Assignment 6. Multithreaded performance

Laboratory

Ordinarily when processes run in Linux, each gets its own virtual processor. For example, when you run the command:

```
tr -cs 'A-Za-z' '[\n*]' | sort -u | comm -23 - words
```

there are three processes, one each for `tr`, `sort`, and `comm`. Each process has its own virtual memory, and processes can communicate to each other only via system calls such as `read` and `write`.

This lab focuses on a different way to gain performance: multithreading. In this approach, a process can have more than one thread of execution. Each thread has its own instruction pointer, registers and stack, so that each thread can be executing a different function and the functions' local variables are accessed only by that thread. However, threads can directly access shared memory, and can communicate results to each other efficiently via the shared memory, so long as they take care not to step on each others' toes.

Synchronization is the Achilles' heel of multithreading, in that it's easy to write buggy programs that have race conditions, where one thread is reading from an area that another thread is simultaneously writing to, and therefore reads inconsistent data (a polite term for "garbage"). This lab does not attack that problem: you will use a prebuilt application that should not have internal race conditions, and you will write an application that is embarrassingly parallel, so that there's no need for subthreads to synchronize with each other.

Lab

Starting with coreutils 8.6, released 2010-10-15, GNU `sort` can use multiple threads to improve performance. This improvement to GNU `sort` was contributed by UCLA students as part of Computer Science 130, the undergraduate software engineering course. This improvement is in the current version of GNU `sort` installed as `/usr/local/cs/bin/sort` in the SEASnet GNU/Linux servers.

Run the command `sort --version` to make sure you're using a new-enough version. Investigate how well the multithreaded sort works, by measuring its performance. First, generate a file containing 10,000,000 random single-precision floating point numbers, in textual form, one per line with no white space. Do this by running the the `od` command with standard input taken from `/dev/urandom`, interpreting the bytes read from standard input as single-precision floating point numbers. (Almost certainly you will occasionally get NaNs, but that's OK; just leave them in there.) Process the output of `od` using standard tools such as `sed` and `tr` so that each floating-point number is on a separate line, without any white space.

Once you have your test data, perhaps in a pipe or perhaps in a file, use `time -p` to time the command `sort -g` on that data, with the output sent to `/dev/null`. Do not time `od` or any of the rest of your test harness; time just `sort` itself.

Invoke `sort` with the `--parallel` option as well as the `-g` option, and run your benchmark with 1, 2, 4, and 8 threads, in each case recording the real, user, and system time. Assuming your `PATH` environment variable is set properly so that `/usr/local/cs/bin` is at its start, you can use `sort --help` or `info sort` for details about how to use the `--parallel` option.

Keep a log of every step you personally took during the laboratory to measure the speed of `sort`, and what the results of the step were. The idea behind recording your steps is that you should be able to reproduce your work later, if need be.

Homework

Modify the simple ray tracer code in Brian Allen's SRT implementation so that the code is multithreaded and runs several times faster on a multicore machine, such as one of the SEASnet Linux servers.

SRT is made of several components. You need to modify only the code in `main.c` and the `Makefile` in order to multithread it. Your implementation should use POSIX threads, so you'll need to include `<pthread.h>` and link with the `-lpthread` library. You'll need to modify the `main` function so that it does something useful with the `nthreads` variable that it computes from the leading digit string in the first operand of the program; currently it errors out unless `nthreads` is 1.

Your new code should invoke `pthread_create` and `pthread_join` to create your threads, and to wait for them to finish. It need not use any other part of the POSIX threads interface; the rest of this (complicated) interface is not needed for this application. It is OK if your new code needs to allocate some additional memory via `malloc` or similar primitives.

Benchmark and test your program with the command "`make clean check`"; this command should output a file `1-test.ppm` that is byte-for-byte identical with the similarly-named file that is output by the unmodified SRT code, a copy of which is in the file `baseline.ppm`. This file is in standard Netpbm format and can be displayed as an image by GIMP and many other graphics tools.

Submit

Submit the following files.

1. A copy of your lab log, as a file `log.txt`.
2. The output of the command `make clean check`, as a text file `make-log.txt`.
3. A gzipped tar file `srt.tgz`, generated by `make dist`.
4. A brief after-action report of your homework, as a text file `readme.txt`. This should discuss any issues that you ran into, and your conclusions about how well your implementation of SRT improves its performance.

If the above files are all in the working directory, the following shell commands should work:

```
gunzip <srt.tgz | tar xf -
(cd srt && make clean check) 2>&1 | diff -u - make-log.txt
awk '200 < length' log.txt readme.txt
```

These commands should output only minor timing differences.

© 2010, 2014–2017 Paul Eggert. See copying rules.
$Id: assign6.html,v 1.17 2017/01/25 00:24:31 eggert Exp $