Northwestern EECS 213
SETI Lab (Beta Test)
Spring 2013

The Search for Extraterrestrial Intelligence (SETI) is a long-standing international effort to try to find possible alien civilizations (yes, little green men) through their radio wave emissions. An inhabited planet like ours is as bright as a small star in the radio spectrum due to humanity's numerous radio transmitters, from big AM, FM, shortwave, and TV stations all the way down to the tiny transmitters used by computers and phones to connect to WiFi and cellular access points. It is assumed that other civilizations will be similar.

SETI uses antennas generally employed for radio astronomy to "look" at other stars, recording *signals* over an interval of time. Unlike a regular radio receiver, these receivers do not *tune* to a specific frequency, but rather listen on many frequencies at once—they are *broadband* receivers. They also do not try to decode or *demodulate* the signals they receive at all. The point is to produce huge amounts of signal data. The signal data contains not only possible alien signals, but also lots of human signals, and lots of noise from all kinds of natural radio sources.

The signal data is analyzed using computer programs that use *signal processing* techniques to sift through the data looking for artificial signals that do not come from our own planet. This is a huge computational challenge, and a significant breakthrough was the SETI@HOME project[1], which allowed ordinary people to volunteer their computers to participate. Currently, there are over 1.3 million registered users whose computers contribute an average of about two thousand trillion floating point operations per second to the effort (2 *PetaFLOPS*).

In this lab, you will tackle a greatly simplified and constrained version of this computation, with the goal of trying to make our class server (or other computers you'd like to try) execute the computation as fast as it can. You will write a program that, given a raw signal from a broadband receiver, will try to hunt for a specific kind of intelligent signal within it, specifically an AM radio signal carrying audio information. Your program will determine whether this AM signal is there, and also which *band* it appears in. We will give you another program, an *AM demodulator*, which given the signal and the band, will extract the audio and let you hear it. The AM demodulator is basically an AM radio, and your program will tell us how to tune it.

Of course, this is a systems course, so our focus won't be on the aliens or the signal processing. The goals of the lab are

- to get you to think about what goes into making a simple parallel algorithm,

[1] http://setiathome.berkeley.edu/

- to expose you to low-level parallel programming on in a shared memory model using pthreads,
- to expose you to Unix I/O, and
- to expose you to the effects of compiler optimization.

The specific algorithm you'll be using also allows you to control how much computation is done per memory read or write, letting you see the effects of the *memory mountain*.

Logistics

This lab can be done in teams of two people.

To start, grab a copy of ~cs213/HANDOUT/setilab_handout.tar, untar it in your protected working directory. You will find the following elements:

- `README` – describes any last minute details
- `Makefile`
- Generation and Playback Programs
  - `siggen` – this binary program will generate a signal for you to look at.
  - `amdemod` - this binary program will demodulate the signal and create an audio signal in a binary format
  - `bin2wav` – this binary program will translate the audio signal to a WAV file, which can be played back with any media player.
- Signal and Filtering Code
  - `signal.[ch]` – load an store signal files in several ways, including text I/O using the stdio system, binary I/O using the Unix open/seek/read/write/close system call interface, and binary I/O using the Unix mmap system call interface.
  - `filter.[ch]` – generate filters and apply them to signals (sequential only)
  - `timing.[ch]` – find out how long things take using various methods
- `band_scan.c` - a sequential version of the program you'll write
- Pthread examples
  - `pthread-ex.c` - how to create and use threads on Unix (and most platforms)]
  - `parallel-sum-ex.c` – how to compute the sum of an array in parallel
- `seti-eval` – allows you to evaluate your program and compare it to others via the web site.

Your first task is to get some signal data from our server. If this were actually SETI@Home, this would be real radio telescope data made available via a project server. For this lab, we've set up a system that generates virtual signals, akin to what you might see if you pointed a radio telescope out into space.

Where SETI@Home looks for signals in a 2.5 MHz frequency band, we are giving you signals in a 200 KHz frequency band to make things more tractable. The signals comprise roughly 30 seconds of data. Note that a radio telescope is subject to the effects of Earth's rotation. This means that as the telescope records data, it can only look at a certain patch of sky for so long. The Aricebo telescope, for example,

scans a given patch of sky for roughly 12 seconds. Because of this, the scientists at SETI@Home are looking for signals that are in the shape of a *Gaussian*. That is, they increase and decrease in signal strength over a 12 second window in the shape of a Gaussian curve. Signals not in this shape are almost certainly some kind of Earth-based interference.

To get your signal, run the `siggen` binary we provide:

```
$./siggen –n netid1_netid2
```

This will give you a custom virtual signal that may or may not contain an alien transmission. Your signal will be in '`sig.bin`' by default unless you specify otherwise with the –o option. This will also give you a file named '`secret`' that contains a secret key that will be used during submission for authenticating you to our server.

You may modify any of the files in the handout directory, except the generation and playback programs, and you can add new files as needed. You can compile `band_scan.c`, and then use it to test out how things work:

```
$ make
$ ./band_scan
usage: band_scan text|bin|mmap signal_file Fs filter_order num_bands
```

Here, the "`text|bin|mmap`" argument indicates how the `signal_file` is to be interpreted. `Fs`, `filter_order`, and `num_bands` relate to the problem, and will be given to you. We will describe them a bit later. For the purposes of this lab, you can assume that `Fs` will always be 400,000 (400 KHz).

If you run `band_scan` on a signal that has an alien, it will say something like this:

```
$./band_scan bin sig.bin 400000 32 10
type:     Binary
file:     sig.bin
Fs:       400000.000000 Hz
order:    32
bands:    10
Load or map file
Read 11999489 samples
signal average power:     0.336757
    0             0.000100 to         19999.999900 Hz:             0.002624
**(meh)
    1         20000.000100 to         39999.999900 Hz:             0.000108
*(meh)
    2         40000.000100 to         59999.999900 Hz:             0.000060
*(meh)
    3         60000.000100 to         79999.999900 Hz:             0.000061
*(meh)
    4         80000.000100 to         99999.999900 Hz:             0.081506
**********************************(WOW)
    5        100000.000100 to        119999.999900 Hz:             0.081501
```

```
************************************(WOW)
    6        120000.000100 to        139999.999900 Hz:             0.000042
*(meh)
    7        140000.000100 to        159999.999900 Hz:             0.000023
*(meh)
    8        160000.000100 to        179999.999900 Hz:             0.000017
*(meh)
    9        180000.000100 to        199999.999900 Hz:             0.000019
*(meh)


<detailed timing information>

Analysis took 10.719594 seconds

POSSIBLE ALIENS 80000.000100-120000.000100 HZ (CENTER 100000.000000 HZ)
```

What this result means is that there is an unusual amount of power centered around the radio frequency 100,000 Hz (100 KHz) in this signal. This may be an alien! If you run against the example file `~cs213/HANDOUT/noalien.sig`, you'll see what a more boring signal looks like. If you run against the example file `~cs213/HANDOUT/alien.sig`, you'll see what an alien signal looks like.

NOTE: We've checked with our scientists, and they tell us that, for the purpose of this lab, aliens will almost certainly be transmitting on frequencies between 50KHz and 150KHz, so if you see some power outside of this band it, it is likely noise or interference and is safe to ignore.

You can then decode the signal:

```
$./amdemod -c 100000 sig.bin
```

Here, the "100000" is the center of the "possible aliens" band you detected using `band_scan`. `amdemod` will produce the output file `sig.out`. You can then convert `sig.out` to a WAV file:

```
$./bin2wav sig.out
```

This will produce the file `out.wav`, which you can play to hear what your alien sounds like.

Your parallel implementation of band_scan will look similar:

```
$ ./p_band_scan
usage: p_band_scan text|bin|mmap signal_file Fs filter_order num_bands
                   num_threads num_processors
```

The two additional arguments denote the number of threads and the number of processors you are to use (processors 0 to num_processors-1). You should round-robin your threads over the processors as a starting point. We'll give you an example `p_band_scan` binary later in the class, and you'll find it on the lab dashboard (more later).

The last line of the output ("`POSSIBLE ALIENS 80000…`") is critical – it is how the testing and grading system finds your answer. Your program needs to match the format that `band_scan` uses exactly.

When you are ready to compare your implementation to others, you can use our evaluation script, like this (don't forget to include your secret with the –k flag):

```
$ ./seti-eval –t team_name -n netid1_netid2 –s ./p_band_scan –k secret
```

This will update the following dashboard:

http://murphy.wot.eecs.northwestern.edu/~cs213/setilab-dashboard.html

Only team names will appear on the dashboard. Your entry will show up with additional information:

- Whether the result is correct
- A WAV file you can click on to hear result
- An analysis of how fast your program was, reported as a scalability graph (more information later)

Note that the evaluation process may take some time—we may run your program with several different signals, filter orders, numbers of bands, numbers of threads, and numbers of processors to produce the report.

The dashboard will also contain this information for our baseline parallel version, labeled "Da TAs".

You can submit as many times as you want and try out different implementations to see just how fast you can make this process go.

When you are ready to hand in your work for grading, run

```
$ ~cs213/HANDOUT/seti-handin –t team_name –n netid1_netid2 –k secret
```

This will take a snapshot of our your directory, as well as run the seti-eval program. Only source files (.c and .h) and Makefiles will be copied, not binaries.

Signal processing in this lab

For the purposes of this lab, a signal is an N element array of double precision floating point numbers, where each number indicates the level of the signal (the intensity of the induced current in the antenna) at a given point in time. A signal like this is more specifically termed a discrete-time signal (from electrical engineering), and is also known as a time series (from statistics). Our discrete-time signals are periodically sampled, meaning that the time between one sample and the next is fixed. The inverse of this sample time is called the sample rate, and it is the "`Fs`" in the above. To make this more concrete, suppose we have a sequence of numbers like this in the signal file:

```
0.03
0.50
-0.2
-0.9
-0.6
0.3
```

Let's suppose that the first value (0.03) arrived at time 0, and further suppose the sample rate (`Fs`) is 1000 per second (or 1000 Hz, or 1 KHz). This means the second value (0.50) arrived at time 0+1/1000 = 0.001 = 1 ms, the next (-0.2) at 2 ms, and so on.

A neat fact of signal processing is that any signal can be represented as a sum of shifted, amplified sine waves at specific frequencies (keyword: Fourier). We can think of the signal as being composed of different "amounts" (amplitudes) of these sine waves, which is essentially what the `band_scan` program computes. Each *band* consists of a group of sine waves. For reasons not important here (keyword: Nyquist), the highest frequency such sine wave we can find is half the frequency of the sample rate, so 500 Hz in the above example.

Now you can better understand what `band_scan` is doing, since you know the `Fs` that was used (400 KHz), and that resulted in being able to look from 0 Hz to 200 KHz. We asked it to look at the sine waves in that range in 10 bins or bands (0 to 20 KHz, 20 to 40 KHz, and so on). It then found a lot of power (meaning high amplitude sines) around 100 KHz.

`band_scan` does its work by applying band pass filters. A band pass filter passes through sine waves that are in a range of frequencies (the band), and rejects others.

If you look at `band_scan.c`, you'll see that for each band of interest, it first generates a filter design. This form of filter is the simplest variety – it looks like another signal in that it's just an array of M doubles, where M=`filter_order`+1. The bigger M (`filter_order`) is, the better the filter is, but the more expensive it is to use. Unless M is truly huge, the time spent in generating the filter is negligible.

Once the filter for the band has been generated, `band_scan.c` next applies it to the signal using a small function called `convolve()`. `convolve()` is the function that does the heavy lifting (keyword: convolution). You can easily read the function's code to see what it does, but here is the challenge: if the filter is of size M and the signal of since N, then `convolve()` does O(N*M) work. Once `convolve()` is finished, the power in the output signal is summed up. This takes O(N) work. Once all the bands are completed, the band-sums are evaluated using thresholds to see if there is something interesting to be found. This takes negligible work. Therefore, if you are asked to use B bands, the whole program does O(B*N*M) work.

Parallelizing the signal processing

Your goal is to do the O(B*N*M) work in less than O(B*N*M) time. Ideally, if you have P processors, you would be able to do it in O(B*N*M/P) time. You may be able to achieve this in some cases. For different values of B, N, M, and P, you may find there are different challenges to achieving high performance. For example, you may be bottlenecked by I/O, just reading/writing the files, or by the CPU, or by the memory system.

Parallelism requires not just that you have the ability to do multiple things at once, but also that no intrinsic ordering or dependencies in the algorithm are violated. If you violate them, the result may be incorrect. Correct and slow is better than fast and wrong, so be careful. So, you are looking at the work to find out what chunks you can correctly do together.

There are at least three independent ways to parallelize `band_scan` that we can think of, plus the algorithm could be changed (see extra credit). You are welcome to try out any approach you think might be interesting. ***The one approach you definitely will want to try first, however, is simply to execute bands simultaneously, that is, to parallelize across the bands. That should be sufficient for achieving the basic learning and performance goals of the lab.***

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 `band_scan` would look at the execution time as a function of (1) the number of processors (P), (2) the signal size (N), and (3) the size of filter (M). Another useful view is called a *speedup curve*, where we fix the problem size (N and M), and vary the number of processors, plotting time-with-1-processor / time-with-P-processors as a function of P. This is the view we will use for reporting results on the web page, and for grading.

Our class server has two processors, each of which has 6 cores, and each core has two hardware threads.[2] For the purposes of this section, *hardware thread* means processor. This means that it can effectively do 2*6*2 = 24 things at once, so we would not expect speedup beyond P=24.

[2] The use of the terms processor, core, hardware thread, software thread (pthread), 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), although each processor can access memory *near* it faster. 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

Two other ways you might find it possible to get better raw performance and/or better speedup include:

- Parallelizing the individual convolutions themselves. If you look at the convolve() function, you'll see that it does O(M) work for each of the N output data points. You could work on those data points in parallel instead of, or in addition to working in parallel across the bands.
- Exploring the impact of different compiler optimizations to increase the performance of the convolve() function. For example, you could try loop unrolling and strip-mining. Even within a single thread, the processor is highly parallel, and many compiler optimizations exist for increasing the "instruction level parallelism" that can take advantage of that.

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

You will need to read the binary input signal file. Although code is provided in `signal.c` to do this, you may want to review it to understand what it's doing, and to better choose between the three techniques offered there. The code supports text-based files using the C stdio library (available wherever C is available), regular Unix I/O, and memory mapped Unix I/O. 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.

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 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*.

---

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.

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 signal). Their writes to the output signal 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.

The `parallel-sum-ex.c` example shows how to use pthreads to sum up an array of doubles in parallel.

Measuring time, performance, and resource usage

The lab code `timing.[ch]` includes three timing tools. The `band_scan.c` code uses them to measure its own activity. You can use them to find out what the bottlenecks are in your code.

The first timing tool, `get_seconds()`, is measuring the passage of real time using the Unix `gettimeofday()` system call. This can be used arbitrarily in any thread since there is exactly one time across the whole system.

The second timing tool, `get_cycle_count()`, is measuring the passage of real time using the processor cycle counter. This is the most accurate measurement, but it's important to note that each core has its own cycle counter, which can make for confusion if a thread migrates from one core to another.

The third timing tool, `get_resources()`, measures resource usage, including time spent using the processor. You can measure the resource usage of your entire process, and of individual threads. This mechanism gives highly detailed information, but the resolution is much lower than the other timing tools.

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 handout code to get a good sense of how things work. Make sure you try both the "alien.sig" and "noalien.sig" signals, and to get your own signal using `siggen`.
2. Read through `band_scan.c` and `filter.c`, focusing specifically on analyze_signal() and convolve(), which are the functions which do the work.
3. Understand the `pthread-ex.c` and `parallel-sum-ex.c` code. Start by just compiling it and running it, and then look at it more carefully. `parallel-sum-ex.c` shows how to sum up the elements of an array in parallel.
4. Develop a strategy to parallelize the band_scan processing. Read the section on this carefully.
5. Implement your design of p_band_scan using pthreads. It must be possible to specify how many threads to use and how many processors to use at run-time.
6. Test your program to make sure it is correct (compare against the sequential `band_scan` program).
7. When it's correct, start participating in the competition using the `seti-eval` script and the web site.
8. Instrument your program with the timing and resource measurement tools so that it reports the time taken to read the signal, read the kernel, do the analysis, as well as the total time it takes.
9. Try to enhance performance based on your instrumentation and other ideas – you can get extra credit if your performance exceeds that of our reference implementation.
10. Do some other extra credit if you have the time, and it looks interesting.

Grading criteria

Your lab will be graded based on three criteria

- 30% Correctness – it should give the same answers as the sequential `band_scan` program.
- 60% Performance – it should perform similarly to our simple parallel implementation (which will be on the dashboard for comparison)
- 10% Code quality – your program should be coherent

For performance, we will run your program on the class server under a controlled environment with a given problem size and a varying number of processors (one thread per processor). This will produce a graph with a curve showing execution time as a function of the number of processors. As long as your curve is within 10% of our simple implementation's curve, you will get full performance credit.

If your implementation is slower, you will lose 10% of the performance points for every 25% it is slower.

Note that our implementation is very simple, and should be easy to match or beat. ***Furthermore, it is important to note that this lab is a beta test (we haven't offered it before), so we are still calibrating the grading criteria. If our criteria turn out to be overly difficult, we will adjust them. Please give us feedback.***

Extra credit

If your implementation is faster than our baseline parallel implementation (whose performance you'll be able to see on the web site), you'll receive extra credit, up to a maximum of 20% more for an implementation that's twice as fast as ours.

If this lab is easy for you, and you'd like more of a challenge, we'd be happy to talk to you about some of the following options:

- Using the Intel/AMD SSE vector instructions or other special instructions to get parallelism within a single thread of execution. Vector instructions operate on lots of data at once.
- Implement convolution through FFT. 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 FFT transform.
- Implement the band_scan as a parallel filter bank.