Optimization tools

After your code is compiled, debugged, and capable of running to completion or planned termination, you can begin looking for ways in which to improve execution speed. In general, the opportunities for optimization fall into three categories, which require progressively more programmer effort. These categories are:

- Improving overall I/O
- Improving use of compiler-generated optimizations
- Analyzing code behavior and rewriting code to optimize performance

1) Improving Overall I/O

a. Using IOBUF

IOBUF is an I/O buffering library that can reduce the I/O wait time for programs that read or write large files sequentially. IOBUF intercepts standard I/O calls such as read and open and replaces the stdio (glibc, libio) layer of buffering with an additional layer of buffering, thus improving program performance by enabling asynchronous prefetching and caching of file data.

IOBUF can also gather runtime statistics and print a summary report of I/O activity for each file. In general, no program source changes are needed in order to take advantage of IOBUF. Instead, IOBUF is implemented by following these steps:

1. Load the IOBUF module:

- **module load iobuf**

2. Recompile (or optionally, relink) the program.

3. Set the IOBUF_PARAMS environment variable as needed

- **setenv IOBUF_PARAMS='*:verbose'**

Note: The simplest parameter specification is:

- **setenv IOBUF_PARAMS='*'**

  This setting matches all files and enables buffering with default parameters.

4. Execute the program.

If a memory allocation error occurs, buffering is reduced or disabled for that file and a diagnostic is printed to stderr. When the file is opened, a single buffer is allocated if buffering is enabled. The allocation of additional buffers is done when a buffer is needed. When a file is closed, its buffers are freed.

The behavior of IOBUF is controlled by the use of environment variables, the most significant of which is **IOBUF_PARAMS**. If this environment variable is not set, the default state is no buffering and the I/O call is passed on to the next layer without intervention.

For more information about valid IOBUF_PARAMS parameters and their usage, see the **iobuf(3) man page**.

b. Improving MPI I/O

When working with MPI code, one of the most effective ways to realize significant improvements in program execution speed is by fine-tuning MPI rank placement and I/O usage. The Cray Message Passing Toolkit (MPT) provides more than forty environment variables designed to help you do just that, the two most significant of which are MPICH_MPIIO_HINTS and MPICH_RANK_REORDER_METHOD.

For a listing of the MPI environment variables and their valid values and uses, see the **intro_mpi(3) man page.**

A full discussion of MPI I/O optimization is beyond the scope of this document.

For more information on this subject, including detailed explanations and examples, see **Getting Started on MPI I/O** in **Message Passing Toolkit Section.**

Optimizing MPI rank placement can require considerably more detailed analysis. **Alternately**, you can use **Cray Performance Analysis Tools** to **instrument your program to study MPI behavior**, and then to generate suggested MPI rank reordering information. For more details, see the **intro_craypat(1), pat_build(1), and pat_report(1) man pages**.

2) Using Compiler Optimizations

Cray Compiling Environment (CCE)

The Cray FORTRAN and C/C++ compilers are optimizing compilers that perform substantial analysis during compilation and generate highly optimized code automatically. The Cray compilers also support a large number of command-line arguments that enable you to exert manual control over compiler optimizations, and fine-tune the behavior of the compiler.

For more detailed information about the Cray Fortran, C, and C++ compiler command-line arguments, see the **crayftn(1), craycc(1), and crayCC(1) man pages**, respectively.

Two of the most useful compiler command-line arguments are the Fortran –rd and C/C++ -h list=m options, which cause the compiler to generate annotated loopmark listings showing which optimizations were performed where in the code.

Together with the -h negmsgs options, which generate listings showing which potential optimizations were not performed, and why, these arguments can help you zero-in on areas in your code that are compiling without error, but not with maximum efficiency.

For more detailed information about generating and reading loopmark listings, see the **Cray Fortran Reference Manual and Cray C and C++ Reference Manual in Compiler Environment Section.**

The Cray compilers also support a large number of pragmas and directives that enable you to exert manual control over compiler optimization behavior. In many cases, code that is not optimizing well can be corrected without substantial changes to the code itself, but simply by applying the right pragmas or directives. For more information about Cray compiler pragmas and directives, see the **intro_directives(1) man page**.

3) Using the Cray Performance Analysis Tools

After you have compiled and debugged your program, you are ready to begin analyzing its performance. The Cray Performance Analysis Tools are a suite of optional utilities that enable you to capture and analyze performance data generated during the execution of your program in order to help you to find answers to two fundamental questions: How fast is my program running? How can I make it run faster?

The Cray Performance Analysis Tools suite consists of three components:

- CrayPat: the program instrumentation, data capture, and basic text reporting tool
- Cray Apprentice2: the graphical analysis and data visualization tool
- PAPI: the Performance Application Programming Interface

To begin working with the performance analysis tools, first load your programming environment of choice, and then load the perftools module.

- **module load perftools**

The performance analysis process consists of three basic steps.

- Instrument your program, to specify what kind of data you want to collect under what conditions
- Execute your instrumented program, to generate and capture the desired data.

- Analyze the resulting data.

i) CrayPat

The CrayPat run time environment, which collects the specified performance data. Accordingly, CrayPat consists of the following major components:

- pat_build, the utility used to instrument programs
- pat_report, the first-level analysis tool used to produce text reports or export data for more sophisticated analysis
- pat_help, the command-line driven online help system

All CrayPat components, including the man pages and help system, are **available only when the perftools module is loaded.**

For successful results, the perftools module must be loaded before you compile the program to be instrumented, instrument the program, execute the instrumented program, or generate a report. If you want to instrument a program that was compiled before the perftools module was loaded, you may under some circumstances find that re-linking is sufficient, but as a rule it's best to load the perftools module and then recompile.

When instrumenting a program, CrayPat requires that the object (.o) files created during compilation be present, as well as the library (.a) files, if any. However, most compilers automatically delete the .o and .a files when working with single source files and compiling and linking in a single step, therefore it is good practice to compile and link in separate steps and use the compiler command line option to preserve these files. For example, if you are using the Cray Compiling Environment (CCE) Fortran compiler, compile using either of these command line options:

- **ftn -c sourcefile.f**

Alternatively:

- **ftn -h keepfiles sourcefile.f**

Then link the object files to create the executable program:

- **ftn -o executable sourcefile.o**

Instrumenting the Program

After the program is compiled and linked, use the pat_build command to instrument the

program for performance analysis. In simplest form, pat_build is used like this:

- **pat_build -O apa executable**

This produces a copy of your original program, which is named executable+pat (for example, a.out+pat) and instrumented for the default experiment. Your original executable remains untouched.

The pat_build command supports a large number of options and directives, including an API that enables you to instrument specified regions of your code. These options and directives are summarized in the pat_build(1) man page and documented more extensively in Using Cray Performance Analysis Tools.

Collecting Data

Instrumented programs are executed just like any other program; either by using the aprun command if your site permits interactive sessions or by using your system's batch commands.

CrayPat supports more than fifty optional run time environment variables that enable you to control instrumented program behavior and data collection during execution.

For example, if you use the C shell and want to collect data in detail rather than in aggregate, consider setting the PAT_RT_SUMMARY environment variable to 0 (off) before launching your program.

/lus/nid00008> setenv PAT_RT_SUMMARY 0

Doing so records data with timestamps, which makes additional reports available in Cray Apprentice2, but at the cost of potentially much larger data file sizes and somewhat increased overhead.

The CrayPat run time environment variables are summarized in the **intro_craypat(1) man page** and documented more extensively in Using Cray Performance Analysis Tools.

Analyzing Data

Assuming your instrumented program runs to completion or planned termination, CrayPat outputs one or more data files. The exact number, location, and content of the data file(s) varies depending on the nature of your program, the type of experiment for which it was instrumented, and the run time environment variable settings in effect at the time of program execution.

All initial data files are output in .xf format, with a generated file name consisting of your original program name, plus pat, plus the execution process ID number, plus a code string indicating the type of data contained within the file. Depending on the program run and the types of data

collected, CrayPat output may consist of either a single .xf data file or a directory containing multiple .xf data files. If the program was instrumented with the -O apa option, a file with the suffix .apa is also generated. This file is a customized template for this program and is created for use with future instrumentation experiments.

To begin **analyzing the captured data**, use the **pat_report command**. In simplest form, it looks like this:

/lus/nid00008> pat_report myprog+pat+PID-nodes.xf

The pat_report command accepts either a file or directory name as input and processes the .xf file(s) to generate a text report. In addition, it also exports the .xf data to a single .ap2 file, which is both a self-contained archive that can be reopened later using the pat_report command and the exported-data file format used by Cray Apprentice2.

The pat_report command provides more than thirty predefined report templates, as well as a large variety of user-configurable options. These reports and options are summarized in the **pat_report(1) man page** and documented more extensively in using Cray Performance Analysis Tools

ii) Cray Apprentice2

Cray Apprentice2 is an optional GUI tool that is used to visualize and manipulate the performance analysis data captured during program execution. Cray Apprentice2 can be run either on the Cray system or, optionally, on a standalone Linux desktop machine. Cray Apprentice2 can display a wide variety of reports and graphs, depending on the type of program being analyzed, the way in which the program was instrumented for data capture, and the data that was collected during program execution.

Cray Apprentice2 is not directly integrated with CrayPat. You cannot launch Cray Apprentice2 from within CrayPat, nor can you set up or run performance analysis experiments from within Cray Apprentice2. Rather, use CrayPat first, to instrument your program and capture performance analysis data, and then use Cray Apprentice2 to visualize and explore the resulting data files.

The number and appearance of the reports that can be generated using Cray Apprentice2 is determined by the kind and quantity of data captured during program execution, which in turn is determined by the way in which the program was instrumented and the environment variables in effect at the time of program execution.

For example, changing the PAT_RT_SUMMARY environment variable to 0 before executing the instrumented program nearly doubles the number of reports available when analyzing the resulting

data in Cray Apprentice2.

To run Cray Apprentice2, load the perftools module, if it is not already loaded.

- **module load perftools**

  Then use the **app2** command to launch Cray Apprentice2.

- **username@login1> app2 [datafile.ap2] &**

Note: Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the app2 command returns an "unable to open display" error, contact your system administrator for help in configuring X Window System hosting and forwarding.

At this point the GUI takes over. If you specified a data file name with the app2 command, the file is opened and parsed and the Overview report is displayed. If you did not specify a data file name, the Open File window opens and you can use standard GUI tools to browse through the file system and select the data file you want to open.

For more information about using Cray Apprentice2, see the **app2(1) man page**.

iii) PAPI

The Performance API (PAPI) is a standard API for accessing microprocessor registers. CrayPat uses PAPI to interface to the Cray system hardware; therefore the module papi is normally loaded as part of the perftools module.

The interface between PAPI and CrayPat is normally transparent to the user. However, advanced users may want to bypass CrayPat and work with PAPI directly.

In this case, you must unload the perftools module and then reload only the papi module.

- **module unload perftools**
- **module load papi**

Note: CrayPat and direct PAPI commands cannot be used at the same time. Therefore, pat_build does not allow you to instrument a program that has also been instrumented using calls to PAPI functions.

For more information about PAPI, see the **intro_papi(3) and papi_counters(5) man pages.**

To see what sorts of information PAPI is capable of capturing on your system, use the **papi_avail** and **papi_native_avail** commands.

Additional information about using PAPI is available through the PAPI website, at http://icl.cs.utk.edu/papi/.