(d) 情報

グリッドコンピューティングを用いた大規模ボリュームデータの可視化

Visualization based on Grid Computing for Large-scale Volume Data

西橋　邦彦 †　檜垣　徹 †　玉木　徹 †　金田　和文 †

Kunihiko Nishihashi †　Toru Higaki †　Toru Tamaki †　Kazufumi Kaneda †

† 広島大学　大学院工学研究科

1　概要

ボリュームレンダリングとは，3 次元格子状のボリュームデータを任意の視点から，対象物の内部までを透かした画像を生成する技術である．医療現場では，CT (Computed Tomography) や MRI (Magnetic Resonance Imaging) などの計測装置から得られたボリュームデータを可視化することで，直観的な観察が可能となり，医師による診断を支援している．ボリュームレンダリングの主な手法に，レイキャスティング法 [5] とスプラッティング法 [4] がある．そのどちらについても，観察対象となるボリュームデータの規模が大きくなれば非常に計算コストが大きくなる．具体的には，ボリュームデータにおける 1 辺のサイズを n ボクセルとすると，ボリュームレンダリングの時間計算量および空間計算量は $O(n^3)$ であるため，大規模なボリュームデータを表示しようとすれば，それだけ時間がかかり，インタラクティブな観察は困難となる．

そのため，高計算コストであるボリュームレンダリングを高速化するために様々な取り組みが行われている．GPU を用いた手法として，[1, 2] では，可視性順序を高速にソートすることで高速化を図っている．また，[3] では，視線方向に対して垂直にボリュームデータをスライスする．それをグループ化し厚みをもたせたスラブとし，ERT 法 [6] の考え方を用い，空のスラブの計算を省くことで計算量を削減している．

並列分散処理アーキテクチャを用いた手法として，[11, 12, 10] では，分散メモリ型並列計算機を用いることで計算を並列化している．また，[13] では，グリッドコンピューティングを用いることで計算を分散化している．本研究では，医療機関や研究施設などに多数ある遊休計算機を有効利用できるという点から，グリッドコンピューティングを用いたボリュームレンダリングの高速化に着目する．

分散ボリュームレンダリングにおいて，ボリュームデータは分割されて各計算機で個別にレンダリングされ，最終的にスクリーンへ合成される．スクリーンへの合成の際，不透明度は重なりをもつ前面側の分割ボリュームデータ (以降，サブボリューム) の影響を受けるため，自分を遮蔽しているすべてのサブボリュームのレンダリング結果がスクリーンへ合成されないと，自分をスクリーンへ合成することはできない．そのため，サブボリュームを各計算機へ送信する順序も工夫する必要がある．

グリッドコンピューティングの場合，上述の高速化手法をそのまま用いて高速化を実現することは望ましくない．なぜならば，グリッドコンピューティングでは各マシンの利用状況によっては処理中に割り当てられる計算パワーが大きく変動するため，ジョブの投入順に処理結果が返ってくるとは限らないからである．そこで，各サブボリュームの遮蔽関係を記憶するオブスタクルフラグを提案する．このオブスタクルフラグを動的に更新することで，最新の遮蔽状況を把握することができる．その遮蔽状況に基づいたジョブスケジューリングを行うことで，全体的な処理の効率化を実現できる．

以降では，まずグリッドコンピューティングを用いたボリュームレンダリング (2 章) について述べる．次に提案手法を示し (3 章)，提案手法を組み込んだ処理手順 (4 章)，評価実験の結果と考察を述べる (5 章)．そして最後にまとめを述べる (6 章)．

2　グリッドコンピューティングを用いたボリュームレンダリング

グリッドコンピューティングでは，一般的に図 1 に示すようなシステムが利用されている．ボリュームデータを分割して各サブボリュームを個別にレンダリングし，スクリーンへレンダリング結果を合成することで最終画像を得る．この処理の流れに従って，大規模なボリュームデータを分散計算して高速化を図る．

ここで重要な点は，各 Agent でのレンダリング結果を合成する順番である．合成順序としては，スクリーンから遠いものから近いもの Back-to-Front と，その逆の Front-to-Back がある．どちらの方法でも本質的には同じ問題をかかえることになるが，提案手法では不透明度が”1”になったときの計算打ち切りが行えるという点で，Front-to-Back を採用する．この場合，レンダリング結果はスプラッティング法のフットプリント同様，他のサブボリュームのレンダリング結果に遮られない (可視性の高い) サブボリュームのレンダリング結果から順に合成を行う必要がある．しかし，グリッドコンピューティングでは既存の計算機を Agent として使用するため，各 Agent の処理能力は不均一であり，また計算途中で第三者に使用されることもある．そのため，単にスクリーンから近いサブボリュームから順に処理を行うと，処理途中で Agent がそのコンピュータ本来の業務に利用された場合や，処理に時間のかかる低スペックな Agent へ送信されてしまった場合に，大きな問題が生じる．

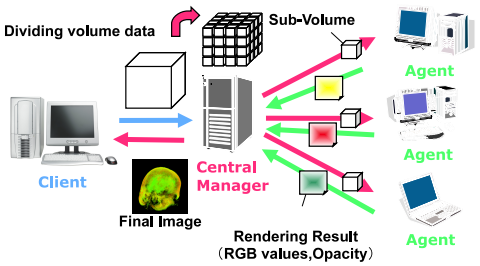


図 1　システム構成

2.1　サブボリューム

ボリュームデータの分割方法として，[3] のように視線方向に対して垂直にスライスしたり，[10] のように各軸方向ごとに切り分けたり，[7] のように Octree 構造に分割する方法がある．このようにボリュームデータの分割を工夫することでレンダリングの処理効率を高めたり，データ容量の削減を図ることができる．しかし，Octree 構造では Central Manager での処理量が増加したり，合成の際の順序が複雑になってしまう．またスライス分割では，合成の際の順序の柔軟性がなくなるため，グリッドコンピューティングには不向きである．

本研究では，図 2 に示すようにグリッド状に等分割する．その理由は前節で述べたように，ある Agent での処理が途中でスローダウンしてしまった場合でも，スライス分割のようにそれ以降のサブボリュームを合成することができず，全体的な処理効率の低下を引き起こしてしまう恐れがなくなるからである．そして，分割構造の単純さから次章で述べるオブスタクルフラグを用いて，効率良くジョブスケジューリングが行えるからである．また，一度分割すれば視線方向が変わっても分割し直さなくてよいという利点もある．

サブボリュームの可視性は，図 2 に示すようにスクリーンに近いものほど高くなる．ここで同色は，可視性が同じレベルにあることを示している．すなわち，合成の際には同色のサブボリューム同士は，どのような順番で合成しても良いことを意味する．この柔軟性がグリッドコンピューティングでは非常に重要となる．

2.2　静的ジョブスケジューリングの欠点

ここでは，説明の便宜上 2 次元に置き換えて説明する．各サブボリュームは各々が遮蔽関係にある．そのため，ボリュームレンダリングの物質内部までを表示するという性質上，レンダリング結果の合成を行う際には順序を考慮する必要がある．合成順序はスクリーンからの可視性に基づき，可視性の高いものから順に合成しなければならない．そのため，効率的な合成処理を行うためにはジョブの送信も可視性の高いものから順に行う．また，Agent へ送信するのは合成可能になったサブボリュームとする．サブボリュームの可視性はスクリーンと，他のサブボリュームの位置関係から決定でき，決定された順序

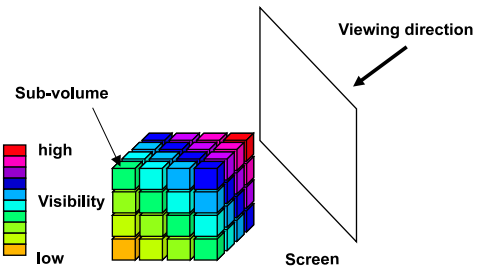
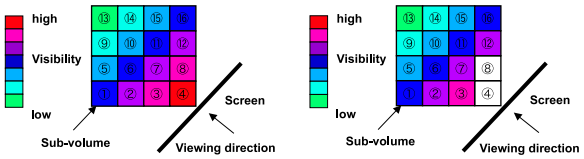


図 2　サブボリュームの可視性

を基に，あるサブボリュームが合成済みになると次のサブボリュームが合成可能となるため，Agent へ送信可能となる．

初期状態での可視性に基づくジョブスケジューリングによって分散ボリュームレンダリングを行った例を図 3 に示す．図 (a) のように，視線方向に基づく初期状態での可視性は，スクリーンに近いほど高い．図 (b) は，サブボリューム 4 とサブボリューム 8 が合成済みになった場合を示す．この状態では，サブボリューム 12 はすでに遮蔽物がなく合成可能なため Agent へ送信可能であるが，初期状態で決定したジョブ送信順序，つまり静的ジョブスケジューリングに基づくと，サブボリューム 3 がまだ合成されていないと，初期状態ではサブボリューム 3 の方が可視性が高いため，サブボリューム 12 は送信できない．静的ジョブスケジューリングでは，このようにジョブの送信待ちが発生し，全体的な処理効率の低下を引き起こしてしまう．これにより，ジョブの送信順序を決定するジョブスケジューリングがいかに重要であるかわかる．



(a)　(b)

図 3　静的ジョブスケジューリング

なお，ジョブスケジューリングを一切考えず，サブボリュームを一斉に Agent へ送信する方法もある．しかし，可視性の高い順に送信してもグリッドコンピューティングの性質上，その順番にレンダリング結果が返ってくるとは限らない．そのため合成待ちが増えてしまい，処理結果を合成するまで一時的に保存しておくメモリを圧迫してしまう恐れがある．

静的ジョブスケジューリングの問題を解決するには，レンダリング結果が合成されるごとに変化するサブボリュームの遮蔽状況を動的に管理すればよい．そこで，次章で遮蔽状況を動的に管理する機構としてオブスタクルフラグを提案する．

3　動的ジョブスケジューリング

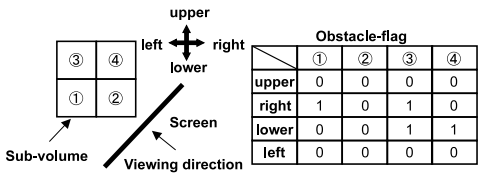
レンダリング結果が合成されるごとにサブボリュームの遮蔽状況は変化する．また，スクリーンへ合成可能であるかは可視性に依存し，あるサブボリュームによってのみ遮られているサブボリュームは，遮っていたサブボリュームが合成されると可視になるため合成可能となる．そこで，遮蔽状況を動的に管理するためにオブスタクルフラグを提案し，それを用いた動的ジョブスケジューリングについて説明する．

3.1　オブスタクルフラグ

ここでは，説明の便宜上 2 次元に置き換えて説明する．サブボリュームはそれぞれが遮蔽関係にあり，合成の際に遮蔽物となるものは，ボリュームデータをどの視線方向から観察するかによって変化する．また，あるサブボリュームに対する遮蔽物は，多くとも 2 つの辺に接するサブボリュームとなる．そこで，注目するサブボリュームに遮蔽物が存在するか否かの情報を保存するため，4 辺分 (4 ビット) のデータ構造を用意する．そして，上下左右それぞれで視線方向に対して直接遮蔽するサブボリュームが存在すれば 1，なければ 0 をフラグとして保存する．そのため，注目するサブボリュームのオブスタクルフラグの全てのビットが 0 であれば，そのサブボリュームがどのサブボリュームからも遮蔽されていないことを示しているので，合成可能であることがわかる．このオブスタクルフラグは，全てのサブボリュームに一つずつ用意する必要がある．

図 4 に示す視線方向からサブボリュームを見た場合，各サブボリュームのオブスタクルフラグは図のようになる．例えば，サブボリューム 3 に注目すると，サブボリューム 3 は右側をサブボリューム 4，下側をサブボリューム 1 によって遮蔽されている．そのため，サブボリューム 3 のオブスタクルフラグは図のようになる．

また，オブスタクルフラグの 1 が立っている個数を care 数と呼ぶことにする．例えば，図 4 では，サブボリューム 2 の care 数は 0，サブボリューム 1 の care 数は 1，サブボリューム 3 の care 数は 2 である．3 次元の場合には，各サブボリュームのオブスタクルフラグのビット数は，面での隣接となるため，6 となる．そして，care 数の最大値は 3 となる．



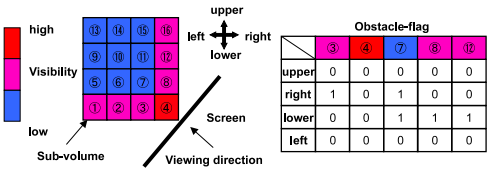
Obstacle-flag

	①	②	③	④
upper	0	0	0	0
right	1	0	1	0
lower	0	0	1	1
left	0	0	0	0

図 4　Obstacle-flag

3.2　オブスタクルフラグに基づくサブボリュームの可視性

2.2 節で述べたように，静的ジョブスケジューリングであると，送信待ちのジョブが発生するため全体的な処理効率の低下を引き起こす．しかし，オブスタクルフラグを参照すると，遮蔽物が存在するか否かを動的に管理できる．図 5 はオブスタクルフラグに基づくサブボリュームの可視性を示したものである．提案手法での可視性のレベルは，オブスタクルフラグの care 数によって設定する．図 3 と比べると，同じ視線方向にもかかわらず，同じ可視性に属するサブボリュームでも，例えばサブボリューム 7 と 12 のように care 数が異なっている．



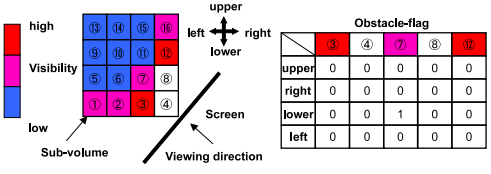
Obstacle-flag

	③	④	⑦	⑧	⑫
upper	0	0	0	0	0
right	1	0	1	0	0
lower	0	0	1	1	1
left	0	0	0	0	0

図 5　オブスタクルフラグに基づくサブボリュームの可視性

3.3　オブスタクルフラグを用いた動的ジョブスケジューリング

図 6 は，サブボリューム 4 と 8 が既に合成済みの場合である．サブボリューム 4 と 8 が遮蔽していたサブボリューム 3, 7, 12 のオブスタクルフラグが更新されている．静的ジョブスケジューリングでは，サブボリューム 12 が合成可能であっても初期状態での可視性に基づいたジョブスケジューリングであるためサブボリューム 3 が合成されないと Agent へ送信することができなかった．しかし，図 6 に示すようにサブボリューム 12 のオブスタクルフラグを参照すると，フラグがすべて 0，すなわち care 数が 0 であるため送信可能と判断できる．よって提案手法での care 数を用いた動的ジョブスケジューリングであると，ジョブの送信待ちが起こらない．また，サブボリュームが合成されるごとに変化する遮蔽状況も動的に管理できていることがわかる．



Obstacle-flag

	③	④	⑦	⑧	⑫
upper	0	0	0	0	0
right	0	0	0	0	0
lower	0	0	1	0	0
left	0	0	0	0	0

図 6　オブスタクルフラグを用いた動的ジョブスケジューリング

4 分散ボリュームレンダリング処理手順

オブスタクルフラグを用いた動的ジョブスケジューリングを組み込んだ分散ボリュームレンダリングの処理手順について説明する．図 7 に示すように，まず Sub-volume List と Obstacle-flag を生成する．Sub-voume List はサブボリュームを管理するリストで，どのサブボリュームがまだレンダリングされていないかを把握するために用いる．オブスタクルフラグを参照して送信可能なサブボリュームがあれば，Processing List へサブボリュームのインデックスを格納するとともにジョブを Central Manager へ送信する．Processing List は現在処理中のサブボリュームを管理するリストで，スクリーンへの合成が終わったかどうかを把握するために用いる．オブスタクルフラグの更新は，スクリーンへの合成がされる度に行う．

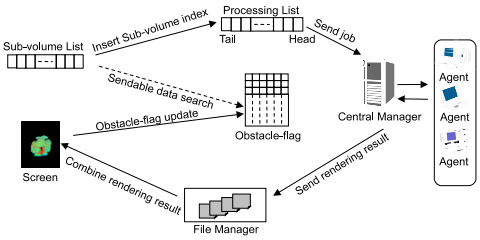


図 7　分散ボリュームレンダリング処理手順

この処理手順では，合成可能なサブボリュームでなければ Agent への送信を行わない．その理由は，もし合成可能でないサブボリュームのレンダリング結果が返ってきてしまった場合，合成可能になるまでメモリに蓄えておかなければならず，それが増えると一時的にメモリを圧迫してしまう恐れがあるからである．これは，Agent の処理能力が不均一というグリッドコンピューティングの性質上，高い確率で起こりうる．分割数やボリュームデータサイズが増えれば尚更である．しかし，その反面 Agent が多ければ多いほど Agent 利用台数の低下を引き起こす．そこで，次節で Agent 利用台数を向上させる例外処理について述べる．

4.1 例外処理による Agent 利用台数の向上

図 8 は，空き Agent があれば，まだ合成可能でないサブボリュームであっても，現在 Agent で処理中のジョブが終了すればただちに合成可能となる状態にあるものを送信する例外処理を組み込んだ処理手順である．空き Agent があれば，例外処理として Processing List を走査し，現在処理中のサブボリュームが直接遮蔽しているサブボリュームを探索する．そして，そのサブボリュームのオブスタクルフラグを参照し，care 数が 1 であれば Processing List へ追加するとともにジョブを Central Manager へ送信する．この作業を空き Agent がなくなるか，条件と合致するサブボリュームがなくなるまで繰り返す．この例外処理によって，合成待ちによるメモリの圧迫を極力防いだ Agent 利用台数の向上が実現できる．

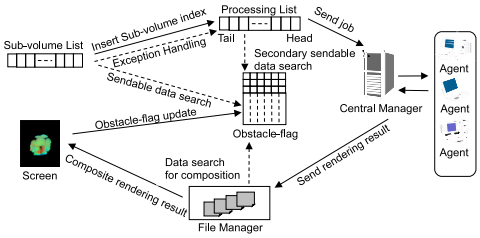


図 8　例外処理を組み込んだ処理手順

5 評価実験

5.1 シミュレーションによる検証

シミュレーションによって，提案手法の有用性の検証を行った．1 つのサブボリュームのレンダリングに 1000sec，500sec，250sec かかるものを 4 台ずつ，計 12 台の Agent を定義した．サブボリュームは全部で 100 個ある．また，Agent は定期的に第三者により使用され，使用されている間はサービスを中断することとする．既に投入されたタスクがあり，未完の場合は処理を中断し，第三者の使用が終了するのを待ち，その後再開する．それぞれの Agent の第三者による使用スケジュールを図 9 に示す．凸状態が，第三者による使用状態を表し，凹状態が，ジョブ受け入れ可能状態を示す．この使用スケジュールは，高スペックな計算機ほど busy と idle の繰り返しの周期が長くなるような状況を想定した．図からわかるように，常時 6 台の Agent がジョブ受け入れ可能状態となっている．

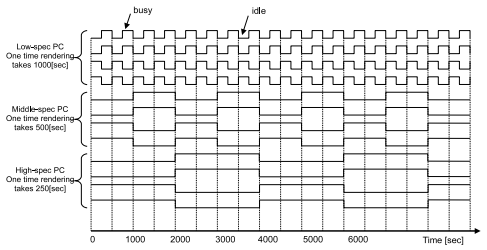


図 9　Agent スケジュール

動的ジョブスケジューリングと静的ジョブスケジューリングの場合では，以下の条件のもとジョブ投入を行うこととする．

＜動的ジョブスケジューリング＞
Agent がジョブ受け入れ可能状態であること．視線方向に対して，自分を遮蔽しているサブボリュームがレンダリング済みであること．
＜静的ジョブスケジューリング＞
Agent がジョブ受け入れ可能状態であること．視線方向に対して，自分を遮蔽しているサブボリュームがレンダリング済みであること．未投入のサブボリュームの中で，一番可視性が高いこと (追い越し厳禁)．

5.1.1 シミュレーション結果

シミュレーション結果を表 1 に，レンダリング進行率のグラフを図 10 に示す．動的ジョブスケジュールを行うことで，経過時間を約 65% まで削減することができている．これは，サブボリュームの処理状況に応じて，動的にジョブクスケジュールを行っているため，レンダリング可能なジョブが常時適切に選択できているためである．これによって，利用可能な Agent を効率的に利用できていることもわかる．

表 1　シミュレーション結果

	Dynamic Job-scheduling	Sequencial Job-scheduling
Elapsed Time [sec.]	10,000	15,500
Agent Utilization [%]	54.2	38.4

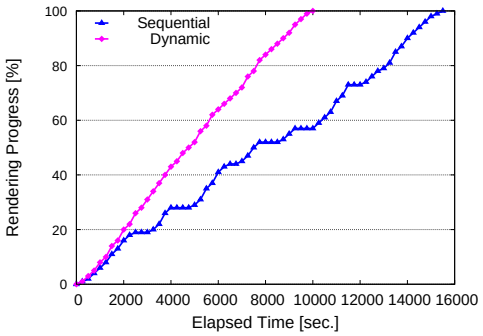


図 10　レンダリング進行率

5.2 テストデータを用いた実機環境での検証

グリッドシステムには，広島大学情報メディア教育研究センターが管理を行っているキャンパスグリッドを使用した [9]．このキャンパスグリッドは，ジョブスケジューラとして Condor[8] を使用している．キャンパスグリッドを構成して

いる Agent のスペックを表 2 に示す．但し，提案手法を組み込んだプロトタイプシステムにとって，次のような不利な条件がある．

広島大学のキャンパスグリッドでは MPI Universe が使用できないため，Standard Universe による逐次的なジョブの投入を行う．そのため，我々が提案しているオブスタクルフラグを用いた動的ジョブスケジューリングを経て，更に Central Manager によって逐次的なスケジューリングがされることになる．これによって，無駄な時間を費やしてしまうことになる．また，キャンパスグリッドには SCore をミドルウェアとしたクラスタも共存しているため，Condor と SCore の競合によって，ジョブのマッチメイキングに時間がかかってしまう．それによって Central Manager から Agent へのジョブの送信間隔が 1 秒以上かかってしまうため，ジョブ数が増えれば増えるほど影響が大きくなり，台数効果が得られない．

実験で使用したテストデータの各種パラメータを表 3 に示す．

表 2　実験環境

Agent 数	OS	CPU	Memory
34	Linux	Xeon 3.06GHz	2GB
469		Pentium4 3.06GHz	990MB

表 3　テストデータパラメータ

解像度 [voxel]	VD size [GB]	分割数	SV size [MB]	Screen size [pixel]
2048^3	16	64	256	3000 × 3600
		512	32	
4096^3	128	512	256	5800 × 7200

VD：ボリュームデータ, SV：サブボリューム

5.2.1　実験結果

図 11 に、経過時間を比較したグラフを示す．但し，Seq：静的ジョブスケジューリング，Dyn：動的ジョブスケジューリング，EH：例外処理を表す．

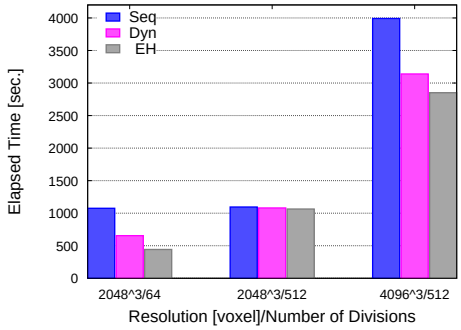


図 11　経過時間の比較

実験結果をまとめると，4096^3 を 512 分割したものと，2048^3 を 64 分割したものでは，動的ジョブスケジューリングや例外処理による経過時間の短縮がされている．4096^3 を 512 分割したものと，2048^3 を 64 分割したものの共通点は，表 3 からわかるように分割ボリュームデータサイズが 256MB ということである．つまり，Agent1 台当たりの処理時間が長いほど動的ジョブスケジューリングの効果があらわれてくると考えられる．そこで，次節で Agent での処理時間に対する効果を調査した実験結果を示す．

5.3　様々な Agent の処理時間における実機環境での検証

Agent1 台あたりの処理時間に応じて動的ジョブスケジューリングの効果がどのように変化するかを調査した．Agent1 台当たりの処理時間は，60 秒，120 秒，180 秒，分割数は 64 とした．

図 12 に，経過時間の比較を，図 13 に Agent 平均利用台数を示す．図 12 より，Agent での処理時間がかかればかかるほど動的ジョブスケジューリングの有用性が顕著になっていることが確認できる．

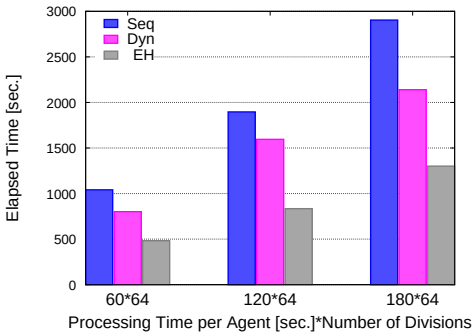


図 12　経過時間の比較

6　まとめ

本論文では，オブスタクルフラグを用いた動的ジョブスケジューリングを提案し，グリッドコンピューティング環境下での実験を行った．その結果，静的ジョブスケジューリングの効率の悪さを，提案手法によって解決できることを示した．

シミュレーションでは，提案手法による Agent 利用率，経過時間の短縮が確認できた．また，レンダリングの進行状況も提案手法のほうが滑らかであった．

実機環境下での実験では，提案手法による Agent 利用台数の向上，経過時間の短縮が確認できた．Agent での処理時間が長いほど，動的ジョブスケジューリングや例外処理が効果的に働いた．

今後の課題として，より大規模なボリュームデータでの検証，ボリュームデータの分割数の増加，大規模な実験環境での検証，より効率の良いジョブスケジューリングの提案などが挙げられる．

参考文献

[1] S. P. Callahan, M. Ikits, J. L. Comba, and C. T. Silva. Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[2] C. Hofsetz, N. Max, and R. Bastos. Object-Space Visibility Ordering for Point-Based and Volume Rendering. *Computer Graphics Forum*, 27(1):91–101, 2008.

[3] H. Y. Keleş, A. Es, and V. İşler. Acceleration of direct volume rendering with programmable graphics hardware. *Vis. Comput.*, 23(1):15–24, 2006.

[4] Lee W. Interactive volume rendering. In *Proc. Workshop on Volume Visualization*, pages 9–18, 1989.

[5] Levoy, M. Display of surface from volume data. *IEEE Computer Graphics & Applications*, 8(3):29–37, 1988.

[6] Levoy, M. Efficient ray tracing of volume data. *ACM Transaction on Graphics*, 9(3):245–261, 1989.

[7] K. Sung. A DDA octree traversal algorithm for ray tracing. In *Eurographics '91*, pages 73–85. North-Holland, 1991.

[8] The Condor Team. Condor Project Homepage. http://www.cs.wisc.edu/condor/.

[9] 広島大学情報メディア教育研究センター. HPC サービス. http://staff.media.hiroshima-u.ac.jp/hpc/HPC_wiki/

[10] 佐々木智充，伊野文彦，藤本典幸，萩原兼一．分散メモリ型並列計算機による高解像度ボリュームレンダリング．**映像情報メディア学会技術報告**, 26(22):7–12, 2002.

[11] 松井学，伊野文彦，萩原兼一．大規模データセットを可視化するための効率の良い並列ボリュームレンダリング．**情報処理学会論文誌：コンピューティングシステム**, 45(11):346–355, 2004.

[12] 松井学，竹内彰，伊野文彦，萩原兼一．累積不透明度の伝搬による並列ボリュームレンダリングの計算量削減．**電子情報通信学会 技術報告書**, 103(249):13–18, 2003.

[13] 檜垣徹，岡部憲史，西橋邦彦，玉木徹，金田和文．グリッドコンピューティングを用いた大規模ボリュームデータの可視化手法に関する研究．**情報処理学会，グラフィクスと *CAD* 研究会**, 2007(84):65–70, 2007.