Course Title: Introduction to Performance Tools

1 Introduction

Performance tools are software used to measure application performance usually with respect to execution time of all or portions of a code. These tools collect data from a running application that can be analyzed to determine if and where there are performance bottlenecks. There are a number of performance tools available ranging from simple utilities to sophisticated software with advanced visualization capabilities for data analysis.

In this tutorial we will describe the following performance tools commonly used to analyze scientific application performance:

- Strace
- Gprof
- Perfsuite
- TAU

This tutorial is not a comprehensive guide on how to use these tools. Rather it is intended as an introduction to using performance tools and how you can apply them to measure the performance of your code. To help you obtain a deeper understanding of the tools, several references are listed at the end of each lesson. These supplement the information presented in the lesson or focus in more detail on an important concept that was presented.

Note: There are many performance tools available to the scientific application developer and only a few of them are described in this tutorial. The tools that are described are used for explanatory purposes only and as such no endorsement of any particular tool is intended.

2 Strace

2.1 Overview

Strace is a Linux utility that traces system calls and signals during program execution. You do not need to recompile your program to use strace which makes it useful even when you do not have access to the source code.

Strace is most commonly used for system administration purposes, but it can also be used for tuning application performance in a variety of ways:

- **Debugging and diagnostics.** Strace can be especially helpful in finding bugs in applications that may not show up in standard error logs or that have not yet been documented (such as those resulting from a just-released system or application update), or in cases where you don't have access to the source code since recompiling is not necessary to perform a trace.
- **Detecting process activity.** Strace can be used to find instances of processes that might be running undetected by ps that could be interfering with or detracting from your application's performance.
- **Tracing a particular system call.** Strace can be used to trace the system calls made by a process while executing, as well as reporting the signals, or software interrupts, that the process receives.
- **Profiling a system call.** In addition to tracing a specific system call, strace can be used to generate a basic profile listing including the number of times a call occurs and the amount of time spent on the calls.

Objectives

After completing this lesson you will be able to:

- State the capabilities provided by strace for tuning program performance.
- Recognize how to run strace on a program.
- Identify the strace option that is useful for obtaining high-level profiling information for an application.
- Interpret a simple strace report.

2.2 Running Strace

Strace is most likely already installed on your system by default. If it is not, the latest version can be downloaded and installed by most package managers, or downloaded from SourceForge at http://sourceforge.net/projects/strace/ and built from source.

In its simplest form, strace can be used without options to invoke a program. The following command line is used to trace the system calls made by the Linux time command:

```
strace time
```

The output produced from tracing the time command is:

```
execve("/usr/bin/time", ["time", "-o", "output.txt"], [/* 31 vars */]) = 0
uname({sys="Linux", node="osage.ncsa.uiuc.edu", ...}) = 0
brk(0)                                  = 0x804c000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=68949, ...}) = 0
old_mmap(NULL, 68949, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb75d8000
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200X\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1573180, ...}) = 0
old_mmap(NULL, 1280684, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xb749f000
old_mmap(0xb75d2000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x132000) =
0xb75d2000
old_mmap(0xb75d5000, 10924, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0)
= 0xb75d5000
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb749e000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb749e4e0, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0xb75d8000, 68949)               = 0
write(2, "Usage: time [-apvV] [-f format] "..., 167Usage: time [-apvV] [-f format] [-o file]
[--append] [--verbose]
       [--portability] [--format=format] [--output=file] [--version]
       [--help] command [arg...]
) = 167
exit_group(1)                           = ?
Process 11444 detached
```

Each line in the strace output contains the system call name, followed by its arguments in parentheses and its return value. For example, in the output listing above the system call to write followed by its arguments and return value is highlighted in yellow.

There are several options you can use with strace to examine the behavior of system calls. For example, you can use the -p option to trace the system calls made by a process. This can be useful for identifying a process that might be hanging the system. The usage information from the strace man page for the -p option is:

-p pid

> Attach to the process with the process ID pid and begin tracing. The trace may be terminated at any time by a keyboard interrupt signal (CTRL-C). strace will respond by detaching itself from the traced process(es) leaving it (them) to continue running. Multiple -p options can be used to attach to up to 32 processes in addition to command (which is optional if at least one -p option is given).

Below is an example use of the -p option to show the system calls made as an xterm waits for user input:

```
[arnoldg@honest2 Linux_PII_CBLAS]$ xterm &
[1] 8003
[arnoldg@honest2 Linux_PII_CBLAS]$ strace -p 8003
Process 8003 attached - interrupt to quit

select(5, [3 4], [], NULL, {0, 877000}) = 0 (Timeout)
ioctl(3, FIONREAD, [0]) = 0
ioctl(3, FIONREAD, [0]) = 0
select(4, [3], [], [], {0, 0}) = 0 (Timeout)
ioctl(3, FIONREAD, [0]) = 0
ioctl(3, FIONREAD, [0]) = 0
```

In the next section we show how to use the -c option to obtain high-level profile information for an application. For information on the other options, see the strace man page.

2.3 Profiling with Strace

Using strace with the -c option will give you a high-level profile of your application. An example use of strace with the -c option on the time command follows:

```
[arnoldg@co-login ~]$ strace -c time uptime
  3:51pm  up 147 days  0:03,  37 users,  load average: 0.45, 0.56, 0.93
0.00user 0.00system 0:00.02elapsed 0%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+165minor)pagefaults 0swaps
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 77.80    0.000333          12        28        26 open
 12.38    0.000053           0       112           write
  9.81    0.000042          42         1           brk
  0.00    0.000000           0         1           read
  0.00    0.000000           0         2           close
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         2           gettimeofday
  0.00    0.000000           0         1           wait4
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         6           mmap
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           mprotect
  0.00    0.000000           0         4           rt_sigaction
  0.00    0.000000           0         1           madvise
  0.00    0.000000           0         2           fstat
  0.00    0.000000           0         1           clone2
------ ----------- ----------- --------- --------- ----------------
100.00    0.000428                   166        27 total
```

The strace report generated by using the -c option (shown above) gives the following information on each system call:

% time	The percentage of the total execution time the program spent in the system call.
seconds	The total number of seconds spent in the system call.
usecs/call	The time spent per call (in microseconds).
calls	The number of times the system call was encountered.
errors	The number of times errors were returned.
syscall	The name of the system call.

2.4 Self Test

<?xml version="1.0" encoding="UTF-8"?> @import url(base.css); @import url(content.css); strace

Strace

Reflection
What capability does strace provide?
Click here

Strace traces system calls and signals for any Linux process. It can be used for tuning application performance in a variety of ways:

- Debugging and diagnostics.
- Detecting process activity.
- Tracing a particular system call.
- Profiling a system call.

True-False Question

When running strace in its simplest form on a progarm named 'myprog' you must first recompile it and then enter the following command: strace myprog.

True ○ False ○

Multi-choice

Which strace option can be used to generate a high-level profile of your application?

- ○ None, using the strace command without options provides profile information.
- ○ -c
- ○ -p

Fill-in-the-blanks

```
[arnoldg@co-login ~]$ strace -c time uptime

3:51pm  up 147 days  0:03,  37 users,  load average: 0.45, 0.56, 0.93
0.00user 0.00system 0:00.02elapsed 0%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+165minor)pagefaults 0swaps
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
77.80    0.000333         12        28        26 open
12.38    0.000053          0       112           write
9.81    0.000042         42         1           brk
0.00    0.000000          0         1           read
0.00    0.000000          0         2           close
0.00    0.000000          0         1           execve
0.00    0.000000          0         1         1 access
0.00    0.000000          0         2           gettimeofday
0.00    0.000000          0         1           wait4
0.00    0.000000          0         1           uname
0.00    0.000000          0         6           mmap
0.00    0.000000          0         1           munmap
0.00    0.000000          0         1           mprotect
0.00    0.000000          0         4           rt_sigaction
0.00    0.000000          0         1           madvise
0.00    0.000000          0         2           fstat
0.00    0.000000          0         1           clone2
------ ----------- ----------- --------- --------- ----------------
100.00    0.000428                   166        27 total
```

Looking at the strace output shown above, fill in the blanks below:

Which system call had the longest duration?

Which system call was called the most?

In which system call did the application spend the most time?

What were the total number of system calls?

Submit Restart Show Answers

2.5 Additional Resources

- Strace man page
- Intro to IO Profiling of Applications (uses strace)
- IO Profiling of Applications: MPI Apps (uses strace)
- Introduction to UNIX Signals and System Calls

3 Gprof

3.1 Overview

Gprof is a standard profiling utility installed by default on most Linux-based systems. You can use gprof to see how much time is being spent in each function of your program along with which functions call or are called by each function. To use gprof you must compile and link your code with profiling enabled. By doing this code will be inserted at the beginning and end of each function in your application that is used for collecting timing information. After compiling, you run your application as you normally would and the inserted code outputs a large amount of raw profile data to a file. You then run gprof to analyze the profile data.

Running gprof produces several forms of output generated from the analysis of the profile data - the flat profile, the call graph, and the annotated source listing. The **flat profile** shows how much time was spent in each function, and how many times a function was called. From this, you can tell which functions in your program are burning most of the cycles. The **call graph** gives specific details about function calls: for each function which function(s) called it and how many times it was called, which functions it calls and how many times it called it. It also gives an estimate of how much time was spent in the subroutines of each function. From this information you can identify which function calls use a lot of time. The **annotated source listing** is a copy of the program's source code with labels indicating the number of times each line of the program was executed.

Lesson Objectives

After completing this lesson you will be able to:

- State the capabilities provided by gprof for analyzing program performance.
- List the steps involved in analyzing a program using gprof.
- List the reports generated by gprof and describe the information they provide.
- Select the appropriate gprof report for retrieving specific information from your program.
- Interpret the data in the two most common reports generated by gprof.

3.2 Running Gprof

The steps for running gprof are:

1. Compile and link your program with profiling enabled.
2. Execute your program to generate a profile data file.
3. Run gprof to analyze the profile data.

We will go briefly over these steps to give you a basic understanding of gprof. For further detail we recommend looking at the additional resources listed at the end of this lesson especially the GNU gprof document.

Compiling a Program with Profiling Enabled

To enable profiling in your program, you must compile and link it as you normally would but with the addition of the -pg option. If you want, you can compile only some of the modules of the program with -pg but if you do the flat profile will only give information about the total time spent in the functions of those modules and nothing about how many times they were called, or from where. It will also greatly reduce the usefulness of the call graph.

Another useful option to include along with -pg is -g. This option instructs the compiler to insert debugging symbols into the program that match program addresses to source code lines. This enables you to perform line-by-line profiling and can be useful once you know what section of your code needs to be optimized. For more on line-by-line profiling with gprof see the GNU Gprof manual.

Generating Profile Data

Once you have enabled profiling in your code, you simply execute the program using the arguments, file names, and so on that you normally use. Your program should run the same as it did before giving the same output while also generating the raw data that will be used by gprof to analyze your code. However, the time spent collecting and the writing the profile data will result in a somewhat slower execution time.

The profile data is written to a file called gmon.out in the *program's current working directory* at the time it exits. If there is already a file called gmon.out, its contents are overwritten. If you don't have permission to write in this directory you will get an error message and the file will not be written. Also, the gmon.out file will not be written properly if your program does not exit normally by either returning from main or calling exit.

Note that using gprof is a quick way to get a high-level view of where an application is spending time and how. However, like any tool there is bound to be some impact on the application's execution and the usefulness of the timing or profiling data generated. In some cases, gprof's timings can be inaccurate but since the learning curve is small and gprof is usually always available, it is useful to know about. There are usually ways to mitigate these inaccuracies. For example, when there are a lot of calls, getting the number of calls and the callpath information can affect the accuracy of the timings. In this case, you can get more accurate timings by choosing to only get timings (and not the call information) by adding the -pg flag only during the link phase. You can do a separate run if needed to get the call path information.

Analyzing Profile Data

After generating the profile data file gmon.out by executing your program, you run gprof to interpret the information contained in it. The command for running gprof is:

```
gprof options [executable-file [profile-data-files...]] [> outfile]
```

where the square-brackets indicate optional arguments. If you omit the executable-file name, a.out is used. If you have generated multiple profile data files and renamed them from gmon.out you can specify them in the optional arguments. Otherwise gmon.out is used by default. If you use multiple profile data files the statistics for them are summed together.

There are several options you can specify when using gprof. If you do not specify any options, the default options -p and -q are used, which prints a flat profile and call graph analysis for all functions. See the gprof man page for a description of the available options.

MPI tip: The default output file for gprof is gmon.out, but MPI doesn't handle that well and you end up with a profile that's missing most of the information. Set the environment variable GMON_OUT_PREFIX and each MPI rank will adds its process id to its gmon output filename. The gmon output files can then be combined in a summary with the gprof -s command.

3.3 Interpreting the Flat Profile

A portion of a flat profile generated using the gprof command with the default arguments is shown below:

```
Flat profile:

Each sample counts as 0.01 seconds.
 %     cumulative   self              self    total
 time    seconds   seconds     calls   s/call   s/call   name
 66.99    172.31    172.31                               Loop_M4_K
 5.45     186.33     14.02                               N4_M4_LOOP
 5.36     200.11     13.78                               __intel_new_memcpy
 3.77     209.81      9.70                               Loop_M4_K
 3.04     217.64      7.83                               N4_K_LOOP_COPYB
 1.64     221.87      4.23       316    0.01     0.01   HPL_dlaswp06N
 1.40     225.46      3.60                               N4_M4_WPARUP
 1.35     228.95      3.49 201118720    0.00     0.00   HPL_rand
 1.14     231.88      2.93 405985272    0.00     0.00   HPL_lmul
 1.09     234.69      2.81                               mkl_blas_mc_dgemm_copybn
 0.85     236.87      2.18       422    0.01     0.01   HPL_dlaswp00N
 0.82     238.97      2.10                               M4_LOOP
 0.65     240.64      1.67       216    0.01     0.01   HPL_dlaswp01N
 0.61     242.20      1.56        10    0.16     0.16   HPL_pdlange
 0.57     243.67      1.47                               AY16_Loop_M16
...
```

In the flat profile, the functions are sorted first by run-time then by number of calls, both in decreasing order, then alphabetically by name.

From the statement given just before the column headers, you can see that each sample counted as 0.01 seconds, which estimates the margin of error in each of the time figures. From the cumulative seconds column, you can see that the program's total execution time was 243.67 seconds. Since each sample counted as 0.01 seconds, 24,367 samples were taken during the run. (For example purposes we assume that our example flat profile listing is complete.)

The meanings of the values in the flat profile columns are defined as follows:

% time	The percentage of the total execution time the program spent in the function. The percentages for all of the functions should add up to 100%.
cumulative seconds	The cumulative total number of seconds spent executing the functions listed to that point in the profile.
self seconds	The number of seconds spent in the individual function.
calls	The total number of times the function was called. This field will be blank if the function was never called or if the number of calls cannot be determined. In the latter case, this is most likely due to the function not being compiled with profiling enabled as in functions from third-party libraries.
self ms/call	The average number of milliseconds spent in the function per call. If the function was not compiled with profiling enabled the field is left blank.
total ms/call	The average number of milliseconds spent in the function and its descendants per cal. If the function was not compiled with profiling enabled the field is left blank.
name	The name of the function.

3.4 Interpreting the Call Graph

Below is the call graph from the gprof analysis in the previous flat profile showing how much time was spent in each function and in its children:

```
granularity: each sample hit covers 2 byte(s) for 0.00% of 257.22 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     67.0  172.31      0.00                 Loop_M4_K [1]
-----------------------------------------------
                 0.00     20.16       2/2          main [3]
[2]      7.8     0.00     20.16       2          HPL_pdtest [2]
                 1.20      8.48       4/4          HPL_pdmatgen [7]
                 0.00      8.93       2/2          HPL_pdgesv [8]
                 1.56      0.00      10/10         HPL_pdlange [22]
                 0.00      0.00       6/6          HPL_indxg2p [117]
                 0.00      0.00       4/8          HPL_numroc [113]
                 0.00      0.00       4/4          HPL_ptimer [121]
                 0.00      0.00       2/21         HPL_grid_info [110]
                 0.00      0.00       2/6          HPL_all_reduce [116]
                 0.00      0.00       2/2          HPL_ptimer_boot [125]
                 0.00      0.00       2/2          HPL_barrier [123]
                 0.00      0.00       2/2          HPL_ptimer_combine [126]
                 0.00      0.00       2/25         PL_broadcast [108]
                 0.00      0.00       2/10890      HPL_dgemv [91]
                 0.00      0.00       2/22         HPL_reduce [109]
-----------------------------------------------
                                                <spontaneous>
[3]       7.8    0.00     20.16                 main [3]
                 0.00     20.16       2/2          HPL_pdtest [2]
                 0.00      0.00       1/1          HPL_pdinfo [133]
                 0.00      0.00       1/1          HPL_grid_init [132]
                 0.00      0.00       1/21         HPL_grid_info [110]
                 0.00      0.00       1/1          HPL_grid_exit [131]
...
```

The dashed lines divide the information into entries for each function. In each entry, the line that starts with an index number in square brackets is the primary line containing statistics for the function that is the subject of the entry. For explanatory purposes we refer to the function which the entry is about as the primary function. (In the primary line highlighted in yellow above, HPL_pdtest is the primary function.) The lines preceding the primary line gives the functions that called the primary function. The lines following the primary line gives the subroutines (also called children in the call graph) called by the primary function. For example, in entry [2] the primary function HPL_pdtest is called from the function main and calls the 14 subroutines listed below it. The entries are sorted by time spent in the function and its subroutines. If the calling function(s) cannot be determined, a dummy caller line is printed with a name value of <spontaneous> and all other fields left blank. This can happen for signal handlers.

The Primary Line

The primary line in a call graph lists information that describes the function which the entry is about. The line starts with an index number in square brackets and lists values for the overall statistics for the function indicated in the name column. In the example call graph above, the primary line for the entry with index value 2 is highlighted in yellow.

The meanings of the values in the primary line columns are:

index	The index number for the function. Entries are numbered with consecutive integers. Each function has an index number.
% time	The percentage of the total time spent in the primary function, including time spent in subroutines called by it.
self	The total amount of time spent in the primary function. This value should be identical to the number printed in the function's seconds field in the flat profile.
children	The total amount of time spent in the subroutine calls made by the primary function. This should be equal to the sum of all the self and children entries of the children called by this function.
called	The number of times the the primary function was called. Two numbers separated by a + sign will be given if the function called itself recursively. The first is the number of non-recursive calls, and the second is the number of recursive calls.
name	The name of the primary function followed by its index number in square brackets.

The Calling Functions Line

If the primary function was called by another function there will be a calling functions line that gives information about the call. Except for the index and % time fields, the fields in the calling functions line are the same as those in the primary line, but because of the different context they have different meanings. In the example call graph above, the calling functions line for the entry with index value 2 is displayed in blue.

The meanings of the values in the calling functions line are as follows:

self	An estimate of the amount of time spent in the primary function itself when it was called by the calling function.
children	An estimate of the amount of time spent in subroutines of the primary function when it was called from the calling function. The sum of the self and children fields is an estimate of the amount of time spent within calls to the primary function from the calling function.
called	Contains two numbers: the number of times the primary function was called from the calling function, followed by the total number of nonrecursive calls to the primary function from all of its callers.
name	The name of the calling function followed by its index number in square brackets.

The Subroutines Line

If the primary function called another function, or subroutine, there will be a subroutines line that gives information about each subroutine called. Just as with the calling functions line, the fields in the subroutines line are the same as those in the primary line, but because of the different context they have different meanings. In the example call graph above, the subroutines lines for the entry with index value 2 are displayed in green.

The meanings of the values in the calling functions line are as follows:

self	An estimate of the amount of time spent directly within the subroutine when it was called from the primary function.
children	An estimate of the amount of time spent in children of the subroutine when it was called by the primary function. The sum of the self and children fields is an estimate of the total time spent in the subroutine when it was called by the primary function.
called	Two numbers, the number of calls to the subroutine from the primary function followed by the total number of nonrecursive calls to the subroutine.
name	The name of the subroutine followed by its index number in square brackets. If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number.

3.5 Example Annotated Source Listing

As described earlier, the annotated source listing is a copy of the program's source code with labels indicating the number of times each line of the program was executed. Below is an example of an annotated source from a simple program, 1cpu.c. The labels are highlighted in yellow.

```
[arnoldg@lgheron c]$ gprof -A 1cpu
*** File /home/arnoldg/c/1cpu.c:
                #include <stdio.h>
                #include <math.h>

                int main(void)
       ##### -> {
                        void work(void);

                        work();
                }

                void work(void)
           1 -> {
                        double f, temp;
                        double sum;
                        double mycos(double);

                        for (f=0.0; f {
                        return(cos(arg));
                }


Top 10 Lines:

     Line      Count

       30  100000001
       12          1

Execution Summary:

        3   Executable lines in this file
        3   Lines executed
   100.00   Percent of the file executed

100000002   Total number of line executions
33333334.00   Average executions per line
```

3.6 Self Test

<?xml version="1.0" encoding="UTF-8"?>   @import url(base.css); @import url(content.css); gprof

Gprof

Reflection
Describe the capabilities provided by gprof for analyzing program performance.

Click here

Gprof provides information about how much time is being spent in each function of a program along with which functions call or are called by each function.

Reflection
What are the steps for using gprof to analyze your code?

Click here

The steps for using gprof are:

1. Compile and link your program with profiling enabled.
2. Execute your program to generate a profile data file.
3. Run gprof to analyze the profile data.

Reflection

List and describe the reports generated by gprof from the program's profile data.

Click here

- Flat profile: provides information about how much time was spent in each function, and how many times a function was called.
- Call graph: gives specific details about function calls - for each function which function called it and how many times it was called and which functions it calls and how many times it called it. It also gives an estimate of how much time was spent in the subroutines of each function.
- Annotated source listing: provides a copy of the program□s source code with labels indicating the number of times each line of the program was executed.

Multi-choice

Which one of the following gprof reports is best for finding the functions in your program that are using the most time?

- ○ annotated source listing
- ○ call graph
- ○ flat profile

Multi-choice

Which one of the following gprof reports is best for identifying which function calls use a lot of time?

- ○ flat profile
- ○ call graph
- ○ annotated source listing

Fill-in-the-Blanks

```
Flat profile:

Each sample counts as 0.01 seconds.
 %     cumulative   self              self    total
 time    seconds   seconds      calls   s/call   s/call   name
 66.99    172.31    172.31                                Loop_M4_K
 5.45     186.33     14.02                                N4_M4_LOOP
 5.36     200.11     13.78                                __intel_new_memcpy
 3.77     209.81      9.70                                Loop_M4_K
 3.04     217.64      7.83                                N4_K_LOOP_COPYB
 1.64     221.87      4.23        316    0.01     0.01    HPL_dlaswp06N
 1.40     225.46      3.60                                N4_M4_WPARUP
 1.35     228.95      3.49   201118720    0.00     0.00    HPL_rand
 1.14     231.88      2.93   405985272    0.00     0.00    HPL_lmul
 1.09     234.69      2.81                                mkl_blas_mc_dgemm_copybn
 0.85     236.87      2.18        422    0.01     0.01    HPL_dlaswp00N
 0.82     238.97      2.10                                M4_LOOP
 0.65     240.64      1.67        216    0.01     0.01    HPL_dlaswp01N
 0.61     242.20      1.56         10    0.16     0.16    HPL_pdlange
 0.57     243.67      1.47                                AY16_Loop_M16
...

granularity: each sample hit covers 2 byte(s) for 0.00% of 257.22 seconds

index % time     self   children   called      name
                                                   <spontaneous>
[1]     67.0   172.31      0.00               Loop_M4_K [1]
-----------------------------------------------
                > 0.00     20.16      2/2          main [3]
```

```
[2]      7.8     0.00     20.16      2        HPL_pdtest [2]
                 1.20      8.48      4/4          HPL_pdmatgen [7]
                 0.00      8.93      2/2          HPL_pdgesv [8]
                 1.56      0.00     10/10         HPL_pdlange [22]
                 0.00      0.00      6/6          HPL_indxg2p [117]
                 0.00      0.00      4/8          HPL_numroc [113]
                 0.00      0.00      4/4          HPL_ptimer [121]
                 0.00      0.00      2/21         HPL_grid_info [110]
                 0.00      0.00      2/6          HPL_all_reduce [116]
                 0.00      0.00      2/2          HPL_ptimer_boot [125]
                 0.00      0.00      2/2          HPL_barrier [123]
                 0.00      0.00      2/2          HPL_ptimer_combine [126]
                 0.00      0.00      2/25         PL_broadcast [108]
                 0.00      0.00      2/10890      HPL_dgemv [91]
                 0.00      0.00      2/22         HPL_reduce [109]
-----------------------------------------------
                                                 <spontaneous>
[3]       7.8    0.00      20.16              main [3]
                 0.00      20.16     2/2          HPL_pdtest [2]
                 0.00       0.00     1/1          HPL_pdinfo [133]
                 0.00       0.00     1/1          HPL_grid_init [132]
                 0.00       0.00     1/21         HPL_grid_info [110]
                 0.00       0.00     1/1          HPL_grid_exit [131]
...
```

Looking at the flat profile and call graph shown above, fill in the blanks below:

In which function did the program spend the most time?

In entry 2, how many seconds did the function HPL_pdtest spend in the function HPL_pdlange?

In entry 2, how many times was the function HL_grid_info called?

Submit Restart Show Answers

3.7 Additional Resources

- GNU gprof Manual
- gprof: a Call Graph Execution Profiler
- Programmer's Toolkit: Profiling programs using gprof

4 PerfSuite

4.1 Overview

PerfSuite is a set of tools and supporting libraries designed to provide an easy-to-use, high-level view of application performance. It differs from traditional time-based statistical profiling tools, such as gprof, in that it enables performance analysis using hardware performance counters. These counters provide performance data such as cycle and instruction counts, cache and memory access statistics, branch behavior statistics, and functional unit and pipeline status. While time-based profiles tell you where your software spends its time, hardware performance measurements can help you understand what the processor is doing and how well your code is being mapped on to the underlying architecture. Hardware measurements also pinpoint particular reasons why the CPU is stalling rather than accomplishing useful work.

PerfSuite can be used without changing a single line of application source code, recompiling, or relinking. It adds minimal overhead to application runtime, an important consideration when collecting performance measurement from long-running applications. Nearly all data (e.g. input, output, configuration) in PerfSuite is represented as XML (eXtensible Markup Language) providing the ability to manipulate and transform the data in many ways. PerfSuite is available for x86, x86-64, and ia64 architectures

Several of the components in PerfSuite are built upon software from other sources. These include:

- The PAPI (Performance API) enables access to hardware performance counter data.
- XML support is provided by the Expat XML parser.
- The Tcl scripting language and associated Tk graphical toolkit provide a foundation for tying several independent tools

together as well as a very convenient and stable base for rapid cross-platform GUI development. In addition to the core Tcl/Tk base, PerfSuite uses the Tcl support library/package tDOM.

Lesson Objectives

After completing this lesson you will be able to:

- State the main capability provided by PerfSuite for analyzing program performance.
- List the steps involved in analyzing a program using PerfSuite.
- Identify the capabilities provided by PerfSuite's command-line utilities.
- Identify the types of reports generated by PerfSuite.
- Describe how to customize your performance analysis using PerfSuite.
- State the command lines needed to generate and analyze Persuite performance data.

4.2 Running PerfSuite

The following PerfSuite command-line utilities are used when analyzing a code:

- **psinv**: provides access to information about the characteristics of a machine (e.g., processor type, cache information, available hardware performance events).
- **psrun**: generates XML files containing raw counter or statistical profiling data from single-threaded, POSIX threads-based and MPI applications.
- **psprocess**: produces a hardware performance report from the performance measurements contained in the XML files generated by psrun (or the libpshwpc library directly).

These utilities can be used in a step-wise fashion to conduct performance analysis:

1. Run psinv to learn about the characteristics of the machine.
2. Decide what you want to measure and whether you are profiling or collecting aggregate counts. If not using the default configuration, select one of the provided configuration files or create a new one to customize your performance analysis.
3. Use psrun to generate XML files containing raw counter or statistical profiling data for your application.
4. Use psprocess to generate a report from the performance data stored in the XML documents by psrun.

In the following sections we briefly describe these steps to give you a basic understanding of how to use PerfSuite. For further detail we recommend looking at the additional resources listed at the end of this lesson.

Installing PerfSuite

If you are a TeraGrid user, PerfSuite is already installed on several of the TeraGrid machines on which you probably have accounts. If it is not already available on your system, you can download it from the PerfSuite software download page at http://perfsuite.ncsa.uiuc.edu/download/. You can either download PerfSuite as a tarball or install it from source, or you can download a small shell script called getsoftware from the PerfSuite site that will retrieve and install PerfSuite and optional supporting software over the network with a single command. Perfsuite is also available on Sourceforge at http://perfsuite.sourceforge.net/.

4.3 Retrieving Machine Information

Before you begin to analyze your code's performance it is best to learn about the architecture of the system on which you plan to run it. You can do that using PerfSuite's command-line utility psinv. The psinv utility displays information about your computer and available hardware performance events. The information generated by psinv is stored in PerfSuite XML output and is useful for later generating derived metrics (or for remembering where you ran your program!). It is available on x86, x86-64, and ia64 platforms. The x86/x86-64 version also shows processor features and descriptions.

The usage for the psinv command is:

```
psinv [ option ]
```

where option is:

Option	Description
-f	Describe features of CPU (IA-32 only).
-h	Provide brief help on usage

-p	List available PAPI events (if PerfSuite was configured with PAPI support).
-v	Show all available information.
-V	Report psinv version.

The following is an example report from psinv:

```
% psinv -v
System Information -
Processors:              2
Total Memory:            2007.16 MB
System Page Size:        16.00 KB

Processor Information -
Vendor:                  Intel
Processor family:        IPF
Model (Type):            Itanium
Revision:                6
Clock Speed:             800.136 MHz

Cache and TLB Information -
Cache levels:            3
Caches/TLBs:             7

Cache Details -
Level 1:
        Type:            Data
        Size:            16 KB
        Line size:       32 bytes
        Associativity:   4-way set associative

        Type:            Instruction
        Size:            16 KB
        Line size:       32 bytes
        Associativity:   4-way set associative

Level 2:
        Type:            Unified
        Size:            96 KB
        Line size:       64 bytes
        Associativity:   6-way set associative

Level 3:
        Type:            Unified
        Size:            4.00 MB
        Line size:       64 bytes
        Associativity:   4-way set associative

TLB Details -
Level 1:
        Type:            Data
        Entries:         32
        Pagesize (KB):   4 8 16 64 256 1024 4096 16384 65536 262144
        Associativity:   32-way set associative

        Type:            Instruction
        Entries:         64
        Pagesize (KB):   4 8 16 64 256 1024 4096 16384 65536 262144
        Associativity:   64-way set associative

Level 2:
        Type:            Data
        Entries:         96
        Pagesize (KB):  4 8 16 64 256 1024 4096 16384 65536 262144
        Associativity:  96-way set associative

PAPI Standard Event Information -
Standard events:        43
Non-derived events:     26
Derived events:         17

PAPI Standard Event Details -
Non-derived:
        PAPI_BR_INS:    Branch instructions
        PAPI_BR_PRC:    Conditional branch instructions correctly predicted
        PAPI_L1_DCA:    Level 1 data cache accesses
        PAPI_L1_DCM:    Level 1 data cache misses
        PAPI_L1_ICM:    Level 1 instruction cache misses
        PAPI_L2_DCA:    Level 2 data cache accesses
        PAPI_L2_DCR:    Level 2 data cache reads
        PAPI_L2_DCW:    Level 2 data cache writes
        PAPI_L2_ICM:    Level 2 instruction cache misses
        PAPI_L2_STM:    Level 2 store misses
        PAPI_L2_TCM:    Level 2 cache misses
        PAPI_L3_DCR:    Level 3 data cache reads
        PAPI_L3_DCW:    Level 3 data cache writes
        PAPI_L3_ICH:    Level 3 instruction cache hits
        PAPI_L3_ICM:    Level 3 instruction cache misses
        PAPI_L3_ICR:    Level 3 instruction cache reads
        PAPI_L3_STM:    Level 3 store misses
        PAPI_L3_TCM:    Level 3 cache misses
        PAPI_LD_INS:    Load instructions
        PAPI_MEM_SCY:   Cycles Stalled Waiting for memory accesses
        PAPI_SR_INS:    Store instructions
        PAPI_STL_ICY:   Cycles with no instruction issue
        PAPI_TLB_DM:    Data translation lookaside buffer misses
        PAPI_TLB_IM:    Instruction translation lookaside buffer misses
        PAPI_TOT_CYC:   Total cycles
        PAPI_TOT_INS:   Instructions completed
Derived:
        PAPI_BR_MSP:    Conditional branch instructions mispredicted
        PAPI_BR_NTK:    Conditional branch instructions not taken
        PAPI_BR_TKN:    Conditional branch instructions taken
        PAPI_FLOPS:     Floating point instructions per second
        PAPI_FP_INS:    Floating point instructions
        PAPI_L1_DCH:    Level 1 data cache hits
        PAPI_L1_ICR:    Level 1 instruction cache reads
        PAPI_L1_LDM:    Level 1 load misses
        PAPI_L1_TCM:    Level 1 cache misses
        PAPI_L2_DCM:    Level 2 data cache misses
        PAPI_L2_ICR:    Level 2 instruction cache reads
        PAPI_L2_LDM:    Level 2 load misses
        PAPI_L3_DCA:    Level 3 data cache accesses
        PAPI_L3_DCH:    Level 3 data cache hits
        PAPI_L3_DCM:    Level 3 data cache misses
        PAPI_L3_LDM:    Level 3 load misses
        PAPI_LST_INS:   Load/store instructions completed
```

4.4 Customizing Your Peformance Analysis

When using PerfSuite you need to decide what you want to measure and whether you are profiling or collecting aggregate counts. By default, PerfSuite collects aggregate event counts over the course of the application run. If you want to customize your performance analysis you need a configuration file, which is an XML document that describes the measurements to be taken. The PerfSuite distribution includes a number of alternative configuration files that you can tailor as needed or you can create a new one. Creating a new configuration file can be done with either a text editor or the command-line utility psconfig described later in this section.

For example, one of the most widely used metrics for performance measurement describes the average number of cycles required to complete an instruction (CPI). You can easily obtain this metric by counting the total number of graduated instructions and the total number of cycles. These events are predefined in PAPI and are called PAPI_TOT_INS and PAPI_TOT_CYC, respectively. The listing below shows a PerfSuite XML configuration file that could be used to measure these events.

```
<?xml version="1.0" encoding="UTF-8" ?>
<ps_hwpc_eventlist class="PAPI">
  <!-- ==================================================
       Configuration file to measure graduated instructions
       and total cycles.
       ================================================== -->
  <ps_hwpc_event type="preset" name="PAPI_TOT_INS" />

  <ps_hwpc_event type="preset" name="PAPI_TOT_CYC" />
</ps_hwpc_eventlist>
```

The XML document root element ps_hwpc_eventlist indicates this configuration is to be used for aggregate counting, where the total count of one or more performance events are measured and reported over the total runtime of your application.

Profiling with Psrun

Setting up for profiling is similar to counting: all you have to do is modify the XML configuration document so that the XML document root element is ps_hwpc_profile rather than ps_hwpc_eventlist and specify a different configuration file such as:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ps_hwpc_profile class="PAPI">
<ps_hwpc_event type="preset"
name="PAPI_BR_MSP threshold="100000" />
</ps_hwpc_profile>
```

When profiling, only one event is allowed in the configuration file and you may set an optional threshold or sampling rate.

Note: You do not need to recompile your code to use PerfSuite's psrun command but if you want to include line numbers in your profile report you must recompile using the -g option.

4.5 Performance Analysis

After you have determined what you want to measure and selected or created a configuration file, you use the psrun command to generate the performance data on your code. The psrun utility can do both hardware performance counting and profiling. In counting mode, it reports overall performance information on your application. In profiling mode, it relates hardware performance event occurrences to your program's source code much like time-based profilers such as gprof. By default, psrun operates in counting mode.

The syntax of the psrun command is:

```
psrun [option] command [args]
```

where option is:

-a, **--annotate** TEXT	Include annotation TEXT in output
-c, **--config** FILE	Specify configuration file
-d, **--domain** DOMAIN	Specify counting domain (default user)
-f, **--fork**	Monitor child processes (default no)
-F, **--format** FORMAT	Select output format (default XML)
-h, **--help**	Provide brief help on usage
-o, **--output** PREFIX	Specify alternate output XML document prefix. The keywords "stdout" and "stderr" are recognized (default is the command name)
-p, **--pthreads**	Enable POSIX thread support
-r, **--resource**	Collect resource usage information
-t, **--threshold** INTEGER	Specify the overflow threshold to be used for profiling

-V, **--version**	Report psrun version

and command [args] is the command line with arguments for your application.

Psrun takes a configuration file in XML format that specifies the desired measurement as input. If you do not specify a configuration file through the command-line option -c, psrun will use a default configuration. You can determine the default configuration file that will be used on your system by entering the command:

```
psrun -h
```

To run psrun on the program myprog without any options and using the default configuration file you would enter the command:

```
psrun myprog
```

If you want to use a custom configuration file, you can do it in one of two ways:

1. using the psrun option '-c filename'
2. setting the PS_HWPC_CONFIG environment variable to the configuration filename

Using the -c filename option, the command line for using psrun to collect profiling data would be:

```
psrun C -c papi_profile_cycles.xml myprog
```

where

- -C instructs PAPI to use xml configurations that are in the install path rather than the current directory.
- -c specifies that the configuration file papi_profile_cycles.xml which directs PAPI to collect file/line data is used rather than the default.

The command lines for running psrun on various types of programs are:

- For shared-memory (OpenMP/POSIX threads) programs: `psrun -p ompprogram`
- For message-passing (MPI) programs: `mpirun -np P psrun -f mpiprogram`
- For hybrid (MPI + OpenMP) programs: `mpirun -np P psrun -f -p hybridprogram`

The output of psrun is an XML document with the root element that contains the resulting performance data as well as information about other characteristics of the computer on which the performance data was collected. By default, the output XML document will be written to a file with the name:

```
command.[threadID].PID.hostname.xml
```

where command will be replaced with the name of the program being measured, PID will be replaced with the process ID of command, and threadID will be replaced with a unique integer thread identifier ranging from 0 to the number of additional threads created by command (threadID will only be present in the file name when support for POSIX threads is enabled with the option -p). You can specify a different output document prefix by using the option -o.

Here is an example XML document created by psrun. How to generate an easier to read report from the XML generated by psrun is described in the next section.

4.6 Post-Processing

After using psrun to generate one or more XML files with your performance data, you can use psprocess to produce a hardware performance report in plain-text format. The report's content varies according to the measurement done.

The sections common to both counting and profiling reports are:

- Report creation details
- Run details
- Machine information
- Run annotation defined by user

If the measurement consisted of aggregate counting, then the report will also include:

- Raw event counts
- Event descriptions and index
- Derived metrics -- calculated from the raw event counts

If the measurement consisted of profiling, then the report will also include:

- Characteristics of the profiling run such as sampling rate
- A breakdown of the samples at various levels of granularity (files, functions, and lines)

An example of a PerfSuite profile report is given in the Use Case: HPL section of this tutorial

These reports are dependent not only on options you supply when running psprocess, but by the information within the XML file generated by psrun.

The syntax of the psprocess command is:

```
psprocess [option] file ...
```

Some commonly used options are:

Option	Description
-a, --annotate	Specify annotation string (must also use -c)
-c, --combine	Operate in combine mode
-h, --help	Display brief help
-o, --output FILE	Specify output file name (default stdout)
-V, --version	Report psprocess version number

Additionals options for the data measured by profiling are:

Option	Description
-s, --show LIST	Select what to report. LIST is a comma-separated list of "modules", "files", "functions", "lines".
-t, --threshold	NUM Set minimum % to report
-x, --xmlout	Produce XML as output

If you generated a single XML file named file.xml using psrun, you would then use the following command to generate a report using psprocess:

```
$ psprocess file.xml
```

If you generated multiple XML files, psprocess can combine the results and then generate a report from the combined file that summarizes the information contained in them with descriptive statistics (mean, max, min, sum, stddev). The command for combining files with psprocess is as follows:

```
$ psprocess -c psrun.*.xml > combined.xml
$ psprocess combined.xml
```

4.7 Self Test

<?xml version="1.0" encoding="UTF-8"?>   @import url(base.css); @import url(content.css); PerfSuite

Reflection

Describe the main capability provided by PerfSuite for analyzing program performance.

Click here

PerfSuite enables performance analysis using hardware performance counters. These counters provide performance data such as cycle and instruction counts, cache and memory access statistics, branch behavior statistics, and functional unit and pipeline status. It can help you understand what the processor is doing and how well your code is being mapped on to the underlying architecture.

Reflection

List the general steps involved in analyzing a program using PerfSuite.

Click here

1. Run psinv to learn about the characteristics of the machine.
2. Decide what you want to measure and whether you are profiling or collecting aggregate counts. If not using the default configuration, select one of the provided configuration files or create a new one to customize your performance analysis.
3. Use psrun to generate XML files containing raw counter or statistical profiling data for your application.
4. Use psprocess to generate a report from the performance data stored in the XML documents by psrun.

Fill-in-the-Blanks

Fill in the blanks with the PerfSuite command-line utility that performs the capability described.

- generates XML files containing raw counter or statistical profiling data from single-threaded, POSIX threads-based and MPI applications.

- produces a hardware performance report from the the XML files containing PerfSuite generated performance measurements.

- provides access to information about the characteristics of a machine (e.g., processor type, cache information, available hardware performance events).

Submit Restart Show Answers

True-False Question

PerfSuite cannot generate a statistical profile report.

True ○ False ○

Reflection

Describe how to customize your performance analysis using PerfSuite.

Click here

Customization of PerfSuite analyses can be done through configuration files. The PerfSuite distribution includes a number of alternative configuration files that you can tailor as needed or you can create a new one using a text editor. When using a custom configuration file, you can either include the filename in your psrun command by using the '-c filename' option or set the PS_HWPC_CONFIG environment variable to the configuration filename.

Multi-choice

Which psrun option is used to determine the default configuration file that will be used on your system.

○ -c
○ -a
○ -h
○ none of the above

Multi-choice

Which of the following options would you use when running psrun with a hybrid MPI/OpenMP program?

○ -f
○ -h
○ -p
○ -f -p

Reflection

Using psrun, you generated an XML file named myprog.12345.co-compute1.xml. What command-line would you issue to generate a hardware performance report on the file?

Click here

psprocess myprog.12345.co-compute1.xml

4.8 Additional Resources

- PerfSuite Website
- Measuring and Improving Application Performance with PerfSuite
- PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux

5 TAU

5.1 Overview

The TAU (Tuning and Analysis Utilities) parallel performance system is a portable suite of tuning and analysis utilities for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. It can be used with programs written in C, C++, Fortran, Java and Python; runs on all HPC platforms; and supports multiple parallel programming paradigms — multi-threading, message passing, mixed-mode, and hybrid. It is distributed under an open source license and can be downloaded from the Software Download Requests page on the TAU website.

With TAU, you can observe events at all levels of resolutions: function, loop, and statements. TAU also provides phase-based profiling to measure performance with respect to application-defined phases of execution and snapshots to characterize changes in the performance profile during the execution of the application. TAU's analysis tools can perform experiment and scaling studies to summarize an application's performance in different environments.

The TAU distribution includes the performance analysis tools ParaProf and PerfExplorer for profile analysis and Jumpshot for trace analysis. Interoperability with other formats such as PerfSuite profiles, Vampir and Scalasca traces is also provided.

TAU is a very powerful set of utilities that you can use to analyze your applications. In this lesson, we will only cover the basics of how to use it so you can begin to understand the capabilities it can provide. The learning curve for TAU is greater than the tools we have described so far but you can also do much more detailed analysis using it.

Lesson Objectives

After completing this lesson you will be able to:

- State the main capability provided by TAU for analyzing program performance.
- List the steps involved in analyzing a program using TAU.
- List the steps in identifying the requirements for selecting a TAU makefile for setting up instrumentation.
- Select a makefile for instrumenting your code.
- Select environment variables to specify the type of data you want to generate for an analysis.
- Select one of TAU's bundled utilities for analyzing your performance data.

5.2 Running TAU

The TAU utilities can be used in a step-wise fashion to conduct performance analysis:

1. Decide what you want to measure, instrument your code, select the appropriate makefile, and recompile your code.
2. Execute your code to generate measurement data.
3. Analyze your data using one of TAU's analysis tools such as Paraprof for viewing profile data or Jumpshot for viewing trace data.

In the following sections we briefly describe these steps to give you a basic understanding of how to use TAU. For further detail we recommend looking at the additional resources listed at the end of this lesson.

Installing TAU

If TAU is not already installed on your system, you can download it from the Software Download Requests page of the TAU website and then refer to the TAU Install Guide for installation instructions.

Running TAU on TeraGrid Systems

TAU is installed on many TeraGrid systems and you can easily add it to your software environment using a predefined softenv key or module. For example, to use TAU with MPI on abe at NCSA, you would enter the command line:

```
soft add +tau
```

To use it at PSC using modules, you would enter the command line:

```
module load tau
```

Note that the command line entered will differ depending on the programming model used. For example, if you have an openmp code, at PSC you would load this module instead:

```
module load tau/2.18-openmp
```

After adding TAU to your environment using softenv or modules, the appropriate TAU compiler wrapper will be added to your PATH and set in your Makefile.

Refer to the resource provider's system documentation for specific instructions on how to use TAU on the system you are running.

5.3 Instrumenting and Compiling Your Code

The first step in using TAU is to instrument and recompile your code. TAU uses three methods of instrumentation. These are:

1. binary rewriting using the Dyninst Tool
2. compiler-based instrumentation
3. source transformation using the Program Database Toolkit (PDT)

Binary Rewriting

TAU uses the Dyninst API to enable instrumentation without the need to edit and recompile your application's source code. The Dyinst API provides binary rewriter support for shared libraries. TAU must be configured with Dyninst in order to use this feature.

You use the tau_run command to instrument your application binary using Dyninst.

Using tau_run select the -o option to name the rewritten binary:

```
% tau_run a.out -o a.inst.out
% mpirun -np 4 a.inst.out
```

Compiler-based Instrumentation

The TAU scripts for compiling instrumented Fortran, C, and C++ programs are tau_f90.sh, tau_cc.sh, and tau_cxx.sh respectively. For example you can use tau_cc.sh to compile a manually instrumented C program by typing:

```
tau_cc.sh -tau_options=-optCompInst samplecprogram.c
```

Choosing a Makefile Stub

There will be several TAU configurations installed on a system based on the options selected when installing TAU. Each configuration includes a single makefile with its associated libraries and scripts. These makefiles are named based on the configuration options chosen. For example, the makefile for TAU configured with MPI, PDT, PGI compilers and the '-nocomm' option is named: Makefile.tau-nocomm-mpi-pdt-pgi.

First, choose an appropriate TAU Makefile stub to use for setting the necessary environment variables. Your choice should be based on the following:

1. Type of measurement (e.g., trace, phase, papi)
2. Your compiler (e.g., Sun, icpc, pgi)
3. MPI, OpenMP, or hybrid instrumentation (MPI+OpenMP)

4. Auto-instrumentation (e.g., pdt)

For example, to generate a callpath profile with MPI select the makefile named Makefile.tau-mpi-pdt.

You then set the environment variable TAU_MAKEFILE to your selected makefile:

```
setenv TAU_MAKEFILE Makefile.tau-mpi-pdt
```

Setting Compiler Options

Next you need to set the TAU options you want using the TAU_OPTIONS environment variable:

```
setenv TAU_OPTIONS
```

Common options include:

- -optVerbose Enable verbose output (default: on)
- -optKeepFiles : Do not remove intermediate files
- -optShared : Use shared library of TAU
- -optCompInst : Use compiler-based instrumentation
- -optPDTInst : Use PDT-based instrumentation
- -optTauSelectFile=/path/to/select.tau : Specify selective instrumentation file

For a list of the available compiler options on your system, enter the following command:

```
tau_compiler.sh -help
```

Source Transformation using PDT

Using PDT, TAU gathers information from source code such as the location of function entry and exit points and uses this information to insert calls for automatic instrumentation. For automatic source instrumentation, you will need a version of TAU built with PDT and a makefile name containing pdt. As with compiler-based instrumentation, the TAU scripts for compiling Fortran, C, and C++ programs are tau_f90.sh, tau_cc.sh, and tau_cxx.sh respectively. To automatically instrument a code with PDT, enter the following command line:

```
tau_cc.sh samplecprogram.c
```

Selective Instrumentation

When using source transformation with PDT, you can also customize your instrumentation to select parts of your code rather than all of it by using a selective instrumentation file. A selective instrumentation file is a text file with instructions on which part of your code you want to instrument. You can automatically create a selective instrumentation file using ParaProf or manually using the following:

- Wildcards for the routine names you want to include specified by a pound sign (#) (to specify a leading wildcard, place the entry in quotes).
- Wildcards for file names specified by an asterik (*).

Below is an example selective instrumentation file taken from the TAU User Guide:

```
#Tell tau to not profile these functions
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
# The next line excludes all functions beginning with "sort_" and having
# arguments "int *"
void sort_#(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST
#Exclude these files from profiling
BEGIN_FILE_EXCLUDE_LIST
*.so
END_FILE_EXCLUDE_LIST
BEGIN_INSTRUMENT_SECTION
# A dynamic phase will break up the profile into phase where
# each events is recorded according to what phase of the application
# in which it occured.
dynamic phase name="foo1_bar" file="foo.c" line=26 to line=27
# instrument all the outer loops in this routine
loops file="loop_test.cpp" routine="multiply"
# tracks memory allocations/deallocations as well as potential leaks
memory file="foo.f90" routine="INIT"
# tracks the size of read, write and print statements in this routine
io file="foo.f90" routine="RINB"
END_INSTRUMENT_SECTION
```

After creating your file, use the

```
-tau_options=-optTauSelectFile=<file>
```

option to enable selective instrumentation.

5.4 Generating Measurement Data

After instrumenting and compiling your code, running it in the usual manner will generate the measurement data files. But before generating measurement data, you need to set some environment variables to specify the type of data you want to generate. A complete list of the TAU environment variables can be found in Appendix A. Environment Variables of the TAU Reference Guide.

Profiling

By default, TAU will store profile data to the current directory; you can specify a different location using the PROFILEDIR environment variable as follows:

```
setenv PROFILEDIR /home/garnold/profiledata/mycode
```

To specify that you want to see the steps TAU takes when your application is running, set the TAU_VERBOSE environment variable:

```
setenv TAU_VERBOSE 1
```

Although you can enable profiling using:

```
setenv TAU_PROFILE 1
```

it is not necessary since TAU does this by default. You must set the value to 0 to disable profiling.

Callpath Profiling

To enable TAU callpath profiling at a depth of ten you would specify:

```
setenv TAU_CALLPATH 1
setenv TAU_CALLPATH_DEPTH 10
```

Using Hardware Counters

To make measurements using hardware counters you first need to make sure that you are using a makefile with papi in its name and find out which PAPI events your system supports (by using the papi_avail command). Then you set the environment variable TAU_METRICS to a colon delimited list of the PAPI metrics you would like to measure.

For example:

```
setenv TAU_METRICS PAPI_FP_OPS\:PAPI_L1_DCM
```

Reducing Measurement Overhead

Profiling with TAU adds overhead to a run. TAU compensates for this by automatically throttling short running functions (functions that take less than 10 microseconds to execute and execute more than 100,000 times). This can be disabled by setting the TAU_THROTTLE environment variable to 0.

You may also customize these settings using the TAU_THROTTLE_NUMCALLS and TAU_THROTTLE_PERCALL environment variables. For example, to change the values to 2 million and 5 microseconds per call use:

```
setenv TAU_THROTTLE_NUMCALLS 2000000
setenv TAU_THROTTLE_PERCALL 5
```

Finally, run the application as you normally would:

```
% mpirun -np 4 matrix
```

Tracing

To enable tracing, you set the TAU_TRACE environment variable to 1. As with profiling, by default the trace data is stored in thecurrent directory; you can specify a different location using theTRACEDIR environment variable.

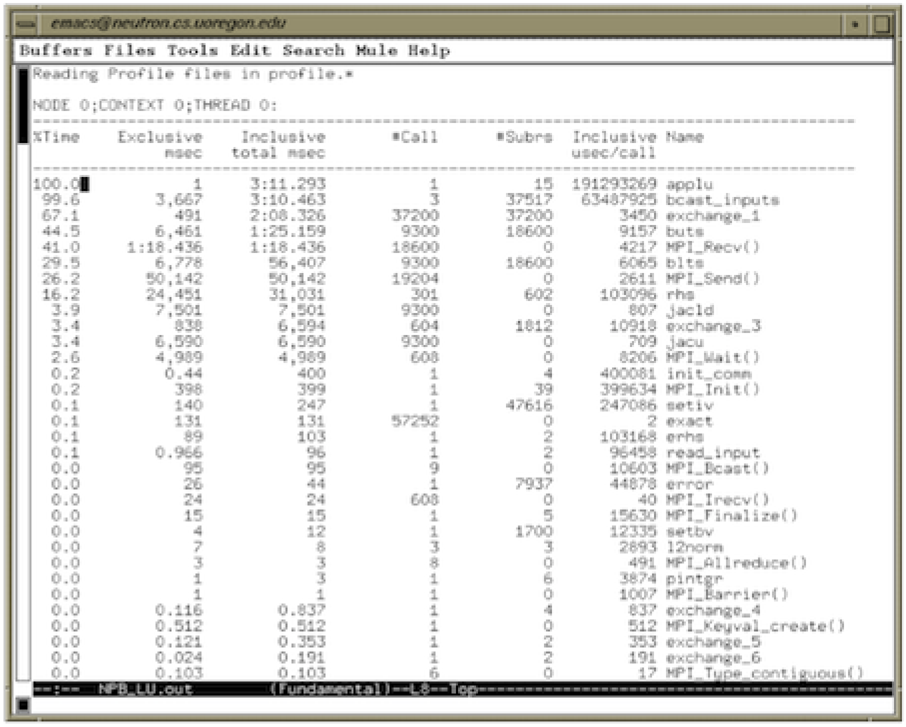
```
setenv TRACEDIR /home/garnold/tracedata/mycode
setenv TAU_TRACE 1
```

After running your code, a tracefile and an event file are generated for each processor. These files can be merged by using the tau_treemerge.pl script and converted into another format using the tau2otf, tau2vtf, or tau2slog2 scripts.

5.5 Analyzing Measurement Data

PProf

TAU's text-based tool, PProf, can be used to quickly display a profile report. Remember that TAU performance data for each metric is organized into subdirectories within the runtime directory. To generate a quick summary of data for any given metric, move to the directory for that metric and run the pprof command. The following graphic shows an example of a display generated by pprof.


emacs@neutron.cs.uoregon.edu
Buffers Files Tools Edit Search Mule Help

```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0            1     3:11.293           1          15  191293269 applu
 99.6        3,667     3:10.463           3       37517   63487925 bcast_inputs
 67.1          491     2:08.326       37200       37200       3450 exchange_1
 44.5        6,461     1:25.159        9300       18600       9157 buts
 41.0     1:18.436     1:18.436       18600           0       4217 MPI_Recv()
 29.5        6,778       56,407        9300       18600       6065 blts
 26.2       50,142       50,142       19204           0       2611 MPI_Send()
 16.2       24,451       31,031         301         602     103096 rhs
  3.9        7,501        7,501        9300           0        807 jacld
  3.4          838        6,594         604        1812      10918 exchange_3
  3.4        6,590        6,590        9300           0        709 jacu
  2.6        4,989        4,989         608           0       8206 MPI_Wait()
  0.2         0.44          400           1           4     400081 init_comm
  0.2          398          399           1          39     399634 MPI_Init()
  0.1          140          247           1       47616     247086 setiv
  0.1          131          131       57252           0          2 exact
  0.1           89          103           1           2     103168 erhs
  0.1        0.966           96           1           2      96458 read_input
  0.0           95           95           9           0      10603 MPI_Bcast()
  0.0           26           44           1        7937      44878 error
  0.0           24           24         608           0         40 MPI_Irecv()
  0.0           15           15           1           5      15630 MPI_Finalize()
  0.0            4           12           1        1700      12335 setbv
  0.0            7            8           3           3       2893 l2norm
  0.0            3            3           8           0        491 MPI_Allreduce()
  0.0            1            3           1           6       3874 pintgr
  0.0            1            1           1           0       1007 MPI_Barrier()
  0.0        0.116        0.837           1           4        837 exchange_4
  0.0        0.512        0.512           1           0        512 MPI_Keyval_create()
  0.0        0.121        0.353           1           2        353 exchange_5
  0.0        0.024        0.191           1           2        191 exchange_6
  0.0        0.103        0.103           6           0         17 MPI_Type_contiguous()
--:--  NPB_LU.out        (Fundamental)--L8--Top-------------------------------------------
```

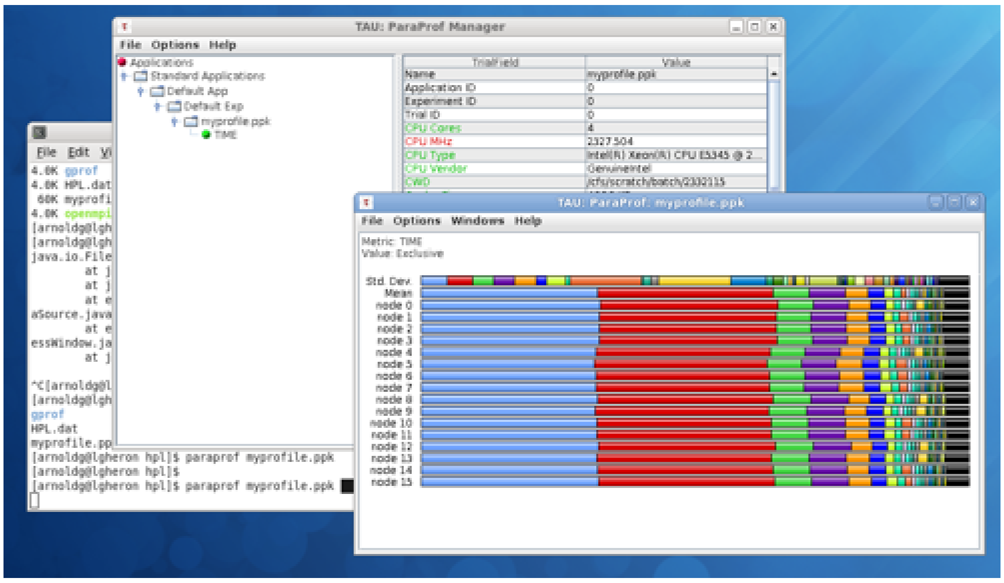
ParaProf

To get more in-depth information than that provided by PProf, you can use ParaProf, TAU's visualization tool. ParaProf is java-based GUI this is included with the TAU distribution. A common setup is to have a local installation of TAU (on your laptop, desktop, or local Linux resource) and manipulate the GUI locally. If you're geographically distant from the supercomputer or cluster, then long network latency will make remote use of the GUI cumbersome. If you have TAU installed with a default configuration and build, copy the files to your local site. The default install of TAU contains the ParaProf viewer you need to process the files.

To make it simpler to move TAU profiles, the TAU suite includes an option to pack up a set of profiles into a single file that can be used with the viewer. The files can be packed at the remote site and then the single file copied to your local site for analysis:

```
#Remote Site:
$ paraprof --pack myprofile.ppk
Loading data...
Packing data...
... sftp , scp, filezilla or other ftp-like client moves the file to the local site...
# Local Site:
$ paraprof myprofile.ppk
```

Launching ParaProf will display manager and profile data windows as shown in the image below.



TrialField	Value
Name	myprofile.ppk
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	4
CPU MHz	2327.504
CPU Type	Intel(R) Xeon(R) CPU E5345 @ 2...
CPU Vendor	GenuineIntel
CWD	/cfs/scratch/batch/2332115

```
4.0K gprof
4.0K HPL.dat
 60K myprofi
4.0K openmpi
[arnoldg@lgh
[arnoldg@lgh
java.io.File
        at j
        at j
        at e
aSource.java
        at e
essWindow.ja
        at j

^C[arnoldg@l
[arnoldg@lgh
gprof
HPL.dat
myprofile.pp
[arnoldg@lgheron hpl]$ paraprof myprofile.ppk
[arnoldg@lgheron hpl]$
[arnoldg@lgheron hpl]$ paraprof myprofile.ppk
```

The ParaProf interface becomes intuitive once you become accustomed to just a few techniques:

- Left click once to get the default action for an item
- Right click any item for more options
- If in doubt, click ... clicking on words and objects (such as line segments) is fair game in ParaProf.

ParaProf provides a multitude of visualization capabilities for analyzing your measurement data and we do not attempt to describe all of them in this tutorial. For a more in-depth look at ParaProph, see the ParaProf User Guide.

Jumpshot

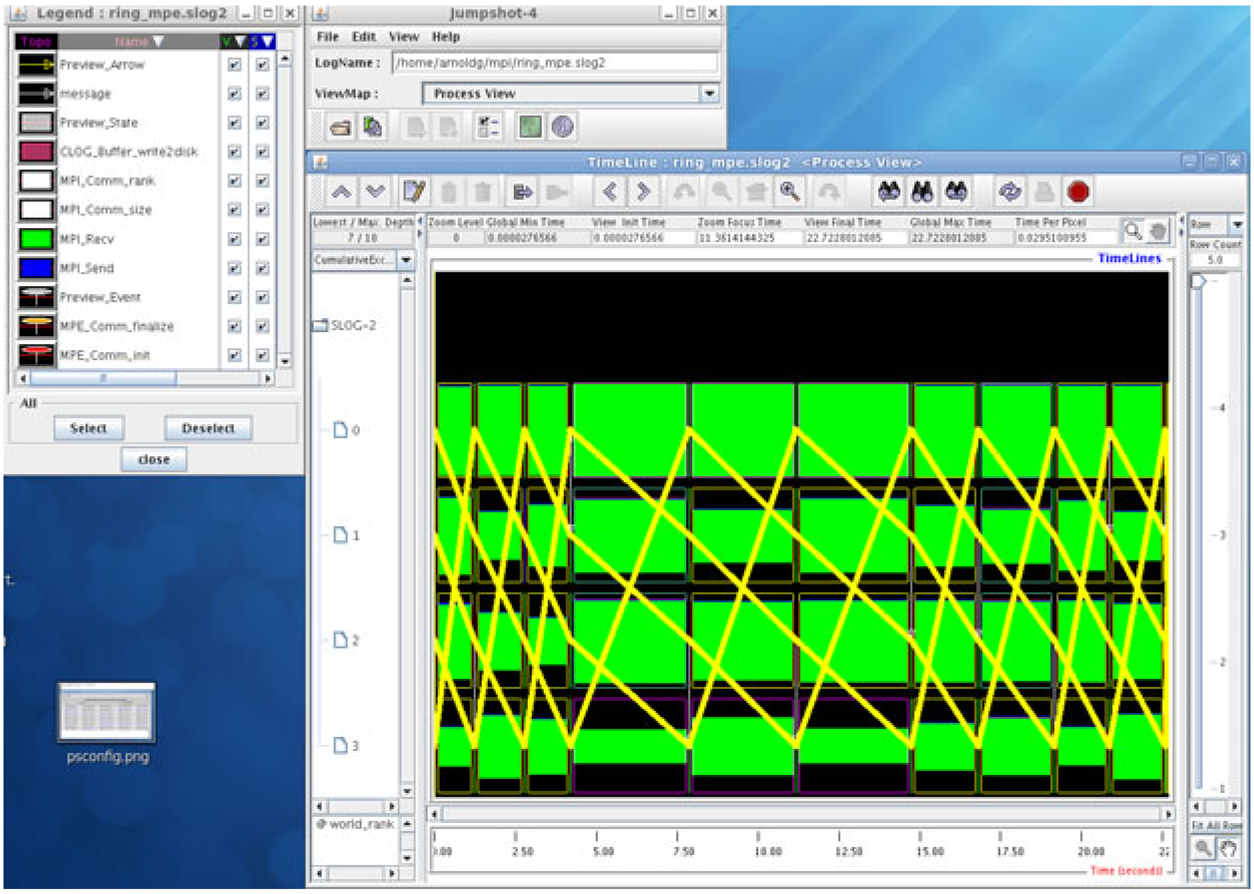
The Jumpshot tool bundled with TAU can be used to analyze trace data. Jumpshot is a Java-based visualization tool for doing postmortem performance analysis. To use Jumpshot you need to first merge and convert your TAU traces to slog2 format using the with tau2slog2 command:

```
% tau_treemerge.pl
% tau2slog2 tau.trc tau.edf -o tau.slog2
```

and then launch Jumpshot using the command:

```
jumpshot tau.slog2
```

Jumpshot's main window will be displayed showing the entire trace. A sample window is shown below (You can click on the image for a larger view.).



For more on using Jumpshot, see the Jumpshot webpage and user guide.

5.6 Self Test

<?xml version="1.0" encoding="UTF-8"?> @import url(base.css); @import url(content.css); TAU-sk

Reflection
List the general steps involved in analyzing a program using TAU.

Click here

The TAU utilities can be used in a step-wise fashion to conduct performance analysis:

1. Decide what you want to measure, instrument your code, select the appropriate makefile, and recompile your code.
2. Execute your code to generate measurement data.
3. Analyze your data using one of TAUs analysis tools such as Paraprof for viewing profile data or Jumpshot for viewing trace data.

Multi-choice
Which is not a consideration when choosing a Makefile stub for instrumenting your code?

- ○ The type of measurement you want to make.
- ○ The number of processors on which you'll be running your code.
- ○ Which parallel programming model you will be using, e.g. MPI or OpenMP.
- ○ What compiler you'll be using.
- ○ Whether you are going to use automatic source instrumentation.

Multi-choice
Which of the following tools in TAU enables you to instrument your code without the need to edit and recompile it?

- ○ Dyinst API

○ Program Database Toolkit

Fill-in-the-Blanks
Read the paragraph below and fill in the missing words.

_______________ is the environment variable used to reduce the overhead that TAU adds to a run.

Submit Restart Show Answers

True-False Question
Both words and line segments are "clickable" in Paraprof and will launch new profile views. (True or false)
True ○ False ○

True-False Question
In TAU, you use environment variables to specify the type of data you want to generate.

True ○ False ○

True-False Question
You must select an appropriate makefile to enable profiling or tracing.

True ○ False ○

Multi-choice
Which of the following tools bundled with TAU can be used to visualize TAU's trace data?

○ PerfExplorer
○ ParaProf
○ Jumpshot

5.7 Additional Resources

We only briefly covered the many capabilities that TAU provides for analyzing the performance of your code. There are many online resources available for additional information. The following are a few that we recommend:

- TAU User Guide
- TAU Quick Reference
- Some Common Application Scenarios
- TAU Examples on Pittsburgh Supercomputing Center website
- TAU documentation at University of Oregon
- TAU Portal at University of Oregon
- TAU User Guide (TeraGrid at PSC)
- User Support for TAU on the TeraGrid
- The TAU Parallel Performance System

6 Use Case: HPL

6.1 Introduction

In this lesson we use a portable implementation of the High Performance Linpack Benchmark for Distributed-Memory Computers (HPL) to show how to use some of the capabilities provided by the following performance tools described in this tutorial:

- Strace
- Gprof

- Perfsuite
- TAU

The Linpack HPL benchmark is the basis for rankings for the Top500 Supercomputing Sites. We chose HPL because first, it is an application used by a large number of researchers engaged in high performance benchmarking; and second, it is already well-optimized enabling us to show what a well-tuned application looks like when using performance tools to analyze code.

If you want to run the HPL examples on your system, instructions for installing it are given in the next section. However, you may also simply follow along with the examples given.

6.2 Installing HPL

To run the HPL software on your system you will need:

1. The HPL source code installed. You can find the source code and installation instructions at http://www.netlib.org/benchmark/hpl/.
2. An implementation of the Message Passing Interface **MPI** (1.1 compliant).We have found that OpenMPI works well with HPL, especially if you've installed HPL on a laptop or standalone desktop. If you do not already have MPI installed, you can download a variety of implementations from http://www.mcs.anl.gov/research/projects/mpi/implementations.html. The source code for OpenMPI is available from http://www.open-mpi.org/.
3. An implementation of either the Basic Linear Algebra Subprograms (BLAS) or the Vector Signal Image Processing Library (VSIPL). The BLAS and ATLAS (a tuned BLAS library) source codes are available from http://www.netlib.org/blas/ and http://sourceforge.net/projects/math-atlas/files/. The VSIPL source code is available from http://www.vsipl.org/documents/.

Note: If you have an XSEDE account, implementations of the required software may already be available. To find available software see the Comprehensive Software Search.

You should be aware that building HPL for the first time can be a complicated process. It depends on the installation of other software packages and libraries and requires some tweaking of the appropriate makefile in order to locate them. In addition, because the installation process on each platform and OS will be slightly different there is no single step-by-step process documented. Also, since each system varies in where they store libraries and compilers, you'll need to make sure paths are set correctly. For instance, the top level directory name may need to be changed from hpl to hpl-2.0; the location of your MPI package depends on which MPI you□ve installed, and so on. Instead of relying on path variables, you may also wish to specify the entire path to ensure accuracy.

After installing HPL, you can test it by issuing the following command from the top level directory:

```
cd bin/<arch>; mpirun -np 4 xhpl
```

The output should look similar to this:

```
[arnoldg@honest2 Linux_PII_CBLAS]$ more xhpl.o2333116
/dev/sda2 on /tmp type ext2 (rw)
----------------------------------------
Begin Torque Prologue (Thu Dec 17 10:35:57 2009)
Job ID:           2333116
Username:         arnoldg
Group:            ags
Job Name:         xhpl
[□]
[1,0]<stdout>:              2 tests completed and passed residual checks,
[1,0]<stdout>:              0 tests completed and failed residual checks,
[1,0]<stdout>:              0 tests skipped because of illegal input values.
[1,0]<stdout>:----------------------------------------------------------------------------
[1,0]<stdout>:
[1,0]<stdout>:End of Tests.
[1,0]<stdout>:============================================================================
2803.264u 55.612s 6:09.33 774.0%        0+0k 0+0io 2457pf+0w
----------------------------------------
Begin Torque Epilogue (Thu Dec 17 10:42:34 2009)
Job ID:           2333116
```

```
Username:         arnoldg
Group:            ags
Job Name:         xhpl
Session:          18599
Limits:           ncpus=1,nodes=2:ppn=8,walltime=00:30:00
Resources:        cput=01:35:19,mem=7808520kb,vmem=10408796kb,walltime=00:06:10
Job Queue:        debug
Account:          aaa
Nodes:            abe0909 abe0911
Killing leftovers...
End Torque Epilogue
----------------------------------------
```

After you've built HPL, you may wish to tune it by modifying the default input data file, HPL.dat,that is located in the HPL executable directory hpl/bin/<arch>/xhpl. For a more in-depth look at the structure and configuration of HPL.dat, see the HPL Tuning documentation page at http://www.netlib.org/benchmark/hpl/tuning.html, Below, is a sample configuration file with simple usage for 16 processors.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
28262     Ns
1            # of NBs
128         NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
4           Ps
4           Qs
16.0         threshold
1            # of panel fact
0         PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4 2 8        NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
0         RFACTs (0=left, 1=Crout, 2=Right)
2            # of broadcast
1 4          BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1 0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
256          swapping threshold
1            L1 in (0=transposed,1=no-transposed) form
1            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8             memory alignment in double (> 0)
```

6.3 Profiling System Calls with Strace

Here we'll use strace to profile HPL's system calls. We chose to use strace because it can be used without modifying the source code you are analyzing. It is also easy to use, you just mix it into your command line and it will trace system calls. The output generated will help you understand what a process is trying to do at a given runtime and therefore how the system time is being spent in your program. Strace is particularly useful for examining I/O performance.

First, we instrument the planned run for strace by modifying the command line in the job script. For our run line we used openmpi with an argument to label the rank information. For strace we used the -c flag which will give you a summary table of all system calls used by the process, including errors.

```
time mpirun --tag-output -np ${NP} -machinefile ${PBS_NODEFILE} \
   strace -c ~/hpl-2.0/bin/Linux_PII_CBLAS/xhpl
```

We used the following perl script to clean up the output and show just the first few lines of strace information for each process:

```
$ more strace_read.pl
#!/usr/bin/perl
# grep 1,4] xhpl.e2297576 | head -10
for ($i=0; $i<12; $i++)
{
   system" grep 1,$i\] xhpl.e* | head -10 ";
}
```

Below are the strace results for the HPL run as processed through the perl script above. Note that most of the system call activity is due to memory allocation (munmap), which is typical of applications that use routines like malloc() and free() to dynamically allocate memory. Notice that read and write are somewhat insignificant as far as percent time per system call because the application does very little in terms of I/O. It reads the hpl.dat file and writes output to standard out but that is minimal. If your application does a lot of I/O you might want to use strace with it to see how that time is spent on the system side. Running strace without the □c option can give you insight into the real-time I/O patterns of a process. The strace man page provides more information and example usage.

```
[arnoldg@lgheron hpl]$ ./strace_read.pl
[1,0]<stderr>:Process 10316 detached
[1,0]<stderr>:% time seconds usecs/call calls errors syscall
[1,0]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,0]<stderr>: 96.89 1.268198 1833 692 1 munmap
[1,0]<stderr>: 1.25 0.016315 16 1011 write
[1,0]<stderr>: 1.02 0.013384 30 446 268 open
[1,0]<stderr>: 0.23 0.003016 15 205 read
[1,0]<stderr>: 0.20 0.002563 0 6404 9 poll
[1,0]<stderr>: 0.16 0.002114 11 193 close
[1,0]<stderr>: 0.11 0.001453 43 34 writev
[1,1]<stderr>:Process 10321 detached
[1,1]<stderr>:% time seconds usecs/call calls errors syscall
[1,1]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,1]<stderr>: 95.54 1.256985 1958 642 1 munmap
[1,1]<stderr>: 2.76 0.036277 58 623 write
[1,1]<stderr>: 1.07 0.014080 32 445 268 open
[1,1]<stderr>: 0.18 0.002397 0 7922 7 poll
[1,1]<stderr>: 0.15 0.002012 10 204 read
[1,1]<stderr>: 0.15 0.001999 118 17 writev
[1,1]<stderr>: 0.09 0.001221 8 150 fstat
[1,2]<stderr>:Process 10318 detached
[1,2]<stderr>:% time seconds usecs/call calls errors syscall
[1,2]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,2]<stderr>: 93.73 1.350478 2084 648 1 munmap
[1,2]<stderr>: 4.87 0.070191 82 859 write
[1,2]<stderr>: 0.78 0.011210 25 445 268 open
[1,2]<stderr>: 0.22 0.003169 16 204 read
[1,2]<stderr>: 0.17 0.002426 0 6822 10 poll
[1,2]<stderr>: 0.14 0.001998 111 18 writev
[1,2]<stderr>: 0.05 0.000740 5 150 fstat
[1,3]<stderr>:Process 10319 detached
[1,3]<stderr>:% time seconds usecs/call calls errors syscall
[1,3]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,3]<stderr>: 95.23 1.358723 2116 642 1 munmap
[1,3]<stderr>: 3.14 0.044855 74 603 write
[1,3]<stderr>: 0.90 0.012786 29 445 268 open
[1,3]<stderr>: 0.43 0.006120 30 204 read
[1,3]<stderr>: 0.20 0.002869 0 7172 8 poll
[1,3]<stderr>: 0.07 0.000998 998 1 execve
[1,3]<stderr>: 0.02 0.000251 0 751 1 mmap
[1,4]<stderr>:Process 10317 detached
[1,4]<stderr>:% time seconds usecs/call calls errors syscall
[1,4]<stderr>:------ ----------- ----------- --------- --------- ----------------
```

```
[1,4]<stderr>: 94.69 1.347330 2115 637 1 munmap
[1,4]<stderr>: 3.46 0.049167 60 819 write
[1,4]<stderr>: 1.08 0.015322 34 445 268 open
[1,4]<stderr>: 0.23 0.003279 16 204 read
[1,4]<stderr>: 0.22 0.003085 181 17 writev
[1,4]<stderr>: 0.19 0.002697 0 6655 10 poll
[1,4]<stderr>: 0.07 0.000998 998 1 execve
[1,5]<stderr>:Process 10322 detached
[1,5]<stderr>:% time seconds usecs/call calls errors syscall
[1,5]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,5]<stderr>: 95.73 1.351924 2109 641 1 munmap
[1,5]<stderr>: 2.75 0.038851 62 623 write
[1,5]<stderr>: 0.65 0.009109 20 445 268 open
[1,5]<stderr>: 0.28 0.003998 235 17 writev
[1,5]<stderr>: 0.23 0.003184 16 204 read
[1,5]<stderr>: 0.19 0.002626 0 6924 11 poll
[1,5]<stderr>: 0.13 0.001906 13 150 fstat
[1,6]<stderr>:Process 10320 detached
[1,6]<stderr>:% time seconds usecs/call calls errors syscall
[1,6]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,6]<stderr>: 95.17 1.253363 1788 701 1 munmap
[1,6]<stderr>: 3.34 0.043949 47 935 write
[1,6]<stderr>: 0.77 0.010086 23 445 268 open
[1,6]<stderr>: 0.32 0.004150 20 204 read
[1,6]<stderr>: 0.21 0.002776 0 7018 12 poll
[1,6]<stderr>: 0.09 0.001246 8 150 fstat
[1,6]<stderr>: 0.08 0.000999 59 17 writev
[1,7]<stderr>:Process 10323 detached
[1,7]<stderr>:% time seconds usecs/call calls errors syscall
[1,7]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,7]<stderr>: 95.46 1.280058 1994 642 1 munmap
[1,7]<stderr>: 2.50 0.033512 56 601 write
[1,7]<stderr>: 1.22 0.016426 37 445 268 open
[1,7]<stderr>: 0.22 0.002998 176 17 writev
[1,7]<stderr>: 0.19 0.002482 0 12130 11 poll
[1,7]<stderr>: 0.15 0.002063 10 204 read
[1,7]<stderr>: 0.07 0.000999 999 1 mkdir
[1,8]<stderr>:Process 7007 detached
[1,8]<stderr>:% time seconds usecs/call calls errors syscall
[1,8]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,8]<stderr>: 96.67 1.322331 1914 691 1 munmap
[1,8]<stderr>: 1.42 0.019445 17 1131 write
[1,8]<stderr>: 0.76 0.010439 23 445 268 open
[1,8]<stderr>: 0.37 0.005091 25 204 read
[1,8]<stderr>: 0.30 0.004053 127 32 writev
[1,8]<stderr>: 0.17 0.002343 0 14286 11 poll
[1,8]<stderr>: 0.14 0.001875 13 150 fstat
[1,9]<stderr>:Process 7006 detached
[1,9]<stderr>:% time seconds usecs/call calls errors syscall
[1,9]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,9]<stderr>: 95.44 1.234502 1926 641 1 munmap
[1,9]<stderr>: 2.91 0.037661 62 603 write
[1,9]<stderr>: 0.86 0.011158 25 445 268 open
[1,9]<stderr>: 0.31 0.003997 235 17 writev
[1,9]<stderr>: 0.20 0.002623 0 8255 11 poll
[1,9]<stderr>: 0.17 0.002210 11 204 read
[1,9]<stderr>: 0.04 0.000466 155 3 futex
[1,10]<stderr>:Process 7008 detached
[1,10]<stderr>:% time seconds usecs/call calls errors syscall
[1,10]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,10]<stderr>: 96.04 1.261437 1947 648 1 munmap
[1,10]<stderr>: 2.44 0.032052 43 751 write
```

```
[1,10]<stderr>: 0.88 0.011548 26 445 268 open
[1,10]<stderr>: 0.17 0.002175 11 204 read
1,10]<stderr>: 0.16 0.002095 0 8442 10 poll
[1,10]<stderr>: 0.15 0.001998 118 17 writev
[1,10]<stderr>: 0.09 0.001149 8 150 fstat
[1,11]<stderr>:Process 7002 detached
[1,11]<stderr>:% time seconds usecs/call calls errors syscall
[1,11]<stderr>:------ ----------- ----------- --------- --------- ----------------
[1,11]<stderr>: 94.78 1.303339 1859 701 1 munmap
[1,11]<stderr>: 3.35 0.046030 64 717 write
[1,11]<stderr>: 1.05 0.014381 32 445 268 open
[1,11]<stderr>: 0.29 0.004042 20 204 read
[1,11]<stderr>: 0.22 0.002998 176 17 writev
[1,11]<stderr>: 0.16 0.002166 0 7727 9 poll
[1,11]<stderr>: 0.12 0.001673 11 150 fstat
```

6.4 Profiling Function Calls with Gprof

Using gprof we can see how much time HPL spends in each function and how many times that function was called. Here we run gprof with the path to the executable.

```
gprof xhpl
```

The call graph profile file (gmon.out by default) is read to generate a full profile as shown:

```
Flat profile:

Each sample counts as 0.01 seconds.
 %     cumulative   self              self    total
 time    seconds   seconds      calls   s/call   s/call   name
 66.99    172.31    172.31                              Loop_M4_K
 5.45     186.33     14.02                              N4_M4_LOOP
 5.36     200.11     13.78                              __intel_new_memcpy
 3.77     209.81      9.70                              Loop_M4_K
 3.04     217.64      7.83                              N4_K_LOOP_COPYB
 1.64     221.87      4.23        316    0.01     0.01   HPL_dlaswp06N
 1.40     225.46      3.60                              N4_M4_WPARUP
 1.35     228.95      3.49  201118720    0.00     0.00   HPL_rand
 1.14     231.88      2.93  405985272    0.00     0.00   HPL_lmul
 1.09     234.69      2.81                              mkl_blas_mc_dgemm_copybn
 0.85     236.87      2.18        422    0.01     0.01   HPL_dlaswp00N
 0.82     238.97      2.10                              M4_LOOP
 0.65     240.64      1.67        216    0.01     0.01   HPL_dlaswp01N
 0.61     242.20      1.56         10    0.16     0.16   HPL_pdlange
 0.57     243.67      1.47                              AY16_Loop_M16
...
granularity: each sample hit covers 2 byte(s) for 0.00% of 257.22 seconds

index % time     self   children   called     name
                                                  <spontaneous>
[1]     67.0   172.31      0.00               Loop_M4_K [1]
-----------------------------------------------
                 0.00     20.16      2/2          main [3]
[2]      7.8     0.00     20.16      2        HPL_pdtest [2]
                 1.20      8.48      4/4          HPL_pdmatgen [7]
                 0.00      8.93      2/2          HPL_pdgesv [8]
                 1.56      0.00     10/10         HPL_pdlange [22]
                 0.00      0.00      6/6          HPL_indxg2p [117]
                 0.00      0.00      4/8          HPL_numroc [113]
                 0.00      0.00      4/4          HPL_ptimer [121]
                 0.00      0.00      2/21         HPL_grid_info [110]
                 0.00      0.00      2/6          HPL_all_reduce [116]
                 0.00      0.00      2/2          HPL_ptimer_boot [125]
                 0.00      0.00      2/2          HPL_barrier [123]
                 0.00      0.00      2/2          HPL_ptimer_combine [126]
                 0.00      0.00      2/25         PL_broadcast [108]
                 0.00      0.00      2/10890      HPL_dgemv [91]
                 0.00      0.00      2/22         HPL_reduce [109]
-----------------------------------------------
                                                 <spontaneous>
[3]       7.8    0.00      20.16              main [3]
                 0.00      20.16     2/2          HPL_pdtest [2]
                 0.00       0.00     1/1          HPL_pdinfo [133]
                 0.00       0.00     1/1          HPL_grid_init [132]
                 0.00       0.00     1/21         HPL_grid_info [110]
                 0.00       0.00     1/1          HPL_grid_exit [131]
...
```

From the flat profile information we see that the majority of time is spent in the function Loop_M4_K. Note that this routine has no source line number or callgraph information given for it. That is because it is from the Intel MKL library that was linked into HPL, which is used by the HPL dgemm routine. We can see that by issuing the following grep command within the MKL library directory:

```
[arnoldg@honest1 em64t]$ nm -o lib* | grep Loop_M4_K
nm: libmkl.so: File format not recognized
nm: libmkl_cdft.a: File format not recognized
libmkl_core.a:_mc_dgemm_kernel_1_b0.o:0000000000000b30 t Loop_M4_K
libmkl_core.a:_mc_dgemm_kernel_1.o:0000000000000b30 t Loop_M4_K
libmkl_core.a:_mc_dgemm_kernel_0_b0.o:0000000000000440 t Loop_M4_K
libmkl_core.a:_mc_dgemm_kernel_0.o:0000000000000440 t Loop_M4_K
libmkl_core.a:_mc_sgemm_kernel_1_b0.o:0000000000000ce0 t Loop_M4_K
libmkl_core.a:_mc_sgemm_kernel_1.o:0000000000000ce0 t Loop_M4_K
libmkl_core.a:_mc_sgemm_kernel_0_b0.o:0000000000000420 t Loop_M4_K
libmkl_core.a:_mc_sgemm_kernel_0.o:0000000000000420 t Loop_M4_K
```

A well-tuned application often uses highly-tuned library routines for much of its computation. If you do not see your own source code line numbers, suspect one of the libraries used by the application.

6.5 Profiling Hardware Events with PerfSuite

6.5.1 Generating Performance Data with Psrun

Before using PerfSuite, we add the perfsuite module to our environment:

```
$ module load perfsuite
```

Now we gather hardware performance information on HPL using PerfSuite's psrun command.

First we wrap the program's main routine with the counters and libraries that it needs using the following modified job script:

```
time mpirun --tag-output -np ${NP} -machinefile ${PBS_NODEFILE} \ psrun
~/hpl-2.0/bin/Linux_PII_CBLAS/xhpl
```

In the script, time gives the total elapsed user and system time on the mpirun command and --tag-output labels the MPI process rank output. The main command, psrun, is followed by the argument giving the program on which PerfSuite is to be run--in this case, xhpl.

Once the job has run, PerfSuite generates a set of .xml files in the runtime directory using the following naming convention for each individual report:

```
[program] [name.process] [id.hostname.xml]
```

```
[arnoldg@honest2 Linux_PII_CBLAS]$ ls
HPL.dat              xhpl           xhpl.e2333078  xhpl.e2333116  xhpl.o2333078
xhpl.o2333116
openmpi-intel.pbs    xhpl.e2332115  xhpl.e2333090  xhpl.o2332115  xhpl.o2333090  xhpl.tau
top.8907.honest1.xml  xhpl.e2332194  xhpl.e2333093  xhpl.o2332194  xhpl.o2333093
top.8910.honest1.xml  xhpl.e2332921  xhpl.e2333111  xhpl.o2332921  xhpl.o2333111
```

Next, we will show how to use a PerfSuite psprocess command line utility to analyze this .xml output.

6.5.2 Analyzing Measurement Data with Psprocess

We use PerfSuite's psprocess command line utility to analyze the .xml files. Here, the -c flag is used with psprocess to combine all xml files into a single new file named combined.xml, which is in turn redirected into a text file with the same name:

```
psprocess -c *xml > combined.xml
psprocess combined.xml > combined.txt
```

The combined.txt file contains summary information derived from analyzing all of the .xml files in the directory.

```
PerfSuite Hardware Performance Summary Report
Version : 1.0
Created : Thu Dec 17 08:52:20 CST 2009
Generator : psprocess 0.5
XML Source : combined.xml
Execution Information
=============================================================================================
Date : Thu Dec 17 08:48:18 CST 2009
Hosts : abe1144 abe1176
Users : arnoldg
Minimum and Maximum Min Max
=============================================================================================
% CPU utilization................................ 95.73 [abe1176] 96.68 [abe1144]
% cycles stalled on any resource................. 29.74 [abe1176] 31.89 [abe1176]
Bandwidth used to level 1 cache (MB/s)........... 5650.32 [abe1144] 6018.82 [abe1144]
Bandwidth used to level 2 cache (MB/s)........... 314.47 [abe1144] 333.17 [abe1144]
CPU time (seconds)............................... 341.59 [abe1176] 344.92 [abe1144]
Floating point operations per cycle.............. 1.16 [abe1144] 1.20 [abe1176]
Floating point operations per graduated instruction 0.49 [abe1144] 0.52 [abe1176]
Graduated instructions per cycle................. 2.30 [abe1176] 2.38 [abe1144]
Graduated instructions per issued instruction..... 0.99 [abe1176] 1.01 [abe1176]
Graduated instructions per level 1 instruction cache miss
 163316.40 [abe1144] 556404.31 [abe1144]
Issued instructions per cycle.................... 2.30 [abe1176] 2.37 [abe1144]
Issued instructions per level 1 instruction cache miss
 162484.07 [abe1144] 561577.21 [abe1144]
Level 1 cache miss ratio (data).................. 0.06 [abe1144] 0.06 [abe1176]
Level 1 cache miss ratio (instruction)........... 0.00 [abe1144] 0.00 [abe1144]
Level 1 data cache accesses per graduated instruction
 0.27 [abe1144] 0.29 [abe1144]
Level 1 instruction cache misses per issued instruction
 0.00 [abe1144] 0.00 [abe1144]
Level 2 cache miss ratio (instruction)........... 0.15 [abe1144] 0.38 [abe1144]
MFLOPS (cycles).................................. 2695.52 [abe1144] 2782.74 [abe1176]
MFLOPS (wall clock).............................. 2587.94 [abe1144] 2683.43 [abe1176]
MIPS (cycles).................................... 5356.21 [abe1176] 5537.33 [abe1144]
MIPS (wall clock)................................ 5127.54 [abe1176] 5344.29 [abe1144]
MVOPS (cycles)................................... 10.94 [abe1144] 13.53 [abe1176]
MVOPS (wall clock)............................... 10.56 [abe1144] 12.97 [abe1176]
Vector instructions per cycle.................... 0.00 [abe1144] 0.01 [abe1176]
Vector instructions per graduated instruction..... 0.00 [abe1144] 0.00 [abe1176]
Wall clock time (seconds)........................ 356.78 [abe1144] 356.85 [abe1144]
Aggregate Statistics Median Mean StdDev Sum
=============================================================================================
% CPU utilization.............................. 96.42 96.35 0.27 1541.60
% cycles stalled on any resource............... 30.90 30.94 0.58 495.05
Bandwidth used to level 1 cache (MB/s)......... 5789.68 5814.30 108.24 93028.83
Bandwidth used to level 2 cache (MB/s)......... 325.41 326.01 4.98 5216.19
```

```
CPU time (seconds)............................ 344.05 343.78 0.97 5500.51
Floating point operations per cycle............ 1.18 1.18 0.01 18.87
Floating point operations per graduated instruction 0.51 0.51 0.01 8.09
Graduated instructions per cycle............... 2.32 2.33 0.02 37.30
Graduated instructions per issued instruction... 1.00 1.00 0.01 15.99
Graduated instructions per level 1 instruction cache miss
 308393.70 306519.78 97314.55 4904316.56
Issued instructions per cycle.................. 2.33 2.33 0.02 37.33
Issued instructions per level 1 instruction cache miss
 308387.96 307029.89 98382.62 4912478.20
Level 1 cache miss ratio (data)................ 0.06 0.06 0.00 0.96

Level 1 cache miss ratio (instruction)......... 0.00 0.00 0.00 0.00
Level 1 data cache accesses per graduated instruction 0.28 0.28 0.00 4.44
Level 1 instruction cache misses per issued instruction
 0.00 0.00 0.00 0.00
Level 2 cache miss ratio (instruction)......... 0.26 0.26 0.06 4.15
MFLOPS (cycles)................................ 2753.03 2744.48 26.89 43911.74
MFLOPS (wall clock)............................ 2645.68 2644.29 25.64 42308.68
MIPS (cycles).................................. 5404.45 5425.92 52.97 86814.69
MIPS (wall clock).............................. 5219.25 5227.93 59.71 83646.85
MVOPS (cycles)................................. 11.70 11.78 0.63 188.47
MVOPS (wall clock)............................. 11.27 11.35 0.60 181.58
Vector instructions per cycle.................. 0.01 0.01 0.00 0.08
Vector instructions per graduated instruction... 0.00 0.00 0.00 0.03
Wall clock time (seconds)...................... 356.82 356.81 0.02 5708.89
```

First, you'll see summary information about the profile itself, including the version, the time and date of the XML file creation, and the process that generated it (psprocess). The execution information shows the time the application (HPL) was run, the machines on which it was run, and the user who ran the application.

In the summary file above, the minimum and maximum values for many metrics are very close. This is characteristic of a well-tuned application. The next section shows minimum and maximum values for various metrics by which the processes are measured as a whole. Big variations in min and max can help identify load imbalance. If they are close, then you know the processes are balanced or performing consistently.

Next, the aggregate statistics section gives you an overall picture of the run. Statistics for a given metric are averaged across all ranks. Here you can see that the standard deviations for each metric are relatively small, which is also characteristic of a well-tuned application: it indicates that all the processes are doing about the same amount of computation as measured by floating point operations, cache accesses, etc. Aggregate statistics are especially useful for comparing to the statistics of just one process, as we'll show in the next example.

One thing to keep in mind as you consider performance is that the counters provided by vendors only measure megaflops used by floating-point hardware. Many commands and applications, especially those on a Windows platform, don't use floating-point hardware rather they work with integers and characters. So if, for example, you're running PerfSuite or another profiling tool in a virtual machine on a Windows OS, you are unlikely to get meaningful MFLOPS results when profiling applications that do not use floating point hardware.

6.5.3 Analyzing Individual MPI Ranks with Psprocess

Now we use psprocess to analyze one of the single XML files generated by psrun, xhpl.19966.abe1176.xml, containing PerfSuite output for a single MPI rank of the job.

```
psprocess xhpl.19966.abe1176.xml > xhpl.19966.abe1176.txt
```

The report for an individual rank, shown below, contains detailed information and hardware counter values for the process. It contains both the actual raw counters and derived statistics for one process. You may want to compare a few of the derived metrics [MFLOPS] with those shown in the averages of the combined file output shown in the previous section.

```
PerfSuite Hardware Performance Summary Report

Version : 1.0
Created : Thu Dec 17 08:52:53 CST 2009
Generator : psprocess 0.5
```

```
XML Source : xhpl.19966.abe1176.xml

Execution Information
==========================================================================================
Collector : libpshwpc
Date : Thu Dec 17 08:48:18 2009
Host : abe1176
Process ID : 19966
Thread : 0
User : arnoldg
Command : xhpl
Processor and System Information
==========================================================================================
Node CPUs : 8
Vendor : Intel
Family : Pentium Pro (P6)
Brand : Intel(R) Xeon(R) CPU E5345 @ 2.33GHz
CPU Revision : 7
Clock (MHz) : 2327.505
Memory (MB) : 7982.85
Pagesize (KB) : 4
Cache Information
==========================================================================================
Cache levels : 2
--------------------------------
Level 1
Type : data
Size (KB) : 32
Linesize (B) : 64
Assoc : 8
Type : instruction
Size (KB) : 32

Linesize (B) : 64
Assoc : 8
--------------------------------
Level 2
Type : unified
Size (KB) : 4096
Linesize (B) : 64
Assoc : 16
Index Description Counter Value
==========================================================================================
 1 Conditional branch instructions................................ 30913811786
 2 Branch instructions............................................ 44730210133
 3 Conditional branch instructions mispredicted.................... 15920572
 4 Conditional branch instructions taken........................... 34624885890
 5 Requests for exclusive access to clean cache line............... 40375337
 6 Requests for cache line intervention............................ 22045229
 7 Requests for exclusive access to shared cache line.............. 176440359
 8 Floating point divide instructions.............................. 1417437
 9 Floating point multiply instructions............................ 482325200218
 10 Floating point operations....................................... 957511373912
 11 Hardware interrupts............................................. 356633
 12 Level 1 data cache accesses..................................... 514731756976
 13 Level 1 data cache misses....................................... 31303836287
 14 Level 1 instruction cache accesses.............................. 797614237497
 15 Level 1 instruction cache misses................................ 5643111
 16 Level 1 load misses............................................. 69120087024
 17 Level 1 cache misses............................................ 32079438588
 18 Level 2 data cache reads........................................ 69567614807
 19 Level 2 data cache writes....................................... 254272087
 20 Level 2 instruction cache accesses.............................. 16917984
 21 Level 2 instruction cache misses................................ 3721147
 22 Level 2 store misses............................................ 1433748938
 23 Level 2 total cache accesses.................................... 69056103234
 24 Level 2 cache misses............................................ 1771764785
 25 Level 2 total cache writes...................................... 259232858
 26 Cycles stalled on any resource.................................. 255419419768
 27 Data translation lookaside buffer misses........................ 568126148
 28 Instruction translation lookaside buffer misses................. 318698
 29 Total cycles.................................................... 800868533488
 30 Instructions issued............................................. 1851080849418
 31 Instructions completed.......................................... 1849771493545
 32 Vector/SIMD instructions (could include integer)................ 4012123186
Event Index
============================================================================================
 1: PAPI_BR_CN 2: PAPI_BR_INS 3: PAPI_BR_MSP 4: PAPI_BR_TKN
 5: PAPI_CA_CLN 6: PAPI_CA_ITV 7: PAPI_CA_SHR 8: PAPI_FDV_INS
 9: PAPI_FML_INS 10: PAPI_FP_OPS 11: PAPI_HW_INT 12: PAPI_L1_DCA
 13: PAPI_L1_DCM 14: PAPI_L1_ICA 15: PAPI_L1_ICM 16: PAPI_L1_LDM
 17: PAPI_L1_TCM 18: PAPI_L2_DCR 19: PAPI_L2_DCW 20: PAPI_L2_ICA
 21: PAPI_L2_ICM 22: PAPI_L2_STM 23: PAPI_L2_TCA 24: PAPI_L2_TCM
 25: PAPI_L2_TCW 26: PAPI_RES_STL 27: PAPI_TLB_DM 28: PAPI_TLB_IM
 29: PAPI_TOT_CYC 30: PAPI_TOT_IIS 31: PAPI_TOT_INS 32: PAPI_VEC_INS

Statistics
============================================================================================
Counting domain..................................................... user
Multiplexed......................................................... yes
Vector instructions per cycle....................................... 0.005
Floating point operations per graduated instruction................. 0.518
Vector instructions per graduated instruction....................... 0.002
Graduated instructions per cycle.................................... 2.310
Issued instructions per cycle....................................... 2.311
Graduated instructions per issued instruction....................... 0.999
Issued instructions per level 1 instruction cache miss.............. 328024.887
Graduated instructions per level 1 instruction cache miss........... 327792.860
Level 1 data cache accesses per graduated instruction............... 0.278
% cycles stalled on any resource.................................... 31.893
Level 1 instruction cache misses per issued instruction............. 0.000
Level 1 cache miss ratio (data)..................................... 0.061
Level 1 cache miss ratio (instruction).............................. 0.000
Level 2 cache miss ratio (instruction).............................. 0.220
Bandwidth used to level 1 cache (MB/s).............................. 5966.726
Bandwidth used to level 2 cache (MB/s).............................. 329.546
MFLOPS (cycles)..................................................... 2782.745
MFLOPS (wall clock)................................................. 2683.428
MVOPS (cycles)...................................................... 11.660
MVOPS (wall clock).................................................. 11.244
MIPS (cycles)....................................................... 5375.854
MIPS (wall clock)................................................... 5183.988
CPU time (seconds).................................................. 344.089
Wall clock time (seconds)........................................... 356.824
% CPU utilization................................................... 96.431

[arnoldg@lgheron PerfSuite]$ cat xhpl.*.txt
PerfSuite Hardware Performance Summary Report

Version : 1.0
Created : Thu Dec 17 08:52:53 CST 2009
Generator : psprocess 0.5
XML Source : xhpl.19966.abe1176.xml
Execution Information
```

```
============================================================================================
Collector : libpshwpc
Date : Thu Dec 17 08:48:18 2009
Host : abe1176
Process ID : 19966
Thread : 0
User : arnoldg
Command : xhpl

Processor and System Information
============================================================================================
Node CPUs : 8
Vendor : Intel
Family : Pentium Pro (P6)
Brand : Intel(R) Xeon(R) CPU E5345 @ 2.33GHz
CPU Revision : 7
Clock (MHz) : 2327.505
Memory (MB) : 7982.85
Pagesize (KB) : 4

Cache Information
============================================================================================
Cache levels : 2
--------------------------------
Level 1
Type : data
Size (KB) : 32
Linesize (B) : 64
Assoc : 8
Type : instruction
Size (KB) : 32
Linesize (B) : 64
Assoc : 8
--------------------------------
Level 2
Type : unified
Size (KB) : 409
Linesize (B) : 64
Assoc : 16
Index Description Counter Value
============================================================================================
 1 Conditional branch instructions................................. 30913811786
 2 Branch instructions............................................. 44730210133
 3 Conditional branch instructions mispredicted.................... 15920572
 4 Conditional branch instructions taken........................... 34624885890
 5 Requests for exclusive access to clean cache line............... 40375337
 6 Requests for cache line intervention............................ 22045229
 7 Requests for exclusive access to shared cache line.............. 176440359
 8 Floating point divide instructions.............................. 1417437
 9 Floating point multiply instructions............................ 482325200218
 10 Floating point operations...................................... 957511373912
 11 Hardware interrupts............................................ 356633
 12 Level 1 data cache accesses.................................... 514731756976
 13 Level 1 data cache misses...................................... 31303836287
 14 Level 1 instruction cache accesses............................. 797614237497
 15 Level 1 instruction cache misses............................... 5643111
 16 Level 1 load misses............................................ 69120087024
 17 Level 1 cache misses........................................... 32079438588
 18 Level 2 data cache reads....................................... 69567614807
 19 Level 2 data cache writes...................................... 254272087
 20 Level 2 instruction cache accesses............................. 16917984
 21 Level 2 instruction cache misses............................... 3721147
 22 Level 2 store misses........................................... 1433748938
 23 Level 2 total cache accesses................................... 69056103234
 24 Level 2 cache misses........................................... 1771764785
 25 Level 2 total cache writes..................................... 259232858
 26 Cycles stalled on any resource................................. 255419419768
 27 Data translation lookaside buffer misses....................... 568126148
 28 Instruction translation lookaside buffer misses................ 318698
 29 Total cycles................................................... 800868533488
 30 Instructions issued............................................ 1851080849418
 31 Instructions completed......................................... 1849771493545
 32 Vector/SIMD instructions (could include integer)............... 4012123186

Event Index
============================================================================================
 1: PAPI_BR_CN 2: PAPI_BR_INS 3: PAPI_BR_MSP 4: PAPI_BR_TKN
 5: PAPI_CA_CLN 6: PAPI_CA_ITV 7: PAPI_CA_SHR 8: PAPI_FDV_INS
 9: PAPI_FML_INS 10: PAPI_FP_OPS 11: PAPI_HW_INT 12: PAPI_L1_DCA
 13: PAPI_L1_DCM 14: PAPI_L1_ICA 15: PAPI_L1_ICM 16: PAPI_L1_LDM
 17: PAPI_L1_TCM 18: PAPI_L2_DCR 19: PAPI_L2_DCW 20: PAPI_L2_ICA
 21: PAPI_L2_ICM 22: PAPI_L2_STM 23: PAPI_L2_TCA 24: PAPI_L2_TCM
 25: PAPI_L2_TCW 26: PAPI_RES_STL 27: PAPI_TLB_DM 28: PAPI_TLB_IM
 29: PAPI_TOT_CYC 30: PAPI_TOT_IIS 31: PAPI_TOT_INS 32: PAPI_VEC_INS

Statistics
============================================================================================
Counting domain...................................................... user
Multiplexed.......................................................... yes
Floating point operations per cycle.................................. 1.196
Vector instructions per cycle........................................ 0.005
Floating point operations per graduated instruction.................. 0.518
Vector instructions per graduated instruction........................ 0.002
Graduated instructions per cycle..................................... 2.310
Issued instructions per cycle........................................ 2.311
Graduated instructions per issued instruction........................ 0.999
Issued instructions per level 1 instruction cache miss............... 328024.887
Graduated instructions per level 1 instruction cache miss............ 327792.860
Level 1 data cache accesses per graduated instruction................ 0.278
% cycles stalled on any resource..................................... 31.893
Level 1 instruction cache misses per issued instruction.............. 0.000
Level 1 cache miss ratio (data)...................................... 0.061
Level 1 cache miss ratio (instruction)............................... 0.000
Level 2 cache miss ratio (instruction)............................... 0.220
Bandwidth used to level 1 cache (MB/s)............................... 5966.726
Bandwidth used to level 2 cache (MB/s)............................... 329.546
MFLOPS (cycles)...................................................... 2782.745
MFLOPS (wall clock).................................................. 2683.428
MVOPS (cycles)....................................................... 11.660
MVOPS (wall clock)................................................... 11.244
MIPS (cycles)........................................................ 5375.854
MIPS (wall clock).................................................... 5183.988
CPU time (seconds)................................................... 344.089
Wall clock time (seconds)............................................ 356.824
% CPU utilization.................................................... 96.431
```

6.5.4 Examining Level 2 Cache Misses

In this example we profile HPL by level 2 cache misses using the configuration file papi_profile_l2tcm.xml.

```
setenv PS_HWPC_FILE usr/apps/tools/PerfSuite/1.0.0a3/share/PerfSuite/xml/pshw

pc/papi_profile_l2tcm.xml

time mpirun --tag-output -np ${NP} -machinefile ${PBS_NODEFILE} \
psrun ~/hpl-2.0/bin/Linux_PII_CBLAS/xhpl

PerfSuite Hardware Performance Summary Report

Version : 1.0
Created : Thu Dec 17 11:19:16 CST 2009
Generator : psprocess 0.5
XML Source : xhpl.18884.abe0909.xml

Execution Information
============================================================================================
Collector : libpshwpc
Date : Thu Dec 17 10:42:18 2009
Host : abe0909
Process ID : 18884
Thread : 0
User : arnoldg
Command : xhpl

Processor and System Information
============================================================================================
Node CPUs : 8
Vendor : Intel
Family : Pentium Pro (P6)
Brand : Intel(R) Xeon(R) CPU E5345 @ 2.33GHz
CPU Revision : 7
Clock (MHz) : 2327.505
Memory (MB) : 7982.85
Pagesize (KB) : 4
Cache Information
============================================================================================
Cache levels : 2
--------------------------------
Level 1
Type : data
Size (KB) : 32
Linesize (B) : 64

Assoc : 8

Type : instruction
Size (KB) : 32
Linesize (B) : 64
Assoc : 8
--------------------------------
Level 2
Type : unified
Size (KB) : 4096
Linesize (B) : 64
Assoc : 16

Profile Information
============================================================================================
Class : PAPI
Version : 3.7.2
Event : PAPI_L2_TCM (Level 2 cache misses)
Period : 100000
Samples : 17118
Domain : user
Run Time : 362.61 (seconds)
Min Self % : (all)
Module Summary
--------------------------------------------------------------------------------
```

```
Samples  Self %    Total %   Module
 16917    98.83%    98.83%   /u/ncsa/arnoldg/hpl-2.0/bin/Linux_PII_CBLAS/xhpl
 196       1.14%    99.97%   /usr/local/openmpi-1.3.2-intel/lib/libmpi.so.0.0.0
 3         0.02%    99.99%   /usr/local/openmpi-1.3.2-intel/lib/libopen-pal.so.0.0.0
 2         0.01%   100.00%   /lib64/tls/libpthread-2.3.4.so

File Summary
--------------------------------------------------------------------------------
Samples   Self %    Total %   File
 15327    89.54%    89.54%    ??
 769       4.49%    94.03%    ../HPL_dlaswp06N.c
 265       1.55%    95.58%    ../HPL_dlaswp01N.c
 248       1.45%    97.03%    ../HPL_pdlange.c
 206       1.20%    98.23%    ../HPL_dlaswp00N.c
 149       0.87%    99.10%    ../HPL_dlacpy.c
 132       0.77%    99.87%    ../HPL_lmul.c
 8         0.05%    99.92%    ../HPL_dlaswp04N.c
 4         0.02%    99.94%    ../HPL_dlaswp02N.c
 2         0.01%    99.95%    ../HPL_pipid.c
 1         0.01%    99.96%    ../HPL_blong.c
 1         0.01%    99.96%    ../HPL_sdrv.c
 1         0.01%    99.97%    ../HPL_dcopy.c
 1         0.01%    99.98%    ../HPL_infog2l.c
 1         0.01%    99.98%    ../HPL_pdlaswp01N.c
 1         0.01%    99.99%    ../HPL_pdrpanllN.c
 1         0.01%    99.99%    ../HPL_rollN.c
 1         0.01%    100.00%   ../HPL_plindx1.c

Function Summary
--------------------------------------------------------------------------------
Samples   Self %    Total %   Function

 8742     51.07%    51.07%    Loop_M4_K
 2199     12.85%    63.92%    N4_M4_LOOP
 1439      8.41%    72.32%    __intel_new_memcpy
 1414      8.26%    80.58%    N4_K_LOOP_COPYB
 769       4.49%    85.07%    HPL_dlaswp06N
 406       2.37%    87.45%    mkl_blas_mc_dgemm_copybn
 306       1.79%    89.23%    K4_M8_LOOP_START
 265       1.55%    90.78%    HPL_dlaswp01N
 248       1.45%    92.23%    HPL_pdlange
 206       1.20%    93.43%    HPL_dlaswp00N
 188       1.10%    94.53%    AY16_Loop_M16
 149       0.87%    95.40%    HPL_dlacpy
 135       0.79%    96.19%    ompi_generic_simple_unpack
 132       0.77%    96.96%    HPL_lmul
 122       0.71%    97.67%    Unit_Triangle4x4
 65        0.38%    98.05%    N4_K_LOOP_COPYB_N
 49        0.29%    98.34%    M4_LOOP
 49        0.29%    98.63%    ompi_generic_simple_pack
 49        0.29%    98.91%    Unit_Loop_MxN4
 25        0.15%    99.06%    N4_M4_WPARUP
 24        0.14%    99.20%    m4_UnAlignedC
 23        0.13%    99.33%    END_N4_K_LOOP_COPYB
 20        0.12%    99.45%    mkl_blas_mc_dgemm_copyan
 15        0.09%    99.54%    _intel_fast_memcpy.J
 10        0.06%    99.60%    mkl_blas_mc_dtrsm_copya
 8         0.05%    99.64%    HPL_dlaswp04N
 8         0.05%    99.69%    K_LOOP_COPYB
 6         0.04%    99.73%    Steps_Any_Loop4
 5         0.03%    99.75%    ??
 4         0.02%    99.78%    HPL_dlaswp02N
 3         0.02%    99.80%    _intel_fast_memcpy
 2         0.01%    99.81%    pthread_spin_lock
 2         0.01%    99.82%    HPL_pipid
 2         0.01%    99.83%    N4_K_LOOP_COPYB_N_TAILS
 2         0.01%    99.84%    mkl_blas_mc_xdgemm_0
 1         0.01%    99.85%    opal_progress
 1         0.01%    99.85%    N4_LOOP
 1         0.01%    99.86%    HPL_infog2l
 1         0.01%    99.87%    Unit_Next_M4
 1         0.01%    99.87%    ompi_convertor_unpack
 1         0.01%    99.88%    ompi_convertor_pack
 1         0.01%    99.88%    MPI_Type_vector
 1         0.01%    99.89%    mkl_blas_mc_dtrsm_lln_r
 1         0.01%    99.89%    PMPI_Type_free
 1         0.01%    99.90%    HPL_sdrv
 1         0.01%    99.91%    HPL_plindx1
 1         0.01%     9.91%    HPL_bcast_blong
 1         0.01%    99.92%    N1_Loop_M8
 1         0.01%    99.92%    N2_Loop_M8
 1         0.01%    99.93%    N_TAILS
 1         0.01%    99.94%    opal_memory_ptmalloc2_int_malloc
 1         0.01%    99.94%    HPL_rollN
 1         0.01%    99.95%    __get_free_dt_struct
 1         0.01%    99.95%    mca_mpool_base_mem_cb
 1         0.01%    99.96%    HPL_pdlaswp01N
 1         0.01%    99.96%    mkl_blas_mc_xdtrsm
 1         0.01%    99.97%    opal_event_loop
 1         0.01%    99.98%    HPL_dcopy
 1         0.01%    99.98%    MPI_Ssend
 1         0.01%    99.99%    mkl_blas_mc_dgemm_kernel_1
 1         0.01%    99.99%    HPL_pdrpanllN
 1         0.01%   100.00%    END_N3_M4_LOOP

Function:File:Line Summary
--------------------------------------------------------------------------------
Samples   Self %   Total %   Function:File:Line

 8742      51.07%   51.07%   Loop_M4_K:??:0
 2199      12.85%   63.92%   N4_M4_LOOP:??:0
 1439       8.41%   72.32%   __intel_new_memcpy:??:0
 1414       8.26%   80.58%   N4_K_LOOP_COPYB:??:0
 406        2.37%   82.95%   mkl_blas_mc_dgemm_copybn:??:0
 306        1.79%   84.74%   K4_M8_LOOP_START:??:0
 188        1.10%   85.84%   AY16_Loop_M16:??:0
 135        0.79%   86.63%   ompi_generic_simple_unpack:??:0
 124        0.72%   87.35%   HPL_pdlange:../HPL_pdlange.c:173
 122        0.71%   88.07%   Unit_Triangle4x4:??:0
 108        0.63%   88.70%   HPL_pdlange:../HPL_pdlange.c:211
 76         0.44%   89.14%   HPL_dlacpy:../HPL_dlacpy.c:195
 66         0.39%   89.53%   HPL_lmul:../HPL_lmul.c:113
 65         0.38%   89.91%   N4_K_LOOP_COPYB_N:??:0
 49         0.29%   90.19%   Unit_Loop_MxN4:??:0
 49         0.29%   90.48%   ompi_generic_simple_pack:??:0
 49         0.29%   90.76%   M4_LOOP:??:0
 40         0.23%   91.00%   HPL_lmul:../HPL_lmul.c:118
 36         0.21%   91.21%   HPL_dlaswp06N:../HPL_dlaswp06N.c:171
 33         0.19%   91.40%   HPL_dlaswp06N:../HPL_dlaswp06N.c:164
 32         0.19%   91.59%   HPL_dlaswp06N:../HPL_dlaswp06N.c:169
 32         0.19%   91.77%   HPL_dlaswp06N:../HPL_dlaswp06N.c:186
 29         0.17%   91.94%   HPL_dlaswp06N:../HPL_dlaswp06N.c:160
 29         0.17%   92.11%   HPL_dlaswp06N:../HPL_dlaswp06N.c:179
 28         0.16%   92.28%   HPL_dlacpy:../HPL_dlacpy.c:174
```

```
27          0.16%   92.43%   HPL_dlaswp06N:../HPL_dlaswp06N.c:151
27          0.16%   92.59%   HPL_dlaswp06N:../HPL_dlaswp06N.c:161
27          0.16%   92.75%   HPL_dlaswp06N:../HPL_dlaswp06N.c:181
27          0.16%   92.91%   HPL_dlaswp06N:../HPL_dlaswp06N.c:188
26          0.15%   93.06%   HPL_dlaswp06N:../HPL_dlaswp06N.c:185
25          0.15%   93.21%   N4_M4_WPARUP:??:0
25          0.15%   93.35%   HPL_dlaswp06N:../HPL_dlaswp06N.c:170
25          0.15%   93.50%   HPL_dlaswp06N:../HPL_dlaswp06N.c:182
24          0.14%   93.64%   HPL_dlaswp06N:../HPL_dlaswp06N.c:155
24          0.14%   93.78%   HPL_dlaswp06N:../HPL_dlaswp06N.c:167
24          0.14%   93.92%   HPL_dlaswp06N:../HPL_dlaswp06N.c:175
24          0.14%   94.06%   HPL_dlaswp06N:../HPL_dlaswp06N.c:187
24          0.14%   94.20%   m4_UnAlignedC:??:0
24          0.14%   94.34%   HPL_dlaswp01N:../HPL_dlaswp01N.c:175
23          0.13%   94.47%   HPL_dlaswp06N:../HPL_dlaswp06N.c:158
23          0.13%   94.61%   HPL_dlaswp06N:../HPL_dlaswp06N.c:174
23          0.13%   94.74%   HPL_dlaswp06N:../HPL_dlaswp06N.c:183
23          0.13%   94.88%   END_N4_K_LOOP_COPYB:??:0
23          0.13%   95.01%   HPL_dlacpy:../HPL_dlacpy.c:279
22          0.13%   95.14%   HPL_dlaswp06N:../HPL_dlaswp06N.c:166
22          0.13%   95.27%   HPL_dlaswp01N:../HPL_dlaswp01N.c:171
21          0.12%   95.39%   HPL_dlaswp06N:../HPL_dlaswp06N.c:180
21          0.12%   95.51%   HPL_dlaswp01N:../HPL_dlaswp01N.c:166
20          0.12%   95.63%   mkl_blas_mc_dgemm_copyan:??:0
20          0.12%   95.75%   HPL_dlaswp01N:../HPL_dlaswp01N.c:179
20          0.12%   95.86%   HPL_dlaswp06N:../HPL_dlaswp06N.c:149
20          0.12%   95.98%   HPL_dlaswp06N:../HPL_dlaswp06N.c:159
20          0.12%   96.10%   HPL_dlaswp06N:../HPL_dlaswp06N.c:176
20          0.12%   96.21%   HPL_dlaswp06N:../HPL_dlaswp06N.c:184
19          0.11%   96.33%   HPL_dlaswp06N:../HPL_dlaswp06N.c:165
19          0.11%   96.44%   HPL_dlaswp06N:../HPL_dlaswp06N.c:168
18          0.11%   96.54%   HPL_dlaswp01N:../HPL_dlaswp01N.c:185
18          0.11%   96.65%   HPL_dlaswp01N:../HPL_dlaswp01N.c:187
18          0.11%   96.75%   HPL_dlaswp06N:../HPL_dlaswp06N.c:178
17          0.10%   96.85%   HPL_dlaswp01N:../HPL_dlaswp01N.c:180
17          0.10%   96.95%   HPL_dlaswp06N:../HPL_dlaswp06N.c:177
17          0.10%   97.05%   HPL_dlaswp06N:../HPL_dlaswp06N.c:189
17          0.10%   97.15%   HPL_dlaswp01N:../HPL_dlaswp01N.c:174
16          0.09%   97.24%   HPL_dlaswp01N:../HPL_dlaswp01N.c:178
16          0.09%   97.34%   HPL_dlaswp01N:../HPL_dlaswp01N.c:188
16          0.09%   97.43%   HPL_dlaswp06N:../HPL_dlaswp06N.c:154
16          0.09%   97.52%   HPL_pdlange:../HPL_pdlange.c:210
15          0.09%   97.61%   _intel_fast_memcpy.J:??:0
14          0.08%   97.69%   HPL_dlaswp01N:../HPL_dlaswp01N.c:184
14          0.08%   97.77%   HPL_dlacpy:../HPL_dlacpy.c:184
13          0.08%   97.85%   HPL_dlaswp01N:../HPL_dlaswp01N.c:181
12          0.07%   97.92%   HPL_dlaswp01N:../HPL_dlaswp01N.c:189
12          0.07%   97.99%   HPL_dlaswp01N:../HPL_dlaswp01N.c:191
12          0.07%   98.06%   HPL_dlaswp00N:../HPL_dlaswp00N.c:148
11          0.06%   98.12%   HPL_dlaswp00N:../HPL_dlaswp00N.c:152
11          0.06%   98.19%   HPL_dlaswp00N:../HPL_dlaswp00N.c:159
10          0.06%   98.25%   HPL_dlaswp01N:../HPL_dlaswp01N.c:186
10          0.06%   98.31%   HPL_dlaswp00N:../HPL_dlaswp00N.c:137
10          0.06%   98.36%   HPL_dlaswp01N:../HPL_dlaswp01N.c:168
10          0.06%   98.42%   mkl_blas_mc_dtrsm_copya:??:0
9           0.05%   98.48%   HPL_dlaswp00N:../HPL_dlaswp00N.c:170
9           0.05%   98.53%   HPL_dlaswp00N:../HPL_dlaswp00N.c:175
9           0.05%   98.58%   HPL_lmul:../HPL_lmul.c:110
9           0.05%   98.63%   HPL_dlaswp00N:../HPL_dlaswp00N.c:155
9           0.05%   98.69%   HPL_dlaswp00N:../HPL_dlaswp00N.c:162
8           0.05%   98.73%   K_LOOP_COPYB:??:0
8           0.05%   98.78%   HPL_lmul:../HPL_lmul.c:112
```

```
8          0.05%   98.83%   HPL_dlaswp00N:../HPL_dlaswp00N.c:139
8          0.05%   98.87%   HPL_dlaswp00N:../HPL_dlaswp00N.c:154
7          0.04%   98.91%   HPL_dlaswp00N:../HPL_dlaswp00N.c:177
7          0.04%   98.95%   HPL_lmul:../HPL_lmul.c:109
7          0.04%   99.00%   HPL_dlacpy:../HPL_dlacpy.c:169
7          0.04%   99.04%   HPL_dlaswp00N:../HPL_dlaswp00N.c:143
7          0.04%   99.08%   HPL_dlaswp00N:../HPL_dlaswp00N.c:146
7          0.04%   99.12%   HPL_dlaswp00N:../HPL_dlaswp00N.c:147
7          0.04%   99.16%   HPL_dlaswp00N:../HPL_dlaswp00N.c:157
7          0.04%   99.20%   HPL_dlaswp00N:../HPL_dlaswp00N.c:166
6          0.04%   99.23%   HPL_dlaswp00N:../HPL_dlaswp00N.c:169
6          0.04%   99.27%   HPL_dlaswp00N:../HPL_dlaswp00N.c:142
6          0.04%   99.30%   Steps_Any_Loop4:??:0
6          0.04%   99.34%   HPL_dlaswp00N:../HPL_dlaswp00N.c:163
5          0.03%   99.37%   HPL_dlaswp00N:../HPL_dlaswp00N.c:172
5          0.03%   99.40%   HPL_dlaswp00N:../HPL_dlaswp00N.c:173
5          0.03%   99.43%   ??:??:0
5          0.03%   99.46%   HPL_dlaswp00N:../HPL_dlaswp00N.c:156
5          0.03%   99.49%   HPL_dlaswp00N:../HPL_dlaswp00N.c:158
4          0.02%   99.51%   HPL_dlaswp00N:../HPL_dlaswp00N.c:174
4          0.02%   99.53%   HPL_dlaswp01N:../HPL_dlaswp01N.c:190
4          0.02%   99.56%   HPL_dlaswp00N:../HPL_dlaswp00N.c:153
4          0.02%   99.58%   HPL_dlaswp00N:../HPL_dlaswp00N.c:164
4          0.02%   99.60%   HPL_dlaswp00N:../HPL_dlaswp00N.c:165
4          0.02%   99.63%   HPL_dlaswp00N:../HPL_dlaswp00N.c:168
3          0.02%   99.64%   HPL_dlaswp00N:../HPL_dlaswp00N.c:171
3          0.02%   99.66%   _intel_fast_memcpy:??:0
3          0.02%   99.68%   HPL_dlaswp00N:../HPL_dlaswp00N.c:149
2          0.01%   99.69%   pthread_spin_lock:??:0
2          0.01%   99.70%   HPL_dlaswp00N:../HPL_dlaswp00N.c:176
2          0.01%   99.71%   HPL_lmul:../HPL_lmul.c:111
2          0.01%   99.73%   mkl_blas_mc_xdgemm_0:??:0
2          0.01%   99.74%   N4_K_LOOP_COPYB_N_TAILS:??:0
2          0.01%   99.75%   HPL_dlaswp02N:../HPL_dlaswp02N.c:177
2          0.01%   99.76%   HPL_dlaswp00N:../HPL_dlaswp00N.c:167
1          0.01%   99.77%   HPL_dlaswp02N:../HPL_dlaswp02N.c:183
1          0.01%   99.77%   Unit_Next_M4:??:0
1          0.01%   99.78%   HPL_dlaswp04N:../HPL_dlaswp04N.c:198
1          0.01%   99.78%   END_N3_M4_LOOP:??:0
1          0.01%   99.79%   MPI_Type_vector:??:0
1          0.01%   99.80%   N_TAILS:??:0
1          0.01%   99.80%   __get_free_dt_struct:??:0
1          0.01%   99.81%   opal_memory_ptmalloc2_int_malloc:??:0
1          0.01%   99.81%   N4_LOOP:??:0
1          0.01%   99.82%   HPL_dlaswp04N:../HPL_dlaswp04N.c:213
1          0.01%   99.82%   mkl_blas_mc_dgemm_kernel_1:??:0
1          0.01%   99.83%   HPL_dlaswp04N:../HPL_dlaswp04N.c:214
1          0.01%   99.84%   HPL_dlaswp04N:../HPL_dlaswp04N.c:215
1          0.01%   99.84%   HPL_dlaswp04N:../HPL_dlaswp04N.c:216
1          0.01%   99.85%   opal_progress:??:0
1          0.01%   99.85%   MPI_Ssend:??:0
1          0.01%   99.86%   N1_Loop_M8:??:0
1          0.01%   99.87%   HPL_dlaswp04N:../HPL_dlaswp04N.c:225
1          0.01%   99.87%   mkl_blas_mc_dtrsm_lln_r:??:0
1          0.01%   99.88%   HPL_infog2l:../HPL_infog2l.c:206
1          0.01%   99.88%   HPL_dcopy:../HPL_dcopy.c:72
1          0.01%   99.89%   PMPI_Type_free:??:0
1          0.01%   99.89%   HPL_pdrpanllN:../HPL_pdrpanllN.c:133
1          0.01%   99.90%   HPL_plindx1:../HPL_plindx1.c:190
1          0.01%   99.91%   opal_event_loop:??:0
1          0.01%   99.91%   ompi_convertor_unpack:??:0
1          0.01%   99.92%   HPL_pipid:../HPL_pipid.c:163
```

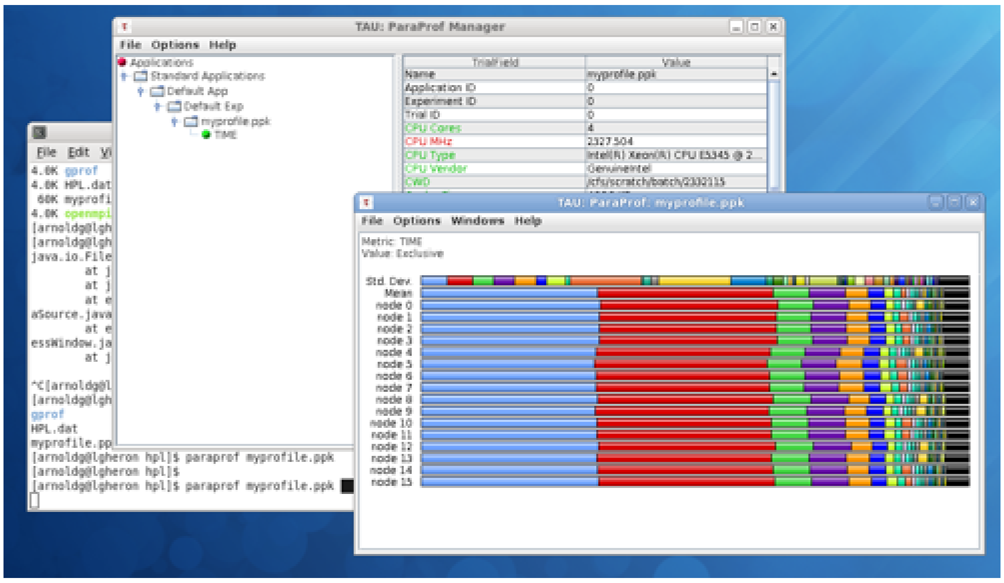
```
1           0.01%   99.92%   HPL_bcast_blong:../HPL_blong.c:266
1           0.01%   99.93%   mca_mpool_base_mem_cb:??:0
1           0.01%   99.94%   HPL_pipid:../HPL_pipid.c:167
1           0.01%   99.94%   HPL_dlaswp01N:../HPL_dlaswp01N.c:163
1           0.01%   99.95%   HPL_pdlaswp01N:../HPL_pdlaswp01N.c:199
1           0.01%   99.95%   ompi_convertor_pack:??:0
1           0.01%   99.96%   N2_Loop_M8:??:0
1           0.01%   99.96%   HPL_dlaswp04N:../HPL_dlaswp04N.c:186
1           0.01%   99.97%   HPL_dlacpy:../HPL_dlacpy.c:280
1           0.01%   99.98%   HPL_dlaswp02N:../HPL_dlaswp02N.c:178
1           0.01%   99.98%   HPL_rollN:../HPL_rollN.c:189
1           0.01%   99.99%   HPL_dlaswp04N:../HPL_dlaswp04N.c:205
1           0.01%   99.99%   mkl_blas_mc_xdtrsm:??:0
1           0.01%   100.00%  HPL_sdrv:../HPL_sdrv.c:83
```

6.6 Profiling a Parallel Application with TAU

In this example, we analyze the profiles generated from instrumenting and compiling HPL with TAU's visualization tool, paraprof:

```
paraprof myprofile.ppk
```

The default view is shown in the following image:

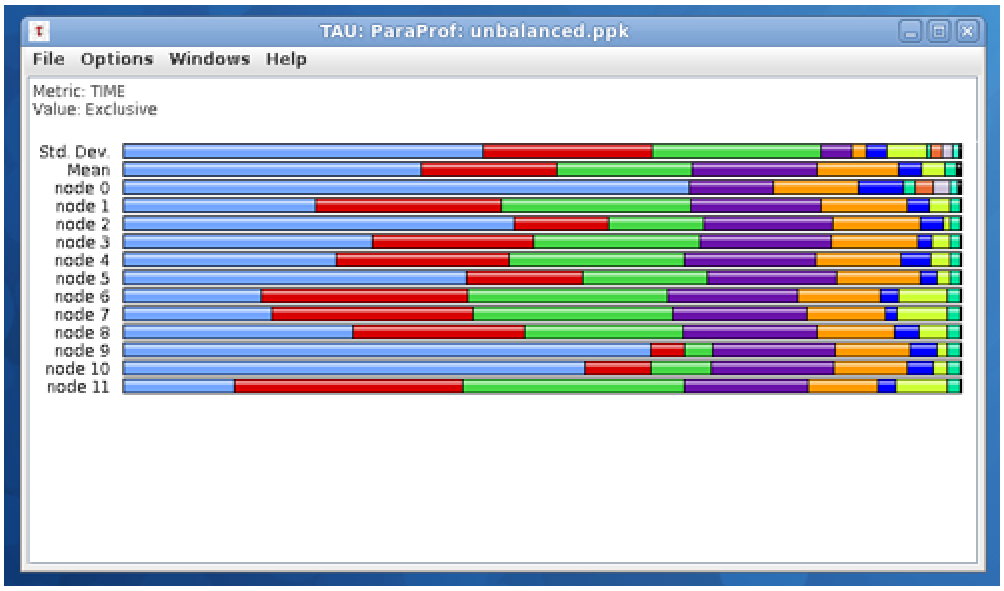


TrialField	Value
Name	myprofile.ppk
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	4
CPU MHz	2327.504
CPU Type	Intel(R) Xeon(R) CPU E5345 @ 2...
CPU Vendor	GenuineIntel
CWD	/cfs/scratch/batch/2332115

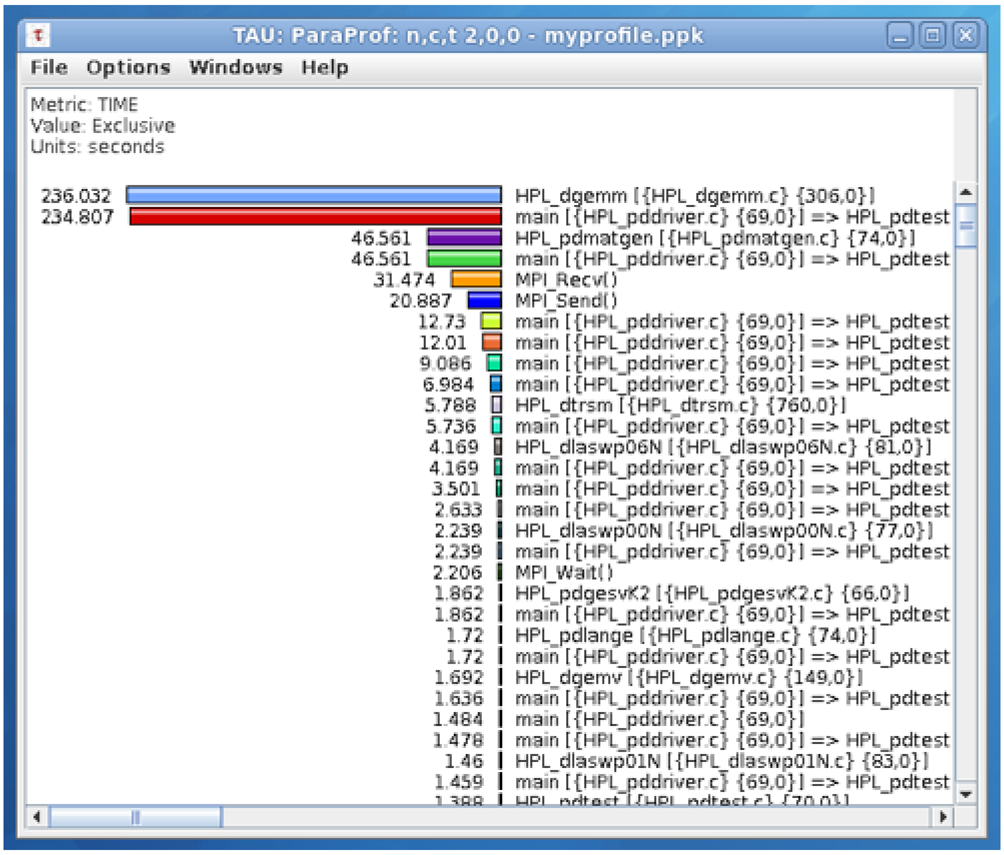
```
4.0K gprof
4.0K HPL.dat
 60K myprofi
4.0K openmpi
[arnoldg@lgh
[arnoldg@lgh
java.io.File
        at j
        at j
        at e
aSource.java
        at e
essWindow.ja
        at j

^C[arnoldg@l
[arnoldg@lgh
gprof
HPL.dat
myprofile.pp
[arnoldg@lgheron hpl]$ paraprof myprofile.ppk
[arnoldg@lgheron hpl]$
[arnoldg@lgheron hpl]$ paraprof myprofile.ppk
```

The graph in the front shows the time spent in routines broken down by color. The evenly sized bars in this view confirms the balanced work seen in the PerfSuite profile. Each node (MPI rank) is doing about the same amount of work. This is what you're looking for in a well tuned application. An application that needs improvement might show a wide variation in the size of the bars of a given color􀀀indicating an unbalanced distribution of work for that routine as shown in the following image:



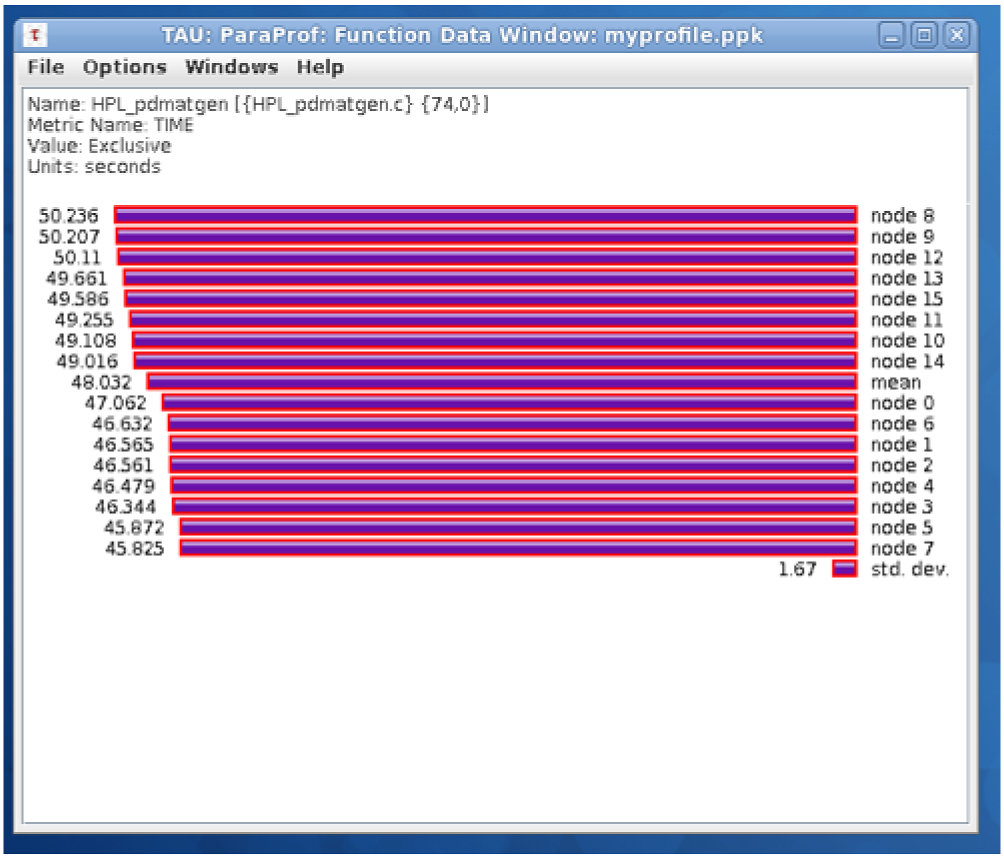
Clicking on the text 'node 2' in the HPL TAU display will show a profile for the second MPI rank of HPL:


TAU: ParaProf: n,c,t 2,0,0 - myprofile.ppk
File Options Windows Help

```
Metric: TIME
Value: Exclusive
Units: seconds

236.032  HPL_dgemm [{HPL_dgemm.c} {306,0}]
234.807  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
 46.561  HPL_pdmatgen [{HPL_pdmatgen.c} {74,0}]
 46.561  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
 31.474  MPI_Recv()
 20.887  MPI_Send()
  12.73  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  12.01  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  9.086  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  6.984  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  5.788  HPL_dtrsm [{HPL_dtrsm.c} {760,0}]
  5.736  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  4.169  HPL_dlaswp06N [{HPL_dlaswp06N.c} {81,0}]
  4.169  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  3.501  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  2.633  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  2.239  HPL_dlaswp00N [{HPL_dlaswp00N.c} {77,0}]
  2.239  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  2.206  MPI_Wait()
  1.862  HPL_pdgesvK2 [{HPL_pdgesvK2.c} {66,0}]
  1.862  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
   1.72  HPL_pdlange [{HPL_pdlange.c} {74,0}]
   1.72  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  1.692  HPL_dgemv [{HPL_dgemv.c} {149,0}]
  1.636  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  1.484  main [{HPL_pddriver.c} {69,0}]
  1.478  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
   1.46  HPL_dlaswp01N [{HPL_dlaswp01N.c} {83,0}]
  1.459  main [{HPL_pddriver.c} {69,0}] => HPL_pdtest
  1.388  HPL_pdtest [{HPL_pdtest.c} {70,0}]
```

Clicking your left mouse button on one of the colored segments will produce a chart showing the data for just that subroutine, and selecting the thread call graph option displays the following:


TAU: ParaProf: Function Data Window: myprofile.ppk
File Options Windows Help

Name: HPL_pdmatgen [{HPL_pdmatgen.c} {74,0}]
Metric Name: TIME
Value: Exclusive
Units: seconds

50.236	node 8
50.207	node 9
50.11	node 12
49.661	node 13
49.586	node 15
49.255	node 11
49.108	node 10
49.016	node 14
48.032	mean
47.062	node 0
46.632	node 6
46.565	node 1
46.561	node 2
46.479	node 4
46.344	node 3
45.872	node 5
45.825	node 7
1.67	std. dev.

7 Use Case: GenIDLEST

Through the TeraGrid's Advanced Support for TeraGrid Applications (ASTA) program, the TeraGrid's Advanced User Support (AUS) staff provides services to users of TeraGrid resources to enhance the research they perform on these resources. These services often include identification of performance problems and code optimization.

In the following presentation to ASTA group members, NCSA's Rick Kufrin describes an ASTA project that identified the cause of a severe slowdown in performance of the GenIDLEST application code running on NCSA's Altix system.

Incorporating Performance Tracking and Regression Testing Into GenIDLEST Using POINT[1] *Tools* (Audio and Slides recording)(PowerPoint slides)

[1] The Productivity from Open, INtegrated Tools (POINT) project is funded as part of the NSF's Software Development for Cyberinfrastructure (SDCI) program. The goal of the POINT project is to provide an open, portable, and robust performance tools environment for the NSF-funded high-performance computing centers. Two of the tools described in this tutorial, PerfSuite and TAU, are core components of the POINT project.

8 Exercises

8.1 Problems

For exercises 1 - 5 we use two codes: cpu1 and matmul. The cpu1 code is a simple program that merely returns some trigonometric values. We used it because it can be run on one or more processors to create artificial system load and it calls functions, which will be listed in the profile summary. The matmul code multiplies two matrices. Matmul, like HPL, is often used for benchmarking performance. You can download these codes by clicking the following links:

- cpu1.c
- matmul.c

1. Compile the cpu1 code with profiling support for use with gprof and include the option that adds debugging symbols to the code. Then run gprof with source code annotation. (Note that at optimization levels higher than -O2 (-O3 , -qhot ...) source code annotation may not produce output with some compilers such as XL.)
2. Recompile the cpu1 code with compiler optimization disabled (-O0 option), run gprof, and save the output. Then recompile using high optimization (-O5), run gprof and compare the results with your disabled compiler optimization run.
3. Profile the matmul code using strace.
4. Use PerfSuite's psrun command to count hardware events in the matmul code.
5. Using PerfSuite, profile the matmul code with respect to cycles (PAPI_TOT_CYC).

6. Using PerfSuite, compare the cache-friendly and cache-unfriendly memory access patterns of these two codes:
   - matvec-goodcache.c
   - matvec-badcache.c
7. The only difference between the two codes is the switching of the two lines "for (i = 0; ...)" and "for (j = 0; ...)" in the "/* Calculation. */" block. Build them using the "-O0" compilation flag: `gcc -O0 -o matvec-goodcache matvec-goodcache.c`
   `gcc -O0 -o matvec-goodcache matvec-badcache.c`
8. Practice using TAU's pprof and paraprof utilities on pi, a simple MPI C++ program with TAU's profiling API calls inside.

   Download the pi tar bundle
9. Use TAU to automatically instrument a simple C++ program called klargest.

   Download the auto instrumentation tar bundle

8.2 Solutions

The following exercise solutions are given as examples only. Your results will differ based on the system and compiler you used for the exercises.

Go to solution 1 2 3 4 5 6 7 8

1. First we compiled the 1cpu code with profiling support (-pg) and included the -g option to add debugging symbols to the code: `arnoldg@bd-login:~/c> xlc -g -pg -o 1cpu 1cpu.c`
   `arnoldg@bd-login:~/c>` Then we ran gprof by default to generate the following flat profile:

```
arnoldg@bd-login:~/c> gprof 1cpu
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 85.51      0.74     0.74        1   743.96   814.33  .work
  8.09      0.81     0.07 100000001     0.00     0.00  .mycos
  6.93      0.87     0.06                             .__gmon_start__

...
```
   Next, we ran gprof with source code annotation:

```
arnoldg@bd-login:~/c> gprof -A 1cpu
*** File /home/arnoldg/c/1cpu.c:
                #include <stdio.h>
                #include <math.h>

       ##### -> int main(void)
                {
                        void work(void);

                        work();
                }

           1 -> void work(void)
                {
                        double f, temp;
                        double sum;
                        double mycos(double);

                        for (f=0.0; f<10000.0; f+= 0.0001)
                        {
                                temp= sin(f);
                                sum += temp;
                                temp= mycos(f);
                                sum += temp;
                                temp= tan(f);
                                sum += temp;
                        }
                        printf("sum= %lf done\n",sum);
                }

   100000001 -> double mycos(double arg)
                {
                        return(cos(arg));
                }



Top 10 Lines:

     Line      Count

       29  100000001
       11          1

Execution Summary:

        3   Executable lines in this file
        3   Lines executed
   100.00   Percent of the file executed

100000002   Total number of line executions
33333334.00   Average executions per line
arnoldg@bd-login:~/c>
```

Note that at optimization levels higher than -O2 (-O3 , -qhot ...) source code annotation may produce no output with the XL compilers.

2. Recompiling the code with -O0 (disable optimization) yields:

```
arnoldg@bd-login:~/c> gprof 1cpuO0
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds     calls  ms/call  ms/call  name
 76.41      0.76     0.76         1   764.06   975.19  .work
 21.11      0.98     0.21 100000001     0.00     0.00  .mycos
  3.02      1.01     0.03                             .__gmon_start__
```

Three functions (work, mycos, and gmon_start) have been caught by the profiler during the unoptimized run. In the % time column on the far left, you can see how much of the total run time each function uses.

Recompiling the code with -O5 (high optimization) yields the following:

```
arnoldg@bd-login:~/c> gprof 1cpuO5 | head -50
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.38      3.75     3.75                             .main
```

As you can see, the only function called is the program itself. This is because, among other things, optimizing the compiler to level -O5, the highest level for IBM compilers, inlines all the functions called by the program. Keep in mind, however, that using the highest level of compiler optimization is a mixed blessing: while it may increase performance significantly, it may also generate incorrect code.

3. Using strace with the -c option will give you a high-level profile of your application.

Whereas other performance tools tend to focus mainly on aspects of code performance, strace is particularly useful because it is the only utility that reports how much system time is devoted to IO (a common bottleneck source). In the example below, IO processes constitute only a small fraction of the overall time duration.

```
arnoldg@bd-login:~/c/matmul> strace -c ./matmul
Time = 8.791451
Rate = 228.176880 MFLOPS
matrix checksum= 251502250000000.0000
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 88.09    0.002004           3       574           rt_sigreturn
  6.33    0.000144         144         1           execve
  1.49    0.000034           7         5           close
  1.45    0.000033           0       131       126 open
  0.66    0.000015           2         9           mmap
  0.53    0.000012           3         4           write
  0.53    0.000012           2         5           fstat
  0.35    0.000008           0       112       105 stat
  0.22    0.000005           3         2           setitimer
  0.18    0.000004           2         2           rt_sigaction
  0.09    0.000002           1         3           brk
  0.09    0.000002           2         1           writev
  0.00    0.000000           0         3           read
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           fadvise64
------ ----------- ----------- --------- --------- ----------------
100.00    0.002275                   857       232 total
arnoldg@bd-login:~/c/matmul>
```

4. In this exercise, first we compiled the matmul code using the -g option so we could see line numbers in our output and then ran perfsuite's psrun command:. Our output shown below includes both raw counter events and statistics. Note that cache hits are greater than cache misses by a 2-to-1 ratio.

```
> xlc_r -O2 -g -qarch=pwr7 -qtune=pwr7 -qcache=auto -qsimd=auto -o matmul.02 matmul.c
> psrun -d all -o count ./matmul.02
Time = 2.206297
Rate = 909.218496 MFLOPS
matrix checksum= 251753752250000.0000

> psprocess count.10846.bd-login.xml
PerfSuite Hardware Performance Summary Report

Version                      : 1.0
Created                      : Mon Jun 28 15:17:07 CDT 2010
Generator                    : psprocess Java version 0.1
XML Source                   : count.10846.bd-login.xml

Execution Information
================================================================================
Collector                    : libpshwpc
Date                         : Mon Jun 28 15:16:55 CDT 2010
Host                         : bd-login
Process ID                   : 10846
Thread                       : 0
User                         : gbauer
Command                      : matmul.02

Processor and System Information
================================================================================
Node CPUs                    : 64
Vendor                       : IBM
Family                       : Power 7
Brand                        : Power7
CPU Revision                 : 2.1
```

```
Clock (MHz)                 : 3864.000
Memory (MB)                 : 63130.38
Pagesize (KB)               : 64

Cache Information
================================================================================
Cache levels                : 3
--------------------------------
Level 1
Type                        : data
Size (KB)                   : 32
Linesize (B)                : 128
Associativity               : 8
Type                        : instruction
Size (KB)                   : 32
Linesize (B)                : 128
Associativity               : 4
--------------------------------
Level 2
Type                        : unified
Size (KB)                   : 256
Linesize (B)                : 128
Associativity               : 8
--------------------------------
Level 3
Type                        : unified
Size (KB)                   : 4096
Linesize (B)                : 128
Associativity               : 0

Event Count Information
================================================================================
Index Description                                                  Counter Value
--------------------------------------------------------------------------------
    1 Branch instructions.....................................   1,003,050,576
    2 Conditional branch instructions mispredicted.............       1,032,517
    3 Conditional branch instructions correctly predicted.......   1,002,707,426
    4 Floating point instructions.............................   1,009,655,367
    5 Floating point operations...............................   1,983,093,122
    6 Cycles integer units are idle...........................   7,531,580,715
    7 Hardware interrupts.....................................               0
    8 Level 1 data cache accesses.............................   2,139,993,201
    9 Level 1 data cache misses...............................   1,063,877,320
   10 Level 1 instruction cache hits..........................   1,738,276,221
   11 Level 1 instruction cache misses........................         152,273
   12 Level 2 data cache misses...............................      63,099,206
   13 Level 2 instruction cache hits..........................          17,852
   14 Level 2 instruction cache misses........................         465,141
   15 Level 3 data cache misses...............................          42,677
   16 Level 3 data cache reads................................      63,099,206
   17 Level 3 instruction cache accesses......................         233,635
   18 Level 3 instruction cache misses........................          76,955
   19 Load instructions.......................................   3,202,604,363
   20 Store instructions......................................       1,266,158
   21 Cycles with no instruction issue........................   6,532,529,181
   22 Total cycles............................................   8,550,549,150
   23 Instructions issued.....................................   6,055,739,406
   24 Instructions completed..................................   6,051,604,296

Event Index
--------------------------------------------------------------------------------
 1: PAPI_BR_INS      2: PAPI_BR_MSP      3: PAPI_BR_PRC      4: PAPI_FP_INS
 5: PAPI_FP_OPS      6: PAPI_FXU_IDL     7: PAPI_HW_INT      8: PAPI_L1_DCA
 9: PAPI_L1_DCM     10: PAPI_L1_ICH     11: PAPI_L1_ICM     12: PAPI_L2_DCM
13: PAPI_L2_ICH     14: PAPI_L2_ICM     15: PAPI_L3_DCM     16: PAPI_L3_DCR
17: PAPI_L3_ICA     18: PAPI_L3_ICM     19: PAPI_LD_INS     20: PAPI_SR_INS
21: PAPI_STL_ICY    22: PAPI_TOT_CYC    23: PAPI_TOT_IIS    24: PAPI_TOT_INS

Statistics
================================================================================
Counting domain...............................................              all
Multiplexed...................................................              yes
Graduated floating point instructions per cycle...............            0.118
Floating point operations per cycle...........................            0.232
Floating point instructions per graduated instruction.........            0.167
Floating point operations per graduated instruction...........            0.328
Floating point instructions per level 1 data cache access.....            0.472
Graduated instructions per cycle..............................            0.708
Issued instructions per cycle.................................            0.708
Graduated instructions per issued instruction.................            0.999
Issued instructions per level 1 instruction cache miss........       39,768.964
Graduated instructions per level 1 instruction cache miss.....       39,741.808
Level 1 data cache accesses per graduated instruction.........            0.354
% floating point instructions of all graduated instructions...           16.684
Percentage of cycles with no instruction issued...............           76.399
Graduated loads and stores per floating point instruction.....            3.173
Graduated loads and stores per floating point operation.......            1.616
Mispredicted branches per correctly predicted branch..........            0.001
Level 1 instruction cache misses per issued instruction.......            0.000
Level 2 cache line reuse (data)...............................           15.860
Level 3 cache line reuse (data)...............................        1,477.530
Level 2 cache hit rate (data).................................            0.941
Level 3 cache hit rate (data).................................            0.999
Level 1 cache miss ratio (data)...............................            0.497
Level 3 cache miss ratio (instruction)........................            0.329
MFLOPS (cycles)...............................................          896.161
MFLOPS (wall clock)...........................................          893.799
MFLIPS (cycles)...............................................          456.264
MFLIPS (wall clock)...........................................          455.061
MIPS (cycles).................................................        2,734.725
MIPS (wall clock).............................................        2,727.515
CPU time (seconds)............................................            2.213
Wall clock time (seconds).....................................            2.219
% CPU utilization.............................................           99.736
```

5. While in most cases perfsuite runs without any instrumentation, in order to do source code level profiling some instrumenting is done at entry and exit. As an example, the following illustrates how to profile code with respect to cycles (PAPI_TOT_CYC).

   First, we get an estimate for the cycle threshold (akin to resolution but don't make it small or overhead becomes an issue).

```
> psprocess -C count.10846.bd-login.xml  | grep PAPI_TOT_CYC

PAPI_TOT_CYC..................................................          3853813
```

   Second, copy and edit the configuration XML file papi_profile_cycles.xml for profiling. `> cp /opt/usersoft/perfsuite/current/share/perfsuite/xml/pshwpc/papi_profile_cycles.xml .`

```
> vi papi_profile_cycles.xml
<ps_hwpc_event type="preset" name="PAPI_TOT_CYC" threshold="3853813"/>
```

   Third, profile using psrun specifying the xml configuration file.

```
> psrun -d all -c papi_profile_cycles.xml -o cycles ./matmul.02
Time = 2.243582
Rate = 894.108608 MFLOPS
matrix checksum= 251753752250000.0000
```

Lastly, analyze profile data using psprocess. At the bottom of the PerfSuite Hardware Performance Summary Report, PerfSuite calls out a portion of the code where a lot of time is being spent--in this case, a nested loop.

```
> psprocess cycles.15920.bd-login.xml
PerfSuite Hardware Performance Summary Report

Version                      : 1.0
Created                      : Mon Jun 28 16:19:55 CDT 2010
Generator                    : psprocess Java version 0.1
XML Source                   : cycles.20245.bd-login.xml

Execution Information
================================================================================
Collector                    : libpshwpc
Date                         : Mon Jun 28 16:16:56 CDT 2010
Host                         : bd-login
Process ID                   : 20245
Thread                       : 0
User                         : gbauer
Command                      : matmul.02

Processor and System Information
================================================================================
Node CPUs                    : 64
Vendor                       : IBM
Family                       : Power 7
Brand                        : Power7
CPU Revision                 : 2.1
Clock (MHz)                  : 3864.000
Memory (MB)                  : 63130.38
Pagesize (KB)                : 64

Cache Information
================================================================================
Cache levels                 : 3
--------------------------------
Level 1
Type                         : data
Size (KB)                    : 32
Linesize (B)                 : 128
Associativity                : 8
Type                         : instruction
Size (KB)                    : 32
Linesize (B)                 : 128
Associativity                : 4
--------------------------------
Level 2
Type                         : unified
Size (KB)                    : 256
Linesize (B)                 : 128
Associativity                : 8
--------------------------------
Level 3
Type                         : unified
Size (KB)                    : 4096
Linesize (B)                 : 128
Associativity                : 0

Profile Information
================================================================================
Class                        : PAPI
Version                      : 4.0.0.3
Event                        : PAPI_TOT_CYC (Total cycles)
```

```
Period                       : 3853813
Samples                      : 2262
Domain                       : all
Run Time (seconds)           : 2.27
Min Self %                   : (all)

Module Summary
--------------------------------------------------------------------------------
  Samples    Self %   Total %  Module

     2262   100.00%   100.00%  /home/gbauer/examples/profiling/matmul.02

File Summary
--------------------------------------------------------------------------------
  Samples    Self %   Total %  File

     2262   100.00%   100.00%  /home/gbauer/examples/profiling/matmul.c

Function Summary
--------------------------------------------------------------------------------
  Samples    Self %   Total %  Function

     2262   100.00%   100.00%  main

Function:File:Line Summary
--------------------------------------------------------------------------------
  Samples    Self %   Total %  Function:File:Line

     2252    99.56%    99.56%  main:/home/gbauer/examples/profiling/matmul.c:33
        7     0.31%    99.87%  main:/home/gbauer/examples/profiling/matmul.c:30
        2     0.09%    99.96%  main:/home/gbauer/examples/profiling/matmul.c:61
        1     0.04%   100.00%  main:/home/gbauer/examples/profiling/matmul.c:21
> cat -n matmul.c
...
    29              for (j=1; j<matSize; j++) {
    30                  sum = 0.0;
    31                  for (k=1; k<matSize; k++) {
    32                      sum += matA[i][k] * matB[k][j];
    33                  }
    34                  matC[i][j] = sum;
    35              }
```

... For comparison, we replaced the loops in the matmul code with ESSL DGEMM (includes matrix transposes in call to DGEMM due to Fortran/C differences). In the following report note the improved MFLOPS rate, and the reduction in issued instructions.

```
> psrun -d all -o essl  ./a.out
Time = 0.079958
Rate = 25088.245760 MFLOPS
matrix checksum= 251753752250000.0000

> psprocess essl.16470.bd-login.xml
PerfSuite Hardware Performance Summary Report

Version                      : 1.0
Created                      : Mon Jun 28 15:32:46 CDT 2010
Generator                    : psprocess Java version 0.1
XML Source                   : essl.16470.bd-login.xml

Execution Information
================================================================================
Collector                    : libpshwpc
Date                         : Mon Jun 28 15:32:39 CDT 2010
Host                         : bd-login
```

```
Process ID                   : 16470
Thread                       : 0
User                         : gbauer
Command                      : a.out

Processor and System Information
================================================================================
Node CPUs                    : 64
Vendor                       : IBM
Family                       : Power 7
Brand                        : Power7
CPU Revision                 : 2.1
Clock (MHz)                  : 3864.000
Memory (MB)                  : 63130.38
Pagesize (KB)                : 64

Cache Information
================================================================================
Cache levels                 : 3
--------------------------------
Level 1
Type                         : data
Size (KB)                    : 32
Linesize (B)                 : 128
Associativity                : 8
Type                         : instruction
Size (KB)                    : 32
Linesize (B)                 : 128
Associativity                : 4
--------------------------------
Level 2
Type                         : unified
Size (KB)                    : 256
Linesize (B)                 : 128
Associativity                : 8
--------------------------------
Level 3
Type                         : unified
Size (KB)                    : 4096
Linesize (B)                 : 128
Associativity                : 0

Event Count Information
================================================================================
Index Description                                                  Counter Value
--------------------------------------------------------------------------------
    1 Branch instructions.......................................     22,987,951
    2 Conditional branch instructions mispredicted..............        197,459
    3 Conditional branch instructions correctly predicted.......     22,397,254
    4 Floating point instructions...............................    329,077,016
    5 Floating point operations.................................  2,117,681,667
    6 Cycles integer units are idle.............................    296,014,088
    7 Hardware interrupts.......................................              0
    8 Level 1 data cache accesses...............................    244,124,656
    9 Level 1 data cache misses.................................     15,543,526
   10 Level 1 instruction cache hits............................    111,430,607
   11 Level 1 instruction cache misses..........................         20,384
   12 Level 2 data cache misses.................................        356,450
   13 Level 2 instruction cache hits............................          1,100
   14 Level 2 instruction cache misses..........................         57,578
   15 Level 3 data cache misses.................................         63,651
   16 Level 3 data cache reads..................................        356,450
   17 Level 3 instruction cache accesses.......................          28,101
   18 Level 3 instruction cache misses.........................          10,363
   19 Load instructions........................................     253,002,409
   20 Store instructions.......................................       6,665,773
   21 Cycles with no instruction issue.........................     114,663,356
   22 Total cycles.............................................     334,702,055
   23 Instructions issued......................................     846,375,443
   24 Instructions completed...................................     833,443,582

Event Index
--------------------------------------------------------------------------------
 1: PAPI_BR_INS      2: PAPI_BR_MSP      3: PAPI_BR_PRC      4: PAPI_FP_INS
 5: PAPI_FP_OPS      6: PAPI_FXU_IDL     7: PAPI_HW_INT      8: PAPI_L1_DCA
 9: PAPI_L1_DCM     10: PAPI_L1_ICH     11: PAPI_L1_ICM     12: PAPI_L2_DCM
13: PAPI_L2_ICH     14: PAPI_L2_ICM     15: PAPI_L3_DCM     16: PAPI_L3_DCR
17: PAPI_L3_ICA     18: PAPI_L3_ICM     19: PAPI_LD_INS     20: PAPI_SR_INS
21: PAPI_STL_ICY    22: PAPI_TOT_CYC    23: PAPI_TOT_IIS    24: PAPI_TOT_INS

Statistics
================================================================================
Counting domain................................................             all
Multiplexed....................................................             yes
Graduated floating point instructions per cycle................           0.983
Floating point operations per cycle............................           6.327
Floating point instructions per graduated instruction..........           0.395
Floating point operations per graduated instruction............           2.541
Floating point instructions per level 1 data cache access.......          1.348
Graduated instructions per cycle...............................           2.490
Issued instructions per cycle..................................           2.529
Graduated instructions per issued instruction..................           0.985
Issued instructions per level 1 instruction cache miss..........     41,521.558
Graduated instructions per level 1 instruction cache miss.......     40,887.146
Level 1 data cache accesses per graduated instruction..........           0.293
% floating point instructions of all graduated instructions.....         39.484
Percentage of cycles with no instruction issued................          34.258
Graduated loads and stores per floating point instruction.......          0.789
Graduated loads and stores per floating point operation.........          0.123
Mispredicted branches per correctly predicted branch...........           0.009
Level 1 instruction cache misses per issued instruction.........          0.000
Level 2 cache line reuse (data).................................         42.606
Level 3 cache line reuse (data).................................          4.600
Level 2 cache hit rate (data)...................................          0.977
Level 3 cache hit rate (data)...................................          0.821
Level 1 cache miss ratio (data).................................          0.064
Level 3 cache miss ratio (instruction)..........................          0.369
MFLOPS (cycles).................................................     24,447.779
MFLOPS (wall clock).............................................     23,170.376
MFLIPS (cycles).................................................      3,799.061
MFLIPS (wall clock).............................................      3,600.559
MIPS (cycles)...................................................      9,621.769
MIPS (wall clock)...............................................      9,119.029
CPU time (seconds)..............................................          0.087
Wall clock time (seconds).......................................          0.091
% CPU utilization...............................................         94.775
```

6. First, we generated measurement data using psrun on NCSA's Abe system. The XML files generated were:
   - matvec-goodcache.3619.honest2.xml
   - matvec-badcache.3626.honest2.xml
7. The output of running psprocess with the generated XML files on Abe:
   - goodcache.out
   - badcache.out

8. Notice the significant differences in "Graduated instructions per cycle", "% cycles stalled on any resource", "MIPS (wall clock)", "CPU time", and "Wall clock time".

9. For this exercise we use pi, a simple MPI C++ program with TAU's profiling API calls inside.

   The following describes how we did the problem on NCSA's Abe system.

   First we loaded the proper Intel compiler and TAU software.
   `module load intel/11.1.072`
   `module load tau` Then, in pi/, we modified the file "Makefile" to include the relevant TAU make file by changing the line `include $(TAUROOTDIR)/include/Makefile` to `include $(TAUROOTDIR)/x86_64/lib/Makefile.tau-icpc-mpi` and built the program: `cd pi`
   `cp -p Makefile Makefile.0`
   `vi Makefile`
   `make` We then executed the program:

   For 1 MPI process: `./cpi` Or, for 4 MPI processes: `mpd &`
   `mpirun -np 4 ./cpi` and used TAU's pprof and paraprof to view the generated profile files: `pprof -m`
   `paraprof`

10. For automatic source instrumentation, you will need a version of TAU built with PDT and a makefile name containing pdt.

    The following describes how we did the problem on NCSA's Abe system.

    First in the makefile change the line: `include $(TAUROOTDIR)/include/Makefile` to `include $(TAUROOTDIR)/x86_64/lib/Makefile.tau-icpc-pdt` `cd autoinstrument`
    `cp -p Makefile Makefile.0`
    `vi Makefile`
    `make` Then we executed the program klargest: `./klargest` and used TAU's pprof and paraprof to view the generated profile files: `pprof -m`
    `paraprof`
