CS 213, Spring 2026
SETI Lab: Quickly Searching for Extraterrestrial Life

Contents

1 Introduction **2**

2 Logistics **3**

2.1 Handout Instructions 3

2.2 Handin Instructions 5

2.3 Evaluation 6

3 Lab Breakdown **7**

3.1 Task: Getting Extraterrestrial Signals 7

3.2 Task: Understand Sequential Band Scan 8

3.3 Task: Build and Evaluate Parallel Band Scan 9

3.4 `seti-perf`, `seti-run`: Similarities and Differences 10

4 Background Knowledge **11**

4.1 Signal Processing 11

4.2 Regular Unix I/O and Memory-Mapped I/O 15

4.3 Pthreads and Processor Affinity 16

4.4 Measuring Time, Performance, and Resource Usage 16

5 Queuing `seti-run` Jobs **17**

6 Advice **18**

7 Testing and Debugging Your Code **19**

8 Extra Credit **19**

1 Introduction

The Search for Extraterrestrial Intelligence (SETI) is a long-standing international effort to try to find alien civilizations (yes, little green men) through their radio wave (and more recently, optical) 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.[1] We make full use of this spectrum to communicate with one another. One of the basic (and fairly naïve) assumptions of early SETI efforts was that this would hold true for other intelligent civilizations. They may, in fact, communicate using technology built on wild physics that we don't yet understand. But for now, we'll have to see what we can uncover with radio waves.

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*[2] 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 many human signals, and a lot of noise from all kinds of natural radio sources (e.g. the supermassive black hole at the center of our galaxy).

The signal data from the radio telescopes is analyzed with computer programs that use *signal processing* algorithms, which sift through the data looking for artificial signals that do not come from our own planet.[3] This is a huge computational (and conceptual) challenge, and a significant breakthrough came with the SETI@HOME project[4], which allowed ordinary people to volunteer their computers to participate in a large-scale, distributed version of these programs. At its peak, there were over 1.3 million registered users whose computers contributed 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 your computer execute the computation as fast as it can[5]. You will write a program that, given a raw signal from a broadband receiver, will try to hunt for signs of intelligence[6] within it, specifically some kind of message. Your program will determine whether a given *frequency band* has unusually high power, and if so, which band it is in. In addition, your program will use another program that we provide to determine whether that high-power signal actually appears to come from an *intelligent* source. For fun, we give you some other programs too; an *AM demodulator* and an audio converter, which given the signal and the band, will convert the signal to audio format and let you hear it. The AM demodulator is basically a simple 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

[1] Perhaps recently discovered extrasolar objects like 1I/'Oumuamua, 2I/Borisov, and 3I/ATLAS are space probes homing in on our radio signals like a certain Harvard astronomer has conjectured. And what are those crazy UAPs, anyway? And what is/was AATIP? Where is my tin foil hat?

[2] They're broadband receivers, but we actually search for *narrow-band* signals in the analysis stage.

[3] Seriously! And in the 1970s, the Big Ear telescope may have found one, the so-called Wow! Signal.

[4] `http://setiathome.berkeley.edu/`

[5] This kind of task, i.e. taking an existing, computationally intensive program, and making it faster with smart programming, is referred to as *performance optimization*.

[6] What would actually constitute *intelligent* information content is a topic of ongoing research!

of the lab are:

- to get you to think about what goes into making a simple parallel algorithm,
- to expose you to low-level parallel programming on a shared memory machine using pthreads,
- to expose you to the Unix I/O programming interface, 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*.

The evaluation of this lab will be done using our small cluster of machines, `amdahl-{1,2,3,4}`. Each of these machines has an Intel Phi Knight's Landing processor which has 64 cores with AVX512 vector units, and 256 hardware threads[7]. Unlike other machines you may use in the class, you will access these machines by submitting a *batch job* through a *queuing system*. For the duration of your job (or until it times out), you will "own" one of the machines. This model, called *space-sharing*, is how a supercomputing resource is typically used, and it's a bit different from the *time-sharing* model used on interactive servers like moore or brooks (or your personal computer).

2 Logistics

This lab can be done in teams of two people.

2.1 Handout Instructions

For this lab, you will need to connect to `moore.wot.eecs.northwestern.edu` via SSH or your tool of choice. You will need the file `setilab-handout.tar`, which you will find in the `~cs213lab/HANDOUT/` directory on the class servers.

Expect limited setup help from course staff if you choose to use tools outside of SSH (such as VSCode).

We highly recommend working on `moore`, as it is available worldwide and has all the tools you will need already installed.[8] **You should test that your code works on `moore` prior to submitting.**

You will need a (protected) directory on a Linux machine in which to do your work. You can create a protected[9] directory like this:

```
unix> cd ~/
unix> mkdir mysetilab
unix> chmod 700 mysetilab
```

[7] Chips and Cheese has an excellent post about these CPUs.

[8] You can generally also work on `brooks`, but we are still having teething pains in getting job submission to work from it. If you work on brooks, it is safest to submit from moore.

[9] The `chmod` command will set things so that only the owner of the directory can read, write, or `cd` into it.

```
unix> cd mysetilab
unix> tar xvf ~cs213lab/HANDOUT/setilab-handout.tar
```

This will cause a number of files to be unpacked in the directory. Your unpacked `setilab-handout.tar` file will contain at least the following files:

- `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.

  `analyze_signal` this program looks for "information content" of your signal, i.e. language.

- 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

- Testing Scripts

  `scripts/seti-run` — allows you to submit your program for testing and evaluation. Evaluation will let you compare it to others via the web site.

  `scripts/seti-look` — allows you to look at the job queue for both your submission and see where you are in line.

  `scripts/seti-cancel` — allows you to cancel one or more of your jobs, either running or pending.

Please be sure to read the `README` file.

We **strongly** recommend that you turn this into a git repository **before** you begin. Debugging parallel code is notoriously difficult, and being able to quickly rollback to a known-good version of your code makes debugging easier.

2.2 Handin Instructions

You will submit your solution to Moore using the `seti-run` script we provide. You may submit as many times as you like; there is no submission limit.

> Your `Makefile` rule to build `p_band_scan` **must** be called `p_band_scan`. See the included `Makefile` for an example of such a rule for the sequential implementation of `band_scan`. There is a rule for a `p_band_scan` target already written for you, please do not rename it. You may (and should) modify the actions inside of the rule though. You may find other hints in the comments in the Makefile.

> Your code will be graded after the assignment is due, to ensure that no other group's solution interferes with yours. To submit, run the following command on `moore` from the directory containing your lab files, replacing all values inside the angle brackets (< and >) **including the brackets** with your values:
>
> ```
> setilab-handout> ./scripts/seti-run -n <netid1_netid2> --eval \
>                                     -t <teamname> -k <secret-key> \
>                                     -a <alien-id>
> ```

For example, a team named `The Fighting Mongooses`[10] whose team members have netIDs `xxx9090` and `xxx434`, whose secret key is `52c83a...`, and believes signal `2` is the one with the aliens would run the evaluation command below:

```
setilab-handout> ./scripts/seti-run -n xxx9090_xxx434 --eval \
                                    -t "The Fighting Mongooses" \
                                    -k 52c83aabbae7b1b8578ae19080b0167c \
                                    -a 2
```

> You can use spaces in your teamname, but you **MUST** quote them like shown above. If you do not, then many things will break.

When submitting, an evaluation run will happen and update the dashboard website as well at `http://moore.wot.eecs.northwestern.edu/~cs213lab/setilab-dashboard.html`

NOTE: This is *not* your final grade! The website is meant to estimate where your performance is and encourage you to compete with other students to make the fastest implementation possible.

[10] `https://www.imdb.com/title/tt0757485/characters/nm0005408`

If you and/or your teammate (if you have one) submit multiple evaluation jobs back-to-back, then the **last one to finish** will end up on the leaderboard! This means that if a really fast evaluation is submitted after a very slow one and the fast one finishes first, the slow one will "win" and end up on the leaderboard. You will still get the performance results from all of your runs in the logs in your local `seti-run-output/` directory.

2.3 Evaluation

In this lab, you will develop the program `p_band_scan.c`, which is a parallel version of `band_scan.c`. `p_band_scan.c` should behave identically to `band_scan.c`, except that it will allow you to vary the number of processors used to compute the solution, making the time to solution faster. Your `p_band_scan.c` must be correct and it should be performance competitive with our implementation. **You may not use OpenMP for your implementation.**

Your `p_band_scan.c` is not allowed to use OpenMP.[11]
You must use pthreads.

Your lab will be graded based on three criteria:

30% Correctness — `p_band_scan.c` should give the same answers as the sequential `band_scan` program, and you should identify the correct signal as containing the alien message.

60% Performance — `p_band_scan.c` should perform similarly to our simple parallel implementation. This is is the "TAs" curve in the performance graph shown on the dashboard.

10% Code quality — your program should be coherent

For performance grading, we will use a performance graph, like the one seen in the dashboard. As long as your curve is within 10% of our simple implementation's curve, you will get full performance credit. However, you will only receive any points for performance if your solution is correct.

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

> **Please note that our implementation is pretty simple and should therefore be easy to match or beat. It is compiled with optimization, however.**

For this lab, you can use any modern Linux machine (for example, Wilkinson Lab machines, Moore, Brooks, your own Linux machine or VM, etc) for development and local testing. You **must** use `moore` for submitting jobs to the amdahl machines, **including for hand-in**.

We will use the amdahl machines (`amdahl-[1-4]`) for the performance and correctness parts of your evaluation.

[11] OpenMP is an extension to C/C++ and other languages that makes writing some kinds of threaded parallel programs very easy. You are welcome to play with OpenMP, and it is a useful thing to know, but we want you to learn using "raw" threads in this lab.

You must NOT ssh to the `amdahl` machines!

3 Lab Breakdown

We now describe the steps you should take.

3.1 Task: Getting Extraterrestrial Signals

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.

Whereas SETI@Home looks for signals across a 2.5 MHz[12]-wide frequency band, we are giving you signals across just a 200 kHz frequency band, to make things more tractable. The signals comprise roughly 30 seconds of data. Typically, these radio signals would be subject to the Earth's rotation. However, to simplify things for you, we will ignore this effect. You can assume that your signal was recorded while the radio telescope was stationary and pointing at a fixed patch of sky.[13]

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

```
setilab-handout> ./siggen -n netid1_netid2 -t "teamname"
```

> Please do not run `siggen` multiple times, unless you absolutely need to change an error you made in your netids! Unlike bomblab and attacklab, you can submit a failing program multiple times with **no** penalty! Do **not** make up fake netids to get "practice" signals.

This will take some time, and it will give you several, custom virtual signals. Only one of these will contain an alien message. Other signals may just contain local interference or old radio transmissions. Each signal will be in `sig-N.bin`, where N will go from 0 to however many signals we decide to give you. You will also receive a file named `secret` that contains a secret key that will be used during submission for authenticating you to our server. `siggen` will also create the `netids` and `teamname` files for you, which will serve as *reminders* of what you chose.

None of `seti-run`'s flags accept files! You must provide the values for these flags as part of the command yourself!

You *must* remember the **exact** netID sequence you used and the **exact** team name you used when you requested the signals. We recommend you put them in a file so you remember them for the future.

[12] The choice of this frequency band is not arbitrary. It's centered around the *Hydrogen line*.

[13] Some astronomers now think this may actually be the right way to go — more telescopes, each dedicated to its own, fixed patch of the sky. The energy required to transmit a signal of significant power over several light years may be too much for an "always on" beacon that a directional sweep is intended to pick up.

3.2 Task: Understand Sequential Band Scan

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:

```
setilab-handout> make
setilab-handout> ./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. F_s, `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 F_s will always be `400000` (400 kHz).

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

```
setilab-handout> ./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)
```

This result means 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 `~cs213lab/HANDOUT/noalien.sig`, you'll see what a more boring signal looks like. If you run against the example file `~cs213lab/HANDOUT/alien_msg.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 50 kHz and 150 kHz, so if you see some power outside of this band it, it is likely noise or interference and is safe to ignore.**

If you've detected a large amount of power in a narrow band, you just might have an alien message! To further verify this, you can see if it has something that might be akin to language by using this program:

```
setilab-handout> ./analyze_signal -p sig.bin
```

The `-p` option tells it to print out high-powered pulse lengths it found in the signal, and is optional. Do you notice anything special about the pulse lengths you find? If it found something very interesting, it will output the following:

```
The information content of this signal appears to be very high!
```

At this point we can pretty much assume we have an alien message on our hands. Make sure to take note of which signal this is; i.e. if this was `sig-0.bin` and it had a message, you will need to provide `0` to the handin program. You can then decode the signal for audio playback:

```
setilab-handout> ./amdemod -c 100000 sig-0.bin
```

Here, the "100000" is the center of the "possible aliens" band you detected using `band_scan`. `amdemod` will produce the output file `sig-0.out`. You can then convert `sig-0.out` to a `WAV` file:

```
setilab-handout> ./bin2wav sig-0.out
```

This will produce the file `out.wav` (it takes a bit of time), which you can play to hear what your alien sounds like.

3.3 Task: Build and Evaluate Parallel Band Scan

Your parallel implementation of `band_scan` will look like this:

```
setilab-handout> ./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. The 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 output format that `band_scan` uses exactly.**

You can run `p_band_scan` directly to debug your work on the machine you're logged into (e.g., `moore`). If you want to try a simple performance test, you can try the `seti-perf` script that is part of the handout. This will do a simple timing run *on your local machine*, comparing with our implementation. This may also be helpful for debugging.

To run, evaluate, and handin your code on the `amdahl` machines, you will use the `scripts/seti-run` command. This command, along with `scripts/seti-look` and `scripts/seti-cancel` form the basis of our queuing system (more about this in Section 5). **These commands will only work correctly only on `moore`!**

When you would like to do a **test** performance run on an `amdahl` machine, you will run this command:

```
setilab-handout> ./scripts/seti-run -n <netid1_netid2> --test
```

This will basically run `seti-perf`, but as a job on an `amdahl` machine. If you look in `./seti-run-output/`, you will see the results of your run after your job finishes. It should run relatively quickly, compared to a full evaluation which can take some time.

When you are ready to do an evaluation run to compare your implementation to others and get it onto the dashboard, you can use `seti-run` like this:

```
setilab-handout> ./scripts/seti-run -n <netid1_netid2> --eval \
                                    -t <team_name> -k <secret> -a 0
```

Note the addition of the `--eval` flag. Also, don't forget to include your secret with the `-k` flag and the alien you found with the `-a` flag. Here, `-a 0` means "I found the alien in signal number 0."

This will take a snapshot of your directory, hand it in to the instructors, evaluate your implementation, and update the dashboard website. Only source files (`.c` and `.h`) and `Makefiles` will be copied, not binaries.

Similar to the performance run, it will give you output once the job is finished. However, it will run a different set of performance tests which can take quite a bit longer. It will also update the dashboard, which you can see at http://moore.wot.eecs.northwestern.edu/~cs213lab/setilab-dashboard.html

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

- An analysis of roughly how fast your program was, reported as a scalability graph (more information in Section 4.1.2) that compares your program with our baseline implementation.
- A ranking of your performance compared to us (we are 1.0) and to other participants.
- A note of whether you found the alien or not—you need to not only select the correct signal number, but the carrier your code finds needs to be correct.

Note that the evaluation process will take some time (approximately 15-20 minutes). We will run your program with several different signals, filter orders, numbers of bands, numbers of threads, and numbers of processors to produce the report. 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've called `seti-run`, a log will be generated within `seti-run-output/` which contains the results from your tests or evaluations by job ID. These are updated in real time as your code runs. If your code had an error, those details will also be in this output.

3.4 `seti-perf`, `seti-run`: Similarities and Differences

There are several scripts included in the handout, many of which have very similar names (and seemingly purposes). However, there are some key differences between all of them that are critical to remember. All of the commands that you will end up running will be one of the following:

- `seti-perf` runs on `moore`, `brooks`, or other machines. `seti-perf` is a small script that you can use to help verify the correctness of your parallelized implementation. You should **not** trust any performance numbers you get from this script, but it useful for seeing whether you've made progress or not.
- `seti-run --test` tests on an Amdahl. When you run `seti-run` with `--test`, you create a **test** job. A test job runs a slightly modified version of `seti-perf` on one of the amdahl machines. You can use this to quickly gauge your performance.
- `seti-run --eval` evaluates on an Amdahl. This is an **evaluation** job. It builds a copy of your code, **hands it in** to the instructors, **and** evaluates it to place on the leaderboard. If you want the teaching group to grade your code, you **must** execute a `--eval` job!

We gave you `seti-perf` to run on Moore, Brooks, etc, locally to let you quickly test that your parallel implementation produces correct behavior and results. We want you to use `seti-run`'s test jobs on the Amdahls so you can quickly get feedback on the machines you are evaluating on. Then the evaluation job does more complete testing to build the leaderboard.

The thing you have to keep in mind is that when you run on Moore or Brooks, you are running on a completely different CPU microarchitecture. Both Moore and the Amdahls use x86_64, but they are wildly different implementations of the same ISA. Further, Moore runs at significantly higher clock frequencies ($2\,\mathrm{GHz}$ or higher *on average*) compared to the Amdahls ($1.5\,\mathrm{GHz}$ *maximum*). All of this (along with many other factors) contributes to Moore's significantly better single-core/single hardware-thread performance compared to any of the Amdahl machines. So you will probably see better performance on low thread counts on Moore than you do on any of the Amdahl machines. However, the Amdahls have 250+ cores, which you will take advantage of to run faster than Moore.

You **CANNOT** directly compare the results of `seti-perf` on Moore or Brooks with the results of `seti-perf` on an Amdahl machine (by running a test `seti-run` job). Moore/Brooks and the Amdahl machines are *significantly* different. To name just a few reasons: Amdahls have 4-way hyperthreading, just $\sim 80\,\mathrm{GiB}$ of RAM, and a very different implementation of x86.

4 Background Knowledge

4.1 Signal Processing

To understand this lab, you need to understand some basics of signal processing and how you might parallelize them.

4.1.1 Signal Processing Basics

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 F_s 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 (F_s) is 1000 per second (or $1000\,\mathrm{Hz}$, or $1\,\mathrm{kHz}$). This means the second value (`0.50`) arrived at time $0 + \frac{1}{1000} = 0.001 = 1\,\mathrm{ms}$, the next ($-$`0.2`) at $2\,\mathrm{ms}$, 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 of such a sine wave we can find is half the frequency of the sample rate, so $500\,\mathrm{Hz}$ in the above example.

Now you can better understand what `band_scan` is doing, since you know the F_s that was used ($400\,\mathrm{kHz}$), and that resulted in being able to look from $0\,\mathrm{Hz}$ to $200\,\mathrm{kHz}$. In the earlier example, we asked `band_scan` to look at the sine waves in that range in 10 bins or bands ($0\,\mathrm{kHz}$ to $20\,\mathrm{kHz}$, $20\,\mathrm{kHz}$ to $40\,\mathrm{kHz}$, and so on). It then found a lot of power (meaning high amplitude sines) around $100\,\mathrm{kHz}$.

`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_and_compute_power()`. `convolve_and_compute_power()` 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 size N, then `convolve_and_compute_power()` does $O(N*M)$ work. As `convolve_and_compute_power()` executes, it also sums up the power in the selected band of the output signal, which is an additional $O(N)$ 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.

4.1.2 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 $O\left(\frac{B*N*M}{P}\right)$ 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 the Extra Credit, Section 8). 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
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. Finally, we can simply plot 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.

Each `amdahl` machine that you will be evaluated on has one processor chip that contains 64 cores, and each core has four hardware threads. For the purposes of this section, hardware thread means processor. This means that it can effectively do $64 * 4 = 256$ things at once, so we would not expect speedup beyond $P = 256$.

> The use of the terms processor, core, hardware thread, software thread (pthread), and process can be a bit confusing.
>
> A machine may have one or more processor. 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 core, but that engine has numerous functional units (adders, multipliers, etc). The purpose of multiple hardware threads per core is basically to keep that engine busy by feeding it work from multiple sources.

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 the OS to map a software thread to a 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.

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_and_compute_power()` 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_and_compute_power()` function.

Note that there is also Extra Credit that involves considering another algorithm.

4.1.3 Carrier Frequencies and Modulation

Each of the signal files has a single signal in it (across a large frequency range). That large range is not what the aliens used in their original signal; they used a much smaller frequency range, we just collected signals in that large range. First we need to remove all the boring relatively low-power noise in the signal, so we filter it first. That filtered signal (which is now the interesting one) still covers a certain bandwidth. To fully answer why the signal covers a certain bandwidth, we need to make a digression into some signal modulation topics.

Our hypothetical aliens are using AM (Amplitude Modulation) radio. The idea of this is that we take a high frequency sine wave (at the "carrier frequency") and then increase and decrease its size (amplitude) in accordance with a lower frequency message signal (such as the alien's voice). The high frequency sine wave

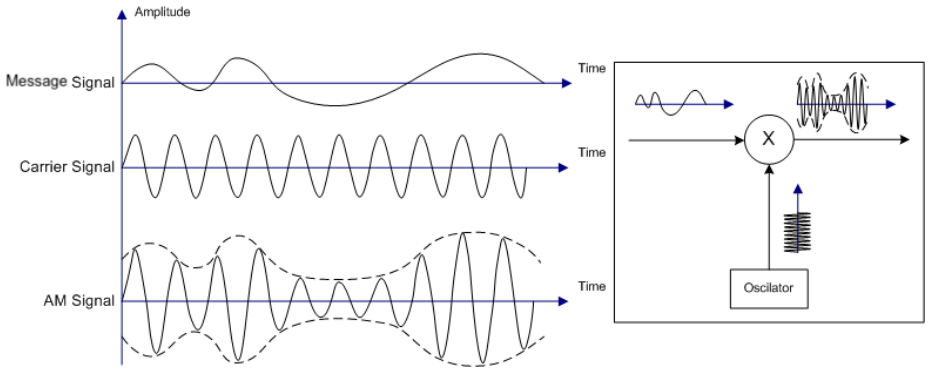


Figure 1: Amplitude Modulation (AM Radio) (Credits to Wikipedia)

is multiplied by the lower frequency message signal, and the resulting wave "carries" the lower frequency message signal as shown in Figure 1.

Notice how the original message signal appears two times in the AM signal, forming the "envelope" around the carrier frequency sine wave. The AM signal is then radiated from an antenna as radio waves. The receiver of this AM signal "tunes" their radio to the carrier frequency (your code will find the frequency to tune to), removes the "lower half" of the AM signal (all the negative values), and then applies a low-pass filter that removes the carrier frequency sine wave. At this point, all that is left is the top half of the envelope, which is the original message signal. This process is called demodulation.

AM is among the easiest-to-understand ways to modulate carrier frequency sine waves with message signals, but it suffers from a bunch of issues. When the alien is speaking quietly the AM signal is very weak (it has low power). This makes it easy for an AM radio signal to be "washed out" by radio frequency noise.

What if we could modulate the frequency of our carrier frequency sine wave instead of its amplitude? This would make the resulting signal always have the same high power, overwhelming noise. Frequency Modulation (FM) is the name of this technique, and is the basis for FM radio.

Wikipedia has a great GIF that illustrates both amplitude and frequency modulation.

4.2 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 of binary files, and memory mapped Unix I/O of binary 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.

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 `exec`-ing 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.

4.3 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 and were also covered in lecture. 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 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.

4.4 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 sys-

tem.

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.

5 Queuing `seti-run` Jobs

SETI Lab is a computationally-intensive assignment. You will be evaluated on one of the Amdahl machines. Each of these machines has an Intel Xeon Phi 7210 "Knight's Landing" processor.[14] This processor is unusual in that runs really slowly (just 1.3 GHz), but has 64 physical cores, and 4 hyperthreads per core for a total of 256 logical cores (hyperthreads)! Further, each of these physical cores has a full AVX-512[15] vector unit! If you write good parallel code, you will be able to scale to hundreds of logical CPU cores, using the entire machine! You can further speed up your code using the vector unit aggressively. The vector unit can potentially do the kind of convolution used in this lab eight steps at time, instead of one step at a time.

We have implemented a queuing system to make sure no two people/groups try to get the same machine simultaneously. If two groups did get the same machine, then your performance results would be invalid. All queuing operations are hidden behind the `seti-run` script, so you do not need to worry about this too much.[16]

We have set hard limits on the number of jobs you may be running simultaneously and the number of jobs you may have in the queue. Notably:

- You may have a maximum of 2 jobs running simultaneously.
- You may have a maximum of 5 jobs total (running and in the queue).
- Your job will run for a maximum of 30 minutes, as measured by the wall time.[17]

There are three commands you can use:

- `scripts/seti-run` will let you queue jobs for performance testing and evaluation. This command is described in detail earlier in the handout. Note that the command might refuse your job

[14] Intel names microarchitectures after landmarks in the Hillsboro, Oregon area where they are headquartered.

[15] You can find information about AVX-512 on Wikipedia and from Intel themselves.

[16] If you are curious, we are building on top of the Slurm Workload Manager, which is commonly used in supercomputers and clusters.

[17] When writing parallel code, there are multiple ways to measure time. Wall time is the amount of time your entire program has been running, so called because this was the time your job lasted when measured by clocks people hang on walls. CPU time is the amount of time your program has spent executing on the CPU. A parallel program that runs on 2 cores that runs for one wall-minute will have actually run for two CPU-minutes.

because you currently have too many running or pending. If it does accept your job, it will give you a job id (a number) to refer to it.

- `scripts/seti-look` will let you see the jobs that you (and everyone) have submitted using `seti-run` that are either running or have pending. Pending just means the job is waiting to run. Jobs move from pending to running in a first-come-first-served manner, sort of like a checkout line at a grocery store.
- `scripts/seti-cancel` allows you to cancel any of your running or pending jobs. You run the command with a list of job IDs to cancel. The course staff can cancel any student's running or pending jobs.

You can instruct the queuing system to keep the contents of your run after it completes, so you can toy with the generated program and debug it on Moore, without having to submit another job. Just pass either the `-K` (capital K) or the `--keep` flags.

6 Advice

We document a suggested approach to this lab below. There are other ways to work with and solve this problem.

The biggest thing to keep in mind is: **don't panic**!

There is a lot here, which can be overwhelming if you've never been given an existing codebase to work on. Part of what we're trying to do is give you the experience of working within an existing, significant codebase. Remember that what you're doing here is writing `p_band_scan.c`. There is a lot of complexity in this overall system, but you are remember that you are not writing it from scratch — you're just parallelizing this one component of it. Our own reference implementation of `p_band_scan.c` is based on `band_scan.c` and consists of only about 180 additional lines of C. 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_msg.sig` and `no_alien.sig` signals, and to get your own signals using `siggen`.
2. Read through `band_scan.c` and `filter.c`, focusing specifically on the `analyze_signal()` and `convolve_and_compute_power()` functions, which do the main 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 Section 4.1.2 carefully. The simplest approach will work fine.
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. Note that you will need to change the `Makefile`, as documented in Section 4.3. You will find comments in the `Makefile` that describe how to do this. Your `Makefile` rule to build `p_band_scan` **must** be called `p_band_scan`.

6. **Test your program to make sure it is correct** (compare against the sequential `band_scan` program). For basic testing, you can directly run `p_band_scan` and compare against the results from `band_scan`.

7. Start evaluating and enhancing your program for performance. For initial performance testing, use `seti-perf` or `seti-run --test`. Once performance is okay, you can use `seti-run --eval` (which must run on `moore`) to get your result onto the scoreboard. The scoreboard[18] will indicate whether or not your program is correct regardless of its performance.

8. For extra credit, you'll need to go further beyond. 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. 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. See Section 8 for more information about this.

Please start early. Note that there are only four machines for ultimate evaluation, so the queue will start to get long as the deadline approaches.

7 Testing and Debugging Your Code

This lab is implementation heavy, so doing thorough testing is important for ensuring correctness and to help with debugging. We have enabled all warnings on the code, but that does **not** mean that your code will produce the correct results. You **must** test your code to make sure it produces equivalent results to our sequential implementation!

> You are writing **parallel** code, which means that `printf` debugging probably won't be helpful!

You will find that `gdb` is very useful here. You can list all the threads via the `info threads` command, and switch to a specific thread using the `thread <k>` command.

You will not receive any debugging help from course staff if you have not made a good-faith effort to debug through testing.

8 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 10% more for an implementation that's twice as fast as ours. Note that your solution MUST still be correct to receive extra credit. Some ideas on how you could reach better performance:

[18] http://moore.wot.eecs.northwestern.edu/~cs213lab/setilab-dashboard.html

- **Understand where the time goes, ideally with profiling.** You could use a *profiler* (like `gprof`) to figure out where the time goes, or you could review other parts of this document to figure this out.
- **Investigate compiler optimizations**, including those that are specific to the machine you are running on. See if the compiler can improve performance for you. `gcc` has hundreds of optimization features you can enable/disable from the command line. You can also try `clang` and Intel's C compiler, `icc`.
- **Get the compiler to vectorize.** Vector instructions operate on lots of data at once and are key to single-thread performance on this hardware. How can you get your compiler to use vector instructions for your code, especially in the inner loop where the majority of time is spent? There are some challenges in the source code for the inner loop that make it difficult for the compiler to vectorize. Can you find and fix them? You also need to tell the compiler to attempt vectorization (a part of optimization), ideally for the specific hardware you have. You can use `objdump -d` to get a sense of the code the compiler is generating.

There are additional extra credit options as well, given below. We suggest you optimize your performace first, since that is likely to be a far easier place to start.

- **Directly use the Intel/AMD SSE or AVX or AVX512 vector instructions** or other special instructions to get parallelism within a single thread of execution. One way to implement this it to write your code using compiler's vector intrinsics, which are extensions to the C types and operators that directly represent the hardware. For example, the type `__m512d` represents a 512 bit AVX512 register holding 8 `doubles`. Another way to implement this is to write your code directly in assembly (either inline assembly or a separate assembly file—we can give you guidance either way).
- **Implement on a GPU.** If you have an NVIDIA GPU, you can learn about the CUDA language/development environment for programming it. For AMD GPUs, HIP is the language/development environmnet. Hanlon also has an NVIDIA K20C GPU you can play with if you do not have one. You can make the CUDA development environment via:

  ```
  Hanlon> source ~cs213/HANDOUT/cuda_env
  ```

- **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.
- We are open to other ideas. Just reach out.