Tools That Led To Increased Program Performance

Patrick McClaughry and Rudolf Eigenmann [*]
CSRD Report 1184

January 1992

Abstract

This papers describes a set of tools that help a programmer 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 which offers help and customization facilities. The new tools mainly facilitate two development phases that were identified as most time-consuming in the optimization project: The process of creating a consistent set of experimental program variants and the analysis and interpretation of compilation and performance results.

1 Introduction

The tools we are going to describe grew out of a successful effort to optimize the Perfect Benchmarks codes for the Alliant FX/8 and the Cedar multiprocessors. In all codes, significant performance improvements were gained using the methodology underlying this paper. Initially, the tools used for this project were the parallelizing compiler Kap/Cedar[EHJP90] and ordinary Unix utilities for manually improving the parallel Fortran code generated by Kap. Additional tool sets were considered, mainly the ones discussed in Section 3. However, they were not deemed useful for our effort because the cost of installing and learning them seemed high compared to the available evidence about their successfulness in optimizing ordinary programs.

Although we did not find the existing program porting environments directly useful for the given project, it was evident that additional tools could increase the productivity. In fact, some early versions of the tools described in this paper were developed as part of the optimization project. The success of these utilities warranted our tool project. The tool design was derived from a careful analysis of the time-consuming development phases in the optimization effort. The results of this analysis are described in [Eig91]. The following paragraphs summarize the important development steps.

[*]Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 305 Talbot Laboratory, 104 South Wright St., Urbana, IL 61801-2932. This work was supported by the U.S. Department of Energy under grant no. DOE DE-FG02-85ER25001.

The first development step usually was to instrument the program with loop-level timing calls. This was followed by the generation and execution of multiple program variants, each corresponding to a set of compiler options plus a set of files that contained individually optimized program sections. The program result data was then 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 program transformations to improve performance. Further potential transformations were derived from many additional sources of information including the source file, compiler listing, the list of successful transformations of other programs, and tools that showed the maximum possible loop parallelism. The next step was to apply the program transformations using a conventional text-editor. This development cycle was repeated several times. Figure 1 shows this cycle.

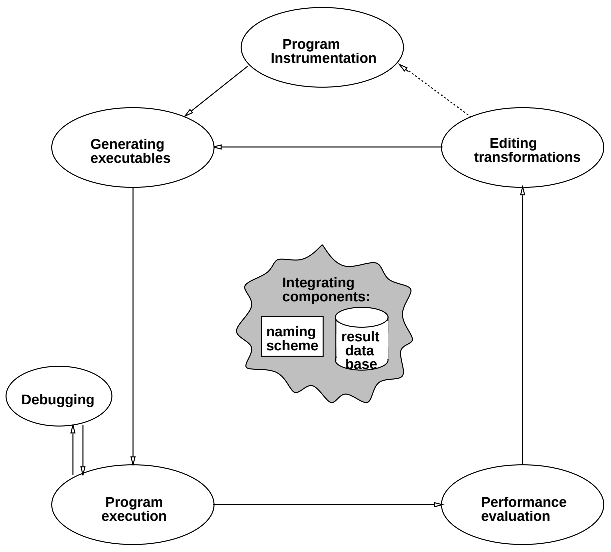

Figure 1: The program optimization cycle

Automatic instrumentation was essential. A simple tool was developed for this purpose and turned out to be a crucial aid. Often there was a need for later re-instrumentation, usually where coarser or finer timing results were desirable.

Generating executable files and running them was time-consuming for two reasons. First, the large number of program variants raised a consistency issue. Errors occurred 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. Further, there were no tools available that could run a set of programs on a specified machine, make sure the input files are available, and transfer the output files to the right place.

Debugging can be time-consuming, however this issue is less significant for porting existing applications than when developing new software. In the porting activities considered in this paper it was of great help to have available the correct program version from which the development started. Our situation could have been improved by the availability of a symbolic debugger. In addition, a number of mistakes could have been detected by simple tools that check the program statically. An example of such a tool is a static "race detector".

The analysis of the program and performance data and the derivation of the potential transformations were the most time-consuming parts of the development process. The huge amount of result data calls for some sort of a database that can manage and filter the data. A methodology was introduced in [Eig91] that determines important factors from this data and helps the user explain the program performance and find transformations for its improvement. Here is a possible place for major growth in new tool designs. They would have to facilitate the analysis of all available sources of information and guide the user to a successful program transformation.

The process of editing the program transformations was comparably fast, although errors introduced in this phase have caused extra debugging sessions. It is mentioned in [Eig91] that transformation-directed editors could be of some help.

The need for a certain degree of tool integration arose from many situations. We have already mentioned the issue of managing the result data of many program runs. The major additional need was for a consistent naming scheme among the programs, their variants, intermediate files, result files, and the program and performance 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. This is often an unwanted side effect. In order to coexist with the Unix environment, it seems feasible for new tools to take small integration steps by providing facilities that support the consistency among file names, object names, and the make commands and options with which the objects are made.

2 Description of Our Tool Set

We have created a tool set that addresses several parts of the porting process (see Figure 1). These services are divided into three components: interactive Cedar Fortran[1] source file utilities, Cedar Fortran profile generation and manipulation utilities, and UNIX level services. The source file utilities contain functionality to aid in instrumentation and compilation of programs. The profile utilities cover the generation and analysis of performance data during the porting operation. We provide UNIX level services because we do not wish to limit access to our tools to the interactive environment. These services permit the user to perform many of the functions available in the interactive environment from the UNIX command line. Before describing each of these parts in greater detail, we will describe the context in which these tools are to be used.

2.1 The intended users, the interface, and the environment

The needs of the users have been a primary motivation in the design of our tool set. 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.

From the beginning we have envisioned tools which could offer an interactive interface as well as a UNIX command line interface. An interactive interface has the benefit of permitting

[1]Our utilities are named Cedar Fortran because they were designed to be used on the Cedar multiprocessor at the Center for Supercomputing Research and Development.

easier "what-if" analysis. Furthermore, an interface which can interact with the user 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. However, we also realize that interactive interfaces can present barriers to users. If the command set is too complex, unfamiliar, or diverse the user may resist using the tool set. This thought has prompted us to consider a UNIX level version of many of the functions provided by our tool set, thus exploiting the knowledge of UNIX present in our users.

Because of its familiarity and ease of use we have chosen Emacs as the interactive environment. Emacs offers a full-featured text editor, file management facilities, interactive help, and a familiar language in which to program (ELisp). Because Emacs is customizable, each user can change many of the elements of our tool set to make the tool set more comfortable to use. For example, commonly used functions can be bound to easily remembered keystrokes. Recognizing that not all users want to use Emacs, we offer a UNIX level interface for many of the functions present in our tool set. Wherever possible we have provided a command which produces results similar to those available inside the Emacs environment. In fact several tools are really UNIX scripts and Emacs is only used to compose the UNIX command for the user.

During the porting and optimizing process a considerable amount of performance data can be generated. We intend to capture this data 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.

In our tool set, PTOPP (Practical Tool Set for the Optimization of Parallel Programs), we provide two levels of access to the database. If the user needs only limited power from the database, they can use PTOPP where many of the interface details are hidden. If the user needs greater power from the database, they can exploit it fully using a UNIX level interface that requires knowledge of SQL and the internal structure of the database. It is our intention that PTOPP provide reasonably complete database services and the UNIX interface be reserved for more complex queries.

2.2 Cedar Fortran mode

Cedar Fortran mode is an Emacs mode (and is thus part of the interactive interface of PTOPP) that provides several helpful functions useful in the porting and optimization of programs. Cedar Fortran mode (CF) is useful during the early part of a porting process. It provides functions to aid in the instrumentation and the compilation of programs. CF can also be useful later in the porting process by offering interactive facilities to view particular program points.

2.2.1 Instrumentation

Instrumentation in our context is the addition of source code to generate event timings. The code is added in pairs; a call to record a start time and a call to record a finish time. These instrumented regions are called intervals. Intervals are given a name during the instrumentation process which is later used to identify important performance measurements. When using CF mode, users have several ways of inserting instrumentation into their program. CF offers an automatic method which instruments the outermost loops of a given module or program. This automatic method is often useful for a first approximation of the most time consuming portions of a program. The instrumentation tool is available at the UNIX level as well. CF also offers interactive insertion and removal of instrumentation calls. By marking a region of Fortran text and invoking a CF command, the user can insert instrumentation calls around the region. Using another instrumentation command the user can comment out instrumentation calls thus removing them from the trace reports. Since instrumentation causes perturbations in the overall runtime of a program, it is common to remove unimportant intervals from the analysis. The instrumentation calls can easily be reinstated again.

2.2.2 Compilation

CF mode offers several ways to compile a program, both interactively and from the UNIX command line. The UNIX level tool is called `cfmake` and it creates a makefile which can be used to generate executable code. `cfmake` allows the user to make a set of consistent compilation commands and options and it supports quick modifications to individual subroutines. Section 2.4 describes the command in more detail. The makefile format is suitable for the UNIX utility `make`. Modification of the makefile by the user permits variations that are outside the scope of the design of the tool. For example, if unusual dependencies between files need to be maintained, appropriate lines can be added to the makefile.

When invoked from within the Emacs environment, compilation takes one of two forms. The most general way of producing an executable from a given source is to invoke the compilation command from within CF. This command composes a UNIX command line using filenames derived from the current context and presents the command in a UNIX shell buffer within Emacs. The user may make changes to this command before execution. After any necessary changes, the command is executed with the output being captured by the Emacs buffer for examination. In a sense this tool is really both a CF tool and a UNIX level tool; it can be invoked in a similar form directly from the UNIX command line.

Nicknamed compilation is a tool which adds simple, user-defined nicknames to certain commonly used compilation schemes. A nickname is a tuple of items expressing the elements of a properly composed compilation command. Some of the items recorded in a nickname are the source files required, the object files required, and the compiler, linker, and preprocessor options. New nicknames are defined by extracting the items from a user supplied compilation command. Once defined, a nickname can be used repeatedly so that the user no longer needs to remember all the component parts of a properly composed compilation command.

While composing a compilation command from within CF the user has quick access to compiler, linker and preprocessor option descriptions. A single keystroke displays these options without the need to abandon the compilation command.

2.2.3 Source-based information

Moving around in a large source file can be time-consuming. CF has several functions which make it easier to navigate through large source files. Since Fortran programs often consist of subroutines, CF offers the facility to locate a particular subroutine or loop in the source file and bring it into view. In the cases when the source file is already instrumented for analysis, CF can locate an interval given the interval name. This tool makes it easy to examine the source code responsible for a particular timing result. The function is used in the Cedar Fortran profile mode. This tool resembles the `etags` utility available with Emacs. At present `etags` works at the subroutine level rather than the interval level, but in the future it may make a good starting point for more elaborate positioning tools.

2.3 Cedar Fortran profile mode

The tasks of generating and manipulating instrumentation timing reports is left to Cedar Fortran profile mode (CFP). CFP has tools which help the user in specifying the tabular reports of instrumentation timings and methods which permit manipulations of the resulting tables.

2.3.1 Profile generation

The fundamental components of a CFP profile are one or more trace timing files produced during the execution of an instrumented program. The format of the input trace timings can, of course, vary depending upon the instrumenting libraries and the type of data generated. It is also likely the user will want to combine certain data to produce derived measurements, i.e. speedup from timings. CFP is designed to address the general problem of composing a profile given a disparate set of instrumentation files.

Users of PTOPP can specify a translation rule that is applied to the performance data as it is being read into the database. This rule tells PTOPP which portions of the data to retain for future use and which to ignore. It also normalizes the data in the database. The translation rule is thus used to capture various formats of performance data.

The user composes a profile by defining a mapping from the data recorded in the database to the desired profile output. For example, a profile column could be specified as the result of dividing parallel execution time by serial execution time to indicate relative speedup of a transformation. PTOPP will include a Profile Description Language for the straightforward expression of such a mapping. These mappings will be saved in the form of templates which can be reused in other contexts. Figure 2 shows a profile consisting of total interval execution times for 3 variants of a program.

2.3.2 Interactive Profile Commands

CFP also permits the user to use the profile as a starting point for more information about the program and its execution. The user can see the raw trace timing data by invoking a CFP function while positioned over the desired interval. This is useful when the user does not wish to recompose a profile but would like to see some aspect of the trace different from that specified in the profile. The user may also attach one or more source files to a profile buffer. Then, when desired, the user can view the relative positions of the interval under analysis in each of the attached files simultaneously. It is also possible to sort the rows of a profile on any column of the profile in increasing or decreasing order. Figure 2 shows a typical CFP buffer with its associated source files.

2.4 Notable Implementation Aspects

Interfacing PTOPP with INGRES. An interesting implementation issue is the interface between PTOPP and the database holding the performance data. The database manager we are currently using is INGRES.

```
Program ID : LW
profile template ... tot-template
Interval Name             S.wi     Vec.wi   CM9x8.wi
    -machine-              c1s        c1s        c4s
    -dataset-
MDMAIN_do2000         3990.194   3092.734    202.099
INTERF_do1000         3707.857   2841.540    162.781
POTENG_do2000          272.430    221.535     13.545
PREDIC_do1000           27.775     43.080      4.156
INTRAF_do1000            8.610      8.554      2.023
CORREC_do1000            7.475      4.853     20.101
INTRAF_do2000            0.921      0.161      0.238
KINETI_do100             0.650      0.288      0.131
BNDRY_do100              0.459      0.352      0.842
INTERF_do2000            0.433      0.416      0.000
POTENG_do1000            0.424      0.198      0.110
MDG_do101                0.201      0.170      0.143
INITIA_do2000            0.056      0.056      0.112
INITIA_do300             0.042      0.047      0.053
```

Figure 2: A typical profile analysis session

As was mentioned previously, we are using Emacs as a front end for the interactive portion of PTOPP. INGRES offers its own command interpreter as well as a C level interface. Since Emacs has facilities for executing UNIX commands within a buffer, we have developed a simple interface using a UNIX script that accesses the INGRES command interpreter. This script accepts an SQL database command, executes it against the INGRES database using input and output redirection, and composes the result in a form understandable by Emacs. PTOPP must then parse the resulting information into the required data structures.

A further complication is that the error messages reported by INGRES appear in the expected data file without a return error code from the UNIX process. Thus errors can only be detected by correct parsing of the output stream. This error parsing must take place in PTOPP, making error detection and recovery tricky.

While this primitive interface is sufficient for our current intentions, we are investigating more robust and flexible ways of supporting communication between PTOPP and the database. We are also examining other database solutions in search of a better match.

A consistent naming scheme. Another interesting aspect is the function used by the PTOPP building blocks to support a consistent naming scheme. The intelligence of this function is based on a *naming file* in which the user describes how names and commands are composed from their items. For example, a filename may consist of a *programId*, a *variantId*, a *traceId*, and a *type_extension*. Similarly, a compilation command may be composed of a *compilername*, a *sourcefile*, and *options*. There is a simple description language for these items that provides operators for string manipulations, and composition alternatives. The tools call the naming function in a form such as "given the executable filename; return the compilation command and source file" or, in general, "given a set of known items; derive the requested set of related items".

The `cfmake` command. A third notable detail is how the `cfmake` command generates executable files. Both *derived* and *composed* files are supported. Derived files are simply the ones whose make commands can be derived by the naming function described above. The user only passes the name of the executable file to the `cfmake` command. As a result, a Unix make file is created and run. `cfmake` generates a composed file when given a number of files containing individually optimized subroutines plus a *default file*. The default file supplies all routines that aren't specified explicitly on the command line. This allows the user to quickly modify individual routines of the program given as the default file. The `cfmake` command works by compiling all routines of the default file separately (if not yet done so) and then linking only those needed.

3 Related Work

There are several programming environments offering tool sets that are similar in some part to what PTOPP offers (Faust [VGGJ+89], Start/Pat [ASM89],SIGMACS [SG90],R^N [CCH+87]). However, the similarities are somewhat superficial. All of these environments, including ours, attempt to make the life of a programmer easier. They all make an effort to integrate the tools into a common interface that tries to be intuitive to use. However, beyond this our approach diverges from many of the others.

Comparing PTOPP to other environments. Perhaps one of the most obvious differences between PTOPP and many of the other parallel programming environment efforts is the user interface environment. Most of the other environments use X-Windows to present their tools to the user. We have chosen a text-based interface as the prototype environment for PTOPP and we will offer graphics capabilities as an option where needed. We believe this will make PTOPP more versatile and familiar. SIGMACS also has an Emacs front end and as such resembles PTOPP more than any of the other environments. During the development of PTOPP we have been concerned with users shunning a tool because the interface is unfamiliar or difficult to learn. Rather than developing an entirely new environment in which to work, we chose to exploit the environment which with all of our users are familiar: UNIX.

Faust and SIGMACS have strong notions of a project. For them a project includes source, object and executable code, compilation information, program data, etc. The environment supports operations on these components of the project, such as file management. The user remains within the environment during the entire process with these operations as the interface to the underlying system. This notion of a project is much stronger than the notion present in PTOPP. PTOPP manages source, object, executable and performance result files through a heavy reliance on its UNIX underpinnings. No explicit project management facilities are provided but rather users are encouraged to use the UNIX directory hierarchy to organize their files. While we likely will expand our notion of a project, possibly using the database, for the time being we feel a UNIX foundation is a solid, familiar way to manage project files.

Another shared feature among many of the other programming environments is a project database. Faust, Start/Pat, R^N and SIGMACS all maintain a project database for their users.

This database often manages access to data about the program under development such as data dependency information and the program source. The user may browse through this data during program porting or development. This use of a database differs from that of PTOPP. PTOPP uses a database to manage the performance data reported after a program execution. Our database does not contain program structure data primarily because the tools necessary to generate this data are not available at this time. We also see maintaining the consistency of this data as a concern for which we do not currently have a solution.

While PTOPP has features which help the user evaluate program transformations, SIGMACS and Start/Pat go beyond this to suggest possible transformations that apply to a particular piece of code. The user may choose from these applicable transformations and the environment makes the changes. While we do expect PTOPP to support feedback from the compiler on the transformations applied or discarded, we do not expect PTOPP to make recommendations on the best transformation.

Start/Pat and R^N include debugging interpreters and the ability to interactively replay the execution of a program. Users of PTOPP use the debugging techniques available at the UNIX level while within Emacs. Several techniques used in the porting process are mentioned in [Eig91]. At this point we do not plan on providing replay/simulation capabilities in PTOPP.

Some of PTOPP's differences. Perhaps because of its close ties to the actual porting of benchmark programs, PTOPP offers some functionality not well supported by the other environments. It is not clear that any of the related work address the automated or manual instrumentation of the source code for production of trace timings. Faust's Impact lets a user examine a trace file, but it is not known how the file was generated. We have found that timing measurements from actual execution runs rather than simulations were useful in the identification of the portions of a program where our optimization efforts would be most fruitful.

Since we use actual execution timings as a basis for our performance optimizations, PTOPP includes aids for generation of executable code. Most of the similar environments are not based on the results of actual program executions. Due to this, little support for program compilation is mentioned. The focus of these environments is more the generation of data to aid in the restructuring of a program than the generation of real world performance figures.

Similarly, PTOPP generates and manipulates the performance figures in a way different than the other environments. The closest similar tool is Faust's Impact which graphically displays an event time line. However, PTOPP is more concerned with presenting the timing figures in such a way that the user can see the success (or failure) of a particular transformation quickly.

Evaluating the success of an environment. Something missing from most of the papers is a discussion of the level of success of each of these environments. Have they been met with wide acceptance? How long does it take a new user to become familiar with the interface? Do they find the environment sufficient or do they go outside the environment to do certain tasks for which the tools were originally designed? Does the environment improve the productivity of the user and if so, how was the increase measured?

PTOPP has been designed in an incremental way directly from experience gained in the porting and optimizing of parallel programs. At each step we have tried to generalize the techniques enough to permit individual variations without unnecessarily overburdening the environment. While this may mean that PTOPP lacks certain features, the tools that make up our tool set have proved their value even before they became a part of PTOPP.

4 Conclusion

Because virtually everything can serve as a tool, it seems necessary to talk about what makes a good tool. We feel that a tool must fill a void in the users current toolbox. A tool should address a particular need or perhaps several needs of the users. Another crucial aspect of a good tool is that it is used. While this might sound obvious, many tools designed to aid a group of users go unused for a number of reasons.

One possible reason a tool may go unused is also something we feel makes a tool useful. A tool should be able to adapt to the needs of the users. Thus the tool remains useful and the user doesn't need to reinvest substantial amounts of time in learning new tools. Along the same lines, if a tool is easy to learn in the first place, it is more likely to be used.

We have collected a set of tools that have grown out of specific needs. These needs were realized by a (albeit small) number of professionals at the Center for Supercomputing Research and Development during the porting of a large number of benchmark applications. We have wrapped our tools in an environment that is easy to learn and use. We have also tried not to stray too far from the familiar and powerful UNIX interface in which all of our users are comfortable. By doing so we have directly addressed some of the most time consuming portions of the porting and optimization process.

References

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

[CCH+87] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, Linda Torczon, and Scott K. Warren. A Practical Environment for Scientific Programming. *IEEE Computer*, pages 75–89, November 1987.

[EHJP90] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. Cedar fortran and its restructuring compiler. In A. Nicolau D. Gelernter, T. Gross and D. Padua, editors, *Languages and Compilers for Parallel Computing II.* MIT Press, 1990.

[Eig91] Rudolf Eigenmann. Towards a methodology of optimizing programs for high-performance computers. Technical Report 1178, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, December 1991.

[SG90] Bruce Shei and Dennis Gannon. SIGMACS: A Programmable Programming Environment. *3rd Workshop on Languages and Compilers for Parallel Programming*, 1990.

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