January 1992
Order Number: 312139-002

iPSC®/860

PARALLEL PERFORMANCE ANALYSIS TOOLS MANUAL

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	04/91
-002	Revision	01/92

RESTRICTED RIGHTS

Preface

This manual describes the iPSC®/860 Parallel Performance Analysis Tools (PAT). The Performance Analysis Tools are a set of tools that allow you to automatically or manually invoke profiling and tracing data gathering to files that can be analyzed later by the PAT utilities.

NOTE

In this manual, the term "iPSC system(s)" refers to any or all of the following SSD products: iPSC®/860, and iPSC®/860S.

A Performance Analysis Tools user will be interested in observing the operating performance of the iPSC system for a variety of reasons. The PAT utilities provide both tabular and graphic reports that allow performance comparisons over time or comparisons with the performance on other machines.

This manual assumes that you are an application programmer proficient in either the C or Fortran language and the UNIX operating system. The manual provides you with enough detail to begin using the PAT utilities with your iPSC system, and full user and reference information to support advanced use of PAT with the iPSC system.

Organization

Chapter 1 "Overview of PAT" presents an overview of the Performance Analysis Tools utility set, and describes the tasks that a user can accomplish using the PAT utilities.

Chapter 2 "Gathering Performance Data" describes the procedures to use for gathering performance data on application programs. The performance data is usually written to separate files when the application terminates. The performance data is then post-processed and examined using the PAT utilities.

Chapter 3	"Using Execution Profiling" presents tutorial and user information on using PAT to perform execution profile analysis on iPSC jobs. The execution profiler operates the same as common profiling tools on sequential computer systems. It monitors subroutine usage and analyzes source/assembly code.
Chapter 4	"Using Communication Profiling" presents tutorial and user information on using PAT to perform communication analysis on iPSC applications. The communication profiler is a utility designed to analyze and quantify the time spent communicating, calculating, and performing I/O functions.
Chapter 5	"Using Event Driven Profiling" presents tutorial and user information on using PAT to perform event driven profile analysis on iPSC applications. This is a tool which allows the analysis of user specified events with particular emphasis on the interactions between multiple processors. It also supports toggles to simplify the task of timing and analyzing blocks of source code.
Chapter 6	"PAT Command Reference" contains reference information on the commands directly related to the collection of PAT profiling information. These commands invoke the PAT profiling utilities so you can analyze performance information in a graphical windowed environment.
Chapter 7	"C System Calls" contains reference information on the C language system calls that are used in compiled C programs to extract PAT profiling information.
Chapter 8	"Fortran System Calls" contains reference information on the Fortran language system calls that are used in compiled Fortran programs to extract PAT profiling information.
Appendix A	"PAT Configuration Information" contains set up instructions for using PAT in the supported environments. The basic software installation and configuration on the SRM are already covered in the Release Notes for the iPSC/860 system software.
Appendix B	"PAT Common Data Structures" is a set of tables that list the default and optional operations that are profiled or traced by the PAT profiling tools.

Notational Conventions

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be used exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

Bold-Italic-Monospace
Identifies user input (what you enter in response to some prompt).

`Bold-Monospace`
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

`<Break>` **`<s>`** **`<Ctrl-Alt-Del>`**

[ ] (Brackets) Surround optional items.

. . . (Ellipsis dots) Indicate that the preceding item may be repeated.

| (Bar) Separates two or more items of which you may select only one.

{ } (Braces) Surround two or more items of which you must select one.

Applicable Documents

For more information, refer to the following manuals:

iPSC® System Manuals

(NEW) *iPSC® System Technical Documentation Guide*
312026-002
Describes the technical documentation that supports the iPSC System and tells how to use the various documents.

(REV) *iPSC®/2 and iPSC®/860 Programmer's Reference Manual*
311708-004
(Replaces 311071-003, 311019-003, and 311831-001)
Provides detailed information on all C and Fortran routines and commands for the iPSC system.

iPSC®/2 and iPSC®/860 User's Guide
311532-007
Overviews the iPSC system, including hardware and software architectures. Tells how to develop and run programs.

(NEW) *iPSC®/860 C Compiler User's Guide*
312130-001
(Replaces 312006-001)
Describes the C cross-compiler and compiler driver for iPSC/860 systems.

(NEW) *iPSC®/860 Fortran Compiler User's Guide*
312131-001
(Replaces 312006-001)
Describes the Fortran cross-compiler and compiler driver for iPSC/860 systems.

i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual
240436-003
Tells how to use the i860 microprocessor assembler and linker. When you order this manual, you also receive the following manuals:

i860™ 64-Bit Microprocessor Object File Utilities Reference Manual
464410-002
Provides reference information for using the i860 microprocessor object file utilities.

i860™ 64-Bit Microprocessor Simulator and Debugger Reference Manual
240437-003
Describes the i860 microprocessor debugger and simulator.

i860™ 64-Bit Microprocessor Programmer's Reference Manual
240329-002
Tells how to use the i860 microprocessor.

Other Manuals

C: A Reference Manual - Harbison and Steele
480628-001
Describes the C programming language.

(NEW) *Effective Fortran 77* - Michael Metcalf
312201-001
Describes the Fortran 77 programming language.

The C Programming Language - Kernighan and Ritchie
122008-002
Describes the C programming language.

Getting the Latest Version of the Examples

The example programs described in this manual, as well as many other examples are available by anonymous **ftp** from **export.ssd.intel.com** as compressed **tar** files. To obtain the examples, you must have Internet access. Make an **ftp** connection, log in as *anonymous* (please use your email address as your password), go into binary mode, enter the directory `pub/R3.3-examples`, and transfer the file *Fexamples.tar.Z* for the Fortran examples and *Cexamples.tar.Z* for the C examples. Here is a sample session that illustrates obtaining the Fortran examples.

```
% ftp export.ssd.intel.com
Connected to export.ssd.intel.com.
220 export FTP server (SunOS 4.0) ready.

Name (export.ssd.intel.com:ted): anonymous
331 Guest login ok, send ident as password.
Password:
230 Guest login ok, access restrictions apply.

ftp> binary
200 Type set to I.

ftp> cd pub/R3.3-examples
250 CWD command successful.

ftp> get Fexamples.tar.Z
200 PORT command successful.
150 Binary data connection for Fexamples.tar.Z
(137.46.201.3,1143)
(53679 bytes)
226 Binary Transfer complete.
local: Fexamples.tar.Z remote: Fexamples.tar.Z
53679 bytes received in .66 seconds (79 Kbytes/s)

ftp> quit
221 Goodbye.
```

When the file is transferred, move it to the directory in which you want your examples to reside, uncompress the file, and then untar it.

```
% uncompress Fexamples.tar.Z
% tar xvf Fexamples.tar
```

A directory called *examples* appears in your current directory. The directory structure under *examples* contains programs, makefiles, and README files. If, you detect any errors when running these programs, please let us know so that we can correct them. We can be reached at **techpubs@ssd.intel.com.**

Table of Contents

Chapter 3
Using Execution Profiling

Chapter 4
Using Communication Profiling

Chapter 4
Using Event Driven Profiling

Chapter 5
PAT Command Reference

Chapter 6
C System Calls

Chapter 7
Fortran System Calls

Appendix A
PAT Configuration Information

Appendix B
PAT Common Data Structures

List of Illustrations

List of Tables

Overview of PAT 1

Introduction

The Parallel Performance Analysis Tools (PAT) is a set of utilities that was developed by the Parasoft Corporation and has been ported to the iPSC system. The PAT utilities are post-processing tools that gather performance data at runtime and then output the data to disk when the application terminates. PAT provides a powerful set of analysis tools that convert the performance data to graphical and tabular forms that can be analyzed interactively on a standard Sun workstation.

Profiling Parallel Programs

The most obvious goal of parallel computing is the acceleration of algorithms that execute too slowly on conventional machines. While other goals, such as fault tolerance, are also important most applications are ported to parallel machines with one aim in mind — running rings around expensive supercomputers.

Since this goal holds such a central position in the realm of parallel processing it is important that users be able to rapidly and effectively analyze their algorithms' performance. Even in cases where absolute speed is not the most important factor, it is crucial to have a thorough understanding of an algorithm to see the strengths and weaknesses of a particular parallelization scheme. In this way it may be possible to see where bottlenecks occur and to devise alternative algorithms to avoid such problems.

The profiling of parallel programs is not as straightforward as on sequential computers. For a sequential computer the only really important piece of information is "How long am I spending in routine XXX?" and "Which routines should I speed up in order to accelerate the code most?". The style of profiling most often used in this context is a simple printout of elapsed times in each routine and, possibly, the number of times each was called and by whom. Armed with this information one can attempt to speed-up certain areas of the algorithm which are known to be heavily used. Alternatively, of course, one might be able to see that there are no real bottlenecks and that, therefore, the code is running as fast as it possibly can on the given hardware.

Parallel programs are trickier because more factors arise which affect their performance. The most obvious, for a message-passing architecture, is the amount of time spent sending and receiving messages. One of the most quoted parameters of such machines is the "Efficiency" or "Overhead" which basically expresses how many times faster *N* processors are than 1. Once the parameter is known one might want to break it down further into times during which I/O is occurring, times when intermediate results are being accumulated globally and times when processors are communicating boundary values, for example.

A final factor which may be extremely important in parallel algorithm development is load balance. This sort of problem can take many forms but is most clearly characterized by differences in execution speed of the different nodes in the parallel machine. Sometimes this can be caused because the workload is not evenly distributed between processors resulting in one node working exceptionally hard and correspondingly slowly. Many times this will slow down the other processors who are waiting to communicate with the slow node degrading the performance of the machine as a whole. Other problems may be more algorithmic in nature — a particular scheme for parallelizing a program may have some inherent defects which make some processors run more slowly than others. Detecting and correcting this sort of problem requires an ability to observe activities in several processors simultaneously at many levels of detail.

PAT Features

The Parallel Performance Analysis Tools utilities are designed to analyze the three major areas that affect parallel computer performance. The three tools each serve one of the following functions:

- The execution profiler tool, `xtool`, monitors time spent in individual routines.
- The communication profiler tool, `ctool`, assesses time spent in communication and I/O.
- The event profiler tool, `etool`, shows the interactions between processors and allows user-specified events to be monitored.

Each profiling tool is kept separate so the user is free to concentrate on particular problems as they arise and can be selective in the amount of information available — it is one thing to provide detailed analysis tools but quite another to present the user with 200M bytes of data to analyze in order to understand the problems. As a result the majority of the tools have graphical interfaces. Menu driven utilities allow the presentation of accumulated data in simple graphical form under the complete control of the user. Optionally, data can be presented in tabular, hardcopy graphical, or hardcopy tabular form for more detailed analysis.

As mentioned in the previous paragraph one has to be rather selective in the data acquired for analysis. One of the more pressing needs for this ability is the fact that performance tools which significantly alter the execution of the target program are of little use. Essentially one ends up analyzing the profiling system rather than the user application! For this reason the tools described in this manual are of the *post mortem* type — that is, data is accumulated during the execution of the user program and then analyzed off-line, after execution has completed.

I/O in parallel computing systems is notoriously slow — especially when compared to the high computing power of typical machines. Even worse, I/O in one processor causes other processors to be affected in routing messages to the outside world. As a result, even limited amounts of real-time I/O can cause significant modifications in program execution which completely invalidate the profiling procedure.

Displaying profiling data "real-time" looks quite attractive but the amount of information rapidly overwhelms the human mind — particularly when more than a handful of processors are involved. Due to the constraints mentioned above it is difficult to present enough context to render a wildly varying display meaningful. Furthermore, saving the data on some physical medium for later use introduces a sequential bottleneck that affects all processors.

The PAT utilities support the following environments:

- SunView
- X Windows
- Hardcopy is supported in Postscript form.

Profiling Methods

The iPSC system uses three basic methods to collect PAT profiling data:

- Compiler switches (automatic/manual)
- Environmental variables (automatic/manual)
- Programmatic interface (manual only)

Refer to Chapter 2 for more information on each of these methods of gathering performance data.

Profiling Example Program

The example screens and listings use a Fortran application program that is supplied with your iPSC/860 system software. You can reproduce the examples used in this manual by copying the source files to your own directory, and then compiling them using the appropriate compiler switches.

The example is a Fast Fourier Transform (FFT) program located at */usr/ips/src/examples/fortran/2dfft* on the iPSC/860 SRM. Copy the entire *2dfft* directory contents over to a duplicate path on your workstation or under your own directory. The example files include a *README* file with important information.

Use the procedures in Chapter 2 to apply the appropriate compiler switches while compiling the example program.

Gathering Performance Data 2

Introduction

The PAT performance data is collected through the use of PAT C or Fortran system calls that are compiled with the application program. These system calls can be placed in the application code automatically (if PAT switches are specified at compile/link time), or they can be added manually to the application source code.

The usual method of gathering profiling data uses switches with the C and Fortran compilers to turn one or more of the profilers on or off for the code being compiled. With this method, appropriate PAT procedure calls are automatically inserted at the start and exit of every procedure/subroutine.

The PAT environment variables can be set on or off after the code has been compiled. Using this method, the code retains the PAT calls introduced with compiler switches, but you interactively control whether performance data is gathered or not.

The manual method of gathering profiling and tracing data requires specific changes to the application source code before the code is compiled. Because of the additional work required for the manual method, you should only use this method if it is necessary to isolate specific performance problems.

Data Gathering Methods

The methods you use to gather performance data will vary depending on the kinds of data you want to examine, the level of detail you want, and the amount of intrusion into your application you want to allow. The remaining sections of this chapter provide detailed information on the data gathering methods supported by the iPSC system for the PAT utilities.

Using Compiler Switches to Profile/Trace

One of the easiest ways to gather profile or trace data on an application is to use the **-Mperf** switch when compiling or linking the application. This switch is available with the iPSC/860 C and Fortran compiler drivers. The **-Mperf** switch is similar to the **-p** compiler switch, except the resulting data files must be processed by the PAT utilities (i.e., they are not compatible with the **prof860** tool). The syntax and descriptions for the switches and arguments appear in the following sections.

Syntax

-Mperf [={prof | comm | event} [={auto | manual}] [, ...] | {auto | manual}]

Arguments

The **-Mperf** switch by itself turns on all PAT instrumentation code. The optional arguments turn on individual performance monitoring code.

prof This argument turns on PAT execution profiling. This switch is the same as the **-p** compiler switch, except the PAT instrumentation code generates only one profiling output file (default file is *exprof.prx*) instead of the output file that is generated for each iPSC node with the **-p** switch. The **prof** switch is both a compiler switch and a link directive. The execution profile output must be displayed with the PAT **xtool** utility, and not the **prof860** utility.

comm This argument turns on PAT communication tracing. When the program terminates, the communication performance data is written (by default) to the *exprof.prc* file. The **comm** switch is a compiler switch and a link directive that also turns on profiling of user routines. The communication performance data can only be displayed with the PAT **ctool** utility.

event This argument turns on PAT event tracing. When the program terminates, the event driven performance data is written (by default) to the *exprof.pre* file. If the **event** option is used during the compile/link phase, the module being compiled will have every procedure/subroutine traced.

After each of the previous arguments, you can specify whether the instrumentation code for that function is performed automatically for the entire application, or manually. The default condition is that each function gathers data automatically.

auto Use the **auto** invocation option to specify that performance monitoring code is to be invoked automatically on the application. This is the default state that applies if there is no invocation option following the prof, comm, or event arguments.

manual Use the **manual** invocation option to specify that performance monitoring code is to be manually invoked using PAT programmatic procedure calls. When the **manual** invocation option is specified, profiling or tracing will not occur unless the appropriate procedure calls (for example, **xprof_int**(), **cprof_off**(), or **eprof_on**()) are encountered in the source code during the compile.

Examples

In the following example, the Fortran linker directives invoke event tracing automatically for the "ftest" program.

```
if77 -Mperf=event -o ftest *.o -node
```

In the next example, C compiler driver directives turn on execution profiling automatically for the "ctest" program.

```
icc -Mperf=prof=auto *.c -o ctest -node
```

In the final example, communications and event performance monitoring options are turned on manually and environment variables are then used to initiate data gathering.

```
icc -Mperf=comm=manual,event=manual ctest -node
setenv EXPROF_SWITCHES "ce"
```

Advantages

Using compiler driver switches is the easiest, most direct way of gathering performance data for the PAT utilities. Using the compiler driver **-Mperf** options, you have the ability to turn on data gathering for any or all of the PAT utilities.

In addition to the default **auto** invocation option of data gathering, you can specify **manual** invocation for any of the **-Mperf** options. The **manual** invocation allows you to further modify the means of data gathering using the programmatic interface. Refer to the section named "Advantages" on page 2-8 for additional advantages related to the **manual** invocation for the **-Mperf** options.

Disadvantages

The disadvantages to the use of compiler driver **-Mperf** options are related more to the use of the default (i.e., **auto**) invocation instead of **manual** invocation. While the automatic method is easy to implement, it does have the following disadvantages:

- One disadvantage is that it is relatively broadbanded, i.e., the entire procedure/subroutine is profiled instead of a possibly smaller code segment that may be of interest.
- Another disadvantage to the automatic method is that you have no programmatic control over the information being gathered. For example, you may only be interested in profiling data for iterations 110-125 of a 1000 iteration loop, or data gathered during the fifth hour of processing for a program that takes 10 hours to complete. With the automatic method you cannot control the amount or type of information that is gathered.
- One final disadvantage to the automatic method of gathering profiling data is that it may have an undesired impact on program performance. Because each procedure/subroutine has at least one entry and one exit system call added to it, programs with many procedures/subroutines can experience a significant growth in processing time and thus a degradation of program performance.

Using Environment Variables to Profile/Trace

You can use environment variables to control the gathering of performance data for any of the PAT utilities. Two environment variables (**EXPROF_SWITCHES** and **EXPROF_OPTS**) specifically affect the data gathering environment of the PAT utilities. The following sections provide detailed information on these environment variables.

The EXPROF_SWITCHES Variable

The **EXPROF_SWITCHES** environment variable is set using the **setenv** shell command. Refer to the appropriate UNIX documentation for detailed reference information on csh environment variables.

Syntax

setenv EXPROF_SWITCHES [" [c] [e] [x] "]

The syntax for using the **EXPROF_SWITCHES** environment variable allows you to enable or disable any or all of the PAT communication (**c**), event (**e**), or execution (**x**) profile/trace tools. Entering ***setenv EXPROF_SWITCHES*** without specifying any switches places all PAT utilities under the **manual** invocation option. Setting any of the **EXPROF_SWITCHES** (e.g., ***setenv EXPROF_SWITCHES "c"***) forces the corresponding tool to go to the **auto** invocation option.

Once the **EXPROF_SWITCHES** environment variable exists, data gathering will ignore any compiler switch (**-Mperf**) settings and will follow the settings of this environment variable.

NOTE

Some UNIX versions support an **unsetenv** command. This command allows you to quit using environment variables and resume use of the **-Mperf** compiler driver switches. If your version of UNIX does not support the **unsetenv** command, you will need to reinvoke your shell in order to resume use of the **-Mperf** compiler driver switches. Most Sun versions of UNIX support the **unsetenv** command, but the UNIX version on the SRM does not.

Examples

In the following example, the **EXPROF_SWITCHES** environment variable is invoked without any switches. This places all PAT utilities under the **manual** invocation option.

```
% setenv EXPROF_SWITCHES
```

In the following example, the c and e environment variable switches are set to put the communication and event tracing tools under the **auto** invocation option.

```
% setenv EXPROF_SWITCHES "ce"
```

In the following example, the **setenv** command is invoked without options to show the current environment variable settings.

```
% setenv
HOME=/home/daleo
SHELL=/bin/csh
TERM=sun-cmd
   .
   .
   .
WINDOW_GFX=/dev/win17
EXPROF_SWITCHES=
%
```

Advantages

The main advantage to using the **EXPRO_SWITCHES** environment variable is that it allows you to profile or not profile a program at runtime based on environment variable settings. This means that you don't need to decide at compile time whether or not you will be profiling the program, though you must compile with the **-Mperf** switch.

NOTE

If profiling is disabled with the **EXPROF_SWITCHES** environment variable, the profiling code will still cause some overhead in the compiled program.

Disadvantages

The main disadvantage to using the **EXPROF_SWITCHES** environment variable is that once you start using this environment variable to control data gathering (i.e., **EXPROF_SWITCHES** exists) you must continue using it until you log out. If your system supports an **unsetenv** command, there is no disadvantage to this approach. This is a problem with some UNIX versions (such as the UNIX version loaded on the SRM), but not with the PAT utilities.

The EXPROF_OPTS Variable

The **EXPROF_OPTS** environment variable is set using the **setenv** shell command. Refer to the *UNIX System V User's Reference Manual* or the on-line man pages for detailed reference information on *csh* environment variables.

Syntax

setenv EXPROF_OPTS "{ elogs | elabs } : *new_value* [; ...] "

The syntax for using the **EXPROF_OPTS** environment variable allows you to set the maximum number of log entries allowed in the default log file. This file is used by the PAT **etool** utility. The default maximum is 1024 log entries and 10 label entries.

This environment variable may be used instead of, or in conjunction with the PAT programmatic interface. Any program that runs more than a few minutes is likely to generate more than the default maximum number of log entries in the log file. Using the **EXPROF_OPTS** environment variable, you can increase the maximum to ensure full coverage for your program. Alternatively, you can use programmatic control to focus on a small area of the program, and also use the **EXPROF_OPTS** environment variable to fine tune the amount of data collected. The **EXPROF_OPTS** environment variable will accept any valid decimal integer as a *new_value*, so a practical maximum is the amount of available memory space.

Examples

In the following example, the **EXPROF_OPTS** environment variable is invoked to change the maximum number of log entries to 15,000:

```
% setenv EXPROF_OPTS "elogs:15000"
```

In the following example, both the log and label values are changed using the same **EXPROF_OPTS** environment variable entry:

```
% setenv EXPROF_OPTS "elogs:15000;elabs:1200"
```

Advantages

The main advantage to using the **EXPROF_OPTS** environment variable is that it allows you to increase the size of the log file to make sure all data is captured without resorting to the programmatic interface. This means that you don't need to add code to the source files to make sure your performance data is captured.

Disadvantages

The only real disadvantage to using the **EXPROF_OPTS** environment variable is that it becomes possible to specify a maximum number of entries that exceeds the available memory or file space. If the available file space is exceeded, system operation becomes unpredictable.

Using the Programmatic Interface to Profile/Trace

The programmatic interface method of data gathering requires you to imbed PAT system calls in the application source code, and then compile and link the application using the **-Mperf** compiler switch and **manual** invocation. You can alternately use the **-Mperf** compiler switch without the **manual** option, and then use the **EXPROF_SWITCHES** environment variable to gather data using the programmatic interface. Entering ***setenv EXPROF_SWITCHES*** without specifying any switches places data gathering conditions under control of the environment variables and places all PAT utilities under the **manual** invocation option.

Advantages

The programmatic interface method of collecting data is not as convenient as the automatic method, but it allows you much more control over the amount and type of data that is collected. Using the manual method, you have full programmatic control over the gathering of profiling information. This means, for example, that you could turn on profiling for only a few procedures/subroutines. You could also add code to turn on profiling for a range of iterations or for a certain time period.

Disadvantages

Programmers using the manual method of collecting profiling data must have a good knowledge of the code that PAT is profiling. One way of gaining this knowledge is to first use the automatic method to get a feel for the general performance for the overall program, and then use the manual method to examine specific areas in more detail.

Using Execution Profiling 3

Introduction

This section describes what might alternatively be called "sequential profiling" since the utilities described are those most familiar in the context of sequential programming. The goal is to analyze the time spent in the various subroutines and functions that make up an application. At its simplest this merely involves tabulating the time spent in each routine. At a second level of detail the user is then able to analyze the source and/or assembly code of their application to better identify hotspots. This allows the user to immediately focus attention on the most time consuming areas which would benefit most by improvement.

Using the data accumulated the user can evaluate, on a node by node basis, the time spent in each function and also that spent "idle" waiting on some external condition such as communication with another processor.

Profiling Overview

This system uses two distinct statistical methods.

- The sampling method regularly samples the processor execution state. Every 10 milliseconds a system routine runs that looks at the current instruction being executed in the user application and increments a counter noting that this particular memory address was in use. In this way one builds up a histogram of the frequencies of hits in various areas of the program and hence the amount of time spent in particular routines.

- The counting method actually counts the number of times the profiled procedures are executed. When execution profiling is selected, two procedures (**__rouent()** and **__rouret()**) are inserted by the compiler at the start (**__rouent()**) and end (**__rouret()**) of each compiled procedure. As the program executes, the **__rouent()** procedure counts the number of times the procedure was executed.

Obviously the sampling technique is not foolproof but if the application executes for a sufficiently long time to collect a reasonable number of samples then one can be fairly confident that the results are representative of the true behavior of the algorithm. (If the code only takes a few milliseconds to run, who cares anyway?). Among the obvious defects in this approach are possibly sick behavior if the program cycles at a similar rate to the routine which logs events. In this case one might always catch the program in a routine that actually doesn't take very long leading to incorrect conclusions — not very likely but possible. A more limiting problem is that it requires a lot of memory to make up the program histogram.

Despite these deficiencies the sampling style of profiling is standard in most sequential computing environments and is part of the profiling system. It has the advantage that it's operation is mostly automatic, and few changes need be made to an existing program in order to profile it. An alternative system, the event driven profiler, is discussed in Chapter 5. The event driven profiler requires more work by the user but is probably quantitatively more reliable in delicate situations.

Automatic Operation

Inserting the **__rouent()** and **__rouret()** functions into your source code and linking to the profiling libraries is accomplished using the **-Mperf** compiler switch. The program is automatically profiled when you load and run the program. Upon program termination the *exprof.prx* file is written automatically by the performance monitoring code.

To help in setting up the various profiling activities, all the PAT tools attempt to automatically initialize themselves to the greatest extent possible. In the case of the execution profiler the ideal situation is one in which the PAT tool is supplied with enough information to make a sensible call to **xprof_init()**. By default, the entire user program is profiled with a scale of one instruction per bin. The default case uses the **auto** invocation option and the **EXPROF_OPTS** environment variable disabled. In this case the following actions would be performed by the PAT tool:

- During program start-up the system call **xprof_inq()** is made. If the appropriate **EXPROF_SWITCHES** environment variable switch is enabled, the execution profiler calls **xprof_init()** to map a default sized buffer to the code of the application being executed and the **xprof_on()** system call starts profiling.

- During the system termination code the **xprof_inq()** function is again used to determine whether or not profiling data should be dumped to disk. The default output file is *exprof.prx* in your current working directory.

Environment Variables

The **EXPROF_SWITCHES** environment variable overrides the **auto/manual** options of the **-Mperf** compiler switch. Refer to "Using Environment Variables to Profile/Trace" on page 2-5 for a detailed description of the actions that can take place when the **EXPROF_SWITCHES** environment variable is used. To summarize, there are several possible results from using the **EXPROF_SWITCHES** environment variable to control execution profiling at execution time, as follows:

-Mperf=prof or **-Mperf=prof=auto**
: If the **EXPROF_SWITCHES** environment variable does not exist, automatic profiling will be performed. If the **EXPROF_SWITCHES** environment variable is set to '**x**', automatic profiling will be performed. If the **EXPROF_SWITCHES** environment variable does not have '**x**' set, manual profiling is in effect.

-Mperf=prof=manual
: If the **EXPROF_SWITCHES** environment variable does not exist, manual profiling will be performed. If the **EXPROF_SWITCHES** environment variable is set to '**x**', automatic profiling will be performed. If the **EXPROF_SWITCHES** environment variable does not have '**x**' set, manual profiling is in effect.

You can determine the current setting of the **EXPROF_SWITCHES** environment variable (or its existence) by typing ***setenv*** from the shell.

Manual Operation

The sampling method uses a number of functions that initialize, bound, and output data on the profiled procedures. The two most elementary profiling functions are **xprof_on()** and **xprof_off()** which are used to enable and disable the profiling system, respectively. This allows the user to maintain fine control over the regions of the application that are actually analyzed — for example it may be sensible to turn off the profiler if one section of code is known to be much more heavily used than any other since one would otherwise be swamped by a vast amount of information about something already understood.

The heart of the profiling system is provided by the **xprof_init()** function. This tool is based on the standard UNIX profiling utility **profil()** and shares the same arguments, as follows:

xprof_init(*buffer, buflen, start_addr, scale***);**

The first argument (*buffer*) is a pointer to a buffer into which the profiling data will be dumped.

The length of the buffer (*buflen*) is the second argument.

The *start_addr* argument specifies the lowest address to be considered in profiling the program. This is most easily discovered by searching through the "program map" files that are often produced by compilers.

The final argument, *scale*, specifies how many bytes to "bin" together. When profiling, the system actually builds a histogram of memory locations and the *scale* argument specifies how wide the histogram bins should be. The *scale* is interpreted as an unsigned, 16-bit, fixed-point fraction with a binary point at the left. The fraction represented by the *scale* argument has a maximum value of 0xFFFF (nearly 1.0) which maps two bytes to a histogram bin. The minimum *scale* value of 0x2 maps 65,536 bytes to each histogram bin. The following list shows how several *scale* values are interpreted:

`scale = 0xFFFF`	Maps each pair of bytes into separate bins.
`scale = 0x8000`	Maps four bytes into a single bin in the data buffer.
`scale = 0x4000`	Maps two instruction words (eight bytes) together into a single histogram bin.
`scale = 0x2`	Maps all instruction words into a single bin in the data buffer, thus producing a non-interrupting core clock.

A mapping function is applied to the program counter discovered by the iPSC system when the *scale* value is applied. First *start* is subtracted and then the result is multiplied by *scale* and divided by 0x10000. The complete mapping function is as follows:

```
bin_number = (PC - start)*scale/0x10000
```

The **xprof_init()**function does not actually turn the profiling system on; to do this you must use the **xprof_on()** function. A sample code to use this profiling subsystem is as follows:

```
/*
 * Sample program demonstrating the setup of the
 * execution profiling system
 */
 #define SCALE                (0x2000)
 int profbuf[2048];
 extern int myfunc();

 main()
 {
 /* Enable profiling system, and turn it on. */

     xprof_init(profbuf, sizeof(profbuf), myfunc, SCALE);
     xprof_on();

 /* Algorithm phase 1., profiler running ..... */

         ..........

 /* Algorithm phase 2., turn profiler off .... */

     xprof_off();
          ..........
```

In this example we choose to profile at a resolution of 16 bytes, selected by the value of the `SCALE` macro, with a 2K byte buffer. Since the individual bins are 2 bytes this means that we have a range of 16K bytes in total (1024 bins, each with a resolution of 16 memory addresses). The starting point is selected as the beginning of the function `myfunc`. Since 16K bytes is not an awful lot of space for a large application we might expect some misses — at times the profiler will detect that the program is executing at addresses outside the range we have covered. These cases are not reported or displayed.

This interface is not especially easy to use because it requires either good guesswork in picking the profiling range or else some time spent looking through a memory map generated by the compiler or linker.

A particularly useful function is **xprof_inq()**. This function looks for the environment variable called **EXPROF_SWITCHES** and checks for the existence of the character 'x' in the defined string. If defined and if 'x' = 1, the **xprof_inq()** function returns 1. If **EXPROF_SWITCHES** is defined and if 'x' = 0, the **xprof_inq()** function returns 0. Since the node program is usually supplied with the same environment variables as were defined in the host operating system you can use this function to control the profiler at runtime rather than hard-coding the calls to **xprof_on()** that enable the system. For example the previous code segment could be changed to the following:

```
/*
 * Sample program demonstrating the setup of the
 * execution profiling system.
 * Use the inquiry function to support runtime
 * operation.
 */
 #define SCALE                (0x2000)
 int profbuf[2048];
 extern int myfunc();

 main()
 {
 /* Enable profiling system, and turn it on. */

    if(xprof_inq()) {
         xprof_init(profbuf,sizeof(profbuf), myfunc,SCALE);
         xprof_on();
    }


/* Algorithm phase 1., profiler running ..... */

         ..........

 /* Algorithm phase 2., turn profiler off .... */

     if(xprof_inq()) xprof_off();
          ..........
```

This code is superior to that shown previously since it does not invoke the profiler unless told to do so by the presence of the appropriate character in the **EXPROF_SWITCHES** environment variable. This allows the program to be profiled or not according to runtime rather than compile time settings. For details explaining how to set environment variables for your system consult the **shell** or **csh** documentation (UNIX systems), the **sh()** or **csh()** online man pages, or the technical reference manual applying to your operating system (if not UNIX).

The function **xprof_dmp()** is used to output the collected profiling information to disk at any time. This function must be called "loosely synchronously" which means that all nodes must call it together. Furthermore the call will not return in any node until it has been called in all others. Usually this code is only required once, at the end of the user program. Since this case arises so often it is performed automatically by the system exit code. As a result most user programs do not require explicit calls to this routine at all.

This function has a single argument — the name of the disk file to be created. This provides a useful capability for profiling separate stages of an application independently. Consider the following sample code:

```
/*
 * Demonstrating the use of the "dump" functions to
 * make multiple output files containing profiles of
 * different phases of an application.
 */
main()
{
 /* Application phase 1. */

     xprof_init( ... );        /* Initialize profiler */
     xprof_on();         /* ...and turn it on */
      ...

     xprof_dmp("phase1.prx");        /* Write out prof data */

 /* Application phase 2. (Don't profile since xprof_dmp
  * turned system off.)
  */
      ...

 /* Application phase 3., reset profiler, and turn on */

     xprof_init( ... );
     xprof_on();
      ...

     xprof_dmp("phase3.prx");
               /* Write out prof data again */

         ...
```

In this case it has been decided that two distinct phases of the application are to be profiled separately. Two sets of profiling commands are used and two calls to **xprof_dmp()** made to write out the accumulated data. Note that a second call to **xprof_init()** and **xprof_on()** must be made to restart the profiler since the call to **xprof_dmp()** zeros all internal profiling information.

As mentioned previously the **xprof_inq** function may be called to inquire about the runtime status of the **EXPROF_SWITCHES** environment variable. If the environment variable does not exist then the **-Mperf=prof** or **-Mperf=prof=manual** switches are used by **xprof_inq()**. Refer to "Using Environment Variables to Profile/Trace" on page 2-5 for more information on the **EXPROF_SWITCHES** environment variable.

Analyzing the Execution Profile — xtool

After a file containing profiling data has been collected it is analyzed with the xtool command. Two arguments must be supplied; the name of the program which is to be profiled and the name of the file containing the written data. If this latter is *exprof.prx* then you can omit this last argument. By default `xtool` uses a graphical interface to display its output but this can be suppressed with the **-p** switch. The simplest way to analyze execution profile data is, therefore, with a command such as:

```
xtool -p program
```

After reading in the symbol table information for your program, the output from this process will consist of several tables. Figure 3-1 shows the output you get when invoking **xtool** with the **-p** switch.

If the procedure was compiled with the **-Mperf=prof** compiler driver switch, there may also be a field indicating the number of calls to each function and the mean time of execution for each. If present, this data will also be displayed in the `# calls` area in Figure 3-1.

The statistical nature of the profiler can again be seen in the +/- entries in the middle column of the display. Because of the finite width of the histogram bins it is not always possible to associate a number of hits with a single routine — the bin may overlap two (or more) functions. In this case **xtool** assigns a portion of the hit count to each of the affected routines and indicates by the +/- notation the possible error.The information provided by this utility is described below.

1. Node identifier. A separate table is presented for the data from each node and is identified by its processor number.

2. Subroutine analysis. For each node a table is presented of the twenty busiest routines based on the number of profiling hits in that function. Occasionally this information might also show up explicit defects in the parallelization scheme selected. If invoked with the **-a** switch, **xtool** shows all routines, not just the busiest twenty.

While the tabular style of output is often the most convenient way of keeping book-keeping checks on a program being optimized, a graphical style usually gives the best feel for the behavior of the program. It can also show detailed information which is not easily accessible from the tabular output.

To invoke the graphical interface to **xtool** one simply omits the **-p** switch from the command line shown earlier. This will start up the display on the *normal display device* for your system. Normally this will be the console of the machine at which you are working. So, for example, the command:

```
xtool program
```

```
Node      0
==========
    Busy : Idle = 1668 : 0 (100.00 )
    misses = 0

        Subroutine      |      Samples      |  Percent  |   # calls
    --------------------|-------------------|-----------|------------
    __flick             |     952           |   57.07   |
    ccopy02             |     176           |   10.55   |
    ccopy_loop100       |     143           |    8.57   |
    _loopi8             |      64           |    3.84   |
    _loopf8             |      55           |    3.30   |
    dcopy_loop10        |      35           |    2.10   |
    _loop0_             |      25 +/- 1     |    1.50   |
    _kcomb_gsf_         |      24           |    1.44   |
    _kcomb_gsi_         |      20           |    1.20   |
    _begin_             |      17           |    1.02   |
    _loopf_big          |      13           |    0.78   |
    _sqrt               |      12           |    0.72   |
    _kcfft1d_           |      12           |    0.72   |
    _loop_big           |      10           |    0.60   |
    _loopf1             |       9           |    0.54   |
    _loopi1             |       7           |    0.42   |
    _loopi4             |       7           |    0.42   |
    _short_loop         |       5           |    0.30   |
    _begin_             |       5 +/- 1     |    0.30   |
    __mrecv             |       4           |    0.24   |
```

Figure 3-1. Tabular Output for xtool

will start up the graphical interface under SunView on a Sun workstation. Sometimes you will wish to use other devices. The X Window System is a common alternative on many systems. To use such a device you perform all the normal setup for that system (such as enabling access control and setting the **DISPLAY** environment variable for X) and then supply a **-T** switch to **xtool**. To use the X Window System, for example, change the above command line to:

```
xtool -TX program
```

The availability of alternate graphics devices varies from system to system as do the key-letters given with the **-T** switch. Consult Appendix A of this manual for more details.

Once you have successfully given the command to start up the graphical interface **xtool** will read the symbol table for your program, as before, and then display its initial menu. Figure 3-2 shows the initial display presented when **xtool** is first invoked

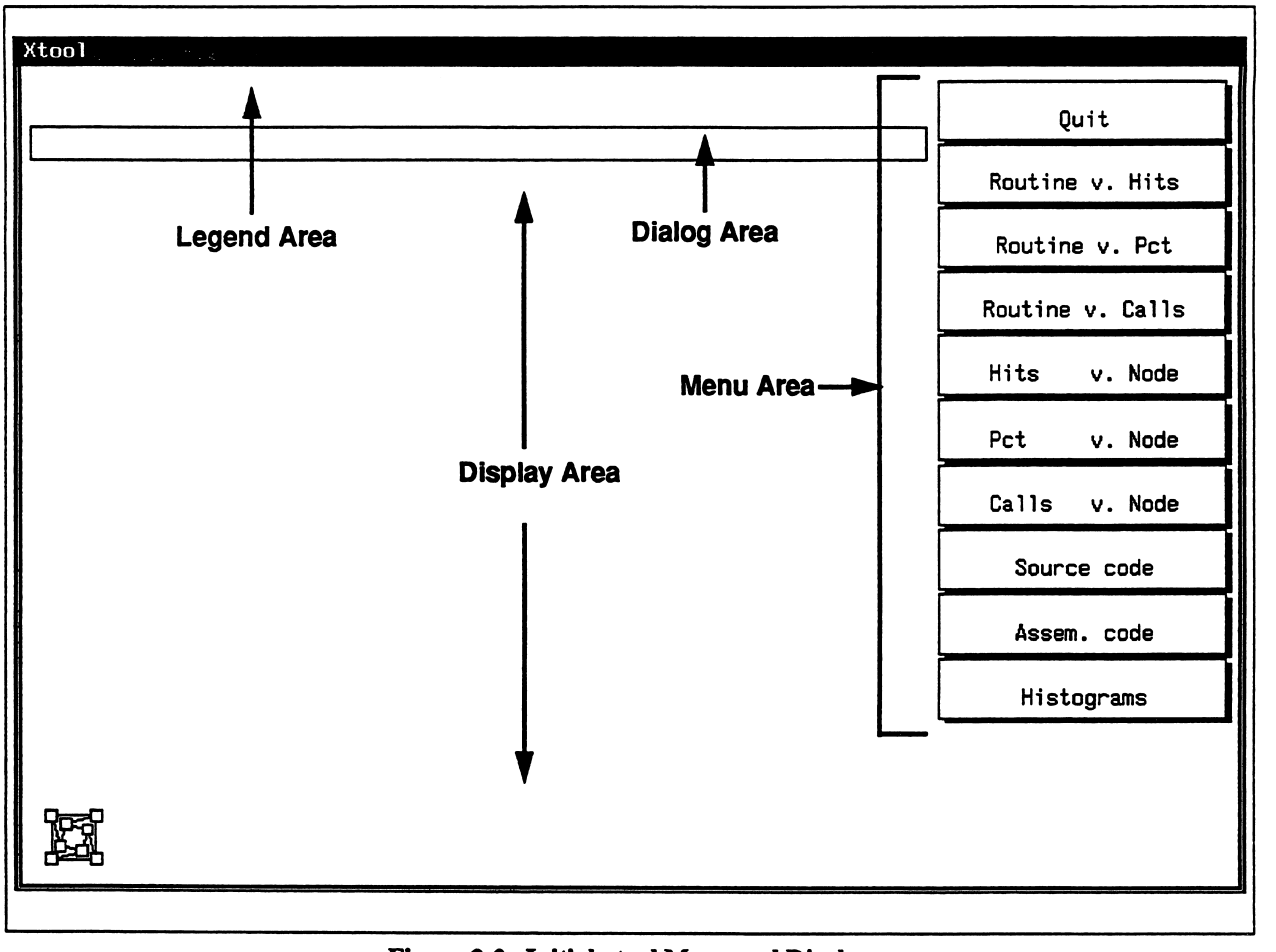

Figure 3-2. Initial xtool Menu and Display

The various sections of this display are used for different purposes while using **xtool** as follows:

Menu Area The various possible selections at any given point are shown in menu form on the right hand edge of the display. Selecting a particular option requires positioning the mouse cursor over the appropriate box and clicking once. If your system doesn't have a mouse the keyboard's arrow keys should be able to move the cursor around. In this case the <HOME> key is used to toggle the speed of cursor movement and "clicking" is achieved by striking any alphanumeric key.

Display Area The largest part of the display is used to show the various graphs and listings that make up the execution profile. Occasionally you will be asked to select some feature from this display to indicate a region of interest for further processing. To do this simply click in the manner described above over some point in this area.

Legend Area This region is used to provide labels and legends for the various graphs.

Dialog Area This area is used to provide simple help and keyboard style interactions with the user. If the commands you invoke from the menu require more information you will sometimes be prompted to enter text here. Enter the string just as you would normally using the keyboard and backspace/delete keys to correct errors. Once the string you wish to enter appears in the dialog area use the **<Enter>** or **<Return>** keys to go back to normal menu processing.

It is possible that the menus may appear garbled or even illegible on your display device. If this is the case you can always terminate **xtool** by selecting the top entry from the main menu — even if you can't read the **Quit** label. Alternatively, of course, you can just abort **xtool** with the usual command sequence for your system. Correcting this problem usually requires specifying an additional switch, **-M**, when starting up. By default **xtool** tries to use three-dimensional push-button style menus. Using the command:

```
xtool -M0 program
```

will make less attractive menus that should be viable on all monitors.

The selections offered by the main menu can be broken down into three groups as follows

Hits vs. Node, Percentage vs. Node, Calls vs. Node
These routines basically display the information shown in the tabular output of Figure 3-1 in graphical form. The first three routines make horizontal bar graphs suitable for small numbers of processors while the last three make more normal vertical line graphs more appropriate for large numbers of processors.

Source Code, Assembly Code
These options allow the display of source code and/or assembly code along with the profile data. This allows the execution of the program to be monitored even at the level of individual machine instructions and is useful for identifying crucial bottlenecks.

Histograms
This options allows the display of the raw data accumulated by the execution profiler. In this form it is easiest to identify important program features that affect the profile and then move on to the source or assembly code options.

Controlling the first set of options is fairly straightforward. Selecting **Routine vs. Pct**, for example yields a graph like the one shown in Figure 3-3. As can be seen, the various popular routines are displayed along the vertical axis and the percentage of the total time spent in each is shown by a horizontal bar for each node. As might well be imagined this type of display can become quite difficult to read when many nodes are shown since the bars for each become extremely thin. By default **xtool** only displays the contributions from the first four nodes found in the profile data.

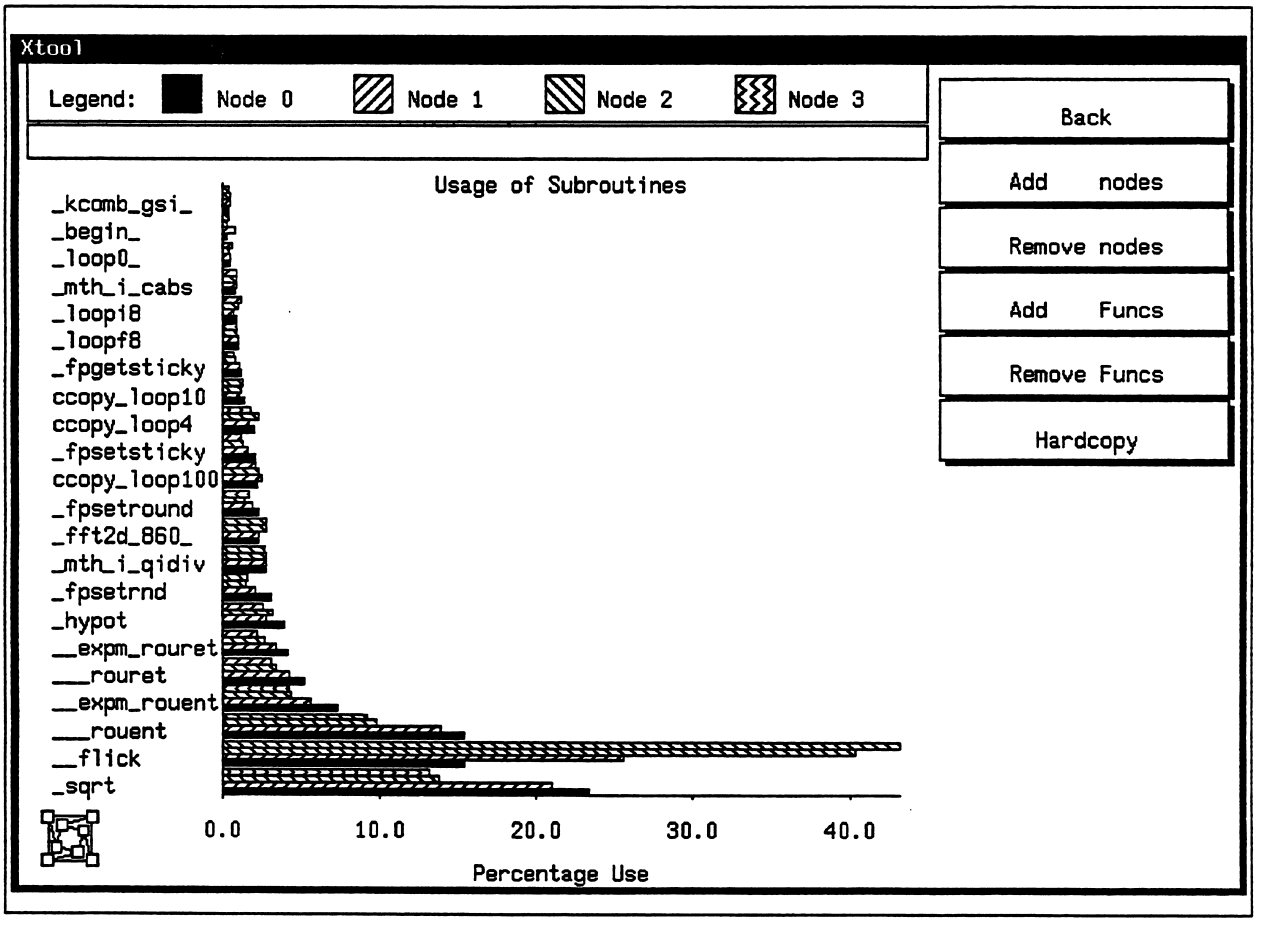

Figure 3-3. "Routine vs. Percentage" Execution Profile

If you wish to look at data from other nodes the **Add Nodes** menu option can be used. Clicking on this item will present you with a menu like that shown in Figure 3-4.

Clicking on a processor number will toggle a check mark by that number. When you have checked the numbers of the nodes whose information you wish to display select **Done** from the top of the menu to add the data. If you mark a node by accident you can remove its check-mark by clicking on it again. Note that options are available to check groups of nodes according to properties often of interest in parallel processing. There may also be multiple pages to this menu if you are using more processors than can fit on a single display. In this case the **Next Page/Previous Page** options can be used to move through the list.

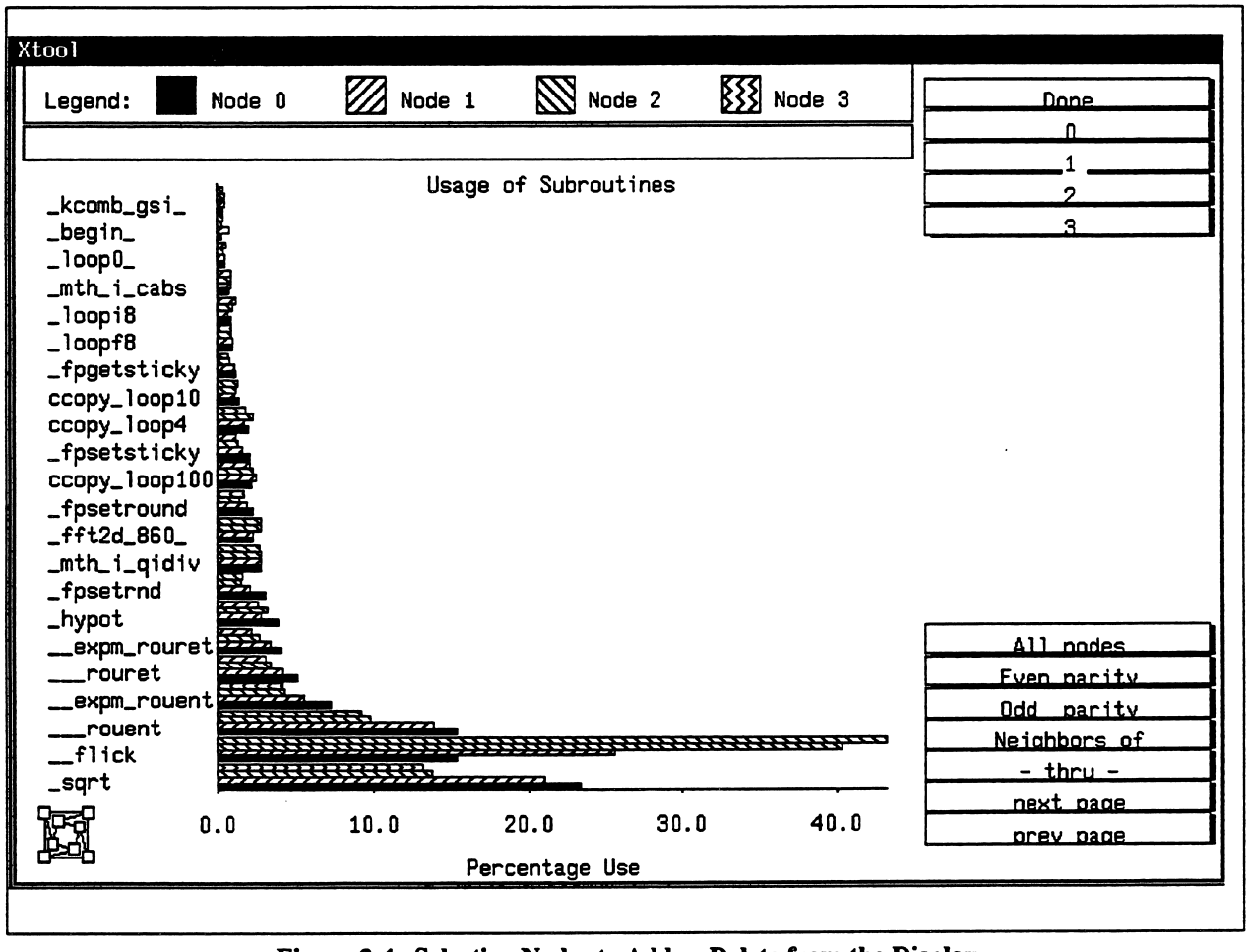

Figure 3-4. Selecting Nodes to Add or Delete from the Display

Several other menu options are common to all the basic information displays as shown in Figure 3-3 and Figure 3-3.

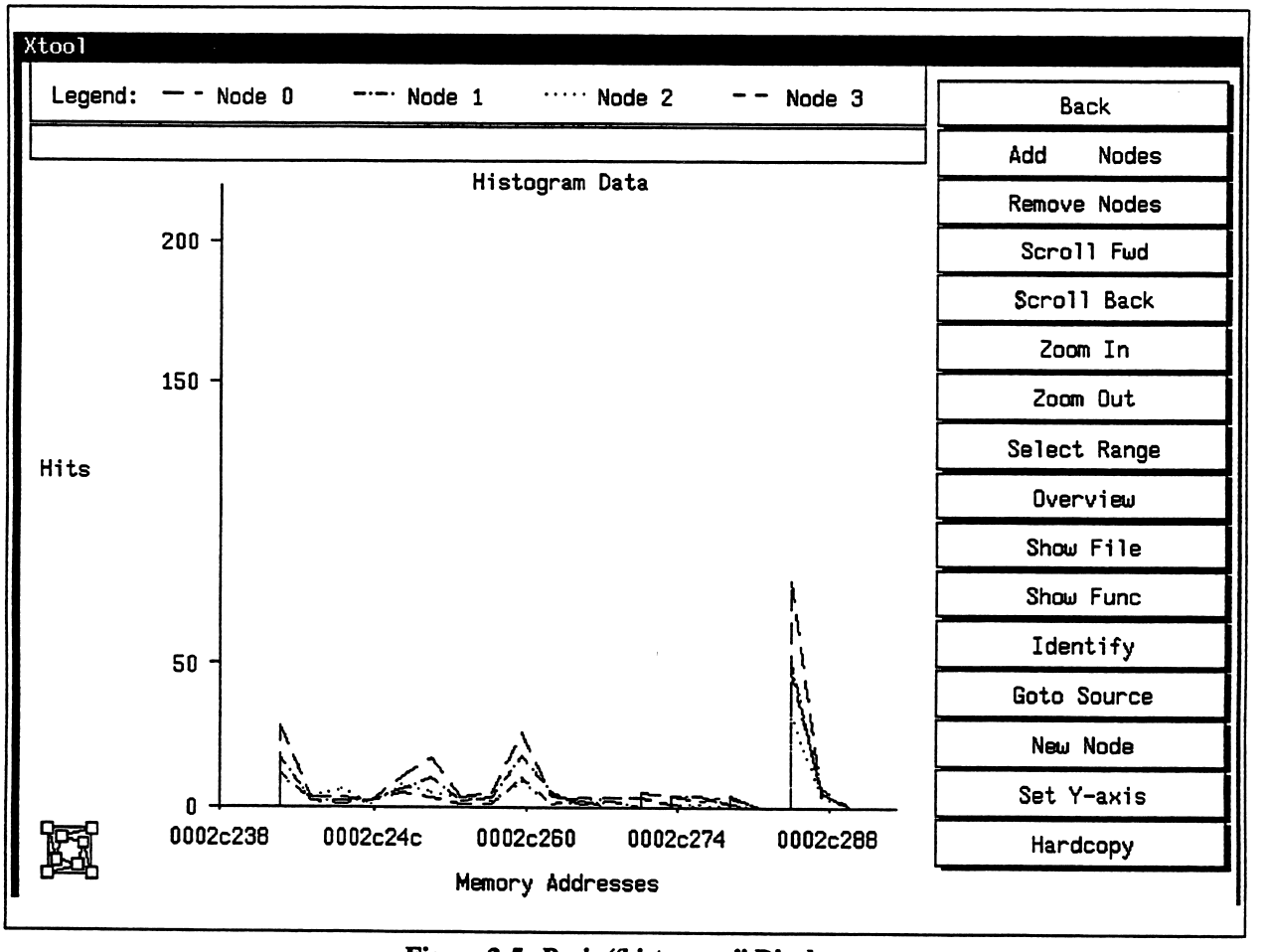

Figure 3-5. Basic "histogram" Display

Remove Nodes This is the opposite of the **Add Nodes** option and causes the checked group of nodes to have their data removed from the display.

Add Funcs Allows additional functions to have their data added to the display. The secondary menu from which selections are made is much like the node selection menu.

Remove Funcs Opposite of the **Add Funcs** option — removes the data for one or more functions from the display.

Hardcopy Create a PostScript file containing a black and white version of the image shown in the **Display Area**. Each time this option is selected a new PostScript file is created by modifying the file name incrementally.

The xxx **vs. Node** graphs (selected from the menu area of Figure 3-2) are very similar to those just discussed and their secondary menus appear just as that shown in Figure 3-3. The important difference between them, however, is that the default behavior is to show data from *no functions*. As a result the initial selection of this item will present a totally blank graph. The **Add functions** menu option should then be used to select those whose data is required.

Possibly the most interesting way to use **xtool** is to select **Histograms** from the main menu. This will present a set of options of which the most basic is **Overview**. Selecting this option results in a display like that of Figure 3-3. Along the horizontal axis is shown the range of memory addresses covered by the profiling histogram. This is determined by the *offset* and *scale* arguments passed to the **xprof_init()** function call. The vertical axis shows the number of histogram hits in the bin at each address. This type of display is interesting because typical programs show sharp spikes in their profiles which give a quick guide to the places of interest. It essentially has the resolution determined by the *scale* parameter and can be tuned arbitrarily by the user, even down to the level of single machine instructions.

The various menu options offered in this display perform the following actions:

Add Nodes, Remove Nodes
: Allows data from more, less or different processors to be displayed. The second level menu is identical in appearance and use to that shown in Figure 3-4.

Scroll Fwd, Scroll Back, Zoom in, Zoom out
: These functions provide simple control of the memory region displayed. Scrolling forward and back moves the display by half its current width while the **Zoom** functions either increase or decrease the amount visible by a factor of two.

Select Range
: This option allows selective zooming. A prompt appears in the dialog area asking for two points to be indicated in the display area. The cursor should be used to indicate an upper and lower bound for the region to be displayed by clicking on points in the Display area.

Overview
: Shows the entire memory range covered by the arguments to the **xprof_init()** system call.

Show File, Show Func
: These options use the information contained in the program's symbol table to display the memory range corresponding to either a single function or a singe source file. They are useful for monitoring the most heavily used functions. In both cases you will be prompted to enter the name of the file or function in the dialog area.

Identify
: This item allows memory addresses on the horizontal axis to be correlated with the appropriate source code. You are prompted to indicate a location in the display area. Clicking at some address causes the source file and line number at that location to be displayed in the dialog window.

Goto Source Selecting this item causes a prompt to be issued asking for a memory address to be indicated in the display area. Once chosen **xtool** switches to its "source code" mode in which the source and/or assembly code are shown together with the histogram data. The various options available in this mode are described later in this section.

New Node This option is used in conjunction with the **Identify** and **Goto Source** options. By default, memory addresses are mapped to program locations according to the information about the program running in node 0. All nodes are required to have the same source program.

Set Y-axis By default the vertical axis is scaled to the maximum histogram bin count for the entire program, irrespective of which part of memory is currently being displayed. This allows quantitative comparisons to be made visually. The scale can, however, be set manually by choosing this option. Any traces too large to fit on the rescaled graphs will be clipped.

Hardcopy This option creates a PostScript file containing the image in the display area. Successive invocations of this option produce separate files with incrementally increasing filenames.

The final displays available in **xtool** are the source/assembly code options. These can be reached by selecting the appropriate item from the main menu, or can be reached with the **Goto Source** command from the histograms menu. In either case a display similar to that shown in Figure 3-3 will appear.

In this mode the display area is divided into two pieces. On the right is shown the source and/or assembly code while the left area shows the raw histogram data as it is allocated to the source code lines through the information provided by the compiler/linker in the symbol table. This type of operation yields the most detail about the operation of the source code and is most often used to identify particular hot-spots which might benefit from reworking or assembly coding.

The various menu options available in this mode are as follows:

Add Nodes, Remove Nodes
Causes the usage information displayed in the left half of the display area to include or exclude particular processors.

Scroll forward, Scroll Back
Allows the source code to be scrolled by half a screen height in the indicated direction. If the display of the usage data is currently enabled this will also be updated to reflect the new source code that becomes visible.

Select File, Select Function
These options allow the source code for a particular source file or function to be displayed.

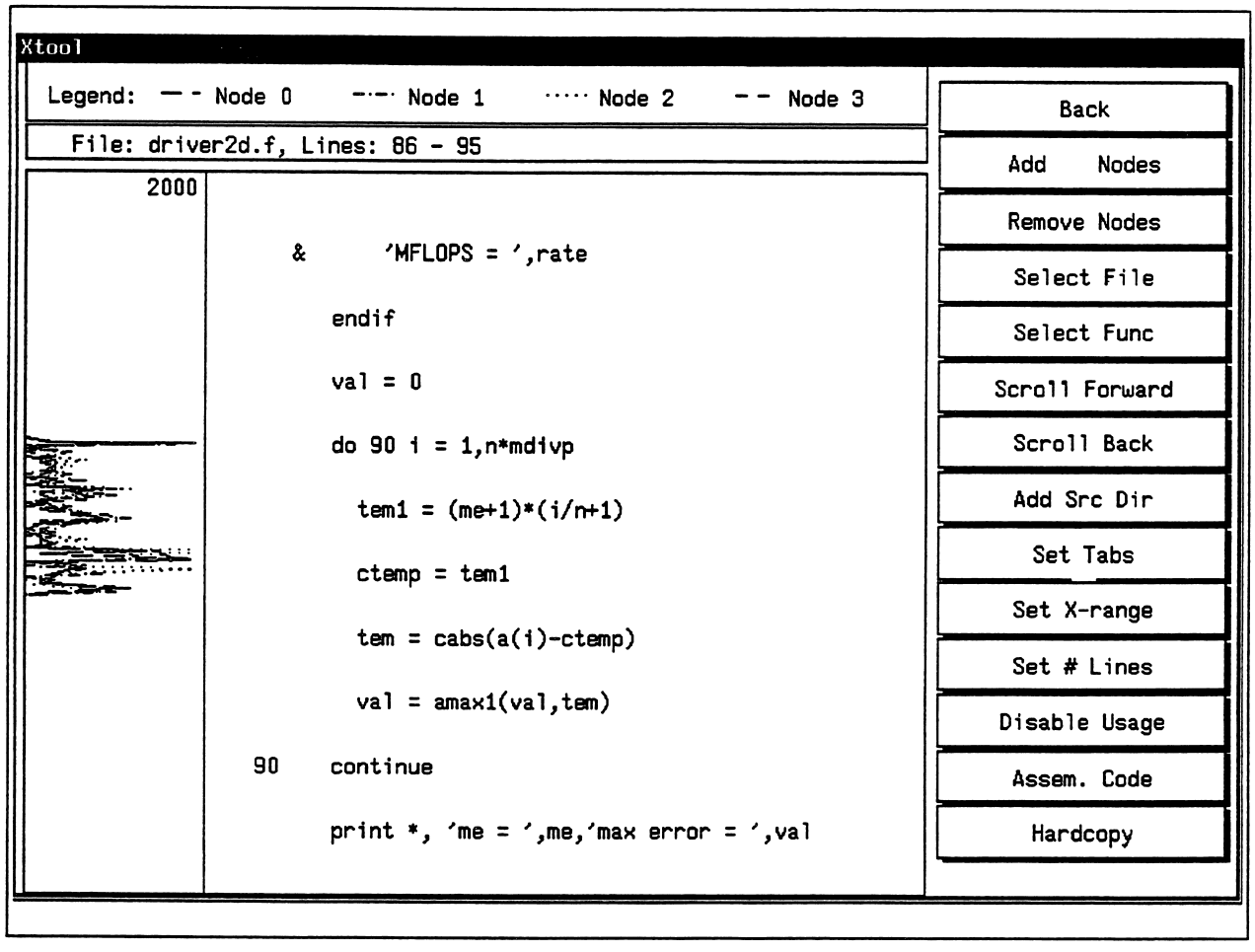

Figure 3-6. Source/Assembly code Display

Add Source Directory

By default **xtool** looks in the current directory for source files to display. Using this option (or the equivalent **-I** on the original command line) causes **xtool** to search additional directories.

Set tabs, Set X-range, Set #lines

These options provide fine control over the appearance of the display. The first is used to expand tab characters in the source code to spaces and the last tells **xtool** how many lines of source code/assembly code to display at once. The **Set X-range** option servers a similar function to the **Set Y-range** option on the histograms menu — it rescales the usage display at the left of the display area.

Disable Usage, Enable Usage

These options, which are mutually exclusive, either cause or suppress the updating of the left part of the display as the source code is scrolled. Since the display of the usage data can sometimes be time consuming this option allows the source code to be moved to interesting areas faster before enabling the usage display.

Assem. code This is a toggle which allows the interleaving of source and assembly code. By default, only the source code is shown. This option allows the additional information to be either added or removed from the display.

Hardcopy In common with the other menus this selection causes a PostScript file to be created with the contents of the display area.

Using Communication Profiling 4

Introduction

This chapter provides user information on the PAT communication profiling tools and system calls. The communication profiler is a tool designed to monitor internode message traffic and estimate overheads in various types of communication.

Communication Tracing Overview

The most obvious difference between a sequential program and its parallel counterpart is in the interaction between the multiple processors. The most basic of these interactions is the communication traffic between nodes.

On each node data is accumulated to measure:

- Time spent calculating, communicating between processors and performing I/O functions. This leads to an estimate of program overheads and efficiency.
- Total number of calls to the communication system. Provides a simple estimate of load imbalances.

As well as keeping track of the above statistics for each node the following data are maintained in each node on a function-by-function basis for every entry point into the communication system:

- Number of calls to each individual function.
- Distribution of return values from each function. Each function in the communication system returns a value indicative of the nature of the communication performed; message length written, message length read, number of objects broadcast, etc. This allows the user to evaluate the communication policy of an algorithm — in particular it may be more effective to bundle up short messages into longer communication packets rather than sending data piecemeal in short messages.

You can use the linker directive **-Mperf=comm** along with the **=auto** or **=manual** qualifiers to link your program with the *libnode.a* library containing the profiled communication procedures. Whenever any of the communication or I/O procedures are invoked the trace information for the number of function calls and the time spent in each function are saved in the communications trace file. The linker directive ensures that only predetermined functions are traced so that the actual communication profile is accurately reflected.

Automatic Operation

You can use the linker directive **-Mperf=comm** to link your program with the *libnode.a* library. The **=auto** qualifier can be added to the directive, but it is unnecessary because this is the default qualifier. The communication trace will be automatically performed when you load and run the program. Upon program termination the *exprof.prc* file (in your current working directory) will be written automatically by the performance monitoring code.

The communication profiler requires no configuration and, as such, can usually be controlled automatically by the system with no intervention from the user. The following actions are performed by the system:

- During system start-up the system call **cprof_inq()** is used to detect whether or not profiling should be performed. If enabled **cprof_on()** is called to start up the profiler.
- During program termination, the **cprof_inq()** function is again used to check for communication profiling. If communication profiling is enabled, the appropriate system call is made to dump the profile data to disk.

Environment Variables

The **EXPROF_SWITCHES** environment variable overrides the run-time effects of the compiler switches. Refer to "Using Environment Variables to Profile/Trace" on page 2-5 for a detailed description of the actions that can take place when the **EXPROF_SWITCHES** environment variable is used. To summarize, there are several possible results from using the **EXPROF_SWITCHES** environment variable to control communication tracing at run-time, as follows:

-Mperf=comm or **-Mperf=comm=auto**
: If the **EXPROF_SWITCHES** environment variable does not exist, automatic communication tracing is performed. If the **EXPROF_SWITCHES** environment variable contains a "**c**", automatic tracing will be performed. If the **EXPROF_SWITCHES** environment variable does not contain a "**c**", manual tracing is in effect.

-Mperf=comm=manual
: If the **EXPROF_SWITCHES** environment variable does not exist, manual communication tracing will be performed. If the **EXPROF_SWITCHES** environment variable contains a "**c**", automatic tracing will be performed. If the **EXPROF_SWITCHES** environment variable does not contain a "**c**", manual tracing is in effect.

You can determine the current setting of the **EXPROF_SWITCHES** environment variable (or its existence) by typing ***setenv*** from the shell.

Manual Operation

You can use the linker directive **-Mperf=comm=manual** to link your program with the communication-traced *libnode.a* library. There is usually no advantage to the use of manual tracing, unless you want to be able to exclude certain functions from the profile. The linker directives ensure that only the actual communication routines are traced. The **manual** qualifier allows you to exclude one or more of the functions, or focus only on a portion of a function.

The communication profiler is almost completely automated requiring little interaction from the user. Two routines **cprof_on()** and **cprof_off()** are provided to control the profiler. They turn the system on and off respectively. This allows complete selectivity as to exactly what portions of the code are profiled.

In a similar manner to the **xprof_inq()** function described earlier the communication profiler includes the function **cprof_inq()**. This function looks for an environment variable called **EXPROF_SWITCHES** and checks for the existence of the character "**c**" in the defined string. If the character "**c**" is found, 1 is returned. If the character "**c**" is not found, 0 is returned. Since the node program is usually supplied with the same environment variables as are defined in the host operating system you can use this function to control the profiler at runtime rather than hard-coding the calls to **cprof_on()** that enable the system.

Unlike the **xtool** and **etool** profiling tools, there is no initialization function (e.g., a function comparable to **xprof_init()** or **eprof_init()**) for the **ctool** communications profiler.

To write the profile data to disk use the **cprof_dmp()** function. This routine has a single argument; the name of the file to be created for the profiler's data. As well as writing out the data this routine also turns off and resets the internal state of the communication profiler so that it can subsequently be re-enabled to profile another section of code.

This latter possibility is useful when an algorithm has multiple phases which can usefully be profiled separately. In the code below, for example, we profile the first and third stages of a program dumping the resulting data to differently named files.

```
/*
 * Demonstrating the use of the "dump" functions to
 * make multiple output files containing profiles of
 * different phases of an application.
 */
main()
{
/* Application phase 1. */

   if(cprof_inq()) cprof_on();
     ...

   if(cprof_inq())                              /* Write out data */
      cprof_dmp("phase1.prc");

/*
 * Application phase 2. (Don't profile since cprof_dmp
 * turned system off.)
 */
     ...

/* Application phase 3., reset profiler, and turn on */

   if(cprof_inq()) cprof_on();

        ...

   if(cprof_inq())                      /* Write out data again.. */
      cprof_dmp("phase3.prc");

      ...
```

Note how this code uses the **cprof_inq()** routine to control the profiling actions. Only if the environment variable is set does the profiler get used. Note also that most node programs do not need to use the **cprof_dmp()** function explicitly since it will be called automatically during the system exit code.

Analyzing the Communication Profile — ctool

After program execution is complete, one or more files should be left containing the communication profiles for the application. These are analyzed with the **ctool** utility. For the moment we will neglect the graphical interface and simply present a tabular version of the profile. This is achieved with the command:

```
ctool -p program_name
```

which expects to find the communication profile in a file called *exprof.prc*. If the file was renamed for some reason then one might instead use:

```
ctool -p program_name phase1.dat
```

to read profiling information from a file called *phase1.dat*.

The result of this command is a table with the following fields:

1. A separate section of the table is provided for each node, and is identified by its processor number.

2. A brief summary of the total times spent calculating, performing I/O and communicating between processors is provided with times in milliseconds.

3. For each node there is a breakdown of the calls to each of the basic communication functions. For each function that was called at least once the number of calls, the total time in that function, and the number of errors returned by that function are shown. Time is measured in milliseconds.

4. The final panel of the display shows a brief analysis of the way in which the various functions were called. The tabulated values show the frequency of return values from a function, binned in logarithmic steps. Thus, the first column indicates the number of times that the given function returned 0 to its caller, the first column the number of times 1 was returned etc. The exact interpretation of this information obviously depends somewhat on the function being called but, in almost all cases, is related to the message length being dealt with. The number of bins displayed can be modified with the **-b** switch. Refer to the **ctool** command manpage for details on using the **-b** switch.

The information tabulated by this command allows a rather detailed study of the algorithmic communication patterns to be made. In particular we have found it invaluable for finding program errors in which too much data is sent in some system call. This type of error tends to show up very clearly in the tabular output.

A graphical interface is available for displaying the data. This is accessed by omitting the **-p** switch from the **ctool** command — in most cases one simply executes:

```
ctool program_name
```

although occasionally you may have to give the name of the file containing the profile data. After a few of seconds of initialization a diagram similar to Figure 4-1 should appear.

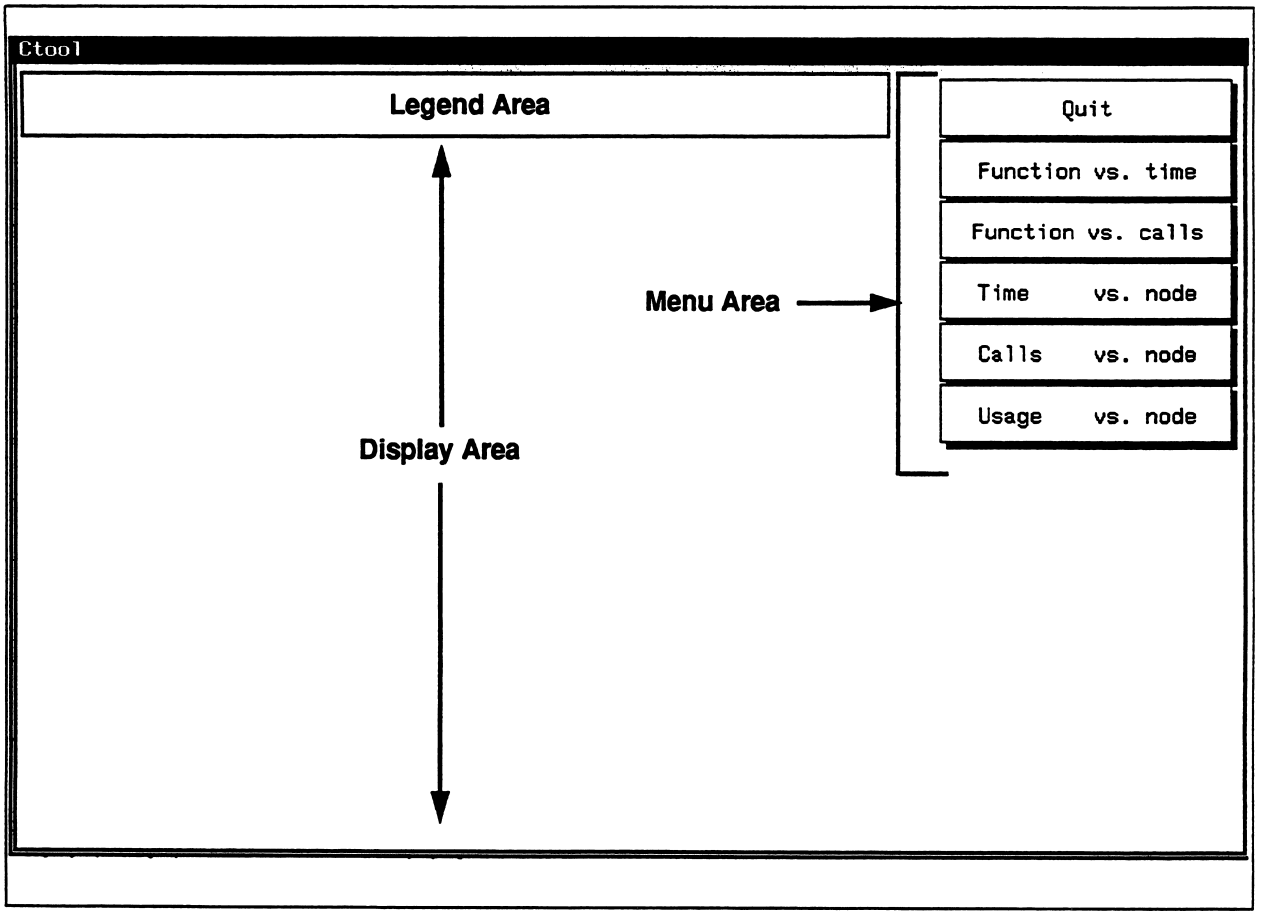

Figure 4-1. Initial ctool Menu and Display

Various pieces of the display are used for special functions

Menu Area Select from the various options available by positioning the cursor over one of the boxes in this area and clicking. The text in each box should help you make the appropriate selection. (If your system has a mouse you should be able to make selections in the usual point-and-click manner. If not use the arrow keys on the numeric keypad to move the cursor and strike any alphanumeric key to make a selection. The <HOME> key toggles between fast and slow cursor motion)

Display Area This area of the display is used to present graphical data. Various types of graphs are available to show different aspects of the communication profile.

Legend Area While data is being displayed graphically this box should contain a legend indicating the meaning of the various items shown.

The selections in the main menu which should now be visible represent various ways of showing profile data. There are essentially three variables involved in each case; the particular function, a node and an interesting quantity which in this case is either time or the number of calls to a particular function. Since graphs are really designed to show only two variables — one on each axis — there are several ways of showing the data.

The first display is **Function vs. Calls.** This presents a horizontal bar chart showing the number of times each function has been called in each processor. An example is shown in Figure 4-2.

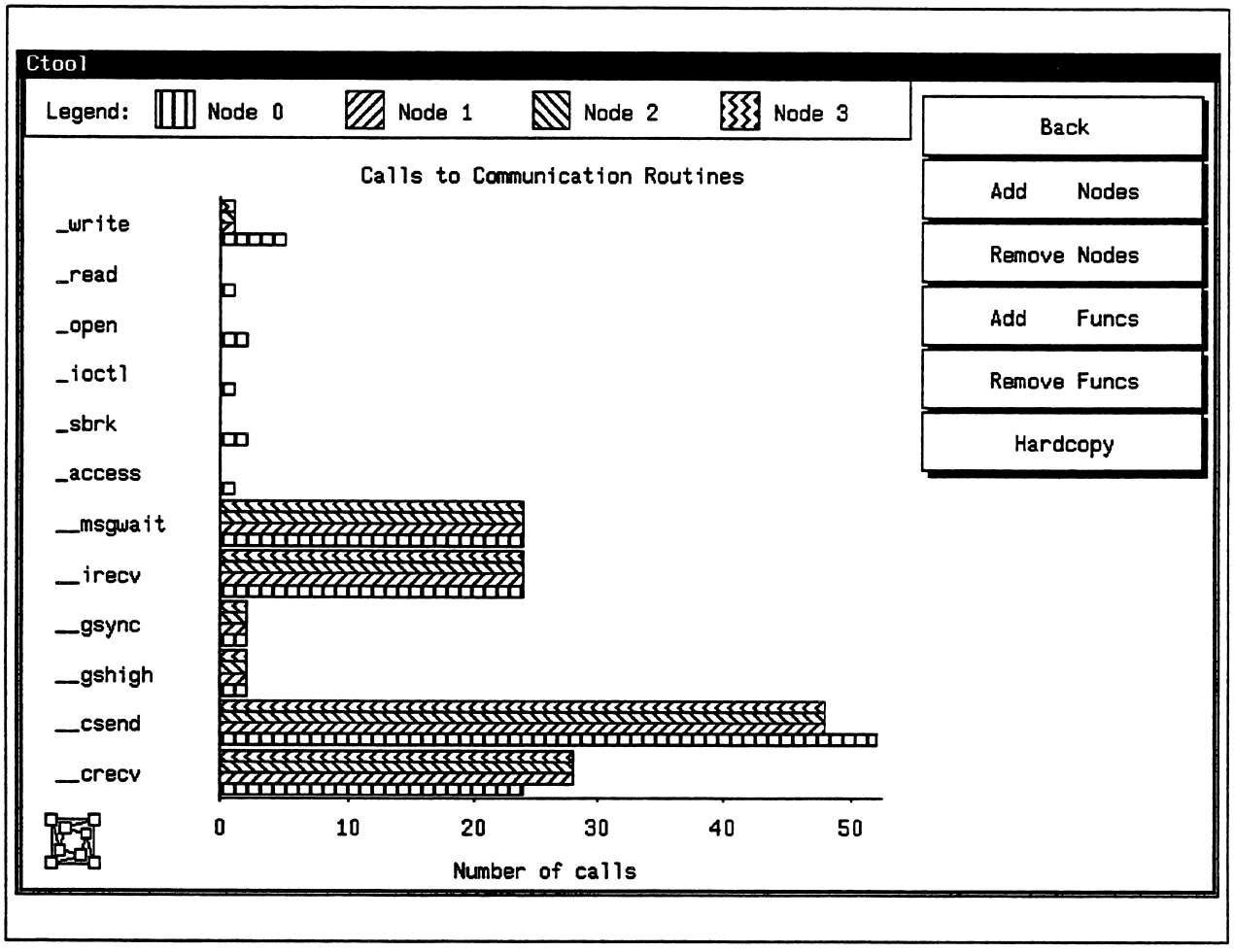

Figure 4-2. Node Selection Menu

Notice that the various processors have been displayed by individual bars (color-coded, if you have a color device) and that a key appears in the **Legend Area** showing which processor is which. In the example shown we assumed that only four processors were being profiled which fit quite nicely on the display. If you have 128 nodes then the bars are going to be awfully thin so **ctool** leaves out all but the first 16 encountered. However, you can control which bars you see by using the menu which should also have appeared (See Figure 4-2). Two of the options are **Add Nodes** and **Remove Nodes** which allow you to either add or delete nodes from the display.

Selecting either of these options yields yet another menu which looks like that shown in Figure 4-2.

This menu is typical of the lower level menus. It has a **Done** entry at the top which takes you back to the previous selection and then several other options. In order to select some nodes for either addition or removal you can:

- Select individual nodes by clicking on their numbers. As you do this a check mark should appear by the side of the name. This mark can be removed by selecting the same node again.
- Select all nodes by clicking on the **All Nodes** box.
- Page through the nodes, if there are too many to fit on the menu all at once. **Page Forward** and **Page Back** move through the set of nodes.
- Select a range of nodes by clicking on the first number, then the **Thru** box and then the second number. You can switch pages in the middle of this operation if the range spans multiple pages.
- Select the neighbors of a particular node. One interesting property of parallel machines is the way in which one node's behavior can affect those connected to it. To select this option click in the **Neighbors of** box and then the node whose neighbors you want to pick.
- Select nodes according to their parity. This concept corresponds to the familiar red-black coloring often used in parallel processing. Node 0 is defined to have even parity and its neighbors to have odd parity. This then extends naturally so that no even parity node is adjacent to another even parity node, and similarly for the odd parity processors.

After selecting one (or more) of the options from this menu, clicking **Done** takes you back to the previous menu level and you will see the consequence of your selection on the **Display Area**. If, for instance, you decided to add all 128 nodes to the display you are probably waiting quite a while for the very tiny graphs to get drawn!

This technique is one way of getting additional node displays onto the screen. Another method is to "lose interest" in particular functions. By default every function that was called at least once has a trace. To remove some functions from the display select the **Remove Funcs** option. The menu that appears looks a little like the node selection menu — it has **Done** at the top and then a list of communication functions. You click away at the names (which should acquire check marks which can be removed by clicking a second time) until the uninteresting ones have been deleted and then select **Done**. This returns to the previous level and updates the **Display Area** with fewer functions, and correspondingly more space for node displays. Another thing to notice is that the horizontal axis rescales whenever functions are added/deleted so it is occasionally useful to remove functions whose usage dwarfs the others to force a magnification of the horizontal axis showing more detail. The deleted functions can always be restored later if required.

The final option on this, and the other display menus, is **Hardcopy**. Clicking on this box saves a copy of the current display (without the attached menus) in a form suitable for printing.

Once all interesting information has been extracted from this display one can click on the **Back** box to return to the main selection menu. The box **Function vs. Time** presents the same style of graph but with the horizontal axis displaying the time spent in each function rather than the number of times it was called as in the previous case. The two boxes **Time vs. Node** and **Calls vs. Node** present alternative views of the profile data.

These options present typical graphs of either the time spent in a routine or the number of calls to a routine against the processor number on the horizontal axis. Individual curves are drawn for each function selected by the user. Clicking on the **Time vs. Node** box, for example, should produce the display shown in Figure 4-3.

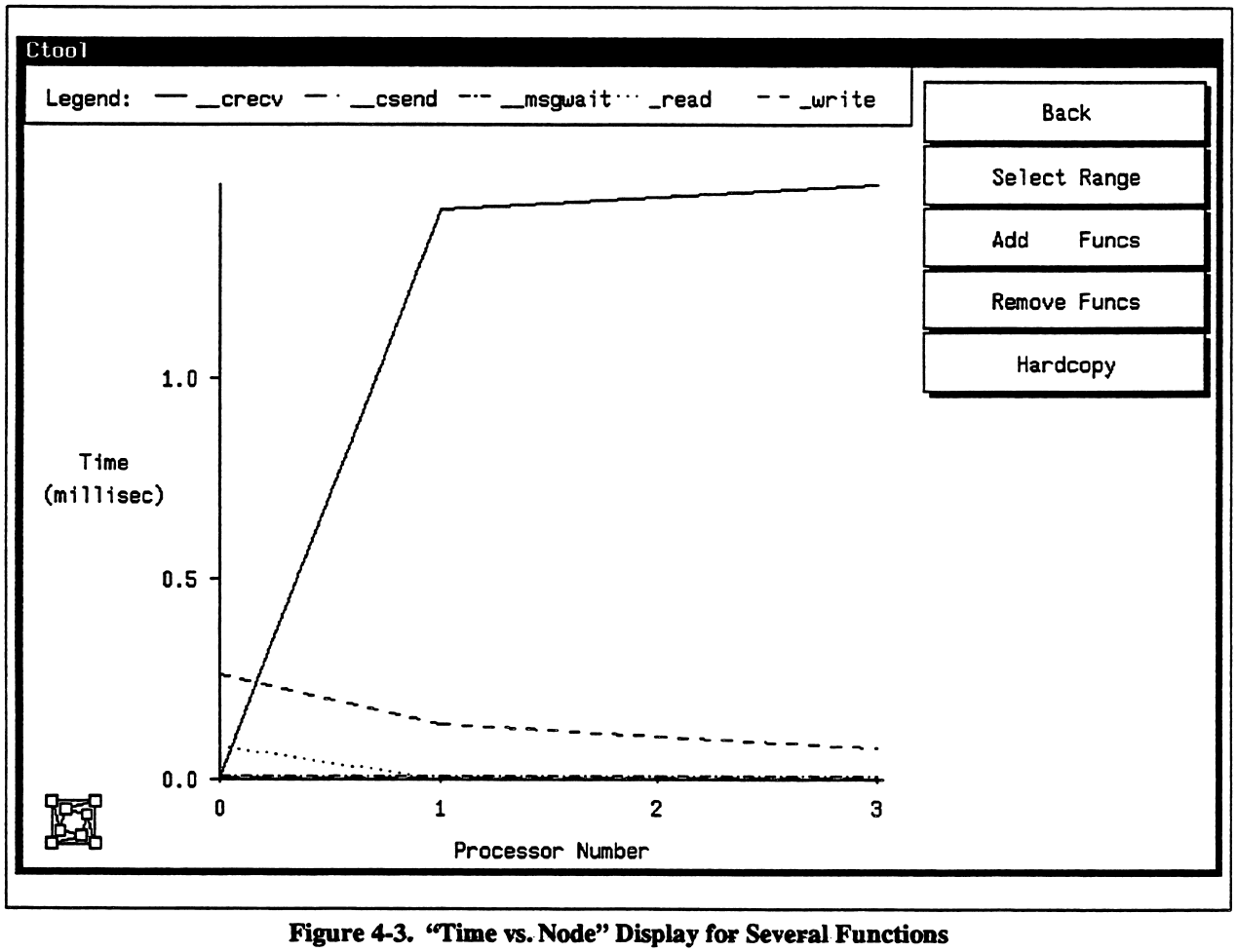

Figure 4-3. "Time vs. Node" Display for Several Functions

This type of display is obviously better suited to showing data from a lot of processors. Even so it occasionally becomes too crowded and the **Add Nodes** and **Remove Nodes** menu selections are available as before. Note, however, that it doesn't make too much sense to pick out weird node combinations for this type of display and, in fact, **ctool** will ignore you if you try. While displaying data in this fashion only the highest and lowest processor numbers are considered and everything in between is also plotted. At least two nodes must be selected for this display.

The final graphical tool available is obtained by selecting **Back** from this menu level and **Usage** from the main menu. The result will look something like that shown in Figure 4-4. Note that the highest node usage establishes the 1.0 (100%) reference, and all other nodes are compared to that one. The usages reported in this display may not correspond exactly to the sums indicated in other displays. This happens because some functions are called by other functions and (depending on the display), this usage may be either counted separately or included with the calling function.

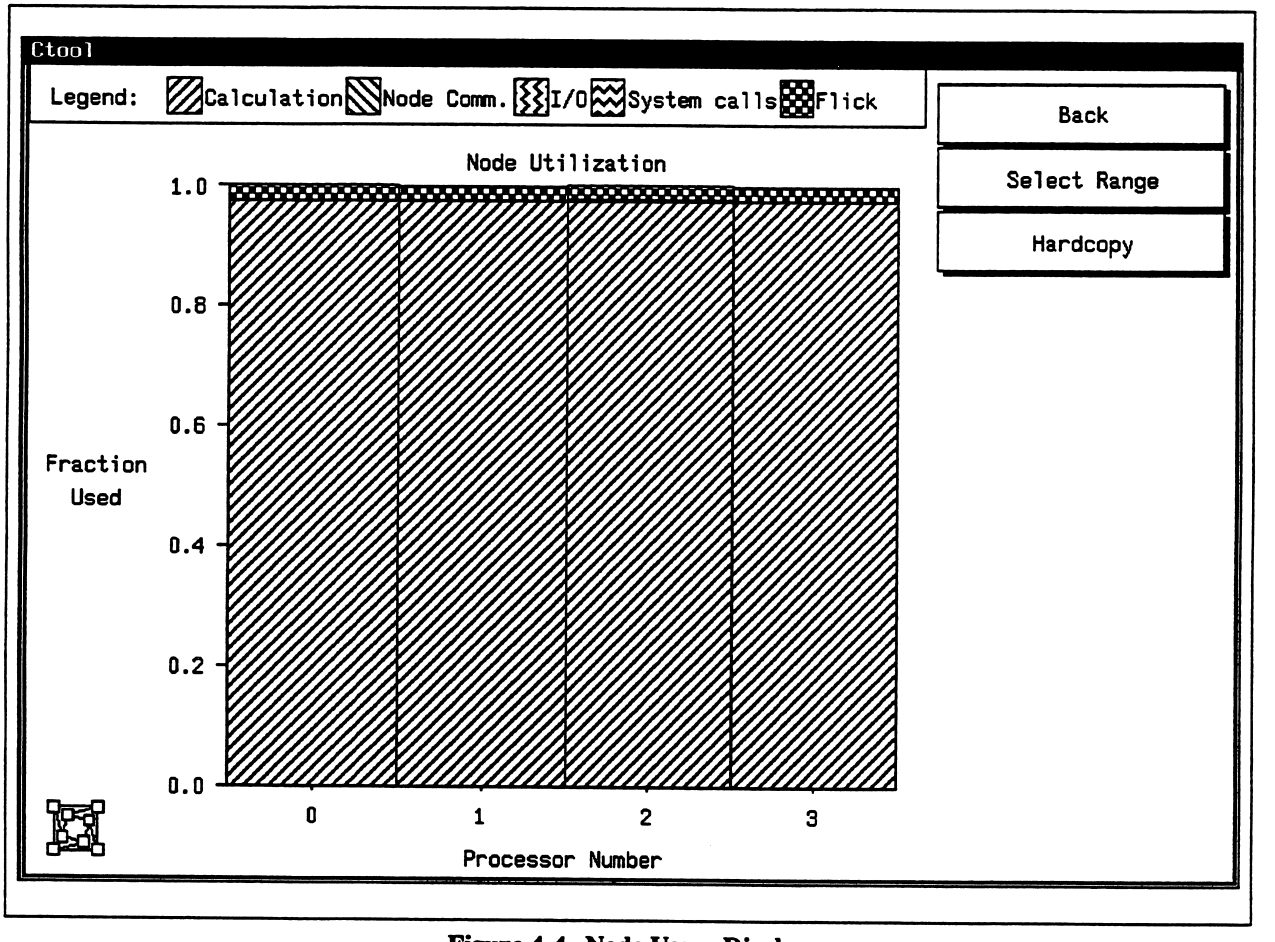

Figure 4-4. Node Usage Display

Along the horizontal axis are the processors just as in the previous displays. For each processor a stacked bar chart is presented showing the division of time between the five fundamental tasks; calculation, node-to-node communication, I/O, system calls, and the **_flick** function.

Quit the profiler by selecting **Quit** from the main display menu.

Using Event Driven Profiling 5

Introduction

This chapter provides user information on the PAT event driven profiling tools and system calls. The event driven profiler is a utility that allows you to examine in detail the interaction between iPSC nodes, by displaying general or user-specified events in the application against time.

Event Tracing Overview

The two profiling techniques discussed so far have been tailored to examining the behavior of nodes in isolation. The event driven profiler is provided to allow more detailed examination of the interaction between various nodes as time progresses through the application. An "event" is a user-specified point in the execution of an application which will be recorded in an internal log for later analysis. In addition to an event's occurrence, you can also record:

- The time at which the event occurred.
- An index value indicating the nature of the event.
- A program variable whose value at the time of the event will be recorded. This will help in later identifying events during analysis.

As well as the above data items which are recorded every time an event occurs the following can optionally be supplied:

- A title that identifies all events with a given index value.
- A `printf`-style format string which will be used when printing the value of the program variable stored at the time of the event.

These last two items are intended to facilitate identification of program events in the analysis phase. They may be omitted if desired.

It is important that a user program containing event specifications does not have to incur the overhead of the profiling system. As with the communication profiler discussed in the previous section one is free to turn the event profiler on or off at will, completely independently of the other profiling systems.

You can use the linker directive **-Mperf=event** along with the **=auto** or **=manual** qualifiers to link your program with the *libnode.a* library containing the profiled system, communication, and I/O procedures. The **=auto** qualifier can be added to the directive, but it is unnecessary because this is the default qualifier. The **=manual** qualifier causes the event profiler to trace only those procedures containing the PAT event tracing procedure calls.

Automatic Operation

You can use the linker directive **-Mperf=event** to link your program with the *libnode.a* library. The **=auto** qualifier can be added to the directive, but it is unnecessary because this is the default qualifier. The event trace will be automatically performed when you load and run the program. Upon program termination the *exprof.pre* file (in your working directory) will be written automatically by the performance monitoring code.

The event profiler can be initialized automatically. The following actions are performed by the system:

- During program start-up the system call **eprof_inq()** is used to detect whether or not profiling should be performed. If enabled **eprof_init()** is invoked with default arguments and **eprof_on()** is called to start up the profiler.
- During the program termination, the **eprof_inq()** function is again used to check for event profiling. If enabled the appropriate system call is made to dump the profile data to disk.

Environment Variables

The **EXPROF_SWITCHES** environment variable overrides the run-time effects of the compiler switches. Refer to the environment variable description in Chapter 2 for a detailed description of the actions that can take place when the **EXPROF_SWITCHES** environment variable is used. To summarize, there are several possible results from using the **EXPROF_SWITCHES** environment variable to control event tracing, as follows:

-Mperf=event or **-Mperf=event=auto**

If the **EXPROF_SWITCHES** environment variable does not exist, automatic event tracing will be performed. If the **EXPROF_SWITCHES** environment variable contains an **'e'**, automatic tracing will be performed. If the **EXPROF_SWITCHES** environment variable does not contain an **'e'**, manual tracing is in effect.

-Mperf=event=manual

If the **EXPROF_SWITCHES** environment variable does not exist, manual event tracing will be performed. If the **EXPROF_SWITCHES** environment variable contains an '**e**', automatic tracing will be performed. If the **EXPROF_SWITCHES** environment variable does not contain an '**e**', manual tracing is in effect.

You can determine the current setting of the **EXPROF_SWITCHES** environment variable (or its existence) by typing **setenv** from the shell. Once the **EXPROF_SWITCHES** environment variable exists, you can turn off any of the settings by setting the **EXPROF_SWITCHES** variable without any arguments.

The **EXPROF_OPTS** environment variable may be used instead of, or in conjunction with the PAT programmatic interface. Using the **EXPROF_OPTS** environment variable, you can increase the maximum number of entries in the log files to ensure full coverage for your program. Alternatively, you can use programmatic control to focus on a small area of the program, and also use the **EXPROF_OPTS** environment variable to fine tune the amount of data collected. Refer to the description in Chapter 2 for more information on the **EXPROF_OPTS** environment variable.

Manual Operation

You can use the linker directive **-Mperf=event=manual** to link your program with the event traced *libnode.a* library. You can selectively use the compiler switch **-Mperf=event=manual** to specify which procedures in your application code will be event traced.

The most obvious of the event profiling calls are **eprof_on()** and **eprof_off()**. Called with no arguments these functions serve to turn on and off (respectively) the event profiling system. This allows fine control over the areas which will be profiled and also lets completed applications run intact without removing the profiling commands. The use of these functions is exactly analogous to the **_on** and **_off** functions of the other profiling subsystems.

The most important function in this section is **eprof_add()** which causes an event entry to be added to the log file. Its usage is:

```
eprof_add(index, datum);
```

The first argument is an identifier for the type of event being recorded which allows one level of identification when analyzing the trace. One example of its use might be to flag all calls to a particular routine with **index = 1** while calls to another routine might have **index = 2**. The second value is another means of identifying events. It is any 32-bit integer value. A good example might be a loop counter or the value returned by some function.

The following code illustrates one use of these functions:

```
/*
 * Sample program demonstrating the use of the event
 * profiler.
 */
#include "expm.h"

main()
{
     int iter;
     float value, crunch();

/* Initialize profiling system to defaults, and turn
 * it on.
 */
     eprof_init(DONTCARE, DONTCARE);
     eprof_on();

/* Set up algorithm, loop several times - each time
 * record an event of type 1 and also record the loop
 * index.
 */
     for(iter=0; iter<100; iter++) {
       eprof_add(1, iter);

/* Now record a type 2 event for the completion of the
 * CRUNCH function and also save the value it returned.
 */
      value = crunch(iter);
      eprof_add(2, (int)value);

      enditer(value);
     }
     exit(0);
}
```

Notice the call to **eprof_init()** in the previous example. This is important. Each call to **eprof_add()** stores additional information in an internal memory buffer on each node. The amount of memory set aside for event logging is determined by the call to **eprof_init()** which must occur before attempting to turn on the profiler, or use any of the event profiler functions. This routine takes two arguments:

```
eprof_init(numlogs, numlabs);
```

in which the first argument specifies how many event log entries should be allocated. If an attempt is made to write more log entries than specified in this call the extra entries are discarded and a warning is issued whenever the data is analyzed. The second argument specifies how many title or label entries will be made to aid in identifying the program state when an event occurs. Either

argument may take the special value DONTCARE, defined in *expm.h* in which case a default is chosen. One important fact about this routine is that it is used to synchronize the clocks on the various processors and must, therefore, be called "loosely synchronously" in each processor. This means that the call must be made at a point where each node is free to communicate with all the others — i.e., there must be no pending communication calls, unread messages, etc. Furthermore any node which makes this call will be blocked until all nodes have made it.

The iPSC/860 nodes each have their own clocks, but the time scale reported with the **etool** display is based on the clock used by node 0. Because of the separate node clocks, it is likely that (with time) the events reported on different nodes will get out-of-sync with the node 0 clock. The **eprof_init()** call uses a **gsync()** call to synchronize the clocks, but since it uses message passing, even this synchronization step may leave the nodes slightly out-of-sync due to inherent communication overhead and transit time.

Having described how the system is initialized and how events are entered into the log one must consider the steps taken to aid in analyzing the profile data. The **etool** utility presents "time-lines" for the individual processors upon which are superimposed the user-specified events. Each event is identified in this display with its index argument from the **eprof_add()** call responsible for its existence. One problem with this style of display, however, is that it is often quite tricky to figure out the correspondence between the time lines and what the actual application is trying to do. This problem is somewhat alleviated by intelligent choices of the `index` parameters. Since each event is labelled with this value one gets a rough guide.

The connection can be strengthened by specifying a title field for each index value with the **eprof_label()** call. This is called with three arguments:

```
eprof_label(index, title, format);
```

The title is merely a character string that will appear in a legend on the display of the time-lines. Each index can have a unique title assigned to it in this manner allowing reasonable identification of the various event types. In the previous example one might add the calls:

```
eprof_label(1, "Top of major iteration loop", "Iteration %d");
eprof_label(2, "After crunching", "crunch returns %d");
```

Note how the title strings identify the purpose of the two types of events.

In addition to displaying the user events on the time-lines **etool** also allows access to a second layer of information — that supplied in the second argument to **eprof_add()**. This information is available upon request and interacts with the last argument to **eprof_label()**. The last argument to the **eprof_label()** function is a C-language format string that is used to display the second argument. Whenever the next layer of detail is requested the user datum corresponding to the selected event formatted according to the last argument to **eprof_label()** to be printed out. Thus, for example, one might inquire about the details for a particular event and be told:

```
1. T = 246.23 ms, "Iteration 39"
```

The information contained here is the index number, the time at which the event occurred and the user data item formatted in conjunction with the format string given to **eprof_label()**. Notice how this information can be used to exactly locate a position on the time line, for example, according to which iteration of the major loop it signifies. Even program bugs might be detected this way since clicking on a type 2 event might yield:

```
2. T = 253.60 ms, "crunch returns -2461"
```

which, in conjunction with the previous output, might be enough to detect that the program is going crazy at iteration 39 since the value returned by the `crunch` function is negative.

Careful use of the labelling facility is the key to using the event profiler. Without it one often has to resort to guesswork in order to relate the events shown on the time lines to the program's behavior. If the labelling is performed carefully, especially the specification of the second piece of information, the data item argument to **eprof_add()**, the event profiler will be a rich source of information about the performance (and maybe even bugs) of an application.

The event profiling tools described here can also be used to record important system events. A particularly important class of interesting events are communication calls between processors. Each application, system, communication, and I/O call also makes entries in the log file. As well as recording which function was called and the value it returned to its caller one can also determine exactly how long each communication call takes. This is invaluable, for instance, in determining the affects of poor load-balance — typically one processor will wait for an excessive amount of time in communicating with an overworked node.

As with the other profiling systems the event profiler has a function **eprof_inq()** which can be used to check for the existence of the character 'e' in the environment string **EXPROF_SWITCHES**. In the same way as for the other profiling systems, this is usually used to provide runtime control of the profiling process.

Measuring Time Intervals with "Toggles"

The event profiler also provides a mechanism for measuring important statistics in relation to section of program code. While the execution profiler described in Chapter 3 is useful for collecting information at the level of individual subroutines or even individual source or assembly instructions it is often important to be able to analyze code at an intermediate level, or to gather statistics about the frequency with which a given program segment is being used.

To facilitate the gathering of such statistics the event profiler uses the toggle concept. A toggle is a structure which gathers information about the time spent within a particular program segment and the number of times the segment is executed. A simple example of its use is shown in the following code:

```
#include "expm.h"

ETOGGLE looptog, grindtog;

main()
{
      float Energy, grind_away();
      int iter, i;

/* Initialize toggle data structures. */

      eprof_toginit(&looptog, "Main iteration loop");
      eprof_toginit(&grindtog, "Calls to grind_away");

/* Start application code, then go into main loop */

         ...

      for(iter=0; iter<100; iter++) {
         eprof_toggle(&looptog);

            /* Other processing going on here.... */

         ........


         for(i=0; i<4; i++) {
            eprof_toggle(&grindtog);
            grind_away(Energy, i);
            eprof_toggle(&grindtog);
         }
         eprof_toggle(&looptog);
      }
/*
 * Dump data to host for later analysis.....
 */
            .........

      exit(0);
}
```

The above example sets up two toggle variables using the ETOGGLE type defined in the *expm.h* header file, and then surrounds interesting pieces of code with identical calls to the **eprof_toggle()** function which alternately start and then stop recording information about the code section. (This is why the tool is known as a toggle — successive calls alternate between turning it on and off.)

The statistics gathered include the time taken to execute the enclosed code and the number of times this code segment is executed.

Each toggle structure *must* be initialized with a call to **eprof_toginit()** as shown at the top of the previous example. This notifies the system of the use of the particular variable and also allows the user to associate a title string with the indicated toggle. A title string makes it easier to analyze the resulting data since the string will be displayed along with the associated data.

Programs may dump profile data to disk at any time using the **eprof_dmp()** function. In common with the previously introduced dump routines this call expects a single argument; the name of the disk file to contain the profiling data. It must be called "loosely synchronously" in all processors and may be repeated as many times as wished to generate independent profile files. Each call to the routine resets the internal state of the profiler and turns it off, including all label definitions.

This routine is invoked automatically by the program exit code. As a result most programs do not need to call this function explicitly.

The **eprof_on()** and **eprof_off()** functions may be used freely to control the periods during which profile data is being accumulated.

Analyzing the Event Profile — etool

The event profile is analyzed with the **etool** utility. To execute this command one types

```
etool program_name log_file_name
```

where the last argument is the name of the file containing the event log. If this has the default name *exprof.pre* then it can be omitted leaving the simple command line:

```
etool program_name
```

This command should result in a display that looks rather like Figure 5-1.

The different areas of the display are used for various purposes in manipulating and interpreting the event log:

Menu Area This is usually the area in which user selections are made which cause various operations to be carried out in the event log display.

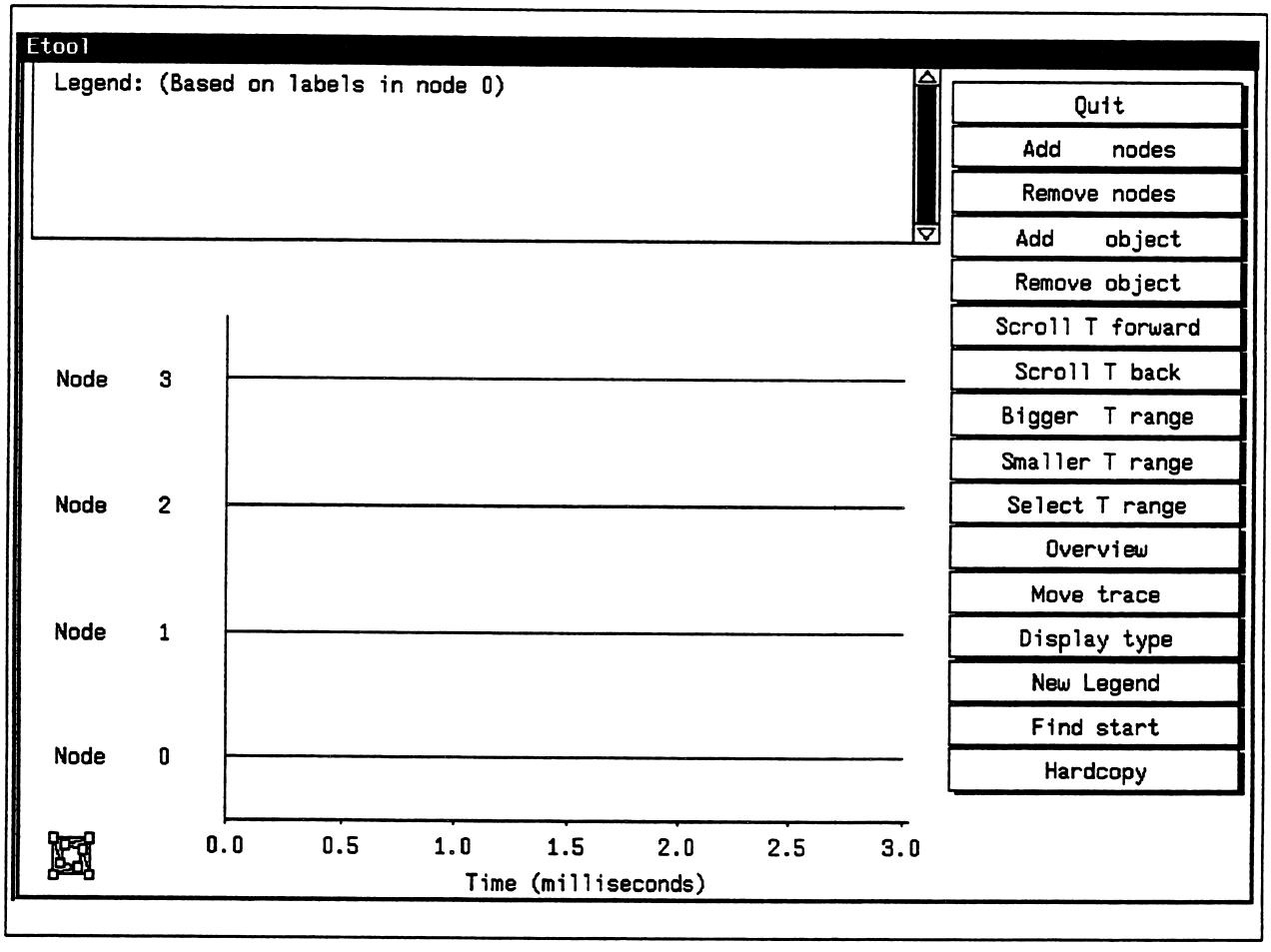

Figure 5-1. Basic Event Profile Display

Display Area This region contains the time-lines for the various processors. A simple horizontal line indicates computing activity while various types of colored and shaded boxes indicate user and system events.

Legend Area This area is used to indicate the meanings of the various objects shown in the **Display Area**. It usually contains an index to the various event types defined in a particular node, although it can also be used to display a key to the encoding of system events.

Dialog Area This region is used to interact with the user. Prompts for user actions are displayed here as well as information concerning events displayed on the screen.

To manipulate and analyze the display one selects from the options in the **Menu Area** on the right using the cursor. Some of the more obvious selections concern the amount of time displayed on the horizontal axis. By default **etool** begins by showing approximately 3 milliseconds of elapsed time. Often there will be no interesting events in this range as typified by the dull display of Figure 5-1. The most naive things to do to correct this situation are the various T-range commands:

Scroll T forward
Scrolls the time-lines forward by half the current width. Thus, if the current display goes from 3 to 4 milliseconds then scrolling would alter the range to $3.5 < T < 4.5$.

Scroll T back Scrolls the time-lines backward by half the current width.

Bigger T range Doubles the range along the horizontal axis while keeping the start point fixed. If the originally display were from 3 to 4 milliseconds then this command would yield the range $3 < T < 5$.

Smaller T range
Zooms in on the time axis by halving the current range while keeping the start point fixed.

Select T range This option allows you to pick out an interesting range from the current display with the cursor. Immediately after selecting this option the message:

```
Select lower time limit
```

will appear in the Dialog Area. You should then move the cursor into the Display Area and click when it is at the lower limit of some interesting range. At this point the prompt in the dialog area changes, asking you to select an `upper limit` and the process is repeated for an upper limit. When this has been selected the time-lines will be redrawn with the horizontal axis displaying the selected range of time values. Note that the range selection takes place *inside* the Display Area and not on the time axis itself. This allows you to pick out interesting *objects* from the display as the guide to an interesting time range rather than having to trace down to the horizontal axis to make the selection.

These options may be used at any time and merely manipulate the data on the display. They are useful when either too much or too little detail is being displayed and one needs to either zoom in or zoom out a little in order to make sense of the events being shown.

Of course these options might not help much if your events are widely spread or start after a lengthy period of program execution. To cover these possibilities two further options are provided:

Find Start As its name implies this option is used to find the first logged event. After selecting it you will be prompted in the Dialog Area to:

```
Select a node
```

To do this move the cursor into the Display Area and click over one of the traces on the screen. This selects that node and, in the current context, looks for the first enabled event in that processor and resets the horizontal range so that this event is displayed.

Overview This option draws all the time-lines in the selected processors from start to finish. User specified events are indicated by vertical bars rather than their full symbols to conserve space. (See Figure 5-2 for an example of the output produced by this command). This is often a useful option to select first followed by **Select T Range** to pick out interesting areas for finer scrutiny.

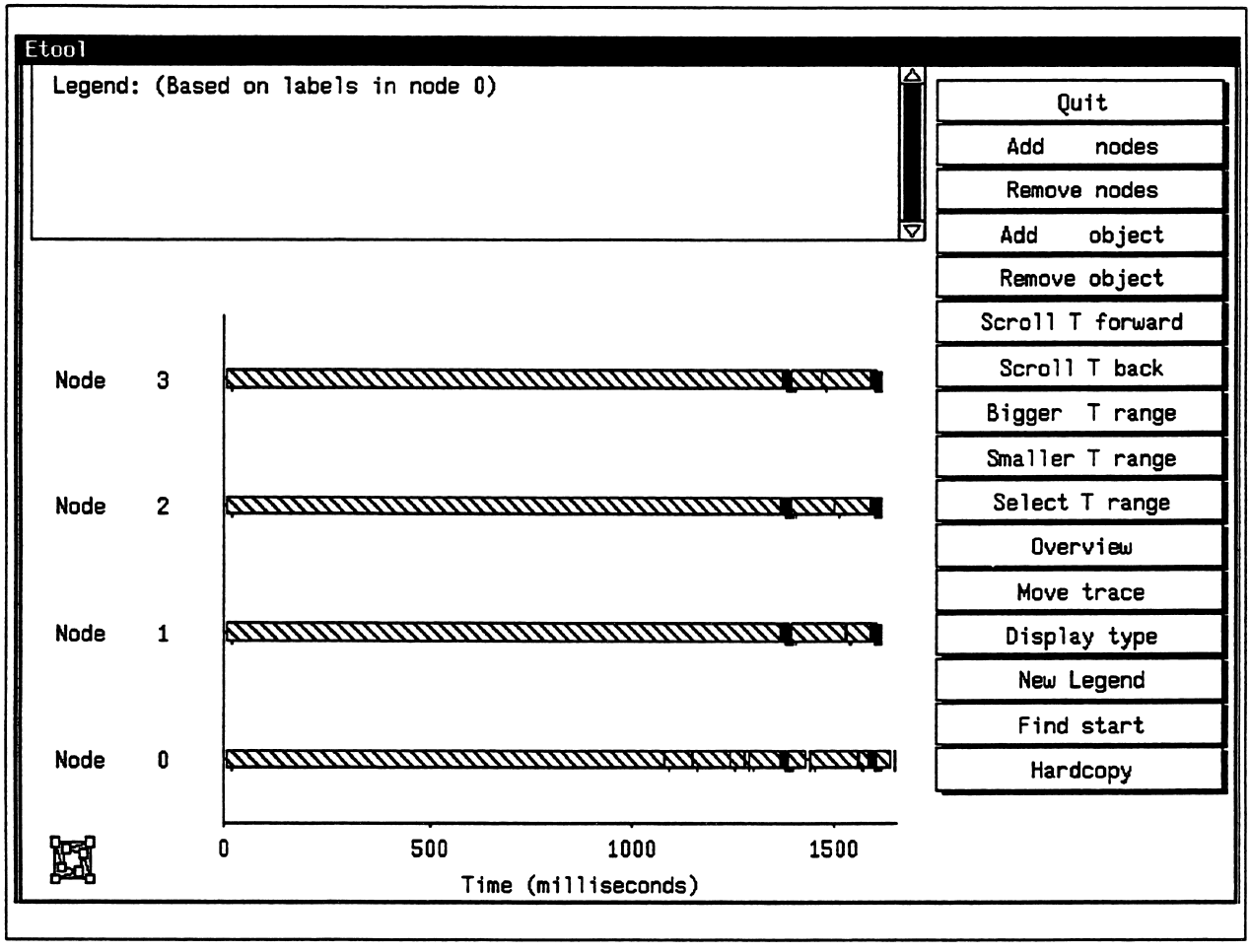

Figure 5-2. Sample Output from the "Overview" Command

These commands are provided to display various regions of interest. Another necessary ability is that of adding more processor time-lines to the display and possibly removing ones already present. By default **etool** displays the traces of the first eight processors encountered in the log-file. This situation can be altered with the **Add Nodes** and **Remove Nodes** commands. Both present a node selection menu which has the same form as that discussed in connection with the **ctool** system. One can select nodes either individually, in ranges, according to their neighbors, according to their parity, or all at once. Having selected some nodes the **Done** option takes you back to the main display menu and performs the requested action with the selected nodes. If this was **Add Nodes** then additional time-lines will be added to the display for each selected node, while the opposite **Remove Nodes** option will remove traces for selected nodes.

The option **Move Trace** is available to re-order the time-lines along the vertical axis. By default this ordering is in order of increasing processor number or selection. Occasionally, however, it is convenient to put together certain traces to better understand the relationships between processors. To do this, select **Move Trace** and you will be prompted as follows in the Dialog Area:

```
Select a trace to move
```

To do this move the cursor into the Display Area and click over a processor time-line. The prompt then changes to:

```
Select a position to move it to
```

at which point you should click in the gap between two processor traces. The display area will be updated with the selected node trace positioned between the two indicated processors.

As well as altering the various quantities displayed along the axes of the Display Area one can also modify the appearance of the time-lines themselves with the following options:

Remove Object

This option allows one to selectively disable various types of events. To disable an object click on **Remove Object**. You will be asked to click over the object you wish to delete. Move the cursor into the Display Area and click over any displayed object. After you select an object, it will no longer be displayed or eligible for detailing. When you have finished removing objects click on **Done** to return to the main menu.

Add Object This option reverses the effect of the previous choice. Upon selection, a menu is displayed containing the objects which have been removed. Click over the items you wish to enable and then on **Done** to return to the main menu and update the display.

These options allow piecemeal addition and deletion of specific events from the display. The **Display Type** option, however, allows sweeping alterations to be made to the time-line display. By default all defined events are shown. These events are represented by numbered boxes on the display lines, the numbers indicating the event index as given in the **eprof_add()** call. However, the system is also potentially logging events, particularly communication calls. This option allows the display type to include or exclude the following types of operations:

Calculation All events that cannot be assigned to one of the other operation types (such as entry/exit from the main routine and calls to math routines) are considered **Calculation** operations.

Node Comm. Events associated with node-to-node communication and data transfers are considered **Node Comm.** operations. Operations such as **_csend**, **_crecv**, and **_csendrecv** are associated with this **Display Type**.

I/O Operations associated with I/O between the iPSC system nodes and the SRM or CFS environment are considered I/O operations. Operations such as **_cread, _write,** and **_cwrite** are associated with this **Display Type.**

System calls System call operations use iPSC or UNIX system calls. In the example program used in this manual, only the **_sbrk** system call is considered.

Flick The **flick** operation occupies the majority of all time in most iPSC programs. Eliminating **flick** tracing from the display often allows you to zero in on the more interesting areas of a program.

The effect of choosing the **All** option is shown in Figure 5-3. By choosing successively smaller time ranges (**Select T range** or **Smaller T range**) you can focus on the area of a program that are of interest.

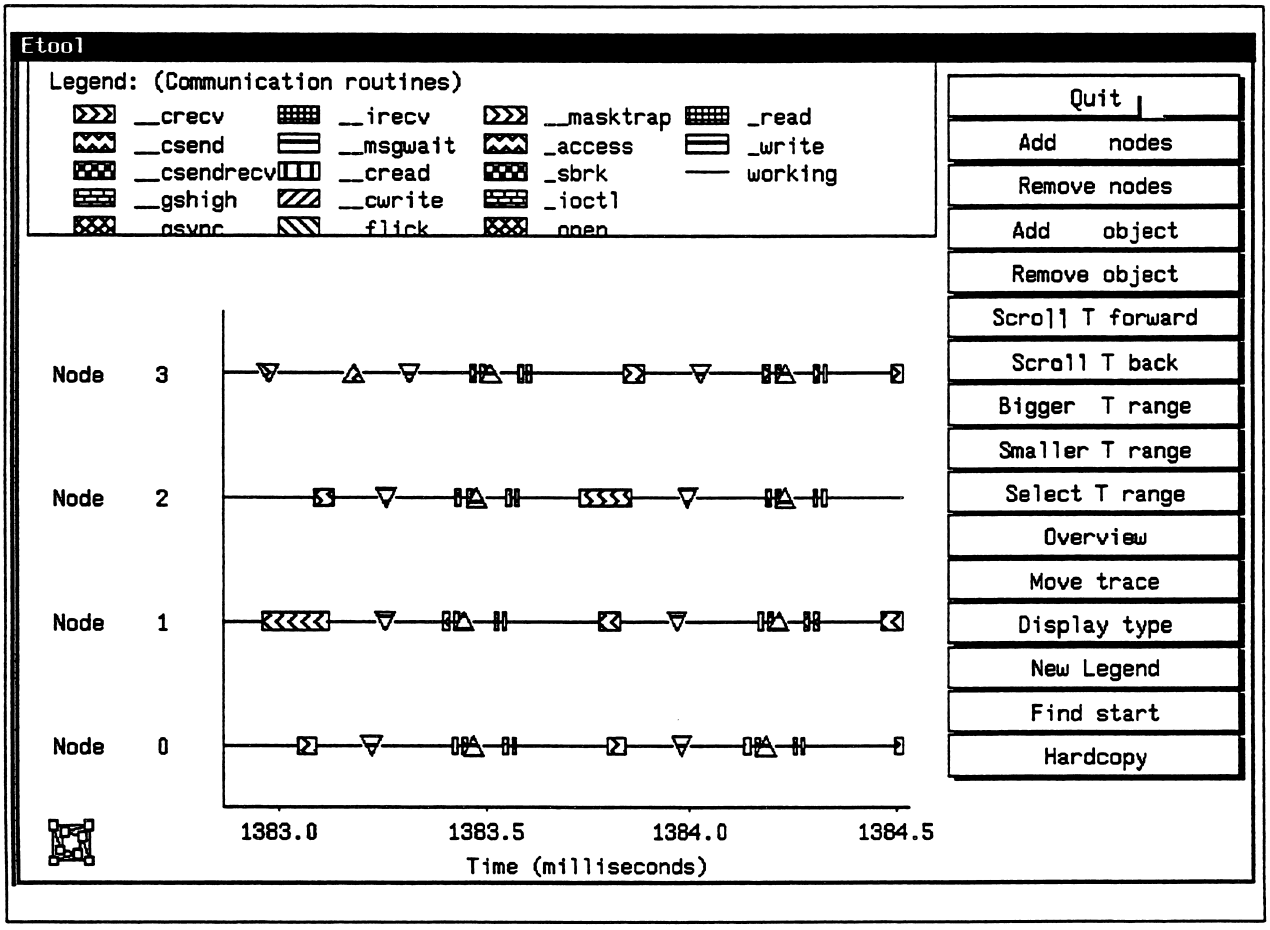

Figure 5-3. Sample Display Showing Both System and User Events

Having added extra data to the Display Area one might be interested in figuring out what they represent. To do this one invokes the **New Legend** option. By default the Legend Area shows the titles that were assigned to user events in node 0 with the **eprof_label()** system call. This information is often enough to understand all user events — their index numbers appear in the boxes on the time-lines and the associated titles appear in the Legend Area. Sometimes, however, the same event number might mean different things in different processors. While this might be classed as bad coding practice it may be unavoidable in real applications and so the **New Legend** option allows you to switch to a different node's set of titles. To do this click on **New Legend** and at the prompt:

```
Select a node or click outside the Display Area
```

move the cursor into the Display Area and click over a processor's time-line. This will immediately switch the Legend Area over to that taken from the indicated node. An alternative possibility is to click outside the Display Area completely. In this case a legend is drawn indicating the coding of the system defined communication events. An example system legend is shown in Figure 5-3.

This last command allow more information to be displayed about the events shown in the Display Area. Usually this will be enough to get a reasonable feel for the part of the application being shown. In order to access the next level of information , simply select the objects of interest in the display area with the cursor. Every time that one clicks on an event in the Display Area extra detail concerning that event appears in the Dialog Area.

For user defined events the information shown includes the index number, the exact time at which the event occurred and the user data value which was supplied to the **eprof_add()** call. The index number is provided to give confirmation that the correct event was actually selected with the cursor — this can get quite tricky in a crowded display although the commands to modify the horizontal axis can be used to alleviate a dense time-line. The time of the event is shown to allow monitoring of execution times — for example if you have an event both at the beginning and end of a function then you can use this value to find out how long it took to execute. The final value is presented either to correlate the displayed data with a point in the application or to understand the way the program is behaving. The supplied data item is processed with the format string optionally supplied in a call to **eprof_label()** and the result appears in the Dialog Area. In the first example of this section, for example, we defined labels containing `"Iteration %d"` and `"crunch returns %d"`. If we click on an event of the first type for which the supplied data value is 39 then the following might appear in the Dialog Area:

```
1. T = 246.23 ms, "Iteration 39"
```

Note that it is not essential to supply a label for an event type in order for this option to succeed. If no format string has been associated with an event then the result of the **Show Detail** click will just be:

```
2. T = 246.23 ms, value = 39
```

in which the first two fields indicate the same information as previously and the last is the value of the user-supplied data item interpreted as an integer. Obviously this is not quite as informative as would be the case if a label were supplied, but occasionally the space saving might be relevant.

While in this mode one can also click on system events. This should produce a message in the Dialog Area which looks like:

```
cread, T = 357.68ms, elapsed 127.54 ms, returned 580
```

The information supplied here is; the name of the communication function invoked (which should correspond to the system event legend if that is displayed), the time at which the communication began, the time taken for the communication to complete and the value returned to the caller. The interpretation of this last piece of information depends upon the particular communication routine invoked but is typically related to the length of the message being transmitted.

There are three basic symbol types used in the etool display: a rectangular box, an upward-pointing triangle, and a downward-pointing triangle. As shown in Figure 5-3, the pattern or color enclosed by the symbol indicates the function being traced, while the symbol shape indicated the operation that the function is performing. The symbol shapes have the following meanings:

Rectangular box

System calls and routines that have no associated call/return operations are indicated with the rectangular box. The routine itself is indicated by the rectangular box, but it is often flanked by triangular symbols that show calls and returns associated with the routine. Selecting this symbol results in a detailed information presentation above the Display Area.

Downward-pointing triangle

A call from a routine, function, or system call. The pattern or color enclosed by the symbol indicates the routine, function, or system call to which this call applies. Selecting this symbol results in a detailed information presentation above the Display Area.

Upward-pointing triangle

A return value to a routine, function, or system call. The pattern or color enclosed by the symbol indicates the routine, function, or system call to which this return value applies. Selecting this symbol results in a detailed information presentation above the Display Area.

The last two options available on the main menu are, hopefully, self-explanatory. **Quit** terminates the **etool** program and returns you to the command line prompt. **Hardcopy** makes a file which, when suitably processed and printed, will show the current state of the graphics screen, less the menus.

Having now discussed all the options available to users of **etool** the question remains: "What can I do with it?". Among the various possibilities are:

- Analysis of time taken in particular routines or pieces of code. Logging events around important code sections and subroutine calls or using toggles allows one to evaluate the time spent in various portions of code.

- Relation of time spent to data conditions. Careful specification of crucial data items as the auxiliary value in **eprof_add()** calls allows the connection to be made between program performance and data dependencies that arise on the nodes.

- Analysis of complex communication patterns and their effect on performance. Enabling the communication profiler while the event profiler is running logs extra information about internode communication.

- Analysis of interprocessor effects such as load imbalance and communication skewing. It becomes immediately apparent if one node is working much harder than the others, or if a particularly crucial communication cycle is being delayed by another processor.

- Analysis of algorithms. In non-deterministic algorithms it may be useful to understand exactly what functions are called and in what order. This can be achieved with suitable event placement. This type of information may be important in understanding the advantages or deficiencies of a particular parallel algorithm.

- Analysis of algorithmic behavior. In certain algorithms it may be important to understand the sequence of events leading to some strange behavior. A good example might be an ill-conditioned matrix problem in which the time taken for an algorithm to operate might depend on some parameter which can be logged and later related to the algorithm performance.

Analyzing the "Toggle" Data — etool -t

The event profiling analysis tool, **etool**, is also used to examine the data collected by the toggle system. To do this we execute the command

```
etool -p -t program_name
```

in which the switches indicate that no graphics should be used ('**-p**') and that toggle data should be analyzed ('**-t**'). The resulting display will appear similar to that shown in Figure 5-4.

Node 0

Description	Total	#Calls	Avge.	Var.
Main iteration loop	478.32	100	4.78	.28
Calls to grind_away	363.96	400	0.91	.03

Node 1

Description	Total	#Calls	Avge.	Var.
Main iteration loop	478.32	100	4.78	.28
Calls to grind_away	363.96	400	0.91	.03

Figure 5-4. Sample Output from the Toggle Utilities

For each toggle is presented the total time spent within that section of code, the number of times executed and mean and variance data. Also shown is the title given to the toggle in the call to **eprof_toginit()**.

Using this system it is possible to build up extremely accurate pictures of program execution.

PAT Command Reference 6

Introduction

The only commands associated with the Parallel Performance Analysis Tools utilities are the commands required for invoking the individual tools. This chapter contains full reference information on the PAT commands.

Executing Commands in "Non-windowing" Operating Systems

When running display tools under operating systems with conventional line-oriented user interfaces, commands are executed by typing their names at the command line prompt.

Usage generally follows the conventional UNIX-style with options being indicated by the '-' character, for example:

```
xtool -p toyland
```

The available options (preceded by the '-' character) are identified in the command reference pages in this chapter.

In keeping with the conventional style all commands exit with status 0 upon successful termination and with non-zero values if errors occur.

Executing Commands in "Windowing" Systems

In windowing systems such as SunView or the X Windows System, the display tools are usually executed by selecting icons from the screen. In most cases a dialog box will then be presented allowing the entry of parameters. In most cases the entries to be made have a one-to-one correspondence to the switches used in the line-oriented interfaces. Usually some mechanism is also provided to "Abort" or "Cancel" the program without executing any commands.

Specifying Numeric Data in Switches

Many of the parameters necessary to the commands listed in this section have numerical values - the number of processors to use, the number of bytes to display, and so on. In most cases these values can be entered with the usual C-style notation as either decimal, octal, or hexadecimal values. If there are any exceptions to the C-style notation for numeric data, the exceptions are identified in the text accompanying the parameter descriptions.

CTOOL CTOOL

Analyze Communication Profile.

Synopsis

ctool [**-b** *nbins*] [**-H...**] [**-p**] [**-T...**] [**-M** *menustyle*] [**-L** *DBname*] [**-m** *PSname*] *program_name* [*log_file_name*]

Domain

This command is available at the system prompt on the system holding the PAT host software. By default, this is a Sun-3 or Sun-4 workstation.

Description

This command is used to examine and analyze the log file created with the **cprof()** communication profiler system calls. The *program_name* argument is required when invoking the **ctool** command. The name of the file containing the profile data (*log_file_name*) may be omitted if it is the default file *(exprof.prc)*.

If the **-p** switch is given this command presents a separate table on *stdout* from each node. The information contained in each table is:

- An identifier showing which node the following data is from.
- A summary of the calculation, communication and I/O times in the processor. In making this classification all inter-node and basic host-node communication comes under the heading "Communication" while genuine I/O requests such as calls to **read, write, cread, cwrite,** etc. are counted as I/O.
- A summary of the time spent in, number of calls to, and errors incurred in each communication function called by the processor. This information is used to give a quick breakdown of the total communication pattern. The error count is also a good place to look for obscure bugs. Each function makes some consistency checks on the supplied arguments and returns an error if they are inconsistent.
- A breakdown of the values returned by the communication functions. The return values are binned logarithmically - the column headed "8" indicates the frequency of return values in the inclusive range 8 through 15. The exact interpretation of this data depends on the particular function being invoked but is usually related to the message length involved in the call. By default data from ten logarithmic bins is included in the output although the **'-b'** switch is provided to override this default.

CTOOL *(cont.)* CTOOL *(cont.)*

One very important use of this system is the detection of programs which are sending too much data in their messages. These will show up very clearly in the histogram output. This data appears on *stdout*.

If the **ctool** command is invoked without the **-p** switch then a graphical interface allows data to be presented in graphical form. The package is menu-driven and quite straightforward to use. A full list of the available options is presented in Chapter 4.

By default the graphical system used is either a Sun-3 or a Sun-4 workstation, along with a PostScript printer for hardcopy output. In certain circumstances these may not be the devices you wish to use. You can use the **-T** switch to redirect display output to the correct device if the default displays are not suitable. Similarly the **-H** switch is used to redirect the hardcopy output.

Options

-b *nbins* Specifies an alternate number of logarithmic bins to display when used in conjunction with the '**-p**' switch. (Default 10).

-H... Specify the type of output device required for hardcopy options. By default hardcopy output is produced in PostScript form. The interpretation of this switch is similar to that of the **-T** option.

-L *DBname* Use the named file as an alternate system call database which describes the instrumented system calls.

-M *menustyle* Use an alternative menu style. By default **ctool** attempts to use three-dimensional menus that may not function correctly on certain monitors. Substituting alternate numeric values in this switch uses other menu types of which **-M0** should be viable on any kind of monitor.

-n *PSname* The base name given to the hardcopy output files created by the system. The actual filename used is created by appending a numeric value and the string *.ps* to the name given to this switch. (Default *cprplot.*)

-p Suppress graphical output. The analysis results are presented in tabular form on stream *stdout*.

-T... Use an alternative graphical device for output. The supported graphical devices and their abbreviations can be found in AppendixA.

program_name Specifies the name of the program from which the profile data was gathered. If the program name is not in the default path, the full pathname must be given.

CTOOL *(cont.)* CTOOL *(cont.)*

log_file_name Specifies the name of the file containing the profile data that the **ctool** utility will be examining. The *log_file_name* may be omitted if it is the default file *(exprof.pre)*.

Examples

To examine the profile data (in a file called *phase3.prof*) from the program named *fft2d.ce* execute the following command:

```
ctool  fft2d.ce phase3.prof
```

To analyze the default profile data file from the program named *fft2d.ce* using simple menus under X-Windows execute the following command:

```
ctool -TX -M0  fft2d.ce
```

Errors

ctool uses an internal data-base to find the names and identifiers of the functions being profiled. If this file is missing you will see the error "`No system call data-base`". This normally means that your system is not installed properly. Either contact your system administrator or use the **-L** switch to specify an alternate path.

If the file is present but garbled or otherwise incorrect **ctool** will appear to function correctly but will either give the wrong names to subroutines or else not list anything at all.

ETOOL ETOOL

Analyze Event Profile.

Synopsis

etool [**-e**] [**-H...**] [**-L** *DBname*] [**-M** *menustyle*] [**-n** *PSname*] [**-p**] [**-t**] [**-T...**] *program_name* [*log_file_name*]

Domain

This command is available at the system prompt on the system holding the PAT host software. By default, this is a Sun-3 or Sun-4 workstation.

Description

This command is used to examine and analyze the event log created with the **eprof()** system calls. The *program_name* argument is required when invoking the **etool** command. The name of the file containing the profile data (*log_file_name*) may be omitted if it is the default file *(exprof.pre)*.

If the **etool** command is invoked without the **-p** switch then a graphical interface allows data to be presented in graphical form. The package is menu-driven and quite straightforward to use. A full list of the available options is presented in Chapter 5.

By default the graphical system used is either a Sun-3 or a Sun-4 workstation, along with a Postscript printer for hardcopy output. In certain circumstances these may not be the devices you wish to use. You can use the **-T** switch to redirect display output to the correct device if the default displays are not suitable. Similarly the **-H** switch is used to redirect the hardcopy output.

ETOOL *(cont.)* ETOOL *(cont.)*

Options

-e Suppress printing of toggles; print only events. This switch is only active when used in conjunction with the **-p** option.

-H... Specify the type of output device required for hardcopy options. By default hardcopy output is produced in PostScript form. The interpretation of this switch is similar to that of the **-T** option.

-L *DBname* Use the named file as and alternate system call database which describes the instrumented system calls.

-M *menustyle* Use an alternative menu style. By default **etool** attempts to use three-dimensional menus that may not function correctly on certain monitors. Substituting alternate numeric values in this switch uses other menu types of which **-M0** should be viable on any kind of monitor.

-n `PSname` The base name given to the hardcopy output files created by the system. The actual filename used is created by appending a numeric value and the string *.ps* to the name given to this switch. (Default *eprplot.*)

-p Suppress graphical output. The analysis results are presented in tabular form on stream *stdout.*

-T... Use an alternative graphical device for output. The supported graphical devices and their abbreviations can be found in Appendix A.

-t Display only the data from the toggle events, suppressing events. This switch is only active when used in conjunction with the **-p** switch.

program_name Specifies the name of the program from which the profile data was gathered. If the program name is not in the default path, the full pathname must be given.

log_file_name Specifies the name of the file containing the profile data that the **etool** utility will be examining. The *log_file_name* may be omitted if it is the default file *(exprof.pre)*.

ETOOL *(cont.)* ETOOL *(cont.)*

Examples

To examine the profile data in a file called *phase3.prof* execute the command:

```
etool fft2d.ce phase3.prof
```

To analyze the default data file using simple menus under X-Windows execute the command:

```
etool -TX -M0 fft2d.ce
```

If only the toggle information is required from the profile data use a command similar to:

```
etool -p -t fft2d.ce toggle.pre
```

Errors

etool uses an internal data-base to find the names and identifiers of the functions being profiled. If this file is missing you will see the error "`No system call data-base`". This normally means that your system is not installed properly. Either contact your system administrator or use the **-L** switch to specify an alternate path.

If the file is present but garbled or otherwise incorrect **etool** will appear to function correctly but will either give the wrong names to subroutines or else not list anything at all.

XTOOL XTOOL

Analyze Execution Profile.

Synopsis

xtool [**-H...**] [**-i**] [**-I** *directory*] [**-M** *menustyle*]
[**-n** *PSname*] [**-p**] [**-t** *topno*] [**-T...**]
program_name [*log_file_name*]

Domain

This command is available at the system prompt on the system holding the PAT host software. By default, this is a Sun-3 or Sun-4 workstation.

Description

This command is used to examine and analyze the log file created with the execution profiler, **xprof()**, system calls. The first argument is the name of the executable program to be profiled and the second is the name of the file containing the profile data. The second argument may be omitted if it has the default value *exprof.prx*. Note that the execution profiler relies on data contained in a symbol table for correct functioning. This normally happens when the application is compiled using the **-Mperf** compile switch.

This command presents a separate table on *stdout* for each node. The information contained in each table is:

- An identifier showing the node that supplied the following data.
- A summary of the busy and idle time in each processor. In this regard we measure CPU time so that the only idle time is when the CPU is not actively executing the process such as when waiting for a message to arrive. All other classes of activity are counted as busy. Note that this interpretation is different from that of **ctool** which distinguishes between calculation and communication time.
- Because the buffer supplied to the profiling function **profil()** may not be large enough to encapsulate the entire program it is possible that the execution profiler will miss occasionally. For example, the program will be executing at an address which lies outside the region mapped by the **profil()** call when it tries to log the profile event.
- A profiling list containing the most heavily used 20 functions in the program. Each shows the fraction of the total profiling events that it corresponds to.

XTOOL *(cont.)* XTOOL *(cont.)*

This data appears on *stdout*.

If the **xtool** command is invoked without the **-p** switch, it is presented in graphical form. The package is menu-driven and quite straightforward to use. A full list of the available options is presented in Chapter 3.

By default the graphical system used is either a Sun-3 or a Sun-4 workstation running Sunview, along with a PostScript printer for hardcopy output. In certain circumstances these may not be the devices you wish to use. You can use the **-T** switch to redirect display output to the correct device if the default displays are not suitable. Similarly the **-H** switch is used to redirect the hardcopy output.

Options

-H... Specify the type of output device required for hardcopy options. By default hardcopy output is produced in PostScript form. The interpretation of this switch is similar to that of the **-T** option.

-i After processing other command line options enter interactive mode as described below. This switch is only active when used in conjunction with the **-p** switch.

-I *directory* Causes the named directory to be searched when looking for the source code to the various routines in the program. By default only the current directory is searched.

-M *menustyle* Use an alternative menu style. By default **xtool** attempts to use three-dimensional menus that may not function correctly on certain monitors. Substituting alternate numeric values in this switch uses other menu types. An **-M0** designation is normally viable on any kind of monitor.

-n *PSname* The base name given to the hardcopy output files created by the system. The actual filename used is created by appending a numeric value and the string *.ps* to the name given to this switch. (Default *xprplot.*)

-p Suppress graphical output. The analysis results are presented in tabular form on stream *stdout.*

-t *topno* Instructs **xtool** to print data from the *topno* most active routines. (Default 20.) This switch is only active when used in conjunction with the **-p** switch.

-T... Use an alternative graphical device for output. The supported graphical devices and their abbreviations can be found in Appendix A.

XTOOL *(cont.)* XTOOL *(cont.)*

Examples

To examine the profile data in a file called *phase3.prof* created by the program **master** execute the following command:

```
xtool master phase3.prof
```

To analyze the default data file using simple menus under X-Windows execute the following command:

```
xtool -TX -M0 myprog
```

If the graphical mode is not available on your system a certain amount of analysis can be performed using the interactive mode. To enter this mode use a command such as:

```
xtool -p -i progname
```

After completing normal **-p** processing you will see the following prompt:

```
xtool>
```

At this prompt the following commands may be executed:

help Prints a summary of available commands.

top # Prints the normal tabular output showing only the indicated number of routines for each node.

quit Terminate **xtool** and return to the shell.

node # Restrict output to the particular node indicated. Use the value **-1** to indicate output from all nodes.

output file Redirect printed output to the named file. Using the command with no file name redirects output to the terminal.

Typing any other word causes **xtool** to attempt to print the histogram information for a function with that name. This can be used in conjunction with the source line numbering information (added to the binary executable by the **-g** linker switch) to match the source code to the execution profile.

C System Calls 7

Introduction

This section of the manual is devoted to a listing of the contents of the subroutine library used for manual performance monitoring.

Synchronization

During initialization and dumping of profiling information all nodes are loosely synchronized. A "loosely synchronous" system call can be perceived as a barrier to the further progress of the program. When one node makes a loosely synchronous call it waits for all other nodes to make the *same* system call (albeit with possibly different arguments). When all nodes have made the call every node proceeds. This concept might be classed "synchronous" but this is too restrictive - it is quite permissible for one node to make the "loosely synchronous" call far ahead of the other nodes. All nodes will, however, be synchronized *after* the call completes.

C Header Files and Macros

Central to the use of the PAT utilities is the C header file *expm.h* which should be included whenever manual performance monitoring functions are being used. This file defines a number of important parameters which have wide usage in the system.

DONTCARE This macro value is used to indicate the default number of logs and labels used for event tracing.

ETOGGLE The "toggle" structure used by the event profiler to measure CPU time and other statistics in indicated program segments.

CPROF_DMP CPROF_DMP

Dump communication profile data to disk from the application program.

Synopsis

cprof_dmp(*filename***)**

Parameter Declarations

char ***filename*;**

Description

This routine is used to write communication profile data to a disk file for later analysis with the **ctool** command. The single argument is the name of the file to which profile data should be written. At the conclusion of the call the communication profiler is disabled as though by a call to **cprof_off()** and its internal state is reset to zero so that further profiling leads to distinct, non-overlapping data.

A call to this function with the filename *exprof.prc* is automatically generated during the program exit if **cprof_inq()** detects that the profiler is enabled at runtime. Because of this, most programs need never use this routine explicitly unless fine control is required over the location and timing of the profile dumps.

Example

The following code is a skeleton of that which might be used to control the communication profiler.

```
main()
{
/* Start off profiler */

   if ( cprof_inq() )cprof_on();

/* Application Phase 1., profiler running */

   ...
```

CPROF_DMP *(cont.)*

CPROF_DMP *(cont.)*

```
/* Phase 1 complete, dump data with cprof_dmp. */

   if(cprof_inq()) cprof_dmp("phase1.prc");

/* Application Phase 2., profiler off since cprof_dmp called */

   ...

/* Application phase 3., turn on profiler again */

   if( cprof_inq() ) cprof_on();
   ...

/* Program over, dump data again and exit */

   if( cprof_inq() ) cprof_dmp("phase3.prc");
   exit(0);
}
```

Notice that because we can selectively profile pieces of code, it sometimes makes sense to collect profiling information about different phases of an algorithm in separate files. In this case multiple calls to **cprof_dmp()** can be made to write out the data from the individual sections.

Fortran Synopsis

SUBROUTINE **KCPDMP**(*fname*)
CHARACTER*80 *fname*

See Also

ctool (command), **cprof()**, **cprof_inq()**

CPROF_INQ CPROF_INQ

Determine runtime status of profiling system.

Synopsis

cprof_inq()

Description

The **cprof_inq()** function first checks the environment variable **EXPROF_SWITCHES**. If the environment variable exists **cprof_inq()** returns an integer value representing the occurrence or non-occurrence of the character '**c**' in the environment variable. If the **EXPROF_SWITCHES** environment variable does not exist the **cprof_inq()** function returns the value of a global variable indicating whether the **auto** or **manual** switch was used on the **-Mperf** link directives. The **cprof_inq()** function is most commonly used to determine at runtime whether or not to enable the communication profiling system.

The performance monitoring linked into the application performs a check for the communication monitoring system during its start-up code. This is used to allow automatic configuration of the communication profiling system. If enabled the system will also automatically generate a call to **cprof_dmp()** upon program termination.

As a result a typical application need contain no explicit calls to the communication profiling routines - they are all made by the performance monitoring code. The only case in which such calls are needed is when more careful control is required over the profiler and the data it dumps.

Example

The following code can be used to control the communication profiler.

```
main()
{
/* Start off profiler */

   if(cprof_inq()) cprof_on();

/* Application Phase 1., profiler running */

   ...
```

CPROF_INQ *(cont.)*

CPROF_INQ *(cont.)*

```
/* Phase 1 complete, dump data with cprof_dmp */

   if(cprof_inq()) cprof_dmp("phase1.prc");

/*  Application Phase 2., profiler off since cprof_dmp called */
   ...

/* Application phase 3., turn on profiler again */

   if(cprof_inq()) cprof_on();
   ...

/* Program over, dump data again and exit */

   if(cprof_inq()) cprof_dmp("phase3.prc");

   exit(0);
}
```

Given this code we can turn the profiler on or off according to the setting of the environment variable "**EXPROF_SWITCHES**". If set to some string containing the letter '**c**' this code will perform profiling. Otherwise no profiling will be done.

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

Fortran Synopsis

INTEGER FUNCTION **KCPINQ**()

See Also

ctool (command), **cprof**()

CPROF CPROF

Control communication profiler.

Synopsis

cprof_on()

cprof_off()

Description

The **cprof_on()** call is used to enable and start the communication profiler. After this call all subsequent calls to the communication system result in entries being made in an internal log-file. The **cprof_off()** call reverses this process. Until a subsequent call to **cprof_on()** no communication profiling will be performed.

The log of profiling information is written to the host file system with **cprof_dmp()**.

Example

The following code can be used to control the communication profiler.

```
main()
{
/* Start off profiler */

   cprof_on();

/* Application Phase 1., profiler running */

   ...
```

CPROF_ON, CPROF_OFF *(cont.)*

CPROF_ON, CPROF_OFF *(cont.)*

```
/*
 * Phase 1 complete, dump data with cprof_dmp
 */

   cprof_dump( "phase1.prc" );

/* Application Phase 2., profiler turned off by previous
 * call to cprof_dmp
 */
   ...

/* Application phase 3., turn on profiler again */

   cprof_on();
   ...

/* Program over, dump data again and exit */

    ...
   exit(0);
}
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

See Also

ctool (command), **cprof_dmp**()

EPROF EPROF

Event driven profiler.

Synopsis

eprof_on()

eprof_off()

eprof_init(*numlog, numlab***)**

eprof_label(*index, title, format_str***)**

eprof_add(*index, datum***)**

Parameter Declarations

int *numlog, numlab*;

int *index*;
char **title, *format_str*;

int *index*;
int *datum*;

Description

These routines make up the interface to the user specified event driven profiling facility. The **eprof_on()** and **eprof_off()** functions enable and disable the event profiler respectively. While disabled, no events are logged even if calls are made to **eprof_add()**.

The routine **eprof_init()** must be called before any of the other profiling calls. The arguments indicate the amount of space (number of bytes) to reserve for "title" and "event" entries - corresponding to each call to the **eprof_label()** and **eprof_add()** functions. The special value **DONTCARE** may be given for either argument indicating that a system selected default should be used. The current overhead allowances for log entries and labels are 24 and 68 bytes respectively.

The header file *expm.h* should be included whenever the manual performance monitoring functions are being used. This file defines important parameters (such as **DONTCARE**) that are widely used in the system.

EPROF *(cont.)*

EPROF *(cont.)*

The **eprof_add()** call is the heart of the event system. It makes a new entry in the log file. Three items are logged; the event *index* and *datum* as given in the function call and the time at which the call is made. The *index* argument is used to differentiate between events at the highest level. This index corresponds to an optional **title** string defined in a call to **eprof_label()**. The *datum* argument is used to identify events at the lowest level. This value is any 32-bit integer value which will be used in conjunction with an optional *format_str* argument defined in a call to **eprof_label()**.

The function **eprof_label()** is used to facilitate event recognition when the log file is subsequently analyzed. Its use is optional. If no calls to **eprof_label()** are made then events will be identified by their *index* argument in the subsequent analysis and the *datum* value will be assumed to be an integer. Making the following call:

```
eprof_label(3, "After return from crunch_func", "Energy = %d");
```

builds in extra information. Together with the event *index* a legend will be presented which connects type 3 with the string "`After return from crunch_func`". Further, when the value of the *datum* argument is shown it will be formatted according to the format string, the following result would be typical:

```
Energy = 23
```

The log of profiling information is written with the **eprof_dmp()** function.

Example

The following is a skeleton of the code that might typically be used to control the event profiler.

```
#include "expm.h"

main()
{
   float Energy, resid, grind(), crunch();
   int iter, i;

/* Start profiler, make labels for indices 1-3, use default sizes
*/

   eprof_init(DONTCARE, DONTCARE);
   eprof_label(1, "Outer loop", "Iteration %d");
   eprof_label(2, "After crunch", "Energy = %f");
   eprof_label(3, "Inner loop", "resid = %f");
   eprof_on();
```

EPROF *(cont.)*

EPROF *(cont.)*

```
/* Start application code, then go into main loop */

   ...

   for(iter=0; iter<100; iter++) {
      eprof_add(1, iter);
      Energy = crunch(iter);

      eprof_add(2, (int)Energy);

      for(i=0; i<4; i++) {
         resid = grind(Energy);
         eprof_add(3, (int)resid);
      }
   }

/* Program over, dump profile data and exit */

    ...
   exit(0);
}
```

The insertion of events like these above can provide significant information about an application. The time between events 1 and 2, for example, indicates the duration of a call to the **crunch** function. Similar information is available about **grind** from events 2 and 3, averaged over the four calls per iteration. The auxiliary *datum* fields will show the interaction between the variables and the program execution rate. It may also show up bugs and/or unexpected behavior which could provide the key to understanding the failings of a particular parallelization scheme.

EPROF *(cont.)* EPROF *(cont.)*

Fortran Synopsis

SUBROUTINE **KEPON**()

SUBROUTINE **KEPOFF**()

SUBROUTINE **KEPINI**(*labbuf, labsiz, logbuf, logsiz*)
INTEGER *labbuf*(*), *labsiz*, *logbuf*(*), *logsiz*

SUBROUTINE **KEPLAB**(*index, title, format*)
INTEGER *index*
CHARACTER*80 *title, format*

SUBROUTINE **KEPADD**(*index, datum*)
INTEGER *index*
INTEGER*4 *datum*

Fortran Description

The Fortran equivalents of the above C functions operate in the obvious way with the exception of the **KEPLAB**() and **KEPINI**() subroutines. The former serves to initialize the event profiling system. The user provides two workspace buffers, *labbuf* and *logbuf* for storing labels and log entries respectively. The size, in bytes, of each buffer is given by the following *siz* parameter. As a guide to appropriate sizes a label entry currently requires 68 bytes while a log entry needs 12.

The **KEPLAB**() function assigns a label to a user "event". The format string which must be provided must be in the notation of the C function **printf**. While the details are complex one only needs to note that the string is printed as specified except that the special sequence **%d** is replaced by the value of the *datum* argument.

A suitable Fortran call which corresponds to that shown earlier in C is as follows:

```
CALL KEPLAB(3, 'After return from crunch_func','Energy = %d')
```

See Also

etool (command), **eprof_dmp**()

EPROF_DMP EPROF_DMP

Dump event profile data to disk from an application program.

Synopsis

eprof_dmp(*filename*)

Parameter Declarations

char **filename*;

Description

This routine is used to write event tracing data to a disk file for later analysis with the **etool** command. The single argument is the name of the file to which profile data should be written. At the conclusion of the call, the event profiler is disabled as though by a call to **eprof_off()** and its internal state is reset to zero so that further profiling leads to distinct, non-overlapping data.

A call to this function with the filename *exprof.pre* is automatically generated during the program exit if **eprof_inq()** detects that the profiler is enabled at runtime. Because of this, most programs need never use this routine explicitly unless fine control is required over the location and timing of the profile dumps.

Example

The following code is a skeleton of that which might typically be used to control the event profiler.

```
#include "expm.h"
ETOGGLE mytog;
int eprof_is_on = 0;

main()
{
   eprof_init(DONTCARE, DONTCARE);
   eprof_label(1, "Outer Loop", "Iteration %d");
   eprof_toginit(mytog, "Timing inner loop");

   if(eprof_inq()) {
      eprof_is_on = 1;
   }
```

EPROF_DMP *(cont.)*

EPROF_DMP *(cont.)*

```
/* Start first part of program using stuff initialized above  */
        .....

/*
 * First phase is over. If the profiler was enabled, dump
 * the data to a file so that we can restart afresh ....
 */
   if(eprof_is_on) eprof_dmp("phase1.pre");
/*
 * Start the second phases of the program with everything
 * reinitialized
 */
   ...
/*
 * When program finally finishes we can let the call to
 * "exit" take care of dumping any data that might be left over.
 */
   exit(0);
}
```

Notice that the **eprof_add()** and **eprof_label()** calls are completely safe even if **eprof_inq()** returns 0 and the profiler is not enabled.

Fortran Synopsis

SUBROUTINE **KEPDMP**(*fname*)
CHARACTER*80 *fname*

See Also

etool (command), **eprof()**, **eprof_inq()**

EPROF_INQ EPROF_INQ

Determine runtime state of the event profiling system.

Synopsis

eprof_inq()

Description

The eprof_inq() function first checks the environment variable EXPROF_SWITCHES. If the environment variable exists eprof_inq() returns an integer value representing the occurrence or non-occurrence of the character 'e' in the environment variable. If the **EXPROF_SWITCHES** environment variable does not exist the **eprof_inq()** function returns the value of a global variable indicating whether the **auto** or **manual** switch was used on the **-Mperf** link directives. The **eprof_inq()** function is most commonly used to determine (at runtime) whether or not to enable the communication profiling system.

The performance monitoring linked into the application performs a check for the event monitoring switch in its startup code and, if present, turns on the profiler with a call to **eprof_init()** and **eprof_on()**. If enabled the system will also automatically generate a call to **eprof_dmp()** at program termination. As a result such applications need only contain explicit calls to **eprof_add()** and **eprof_label()** - all control functions are performed by the performance monitoring code. The only case in which such calls are needed is when more careful control is required over the profiler and the data it dumps.

Example

The following code is a skeleton of that which might typically be used to control the event profiler.

```
#include "expm.h"
ETOGGLE mytog;
int eprof_is_on = 0;

main()
{
   eprof_init(DONTCARE, DONTCARE);
   eprof_label(1, "Outer Loop", "Iteration %d");
   eprof_toginit(mytog, "Timing inner loop");

   if(eprof_inq()) eprof_is_on = 1;
```

EPROF_INQ *(cont.)*

EPROF_INQ *(cont.)*

```
/*
 * Start first part of program using stuff initialized above.
 */
   .....

/*
 * First phase is over. If the profiler was enabled, dump
 * the data to a file so that we can restart afresh ....
 */
   if(eprof_is_on) eprof_dmp("phase1.pre");

/*
 * Start the second phases of the program with everything
 * reinitialized
 */
   ...

/*
 * When program finally finishes we can let the call to
 * "exit" take care of dumping any data that might be left over.
 */
   exit(0);
}
```

Fortran Synopsis

INTEGER FUNCTION KEPINQ()

See Also

etool (command), **eprof()**

EPROF_TOGGLE EPROF_TOGGLE

Statistical analysis of code sections.

Synopsis

eprof_toginit(*togptr*, *label*)

eprof_toggle(*togptr*)

Parameter Declarations

ETOGGLE **togptr*; /* *structure defined in expm.h* */
char **label*;

Description

These routines allow selective analysis of particular sections of code. By surrounding code segments with calls to the **eprof_toggle**() one can obtain statistics relating to the number of times the particular code section was called and the average and total times spent in these sections. The data is collected in exactly the same manner as the "event profiling" information obtained through calls to **eprof_add**(). The same commands are available to dump the profiling data as are used by the other "**eprof**" utilities.

Each toggle data structure must be initialized with a call to **eprof_toginit**() before it can be used for data collection. This function expects to be passed two values. The first value is a pointer to a structure of type ETOGGLE, defined in the C header file *expm.h*. The second value is a string that will later be used to identify the collected statistics when analyzed with **etool**.

The log of profiling information is written to the host file system by the **eprof_dmp**() function.

EPROF_TOGINIT, EPROF_TOGGLE *(cont)*

Example

The following example demonstrates the use of the toggle ideas.

```
#include "expm.h"

ETOGGLE looptog, grindtog;

main()
{
   float Energy, grind_away();
   int iter, i;

/* Initialize toggle data structures. */

   eprof_toginit(&looptog, "Main iteration loop");
   eprof_toginit(&grindtog, "Calls to grind_away");

/* Start application code, then go into main loop */

   ...

   for(iter=0; iter<100; iter++) {
      eprof_toggle(&looptog);

         /* Other processing going on here.... */

            ........


      for(i=0; i<4; i++) {
         eprof_toggle(&grindtog);
         grind_away(Energy, i);
         eprof_toggle(&grindtog);
      }
      eprof_toggle(&looptog);
   }
/*
 * Dump data to host for later analysis.....
 */
   .........

   exit(0);
}
```

EPROF_TOGINIT, EPROF_TOGGLE *(cont)*

The toggle data will be stored in a file with the name *exprof.pre* (unless overridden by some other function call) together with the normal "event" data which may have also been collected with calls to **eprof_add**().

To analyze the data for program *fft2d*, execute the **etool** command with:

```
etool -p -t fft2d
```

This combination of switches both suppresses the normal graphical output and also restricts attention to the toggle data.

The list of initialized toggles for each node is displayed, together with the number of times each code section was used, the total time elapsed in this section, the average time per call, and the variance of these times. Using this information it is possible to build up a very accurate picture of the performance of a parallel program.

NOTE

The **eprof_dmp()** call clears all label and toggle information. Any new labels must be defined if related information is desired.

Fortran Synopsis

SUBROUTINE **KEPTGI**(*toggle, text*)
INTEGER *toggle*(16)
CHARACTER*80 *text*

SUBROUTINE **KEPTOG**(*toggle*)
INTEGER *toggle*(16)

Fortran Description

The Fortran equivalents of the above C functions operate in the obvious way with the exception that no header file is available in Fortran to define the toggle data structure. Instead Fortran programs should use arrays of 16 integer values as the appropriate variables.

See Also

etool (command), **eprof_dmp**()

XPROF_DMP XPROF_DMP

Dump execution profile data to disk

Synopsis

xprof_dmp(*filename***)**

Parameter Declarations

char ***filename**;

Description

This routine is used to write execution profile data to a disk file for later analysis with the **xtool** command. The single argument is the name of the file to which profile data should be written. At the conclusion of the call the execution profiler is disabled as though by a call to **xprof_off()** and its internal state reset to zero so that further profiling leads to distinct, non-overlapping data.

A call to this function with the filename "*exprof.prx*" is automatically generated during the program exit if **xprof_inq()** detects that the profiler is enabled at runtime. Because of this most programs need never use this routine explicitly unless fine control is required over the location and timing of the profile dumps.

Example

The following code is a skeleton of that which might typically be used to control the execution profiler.

```
#define PROFSIZE (8192)
#define PROFSCALE (0x4000)
int profbuf[PROFSIZE];

extern int myfunc();              /* Low address for profiling */
                                 /* Found from the compiler map */
main()
{
/* Start off profiler */

if( xprof_inq() ) {
   xprof_init( profbuf, sizeof(profbuf),myfunc, PROFSCALE );
      xprof_on();
   }
```

XPROF_DMP *(cont.)* XPROF_DMP *(cont.)*

```
/* Application Phase 1., profiler running */

   ...

/* Phase 1 complete, dump data with xprof_dmp */

   if(xprof_inq())xprof_dmp("phase1.prx");

/* Application Phase 2., profiler off since xprof_dmp called */

   ...

/* Application phase 3., turn on profiler again */

   if(xprof_inq()) xprof_on();
   ...

/* Program over, dump data again and exit */

   if(xprof_inq()) xprof_dmp("phase3.prx");
   exit(0);
}
```

Notice that since we can selectively profile pieces of code it sometimes makes sense to write out the profile data from separate program segments independently to separate files for simplicity in later analysis. In this case multiple calls to **xprof_dmp()** are used to create the profile files.

Fortran Synopsis

SUBROUTINE **KXPDMP**(*fname*)
CHARACTER*80 *fname*

See Also

xtool (command), **xprof_init()**, **xprof()**

XPROF_INIT XPROF_INIT

Low level execution profiler.

Synopsis

xprof_init(*buffer, buflen, start, scale*)

Parameter Declarations

char **buffer*;
void **start*;
int *buflen, scale*;

Description

This routine initializes the execution profiler. Every 10 milliseconds the program counter of the user application is examined and a histogram entry in the memory area denoted by *buffer* is incremented. The size of the histogram is *buflen* bytes and its first bin starts at the address specified as *start* — normally the name of some program subroutine.

In order to decide which histogram entry to increment a mapping function is applied to the program counter discovered by the system. Note that the i860 instructions are each four bytes long. First *start* is subtracted and then the result is multiplied by *scale* and divided by 0x10000. The complete mapping function is as follows:

```
bin_number = (PC - start)*scale/0x10000
```

The overall effect of the *scale* parameter is to map groups of adjacent program locations into the same histogram bin. A program location is a two-byte value. The value `scale = 0x10000` maps every program location into a separate histogram bin, `scale = 0x8000` maps each pair of program locations into a single bin, `scale = 0x4000` every group of four, and so on.

Using combinations of the *buflen*, *start*, and *scale* parameters it is possible to allocate various memory ranges to be profiled. Note that no errors are incurred if the range is not large enough resulting in a calculated `bin_number` which is out of the histogram range.

The **xprof_init**() call does not enable the profiler. An explicit call to **xprof_on**() must be made to begin gathering profile data.

XPROF_INIT *(cont.)* XPROF_INIT *(cont.)*

Example

The following code is a skeleton of that which might typically be used to control the execution profile.

```
#define PROFSIZE (8192)          /* Size of profiler buffer */
#define PROFSCALE (0x2000)       /* Map eight bytes per bin */
int profbuf[PROFSIZE];

extern int myfunc();           /* Low address for profiling */
                              /* Found from the compiler map */
main()
{
/* Start off profiler */

   xprof_init( profbuf, sizeof(profbuf),myfunc, PROFSCALE );
   xprof_on();

/* Application Phase 1., profiler running */

      ...
```

The choice of the *start* argument is most conveniently made in conjunction with the linker map provided by the linker. This usually contains a list of the addresses of all the functions in an application. The address list can be used to find the minimum address.

Fortran Synopsis

SUBROUTINE **KXPINI**(*prbuf, prlen, start, scale*)
INTEGER *prbuf*(*), *prlen, start, scale*
EXTERNAL *start*

See Also

xtool (command), **xprof()**

XPROF_INQ XPROF_INQ

Determine runtime status of execution profiler.

Synopsis

xprof_inq()

Description

The xprof_inq() function first checks the environment variable EXPROF_SWITCHES. If the environment variable exists, xprof_inq() returns an integer value representing the occurrence or non-occurrence of the character 'x' in the environment variable. If the **EXPROF_SWITCHES** environment variable does not exist, the **xprof_inq()** function returns the value of a global variable indicating whether the **auto** or **manual** switch was used on the **-Mperf** link directives. The **xprof_inq()** function is most commonly used to determine (at runtime) whether or not to enable the execution profiling.

The performance monitoring linked into the application performs a check for the execution profiling during its startup code. This is used to allow automatic configuration of the execution profiling system. If enabled the system will also automatically generate a call to **xprof_dmp()** upon program termination.

Example

The following code is a skeleton of that which might typically be used to control the execution profiler. All the necessary calls are shown to setup and run the profiler.

```
#define PROFSIZE (8192)
#define PROFSCALE (0x4000)
int profbuf[PROFSIZE];

extern int myfunc();              /* Low address for profiling */
                                  /* Found from the compiler map */
main()
{
/* Start off profiler */

   if(xprof_inq()) {
      xprof_init(profbuf, sizeof(profbuf),myfunc, PROFSCALE);
      xprof_on();
   }
```

XPROF_INQ *(cont.)* XPROF_INQ *(cont.)*

```
/* Application Phase 1., profiler running */

   ...

/* Phase 1 complete, dump data with xprof_dmp */

   if(xprof_inq()) xprof_dmp("phase1.prx");

/* Application Phase 2., profiler off since xprof_dmp called */

   ...

/* Application phase 3., turn on profiler again */

   if(xprof_inq()) xprof_on();
   ...

/* Program over, dump data again and exit */

   if(xprof_inq()) xprof_dmp("phase3.prx");
   exit(0);
}
```

Note that profiling will only be performed if **xprof_inq()** returns a non-zero value. In this way the behavior of the code can be deferred till runtime, a much more flexible approach than during compilation.

Notice also that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

Fortran Synopsis

INTEGER FUNCTION **KXPINQ()**

See Also

xtool (command), **xprof_init()**, **xprofcp()**, **xprof()**

XPROF XPROF

Control execution profiler.

Synopsis

xprof_on()

xprof_off()

Description

The **xprof_on()** call is used to enable and start the execution profiler which must have been previously initialized with a call to **xprof_init()**. Subsequently a periodically scheduled event occurs which causes the program counter of the user application to be "logged" in an internal structure. The **xprof_off()** call reverses this process. Until there is a subsequent call to **xprof_on()** no execution profiling will be performed.

If the application was compiled with **-Mperf=manual**, the profiler is initially off and must be explicitly enabled with calls to **xprof_init()** and **xprof_on()**.

The log of profiling information is written to the host file system by the **xprof_dmp()** function. When the profiler is enabled, application subroutine calls are also logged.

Example

The following code is a skeleton of that which might typically be used to control the execution profiler.

```
#define PROFSIZE (8192)
#define PROFSCALE (0x4000)
int profbuf[PROFSIZE];

extern int myfunc();            /* Low address for profiling */
                               /* Found from the compiler map */
main()
{
/* Start off profiler */

   xprof_init(profbuf, sizeof(profbuf), myfunc, PROFSCALE);
   xprof_on();
```

XPROF_ON, XPROF_OFF *(cont.)*

XPROF_ON, XPROF_OFF *(cont.)*

```
/* Application Phase 1., profiler running */

   ...

/* Phase 1 complete, dump data with xprof_dmp */
   xprof_dump( "phase1.prx" );
   ...

/* Application Phase 2., profiler off since data dumped */

   ...

/* Application phase 3., turn on profiler again */

   xprof_on();
   ...

/* Program over, dump data again and exit */

    ...
   exit(0);
}
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

Fortran Synopsis

SUBROUTINE **KXPON**

SUBROUTINE **KXPOFF**

See Also

xtool (command), **xprof_init()**, **xprof_dmp()**

Fortran System Calls 8

Introduction

This section of the manual is devoted to a listing of the contents of the Fortran subroutines used for manual performance monitoring.

Synchronization

During initialization and dumping of profiling information all nodes are loosely synchronized. A "loosely synchronous" system call can be perceived as a barrier to the further progress of the program. When one node makes a loosely synchronous call it waits for all other nodes to make the *same* system call (albeit with possibly different arguments). When all nodes have made the call every node proceeds. This concept might be classed synchronous but this is too restrictive. It is quite permissible for one node to make the "loosely synchronous" call far ahead of the other nodes. All nodes will, however, be synchronized *after* the call completes.

KCPDMP KCPDMP

Write communication profile data to a file.

Synopsis

SUBROUTINE **KCPDMP**(*fname*)

Parameter Declarations

CHARACTER*80 *fname*

Description

This routine is used to write communication profile data to a disk file for later analysis with the **ctool** command. The single argument is the name of the file to which profile data should be written. At the conclusion of the call the communication profiler is disabled by a call to **KCPOFF()** and its internal state is reset to zero so that further profiling leads to distinct, non-overlapping data.

A call to this function with the filename *exprof.prc* is automatically generated during the program exit if **KCPINQ()** detects that the profiler is enabled at runtime. Because of this automatic operation, most programs need never use this routine explicitly unless fine control is required over the location and timing of the profile dumps.

KCPDMP *(cont.)*

KCPDMP *(cont.)*

Example

The following code is a skeleton of that which might be used to control the communication profiler in an application program.

```
      PROGRAM CBXPRF
C
C-- Start off profiler. This code is not strictly
C-- necessary since it is equivalent to the check
C-- made automatically by Performance Monitoring code.
C
      ISTAT = KCPINQ()
      IF(ISTAT .NE. 0) CALL KCPON
C
      ...
C
C-- Program over, dump data again and exit. Again
C-- this code is superfluous since it duplicates the
C-- action of Performance Monitoring code.
C
      IF(ISTAT .NE. 0) CALL KCPDMP('exprof.prc')
      STOP
      END
```

Notice that because we can selectively profile pieces of code, it sometimes makes sense to collect profiling information about different phases of an algorithm in separate files. In this case multiple calls to **KCPDMP()** can be made to write out the data from the individual sections.

See Also

ctool (command), **KCPCP()**, **KCPROF()**

KCPINQ KCPINQ

Determine runtime status of execution profiler.

Synopsis

INTEGER FUNCTION **KCPINQ()**

Description

The KCPINQ() function first checks the environment variable EXPROF_SWITCHES. If the environment variable exists KCPINQ() returns an integer value representing the occurrence or non-occurrence of the character 'c' in the environment variable. If the **EXPROF_SWITCHES** environment variable does not exist the **KCPINQ()** function returns the value of a global variable indicating whether the **auto** or **manual** switch was used on the **-Mperf** link directives. The **KCPINQ()** function is most commonly used to determine (at runtime) whether or not to enable the communication profiling system.

The performance monitoring linked into the application performs a check for the communication monitoring system during its start-up code. This is used to allow automatic configuration of the communication profiling system. If enabled the system will also automatically generate a call to **KCPDMP()** upon program termination.

A typical application need not contain explicit calls to the communication profiling routines. These calls are all made by the performance monitoring code. The only case in which explicit calls are needed is when more careful control is required over the profiler and the data it writes to disk.

KCPINQ *(cont.)*

KCPINQ *(cont.)*

Example

The following code is a skeleton of that which might be used to control the communication profiler in an application program.

```
      PROGRAM CBXPRF
C
C-- Start off profiler. This code is not strictly
C-- necessary since it is equivalent to the check
C-- made automatically by Performance Monitoring code.
C
      ISTAT = KCPINQ()
      IF(ISTAT .NE. 0) CALL KCPON
C
      ...
C
C-- Program over, Dump data again and exit. Again
C-- this code is superfluous since it duplicates the
C-- action of Performance Monitoring code.
C
      IF(ISTAT .NE. 0)CALL KCPDMP('exprof.prc')
      ENDIF
      STOP
      END
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

See Also

ctool(command), **KCPROF**(), **KCPDMP**()

KCPON, KCPOFF KCPON, KCPOFF

Control communication profiler.

Synopsis

SUBROUTINE **KCPON()**

SUBROUTINE **KCPOFF()**

Description

KCPON() is used to enable and start the communication profiler. After this call all subsequent calls to the communication system result in entries being made in an internal log-file. **KCPOFF()** reverses this process - until a subsequent call to **KCPON()** no communication profiling will be performed.

The log of profiling information is written to the host file system with **KCPDMP()**.

Example

The following code is a skeleton of that which might typically be used to control the communication profiler.

```
      PROGRAM PRFTST
C
C-- Start off profiler.

        CALL KCPON()
C
C-- Application Phase 1., profiler running.
C
        ...
C
C-- Phase 1 complete, dump data with KCPDMP.
C
        ...
C
C-- Application Phase 2., profiler turned off by
C-- previous call to KCPDMP.
C
        ...
```

KCPON, KCPOFF *(cont.)*

KCPON, KCPOFF *(cont.)*

```
C
C-- Application phase 3., turn on profiler again.
C
   CALL KCPON()
      ...
C
C-- Program over, dump data again and exit. The STOP statement
C-- will take care of dumping data to the host automatically.
C
      ...
   STOP
   END
```

Notice that we can selectively profile pieces of code. In this mode it makes sense to dump out the profile data independently to separate files for simplicity in later analysis.

See Also

ctool(command), **KCPDMP()**

KEP KEP

Event driven profiler.

Synopsis

SUBROUTINE **KEPON**()

SUBROUTINE **KEPOFF**()

SUBROUTINE **KEPINI** (*labbuf, labsiz, logbuf, logsiz*)

SUBROUTINE **KEPLAB** (*index, title, format*)

SUBROUTINE **KEPADD** (*index, datum*)

Parameter Declarations

INTEGER *labbuf*(*), *labsiz*, *logbuf*(*), *logsiz*

INTEGER *index*
CHARACTER*80 *title, format*

INTEGER *index, datum*

Description

These routines make up the interface to the user specified event driven profiling facility. **KEPON**() and **KEPOFF**() enable and disable the system respectively. While disabled no events are logged even if calls are made to **KEPADD**().

The routine **KEPINI**() must be called before any of the other profiling calls. The arguments indicate two buffers to be used for "title" and "event" entries which must be supplied by the calling program. Each entry corresponds to a single call to the **KEPLAB**() and **KEPADD**() subroutines. As a guide to the amount of space which should be provided, the current allowances for log entries and labels are 12 and 68 bytes respectively. Note that the *logsiz* and *labsiz* arguments should be given in bytes.

KEP *(cont.)*

KEP *(cont.)*

KEPADD() is the heart of the event system. It makes a new entry in the log file. Three items are logged; the event *index* and *datum* as given in the function call and the time at which the call is made. The *index* argument is used to differentiate between events at the highest level. This index corresponds to an optional *title* string in a call to **KEPLAB**(). The *datum* argument is used to identify events at the lowest level. This will be used in conjunction with the *format* argument supplied to a call to **KEPLAB**().

The function **KEPLAB**() is used to facilitate event recognition when the log file is subsequently analyzed. Its use is optional. If no calls to **KEPLAB**() are made then events will be identified by their *index* argument in the subsequent analysis and the *datum* value will be assumed to be an integer. Making the following call:

```
CALL KEPLAB(3, 'After return from crunch', 'Energy = %d')
```

builds in extra information. Together with the event *index* a legend will be presented which connects type 3 with the string "`After return from crunch`". Furthermore, when the value of the *datum* argument is shown it will be formatted according to the format string. A typical result would be as follows:

```
Energy = 23
```

Note that the Fortran string is interpreted according to the conventions associated with the C function **printf**. At its simplest, this merely means that text is printed as entered and the special string '`%d`' is replaced with an integer value.

The log of profiling information is written to the host file system with **KEPDMP**().

Example

The following code is a skeleton of that which might typically be used to control the event profiler.

```
      PROGRAM EPRTST
C
      REAL ENERGY, RESID, GRIND, CRUNCH
      INTEGER ITER, I
      INTEGER LOGBUF(2048), LABBUF(256)
```

KEP *(cont.)*

KEP *(cont.)*

```
C
C-- Start profiler, make labels for indices 1-3.
C
   CALL KEPINI(LABBUF, 1024, LOGBUF, 8192)
   CALL KEPLAB(1, 'Outer loop', 'Iteration %d')
   CALL KEPLAB(2, 'After crunch', 'Energy = %d')
   CALL KEPLAB(3, 'Inner loop', 'resid = %d')
   CALL KEPON
C
C-- Start application code, then go into main loop.
C
      ...
   DO 10 ITER=1,100
      CALL KEPADD(1, ITER)
      ENERGY = CRUNCH(ITER)
      CALL KEPADD(2, INT(ENERGY))
      DO 20 I=1,4
         RESID = GRIND(ENERGY)
         CALL KEPADD(3, INT(RESID))
 20   CONTINUE
 10   CONTINUE
C
C-- Program over, dump profile data and exit.
C
      ...
   STOP
   END
```

The insertion of events like these above can provide significant information about an application. The time between events 1 and 2, for example, indicates the duration of a call to the CRUNCH function. Similar information is available about GRIND from events 2 and 3, averaged over the four calls per iteration. The auxiliary *datum* fields will show the interaction between the variables and the program execution rate. It may also show up bugs and/or unexpected behavior which could be the key to understanding the failings of a particular parallelization scheme.

See Also

etool (command), **KEPDMP()**

KEPDMP KEPDMP

Write event profile data to a file.

Synopsis

SUBROUTINE **KEPDMP**(*fname*)

Parameter Declarations

CHARACTER*80 *fname*

Description

This routine is used to write event profile data to a disk file for later analysis with the **etool** command. The single argument is the name of the file to which profile data should be written. At the conclusion of the call the event profiler is disabled as though by a call to **KEPOFF**() and its internal state reset to zero so that further profiling leads to distinct, non-overlapping data.

A call to this function with the filename *exprof.pre* is automatically generated during the program exit if **KEPINQ**() detects that the profiler is enabled at runtime. Because of this most programs need never use this routine explicitly unless fine control is required over the location and timing of the profile dumps.

Example

The following code is a skeleton of that which might typically be used to control the event profiler.

```
      PROGRAM EPRTST
C
      INTEGER LOGBUF(2048), LABBUF(256)
C
      REAL ENERGY, RESID, GRIND, CRUNCH
      INTEGER ITER, I
C
C-- Setup profiler and make labels for indices. If
C-- asked to do so at runtime start the thing up.
C
```

KEPDMP *(cont.)*

KEPDMP *(cont.)*

```
      CALL KEPINI(LABBUF, 1024, LOGBUF, 8192)
      CALL KEPLAB(1, 'Outer loop', 'Iteration %d')
      CALL KEPLAB(2, 'After crunch', 'Energy = %d')
      CALL KEPLAB(3, 'Inner loop', 'resid = %d')
      ISTAT = KEPINQ()
      IF(ISTAT .NE. 0) CALL KEPON
C
C--Start application code, then go into main loop.
C
         ...
      DO 10 ITER=1,100
         CALL KEPADD(1, ITER)
C
         ENERGY = CRUNCH(ITER)
         CALL KEPADD(2, INT(ENERGY))
C
         DO 20 I=1,4
            RESID = GRIND(ENERGY)
            CALL KEPADD(3, RESID)
 20      CONTINUE
 10      CONTINUE
C
C-- Program over; dump data to host for later analysis.
C
      IF(ISTAT.NE.0) CALL KEPDMP('exprof.pre')
      STOP
      END
```

The strange looking strings passed as the third argument to the **KEPLAB()** functions are actually going to be passed to the C string formatting routine **sprintf()**. All that is really important for this application is that the characters "%d" will be replaced by the decimal value supplied as the last argument to **KEPADD()**.

Notice that the **KEPADD()** and **KEPLAB()** calls are completely safe even if **KEPINQ()** returns 0 and the profiler is not enabled.

See Also

etool (command), **KEPROF()**

KEPINQ KEPINQ

Determine runtime status of the event profiling system.

Synopsis

INTEGER FUNCTION KEPINQ()

Description

The **KEPINQ()** function first checks the environment variable **EXPROF_SWITCHES**. If the environment variable exists **KEPINQ()** returns an integer value representing the occurrence or non-occurrence of the character 'e' in the environment variable. If the **EXPROF_SWITCHES** environment variable does not exist the **KEPINQ()** function returns the value of a global variable indicating whether the **auto** or **manual** switch was used on the **-Mperf** link directives. The **KEPINQ()** function is most commonly used to determine at runtime whether or not to enable the event profiling system.

The performance monitoring linked into the application performs a check for the event monitoring system during its startup code. This is used to allow automatic configuration of the event profiling system. If enabled, the system will also automatically generate a call to **KEPDMP()** upon program termination.

As a result a typical application need contain no explicit calls to the event profiling routines - they are all made by the performance monitoring code. The only case in which such calls are needed is when more careful control is required over the event profiler and the data it dumps.

Note that the **KEPROF()** functions are always available to have fine control over the profiling process, even when the runtime system is able to initialize and dump the profiling data.

Example

The following code is a skeleton of that which might typically be used to control the event profiler.

```
      PROGRAM EPRTST
C
      INTEGER LOGBUF(2048), LABBUF(256)
C
      REAL ENERGY, RESID, GRIND, CRUNCH
      INTEGER ITER, I
```

KEPINQ *(cont.)*

KEPINQ *(cont.)*

```
C
C-- Setup profiler and make labels for indices. If
C-- asked to do so at runtime start the thing up.
C
   CALL KEPINI(LABBUF, 1024, LOGBUF, 8192)
   CALL KEPLAB(1, 'Outer loop', 'Iteration %d')
   CALL KEPLAB(2, 'After crunch', 'Energy = %d')
   CALL KEPLAB(3, 'Inner loop', 'resid = %d')
   ISTAT = KEPINQ()
   IF(ISTAT .NE. 0) CALL KEPON
C
C--Start application code, then go into main loop.
C
      ...
   DO 10 ITER=1,100
      CALL KEPADD(1, ITER)
C
      ENERGY = CRUNCH(ITER)
      CALL KEPADD(2, INT(ENERGY))
C
      DO 20 I=1,4
         RESID = GRIND(ENERGY)
         CALL KEPADD(3, RESID)
 20   CONTINUE
 10   CONTINUE
C
C-- Program over; dump data to host for later analysis.
C
   CALL KEPDMP('exprof.prc')
   STOP
   END
```

The strange looking strings passed as the third argument to the **KEPLAB()** functions are actually going to be passed to the C string formatting routine **sprintf()**. All that is really important for this application is that the characters "%d" will be replaced by the decimal value supplied as the last argument to **KEPADD()**.

Notice that the **KEPADD()** and **KEPLAB()** calls are completely safe even if **KEPINQ()** returns 0 and the profiler is not enabled.

See Also

etool (command), **KEPROF()**

KEPTGI, KEPTOG

KEPTGI, KEPTOG

Calculate program statistics.

Synopsis

SUBROUTINE **KEPTGI**(*toggle, label*)

SUBROUTINE **KEPTOG**(*toggle*)

Parameter Declarations

INTEGER *toggle*(16)
CHARACTER*80 *label*

INTEGER *toggle*(16)

Description

These routines allow selective analysis of particular sections of code. By surrounding code segments with calls to **KEPTOG()** one can obtain statistics relating to the number of times the particular code section was called and the average and total times spent in these sections. The data is collected in exactly the same manner as the event profiling information obtained through calls to **KEPADD()**. The same commands are available to write the profiling data to disk as are used by the other "**KEPROF**" utilities.

Each toggle data structure must be initialized with a call to **KEPTGI()** before it can be used for data collection. This function expects to be passed an array of integers and a string that will later be used to identify the collected statistics when analyzed with **etool**.

The log of profiling information is written to the host file system with **KEPCP()** or **KEPDMP()**.

KEPTGI, KEPTOG *(cont.)*

KEPTGI, KEPTOG *(cont.)*

Example

The following example demonstrates the use of the "toggle" ideas.

```
      PROGRAM TOGTST
      INTEGER LPTOG(16), GRNTOG(16)
      REAL*4 ENERGY, GRIND
      INTEGER ITER, I
      COMMON/XPRESS/NOCARE,NORDER,NONODE,IHOST,IALNOD,IALPRC
C
C-- Initialize toggle data structures.
C
      CALL KEPTGI(LPTOG, 'Main iteration loop')
      CALL KEPTGI(GRNTOG, 'Calls to GRIND')
C
C-- Start application code, then go into main loop.
C
         ...
C
      DO 10 ITER = 1, 100
         CALL KEPTOG(LPTOG)
C
C--      Other processing going on here....
C
                  ........
C
         DO 20 I = 1, 4
            CALL KEPTOG(GRNTOG)
            ENERGY = GRIND(I)
            CALL KEPTOG(GRNTOG)
 20      CONTINUE
         CALL KEPTOG(LPTOG)
 10      CONTRINUE
C
C-- Dump data to host for later analysis...
C
            .........
C
      STOP
      END
```

KEPTGI, KEPTOG *(cont.)*

KEPTGI, KEPTOG *(cont.)*

The toggle data will be stored in a file with the name *exprof.pre* (unless overridden by some other function call) together with the normal event data which may have also been collected with calls to **KEPADD()**.

To analyze the data that was generated from program *fft2d*, we execute the **etool** command as follows:

```
etool -p -t fft2d
```

This combination of switches both suppresses the normal graphical output and also restricts attention to the toggle data.

For each node is displayed the list of initialized toggles together with the number of times each code section was used, the total time elapsed in this section, the average time per call and the variance of these times. Using this information it is possible to build up a very accurate picture of the performance of a parallel program.

See Also

etool (command), **KEPROF()**, **KEPDMP()**.

KXPDMP KXPDMP

Write execution profile data to a file.

Synopsis

SUBROUTINE **KXPDMP**(*fname*)

Parameter Declarations

CHARACTER*80 *fname*

Description

This routine is used to write execution profile data to a disk file for later analysis with the **xtool** command. The single argument is the name of the file to which profile data should be written. At the conclusion of the call, the execution profiler is disabled as though by a call to **KXPOFF()** and its internal state is reset to zero so that further profiling leads to distinct, non-overlapping data.

A call to this function with the filename *exprof.prx* is automatically generated during the program exit if **KXPINQ()** detects that the profiler is enabled at runtime. Because of this, most programs need never use this routine explicitly unless fine control is required over the location and timing of the profile dumps.

Example

The following code is a skeleton of that which might typically be used to control the execution profiler.

```
      PROGRAM XPRTST
C
      INTEGER PRFBUF(2048), PRFSCL
C
C-- This value is 0x2000 (hexadecimal)
C
      PARAMETER (PRFSCL = 8192)
C
C-- This is the name of a function in the program, low
C-- in memory. A suitable candidate can usually be found
C-- by looking through the "linker map".
C
```

KXPDMP *(cont.)*

KXPDMP *(cont.)*

```
      EXTERNAL F_MAIN
C
C-- Start up profiler if user selected -mx option.
C
      ISTAT = KXPINQ()
      IF(ISTAT .NE. 0) THEN
         CALL KXPINI(PRFBUF, 8192, F_MAIN, PRFSCL)
         CALL KXPON
      ENDIF
C
C-- Execute application code with profiler running.
C
         ...
C
C-- Program over, dump data and exit.
C
      IF(ISTAT .NE. 0) CALL KXPDMP('exprof.prx')
      STOP
      END
```

Notice that since we can selectively profile pieces of code it sometimes makes sense to write out the profile data from separate program segments independently to separate files for simplicity in later analysis. In this case multiple calls to **KXPDMP()** are used to create independent files.

See Also

xtool (command), **KXPINI()**, **KXPROF()**

KXPINI KXPINI

Low level execution profiler.

Synopsis

SUBROUTINE **KXPINI**(*buffer, buflen, start, scale*)

Parameter Declarations

INTEGER *buffer*(*), *buflen, start, scale*

Description

This routine serves to initialize the execution profiler. Every few milliseconds the program counter of the user application is examined and a histogram entry in the memory area denoted by *buffer* is incremented. The size of the histogram area is *buflen* bytes.

In order to decide which histogram entry to increment, a mapping function is applied to the program counter discovered by the system. Note that the i860 instructions are each four bytes long. First *start* is subtracted and then the result is multiplied by *scale* and divided by 0x10000 (Hexadecimal). The complete mapping is as follows:

```
BIN = (PC - START)*SCALE/0x10000
```

The overall effect of the *scale* parameter is to map groups of adjacent program locations into the same histogram bin. A program location is a two-byte value. The value `scale = 0x10000` maps every program location into a separate histogram bin, `scale = 0x8000` maps each pair of program locations into a single bin, `scale = 0x4000` every group of four, and so on.

Using combinations of the *buflen*, *start* and *scale* parameters it is possible to allocate various memory ranges to be profiled. Note that no errors are incurred if the range is not large enough. this results in a calculated `BIN` which is out of the histogram range.

KXPINI() does not enable the profiler. An explicit call to **KXPON()** must be made to begin gathering profile data.

KXPINI *(cont.)* KXPINI *(cont.)*

Example

The following code can be used to control the execution profile:

```
      PROGRAM XPRTST
C
      INTEGER PRFBUF(2048), PRFSCL
      PARAMETER(PRFSCL = 8192)
C
C-- This is the name given to a particular routine in
C-- the program which is known to occur low in memory.
C-- This information can usually be obtained from a
C-- "linker map".
C
      EXTERNAL F_MAIN
C
      CALL KXPINI(PRFBUF, 8192, F_MAIN, PRFSCL)
      CALL KXPON
C
C-- Application code, profiler running....
C
          ...
```

The choice of the *start* argument is most conveniently made in conjunction with the linker map provided by the compiler. This usually contains a list of the addresses of all the functions in an application and can be used to find the smallest address.

See Also

xtool (command), **KXPROF()**

KXPINQ KXPINQ

Determine runtime status of the execution profiler.

Synopsis

INTEGER FUNCTION KXPINQ()

Description

The **KXPINQ()** function first checks the environment variable **EXPROF_SWITCHES**. If the environment variable exists **KXPINQ()** returns an integer value representing the (non)-occurrence of the character 'x' in the environment variable. If the **EXPROF_SWITCHES** environment variable does not exist the **KXPINQ()** function returns the value of a global variable indicating whether the **auto** or **manual** switch was used on the **-Mperf** link directives. The **KXPINQ()** function is most commonly used to determine at runtime whether or not to enable the execution profiling system.

The performance monitoring linked into the application performs a check for the communication monitoring system during its start-up code. This is used to allow automatic configuration of the communication profiling system. If enabled the system will also automatically generate a call to **KXPDMP()** upon program termination.

Example

The following code is a skeleton of that which might typically be used to control the execution profiler.

```
      PROGRAM XPRTST
C
      INTEGER PRFBUF(2048), PRFSCL
C
C-- This value is 0x2000 (hexadecimal)
C
      PARAMETER (PRFSCL = 8192)
C
C-- This is the name of a function in the program, low
C-- in memory. A suitable candidate can usually be found
C-- by looking through the "linker map".
C
```

KXPINQ *(cont.)*

KXPINQ *(cont.)*

```
      EXTERNAL F_MAIN
C
C-- Start up profiler if user selected -mx option.
C
      ISTAT = KXPINQ()
      IF(ISTAT .NE. 0) THEN
         CALL KXPINI(PRFBUF, 8192, F_MAIN, PRFSCL)
         CALL KXPON
      ENDIF
C
C-- Execute application code with profiler running.
C
         ...
C
C-- Program over, dump data and exit.
C
      IF(ISTAT .NE. 0) CALL KXPDMP('exprof.prx')
      STOP
      END
```

See Also

xtool (command), **KXPINI()**, **KXPROF()**

KXPON, KXPOFF KXPON, KXPOFF

Control execution profiler.

Synopsis

SUBROUTINE **KXPON()**

SUBROUTINE **KXPOFF()**

Description

KXPON() is used to enable and start the execution profiler which must have been previously initialized with a call to **KXPINI()**. Subsequently a periodically scheduled event occurs which causes the program counter of the user application to be "logged" in an internal structure. **KXPOFF()** reverses this process - until a subsequent call to **KXPON()** no execution profiling will be performed.

The profiler is initially off and must be explicitly enabled with calls to **KXPINI()** and **KXPON()**.

The log of profiling information is written to the host file system with **KXPDMP()**.

Example

The following code is a skeleton of that which might typically be used to control the execution profiler.

```
      PROGRAM XPTEST
C
      INTEGER PRFBUF(2048), PRFSCL
C
C-- This value is 0x2000 (hexadecimal)
C
      PARAMETER(PRFSCL = 8192)
C
C-- This is the name of a function found to be in
C-- low memory by perusing the "linker map".
C
```

KXPON, KXPOFF *(cont.)*

KXPON, KXPOFF *(cont.)*

```
      EXTERNAL F_MAIN
C
C-- Start off profiler.
C
      CALL KXPINI(PRFBUF, 8192, F_MAIN, PRFSCL)
      CALL KXPON
C
C-- Application code, profiler running.
C
         ...
C
C-- Program finishes. Dump data and exit.....
C
         ...
      STOP
      END
```

See Also

xtool (command), **KXPINI()**, **KXPDMP()**

PAT Configuration Information A

Introduction

The PAT utilities configuration information include the information that is used to control data gathering functions, and the information that is used to control the display and output of profile/trace data. This appendix provides the configuration information required to use the PAT utilities on different output devices. The PAT utilities support the following graphical environments:

- SunView
- X Window System
- Hardcopy is supported in PostScript form.

In order to change from one supported output device/environment to another, you can set command line switches on any of the PAT tools (**xtool**, **ctool**, or **etool**).

Changing to the Sunview Environment

The SunView environment is the default environment for the PAT utilities. In order to use the facilities of the SunView environment while using the PAT utilities, invoke any of the tools with default settings. For example, the command:

```
xtool program
```

will start up the graphical interface for the execution profiling tool under SunView on a Sun workstation.

You can suppress the graphical interface on any of the tools by invoking the tool with the **-p** switch. This causes the tool to be invoked using a tabular interface that should be usable by any ASCII terminal. In many cases, this graphical summary is more useful than the graphical output. For example, if the communication trace tool is invoked with the tabular interface, you might get the following output:

```
{intrepid:48} ctool -p fft2d
Ctool Version 3.1.2 -- Copyright (C) 1991 ParaSoft
Reading symbol table "fft2d": 100%
Symbol table: 766 public, 79 local.


Node      0
===========
    Elapsed time                 :    65.16 milliseconds
    Calculation                  :    63.56 milliseconds
    Node Comm.                   :     0.01 milliseconds
    I/O                          :     0.01 milliseconds
    System calls                 :     0.00 milliseconds
    Flick                        :     1.58 milliseconds

    Routine    | calls    time    errs |   0    1    2    4    8   16   32   64  128  256
  ------------------------------------------------------------------------------------------
  __crecv      |    24     0.00       0 |  24    0    0    0    0    0    0    0    0    0
  __csend      |    52     0.00       0 |  24    0    0    4    0    0    0    0    0   12
  __gshigh     |     2     0.00       0 |   2    0    0    0    0    0    0    0    0    0
  __gsync      |     2     0.00       0 |   2    0    0    0    0    0    0    0    0    0
  __irecv      |    24     0.00       0 |   0    0   24    0    0    0    0    0    0    0
  __msgwait    |    24     0.01       0 |  24    0    0    0    0    0    0    0    0    0
  _access      |     1     1.07       0 |   1    0    0    0    0    0    0    0    0    0
  _sbrk        |     2     0.00       0 |   0    0    0    0    0    0    0    0    0    0
  _ioctl       |     1     0.01       0 |   0    0    0    0    0    0    0    0    0    0
  _open        |     2     0.16       0 |   0    0    1    0    0    0    0    0    0    0
  _read        |     1     0.08       0 |   0    0    0    0    0    0    0    0    0    1
  _write       |     5     0.26       0 |   0    0    0    0    0    2    1    2    0    0


Node      1
===========
    Elapsed time                 :    65.14 milliseconds
        .
        .
        .
```

The graphical interface contains the same information, but you must view three different screens in order to get the same information you get in a single table. For example, Figure A-1 shows the screen you get for the "Time vs. node" selection in the graphical window.

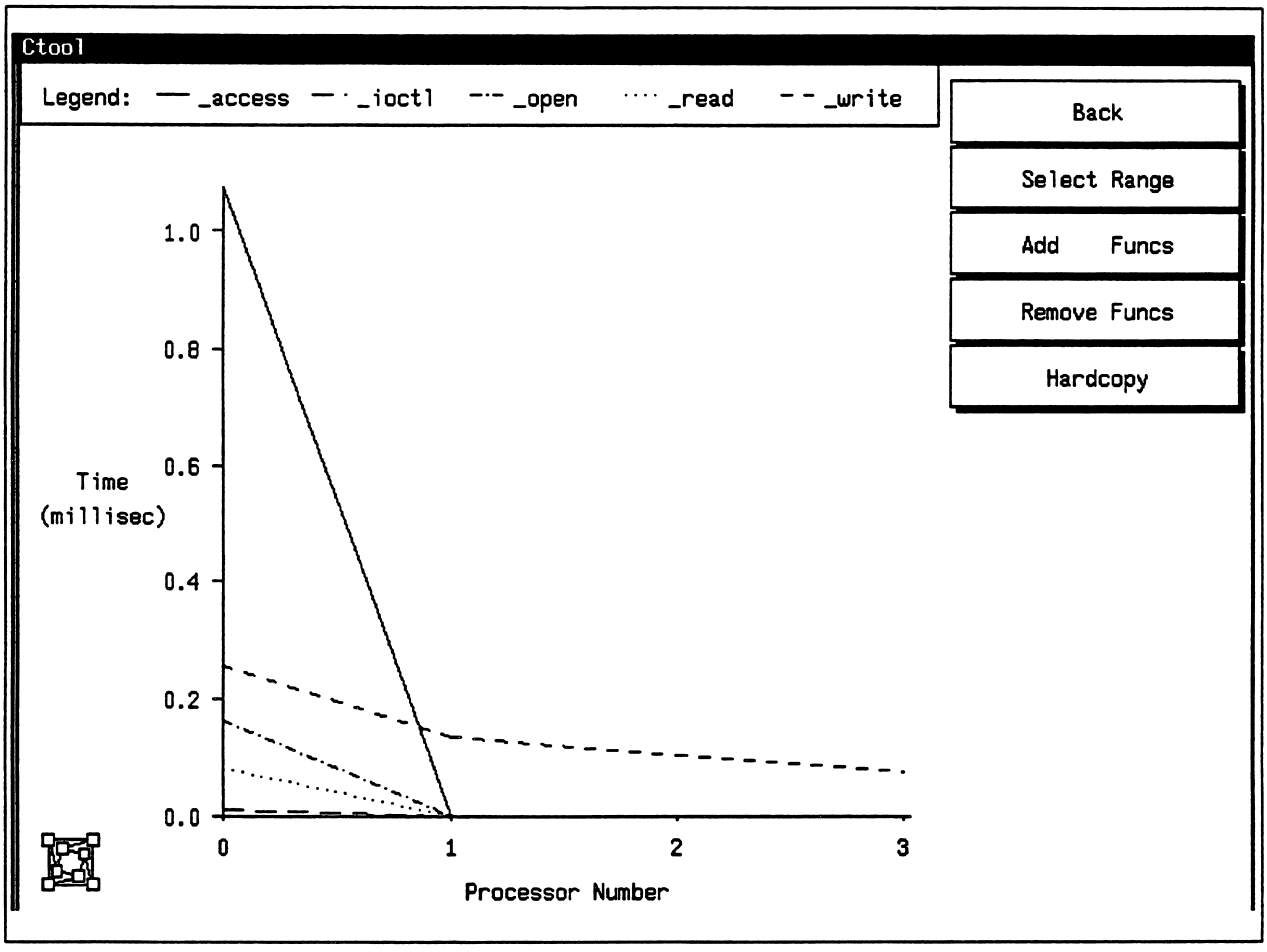

Figure A-1. "Time vs. Node" Graphical Display

Changing to the X Window System Environment

You can specify the X Window System as the graphical display environment by invoking any of the PAT utilities using the -T switch and specifying "X". For example, the following command line invokes the event tracing tool using the X Windows System:

```
{intrepid:48} etool -TX fft2d
```

The user interface and tool operation under the X Windows System is the same as under the default SunView environment.

Preparing for Output to a Postscript Device

When you select **Hardcopy** from any of the PAT utilities, PAT creates a file and writes a one page, PostScript-formatted graph to the file. The graph corresponds to the image currently shown in your PAT display and legend windows, but without borders or menu area information.

After PAT has created the graphic file, you must explicitly send the file to a PostScript-compatible printer or other output device. PAT writes to the output file, not to the printer. The print output file name uses the following format:

*Tool***rplot***Number*.*PSname*

The print output file is created in your current working directory. The parameter-like (i.e., italicized) portions of the print output file name have the following meanings:

Tool This portion of the filename is a letter. A "**c**" corresponds to print output files from **ctool**. An "**x**" corresponds to print output files from **xtool**. An "**e**" corresponds to print output files from **etool**.

Number This portion of the filename is an integer that corresponds to the plot number created during this session. PAT supplies this number automatically. The first plot is number 0, the second is number 1, and so on. The numbering is restarted at the beginning of each session, and existing files are overwritten by files of the same name.

PSname This file extension is "**ps**" by default. You can change the default extension when the tool is invoked by using the **-n** switch and supplying a new *PSname*.

A brief example shows how to obtain hardcopy output with the PAT utilities. If you had selected **Hardcopy** three times (for three different plots) during a **ctool** session, the following UNIX command would print the second plot to a PostScript printer named "psx":

```
{intrepid:55} lp -d psx crplot1.ps
```

PAT Common Data Structures B

Introduction

This appendix provides tables that list the default and the optional operations that are profiled/traced by the PAT utilities. The following tables list the commands, calls, functions, and routines that PAT uses when building the profile or trace files.

Table B-1. NX Communication Functions (1 of 2)

Call/Function	Parameters			Language Support	
	param 1	param 2	param 3	C	Fortran
_cprobe	typesel		return	✔	✔
_crecv	typesel	len	return	✔	✔
_csend	type	node	len	✔	✔
_csendrecv	type	tonode	typesel	✔	✔
_flushmsg	typsel	node	return	✔	✔
_gcol	xlen	ylen	return	✔	✔
_gcolx			return	✔	✔
_gdhigh	n		return	✔	✔
_gihigh	n		return	✔	✔
_gshigh	n		return	✔	✔
_gdlow	n		return	✔	✔
_gilow	n		return	✔	✔
_gslow	n		return	✔	✔

Table B-1. NX Communication Functions (2 of 2)

Call/Function	Parameters			Language Support	
	param 1	param 2	param 3	C	Fortran
_gdprod	n		return	✔	✔
_giprod	n		return	✔	✔
_gsprod	n		return	✔	✔
_gdsum	n		return	✔	✔
_gisum	n		return	✔	✔
_gssum	n		return	✔	✔
_giand	n		return	✔	✔
_gland	n		return	✔	✔
_gior	n		return	✔	✔
_glor	n		return	✔	✔
_gixor	n		return	✔	✔
_glxor	n		return	✔	✔
_gopf	xlen		return	✔	✔
_gsendx	xlen	nlen	return	✔	✔
_gsync			return	✔	✔
_hrecv	typesel	len	return	✔	✔
_hsend	type	node	len	✔	✔
_hsendrecv	type	tonode	typesel	✔	✔
_iprobe	typesel		return	✔	✔
_irecv	typesel	len	return	✔	✔
_isend	type	node	len	✔	✔
_isendrecv	type	tonode	typesel	✔	✔
_msgcancel	id		return	✔	✔
_msgdone	id		return	✔	✔
_msgwait	id		return	✔	✔

Table B-2. NX I/O Functions

Call/Function	Parameters			Language Support	
	param 1	param 2	param 3	C	Fortran
_cread	fildes	len	return	✔	✔
_cwrite	fildes	len	return	✔	✔
_eseek	fildes	whence	void	✔	✔
_esize	fildes	whence	void	✔	✔
_estat			return	✔	✔
_festat	fildes		return	✔	✔
_iodone	id		return	✔	✔
_iomode	fildes		return	✔	✔
_iowait	id		return	✔	✔
_iread	fildes	len	return	✔	✔
_iseof	fildes		return	✔	✔
_iwrite	fildes	len	return	✔	✔
_lsize	fildes	offset	return	✔	✔
_restrictvol	fildes	nvol	return	✔	✔
_setiomode	fildes	mode	return	✔	✔

Table B-3. NX System Functions

Call/Function	Parameters			Language Support	
	param 1	param 2	param 3	C	Fortran
_flick			void	✔	✔
_fork			return	✔	✔
_handler	etype		return	✔	✔
_killcube	node		return	✔	✔
killproc	node		return	✔	✔
_killproc	node		return	✔	✔
_led			void	✔	✔
load	node		return	✔	✔
_load	node		return	✔	✔
_masktrap	state		return	✔	✔
waitall	node		return	✔	✔
_waitall	node		return	✔	✔
waitone	node	cnode	ccode	✔	✔
_waitone	node	cnode	ccode	✔	✔

Table B-4. UNIX Supported Functions (1 of 2)

Call/Function	Parameters			Language Support	
	param 1	param 2	param 3	C	Fortran
access	amode		return	✔	
alarm	sec		return	✔	
brk	endds		return	✔	
sbrk	incr		return	✔	
chdir			return	✔	
chmod	mode		return	✔	
chown	owner	group	return	✔	
close	fildes		return	✔	
creat	mode		return	✔	
dup	fildes		return	✔	
execl			return	✔	
execle			return	✔	
execlp			return	✔	
execv			return	✔	
execve			return	✔	
execvp			return	✔	
exit	status		void	✔	
gtty	fildes		return	✔	
ioctl	fildes	request	return	✔	
kill	pid	sig	return	✔	
link			return	✔	
lseek	fildes	offset	whence	✔	✔
mkdir	mode		return	✔	
nice	incr		return	✔	
open	oflag	mode	return	✔	
pause			void	✔	

Table B-4. UNIX Supported Functions (2 of 2)

Call/Function	Parameters			Language Support	
	param 1	param 2	param 3	C	Fortran
pipe	fildes		return	✔	
read	fildes	nbyte	return	✔	
rmdir			return	✔	
sighold	sig		return	✔	
sigignore	sig		return	✔	
signal	sig		return	✔	
sigpause	sig		return	✔	
sigrelse	sig		return	✔	
sigreturn			return	✔	
sigset	sig		return	✔	
stat			return	✔	
fstat	fildes		return	✔	
statfs	len	fstyp	return	✔	
fstatfs	fildes	fstyp	return	✔	
sync			return	✔	
unlink			return	✔	
wait			return	✔	
write	fildes	nbyte	return	✔	

Index

M

O

P

R

S

T

X