HPCE CW2 - Streaming parallelism and makefiles

Issued: 2014/01/29

Due: 2014/02/12, 23:59

Last updated: 2014/02/05, 02:11

1 Goals

Performance does not always come from writing code, sometimes it is about the way you make use of existing code and programs. The philosophy of the unix shell is to create many small programs with orthogonal functionality, then to combine them in many ways to provide different high-level functionality. This philosophy also often allows large amounts of parallelism, which can allow surprisingly good scaling over multiple processors.

The overall goals of this coursework are:

- Get experience in using the command line and shell-script, for those who are used to GUI-based programming.
- Some (minimal) experience in automating setup.
- Understand process pipelines, and how they can improve efficiency and enable parallelism.
- Explore the built-in parallelism available in the shell.
- Look at makefiles as a way of creating dependency graphs.
- Use the built-in parallelism of make, and explore it's advantages and limitations.

The goal is not to produce ultra-optimised C code (though if you do, and it works, then that is nice too). Also, I make no apologies for making you go through some slightly circuitous steps on the way to the streaming and parallelism bits later on in the exercise. For some of you it will make you do things you already know how to do, or conflict with the ways that you've done things in personal projects or industry. [1]

[1] "bash? bash! Real programmers use (ksh,asht,csh...) as their shell." Actually, I used

2 Environment Setup

This coursework should be performed in a posix-like environment, so that could be Linux, OS-X, or Cygwin. Particular things it relies on are:

Bash There are many shells (the program that proves the interactive command line), and there are some differences in input syntax. This coursework explicitly targets the Bash shell, as it is installed in most systems (even when it isn't the default).

Make While traditionally used just for building programs, "make" can also be used for the general co-ordination of many programs working together. We will specifically use GNU make.

C++ toolchain Some of the audio processing will be done with programs compiled by you, so you need a command line toolchain setup. It doesn't really matter if it is gcc, clang, or icc, though if you don't have any strong preference, stick with the gcc.

First start a command line terminal – the exact mechanism depends on your OS. You should now be looking at a command line shell, with some sort of blinking text prompt. The first thing to check is that you are in bash, rather than some other shell. Type the command:

```
echo $SHELL
```

This displays the name of the current shell, which should look something like "/bin/bash". If it says something different, you'll need to type:

```
bash
```

which should then drop you into a bash shell (the prompt may change slightly). If it complains that bash can't be found, then you'll need to use your system's package manager to install it.

You should now be in bash, so next make sure that you have "make" installed:

```
make -v
```

If it complains that make can't be found, then you'll need to install it using your package manager. If make is listed as anything other than "GNU Make", then you may need to install the GNU version – however, try following the coursework first, and it may just work ok.

Change to a directory you want to do do your work in, then download the coursework tarball using curl:

```
curl http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/hpce_cw2_spec.tar.gz  \
   -o hpce_cw2_spec.tar.gz
```

lisp as my shell for a while. I wouldn't recommend it, about the only advantage is being able to say "well, I use lisp as my shell" smugly at conferences. Until you find someone else uses APL.

(Note the continuation slash at the end, this is all one command). Again, if curl isn't found, get it through your package manager. If you are trying to get the zip file with a web browser, try again with curl. Once it has finished, check you know where it is (`ls`).

Once you've got the tarball file, choose a working directory, for example "hpce_cw2", and extract the tarball there:

```
mkdir hpce_cw2
cd hpce_cw2
tar -xzf ../hpce_cw2_spec.tar.gz
```

You should now have a number of directories (similar to the first coursework), some of which contain files. This will be your working directory, and your submitted zip should follow the same structure.

3 Part A : Build automation

The point of this section is to set up a framework for later work in audio processing, and to do that we're going to use two tools: `sox` and `lame`. The aim here is to create a single makefile which is able to download and build both packages, creating a known environment no matter what software is currently installed. This may seem a slightly pointless task, but being able to replicate an environment by typing `make` and then going away and getting a coffee can often be useful.

3.1 Building sox

Before automating, you need to build it manually. For lots of open source packages, there are a number of steps for doing this:

- Download the tarball.
- Extract the sources from the tarball.
- Run the "./configure" script, which allows the package to look at your environment and see what is currently installed, and allows the user to specify where the new package will be installed.
- Execute "make" to build all the sources and produce the executable.
- Do "make install" to install the binaries and documentation.

The process is not completely standardised, but for many packages it looks the same.

3.1.1 Download the tarball

We'll use a version of sox that I downloaded from their website, but which will be hosted locally to avoid us hammering their website. As before, we'll use `curl` to download the source package:

```
curl http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/sox-14.4.1.tar.gz \
    -o packages/sox-14.4.1.tar.gz
```

Note that this is following the same pattern as before: the first argument to `curl` is the URL we want to access, and the argument after "`-o`" is the location it should be stored. As with most unix programs, you can use "`--help`" to find out the arguments it takes (for example `curl --help`), or `man` or `info` to get more detailed documentation (`man curl`, or `info curl`). If they exist, the info pages are usually more detailed and better structured (and the man page may just be a stub), though some tools only have man pages.

Here we are specifying that the file should be downloaded to the "packages" directory. Once curl finishes, do `ls packages` to check that you can see the download file there.

Note that this is a distinct step in the process: we started with nothing, executed the curl command, and the output of that command was the tarball.

3.1.2 Unpack the sources

We now need to extract the contents of the tarball. To keep things clean, we'll extract the contents to the directory called "build". This will keep things separated, so that if necessary (e.g. to save space), we can delete the contents of "build" without losing anything.

You should still be at the base directory, but that is not where we want to extract. Do `cd build` to get into the build directory. The general command to extract things is "`tar -xzf` *`some-file`*", where *some-file* is the path to a tarball you want to extract. The "`-xzf`" option specifies what you want to do: "`x`" means eXtract; "`z`" means the tarball is compressed with gzip; and "`f`" means that we are specifying the input tarball using a File path.

The tarball is still in the packages directory, so we'll need to specify a relative path. You may wish to do:

```
ls ..
ls ../packages
ls ../packages/sox-14.4.1.tar.gz
```

just to check you know where you are pointing. Knowing that the relative path from the "build" directory is "../packages/sox-14.4.1.tar.gz", we can now extract the files using:

```
tar -xzf ../packages/sox-14.4.1.tar.gz
```

This will extract a number of files and directories in the current (i.e. "build") directory. Change directory into the new "sox-14.4.1", and have a look around.

In particular, note that there is a file called "configure" which will be what we use next.

This step is now complete, and again we can summarise the step in terms of inputs, commands, and outputs: the input is the tarball; the command is tar; and the output is the extracted files, in particular the file called configure.

3.1.3 Configure the package

The source files for sox are designed to work on multiple unix-like platforms, but to achieve this they have to enable and disable certain features, or specify different `#define`s at compile time. A commonly used approach is to use GNU Autotools, which attempts to make this "easy", and is reasonably portable. [2]

The "configure" file produced by the previous stage is actually a program that we need to run. Make sure that you are in the "build/sox-14.4.1" directory, and do:

```
./configure --help
```

This will show you all the options that you can pass when configuring the package, so there are lots of options affecting what gets included or excluded, and multiple flags that can be passed in.

The only flag that we need to worry about is "`--prefix`", which tells `configure` where it should install the eventual binaries. By default it will install it in a system-wide location, often "/bin" or "/local/bin". However, we are trying to create a local environment that we control, rather than modifying the entire systems. On many systems you also won't be allowed to write to the system-wide directories (certainly in college, and often in an industrial context), so it is critical that you make sure it installs it in a place you control.

We're going to specify the installation directory as "local" in your base directory. From where you are you should be able to do:

```
ls ../../local
```

to check the relative path from where you are, and can then do:

```
./configure --prefix=../../local
```

Well, except... it won't (at least on systems I tried). The error message should tell you what the problem is, and to solve it you can use the following:

1. The `pwd` command prints the current directory.
2. Enclosing a command in back-ticks captures the output of that variable, so for example:

   ```
   X=‘pwd‘
   echo $X
   ```

[2] Although nobody really thinks that autotools are good – they just mostly work and everything else is worse. See for example Poul-Henning Kamp's commentary.

captures the output of `pwd` into the shell variable X, then prints the value of X using echo. (make sure they are back-ticks, usually in the top-left key on the keyboard).

3. You can concatenate strings in the shell, so if you do:

   ```
   X=‘pwd‘
   echo "$X/../../local"
   ```

   it should now print your current directory followed by the relative path to local. However, because it starts with "/", this is now an absolute path, despite the ".." components later on.

4. Collapsing the two steps, you can do:

   ```
   echo "‘pwd‘/../../local"
   ```

   to combine the two.

This leads us to a (rather hacky, but portable) solution:

```
./configure --prefix="‘pwd‘/../../local"
```

You should now see huge amounts of configuration messages go past as it works out what your system looks like.

It is possible *that it may find some errors, for example a missing library. Some of those can be fixed easily, but if you find uncorrectable, switch to a controlled environment like Cygwin. Building the environment should not involve any debugging or difficulty, so don't get caught up trying to debug problems.*

Once configuration has finished, a new file called "Makefile" will have appeared, which is the input to the next stage. Again, notice the pattern: the required input is the configure file; the action is to run that configure file; and the output is the Makefile.

3.1.4 Building the code

We now have a makefile, which contains the list of all files which must be built, as well as the order in which they have to be built. So for example, an object file must be built before a library containing it can be linked. However, for now we can just run the file:

```
make
```

You'll now (hopefully) see a mass of compilation messages going past. To start with, there will be lots of individual C files being compiled, then you'll start to see some linking of libraries and executables. Once this process is finished, the whole thing should be compiled. If you do:

```
ls -la src/sox
```

Then you should be able to see the sox executable, and if you want you can run it with:

```
src/sox
```

However, it is not yet in the location where we want it installed.

Again, there should not be any compilation errors once make starts – if anything strange happens at this stage, you may wish to switch to a clean cygwin install.

Pushing the pedagogical point: in this stage the input file was the Makefile, the process was calling make, and the output is the executable.

At this point we can start to exploit the parallelism of make. Do

```
make clean
```

This is the convention for getting a makefile to remove everything it has built, but *only* the things it has build – all source files will be untouched. You'll see that "src/sox" has disappeared, along with all the object files.

Now do:

```
make -j 4
```

You should see a similar set of compilation messages go past, but you'll see them go by faster, and if you look at a process monitor, you'll see multiple compilation tasks happening at once. If you have 4 or less CPU cores, you'll see most of the cores active most of the time, while if you have more than 4 cores then the cores will all be active but not running at 100%.

This type of process-level parallelism is extremely useful because:

- The parallelism is declarative rather than explicit: internally the makefile does not describe what should execute in parallel, instead it specifies a whole bunch of tasks to perform, and any dependencies between those tasks.
- It is very safe: there is usually no chance of deadlocks or race conditions.
- It allows you to take existing sequential programs with no support for multi-core and scale them over multiple CPUS, without needing to rewrite them.

There are limitations to this approach, but if a make-like solution to describing parallelism can be used, it is usually very efficient in terms of programmer time, and fairly efficient at exploiting multiple cores.

3.1.5 Installing

The final step here is to install the binary to its target location, in our case the "local" directory. To install, just do:

```
make install
```

You'll see various things being copied, and a few other commands being run. The installation directory is "local", so if you do:

```
ls ../../local/
```

You'll see that you have a shiny new "local/bin", "local/include", and so on created, and hopefully if you do:

```
ls ../../local/bin
```

You'll see the sox binary there.

So our final stage took the sox binary in one place, and used the installer to move it to the target location.

If you return to the base folder, you can now run sox as `local/bin/sox`.

3.2 Building lame

Sox is able to play audio, but by default is not able to handle things like mp3s. We'll use lame for these purposes, so we'll also need to follow the same steps.

The version of lame we'll use is lame-3.99.5, and it is available at:

```
http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/lame-3.99.5.tar.gz
```

The steps you should follow will be exactly the same as for sox, but you'll need to change the various names from "sox" to "lame" as you go through.

3.3 Checking it works

Congratulations: you should now have both lame and sox installed in "local/bin". We'll now do some (not very impressive) parallel processing. From your base directory, execute this (it is all one command, you can copy and paste it in) [3]:

```
curl http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/bn9th_m4.mp3 \
    | local/bin/lame -  --mp3input --decode  - \
    | local/bin/sox  - -d
```

Be vary careful about the ends of lines in these commands, as it is easy to accidentally send binary files to the terminal. If that happens, you'll get a screen of weird characters, and your terminal might lockup. If that happens, just kill the terminal and open another one.

The three commands correspond to three stages in a parallel processing pipeline:

1. Curl is downloading a file over http, and sending it to stdout.
2. Lame is reading data from its stdin ("`-`"), treating that input data as mp3 ("`--mp3input`") and decoding it to wav ("`--decode`"), then sending it to stdout (second "`-`").

[3] Source for mp3. Creative commons, I believe.

3. Sox is reading a wav over its stdin ("`-`"), and sending it to the audio output device ("`-d`").

If you look at a process monitor while it is playing (e.g. taskmgr in windows, or top/htop in unix), then you'll see that all three processes are active at the same time. However, the processing requirements are very small.

If you instead try:

```
curl http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/bn9th_m4.mp3 \
    | local/bin/lame -  --mp3input --decode  - \
    | local/bin/lame -  - \
    > tmp/dump.mp3
```

then here we are decoding, then re-encoding (i.e. transcoding). Because we are writing to a file ("`> tmp/dump.mp3`"), there are no limits on playback speeds, so all processes run as fast as they can. In this case you'll find either that the curl process is the limit (if you have slow internet), or more likely that the second lame process starts taking up 100% of CPU. The first lame process will probably take a much smaller, but still noticeable amount of CPU time.

To emphasise this effect, do the following:

```
curl http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/bn9th_m4.mp3 \
    | local/bin/lame --mp3input --decode - - \
    | local/bin/lame - - \
    | local/bin/lame --mp3input --decode - - \
    | local/bin/lame - - \
    > tmp/dump.mp3
```

This is now completely pointless, but if you run it, you will probably see that two of the lame processes are now taking up an entire CPU each. So by connecting streaming processes in this way, we are enabling multiple cores to work together on the same problem.

3.4 Automating the build

You've now built two tools, but there were quite a few steps involved. If you now wanted to do the same experiment on a different machine, you'd have to go through each of the manual processes again. Here we're going to build a makefile which will automate this process.

Note: This automation and the makefile is part of the submission – my tests will initially get your submission to build its environment, before running your code.

We're going to do this by creating a makefile, similar to the makefile you used to build sox and lame, but much simpler. Note that there is a lot of documentation available for make, either on the GNU website, or by typing `info make`.

Makefiles primarily consist of rules, with each rule explaining how to create some target file. For example, we want our build system to eventually create the target executable "local/bin/sox". Each rule consists of three parts:

target The name of the target file (or files) that the rule can build.

dependencies Zero or more files which must already exist before the rule can execute.

commands Zero or more shell commands to execute in order to build the target.

The general format of a rule is:

```
target : dependencies
    command(s)
```

Be aware that the space before the command must be a single tab, not multiple spaces. The makefile will contain multiple rules, each of which describes one step in the process.

In the case of building sox, we used the following rules:

Stage	target	dependencies	command	directory
Download	packages/sox-14.4.1.tar.gz		curl	
Unpack	build/sox-14.4.1/configure	packages/sox-14.4.1.tar.gz	tar	build
Configure	build/sox-14.4.1/Makefile	build/sox-14.4.1/configure	configure	build/sox-14.4.1
Build	build/sox-14.4.1/src/sox	build/sox-14.4.1/Makefile	make	build/sox-14.4.1
Install	local/bin/sox	build/sox-14.4.1/src/sox	make install	build/sox-14.4.1

So each of these corresponds to each of the stages you performed manually (you'll notice I deliberately pointed out the target, commands, and dependencies at the end of each stage). I've also included the directory we were in when the command was executed, as we were occasionally using relative paths.

3.5 Clean the environment

The makefile is going to replicate the steps you performed to install sox and lame, so first you need to delete everything you did manually. So delete:

- The tarballs from "packages".
- The two build directories from "build".
- All the directories from "local".

You should now be back to just what came out of the original coursework tarball.

3.6 Create the makefile

In the base directory of your project, create a text file called "makefile". This should have no extension: windows may secretly add a ".txt" on the end, and lord knows what MacOS does, but you need it to have no extension. You can check it by doing `ls` to see what the real filename is. If some sort of extension has been added, just rename it, .e.g:

```
mv makefile.txt makefile
```

Open the text file in a text editor, and add the variable and rule:

```
SRC_URL = http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2

packages/sox-14.4.1.tar.gz :
    curl $(SRC_URL)/sox-14.4.1.tar.gz -o packages/sox-14.4.1.tar.gz
```

This first defines a convenience variable called `SRC_URL`, which describes the web address, and then defines the actual rule: the target is "packages/sox-14.4.1.tar.gz"; there are no dependencies (as it comes from the network); and the command just executes curl. Where the variable `SRC_URL` is referenced using $(SRC_URL) it will expand into the full path.

Note-1: the white-space before "`curl`" is a tab character. Do not use four spaces, or it won't work.

Note-2: if you are in windows (cygwin), you need to make sure you use unix line-endings in your makefile. If necessary, force your text editor to use that mode.

If you go back to your command line (with the same directory as the makefile), and execute:

```
make packages/sox-14.4.1.tar.gz
```

it should execute the rule, and download the package. The argument to make specifies the target you want to build, and it will execute any rules necessary in order to build that target.

3.6.1 Start adding rules

So now the makefile can download, and we need to unpack the file. Add this rule to the end of the makefile:

```
build/sox-14.4.1/configure : packages/sox-14.4.1.tar.gz
    cd build && tar -xzf ../packages/sox-14.4.1.tar.gz
```

Make sure there is an empty line between this rule, and the previous rule.

This tells make how to build the configure target, so you can now do:

```
make build/sox-14.4.1/configure
```

Note that make does not re-execute the download rule, it only executes the tar rule. This is because if a file already exists, then it won't bother to re-execute the rule that produced it.

For the rule's command, we used:

```
cd build && tar -xzf ../packages/sox-14.4.1.tar.gz
```

Originally we performed the command from within the "build" directory, so to use the same tar command we first enter the correct directory and then ('`&&`') execute the tar command. Changing directories within a command only affects that particular command – any further commands will execute from the original working directory.

3.6.2 Fill in the rest of the stages

You can now follow the same process for the rest of the stages, adding a rule to the makefile for each.

Two of the stages involve calling another makefile from within make – this is perfectly valid, but in order to allow things like parallel make to work well, we need to use the convention for recursive make. So where you would naturally write the command as:

```
build/sox-14.4.1/src/sox : build/sox-14.4.1/Makefile
    cd build/sox-14.4.1 && make
```

instead write it as:

```
build/sox-14.4.1/src/sox : build/sox-14.4.1/Makefile
    cd build/sox-14.4.1 && $(MAKE)
```

Once you have added all the rules, doing:

```
make local/bin/sox
```

should result in all the sox build and install stages running. However, they will only run if necessary – if you run the same command again, nothing will happen, because the target already exists.

3.6.3 Add support for lame

The same steps can be followed to add the commands to build lame to the same makefile – simply add the equivalent rules after the rules for sox. You should mostly be able to copy and paste the rules, then modify where necessary to work with lame. [4]

3.6.4 Create a dummy target

We'd also like an easy way to tell make to build all the tools we need, so the final thing we'll do is create a dummy target called "tools". Add the following line to the end of your makefile:

```
tools : local/bin/sox local/bin/lame
```

This is a dummy rule, because it has no commands, so the "tools" target is considered to exist as long as both sox and lame have been built. You can then type:

```
make tools
```

and it will check that both tools have been built, and if necessary perform the steps to build them.

[4] There are ways to templatise things like this, but we won't go into them here.

3.7 Conclusion

We now have an automated build system for getting and downloading the tools. As a final test, clean the environment as before (making sure not to delete your makefile), then do:

```
make -j 4 tools
```

This should download and build all the tools, and as a bonus you should see the compilation of both lame and sox proceeding in parallel. Some of the output will be messed up, because concurrent tasks are writing to the terminal, but the resulting files will be ok.

Note-1: *The test tools for submissions will* `make tools` *as the first step, so do make sure it works from a clean start.*

Note-2: *These makefiles will look essentially identical, so don't worry about us thinking you have plagiarised.*

4 Part B : Processing with pipes

This section will play around with the concept of pipeline parallelism and streaming, all in the context of streaming audio.

We'll be working in the directory "audio", and will rely on the two tools built in the previous section having already been built. We will be using C rather than C++, but feel free to use C99 constructs.

4.1 Some helper functions

First we'll define some helper scripts to reduce typing. In the audio directory, create a text file called "mp3_url_src.sh" and enter the following text:

```
#!/bin/bash
curl $1 | ../local/bin/lame --mp3input --decode - -
```

then create another file called "audio_sink.sh", and give it the contents:

```
#!/bin/bash
../local/bin/sox  - -d
```

Save both files, and as before, be very careful about line endings on windows.

If weird things are happening, try doing `cat -v your_file.sh`*, and check that there aren't any special characters at the end of the line.*

Then, (with "audio" as your current directory) do:

```
./mp3_url_src.sh http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/bn9th_m4.mp3 \
    | ./audio_sink.sh
```

This is equivalent to the streaming playback we set up in the previous section, but we have hidden the details of the command line arguments. By wrapping

them up in little scripts, we have hidden the details, but can still connect them together via their stdin and stdout.

Streaming from an URL is wasting bandwidth, so download the mp3 by doing:

```
curl http://cas.ee.ic.ac.uk/people/dt10/teaching/2013/hpce/cw2/bn9th_m4.mp3 \
    > bn9th_m4.mp3
```

So we're downloading the file as before, but redirecting the output to a local file. [5] You can check that the local file is a valid mp3 by playing it in a normal mp3 player.

From this point on, feel free to substitute any mp3s while testing the graphs, or turn the volume down, as otherwise it gets very boring when you're debugging. You will be explicitly told if you need to work with a particular mp3, or actually listen to the output.

Create a new script file called "mp3_file_src.sh", and give it the body:

```
#!/bin/bash
../local/bin/lame --mp3input --decode $1 -
```

You should now be able to do:

```
./mp3_file_src.sh bn9th_m4.mp3  |  ./audio_sink.sh
```

Here we have kept the same sink (audio out), but we are using a different source which reads from a file rather than an url. This ability to swap in and out parts of the chain is one of the great strengths of the unix pipe philosophy.

If we want to look at the speed of decoding, without being constrained to the actual playback speed (the audio sink will only accept samples at normal audio frequency), we can redirect the output to a file. However, if we are only interested in processing speed, we might as well avoid writing to a real file, so there is no chance that disk speed will slow us down. For this we can use `/dev/null`, which is a special file which throws away anything written to it.

To time the command, we can use... `time`. This takes as an argument the command to execute, then says how long it took. So if you do:

```
time ( ./mp3_file_src.sh bn9th_m4.mp3  > /dev/null )
```

you'll notice that the CPU usage shoots up to 100% (at least on one CPU), and it will probably decode the file in a few seconds.

4.2 Getting raw data

The default output for lame when decoding is a wav file, containing the raw data from the stream. For example, if you do:

```
./mp3_file_src.sh bn9th_m4.mp3  > ../tmp/dump.wav
```

[5] This performs the same function as using the "`-o`" option you used to get the coursework spec.

then you should be able to play the resulting wav in a standard media player.

The media player can do this because wav files contain meta-data, as well as the raw sound samples. So the header describes things like the sampling rate, the number of channels, the bits per sample, and the endian-ness. We don't want to bother with the header, so for the rest of this coursework, whenever we pass audio data around it will be:

- Dual channel (stereo).
- 16 bits per sample; signed; little-endian.
- 44.1 KHz.

By default lame will add a wav header, so we need to get rid of that. Sox is also expecting a wav header so it knows what kind of data it is, so we'll need to manually specify that instead.

If you look at the documentation for lame, it specifies the "-t" option to get raw output, so update both "mp3_file_src.sh" and "mp3_url_src.sh" to pass that flag to lame.

The documentation for sox details multiple settings which can be used to specify the input file configuration, you can find it by looking for "Input & Output File Format Options". First note that the settings can be applied to either the input or output stream of sox, depending on which filename it appears before. In this case we want to specify what the format of the input stream is, so in "audio_sink.sh" insert the flag "`-t raw`" before the hyphen representing stdin, along with options to specify the format of the audio data described above.

Now try running the same streaming command as before:

```
./mp3_file_src.sh bn9th_m4.mp3  |  ./audio_sink.sh
```

If all has gone well, the data streaming between them will now be raw binary, rather than a wav file, but the effect should be exactly the same. If it sounds weird or distorted, double check that you have set the correct format on both side.

While you are modifying settings, you may notice that both lame and sox produce quite a lot of diagnostic information as they run. You may wish to suppress this in your scripts, using "`--silent`" and "`--no-show-progress`" options, respectively.

4.3 A simple pass-through filter

Now we'll now finally start adding our own stuff to the pipeline. You will find a file in "audio" called "passthrough.c". Compile it by doing:

```
make passthrough
```

Note that even though we don't have a makefile in the directory, make has some default rules for common processes. In this case, make knows how to turn a file called "XXXX.c" into an executable called "XXXX".

The original distribution was missing "`#include <stdint.h>`" from "passthrough.c", which meant it didn't compile in some environments (no definition of `int16_t`). This is updated in the latest tarball, or you can add it to the top of "passthrough.c" yourself.[6]

Insert your new filter into the chain:

```
./mp3_file_src.sh bn9th_m4.mp3  | ./passthrough  | ./audio_sink.sh
```

If all goes well it should sound the same.

However, if you look at the processes in a process manager, you'll most likely see that "passthrough" is taking up a lot of CPU time, probably more than lame and sox (though it varies between systems). To make this effect clearer, use the same approach as before to remove the speed limitations:

```
./mp3_file_src.sh bn9th_m4.mp3 |  ./passthrough  > /dev/null
```

Depending on your system, you may now see that "passthrough" is taking up one entire CPU core, while lame is taking less CPU time. At the very least, "passthrough" will be using a lot of CPU to do not very much.

One possible problem is that we are compiling it with no optimisations. Create a makefile in the "audio" directory, and add the statement:

```
CPPFLAGS += -O2 -Wall -lm
```

here we are adding three flags that will be passed to the C compiler: "-O2" turns on compiler optimisations; "-Wall" enables warnings (generally a useful thing to have for later stages); and "-lm" tells it to link to the maths library. If you wish, you could also add `-std=c99` to the flags, if you want to use C99 constructs.

Now build it again with:

```
make passthrough
```

You will likely find that nothing happened, because from make's point of view the target "passthrough" already exists, and has a more recent timestamp than "passthrough.c". You have two ways to force it to do the make:

1. *Touch* the source file "passthrough.c', i.e. make sure the file's modification date is updated to now. You could do that by simply modifying and saving the C file, or by using the `touch` command.

2. Pass the "-B" flag to make, in order to force it to ignore timestamps and build everything. So you could do `make -B passthrough`.

If you run the optimised version, you'll still find much the same problem, with "passthrough" being very slow (at least for the amount of work being done).

The problem here is excessive communication per computation. Passthrough is pretty much all communication: all it does is read from one pipe and write

[6] Thanks to Richard Evans.

to another. The problem is in the fixed-costs versus per-byte costs: there is quite a lot of OS overhead to call `read` and `write`, so for very small transfers the per-byte cost is dwarfed by the cost of setting up the transfer. This is very similar to the argument we used for vectorisation, where the scheduling cost (i.e. doing a read) is high, while the calculation cost (actually getting each byte) is relatively cheap.

The solution is that instead of reading just one time sample (i.e. four bytes), each call to read and write should move multiple samples, using a larger buffer.

Task: Modify passthrough so that instead of processing 1 time sample (4 bytes), it processes n samples (n*4 bytes). If the executable is given no arguments, it should use a default of 512 samples. If a positive integer is passed to passthrough, then that should be the number of samples used to buffer.

Once you have got passthrough modified, take some time to experiment with different buffer sizes, and look at the difference in speed – at the default buffer size "passthrough" should take negligible time, while for smaller sizes it will start to become the main bottleneck.

4.4 Generating signals

Task: Write a C program called "signal_generator.c" which can synthesis sine waves of a given frequency (I would suggest copying and modifying "passthrough.c"). This program should have one argument, which is a floating-point number giving the desired frequency (f). If there is no argument, then the default frequency should be 50hz. The generator should use a batch size of 512 samples, i.e. each time it writes to stdout it will write 2048 bytes.

The "-lm" flag previously suggested for CPPFLAGS in the makefile was missing from the original version – if you are getting linker errors with `sin`*, try adding it.*[7]

If we consider the very first sample output to have time $t = 0$, then at time t both left and right output should have the value:

$$30000 \sin(t 2\pi f) \tag{1}$$

rounded to 16 bit.

If you pipe the output of your signal generator to "audio_sink.sh", then you should hear a sine wave. It will get annoying quickly.

4.5 Merging signals

Task: Write a C program called "merge.c", which takes two input streams, specified as files on the command-line, and merges them to a single output stream which is specified to stdout. The merging function should take an equally weighted blend from both streams, so if you supply the same input for both arguments, you'll end up with the same thing back. Unlike before, you'll actually

[7] Thanks to Richard Evans.

have to open the two input files, rather than just using stdin. Use a batch size of 512 samples again.

In order to test this program, you'll need two raw files. You can generate them just as:

```
./signal_generator 1000 > ../tmp/sine1000.raw
./signal_generator 2000 > ../tmp/sine2000.raw
```

For each command, let it run for a couple of seconds, then kill it with ctrl-c. You can now test your program "merge", by doing:

```
./merge ../tmp/sine1000.raw ../tmp/sine2000.raw
    | ./audio_sink.sh
```

Going via temporary files means that data is forced onto disk (or at least to disk cache), but really we'd prefer things to stay in memory. One way of doing this is to use another feature of bash called input redirection. Do the command:

```
./merge  <(./signal_generator 1000)  <(./signal_generator 2000)
    | ./audio_sink.sh
```

This process can actually be nested, so for example, you could write:

```
./merge \
    <(./merge \
        <(./mp3_file_src.sh bn9th_m4.mp3) \
        <(./merge <(./signal_generator 600)  <(./signal_generator 700)) \
    ) \
    <(./merge \
        <(./merge <(./signal_generator 800)  <(./signal_generator 900)) \
        <(./merge <(./signal_generator 1000)  <(./signal_generator 100)) \
    ) \
    | ./audio_sink.sh
```

This sounds terrible, [8] but what is important is that you have connected these sequential programs into a single parallel processing graph. There are no intermediate files, so it can keep going for ever.

The effect of this is more obvious if you redirect the output to `/dev/null`:

```
./merge \
    <(./merge \
        <(./mp3_file_src.sh bn9th_m4.mp3) \
        <(./merge <(./signal_generator 600)  <(./signal_generator 700)) \
    ) \
    <(./merge \
        <(./merge <(./signal_generator 800)  <(./signal_generator 900)) \
        <(./merge <(./signal_generator 1000)  <(./signal_generator 100)) \
    ) \
    > /dev/null
```

[8] And looks terrible. However, you could hide it in a script.

The CPU usage should shoot up, and on my 8-core desktop I get about 6 of the cores fully utilised.

4.6 FIR filtering

The final function we'll add is discrete time FIR filtering. FIRs are examples of useful but time consuming filters that need to be applied in real-time streaming systems, but bear in mind I'm not a DSP expert – feel free to laugh at the naivete of my coefficients.

We can represent one channel of our discrete-time input audio as the sequence $x_1, x_2, ...$ and we'll filter it with a k-tap FIR filter. The filter has k real coefficients, $c_0, ..c_{k-1}$, and the output sequence $x'_1, x'_2, ...$ is defined as:

$$x'_i = \sum_{j=0}^{k-1} x_{i-j} c_j \tag{2}$$

In the "audio/coeffs" directory you'll find a number of coefficient sets for notch filters, with filenames of the form "fXXX.csv", where XXXX is the centre frequency. Different filters have different orders.

Note that the original coursework tarball didn't contain the coeffs directory – if you have got this far on the original tarball, then you can download again to another directory and simply copy across.[9]

Task: Write a program called "fir_filter.c" that takes as an argument a filename specifying the FIR coefficients, which will apply the FIR filter to each channel of the data coming from stdin, and write the transformed signal to stdout. All internal calculations should be in double-precision, but input and output should be in stereo 16 bit as normal.

So for example, if you write:

```
./merge <(./mp3_file_src.sh bn9th_m4.mp3) <(./signal_generator 800) \
    | ./audio_sink.sh
```

you'll get audio mixed with a sine wave, but if you do:

```
./merge <(./mp3_file_src.sh bn9th_m4.mp3) <(./signal_generator 800) \
    | ./fir_filter coeffs/f800.csv \
    | ./audio_sink.sh
```

then hopefully most of the sine-wave disappears. If you double up the filter:

```
./merge <(./mp3_file_src.sh bn9th_m4.mp3) <(./signal_generator 800) \
    | ./fir_filter coeffs/f800.csv \
    | ./fir_filter coeffs/f800.csv \
    | ./audio_sink.sh
```

then it should be completely gone.

[9] Thanks to Robert Bishop.

4.6.1 Some benchmarking

We're now going to benchmark two different ways of using our FIRs: first by using intermediate files, then by using direct streaming.

Task: Create a script called "corrupter.sh", which takes an audio stream as input, mixes it with sine waves, and produces an output. This script should be streaming, i.e. it can run forever as long as it keeps getting input. The ratios should be:

Source	Weight
Input	25%
500hz	12.5%
600hz	12.5%
800hz	12.5%
1000hz	12.5%
1200hz	12.5%
1400hz	12.5%

Task: Create a file called "all_firs_direct.sh". This should take an audio stream from stdin, and produce on stdout another stream that has had applied a notch filter at 500, 600, 800, 1000, 1200, and 1400 hz (using the f500,f600,...,f1400, coefficient files provided). This script should not use any intermediate files.

Task: Create another file called "all_firs_staged.sh". This will do exactly the same job as "all_firs_direct.sh", except each stage will read input from a named file, and write to a named file. We *may* wish to call this script in parallel with itself, so be careful about what the intermediate named files are called.

A useful pattern within the script may be to use mktemp, so for example:

```
T1=`mktemp ../tmp/XXXXXXXX`;
./mp3_file_src.sh bn9th_m4.mp3 > $T1;
cat $T1 | ./audio_src.sh
```

will create temporary file and store the name in variable T1, then decode the entire file to T1, then send it to the audio source.

Once you have written these three scripts, you can then try to work out how much time is saved by avoiding disk, by doing:

```
time (./mp3_file_src.sh bn9th_m4.mp3 | ./corrupter.sh | ./all_firs_direct.sh > /dev/null )
```

versus:

```
time (./mp3_file_src.sh bn9th_m4.mp3 | ./corrupter.sh | ./all_firs_staged.sh > /dev/null )
```

You may want to use a short mp3 file, unless your fir is blazingly fast.

Exact results will vary with your system, but what you'll probably see is that in the direct case, all six fir_filters are working together. If you've got two CPUs, both should be at 100%; for four CPUs there is just enough work to occupy all of them; and for eight CPUs, you'll see some of the fir_filters with more taps working flat-out, while some of the filters with fewer taps are operating a little under load.

In the staged case, which may appear the more natural approach if you have a bunch of discrete tools to use together, each of the stages in turn will occupy 100% of just one CPU. As only one thing is being done at once, you lose the ability to exploit the extra cores, and on my eight-core desktop it takes just over four times longer than the streaming method.

4.6.2 Conclusion

As a final stage, add the following to the end of your makefile:

```
tools : merge correlation fir_filter passthrough signal_generator
```

This will allow users to go into the "audio" directory and type "make tools", and all your files will be updated.

5 Part C : Doing something useful

One idea of this coursework is to introduce you to some tools, and give you some grounding in the techniques used in future courseworks, but the idea is also to give you something useful to use. A very common need in projects (both in academia and industry), is to execute a set of tools over some combinations of input criterion, and gather summary statistics into a table.

In this case, we're going to analyse the result of applying a number of different filters, and we're going to examine the correlation between the original file and the filtered file at various timescales.

5.1 Performing some analysis

Task: Copy "merge.c" to a new file called "correlation.c" in the "audio" directory. This file will still take two input streams, but will also now take a third integer argument specifying window size. The goal of this file is to:

1. Convert each of the inputs to mono through simple averaging.
2. Calculate a non-overlapping windowed Root-Mean-Squared (RMS) over each stream.
3. Track the correlation between windowed RMS across the streams.

So mathematically, if we have $A_L[i]$ and $A_R[i]$ as samples from the first stream, and $B_L[i]$ and $B_R[i]$ as samples from the second stream, we will first calculate the mono samples:

$$A_M[i] = (A_L[i] + A_R[i])/2 \tag{3}$$
$$B_M[i] = (B_L[i] + B_R[i])/2 \tag{4}$$

We'll then take those mono samples, and calculate the RMS over windows of size w (passed on the command line):

$$A_{RMS}[j] = \sqrt{\frac{1}{w}\sum_{i=0}^{w-1} A_M[j*w+i]^2} \tag{5}$$

$$B_{RMS}[j] = \sqrt{\frac{1}{w}\sum_{i=0}^{w-1} A_M[j*w+i]^2} \tag{6}$$

Over the course of the streams we'll end up with n distinct RMS windows (depending on the input stream sizes), and so finally we calculate the correlation:

$$r = \frac{\sum_{i=0}^{n-1} A_{RMS}[i] \times B_{RMS}[i]}{\sum_{i=0}^{n-1} B_{RMS}[i]^2 \times \sum_{i=0}^{n-1} B_{RMS}[i]^2} \tag{7}$$

The C code to do this consists of simply tracking the running squared sum within each window, then at the end of each window calculating the RMS and updating two sum of squares and a sum of products.

Once the calculation is finished, the program should print a single floating-point number to stdout containing the correlation. So this program acts as a sink – audio comes in, but only a single correlation comes out.

Once the code is working, you can try doing this:

```
./correlation \
    <(./mp3_file_src.sh bn9th_m4.mp3)  \
    <(./mp3_file_src.sh bn9th_m4.mp3 | ./fir_filter coeffs/f3000_k100.csv ) \
    1024
```

If you adjust the window size, you should see that the correlation decreases as it gets smaller, and increases as it gets larger. This makes sense, as you'd expect FIR filters to introduce large amounts of local difference, and very little large scale differences.

There is a certain amount if redundancy going on here, as the mp3 is being decoded twice, so I could actually save that redundancy by creating a fifo file. Using fifos you can actually make arbitrary streaming graphs, but that goes beyond the intent of this coursework and course – while extremely useful, it is not "safe" parallelism, as it is possible to deadlock.

5.1.1 Combining pipe-based and streaming parallelism

Move into the "experiment" directory. Within that directory there is a "coeffs" directory, containing a number of FIR coefficients, and an "inputs" directory, which is currently empty. Select three mp3 files (it doesn't matter what, as long as they are stereo 44.1KHz), and copy them into the inputs directory.

Imagine we are comparing the behaviour of a particular set of filters, and we want to explore it's correlation behaviour. We can't just run it on one input

mp3, as that file might be be an outlier. Also, we can't just check one correlation window, as we don't know which window size is important. What we want to do is:

1. For each input mp3:
2. For each filter:
3. For each window:
4. Calculate the correlation

We could do this using a shell-script containing three for loops which executes the right commands, but there would be some problems:

- There is no parallelism in this process – even if you have multiple cores, they won't be used.
- If any one of the inputs changed, then *all* outputs must be recalculated when updating the correlations.
- If a new coefficient is added, then *all* outputs must be recalculated to add the new correlations.
- If the process is interrupted, there is no easy way of continuing, so the process would start from scratch.
- In a naive implementation each filter would be applied each time we calculate the correlation, rather than in the outer loop.
- There is no simple way of building just one of the correlations, or a small subset of the correlations.

Some of these problems can be fixed with extra logic in the script, but eventually it becomes very complicated. Better to use a tool designed for the job.

The "experiment" directory contains a makefile which does all of this in a (relatively) short script. You can execute it by doing:

```
make -j 4 all.csv
```

You should see that immediately four filtering processes start, all of which are handling different pairs of inputs and filters. After a while they will finish and multiple analysis tasks will start executing – there are a lot more analysis tasks than filtering, but they execute a lot quicker. Depending on your machine, you may see a transition zone when there are some filtering tasks and some analysis tasks running concurrently. That is because make considers true dependencies when scheduling the tasks, so if an analysis task is ready to run (i.e. its input has been generated), then it doesn't matter if other filtering tasks are still going.

Now try modifying one of the input coefficients files, e.g. tweak one of the coefficients slightly. If you now ask make to build "all.csv" again, it will only rebuild those wavs which depend on the filter, and only re-analyse those that depend on the filter.

6 Conclusion

Task: To prepare this coursework for submission, you need to clean your submission of all the intermediate files, such as executables, downloaded packages, mp3 files, ... What you should be left with is the original structure and files, and the various scripts, makefiles, and c files that you have created. *Make sure you don't delete the source files you have created.* Once you have cleaned the directories, go into the root directory and do:

```
tar -czf ../hpce_cw2_your-login.tar.gz .
```

where "hpce_cw2_your-login" is your standard college login. For example, mine would be called "hpce_cw2_dt10.tar.gz". This should create a submission tarball in the directory below your work. Note that the tarball should be quite small, around a 100 KB to 200 KB. If it seems much bigger, make sure you haven't accidentally included any mp3s or executables.

Before submitting, you should do a number of checks to make sure it works:

1. Extract the tarball to a fresh directory.
2. Run `make tools` in the base of the submission to build sox and lame.
3. Go into the audio directory and run `make tools`.
4. Choose an mp3, and do `./mp3_file_src.sh your_mp3.mp3 | ./corrupter.sh | ./all_firs_direct.sh > /dev/null`.
5. Go into the experiment directory, copy an mp3 into "inputs", and try running `all.csv`.

If that all works, submit your tarball via blackboard.

Congratulations! You now know how to:

- Build and install random software packages without being root.
- Automate the creation of build environments.
- Declare dependency graphs using makefiles.
- Perform parallel processing using makefiles.
- Exploit recursive parallelism. [10].
- Integrate together multiple disparate tools using pipes.
- Exploit streaming parallelism using pipes.

[10] Remember that sox and lame will built together

7 Endnotes

7.1 Errata

2014/02/05 - 02:04 : Missing coeffs directory in the spec file – a classic failure to specify dependencies correctly on my part. Thanks to Robert Bishop.

2014/02/04 - 14:25 : Some odd unicode thing making the "–no-show-progress" flag shown as "noshowprogress". Plus some good suggestions on improving the portability of the instructions relating to linking and header files: thanks to Richard Evans for all of those.

7.2 Estimates of time taken

Based on me running through:

- Reading time: 0.5 hours.
- Following instructions: 1.5 hour (not including C code).
- Writing C code: 3 hours (estimate only).