修士論文

分割統治法を用いたタスク並列 Tree-based AMR アルゴリズム

A Task Parallel Tree-based AMR Using Divide-and-Conquer method

平成 25 年 2 月 6 日提出

指導教員　　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-116422 河野 瑛

概 要

微分方程式の数値シミュレーションは工学・自然科学等様々な分野で重要な問題であり，特に近年並列計算環境を用いた大規模計算によって大きな成果が挙げられている．これらのシミュレーションは一般に計算対象空間を格子を用いて離散化することで行われる．計算の精度は格子のサイズに依存し，より高精度な計算にはより大きな計算量，データサイズが必要となる．しかしながら多くの場合，高精度が必要となるのはシミュレーション領域のごく限られた部分であることが多く，こういった場合には本来必要の無い部分にまで細粒度な格子が用いられることになる．

このように領域中の場所によって求められる精度が異なるようないわゆるマルチスケール問題に対して効果を発揮するのが Adaptive Mesh Refinement(AMR) 法である．AMR 法では高精度が必要な部分に局所的に細かい格子を用いて離散化を行い，全体は粗い格子で離散化を行うことにより，計算量やメモリ量を節約しつつ高解像度の計算を実現する．特に高精度が必要な部分に対して再帰的に空間を均等に分割することによって得られる格子を用いる Tree-based AMR 法に対して，近年大規模な並列計算を行う研究が盛んに行われている．Tree-based AMR 法は空間的に不均一なデータ分布を持ち，また計算が進むに連れて格子が変化していくために，従来用いられてきたようなデータを分轄することによって並列化を行う SPMD モデルを用いて効率的な並列プログラムを記述することは難しい問題である．

一方でこのような並列化の難しいアプリケーションや階層化・複雑化の進む計算機環境を背景に，高生産に並列計算を行うことができる並列プログラミングモデルやプログラミング言語への要請が高まっている．中でもタスク並列モデルは，プログラム中の任意の場所で生成できる並列実行可能な『タスク』を用いた柔軟な並列性の記述が可能であり，特に再帰を用いる分割統治法のようなアルゴリズムも直感的かつ効率的に並列化することができる．また，処理系によって自動的に計算プロセス間の負荷分散がおこなわれるため，動的なアプリケーションに対しても効率的な並列化が可能である．

本研究ではこのような背景から，タスク並列モデルを用いた Tree-based AMR アルゴリズムを提案している．提案手法では，タスク並列処理系による自動的な負荷分散や見通しの良い並列化を実現しているだけでなく，分割統治法に基づいたアルゴリズムにより効率的な処理を可能にしている．特に格子中の隣接セルへのアクセスを伴うような操作に対しては，従来全体で $O(n\log(n))$ の計算コストを要するか，もしくは高速化のためのデータ構造を別に用意する必要があったのに対して，木構造のデータ分散が偏りすぎていないという前提のもとでシンプルなデータ構造を保ちながら $O(n)$ の計算コストを実現している．実際に，移流方程式と拡散方程式の差分解法に基づいた Tree-based AMR シミュレーションの実装を行い共有メモリ環境で良好な並列性能を確認した．

目 次

図目次

表目次

第1章　序論

1.1　大規模数値計算

コンピュータシミュレーションは近年，実験・理論に続く第三の科学と呼ばれ，科学研究における重要な手段となっている［44］．特に近年の並列計算環境の目覚しい発展に伴ってますます大規模なデータや計算を扱うことが可能になっており，数値シミュレーションの重要性は年々増している．最新の研究では，IBM のスーパーコンピュータ BlueGene/Q で 786,432 コアを用いて約２兆粒子による宇宙物理のシミュレーションを行った例や［17］，京コンピュータで 442,368 コアを用いて 107,292 原子によるシリコンナノワイヤの電子状態計算を行った例などがある［18］．

これらのコンピュータシミュレーションでは様々な問題が対象となるが，その中でも特に偏微分方程式の数値解析は非常に重要である．偏微分方程式を用いることで自然科学や工学における様々な現象を記述することが出来る．例えば流体の運動を記述するためにはナビエ・ストークス方程式やオイラー方程式が用いられるし，量子力学の中心的な方程式はシュレディンガー方程式である．これらの偏微分方程式によって表現される問題は，多くの場合解析的に解くことが不可能であるため，しばしばコンピュータを用いた数値解析は非常に重要な手段である．

偏微分方程式を離散化して計算するための手法としては，差分法，有限要素法，有限体積法，など様々な方法が考案されているが，これらの手法の中で現在使われている多くは，計算対象空間を格子（メッシュ）を用いて離散化する格子法に属する．格子法では，一般に格子のサイズを細かく取るほど高精度の解を得ることができる．従って高精度なシミュレーションを行うためには細かい格子を用意する必要があるが，当然それに伴って計算量や必要メモリ量も増大することになる．具体的に考えると，3 次元空間のシミュレーションで 2 倍の精度を得るためには，必要なメモリ量は $2^3 = 8$ 倍であり，時間解像度も考えると計算量は 16 倍になる．このように計算精度を高めるためには莫大な計算量の増加が必要となるので，高精度に多次元シミュレーションを行うことは容易ではない．

しかしながら多くの場合，高解像度が必要となるのはシミュレーション領域のごく限られた部分であることが多い．即ち，物質の表面や不連続領域など物理量の勾配が大きな部分のみが高解像度を必要とするのであって，その他の領域は相対的に低い解像度で良いはずである．このように，計算対象中で必要な計算スケールが異なるような部分，性質を持つような問題はマルチスケール問題と呼ばれ，格子法の対象となる問題でも多く見られる．例えば，Bursteede らによる地球全体のマントル対流のシミュレーション［7, 8］の場合では，高解像度が必要なプレートの境界部分では $O(1)$ km の格子サイズが必要なのに対して，他の場所では $O(10^4)$km 程度のサイズの格子で十分な部分も多く，また時間解像度に関しても $O(10^4)$ 年程度から，$O(10^8 - 10^9)$ 年程度までと必要な解像度に大幅な差が存在することが報告されている．従って，空間が 3 次元であることに注意して時間的・空間的な精度を合わせて考えると，一様な格子を用いる計算で十分な解像度を得るためには，$O(10^{16} - 10^{17})$ の自由度が必要であり，現在のコンピュータ環境では現実的な時間で計

算することは不可能である．

1.2　Adaptive Mesh Refinement

このような事実を踏まえて，Adaptive Mesh Refinement (AMR) 法では，高精度が必要な部分に局所的に細かい格子を用いて離散化を行い、全体は粗い格子で離散化を行うことにより、計算量やメモリ量を章約しつつ高解像度の計算を実現する．また時間解像度についても，一般に格子が小さくなれば時間刻みも細かくする必要があるが，AMR 法ではこのタイムステップについても格子の大きさによって別に進行させることで計算量を削減することのできる方法が存在する．従って AMR 法とは，一般的な微分方程式及びその時間発展を扱う問題を，離散化誤差の大きいところで空間的・時間的に局所的に分解能を向上させることにより，僅かな計算機資源で全体的に小さな離散化誤差で計算を行い，かつ計算時間を短縮させる方法，といえる．1.1 章で述べた Burstedde らによるマントル対流シミュレーションでは AMR を用いることによって，一様なメッシュを用いる場合と比べて $O(10^3)$ の自由度を削減できたことが報告されている．

AMR は 1984 年に Berger らによって初めに考案された方法であり［4］，これまでに様々な手法が考案され，研究がなされてきた［4, 5, 7–9, 11, 20, 21, 23–25, 30–32, 34, 36, 38, 39, 41–43, 45］．

AMR は用いる格子の種類によって *Block-structured*, *Unstructured*, *Tree-based* の大きく 3 種類に分けることができるが，本研究では近年特に効率的な大規模並列計算に関する研究が多く行われている Tree-based AMR［7–9, 20, 21, 23, 24, 36, 38, 39, 45］を対象とする．個々の AMR 手法の特徴や得失については 2.1 章で述べる．

Tree-based AMR は 8 分木構造と対応する非構造メッシュを用いるため，空間的にデータの分布が不均一であり，またシミュレーションの時間が進むにつれて格子が変化していく動的な性質を持つので， 並列計算における負荷分散が難しく，見通しよく簡潔に並列化を行うことは簡単ではない．加えて，解の精度を保証するために 2:1 バランス条件と呼ばれるセル間の細分化レベルに関する制約が設けられる事が多く，この制約を満たすための処理（バランス処理）が必要となる．バランス処理は Tree-based AMR における他の操作と異なり，効率的に行うためには処理を行うノードの順番を適切に設計する必要があるため，特に並列プログラムにおいてスケーラブルなアルゴリズムを構築することは重要な問題となっている．

1.3　高生産並列プログラミング言語とタスク並列モデル

一方，ここ数年の計算機の進化の傾向として，マルチコア化，メモリ配置の不均一化（NUMA）・階層化，GPU・アクセラレータ等を用いたヘテロジニアスな計算環境の増加などが顕著である．

現在，並列プログラミングモデルのデファクトスタンダードとなっているのは分散メモリ環境においては MPI (Message Passing Interface)［27］であり，共有メモリ環境においては OpenMP［6］がそうである．これらのプログラミングモデルでは，基本的にハードウェアの物理的な構成に素直にマッピングできるように設計されており，プリミティブなパラダイムであるといえる．従って，実際に行われる計算や通信が把握しやすくチューニングによって性能を引き出しやすいという長所を持つ一方で，逐次プログラムからの飛躍が大きくプログラミングが難しいという欠点をもっている．

例えば MPI では，各実行プロセスにアドレス空間が分離しており，データのやりとりはプロセス間の双方向通信によって行われる．従って，あるプロセス p が自分の持っていないデータ d を得たい時には，d をプロセス q が保有している情報を管理する必要があり，その上でプロセス p がプロセス q から d を受信するということとプロセス q がプロセス p に d を送信するという記述を行う必要がある．

また，MPI や OpenMP では，データを静的に分割することによって並列化を行う SPMD モデルと呼ばれるモデルがよく用いられる．しかし負荷が動的に変化するアプリケーションに対しては，プログラムが進むに連れて負荷分散が崩れてしまうので，データを再分割するなどして調整してやる必要がある．

このようにプログラミングモデルによる制約によって，既存の並列アプリケーションのアルゴリズムやプログラムが複雑になっているという弊害も存在する．特に計算機環境が階層化，複雑化，多様化していくことを鑑みると，こういった弊害は今後ますます顕著になると考えられる．

このような背景から，より高水準・高生産に並列プログラムを記述することのできるプログラミングモデルやプログラミング言語が近年盛んに研究されている．例えば，Intel TBB［35］, Cilk［14］, Chapel［19］, X10［10］といった比較的新しい並列言語では，並列実行可能なタスクの柔軟な生成・実行を可能にする機構を備えている．Chapel や X10, UPC といった言語では PGAS (Partitioned Global Address Space) と呼ばれる仮想的に共有される大域アドレス空間を提供することで分散環境におけるノード間の通信を抽象化して扱えるように設計されている．他にも Charm++［22］では並行オブジェクトを用いた大規模数値計算を行なっている．

本研究では，これらの中でも特にタスク並列モデルに着目する．タスク並列モデルでは，プログラム中の任意の場所で「タスク」と呼ばれる並列処理可能な計算単位を生成することによって並列性を記述する．従って，従来の MPI 等で用いられる SPMD モデル等では難しい，多重ループや再帰等を用いたプログラムも柔軟かつ直感的な並列化できる．また，一般にタスク並列モデルでは SPMD モデルとは異なり，実行可能タスクを実行プロセス数よりも多く生成し，処理系が自動的にタスクの割り当てを行う．これは即ち，細粒度のタスクを生成しておけば処理系が自動的に負荷分散を行うことを意味する．従って今回とりあげる AMR に代表されるような，SPMD モデルでは均一な負荷分散が難しいアプリケーションや，負荷が動的に変動するようなアプリケーションを直感的な並列化を行うのに大変有効な手段である．

1.4　本研究の目的と貢献

以上を鑑みて，本研究では Tree-based AMR を見通しよく効率的に並列化することを目的とし，タスク並列モデルを用いた並列アルゴリズムを提案する．

提案手法では Tree-based AMR における主要な操作全てが再帰を用いた分割統治法によって記述される．特に，ノード間の依存関係から直感的な並列化が難しいバランス操作についても分割統治法を用いて見通しよく並列化を行うことに成功している．特に，バランスアルゴリズムに関しては木構造が偏りすぎていないという前提のもとで $O(n)$ の計算コストで実現している．これは従来の手法が $O(n\log(n))$ ないし，隣接アクセスを高速化するための特別なデータ構造を用いて $O(n)$ を達成していたことに対し，有用な貢献であるといえる．また，提案する並列アルゴリズムでは全体同期を用いず全ての処理を局所同期で進めるため，アルゴリズム上のボトルネックとなる部

分が少ない．

本研究では実験によってバランス操作に対する良好な並列性能を確認した．また，流体力学など様々な分野で用いられる微分方程式である拡散方程式と移流方程式の差分法によるシミュレーションを題材に，Tree-based AMR を実装・評価を行い，提案手法の有効性を示した．

1.5　本稿の構成

本稿の構成は以下のようになっている．

2 章 本研究の対象アプリケーションである Tree-based AMR について，特にメッシュの生成における拘束条件を満たすためのプロセスであるバランス処理について説明する

3 章 本研究で用いる，タスク並列処理系と分割統治法について述べる

4 章 既存研究について述べる．特に Tree-based AMR の並列バランスアルゴリズム及びデータ構造に関する近年の動向について詳しく説明する

5 章 本研究で提案する，分割統治法を用いた Tree-based AMR におけるバランスアルゴリズムについて述べる

6 章 本研究で提案する，タスク並列モデル向けの Tree-based AMR の全体像及び各操作について詳しく説明する．

7 章 性能評価を行う．バランス処理に対する評価と，実際の差分法シミュレーションを用いたアプリケーション全体の評価を行う

8 章 まとめと今後の展望について述べる．

第2章　Tree-based AMR

本章では AMR，特に本研究の対象手法である Tree-based AMR について述べる．2.1 章でまず AMR の分類とその特徴を述べ，2.2 章以降では Tree-based AMR について述べる．

2.1　Adaptive Mesh Refinement の分類

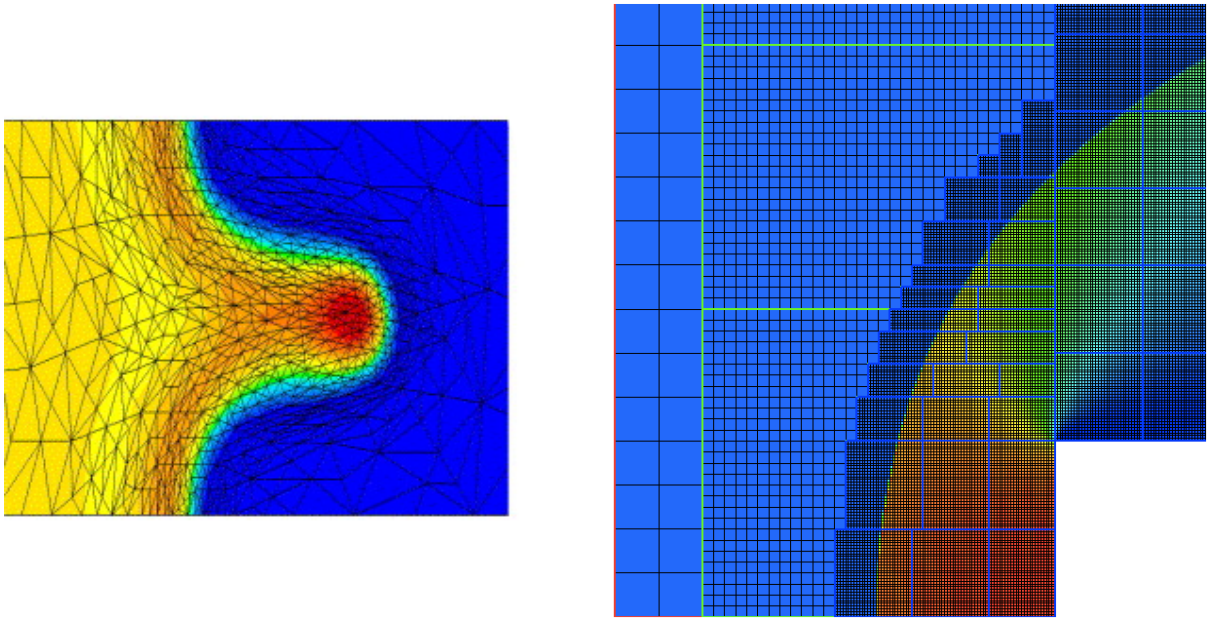
1.2 章で述べたとおり， AMR は用いる格子の種類によって *Block-structured*, *Unstructured*, *Tree-based* の大きく 3 種類に分けられる．以下ではそれぞれについて簡単に述べる．

Block-structured AMR　細分化レベルの異なる複数の矩形の一様メッシュを階層的に重ねて用いる手法で，*patch-based* AMR とも呼ばれる．図 2.1(b) は Cowles らによる Block-structured AMR フレームワーク *SAMRAI* [43] を用いた，流体のダム崩壊シミュレーション [12] の様子を可視化したものである．図中では物理量が色で表現されており，物理量の勾配の大きな部分により細かい格子が重ねられている様子が見てとれる．

Block-structured AMR では一様なメッシュを用いるので，データ構造が単純であり既存の一様メッシュ用のソルバを流用することができるという長所を持つ一方で，複雑な形状に対しては効率的なメッシュの生成が難しいという欠点を持っている．Block-structured AMR の代表的な並列ソフトウェアとしては，Enzo [32], Chombo [11], SAMRAI [43] などがある．

Unstructured AMR　三角形，四面体，六面体等からなる非構造メッシュを用いる手法である．Unstructured AMR では複雑な形状へも柔軟に細分化を行うことができる一方で，一様メッシュのような構造的なメッシュ形状でないために隣接セルの対応関係などのデータ構造を管理する必要があり，メモリ量が大きくなりがちであり，またソルバも複雑になることが多い．図 2.1(a) は Sun らによる薄膜流シミュレーション [37] の様子を可視化したものである．Unstructured AMR の代表的な並列ソフトウェアとしては PYRAMID [30] や ParFUM [25] 等がある．

Tree-based AMR　立方体の形をした計算領域を，各座標軸で均等に分割することによって細分化を行う．この処理を再帰的に繰り返すことによって木構造で表される階層的な細分化がなされたメッシュを得る．Block-structured AMR では細分化レベルの異なる格子を別の格子として扱っていたのとは異なり，全体で 1 つの格子として扱う．3 次元空間ではメッシュは 8 分木の葉と対応し，2 次元では 4 分木の葉と対応する．図 2.2 は Popinet らによる Tree-based AMR ソフトウェア Gerris を用いた，船舶周りの空気の流れシミュレーション [34] を可視化したものである．



(a) Sun らによる Unstructured AMR を用いた薄膜流シミュレーション［37］

(b) Cowles らによる Block-structured AMR フレームワーク”SAMRAI”［43］を用いた，流体のダム崩壊シミュレーション［12］

図 2.1. Block-strucutred AMR 及び Unstructured AMR で用いられるメッシュの例［12, 37］

このように 3 種類の AMR ではそれぞれ異なるアプローチをおこなっており，それぞれ得失がある．本研究の対象である Tree-based AMR はメッシュの柔軟性とデータ構造の簡潔性の観点から Unstructred AMR と Block-based AMR の中間のような性質を持っており，近年特に効率的な大規模並列計算に関する研究が多く行われている［7–9, 20, 21, 23, 24, 36, 38, 39, 45］Tree-based AMR を対象に研究を行う．

2.2　Octree

Tree-based AMR で用いられる格子は木構造と対応しており，3 次元では *octree*，2 次元では *quadtree* と呼ばれる．Octree では子ノードを持つ場合その数は必ず 8 つであり，親ノードの対応する領域を x 軸，y 軸，z 軸でそれぞれ均等に分割してできる 8 つの部分領域が各子ノードに対応する．Tree-based AMR ではこの分割処理を再帰的に適用することによって格子構造を得る．従って，Tree-based AMR におけるメッシュはリーフノードによって構成されており，各ノードは特定の立方体領域と対応する．図 2.3 は quadtree の概念図である．

本稿中では，octree 中の各ノードは *octant* あるいは単にノード，セルと呼び，計算格子すなわちリーフセルの集合を指す場合にはメッシュ，格子と呼ぶ．

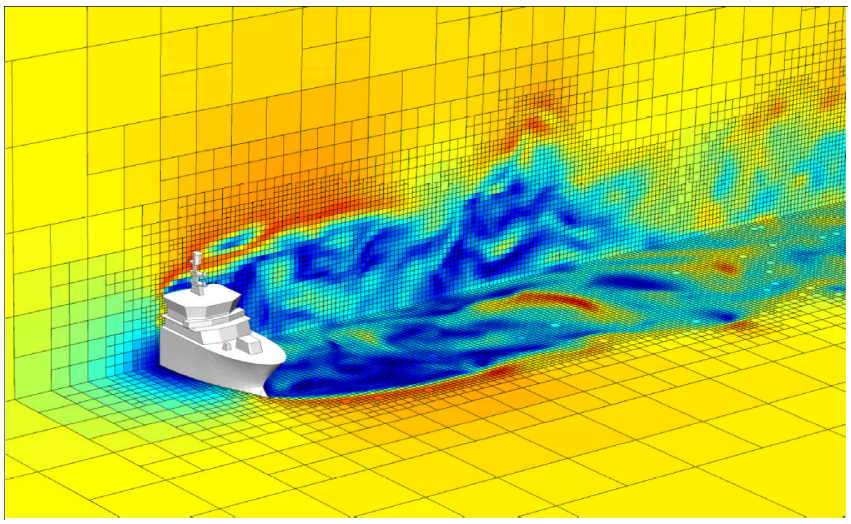


図 2.2. Popinet らによる Tree-basedAMR ソフトウェア Gerris を用いた，船舶周りの空気の流れシミュレーション［34］

2.3　2:1 バランス条件

Tree-based AMR では一般に，使用するメッシュに **2:1 バランス条件**と呼ばれる制約を設ける．これは『互いに隣り合うノードの大きさの比が 1:1 もしくは 2:1 で無ければならない』というもので，以下のように言い換えることもできる．

- 隣り合う領域の細分化レベルの差は高々1 でなければならない

図 2.4 ではこの条件を図示している．こういった制約が設けられるのは，隣り合うセル同士の大きさが異なる場合には粗いセルと細かいセルとの間で計算時には補間が行われるため，解の精度が低下してしまうからである．

バランス条件が適用される隣接ノードの定義は 3 つのタイプが存在する．1 つ目は *2-balanced* と呼ばれ，面を共有するノードに対してバランス条件を課す．従ってこの場合 1 つのノードに対して 6 個のノードとの間でバランス条件を満たしていなければならない．2 つ目は *1-balanced* で，面ないし辺を共有する場合で，18 個のノードとの間で比較されることになる．3 つ目は *0-balanced* で面，辺，頂点を共有するノードに対して定義される．このときには，26 個のノードとの間にバランス条件が課せられる．本研究ではこのうち最も厳しい 0-balanced の場合を前提として議論を行うが，他の場合でも同様である．

2.4　2:1 Octree Balancing

2:1 バランス条件の存在のために，Tree-based AMR ではこれを満たすための処理が必要となる．即ち，octree の全てのリーフノードとその隣接ノードがバランス条件を満たしているかをチェックし，満たされていない場合適切に粗いノードの細分化を行う処理である．この処理は *2:1 Octree*

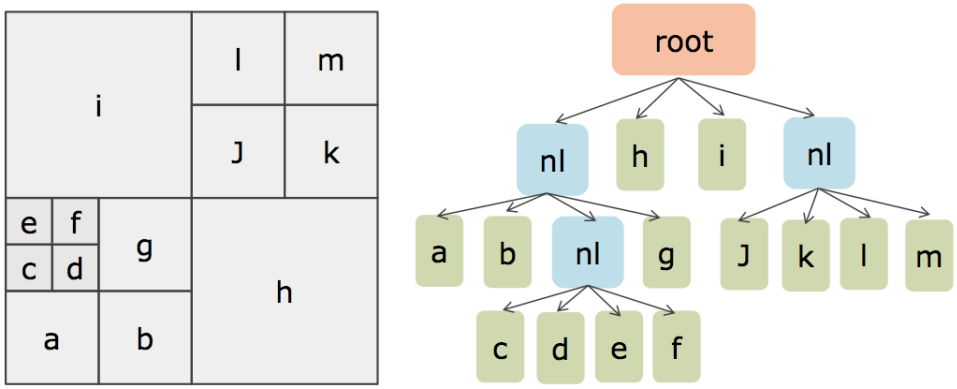


図 2.3. Tree-based AMR で用いられる 4 分木構造（Quadtree）．右図においてアルファベットが書かれているノードはリーフノードであり，左図における同じアルファベットが書かれた領域と対応している．右図中の root と書かれたノードは木構造のルートであり，nl とマークされたノードはリーフノードでは無い（子ノードを持つ）ノードである．

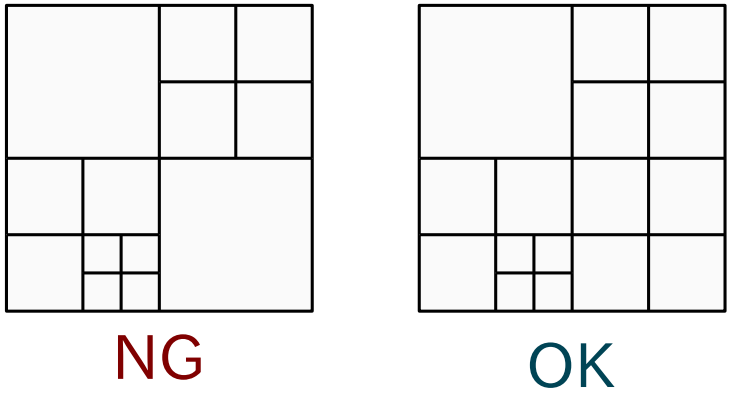


図 2.4. 2:1 バランス条件が課せられたメッシュでは，隣り合う領域は細分化レベルの差が高々 1 でなければならない．左図では右下の 1 レベルしか細分化されていない領域と，3 レベル細分化された領域が隣接しているために，2:1 バランス条件を満たしていない．

balance と呼ばれるが，本稿では，今後この処理をさして『バランス処理を行う』ないし単に『バランスする』と表記する．

バランス処理の最も単純なアルゴリズムは Yerry [46] らによって 1984 年に開発されたものである．この手法では単に木構造中の全てのリーフノードを幅優先探索し，隣接ノードとの細分化レベルの差を比較，必要に応じて細分化を行う．しかしながら，この方法では以下で述べる *ripple effect* と呼ばれる現象の存在から，複数回の施行を必要とする．

Ripple effect とは，あるノードの細分化が引き金となって連鎖的に細分化が起きる現象のことである．図 2.5 はその典型的な状況である．Yerry らの方法では ripple effect が起きた場合に，既にチェックしたノードにおけるバランスが崩れてしまう可能性がある．そのために全てのノードで

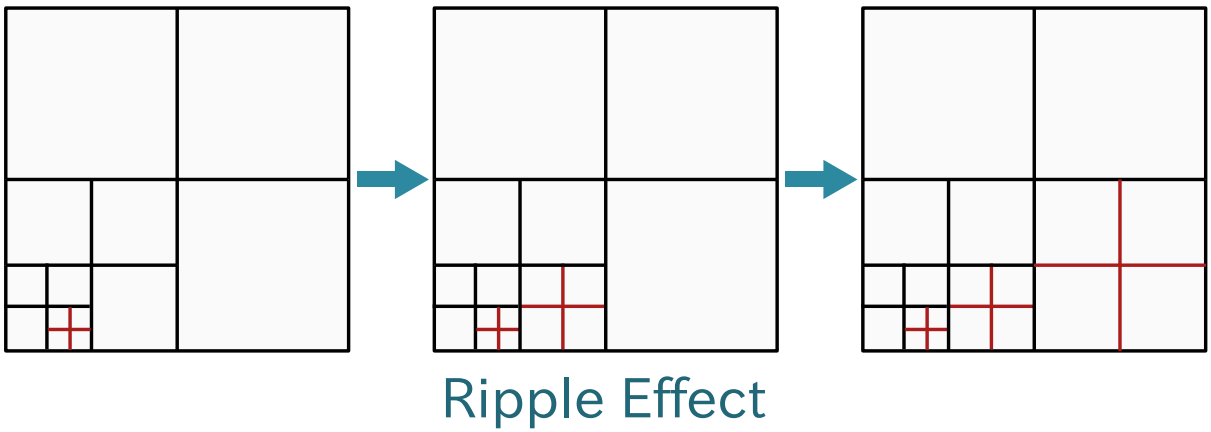


図 2.5. Ripple Effect の例．ある領域の細分化が引き金となって，連鎖的に細分化が引き起こされる．

細分化が起きなくなるまで，全ノードに対する施行を繰り返す必要が生じるのである．

現在 Ripple effect を回避するために多く使われているのは Tu らによって 2005 年に提案された Prioritized Ripple Propagation (PRP) アルゴリズム［41］である．PRP アルゴリズムは，バランス処理によるあるノードの細分化は必ずより細かいノードによって引き起こされる，という観察に基づいている．従って，最も細分化レベルの高い（小さい）ノード群からバランス処理を始め，次に 2 番目に細かいノード群，という具合に細かいノードから粗いノードに順番に処理を進めていけば良いという訳である．

このように，Ripple effect の影響を受けず効率的にバランス操作を進めるためには適切な順番で各ノードに対する処理をを行う必要があり，ノード間に依存関係があるといえる．この依存関係がバランス操作を複雑たらしめる要因であり，特に並列計算を行う際には顕著になる．並列アルゴリズムに関しては 4 章で述べる．

第3章　タスク並列モデルと分割統治法

3.1　タスク並列モデル

1.3 章で述べたように，タスク並列モデルではプログラム中の任意の場所で「タスク」と呼ばれる並列処理可能な計算単位を生成することによって並列性を記述する．これらのタスクは処理系によって自動的に実行プロセスにスケジューリングされる．従ってデータを静的に分割して各実行プロセスに割り当てる SPMD モデルに比べて柔軟な並列化が可能である．例えば逐次プログラム中の多重ループや再帰などの制御構造は SPMD モデルで並列化する際にはプログラム構造を変更する必要があるが，タスク並列モデルではタスクの生成とそれらのタスクの終了同期を指示を意味する文を追加するだけで並列化が可能であり，直感的であるといえる．

3.2　分割統治法

分割統治法は大きな問題を小さな問題に再帰的に分割していき，解かれた小さな問題を逆順に統合していくことで最終的に当初の問題の解を得る手法である．分割統治法はマージソートや高速フーリエ変換等で用いられるよく知られた手法であり，再帰を用いて記述されることが多いことからも一般にタスク並列モデルと相性が良いことが知られている．また木構造における種々の操作は親から子を辿るプロセスが本質的に分割統治と同様のプロセスであるので，やはりタスク並列モデルと相性が良い．

以下のリスト 3.1 示すのは本研究で提案するアルゴリズムで頻繁に用いられるプログラムパターンを，タスク並列言語である Cilk [14] を用いて記述したものである．

リスト 3.1. Octree Traverse

```
1  Cilk void
2  OctreeOperation( Octant* o )
3  {
4     int i;
5
6     if( is_not_leaf( o ) ) {
7         for(i=0;i<8;i++) {
8             spawn OctreeOperation( o->child[i] );
9         }
10        sync;
11    }
12 }
13
```

上記コードで用いられている spawn はタスク呼び出しを意味し，sync はタスクの終了の待ち合わせを意味する Cilk のキーワードである．即ち上記コードでは 8 つの子ノードに対する処理がタ

スクとして呼び出され，並列に処理されることになる．これらのキーワードは OpenMP [6] における `#pragma omp task` と `#pragma omp task wait`，Intel TBB [35] における `task_group::run` と `task_group::wait` と同等のものである．これらのタスク並列処理系では一般に OS の提供するスレッド機能を用いる場合と比べて極めて高速に動作する．

上記コードからわかるとおり，分割統治法を用いた octree に対する処理は，タスク並列モデルを用いることで逐次プログラムとほとんど差異の無い見通しの良い並列化が可能である．

3.3　ワークスティーリング

タスク並列処理系では一般にプログラムの実行が開始された時点では 1 つの実行プロセスが 1 つのタスクを実行している状態である．新たにタスクが生成されたときにワークスティーリング [13, 26] と呼ばれる機構によって別の実行プロセスにタスクの実行が移り，並列実行およびその負荷分散が行われる．ワークスティーリングの手法は様々なものがあるが，Tree-based AMR では主要な操作が各リーフノードでの隣接ノードとの処理であるので，メモリのローカリティの観点からできるだけ物理的に近いノードを同じ実行プロセスに担当することが効率的な計算につながると考えられる．即ち分割統治法によって分割され再帰的に生成されるタスク群の中で，できるだけ大きな粒度で作られたタスク，つまり octree の根に近いところで生成されたタスク，がスティールすることが望ましいことがわかる．

これを実現するために効率的と思われるワークスティーリングの手法としては，以下の 3 つの方針を用いることが考えられる．1 つ目は *work first* 実行である．*Work first* 実行とは，タスクが生成された際に直ちに生成されたタスクの実行を開始する手法であり，逐次実行と同様の順番でタスクが実行に生成される．2 つ目は *LIFO* 実行であり，これはあるタスクが終了した時にその親タスクが他のプロセスにスティールされていない場合にその親の実行に戻るという方針である．3 つ目は *FIFO* スティールである．これはワークスティーリングの際に，スティール対象となったプロセスのうちで最も古いタスク，即ち木構造に沿ったタスク生成を行う場合には最も根に近いタスクをスティールする方針である．

これらの方針を用いることで，木構造においてできるだけ大きな粒度での負荷分散を行うことが可能になる．現在これらの方針を使用できるタスク並列処理系としては，Cilk と MassiveThreads [28] が挙げられる．

第4章　関連研究

本章では関連研究について述べる．まず 4.2.3 章で本研究の提案アルゴリズムのベースとなっている BBP アルゴリズムについて述べる．その後 4.1 章と 4.2 章で近年の研究で用いられている Tree-based AMR におけるデータ構造とバランスアルゴリズムについて述べる．

4.1　データ構造

Tree-based AMR における処理で最もコストの高い処理はバランスアルゴリズムを初めとする，全リーフノードとその隣接ノードとの間で行う処理である．既存の多くの Tree-based AMR 実装ではこの種の操作は，各リーフノードに対する処理の中で隣接ノードを探索することによって実現されている．一様な格子を用いた手法では，隣接ノードへのアクセスはインデックスを指定した配列へのアクセスであるので自明に $O(1)$ で行うことができる．一方で Tree-based AMR の場合にはデータが木構造状であるため，そうはいかない．最もシンプルな木構造はポインタベースのものであるが，この場合隣接ノードへのアクセスは，位置情報を保持しながら木を登り適切な位置から降っていく方法か，ないし位置情報をもとに根から探索を行う方法で行われる．これらの方法は平均で $O(\log(n))$ のコストを要し，全体では $O(n \log(n))$ を必要とする．そのため多くの既存システムでは，この隣接探索コストを削減するためのデータ構造の工夫が数多くなされてきた．

一方で，従来の並列 Tree-based AMR アプリケーションでは MPI を初めとする SPMD モデルでの並列化が多く試みられてきたので，負荷分散には静的な領域の分割が用いられてきた．Tree-based AMR では各ノードにおける計算コストがほぼ一定であるので，均等な負荷分散にはできるだけ全計算プロセスが同一なノード数を担当するために領域を分割する必要があるが，octree メッシュは木構造であるために均等な負荷分散が簡単ではない．このため，負荷分散のコストを削減するためのデータ構造の工夫もこれまで数多く行われている．

以下ではこれらの議論を踏まえて，現在実際に多くの成果を上げている研究で用いられている 2 つのデータ構造を紹介する．

4.1.1　Linear Octree

Linear Octree は Tree-based AMR では非常によく用いられる octree の表現方法である［7–9, 20, 36, 39, 41, 42］．Linear Octree では従来のポインタベースの木構造を用いず，ノードの位置及び細分化レベルを用いて存在しうるノード全てに一意な一次元の ID を割り当てることのできる機構を用意する．この仕組みを利用することによって全ノードをシリアライズすることが可能になり，データ構造としては一次元の線形リストあるいは配列とすることができる．従って単純に均等なノード数となるようリストや配列を分割することによって用意に均等な負荷分散を実現すること

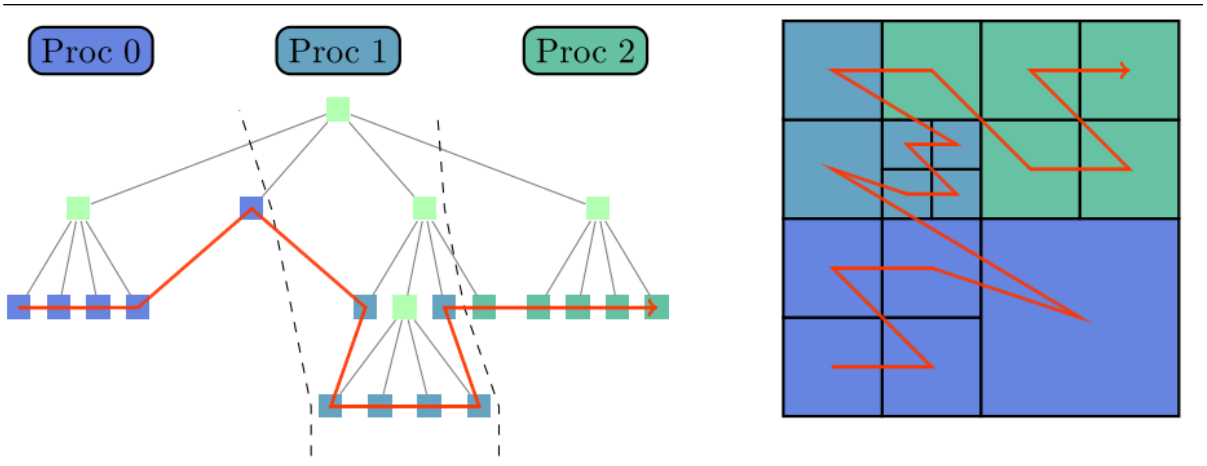


図 4.1. Burstedde らによる Linear Octree の図 [8]. 木構造メッシュのリーフノードをシリアライズすることによって，均等なノード数をもつような領域分割を用意に行うことができる．

ができることが最大の利点である．このとき，既存のアプリケーションではリーフノードの情報のみを格納しその他の情報は用いないことが多い．

ID の機構としては Space Filling Curve (SFC) がよく用いられる．中でも Morton Order (Z-curve) もしくは Hilbert Curve が殆どのアプリケーションで用いられている．このとき，SFC ではできるだけ局所性を保つような順番，即ちできるだけ物理的に近くのノードが ID でも近くに配置されるように順序付けがなされているので，領域分割に際してもある程度高いローカリティを持てるようになっている．これらの ID は，細分化レベルの最大値を L とすると位置情報を表す 次元数 $\times L$ bit と細分化レベルを表す $ln(L+1)$ bit の和のサイズの ID を用いて任意のノードを表現することができる．図 4.1 は Morton Order を用いた Linear Octree 及びそれを利用した領域分割の様子を示している．

Linear Octree を用いて隣接探索を行う場合，隣接セルの ID は $O(1)$ でわかるものの[1]，実際に隣接セルにアクセスするためには線形探索を必要とするためやはり $O(\log(n))$ のコストを必要とする．そのため，探索コストを削減するための方策としては別のデータ構造が設けられることが多い．たとえば，Burstedde らによる Tree-based AMR 用ライブラリ *p4est* [7–9, 20] では ID に対するメモリ位置をアプリケーションレベルでキャッシュすることによってコスト削減を行なっている．

4.1.2　FTT

Fully Threaded Tree (FTT) は，隣接探索コストを削減することを主眼に Khokhlov によって開発された octree データ構造であり [23]，近年では gerris [34], RAMSES [16, 40] などで用いられている．このために FTT では，シンプルなポインタベースの木構造で持たれる親から子へのポインタ以外に，子から親へのポインタと隣接セルへのポインタを保持する．このように，隣接セル情

[1] 隣接セルの細分化レベルはわからないので，性格には隣接セルである可能性のある ID が複数個存在する．しかし $O(1)$ であることに変わりはなく，SFC の性質上必ず近くに（親と子のセル ID は連続する）存在するので複数個存在することが後の探索にも本質的に影響を与えることはない

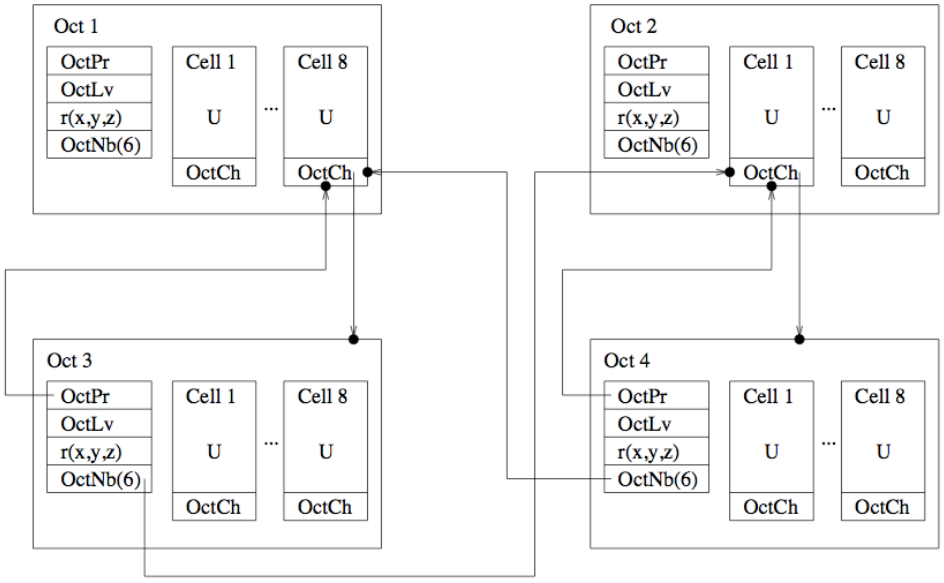


図 4.2. Khokhlov らによる Fully Threaded Tree (FTT) の概念図．図中の Cell 1, 2 等はセルの実体であり，Oct 1,2,3,4 は親子関係，兄弟関係，隣接関係等を管理している．図中の矢印はポインタを表している．矢印からわかるように，Oct 1 は Oct 3 の親であり，Oct 2 は Oct 4 の親である．また Oct3 と Oct4 は隣接しており，各々の親を介して参照を行なっている．

報を保持することによって FTT 作成後の隣接参照を $O(1)$ に削減することに成功している．細分化によって新たなノードが作られた際にも，Tree-based AMR ではあるノードの隣接セルは必ず兄弟ノードか親の隣接セル，ないし親の隣接セルの子か孫になるので，親の情報を用いて低オーバヘッドに隣接を探索してノードを追加することが可能である．また，FTT では隣接セルへの保持の仕方が工夫されており，隣接セルが自信の兄弟セルでない，即ち親の隣接セルもしくはその子孫ある，場合には「親の隣接セル」へのポインタを保持する．このように兄弟以外の隣接参照に必ず親を経由することによって，どのノードが削除された場合にも隣接するノードの情報を変更する必要がなくなる．即ちノードの増減に関して独立に対応できるという利点を持ち，これは並列処理を行う際に非常に有用である．図 4.2 は Khokhlov らによる FTT の概念図である．

一方で Linear Octree とは異なり，ポインタベースの木構造を用いているので領域分割を行うのは簡単ではない，前述の gerris [34] では，領域分割のためにグラフ分割を用いている [1]．しかしながら，最適なグラフ分割は NP 困難に属する問題であるので，いくつかのヒューリスティックを用いてある程度の粒度での負荷分散を行なっている．

4.2　並列バランス処理

前章で述べたとおりバランス処理は Tree-based AMR で最もコストの高い処理の 1 つであり，この部分を高速かつスケーラブルに行うアルゴリズムは非常に意味が大きい．2.4 において，ナイーブなアルゴリズムでは複数回の反復的な施行を必要とし，逐次プログラムにおいては Prioritized

Ripple Propagation (PRP) アルゴリズムがよく使われることは既に述べた．本章では，近年用いられる並列アルゴリズムについて解説する．

4.2.1 Parallel PRP アルゴリズム

PRP アルゴリズムの並列バージョンである Parallel PRP アルゴリズムは Tu らによって 2005 年に提案され［41］，これまで多くのアプリケーションで使われてきた［8, 41］．しかしながら PRP アルゴリズムでは細分化レベルの高いノードから順に処理を行なっていくので，SPMD モデルによる並列プログラムではレベル毎に同期を必要とする．特に分散環境における並列計算ではこの同期には全体通信を行う必要があるが，全体通信は非常にコストが高く多くの場合スケーラビリティを阻害してしまう．また負荷分散においては一般に各プロセスのノード数を均一化することを目指すが，これは各レベルにおけるノード数を均一化するわけではないのでレベルごとに負荷分散が偏ってしまう問題が生じる．このような理由から，特に大規模計算ではあまり良い性能を示せていない．

4.2.2 One pass アルゴリズム

近年研究が進められている大規模計算向けの Tree-based AMR の多くでは，*one pass* アルゴリズムと呼ばれる，レベル毎の同期を必要とせず一度の通信フェーズのみでバランス処理を行うことのできる手法がベースとして採用されている［2, 7, 9, 20, 36, 39］．One pass アルゴリズムは Sundar らによって 2008 年に提案された手法であり，*Insulation Layer* という概念を導入して「各実行プロセスの担当する領域内のノードを細分化する可能性のある他プロセスのノード」を算出することによって，1 回のみの通信フェーズでのバランス操作を実現している．Insulation layer は octree メッシュ中の任意のノードに対して定義される領域で，あるノード o の細分化を引き起こす可能性のあるノードは全て o の insulation layer$I(o)$ に含まれるという特徴を持つ．言い換えると $I(o)$ の外部に存在するノードがどれだけ細分化されても，それによって o の細分化が起こることは無い，ということが言える．

One pass アルゴリズムでは実際の隣接ノードだけでなく，他プロセスの持つ自プロセスのノードを細分化する可能性のあるノードも通信するために parallel PRP アルゴリズムに比べて通信量は増えるものの，レベルごとの同期に代表される全体通信の回数を劇的に削減したことによって性能上の大きな向上を得ることに成功している．近年では Bengerth らや Isaac らによって one pass アルゴリズムにおける通信されるノード数や通信の最適化などが行わるなどの改善が続けられている．

4.2.3 Balance By Parts アルゴリズム

Balance By Parts (BBP) アルゴリズムは Tu らによって 2004 年に提案された手法で，分割統治法を用いた octree の 2:1 バランスアルゴリズムである［42］．本研究で提案するバランスアルゴリズムはこの BBP アルゴリズムに基づいたものである．BBP アルゴリズムは以下のように動作する．

1. 木構造に沿って領域を部分領域（木構造の子ノード）に分割する

2. 各部分領域内部でバランス処理を行う
3. （内部的にバランスされた）各部分領域間でバランス処理を行う

例えば深さが 2 の子ノード（部分領域）まで分割されるとすると，全計算領域（木構造中の根ノード）に対するバランス処理は $8^2 = 64$ 個の部分領域（子ノード）に対する処理に分割され，まずそれぞれの子ノードで内部的にバランス処理が行われる．このとき，各子ノード内部でのバランスは 2.4 で述べた PRP アルゴリズムによって行われる．その後，内部的にバランスされた各子ノード間をバランスすることによって全体としてバランス条件を満たす，という流れになる．子ノード間のバランス処理は以下のように行われる: 1. 面で接する子ノード同士でバランス処理を行う，2. 辺で接する 2 つの子ノード同士でバランス処理行う，3. 頂点で接する 2 つの子ノード同士でバランス処理を行う．このとき 2 つの子ノードのバランス処理には，各々のノードの子孫のうち実際に接している，つまり境界と接するノードのみを考えればよいことが Tu らによって示されている．

BBP アルゴリズムが導入されている Tu らの研究では，格子全体がメモリに乗らないような巨大なサイズを想定していた．そのため，まずメモリに収まる大きさの子ノードまで分割を行い，その後に各子ノードの境界上のノードのみを取り出してバランスするという方針で設計されていた．我々の知る限りでは，これ以降に行われた研究では BBP アルゴリズムは用いられていない．これは，多くの Tree-based AMR の研究では扱う計算ノード数を増やし，ディスク上のデータは考慮されなくなったためだと考えられる．

4.3　Langer らによる分散オブジェクトを用いた手法

Langer らは並列オブジェクトをサポートする並列言語 charm++［22］を用いて，前章で述べた手法とは異なるアプローチで Tree-based AMR を記述している［24］．SPMD モデルに基づく多くのアルゴリズムで計算ノードに対して処理を記述するのに対し，並列オブジェクトを用いる charm++ では木構造中のノードに対する記述を行う．通信にはオブジェクト間のメッセージを用いて行っている．Langer らは，前章で述べたような既存手法では高並列環境になるにつれ全体通信がボトルネックとなることを指摘し，並列オブジェクトを用いて全体通信を一度しか必要としないアルゴリズムを提案している．

Langer らの手法では，octree 中の各ノードが並列オブジェクトとなる．従ってバランス処理についても各ノードで非同期に行われる．図 4.3 に示すように，あるノードが細分化を行う際にはそのノードは隣接ノードにオブジェクト間のメッセージを用いて教える．このようにバランス処理は全て非同期に行われるが，非同期であるために各オブジェクトがバランス処理の終了を知る方法が必要でありそのために終了検知アルゴリズムを用いている．彼らは実際に最大で 32K プロセッサを用いた Tree-based AMR 計算を行なっており，既存手法よりも少ないものの，やはり並列度が高まるに連れて全体通信の占める割合が高くなることを報告している．

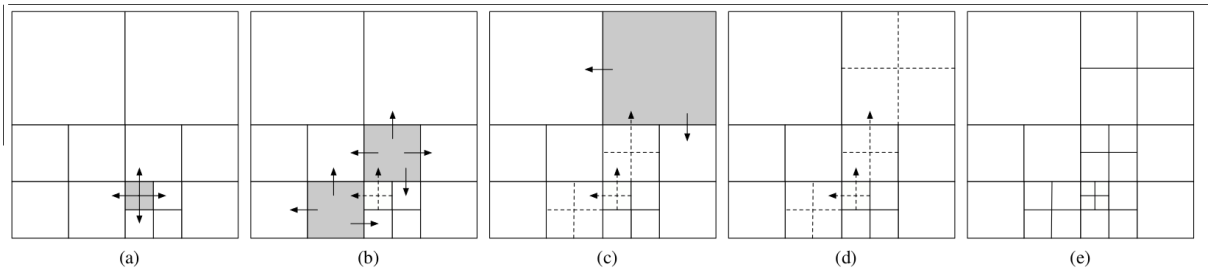


図 4.3. Langer らによる並列オブジェクトを用いたバランスオブジェクト．図中の各部分領域，すなわち octree 中の各ノードが並列オブジェクトである．あるノードが細分化する場合，隣接ノードにその旨を伝える．各ノードにおける処理は完全に非同期に起こる．しかし，非同期であるため終了検知アルゴリズムを必要とする

第5章　分割統治法を用いたバランスアルゴリズム

本章では，本研究で提案する分割統治法を用いた octree をバランスするアルゴリズムについて述べる．バランス処理の入力は 2:1 バランス条件が満たされていない octree であり，出力は 2:1 バランス条件が満たされた octree である．

2 章で述べたように，octree のバランス処理の要点は全リーフ octant がその隣接セルとの細分化レベルを比較するという処理である．従って本章ではバランスアルゴリズムとして説明するが，Tree-based AMR 中の操作で全リーフノードが隣接ノードとの間で何らかの処理を行うような操作であれば，同様の方法で適用可能である．

なお，アルゴリズム中にある `spawn` と `sync` はそれぞれ，3 章で説明したタスク並列モデルにおけるタスクの生成と生成されたタスク群の完了のための同期を意味するキーワードである．

本章の構成は以下である．まず 5.1 章でアルゴリズム全体を概観し，5.2 章と 5.3 章で要素アルゴリズムを説明する．その後，5.5 章で提案手法の正しさについて述べ，さらに，5.6 章で提案アルゴリズムの時間計算量が $O(n)$ であることについて証明を行う．最後に，5.7 章で提案アルゴリズムの考察を行う．

5.1　アルゴリズムの概観

提案アルゴリズムでは octree のバランス処理を分割統治法を用いて行う．即ちある領域をバランスするという処理を，1. 均等に分割された部分領域をバランスする，2. 内部的にバランスされている部分領域間をバランスする，という手続きに沿って行う．ここでいう部分領域というのは木構造中における子ノードであり，木構造に沿って再帰的に処理を行うことを意味する．即ち，ある octant をバランスするという処理を root octant から leaf octant まで再帰呼び出しによって行うということである．Algorithm 1 でこの操作を具体的に記述している．

Algorithm 1 `BalanceOctant`

Input: An octant o

```
1: if o is a leaf then
2:     return
3: end if
4: for each o's child c do
5:     spawn BalanceOctant(c)
6: end for
7: sync
8: BalanceChildren(o)
```

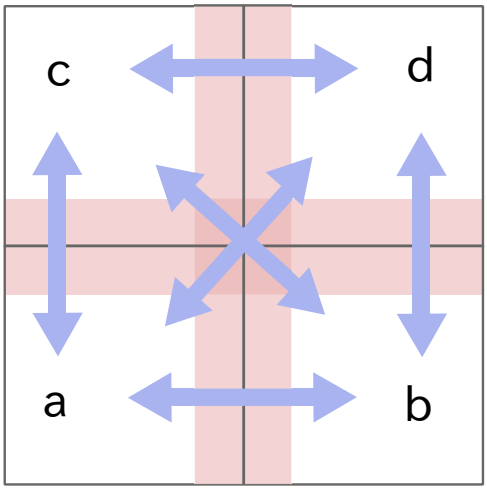


図 5.1. a，b，c，d はそれぞれ 4 分木中の，既に内部がバランス済みのノードである．これら 4 つのノードの親ノードにおいて子ノード間のバランス処理を行う（BalanceChildren）．この処理は図の矢印で示される 6 組のノード同士のバランス処理に分割され，その際にバランス条件を比較する必要となるのは各ノードのうち隣接する辺ないし，角に接する子孫ノードのみである．これらのノードが図中の色つきの部分で模式的に示されている．

BalanceOctant 関数において，octant o はまず 8 つの子ノードに対して同じ関数を再帰的に呼び出す．この処理によって，各子ノードが内部的にバランスされていることが保証される．その後，BalanceChildren 関数によって各子ノード間におけるバランス処理を行い，octant o がバランスされていることを保証する．子ノード間をバランスする BalanceChildren 関数については次章で述べる．

5.2　子ノード間のバランスアルゴリズム

ある octant に対して，各子ノードが内部的にバランス条件を満たしているときに，子ノード間のバランス処理を行うことを考える．提案手法では，この処理を全ての隣接する子ノード間のバランス処理に分割する．隣接する子ノードの組の数は，3 次元のとき $\binom{8}{2} = 28$ で，2 次元のとき $\binom{4}{2} = 6$ であり，この数だけ処理が行われることになる．隣接する子ノード同士は面，辺ないし角を共有している．従って提案手法ではこれらの子ノード間の処理を，どのように接しているか，でグループ分けを行い，そのグループごとに処理を行う．即ち，面を共有しているのか，辺を共有しているのか，点を共有しているのかの 3 グループ及び隣接している方向でグループ分けを行う．図 5.1 は 2 次元の場合を表しており，6 つの矢印で示されているのが隣接するノードの組である．この図で水平方向に x 軸，垂直方向に y 軸を仮定すると，x 軸と並行な辺で接しているのは $\{a, c\}$, $\{b, d\}$ の 2 組であり，y 軸と平行な辺で接しているのは $\{a, b\}$, $\{c, d\}$ の 2 組，頂点で接しているのは $\{a, d\}$,$\{b, c\}$ の 2 組である．同様に 3 次元の場合には，xy 平面，yz 平面，zx 平面と平行な面で接するノードの組がそれぞれ 4 組ずつ，x 軸，y 軸，z 軸と平行な辺で接するノードの組がそれぞれ 4 組みずつ，頂点で接するノードの組が 4 組の計 26 組である．

Algorithm 2 は 2 次元の場合を記述しており，3 次元の場合も同様に拡張可能である．アルゴリズム中の `BalanceEdge` 及び `BalanceCorner` はそれぞれ，辺を共有するノード同士のバランス処理と頂点を共有するノード同士のバランス処理を示している．

Algorithm 2 `BalanceChildren`

Input: An octant o

```
1: for each o's children a and b that neighbors on an edge parallel to x-axis do
2:     spawn BalanceEdge(a,b)
3: end for
4: sync
5: for each o's children a and b that neighbors on an edge parallel to y-axis do
6:     spawn BalanceEdge(a,b)
7: end for
8: sync
9: for each o's children a and b that neighbors on a corner do
10:    spawn BalanceCorner(a,b)
11: end for
12: sync
```

これらのグループ分けは本質的に必要なものではないが，並列処理を行う際には非常に有用である．なぜならば，グループの異なる子ノード同士の処理，例えば図 5.1 における $\{a, b\}$ と $\{a, c\}$ の組の処理，を同時に行ってしまうと，2 つのプロセスが同時に 1 つのノードの情報を書き換える可能性があるために競合してしまうからである．このためにロックのような排他制御を行うことは，一般に並列性能を著しく落としてしまう原因となりうる．

従って提案手法ではこのグループ分けに沿って処理を並列化し，グループの処理間では同期を行うことによってこの種の競合状態を回避している．この同期処理は，Algorithm 2 中では `sync` キーワードによって表現されている．なお，次章で述べるように `BalanceEdge`，`BalanceCorner` においても再帰呼び出しを用いたタスク生成が行われるため，並列実行可能なタスクの数が制限されることは無い．

5.3　隣接する 2 つの octant をバランスするアルゴリズム

本章で考えるのは，Algorithm 2 中で呼ばれる `BalanceEdge`, `BalanceCorner` の両者，即ち互いに内部的にバランスされた 2 つの隣接するノード間をバランスするアルゴリズムである．

このとき実際にバランス条件を比較する必要のある octant は，両ノードの境界上にあるものだけである．即ち，両ノードの子孫ノードのうち接する面ないし辺，頂点に接しているものだけである．図 5.1 ではこれらの octant の存在する領域が模擬的に赤く示されている．境界上に存在する octant のみを考えれば良い理由とその証明については 5.5 で述べる．

提案手法ではこの処理についても再帰的に行う．つまり，隣接する 2 つのノードのバランス処理を境界上に存在する子ノード間の処理に分割する．2 次元の場合，子ノードが辺で接する様子

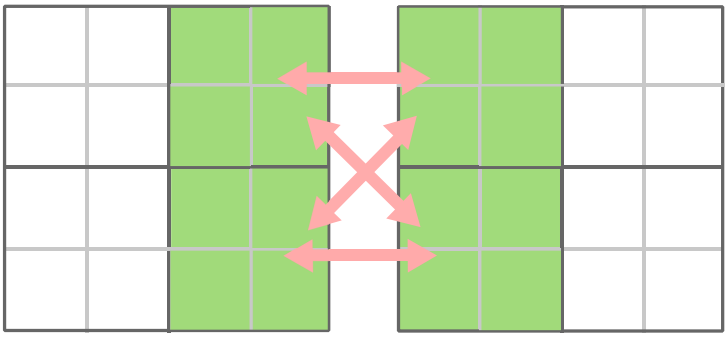


図 5.2. 辺で接する 2 つのノードのバランス処理は，図中の色のついた領域で示されている境界上の子孫ノードが対象となる．この処理は図中の矢印で示される子ノードの組同士でのバランス処理に細分化され，再帰関数として呼ばれる

は 図 5.2 のようになる．図中で色の付いている領域が境界上の子ノードを表しており，このとき分割される子ノードの組が矢印で表現されている．このように 2 次元での辺で接するノード間のバランス処理（BalanceEdge）は，2 組の辺で接する子ノード間のバランス処理（BalanceEdge）と 2 組の頂点で接する子ノード同士のバランス処理（BalanceCorner）に分割され，それぞれが再帰的に実行される．Algorithm 3 はこの処理を記述している．Algorithm 4 にあるように，頂点で接する場合も同様に再帰を行うが，この場合は境界上のノードは 1 組のみであるため，再帰呼び出しも 1 つである．3 次元の場合も同様に考えることができ，面で接するノード間のバランス操作（BalanceFace）は図 5.3 からわかる通り，4 組の子ノード同士の BalanceFace，4 組の BalanceEdge が 2 種類の計 8 組，そして 4 組の BalanceCorner を再帰的に呼び出すことによって実現される．

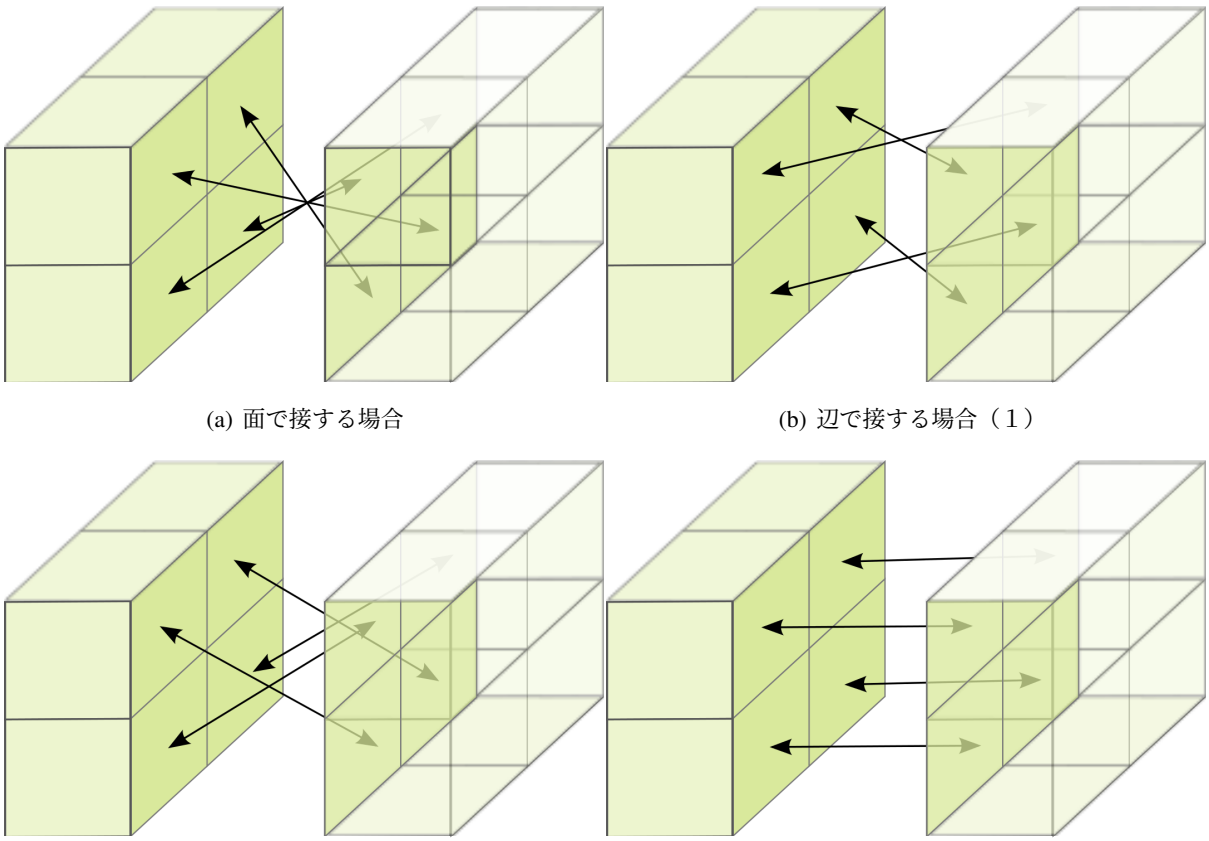
また BalanceEdge における sync キーワードからわかるとおり，BalanceCorner のときと同様にグループ分けを行って競合を回避している．

BalanceEdge, BalanceCorner の両処理で初めに呼ばれている処理，CheckAndRefine2Octants では，バランス条件が満たされていない場合に粗いノードを細分化する機能と，更なる再帰が必要かを判断する 2 つの機能をつ．これについては次章で詳述する．

5.4　CheckAndRefine2Octants

本説で説明するのは，前章で説明した BalanceEdge 及び BalanceCorner の両関数の初めで呼ばれている関数，CheckAndRefine2Octants についてである．具体的なアルゴリズムについては Algorithm 5 に記述している．この処理は 2 つの機能を持つ．即ち，2 つのノードを比べ必要なら片方を細分化する（バランスする）機能と更に処理を分割して再帰する必要があるかどうかを判断する機能である．

一つ目の目的については Algorithm 5 中の 3〜10 行目，11〜18 行目が相当し，2 つの隣接するノード間で 2:1 バランス条件が満たされていないことがわかれば，粗いノードを 1 レベル細分化する．この細分化処理は，Algorithm 5 中では MakeChildren として表記されている．具体的には以下のように動作する．



(a) 面で接する場合　(b) 辺で接する場合（1）

(c) 辺で接する場合（2）　(d) 頂点で接する場合

図 5.3. 面で接する 2 つのノードのバランス処理は，子ノード同士のバランス処理に分割される．これらの処理が行われる子ノードの組は上図のように接し方によって 4 つのグループに分けられ，グループ毎に同期的に処理が進む．

Algorithm 3 `BalanceEdge`

Input: Neighboring octants *a* and *b*

```
1:  if CheckAndRefine2Octants(a,b) is true then
2:      for each a's child aa and b's child bb that neighbors on an edge do
3:          spawn BalanceEdge(aa,bb)
4:      end for
5:      sync
6:      for each a's child aa and b's child bb that neighbors on a corner do
7:          spawn BalanceCorner(aa,bb)
8:      end for
9:      sync
10: else
11:     return
12: end if
```

Algorithm 4 `BalanceCorner`

Input: Neighboring octants *a* and *b*

```
1: if CheckAndRefine2Octants(a,b) is true then
2:    for each a's child aa and b's child bb that neighbors on a corner do
3:       spawn BalanceCorner(aa,bb)
4:    end for
5:    sync
6: else
7:    return
8: end if
```

まず，2 つのノードどちらもリーフノードであるならばバランス条件は満たされているし，どちらもリーフノードでは無い場合には子ノードを見なければわからないので，これらの場合には細分化は起きない．次に 1 つのノード *a* がリーフノードでもう 1 つのノード *b* がリーフノードで無い場合には *b* の子ノードのうち *a* に接しているものについて調べる．即ち，*a* に接する *b* の子ノードがリーフノードであるならば（*b* は高々1 レベルしか *a* より細かくないならば）バランス条件は満たされているので細分化は行われない．そうでない場合，即ち *a* に接する *b* の子ノードがリーフノードでない（*b* が *a* より 2 レベル以上細分化されている）場合，バランス条件が満たされていないので *a* を細分化する．

二つ目の目的は Algorithm 5 中の 2,7,10,15,18,20 行目に現れる `return` で返される真偽値がそうである．`true` であれば再帰を必要とし，`false` であれば必要としない，即ち処理を終了して良いことを意味する．具体的には，両者がリーフノードである場合には `false` であり，どちらもリーフノードでない場合には `true` である．どちらか一方がリーフノードである場合には，粗いノードが細分化された時のみ `true` を返す．

5.5　提案アルゴリズム

本説では提案アルゴリズムが正しさについて述べる．提案手法は分割統治法に基づいて部分領域内でのバランス処理を行った後に部分領域間でのバランス処理を行う．この処理については Tu らによる *balance by parts* (BBP) アルゴリズムと同様であり，このアルゴリズムの正しさについては［42］で証明されている．この証明の直感的な説明は，*Internal Octant* と *Boundary Octant* の 2 つの概念を用いて行うことができる．Internal Octant とはあるノードの子孫ノードにおいて境界と接していない，内側に存在するノードのことを言い，Boundary Octant はその反対に境界と接するノードのことである．ここで重要な事実は図 5.4 からわかる通り，Boundary Octant は決して 隣接する Internal Octant より細かくならないということである．従ってあるノードが内部的にバランスされているとき，外部のノードがどれだけ細分化されようともその影響をうけて細分化されるのは Boundary Octants のみであり，Internal Octants が細分化されることはない．それゆえ本アルゴリズムで隣接ノード間をバランスする際に実際にバランス条件を比較する必要があるのは，境界上の子孫ノードのみで良いという事ができる．

Algorithm 5 `CheckAndRefine2Octants`

Input: Neighboring octants a and b

Output: *true* if they need more recursive calls, *false* otherwise

```
1: if a is a leaf and b is a leaf then
2:     return false
3: else if a is a leaf then
4:     for each b's child octant bb that is adjacent to a do
5:         if bb is not a leaf then
6:             MakeChildren(a)
7:             return true
8:         end if
9:     end for
10:    return false
11: else if b is a leaf then
12:    for each a's child octant aa that is adjacent to b do
13:        if aa is not a leaf then
14:            MakeChildren(b)
15:            return true
16:        end if
17:    end for
18:    return false
19: else
20:    return true
21: end if
```

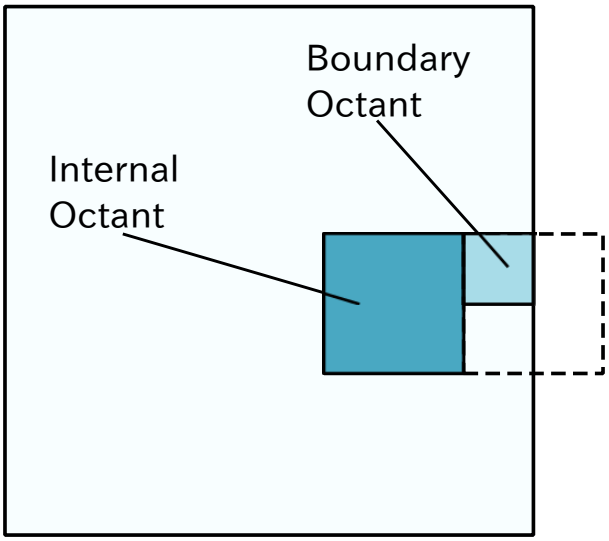


図 5.4. Internal Octants と Boundary Octants の概念図．Boundary Octants が隣接する Internal Octants より細くなることは起こりえない

5.6　提案アルゴリズムの計算量の証明

本章では，バランス処理が行われたあとのリーフセルを n としたとき，octree が過度に偏っていないという前提条件のもとで提案するバランスアルゴリズム，つまり `BalanceOctant` の時間計算量が $O(n)$ であることを示す．具体的には octree 中のノードの子ノードの数が L であるとき（即ち 2 次元では $L = 4$,3 次元では $L = 8$），以下を仮定する．

> あるノード内部に n 個のリーフノードが存在するとき，どの $L/2$ 個の子ノードの集合をとっても，その集合内部のリーフノードの合計が γn を超えないような定数 $\gamma < 1$ が存在する．

次の 3 つの定義を行う．

Definition 1
- $C(n,m)$ *:* バランス後の内部に存在するリーフセルの数がそれぞれ n 個以下，m 個以下であるような 2 つのセル a,b に対する `BalanceCorner(`a,b`)` の最悪計算量
- $E(n,m)$ *:* バランス後の内部に存在するリーフセルの数がそれぞれ n 個以下，m 個以下であるような 2 つのセル a,b に対する `BalanceEdge(`a,b`)` の最悪計算量
- $V(n)$ *:* バランス後の内部に存在するリーフセルの数が n 個以下であるようなセル a に対する `BalanceOctant(`a`)` の最悪計算量

次の三つの式を証明する．

Lemma 2　*1.*

$$C(n,m) \in O(\log(n+m))$$

2.

$$E(n,m) \in (n+m)^{\gamma}$$

3.

$$V(n) \in O(n)$$

以下で証明を行う．

1. $C(n,m) \in O(\log(n+m))$: Algorithm 4 から分かる通り，`BalanceCorner(`a`,` b`)` は 0 個ないし 1 個の再帰呼び出しを行う．前者のケースでは，計算コストは明らかに $O(1)$ である．後者のケースで，実行される再帰呼び出しを `BalanceCorner(`a_1`,` b_1`)` とする．このとき a_1 は a の子であり b_1 は b の子である．a_1 に含まれるリーフセルの数を αn，b_1 に含まれるリーフセルの数を βn とすると，以下が言える．

   $$C(n,m) \leq c + C(\alpha n, \beta m)$$

   ここで c は定数である．$C(\alpha n, \beta m)$ に対する再帰呼び出しの数が高々 $\log_{1/\gamma}(\alpha n + \beta m)$ であることを用いて，上の式から $C(n,m) \in O(\log(n+m))$ を容易に導くことが出来る．
   今回は，特に以下を示す．

   $$C(n,m) \quad \leq \quad c \log_{1/\gamma}(n+m)$$

   これは先程の式から，以下のように導出できる．

   $$\begin{aligned} C(n,m) &\leq c + C(\alpha n, \beta m) \\ &\leq c + c \log_{1/\gamma}(\alpha n + \beta m) \\ &\leq c + c \log_{1/\gamma}(\gamma(n+m)) \\ &\leq c + c \log_{1/\gamma} \gamma + c \log_{1/\gamma}(n+m) \\ &\leq c \log_{1/\gamma}(n+m). \quad \text{(Q.E.D)} \end{aligned}$$

2. $E(n,m) \in (n+m)^{\gamma}$: Algorithm 3 から分かるとおり，`BalanceEdge(`a`,` b`)` は 0 個ないし 4 個の再帰呼び出しを行う．前者のケースでは，計算コストは明らかに $O(1)$ である．後者のケースで，4 つの再帰呼び出しを以下のように定義する．

   - `BalanceEdge(`a_1`,` b_1`)`,
   - `BalanceEdge(`a_2`,` b_2`)`,
   - `BalanceCorner(`a_1`,` b_2`)`,
   - `BalanceCorner(`a_2`,` b_1`)`,

   ここで，a_1 と a_2 は a の子であり，b_1 と b_2 は b の子である．
   a_1, a_2, b_1, b_2 がそれぞれ $\alpha_1 n, \alpha_2 n, \beta_1 m, \beta_2 m$ 個のリーフセルを含むとすると，既に示した $C(n,m) \in O(\log(n+m))$ の式から， 定数 d を用いて以下が言える．

   $$E(n,m) \leq d \log(n+m) + E(\alpha_1 n, \beta_1 m) + E(\alpha_2 n, \beta_2 m)$$

   この式から以下を示す．

   $$E(n,m) \in O((n+m)^{\gamma})$$

特に以下を示す.

$$E(n,m) \leq A(n+m)^{\gamma},$$

ここで A は以下である.

$$A = \frac{d}{(1-2(\gamma/2)^{\gamma})\gamma e \ln 2}.$$

まず D を定義する.

$$D = \left\{ \left(\frac{\alpha_1 n + \beta_1 m}{n+m} \right)^{\gamma} + \left(\frac{\alpha_2 n + \beta_2 m}{n+m} \right)^{\gamma} \right\}$$

D を用いて

$$\begin{aligned} E(n,m) &\leq d\log(n+m) \\ &\quad + A(\alpha_1 n + \beta_1 m)^{\gamma} + A(\alpha_2 n + \beta_2 m)^{\gamma} \\ &\leq d\log(n+m) + AD(n+m)^{\gamma} \quad (*), \end{aligned}$$

ここで x と y を以下のようにする.

$$\begin{aligned} x &= \left(\frac{\alpha_1 n + \beta_1 m}{n+m} \right), \text{ and} \\ y &= \left(\frac{\alpha_2 n + \beta_2 m}{n+m} \right) \end{aligned}$$

$x + y \leq \gamma$ であるから,

$$\begin{aligned} D &= x^{\gamma} + y^{\gamma} \\ &\leq \left(\frac{\gamma}{2} \right)^{\gamma} + \left(\frac{\gamma}{2} \right)^{\gamma} \\ &\leq 2\left(\frac{\gamma}{2} \right)^{\gamma} \end{aligned}$$

従って,

$$\begin{aligned} (*) &\leq d\log(n+m) + A2\left(\frac{\gamma}{2} \right)^{\gamma} (n+m)^{\gamma} \\ &\leq \left(d\frac{\log(n+m)}{(n+m)^{\gamma}} + A2\left(\frac{\gamma}{2} \right)^{\gamma} \right)(n+m)^{\gamma}. \quad (*) \end{aligned}$$

この値は 1 より小さい定数である.
次に上式 1 つ目の括弧中の初項の上限を定める. $x = n+m$ とすると,

$$\frac{\log(n+m)}{(n+m)^{\gamma}} = \frac{\log x}{x^{\gamma}}$$

この式は $x = e^{1/\gamma}$ だるときに最大となりその値は以下である.

$$\frac{\log e^{1/\gamma}}{(e^{1/\gamma})^{\gamma}} = \frac{1}{\gamma e \ln 2}$$

従って,

$$\begin{aligned}(*) &\leq \left(\frac{d}{\gamma e \ln 2} + A2\left(\frac{\gamma}{2}\right)^{\gamma}\right)(n+m)^{\gamma} \\ &= \left(\frac{d}{\gamma e \ln 2} + \frac{d}{(1-2(\gamma/2)^{\gamma})\gamma e \ln 2} 2\left(\frac{\gamma}{2}\right)^{\gamma}\right)(n+m)^{\gamma} \\ &= \frac{d}{(1-2(\gamma/2)^{\gamma})\gamma e \ln 2}(n+m)^{\gamma} \quad \text{(Q.E.D)}\end{aligned}$$

3. $V(n) \in O(n)$: 既に示した 2 つのケースと同様に，`BalanceOctant(`a`)` は $O(1)$ のコストである 0 個の再帰呼び出しを行うか，以下のような再帰呼び出しを行う.
   - `BalanceFace(`a_i`,` b_i`)`, for $i = 1, 2, 3, 4$
   - Four `BalanceEdge`'s and two `BalanceCorner`'s

   下の呼び出しの計算コストは $O(n^{\gamma})$ である従って定数 v を用いて,

$$V(n) \leq vn^{\gamma} + \sum_{i=1,2,3,4} V(\alpha_i n)$$

.
以下で，$V(n) \in O(n)$ を示す.
$U(n)$ を以下のように定義する.

$$U(n) = \max_{x \leq n} \frac{V(x)}{x}$$

これを用いて,

$$\begin{aligned}\frac{V(n)}{n} &\leq vn^{\gamma-1} + \sum_{i=1}^{4} \frac{V(\alpha_i n)}{n} \\ &\leq vn^{\gamma-1} + \sum_{i=1}^{4} \alpha_i \frac{V(\alpha_i n)}{\alpha_i n} \\ &\leq vn^{\gamma-1} + \sum_{i=1}^{4} \alpha_i U(\gamma n) \\ &\leq vn^{\gamma-1} + U(\gamma n)\end{aligned}$$

n は任意の数であるから,

$$U(n) \leq vn^{\gamma-1} + U(\gamma n).$$

これを再帰的に適用して,

$$\begin{aligned}U(n) &\leq vn^{\gamma-1} + U(\gamma n) \\ &\leq vn^{\gamma-1} + v(\gamma n)^{\gamma-1} + U(\gamma^2 n) \\ &\leq vn^{\gamma-1} + v(\gamma n)^{\gamma-1} + v(\gamma^2 n)^{\gamma-1} + U(\gamma^3 n) \\ &\leq vn^{\gamma-1} + v(\gamma n)^{\gamma-1} + \cdots + v(\gamma^{l-1} n)^{\gamma-1} + U(\gamma^l n) \\ &= vn^{\gamma-1}(1 + \delta + \cdots + \delta^{l-1}) + U(\gamma^l n) \\ &= vn^{\gamma-1}(1 + \delta + \cdots + \delta^{l-1}) + U(1) \quad (*)\end{aligned}$$

ここで，$\delta = \gamma^{\gamma-1}$ かつ $l = \left\lceil \log_{1/\gamma} n \right\rceil$ である．従って，

$$\begin{aligned}
(*) &= vn^{\gamma-1}\frac{\delta^l - 1}{\delta - 1} + U(1) \\
&= vn^{\gamma-1}\frac{\gamma^{(\gamma-1)l} - 1}{\delta - 1} + U(1) \\
&= vn^{\gamma-1}\frac{(1/n)^{\gamma-1} - 1}{\delta - 1} + U(1) \\
&\leq vn^{\gamma-1}\frac{(1/n)^{\gamma-1} - 1}{\delta - 1} + U(1) \\
&\leq \frac{v}{\delta - 1} + U(1),
\end{aligned}$$

この値は定数である．つまり，

$$\max_{x \leq n} \frac{V(n)}{n} \in O(1),$$

従って，

$$\frac{V(n)}{n} \in O(n).$$

5.7　考察

提案アルゴリズムは 4.2.3 章で述べたように，Tu らによる Balance By Parts アルゴリズム［42］を元にしている．BBP アルゴリズムと提案手法の最も大きな違いは，提案手法ではバランス処理全体を分割統治法で行なっている点と，再帰呼び出しを用いている点である．

BBP アルゴリズムではメモリに収まらないような巨大な octree を対象としていたため，バランス処理の際にメモリにサイズに収まるようにすることを目的としていた．全体 octree を分割して，各 suboctree 内部は PRP アルゴリズムでバランス処理を行い，内部的にバランスされた suboctree 間をバランスする際には，境界と接するノードだけを取り出してバランス処理を行う．これに対して提案手法では，バランス処理全体を再帰を用いた分割統治で行なっている．提案手法による利点としては以下のような特徴が挙げられる．

- 過度に偏っていないという前提のもと，$O(n)$ の計算コスト
- 直感的かつ容易に並列化が可能
- 全体同期を必要としない
- タスク並列処理系による自動的な負荷分散
- 最小限のシンプルなデータ構造

一方で提案手法はタスク並列処理系を用いており，このまま分散環境で効率的なプログラムを記述できるわけではない．分散環境で効率的に動作するタスク並列モデルに関しては様々な研究がなされているが，これまでのところは実用レベルに達していないため，本研究の提案アルゴリズムについても何らか改善を行う必要があると考えられる．我々のグループでは現在分散環境向けの軽量スレッド処理系 MassiveThreads/DM を開発中であり，こちらとあわせて改善を続けていく必要がある．

第6章　タスク並列Tree-based AMR

本章では今回用いる問題設定と，本研究で提案するタスク並列モデルを用いたTree-based AMRプログラムについて具体的に説明する

6.1　問題設定

Tree-based AMRは適応的なoctreeメッシュを用いる格子法一般に対する呼称であり，一口にTree-based AMRといっても，対象となる問題や用いる計算手法，設定などによって必要なデータ構造や行う計算の性質は大きく異なる．

以下で，本研究で用いた問題設定を述べる．

数値解析手法　本研究では有限差分法を用いる．有限差分法は微分方程式を解くための数値解析手法の1つであり，格子点間の変数値の差（**差分**）を用いて微分作用素を近似する．一様格子を用いる場合には特にステンシル計算と呼ばれるような形になる．他の手法としては有限要素法や有限体積法，境界要素法などが挙げられる．有限要素法では解析対象領域を小領域（要素）に分割し，各要素内で支配方程式に重み関数乗じて積分した弱形式で計算を行う．各要素は節点で隣接要素と繋がっており，節点に物理量を配置する．一般に巨大な行列計算に帰着される．有限体積法では各小領域内で物理量が保存されるように離散化を行い，節点ではなく要素内の計算点に物理量を配置する．

このように用いる手法によってデータ構造や計算の性質が大きく異なる．今回は比較的単純である有限差分法を用いるが，提案手法は原則的に他の手法にも適用可能である．

空間差分スキーム　本研究では2次中央差分ないし1次片側差分を用いる．この制限の意味するところは，あるノードにおける差分の計算には隣接ノードのみを用いるということである．2次片側差分や4次中央差分等の高次差分スキームを用いる場合には2つ隣のノードの値を差分計算で必要する．具体的に言い換えると，Φ_i の値を計算するのに Φ_{i-1}，Φ_{i+1} を用いるスキームに対応していて，Φ_{i-2}, Φ_{i+3} のような項を用いるスキームに対応していないということである．ただしこれは今回簡単のために設けた制約であって，例えばoctree中のリーフノードに1つの格子点を設けるのではなく，ある程度の大きさをもったメッシュ（例えば 8×8 の一様メッシュ）を対応させるなどすれば比較的容易に実現できる．

時間発展スキーム　本研究では陽解法を対象とする．陽解法では時刻 t の値を用いて $t + \Delta t$ の値を計算する．一方陰解法では $t + \Delta t$ の値を仮定して場の支配方程式に代入し，誤差が十分小さくなるまで反復計算を行う．これは巨大な行列式による連立一次方程式を解くと操作に帰着される．

一般に陽解法ではタイムステップを細かくとる必要があるが計算負荷が小さい．反対に陰解法ではタイムステップを長くとることができるが，一度の計算負荷が大きい．陰解法を用いた場合計算負荷・メモリ負荷が行列計算が大きくなってしまうため，今回は陽解法を用いている．

細分化・集約化の判断方法　細分化・集約化の判断には，Parthasarathy らによって提案されている隣接ノードとの物理量の差の統計量から判断する手法［33］を用いている．この方法では，全リーフノードにおける隣接ノードの物理量の差の最悪値と，差の平均と標準偏差を比較することによって判断を行う．即ち，平均を μ，標準偏差を δ，あるノードにおける隣接ノードとの物理量の差の最大値を val_{max} とすると以下である．

$val_{max} - \mu \geq K_{high} \cdot \delta$ **である場合**　細分化する

$val_{max} - \mu \leq K_{low} \cdot \delta$ **である場合**　集約化する

その他の場合　何もしない

K_{high}, K_{low} は予め定められた定数パラメータである．

このように細分化・集約化を行う前にまずこれらの統計量を収集・計算する必要があるため，実際の操作としてはまず統計量の収集処理があり，その後に実際の細分化・集約化の処理が行われる．

6.2　データ構造

今回のシミュレーションプログラムでは，以下のような octree データ構造を用いる．

リスト 6.1. データ構造

```
1 struct Octant {
2   int level;             // 細分化レベル
3   double val[2];         // 物理量．２つを使ってダブルバッファリングを行う
4   double highest_diff;   // 隣接セルとの物理利用の差の最悪値
5   struct Octant* child; // 子ノードへのポインタ．動的に確保される
6 };
```

この例は最もシンプルな場合を示している．当然，問題や離散化方法によって扱う物理量やその他の値は異なる．本提案手法のデータ構造として重要なのは，octree メッシュに関する情報が細分化レベルと子へのポインタという最低限のデータしか必要としないということである．

6.3　タスク並列モデルを用いた Tree-based AMR 中の操作

Tree-based AMR では基本的に全ての操作は octree メッシュに対して行われ，それらの操作は以下の 2 種類に分けることができる．

- 全てのリーフノードに対して何らかの処理を行う
- 全てのリーフノードに対して，隣接ノードとの間で何らかの処理を行う

以下でこれらの処理を実現する具体的なアルゴリズムを述べる．

6.3.1　全リーフノードに対する処理

以下のリスト 6.2 のように記述される．

リスト 6.2. Octree 上の全リーフノードに対する処理

```
1  Cilk void
2  OctreeOperation( Octant* o )
3  {
4     int i;
5     if( is_leaf( o ) ) {
6       LeafOperation(o);
7     } else if(o->level < CUTOFF_LEVEL) {
8         for(i=0;i<8;i++)
9             spawn OctreeOperation( o->child[i] );
10        sync;
11    } else {
12        for(i=0;i<8;i++)
13            OctreeOperation( o->child[i] );
14    }
15 }
```

コード中の関数 LeafOperation が，リーフノード即ちメッシュのセルに対しての処理を表している．CUTOFF_LEVEL は予め定められた，タスク生成のしきい値である．例えば CUTOFF_LEVEL が 3 であれば深さ 3 未満の浅いノードまではタスクとして並列に実行され，深さ 3 以上での処理は逐次実行される．この場合 $1 + 8 + 8^2 + 8^3$ 個のタスクが生成されることになる．

6.3.2　全リーフノードに対する隣接ノードアクセスを伴う処理

隣接アクセスを伴う処理は，バランス処理や計算部分で用いられる重要な操作である．5 章で説明したバランスアルゴリズムと同様のアルゴリズムでも行うことができるが，バランス処理のようなノード間の処理の依存関係が無いようであれば，以下のリスト 6.3 に示すように再帰する際に予め隣接ノードを与えてやることでさらに高速に処理することが可能である．

リスト 6.3. Octree 上の全リーフノードに対する隣接ノードへのアクセスを伴う処理

```
// 引数は処理対象ノードとその左右上下の隣接ノード
Cilk void
OctreeNeighborhoodOperation( Octant* o, Octant* left, Octant* right,
                                        Octant* up,   Octant* down)
{
   if( is_leaf( o ) ) {
     NeighborOperation(o,left); //隣接ノードとの処理
     NeighborOperation(o,right);
     NeighborOperation(o,up);
     NeighborOperation(o,down);
   } else if( o->level < CUTOFF_LEVEL){
     // 左上の子ノード o->child[LEFTUP] についての処理
     // 隣接ノードが子を持たない（1 レベル粗い）場合との場合分けが必要
     Octree* child_left  = left->is_leaf ? left : left->child[RIGHTUP];
     Octree* child_right = o->child[RIGHTUP];
     Octree* child_up    = up->is_leaf   ? up : up->child[LEFTDOWN];
     Octree* child_down  = o->child[LEFTDOWN];
     spawn OctreeNeighborhoodOperation( o->child[LEFTUP],
                                child_left, child_right, child_up, child_down);
     // 他の子ノードに対しても同様に再帰呼び出しを行う
     ...
     ...
     sync;
   } else {
     // 上と同様の処理を逐次に行う
     ...
     ...
   }
}

```

リスト 6.3 は簡単のために 2 次元での例を示しているが，同様に 3 次元の場合も記述可能である．

6.4　タスク並列 Tree-based AMR シミュレーションの流れ

6.4.1　計算の流れ

0. 初期メッシュの生成・初期化
1. 隣接セルとの値の差の平均・標準偏差を求める
2. メッシュの細分化・集約化
3. 2:1 バランス
4. 差分計算
5. 値の更新（時間を進める）
6. 外力の付加
7. 1 へ

初期化を行った後の 1〜6 の操作は，定められた回数反復されることになる．

以下でそれぞれのフェーズについて述べる．ただし，3 の 2:1 バランス処理については既に 5 章で述べたとおりなので省略する．

6.4.2　初期メッシュの生成・初期化

このフェーズはアプリケーションの実行時に一度だけ実行される初期化フェーズであり，行う処理は初期メッシュの生成と値の初期化である．まず初期メッシュの生成は以下のリスト 6.4 に示すされるように，簡単に行うことできる．

リスト 6.4. 初期メッシュの生成

```
1 void
2 InitialOctree(Octant* o)
3 {
4    if(o->level < INITIAL_LEVEL) {
5        MakeChildren(o);
6        for(int i=0;i<8;i++)
7            InitialOctree( o->child[i] );
8    }
9 }
```

コード中の INITIAL_LEVEL は予め決められた値で，初期メッシュとしてこのレベルまで展開された octree メッシュが用いられる．つまり，2 次元で初期レベルが 10 ならば，1024x1024 のメッシュが初期メッシュとなる．

なおコードでは逐次処理として記述されているが，初期メッシュサイズが大きい場合には再帰呼び出しに spawn キーワードを付け加えることによって簡単に並列化可能である．

メッシュに対する初期値のセットはリーフノードに対する処理であるので，リスト 3.1 の構造を用いて実現できる．

6.4.3　隣接セルとの値の差の平均・標準偏差を求める

6.1 章で述べた通り，細分化・集約化の判断には，リーフノードと隣接ノードの物理量の差を用いる．このために，まず全体の平均と標準偏差を求め，各リーフノードで最悪値を知る必要がある．平均 μ と標準偏差 δ は以下のように求めることができる．

$$
\begin{aligned}
\mu &= \frac{\sum_{i=0}^{n} x_i}{n} \\
\delta &= \sqrt{\frac{\sum_{i=0}^{n}(x_i-\mu)^2}{n-1}} \\
&= \sqrt{\frac{\sum_{i=0}^{n}(x_i^2)-n\mu^2}{n-1}}
\end{aligned}
$$

従ってこの操作では，全ての隣接するリーフノード間の物理量の差及び差の 2 乗を求めて集めることと，各リーフノードで最悪値を求めることが必要となる．この操作は,『全リーフノードに対する隣接ノードアクセスを伴う処理』，でありリスト 6.2 と同様の構造を用いて実現可能である．

6.4.4　メッシュの細分化・集約化

この操作ではメッシュの細分化ないし集約化を行う．判断を下すためのデータ，つまり全体の平均，標準偏差及び各ノードにおける隣接ノードとの差の最大値，は 6.4.3 章の操作で既に各ノードが保持している．従って，この操作は『全リーフノードに対する操作』であり，リスト 3.1 と同様の構造で実現することができる．ただし集約化が行われるのは同じ親ノードの子ノード全てで集約化の判断がなされた時のみであり，また一度の操作でリーフノードから二段階以上集約化されることは無い．具体的なアルゴリズムをリスト 6.5 に示した

リスト 6.5. octree の細分化・集約化

```
1  // 入力ノードが集約化の基準を満たす場合,(親に）真を返す
2  bool
3  RemeshOctree(Octant* o)
4  {
5     if(is_leaf(o)) {                // 子ノードの処理
6       if(need_to_refine(o)) {
7         make_children(o);           // 細分化する
8         return false;               // 当然集約化は起きない
9       }
10      else if(need_to_coarsen(o)) {
11        return true;                // 集約化の基準を満たすことを親に伝える
12      }
13      else {
14        return false;
15      }
16    }
17    else {                           // 親ノードの処理
18      coarsen_flag = true;
19      for(int i=0;i<8;i++)
20        coarsen_flag &= RemeshOctree( &(o->child[i]) );
21      if(coarsen_flag){              // 全ての子ノードが集約化の基準値を満たすなら
22        delete_children(o);          // 親が子ノードを消去する（集約化する）
23      }
24      return false;                  // 親ノードが集約化されることはない
25    }
26 }
```

6.4.5　差分計算と値の更新

このフェーズでは各リーフノードで，隣接ノードと自ノードの物理量を用いて微分作用素を計算する．一様格子を用いる場合にはこの操作はステンシル計算と呼ばれ，例えば二次元で中央差分を用いた計算なら

$$A_{i,j}^{n+1} = aA_{i,j}^{n} + bA_{i-1,j}^{n} + cA_{i+1,j}^{n} + dA_{i,j-1}^{n} + eA_{i,j+1}^{n}$$

というように表現される処理である．

Tree-based AMR では隣接セルの大きさが異なるためにこういったことはできない．従って図 6.1 に示すように，隣接ノードと同じレベルのノードの存在する計算点での値が補間・補外される．

また，更新される $A_{i,j}$ の値は隣接ノードの計算で使われるため，プログラム上では同じ変数に書きこむことはできない．そのため，章で述べたように変数を 2 つ用意しておき，イテレーショ

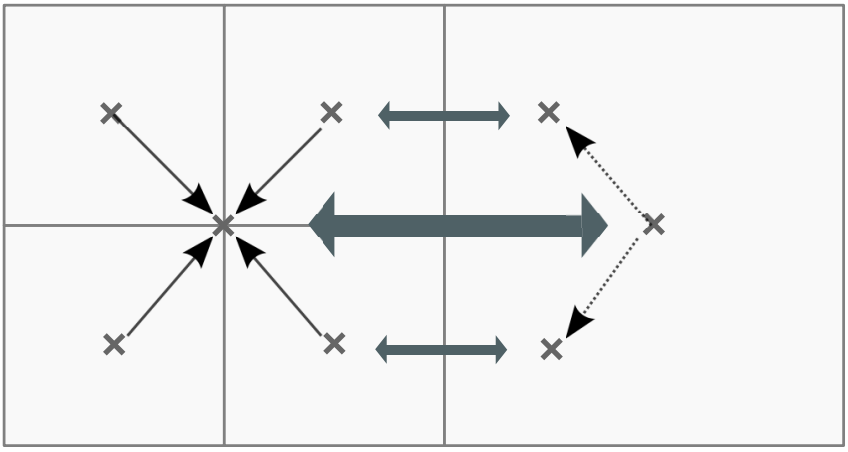


図 6.1. サイズの異なるノード間での差分計算．粗いノードには細かいノードを集約した値が加算される．細かいノードには粗いノードから補外された値が加算される

ン毎に交互に用いることによってこの問題を解決している．このような手法はダブルバッファリングと呼ばれ，よく用いられる方法である．

この操作は,『全リーフノードに対する隣接ノードアクセスを伴う処理』，でありリスト 6.2 と同様の構造を用いて実現可能である．

第7章　評価

本章では，本研究で提案するタスク並列を用いた Tree-based AMR の評価を行う．

7.1　実装

今回提案手法を C 言語とタスク並列処理系向け軽量スレッドライブラリ MassiveThreads［28, 29］を用いて実装した．

7.2　実験環境

評価実験には共有メモリ x86-64 コンピュータを用いた．詳細を表 7.1 に示す．またハードウェアの設定として以下の 3 つの機能をオフにしている

- Hardware Prefetch
- Adjacent Cache Line Prefetch
- DCU Streamer Prefetcher

これらの機能はいずれもキャッシュを強力にするための機能であるが，一般にシーケンシャルなメモリアクセスに対する最適化を提供するものであり，ランダムなメモリアクセスが多い場合にはかえって性能に悪影響を及ぼすことがある．事前に計測を行ったところ実際に並列性能を低下させることがわかったので，今回の評価実験ではこれらの機能は無効にしている．

7.3　バランスアルゴリズムの評価

7.3.1　問題設定

評価のための問題設定は Sundar ら［39］と同様の，N 体問題における Treecode［3］や高速多重極展開法［15］で用いられる 8 分木生成モデルを用いた．即ち空間中に粒子が存在しており，全てのセルが高々1 つの粒子しか含まなくなるまで，空間を再帰的に 8 分割していくような処理によっ

アーキテクチャ	モデル	周波数	コア数/ソケット	ソケット数	コア数	L1	L2	L3
Sandy Bridge	E5-2660	2.2GHz	4	4	16	32K	256K	20M

表 7.1. 評価実験で用いた機器のスペック

.

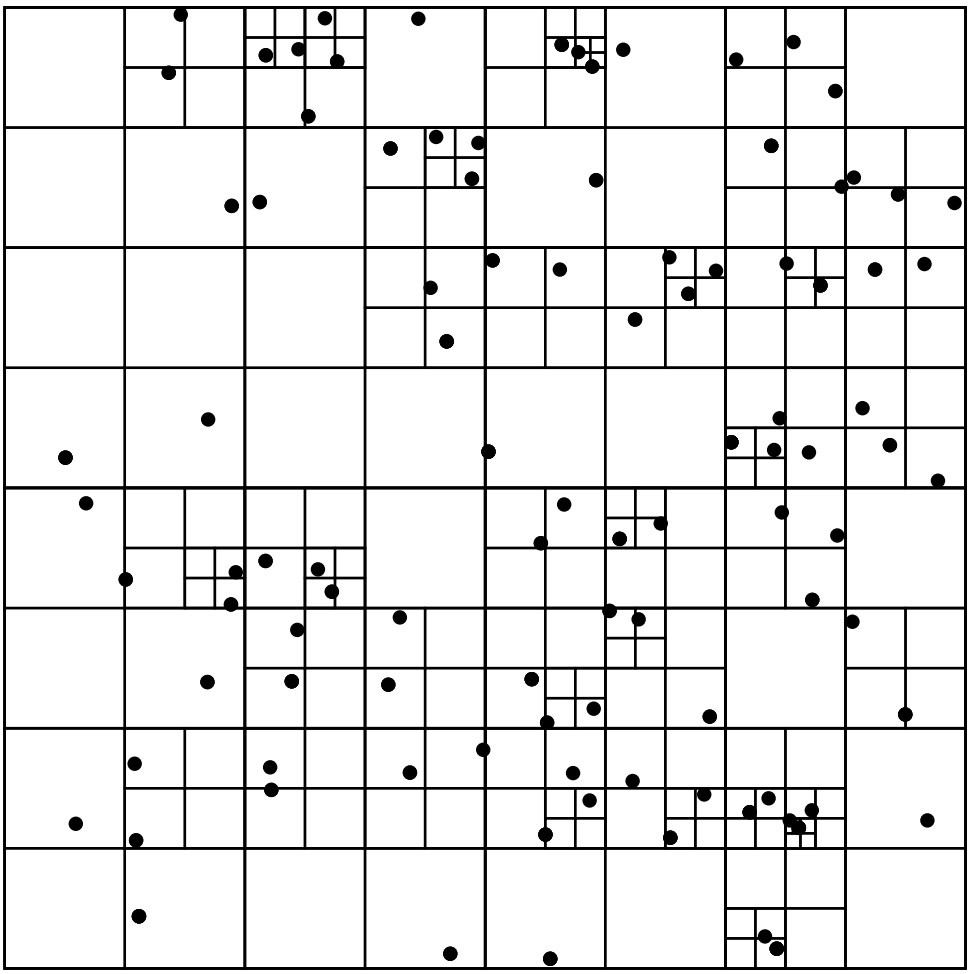


図 7.1. Barnes-Hut Tree．正方領域中に粒子が存在し，1 つの領域内に高々1 つの粒子が存在するもしくは予め定められた最大レベルに達するまで，領域の均等な 4 分割が繰り返される．

て得られる木構造である．この問題設定によって作られた 8 分木をバランスする時間が評価対象となる．図 7.1 は 2 次元の場合のメッシュと粒子の図である．

このモデルでは粒子の分布を調整することよって木構造の偏りを調整することが可能で，今回は一様な分布とガウス分布を用いて評価を行った．一様な分布の場合には木構造に偏りがなく初めからバランスされているため，バランス処理によるセルの細分化は起きず，また並列性能を出しやすい設定と言える．一方ガウス分布では極端に偏った木構造が生成されるため多くの細分化が起こり，また並列性能を出しにくい設定と言える．

具体的なパラメータの設定は表 7.2 の通りである．一般にバランス処理の問題サイズはバランス処理後のリーフセルの数で考えられ，この数と問題サイズが一致するようになっている．

なお，本実験では大量の動的なメモリ確保が行われる．この際に標準のメモリ割り当て関数 `malloc`(glibc 2.11.3) が並列実行のボトルネックとなることを避けるため，プログラムの初めで予め十分なメモリを確保している．

タイプ	問題サイズ	粒子数	バランス前のリーフセル数	バランス後のリーフ数	最大の深さ
Gaussian	1M	180K	608K	1M	14
Gaussian	2M	361K	9.7M	2M	15
Gaussian	4M	720K	9.7M	4M	14
Gaussian	8M	1.43M	4.82M	8M	16
Gaussian	16M	2.94M	9.94M	16M	16
Gaussian	32M	5.8M	19.6M	31.8M	17
Regular	16M	16.8M	16.8M	16.8M	8
Regular	32M	10.5M	32.1M	32.1M	9
Regular	63M	23.1M	63.7M	63.7M	9

表 7.2. 問題サイズとパラメータ設定

.

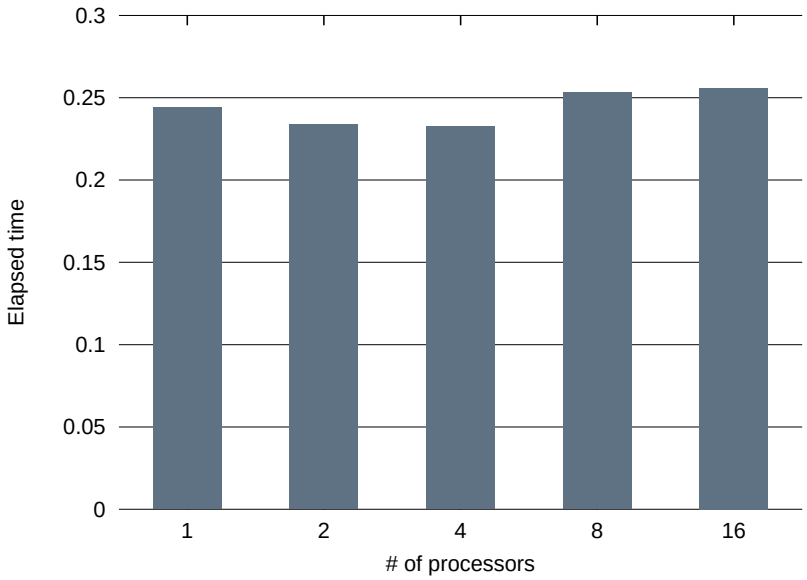


図 7.2. ウィークスケール．1 プロセスあたりの問題サイズが 1M である．

7.3.2　パフォーマンス

図 7.2 はガウス分布の場合における，Weak Scale のグラフである．この実験では 1 プロセッサあたりの問題サイズを 1M としている．グラフからわかる通り，非常に良い並列性能を示している．またこの結果は，提案アルゴリズムが $O(n)$ であることの傍証にもなっている．

図 7.3 は一様分布，ガウス分布ともに 16M, 32M, 64M でサイズを固定した時の逐次計算と比べたスピードアップのグラフである．最大 16 並列の際に，一様分布の場合で約 15 倍，ガウス分布の場合でも約 14 倍と非常に良い性能が観測できた．

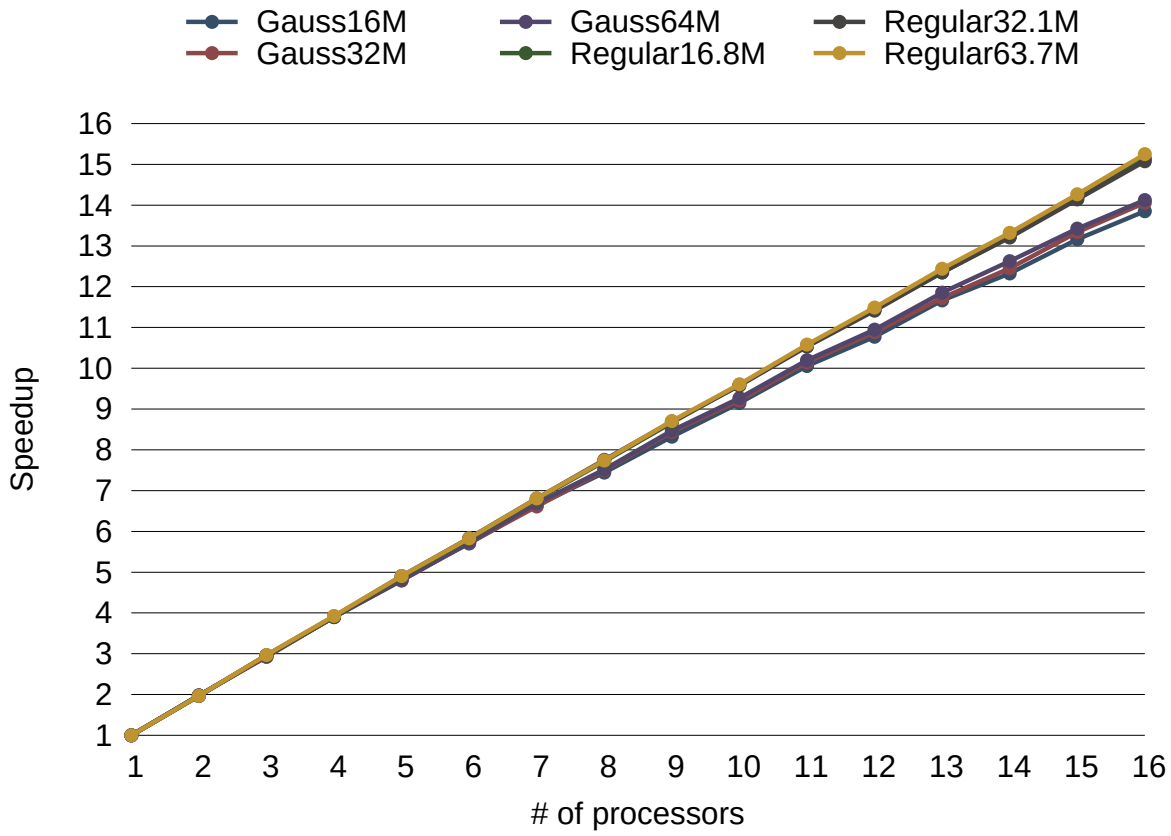


図 7.3. ストロングスケール．一様な分布の場合には 16 並列で約 15 倍，ガウス分布の場合でも約 14 倍のスピードアップが得られた．

7.4　移流方程式のシミュレーション

Tree-based AMR による計算の題材として，有限差分法を用いて移流方程式のシミュレーションを行った．移流方程式は特に流体力学との関係が深く，様々な科学技術計算の中でも最も重要な方程式の 1 つである．拡散方程式と比べると，細分化が必要となる領域が移動していくため動的なロードバランスが難しい問題と言える．

7.4.1　定式化

2 次元の定常移流方程式は以下のように表される．

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} + v\frac{\partial f}{\partial y} = 0$$

移流方程式は物理的には波の伝搬と対応しており，数値安定性の観点から空間差分には風上差分が適していることが知られている［47］．時間差分には拡散方程式と同様に陽解法を用いる．どちらも一次精度を用いて，以下のように離散化できる．

$$\frac{f_{i,j}^{n+1}-f_{i,j}^{n}}{\Delta t}+u\frac{f_{i,j}^{n}-f_{i-1,j}^{n}}{\Delta x}+v\frac{f_{i,j}^{n}-f_{i,j-1}^{n}}{\Delta x}=0 \qquad (u\geq 0, v\geq 0)$$

$$\frac{f_{i,j}^{n+1}-f_{i,j}^{n}}{\Delta t}+u\frac{f_{i+1,j}^{n}-f_{i,j}^{n}}{\Delta x}+v\frac{f_{i,j}^{n}-f_{i,j-1}^{n}}{\Delta x}=0 \qquad (u< 0, v\geq 0)$$

$$\frac{f_{i,j}^{n+1}-f_{i,j}^{n}}{\Delta t}+u\frac{f_{i,j}^{n}-f_{i-1,j}^{n}}{\Delta x}+v\frac{f_{i,j+1}^{n}-f_{i,j}^{n}}{\Delta x}=0 \qquad (u\geq 0, v< 0)$$

$$\frac{f_{i,j}^{n+1}-f_{i,j}^{n}}{\Delta t}+u\frac{f_{i+1,j}^{n}-f_{i,j}^{n}}{\Delta x}+v\frac{f_{i,j+1}^{n}-f_{i,j}^{n}}{\Delta x}=0 \qquad (u< 0, v< 0)$$

7.4.2　タイムステップの決め方

CFL 条件から［47］，以下のように決める．

$$\Delta t=\beta * Min(\Delta x/u, \Delta y/v)$$

上の式で β は安全係数であり今回は 0.2 を用いた．

7.4.3　細分化・集約化とバランス

細分化・集約化処理は 6.4.3 章，6.4.4 章で述べた操作を行う．6.1 章で説明したように，具体的には以下のような基準を用いる．

$val_{max}-\mu \geq K_{high}\cdot\delta$ **である場合**　細分化する
$val_{max}-\mu \leq K_{low}\cdot\delta$ **である場合**　集約化する
その他の場合　何もしない

今回この式における定数は，$K_{high}=2$ と $K_{low}=0.1$ とした．

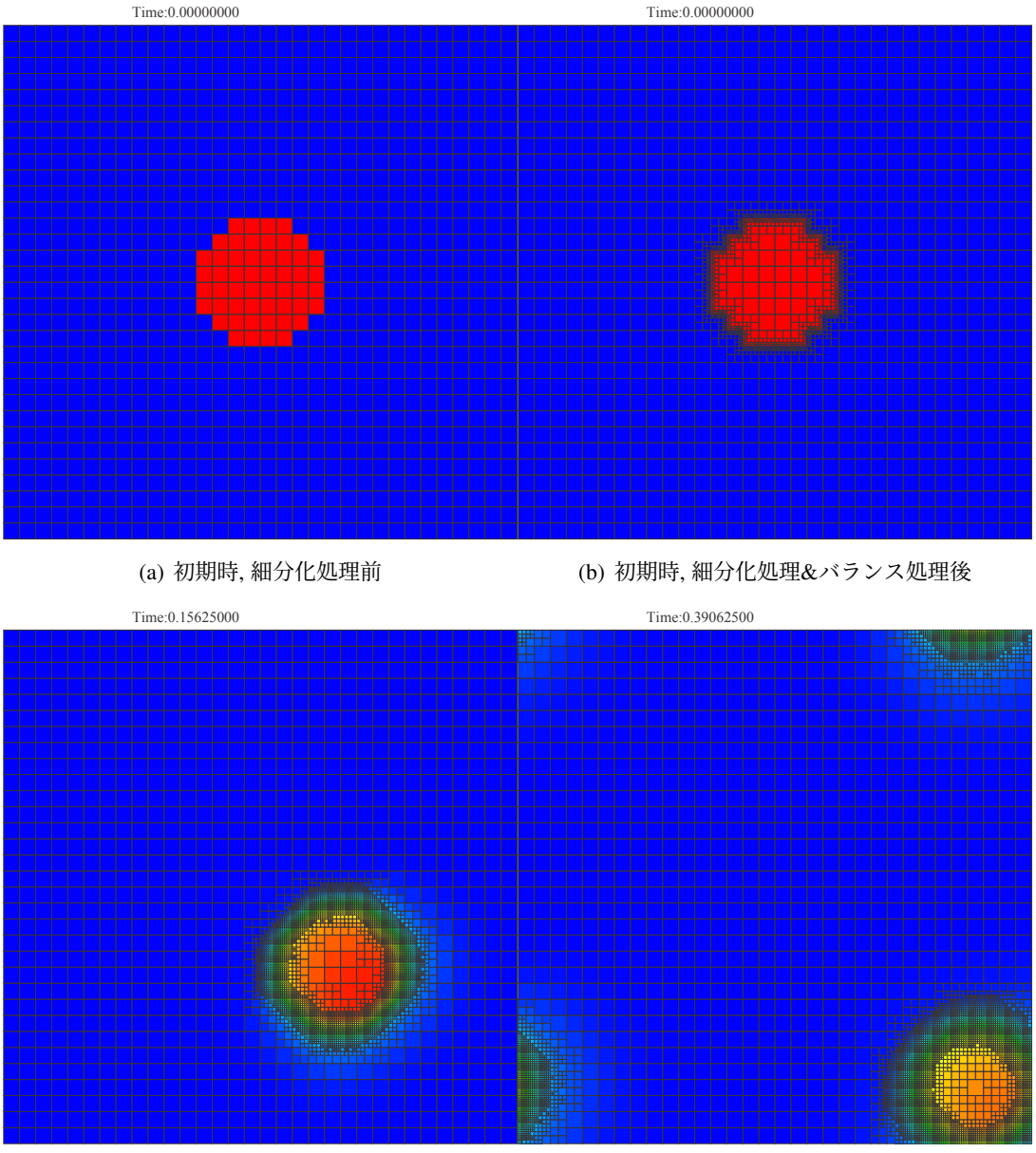
この細分化・集約化処理とそれに伴うバランス処理は，本来シミュレーション中のある程度のタイムステップ毎に（例えば 100 回に 1 度）行えば良いものであるが，今回はこれらの操作のパフォーマンスを評価するために毎タイムステップ行なっている．

7.4.4　パラメータ

タスク生成のしきい値として，6.3 章で述べた `CUTOFF_LEVEL` を 4 としている．従って，単純な全リーフノードに対する処理ではタスクの数は $1+4+4^2+4^3+4^4$ となる．

図 7.4 は初期レベル 5，最大レベルを 8 としたときのシミュレーションを可視化した画像である．今回値の初期配置としては図 7.4 のように，中央に円形に高い値の場所が来るような配置を用いた．

また以降の性能評価では最小レベル 13，最大レベル 19 としたときの実験結果を示すが，この設定では平均して 90M ノード，67M リーフノードに対する計算が行われている．



(a) 初期時, 細分化処理前　(b) 初期時, 細分化処理&バランス処理後

(c) 200 タイムステップ後　(d) 500 タイムステップ後

図 7.4. Tree-based AMR を用いた移流方程式のシミュレーションを可視化した例．初期レベルは 5，最大レベルは 8 としている．

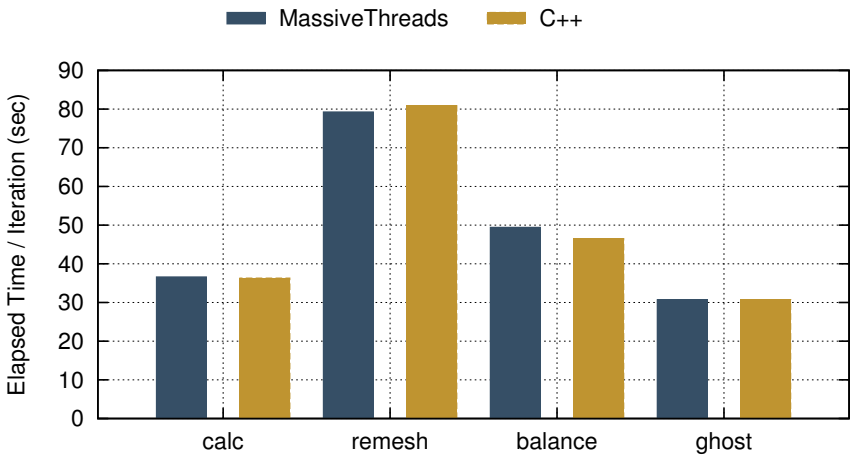


図 7.5. MassiveThreads の逐次実行と並列化を行なっていない C++プログラムの実行の，1 イテレーションのあたりの各フェーズの実行時間の比較

7.4.5　MassiveThreads と C++の逐次性能の比較

図 7.5 は，最小レベル 13，最大レベル 19 としたときの，MassiveThreads の逐次実行と並列化を行なっていない C++プログラムの実行の，1 イテレーションのあたりの各フェーズの実行時間の比較のグラフである．MassiveThreads では，先に述べたとおり `CUTOFF_DEPTH` を 4 としており，$1+4+4^2+4^3+4^4$ 個のタスクが生成されている．グラフから明らかな通り C++と MassiveThreads の間には殆ど差はなく，今回のプログラムでは MassiveThreads のタスク生成やタスクの切り替え等のオーバヘッドによる性能への影響はほぼ無い事が確認できた．

7.4.6　各操作の実行時間と割合

図 7.6 と図 7.7 はそれぞれ，最小レベルを 13，最大レベルを 19 とした時の各フェーズの実行時間とその割合を示したグラフである．計算時間中で支配的な順に，Remesh, Calc, Balance, Ghost となっている．Remesh では全リーフノード間の物理量の差を集めた後，その後全リーフノードに対して細分化ないし集約化を行う．つまり，全リーフノードに対する処理，と全リーフノードに対する隣接アクセスを伴う処理，の両方が行われている．他の 3 つのフェーズ，Calc, Balance, Ghost では隣接アクセスを伴う処理が行われている．ただし Calc ではリスト 6.3 で述べた，バランス操作で用いるアルゴリズムよりも高速なアルゴリズムが用いられている．それにも関わらず他の 2 つの操作よりも実行時間が長いのは，他の 2 つがほとんどの場合で木構造を辿るだけで書くノードの保持するデータに対する読み書きや計算を行わないのに対して，Calc の処理では多く行うからだと思われる．この事実は，図 7.7 で並列度が高くなるに連れて Calc の割合が少しずつ大きくなっていることにも関係していると思われる．全てのフェーズが全てどの並列度でもある程度の割合を占めていることから，どのフェーズに対しても効率的に並列化を行うことが重要であるとわかる．

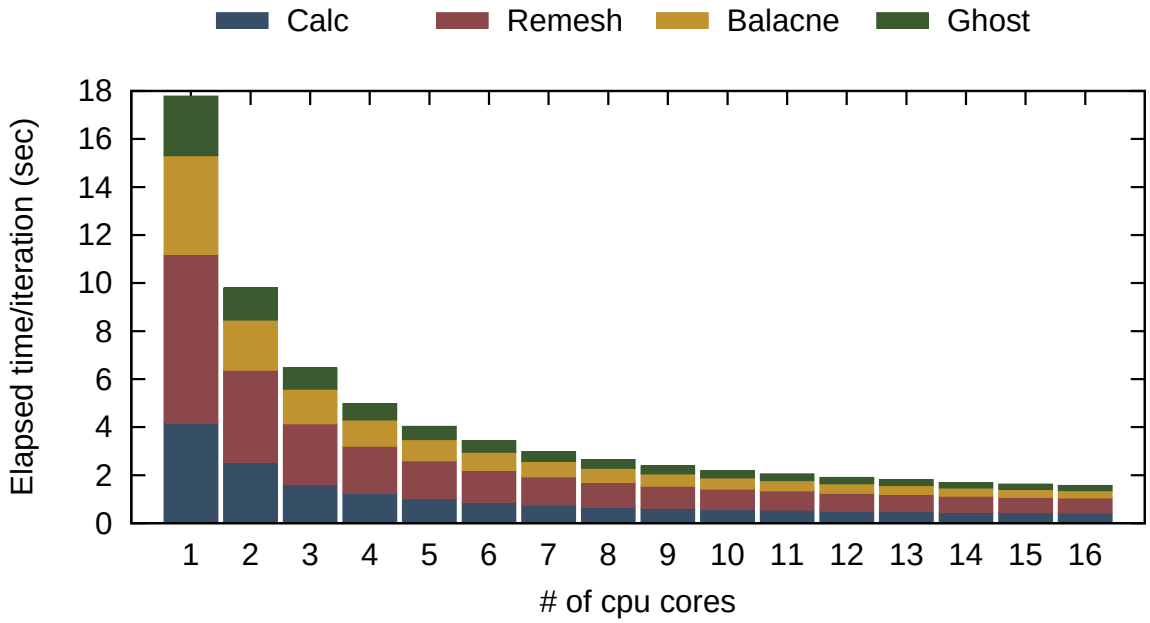


図 7.6. 移流シミュレーションおける 1 イテレーションあたりの実行時間

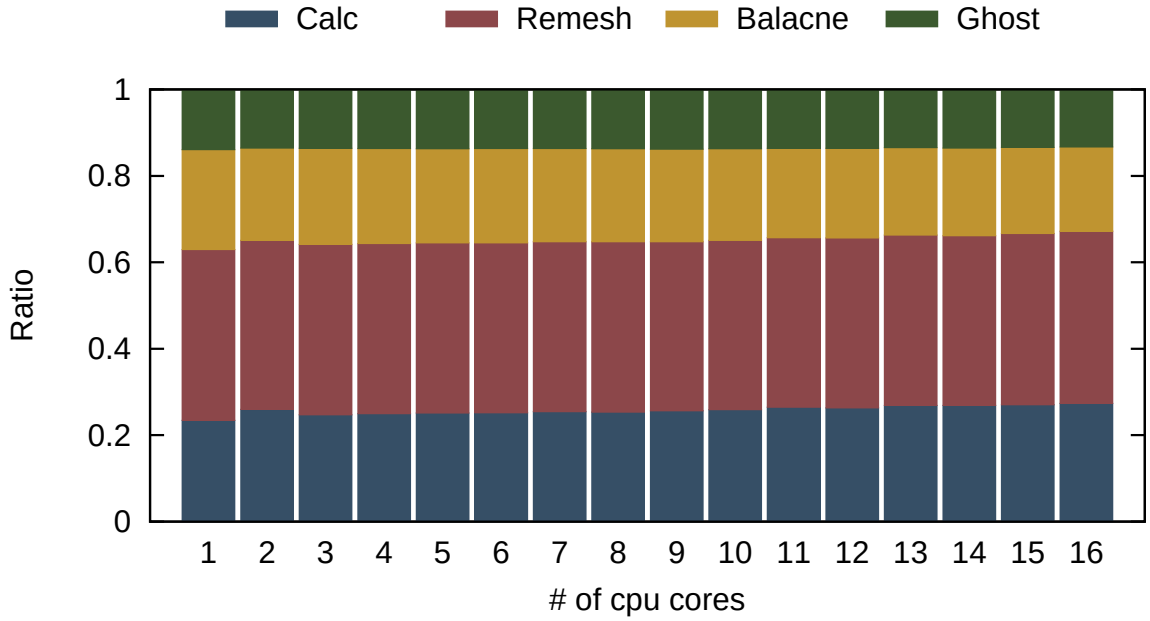


図 7.7. 移流シミュレーションにおける各操作の実行時間割合.

7.4.7　各操作のスケーラビリティ

図 7.8 は，前章と同じ実験での各フェーズ及び全体の並列性能を示している．バランス操作に対しては 16 コアで約 14 倍と非常に有効な性能を示している．全体では約 11 倍強といったところで，並列化効率としては約 71%である．一方で Calc フェーズの並列性能の伸び相対的に悪く，最大でも 10 倍程度のスピードアップとなっている．これは図 7.7 からもわかる通りで，高並列計算

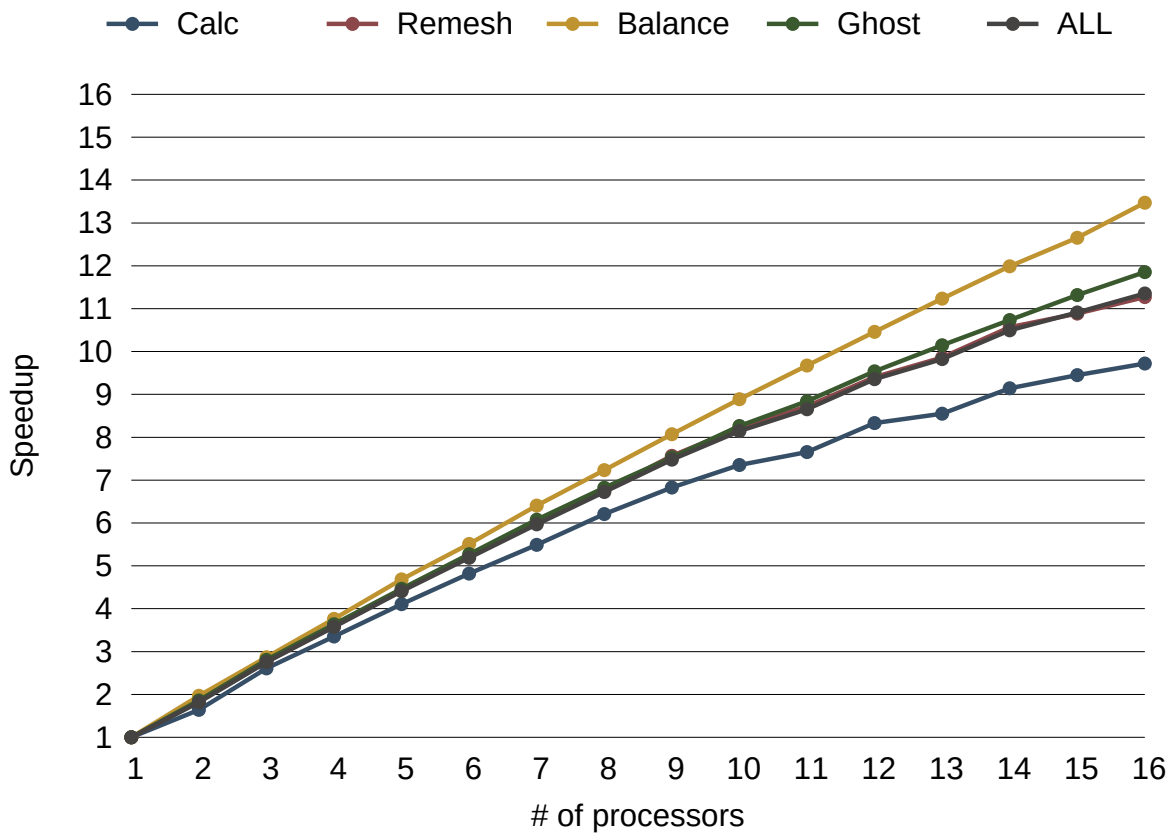


図 7.8. 各操作及び全体の台数効果

では計算全体の高速化の足を引っ張る形となっている．

7.5　拡散方程式のシミュレーション

次に，Tree-based AMR による計算の題材として，有限差分法を用いて拡散方程式の数値シミュレーションを行った．拡散方程式は自然科学・工学等幅広い分野で重要な現象を表す式であり，効率的な並列数値計算への要請は大きい．今回は特に熱伝導方程式を念頭にシミュレーションを行った．熱伝導方程式は拡散方程式によって表される代表的な方程式である．

7.5.1　定式化

以下に示すのは 2 次元熱伝導方程式である．

$$\rho c \frac{\partial f}{\partial t} = \lambda_x \frac{\partial^2 f}{\partial x^2} + \lambda_y \frac{\partial^2 f}{\partial y^2} + Q(x, y, t)$$

上の式で ρ は密度，c は熱容量，T が温度，λ が熱伝導度，$Q(x, y, t)$ は外力を表している．今回は ρ, c, λ を定数ととした．上式右辺の第一項，すなわち T の x での 2 階微分の項は中央差分を用いて次のような離散近似を行う．

$$\frac{\partial^2 f_{i,j}}{\partial x^2} = \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{\Delta x^2}$$

また，時間微分には単純なオイラー陽解法を用いると以下のように表すことができる．

$$f_{i,j}^{n+1} = f_{i,j}^{n} + \Delta t \left.\frac{\partial f_{i,j}}{\partial t}\right|_{t=n}$$

y についての 2 階微分も同様にして，以上を合わせると拡散方程式は以下のように離散化できる．

$$f_{i,j}^{n+1} = f_{i,j}^{n} + \Delta t \left\{ \frac{\lambda_x}{\rho \cdot c} \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{\Delta x^2} + \frac{\lambda_y}{\rho \cdot c} \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{\Delta y^2} \right\} + Q(i, j, n)$$

定数に関しては，熱拡散率 $\alpha = \frac{\lambda}{\rho \cdot c} = 0.5$ として計算を行った．

7.5.2　タイムステップの決め方

今回用いた離散化方法で，数値的に安定となる条件は以下のようになる．

$$\Delta t \leq \frac{1/2\alpha}{1/\Delta x^2 + 1/\Delta y^2}$$

従って今回は，最も小さいセルの一辺の長さ（$= \Delta x = \Delta y$）を右辺に代入し，安全係数 0.5 を掛けたものをタイムステップとして計算を行った

7.5.3　細分化・集約化とバランス

7.4.3 章で説明した，移流方程式での方法と同様.

7.5.4　パラメータ

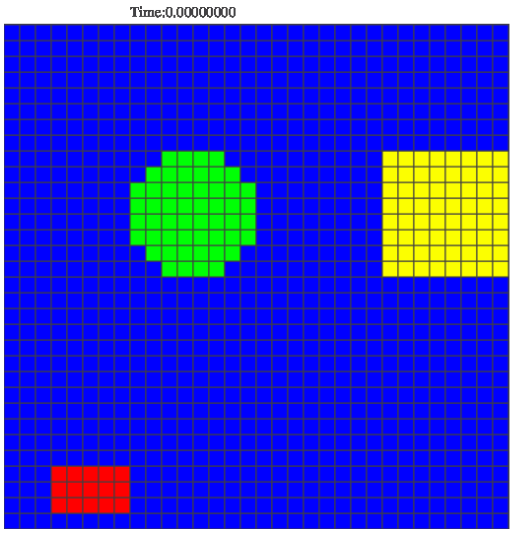
今回の評価実験では，主に初期レベル 13，最大レベル 19 としている．この設定の時，ノード数は平均で約 90M 個，リーフ数は平均約 67M 個となる．図 7.9 は本シミュレーションを可視化した画像である．この計算では初期レベルを 5，最大レベルを 8 としている．

7.5.5　各フェーズの実行時間とその割合

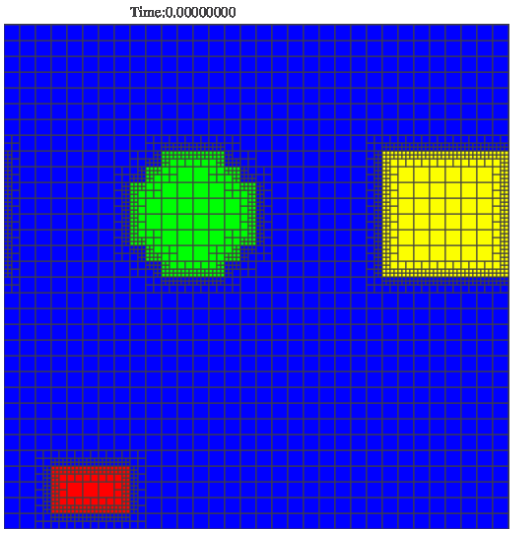
図 7.10，図 7.11 は最小レベル 13，最大レベル 19 としたときの各操作の平均時間とその割合を示したグラフである．全体として図 7.6，図 7.7 と同様の分布となっている．

7.5.6　各フェーズの台数効果

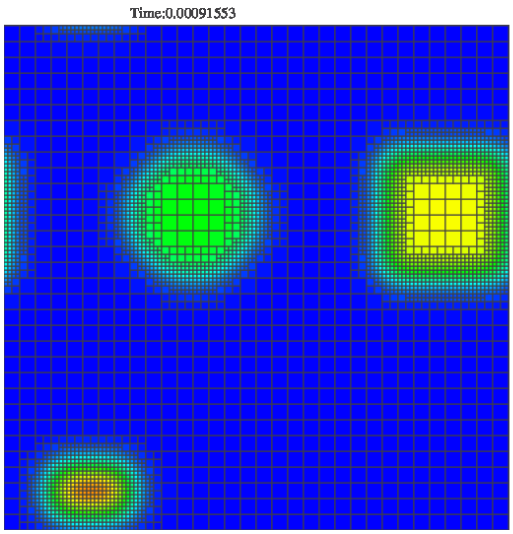
図 7.12 は初期レベル 13，最大レベル 19 としたときの各フェーズのストロングスケールを示したグラフである．バランス処理が最も良くスケールしていて，16 並列時に約 12 倍のスピードアップを得ている．一方で最も並列性能を落としているのは Calc 操作であり，約 9 倍の向上にとどまっている．この傾向は移流方程式のときにもみられたものである．



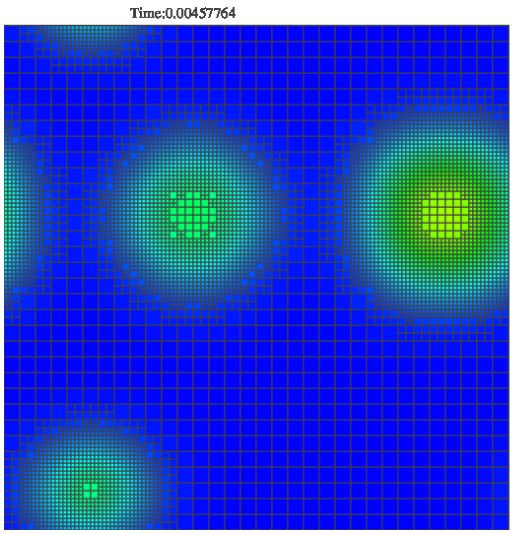
(a) 初期時, 細分化処理前



(b) 初期時, 細分化処理&バランス処理後



(c) 30 タイムステップ後



(d) 150 タイムステップ後

図 7.9. Tree-based AMR を用いた熱伝導方程式のシミュレーションを可視化した例．初期レベルは 5，最大レベルは 8 としている．

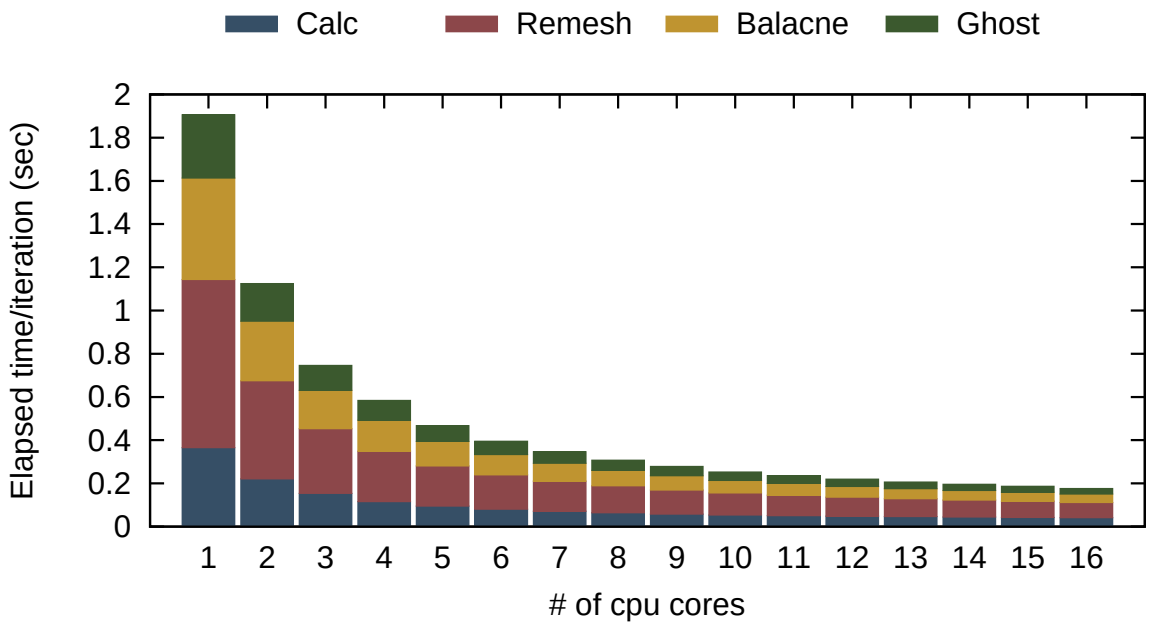


図 7.10. 拡散方程式拡散方程式シミュレーションでの 1 イテレーションあたりの実行時間

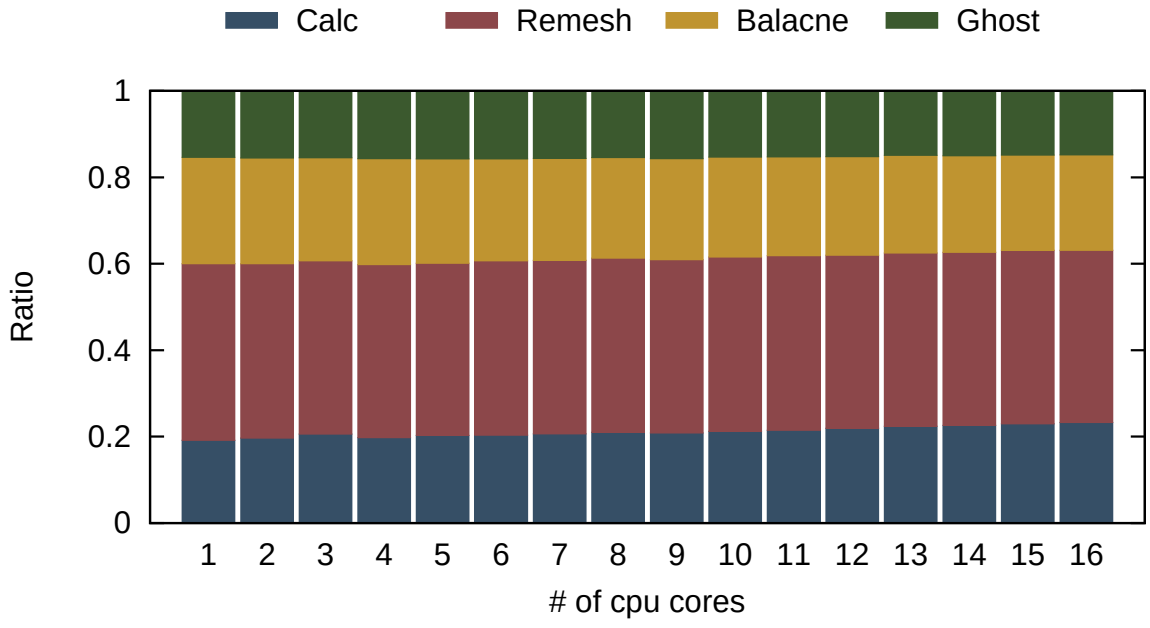


図 7.11. 拡散方程式シミュレーションでの各操作の実行時間割合.

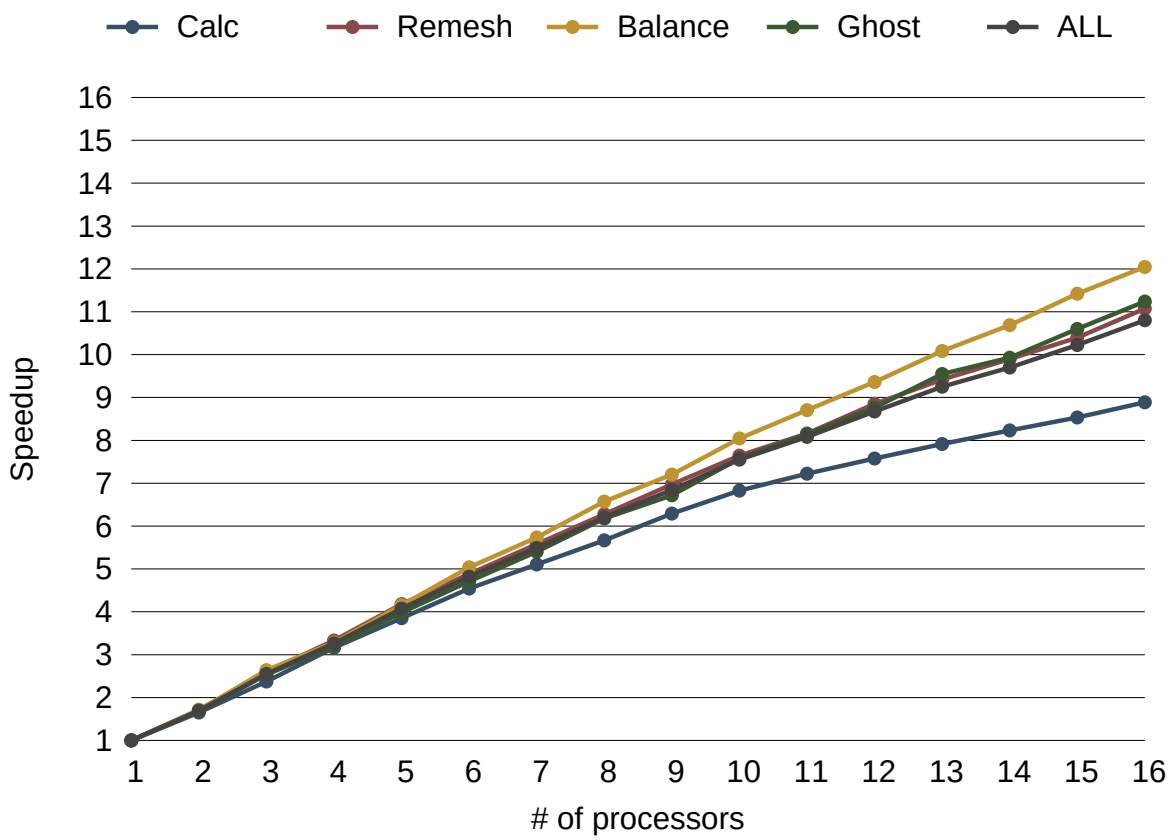


図 7.12. 各操作及び全体の台数効果

第8章　結論

8.1　まとめ

本研究では Tree-based AMR を効率的かつ見通しよく並列化を行うことを目的として，タスク並列モデルを用いたアルゴリズムを提案した．提案手法は分割統治法に基づいており，Tree-based AMR における種々の操作を効率良く記述することができる．またバランス処理を初めとする隣接ノードへのアクセスを伴う処理に対しては，$O(n)$ のコストで処理を行うことができる．これは従来の手法では $O(n\log(n))$ のコストを必要とするか，もしくは隣接アクセスを高速化するための特別なデータ構造を用いて $O(n)$ を実現していたことと比べて，大きな意味がある．Tree-based AMR はデータの分布が不均一であり，また計算が進むにつれて格子が変化していくため，並列プログラムにおける負荷分散が難しく様々な研究が行われてきた．これに対して提案手法ではタスク並列処理系が自動的に負荷分散を行うため，見通しよく効率的なプログラムを記述できる．

本研究では実際に提案するアルゴリズムを，タスク並列処理系 MassiveThreads を用いて実装し評価を行った．Tree-based AMR において最もコストの高い処理の１つであるバランス処理はほぼ理想的な並列性能に達した．また，工学・自然科学分野で頻繁に現れる重要な方程式，移流方程式と拡散方程式の有限差分法によるシミュレーションプログラムを用いて実装・評価を行い，提案手法の有効性を示した．

8.2　今後の展望

今回の研究では，他の手法との比較が行えていない．今後，実際に用いられている有力なアルゴリズムやアプリケーションとの比較を通して本提案手法の有効性を示していきたい．

また本研究で提案したアルゴリズムはタスク並列処理系を前提においている．しかしながら，現在のところ分散環境におけるタスク並列処理系の中で既存の SMPD 型の処理系に比肩する性能をもつものはまだない．従って本研究の目的である，見通しよく並列プログラムの記述が可能で，かつ高性能な Tree-based AMR アプリケーションは，分散環境では達成できていない．今回の研究成果を，分散環境におけるタスク並列処理系の実装にフィードバックし，両者の性能・使い勝手を高めていくことが必要だと考えられる．

他にも，6.1 章で述べたように，今回対象外とした手法や問題設定への適用も今後の課題である．これらの中でも特に有限要素法や有限体積法，境界要素法などの数値解析手法を用いる場合には計算の性質や必要となるデータ構造が大きく異なるので，これらへの提案手法の適用は更なる検討を要することが予想される．

参考文献

[1] Gilou Agbaglah, Sébastien Delaux, Daniel Fuster, Jérôme Hoepffner, Christophe Josserand, Stéphane Popinet, Pascal Ray, Ruben Scardovelli, and Stéphane Zaleski. Parallel simulation of multiphase flows using octree adaptivity and the volume-of-fluid method. *Comptes Rendus Mécanique*, 339(2-3):194–207, February 2011.

[2] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software*, 38(2):1–28, December 2011.

[3] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[4] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484–512, March 1984.

[5] M.J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, May 1989.

[6] OpenMP Architecture Review Board. Openmp application program interface. July 2011.

[7] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas Wilcox. Extreme-Scale AMR. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, November 2010.

[8] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Georg Stadler, Eh Tan, Tiankai Tu, Lucas C. Wilcox, and Shijie Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 62:1—-62:15. IEEE Press, November 2008.

[9] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. p4est : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, January 2011.

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10. *ACM SIGPLAN Notices*, 40(10):519, October 2005.

[11] P Colella, DT Graves, JN Johnson, ND Keen, TJ Ligocki, DF Martin, PW McCorquodale, D Modiano, PO Schwartz, TD Sternberg, and Van Straalen. Chombo Software Package for AMR Applications - Design Document. 2011.

[12] Geoffrey Cowles. Morphodynamic modeling using adaptive mesh refinement. http://www.smast.umassd.edu/CMLAB/.

[13] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, page 1, New York, New York, USA, November 2009. ACM Press.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*, volume 33, pages 212–223, New York, New York, USA, May 1998. ACM Press.

[15] L Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, December 1987.

[16] Thomas Guillet and Romain Teyssier. A simple multigrid scheme for solving the Poisson equation with arbitrary domain boundaries. *Journal of Computational Physics*, 230(12):4756–4771, June 2011.

[17] Salman Habib, Vitali Morozov, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Tom Peterka, Joe Insley, David Daniel, Patricia Fasel, Nicholas Frontiere, and Zarija Lukić. The universe at extreme scale: multi-petaflop sky simulation on the BG/Q. page 4, November 2012.

[18] Yukihiro Hasegawa, Motoyoshi Kurokawa, Hikaru Inoue, Ikuo Miyoshi, Mitsuo Yokokawa, Jun-Ichi Iwata, Miwako Tsuji, Daisuke Takahashi, Atsushi Oshiyama, Kazuo Minami, Taisuke Boku, Fumiyoshi Shoji, and Atsuya Uno. First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the K computer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, page 1, New York, New York, USA, November 2011. ACM Press.

[19] Cray Inc. Chapel specification version 0.92. October 2012.

[20] Tobin Isaac, Carsten Burstedde, and Omar Ghattas. Low-Cost Parallel Algorithms for 2:1 Octree Balance. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 426–437. IEEE, May 2012.

[21] Hua Ji, Fue-Sang Lien, and Eugene Yee. A new adaptive mesh refinement data structure with an application to detonation. *Journal of Computational Physics*, 229(23):8981–8993, November 2010.

[22] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++. *ACM SIGPLAN Notices*, 28(10):91–108, October 1993.

[23] A.M Khokhlov. Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations. *Journal of Computational Physics*, 143(2):519–543, July 1998.

[24] Akhil Langer, Jonathan Lifflander, Phil Miller, Kuo-Chuan Pan, Laxmikant V. Kale, and Paul Ricker. Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 100–107. IEEE, October 2012.

[25] Orion S. Lawlor, Sayantan Chakravorty, Terry L. Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant V. Kalé. ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235, September 2006.

[26] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM conference on LISP and functional programming - LFP '90*, pages 185–197, New York, New York, USA, May 1990. ACM Press.

[27] Message Passing Interface Forum. http://www.mpi-forum.org/.

[28] Jun Nakashima. MassiveThreads: A Lightweight Thread Library for High Productivity Languages. http://code.google.com/p/massivethreads/, Last accessed Sep 22 2012.

[29] Jun Nakashima and Kenjiro Taura. Multithread Framework That Manages both Efficient I/O and Lightweight Thread Management (In Japanese). *IPSJ Transaction (PRO)*, 4(1):13–26, March 2011.

[30] C.D. Norton, J.Z. Lou, and T.A. Cwik. Status and directions for the PYRAMID parallel unstructured AMR library. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 1224–1231. IEEE Comput. Soc, 2001.

[31] D.R. O'Hallaron. A Computational Database System for Generatinn Unstructured Hexahedral Meshes with Billions of Elements. In *Proceedings of the ACM/IEEE SC2004 Conference*, pages 25–25. IEEE, November 2004.

[32] Brian W. O'Shea, Greg Bryan, James Bordner, Michael L. Norman, Tom Abel, Robert Harkness, and Alexei Kritsuk. Introducing Enzo, an AMR Cosmology Application. page 10, March 2004.

[33] V. Parthasarathy, Y. Kallinderis, and Kengo Nakajima. Hybrid Adaptation Method and Directional Viscous Multigrid with Prismatic-Tetrahedral Meshes. *AIAA paper 95-067*, 1995.

[34] Stéphane Popinet. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *Journal of Computational Physics*, 190(2):572–600, September 2003.

[35] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[36] Rahul S. Sampath, Santi S. Adavani, Hari Sundar, Ilya Lashuk, and George Biros. Dendro: parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 18:1—-18:12, Austin, Texas, November 2008. IEEE Press.

[37] Pengtao Sun, Robert D. Russell, and Jinchao Xu. A new adaptive local mesh refinement algorithm and its application on fourth order thin film flow problem. *Journal of Computational Physics*, 224(2):1021–1048, June 2007.

[38] Hari Sundar, Rahul S. Sampath, Santi S. Adavani, Christos Davatzikos, and George Biros. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*, page 1, New York, New York, USA, November 2007. ACM Press.

[39] Hari Sundar, Rahul S. Sampath, and George Biros. Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, January 2008.

[40] R. Teyssier. Cosmological Hydrodynamics with a Adaptive Mesh Refinement: a new high resolution code called RAMSES. *Astronomy and Astrophysics*, 385(1):337–364, April 2002.

[41] TIANKAI Tu, D.R. O'Hallaron, and Omar Ghattas. Scalable Parallel Octree Meshing for TeraScale Applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 4–4. IEEE, November 2005.

[42] Tiankai Tu and David R. O 'hallaron. Balance Refinement of Massive Linear Octree Datasets. Technical report, 2004.

[43] Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, and Noah Elliott. Large scale parallel structured AMR calculations using the SAMRAI framework. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '01*, pages 6–6, New York, New York, USA, November 2001. ACM Press.

[44] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 1 edition, 5 2002.

[45] Jingjin Wu, Zhiling Lan, Xuanxing Xiong, Nickolay Y. Gnedin, and Andrey V. Kravtsov. Hierarchical task mapping of cell-based AMR cosmology simulations. In *ACM/IEEE SC 2012 Conference (SC'12)*, page 75, November 2012.

[46] Mark A. Yerry and Mark S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal for Numerical Methods in Engineering*, 20(11):1965–1990, November 1984.

[47] 藤井 孝蔵. **流体力学の数値計算法**. 東京大学出版会, 4 1994.

発表文献

[1] 河野瑛，田浦健次朗．タスク並列モデルを用いた Tree-based AMR の評価．**並列／分散／協調処理に関するサマーワークショップ（*SWoPP2012*）**, 鳥取, 2012/8.

[2] Akira Kono, Kenjiro Taura. A task parallel algorithm for 2:1 octree balance. *18th International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS2013)*, Boston, 2013/5. (submitted)

謝辞

まず，本研究を行うにあたり研究の方針から実装の細部に至るまで数多くの助言をいただいた田浦建次朗准教授に深く感謝します．研究生活の中でモチベーションを失いがちな私に対して，ときには厳しい教師として，ときには頼りがいのある先輩として，ときには私を励ましてくれる同僚のように接して頂き，どうにかここまで研究をまとめることができました．学部生の時に研究室に配属されてから，先生のもとで指導を受けたこの三年間で得たものは，私の中で本当に大きな意味を持っています．

また，情報基盤センターの中島研吾先生には，Tree-based AMR の背景知識や詳細な仕様の策定にあたって有用なアドバイスをいくつも頂きました．D2 年の中島潤先輩と D1 の秋山茂樹先輩，同期の池上克明君には度々研究やプログラミングに関する具体的な相談に乗って頂き，研究を進める上で大変お世話になりました．また，同期の堀内美希さんや M1 の中谷翔君には日々を研究室で過ごす上で，様々な面で支えて頂きました．ポスドクの頓楠さん，M1 の林伸也君，中澤隆久君，アムガラン・ガンバト君には研究室の運営やミーティングでお世話になりました．

ここには書ききれませんが，その他にも研究生活を送る中で本当に多くの方々にお世話になりました．この場を借りて厚くお礼申し上げます．最後に，これまで支えてくださった家族に感謝いたします．本当にありがとうございました．