A Parallel FLAC Encoder

Christopher Peplin
peplin@umich.edu

Maxwell Miller
mgmiller@umich.edu

April 17, 2008

1 Introduction

Our project began with the intention to parallelize the FLAC encoder and create a multi-threaded transcoder (FLAC to MP3) that operated at the audio block level. Upon acquiring the FLAC source code (which includes the C and C++ libraries, flac command line encoder/decoder and some example programs), we realized the need to scale back our expectations of the project. The FLAC API is an extensive, robust library which was built from the ground up, unfortunately, as a serial application.

We spent the first two weeks attempting to understand the flow of the basic flac encoder when dealing with a typical audio file (2 channel, 16-bit WAV). After that, we identified possible points of parallelism at two levels of the encoding process: the high level (the frontend, above the library), and low level (within the library).

This report summarizes the serial algorithm, it's potential for parallelism, and our issues with the library design in implementation. Ultimately, we were successful at parallelizing the algorithm at a high level using a pipeline from Intel's Threading Building Blocks (TBB). This project has definitive implications on the future of the FLAC project, but its integration depends on the evolution of the flac program (which is currently written in C, and thus incompatible with TBB) and the cooperation of the project owner (Josh Coalson) and community.

2 Serial Problem

FLAC is a lossless audio codec that converts WAV files to compressed FLAC files that are seekable and streamable. A FLAC file contains a stream header followed by a series of audio frames which hold a small piece of the encoded audio along with enough information to decode that frame. The encoding algorithm can be described at a high level in this way:

1. Read WAV file header and confirm its parameters are compatible with this encoder
2. Open the output file stream
3. Write stream header, including metadata such as artist/album information and an MD5 checksum
4. Encode:

```
while( there is audio left to encode ){
read a block of audio
encode block into a FLAC frame
write frame to output stream
}
```

5. Write last block and stream footer

Each block of audio is processed in 3 stages.

1. **Inter-channel Decorrelation** The encoder decides whether or not to split the audio into channels (left & right), and if so, which method. One option is a simple left/right split, and another (which often garners significant extra compression) is mid and side channels.
2. **Modeling** The audio signal is approximately modeled by a function in one of three ways. The function should ideally represent the original audio with fewer bits.
   - Constant signal frames fit to a simple, constant function with only one argument
   - Extremely busy or random frames are modeled verbatim, as themselves
   - All other frames are modeled using general linear predictive coding (LPC)
3. **Residual Coding** Once modeled, the model is subtracted from the original audio to produce the residual, or error. This stream is then encoded using Rice codes (a special set of Huffman codes) and run length encoding.

The bulk of the FLAC project is a C library (libFLAC) that implements the codec as per the format specification. There is also C++ library of wrapper classes (libFLAC++). Any actual FLAC encoder or program using the FLAC format must implement the encoding algorithm as described above, using functions provided in libFLAC. The reference encoder in the source, flac, provides one such implementation.

Buffering & Blocking There are two important considerations regarding speed and compression ratio with the serial problem. There are two buffers - one reads directly from the input file and feeds its data to the second buffer that holds the actual block of audio to be processed. The second buffer is of constant size (one block of samples), but the first is variable. The example encoder frontend reads 1024 samples into the first buffer, and calls the process frame function. That function funnels data from the first buffer to the second, waiting until the second has one full block of audio before actually processing (encoding) the frame. Thus, the variable size of the first buffer is important to consider in relation to available memory, disk read speed and cache size.

Block size is also an important parameter. Similar to thread overhead problems in parallel programming, too small of a block size will lower the overall compression while too large a block will not allow the compressor to generate an efficient model for the audio.

Serial Performance In terms of real world performance, the FLAC encoder typically encodes a 3-4 minute song in 4-6 seconds which translates into approximately 40 seconds per album. By profiling the reference encoder with gprof, we discovered that 94-98% of that time is spent within the different variants of process_frame (the middle step of the while loop from above).

Truncated profiling results showing the prime candidates for parallel speedup:

```
index  \% time  self   children    called     name
                0.00    0.00       1/3080        FLAC__stream_encoder_finish [37]
                0.00   12.99    3079/3080        FLAC__stream_encoder_process [6]
[7]     96.5    0.00   13.00    3080          process_frame_ [7]
                0.00   12.34    3080/3080        process_subframes_ [8]
                0.00    0.47    3080/3080        FLAC__MD5Accumulate [22]
<truncated>
```

3 Concurrent Architecture

We spent a large amount of time after beginning this project identifying which classes and source files were of interest. We narrowed our modifications down to the example C++ encoder and both the libFLAC and libFLAC++ libraries. At first look, we had no problem finding embarrassingly parallel loops. Many of the functions in the library iterate over large data sets with computationally intensive processes (process_frame, which does the LPC calculations, is a good example). However, many of the loops turned out to usually iterate over much smaller numbers of items than we expected.

After analyzing the actual runtime and size of each loop, our top three parallel plans were:

1. A 4-6 stage pipeline within process_frame, which is called on each frame.
2. A parallel_for within stream_encoder_process for a loop that iterates over each sample in the block.
3. A 4 stage pipeline at the encoder frontend level, mimicking the pseudocode demonstrated in the Serial Problem section.

Our program uses plan 3, because it cleanly separates I/O from computation in a situation very similar Intel's example code for the TBB pipeline. Each frame is processed completely independent of any other, so data dependencies are in theory minimal. Plan 1 is also clearly separated, but the overhead of a pipeline for each frame is too much, and would limit any speedup drastically. We determined that the work in loop of the second plan was ultimately insignificant in comparison to other areas of the encoder.

Before explaining the hazards of this design, we present our pipeline filter plan and the read/write dependencies between each filter. These filters are defined in filters.h, and the type of token passed among them is in pipelinestruct.h.

InputFilter
: read: infile, shared encoder
: write: shared encoder, raw WAV buffer, byte counters

PCMFilter (parallel)
: read: byte counter, shared encoder, raw WAV buffer
: write: PCM buffer

ProcessFilter (parallel)
: read: PCM buffer, byte counter
: write: parallel encoder

OutputFilter
: read: parallel encoder
: write: outfile, shared encoder

While designing the pipeline, it became obvious that libFLAC was built from the ground up as a serial library. Every function in the library is based the StreamEncoder struct, which stores all information about the audio file, encoding status and state, progress measures and every buffer minus the first, variable sized buffer of raw WAV data. This struct is very convenient in serial, as it imitates object oriented functionality in C. In parallel, the fact that this much state is held by one object severely restricts the parallelism. To solve this problem, we rewrote a handful of library functions to be pipeline-safe. Instead of working on a single encoder struct, they now take two encoder structs as arguments - a shared encoder read by every filter and another encoder mutually exclusive to a token. The motive behind this is to give each pipeline token a unique, local encoder to keep encoding state and buffers. Anytime file metadata or progress counters are requested in a function, they instead read the shared encoder (which is thread safe).

4 Performance Results

The parallel version compares very favorably to the serial example encoder. Both encoders are still I/O bound at times (especially on large files). CPU usage sometimes drops below 100%, but this has to do more with buffer sizes than the parallelism. We used a test suite of 6 audio files, which we encoded while timing using the two encoders, then verified by hand that the resulting audio was not corrupted. We saw very large speedups in all of the test cases, which we attribute to the fact that the intense computation is now parallel, and the I/O and computation are also separated into multiple threads by the pipeline structure. File size did not have a noticeable impact on encoding time.

- test1.wav (530MB) - Nine Inch Nails - Ghosts, Disc 1
  - Quiet, ambient CD with lots of constant and silent frames
- test2.wav (584MB) - Nine Inch Nails - Ghosts, Disc 2
  - Heavier industrial sound, a good mix of the three frame types)
- test3.wav (17MB) - Wolf Eyes - Dead in a Boat
  - Noise rock, stress verbatim frames
- test4.wav (58MB) - Charles Mingus - Wednesday Night Prayer Meeting
  - Acoustic jazz, lots of ambient noise, no silent frames
- test5.wav (57MB) - Fiona Apple - Window
- test6.wav (22MB) - Tom Waits - Lie to Me
  - Electric blues with louder, full frames than the previous track

Results Run on a dual core AMD Athlon X2 4200+, averaged over 3 runs:

```
Starting testing...

Encoding ./test1.wav with serial encoder...
```

```
average wall time  0m48.408s
Encoding ./test1.wav with parallel encoder...
average wall time  0m27.92s
73% speedup

Encoding ./test2.wav with serial encoder...
average wall time  1m2.962
Encoding ./test2.wav with parallel encoder...
average wall time  0m35.693
76% speedup

Encoding ./test3.wav with serial encoder...
average wall time  0m1.971s
Encoding ./test3.wav with parallel encoder...
average wall time  0m1.118s
76% speedup

Encoding ./test4.wav with serial encoder...
average wall time  0m6.209s
Encoding ./test4.wav with parallel encoder...
average wall time  0m2.963s
109% speedup

Encoding ./test5.wav with serial encoder...
average wall time  0m6.524
Encoding ./test5.wav with parallel encoder...
average wall time  0m2.65
146% speedup

Encoding ./test6.wav with serial encoder...
average wall time  0m1.844s
Encoding ./test6.wav with parallel encoder...
average wall time  0m0.973s
89% speedup

Testing finished.
```

Average speedup: 94%

5 Lessons Learned

This project was an immense learning experience for both of us. This is our first time touching a project of this magnitude, and it was eye opening to see their code organization and documentation. We also had to understand automake, and its Makefile generation scripts. We still do not know how to use this tool to its fullest, but we could figure out was very helpful for adding our libraries and new files.

Our confidence in parallel design patterns was strengthened by planning and implementing the encoder. Countless times during the planning stage, something we learned in lecture really

solidified in our minds. Examples of this include being wary of thread overhead and watching for blocking I/O. Most importantly, we saw firsthand how very difficult it can be to parallelize code written for serial execution. It is always a better situation to be working with code already optimized for minimum data dependencies. The majority of our time was spent separating the shared and mutually exclusive encoders, and still, there are some issues we have yet to resolve. Namely, the following extra requirements/limitations are placed on the encoder:

- No loose mid-side stereo
- Verification while encoding
- Verification after decoding (using the MD5 checksum)

These are not serious problems, and could be rectified with additional effort. We made a decision to focus elsewhere on this project, as these options eluded complete understanding and aren't common encoding options.

6 Conclusion

FLAC is an embarrassingly parallel encoder that looked very simple to parallelize. The vast source code made that job much harder, but ultimately there wasn't any more difficulty in the actual parallel design besides for what we anticipated. Intel's Threading Building Blocks freed us up to break down the library into functions we could use without worrying about the details of threads. At the scale of the TBB finger exercise, TBB's usefulness is not entirely clear. Integrated into a large project like this, TBB's algorithms were extremely helpful. We hope to see the library in common use, so a program like this could be distributed.

Our encoder is not full featured. Based on the example C++ encoder, it encodes a very limited subset of the types of files understood by the full-fledged FLAC reference encoder. There is no reason why our pipeline design could not be melded with the robust encoder, but it is beyond the time frame of this class. The reference encoder is a C program, so any further modifications would and should start as a new, C++ reference encoder. Time permitting, we will pursue this idea with the creator of FLAC and search for interested developers in the FLAC community. There are hints of interest in a multi-threaded FLAC encoder on the web, so enlisting the help of other develops should not be impossible with the solid start we've completed already.

Source Files created or modified in FLAC source:

```
examples/cpp/encode/file/filters.h
examples/cpp/encode/file/filters.cpp
examples/cpp/encode/file/pipelinestruct.h
examples/cpp/encode/file/main_parallel.cpp
include/FLAC++/encoder.h
include/FLAC/stream_encoder.h
src/libFLAC++/stream_encoder.cpp
src/libFLAC/stream_encoder.c
```

7 References

- FLAC Format Specification - http://flac.sourceforge.net/format.html
- FLAC Project Documentation - http://flac.sourceforge.net/api/index.html