タスク並列処理系における CPU の利用効率に着目したスケジューリング手法

大筒 裕之[†] 中島 潤[†] 田浦 健次朗[†]

一般にタスク並列処理系は，アイドル状態になったワーカスレッドが他のワーカスレッドからタスクを奪うワークスチールによって，動的に負荷分散を行なっている．しかし，多くのタスク並列処理系では，プロセス全体で実行可能なタスクの数がワーカスレッド数に対して少ない場合には，アイドル状態のワーカスレッドが成功しないワークスチールを行い続けてしまう．このような状態は CPU 資源を有効に利用できているとは言えない．

そこで本研究では，(1) スチール可能なタスクが存在しない場合はワーカスレッドを可能な限り速やかにスリープさせる (2) 新たなタスクが生まれたらスリープしているワーカを速やかに起こす，という 2 つの方針を採る．それにより，並列処理系の台数効果への悪影響を抑えつつ，余剰のワーカが CPU を消費しないようにワーカスレッドの挙動を管理する方式を提案する．また，我々が開発しているタスク並列処理系である MassiveThreads にこの手法を実装し，評価を行った．

Scheduling strategy focused on resource efficiency in Task-Parallel framework

Hiroyuki Odo ,[†] Jun Nakashima [†] and Kenjiro Taura [†]

Most task parallel frameworks balance loads dynamically by an idle worker thread that steals a task from another worker thread. In most frameworks, however, the idle worker thread fails in work-stealing when the number of runnable tasks is less than that of worker threads which leads to waste of CPU resources.

Our work uses following two strategies: (1) Make worker threads sleep as soon as possible when no task to be stolen exists. (2) wake up sleeping workers as soon as possible when new tasks are created. Our approach prevents idle worker threads from consuming CPU resources and keeps well load-balancing. We implemented this methods into our task parallel framework called MassiveThreads and evaluated its performance.

1. はじめに

1.1 背景

近年，排熱や消費電力の問題から，逐次計算の性能はピークを迎え[5]，計算の高速化は並列計算が主となってきている．計算機システムの大規模化も世界中で進み，並列アプリケーション記述の難化や，消費電力の肥大化が問題視されている．

並列アプリケーション記述の問題に対するアプローチとして，Cilk[2] や MassiveThreads[6] 等のタスク並列プログラミングモデルを用いた並列処理系の開発が進んでいる．タスク並列モデルでは，実行の最小単位をタスクとしてプログラムの実行中に動的に生成する．この時，計算の分割が不均一であっても負荷分散のための処理を記述する必要はなく，処理系によって自動的に負荷分散が行われる．

ワークスチールを用いて負荷分散を行う多くのタスク並列処理系では，アプリケーション全体で実行可能なタスクの数がワーカスレッド全体の数よりも少ない場合に，一部のワーカスレッドは成功しないワークスチールを行い続けてしまう．このようなワーカスレッドのビジーウェイトは，消費電力の無駄につながる． また，近年になって開発された Intel の Turbo Boost Technology[4] や AMD の Turbo CORE といった，使用されているプロセッサの数が少ない場合にはプロセッサの動作周波数を動的に上げる技術も考慮すると，実行可能なタスクが 1 つしかない場合に，計算に使うワーカスレッド以外の動作を止めることで計算しているワーカスレッドの動作周波数が上がり，性能向上の余地があると考えられる．

1.2 目的

本研究の目的は，並列計算を行なっている時に余剰のワーカスレッドが CPU を消費しないように，ワーカスレッドの挙動を管理する方式を新たに提案することである．

† 東京大学
the University of Tokyo

我々は，この問題を解決するためにタスクが存在しない場合には速やかにワーカスレッドをブロックし，新たにタスクが生成された時にワーカスレッドのブロックを解除するという方針を採る．それによって，台数効果や計算速度への悪影響を抑えつつ，有限の計算資源を有効に活用できると考える．

2. 関連研究

2.1 Work Stealing

タスク並列モデルでは，図 **1** のように各ワーカスレッドに対してタスクを格納するためのキューを設けている．それぞれのワーカスレッドが自分のタスクキューにアクセスする場合は，図 **2** の下部分のように，LIFO のポップと同じ動作をする．自身のタスクキューに格納されたタスクを全て計算を終えた時は，ランダムに選んだ他のワーカスレッドのタスクキューにアクセスして，タスクを奪ってくる．この，他のワーカスレッドからタスクを奪ってくる動作のことをワークスチールと呼んでいる．ワークスチールは，再帰的に作られるタスクの中で最も根元に近い部分のタスクを奪うため，図 2 の上部分のように FIFO でアクセスしている．Blumofe らは，ランダムなワークスチーリングが，strict な計算を効率良く実行できることを示している[3]．しかし本研究の場合は，ワーカスレッドがブロックする時に，タスクが残っていないか他のワーカスレッド全てを確認しなくてはならない．ランダムなワークスチーリングだけではこれを達成するのに「全てのワーカスレッドに順番にワークスチールする」方法よりも多くの回数を要するため，本研究ではこの部分に拡張を施す．

2.2 Power-aware application level scheduler

Araujo らは，プログラマにタスクの計算コストを指定させるものと，ヒューリスティックを用いて最大の計算コストを予測する手法を提案した[1]．更に，計算コストの小さいものを動作周波数を低く設定したプロセッサに割り当てる事によって，並列処理系の消費電力を抑えようとしている．

報告されている実験結果によると，ワーカスレッド 12 個に対して台数効果がおよそ 2.4 倍となっている．電力消費を抑えることには成功しているが，台数効果があまり高くなく，本研究が目標とする台数効果と低消費電力の両立ができているとは言えない．

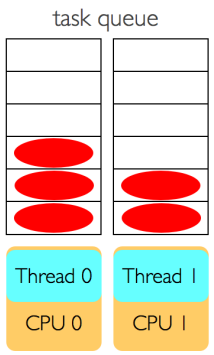

図 **1** タスク並列モデル
Fig. 1 Task-parallel model

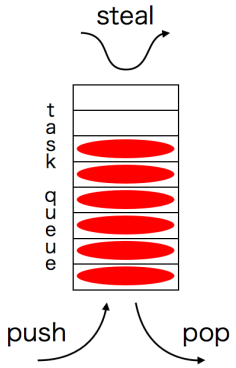

図 **2** タスクキュー
Fig. 2 Task queue

3. 提案手法

3.1 概　　要

多数のワーカスレッドがタスクを多く保持している状態ではランダムにワークスチールを行う事で台数効果を担保し，タスクが減ってきた所で，ワーカスレッド全体をチェックしてスリープさせる．このような戦略を採ることで，タスクの実行に関わっていないワーカスレッドをブロックし，高々タスク数だけのワーカスレッドが計算を行なっている状態を作り出す．

また，ブロックしているワーカスレッドが存在する時に新しくタスクを生成する場合には，そのブロックしているワーカスレッドが最初にワークスチールするターゲットを，タスク生成者に設定を行なう．設定後，ワーカスレッドのブロックを解除することで，解除されたばかりのワーカスレッドがターゲットを探す回数を減らすことが出来る．

これらの手法を用いることにより，ワーカスレッドは寝たり起きたりを繰り返しながら計算を実行する事になる．寝てから起きるまでには一定のオーバヘッドがあり，台数効果を損なう可能性がある．

3.2 アルゴリズム

本研究の提案手法は，1 度ランダムに選んだワーカスレッドに対するワークスチールに失敗した時，ワーカスレッド全体に対して順番にワークスチールを行い，タスクが無いことを確認してからスリープするというものである．

この節では，ワーカスレッドがスリープする直前の，ワーカスレッド全体をチェックする部分のアルゴリズムについて説明する．まず，全てのワーカスレッドを

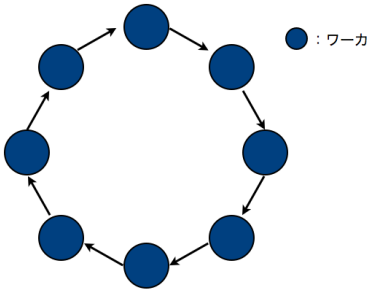

図 3　周回ワークスチーリング
Fig. 3　circle work stealing

対象にした線形探索をするために，図 3 のようにワークスチールする時のシーフ → ビクティムの関係を円周上に並べたものを考える．この時，ワークスチールに失敗した場合の次のターゲットは，図 3 において今回のターゲットが指しているワーカスレッドとなる．

我々の提案する手法では，線形探索中のシーフはワークスチール時，次のように動作する．

（1） 相手がタスクを持っている場合：通常通りワークスチールを行なう．
（2） 相手がタスクを持っておらず，ワークスチールを行なっている場合：ブロックする．
（3） 相手が唯一持っているタスクを計算中の場合：何もせず，次のワーカスレッドにワークスチールする．
（4） 相手がブロックしている場合：(3) 同様，次のワーカスレッドにワークスチールする．

(2) のように，他にワークスチールを行なっているワーカスレッドを見つけた時にブロックする狙いは，次の通りである．まず，ブロックしようとしているワーカスレッドを A，A が見つけたワークスチール中の別のワーカスレッドを B とする．両者は向きが一方向の周回ワークスチーリングを行なうため，A と B の間にあるワーカスレッドはタスクが無い事がわかっている．B から A までのワーカスレッドに関しては，B が周回ワークスチーリングでチェックを行えば全てのワーカスレッドに関するチェックが終わっている事が期待される．よって，A は B を見つけた時点でブロックすることで，ブロックするまでに A が行わなくてはいけない動作が少なくなり，本研究の狙いである「できるだけ早くスリープさせる」という点で改善される．

3.3 実　　装

今回，提案手法を実装する環境として，我々が開発しているタスク並列処理系の MassiveThreads を選択した．

本研究で提案する手法を，MassiveThreads のスケジューラに実装した擬似コードをアルゴリズム 1 に示す．以下では，ワーカスレッドの総数を N とする．各ワーカスレッドには通し番号が振ってあり，rank($< N$) は自分の番号であるとする．また，ws_hint は，タスク生成者がブロックしているワーカスレッドを起こす時に設定するワークスチールのターゲットである．

アルゴリズム 1：提案手法スケジューラ

```
void Scheduler_ex() {
 int victim; // ワークスチールのターゲット
 while(プログラム終了まで) {
   Task t = (タスクキューからポップ);
   if(t == null){ /* タスクキューが空 */
     victim = (ランダムに選ぶ);
     t = (ワーカ"victim"からワークスチール);
   }
   if(t == null) { /* ワークスチール失敗 */
     victim = rank + 1;
     while(t != null && victim != rank){
       t = (ワーカ"victim"からワークスチール);
       if(t == null){
         if("victim"がワークスチール中){
           worker_block(rank); //ブロックする
           t = (ws_hint からワークスチール);
         }
         else
           victim = (victim+1) % N;
   } } }
   if(t != null)
     execute(t); /* タスクが見つかったら実行 */
}
```

4. 評 価 実 験

4.1 実 験 環 境

評価実験には，表 1 に示した 2 つの共有メモリ構成のマシンを用いた．

表 1　実験環境
Table 1　Experiment environment

	マシン T	マシン M
プロセッサ	Intel Xeon E7540 freq: 2.0GHz physical: 24 cores logical: 48 cores	AMD Opteron 8354 freq: 2.2GHz physical: 32 cores logical: 32 cores
キャッシュ	L1D: 32KB/core L2: 256KB/core L3:18MB/socket	L1D: 64KB/core L2: 512KB/core L3: 2MB/socket
Turbo Boost / Turbo CORE	○	×

4.2 逐次性能への影響

はじめに，並列化を行わない単純な逐次プログラムについて，我々の提案手法がどのような影響を与えるのか調査した．実験には，再帰的にフィボナッチ数列の第 38 項を計算するプログラムを用いた．プログラ

ムの実行時間をマシン M とマシン T で計測した結果を図 4 に示す．

マシン M において，提案手法の実行時間が既存の物に対して最大で 1%程度短くなっていた．この原因としては，次のようなものが考えられる．既存の MassiveThreads においてシーフは，逐次計算を行なっているメインスレッドにワークスチールを行う毎に，メインスレッドの情報をキャッシュに読み込んでいる．そして，メインスレッドが自身の情報を更新する度に，その時点でメインスレッドの情報を持っているシーフのキャッシュに対して，キャッシュインバリデートを行わなくてはならない．提案手法を実装した MassiveThreads では，余剰のワーカスレッドはブロックしているためキャッシュインバリデートを行う必要がなく，これが実行時間の差として表れている可能性がある．

また，マシン T の方では実行時間がおよそ 22%短くなっている．こちらの原因は，MassiveThreads がデフォルトで論理コアの数と同じ数だけワーカスレッドを作っている事にあると考えられる．マシン T の CPU は，物理コア 1 つに対し，ハイパースレッディングの技術を用いて論理 2 コアとなっている． 今回の実験において，既存の MassiveThreads では，計算を行なっているメインスレッドと，タスクを持たないシーフが 1 つの物理コアに同居している．このことから，本来メインスレッドが計算に費やすべき時間の内，何割かがシーフの無意味なワークスチールに割かれてしまい，実際に計算にかかる時間よりも実行時間が遅くなってしまっていると考えられる．

また，傍証として，マシン T において，生成されるワーカスレッドの数を物理コアの数と同じ 24 個に設定して同様の実験を行った場合には，図 **4** のような大きな差は見られなかった．

4.3 台数効果への影響

続いて，細かいタスクを大量に生成するプログラムを用いて，負荷分散が上手く行われていることの調査を行う．実験にはフィボナッチ数列の第 n 項を

$$fib(n) := fib(n-1) + fib(n-2)$$

という再帰呼び出しを用いて求めるプログラムを用いた．ここで，ひとつひとつの再帰呼び出しをタスクとしている．また，使用する物理コアの数と台数効果の対比を調べるため，実験環境として，ハイパースレッディングの機能が搭載されていないマシン M を選択した．

実験によって得られた台数効果のグラフを図 **5** に示す．グラフから，ワーカスレッド 32 個の場合には

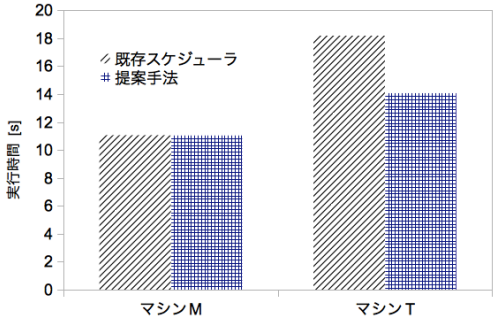

図 **4** 逐次計算の実行時間
Fig. 4 Execution time of serial application

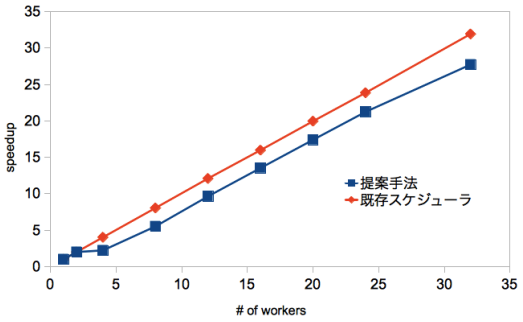

図 **5** フィボナッチ数列の並列計算の台数効果
Fig. 5 Speedup for the parallel fibonacci

台数効果がおよそ 13 %低下しているのが読み取れる．ワーカスレッド 2 個の場合には台数効果が変動しておらず，4 個になった所から低下している事から，ワークスチーリングのアルゴリズムに周回ワークスチーリングを組み込んだことで，ランダムなワークスチーリング単体だった既存の物よりも負荷分散が遅くなっているのが原因であると推察される．

4.4 消費電力の変化

最後に，マシン T に搭載されている IPMI(Intelligent Platform Management Interface) を用い，同マシンで消費電力の計測を行った．ワーカスレッドをスリープさせる場合とさせない場合の消費電力量の変化を調べるため，計測時には 4.2 で使ったものと同じように，フィボナッチ数列の第 n 項を逐次的に計算するプログラムを実行させた．

計測結果を図 **6** に示す．この図は，横軸にプログラムの実行時間，縦軸に IPMI による消費電力の最大値を示している．図 6 の色のついた部分の面積が，プログラムの実行全体での消費電力量を表している．

図 6 によれば，フィボナッチ数列を逐次計算するプ

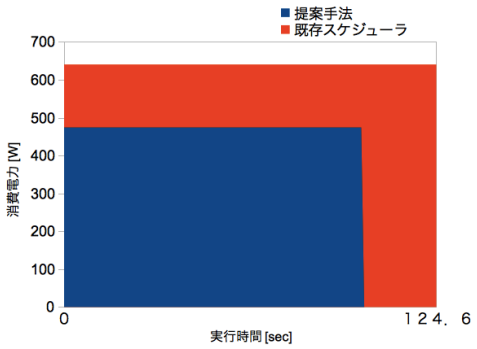

図 6　逐次計算時にかかる消費電力
Fig. 6　Power consumption of serial execution

ログラムにおいては，計算全体で電力の消費量をおよそ 40%削減することに成功している．これは，4.2 で述べたようにマシン T では HyperThreading によって物理コアの時間を奪い合いながら実行が進むため，既存の MassiveThreads では実行時間が長くなっているのが大きな原因となっている．今回は計測を行なっていないが，マシン M でも同様の実験を行った場合には，プログラムの実行時間は既存の MassiveThreads と提案手法を実装したものとでほぼ同じとなるため，削減できる電力の消費量はおよそ 26%程度であると期待される．

5. おわりに

5.1 まとめ

本研究では，タスク並列処理系 MassiveThreads を基盤として，台数効果への悪影響を抑えつつ余剰な CPU コアの使用を控えるタスクスケジューリング手法を実装した．我々の提案手法により，32 並列での台数効果の低下を 13%に抑えつつ，ワーカスレッドをコア数と同じだけ展開している状態での逐次計算時に消費される電力を最大およそ 40%削減することができた．

5.2 今後の課題

本稿では，全てのワーカスレッドを探索するために周回ワークスチーリングを使って線形探索を行なっているが，他の探索方法での実装を模索していく事が必要であると考えられる．

参考文献

1) Araujo, A. S. D., Camargo, C. A. D. S., Cavalheiro, G. G. H. and Pilla, M. L.: Towards a Power-Aware Application Level Scheduler for a Multithreaded Runtime Environment, *2010 22nd International Symposium on Computer Architecture and High Performance Computing Workshops*, pp. 43–48 (2010).
2) Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y.: Cilk: an efficient multithreaded runtime system, *SIGPLAN Not.*, Vol. 30, No. 8, pp. 207–216 (1995).
3) Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, *J. ACM*, Vol. 46, No. 5, pp. 720–748 (1999).
4) Dan huynh, Tyrone Hill, J. J. G. N.: Intel Optimized Turbo Boost Technology and Thermal Management (2010).
5) Ramanathan, R.: Intel Multi-Core Processors Making the Move to Quad-Core and Beyond(White Paper) (2006).
6) 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 (2011).