EECS 213 Parallelism Lab (Beta Test)

In this lab, you will build a parallel implementation of image filtering via convolution and measure its performance on one or more machines. The lab will hopefully get you to think about what goes into making a parallel algorithm, and expose you to low-level parallel programming on in a shared memory model using pthreads, as well as to Unix I/O.

This lab can be done in teams of two people. Please email me and the TA about your team as soon as possible.

Your implementation will:

1. Load a binary image from a file. (There are at least two approaches you can try)
2. Load a filter from a file. (Again, at least two possible approaches)
3. Apply the filter to the image making the best use of the processors/cores that are available. There are many possible approaches, involving different algorithms, numbers of threads, mapping of threads to processors, use of the memory system, etc.
4. Store the binary image back to a file. (Again, at least two possible approaches).

You will measure how fast each of these stages is, on any machine you choose. Furthermore, you will measure the speed as a function of the number of processor cores you use, producing a *scalability* measurement. You'll report performance on the discussion group, and we will have a little friendly competition on (a) how fast our implementations are, and (b) how well they *scale*.

~cs213/HANDOUT/parallel-lab on the 213 machine contains example programs that you will want to look at. Currently, there are examples for image convolution, basic Unix I/O, memory-mapped I/O, pthreads, and timers. I may add other examples if needed.

Making an image and a kernel

For the purposes of this lab, an image will be an N by N array of double precision floating point numbers (pixels), where each number indicates intensity (0=black, increasingly larger numbers mean increasing brightness). A convolution kernel, which will be described in more detail later, is an M by M image, where M is an odd number. When stored in a file, an image or kernel is stored in raw binary form – that means that that the raw bytes of the array are written into the file in the same order that they appear in memory, with no conversion.

We will supply some example images and kernels, but it may be fun to make your own. The 213 machine has a tool called ImageMagick installed. ImageMagick is commonly installed on Linux machines and is free, open-source software you can download and install yourself. The ImageMagick convert command can convert images from just about any format into one suitable for this class. For example, this converts myimage.jpg to myimage.gray:

convert myimage.jpg –scale 512x512! –depth 64 -define quantum:format=floating-point myimage.gray

You can also use the ImageMagick display command to display an image to you:

display –size 512x512 –depth 64 –define quantum:format=floating-point myimage.gray

Note: There are currently some issues with the above commands on the 213 machine. An update will be emailed later.

Image filtering via convolution

There are many forms of image filtering. The one we will focus on here is *convolution with a kernel.* This filtering technique is what is behind many Photoshop filters, such as sharpening, blurring, etc. Convolution can be done in 1D (audio filtering, for example), 2D (image filtering), and in higher dimensions.

A sequential version of convolution for 2D images is given in the file *conv-ex.c*. This is commonly referred to as *boxcar convolution* because you can visualize the kernel as a box marching across the pixels of the input image. At each pixel, we compute a weighted sum of that pixel and the neighboring pixels. The *weights* are the values in the kernel. Different weights give you different filtering effects. This process can implement any *linear filter* just by choosing the right size kernel (its *extent*) and setting its weights correctly. One subtlety is that parts of the kernel can be "hang off the corners and edges" of the input image as it marches across the image. In this case, the "missing" parts of the input image are treated as zero.

Regular Unix I/O and memory-mapped I/O

You will need to read the input image file and the kernel file, as well as write the output image file. There are at least two ways to do this.

The file *io-ex.c* gives an example of using regular Unix I/O to read and write files. Regular I/O consists of the use of the Unix system calls open, read, write, lseek, and close. You can learn much more about regular I/O in the book, and in the handout *Unix Systems Programming In A Nutshell*, available on the web page. The main idea is that regular I/O is explicit – you need to tell the operating system exactly what to do and when to do it using the system calls.

The file *mmap-ex.c* gives an example of using memory-mapped I/O to read and write files. In memory-mapped I/O, we ask the operating system (via the mmap system call) to map the file into our address space so that we can treat it like a chunk of data in memory. Recall that in execing a program, the operating system memory maps portions of the executable program file into the address space and then jumps to it. The mmap system call gives us access to the same functionality. Memory-mapped I/O is

implicit – you just read and write memory, and the operating system translates that into actual I/O as needed.

Pthreads and processor affinity

You will need to partition the work of doing the image convolution among multiple processors. To do this, you will use threads, which are explained in some detail in your book. In Linux, the threading interface is called pthreads.

The file *pthread-ex.c* shows how to use the basic pthread system calls. It is very important that you supply the –pthread option to gcc when compiling code that uses pthreads (see the Makefile). The pthread_create system call creates a new thread that starts executing in the function you specify. That is, it looks like a function call, but the caller does not wait for the callee to finish! Instead, the caller and callee continue to run simultaneously. The caller can explicitly wait for the callee to finish by using the pthread_join system call.

Note that this is quite similar to the fork and wait system calls discussed in more detail in class, the book, and the systems programming handout. However, while fork creates an entirely new process that is an independent clone of the parent process, pthread_create creates a new thread of execution (that starts at the callee function) within the current process. The new thread of execution shares all the memory and other state of the current process with the thread that created it, and with all of the other threads in the process. This means they must carefully coordinate access to the shared memory to avoid serious and difficult to track down bugs. While this is extremely challenging, it is outside the scope of this lab. In this lab, your threads only need to read from shared memory (the input image and kernel) . Their writes to the output image do not need to overlap.

You can create as many threads as you want (and that the operating system has memory to track). The operating system will interleave the execution of these threads in time and across the processors available on the system. That is, the operating system can switch from thread to thread on any given processor, and it can move a thread from one processor to another. This scheduling activity happens on the order of every millisecond or so. It does this to "balance the load" and to maintain "fairness" among all the threads in the system. However, it is sometimes convenient, especially in a parallel program, to directly control which processor a thread runs on. This is known as "processor affinity". A thread can advise the operating system of the set of processors it would like to be run on. The *pthread-ex.c* example shows how a thread can request that it only run on a specific processor.

Measuring time and performance

You will measure the performance of your program using the passage of real time. The most accurate measure of real time on the system is the processor cycle counter, which counts the number of clock ticks (1GHz processor = one billion ticks per second) that have occurred since the processor was booted. There is a special instruction to directly read this value. In addition, on Linux, the Unix system call gettimeofday is built on top of the cycle counter. Gettimeofday returns the number of seconds and

microseconds that have passed since the beginning of January 1, 1970. The file *timer-ex.c* illustrates how to use both the cycle counter and gettimeofday to measure how long a chunk of code takes to execute. BTW, the code for using the cycle counter also illustrates a very important feature of the C language – inline assembler. The C language lets us drop to assembler level, and directly write instructions that machine will actually execute, whenever we need to do this. Here, we need to execute a specific instruction to read the cycle counter.

Note that the passage of real time also counts time spent in the operating system (for example, doing exception handling and context switches), time waiting on slow I/O devices to catch up, and time spent running threads other than your own. For the purposes of this lab, we will ignore this, but, if you're interested in exactly where your time is going, you might want to look at the getrusage system call.

When you measure performance of your program, you will do so on physical machines other than the 213 machine, and those machines should have as many processor cores and/or hardware threads as possible. 213.cs.northwestern.edu is a virtual machine (see http://pdinda.org/vlab if you're curious about how and where it runs). There are two consequences. First, it is configured as a single core, single processor virtual machine, which makes it rather uninteresting for parallel programs. Second, because the hardware is virtualized, things like the cycle counter and gettimeofday are much much less accurate than described above. You can write and debug your programs on 213, but when you are ready to do performance testing, you should at least do so on the Tlab and Wilkinson Lab machines. The Tlab machines have a single processor that has two hardware threads of execution (Intel calls this "hyperthreading"). The Wilkinson lab machines have a single processor that has four cores ("quadcore"), each of which can have one or two hardware threads. When you hand in your code, we will also try it out on a dual quadcore (2 processors, 4 cores each) machine, and possibly a quad quadcore (4 processors, 4 cores each). [1]

For the purposes of this lab, we will use hardware thread, core, and processor interchangeably.

[1] The use of the terms processor, core, hardware thread, thread/pthread/software thread, and process can be a bit confusing. Here is what it means. A machine may have one or more processors. Each processor is a separate chip mounted on the motherboard of the machine. The processors share the main memory system (DRAMs). A processor can have one or more cores. A core is a complete execution unit plus one or more levels of memory cache. Each core can independently fetch, decode, and execute instructions. Usually, the cores of a processor share an L2 or L3 cache. Each core may have one or more hardware threads. A hardware thread ("hyperthread" is what Intel likes to call this) consists of hardware that can fetch and decode instructions. All the hardware threads of a core share the single execution engine of the core. Their purpose is basically to keep that engine busy by feeding it work. The operating system creates the abstraction of software threads, which are the pthreads you will program in this lab. The OS dynamically maps software threads onto hardware threads. You can ask that it maps a software thread to specific hardware thread on a specific core on a specific processor. The OS also creates the abstraction of processes, which contain one or more software threads running in a shared memory space. These processes are accessed by the programmer through fork/wait and similar system calls. The OS implements processes using both software threads and virtual memory management, both of which are tightly coupled with the hardware. In addition, some programming languages (e.g., Scheme, some Java implementations, etc) implement another level of threads and processes on top of the operating system supplied software threads (pthreads) and processes.

When we think of the performance of a parallel program, we need to think beyond just the basic run-time for an example. In particular, we are also interested in how the program *scales* with the problem size and with the number of processors. In a perfectly scalable program, we can always double the number of processors and expect the execution time to be cut in half. Very few parallel programs work this way, in fact, if they do, they are usually called "embarrassingly parallel." A good performance measurement of a convolution would look at the execution time as a function of (1) the number of processors, (2) the size of image, and (3) the size of convolution kernel. Another useful view is called a "speedup curve", where we fix the problem size (image and kernel), and vary the number of processors, plotting time-with-1-processor / time-with-p-processors as a function of p.

What to do

Here is a suggested approach to this lab. Keep in mind this lab is a beta test. Don't panic! Ask questions and get help.

1. Read and play with the example code to get a good sense of how the basic tools work.
2. Write a sequential version of convolution that can operate on images and kernels whose size is only known at run-time. The example code has N and M hard-coded. You'll need a version that works if N and M are given when the program is run.
3. Figure out how to read and write image and kernel files.
4. You now have a sequential program that you can use to test your eventual parallel program. The program will have a command-line interface like this:

   seq-conv N input-image M kernel output-image

   Here, the input and output images are NxN and the kernel is MxM.
5. Develop a strategy to parallelize convolution. For the purpose of this project, you can assume that you will not use more than 32 processors, and that each image will have at least 32 rows. As an example of developing a strategy, notice that the convolution example code is a four-deep loop nest. The outer loop iterates over the rows of the output image, and computing each row is independent of computing every other row. Could you take advantage of this?
6. Implement your parallel convolution using pthreads. It must be possible to specify how many threads to use and how many processors to use at run-time.
7. Test your program to make sure it is correct (compare against your sequential program). Your parallel program will have a command-line interface like this:

   par-conv N input-image M kernel output-image T P

   Here, T is the number of threads to use, and P is the number of processors to use.
8. Instrument your program with timers so that it reports the time taken to read the image, read the kernel, do the convolution, and write the output image, as well as the total time it takes.

9. Run your program on the fastest machine, or the machine with the most processors, that you can find. Again, this needs to be some machine other than 213. At least try on Tlab and Wilkinson Lab. You will record each of your instrumented times as a function of:

   Image size: 256, 512, 1024, 2048 4096
   Kernel size: 5, 9, 17, 33, 65, 128, 256
   Processors: 1, 2, 4, 8, … up to however many you have
   Threads: 1, 2, 4, 8, 16, 32

   Notice that performance is likely to be different from the first execution to the second for each case due to caching effects in the operating system. You should report both the "cold" (first time) numbers and the "warmed up" (second time) numbers.

   You will then report these numbers (ideally, graphed) on the discussion group, along with info about the machine:

   cat /proc/cpuinfo (description of the processors on the machine)
   uname –a (description of the kernel, etc)
   hostname (name of the machine)
10. Maybe do some extra credit

Hand in and grading

After you're satisfied with your performance, you will hand in, by sending email to me and the TA, your source code, and your performance data (a copy of what you already posted on the discussion group).

We will grade your lab based on completeness and correctness. Essentially, if you have made a good effort to build a parallel convolution and it works, you will receive full credit. The goal of the performance competition is to have a little fun, not to affect your grade. However, I do want you to participate!

Since this is a beta test, please immediately post any concerns and issues on the discussion group and email me and the TA. Active discussions about the project on the discussion group are encouraged!

Extra Credit

You can earn extra credit by trying different, additional approaches to making convolution fast. Some examples:

- Use the Intel/AMD SSE vector instructions to get parallelism within a single thread of execution.
- Implement convolution through FFT or DCT. This is an alternative algorithm to the one given in the example code that is asymptotically much better. However, to make it parallel, you would

have to figure out a parallel 2D FFT or DCT transform. (BTW, 2D DCT is at the heart of JPEG encoding/decoding).

- Implement parallel convolution using plus scans (parallel prefix operations using the + operator). If this sounds interesting, see me and I will give you a relevant paper (which doesn't appear to be online).
- Implement parallel convolution on your NVIDIA graphics card using CUDA. (A modern graphics card is essentially a parallel computer with anywhere from 16 to 256 or more hardware threads, albeit very weird hardware threads).