卒業論文

PC クラスタ上での Ogg Vorbis エンコーダの並列化

氏　　名 : 井ノ口　春寿
学籍番号 : 2210030037-4
指導教員 : 山崎　勝弘　教授

提出日　　: 2007 年 2 月 16 日

立命館大学　理工学部　情報学科

内容梗概

コンピュータの高性能化に伴い、そのコンピュータを複数台ネットワークで接続した PC クラスタの性能向上も進んでいる。PC クラスタの高速演算処理は、地球シミュレーションに代表されるように、最先端情報処理技術に欠かせない重要な要素となっている。そのため今後ますます PC クラスタの活用が見込まれている。

しかし、PC クラスタには特有の問題がある。それは逐次処理とは異なり並列プログラミングが必要とされること、また PC クラスタのノード間で情報のやりとりを行う点である。MPI では、この情報のやりとりをプログラマが記述する必要があり、データを大量に送受信する時に、PC クラスタの性能に大きく影響を与え問題となる。

そこで本論文では、PC クラスタの性能測定と音声圧縮プログラムの並列化を行い、SCore 上で動作させて並列化によるデータ通信の影響について考察する。

まず PC クラスタの性能測定では、クラスタの世界で広く用いられている HPL ベンチマークを用いて Raptor Cluster の性能測定を行い、性能の実測値を測定している。このことにより、PC クラスタの特徴である高性能化を数字で確認することができ、他の PC クラスタと比較することが可能な点で有用である。

HPL での、Raptor Cluster（8 ノード）の性能は 27.28Gflops である。

また音声圧縮プログラムの並列化では、Ogg Vorbis を用いて実験を行っている。音声圧縮技術が音楽配信サービスなどで用いられ、今日の社会で重要な役割を担っていること、また音声圧縮には時間がかかり、PC クラスタでの高速化には意義があると考えたからである。本研究では、Ogg Vorbis エンコーダを MPI による並列化を行い、ノードごとのデータ割当量を変更して実験を行い、それらの性能を比較する。実験で、ノードごとのデータ割当量を変化させることにより通信量を変化させ、クラスタの通信の特性を調査している。またクラスタのノードを 1 台から 8 台まで変化させ性能の比較を行った。

MPI による並列化の効果として、Raptor Cluster（8 ノード）で最高 3.51 倍の速度向上を達成した。またこれらの実験ではある程度の速度向上が得られたが、全体処理の時間は理想的な速度向上と程遠い結果となった。その原因として、PC クラスタのノード間通信において、通信回数と 1 回あたりの通信量にトレードオフ関係があり、ノード間通信が実行時間の増加をもたらしたためである。

目次

図目次

表目次

1 はじめに

近年の PC クラスタの性能向上はめざましく、1 ペタ flops のクラスタが “TOP500”（[5]）に登場するのも時間の問題であると考えられている。これは PC クラスタの構成要素である PC 自体の高性能化による物に他ならない。そのことにより今まで時間がかかりすぎて不可能であると思われていたことが実現してきた。例えば、DNA 解析・地球規模の気候変動シミュレーション・暗号解析などである。これらのことは数十年前には不可能であると考えられてきたことである。PC クラスタの高性能化はその実行時間を短縮させ、現実的な時間で解けるようにした。今後、今現在時間がかかりすぎて不可能であると思われていることが、PC クラスタ自体の高性能化もさらに進んでいき可能になることであろう。

しかし、その性能を最大限に用いるためにはどのように PC クラスタを用いるべきなのか研究する必要があり、PC クラスタを支える並列化に関する研究が必要である。というのも、プログラムの並列化を行うにあたり並列化独自の問題が生じてくるからだ。例えば、何も対策をしないでプログラムの並列化を行うと、PC クラスタの台数を増やしたときに並列効率の減少を招く恐れがある。なぜなら不必要なデータが割り当てられたり、データを転送する際にブロードキャスト通信で行うと通信によるボトルネックの発生など逐次プログラムでは考えられなかった問題が生じてくるからである。そのため PC クラスタの特性の調査は PC クラスタの能力を最大限引き出すために重要である。PC クラスタの特性を決定する要素には、プログラム自体の計算量・通信量・メモリ使用量があるが、プログラム自体の計算量とメモリ使用量は逐次プログラムと何ら変わるところがないから、通信量について着目して研究を行っている。

そこで PC クラスタの通信問題を中心とする特性調査を研究の目的として、本研究ではベンチマークプログラムと音声圧縮プログラムにより研究を行っている。

まずベンチマークには、クラスタの世界で広く用いられている HPL ベンチマークによりクラスタの理論的な性能を測定し、ノード数の増減による性能評価を行った。HPL を用いた理由として、HPL が多くの PC クラスタで実行されており、性能特性の調査を行うのに必要十分な要件を満たすと考えたからである。また PC クラスタの特徴である高性能化を数字で確認することができ、他の PC クラスタと比較することが可能な点で有用であると考えたからである。もっとも HPL ベンチマークの結果は PC クラスタの 1 つの側面を表しているにすぎないため、この結果のみで PC クラスタの特性を判断することはできない。しかし世界共通の統一的な性能指標を客観的に示すことができるため用いることとした。

本研究では、メモリ使用量を変化させてクラスタの性能測定とノードの増減による性能比較を行った。

次に、音声圧縮プログラムを並列化して、クラスタで実行し性能評価を行った。今回音声圧縮プログラムを選択した理由として、音声圧縮技術がオンライン音楽配信サービス・携帯電話の音声・地上波デジタル放送の音声・DVD の音声などで幅広く用いられている技術であり、今日の社会で重要な役割を担っていることがある。また音声圧縮には時間がかかり、PC クラスタでの高速化には意義があると考える。本論文で、音声圧縮には Ogg Vorbis を用いている。Ogg Vorbis を用いた理由としては、多くの音声圧縮プログラムのソースコードが手に入らない状況下で Ogg Vorbis はフリーソフトで開発されており音質にも定評があることを考慮して決定した。実験では、並列化した Ogg Vorbis エンコーダをクラスタで性能測定し実際的なアプリケーションによる性能評価を行っている。

本研究では、Ogg Vorbis エンコーダを MPI による並列化を行い、ノードごとのデータ割当量を変更して実験を行い、それらの性能を比較する。なお MPI で並列化を行ったのは分散メモリのクラスタにおいて一般に利用されており標準規格があるためである。実験で、ノードごとのデータ割当量を変化させることにより通信量を変化させ、クラスタの通信の特性を調査している。またクラスタのノードを 1 台から 8 台まで変化させ性能の比較を行った。

以上のように、2 つの観点から並列化に関する研究を行い、ノード間通信に重要な意味があることを実験により確かめた。今後 PC クラスタの高性能化が見込まれる中で、PC クラスタの特性を調査したことには意義がある。また研究を行うことにより、PC クラスタの発展に寄与したいと考えた。

最後に本論文の構成は、第 2 章で PC クラスタと並列プログラミングの概要と HPL ベンチマークについて述べ、第 3 章で Ogg Vorbis エンコーダ・第 4 章で並列化手法について説明する。そして、第 5 章で実験結果と考察について報告する。

2 PC クラスタと並列プログラミング

2.1 PC クラスタとは

PC クラスタとは複数台の PC（コンピュータ）を組み合わせ、それぞれの PC が協調動作するように設計されたシステム全体を言う。協調動作を行わせるためには、専用のソフトウェアが必要となり、クラスタ内の PC がネットワークで接続されている必要がある。また PC クラスタを動作させて計算を行わせるためには並列プログラムが必要となる。これらの要件を備えることによって、PC クラスタは PC クラスタを構成する個々の PC を意識することなく、まるで 1 台の PC のように扱うことができる。

PC クラスタが利用されるのは主にプログラムの高速化である。この用途に利用されるクラスタは High-performance clusters （高性能クラスタ、以下 HPC と約す）とも呼ばれる。HPC はクラスタの各 PC（ノード）に処理を分散させることにより高速化を実現している。

HPC の並列プログラミング実行環境として、SCore と Beowulf が有名である。

SCore は、経済産業省による超並列処理研究推進委員会である新情報処理開発機構（RWCP）にて開発された実行環境である。SCore の特徴として、共有メモリを前提とした OpenMP を分散メモリで利用できる点がある。これにより分散メモリの PC クラスタで OpenMP を利用できる。

また Beowulf は NASA の Thomas L. Sterling と Donald Becker によって開発された実行環境である。

2.2 並列プログラミング

例えば、C 言語で書かれたプログラムを PC クラスタ上で動かし、並列に動作させプログラムの高速化を図ることはできない。なぜなら、C 言語は 1 台のコンピュータで動かすこと（逐次実行）を前提としているからだ。

そのため PC クラスタを動作させるために必要となるのが並列プログラミングである。並列プログラミングとして有名なのが MPI と OpenMP である。

2.2.1 MPI について

MPI とは、Message Passing Interface の略である。

MPI を用いることにより PC クラスタ上で並列動作させることができるようになる。ただ MPI は新しいプログラミング言語ではなくライブラリとして提供される。そのため C 言語で書かれたプログラムにリンクさせることにより並列化を行うことができる。ただし MPI が自動的にプログラムを並列化させるわけではなく、プログラマーが並列に動作させたい部分を指定する必要がある。

MPI は分散メモリ環境を前提としており、MPI を用いたプログラムは、C 言語でのネットワークプログラムのサーバ・クライアントの考えに近い。つまり命令を指示するサーバと命令に従って行動するクライアント（クラスタではノードと呼ばれる）という考えにたち、命令やデータ（MPI ではメッセージと呼ばれる）をサーバがクライアントに伝え、クライアントがデータを処理し、サーバに送り返すという処理を行う。並列化を行っているとクライアントが複数存在していることになる。そのため処理が高速になる。

2.2.2 OpenMP について

OpenMP はライブラリであり、プログラマーが並列化させたい部分を指定する点は MPI と同様である。

ただ、OpenMP は共有メモリ環境を前提にしている点で MPI と異なる。また MPI では明示的にノード間通信をする場合に、MPI 専用の通信関数を呼び出さなければならないが、OpenMP はディレクティブを挿入することに並列化でき、ノード間通信をプログラムする必要がない。

MPI と比べると、OpenMP は共有メモリを前提としているので、SMP（対称型マルチプロセッシング）では性能向上が見込まれる。その理由として、OpenMP では異なるスレッドにおいて同一アドレスを参照でき、スレッドごとにデータの移動が不要なためである。

2.3 PC クラスタの性能測定

PC クラスタはどの程度の性能を発揮するのかを客観的に調べるために、今後の実験で使用する Raptor Cluster の性能測定を行った。

なお性能測定には、ネットワーク測定（NetPIPE）とベンチマーク測定（HPL Benchmark）を行いノード間通信とクラスタ全体の両面から調べた。

2.3.1 Raptor Cluster の構成

Raptor Cluster の構成は以下の図 1 のようになる。Raptor Cluster は 1 台のサーバホスト（spino）と 8 台計算ノード（raptor00-raptor07）から構成される。それぞれのノードは Gigabit Ethernet Switch を通してノード間通信を行い、またサーバとの通信を行う。

計算機の性能は図 1 に示したとおりである。サーバホストである spino は Xeon プロセッサの 2.8GHz で 2GB のメモリを積んでいる。またノード側である raptor は Pentium4 プロセッサの 3.2GHz で 2GB のメモリを積んでいる。なお計算機どおしをつなぐイーサネットスイッチには最大帯域 1Gbps のものが用いられている。なお、並列プログラミング環境として SCore を利用している。

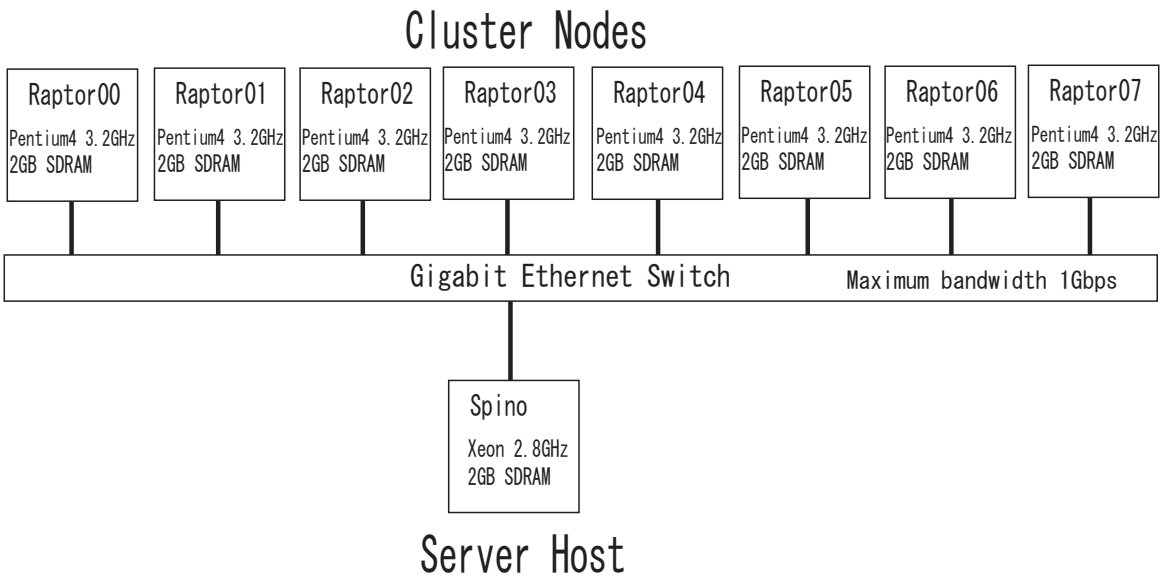


図 1　Raptor クラスタの構成

2.3.2 ネットワーク測定

ネットワーク測定では、クラスタのノード（Raptor00, Raptor01）間の通信速度を NetPIPE (A Network Protocol Independent Performance Evaluator) を用いて計測した。NetPIPE ではさまざまなプロトコルで計測できるが今回は TCP, MPI で測定した (図 2)。

計測環境

NetPIPE　3.6.2　<http://www.scl.ameslab.gov/netpipe/>

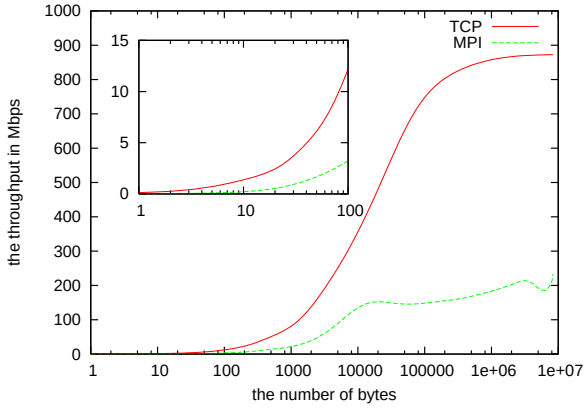


図 2　ブロックあたりのバイト数と通信速度

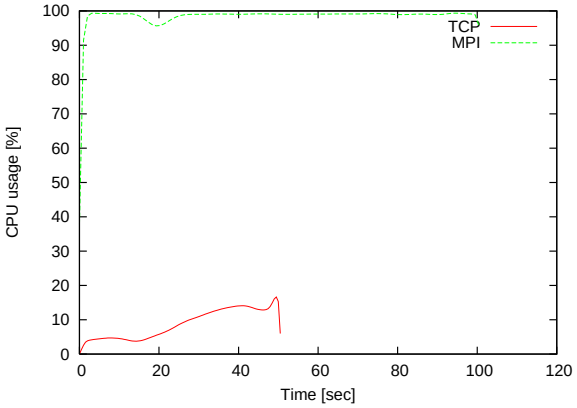


図 3　測定時の CPU 使用率

図 2 の結果から、TCP での通信速度に比べて MPI での通信が極端に遅いことが分かる。

この原因をいろいろと検討したが、CPU がボトルネックになっているのではないか疑った。なぜなら前に高速通信の実験を行った際に、負荷が思っていた以上に大きかったからである。

そこでネットワーク計測中の CPU 使用率を測定するために Linux コマンド top を用いて 0.5 秒間隔で CPU 使用率を計測した (図 3)。この方法では概略的なことしか分からないが、予測通りの結果を得た。

すなわち図 3 から分かるように CPU 使用率が TCP は 20% 以下なのに MPI では常時 100% に近い値を示し、CPU がボトルネックになっているとの確証を得た。

2.3.3　HPL Benchmark

クラスタの性能測定には HPL（A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers）を用いた。HPL とは密行列 LU 分解計算の浮動小数点演算実行速度を計測するプログラムである。

HPL の特徴として利用者がいろいろなパラメータを変更できるところにある。この中でも問題サイズは計算実行時間に重大な影響を及ぼし、搭載されているメモリサイズによって最適な問題サイズは異なる。

また HPL は世界最速を競う “TOP 500”（[5]）でも用いられている公式ベンチマークとなっている。“TOP 500” では、多くの計算機で HPL によって性能が測定された結果を公表している。

“TOP 500” の歴史は古く 1993 年より毎年 2 回ずつ更新され、その時代の最速の 500 台を知ることができる。そのため、このサイトからクラスタやスーパコンピュータの性能がどのように向上してきたかが分かる。

計測環境

HPL　1.0a　<http://www.netlib.org/benchmark/hpl/>
BLAS　GotoBLAS　<http://www.tacc.utexas.edu/>
gcc, g77 **最適化オプション**　*-fomit-frame-pointer -O3 -funroll-loops*
Linpack **共通パラメータ**　P = 2, Q = 4, PFACT = *Crout*, NBMIN = 4, NDIV = 2, RFACT = *Right*, BCAST = 2*ringM*, DEPTH = 0, SWAP = *Mix* (*threshold* = 64), L1 = *transposed form*, U = *transposed form*, EQUIL = *yes*, ALIGN = 8 *double precision words*

8 ノードで HPL をテストしたときの結果を以下の図に示す。FLOPS (Floating point number Operations Per Second) の最大値を求めるために、まず N を 1000 ずつ変化させ最大となる N を求めた (図 4)。これとは別に最適な NB を求めるため、N を 20000 で固定し NB を変化させ最大となる NB を求めた (図 5)。

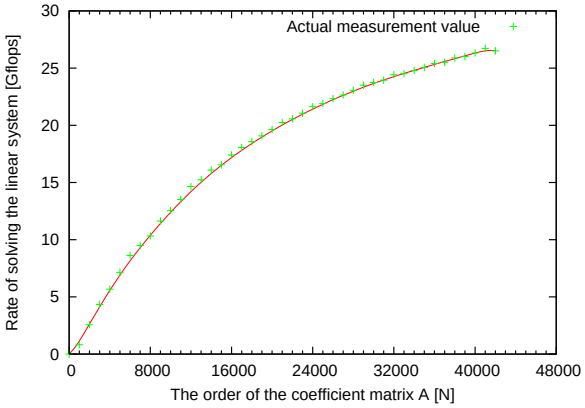


図 4　N と FLOPS との関係 (NB = 200)

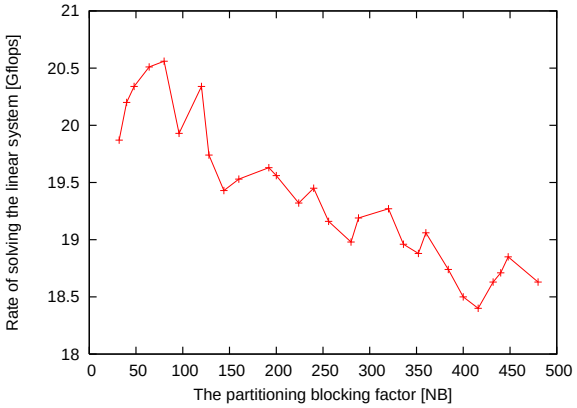


図 5　NB と FLOPS との関係 (N = 20000)

上記のように FLOPS を最大化させる N と NB が確定 ($N = 42000, NB = 80, 120, 200$) し、今度はこの条件で FLOPS を測定した (表 1)。HPL で結果を左右する重大なパラメータは問題サイズ (N) とブロック分割サイズ (NB) である。そのためそれ以外のパラメータは上述のように固定した。

表 1　最大 FLOPS の決定表

NB	80	120	200
FLOPS	27.58	27.78	26.51

上記の表 1 から、最大 FLOPS は 27.78[Gflops] となった。

HPL Benchmark まとめ

実験を行った結果、最大となる FLOPS(R_{max}) は 27.78[Gflops] (N（問題サイズ） = 42000, NB（ブロック分割サイズ） = 120) となった。Raptor Cluster の性能の理論値 (R_{peak}) は、51.2Gflops (3.2GHz × 2 × 8$nodes$[*1]) となるので今回の測定で 54%($R_{max}/R_{peak} \times 100 = 27.78/51.2 \times 100 = 54.257\cdots$) の性能を引き出せたこととなる。

また同様に 1 ノード ($N = 10000, P = 1, Q = 1$), 2 ノード (18000, 1, 2), 4 ノード (27000, 2, 2) についても実験を行った（括弧内で示した条件以外は 8 ノード実行時と同様)。

これを 2006 年 11 月に発表された “TOP 500” の最新のリストと比べると以下の表 2 のようになる。

表 2　HPL Benchmark 実行結果まとめ（R_{max}: 測定値, R_{peak}: 理論値, 2006 年 11 月現在）

Rank	Computer	Processors	Year	R_{max}[Gflops]	R_{peak}[Gflops]
1	BlueGene/L - eServer Blue Gene Solution	131072	2005	280600	367000
9	TSUBAME Grid Cluster	11088	2006	47380	82124.8
14	Earth-Simulator	5120	2002	35860	40960
500	Blade Cluster BL-20P	800	2005	2736.9	4896
—	Raptor Cluster（8 ノード）	8	2006	27.78	51.2
—	Raptor Cluster（4 ノード）	4	2006	16.50	25.6
—	Raptor Cluster（2 ノード）	2	2006	8.930	12.8
—	Raptor Cluster（1 ノード）	1	2006	5.202	6.4

また実験からどの程度の性能を引き出したのかを調べた。なお性能測定には $A_n = R_{max}/R_{peak} \times 100$（$n = 1, 2, 4, 8$ ノード）という指標を用いた。

[*1] 動作周波数 × 浮動小数点演算器 × ノード数によって計算。

この指標によると、$A_1 = 81.281\%$, $A_2 = 69.765\%$, $A_4 = 64.453\%$, $A_8 = 54.257\%$ となった。ノード数を増やすごとに、理論値からの乖離が目立つ結果となった。

この結果について、1 ノード実行で約 81% の性能であることから考えると、2 ノード実行では約 66% 、4 ノード実行では約 43% 、8 ノード実行では約 19% と理論的にはなるはずである。

しかし現実にはそうなっていないのは、HPL Benchmark の特徴として、性能は問題サイズ（メモリサイズに依存）に影響を受けるためである。具体的には、問題サイズを増加させると効率が上がり、結果として、性能が向上するからである。

3 Ogg Vorbis エンコーダ

3.1 Ogg Vorbis とは

Ogg Vorbis は、非営利組織クシフォフォルス財団によって開発されたパテントフリーの音声ファイルフォーマットである。Ogg Vorbis で 1 つのオーディオ圧縮コーデックを表すわけではなく、Vorbis という非可逆オーディオ圧縮コーデックと Ogg というコンテナフォーマットを合わせたものである。

Ogg はコンテナにすぎないので、ビデオ圧縮コーデックの Theora 等を入れることができるが、一般に Ogg といえば Ogg Vorbis を指す。

Vorbis の開発が始まったのは 1999 年で、広く使われていた MP3 (MPEG-1 Audio Layer 3) のライセンス料が変更されていきなりライセンス料が必要となったことが切っ掛けとなった。MP3 は特許の制限を受けるため、MP3 の代替として Vorbis は特許の制限を受けず自由に利用できる音声圧縮を目指して開発が続けられている。

Vorbis の仕様はパブリックドメイン、ライブラリは BSD-style ライセンスで関連するツールは GPL (GNU General Public License) で公開されている。また Ogg コンテナフォーマットは RFC 3533 (The Ogg Encapsulation Format Version 0) で正式に規定されている。

3.2 Ogg Vorbis エンコーダのアルゴリズム概要

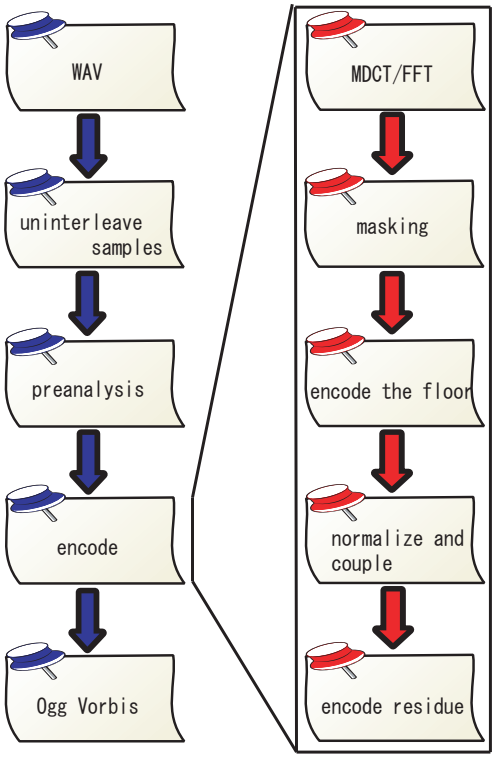
Ogg Vorbis のデコード処理については、Vorbis I 仕様書 [8] に詳しく書かれている。しかし、エンコード処理については全く触れられていない。つまり Ogg Vorbis は、デコード処理のみ決められているといえる。

デコード処理だけ仕様書によって決めれば、エンコーダはデコーダが処理できるようなデータを出力すればどのようなアルゴリズムを使っても良いということになる。

このことの利点として、エンコードのアルゴリズムを容易に変更できるということがある。事実、Ogg Vorbis エンコーダは公式のものを含めて数種類[*2] あるがどのエンコーダで作成したデータも同じデコーダでデコードできる。

具体的に、今回用いた公式のエンコードライブラリを用いたエンコーダのアルゴリズムは以下の図 6 のようになっている。上記の通りエンコード処理のアルゴリズムについては書かれていなかったので、ライブラリのソースコードを実際に調べた結果を書く。

[*2] 有名なものとして、aoTuV <http://www.geocities.jp/aoyoume/aotuv/>, Vorbis GT3 <http://www.sjeng.org/vorbisgt3.html>がある。



(a) 全体の流れ　(b) エンコード処理

表 3　encode 部分の用語解説

用語	解説
MDCT	修正離散コサイン変換 (Modified Discrete Cosine Transform) のこと。離散信号の時間領域を周波数領域へ変換する。
FFT	高速フーリエ変換 (Fast Fourier Transform) のこと。離散信号の時間領域を周波数領域へ変換する。
masking	不要な信号を取り除くこと。
floor	エンコード処理の基となるデータ。
normalize	音量を調整すること。
couple	Channel Coupling 処理のこと。ステレオ音声情報を効率的に符号化することによりビットレートを引き下げることができる。
residue	上記処理を終えた音声データのこと。

図 6　Ogg Vorbis エンコーダのアルゴリズム

図 6 の左側の流れ（青い部分）すなわちエンコード処理の全体像については以下のようになる。

3.2.1　uninterleave samples（データ変換）

libvorbis エンコードライブラリは、RAW 形式の WAV データを入力として扱うことができない。そこでエンコード前に処理を行いライブラリが扱えるデータに変換する。量子化数 16 ビットの WAV を実際に変換するときのアルゴリズムは以下の図 7 ようになる。

なお各変数について、$nSamples$ は WAV ファイルの入力サンプル数、$vi.channels$ は WAV ファイルのチャネル数を表す。データ構造として $pArray\,(data)$ は $nSamples \times vi.channels$ サイズの 1 次元 16 ビットの整数型で表現されている入力 WAV データ、$buf\,[j][i]$ は配列長 $nSamples$ と $vi.channels$ の uninterleave samples 後の 2 次元配列で表現された WAV データを表す。また演算子として、shl は左シフト演算を表している。

$$
\begin{aligned}
&i := 0;\\
&\textbf{while } i < nSamples \textbf{ do}\\
&\quad \textbf{begin}\\
&\qquad \textbf{for } j := 0 \textbf{ to } vi.channels - 1 \textbf{ do}\\
&\qquad\quad \textbf{begin}\\
&\qquad\qquad buf\,[j][i] := smallInt\,((pArray\,(data)\,[i\ shl\ vi.channels + j\ shl\ 1 + 1]\ shl\ 8)\ or\\
&\qquad\qquad\qquad\qquad\qquad\qquad pArray\,(data)\,[i\ shl\ vi.channels + j\ shl\ 1 + 0])\,/32768;\\
&\qquad\quad \textbf{end};\\
&\qquad i := i + 1;\\
&\quad \textbf{end};
\end{aligned}
$$

図 7　量子化数 16 ビットの WAV 変換処理

理論的には Vorbis は 255 チャンネルまで扱えるが、制約から実用上は 2 チャンネルの音声信号を扱う場合がほとんどである。そこで上の図 7 において、$vi.channels$ は 2 と等価となる。$vi.channels = 2$ とした場合、左側と右側の音声とを分けて 2 回処理されることとなる。

処理の前提として、量子化数 16 ビットの WAV データは 16 ビット（2 バイト）ごとに左右左右左 …… と格納されており、リトルエンディアンで記録されているということに留意しなければならない。そこで 2 バイト目のデータを 8 ビット左シフトしてから 1 バイト目のデータと論理和を取るわけである。

また量子化数 16 ビットの WAV データは符号付きの 16 ビットのデータなので、$-32768\left(-2^{16}/2\right) \leq pArray[\,][\,] \leq 32767\left(2^{16}/2-1\right)$ の範囲がある。よって $2^{16}/2 = 32768$ で割ることにより $-1 \leq buf[j][i] \leq 1$ のデータを得るのである。

簡単にまとめると、文字型で取得した 16 ビットを 16 ビットの整数型に変換し、さらに $2^{16}/2 = 32768$ で除して $-1 \leq buf[j][i] \leq 1$ の浮動小数点数を得ている処理と言える。

3.2.2 preanalysis（前処理）

関数 vorbis_analysis_blockout を呼び出すことにより実行され、主にエンコード処理の前処理を行う。例えば、エンコードに必要な音声データが十分なのかといったテストが行われる。

3.2.3 encode（データ圧縮）

関数 vorbis_analysis を呼び出すことにより実行され、エンコード処理の本体となる。プログラマーが呼び出すのはこの関数だが、エンコード処理のコアとなっている関数は mapping0_forward となっている。この関数を読むだけではエンコード処理の詳細を知ることはできないので、仕様書を見ながら読む必要がある。以下 encode 処理の具体的な中身を挙げる。

3.3 Ogg Vorbis エンコーダのエンコード処理

3.3.1 window the PCM data

図 6 の全体の流れでは書かなかったが、MDCT/FFT を行う前に入力 PCM データに窓関数を適用する必要がある。

窓関数とは以下のような定義となっている [2]。

> 窓関数とは、インパルス応答算出の基となった不連続的（方形的）な周波数特性に替えて、連続的な周波数特性を基にするもので、結果的には、インパルス応答の山裾をより速く 0 に漸近させるものである。

窓関数を適用する理由としては、音声データの一部を切り出したからということが挙げられる。実際に 1 回にエンコードされる音声は図 8 にあるように音声データの一部分のみである。図 8 からも分かるように音声の一部を切り出すとデータの両端に前後の波形の影響がでてしまう。

このことによって次から説明する MDCT, FFT などに影響を与える。そもそも MDCT, FFT 等の周波数解析は切り出したデータを 1 周期と見なしてこの周期が永遠に続いているという仮定を基に成り立っている。とすれば、図 8 の左端と右端が極端に異なる場合つなぎ合わせたときに極端な変化が生じることに繋がり、周波数解析を行った場合に高周波数成分に歪みが生じる。

そのまま処理を続けると、それ以降の処理（周波数分析）にも影響が出てしまう。結果としてエンコードの性能低下につながる。このことを防ぐために窓関数が適用される。Vorbis で適用される窓関数は以下の式 (1) の通りである。

$$f(x) = \sin\left(\frac{\pi}{2} sin^2\left(\frac{\pi}{K}(x+0.5)\right)\right), \quad 0 \leq x \leq K-1 \tag{1}$$

また窓関数適用でどのように入力 PCM 信号が変わったかを図 8 と図 9 で示している。

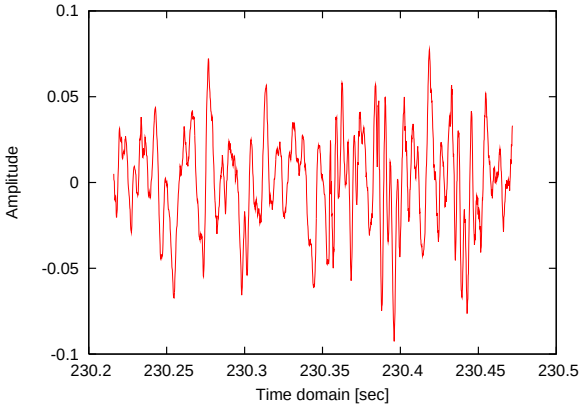


図 8　入力 PCM データ

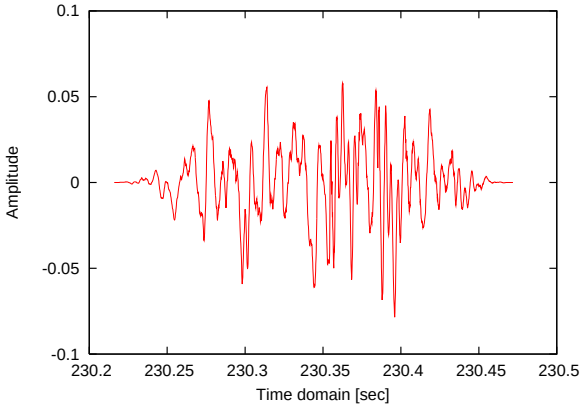


図 9　ウインドウ適用後の PCM データ

入力され uninterleave samples 後の音声データは図 8 になる。この音声データは全体の音声データの一部である。

図 8 の左端・右端に着目すると 0（無音）に収束しておらず、このままの状態で MDCT 等を行うと高音域にノイズが生じるため、正確に周波数分析できない。そこで、window 処理を行い端を 0（無音）に収束させる（図 9)。

3.3.2　MDCT

MDCT とは Modified Discrete Cosine Transform のことであり、修正離散コサイン変換と訳される。MDCT を行うことにより離散信号の時間領域を周波数領域へ変換することができる。

$x(k)$ を時間領域のサンプルとする。$x_t(k), 0 \leq k \leq n-1$ のサンプルはブロック番号 t の周波数領域のサンプル $X_t(k), 0 \leq k \leq n/2-1$ を計算するのに使われるとすると、MDCT の基本式は以下の式 (2) になる [4]。

$$X_t(m) = \sum_{k=0}^{n-1} f(k) x_t(k) \cos\left(\frac{\pi}{2n}\left(2k+1+\frac{n}{2}\right)(2m+1)\right), \quad 0 \leq m \leq \frac{n}{2}-1 \tag{2}$$

ただ式 (2) がそのまま用いられる訳ではない。実際には高度に最適化されたアルゴリズムが用いられている [4]。

STEP 0

窓関数が適用された時間領域のサンプル $x_k, 0 \leq k \leq n-1$ に対しては式を適用する。

$$u_k = \begin{cases} -x_{k+\frac{3n}{4}} & 0 \leq k \leq \frac{n}{4}-1 \\ x_{k-\frac{n}{4}} & \frac{n}{4} \leq k \leq n-1 \end{cases}$$

STEP 1

$0 \leq k \leq \frac{n}{4}-1$ のデータに対して以下の式を適用する。

$$\begin{aligned} v_{n-4k-1} &= (u_{4k} - u_{n-4k-1})\, A_{2k} - (u_{4k+2} - u_{n-4k-3})\, A_{2k+1} \\ v_{n-4k-3} &= (u_{4k} - u_{n-4k-1})\, A_{2k+1} + (u_{4k+2} - u_{n-4k-3})\, A_{2k} \end{aligned}$$

STEP 2

$0 \leq k \leq \frac{n}{8}-1$ のデータに対して以下の式を適用する。

$$\begin{aligned} w_{\frac{n}{2}+3+4k} &= v_{\frac{n}{2}+3+4k} + v_{4k+3} \\ w_{\frac{n}{2}+1+4k} &= v_{\frac{n}{2}+1+4k} + v_{4k+1} \\ w_{4k+3} &= \left(v_{\frac{n}{2}+3+4k} - v_{4k+3}\right) A_{\frac{n}{2}-4-4k} - \left(v_{\frac{n}{2}+1+4k} - v_{4k+1}\right) A_{\frac{n}{2}-3-4k} \\ w_{4k+1} &= \left(v_{\frac{n}{2}+1+4k} - v_{4k+1}\right) A_{\frac{n}{2}-4-4k} + \left(v_{\frac{n}{2}+3+4k} - v_{4k+3}\right) A_{\frac{n}{2}-3-4k} \end{aligned}$$

STEP 3

$0 \le l \le \log_2(n) - 4$ のデータに対して以下の式を適用する。ここで $k_0 := \frac{n}{2^{l+2}}$ $k_1 := 2^{l+3}$ とおく。
さらに $0 \le r \le \frac{n}{2^{l+4}-1}$ かつ $0 \le s \le 2^{l+1} - 1$ のデータに対して以下の式を適用する。

$$
\begin{aligned}
\hat{u}_{n-1-k_0 2s-4r} &= w_{n-1-k_0 2s-4r} + w_{n-1-k_0(2s+1)-4r} \\
\hat{u}_{n-3-k_0 2s-4r} &= w_{n-3-k_0 2s-4r} + w_{n-3-k_0(2s+1)-4r} \\
\hat{u}_{n-1-k_0(2s+1)-4r} &= \left(w_{n-1-k_0 2s-4r} - w_{n-1-k_0(2s+1)-4r}\right) A_{rk_1} \\
&\quad - \left(w_{n-3-k_0 2s-4r} - w_{n-3-k_0(2s+1)-4r}\right) A_{rk_1+1} \\
\hat{u}_{n-3-k_0(2s+1)-4r} &= \left(w_{n-3-k_0 2s-4r} - w_{n-3-k_0(2s+1)-4r}\right) A_{rk_1} \\
&\quad - \left(w_{n-1-k_0 2s-4r} - w_{n-1-k_0(2s+1)-4r}\right) A_{rk_1+1}
\end{aligned}
$$

STEP 4

$1 \le i \le \frac{n}{8} - 2$ のデータに対して以下の式を適用する。
$j = \mathrm{BITREVERSE}(i)$
$\mathrm{IF}(i < j)\mathrm{THEN}$

$$
\begin{aligned}
\hat{v}_{8j+1} &= \hat{u}_{8i+1} & \hat{v}_{8i+1} &= \hat{u}_{8j+1} \\
\hat{v}_{8j+3} &= \hat{u}_{8i+3} & \hat{v}_{8i+3} &= \hat{u}_{8j+3} \\
\hat{v}_{8j+5} &= \hat{u}_{8i+5} & \hat{v}_{8i+5} &= \hat{u}_{8j+5} \\
\hat{v}_{8j+7} &= \hat{u}_{8i+7} & \hat{v}_{8i+7} &= \hat{u}_{8j+7}
\end{aligned}
$$

STEP 5

$0 \le k \le \frac{n}{2} - 1$ のデータに対して以下の式を適用する。

$$\hat{w}_k = \hat{v}_{2k+1}$$

STEP 6

$0 \le k \le \frac{n}{8} - 1$ のデータに対して以下の式を適用する。

$$
\begin{aligned}
\tilde{u}_{n-1-2k} &= \hat{w}_{4k} & \tilde{u}_{n-2-2k} &= \hat{w}_{4k+1} \\
\tilde{u}_{\frac{3n}{4}-1-2k} &= \hat{w}_{4k+2} & \tilde{u}_{\frac{3n}{4}-2-2k} &= \hat{w}_{4k+3}
\end{aligned}
$$

STEP 7

$0 \le k \le \frac{n}{8} - 1$ のデータに対して以下の式を適用する。

$$
\begin{aligned}
\tilde{v}_{\frac{n}{2}+2k} &= \left(\tilde{u}_{\frac{n}{2}+2k} + \tilde{u}_{n-2-2k} + C_{2k+1}\left(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}\right) + C_{2k}\left(\tilde{u}_{\frac{n}{2}+2k+1} + \tilde{u}_{n-2-2k+1}\right)\right)/2 \\
\tilde{v}_{n-2-2k} &= \left(\tilde{u}_{\frac{n}{2}+2k} + \tilde{u}_{n-2-2k} - C_{2k+1}\left(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}\right) - C_{2k}\left(\tilde{u}_{\frac{n}{2}+2k+1} + \tilde{u}_{n-2-2k+1}\right)\right)/2 \\
\tilde{v}_{\frac{n}{2}+1+2k} &= \left(\tilde{u}_{\frac{n}{2}+1+2k} - \tilde{u}_{n-1-2k} + C_{2k+1}\left(\tilde{u}_{\frac{n}{2}+1+2k} + \tilde{u}_{n-1-2k}\right) - C_{2k}\left(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}\right)\right)/2 \\
\tilde{v}_{n-2k-1} &= \left(-\tilde{u}_{\frac{n}{2}+1+2k} + \tilde{u}_{n-1-2k} + C_{2k+1}\left(\tilde{u}_{\frac{n}{2}+1+2k} + \tilde{u}_{n-1-2k}\right) - C_{2k}\left(\tilde{u}_{\frac{n}{2}+2k} - \tilde{u}_{n-2-2k}\right)\right)/2
\end{aligned}
$$

STEP 8

$0 \leq k \leq \frac{n}{4} - 1$ のデータに対して以下の式を適用する。

$$X_k = \tilde{v}_{2k+\frac{n}{2}} B_{2k} + \tilde{v}_{2k+1+\frac{n}{2}} B_{2k+1}$$
$$X_{\frac{n}{2}-1-k} = \tilde{v}_{2k+\frac{n}{2}} B_{2k+1} - \tilde{v}_{2k+1+\frac{n}{2}} B_{2k}$$

定義

$0 \leq k \leq \frac{n}{4} - 1$ のデータに対して以下の式を適用する。

$$A_{2k} = \cos\left(\frac{4k\pi}{n}\right) \quad A_{2k+1} = -\sin\left(\frac{4k\pi}{n}\right)$$

$0 \leq k \leq \frac{n}{4} - 1$ のデータに対して以下の式を適用する。

$$B_{2k} = \cos\left(\frac{(2k+1)\,\pi}{2n}\right) \quad B_{2k+1} = \sin\left(\frac{(2k+1)\,\pi}{2n}\right)$$

$0 \leq k \leq \frac{n}{8} - 1$ のデータに対して以下の式を適用する。

$$C_{2k} = \cos\left(\frac{2\,(2k+1)\,\pi}{n}\right) \quad C_{2k+1} = -\sin\left(\frac{2\,(2k+1)\,\pi}{n}\right)$$

図 9 の音声データに対して MDCT を適用すると、図 10 になる。

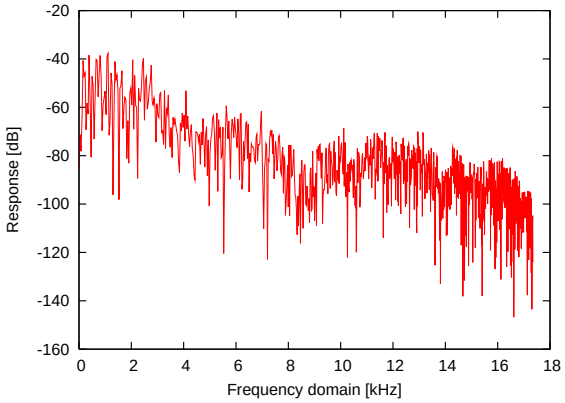


図 10　MDCT

3.3.3　FFT

FFT とは Fast Fourier Transform のことであり、高速フーリエ変換と訳される。FFT を行うことにより離散信号の時間領域を周波数領域へ変換することができる。

高速フーリエ変換とは文字通りフーリエ変換 (Discrete Fourier Transform; DFT) を高速化したものである。

フーリエ変換の基本式は式 (3) のようになる。

$$X(k) = \frac{1}{NT} \sum_{n=0}^{N-1} x(nT) e^{-j\frac{2\pi}{N}kn} \quad 0 \leq k \leq N-1 \tag{3}$$

これの式を高速化させたものを FFT と呼び、基本式は以下の式 (4) のようになる。

$$X(k) = \frac{1}{2}\left(X_0(k) + e^{-j\frac{2\pi}{N}k} X_1(k)\right) \tag{4a}$$

$$
\begin{aligned}
X\left(\frac{N}{2}+k\right) &= \frac{1}{2}\left(X_0(k) + e^{-j\frac{2\pi}{N}\left(\frac{N}{2}+k\right)}X_1(k)\right) \\
&= \frac{1}{2}\left(X_0(k) - e^{-j\frac{2\pi}{N}k}X_1(k)\right)
\end{aligned}
\tag{4b}
$$

適用するデータの範囲は、$0 \leq k \leq \frac{N}{2} - 1$ となる。

図 9 の音声データに対して FFT を適用すると、図 11 になる。

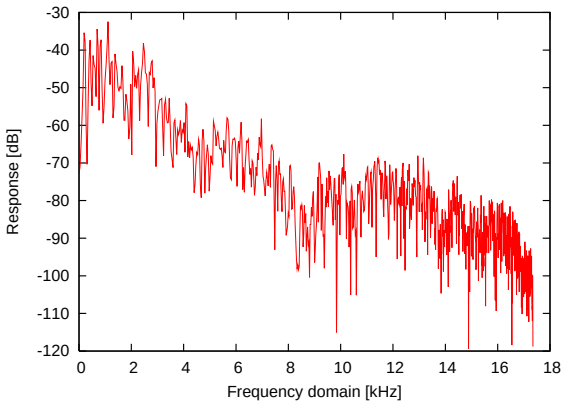


図 11　FFT

3.3.4　masking

masking とは、心理音響学により研究されてきた人間には聞こえない音声データを消去することによりデータ圧縮を行う処理である。

MDCT や FFT の結果を基に noise（図 12）と tone（図 13）を検出する。さらにその検出した noise と tone を基に mask（図 14）を計算する。

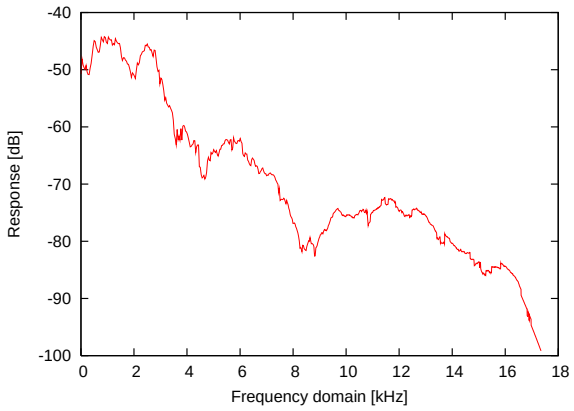


図 12　noise

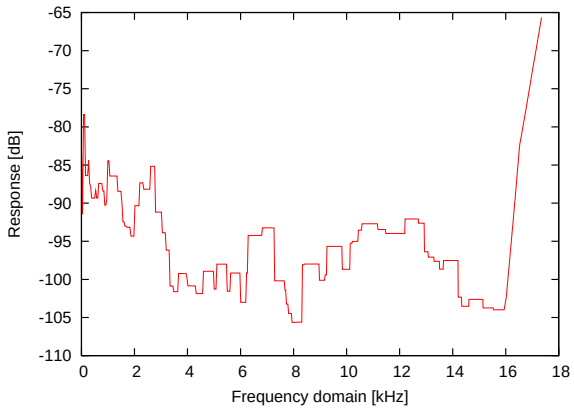


図 13　tone

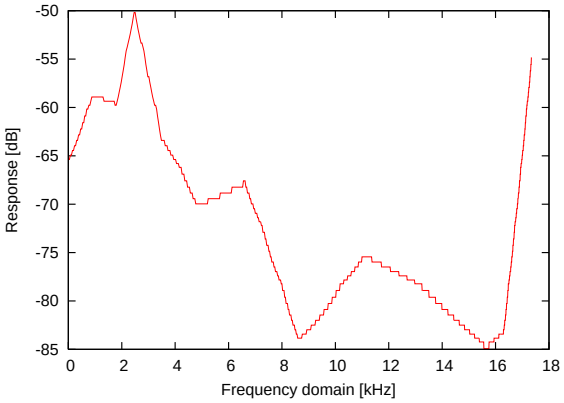


図 14　noise, tone から計算された mask

3.3.5　encode the floor

周波数領域の包絡線区分線形関数（低解像度のスペクトル）をハフマン符号化とベクトル量子化する。

3.3.6　normalize and couple

(1) normalize

normalize とはディジタル音声信号の音量を引き上げるまたは引き下げる処理である。通常は音声データの音量を最大レベルに引き上げることが行われている。

Ogg Vorbis エンコーダでも normalize 処理を用いているが、一般的に行われているように曲全体を normalize 処理するのではなく、曲の部分ごとにそれぞれに最適な音量に変換している。具体的には、曲の静かな部分では音量を引き上げる処理を行い、曲のうるさい部分では音量を引き下げる処理を行っている。

この処理によって、エンコードされた音声データは曲の最中で音量調節不要となり曲自体を引き立てる。

(2) couple

couple とはディジタルステレオ音声信号の左右の音声の特徴が似ている箇所をまとめることにより、データ量の節約をする処理である。このことにより節約したデータ量を用いてより多くのビットレートを割り当てられることになるため、音声データの高音質化に貢献する処理である。

Ogg Vorbis では、Dual Stereo, Lossless Stereo, Phase Stereo, Mixed Stereo が考えられている。

3.3.7　encode residue

スペクトルから低解像度のスペクトル（floor）控除後の residue をベクトル量子化とハフマン符号化する。

つまり最後に残った residue（図 15）が圧縮されエンコード処理は完結する。

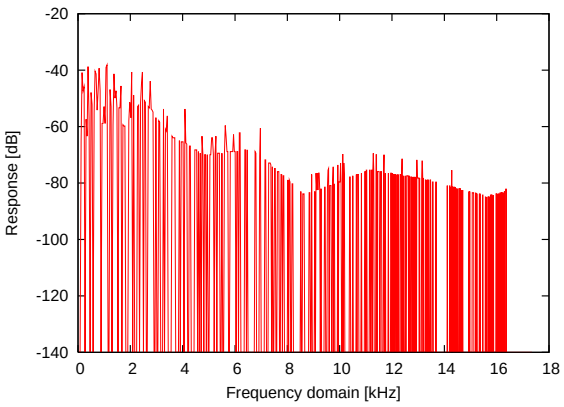


図 15　residue

3.3.8 エンコード処理実行割合

エンコーダによるエンコーダ処理の実行時間の割合を調べるためプロファイラ[*3] を用いて、実際にクラスタのノード (Raptor00) でエンコーダを実行し情報を得た。なお入力には 98 秒 (16922kbyte) の WAV ファイルを用いた。図 16 に結果を示す。なお簡略化のため重要性がある関数のみを書いた。

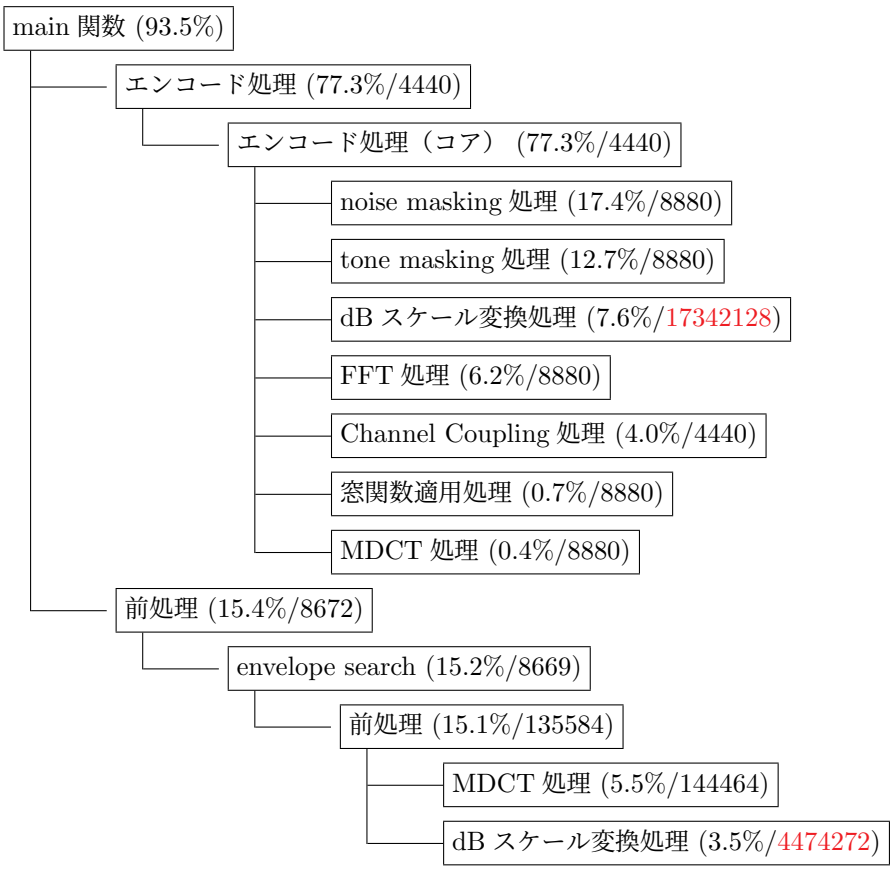


図 16　プロファイラの実行結果（全実行時間に対する割合 [%]/親関数からの呼び出し回数）

エンコーダは主に 2 つの部分に分けられる。1 つ目がエンコード処理本体と 2 つ目がその前処理となる。

Vorbis[*4] の特徴としては MDCT や FFT よりも Masking 処理に時間をかけていることが図 17 から分かる。具体的に説明すると、Masking 処理の総実行時間に占める割合は 30.1% (17.4% + 12.7%) にも及ぶ。これに対して離散周波数分析処理の総実行時間に占める割合は 12.1% (6.2% + 0.4% + 5.5%) にとどまる。

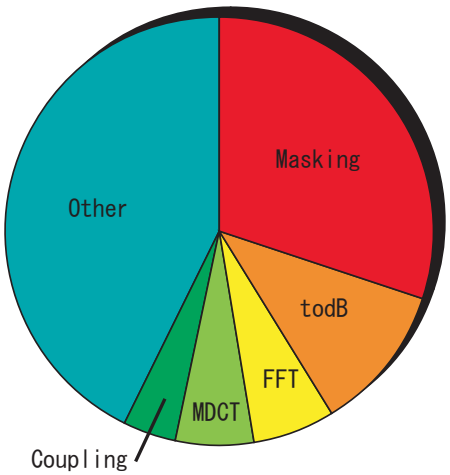


図 17　処理別のエンコーダ実行時間割合

Masking 処理の他に、例えば Channel Coupling 処理やその他より高音質で圧縮できるようにするための処理に多くの時間を割り当てていることが分かる。

あと、図 16 よりデシベルへのスケール変換 (関数 todB) の呼び出し回数が異常に多く合計すると 177895400(17342128 + 4474272) 回呼び出されていることも分かる。また総実行時間に占める割合は 11.1% (7.6% + 3.5%) になる。

今回は main 関数 (93.5%) 部分を並列化を行ったので、理論的には処

[*3] 関数の呼び出し関係や関数ごとの実行時間・呼び出し回数等のプログラムの挙動を解析するツール。
今回の測定には GNU gprof <http://www.gnu.org/>を用いた。

[*4] あくまで公式のエンコーダを用いた場合

理速度は $\frac{100}{6.5+93.5/n}$ 倍（n はノード数）となる。

4 Ogg Vorbis エンコーダの並列化手法

4.1 使用したソースコード

並列化を行うにあたり、Ogg Vorbis エンコーダを一から作成するのは大変なので以下のサンプルコードとライブラリを書き換えて MPI により並列化を実現した。

libvorbis-1.1.2　Vorbis 関連のライブラリ - BSD ライセンス - <http://xiph.org>
vorbis-tools-1.1.1　Vorbis 関連のツール（vcut を利用） - GPL ライセンス - <http://xiph.org>

今回用いたソースコードはそれぞれ BSD ライセンスと GPL ライセンスの元で公開されている。

4.2 分割処理の問題点と解決方法

並列化し最初に実行したときに以下の 3 つの問題点を見つけた。

(1) granulepos が正しくないため、総演奏時間が正しく表示されない

granulepos とは、ogg_packet に記録されているファイル先頭からの絶対値を示す。Ogg Vorbis では、総演奏時間を最後の ogg_packet の granulepos に基づいて計算している。計算方法は単純で最後の granulepos をサンプリングレートで除して ($total\ playback\ time = granulepos/sampling\ rate$) 求めている。

granulepos が正しくない原因としては、並列化を行うとそれぞれのノードが独自に granulepos をカウントしてしまうため granulepos の最終値が実際の $\frac{1}{ノード数}$ となってしまうためだ。これに影響して総演奏時間も実際の $\frac{1}{ノード数}$ となってしまう。そのためアプリケーションによっては正しい演奏時間が表示されなかったり（Winamp[*5] で確認)、正しい演奏時間が表示されず誤った演奏時間で中断するもの（Sound Player Lilith[*6] で確認）があった。

そこで granulepos が正しい値となるようにプログラムの修正を行い、正しい演奏時間で最後まで演奏できていることを確認 (Winamp) した。

(2) 音がとびとびに聞こえる

音がとびとびに聞こえる問題の原因として、libvorbis ライブラリ内の関数 vorbis_analysis_blockout (libvorbis-1.1.2/lib/block.c) で必要な音声データがそろうまでバッファリングしていることが挙げられる。バッファリングが問題となるのはノードの境目で、そこでバッファリングした音声データが次にノードに割り当てられたブロックの最初に出力されることが原因で、音がとびとびに聞こえるという問題が起こる。

そこで関数 vorbis_analysis_blockout でバッファリングを行わないようにソースコードを変更し、音飛びがないことを確認 (Winamp) した。

(3) その他の問題

その他ノードに割り当てるデータ量 (D) に起因する問題として以下のものがある。

1. D を小さくする（例: $D = 4096byte$）と音が切れ切れになる
2. D を極端に小さくする（例: $D = 512byte$）とエラーで止まる

1. の問題の原因としては、関数 vorbis_analysis_blockout( ) でバッファリングを行わないように変更したためである。そのためエンコードに必要な音声データの不足分をパディングで補ったため、無音のパディングの部分頻繁に発生し音が切れ切れに聞こえたからである。

[*5] 音楽ファイルを再生できる有名なマルチメディアプレイヤーソフト。様々なファイル形式の再生に対応している。
なお検証にはバージョン 5.3 を利用した。(Nullsoft, Inc. <http://www.winamp.com/>)

[*6] Winamp と同じマルチメディアプレイヤーソフト。
なお検証にはバージョン 0.991b を利用した。(Project9k <http://www.project9k.jp/>)

2. の問題の原因としては、D を極端に小さくしたためファイル入出力が間に合わなかった等の原因が推察される。

ただし D を小さくすると、ノード間通信が頻発するなどの影響で実行時間が極端に長くなるのは明らかなので無視することとした。

4.3 具体的な並列化手法

並列化の全体像は以下の図 18 のようになる。まず番号 0 のノードがハードディスクにアクセスし必要な WAV データを一括して取得する。次に番号 0 のノードは各ノードが必要な WAV データを送りエンコードさせる。そして各ノードでエンコードされた OGG データを番号 0 のノードに送り返す。番号 0 のノードは送られてきた OGG データをハードディスクに書き込む。以上のことを繰り返してエンコードを並列化して実行する。

また以下の図 19 にあるように番号 0 のノードが取得する WAV データはノード割当量 × ノード数となる。このうちノード割当量が 1 ノードに割り当てられる（ブロック分割）。

具体的な秒数で示すと、番号 0 のノードはノード割当量 × ノード数 / サンプリング周波数 [秒] の WAV データを読み込み、各ノードにノード割当量/サンプリング周波数 [秒] の WAV データを割り振ったことになる。例えば、ノード割当量が 200[KB] でノード数が 8 の場合は、番号 0 のノードが取得する WAV データは 6553600$(1024 \times 200 \times 4 \times 8)$[byte] となり、具体的には $37.1519\cdots((1024 \times 200 \times 4 \times 8)/(4 \times 44100))$ 秒となる。また各ノードには 819200$(1024 \times 200 \times 4)$[byte] すなわち $4.6439\cdots((1024 \times 200 \times 4)/(4 \times 44100))$ 秒が割り当てられることになる。

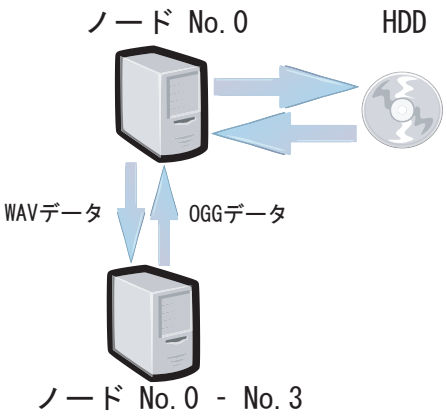


図 18　4 ノード実行時の並列化の全体像

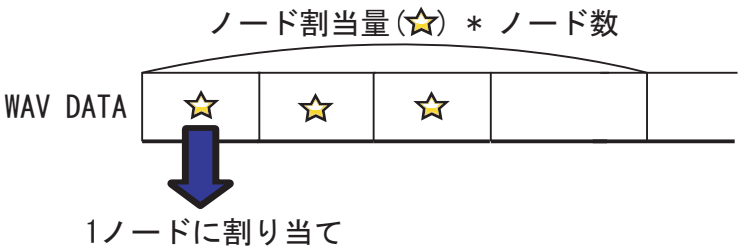


図 19　WAV データの割り当て方法

以上の方法はネットワークプログラムでのサーバ・クライアント方式に類似する方法である。具体的に異なる部分は、サーバにあたるノードでクライアントで行う処理を行うかどうかであり、今回の方法ではサーバに相当するノードでもクライアントと同様の処理を行っている。

5　実験と考察

5.1　実験環境

実験に用いたクラスタは、PC クラスタの性能測定で性能を計測した Raptor クラスタで行った。

また入力には 98 秒 (16922kbyte) の WAV ファイルを用いた。この WAV ファイルは CD 音源であり、44.1kHz でサンプリングされ 16bit で量子化されたステレオ音声のものである。よってこの WAV ファイルのビットレートは 1411kbps である。

今回、エンコーダのオプションには基本音質設定 0.4 を用いた。この設定で、およそ可変ビットレートの 128kbps の Ogg Vorbis ファイルにエンコードされるので、ファイルサイズは $\frac{1}{11}$ となる。

5.2　実験結果

並列化した Ogg Vorbis エンコーダを Raptor Cluster 上で実行した結果は以下のようになった。

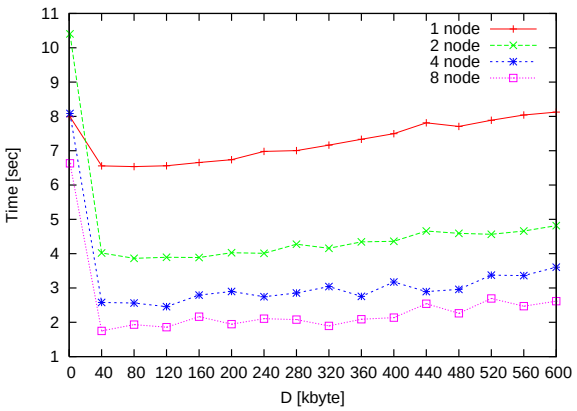


図 20　ノードに割り当てるデータ量と実行時間の変化

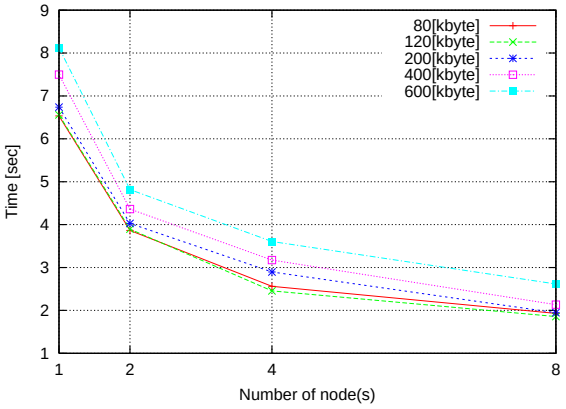


図 21　ノードに割り当てるデータ量を変えた時のノード数と実行時間の変化

実験はノード割当量と実行時間の関係、ノード数増減による実行時間の関係という 2 つの観点より行った。

並列化によってノード間通信という新たな現象が生じ、これはノード割当量というパラメータにより左右される。また並列化によって並列実行という新たな現象が生じる。これらのことから 2 つの観点により実験を行うことは有用であると考えた。

そこでノード割当量・ノード数と実行時間の関係を調べるため実験を行い結果を得た（図 20, 図 21）。また表 4 においてノード数増減による速度向上を調べた。

表 4　ノード数とノード割当量の関係

ノード割当量	1 ノード	2 ノード	4 ノード	8 ノード
80	6.536 (1)	3.865 (1.69)	2.562 (2.55)	1.931 (3.38)
120	6.536 (1)	3.865 (1.69)	2.562 (2.55)	1.931 (3.38)
200	6.739 (1)	4.028 (1.67)	2.897 (2.33)	1.946 (3.46)
400	7.498 (1)	4.362 (1.72)	3.175 (2.36)	2.135 (3.51)
600	8.125 (1)	4.817 (1.69)	3.607 (2.25)	2.616 (3.11)

表中数値は実行時間（秒）・括弧内は速度向上比（倍）

5.3　考察

以上のように、エンコーダを並列化し結果を得た。今回はネットワークプログラムのサーバ・クライアント方式に類似する方法（サーバにおいてもクライアントと同様の処理を行う点で異なる）で処理を行った。この方法では、サーバにかかる負荷とクライアントにかかる負荷が異なるため、それぞれを分けて考える。

まずサーバ側について考える。サーバにおいては、通信に占める割合は 50% 前後とほぼ等しい結果となった（図 22）。このことから、通信時間が実行時間に重大な影響を及ぼすと言える。今、図 22 より通信時間の割合がデータ割当量の増減に影響をあまり受けていないので、実行時間に重大な影響がないかのように思われる。しかし、エンコーダ実行時間はデータ割当量の増減に影響を受けている（表 4）。このことについてクライアント側から考察を行う。

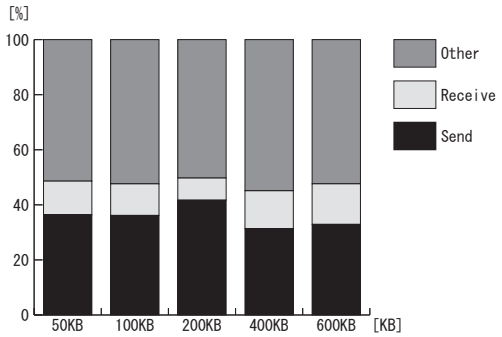


図 22　データ割当量と Raptor00（サーバ）におけるエンコーダ実行時間の割合（8 ノード）

表 5　ノード（Raptor00 除く）におけるエンコーダ実行時の分析

ノード割当量 [KB]	送信時間 [sec]		受信時間 [sec]		送信データ量 [KB]		受信データ量 [KB]	
	平均	標準偏差	平均	標準偏差	平均	標準偏差	平均	標準偏差
50	0.013	0.004	0.750	0.012	181.924	3.038	2400	0
100	0.055	0.039	0.847	0.007	178.361	6.966	2800	0
200	0.131	0.099	1.208	0.017	176.621	30.633	3200	0
400	0.239	0.132	1.447	0.118	166.385	47.618	4800	0
600	0.292	0.318	1.474	0.036	175.277	67.045	4800	0

8 ノード, 送信: ノードからサーバ, 受信: サーバからノード

クライアント側の実行結果を分析したのが表 5 となる。上述のようにデータ割当量によって通信時間の割合に変動が生じなかったことから、エンコーダ実行時間は通信時間自体に影響を受けたと考えられる。このことは表 5 からも証明できる。すなわち、表 5 の送信時間に着目すると送信時間の平均は確かにノード割当量に応じて増加している。しかしノード割当量を増減させても結果的に送信するデータは、表 5 の送信データの平均の欄からも明らかなようにあまり変動しないことからすると不自然である。

そこで表 5 で送信時間の標準偏差を計算した。これによれば、ノード割当量増加に伴って標準偏差が増加しノード間に送信時間の差異が生じていることが分かる。つまりサーバに集中してデータが送信されることで、データ送信ノード以外が待機状態になり送信時間にばらつきが生じたと考えられる。

以上の考察から、ノード間の通信時間にばらつきが生じ、またノード割当量を増加させるに伴ってばらつき（標準偏差）が増大したことが分かる。エンコード処理は全てのノード間通信がサーバ側で完結しないと終了しないので、ばらつきが増大することは不利である。よって、クライアント側でのこのばらつき増減がエンコーダ実行時間全体に影響を与えたと言える。

6 おわりに

今回の研究では 2 つのことに挑戦した。1 つは PC クラスタの性能測定を行うことであり、もう 1 つは Ogg Vorbis エンコーダの並列化である。

まず PC クラスタの性能を実際に測定することで、クラスタとはどれほど高速なのかという疑問が解けた。具体的には、HPL によって測定した Raptor Cluster（8 ノード）の性能は 27.78Gflops であり、また実測することにより今日のスーパコンピュータと呼ばれている上位 500 位との比較が可能となりよりクラスタの世界を身近に感じることができた。

次に Ogg Vorbis エンコーダの並列化を行うことにより、クラスタ上での並列プログラムの作成から性能測定までの一連の流れを確認できた。

Ogg Vorbis エンコーダの並列化をしているときは、並列効果が得られるのだろうかという不安があったが、最終的な並列化によって 3.51 倍（8 ノード）という性能が得られた。また Ogg Vorbis エンコーダのソースコードを読み進めることによって、音声圧縮についての知識が得られ、そのことによっておもしろい分野であると感じた。しかし、音声圧縮は心理音響学の分野などに深く関係し、さらに Ogg Vorbis では最新の技術が用いられていることから理解できない部分もあった。そこで卒業してから、それらの分野について勉強を行っていきたいと感じた。

謝辞

本研究の機会を与えて下さり、数々の助言を頂きました山崎勝弘教授に深く感謝致します。

また、本研究に必要不可欠なクラスタの整備をなされている Duy 氏、及び色々な面で貴重助言や励ましを下さった研究室の皆様に心より深く感謝いたします。

参考文献

[1] PC クラスタコンソーシアム.
http://www.pccluster.org/.

[2] 兼田護: ディジタル信号処理の基礎, 森北出版, 2000.

[3] Erik Montnemery and Johannes Sandval: Ogg/vorbis in embedded systems, Master's thesis, 2004.

[4] T. Sporer, K. Brandenburg, and B. Edler: The use of multirate filter banks for coding of high quality digital audio, the 6th European Signal Processing Conference, pp. 211–214, 1992.

[5] Top500 supercomputer sites.
http://www.top500.org/.

[6] Ye Wang, Leonid Yaroslavsky, Miikka Vilermo, and Vaananen: Some peculiar properties of the mdct.

[7] Keith Wright: Notes on ogg vorbis and the mdct, 2003.
http://www.free-comp-shop.com/vorbis.html.

[8] Xiph.org foundation.
http://www.xiph.org/.

[9] 井澤裕司: ディジタル信号処理（基礎編）. 信州大学工学部
http://laputa.cs.shinshu-u.ac.jp/~yizawa/InfSys1/basic/index.htm.

付録 A　並列化したエンコーダの正当性の検証

今回作成した Ogg Vorbis エンコーダで実際に問題点が解決されているか確かめるために、音声ファイル (98 秒) を入力し OGG ファイルを得た。

OGG ファイルの情報を取得できる ogginfo[*7] でその OGG ファイルを調べると、4 ノードでの実行時の問題点修正前 (図 23) と後 (図 24) の情報は以下のようになった。

```
01: [haru@a00 examples]$ ogginfo TEMP.ogg
02: Processing file "TEMP.ogg"...
03:
04: New logical stream
05: (#1, serial: 0076adf1): type vorbis
06: Vorbis headers parsed for stream 1,
07: information follows...
08: Version: 0
09: Vendor: Xiph.Org libVorbis I 20050304
10: Channels: 2
11: Rate: 44100
12:
13: Nominal bitrate: 128.000000 kb/s
14:
15: <!-- 中略 -->
16:
17: Warning: granulepos in stream 1
18: decreases from 511744 to 128
19: Warning: granulepos in stream 1
20: decreases from 1023744 to 511872
21:
22: <!-- 中略 -->
23:
24: Vorbis stream 1:
25:         Total data length: 1468358 bytes
26:         Playback length: 0m:23.219s
27:         Average bitrate: 505.895217 kb/s
28: Logical stream 1 ended
29: Warning: illegally placed page(s) for
30: logical stream 1
31: This indicates a corrupt ogg file:
32: Page found for stream after EOS flag.
33: [haru@a00 examples]$
```

図 23　問題点修正前

```
[haru@a00 examples]$ ogginfo test322.ogg
Processing file "test322.ogg"...

New logical stream
(#1, serial: 0076adf1): type vorbis
Vorbis headers parsed for stream 1,
information follows...
Version: 0
Vendor: Xiph.Org libVorbis I 20050304
Channels: 2
Rate: 44100

Nominal bitrate: 128.000000 kb/s

<!-- 中略（エラーなし） -->

Vorbis stream 1:
        Total data length: 1354372 bytes
        Playback length: 1m:38.201s
        Average bitrate: 110.334072 kb/s
Logical stream 1 ended
[haru@a00 examples]$
```

図 24　問題点修正後

これらの図より問題点が解決されていることが分かる。まず問題点修正前 (図 23) の 17〜20 行目にあった警告文が修正後 (図 24) では修正されていることが分かる。

また問題点修正前 (図 23) の 26 行目にある演奏時間が 23.219 秒 (実際入力 98 秒) とおかしいのに対して、問題点修正後 (図 24) の 19 行目にある演奏時間は 98.201 秒と正しいことが分かる。

以上より、作成したエンコーダの正当性を検証した。

[*7] Xiph.Org Foundation によって開発された vorbis-tools のプログラム群の一つ。Ogg ファイルの情報を取得したり、正当性を検証できる。