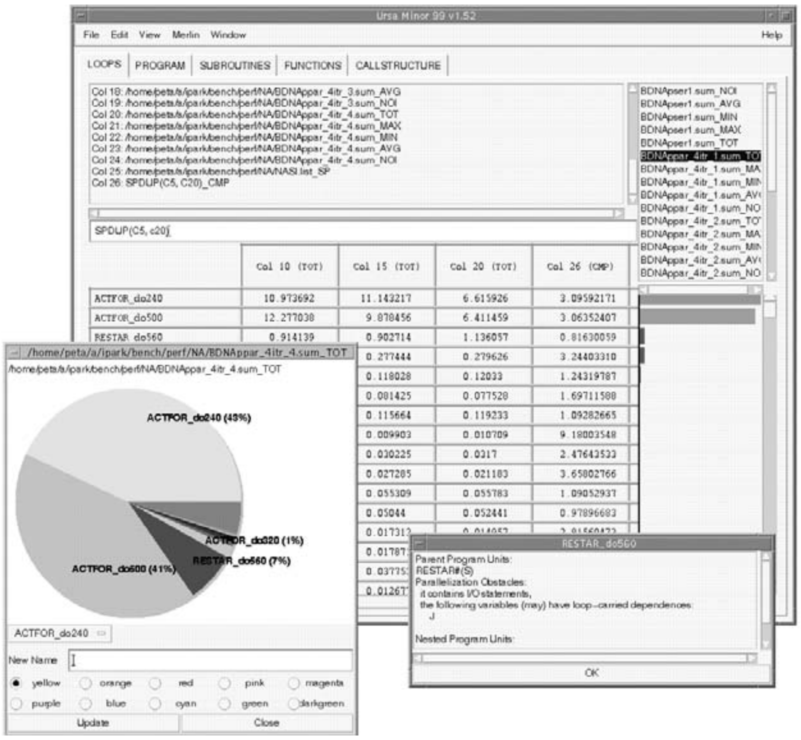
Tool Functionality

Performance data and program structure visualization The Ursa Minor tool presents information to the user through two main display windows: The Table View and the Structure View. The Table View shows performance data as text entries for subroutines, functions, loops, blocks, or other, user-defined program sections. The Structure View visualizes the program's subroutine call and loop nest structure.

Figure 2 shows the Table View of Ursa Minor in use. The Table View displays data such as average execution time, the number of invocations of code sections, cache misses and text indicating if loops are serial or parallel. Users can manipulate the data through various features this view provides. The Table View is a tabbed folder that contains one or more tabs with labels. Each tab corresponds to a "program unit group", which means a group of data of a similar type. For instance, the folder labeled "LOOPS" contains all the data regarding loops in a given program. When reading predefined data inputs such as timing files, Polaris listing files, and simulation results from MAX/P [12], Ursa Minor generates predefined program unit groups, (e.g., LOOPS, PROGRAM, etc.). Users can create additional groups with their own input files using a proper format.

A user can rearrange columns, delete columns, sort the entries alphabetically or based on execution time. The bar graph on the right side shows an instant normalized graph of a numeric column. After each program run, the user can include the newly collected information as additional columns in the Table View. 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 has to focus next.

In addition, users can set a display threshold for each column so that an item that is less than a certain quantity is displayed in a different color. For example, this feature allows users to easily identify code sections with poor speedup. One or more rows and columns can be selected so that they can be manipulated as a whole. Data that would not fit into a table cell, such as the compiler's explanation for why a loop is not parallel, can be displayed in a separate window on demand. Finally, Ursa Minor is capable of generating


Ursa Minor 99 v1.52
File Edit View Merlin Window Help
LOOPS PROGRAM SUBROUTINES FUNCTIONS CALLSTRUCTURE
ACTFOR_do240 (43%)
ACTFOR_do320 (1%)
RESTAR_do560 (7%)
ACTFOR_do500 (41%)
ACTFOR_do240
New Name
yellow orange red pink magenta
purple blue cyan green darkgreen
Update Close
OK

```
Col 18: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_3.sum_AVG
Col 19: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_3.sum_NOI
Col 20: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_TOT
Col 21: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_MAX
Col 22: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_MIN
Col 23: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_AVG
Col 24: /home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_NOI
Col 25: /home/peta/a/ipark/bench/perf/NA/NASI.list_SP
Col 26: SPDUP(C5, C20)_CMP
```

```
BDNApser1.sum_NOI
BDNApser1.sum_AVG
BDNApser1.sum_MIN
BDNApser1.sum_MAX
BDNApser1.sum_TOT
BDNAppar_4itr_1.sum_TOT
BDNAppar_4itr_1.sum_MA
BDNAppar_4itr_1.sum_MIN
BDNAppar_4itr_1.sum_AV
BDNAppar_4itr_1.sum_NO
BDNAppar_4itr_2.sum_TOT
BDNAppar_4itr_2.sum_MA
BDNAppar_4itr_2.sum_MIN
BDNAppar_4itr_2.sum_AV
BDNAppar_4itr_2.sum_NO
```

```
SPDUP(C5, c20)
```

	Col 10 (TOT)	Col 15 (TOT)	Col 20 (TOT)	Col 26 (CMP)
ACTFOR_do240	10.973692	11.143217	6.615926	3.09592171
ACTFOR_do500	12.277038	9.878456	6.411459	3.06352407
RESTAR_do560	0.914139	0.902714	1.136057	0.81630059
		0.277444	0.279626	3.24403310
		0.118028	0.12033	1.24319787
		0.081425	0.077528	1.69711588
		0.115664	0.119233	1.09282665
		0.009903	0.010709	9.18003548
		0.030225	0.0317	2.47643533
		0.027285	0.021183	3.65802766
		0.055309	0.055783	1.09052937
		0.05044	0.052441	0.97896683
		0.017312	0.014057	2.01560472
		0.01787		
		0.03775		
		0.01267		

```
/home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_TOT
/home/peta/a/ipark/bench/perf/NA/BDNAppar_4itr_4.sum_TOT
```

```
RESTAR_do560
Parent Program Units:
RESTAR#(S)
Parallelization Obstacles:
  it contains I/O statements,
  the following variables (may) have loop-carried dependences:
    J

Nested Program Units:
```

Figure 2: Table View of the Ursa Minor tool. The user has gathered information on program BDNA. After sorting the loops based on execution time, the user inspects the percentage of three major loops (ACTFOR_do240, ACTFOR_do500, RESTAR_do560) using a pie chart generator. Computing the speedup (column 26) with the Expression Evaluator reveals that the speedup for RESTAR_do560 is poor, so the user is examining more detailed information on the loop.

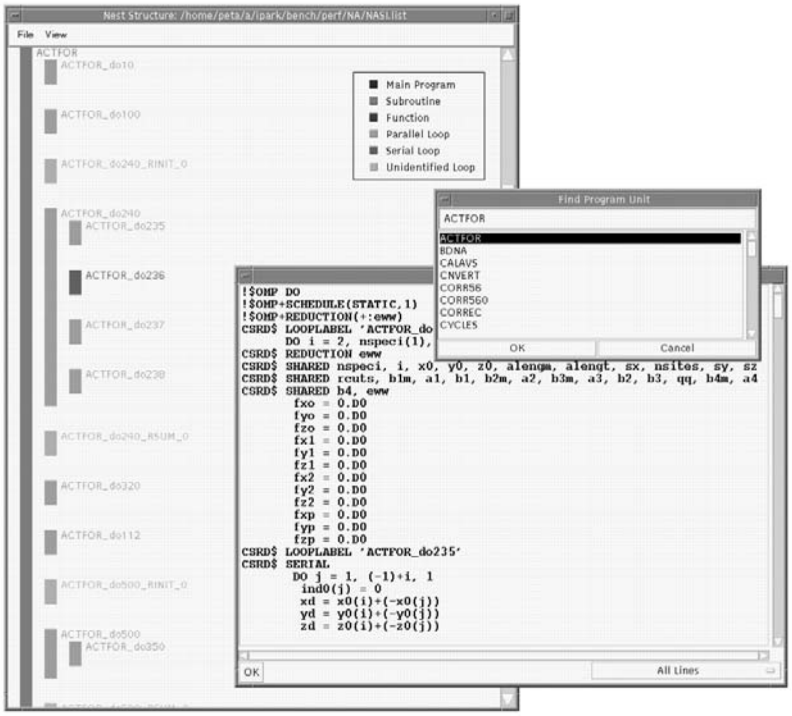
pie charts and bar graphs on a selected column or row for instant visualization of numeric data within the tool.

Another view of Ursa Minor displays the program's calling structure, which includes subroutine, function, and loop nest information, as shown in Figure 3. Each rectangle represents either a subroutine, function, or loop. The rectangles are color-coded to display additional attributes. Users can activate the source viewer for each rectangle by a mouse click. We have added a flexible zoom utility and the support for OpenMP directives in the source viewers. This display helps one understand the program structure for tasks such as interchanging loops or finding outer and inner candidate parallel loops.

The Ursa Minor user interface is configurable. Users can change the look of the various displays and many other parameters. Most tool functions can be mapped to keyboard shortcuts. Furthermore, an "on-line tutorial" allows users to explore important features of the tool with sample input data.

Expression Evaluator The ability to compute derivative values of raw performance data is critical in analyzing the gathered information. For instance, the average timing value of different runs, speedup, parallel efficiency, and the percentage of the execution time of a code section with respect to the overall execution time of the program are common metrics used by many programmers. Instead of adding individual utilities to compute these values, we have added the Expression Evaluator for user-entered expressions. We have provided a set of built-in mathematical functions for numeric, relational, and logical operations. Nested operators are allowed, and any reasonable combination of these functions are supported. The Expression Evaluator has a pattern matching capability as well, so the selection of a data set for evaluation becomes simplified.

The Expression Evaluator also provides users with query functions that apprehend the static analysis data from a parallelizing compiler. These functions can be combined with the mathematical functions,



```
!$OMP DO
!$OMP+SCHEDULE(STATIC,1)
!$OMP+REDUCTION(+:eww)
CSRD$ LOOPLABEL 'ACTFOR_do
      DO i = 2, nspeci(1),
CSRD$ REDUCTION eww
CSRD$ SHARED nspeci, i, x0, y0, z0, alengm, alengt, sx, nsites, sy, sz
CSRD$ SHARED rcuts, b1m, a1, b1, b2m, a2, b3m, a3, b2, b3, qq, b4m, a4
CSRD$ SHARED b4, eww
        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$ SERIAL
        DO j = 1, (-1)+i, 1
         ind0(j) = 0
         xd = x0(i)+(-x0(j))
         yd = y0(i)+(-y0(j))
         zd = z0(i)+(-z0(j))
```

Figure 3: Structure View of the Ursa Minor tool. The user is looking at the subroutine and loop call graph generated for program `BDNA`. Using the "Find" utility, the user set the view to subroutine `ACTFOR`, and opened up the source view for the parallelized loop `ACTFOR_do240`.

allowing queries such as "loops that are parallel and whose speedups are less than 1" or "loops that have IO and whose execution time is larger than 15 % of the overall execution". The Expression Evaluator is a powerful utility that allows manipulating and restructuring the input data to serve as the basis for the users' performance modeling through a common spreadsheet-like interface.

The Merlin Performance Advisor Identifying performance bottlenecks and finding the right remedies usually take experience and intuition, which novice programmers lack. Acquiring this expertise requires many trials and studies. Even for those programmers who have experienced peers, the transfer of knowledge takes time and effort. We have used a combination of the above Expression Evaluator and a knowledge-based database to create a framework for easy "transfer of experience". Merlin is an automatic performance data analyzer that allows experienced programmers to tell novice programmers how to handle specific performance symptoms.

Merlin navigates through a knowledge-based database ("map") that contains the information on diagnosis and solutions for various performance symptoms. Advanced programmers write maps based on their experience, and novice programmers can make use of this experience by activating Merlin. A Merlin map enables multiple cause-effect analyses of performance and static data efficiently. Merlin also assist users in "learning by examples". Merlin is able to work with any map as long as the map is in the correct format, which widens its applicability. A more detailed description of Merlin is available in [13]. We present a case study emphasizing the use of Merlin in Section 4.3.

Internal Organization of the Ursa Minor tool

Figure 4 illustrates the interactions between Ursa Minor modules and the various data files. The Database Manager handles interaction between the database and other modules. Upon users' requests, it fetches the required data items or creates and modifies database entities. The GUI manager coordinates various windows

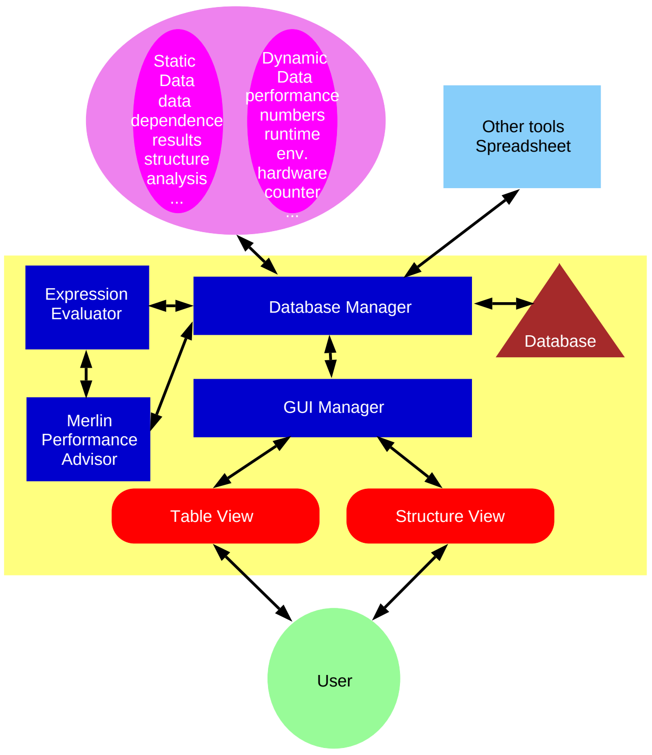


Figure 4: Internal organization of the Ursa Minor Tool.

and views and handles user actions. It also takes care of data consistency between the database and the display windows. The Expression Evaluator is the facility that allows users to perform various operations on the current database. This module parses the user-entered commands, applies the operations, and updates the views accordingly. Finally, Merlin is the guidance system capable of automatically conducting performance analysis and presenting suggestions. The Ursa Minor tool is written in 20,000 lines of Java.

3.2 InterPol: Interactive Tuning Tool

Overview

InterPol is an interactive utility that allows users to apply selected optimization techniques on program or program sections [15]. Users can select target code sections from an entire program source, then either run a custom-built optimizer or make manual changes. It allows users to build their own compiler from numerous optimization modules available from the Polaris parallelizing compiler infrastructure. During program optimization, InterPol keeps track of the program sections being modified, relieving programmers of file and version management tasks. In this way, programmers are free to apply selected techniques on specific regions, change code manually, and generate a working version of the entire program without exiting the tool. During the optimization process, the tool can display static analysis information generated by the underlying compiler, which can help users in further optimizing the program. For those who are not familiar with the techniques available from parallelizing compilers, the tool provides greater insights into the effect of code transformations.

Figure 5 illustrates the major components of InterPol. Users select code regions using the Program Builder and orchestrate optimization techniques through the Compiler Builder. The Compilation Engine takes inputs from these builders, executes the selected compiler modules, and displays the output program. If the user wants to keep the newly modified code segments, the output will go into the Program Builder, replacing the old segments. Instead of running the Compilation Engine, users may choose to add changes to the code manually. All of these actions are facilitated by a graphical user interface. Users are able to store the current program variant at any point in this scenario.

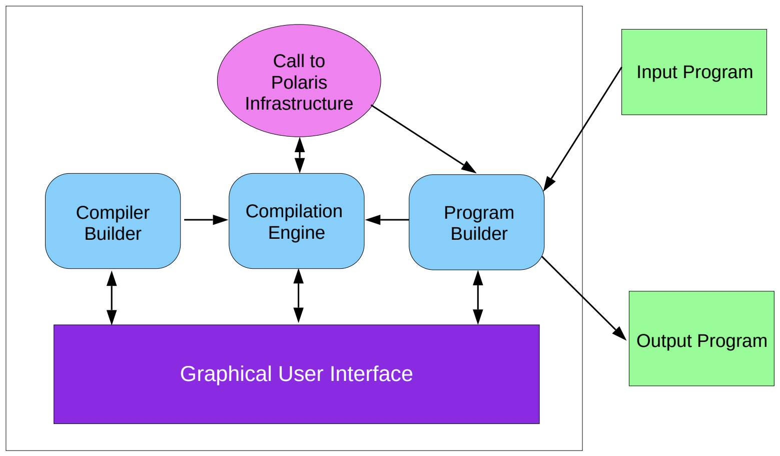


Figure 5: Internal organization of InterPol. Three main modules interact with users through a Graphical User Interface. The Program Builder handles file IO and keeps track of the current program variant. The compiler Builder allows users to arrange optimization modules in Polaris. The Compilation Engine combines the user selections from the other two modules and calls Polaris modules.

Tool Functionality

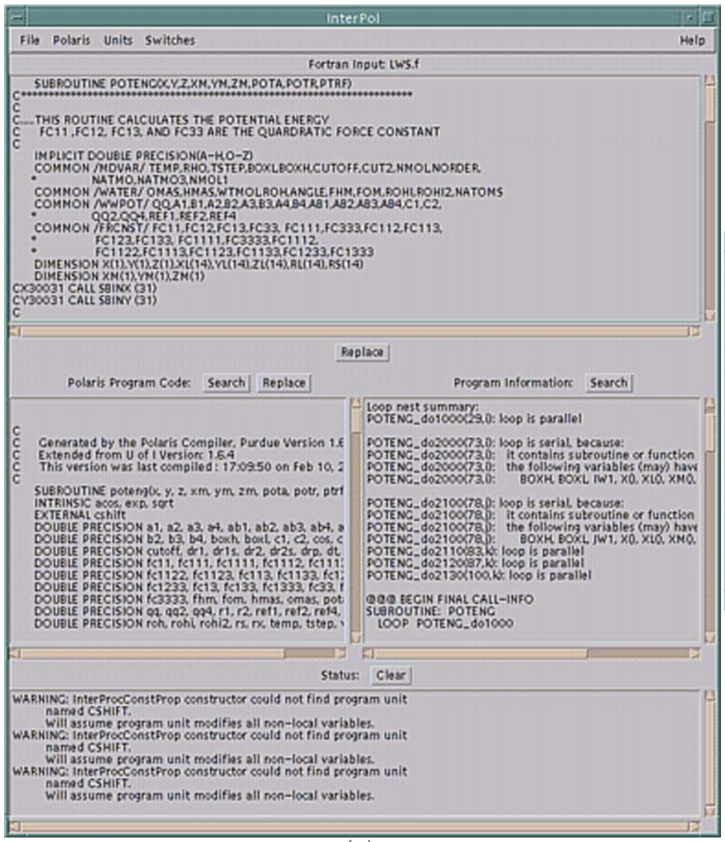
Figure 6(a) shows the graphical user interface of InterPol. Target code segments and the corresponding transformed versions are displayed in separate areas. The static analysis information is given in another area whenever a user activates the compiler. Finally, the Program Builder interface provides an instant view of the current version of the target program.

InterPol is written in Java. The underlying parallelization and optimization tool is the Polaris compiler infrastructure [4]. Various Polaris modules form building blocks for a custom-designed parallelizing compiler. InterPol is capable of combining these modules in any order. Overall, more than 25 modules are available for this purpose. For example, Polaris includes several different data dependence test modules, which can be arranged by InterPol, allowing the user to compare and evaluate these tests. Executing this custom-built compiler is as simple as clicking a menu and the result is displayed directly on the graphical user interface. Figure 6(b) shows the Compiler Builder interface in InterPol. More detailed configuration is also possible through InterPol's Polaris switch interface, which controls the behavior of the individual passes.

The Program Builder keeps and displays the up-to-date version of the whole program. Users select program segments from this module, apply automatic optimization set up by the Compiler Builder and/or add manual changes. The Compiler Builder is accessible at any point, so users can apply entirely different sets of techniques to different regions. The current version of the program is always shown in the Program Builder text area. In this way, InterPol allows for a highly interactive and incremental process of modifying and tuning a parallel program.

During the optimization process, InterPol can display program analysis results generated by running Polaris modules. This includes data dependency test results, induction and reduction variables and array access patterns. InterPol provides the environment for combining this information with the programmer's knowledge of the underlying algorithms, the program's dynamic behavior, and the input data.

Figure 7.a demonstrates the functionality of InterPol through a small example program. Figure 7.b shows the code after being simply run through the default Polaris configuration with the inlining switch set to inline subroutines of 10 statements or less. Two important results can be seen: (1) subroutine one is not inlined due to the inlining pass executing prior to deadcode elimination, and (2) the loops in subroutine two are not found to be parallel because of subscripted array subscripts, which the Polaris compiler cannot analyze.



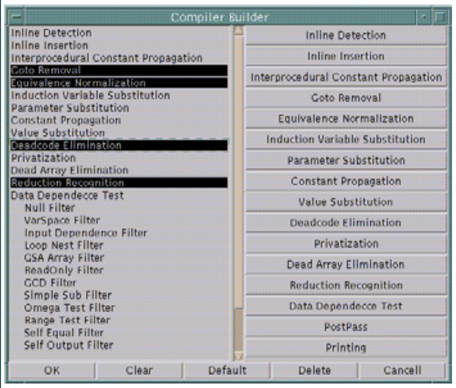
```
      SUBROUTINE POTENG(X,Y,Z,XM,YM,ZM,POTA,POTR,PTRF)
C*******************************************************************
C
C
C....THIS ROUTINE CALCULATES THE POTENTIAL ENERGY
C     FC11 ,FC12, FC13, AND FC33 ARE THE QUARDRATIC FORCE CONSTANT
C
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
      COMMON /MDVAR/ TEMP,RHO,TSTEP,BOXL,BOXH,CUTOFF,CUT2,NMOL,NORDER,
     *              NATMO,NATMO3,NMOL1
      COMMON /WATER/ OMAS,HMAS,WTMOL,ROH,ANGLE,FHM,FOM,ROHI,ROHI2,NATOMS
      COMMON /WWPOT/ QQ,A1,B1,A2,B2,A3,B3,A4,B4,AB1,AB2,AB3,AB4,C1,C2,
     *              QQ2,QQ4,REF1,REF2,REF4
      COMMON /FRCNST/ FC11,FC12,FC13,FC33, FC111,FC333,FC112,FC113,
     *               FC123,FC133, FC1111,FC3333,FC1112,
     *               FC1122,FC1113,FC1123,FC1133,FC1233,FC1333
      DIMENSION X(1),Y(1),Z(1),XL(14),YL(14),ZL(14),RL(14),RS(14)
      DIMENSION XM(1),YM(1),ZM(1)
CX30031 CALL SBINX (31)
CY30031 CALL SBINY (31)
C
```

```
C
C    Generated by the Polaris Compiler, Purdue Version 1.6
C    Extended from U of I Version: 1.6.4
C    This version was last compiled : 17:09:50 on Feb 10, 2
C
      SUBROUTINE poteng(x, y, z, xm, ym, zm, pota, potr, ptrf
      INTRINSIC acos, exp, sqrt
      EXTERNAL cshift
      DOUBLE PRECISION a1, a2, a3, a4, ab1, ab2, ab3, ab4, a
      DOUBLE PRECISION b2, b3, b4, boxh, boxl, c1, c2, cos, c
      DOUBLE PRECISION cutoff, dr1, dr1s, dr2, dr2s, drp, dt,
      DOUBLE PRECISION fc11, fc111, fc1111, fc1112, fc111
      DOUBLE PRECISION fc1122, fc1123, fc113, fc1133, fc1
      DOUBLE PRECISION fc1233, fc13, fc133, fc1333, fc33, f
      DOUBLE PRECISION fc3333, fhm, fom, hmas, omas, pot
      DOUBLE PRECISION qq, qq2, qq4, r1, r2, ref1, ref2, ref4,
      DOUBLE PRECISION roh, rohl, rohi2, rs, rx, temp, tstep,
```

```
Loop nest summary:
POTENG_do1000(29,0): loop is parallel

POTENG_do2000(73,0): loop is serial, because:
POTENG_do2000(73,0):   it contains subroutine or function
POTENG_do2000(73,0):   the following variables (may) have
POTENG_do2000(73,0):       BOXH, BOXL, IW1, X0, XL0, XM0,

POTENG_do2100(78,0): loop is serial, because:
POTENG_do2100(78,0):   it contains subroutine or function
POTENG_do2100(78,0):   the following variables (may) have
POTENG_do2100(78,0):       BOXH, BOXL, JW1, X0, XL0, XM0,
POTENG_do2110(83,k): loop is parallel
POTENG_do2120(87,k): loop is parallel
POTENG_do2130(100,k): loop is parallel

@@@ BEGIN FINAL CALL-INFO
SUBROUTINE:  POTENG
  LOOP  POTENG_do1000
```

```
WARNING: InterProcConstProp constructor could not find program unit
        named CSHIFT.
        Will assume program unit modifies all non-local variables.
WARNING: InterProcConstProp constructor could not find program unit
        named CSHIFT.
        Will assume program unit modifies all non-local variables.
WARNING: InterProcConstProp constructor could not find program unit
        named CSHIFT.
        Will assume program unit modifies all non-local variables.
```

(a)



(b)

Figure 6: User Interface of InterPol: (a) the main window and (b) the Compiler Builder.

Figure 8.a shows the resulting program after adding a deadcode pass prior to the inlining pass in the Compiler Builder, and running the main program and subroutine one from Figure 7.a through this "new" compiler. Finally, in Figure 8.b, subroutine two has been parallelized by hand, and included into the Program Builder. Through simple interactions with InterPol, a code for which Polaris was only able to parallelize a single innermost loop, has both its outermost loops parallelized.

3.3 Tool Support in Each Step

Our tools have been designed and modified based on the parallel programming methodology introduced in Section 2. Figure 1 gives the overview of how these tools can be of use in each step of the methodology. The functionality of Ursa Minor and InterPol, combined with the Polaris instrumentation module, cover all the aspects of the proposed methodology. Getting execution time by running instrumented applications only requires simple UNIX commands, thus no special-purpose tools are needed. Ursa Minor mainly contributes to the performance evaluation stages. InterPol and Polaris offer aid in the parallelization and manual tuning stages.

4 Evaluation

We evaluate our environment as follows. In Section 4.1, we measure the time consumed by a number of parallel programming tasks accomplished with and without our tools. In Section 4.2 and 4.3, we present case studies that demonstrate the use of our methodology and toolset. These studies provide details of many parallelization and tuning steps and the tool functionality.

```
PROGRAM EXAMPLE
REAL A(100,100),B(100,100)
REAL C(100)
INTEGER I
DO I = 1, 100
 CALL ONE(A,B,I)
 C(I) = I
ENDDO
CALL TWO(A,B,C)
WRITE (6,*) A
WRITE (6,*) B
END

SUBROUTINE ONE(A,B,I)
REAL A(100,100),B(100,100)
INTEGER DEADCODE
DEADCODE = 1
DEADCODE = 2
DEADCODE = 3
DEADCODE = 4
DEADCODE = 5
DO J = 1,100
 A(J,I) = 0
 B(J,I) = 0
ENDDO
END

SUBROUTINE TWO(A,B,C)
REAL A(100,100), B(100,100)
REAL C(100)
DO I = 1, 100
 DO J = 1, 100
  A(C(J),C(I)) = I+J
  B(C(J),C(I)) = I*J
 ENDDO
ENDDO
END
```

(a)

```
PROGRAM EXAMPLE
REAL A(100,100),B(100,100)
REAL C(100)
INTEGER I
DO I = 1, 100
 CALL ONE(A,B,I)
 C(I) = I
ENDDO
CALL TWO(A,B,C)
WRITE (6,*) A
WRITE (6,*) B
END

SUBROUTINE ONE(A,B,I)
REAL A(100,100),B(100,100)
!$OMP PARALLEL DO
DO J = 1,100
 A(J,I) = 0
 B(J,I) = 0
ENDDO
!$OMP END PARALLEL DO
END

SUBROUTINE TWO(A,B,C)
REAL A(100,100), B(100,100)
REAL C(100)
DO I = 1, 100
 DO J = 1, 100
  A(C(J),C(I)) = I+J
  B(C(J),C(I)) = I*J
 ENDDO
ENDDO
END
```

(b)

Figure 7: Contents of the Program Builder during an example usage of the InterPol tool: (a) the input program, (b) the output from the default Polaris compiler configuration.

4.1 Efficiency of Tools for Common Tasks in Parallel Programming

The main objectives of this experiment is to produce quantitative measures for the efficiency of Ursa Minor. To this end, we have selected 10 tasks that are commonly performed by parallel programmers using parallel directive languages. These tasks are listed in Table 1.

task01	Compute the speedup of the given program on 4 processors in terms of the serial execution time.
task02	Find the most time-consuming loop based on the serial execution time.
task03	Find the inner and outer loops of that loop.
task04	Find the caller(s) of the subroutine containing the most time-consuming loop.
task05	Compute the parallelization and spreading overhead of that loop on 4 processors.
task06	Compute the parallel efficiency of the second most time-consuming loop on 4 processors.
task07	Export profiles to a spreadsheet to create total execution time chart (on varying number of processors) containing 5 of the most time-consuming loops.
task08	Count the loops the speedups of which are below 1.
task09	Count the loops that are parallel and whose speedups are below 1.
task10	Compute the parallel coverage and the expected speedup based on Amdahl's Law.

Table 1: Common tasks in parallel programming, used to measure our tool performance

Task 1 is a simple calculation; users may use either a calculator or the Expression Evaluator from Ursa Minor with comparable efficiency. Task 2 evaluates the table manipulation utilities (sorting and rearranging) for the performance data. Tasks 3 and 4 measure the efficiency of the Structure View and the utilities that it provides. The Expression Evaluator is the main target for evaluation in Tasks 5 and 6. Task 7 tests the ability to rearrange the tabular data and export it to other spreadsheet applications. Tasks 8, 9, and 10 evaluate the combined usage of multiple utilities (sorting, the Expression Evaluator, query functions, the

```
PROGRAM EXAMPLE
REAL A(100,100),B(100,100)
REAL C(100)
INTEGER I
!$OMP PARALLEL DO
DO I = 1, 100
 DO J = 1,100
  A(J,I) = 0
  B(J,I) = 0
 ENDDO
 C(I) = I
ENDDO
!$OMP END PARALLEL DO
CALL TWO(A,B,C)
WRITE (6,*) A
WRITE (6,*) B
END

SUBROUTINE TWO(A,B,C)
REAL A(100,100), B(100,100)
REAL C(100)
DO I = 1, 100
 DO J = 1, 100
  A(C(J),C(I)) = I+J
  B(C(J),C(I)) = I*J
 ENDDO
ENDDO
END
```

(a)

```
PROGRAM EXAMPLE
REAL A(100,100),B(100,100)
REAL C(100)
INTEGER I
!$OMP PARALLEL DO
DO I = 1, 100
 DO J = 1,100
  A(J,I) = 0
  B(J,I) = 0
 ENDDO
 C(I) = I
ENDDO
!$OMP END PARALLEL DO
CALL TWO(A,B,C)
WRITE (6,*) A
WRITE (6,*) B
END

SUBROUTINE TWO(A,B,C)
REAL A(100,100), B(100,100)
REAL C(100)
!$OMP PARALLEL DO
DO I = 1, 100
 DO J = 1, 100
  A(C(J),C(I)) = I+J
  B(C(J),C(I)) = I*J
 ENDDO
ENDDO
!$OMP END PARALLEL DO
END
```

(b)

Figure 8: Contents of the Program Builder after user interaction with the InterPol tool: (a) the output after placing an additional deadcode elimination pass prior to inlining and (b) the program after manually parallelizing subroutine two.

static information viewer, and the display option control) provided by Ursa Minor.

Experiment

We have asked four users to participate in this experiment. They were asked to perform the tasks shown in Table 1 one by one. We chose two different sets of performance data for the experiment. These datasets contain timing profiles of FLO52Q from the Perfect benchmarks [3] under two different environments. Thus, the number of data items are the same in both datasets, but the timing numbers are different. First, the participants were asked to perform the tasks without our new tools. They were allowed to use any existing tools and scripts. Then, they performed the tasks using our new tools with the other dataset.

The time to invoke the tools and load input files was counted separately as *loading time*. Time to convert data files for different tools are also included in the loading time. The loading time reflects the level of integration of tools.

The four users represent different classes of programmers. User1 is an expert performance analyst who has written many special-purpose scripts to perform various jobs, such as tabularizing and sorting. User1 does use our tools but relies more on these scripts. User2 has also been working on performance evaluation for a while and is considered an expert as well. He uses only basic UNIX commands, rather than scripts. However, his skills with the basic UNIX commands are very good, so he can perform a complex task without taking much time. User2 started using our tools only recently. User3 is also an expert performance analyst, but his main target programs are not shared memory programs. He has been using our tools for a long time, but with distributed memory programs. Finally, user4 is a novice parallel programmer. His experience with parallel programs are limited compared to the others. He had to read our methodology and tried to use our tools in benchmarking research.

Table 2 shows the time for these users to perform the assigned tasks. User2, 3, and 4 decided that tasks 9 and 10 cannot be performed within reasonable time, so they gave estimated times instead. All of the users used a commercial spreadsheet later in the session, but user4, the novice programmer started doing the tasks after he set up the spreadsheet and imported the input files. User1 used his scripts for many of the tasks.

	user1	user2	user3	user4	average
task01	21	41	73	61	49
task02	14	9	3	43	17.25
task03	22	29	97	77	56.25
task04	28	6	46	48	32
task05	75	44	217	132	117
task06	25	43	27	44	34.75
task07	94	400	197	275	241.5
task08	67	208	211	594	270
task09	258	280	420	600	389.5
task10	220	400	540	600	440
loading	60	79	128	864	282.75
total	884	1,539	1,959	3,338	1,930

Table 2: Time (in seconds) taken to perform the tasks without our tools.

	user1	user2	user3	user4	average
task01	7	18	23	19	16.75
task02	3	2	16	14	8.75
task03	6	4	3	9	5.5
task04	4	6	18	3	7.75
task05	52	61	85	98	74
task06	14	9	18	19	15
task07	47	104	202	100	113.25
task08	81	39	54	72	61.5
task09	72	33	46	49	50
task10	163	400	138	600	325.25
loading	57	73	99	90	79.75
total	506	749	702	1,073	757.5

Table 3: Time (in seconds) taken to perform the tasks with our tools.

In the second part of the experiment, all users were allowed to use our tools to perform the same tasks. The results are shown in Table 3. User1 used a combination of a spreadsheet and Ursa Minor to perform tasks 8, 9, and 10. The others used a spreadsheet for task 7 only. User4 was not sure that he can finish task 10 even with our tool support, so he gave an estimated time.

These tables show that our tool support improves the time to perform common parallel programming tasks considerably. Figure 9 shows the overall times to finish all the tasks. They indicate that our tool support not only saves time, but also makes the process easier for novice programmers, resulting in similar times for all users to perform the tasks when using our tools. The work speedups for the four users are 1.75, 2.05, 2.79, and 3.11, respectively.

The strength of the presented approach lies not only in the fact that the tools offer efficient ways of performing individual tasks, but also in the integration of these features in a common environment. This is demonstrated by the savings in the loding time in our experiment. Users do not have to deal with several tools and commands. There is no need to open the same file into many different tools. For instance, users can open the Structure View to inspect the program layout and examine and restructure the performance data from the same database.

4.2 Case Study: Manual Tuning of ARC2D

In this section, we present a case study illustrating the manual tuning process of program ARC2D from the Perfect benchmark suite [3]. In this study, a programmer has tried to improve the performance of the

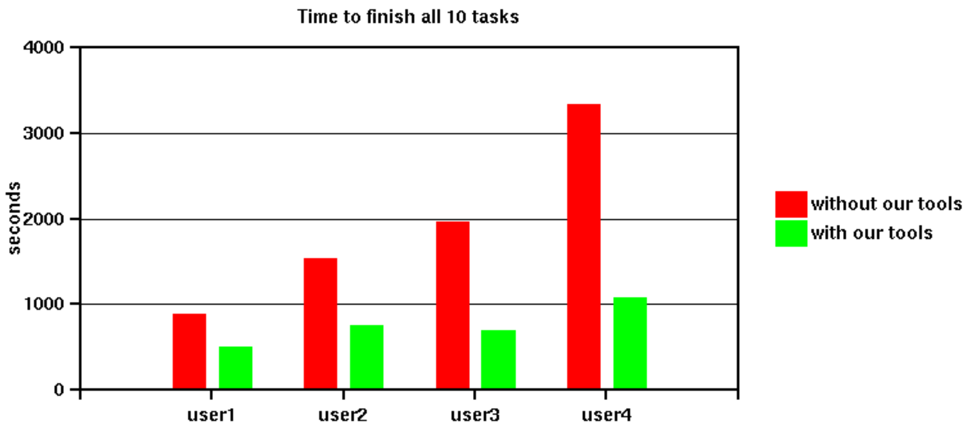


Figure 9: Overall times to finish all 10 parallel programming tasks.

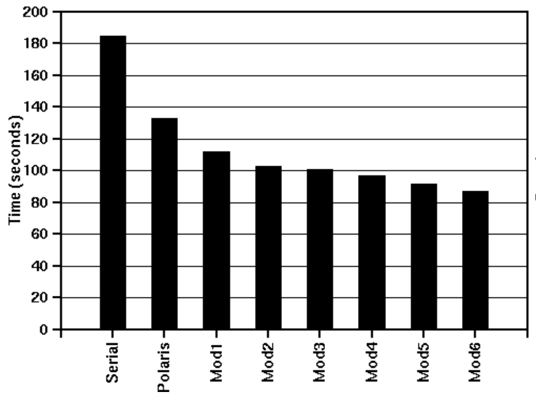
program beyond that achieved by the Polaris parallelizing compiler. The target machine is a HyperSPARC workstation with 4 processors.

Polaris was able to parallelize almost all loops in ARC2D. However, the speedup of the resulting executable was only 1.38 on 4 processors. Using Ursa Minor's Structure View and sorting utility, the programmer was able to find three loops to which loop interchange can be applied: `FILERX_do19`, `XPENTA_do3`, and `XPENT2_do3`. After loop nests were interchanged in these loops, the total program execution time decreased by 22 seconds, increasing the speedup from 1.39 to 1.65.

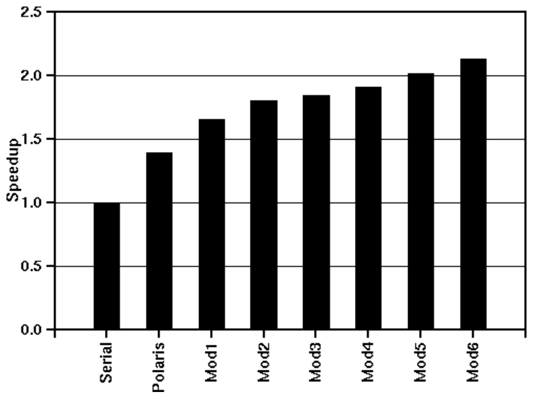
As the result of this modification, dominant program sections have changed. The programmer re-evaluated the most time-consuming loops using the Expression Evaluator to compute new speedups and the percentage of loop execution time over the total time. The most time-consuming loop was now the `STEPFY_do420` nest, which consumed 27% of the new parallel execution time. The programmer examined the nest with the source viewer and noticed two things: (1) there were many adjacent parallel regions and (2) the parallel loops were not always distributing the same dimension of the work array. The programmer merged all of the adjacent parallel regions in the nest into a single parallel region. The new parallel region consisted of four consecutive parallel loops. The first two loops distributed the work array across its innermost (stride-1) dimension. The second two nests were doubly nested and distributed the work array across its second innermost dimension. The effect of these changes were two-fold. First, the merging of regions should eliminate parallel loop fork/join overhead. Second, the normalization of the distributions within the subroutine should improve locality. After this change, the speedup of the loop improved from 1.19 to 1.50.

The programmer was able to apply the same techniques (fusion and normalization) to the next 3 most time-consuming loops (`STEPFX_do300`, `FILERX_do15`, and `YPENTA_do1`). These modifications result in a speedup gain from 1.50 to 2.02. Finally, the programmer applied the same techniques to the next most time-consuming sections `XPENTA`, `YPENT2`, and `XPENT2` according to the newly computed profiles and speedups. This modification improved the speedup to 2.12.

In summary, applying loop interchange, parallel region merging and distribution normalization, yielded an increase from the out-of-the-box speedup of 1.38 to a speedup of 2.12. This corresponds to a 35% decrease in execution time. Figure 10 shows the improvements in the total program performance as each program optimization was applied. Ursa Minor allowed the user to quickly identify the loop structure of the program and sort the loops to identify the most time-consuming code sections. After each modification, the user was able to add the new timing data from the modified program runs, re-calculate the speedup and see if an improvement was worthwhile. In this case study, the user has followed the methodology to improve the performance significantly. The tool features that proved most important were data arrangement, the Structure View, the source viewer, and the Expression Evaluator.



(a)



(b)

Figure 10: The (a) execution time and (b) speedup of the various version of ARC2D (Mod1: loop interchange, Mod2: `STEPFY_do420` modification, Mod3: `STEPFX_do300` modification, Mod4: `FILERX_do15` modification, Mod5: `YPENTA_do1` modification, Mod6: modification on `XPENTA`, `YPENT2`, and `XPENT2`).

Techniques	Criteria
Serialization	speedup < 1
Loop Interchange	# of stride-1 accesses < # of non stride-1 accesses
Loop Fusion	speedup < 2.5

Table 4: Optimization techniques and application criteria.

4.3 Case Study: Using the Performance Advisor

In this section, we present a simple performance map for the Merlin performance advisor, based solely on execution timings and static compiler information. Such data can easily be obtained by a novice user from a compiler listing and timing profiles. The map used in this experiment is designed to advise programmers in improving the performance of programs optimized by a parallelizing compiler such as Polaris [4]. In this case study, we assume that a parallelizing compiler was used as the first step in optimizing the performance of the target program and that the compiler's program analysis information is available. The performance map aims at increasing this initial performance.

Based on our experiences with parallel programs, we have chosen three techniques that are easy to apply and may yield considerable performance gain. These techniques are serialization, loop interchange, and loop fusion. All of these techniques are present in modern compilers. However, compilers may not have enough knowledge to apply them most profitably [14], and some code sections may need small modifications before the techniques become applicable automatically.

Performance Map Description

The performance map used in this experiment includes criteria for the application of the techniques shown in Table 4. If the speedup of a parallel loop is less than 1, we assume that the loop is too small for parallelization or that it required extensive modification. Serializing it prevents performance degradation. Loop interchange may be used to improve locality by increasing the number of stride-1 accesses in a loop nest. Loop interchange is commonly applied by optimizers; however, our case study shows many opportunities missed by the backend compiler. Loop fusion can likewise be used to increase both granularity and locality. The criteria shown in Table 4 represent simple heuristics and do not attempt to be an exact analysis of the benefits of each technique. We simply assumed the threshold of the speedup as 2.5 to apply loop fusion. In all cases the user will have to measure the benefit of the suggested optimization.

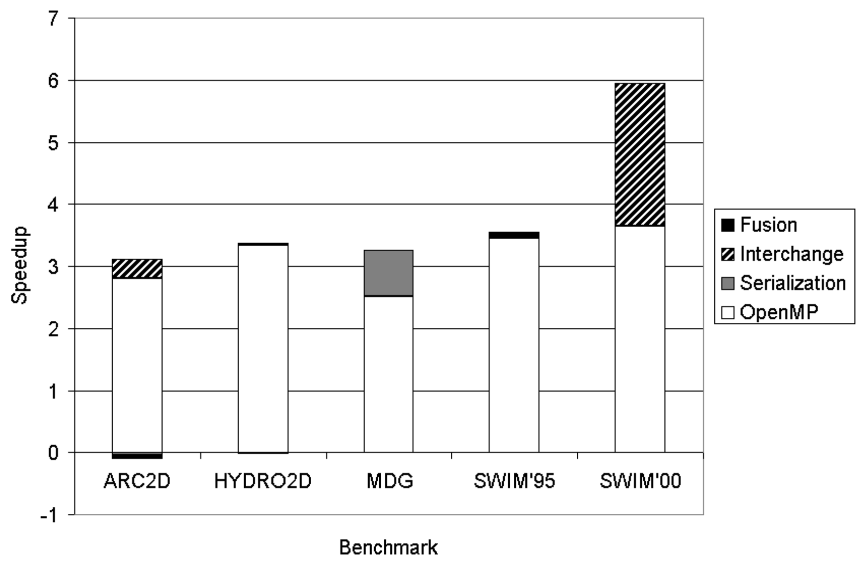


Figure 11: Performance improvements achieved by applying the performance map. The speedup is with respect to the serial code on a Sun Enterprise 4000 system. Each graph shows the cumulative speedup when applying techniques.

Experiment

We have applied these techniques based on the criteria in Table 4. We have used a Sun Enterprise 4000 with six 248MHz UltraSPARC processors. The OpenMP code is generated by the Polaris OpenMP backend. The results on five programs are shown in Figure 11. They are `SWIM` and `HYDRO2D` from the SPEC95 benchmark suite, `SWIM` from the SPEC2000 suite, and `ARC2D` and `MDG` from the Perfect Benchmarks. We have incrementally applied these techniques starting with serialization. Figure 11 shows the resulting improvement. The decrease in execution time ranges from -1.8% for fusion in `ARC2D` to 38.7% for loop interchange in `SWIM'00`. For `HYDRO2D`, the application of the MERLIN suggestions did not noticeably improve performance.

Among the codes with large improvements, `SWIM` from SPEC2000 benefits most from loop interchange. It was applied under the suggestion of MERLIN to the most time-consuming loop, `SHALOW DO3500`. Likewise, the main technique that improved the performance in `ARC2D` was loop interchange. `MDG` consists of two large loops and numerous small loops. Serializing these small loops was the sole reason for the performance gain. Table 5 shows a detailed breakdown of how often techniques were applied and their corresponding benefit.

Using this map, considerable speedups are achieved with a relatively small effort. Novice programmers can simply run MERLIN to see the suggestions made by the map. Experienced programmers can flexibly update the map without modifying MERLIN. Thus if new techniques show potential or the criteria need revision, the performance map can easily and incrementally be enhanced.

5 Related Work

Numerous tools exist to help programmers develop well-performing parallel programs. The important role of tools to aid the process of parallel program development and performance tuning is widely acknowledged. Among the supporting tools are those that perform automatic parallelization, performance visualization, instrumentation, and debugging. Many of the current tools are summarized in [5, 6].

Several tools have attempted to integrate different parallel programming tasks. Pablo and the Fortran D editor [1] combine program optimization and performance visualization. The SUIF Explorer [17] and FORGExplorer [2] have a similar goal. The KAP/Pro Toolset [16] consists of tools for automatic parallelization, performance visualization, and debugging. The focus of the Annai Tool Project [23] is on the aspects of parallelization, debugging, and performance monitoring. Faust [10] attempted to create the most

Benchmark	Technique	Number of Modifications	% Improvement
`ARC2D`	Serialization	3	-1.55
	Interchange	14	9.77
	Fusion	10	-1.79
`HYDRO2D`	Serialization	18	-0.65
	Interchange	0	0.00
	Fusion	2	0.97
`MDG`	Serialization	11	22.97
	Interchange	0	0.00
	Fusion	0	0.00
`SWIM'95`	Serialization	1	0.92
	Interchange	0	0.00
	Fusion	3	2.03
`SWIM'00`	Serialization	0	0.00
	Interchange	1	38.69
	Fusion	1	0.03

Table 5: A detailed breakdown of the performance improvement due to each technique suggested by the Merlin performance advisor.

comprehensive environment, encompassing code optimization and performance evaluation. DEEP/MPI [19] augments a performance evaluation utility with a procedure-level performance advisor. Both WPP/Aivi [22] and CAPO [18] provide a parallelizing compiler and a graphical tool to visualize static program analysis information.

	performance data visualization	program structure visualization	display compiler analysis	automatic parallelization	interactive compilation	support performance modeling	automatic analysis/guidance	debugging
Pablo/Fortran D Editor	√		√	√	√			
SUIF Explorer	√		√	√	√		√	
FORGExplorer			√	√	√			
KAP/Pro Toolset	√			√				√
Annai Project	√							√
DEEP/MPI	√	√					√	
Faust	√	√	√	√	√			
Ursa Minor/InterPol	√	√	√	√	√	√	√	

Table 6: Feature comparison of parallel programming environments

Table 6 shows the availability of features in these environments. The parallelization utility available from the Pablo/Fortran D Editor is actually semi-automatic. The guidance system (Parallelization Guru) of the SUIF Explorer points to dominant and possibly problematic code sections. DEEP/MPI's advisor is limited to fixed, procedure-level analysis. The table shows that, except for debugging, our environment provides the most comprehensive support. Ursa Minor's support for performance modeling is a feature not provided by any other tool. It includes capabilities for querying, filtering and abstracting performance and program analysis data. This allows users to reason about the performance of a program in a flexible manner. The configurable, loop-level performance guidance provided by Merlin is a unique feature of our environment as well. InterPol allows users to “build” their own parallelizing compiler, a feature also not available in other tools. Overall, the Ursa Minor/InterPol toolset offers the most versatile and flexible features. Furthermore, in contrast to most other environments, our tools exists in Web-accessible forms. Any user with a standard Web browser can make use of this system, including complete on-line documentation and tutorials.

6 Conclusion

Our effort to create a parallel programming environment has resulted in a parallel program development and tuning methodology and a set of supporting tools. We have developed the tools with the goal to provide an integrated, flexible, accessible, portable and configurable tool environment that supports the underlying methodology. Our toolset integrates static program analysis with performance evaluation, while supporting data visualization and interactive compilation. Performance data management is also simplified with our tools.

We have evaluated our environment both quantitatively and qualitatively through case studies and an experiment to measure tool efficiency, We have found that it provides effective support for developing well-performing parallel programs.

The clear focus on providing tools that support an underlying programming methodology is one of the most distinguishing aspects of the presented work. In doing so we have addressed one of the grand challenges in software engineering in general, and parallel program optimization in particular. The challenge is that programming is not a systematic discipline. There are no textbooks that teach a programmer the concrete steps that must be taken to create a well-performing piece of software. Because of this, software design in difficult, time-consuming, expensive, and requires experienced programmers. We feel that there will probably always be parts of software design that take intuitive skills and are thus hard to learn systematically. However, there are also many steps that are repetitive and can be followed in a methodological manner. Defining these steps clearly and providing supporting tools will not only help the less experienced programmers, it will also put the programming discipline on a more solid scientific basis. The presented paper is a contribution to this end.

References

[1] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J. C. Wang, and D. A. Reed. An integrated compilation and performance analysis environment for data parallel programs. In *Proc. of Supercomputing Conference*, pages 1370–1404, 1995.

[2] Applied Parallel Research Inc. *Forge Explorer*, 2000. http://www.apri.com.

[3] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):9–40, Fall 1989.

[4] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[5] J. Brown, A. Geist, C. Pancake, and D. Rover. Software tools for developing parallel applications. 1. code development and debugging. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[6] J. Brown, A. Geist, C. Pancake, and D. Rover. Software tools for developing parallel applications. 2. interactive control and performance tuning. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[7] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computing in Science and Engineering*, 5(1):23–30, January 1999.

[8] Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Transactions of Parallel and Distributed Systems*, pages 5–23, January 1998.

[9] Rudolf Eigenmann, Insung Park, and Michael J. Voss. Are parallel workstations the right target for parallelizing compilers? In *Lecture Notes in Computer Science, No. 1239: Languages and Compilers for Parallel Computing*, pages 300–314, March 1997.

[10] Vincent Guarna Jr., Dennis Gannon, David Jablonowski, Allen Malony, and Yogesh Gaur. Faust: An integrated environment for the development of parallel programs. *IEEE Software*, 6(4):20–27, July 1989.

[11] Nirav H. Kapadia and José A.B. Fortes. On the design of a demand-based network-computing system: The Purdue university network computing hubs. In *Proc. of IEEE Symposium on High Performance Distributed Computing*, pages 71–80, Chicago, IL, 1998.

[12] 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, Manual ECE-HPCLab-97201, 1997.

[13] Seon Wook Kim, Insung Park, and Rudolf Eigenmann. A performance advisor tool for novice programmers in parallel programming. In *Proc. of the 13th annual Workshop on Languages and Compilers for Parallel Computing*, August 2000.

[14] Seon Wook Kim, Michael J. Voss, and Rudolf Eigenmann. Performance analysis of parallel compiler backends on shared-memory multiprocessors. In *Proc. of the Tenth Workshop on Compilers for Parallel Computers*, pages 305–320, January 2000.

[15] Stefan Kortmann, Insung Park, Michael Voss, and Rudolf Eigenmann. Interactive and modular optimization with INTERPOL. In *Proc. of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1963–1967, June 2000.

[16] Kuck and Associates Inc. *KAP/Pro Toolset*, 2000. http://www.kai.com.

[17] W. Liao, A. Diwan, R. P. Bosch Jr., A. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Proc. of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, August 1999.

[18] NAS Systems Division. *CAPO (CAPTools-based Automatic Parallelizer using OpenMP)*, 2001. http://www.nas.nasa.gov/Groups/Tools/CAPO/.

[19] Pacific-Sierra Research. *DEEP/MPI: Development Environment for MPI Programs Parallel Program Analysis and Debugging*, 2000. http://www.psrv.com/deep_mpi_top.html.

[20] Paramount Research Group, Purdue University. *Program Parallelization and Tuning Methodology*, 2000. http://peak.ecn.purdue.edu/ParaMount/UMinor/meth_index.html.

[21] Insung Park, Michael J. Voss, Brian Armstrong, and Rudolf Eigenmann. Supporting users' reasoning in performance evaluation and tuning of parallel applications. In *Proc. of the Twelth IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2000.

[22] Makoto Satoh, Yuichiro Aoki, Kiyomi Wada, Takayoshi Iitsuka, and Sumio Kikuchi. Interprocedural parallelizing compiler WPP and analysis information visualization tool Aivi. In *Proc. of EWOMP'2000: European Workshop on OpenMP*, Edinburgh, Scotland, U.K., September 2000.

[23] B. J. N. Wylie and A. Endo. Annai/PMA multi-level hierarchical parallel program performance engineering. In *Proc. of International Workshop on High-Level Programming Models and Supportive Environments*, pages 58–67, 1996.