Vol.2013-HPC-140 No.18
2013/8/1

ワークスチーリング戦略のカスタマイズによるAMR法の高速化

中島潤[1,a)]　河野瑛[2]　田浦健次朗[1,b)]

概要：タスク並列処理によって記述されたアプリケーションを効率的に実行するためには，アプリケーションやハードウェアに関する知識を考慮してタスクのスケジューリングを行うことが必要である．これを達成するために，我々は以前にワークスチーリングの戦略をカスタマイズ可能なタスク並列処理の実行時システムによって，プログラマがスケジューリング方針を最適化する手法を提案した．本稿では提案手法の有効性を評価することを目的として，適合細分化格子法 (AMR) による移流計算に対して提案手法を適用し，ワークスチーリング戦略をカスタマイズして評価を行なった結果について述べる．

キーワード：タスク並列処理，スケジューリング，適合細分化格子法

Accelerating Adaptive Mesh Refinement by Customizing Work Stealing Strategy

Jun Nakashima[1,a)]　Akira Kono[2]　Kenjiro Taura[1,b)]

Abstract: In order to execute task-parallel application efficiently, it is important to take characteristics of application and hardware into account. In order to address this issue, previously we proposed a runtime system which can customize the strategy of work stealing. In order to evaluate the effectiveness of our proposal, we applied the customization to an advection calculation using Adaptive Mesh Refinement. This paper describes the customized work stealing strategy and its evaluation results.

Keywords: Task Parallelism, Scheduling, Adaptive Mesh Refinement

1. はじめに

ノード内のコア数の増加，NUMA アーキテクチャによるメモリの多階層化，アクセラレータの導入などによって現在の並列計算機は非常に大規模かつ複雑なものとなっており，さらに今後ますます複雑化していくものとみられている．それに対して現在広く普及しているプログラミング処理系の多くは，並列計算機のハードウェアを単純に抽象化した機能を提供しており，このような大規模並列計算機の性能を引き出すようにアプリケーションを記述することは困難である．そのため，ハードウェアをより高度に抽象化したプログラミングモデルを提供しており，平易な記述で高い実行性能を達成できる生産性の高い並列プログラミング環境の構築が並列計算における重要な課題となっており，これを解決するため Chapel[1] や X10[2] をはじめとする様々なプログラミング処理系が提案され，活発に開発が行われている．

これらの処理系の多くは，アプリケーションの並列性を表現するための機能としてタスク並列処理を提供している．タスク並列処理では，処理全体が細粒度に分割して軽量なスレッドに割り振られ (この軽量なスレッドは一般にタスクと呼ばれる．本稿でも以降タスクと呼ぶ)，それらを実行時システムがハードウェア (多くの場合計算ノード内

1 東京大学
University of Tokyo
2 日立製作所
Hitachi
a) nakashima@eidos.ic.i.u-tokyo.ac.jp
b) tau@eidos.ic.i.u-tokyo.ac.jp

の CPU コアであるため，本稿でも以降 CPU コアであるとする) に適切に割り振って実行する．十分に多くのタスクに処理が分割されていれば，処理系がタスクを CPU コアに割り振ることによって動的な負荷分散が自動的に行われるため，タスク並列処理を用いることで再帰や多重ループをはじめとする，処理を単純に分割するだけでは効率的な並列化ができないアプリケーションを平易に記述することができる．

アプリケーション内で多くのタスクが生成，破棄されるため，タスク並列処理の実行時システムには，小さなオーバーヘッドでタスクを生成，破棄したうえで，それらを効率的に各 CPU コアにスケジューリングすることが求められる．これを達成するためのスケジューリング手法としては，Lazy Task Creation[3] が広く知られているが，この手法はタスク間の負荷のばらつきなどのアプリケーションの特性やメモリ階層などのハードウェアの特性を考慮したスケジューリングを行わない．

そのため，これらの特性を活かすことが特に重要になる大規模な分散メモリ環境では，単に Lazy Task Creation を利用するだけでは効率的なスケジューリングを行えないと考えられる．今後並列計算機の複雑化にしたがって，この影響がますます大きくなることを考えると，アプリケーションやハードウェアの特性について配慮したタスクのスケジューリングを行うことは，タスク並列処理を用いて大規模並列計算機上で高い実行性能を達成するうえでの重要な課題であるといえる．

この課題を解決するために，我々はタスクのスケジューリングの戦略をカスタマイズすることが可能な機能を備えたタスク並列処理系によって，プログラマがその方針をアプリケーションやハードウェアにあわせて最適化する手法 [4] を提案しており，さらに単純なベンチマークを用いてその有用性を評価している．

我々はより実際のアプリケーションに近く，なおかつタスク並列処理の利点を活かせるアプリケーションを用いて提案手法の有用性を評価することを目的として，河野ら [5] による，タスク並列処理を用いて実装された適合細分化格子法 (AMR) による移流計算を題材として，これに計算粒度の不均衡と NUMA アーキテクチャを考慮するようなスケジューリング戦略のカスタマイズを適用して評価を行なった．本稿では，タスク並列 AMR に対して適用したスケジューリング戦略とその評価について述べる．

2. タスク並列処理のスケジューリング

2.1 Lazy Task Creation

タスクの生成や合流処理を並列に行うため，タスク並列処理によるアプリケーションは多くの場合，再帰的にタスクを生成する形で記述される．そのような形で記述されたタスク並列処理によるアプリケーションを効率的に実行するためのスケジューリング手法として，Cilk[6] をはじめとする，共有メモリ環境を対象とした多くのタスク並列処理系において用いられている手法として Lazy Task Creation[3] がある．

Lazy Task Creation では，各 CPU コア上にタスクを実行するための OS レベルのスレッド (本稿ではワーカーと呼称する) と，実行可能なタスクを格納するためのキュー (本稿ではランキューと呼称する) を配置し，それぞれのワーカーがランキューからタスクを取りだし，実行することでタスク並列処理を行う．

各ワーカーは以下の方針に基づいてタスクを実行する:

- タスクを作成する際は，それまでのタスクを中断してランキューの先頭に格納して，新しく作成したタスクを即座に実行する
- タスクが終了あるいは中断した際はランキューの先頭からタスクを取りだして実行する
- ランキューにタスクがない際には他のワーカーをランダムに選択して，その末尾からタスクを奪ってくる (ワークスチーリング)

ランキュー末尾からのワークスチーリングによって奪われるタスクは，タスクの親子関係の木構造の根に近く，将来的に多くの子タスクを生成する，すなわち粒度が大きいタスクであるため，少ないスチール回数で効率的に負荷分散をとることができる．また，ワークスチーリングが発生した箇所を除けば，それぞれのタスクは逐次に再帰呼び出しを行ったのとまったく同じ順序で実行されるため，再帰の実行順を利用してメモリアクセスなどの最適化を行った逐次プログラムをタスク並列処理によって並列化した際にも，まったく同様に最適化の効果が得られることが期待される．

ワークスチーリングの際にタスクを奪う対象は実際に奪われるタスクの粒度や，ハードウェアの構成などの要素を考慮することなくランダムに選ばれる．そのため，特にこれらの要素に配慮することが重要な場合に大きな性能低下が発生しうる．例えば分散メモリ環境においては，データにアクセスするオーバーヘッドがノード内とノード間で大きく異なるため，ランダムなワークスチーリングではデータの配置を意識したタスクのスケジューリングができないことが原因で実行性能が大幅に低下すると考えられる．

2.2 ワークスチーリング戦略のカスタマイズ

様々なハードウェアやアプリケーションの特性を考慮した，効率的なタスクのスケジューリングを行うことを目的として，我々はスケジューリング戦略をアプリケーションやハードウェアにあわせてプログラマがカスタマイズする手法 [4] を提案して，それを MassiveThreads[7], [8] 上に実装している．この手法では，タスクの実行順や，大きい粒度のタスクのスチールを行えるといった Lazy Task Creation

の利点を維持しつつスケジューリング戦略をカスタマイズするために，カスタマイズ可能な範囲をワークスチーリングの戦略に絞り，それに必要な機能をプログラマに提供している．

提案する手法を用いてワークスチーリング戦略のカスタマイズを行う際の手順を以下に示す．

2.2.1 ヒントの割り当て

```
1  void user_task (void *param){
2    ...
3    // 追加部分: ヒントを割り当てる
4    struct {...} hint;
5    set_task_hint(&hint,sizeof(hint));
6    // 再帰的にタスクを作成
7    spawn_task(user_task,param1);
8    spawn_task(user_task,param2);
9    ...
10 }
```

まず，アプリケーションに変更を加え，ワークスチーリングのヒントとして用いられる任意のデータを各タスクに割り当てる．ヒントの内容や割り当ては対応するタスクから任意のタイミングで変更，解除が可能である．ヒントとして割り当てるデータの種類は問われないが，確保，解放の手間を簡略化できることを考えると，タスク内のローカル変数を割り当てることが最も望ましい[*1]．

Lazy Task Creation においては，タスクを作成する際に新しく作成されたタスクが即座に実行され，親のタスクがランキューに格納されて他のワーカーにスチールされる．したがって，ここで割り当てられるデータの内容は，タスクを作成した後に実行する処理に関するものとするべきである．例えば上の疑似コードでは，6 行目でタスクが作成されたことによって，このタスク他のワーカーにスチールされた後は，そのワーカー上で 7 行目の `param2` を引数とするタスクを作成するコードが実行される．したがって，5 行目で割り当てられるヒントの内容は，7 行目のタスク作成に関するものであることが望ましい．

2.2.2 ユーザー定義のワークスチーリング関数の実装

```
1  // ワークスチーリング関数: ワーカーがアイドル状態
2  // になった際に呼び出される
3  void user_defined_steal(int id)
4  {
5    task_handle t_stolen;
6    // 1.スチール対象の候補を決める
7    int candidates[] = get_random_workers();
8    // 2.ヒントを読み取る
9    struct {...} hints = readydeque_peek(candidates);
10   /*
11    * 3.ヒントをもとに実際にスチールを行う
12    * 対象や順序を決める
13    */
14   int targets[]=...;
15   // 4.スチールを行う
16   for t in targets{
17     t_stolen = readydeque_trysteal(t,
18       steal_confirm, data);
19     if (t_stolen)break;
20   }
21   return t_stolen;
22 }
23
24 // 確認用関数: ワークスチーリングが成功することが
25 // 確定した際呼び出される
26 int steal_confirm(task_handle_t stolen,
27   void *userdata)
28 {
29   int ok;
30   /*
31    * 5.ここで実際にスチールするか判断
32    */
33   // 0以外を返すとスチール完了
34   // 0を返すと中断
35   return ok;
36 }
37
38 int main(int argc, char **argv)
39 {
40   ...
41   // 6.ワークスチーリング関数を切り替え
42   set_steal_func(user_defined_steal);
43   ...
44 }
```

次に，ヒントを利用してワークスチーリングを行うユーザー定義のワークスチーリング関数を実装して，標準のワークスチーリング関数と切り替える．ワークスチーリング関数はワーカーのランキューにタスクがなくなった際に呼び出されるコールバック関数で，アイドル状態のワーカーの ID を引数としてとる．典型的には，ワークスチーリング関数は，以下のような手順でワークスチーリングを行うように実装される．

候補選択　ワークスチーリング対象の候補となる複数のワーカーを選択する．

ヒントの読み取り　それぞれの候補から盗むことができるタスクのヒントを読み取る．

スチール対象の決定　ヒントをもとにスチールを行う対象のワーカーやそれらの間の順序を決定する

スチールの試行　対象のワーカーに対してワークスチーリングを試行する．この際，コールバック関数としてスチールされたタスクを確認するための関数を同時に指定する．

結果の確認　ワークスチーリングの際にヒントを読み取ったものと同じタスクがスチールされるとは限らないため，実行するには望ましくないタスクがスチールされてしまう可能性がある．それに対処するため，スチールが成功することが確定した段階で，スチール試行時に指定したユーザー定義の確認用コールバック関数が呼び出される．この関数はスチールしたタスクをもとに，スチールを確定するか否かを判断する．

ユーザー定義のワークスチーリング関数，確認用関数，

*1 [4] で発表した際にはタスクの作成時にヒントの初期データとサイズを割り当て，それ以降は内容を変更することしかできなかったが，現在は改良が加えられ，上に説明したような動作となっている．また，タスク作成時に初期データを割り振る手法についても引き続き利用可能である．

およびワークスチーリング関数の切り替え部分の例を上の擬似コードに示す．

3. 関連研究

ハードウェアやアプリケーションの特性に配慮した効率的なタスクのスケジューリングを行うことを目的として，様々なスケジューリング手法が提案されている．

例えば，Chen らが提案する CATS[9] は，プロファイリングによって得られたタスクが利用するデータサイズを推測し，共有キャッシュからはみ出さないようにタスクの配置を行う．また，Wimmer ら [10] は一度に盗むタスクの数や実行順などをアプリケーションの特性に応じてプログラマが自由に設定できるスケジューラを提案している．

分散メモリ環境では，データにアクセスするオーバーヘッドがノード内とノード間で大きく異なるため，分散メモリ環境を対象としたタスク並列処理系の多くが，それに配慮したスケジューリングを行っている．

例えば，Scioto[11], [12] では，タスクが作成される際，そのタスクがノード内で実行されることが望ましい場合はランキューの先頭に追加するのに対し，別のノードで実行されるのが望ましい場合は，ランキューの末尾に追加して，他のノードに効率よくスチールされるようにしている．広域分散環境を対象としたタスク並列処理系である Satin[13], [14] は，広域ネットワーク上で効率的な負荷分散を行うために，ネットワークの階層構造にも配慮してワークスチーリングを行っている．

これら既存のスケジューリング手法の多くは，タスク並列処理系に組み込まれており，実際の処理はプログラマから隠蔽されている．そのため，スケジューリング手法に対して最適化を行う余地はあくまでタスク並列処理系が API として公開している範囲に限定されており，それ以上の高度なチューニングを行うためには，タスク並列処理系の内部的な実装に踏み込んで，スケジューラを改変することが求められる．アルゴリズムのみが発表されており，実装が入手できない手法をアプリケーションに対して適用する際についても同様に，何らかの既存のタスク並列処理系を改変してアルゴリズムを実装する必要がある．

それに対して我々が提案しているスケジューリングのカスタマイズ手法は，タスク並列処理系の実装とユーザーがカスタマイズする部分とは分離されているため，処理系の内部的な実装に踏み込むことなくスケジューリングの方針をカスタマイズすることができる．しかもスケジューリング方針は C/C++言語の関数として実装されるため，カスタマイズの自由度が非常に高い．

さらに，既存のスケジューリング手法の中には，実行順が逐次の再帰プログラムと異なるものがあるのに対して，我々が提案しているカスタマイズ手法では，カスタマイズの範囲がワークスチーリングの戦略に限局されていることにより逐次再帰と同じ実行順が保証されているため，それに由来する利点を失うことなくカスタマイズを行うことができる．

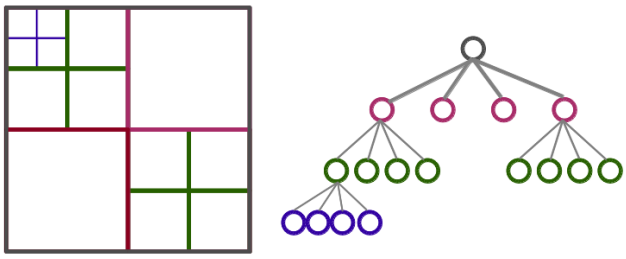

図 1　格子とそれに対応する四分木 ([15] より引用)
Fig. 1　Mesh and Corresponding Quadtree (cited from [15])

4. タスク並列処理による Tree-Based AMR

この章では，スケジューリングのカスタマイズの題材として用いた，河野ら [5] によるタスク並列 Tree-Based AMR を用いた二次元空間の移流計算について説明する．

4.1 Tree-Based AMR

数値シミュレーションによって偏微分方程式を解く際には計算対象の空間を離散化して，離散化された空間上の格子点に対する計算を行うことで解を得る．得られる解の精度は格子の細かさによって決まるため，高い精度の解を得るためには非常に多くの格子点が必要になり，それに伴って計算量も非常に大きくなる．しかし，実際には細かい格子による計算が必要な領域は領域内の一部であることが多い．そこで，高精度な計算が必要な部分には細かい格子を，低い精度の計算でも十分な部分には粗い格子を動的に割り当てることで，計算量の削減と高い精度の計算を両立する手法が適合細分化格子法 (AMR)[16] である．

Tree-Based AMR は AMR の一種で，立方体 (二次元の場合は正方形) の形状の直交格子を各座標軸に沿って再帰的に二分割して細分化することで，適合的な格子を生成するものである．計算領域を離散化する格子全体が四分木 (三次元の場合は八分木) で表現され，計算対象の格子点は四分木のリーフノードに対応している (**図 1**)．

4.2 計算の流れ

タスク並列 Tree-Based AMR による移流計算は，C++ 言語によって記述されており，MassiveThreads を用いて並列化されている．

各タイムステップは以下に述べる **calculate**，**check**，**remesh**，**balance**，**ghost** の 5 つのフェーズからなり，いずれのフェーズも再帰的にタスクを生成して四分木全体を並列に処理するような形で記述されている．

calculate　一次精度風上差分法によって各格子点のタイムステップを進める．

check　各格子点について，細分化，集約化の必要があるかどうかの判断は Parthasarathy ら [17] による，隣接格子点との物理量の差の統計量から判断する手法によって行われる．このフェーズではその判断に必要となる，各格子点における隣接する格子点との物理量の差の平均，および標準偏差を計算する．

remesh　**check** フェーズで計算した統計量と，隣接格子点との物理量の差の最大値を比較して，その結果をもとに格子の細分化，および集約化を行う．具体的には，val_{max} を隣接格子点との物理量の差の最大値，μ を平均，δ を標準偏差，K_{high}, K_{low} を定数としたとき，細分化，および集約化の判断は以下の式にしたがって行われる．

- $val_{max} - \mu \geq K_{high}\delta$ のとき:細分化
- $val_{max} - \mu \leq K_{low}\delta$ のとき:集約化

細分化，および集約化の際には物理量の補間が必要だが，この実装では簡単化のため，細分化する際には単に粗い格子の物理量をコピーしている．また，集約化を行う際には細かい格子の平均をとっている．

balance　解の精度を保つためには，隣接する格子の細分化レベル差が 1 以内に収まっている必要があるため，隣接する格子間の細分化レベルを比較して，その制約を満たすように細分化を行う．このフェーズは河野ら [15] によって提案された分割統治法に基づくアルゴリズムによって実装されている．

ghost　細分化レベルが異なる境界の計算を正しく行うために，両者の境界にある粗い格子の一つ下のレベルにダミーの格子 (ghost cell とよばれる) を配置して，粗い格子の物理量をそれに補間する．この補間も簡単化のため，粗い格子の物理量をコピーによって実装されている．

calculate，**remesh** フェーズは四分木全体を並列にたどって処理を行うのみなのに対して，**check**，**balance**，**ghost** フェーズは，各タスクが再帰的に子タスクを生成して全ノードに対する処理を行った後，子タスクの終了を待ち，改めて子ノードどうしが接する部分に対して再帰的にタスクを生成して処理を行う，という複雑な制御構造をもっている．

なお，典型的な AMR の実装では毎タイムステップ実行する必要があるのは **calculate** と **ghost** フェーズのみで，他のフェーズは適宜間隔を空けて行われるものであるが，本稿の評価では全フェーズを毎タイムステップごとに実行している．

4.3　スケジューリングの課題

上に述べたタスク並列 Tree-Based AMR の特性から導かれるスケジューリングの課題としては，以下の二点が挙げられる．

不均衡な木の構造　AMR の格子を表す四分木は，細分化が行われた一部の領域にノードが集中する不均衡なものとなるため，ランダムワークスチーリングによる負荷分散が前提としている，再帰の親子関係の根に近いほど粒度が大きいという特徴が成り立たない．そのためスチールの回数が増加し，それに伴うオーバーヘッドによって実行性能が低下すると考えられる．

メモリへの負担　フェーズごとに木構造をたどる必要があるため，メモリのバンド幅が AMR の性能に大きく影響する．近年主流となっている NUMA 構成のマシンでメモリバンド幅を活かすためには，タスクとデータが配置されるソケットをできるだけ一致させる必要があるが，ランダムワークスチーリングはこれを考慮しないため，メモリのバンド幅を十分に活かすことができない．

5.　ワークスチーリング戦略のカスタマイズ

この章では，4.3 節で述べた課題を解決するために実装したワークスチーリング戦略のカスタマイズ内容について説明する．

5.1　方針

ワークスチーリング戦略のカスタマイズは以下の二つの方針に基づいて行なった．

スチール粒度の向上　不均衡な木構造においてもできるだけ粒度の大きなワークスチーリングを行うために，各タスクに粒度を表す指標を付加して，それをもとに粒度の大きなタスクを選択してスチールする．各フェーズの処理は木を再帰的にたどるものであるため，粒度の指標としては，タスクが処理を行う四分木を根とした部分木のノード数とする．スチール粒度の向上によって，スチールの回数ならびにオーバーヘッドが減少するだけでなく，後述するハードウェアプリフェッチ機構に配慮したメモリ再配置がより効果的に機能することが期待される．

NUMA 構成への配慮　四分木の各ノードがどのワーカーで処理されたかを記録しておき，それをもとに，あるワーカーがワークスチーリングを試みる際は，できるだけ直前のフェーズにそのワーカー上で処理されたノードを利用するタスクをスチールするようにする．さらに，定期的に木構造の各ノードをワーカーに対応する CPU ソケット上のメモリに再配置することで，ソケットをまたぐメモリアクセスの頻度を削減し，NUMA 構成のマシンにおけるメモリアクセスの効率を高める．ハードウェアプリフェッチ機構を活用するために，メモリの再配置の際には，タスクの実行順に沿ってノードのメモリアドレスができるだけ連続になるように留意する．本稿執筆には，このメモリ再配置

機能の実装が間に合わなかったため，これは今後行う予定である．

あるワーカーがワークスチーリングを試みた際，粒度が大きいが直前に違うワーカーで実行されていたタスクと，粒度が小さいが直前にそのワーカーで実行されたタスクが同時にスチール可能になる場合など，これらの方針が競合することも考えられるが，その際は後者を優先する．タイムステップごとの四分木の構造変化はそれほど大きくないため，直前に行われたスケジューリングと同じ方針をとることによる性能劣化は少ないことが期待されるためである．

5.2 追加のデータ構造

```
1  struct Tree {
2    // タスク並列Tree−Based AMR のデータ構造
3    double T[2]; // 物理量
4    struct Tree *child; // 子ノードへのポインタ
5    double highest_diff; // 隣接格子点との差分の最大値
6    int level; // 細分化レベル
7    bool is_leaf; // 葉ノード (=格子点)であればtrue
8    // ワークスチーリング戦略をカスタマイズする
9    // ために追加するデータ
10   int n_nodes; // 自身を根とした部分木のノード数
11   int last_worker; // 最後に実行したワーカー
12 }__attribute__((aligned(64)));
13
14 // 各タスクに割り当てるヒント
15 struct ws_hint{
16   int n_nodes;
17   int last_worker;
18 };
```

まず，粒度の指標となる部分木のノード数と，最後にそのタスクに関する処理を実行したワーカーを格納するために，四分木のノードを表す構造体にそれらを表す整数 `n_nodes` と `last_worker` を追加する．また，各タスクに割り当てるヒントも同様に，`n_nodes` と `last_worker` からなる構造体として定義している．

上にデータ構造の宣言部の疑似コードを示す．なお，構造体のメンバ数自体は増加しているが，false sharing を避けるために構造体のサイズはキャッシュラインのサイズで切り上げられているため，追加する前後で実際に確保されるデータの大きさは変化していない．

5.3 スケジューリングヒントの割り当て

```
1  void traverse_tree (Tree *t){
2    // ヒントを確保,初期化してタスクに割り当てる
3    struct ws_hint hint;
4    hint.n_nodes = t−>n_nodes;
5    set_task_hint(&hint,sizeof(hint));
6    // last_worker を更新
7    t−>last_worker = get_worker_num();
8    // 子ノードを並列にたどるタスクを作成する
9    for (i=0;i<4;i++){
10     // child[i+1]が盗まれた後最初に実行されるので
11     // それが実行されたワーカーをヒントに指定
12     if (i<3)
13       hint.last_worker = t−>child[i+1].last_worker;
14     spawn_task(traverse_tree,t−>child[i]);
15   }
16   // 終了待ち
17   sync_tasks;
18   // ヒントを解除
19   set_task_hint(NULL,0);
20   /*
21    * check, balance, ghost フェーズでは,
22    * ここでさらに再帰的にタスクを作成する
23    */
24   ...
25   // 最後にノード数を更新 (Calc フェーズのみ)
26   t−>n_nodes = 1 + t−>child[0] + t−>child[1]
27     +t−>child[2] + t−>child[3];
28 }
```

各フェーズで再帰的にタスクを作成して四分木をたどる部分の擬似コードを上に示す．子タスクを作成する直前にローカル変数としてスケジューリングヒントを確保して，対応する四分木のノード数をヒントに書き込んだ後タスクに割り当てる．次に，ノード構造体の `last_worker` メンバに現在タスクを実行しているワーカーの ID を書き込む．

子タスクが生成される際には，現在のタスクが他のワーカーにスチールされて，別のワーカーで次の子タスクが生成される．したがって，ヒントの `last_worker` メンバにはタスクが作成される直前に，次に作成されるタスクがもつ子ノードの `last_worker` を書き込む．

さらに，タスクが終了する直前に，子のノード数をもとに，自身を親とする部分木のノード数を更新する．これはどのフェーズでも行うことができるが，各フェーズにこの処理を追加してオーバーヘッドを評価した際，最も性能低下の度合いが小さかった calc フェーズでのみ行うものとした．

なお，**check**，**balance**，**ghost** の 3 つのフェーズでは，子に対して再帰的にタスクを作成して処理を行なった後，ノード間の接する部分を対象にしたタスク並列処理を改めて行う．ここにもヒントを割り当てることが可能だが，この部分の再帰はノード間の接する部分のみをたどるものであるため，部分木全体のノード数は正確な粒度の指標にはならないと考えられる．そのため，現在の実装ではこれらのタスクを作成する前にヒントの割り当てを解除している．今後の課題として，この部分にも適切な指標を割り当てることについても検討を行っている．

5.4 ワークスチーリング関数

```
1  int n_candidates;// 候補ワーカー数
2
3  // ワークスチーリング関数: ワーカーがアイドル状態
4  // になった際に呼び出される
5  void mysteal(int id)
6  {
7    task_handle t_stolen;
8    // スチール対象の候補を決める
9    int candidates[n_candidates]
10     = get_random_workers();
11   // ヒントを読み取る
12   struct ws_hint hints[n_candidates]
```

```
13      = readydeque_peek(candidates);
14    for (int i=0;i<n_candidates;i++){
15      int target;
16      // ヒントとワーカーID が一致する
17      // タスクの中で最もノード数の
18      // 大きいものを選択
19      target = pick_the_best_candidate(id,
20        candidates, hints);
21      // ワークスチーリングを試行
22      t_stolen = readydeque_trysteal(candidates[target],
23        steal_confirm, &hints[target]);
24      hints[target].n_nodes=−1;
25      if (t_stolen)break;
26    }
27    return t_stolen;
28  }
29
30  // 確認用関数: ワークスチーリングが成功することが
31  // 確定した際呼び出される
32  int mysteal_confirm(task_handle_t stolen,
33    void *userdata)
34  {
35    struct ws_hint *hint_read = userdata;
36    struct ws_hint *hint_stolen
37      = get_hint_data(stolen);
38    // ノード数が読み取ったものより
39    // 小さければキャンセル
40    if (hint−>n_nodes > hint_stolen−>n_nodes)
41      return 0;
42    if (get_worker_num() == hint−>last_worker){
43      // 現在のワーカーID と一致する
44      // タスクを盗んだ場合は
45      // last_worker も比較
46      if (hint−>last_worker
47        != hint_stolen−>last_worker)
48        return 0;
49    }
50    return 1;
51  }
```

ワークスチーリングを行う手順を上の擬似コードに示す．まず，複数のワーカーからワークスチーリング対象をランダムに選択して，それらからヒントを読み取る．次に，それらのヒントを比較して，ワーカーの ID とヒントに保存されている ID が一致するタスクに対して，粒度が大きい順にスチールを試行する．ワーカーの ID とヒントに保存されている ID が一致するタスクがない場合は，単に粒度が大きい順にスチールを試行する．ノード間の接する部分を再帰的にたどるタスクはヒントの情報をもたないため，これらのタスクは粒度が 0 であるものとして，最も低い優先度でスチールする．

ワークスチーリングの確認関数では，実際にスチールしたタスクと以前に読みだしたヒントのタスクの粒度を比較して，想定よりも小さい粒度のタスクを盗んでしまった場合はこれをキャンセルする．また，ワーカーの ID とヒントに保存されているワーカーの ID が一致する場合は，これについても確認して，一致しないならばキャンセルする．

6. 評価

表 1 に示す共有メモリ環境を用いて，5 章で述べたワークスチーリング戦略のカスタマイズに関する評価を行なった．

表 1 評価に用いた環境
Table 1 Experimental Setup

CPU	Opteron 6172 (2.1GHz) 4 ソケット 12 コア × 4 = 48 コア
メモリ	128GB (32GB × 4 ソケット)
キャッシュ	L1D: 64KB/コア L2: 512KB/コア L3: 12MB/コア
OS	Linux 2.6.32 (Debian)
C コンパイラ	GCC 4.7.2
最適化オプション	-O3
numactl パラメータ	`--interleave=all`

表 2 移流計算の問題設定
Table 2 Problem Setting

空間の大きさ	(1.0,1.0)
速度場	(0.1,0.1)
境界条件	周期境界条件
CFL 数	0.4 (最も細かい格子)
タイムステップ数	500
細分化レベル	最小 8 (256×256) 最大 19
細分化閾値 K_{high}	0.1
集約化閾値 K_{low}	0.05
木のリーフあたりの格子点数	1
細分化を行う周期	毎タイムステップ
逐次再帰に移行する閾値	2048 ノード

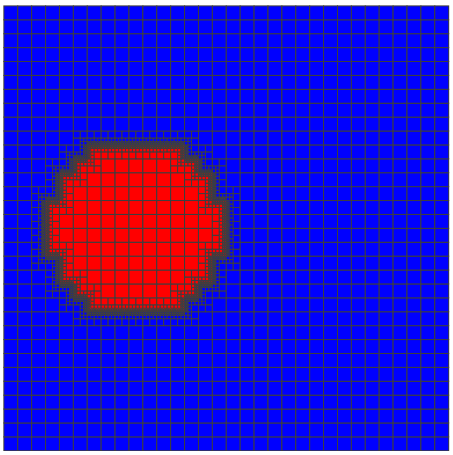

図 2 細分化レベルを最小 5，最大 8 とした場合の初期条件．赤い部分は物理量が大きい部分，青い部分は物理量が小さい部分である．
Fig. 2 Initial Condition (refinement level=[5,8]). Red parts have high value and blue parts have low value.

6.1 問題設定

計算に用いる問題には，二次元空間の移流方程式を**表 2** に示す設定のもとで解くものを用いた．初期条件には，円盤状に物理量が大きい部分が存在している，**図 2** のような物理量の分布を用いた．これは Langer ら [18] による Charm++[19] によって記述された AMR の実装の評価に用いられているデータセットを参考にして設定したも

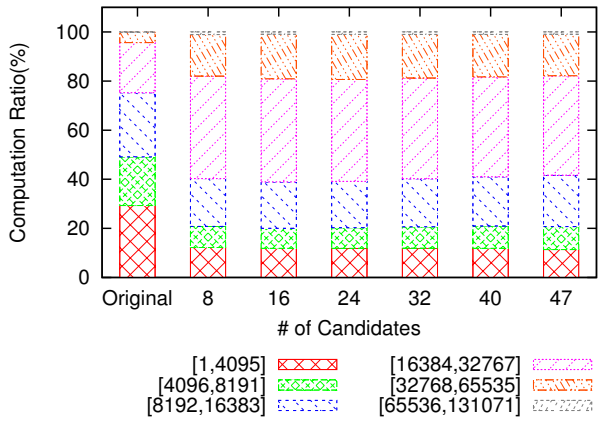

図 3　**Calculate** フェーズのタスクの粒度
Fig. 3　Computation Granularity on **Calculate** Phase

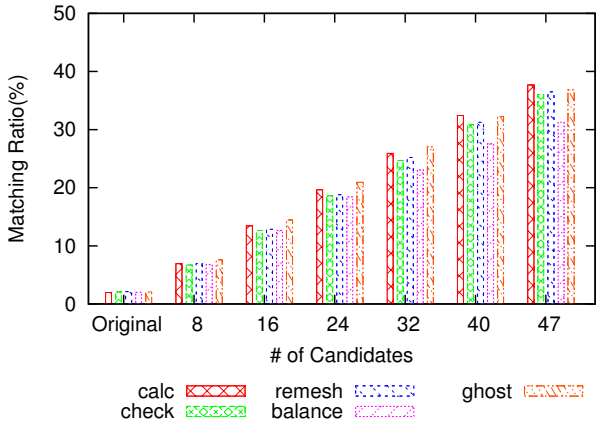

図 4　直前のフェーズと同じワーカーでノードの処理が実行される確率
Fig. 4　Worker Matching Ratio vs. # of Candidates

の[*2]で，物理量が大きい部分と小さい部分の境界部分のみに細分化が行われるため四分木が不均衡なものとなり，さらに境界部分が時間発展によって移動するため細分化や集約化が頻発するような設定となっている．

6.2　計算の粒度

ワークスチーリングのカスタマイズによるタスクの粒度向上の効果を確認するために，ワーカー数を 48 に固定して，ワークスチーリング対象の候補数を変化させた際に，計算がどの程度の粒度で行われているかを大まかに計測した．この計測は，**calculate** フェーズにおいてスチールされることなく連続で実行された部分木のノード数を 2 のべき乗間隔に集約したヒストグラムを作成して，それに対して区間の最小値をかけ，それぞれの粒度が全体に占める割合を大まかに計算することで行なっている．

得られた結果を図 **3** に示す (粒度が 1 ノードから 4095 ノードまでの区間は統合してある)．カスタマイズを行わない場合 (グラフ内の “Original” 以降のグラフにおいても同様) と比較して，カスタマイズを行った場合は大きい粒度の計算が占める割合が増加していることがわかる．**calculate** フェーズにおいては候補数の変化による粒度の変化はほとんどみられなかったが，他のフェーズにおいて粒度の傾向が同様になるとは限らないため，今後他のフェーズについても同様な調査を行う予定である．

6.3　直前のタイムステップとのワーカーの一致度

ワーカー数を 48 に固定して，ワークスチーリング対象の候補数を変化させた際の，木の各ノードの処理が直前のフェーズと同じワーカーで実行される確率を図 3 に示す．ただし，**check**，**balance**，**ghost** フェーズにおける，ノード間の接する部分のみを再帰的にたどる部分に関してはこれに含まれない．カスタマイズを行わない場合 (グラフ内の “Original”) は直前のフェーズと同じワーカーでノードの処理が行われる確率はおよそ 2%であるのに対して，カスタマイズを行った場合はその確率が 30%以上に増加している．

[*2] 細分化や集約化の方針，粗い格子から細かい格子への補間方法，四分木の葉あたりの格子点数，および細分化を行う周期が Langer らの評価と異なっている．

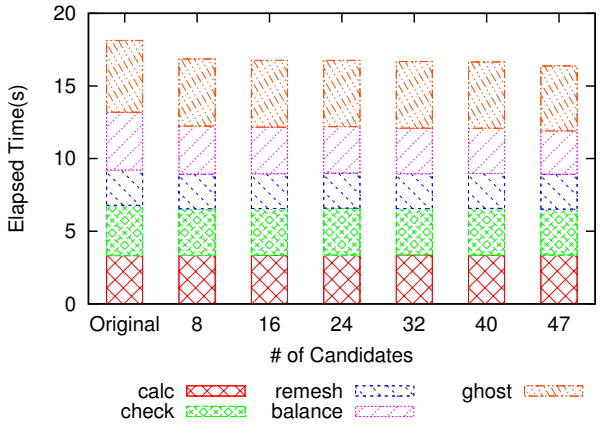

図 5　候補数を変化させた際の実行時間
Fig. 5　Execution Time vs. # of Candidates

6.4　フェーズごとの実行時間

ワーカー数を 48 に固定して，ワークスチーリング対象の候補数を変化させた際のフェーズごとの実行時間を図 **5** に示す．カスタマイズを行わない場合と比較して，カスタマイズを行なった場合は特に **balance**，**ghost** フェーズの実行時間が減少している．候補数の変化による実行時間の変化はランダムワークスチーリングとの差と比較してわずか (候補数 8 ワーカーの場合と 48 ワーカーの場合との差が 0.5 秒程度) ではあるが，候補数を増やすほど実行時間は短くなっている．

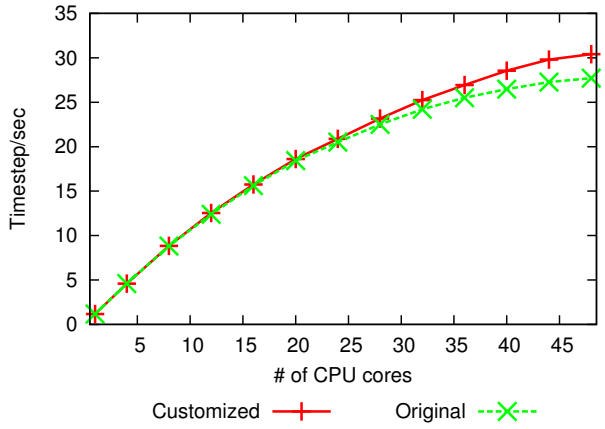

図 6　ワーカー数を変化させた際の実行性能
Fig. 6　Performance vs. # of Workers

6.5　実行性能

ワークスチーリング対象の候補数を [(ワーカー数) − 1] として，ワーカー数を変化させた際の実行性能を図 6 に示す．縦軸の単位は一秒あたりのタイムステップである．カスタマイズを行なったもの (グラフ内の “Customized”) はカスタマイズを行わないものと比較してワーカー数が多い部分で性能が向上しており，性能向上の割合が最も大きい 48 ワーカーの場合は単位時間あたりのタイムステップ数が 9.8%向上している．

NUMA に配慮したメモリ再配置機構がまだ実装されていないことから，この性能向上の理由は主にタスクの粒度が向上したことによると推測される．今後より詳細な解析を行い，性能向上の理由を明らかにする予定である．

7.　まとめと今後の課題

タスク並列処理を用いた Tree-Based AMR による移流計算の性能を向上させるために，部分木のノード数を指標として，スチールの粒度を向上させると同時に，NUMA 環境に配慮して直前に実行されたワーカーにできるだけタスクが配置されるようにワークスチーリング戦略のカスタマイズを行なった．

評価を行った結果，カスタマイズによってタスクの粒度，直前に実行されたワーカーに配置される確率の両方が改善していることが確認できた．また，48 コア使用時の実行性能も 9.8%改善された．

今後は以下の実装や評価を予定しており，これらを通じてタスク並列処理による Tree-Based AMR ならびに本稿で述べたカスタマイズ手法の有効性を明らかにする予定である．

- NUMA 環境に配慮したメモリの再配置を行う機構の実装および評価
- 今回得られた結果のより詳細な解析
- Langer らによる実装と同一条件での性能比較

謝辞　本研究は，JST CREST「高性能・高生産性アプリケーションフレームワークによるポストペタスケール高性能計算の実現」および JSPS 特別研究員奨励費 (課題番号 248391) の助成を受けて行われたものである．

参考文献

[1] Cray. The chapel parallel programming language. `http://chapel.cray.com`.

[2] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 519–538, New York, NY, USA, 2005. ACM.

[3] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 2, No. 3, pp. 264–280, 1991.

[4] Jun Nakashima, Sho Nakatani, and Kenjiro Taura. Design and Implementation of a Customizable Work Stealing Scheduler. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pp. 9:1–9:8, Eugene, Oregon, 2013. ACM.

[5] 河野瑛, 田浦健次朗. 分割統治法を用いたタスク並列 Tree-based AMR アルゴリズム. Master's thesis, 東京大学, February 2013. `http://hdl.handle.net/2261/54203`.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, Vol. 30, No. 8, pp. 207–216, 1995.

[7] 中島潤, 田浦健次朗. 高効率な I/O と軽量性を両立させるマルチスレッド処理系. 情報処理学会 論文誌 プログラミング (PRO), Vol. 4 No. 1, pp. 13–26, March 2011.

[8] Jun Nakashima. MassiveThreads: A Lightweight Thread Library for High Productivity Languages. http://code.google.com/p/massivethreads/.

[9] Quan Chen, Minyi Guo, and Zhiyi Huang. CATS: Cache Aware Task-Stealing based on Online Profiling in Multi-socket Multi-core Architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pp. 163–172, New York, NY, USA, 2012. ACM.

[10] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Work-stealing with Configurable Scheduling Strategies. In *Proc. of PPoPP '13*, pp. 315–316, 2013.

[11] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Proc. of ICPP '08*, pp. 586–593, 2008.

[12] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable Work Stealing. In *Proc. of SC '09*, pp. 53:1–53:11, 2009.

[13] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient Parallel Divide-and-Conquer. In *Proc. from the 6th International Euro-Par Conference on Parallel Processing*, pp. 690–699, 2000.

[14] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In *Proc. of PPoPP '01*, pp. 34–

43, 2001.

[15] 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. Technical Report 35, 東京大学大学院情報理工学系研究科, 東京大学大学院情報理工学系研究科, jul 2012.

[16] Marsha J. Berger and Joseph E. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Technical report, Stanford University, 1983.

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

[18] Akhil Langer, Jonathan Lifflander, Phil Miller, Kuo-Chuan Pan, , Laxmikant V. Kale, and Paul Ricker. Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012)*, New York, USA, October 2012.

[19] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pp. 91–108. ACM Press, September 1993.