特別研究報告書

並列ボリュームレンダリングにおける画像重畳処理に関する研究

指導教員　　富田 眞治 教授

京都大学工学部情報学科

吉良 裕司

平成17年2月10日

並列ボリュームレンダリングにおける画像重畳処理に関する研究

吉良 裕司

内容梗概

近年の計算機性能の急速な向上に伴い，大規模かつ高精度な数値シミュレーションを行い，その結果を可視化し提示することが求められている．そしてその際には，ボリュームレンダリングによって可視化が行われるようになってきている．ボリュームレンダリングは対象を3次元的に中身の詰まったボリュームとして表現し，その内部構造や動的特性を表示するための技術である．ボリュームレンダリングによって可視化を様々な視点から対話的に行うことによって，ボリューム空間中の複雑な3次元構造を容易に理解することが出来る．そのため，対象へのインタラクティブな操作に対応してシミュレーションを行うとともに，その結果を実時間で可視化することが科学や医療などの様々な分野で求められている．

しかし，ボリュームレンダリングは表示させる画像のピクセル数に応じた積分計算が必要となる．さらに，ボリュームデータは空間の最小単位であるボクセルの集合で表されているため，そのデータ量は膨大になる．そのために1秒間に30フレームの描画を行う実時間可視化は一般に困難であり，並列計算機や専用のハードウェアを用いる必要があった．

一方，近年のPCグラフィクスカードは描画速度と機能が飛躍的に向上している．そのため，非常に高速に3次元の描画が可能となった．テクスチャマッピングやアルファブレンディングの機能がサポートされたことや，グラフィクスカードの記憶容量の増大などの影響が大きい．しかし，医療データ等のような大規模なデータを相応のサイズで実時間に可視化するためには未だに十分な能力を備えているとはいえない．

本稿では，特別なハードウェアを用いずに，大規模なデータを実時間に可視化することを目指し，グラフィクスカード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構築について報告する．グラフィクスカードによるボリュームレンダリングでは主記憶からグラフィクスカードへのデータ転送の時間が問題となる．そこで，グラフィクスカードを備えたＰＣクラスタの各ノードが，ボリュームレンダリングを分割してレンダリングすることでその問題を回避する．このとき，各スレーブノードで生成されるレンダリング結果を1つのマスター

ノードへ集めるための通信時間が問題となる．そこで，最も単純な方法として逐次的にレンダリング結果を集めていく方法を挙げ，その問題点として画像データを送るスレーブノードの中にはアイドリング時間が非常に長くなる点に注目した．その解決策としてノード間通信を木構造で行うと同時に重複度を考慮したノード間の合成も行うことで，より効率的な通信を目指した．

その結果，現在の実装でマスターの受信するデータの総量は，ランレングス符号化により中間画像の圧縮を行った逐次的な合成の場合に比べて削減できたが，ノード間の重畳処理全体にかかる時間は増大してしまった．その原因として，より多くの合成をスレーブノードで行おうとしたために合成で時間がかかってしまい，マスターノードがアイドルとなってしまう時間が発生したことが挙げられる．また，重複度に大きく依存するシステムとなっていて，各重複度の割合により木構造の各ステージの負荷が偏ってしまったために視点による速度の差が大きくなった．

Research about Image Composition in Parallel Volume Rendering

Yuji KIRA

Abstract

Today's rapid improvement of the computer performance brings a high expectation of a large scale numerical simulation and visualization of the result. And volume rendering is coming to be used for visualization. In volume rendering, the object regarded as a three dimentional volume which is filled. And volume rendering is a technology to display the internal structure and dynamic characteristic of an object. We can understand complicated three dimentional structure in volume space easily by visualizing by volume rendering technique interactively from various viewpoints. Simulation and real time visualization corresponding to intractive operation are required in the various fields such as science and medical treatment.

However, volume rendering needs integral computation in proportion to pixels of the image. Volume data consists of a set of voxels that are the minimum elements of space. And the size of volume data is generally enormous. Therefore it is difficult to render 30 images per second with volume rendering. So, it used to realize real time volume rendering only with a parallel computer or a special hardwear.

On the other hand,today's great progress of PC graphics card makes it possibe to acchieve high speed rendering in 3 dimention. It is affected mostly by the support of texture mapping, alpha-blending function and an increase of the graphics memory. But it is not enough for visualization of a large scale data such as medical data in suitable size and real time.

In this paper, we describe the parallel volume rendering system using graphics cards to realize real time volume rendering of large scale data set without a special hardware for volume rendering. It takes much time to load volume data in graphics card from main memory. Then, we can avoid that matter by separating the volumedata into subvolume, distribute them to each node which have a graphics card and render them.In this situation, time of the gathering the results of rendering a subvolume becomes our concern. Then we propose

the simplest way to gather the results of rendering, which is gathering them one after another. And we focused on the point that some nodes' idling time accepted by a master node for sending image data become very long. So we aim for more efficient way to communicate image data by using tree structure in communication. And the same time, we introduce composition by refering numbers of overlaps.

As the result, the speed of gathering image data from subnodes in tree structure are slower than the way to gather them in one after another using Run-Length encoding. Although, we could reduce the amount of data which master node receives. We think that the reason of that result lies in the situation which some slave nodes take much time to compose and master node can not receive image data while sub nodes are composing. The system depends much on the numbers of overlaps. And the system is fast in some viewpoints and very slow in other viewpoints. Because rates of number of overlaps decide how heavy at each stage in tree structure.

並列ボリュームレンダリングにおける画像重畳処理に関する研究

目次

第 1 章　はじめに　1

第 2 章　並列ボリュームレンダリング　1

2.1　ボリュームレンダリング 2

2.2　グラフィクスカードによるボリュームレンダリング 3

2.2.1　アルファブレンディング 3

2.2.2　テクスチャマッピング 4

2.2.3　テクスチャベースボリュームレンダリング 5

2.3　並列化 6

2.3.1　基本アルゴリズム 6

第 3 章　ノード間合成の手法　7

3.1　逐次的合成 7

3.1.1　ランレングス符号化 8

3.2　Binary Swap Composition 8

3.3　２分木による合成 10

3.3.1　Bounding Box 12

3.3.2　重複度 12

3.3.3　ブロック分割 12

3.4　提案方式 13

第 4 章　評価　16

4.1　環境 17

4.2　逐次的合成の結果 19

4.3　２分木による合成の結果 19

4.4　考察 21

第 5 章　まとめ　23

謝辞　24

参考文献　24

第1章　はじめに

近年，汎用PCの性能は急速に向上してきている．それに伴い，以前はスーパーコンピュータを用いなければ困難であった，大規模かつ高精度な数値シミュレーションを安価な汎用PCで行うことが可能となってきた．そのような数値シミュレーションの1つとして，大規模な3次元データを実時間内でシミュレートし，更に可視化することが医療等の分野において必要となっている．可視化することにより，人間にとって理解することが容易になり，様々な仮想実験/仮想体験型のシミュレーションが可能となる．

代表的な可視化手法にボリュームレンダリングがある．ボリュームレンダリングとは，対象を3次元的に中身の詰まったボリュームとして表現し，その複雑な内部構造や動的特性を表示するための技術である．ボリュームレンダリングは，表示させる画像のピクセル数に応じた積分計算が必要となるために，数値計算が膨大なものとなる．更に，ボリュームデータは3次元空間全体を格子状に分割し，各格子ごとにその地点のデータを持たせているためにデータ量も膨大になる．このような大規模なデータを扱う場合には，膨大な演算量，グラフィクスカードの記憶容量やメモリ大域幅の制限から，機能の向上が目覚しいといえども，単一のグラフィクスカードで実時間に可視化することは不可能である．

そこで，PCクラスタを用いて大規模なデータのボリュームレンダリングを並列化することにより，実時間可視化を可能とすることを目指す．クラスタを用いて並列化を行うと必ず通信が必要となる．PC間の通信をどのように行うかは，PCクラスタを扱う際に非常に重要な問題となる．本稿では，PCクラスタを用いた並列ボリュームレンダリングにおける通信時間の削減を目標とし，各PCの所持するデータをPC間でどのように合成するかについて考察した．以下，2章で並列ボリュームレンダリングについて説明し，3章で通信時間を削減するための手法を述べ，4章でそれらの評価を行う．

第2章　並列ボリュームレンダリング

まず，本稿で使用する重要な語句を説明する．その後，ボリュームレンダリングとグラフィクスカードを用いたボリュームレンダリングについて記し，ボリュームレンダリングの並列化について述べる．

2.1 ボリュームレンダリング

ボリュームデータは空間の最小単位であるボクセル (voxel) の集合で表されている [1]．ボクセルは 3 次元座標上の点からサンプリングされたもので，もとの対象物について，その地点での 1 つもしくは複数の観測値や計算値（例えば，傾向強度，密度，電荷，湿度など）を持たせることにより，空間全体をデータとして保持している．格子点以外の地点のデータは，その地点近傍の格子点から線形補間を用いて求める．これにより，空間内の全ての地点におけるデータを得ることが出来る．スクリーンのピクセルごとにボリュームデータ内へ光線を飛ばし，各ボクセルに割り当てられた不透明度，シェーディング値，そして入射光の強度を元とした，視線方向へのボクセル値の積分をボリューム空間を抜けるまで行う．それにより，各ピクセルの値を求め，レンダリング画像の作成を行う．ボリュームレンダリングの積分計算の順番により，次の 2 つの方法がある．それを図 1 に示す．

- Front to Back
  スクリーンの各ピクセルから光線を飛ばして，スクリーンに近いほうからその光線上の各標本点の累積値を計算する．この場合，カラーと不透明度の累積を求める計算式が必要となるが，不透明度の累積を用いて Early Ray Termination (ERT) による高速化が可能となる．ERT とは，不透明度の累積値が十分に不透明であるという閾値を上回った場合，それ以降の物体からの影響はごく小さいと考え，そこで計算を打ち切る手法である．

$$C = C_d + A_s(1 - A_d)C_s \tag{1}$$

$$A = A_d + (1 - A_d)A_s \tag{2}$$

- Back to Front
  スクリーンの各ピクセルから光線を飛ばして，その光線上の標本点の値をスクリーンに遠いほうから足し合わせる．
  遠い標本点から順に足していくので，空間内のすべての標本点の累積計算をしなければならないが，透明度に関する累積計算をする必要がなくなる．

$$C = A_sC_s + (1 - A_s)C_d \tag{3}$$

式 (1)(2)(3) において，C はカラー値，$C = (R, G, B)$，A は不透明度 (アルファ) としている．添え字 s (source) は現在の標本点の値を示し，添え字 d (destination)

は光線が現在の標本点にあたるまでの累積を示す．このように，前後の標本点の値をアルファ値の示す比率で線形内挿して合成画像の色を算出することをアルファブレンディングという．

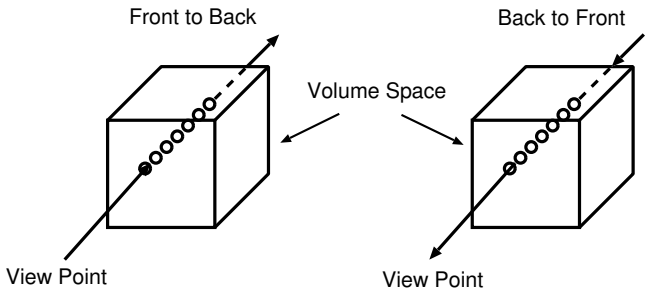

図 1: ボリュームレンダリング

2.2 グラフィクスカードによるボリュームレンダリング

2.2.1 アルファブレンディング

アルファブレンディング [2] とは全面の画像要素の色と後面の画像要素の色とを，アルファ値の示す比率で線形内挿して，合成画像の色を算出するものである．アルファ値とは各ピクセルの色を決定する RGB(赤，緑，青) 以外のもう一つの要素で，素材の不透明度を決定するものであり，通常 0.0 から 1.0 の間の値をとる．1.0 は素材が不透明で後ろにあるものは隠されるということを示し，0.0 は素材が完全に透明で見えないことを表す．アルファブレンディングでは前面のカラー値－ソース (source) －が，現在保存されている後面のカラー値－デスティネーション (destination) －と組み合わせて処理される．

ソースとデスティネーションの混合係数をそれぞれ $(A_s, 1 - A_s)$ とし，ソースとデスティネーションの RGBA 値を $(R_s, G_s, B_s, A_s)(R_d, G_d, B_d, A_d)$ で表示すると，最終的な RGBA 値は次のように求められる．

$$\begin{aligned}(R, G, B, A) = (&R_s + R_d(1 - A_s), G_s A_s + G_d(1 - A_s), \\ &B_s A_s + B_d(1 - A_s), A_s A_d + A_d(1 - A_s))\end{aligned} \tag{4}$$

2.2.2 テクスチャマッピング

テクスチャは物体の表面の質感を表現するために貼り付けるデータであり，テクスチャを物体をあらわす図形オブジェクトに貼り付けることをテクスチャマッピング [2] という．テクスチャマッピングは，非常に複雑な画像を大掛かりな図形モデルを構築することなく生成できる．ポリゴンの各頂点がテクスチャ画像上のどの点に対応しているのかという情報を予め関数として持っており，ポリゴンを描画する時に，それをもとにポリゴンに対応するピクセルの 1 点 1 点が，テクスチャ上のどの位置（色）を参照しているのかを求めてその色でピクセルを着色することでテクスチャの描画を実現する．

テクスチャ座標はポリゴンの各頂点がテクスチャ画像上のどの位置に対応するのかを表すものである．テクスチャ座標を図 2 に示す．通常は 0.0〜1.0 の値を取り，テクスチャ画像の左下が，$(s = 0, t = 0, r = 0)$ で，右上が $(1, 1, 1)$ となる．

$$\boldsymbol{s} = \begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

テクスチャ座標 (同次座標系)　オブジェクト座標 (同次座標系)

$\boldsymbol{M}(4\times4)$:変換行列 (平行移動, 回転等)

$\boldsymbol{T}(4\times4)$:オブジェクト座標とテクスチャ座標の対応を表す関数

とすると

$$\boldsymbol{s} = \boldsymbol{MTx} \tag{5}$$

によりオブジェクト座標とテクスチャ座標の対応を定め，それにより 3 次元オブジェクト空間内の，ある座標でのカラーをテクスチャにより決定する．

4 つのテクスチャ座標 (s, t, r, q) がテクスチャ行列で乗算される場合，その結果のベクトルは同次テクスチャ座標として解釈される．この q 座標は複数の射影や，透視変換が必要な場合に使用する．また，1 次元テクスチャの場合は s のみを，2 次元テクスチャでは s, t のみを，3 次元テクスチャでは s, t, r を用いる．

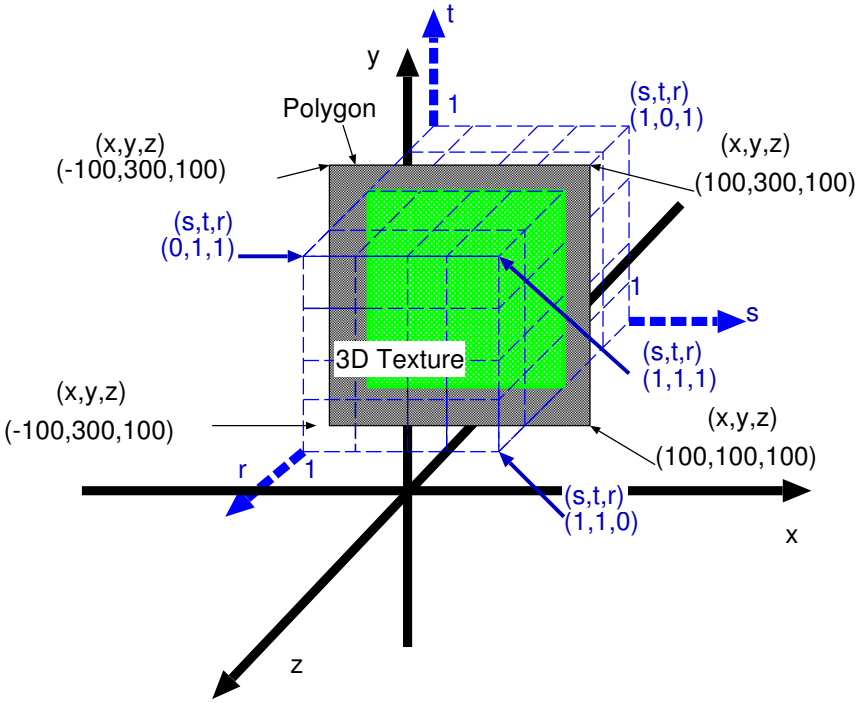

図 2: オブジェクト座標とテクスチャ座標

2.2.3 テクスチャベースボリュームレンダリング

グラフィクスカードでサポートされたテクスチャマッピングとアルファブレンディグを用いて，ボリュームレンダリングを行う．ボリュームデータのある軸に垂直な断面をテクスチャデータとしてポリゴンにマッピングし，それらを不透明度を用いて合成する．それにより，式 (3) の計算を全てのピクセルに対して実行することが出来る (図 3).

視線に垂直な平面（ポリゴン）を複数用意し，それらにボリュームデータを厚みを持ったソリッドテクスチャ（3 次元テクスチャ）として与える．3 次元のテクスチャ座標とテクスチャ変換行列によってテクスチャが割り当てられている空間内の任意の平面に対するテクスチャ（物体の任意の断面）を構成することが出来る．この座標を用いて視線に対して垂直に並んだ各ポリゴン上にマッピングされるテクスチャを求める．そして，視点から遠いポリゴンからアルファブレンディングにおいて，各要素の混合係数を $(A_s, 1 - A_s)$ とし，式 (3) の計算を行う．

視点変更のたびに各ポリゴンへのテクスチャ座標が再計算される．即ち，視点変更において実際に変更されるのはポリゴン上の図柄であり，ポリゴンの位

置は常に一定である．

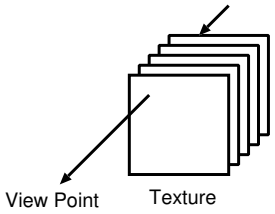

図 3: テクスチャベース

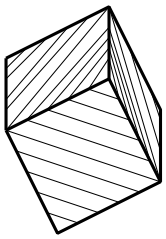

図 4: ３次元テクスチャ

2.3 並列化

2.3.1 基本アルゴリズム

ボリュームレンダリングは光線に沿ったボリューム空間内の標本点の透明度の累積をスクリーン上のピクセル毎に行うものである．したがって，その処理はピクセル毎に独立であり，その計算は空間内を通過する光線の積分計算である．よって，空間をいくつかに分割し，それぞれの区間に対する演算結果を足し合わせることで最終結果を得ることができる．そこで，ボリュームデータを x, y, z 全ての軸方向に 2 等分し，図 5 のように 8 つのデータ (サブボリューム) に分割する．その分割したデータそれぞれをクラスタの各ノードが分担してテクスチャベースでボリュームレンダリングし，各サブボリュームごとのレンダリング結果（中間画像）を生成する．

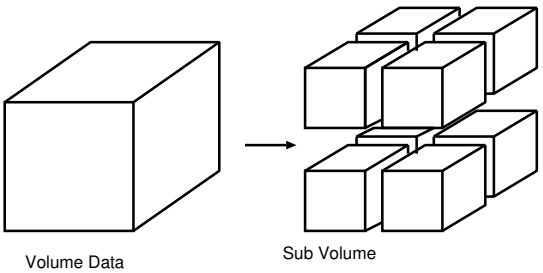

図 5: データ分割

そして，フレームバッファ上でレンダリング結果が存在する部分だけを読み出し，それを一つのマスターに集める．データは最初に分配されるだけで回転等のパラメータの変更による再描画において，各ノードは現在保持しているサ

ブボリュームに関してのみレンダリングを行う．各サブボリュームからそれに対応するオブジェクト座標での代表点をとり，サブボリュームのスクリーンからの距離を得る．その距離を用いて，各ノードで生成されたレンダリング結果の画像を，視点からの距離を考慮してスクリーン上の適切な位置に式 (3) によりアルファ値で合成することにより最終結果を得る (図 6).

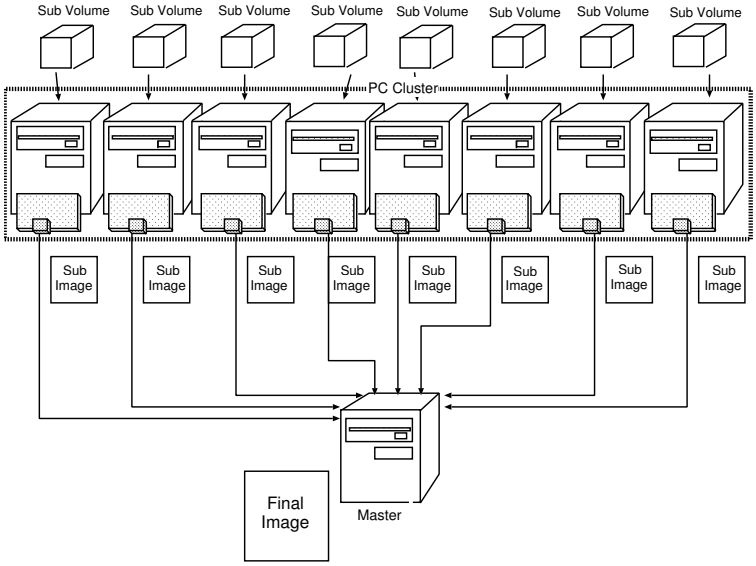

図 6: 並列ボリュームレンダリングシステム

第 3 章　ノード間合成の手法

本章では，各ノードが保持する画像データを並列処理によって合成する手法を示し，その高速化手法について述べる．

3.1　逐次的合成

最も単純な手法である．各ノードが，自身の担当するサブボリュームのレンダリングを終了し，フレームバッファ上のレンダリング結果を読み出した後，その画像データを一斉にマスターノードへその結果を送信する手法である．マスターノードは，視点からの距離が最も遠いサブボリュームを担当しているノードから順に，逐次的に受信する．

ノード間合成にかかる時間は視点の位置に依らず，安定した速度が得られる．しかし，全てのスレーブノードがほぼ同時にマスターノードへ送信を始めよう

とするが，マスターノードは複数のスレーブノードから同時に受信することはできない．そのため，マスターノードに送信できるようになるまで待っていなければならないスレーブノードが存在する．待っている間は合成処理に関する作業は行っていない．マスターノードへの送信が可能になるときが後になるほど，仕事をしない時間が長くなる．そこで，この待機している時間を利用して合成に関わる処理を行うことでより高速に最終合成を行うことを考える．

逐次的合成処理において，中間画像を圧縮し，送信するデータ量を軽減する手法としてランレングス符号化が挙げられる [3].

3.1.1 ランレングス符号化

ランレングス符号化は繰り返しの情報を圧縮することによって，データのサイズを減少する方式である．同じ色の連続している部分を破棄して，データを圧縮する．これを各ノードの保持する画像中の透明部分に関してのみ行う．透明 ($A = 0$) であるピクセルは RGB の値に関わらず可視化結果に影響はないので，RGB の値をマスターに送信する必要はない．そこで，送信する画像の中に含まれる連続する $A = 0$ となるピクセルの個数を RGB の領域を用いて表し，データ量を減らす (図 7)．これにより，通信データ量を減らすことができるので，画像データの転送時間を減らすことができる．

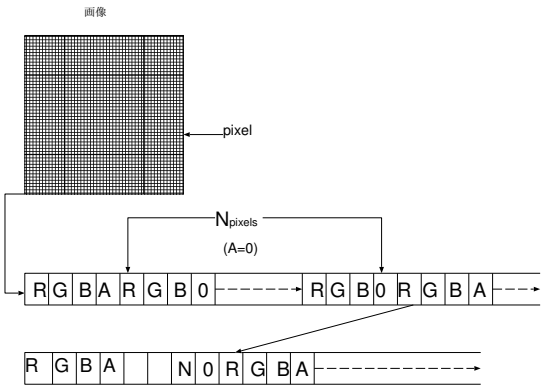

図 7: ランレングス符号化

3.2 Binary Swap Composition

ノード間の合成手法の一つに，Binary Swap Composition (以下，BSC)[4] という手法がある．この手法は，通信量の削減と合成処理における負荷の均衡を

とる点において優れている [5] 手法である．

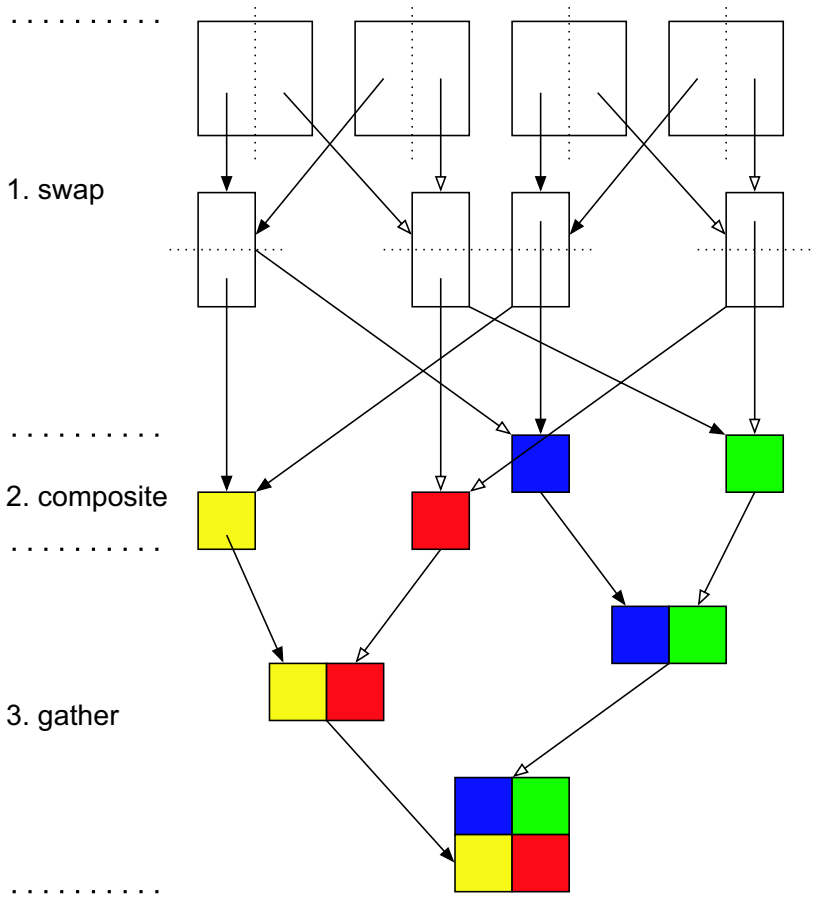

図 8: Binary Swap Composition Tree

N 台 のプロセッサで最終合成を行う場合を考える．BSC は図 8 に示すように，交換 (Swap)，合成 (Composite)，収集 (Gather) という 3 つのフェイズから構成される．

- Swap
  各プロセッサは，合成すべきスクリーンを 2 分割する．自身と同じ領域を持つプロセッサと，互いが異なる領域を担当するように，分割したスクリーンを交換する．
  この交換を $log_2 N$ 段繰り返す．
- Composite
  $log_2 N$ 段の交換後には，各プロセッサは自身が担当するスクリーン領域（全体の N 分の 1）の，全てのプロセッサによるサブスクリーンを持つことに

なる．即ち，最終合成を行うことが可能な条件を満たしている．したがって，各プロセッサが自身の担当領域を最終合成することにより，並列最終合成を行うことが出来る．

- Gather

  並列最終合成が終わったあとでは，単純に領域を繋ぎ合わせるだけで最終画像を得ることが出来る．この繋ぎ合わせは，N 枚の最終合成画像を通常の木による合成を行うことで，1 台のプロセッサに集めることで完了する．

この手法は，交換のフェイズにおいて，1 段交換を行うたびに，プロセッサ間でやり取りされるサブスクリーンの大きさは半分になる．

また，最終合成についても，各ノードがそれぞれ別の領域を並行して合成を行うので，負荷の均衡がとれている．

しかし，この手法は次に述べるような点で我々の行っている研究には不向きである．

- スクリーンサイズ

  この手法の性質から，各ノードは最終画像と同じ大きさのスクリーンサイズを持つことになる．これにより，フレームバッファから画像情報を読み出す処理時間が増大してしまう．

- 描画領域の局所性

  我々が扱う並列ボリュームレンダリングは，ボリュームデータを 8 等分したうちの 1 つを各ノードが担当する．このため，各ノードの描画領域はまとまっていて，スクリーンのどこにあたるかを予め求めることができる．また，その領域は全体の 4 分の 1 ほどである．つまり，何も描画されていない領域が全体の 4 分の 3 を占める．そのような条件下で BSC を行うと，全く画像が描画されていない領域が Swap に多く含まれることになる．

このことから，BSC が速いといえるのは，各ノードの描画領域が最終画像と同じ大きさのスクリーン全体に及ぶ時であるといえる．

3.3 2 分木による合成

2 分木 (Binary Tree) による合成では，8 個の CPU をそれぞれスレーブノードとする．逐次的合成の場合は，マスターノードが単純に視点から遠い順に受信して Back to Front の計算式で合成すればよい．しかし，2 分木による合成の場合に最終的な画像を正しく表示させるためには，スレーブノード間合成では常

に隣接するサブボリュームと合成しなければならない．このことは，逐次的合成でもいえることだが，視点から遠い順に受信することでそのルールは守られている．2 分木では，視点からの距離だけから合成相手を決めることはできない．そこで，ノード間合成をする際には合成する相手を正しく選ばなければならない．

本実験では，図 5 のようにボリュームデータを 8 個のサブボリュームに分割しているので，任意のサブボリュームに隣接するサブボリュームは 3 つである．そして，その 3 つのサブボリュームはそれぞれボリュームデータ固有の座標系の x, y, z 方向と対応させることができる (図 9)．また，視線ベクトルが (x, y, z)

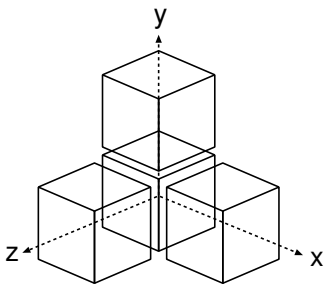

図 9: 隣接するサブボリュームと軸の対応

と表されるとする．あるサブボリューム V_1 について，z 方向に隣接するサブボリューム V_2 の投影される領域と，V_1 の投影される領域との重なる部分領域の大きさは，視線ベクトルの z 成分の絶対値の大きさに依存する．例えば，ボリュームデータを正面から見ると視線ベクトルは (0,0,1) とおける．このとき，どのサブボリュームの投影領域も，x, y 方向に隣接する 2 つのサブボリュームの投影領域とは全く重ならない．しかし，z 方向に隣接するサブボリュームの投影領域を考えると，互いに完全に重なっている．このように，視線ベクトルの各成分の絶対値が大きいほど，その方向に隣接する 2 つのサブボリュームの投影領域の重なりは大きくなる．

そこで，視線ベクトルの 3 つの成分の内，絶対値が大きい成分（主軸）方向から合成するように 2 分木を構成した．主軸方向を基に合成を行うことにより，隣接するサブボリュームと合成しているので，正しい最終画像を得ることができる．また，合成するために選ばれた 2 つのサブボリュームの中間画像の重なる部分が最も大きくなる．すると，合成結果のデータサイズがより小さくなる

ので，通信データ量の削減が可能となる．こうして逐次的合成で問題となった長いアイドリング時間の削減が行えるが，ノード間で通信されているデータに目を向けると，ノード間合成の際に必要のない画像データも送受信されていることが分かる．ノード間合成の際に必要のない画像データとは，そのデータを他のノードに送信しても，合成に用いられないデータのことを指す．このような画像データの送受信を減らすことで高速なノード間合成の達成を目指す．

今必要となる情報は，スクリーン上のある領域に画像データを持つノードの数である．以下において，このノードの数を重複度 [6] と呼ぶこととする．これを求めるために，まずはバウンディングボックスについて述べる．

3.3.1 Bounding Box

重複度を求めるためには，8 つのノードによるレンダリングの結果が描画されるスクリーン上の位置をそれぞれ求めなければならない．各ノードが担当するサブボリュームは 8 つの頂点を持ち，それらを回転させた結果を描画する．つまり，8 つの頂点の回転後の位置を求めれば，頂点の x 座標の最大値と最小値，y 座標の最大値と最小値から，サブボリュームが描画されている領域を含む最小の四角形を得ることが出来る．この四角形をバウンディングボックスと呼ぶ (図 10)．

3.3.2 重複度

各サブボリュームに対して，バウンディングボックスを 1 つずつ求めることが可能であるので，ボリュームデータを図 5 のように分割した場合は計 8 つのバウンディングボックスが得られる．この場合，スクリーン上のある領域について，8 つのバウンディングボックスのうちいくつのバウンディングボックスがその領域を含んでいるかを求めることが出来る．このことを重複度 (Number of Overlap) とよぶ．重複度は，その領域についてのノード間の合成回数と言い換えることも出来る (図 11)．ある領域を含んでいるバウンディングボックスの数が 1 つのときに値が 1 となるがそれを重複度 1 とよぶのは重複という言葉の意味に反するが，8 つのバウンディングボックスのうち 1 つのバウンディングボックスも含まない領域については値を 0 としているので便宜上そうよぶこととする．

3.3.3 ブロック分割

画像データには局所性があるので，ある領域の重複度と，その近傍の領域の重複度が等しいことが多い．そこで，各ノードで生成される中間画像を，縦横

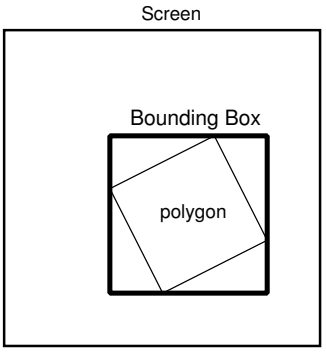

図 10: Bounding Box

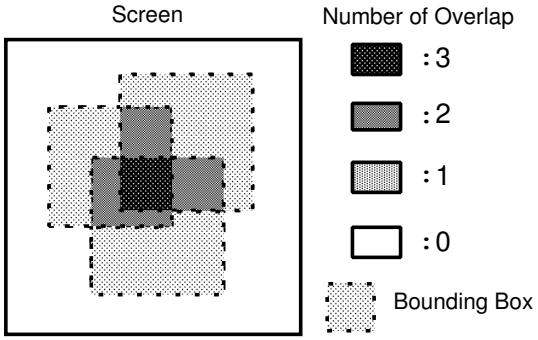

図 11: 重複度

それぞれ 2^n ピクセルの大きさのブロックに分割し，以降の処理をブロック単位で考えることとする．各スレーブノードが生成する中間画像を N^2 ピクセルとする．この中間画像を，縦横それぞれ m 個、計 m^2 個のブロックに分割する．各ブロックの大きさを 2^n(n は 0 から 9 の整数) ピクセルにする．つまり，$m = N \div 2^n$ となる (図 12)．図では，$m = 8$ である．

この時に，サブボリュームが全く描画されていないブロックのデータは持たないようにする．これにより，各ノードが持つ画像データのサイズを小さくすることが可能となる．このブロックごとに重複度を求める．それは，バウンディ

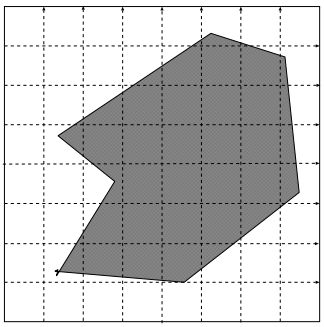

図 12: ブロック分割

ングボックスを求めるとその領域と重複するブロックが定まるので，そのブロックの重複度を 1 増やすことで行われる．

3.4 提案方式

重複度を利用した次のようなノード間合成の方式を提案する．

スクリーン上の重複度の分布を，Bounding Box から求めた場合は図 13 のよ

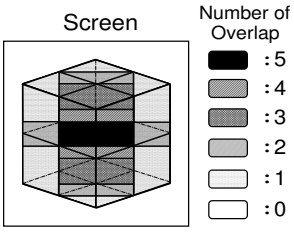

図 13: Bounding Box による重複度の分布

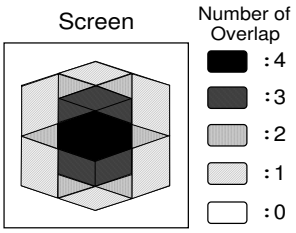

図 14: ピクセル単位による重複度の分布

うになる．各スレーブノードはこの分布を計算してブロックごとの重複度を予め求めておく．ただし，ピクセル単位で求めた重複度は図 13 とは異なり，図 14 のようになる．

ブロック分割を行い，画像データをブロック単位で持つようにするためには，ピクセル単位の Bounding Box をフレームバッファから読み出すだけでは不十分である．そこで，フレームバッファから読み出す領域は，ブロック単位の Bounding Box とする．ピクセル単位の Bounding Box を求めるときに，その領域の最大の x, y，最小の x, y を計算により求めた．この座標と，ブロックの 1 つあたりの大きさから，ブロック単位の Bounding Box の 4 つの頂点ブロックが求められる．

フレームバッファから画像データを読み出した後，画像データをブロック毎に分割して合成に必要なブロックのデータのみを持つ．合成に不要なブロックとは，そのブロックに含まれるピクセルが全て透明なブロックのことである．この時に，ブロックごとの重複度を参照して重複度 1 のブロックのデータを持つ配列 a_1 と，重複度 2 以上のブロックのデータを持つ配列 a_{2+} とに分けておく．このような配列をまとめて a[ ] で表す．ノード間合成を行うときには，a[ ] だけでは合成する相手のブロックを特定することができないので，各 a[ ] ごとに，次のようなデータも用意する．

- a[ ] に格納されているブロックの総数を覚えておく変数 p
  別のノードに画像データを送るときは，まずはこの変数を送信する．そしてこの変数の値が 0 でない時に互いの送受信のためのプロセスを実行する．
- a[ ] に格納されているブロックがスクリーン上でどの位置にあたるかを示す値を覚える配列 s[ ]
- ブロックのスクリーン上の位置が与えられたときに a[ ] の中に，そのブロックが含まれているかを示す配列 t[ ]
  a[ ] に対象のブロックが存在することを示すために，先頭から何番目のブ

ロックかを表す値が格納されている．対象のブロックが存在しないことを示すためには，スクリーンの総ブロック数（定数）を格納しておく．つまり，ブロック間の合成を行うときには t[s[ ]] の値が総ブロック数なら a[ ] の中にそのブロックは保存されていない．t[s[ ]] の値が総ブロック数よりも小さい値なら，a[ ] の先頭から t[s[ ]] 個めのブロックが探しているブロックである．

ここで，スレーブノード数が 8 台である場合の重畳処理の流れを図 15 に示す．

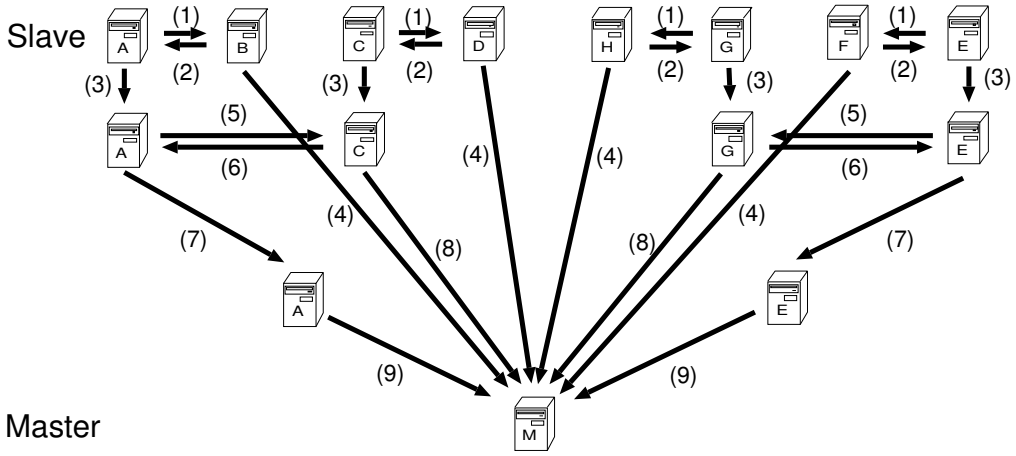

図 15: 重畳処理の流れ

8 台のスレーブノードがそれぞれA～Hのどの場所に配置されるかは，3.3 の最初に述べた視線ベクトルの各成分から決まる．以下，図 15 の矢印の番号順に説明する．

(1) スレーブノード A，C，E，G が持つ a_1 をそれぞれスレーブノード B，D，F，G へ送る．

(2) スレーブノード B，D，F，G が持つ a_{2+} をそれぞれスレーブノード A，C，E，G へ送る．

(3) スレーブノード A，C，E，G は，自身の a_{2+} と他ノードの a_{2+} を持っている．この 2 つの配列をブロックごとに合成して，重複度が 2 でかつ合成を行ったブロック a_2 と，重複度が 2 で合成を行っていないブロックと重複度が 3 以上のブロックの配列 a_{3+} というように 2 つの配列に分ける．このとき，全てのブロックが合成できるのではなく，どちらか一方しか画像デー

タを所持していないために合成を行えないブロックも存在する．そのようなブロックは，配列にコピーするだけである．

(4) スレーブノード B，D，F，H は，自身の a_1 と他ノードの a_1 を持っている．この集合に含まれるブロックは全て重複度が 1 なので，全く合成する必要が無い．つまり，この 2 つの配列のブロックを合成しようとしても，共通のブロックを持たないので合成はできない．そこで，ここでは 2 つの配列を別々にマスターノード M に送る．

(5) スレーブノード A，E が持つ a_2 をそれぞれスレーブノード C，G へ送る．

(6) スレーブノード C，G が持つ a_{3+} をそれぞれスレーブノード A，E へ送る．

(7) スレーブノード A，E は，自身の a_{3+} と他ノードの a_{3+} を持つ．この 2 つの配列をブロックごとに合成して，1 つの配列 a_3 にまとめる．このとき，(3) と同様に全てのブロックが合成できるのではなく，どちらか一方しか画像データを所持していないブロックも存在する．そのようなブロックは合成を行えないので，配列に加えるだけである．

(8) スレーブノード C，G は，自身の a_2 と他ノードの a_2 を持っている．この配列に含まれるブロックは全てそれ以上の合成は行わないので，(4) と同様にこの 2 つの配列を別々にマスターノード M に送る．

(9) (7) の処理によって得られた配列をマスターノード M に送る．

(1)〜(9) により全ての集合を受け取ったマスターノードは，その画像データをスクリーンに描画する．

マスターノードが最終画像を表示するためには，少なくともウィンドウに描画される透明でない部分の画像データは受け取らなければならない．その画像データを，マスターノードができるだけ常に受け取っている状態にしておこうとするのが，この方式の考え方である．そこで，重複度を考え，ノード間の合成に必要のなくなった領域から受信するようにしている．

図 15 ではマスターノードと 8 つのスレーブノードで説明したが，スレーブノードの台数が多くなったときにもマスターノードの上のレベルまでは全て同じ操作を繰り返せばよい．

第 4 章　評価

4.1 環境

OpenGL グラフィクスライブラリ，ＭＰＩ-2 を用いて PC クラスタ上で汎用グラフィクスカードによる並列ボリュームレンダリングを次のような環境で行った．

CPU	Pentium4 3.0GHz
Memory	1.0GB/DDR400
Mothyerboard	ASUS P4C800-E Delux
Graphics Card	GeForceFX5950 Ultra
Graphics Memory	256MB DDR
OS	Debian GNU/Linux(kernel 2.4.22)
Network	1000BaseT Ethernet
実測性能	およそ 500Mbps

表 1: 実装環境

PC クラスタは 8CPU で，別にマスターが 1 台ある．このグラフィクスカードのビデオメモリは 256MB であるが，テクスチャデータの格納以外にも，Z バッファ，フレームバッファ等にも使用されるので実際にテクスチャデータ格納に割り当てることが出来るメモリの大きさは 256MB より小さくなる．

並列化はこの PC クラスタの各ノードに，ボリュームデータを縦方向に 2 分割，横方向に 2 分割，奥行き方向に 2 分割，すなわち全体を 8 分割したサブボリュームを１つずつ割り当て，個別にボリュームレンダリングを行う．その結果を第 3 章で述べた手法によりノード間合成し，最終的にマスターで表示する．

最終的な合成結果を表示するスクリーンのサイズは 1024×1024 である．各ノードは，8 分割されたサブボリュームを担当しているので，レンダリング結果を表示するスクリーンのサイズは，最小で 512×512 とすることができる．よって，逐次的合成を行うときにはスレーブノードのスクリーンサイズは 512×512 とした．しかし，投影領域が 512×512 であれば，それが投影されるウィンドウサイズが 1024×1024 であっても，描画処理にかかる時間の差は非常に小さかった．ただし，フレームバッファから画像データを読み出して配列に保存する処

理の時間には，その領域の広さが大きく関わっているのでバウンディングボックスを用いて最小の領域を読み出すようにする．この事と，ウィンドウサイズを 1024×1024 で行うとブロック分割の実装が容易になるという事から，木構造による合成の際にはスレーブノードのスクリーンサイズは 1024×1024 とした．スクリーンサイズは，視点変更の際にボリュームデータの投影がスクリーンからはみ出ることがないように，十分な大きさに設定した．正面から見たとき，サブボリュームをマッピングするポリゴンがスクリーンへ投影される領域はおよそ 300×300 ピクセルとなる．そのため，この視点の時に各ノードが最初に所持する中間画像データは $300 \times 300 \times 4(RGBA$ 各要素 $8bit) = 360KB$ となる．

また，通信速度の実測は 500MB ほどであった．

この実験では次のようなボリュームデータを使用した．

- 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．
- 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．

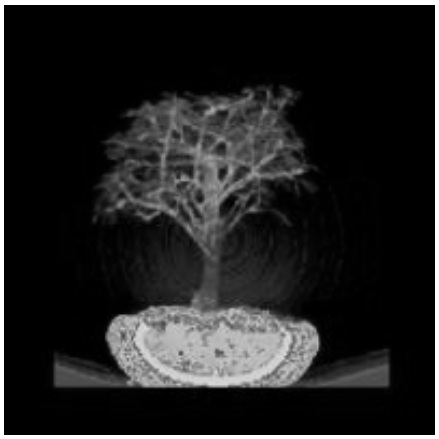

図 16: Bonsai

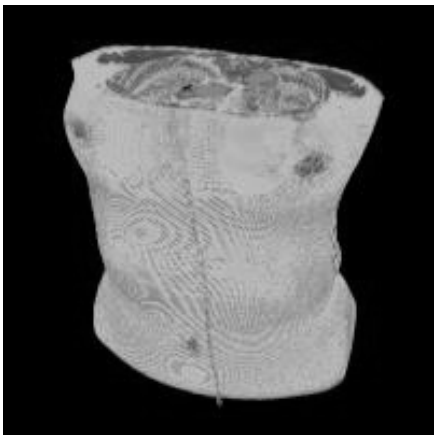

図 17: Chest

ボリュームデータは全て RGBA4Byte なので，テクスチャデータとしてグラフィクスメモリに格納する際には，上記のサイズの 4 倍の大きさとなる．ただし，ボリュームデータを 8 つのサブボリュームに分割するので，各ノードがグラフィクスメモリに保存するサブボリュームのサイズは，ボリュームデータのサイズ $\times 4(RGBA) \div 8$ となるので，Bonsai の場合はおよそ 8.4MB，Chest の場合はおよそ 67.1MB となる．これは，グラフィクスメモリに十分収まるサイズ

であるのでスワップによる描画速度の低下は見られなかった.

レンダリングの時間は，ノード間合成の手法，すなわち逐次的な合成の場合と 2 分木による合成の場合とではほとんど変化はない．というのは，ノード間合成はフレームバッファから読み出した画像データを合成することであって，レンダリング時間とは独立であるためである．ただし，先に述べたようにノード間合成の手法によってスレーブノードのスクリーンサイズを変えていることによる差や，ボリュームデータのサイズでテクスチャデータのサイズが異なることによる差も存在する.

最終画像を 1024×1024 に設定しているため，奥行き方向の解像度も 1024 にするべきではあるが，本研究の目的はノード間合成にあるのでレンダリング部は奥行きの解像度を 1/4 に下げてその時間が短くなるようにした.

4.2 逐次的合成の結果

各スレーブノードが担当するサブボリュームをレンダリングを終えてから，その結果をマスターノードへ逐次的に送信して最終画像が表示されるまでの時間をランレングス符号化 を適用した場合としない場合の両方について，24 の視点で計測した．計測結果を図 18 に示す．縦軸の描画時間の単位は msec である．視点は，図 9 において z の正の方向から x の正の方向へ向かう角度を α，$x-z$ 平面から y の正の方向への角度を β とし，α $= 0$, β $= 0$ から β のみを 15 度ずつ 180 度まで増やした場合と，α $= 45$ 度, β $= 0$ 度 から同じく β のみを 15 度ずつ 180 度まで増やした場合の 2 系統について計測した.

ランレングス符号化は，各スレーブノードで得られた中間画像に対して行い，転送する画像データ量を削減するための手法で，視点による影響は少ない．そのため，連続する透明領域が多い場合は効果的にデータが削減できる．Bonsai と Chest の中間画像は，ランレングス符号化 によって圧縮すると，それぞれ 39〜54 %, 43〜65 % のデータ量になった.

4.3 ２分木による合成の結果

逐次的合成と同様の条件で，各スレーブノードが担当するサブボリュームのレンダリングを終えてから，その結果を他のノードと適宜交換しながらマスターノードで最終画像が表示されるまでの時間を同じ 24 の視点から計測した．このノード間合成時間の計測結果を図 18 に示す．その時の重複度の分布を図 19 に

示す．

ブロック分割を行ったときのブロック 1 つの大きさは図 18 では 4×4 である．

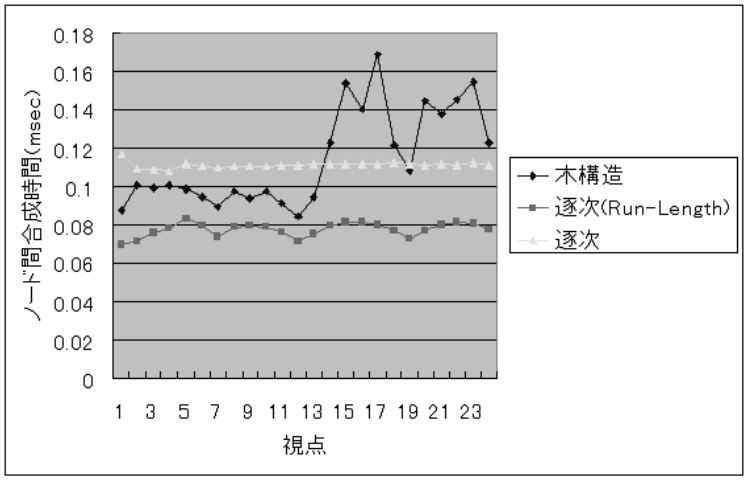

図 18: 各合成手法の描画時間 (Chest)

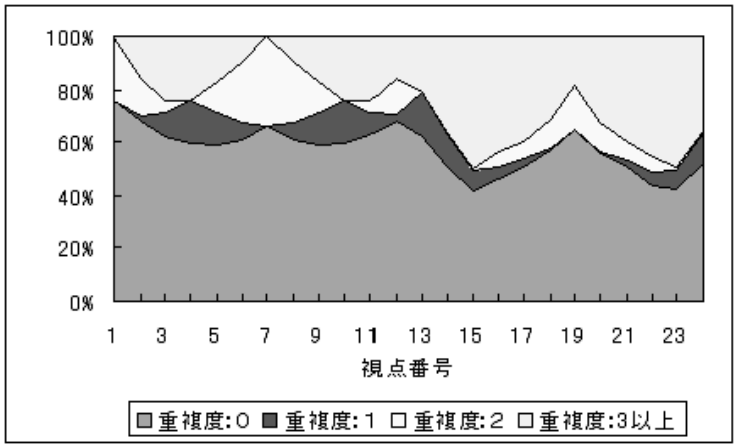

図 19: 各重複度の割合 (Chest)

そして，マスターが受け取った画像データの総量を比較したグラフが図 20 である．1024×1024 の画像を 1 枚表示するために必要なデータ量は 4MB である．

また，中間画像を分割するブロックのサイズを変化させたときの，平均の描画時間（レンダリングとノード間合成を合わせた時間）を同じ視点で計測した

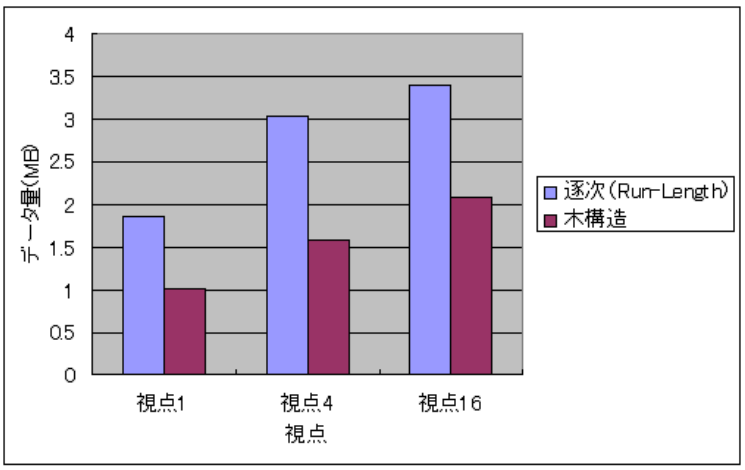

図 20: マスターの総受信量 (Chest)

結果を表 2 に示す.

ブロックサイズ	2	4	8	16
描画時間 (msec)	183.402	179.614	181.214	189.890

表 2: ブロックの大きさによる速度の変化

4.4 考察

図 18，19 を見ると，重複度を考慮した木構造によるノード間合成を行ったときの描画速度は，逐次的にランレングス符号化を行った合成に劣っている．また，重複度が 3 以上のブロックが増える視点のときは，急激に描画時間が増加している．

マスターノードがアイドル状態にならないように，スレーブノードがマスターノードへ画像データを送っているようにしたい．逐次的合成ではそれが実行されているが，合成はマスターノードが行っていた．スレーブノードで合成も行いながらマスターが常に受信しているようにできれば逐次的合成よりも描画速度が速くなると思われる．マスターノードの処理の中で，どこに時間がかかっていたかを調べると，次のようになった．この時間は，ボリュームデータを真正面から見た場合である．つまり，画像データを持つ全てのブロックの重複度は 2 である．計測した時間は，次のように計測した．画像データを受信する前に int 型の変数 p を必ず受信するようにしているので，各受信操作について，マ

スターノードが動き始めてから変数 p を受信するまでの時間を測った．(4) などは，図 15 の矢印をさす番号である．時間の単位は msec である．

1 :マスターノードが動き始めてから 1 つ目の受信 (4) を始めるまで：43.0
これは，ほぼレンダリングの時間である．

2 :1 つ目の (4) を受信し始めてから最後の (4) を受信し始めるまで：1.5
真正面から見た場合なので，(4) で受信する重複度が 1 のブロックは存在しないため非常に小さい．

3 :(4) が終わってから，(8) の受信を始めるまで：15.3
この時間は，マスターはずっとアイドル状態である．

4 :1 つ目の (8) の 1 回目を受信し始めてから 1 つ目の (8) の 2 回目を受信し始めるまで：13.7
1 つ目の (8) の 1 回目で 4×4 ピクセルのブロックを 1059 個（67.8kB）受信し，それを最終画像に貼り付けている．

5 :1 つ目の (8) の 2 回目を受信し始めてから 2 つ目の (8) の 1 回目を受信し始めるまで：11.2
1 つ目の (8) の 2 回目で 4×4 ピクセルのブロックを 850 個（54.4kB）受信し，それを最終画像に貼り付けている．

6 :2 つ目の (8) の 1 回目を受信し始めてから 2 つ目の (8) の 2 回目を受信し始めるまで：13.9
2 つ目の (8) の 1 回目で 4×4 ピクセルのブロックを 1073 個（68.7kB）受信し，それを最終画像に貼り付けている．

7 :2 つ目の (8) の 2 回目を受信し始めてから 1 つ目の (9) を受信し始めるまで：11.9
2 つ目の (8) の 2 回目で 4×4 ピクセルのブロックを 914 個（58.5kB）受信し，それを最終画像に貼り付けている．

8 :1 つ目の (9) と 2 つ目の (9) で受信するブロックは合わせて 69 個：合わせて 1msec 以下

この 1〜8 の処理のうち，3 ではマスターは仕事をしていない．これは，この段階のスレーブノードによる合成に時間がかかっているからである．この段階の合成とは，図 15 の (6) の後の合成に相当する．この合成を行うスレーブノードは 2 台存在するので，どちらかは合成を行わずにマスターノードへデータを送信し，もう片方は通常通りに合成を行うようにすれば高速化が可能ではない

かと考える．

また，上記の 1〜8 の時間は，ボリュームデータを真正面から見ているのでやや特殊な場合である．図 19 を見ると，重複度が 3 以上のブロックの割合が大きくなっている．これはつまり，図 15 の (9) で転送するデータが多いことを示す．現在の実装では 8 台のスレーブノードとは別にマスターノードを用いているが，リモートホストを用いることの利便性を考えなければ 8 台だけで並列ボリュームレンダリングを行ったほうが速度は速くなる．というのは，木構造の最後のステップのデータの受信が 1 つ行われなくなるためである．

第 5 章　まとめ

本稿では，汎用グラフィクスハードウェアによる並列ボリュームレンダリングのノード間重畳処理に，重複度を考慮した木構造による手法を実装し評価を行った．その結果，現在の実装のままでは逐次的合成のときに問題となっていたアイドリング時間が再び問題となり，更にノード間の通信や合成に時間がかかる状態であることが分かった．また，合成をせずにただコピーだけ行われるブロックが合成処理のたびに現れてしまうことも，全体の高速化の妨げとなる．ポインタを使ったデータ構造にすると，ノード間で通信が行われるたびに新しくポインタをつける必要がでてくることになる．現状では，ブロックの大きさが 4×4 ピクセルの時が最速であり，スクリーンサイズに比べて非常に小さいのでブロックごとにポインタをつけるようにするとそのオーバヘッドも生じるおそれがある．本稿で述べた手法は重複度に拠りすぎていて，最後のステップの送信以外は完全に合成が終わったブロックのみマスターノードに送るようにしていたためにこのような結果になった．最後だけではなく，途中でもマスターが合成を行うようにすれば高速化が行えるのではないだろうか．

現在我々は，1 台の PC をマスターノード，8 台の PC をスレーブノードとしてクラスタを組んでいる．今後はより大規模なデータを秒間 30 枚で描画する実時間可視化を実現するために，クラスタの規模を大きくする予定である．本稿では逐次的合成が最も速いという結果になった．しかし，クラスタの規模が大きくなるにつれて逐次的合成で行ったときの問題がより顕著に現れるのは明らかである．このことからも，より効率的なノード間の合成手法が必要となる．

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚なる謝意を表します.

また，本研究に関して適切な御指導を賜った森眞一郎助教授，中島康彦助教授，五島正裕助手に心から感謝致します．さらに，日頃からご助力頂いた京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝致します.

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夫氏よりご提供いただきました．盆栽のボリュームデータは volvis から利用させていただきました.

参考文献

[1] 中嶋 正之, 藤代 一成: コンピュータヴィジュアリゼーション, 共立出版, 2000

[2] Mason Woo, Jackie Neither and Tom Davis: OpenGL プログラミングガイド, ピアソンエデュケーション, 1997

[3] 丸山 悠樹, 中田 智史, 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, 情報処理学会論文誌, June, 2004

[4] Kwan-Liu Ma, James S. Painter, Charles D. Hansen and Michael F. Krogh: Parallel Volume Rendering Using Binary-Swap Image Compositon, IEEE Comput. Graph. and Appl., July, 2003

[5] 緒方 正人, 村木 茂, マ クワンリュウ, 劉 学振: 高並列ボリュームレンダリングを可能にするパイプライン画像重畳装置の設計と評価, 電子情報通信学会論文誌 Feb., 2003

[6] Aleksander Stompel, Kwan-Liu Ma, Eric B.Lum, James Ahrens and John Patchett: SLIC:Scheduled Linear Image Compositing for Parallel Volume Rendering, IEEE Symposium on Parallel and Large-Data Visualization and Graphics, Oct., 2003