汎用グラフィクスカードを用いた 並列ボリュームレンダリングシステム

丸山悠樹† 中田智史† 高山征大†
篠本雄樹† 五島正裕† 森眞一郎†
中島康彦†† 富田眞治†

本稿では汎用グラフィクスハードウェアを用いて, 並列ボリュームレンダリングを行うシステムの実装について報告する. 実装したシステムでは大規模かつ高速な描画を行うために, 適応的サンプリングによるデータの圧縮と, 中間画像の圧縮による通信時間の削減を行った. この結果, $512 \times 512 \times 1024$ のボリュームデータに対し, ほぼ実時間でのレンダリングが可能となった.

A Parallel Volume Rendering System with Commodity Graphics Hardware

Yuki Maruyama ,† Satoshi Nakata ,† Motohiro Takayama,†
Yuki Shinomoto,† Masahiro Goshima ,† Shinichirou Mori ,†
Yasuhiko Nakashima†† and Shinji Tomita †

In this paper, we report the implementation of parallel volume rendering systems using commodity graphics hardware. In order to render large volume data with high frame rate, we have also implemented two kinds of data compression schemes: volume data compresson based on adaptatively sampling technique and intermediate-image compression to reduce communication time. Consequently, we could realize real time rendering of $512 \times 512 \times 1024$ volume data using four commodity graphics hardwares.

1. はじめに

近年の計算機処理能力の向上による大規模シミュレーションシステムの実用化に伴ない, より大規模な3次元データの解析を支援する可視化システムの実用化が求められている. このような大規模な3次元データの解析を支援する可視化方法の一つとしてボリュームレンダリングが挙げられる. ボリュームレンダリングを用いることにより, 複雑な3次元構造の理解が容易となるため, 工学, 医学などの分野で幅広く利用されている. しかし, 膨大な計算量が必要とされるため, 専用ハードウェアを用いる場合を除き, リアルタイムに可視化することは一般に困難であった. しかし, 汎用グラフィクスハードウェアの機能の向上とともに, その機能を利用してボリュームレンダリングを行うことが可能となってきており, これを並列化して用いることで大規模なデータの可視化が可能となった.

本稿では, このような汎用グラフィクスカードを用いた並列ボリュームレンダリングシステムを ATI 社の RADEON9700PRO ならびに NDIVIA 社の GeForceFX5950Ultra を用いて実際に構築し, その評価を行なった結果を報告する.

以下, 2 章で研究の背景となるボリュームレンダリングについて説明し, 3 章で汎用グラフィクスハードウェアによるボリュームレンダリング手法について述べる. 4 章では並列化手法について述べ, 5 章で具体的な実装について述べた後, 6 章で評価結果を示す. 7 章ではシステムの実装および評価結果をふまえた考察を行なうとともに関連研究について言及し, 8 章で結論を述べる.

† 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
†† 京都大学大学院経済学研究科/JST
Graduate School of Economics, Kyoto University/JST

2. Volume Rendering

我々が対象としているボリュームレンダリングは3次元のスカラー場をボクセルの集合として表現し, 2次

元平面へ投影することにより, 複雑な内部構造や動的特性を可視化する手法である. ボリュームレンダリングは大別して, すべてのサンプル点の寄与を計算して全体を表示する直接法と, 前処理によって表示する情報を抽出してデータの一部分を表示する間接法の二種類に分類され, 通常ボリュームレンダリングという場合は直接法のことを指す. 直接法によるボリュームレンダリングでは対象空間内のボクセルすべての寄与を計算して 2 次元平面へ投影する. このため膨大な計算時間と記憶容量が必要とされ, 大型計算機や特殊ハードウェアを用いる場合に利用が限られており, リアルタイムに可視化☆ することは一般に困難であった.

3. Texture Based Volume Rendering

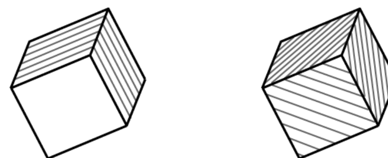
ボリュームレンダリングは描画面の各画素から視線方向に沿ってボクセル値の持つ色情報を積分していく. これを離散化すると, スクリーン上の各ピクセルごとに発生する視線に沿って, 視線と交差するボクセル値のサンプリングを視線上のボクセルがなくなるまで繰り返し, ピクセル値を求めることになる. この方法は視点から近い順にサンプリングする方法 (front to back) と, 視点から遠い順にサンプリングする方法 (back to front) に分けられる. back to front の場合, ボクセルの値を視点に近い順から, v_0, v_1, $\cdots$, v_n とし, RGB の各色情報 c_k と不透明度 α_k がボクセル値 v_k の関数で表されるとすると, ピクセル値は

$$P = \sum_{i=0}^{n} \alpha(v_i) c(v_i) \prod_{j=0}^{i-1} (1 - \alpha(v_j)) \qquad (1)$$

と表される. このピクセル値計算式は累積値 C_k を用いて次式のような漸化式に変形される.

$$C_{k-1} = \alpha(v_i) c(v_k) + (1 - \alpha(v_i)) C_k \qquad (2)$$

ここで $P = C_0$ である. 式 (2) はボリュームのサンプリングを視線方向に従って一定のサンプリングで行い, 描画面に遠い方から順に RGB 値を α ブレンディングすることでボリュームレンダリングができることを示している. α ブレンディングとは二枚の画像の持つ RGB 値を, α 値の示す比率で線形内挿して, 合成画像の RGB 値を算出する方法である. ボリュームをある軸に対して垂直なスライスの重ね合わせで表現する. そのスライスをテクスチャとしてポリゴンにマッピングし, それらを視点から遠い順に順次 α ブレンディングすることでボリュームレンダリングを行う. この手法を用いることで, 汎用グラフィクスハードウェアの機能を利用した高速処理が可能である[1). グラフィクスハードウェアが 3 次元テクスチャをサポートしている場合には, 視線に対して垂直な面を用意し, ボリュームデータを 3 次元テクスチャとして扱う方法が可能である (図 1-(b)). 3 次元テクスチャが利用できない場合は, ボリュームデータを各座標軸に対して垂直なスライスとして 3 つ用意し, 視線方向とスライスの法線のなす角が一番小さい軸のスライスに対して, 2 次元テクスチャとしてマップする方法を用いる (図 1-(a)). 本稿で用いるグラフィクスハードウェアはともに 3 次元テクスチャをサポートしており, ボクセル値を 32bit の RGBA データ (各要素 8bit) として描画した場合, $256 \times 256 \times 256$ のボリュームデータをリアルタイムに描画することができる. したがって, 本稿では 3 次元テクスチャによるボリュームレンダリングを扱うことにする.



(a) 2 次元テクスチャ (b) 3 次元テクスチャ

図 1 テクスチャベースのボリュームレンダリング

しかし, さらに大きいサイズのボリュームデータを描画する場合, 演算速度, メモリバンド幅, メモリ容量の制限などがボトルネックとなり, 描画速度が低下する. RADEON9700PRO の場合, Core Clock: 325MHz, Pipeline Unit: 8, Memory Pipeline: 310MHz DDR, Memory: 256bit 128MB という仕様である. 一番のボトルネックはメモリ容量で $256 \times 256 \times 256$ のボリュームデータを描画することが限界であるが, 仮にメモリ容量の制限がなかったとしても, 演算速度は 2.6Gpixel/sec, メモリバンド幅は 19.8GB/sec であるため, 1024^3 のボリュームデータの場合, たかだか 5fps 程度しか描画できないということがわかる☆☆. このよ

☆ 本稿では 30FPS のフレームレートを仮定しているが, 1GB を越えるような大規模ボリュームデータに対しては 10FPS 程度でもリアルタイムと呼ぶことが多い (cf. IEEE Visualization 会議録等).

☆☆ 1 ボクセルを 4 バイトで表現した 3D テクスチャでは, N^3 サイズのボリュームデータに対応する 3D テクスチャサイズは $4N^3$ バイトとなる. 一回のレンダリングでボリュームデータ全体に対して一回だけアクセスが起ると仮定すると, 秒間 F 枚の描画性能を出すためには, $4FN^3$[Byte/sec] のメモリバンド幅が最低限必要である. 仮に N=1024 とした場合, 秒間 5 枚の描画でも 20GB/s のメモリバンド幅が必要である.

うに, 現状の汎用グラフィクスハードウェアの性能は大規模なボリュームデータの描画を行うにはまだまだ性能が不足しており, 複数の汎用グラフィクスハードウェアを用いた並列化を行う必要がある.

4. 並　列　化

汎用グラフィクスハードウェアのテクスチャマップ機能と α ブレンディングを用いることにより, ボリュームレンダリングの高速処理が可能になる. しかし, テクスチャを保持するグラフィクスハードウェアのメモリ容量には制限があるため, メモリ容量を上回るサイズのデータを描画することはできない. 描画するデータがグラフィクスハードウェアのメモリ容量を超えない大きさのテクスチャに分割して描画させることにより, そのままでは描画できない大きなサイズのデータを描画することが可能となる. しかし, 保持しているテクスチャデータのサイズがグラフィクスメモリの容量を超える場合, テクスチャデータはメインメモリーに格納され, テクスチャは描画に使われる度にグラフィクスカードに転送される. このため, テクスチャデータの転送時間がボトルネックとなり描画速度が急激に低下する. そこで複数の汎用グラフィクスハードウェアを用いて並列化を行う. これによりデータサイズの大きなボリュームデータを扱うことができる[2].

4.1 基 本 方 針

まず, $N_x \times N_y \times N_z$ のボリュームデータを x, y, z 方向に d 等分に分割し, P 台のノードそれぞれで発生させた d^3/P 個のプロセスにそれぞれを割り当てる. 各サブボリュームをそれぞれのグラフィクスカードに与えてボリュームレンダリングを行い, 中間画像を生成する (図 2). このようにして生成された d^3 枚の中間画像を 2 次元テクスチャとして扱い, 視点からの距離の遠いものから α ブレンドして一つの画像にまとめることにより最終結果を得る. 図 3 にシステム構成図を示す. 図の構成では Master ノードからの視線情報を基に 4 台の Slave ノードで構成する PC クラスタで中間画像を生成し, その結果をマスタに集めた後に最終合成が行なわれる. 今, ボリュームデータの分割数 d を 2 とすると, 図 3 のシステム構成では, 各 Slave ノードが 2 つのサブボリュームを担当することになる. 各スレーブノードの動作を図 4 に示す. なお, 現在の実装では描画処理と通信処理のオーバラップは行なっていない.

4.2 適応的サンプリング

並列化により大きなデータを扱うことができるが, さらに大きなデータを扱う方法として, ボリュームデータをブロックで分割し, ブロック内のデータの局所性を利用して, データ量を削減することでメモリ利用の効率化を行う適応的サンプリングという方法が提案されている[4]. この手法を用いることにより, ほとんど画質を劣化させることなく, データ量を大幅に削減することができる.

そこで, ボリュームデータの局所性を利用してデータ量を削減することにより, メモリ利用の効率化を行うことを考える. 他にもグラフィクスハードウェアのメモリ容量を上回るサイズの大きなデータを描画する方法として, テクスチャデータをメインメモリに保持

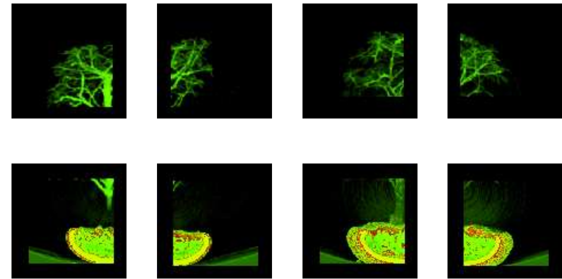


図 2　中間画像 ($d = 2$)

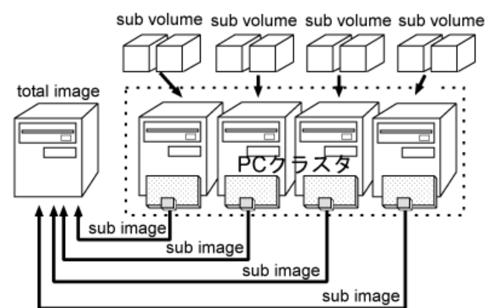


図 3　並列化

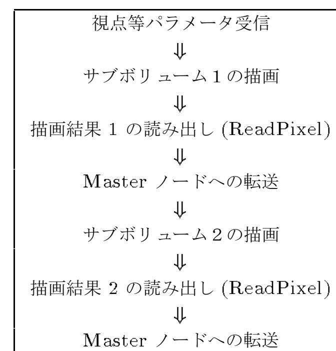


図 4　Slave ノードの一連の動作

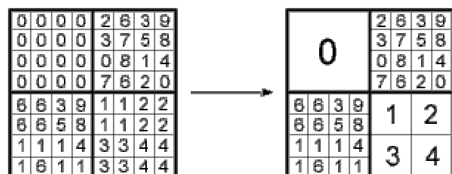


図 5 適応的サンプリング

しておき, グラフィクスメモリに容量を越えない量だけデータを転送させる方法が考えられるが, 今回使用する GeFoece4 Ti4600 の場合でも 1MB あたり 1.85msec 程度の転送時間を必要とすることがわかっている. この方法を用いる場合, 1 回の描画ごとにグラフィクスハードウェアにテクスチャデータを転送する必要があるため, $256 \times 256 \times 256$ のボリュームデータの場合, 1 フレームあたりの描画に少なくとも 116.4msec はかかってしまうため, 高速な描画処理を行うことはできない. リアルタイムに可視化するためには, 常に全てのデータをグラフィクスメモリに収まりきる形で保持しておくほうが望ましい. そこで, データを固定解像度のブロックで表現し, ブロック内の局所性を利用して, データ量を削減することでメモリ利用の効率化を行う.[4)]

$N_x \times N_y \times N_z$ のボリュームデータを x, y, z 方向それぞれに b 等分に分割した場合を考える. ただし, N_x/b, N_y/b, N_z/b は, OpenGL の制約上, 2 のべき乗である必要がある. このようにして分割された各ブロックに対して, x, y, z 方向それぞれのデータサイズを 1/2 のサイズとすることで解像度を順に一段階ずつ減少させ, その際に生じる元のデータとの誤差を計算していく. 解像度を下げる前の各ブロックのサンプル点のうち, 解像度を一段階下げたときの各ブロックのサンプル点に含まれるボクセル値の平均値を, 解像度を一段階下げたときのボクセル値と定義する. また, 誤差はサンプリングした値との自乗誤差と定義する. このようにして解像度の変更された各ブロックを視点からの距離の遠いブロックから順に描画していく. 図 5 に 2 次元平面に対して行った適応的サンプリングの例を示す. ブロック全体の誤差が 0 の場合には画質を全く変えずに, データ量を削減することができる. 全てのブロックで誤差計算を行い, 誤差が許容値以下であるブロックの解像度を下げていく. この操作を繰り返すことで, 全体としてデータのサイズを減少させていき, グラフィクスハードウェアで使用するメモリを最小限に抑えることができる.

ブロックに分割されたボリュームを V とし, ブロック内の座標 (i,j,k) におけるボクセル値を $v_n(i,j,k)$, 解像度を一段階下げたときのボクセル値を $v_{n-1}(i,j,k)$ とすると, 各ブロックの誤差は

$$\sum_{(i,j,k)\in V} |v_n(i,j,k) - v_{n-1}(i,j,k)|^2 \qquad (3)$$

と定義される.

実際のデータに対して誤差の許容値を変えながら, ブロック単位の適応的サンプリングを行い, データを削減した時の画質を評価した結果, ほとんど画質の劣化なく, データ量を大幅に削減することができることを確認した[9)].

我々の実装においては, 前節で述べた d^3 個のサブボリューム毎に独立に適応的サンプリングを行なう. また, 適応的サンプリングにより 1 つのサブボリュームが複数のブロックに分割されるが ($d < b$ の場合), サブボリューム内の全てのブロックの描画結果を一枚の中間画像に合成した後 Master ノードへ転送を行なう.

4.3 中間画像の圧縮による高速化

一般に断層撮影などから得られるボリュームデータは全体の 70% から 95% が透明領域である. このことから, 生成される中間画像には透明領域が多数含まれていることが多い. この透明領域では RGB の値はどのような値をとっても影響はない. したがって, 不透明度 A が 0 であるピクセルの情報を圧縮することにより, 通信データ量を削減することができる. 生成された中間画像のデータはすべてのピクセルの RGBA 値 (各 1Byte) が順に並んだ 1 次元配列である. $A = 0$ となるピクセルがいくつか連続するとき, 最初のピクセルの RGB 成分に相当する 3Byte 分の情報を使って, 連続するピクセルの個数を表し, 4Byte 目を 0 として圧縮することにより通信データ量の削減を行う (図 6). 例えば, $A = 0$ となるピクセルが N 個続いた場合, $4N$Byte から 4Byte へデータを圧縮することができる. この手法を用いることにより, 中間画像の通信によるオーバヘッドを低下させ, 描画の高速化を図る.

5. 実　　装

OpenGL1.2 グラフィクスライブラリを用いて PC クラスタ上に実装した. 今回行った実装は二種類の環境で評価を行う. 表 1 に示す実装環境の PC クラスタを PC クラスタ 1, 表 2 に示す実装環境の PC クラスタをクラスタ 2 とする. なお, Master は生成された中間画像を 2 次元テクスチャとして扱い, α ブレンドして一つの画像にまとめるためのノード, Slave はサブボリュームのボリュームレンダリングを行うためのノードである. Slave ノード数は特に断りのない場合

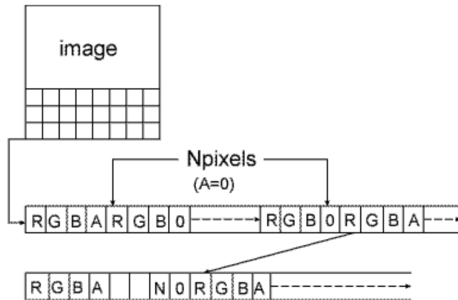


図 6　中間画像の圧縮

表 1　クラスタ 1 の実装環境

	Master
CPU	Pentium4 1.8GHz
Memory	512MB/PC133
Motherboard	DFI NB72-SC
Gfx Card	GeForce4 Ti4600
Gfx Memory	128MB DDR
Gfx Driver	nvidia Ver.1.0-5328 (2003/12/17)
OS	Red Hat Linux 8.0 (kernel 2.4.18)
	Slave
CPU	Pentium3 1.0GHz
Memory	512MB/PC133
Motherboard	EPOX EP-3S2A5L
Gfx Card	RADEON9700PRO
Gfx Driver	fglrx (2003/09/23)
Gfx Memory	128MB DDR
OS	Red Hat Linux 7.3 (kernel 2.4.18)
NODE	4 台
Network	100BaseTX Ethernet
(スイッチ)	PLANEX 社 FGX-08-TXS
(カード)	INTEL EXPRES-PRO-100 カード
(ライブラリ)	gcc+LAM6.5.9/MPI
(実測性能)	約 90Mbps (64KB 転送時)

は 4 台で測定したデータを用いている．両表でのネットワークの実測性能は，簡単な pinpong プログラムを用いて，128^2 ピクセルの中間画像に相当するサイズ (64KB) のデータを送受した場合の通信性能の実測値を示している．

OpenGL1.2 では，α ブレンディングを行う際に前後の 2 枚の画像が持つ RGBA 値に乗じる α 値を混合係数として設定する．色情報 (source) が，バッファに保存されている色情報 (destination) と，α ブレンドされて処理されるとすると，source と destination の RGB 値を C_s, C_d, α 値を A_s, A_d, 混合係数 (B_s, B_d) をとして，α ブレンド後のバッファの RGB 値と α 値は次式のように表される．

$$C = B_sC_s + B_dC_d \quad (4)$$

$$A = B_sA_s + B_dA_d \quad (5)$$

並列化を行う場合，スクリーンに対して後方にあるサブボリュームから生成した画像と，その前方にあるサブボリュームから生成した画像とを重ね合わせなければならない．そのため，各サブボリューム全体の不透明度を計算しておく必要がある．RGB 値と α 値の計算式は RGB 値の混合係数を $(A_s, 1-A_s)$ とし，α 値の混合係数を $(1, 1-A_s)$ として次式のように表される．

$$C = A_sC_s + (1-A_s)C_d \quad (6)$$

$$A = A_s + (1-A_s)A_d \quad (7)$$

OpenGL1.2 では混合係数は RGBA 値共通の値として設定され，RGBA の各要素ごとに異なる混合係数を設定することができないため，式 (5) の RGB 値と式 (6) の α 値を同時に求めることはできない．RGB 値と α 値を別々に求める方法も考えられるが，この方法だと α ブレンディングを 2 回行うことになるため処理速度が低下する．そこで，テクスチャデータが透明度を考慮したカラー値 (intensity) を持つというモデルであると考え，各ボクセルの持つ RGB 値に予め α 値を乗じておくようにする．このように，テクスチャデータの RGBA 値を (AR, AG, AB, A) としておき，混合係数を $(1, 1-A_s)$ として計算することで正確な画像を得ることができる．

表 2　クラスタ 2 の実装環境

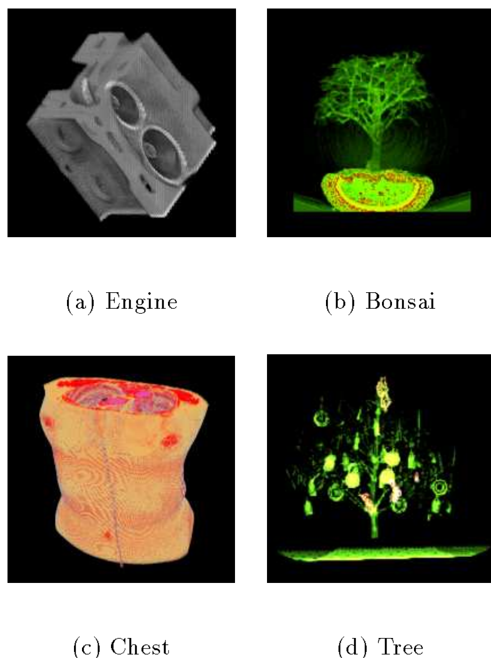
	Master
CPU	Pentium4 3.0GHz
Memory	1.0GB/DDR400
Motherboard	ASUS P4C800-E Delux
Gfx Card	GeForceFX5950 Ultra
Gfx Memory	256MB DDR
Gfx Driver	nvidia Ver.1.0-5328 (2003/12/17)
OS	Debian Linux (kernel 2.4.22)
	Slave
CPU	Pentium4 3.0GHz
Memory	1.0GB/DDR400
Motherboard	ASUS P4C800-E Delux
Gfx Card	GeForceFX5950 Ultra
Gfx Memory	256MB DDR
Gfx Driver	nvidia Ver.1.0-5328 (2003/12/17)
OS	Debian Linux (kernel 2.4.22)
NODE	4 台
Network	1000BaseT Ethernet
(スイッチ)	Bufferlo LSW-GT-8W
(カード)	Onboard CSA-base GbE コントローラ
(ライブラリ)	gcc+LAM6.5.9/MPI
(実測性能)	約 700Mbps (64KB 転送時)

6. 評　　価

6.1 評価データの仕様と測定方法

評価では，サイズの異なる以下の 4 種類のボリュームデータを使用した．Chest 以外は，ボリュームレンダリング処理の評価によく用いられるデータセットである．

- Engine : エンジンのボリュームデータでサイズ



(a) Engine (b) Bonsai

(c) Chest (d) Tree

図 7 ボリュームデータ

は $256 \times 256 \times 128$ である (図 7(a)). 3D テクスチャ表現では 32MB となる.

- Bonsai : 盆栽のボリュームデータでサイズは $256 \times 256 \times 256$ である (図 7(b)). 3D テクスチャ表現では 64MB となる.
- Chest : 人間の胸部のボリュームデータでサイズは $512 \times 512 \times 512$ である (図 7(c)). 3D テクスチャ表現では 512MB となる.
- Tree : クリスマスツリーのボリュームデータでサイズは $512 \times 512 \times 1024$ である (図 7(d)). 3D テクスチャ表現では 1GB となる.

不透明度については各ボリュームデータともにボクセル値と等しい値をとっている. スクリーンのサイズは各ボリュームデータともに 256^2, 中間画像を生成する各ノードのサブスクリーンのサイズは 128^2 とした. スライスの数はボリュームデータの各軸方向のサイズの最大値とした.

ボリュームレンダリングにかかる描画速度は視点を変更しながら 256 枚の描画を行い, それに要した時間から平均の描画速度を求めている☆. 特に指定しない場合, 視点はボリューム空間の中心を原点とし, 原点を通る水平面上 (z=0 の平面, z 軸は紙面の上下方向とする) の半径 R の円周上を $\frac{360}{256}$ 度づつ移動するものとする. なお R はボリュームデータの各軸方向のサイズの最大値の 2 倍とした.

6.2 単純な並列化の効果

並列化を行わずに 1 台で実行させた時の描画速度と, 適応的サンプリングも中間画像の圧縮も行わずに並列化を行ったときの描画速度を表 3, 表 4 に示す. 一括は 1 つのサブボリュームを単一の 3 次元テクスチャとしてグラフィクスカードで描画させた時の描画速度を, 分割は個々のサブボリュームをさらに 8 等分 (元のボリュームデータを 4^3 個の 3 次元テクスチャに分割したものと等価) して描画させた時の描画速度を示している. 表中の 「-」は, 3 次元テクスチャがグラフィクスカードのメモリに入りきらずに描画ができなかったことを示す.

クラスタ 1 では並列化による著しい通信時間のオーバヘッドが見られ, クラスタ 2 ではリニアではないものの並列化による速度向上がみられる. これはネットワークの通信性能の違いが原因である. 4 章で述べた通り, 今回の実装では Slave ノードで作成された中間画像 (128^2 ピクセル × 4B/ピクセル = 64KB) を全て Master ノードに送っている. 一回のボリュームレンダリング処理につき, Master ノードは 8 枚の中間画像を受け取ることになり合計で 512KB のデータを受信する必要がある. クラスタ 1 およびクラスタ 2 の通信時間の実測性能 (表 2 および 表 3 参照) を元に計算すると, このデータ受信に要する時間は, それぞれ 44ms および 5.7ms となる. したがって, ネットワーク性能に起因する描画速度の上限は, 22.5FPS ならびに 175FPS となる. これ以外にも, グラフィクスカードからの画像読み出し等のオーバヘッドが存在することから, 4 台構成のクラスタ 1 での性能低下の原因は合成処理であることが分る.

次に分割の効果を見ると, クラスタ 1 においては Engine, Bonsai のボリュームデータがグラフィクスメモリに入りきるサイズであるため, 分割して描画処理が増加したことによる描画速度の低下が見られる. 一方, Chest, Tree は一括の場合, 描画が不可能であるが, 分割することで低速度ながらも描画可能となった. さらに Chest は並列化によって使用可能なグラフィクスカードの総メモリ量が増えたため, Engine, Bonsai

☆ OpenGL の API を介してグラフィクスカードに送られた描画命令はパイプライン処理される. また, 処理の終了を通知する手段も別段設けられていないため, 1 枚の描画時間を計測することができない. そのためパイプラインが十分定常状態に落ち着く程度の連続描画を行ないそれに要した時間から平均描画速度を求めている.

表 3 単純な並列化の効果-クラスタ 1-

Data	描画速度 [fps]			
	1 台		4 台	
	一括	分割	一括	分割
Engine	102	39.9	17.7	18.0
Bonsai	78.3	34.8	17.2	17.6
Chest	-	0.066	-	13.2
Tree	-	0.007	-	0.77

表 4 単純な並列化の効果-クラスタ 2-

Data	描画速度 [fps]			
	1 台		4 台	
	一括	分割	一括	分割
Engine	34.8	31.1	106	97.4
Bonsai	28.5	24.5	92.4	84.1
Chest	0.49	0.77	31.8	27.9
Tree	-	0.05	0.73	10.7

に近い描画速度を達成している．クラスタ 2 でも同様の特徴が見られるが, グラフィクスメモリがクラスタ 1 の 2 倍あるため, 単純な並列化だけで Chest の高速描画が可能となっていることがわかる．

6.3 適応的サンプリングの効果について

適応的サンプリングを行うブロックの大きさは, 小さいと描画時のオーバーヘッドや使用メモリが増加し, 大きいとデータの局所性を利用しにくくなるため, 最適値は処理系とデータに依存する． 誤差の許容値を 5%, 分割数を $d = 2$, slave マシンを 4 台とし, 分割してできるブロックの数を変えながら, 描画速度, 元データに対する適応的サンプリング後のデータサイズの比率 (縮小率), 適応的サンプリングにかかった時間を比較した結果を表 5, 表 6 に示す．

クラスタ 1 においては, Engine, Bonsai はブロックの数が 4^3 の時, Chest, Tree はブロックの数が 8^3 の時が最も効率が良いことがわかる．一方, クラスタ 2 においては, Engine, Bonsai, Chest はブロックの数が 8^3 の時☆, Tree はブロックの数が 16^3 の時が最も効率が良い．

表からは, ボリュームデータがグラフィクスメモリに入りきらないデータ (Chest,Tree) に関してはデータを圧縮することにより高速描画が可能となること, また, ボリュームデータがグラフィクスメモリに入りきるデータ (Engine, Bonsai) に関しても, 分割して描画処理が増加したことによる描画速度の低下はブロック数 16^3 の場合を除きほぼ見られないことが確認できる．また, Tree に関してはブロック数が 8^3 から 16^3 に変るとボリュームデータが半分に圧縮されるにも関

☆ Engine に関しては表では 2^3 と同じであるが, 小数点以下まで見ると 8^3 がわずかに優れている

表 5 適応的サンプリングの効果-クラスタ 1-

Data	ブロック数	描画速度 [fps]	縮小率 [%]	前処理 [sec]
Engine	2^3	17.7	(非圧縮)	
	4^3	18.2	93.6	0.79
	8^3	16.9	50.2	1.14
	16^3	11.3	34.2	1.48
Bonsai	2^3	17.2	(非圧縮)	
	4^3	17.8	84.4	1.30
	8^3	15.8	53.9	1.66
	16^3	10.4	37.5	2.30
Chest	2^3	-	(非圧縮)	
	4^3	13.5	100	3.69
	8^3	15.8	77.2	6.03
	16^3	10.4	66.8	8.81
Tree	2^3	-	(非圧縮)	
	4^3	1.11	71.9	19.9
	8^3	12.7	44.9	20.9
	16^3	8.14	22.3	22.5

わらず, グラフィクスメモリが少ないクラスタ 1 でブロック数 8^3 が最適であるのに対して, クラスタ 2 ではブロック数 16^3 が最適となっている．これは, クラスタ 1 では分割に伴う処理に起因する Slave ノードの CPU 負荷の増加が, データ量が半分になることの効果よりも大きかったためである．以上のように, 適応的サンプリングを用いたデータ圧縮は, ブロック分割数を適切に設定することでよい効果的が得られることが分った．

しかし, 今回使用しているボリュームデータは時系列によって変化の起こらない静的なデータを用いており, シミュレーション結果の実時間可視化のように時間変化する動的なデータ可視化の場合は適応的サンプリングにかかる時間を考慮しなければならない．このため, 適応的サンプリングは静的なデータに対しては有効な手法であるといえるが, 動的なデータに対してはデータの更新頻度との関係で必ずしも有効でない場合も存在する．

6.4 中間画像の圧縮効果

中間画像の圧縮効果は, 透明領域が多ければ通信データ量を減らすことができるが, 透明領域が少ない場合は圧縮にかかる時間が逆にオーバヘッドとなってしまう．slave マシンを 4 台とし, 中間画像の圧縮を行った時と行わない時の描画速度の比較を行った結果を表 7 に示す．縮小率は中間画像が最も小さなサイズに圧縮された時と最も大きなサイズに圧縮された時の縮小率を示している．分割数は $d = 2$, 適応的サンプリングの誤差の許容値は 5%, ブロックの数はクラスタ 1 で Engine, Bonsai が 4^3, Chest, Tree は 8^3, クラスタ 2 で Engine, Bonsai, Chest が 8^3, Tree は 16^3 とした．クラスタ 1 では中間画像の圧縮による描画速度の高速

表 6　適応的サンプリングの効果-クラスタ 2-

Data	ブロック数	描画速度 [fps]	縮小率 [%]	前処理 [sec]
Engine	2^3	106	(非圧縮)	
	4^3	98.7	93.6	0.087
	8^3	106	50.2	0.260
	16^3	62.1	34.2	0.368
Bonsai	2^3	92.4	(非圧縮)	
	4^3	91.0	84.4	0.28
	8^3	100	53.9	0.47
	16^3	61.9	37.5	0.70
Chest	2^3	31.8	(非圧縮)	
	4^3	27.9	100	0.82
	8^3	33.6	77.2	2.02
	16^3	32.6	66.8	2.87
Tree	2^3	0.73	(非圧縮)	
	4^3	13.7	71.9	6.64
	8^3	19.2	44.9	8.11
	16^3	21.7	22.3	10.0

化が見られる．一方, クラスタ 2 では描画速度の高速化はほとんど見られず, むしろ低下してるものも見られる．

まずクラスタ 1 に着目すると, 6.2 節で議論したとおり, Master ノードの受信データ量とネットワーク性能で決まる非圧縮時の描画速度の上限は 22.5FPS であったのが, 圧縮することで仮にデータ量が 50%になると上限が 45FPS となる．表 7 の結果を見ると, Tree を除き, ほぼ圧縮率に比例した速度向上が得られており, このことから, クラスタ 1 ではグラフィクスカードの描画性能ではなく, 合成処理の性能が全体の性能を支配していることが分る．ただし, Tree に関してはメモリ容量に起因する描画速度の上限が支配的であり, 圧縮率に見合った速度向上は得られていない.

次にクラスタ 2 に着目すると, ネットワーク性能に起因する非圧縮時の描画速度の上限は 175FPS であった．したがってクラスタ 2 では非圧縮の状態でもネットワークの飽和状態は発生しない．しかしながら, 描画速度が高くネットワークが高負荷状態になる Engine および Bonsai では, 中間画像の圧縮により各々 10% および 5%の速度向上が得られている．一方, 描画速度が低くネットワークが低負荷状態である Chest や Tree では, わずかではあるが圧縮／解凍処理のオーバヘッドに起因する速度低下が起っている.

6.5　並列化の効果

ここでは, 適応サンプリングと中間画像の圧縮の両方を適用した場合における並列化の効果を評価する．並列化を行わずに 1 台で実行させた時の描画速度と並列ボリュームレンダリングを行ったときの描画速度を表 8 に示す．分割数は $d = 2$, 適応的サンプリングの誤差の許容値は 5%, ブロックの数は中間画像の圧縮効

表 7　中間画像の圧縮効果

Data	クラスタ 1 描画速度 [fps]		クラスタ 2 描画速度 [fps]		縮小率 [%]
	非圧縮	圧縮	非圧縮	圧縮	
Engine	18.2	34.6	106	116	48 ～ 79
Bonsai	17.8	38.1	100	105	37 ～ 64
Chest	15.8	27.5	33.6	33.4	42 ～ 82
Tree	12.7	14.4	21.7	21.6	41 ～ 73

表 8　並列化の効果

Data	クラスタ 1 描画速度 [fps]		クラスタ 2 描画速度 [fps]	
	1 台	4 台	1 台	4 台
Engine	34.5	34.6	35.0	116
Bonsai	34.7	38.1	31.4	105
Chest	0.45	27.5	1.35	33.4
Tree	0.33	14.4	6.30	21.6

果の実験で使った値を用いる．Engine, Bonsai, Chest に関してはほぼリアルタイムに可視化できていることがわかる．データのサイズが大きくなると描画速度が著しく低下してしまうが, これはグラフィクスカードが保持するテクスチャデータのサイズがグラフィクスメモリの容量を超えてしまったためだと考えられる．逆に, $512 \times 512 \times 512$ のサイズ以下のボリュームデータは各ノードの汎用グラフィクスカードのメモリ容量以下であったため, 高速な描画が可能であることがわかる．クラスタ 2 で 1 台構成の場合に, Tree が Chest よりも高速に描画ができるという逆転現象が起きているが, これは, 適応サンプリングの結果 Tree のデータサイズが元のデータの 22.3%に圧縮されたのに対して, Chest では 77.2%にしか圧縮されておらず圧縮後のデータサイズが逆転したのが原因である．同じ状況で 4 台構成の場合に逆転が起らないのは, データサイズの逆転によるメモリアクセス時間短縮の効果が 1 台の場合に比べて少ないからである.

7.　考察および関連研究

7.1　考　　察

本節では, 汎用グラフィクスカードを用いた並列ボリュームレンダリングシステムの実装を踏まえたいくつかの考察を行なう．表 9 は今回使用したグラフィクスカードの諸元である.

7.1.1　汎用グラフィクスカードの利用に関して

3D テクスチャを用いたボリュームレンダリングでは, 3D テクスチャをサンプリングしたスライス (2D テクスチャ) を一時的に作成しなければならないが (図 1), この際のメモリ・アクセスパターンは視点位置によって大きく変化する．したがって, メモリの実効転送速度が低下し, これによる描画速度の視点依存性が

表 9　グラフィクスカードの主な諸元

	RADEON 9700PRO	GeForce FX5950Ultra
Core Clock	325MHz	475MHz
Pixel Pipeline Unit	8	8
ピクセル計算性能	2.4G pixel/sec	3.8G pixel/sec
Memory Clock	310MHz DDR	475MHz DDR2
Memory bit 幅	256bit	256bit
Memory バンド幅	19.3GB/s	30.4GB/s
Memory 容量	128MB	256MB

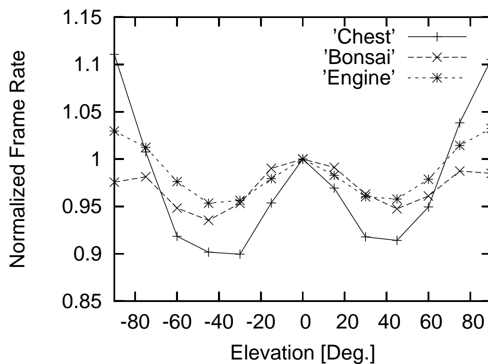


図 8　描画速度の視点依存性

発生する．さらに，グラフィクスカード内には 2D テクスチャのアクセスに最適化されたプリフェッチ機構付きキャッシュが存在しており，これも視点依存性を助長する一因となっている．視点位置に依存した描画性能の大幅な変動が許容できないシステムにおいては注意が必要である．

参考までに，クラスタ 2 上で実際に測定した視点依存性のグラフを図 8 に示す．いままでの測定では，z=0 平面上の半径 R の円周上で視点移動を行なったが，この円周を x 軸の回りに仰角 $\phi(90 \geq \phi \geq -90$, 15 度間隔) だけ回転し，その円周上で視点移動を行ない描画時間を測定したものである．測定は 4 ノード構成のクラスタ 2 において，何ら最適化を行なわない状態で行なった．サブボリュームの分割も行なっていない (表 4 の 4 台，一括に相当する構成)．

図の縦軸は仰角 0 度の場合の描画速度を 1 として正規化した描画速度である．Engine, Bonsai, Chest の 3 つのデータに対して計測を行なっているが，何れの場合も，±45 度付近で最も速度が低下している．この時，最も高速な場合に比べて，Engine, Bonsai では 5%, Chest では 20%もの速度低下が起っていることが分る．

7.1.2　最適化に関して

今回の実装では，データ圧縮に関する最適化として，ボリュームデータの圧縮ならびに 中間画像の圧縮を実装した．

今回は実装していないが，演算量そのもを減らす最適化手法として，早期視線終端 (Early Ray Termination:ERT)[5] がある．しかし，本来はピクセル単位の最適化であるため，スライス単位で処理を行なうテクスチャマッピングベースのボリュームレンダリングへはそのままは適用はできない．しかしながら，我々が提案した ERT-table 法[8] と，今回実装した適応的サンプリング法を組合わせることで，テクスチャマッピングベースのボリュームレンダリングアルゴリズムへの適用も可能であると考える．

次に，並列処理という観点では中間画像の合成処理オーバヘッドの軽減が必須である．今回の実装ではノード数が 4 台であったため単純な合成処理を実装したのみであるが，既に提案されている最適化手法等[6),7)]，を適用することでスケーラビリティの確保が期待できる．

7.2　関 連 研 究

テクチャマッピングを応用したボリュームレンダリング手法自体は 10 年近くまえに提案されていたが[12]，高価な専用グラフィクスアクセラレタを登載したハイエンドのシステムでしか利用できなかった．汎用グラフィクスカードにテクスチャマッピングや α ブレンディングの機能が登載され，かつ 3 次元テクスチャが扱えるようになった頃を契機に汎用グラフィクカードを用いたボリュームレンダリングシステムが多く開発／発表されるようになった (例えば文献 1)，13))．グラフィクスカードのメモリ不足問題に対して適応的サンプリング法を提案した文献 4) などもその一つである．

PC クラスタの各ノードに汎用グラフィクスカードを登載したシステムも複数存在している (例えば文献 3)，10)，11).)．文献 10) のシステムは我々同様，全てを汎用部品で構成している．このシステムは 2 世代前のグラフィクスカード (GeForce3) を用いた 2D テクスチャを前提にした実装であり，当時の環境下で何が性能のボトルネックになるかを議論している．文献 11) では，サブボリュームのレンダリングノードへの割当や部分合成の並列処理におけるスクリーンの分割法に関する議論を行なっている．文献 3) は，ボリュームレンダリング専用アクセラレータを登載したシステム[2] のグラフィクスハードウェア部分を汎用グラフィクスカードに置き換えたものである．ただし，部分画像の合成に専用のネットワークと専用のハードウェアを別途用意している．いずれのシステムもボリューム

データや部分画像の圧縮と組合わせた評価は行なっていない. また, 視点依存性に関する議論や OpenGL1.2 を用いた α ブレンディングを並列実装する場合の問題点の指摘はなされていない.

8. 結　　論

本稿では汎用グラフィクスカードを用いた並列ボリュームレンダリングシステムの実装の報告を行うとともに, ブロック化による適応的サンプリング, ならびに中間画像の圧縮を用いたデータ圧縮技術を併用した場合の性能評価結果を示した. この結果, サンプルとして使用したボリュームデータを 4 台のクラスタを用いてほぼリアルタイムに描画することができた. しかし, 時系列によって変化の起こりうる動的なデータの可視化には圧縮処理に時間がかかるため, 適応的サンプリングによるボリュームデータの圧縮方法は向かない. また, 中間画像の圧縮はネットワークが 100BaseTX のクラスタでは効果があったものの, ネットワークが 1000BaseT のクラスタでは効果が見られなかった.

現在の実装ではクラスタ 1 とクラスタ 2 で多くの構成パラメータが異なるため, 両者の比較という形での議論は行なわず, 異なる 2 つのシステムでの実験結果の提示にとどめている. 構成パラメータを変更した場合の比較に関しては今後, 別論文としてまとめたいと考えている.

9. 謝　　辞

日頃より御討論いただく京都大学大学院情報学研究科富田研究室の諸氏に感謝します.

本研究の一部は文部省科学研究費補助金 (基盤研究 (B)(2) 課題番号 13480083 ならびに 特定領域研究 (C)(2) 「情報学」課題番号 13224050) による.

本稿で用いた胸部のボリュームデータは, (株) ケイジーティー 宮地英生 氏より御提供いただいたものである. また, エンジン, 盆栽のボリュームデータは volvis ☆から, クリスマスツリーのボリュームデータは The Computer Graphics Group XMas-Tree Project ☆☆から利用させて頂いた.

参 考 文 献

1) C.Rezk-Salama, K.Engel, M.Bauer, G.Greiner, T.Ertl: "Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Texturesand Multi-Stage Rasterization", In Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware, 2000.
2) S.Muraki, M.Ogata, K.Ma, K.Koshizuka, K.Kajihara, X.Liu, Y.Nagano and K.Shimokawa, "Next-Generation Visual Supercomputing using PC Clusters with VolumeGraphics Hardware Devices", Proceedings of IEEE/ACM Supercomputer Conference, 2001.
3) Muraki, S., et al.: A PC Cluster System for Simultaneous Interactive Volumetric Modeling and Visualization, *Proc. of IEEE Symp. on Parallel and Large-Data Visualization and Graphics*, pp. 95–102 (2003).
4) 山崎, 加瀬, 池内: "PC グラフィクスハードウェアを利用した高精度 ·高速ボリュームレンダリング手法", 情報処理学会 CVIM-130-10, Nov. 2001.
5) Levoy, M.: Efficient Ray tracing of volume data, ACM Transactions on Graphics, Vol.9, No.3, pp.245–261(1990).
6) Henry Fuchs, Gregory D. Abram and Eric D. Grant: Near real-time shaded display of rigid objects, Proc. of ACM SIGGRAPH, pp.65–72,(1983).
7) Ma, K.-L., et al.: Parallel volume rendering using binary-swap compositing, IEEE Computer Graphics and Applications, Vol.14, No.4, pp.59–68 (1994).
8) 高山征大, 他 :セル投影型並列ボリュームレンダリングへの Early Ray Termination の適用, Visual Computing シンポジウム 2004(採録決定).
9) 丸山悠樹, 他: 汎用グラフィックスカードを用いた並列ボリュームレンダリングシステムの実装, 情報処理学会研究報告 (2003-ARC-154), Vol.2003, No.84, pp.61–66, Aug. 2003.
10) Marcelo Magallon, Matthias Hopf and Thomas Ertl: Parallel Volume Rendering Using PC Graphics Hardware, Proc. Pacific Graphics , 2001.
11) Antonio Garcia and Han-Wei Shen: An Interleaved Parallel Volume Renderer with PC-Clustes, Proc. Eurographics Workshop on Parallel Graphics and Visualization, pp.51–59, 2002.
12) B. Cabral, N. Cam, and J. Foran: Accelerated Volume Rendering and Tomographic Reconstraction Using Texture Mapping Hardware, Proc. Symp. on Volume Visualization, pp.91-98, 1994.
13) 松田浩一, 大田敬太, 藤原俊朗, 土井章男: 普及型 PC における 2 次元テクスチャを用いた 3 次元画像の任意断面表示手法, 電子情報通信学会 論文誌 D-II, Vol.J85-D-II, No.8, pp1351–1354(2002).

☆ http://www.volvis.org/
☆☆ http://ringlotte.cg.tuwien.ac.at/datasets/XMasTree/