Lab 8: Image processing on Multicore Architecture

CPSC 1021 – Spring 2020

Introduction

During this lab you will:

1. Make use of modern multicore CPU architecture.
2. Make use of modern concurrent processing to parallelize a section of your code.
3. Use 2D float arrays to create a PPM image.
4. Use multiple cores so that each writes a chunk of image rows into memory.

Part 0 - Background

What is a Multicore Architecture?

Multicore architecture is also called multicore technology. It describes a computer with multicore processor/s. Multicore processors are processors that implement the core logic of multiple processors. In this case one integrated circuit unit (ICU), also called a die, contains more than one processor core, and all of them work together as a single unit. This enables to create a system that has a significant boost in speed and efficiency and increased overall system performance, while using the same or less amount of energy. Multicore processors are used in many mobile devices, desktops and servers and are common for such tasks as 3-D games, encoding and editing videos.

Multicore architecture makes parallel computing possible. Parallel computing divides computing tasks between the cores and executes them at the same time, in parallel. The results are then delivered by means of a single shared gateway. Processors that have two cores are called dual core processors, those that have four cores – quad core processors, etc.

Each core can have two threads. A thread is a unit of execution. So, if you have 4 cores, you can run two threads on each core, with the total of 8 threads!

So, what is the highest number of cores we can have on one CPU today? As of today, we have two processors with 8 cores and 16 threads – Intel i9 9900k and AMD Ryzen 7 2700x. AMD also announced a new AMD Ryzen 9 3850x processor with 16 cores and 32 threads, but it is not available on the market yet.

Here is a 5-minute video that explains multicore technology and how it can be used.
https://www.youtube.com/watch?v=d_DYHdIhsr4

Multicore technology should not be confused with multiprocessor technology. Multiprocessor means one computer has multiple CPUs, while multicore has multiple cores or processing units on one processor. Can one computer have multiple multicore CPUs? Sure thing, just imagine the processing power!

You will learn more about this in the computer organization course, but for now you will experiment using multicore machines to process a ppm image.

Determining the number of CPUs and CPU cores on a Linux machine

This lab will be done on one of the SoC *babbage* machines (as usually). There are several useful commands that will allow you to obtain system information.

`uname` - displays general system information. You can *man* it to see what flags can be used with this command to get specific system info. If you just type it on the command line without any flags, you will see the type of the OS your machine uses: Linux. Go ahead and try this command with different flags to see what you can learn about your system.

`uname -p` - displays the type of the processor: x86_64.

However, this is also a general type of information, that does not display the number of system CPUs and cores per CPU. Luckily, Linux has another command that shows the number of processors:

`nproc` - this one display the number of processors, but nothing else.

`lscpu | less` - (list cpu info) This command displays a lot of details: your system's cpu information, including the processor model, number of cpus, number of cores per cpu, number of threads per core, processor architecture, etc. *Piping* this command into *less* allows you to see less info on each screen and allows you to scroll up and down. Finally, a useful command for a computer scientist like yourself! Go ahead, run this command and find out all you can about your machine. You can learn more about managing your system in the future Linux System Administration Course (CPSC4240).

`less /proc/cpuinfo` - this command displays even more detailed information about each processor on the system. Try it to see what you find out.

Part I – Kickin' it old school: write out a PPM image using 2D arrays

Use three `float` 2D arrays to represent the pixel contents of an image (the payload):

```
int      h = 480, w = 640;
float    R[h][w];
float    G[h][w];
float    B[h][w];
```

and then use a double-nested `for` loop to set the pixel colors to whatever you like, e.g., red:

```
for(int y = h - 1; y >= 0; y--) {
  for(int x = 0; x < w; x++) {
       R[y][x] = 1.0;
       G[y][x] = 0.0;
       B[y][x] = 0.0;
  }
}
```

Output the PPM image:

```
// open output file
out = fopen("color.ppm","w");
if(!out) {
   printf("Error in opening output image\n");
   return 1;
}

// write output
fprintf(out,"P6\n%d %d\n255\n", w, h);
for(i = 0; i < h; i++) {
  for(j = 0; j < w; j++) {
    r = (unsigned char)(R[i][j] * 255.0);
    g = (unsigned char)(G[i][j] * 255.0);
    b = (unsigned char)(B[i][j] * 255.0);
    fwrite(&r, sizeof(unsigned char), 1, out);


    fwrite(&g,sizeof(unsigned char),1,out);
    fwrite(&b,sizeof(unsigned char),1,out);
  }
}
```

Part II – How many cores?

We will attempt multicore programming using the OpenMP programming API. To do so, your code should `#include <omp.h>` and your source code must be compiled with the `–fopenmp` flag during compilation *and* linking. You can, of course, add this flag to the `Makefile` for convenience.

After setting up the code as shown above and testing it to make sure it outputs a color image, find out how many cores the machine you are logged in to has by putting this bit of code into your `main()` function:

```
// figure out how many threads we have available
#pragma omp parallel private(tid)
{
  if((tid = omp_get_thread_num()) == 0)
    ncores = omp_get_num_threads();
}
fprintf(stderr, "num cores: %d\n", ncores);
```

where `tid` and `ncores` are both `int` types.

Part III – How many rows per core?

Now that you know how many cores you have, to fill in an image of height `h`, figure out how many rows each core should fill in:

```
// calculate chunk, splitting up work as evenly as possible
chunk = h/ncores;
fprintf(stderr,"h: %d\n", h);
fprintf(stderr,"chunk (%d/%d): %d\n", h, ncores, chunk);
```

Part IV – The tricky part

To parallelize the code, you have to tell OpenMP which parts of memory (the variables) are private (unique) to each core (meaning each core gets its own copy) and which are shared (all cores see the same memory and the memory contents are not unique to any of the cores).

In this instance, the variables `w, h, R, G, B`, do not change across cores, and so are shared. However, the variable `tid` (used to represent thread id) is unique to each core, since each core gets its own thread id.

You tell OpenMP what's shared and what is private using this `#pragma` call:

```
#pragma omp parallel \
        shared(w,h,R,G,B) \
        private(tid)
```

and you place this two lines above the double-nested for loop where you set the image colors.

Part V – Parallelizing the outer for loop

To parallelize the outer for loop, insert this `#pragma` call just above the two outer for loops and the first `#pragma` call above in Part III.

```
#pragma omp for schedule(static,chunk) nowait
```

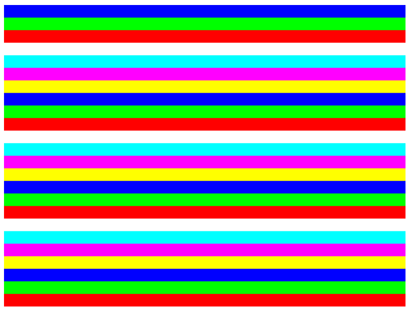
This call tells OpenMP to unroll the for loop (that loops over the rows) so that each "chunk" (or band) of rows is run on its own core so that the whole outer `for` loop runs in parallel.

Part VI – The sort of tricky part: putting it all together

Now, to make each core write its own color, you need to figure out which core is executing at any given time. Remember that there might be 8 cores running *at the same time* (in parallel). To figure out which is which, you use this bit of code:

```
        tid = omp_get_thread_num();
```

inside the double nested `for` loop. This will give you an integer in the range `[0,ncores]`. Now, your task is to use this thread id to fill in its own color band with its own specific color. For example, below, on a 24-core machine, each core would fill in one of 7 colors (e.g., `tid % 7`) to produce the output image below.



Create a makefile to build this code. Make sure to make clean before you create and submit your tarball.

NOTE: If you submit the archive that cannot be open, or is corrupt, you will not be given the opportunity to redo the work and resubmit. You have one shot to get it right.

What/How to submit

That's it, you are done! You can now submit your zipped tarball to canvas to be graded.