メモリ効率の良いスレッド生成のためのスタック領域のリンクリスト化

折笠 雄太朗　千葉 滋

本研究ではスレッド生成時にスタック領域を盗み出して割り当てる方法を提案する．スタック領域の盗み出しは，既存のスレッドのスタック領域の未使用部分を，生成するスレッドの実行に割り当てることであり，多数のスレッドが生成された場合でもスタック領域確保によるメモリ消費を抑えることができる．本提案手法ではスタック領域を連続した大きなメモリ領域にするのではなく，固定長の短いメモリ領域を連結したリンクリストにする．短いメモリ領域を使い切った時は gcc の split stack 機構を用いて検出し，新しいメモリ領域を連結する．従来提案されていた類似の手法，例えば StackThread/MP では，利用不可能なメモリ領域が大量に発生する場合がある．本論文の提案手法ではそのような状況でも利用不可能なメモリの量を抑えることができる．

1　はじめに

大規模な科学計算問題が与えられたとき，問題を均等に分割し並列に処理し，処理速度が CPU コア数でスケールさせる理想的である．しかし例えばアンバランスな木の全探索などでは，それは困難である．単純に木の 1 node に 1 スレッドを割り当てると全ての CPU コアを使い切れるが，スレッド生成・管理のコストが無視できない程大きくなりうる．このコストを嫌って，プログラマが自分でなるべくスレッドを作らずに，ロードバランスさせるコードを書くことは可能であるが，そのようなプログラムは実装が複雑になる．それよりも，ライブラリや言語処理系のスレッド機構が大量のスレッドを効率よくロードバランスさせながら実行するようにし，プログラマはそのスレッド機構を用いて，実装を気にせず多数のスレッドを生成するプログラムを書いた方が保守性の高いプログラムになることが期待できる．

上記のようなスレッド機構がすべき最適化の一つが，スレッド実行のためのメモリの消費量をなるべく少なくすることである．単純な実装では非常に多数のスレッドを同時に生成すると，各スレッドが用いるスタック領域が大量のメモリを消費してしまう．すぐに終了するスレッドや，ほとんど同期待ちしかしないようなスレッドが大量に生成されるようなプログラムでは，確保されたスタック領域がほとんど使われず無駄になってしまう．

そこで，多数のスレッドを生成したときにもスタック領域を大量に消費してしまわないように，スレッド生成時にスタック領域を盗み出して割り当てる方法を提案する．ここで言う「盗み出す」とは既存のスレッドのスタック領域の未使用部分を，生成するスレッドの実行に割り当ててしまうことを例えている．提案手法の実現のために，スタック領域を不連続な短いメモリ領域に分割し，各領域をリンクリストの様に連結して用いる．スタック領域のリンクリスト化は gcc の split stack 機能を利用し実装する．類似の手法として StackThreads/MP [4] があるが，この方法では，論理コア毎に連続した 1 本のスタック領域を用意し，それをコア上で実行される複数のスレッドが共有する．この手法では状況によっては，利用不可能なメモリ領域が大量に発生する場合があるが，本論文の提案手法ではそのような状況でも利用不可能なメモリの量を抑えることができる．

Yutaro Orikasa, Shigeru Chiba, 東京大学情報理工学研究科, Grad. School of Information Science and Technolgoy, The University of Tokyo.

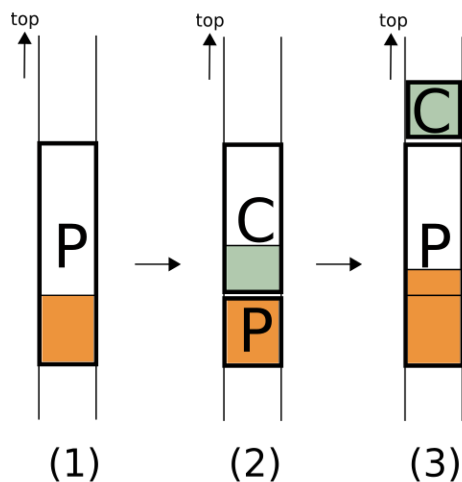

図 1　子スレッドのスタック領域の割り当て

2　大規模並列時のスレッドのスタック領域

並列プロセッサの普及により，性能を出すためにプログラマはスレッドを使った並列プログラムを書くことが求められている．しかし非常に多数のスレッドを同時に生成すると，各スレッドが用いるスタック領域が大量のメモリを消費し，場合によってはメモリ不足でプログラムが動作しなくなってしまう．とくに，スタック領域の大半は確保されてもほとんど使われないと考えられるので，各スレッドに大きなメモリ領域を割り当てるとメモリ効率が著しく悪化する．一方，各スレッドに割り当てるメモリ領域を小さくしすぎると，スタックオーバーフローでスレッドの実行が途中で中断してしまう危険性がある．

この問題を回避する手法のひとつが，スレッドに対するスタック領域の割り当てを遅延させる手法である．あるスレッド（親スレッドと呼ぶ）が新しいスレッド（子スレッドと呼ぶ）を生成したときには，子スレッド専用のスタック領域をすぐには割り当てず，親スレッドのスタック領域の未使用部分（現スタックポインタより上の部分）を使って子スレッドを実行する．親スレッドは他に空きコアができるまで停止する．例えば図 1 では，まず (1) 親スレッドがスタック領域 P のうち橙色の部分を使って動いているとする．ここで (2) 子スレッドが生成されると，スタック領域 P の未使用部分 C をスタック領域として子スレッドを実行する．

このようにすると，子スレッドの実行時間が短く，親スレッドの実行が再開される前に終了すれば，新たなスタック領域を確保する必要がなくなり，メモリ消費を小さくできる．一方，子スレッドが途中で同期のために停止した場合は，子スレッドが終了する前に親スレッドの実行を再開しなければならない．その場合，親スレッドの実行を再開する時点で子スレッドに専用のスタック領域を新たに割り当て，スタック領域の中身を旧領域から新領域にコピーして退避する．図 1 の (3) に示すように，子スレッドのスタック領域 C を元の元のスタック領域 P の外に移し，スタック領域 P を元の大きさに戻す．これにより親スレッドが自身のスタック領域を利用できるようにする．また子スレッドが実行再開後に新しいスタック領域を利用するように，停止した子スレッドのスタックポインタやフレームポインタを新しいスタック領域を指すように変更する．多数のコアをもつマシンなどで，子スレッドが同期のため停止する前に親スレッドの実行を別のコアを使って再開する場合は，タイマー割り込みなどを用いて子スレッドをいったん強制的に停止させる．

しかしながら，スタック領域の中身（図 1 のスタック領域 C の緑の部分）を旧領域から新領域にコピーすると，コピーの時間が実行時ボトルネックとなる可能性がある．また旧スタック領域内をさすポインタが存在する場合は，そのポインタが新領域をさすように，値を変更しなければならない．このため，この手法は C 言語には適さないという問題もある．この問題を避けるには，親スレッドと子スレッドを同時に動かさないこととし，親スレッドが一時停止して，子スレッドの実行が再開するときは，親スレッドのスタック領域の中身を退避して，子スレッドのスタック領域の中身を親スレッドのスタック領域へ復元する，という手法が考えられる．つまり図 1 の (3) でコピーしたスタック領域 C を再び元の位置に戻してもよいが，スタック領域のコピーのオーバーヘッドが非常に大きくなることが予想される．また多数のコアがある場合，この手法では多数のコアを有効活用できない可能性がある．

3　親スレッドのスタック領域の盗出し

我々は，多数のスレッドを生成したときにもスタック領域を大量に消費してしまわないように，子スレッドが親スレッドのスタック領域を盗み出す方式を提案する．本方式では，前章で説明した方式のように，子スレッドは親スレッドのスタック領域の未使用部分を使って実行される．しかしながら前章の方式と異なり，子スレッドはそのスタック領域を終了するまで親スレッドに返却しない．言わば盗んでしまう．

子スレッドが終了する前に親スレッドの実行を再開するときは，親スレッドは新しいメモリ領域を得て，それを既存のスタック領域にリンクリストのように連結して用いる．新しいメモリ領域として大きな領域は確保しないものとし，メモリの消費を抑える．図 2 に示すように，親スレッドのスタック領域 P の一部をスタック領域に使って子スレッドを動かす (2) までは図 1 と同じであるが，親スレッドの実行を再開する (3) では，子スレッドのスタック領域 C はそのままにして，親スレッドのスタック領域を追加で確保し，元のスタック領域と連結して用いる．リンクリスト状に連結されたスタック領域の管理には gcc の split stack 機能を用いる．Split stack は関数呼び出し時にスタック領域の残りサイズを計算し，不足が検出された場合に新たにメモリ領域を確保し，既存のスタック領域に追加する．提案方式では，各スレッドに一回に割り当てるスタック領域を小さくし，全体としてメモリの消費量を抑える．例えば図 2 の (1) で親スレッドのスタック領域 P は，図 1 に比べて小さくなっている．

3.1　アルゴリズム

提案する方式では，スレッドの切り替えはレジスタの退避によって行い，スケジューリングには work steal アルゴリズムを利用する．その上で，スタック領域の管理にスタック領域の盗出し法を用いる．

スレッドを生成すると親スレッドは一停止する．このとき thread entry と呼ぶデータ構造を作ってレジスタ等を退避し，生成する子スレッドの実行を開始する．作成した thread entry は論理コアごとの deque に入れて管理し，スケジューリングをおこなう．親スレッドの一時停止時に退避した entry は親スレッドを実行していた論理コアの deque の末尾に push する．子スレッドが終了したとき，親スレッドの実行が一度も再開されていない場合は，親を deque から取り除き，親の実行を再開する．そうでない場合は以下のように work steal スケジューリングを行う．各論理コアは自分の deque の末尾から thread entry を取り出す．もし，自分の deque が空なら，空でない他の論理コアの deque の先頭から thread entry を取り出す．論理コアは，取り出した thread entry 中の退避されたレジスタなどを復元し，そのスレッドを再開する．

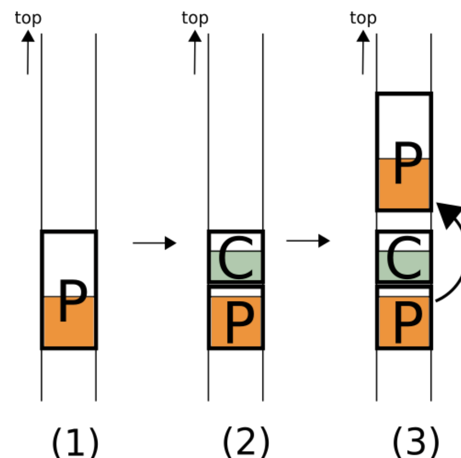

図 2　親スレッドのスタック領域の盗みだし

子スレッドのスタック領域の割り当て (盗出し) は，本方式では以下のように行う．子スレッドのスタック領域には，親スレッドのスタック領域の上方の余りを使う．上方とは，スタックの成長方向のことを指す．上方の余りとは，親スレッドのスタックポインタ (以下 sp) からマージン領域のサイズ分進んだ位置から，親スレッドの元のスタック領域の上限の間の領域のことである．マージン領域は後述する split stack の動作のに使用するスタック領域である．子スレッドのスタック領域の開始位置が決まったら，その位置を親スレッドのスタック領域の新しい上限とする．親が再開する前に子が終了した場合は，親のスタック領域の上限を増やし，子が使用していたスタック領域を親に返

却する．親が再開した後で子が終了した場合も子が使用していたスタック領域は親に返却するが，それは後述するように親のスタック領域が縮んで，子スレッドが使用していた領域と連続領域となったときである．

子スレッドの実行中に親スレッドの実行を再開するときは，新しいメモリ領域を確保し，それを短くなった既存のスタック領域と連結して利用する．短いスタック領域を連結して一つのスタック領域として用いるために，gcc の split stack 機能を用いる．この機能により，関数呼び出しのたびに，呼び出す関数の実行に必要なスタック領域の大きさ (以下 *frame-size*) と，スタックポインタ sp とスタック領域の上限の間の大きさを比較する．比較の結果，*frame-size* 分の領域が sp より上にある場合はそのまま関数を実行する．領域がない場合は，まずスタック領域の上限のすぐ上に終了した子スレッドのスタック領域があれば，その分だけスタック領域の上限を増やす（このとき子スレッドが盗んだスタック領域が親に返却される）．その上で再び *frame-size* と，sp とスタック領域の上限の間の大きさを比較し，*frame-size* 分の領域が sp より上にある場合は呼ばれた関数を実行する．領域がない場合，新たにメモリ領域を確保し，既存のスタック領域に追加して連結する．確保するメモリの大きさは，固定長の小さなものか，それが *frame-size* 未満なら，*frame-size* と split stack のためのマージン領域の和である．呼ばれた関数は追加されたメモリ領域をスタック領域の一部として実行される．

関数の実行が終了して戻るとき，その関数の呼び出し時に既存のスタック領域にメモリ領域を追加して連結していた場合，追加した領域をスタック領域から外す．外したメモリ領域の一部が子スレッドやその子孫のスレッドにより使用されていなければ，そのメモリ領域を解放する．

3.2　メモリ効率

本提案方式では，スタック領域を比較的小さい領域に分割して管理する．したがって多数のスレッドが生成され，その多くが短い時間で終了するようなプログラムの場合に，スタック領域のためのメモリを効率よく利用して実行できる．スタック領域のために全体として必要なメモリ量を小さく抑えられることが期待できる．

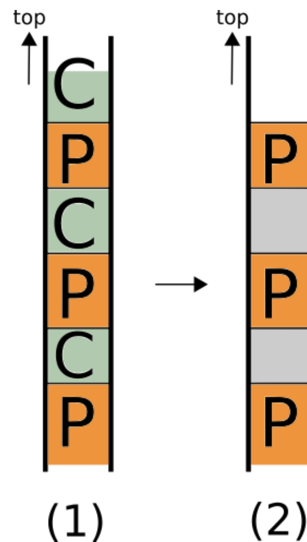

図 3　StackThreads/MP のスタック領域の管理

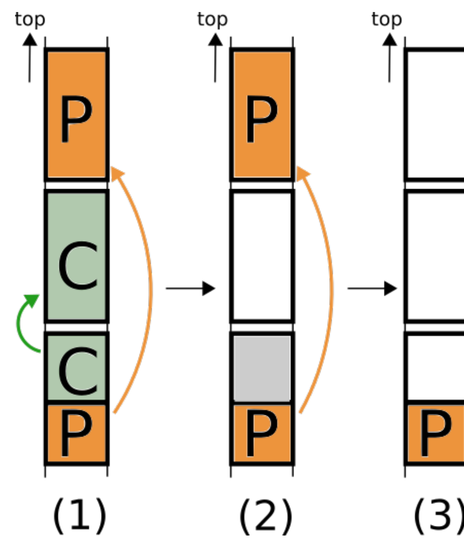

図 4　提案手法での歯抜けのスタック領域の扱い

類似の手法として StackThreads/MP [4] で用いられているスタック領域の管理法があるが，状況によっては利用されないメモリ領域が大量に発生する場合がある．

StackThreads/MP では，論理コア毎に連続した 1 本のスタック領域を用意し，それが同一コア上で実行

される複数のスレッドによって共有される．例えば，親スレッドと子スレッドが交互に実行された場合，それぞれのスタックフレームが 1 本のスタック領域上に交互に並ぶことになる．したがって子スレッドが先に終了した場合スタック領域上に多数の歯抜けの領域が図 3 のようにできてしまう．この灰色で示した歯抜けの領域は親スレッドが終了するまで再利用されない．StackThreads/MP では，短い細切れのスタック領域を連結して用いることができないからである．

一方，提案手法では，子スレッドと親スレッドが交互に実行された後に，先に子スレッドだけが終了した場合も，子スレッドがスタック領域として利用していたメモリの大半を再利用できる．子スレッドのスタック領域は，親スレッドから盗んだスタック領域と，新規に子スレッドが確保して占有するスタック領域の 2 つがある．子スレッドが占有する領域は，子スレッドの終了（正確にはその領域を確保した関数呼び出しの終了）と共に解放され，再利用可能になる．親スレッドのスタック領域から子スレッドが盗んだ領域も，親スレッドが既にそのスタック領域を利用していなければ，その領域全体を解放して再利用できる．親スレッドがまだそのスタック領域を利用中の場合は再利用できないが，そのような領域は小さい．なぜなら，子スレッドに盗まれた領域は，親のスタック領域の上方の未使用領域だけであるからである．

例えば図 4 のように，親スレッドと子スレッドのスタック領域が (1) のように配置されていたとする．P が親の，C が子のスタック領域を示す．スタック領域 C の下側の四角部分が，親スレッドから盗んだ領域である．ここで子スレッドが終了すると，(2) のように歯抜けの未使用領域ができるが，白いスタック領域は全体を解放でき，灰色のスタック領域だけが未使用のまま残る．この灰色のスタック領域は (3) で，親スレッドのスタック領域が縮み，スタック領域 P の下側の部分だけが残ったときに，親スレッドのスタック領域に連結されて以後使われる．

4 関連研究

StackThreads/MP [4] と本論文の提案方式の比較は既に 3.2 章で述べた．StackThreads/MP 以外にも，多数のスレッドを効率よく管理する手法の研究は数多くある．

Lazy Task Creation [3] は本手法で採用している work steal スケージューリング・アルゴリズムに加え，2 章で言及したスタック領域をコピーする方法を lisp 処理系上で提案している．Lisp ではスタック領域内を指すポインタは存在せず，スタック領域のコピー時にポインタ値の変更をする必要がないため，この手法は有効である．一方，本論文の提案手法は C 言語上での使用を想定しており，スタック領域をコピーしない．このためポインタ値の変更をする必要がなく，メモリをコピーするコストもない．

Cilk-5 [1] は work steal スケジューリングを行う並列計算言語である．本論文の提案手法と異なり，スレッドが作成されるごとにそのスレッドの実行に十分な大きさのスタック領域を割り当てる．例えば再帰的にスレッドが作られるようなプログラムでは，スタック領域が余っている状態で子スレッドを多数生成するので，使われていないスタック領域が大量に発生する．本論文の手法ではスタック領域の余っている部分を子スレッドに割り当てるので，よりメモリ効率がよい．

Tascell [2] は，ある論理コアが他の論理コアからタスクを要求されると，ある過去の時点へバックトラックしタスクを生成する仕組みの並列計算言語である．本論文の提案手法と異なり，親子のタスクが同時並行に動くことができないが，スレッドの生成と管理のコストが非常に小さい．

5 まとめ

性能を出すためにプログラマはスレッドを使った並列プログラムを書くことが，並列プロセッサの普及により求められている．しかし非常に多数のスレッドを同時に生成すると，各スレッドが用いるスタック領域が大量のメモリを消費してしまう．場合によってはメモリ不足でプログラムが動作しなくなってしまう．我々は，多数のスレッドを生成したときもスタック領域を大量に消費してしまわないように，子スレッドが親スレッドのスタック領域を盗み出す方式を提案した．

現状ではレジスタの退避及び復元処理と work steal スケジューリングの部分まで実装が完了している．現在は提案手法の実現のために gcc の split stack のランタイムライブラリ部分の改造を行なっている．実装は C++ 言語で行い，GNU/Linux 上で動作することを前提としている．

参考文献

[1] Frigo, M., Leiserson, C. E., and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, ACM, 1998, pp. 212–223.

[2] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T.: Backtracking-based Load Balancing, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, ACM, 2009, pp. 55–64.

[3] Mohr, E., Kranz, D. A., and Halstead, Jr., R. H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, New York, NY, USA, ACM, 1990, pp. 185–197.

[4] Taura, K., Tabata, K., and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, ACM, 1999, pp. 60–71.