Practical Tools for Optimizing Parallel Programs

Rudolf Eigenmann **Patrick McClaughry**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign*

Abstract

This paper describes a set of tools that help a programmer to be more efficient in optimizing scientific programs for a parallel computer. The design of these tools emerged from experience gained during a successful optimization effort on a set of representative supercomputer application codes. We have developed a number of utilities that complement available Unix tools. Additional tools offer a higher degree of interactivity; they are currently built into the Emacs editor. The new tools mainly facilitate two development phases that were identified as time-consuming in the parallel programming project: The process of creating and maintaining a consistent set of experimental program variants and the analysis and interpretation of compilation and performance results.

1 Introduction

1.1 Two related motivations

The initiative to invest in a tool design effort came from the need for tools in our project. The goal of the project was to optimize a large set of real applications for the Alliant FX/8 and the Cedar machine, including the Perfect Benchmarks® programs [3]. Initially, our tools consisted of the parallelizing compiler Kap/Cedar[2] and ordinary Unix utilities for manually improving the parallel Fortran code generated by Kap.

Additional motivation came from the many open issues in tool design. For example, there is a big discrepancy between the efforts being invested in design projects for new tools and the resulting benefits for the user community. This concerned us directly, because, despite many announced products, we have not found tools that could help us reach our objective more quickly. Perhaps one reason for this is the lack of measures to assess tools and programming environments. Some concrete issues resulted from our program optimization project. An overview of the program optimization cycle and the applied methodology is described in [1]. Our major findings are, that there is a need for facilities for two development phases that were identified as time-consuming: The process of creating a consistent set of experimental program variants and the analysis and interpretation of compilation and performance results.

1.2 Design principles

Keep it simple The major principle we intended to observe was to keep new tools simple. We intended not only to keep the time invested in this project to a minimum; we also believe that large tools may become too complex to use. Certainly, if we managed to design simple tools that provided the power needed, we would feel very successful. There is precedent that this objective can be achieved: users of elaborate programming environments quite often report that only a few features can be credited for the success of their mission.

Users are advanced parallel programmers The users for which our tools were designed are professionals with a solid understanding of UNIX and Fortran. Our tool set does not attempt to be a parallelizing tutor nor does it attempt to fully automate the task. The user is assumed to be comfortable with the notions of parallel architectures and the types of transformations useful on programs for such architectures.

This work complements other projects of our research group that develop parallelizing compilers [2, 3] capable of optimizing sequential programs automatically for parallel machines. Eventually these two projects will merge in order to satisfy a spectrum of users, from application programmers that are unable to invest the time in optimizing parallel programs to expert programmers who want to exploit all features of the parallel machines.

Stay at Unix level and provide interactivity where needed By building on top of the Unix operating system we hook on to the most widely used platform for parallel programming. Functionality available at the Unix level should not be reinvented and not be impeded.

Sometimes a more interactive interface is needed which can permit easier "what-if" analysis and can offer help or guidance for commands and options either not yet fully learned, or for which the user does not wish to remember the syntax.

* This work was supported in part by the U.S. Department of Energy under grant no. DOE DE-FG02-85ER25001.

An important component of interactive interfaces is a user interface management system. The requirement to provide state-of-the-art features such as customizable command accelerators[1] and help facilities often conflicts with the available development time. For this reason we chose Emacs as our first interactive interface. Emacs offers a full-featured text editor, file management facilities, interactive help, and a familiar language in which to program (ELisp). Replacing Emacs by an alternative user interface management system that provides similar features is straightforward.

Data base access During the porting and optimizing process a considerable amount of performance data can be generated. These data are captured in a database accessible from within the interactive environment as well as at the command line. The introduction of a database to manage the performance data adds significantly to the power of our tool set; however, that power comes at a cost. In order to exploit the power of the database, a user must become familiar with its interface and structure. This trade-off must be chosen carefully.

2 A Tour through PTOPP

Our basic approach to optimizing programs was straightforward: first, the time-consuming parts of the programs were determined; then these code-sections were improved one by one.

Most often the analysis was done on a loop-by-loop basis. The first step was to get a rank list of time-consuming loops. For each of these loops, potential improvements were then determined by looking at the automatically parallelized code, the compiler listing, and timing profiles. The growing list of transformations that had already proven useful in our experiments was another valuable source of information. Applying transformations and experimenting with them was the next major step; often this resulted in a changed loop profile, yielding a new display of the most time-consuming program sections. This process was reiterated until there were diminishing returns.

The program-optimization process is shown in Figure 1 and described in more detail in [1]. The following sections explain the steps by examples, describe the tools we provided, and discuss the design.

2.1 Program instrumentation

We usually started optimizing programs by identifying the time-intensive loops. For this purpose all loop nests in the serial source code were instrumented. All optimized program variants were derived from the instrumented source code so that loop-by-loop speedups could be computed from the profiles.

1. commands bound to a single keystroke or key sequence

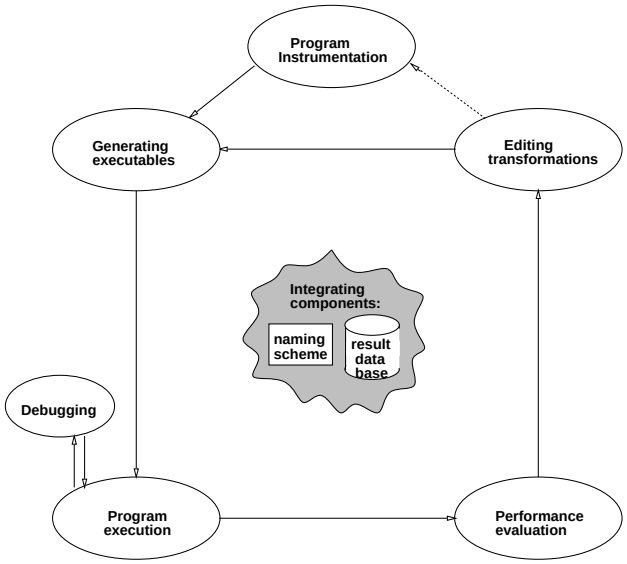

Figure 1: The program optimization cycle

PTOPP provides facilities so that program instrumentation can be done fully implicitly through the `cfmake` command, with the `instrument` command, or within the editor while looking at the source code. The `cfmake` command will be described later. It usually makes the explicit call to the instrumentation utility unnecessary. If the program instrumentation facility is called from within the editor, the resulting instrumented source file is displayed. Loops are embraced with calls to a timer and profile generation library function as shown below.

```
      call start_interval(23)
      DO 100 i=1,n
        ....
100   CONTINUE
      call end_interval(23)
```

Although automatic instrumentation works well, in a later iteration of the program optimization process the user may wish to add or delete some of these calls. The editor facilitates this through Emacs functions that interactively instrument or de-instrument a specified region of text.

Discussion Although the wish for advanced profiling facilities is stated often by software engineers, the features that contribute to more highly optimized programs are quite unclear. In our experience the combination of initial automatic instrumentation and interactive modification facilities was very useful. Loop-level profile information was essential, and because of this, the Unix *gprof* utility was not sufficient for our purposes. Functionality that is not yet available and that we sometimes wished for is combined inclusive and exclusive profiling, the display of the dynamic call tree, and the full record of original trace information.

Some of this functionality is available from a related tool project [6].

2.2 Generation and execution of program variants

The program instrumentation step is followed by the generation and execution of multiple program variants, each corresponding to a set of compiler options plus a set of files that contain individually optimized program sections. Programs are executed on a number of different machines and configurations. The result data provide the initial information for analyzing the program performance. Examples of program variants of interest for this purpose are serial, vectorized, and vector-concurrent optimizations.

PTOPP provides the `cfmake` command at the Unix level, which, in its simplest form, takes the name of an executable file as a parameter. It derives all necessary make steps from the name and generates and starts a Unix make file. The make steps include the automatic instrumentation of the program, if necessary. The `cfmake` command also supports the composition of a file from a number of sources, such as from a set of individually optimized program modules plus "the rest" from another file. The naming conventions, by which the utility derives all make steps, are specified in a separate language designed for this purpose and can be customized flexibly. For example, in our conventions

```
cfmake testC.wi
```

will make an executable code starting with the source file `test.f`, instrumenting it, compiling it using the Cedar optimizer, and linking it with the library that would generate a wallclock time profile.

Similarly, if the user wishes to optimize parts of the file `test.f` individually, he or she can extract and modify these parts into files `a.f` and `b.f`, say, and then issue the command

```
cfmake testC.cflib a.f b.f -o testM1
```

This will compose an executable code named `testM1`, which contains the modules `a.f` and `b.f` instead of the ones that are part of file `test.f`.

The `cfmake` commands supports the creation of a series of related program variants and names them so they can be clearly identified.

For smaller programming experiments PTOPP provides functions within the editor that let the user compile and link the file being edited. A simple Emacs function supports the generation of a Unix-type command line by either editing the last such command issued or by picking from a set of prepared command line templates. It also provides descriptions of available compilation command options while composing the command.

For program execution, object files need to be copied to the target machine, input files must be made available, and after the execution, output files must be given the proper name so they can be associated with the executed code variant and machine configuration. In our experiments this functionality is provided by a Unix script that is written for the specific needs of our situation.

Discussion At the beginning of our project, generating executable files and running them was time-consuming for two reasons. First, the large number of program variants raised a consistency issue. Before we realized supporting tools, we struggled with many errors such as forgetting a compiler option or a variable of the runtime environment, which made "performance anomalies" appear among the program variants. Second, the Unix make utilities were not powerful enough to support quick changes to one subroutine of a given large program. Splitting up files into subroutines was no feasible option because the number of file variants for experimenting with optimizations was already unmanageable enough.

The `cfmake` command was a great help in managing files, their names, and corresponding make commands. We used complementary simple Unix facilities such as a log file of all cfmake commands with time stamp, so that we could check make commands and the date they were issued, and re-make codes at any time with as few keystrokes. These features come at the price of creating and sticking to the naming conventions. The naming conventions are defined in a separate language, SDL (string definition language), the description of which is beyond the scope of this paper. Early experience shows that users may be willing to use predefined naming conventions so that the price of maintaining SDL files is reasonable.

The program execution phase is not yet facilitated by PTOPP. The mentioned script was very important because it provides many services for a single command, including waiting for target machines to be up. It also plays its role in maintaining the naming conventions by renaming output files properly. The inclusion of this functionality into a flexible tool is an increasingly important issue as we extend the range of programs and target machines.

2.3 Debugging

PTOPP does not include debugging tools. In our experiments we used primitive debugging facilities such as undoing most recent program modifications and, in worst case, "binary error search" by swapping modules in and out of an object code until we found the module causing the problem. This was usually feasible for our type of work because our objective was to optimize an existing program, of which we had a working original version.

We also had access to a low-level debugger that let us inspect the machine state at a binary level. It served well for indicating the source code position at runtime traps and inspecting register contents.

Being disciplined is certainly essential for keeping debugging time low. This is obvious when it comes to being careful about typos. It is of further importance

in terms of carefully naming intermediate program variants that are created during debugging sessions. A successful method was to create a new filename with every single modification. This way we could always find the version that "used to work or fail", and since we recorded compilation commands, the options a file was made with could always be determined. In our experiment this method was maintained manually. Future PTOPP version may provide supporting facilities.

2.4 Performance analysis

In order to find program transformations that could improve the performance, the program result data were collected and analyzed for a number of *optimization factors*. These factors provided information about the program performance on a loop-by-loop basis and gave initial hints for optimizing program transformations. Examples of such factors are the loop speedup and the *globalization penalty* (i.e., the performance loss when putting data in global instead of in local memory). The methodology of analyzing these data is described in [1].

PTOPP provides many facilities to support this phase of the development process. The INGRES database is used for managing the large amount of program result data. Currently, it is the user's responsibility to enter these data into the database, usually by filling out part of the following Form:

```
Experiment ID :
Program ID : NA
Variant ID : A.wi
Machine ID : c4s
DataSet Name :
TimeStamp : 28-Mar-92 02:27:06
MFLOPS rate : 2.066728
CPU time : 438.1485
Wallclock time : 468.0000
Status : VALID
Verification File Name : NAA.wi.ver.c4s
Sumfile File Name : NAA.wsum.c4s
```

The form provides fields for entering short forms of the name of a program, the variant (such as automatically vectorized or manually optimized variant XY), the machine and configuration on which the program was executed, the chosen data set, the time the program was run, and the performance data. There is interactive support for filling out this form. Usually the *verification file name* is the only field filled out. All other information can be derived automatically from the name and the content of this file. The verification file is a file produced at the end of each program run and indicated whether the program ran correctly and in what time. If the name or content of this file cannot be used or the user does not wish to use it, then any or all fields can be overwritten before giving the command to enter the information into the database. The system fills in an *Experiment ID* which one can think of as the page number of the "book" in which the results are recorded.

The *Sumfile* contains profile result data. Its name is also usually derived automatically by naming conventions from the verification file name, and its content is read into the database. For each loop nest of each subroutine in the program it gives the following information:

```
BNDRY_do100 100 AVE: 0.005819 MIN: 0.005615
                MAX: 0.006104 TOT: 0.581856
```

which says that loop with label 100 of subroutine `BNDRY` was executed 100 times, and the average, minimum, maximum, and accumulated timings are as shown.

Once result data for a number of program variants are available, the user may want to look at the most time-consuming loops and compare the timings. The program variants, such as *serial, vector-concurrent*, and *manual variant XY*, were entered in the form shown above and can now be used in a similar form to select the data for the profile display. If the user prefers, experiment ID numbers can be used instead. The top window in Figure 2 shows the resulting display.

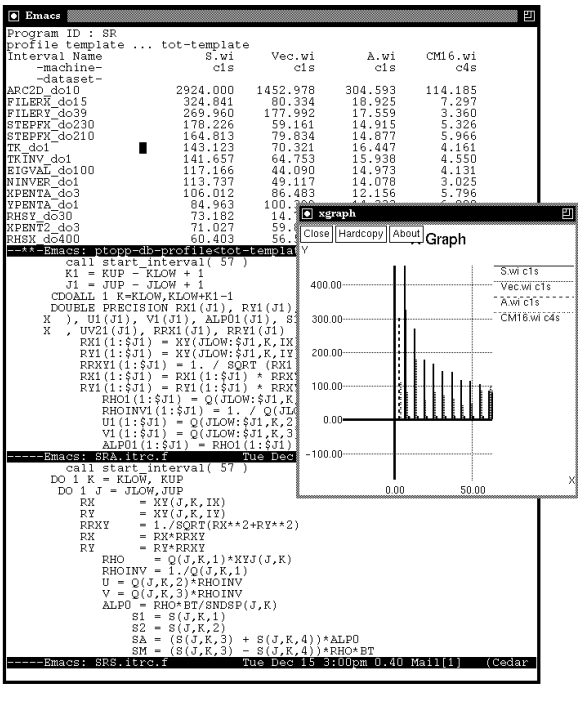

Figure 2: A typical profile analysis session

The loop timing numbers are a useful starting point for understanding the performance behavior of the program. The user may now wish to see the loops in the source code that correspond to some of the timing numbers. This can be done interactively. In Figure 2 the user has selected two program variants for source display and is looking at loop 1 in subroutine `tk`. Commands are available to scan quickly through the most time-consuming sections of a program and review the transformations done to them as well as the resulting performance. Optionally the user can re-

quest a graphical display of the timing information in various styles.

A number of questions recur with each program analysis session, and they have been put together as a methodology of performance analysis [1]. A simple one of these questions is "what is the resulting loop speedup". The profile display can give an answer by adding a new column that divides two given columns. The PTOPP facilities provide a flexible mechanism to generate such derived columns and switch between a range of displays [4]

Discussion The analysis of the program and performance data and the derivation of the potential transformations were among the most time-consuming parts of the development process. The huge amount of result data called for both a database that can manage the data and a methodology that can abstract the data. Both items come at a cost. Introducing a database comes with new notions, and a few additional commands. Furthermore, the user has to learn to think in terms of the methodology. Having gone through this exercise we believe that the benefits outweigh the costs. The database opens a new world of power over the gained data. There is a query language (SQL) that lets the advanced user compose new displays of the data and explore many questions and answers. The methodology turned out to provide a welcome terminology in which to argue about performance effects and explain the behavior of many programs.

These facilities have given us much help for finding new program transformations. Although they do not directly point out specific actions to take, recipes are given that lead to optimizing transformations, depending on the answers to the questions mentioned above. As our project proceeds we expect to refine this methodology of guiding the user through the optimization steps. In many cases, recipes are difficult to give. For example, if loops are not (yet) parallel, the user has to find out whether they can be transformed into parallel forms. Most useful for this purpose is the hit list of most effective transformations found in our project so far. This process is described more detailed in [1].

2.5 Editing transformations

The process of editing the program transformations was comparably fast, although errors introduced in this phase have caused extra debugging sessions in our experiments. It is mentioned in [1] that transformation-directed editors could be of some help. Some of these transformations are not difficult to edit, but they require a careful interprocedural analysis of the program. Examples of such transformations are *privatizing* of arrays and turning a loop into a concurrent loop. The editing action of array privatization is just moving the data declaration from its original place to the loop header, and making a loop concurrent is replacing the keyword `DO` by `DOALL`. However, the analysis of privatizable arrays requires careful definition-use investigations, and making sure a loop is fully parallel requires data-dependence analysis. For our experiments we had only the compiler-generated data-dependence information available, which did not detect privatizable arrays. Tools that promise to be very useful for this analysis are being developed.

Other transformations are easier to analyze but more tedious to edit. They may be the primary candidates for support in transformation tools. Examples are *stripmining* and *loop coalescing*. Such tools are also being designed in related projects [5].

3 Questions and Answers

Assessing the usefulness of new tools is difficult without hands-on experience. Therefore, in this paper we are not trying to argue why our tools are better or worse than others, but we will provide a wide range of opinions by reporting a number of questions and comments we have received so far, and our replies to them.

Did PTOPP really improve the situation? How? If there is any "proof" that PTOPP helps programmers do a better job, it is that the tools were designed out of direct needs in a project that successfully optimized one of the most widely accepted workloads for supercomputers. So far, six people have used the tools in this project. PTOPP helped primarily in two areas: keeping related files in a consistent state and selecting and juxtaposing related information. The former is achieved by naming conventions that reflect program variants, such as compilation, link, execution options, and manual transformations. The latter is achieved though a database and interactive query facilities.

The tool set does not appear to be state of the art. True, PTOPP does not provide the newest interaction technology, such as color-graphical displays and audio. Instead, our focus was on providing the functionality needed in a substantial program optimization project. Although we do not say that modern interaction instruments are the wrong approach to programming environments, our design was influenced by the observation that the cost of learning elaborate interfaces can offset the benefit. Even in terms of user interfaces PTOPP provides some features that state-of-the-art tools often lack. Examples are flexible customization and help capabilities, and the ability to provide much of the functionality through non-graphics terminals.

You claim that all you need is Emacs; this is certainly wrong. Basing the interactive portion of PTOPP on the Emacs editor has allowed us to do what we just said: work on the functionality and still provide an acceptable user interface. For those who use Emacs as their main editor, learning PTOPP is expected to be quite easy. Learning PTOPP plus Emacs is more

difficult, but probably not much more so than learning any other new user interface, such as X-windows-based environments. To those who are fundamentally opposed to Emacs, we can only suggest that they try it.

Are integrated tools important, and what does PTOPP do about them? The need for a certain degree of tool integration arose from many situations in our programming experience. We have addressed the issue of managing program variants and their result data. It is unclear to what extent tool integration is desirable. Highly integrated tool sets tend to obscure the interface to the underlying operating system from those who wish to use it. For example, in order to maintain the consistency expected from integrated environments, they may need to disallow low-level actions such as explicitly renaming files, or copying files from one directory to another. This is often an unwanted side effect. In order to coexist with the Unix environment, PTOPP's approach is to to take small integration steps by providing facilities that support the consistency among file names, object names, and the make commands.

You seem to ignore parallelizing and interactive restructurers. Are they not important? They are! Our tool project is complementing our primary project of creating parallelizing restructurers [2, 3, 5]. That's why little is said about such tools in this paper. In Section 2.5 we mentioned the potential benefit users can expect from interactive restructurers. So far, PTOPP attempted to facilitate the programming tasks that are mechanical in nature, whereas restructuring tools will need to aid the programmer in making many intelligent decisions about program transformations. The integration of all these tools is a long-term goal.

Where are limitations of your tools? Our tool project is a small-scale effort. It would need extensions on many fronts to become a complete tool set. Development phases that are unsupported so far are debugging, program execution, and editing transformations. For each existing tool area in PTOPP there exists a wish list from the user community that will allow us to make the facilities more useful. Finally, the tools need to be ported to further machine environments in order to be of value to a large user community.

Where do you go from here? We will continue to use and extend PTOPP in a strictly need-driven way. The main application will be our project of porting and optimizing large applications to new parallel machines. In the past the Alliant FX/8 and Cedar machines were the targets. In the future we will look at newer machines, which will require us to port the tools to different platforms.

Conclusions

We have presented a set of Unix-based tools that led us to optimize successfully variants of a wide range of application programs, including the Perfect Benchmarks® codes. In this parallel programming effort we identified the management of the variety of files generated during program optimization and the inspection and analysis of program result data as time-consuming programming phases. In a strictly need-driven way we have added simple tools to facilitate these tasks.

The relative success of this modest programming environment project leads us to believe that there is a way out of a big dilemma in which current programming environment research appears to be: the discrepancy between the wealth of both tool research papers and commercial developments, and the usefulness of delivered environments. In contrast, we were able to create simple facilities that were of significant help. Although there is more proof of effectiveness we have to deliver, in terms of widening the tool user community and machine spectrum, the results indicate that tool developments should go in these directions: (1) Tools must be designed in a need-driven way.(2) New tools should be simple extensions of available environments with which users have experience. (3) Interactive interfaces can be built at reasonable costs using existing platforms as well. (4) Although very difficult, we *must* quantify the usefulness of new tools. By lack of a direct goodness measure we propose that the accomplished mission in which newly proposed tools have proven their value be clearly reported. In our case we have successfully optimized one of the most widely accepted program suites for parallel machines.

References

[1] R. Eigenmann, *Toward a Methodology of Optimizing Programs for High-Performance Computers*, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., Report No. 1178, August 1992.

[2] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua, *Restructuring Fortran Programs for Cedar*, Proceedings of ICPP'91, St. Charles, IL, 1 (August 12-16, 1991), pp. 57–66.

[3] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua, *The Cedar Fortran Project*, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. Res. & Dev., Report No. 1262, 1992.

[4] P. E. McClaughry, *PTOPP - A Practical Toolset for the Optimization of Parallel Programs*, Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1992.

[5] P. M. Petersen, *Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques*, PhD thesis, University of Illinois at Urbana-Champaign, January 1993.

[6] S. Sharma, R. Bramley, P. Sinvahl-Sharma, J. Bruner, and G. Cybenko, *P3S: Portable, Parallel Program Performance Evaluation System*, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., Report No. 1170, September 1992.