遅延タスク生成の反復計算向け拡張

馬谷誠二[†] 八杉昌宏[†]
小宮常康[†] 湯淺太一[†]

不規則な問題を並列処理するのには，実行時のスレッド生成が可能なプログラミング言語が適している．そのような言語のなかには，スレッド生成時に他のプロセッサに授受可能なタスクの生成を行うように実装されていることも多い．しかし，この方法では実行時間のうちタスク生成にかかるオーバヘッドの占める割合が高くなってしまう．これに対し，Lazy Task Creation（LTC）を用いた実装ではアイドルなプロセッサが現れたときにはじめて，新しいタスクを生成することで必要のないタスク生成にかかるオーバヘッドを減らしている．さらに，タスクの分割を根本で行うことによりほぼ最小限のタスクの授受で各プロセッサに仕事（work）が十分に割り当てられる．本研究では，実行時スレッド生成を行う並列言語 OPA に LTC を導入する．また，OPA には明示的なループ構文が存在しており，そのような反復コードの実行中にスレッド生成を行うことも可能であるが，これを通常の LTC により実行させた場合，仕事を均等に分けることができず，そのためにタスク生成・授受の回数も抑えられない．本研究では，反復コード実行中には通常と異なる方法で LTC を行わせることにより，そのような状況においても効率の良い負荷分散を行えるようにした．

Extending Lazy Task Creation for Iterative Computation

Seiji Umatani,[†] Masahiro Yasugi,[†] Tsuneyasu Komiya[†]
and Taiichi Yuasa[†]

Programming languages which feature runtime thread creation are quite suitable for parallel processing of irregular problems. In their implementations, the creation of a thread often means the creation of a task which is a piece of work transferable to other processors. Such implementations, however, often incur a large overhead of task creation in the overall execution. In contrast, implementations using Lazy Task Creation (LTC) reduce the overhead by creating a new task only when some other processor become idle. The new task is extracted by dividing the present running task. Furthermore, by dividing it around the root, each processor can have sufficient amount of work with the almost minimum number of task transfers. In this study, we introduce LTC in parallel language OPA which features runtime thread creation. OPA has explicit loop syntax, and enables thread creation during a loop execution. If we use the usual LTC for such loop, we cannot divide work equally and the number of task creations and task transfers will become large. In order to obtain the LTC's efficient load-balancing even in such cases, we propose an extension of LTC to support iterative thread creation.

1. はじめに

実行時スレッド生成が可能な言語においては，データ並列的な規則的な処理だけでなく，探索問題や最適化問題といった不規則な並列処理も効率良く記述/実行することができる．

そのような言語の 1 つとして，オブジェクト指向並列言語 OPA [11] があげられるが，現在の OPA の実装においてはスレッド生成時にタスクの生成と割当てが行われている．

一般に，細粒度並列処理においてタスクの負荷分散にかかるオーバヘッドが大きくなると処理系全体の性能の低下につながる．特にすべてのプロセッサに十分な量の仕事（work）が割り当てられている状況での負荷分散にかかるオーバヘッドは明らかに無駄である．Lazy Task Creation（LTC）[4] はそのような状況におけるオーバヘッドを抑えることで効率の良い並列処理を可能にする．

本研究では，OPA でのスレッド生成を，LTC 方式による実装に変更することで効率の良い負荷分散を行わせることを目的とする．

また，反復実行コード中にスレッド生成が用いられ

† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

ているような場合には，通常の LTC でも計算以外のオーバヘッドの割合が高くなってしまうが，それについても十分な性能が出せるような対応を図る．

以下，2 章で LTC の概要，およびループ内でスレッド生成が用いられるコードを LTC により実行した場合に問題となる状況について説明する．

3 章で OPA 処理系の概要を述べた後，OPA における LTC の実装については 4 章で，スレッド生成を含むループへの対応については 5 章で述べる．

6 章で性能評価について触れ，7 章で関連研究との比較を行う．

2. LTC の概要

2.1 オリジナルの LTC を用いたタスク分割

プログラミング言語 OPA は，Java 言語に似た構文を持つ一方で次のような並列構文を提供する．

```
f(){
  ...;
  par obj.g();
  ...;
}
```

この場合，`f()` を実行中であったスレッドとは別に，`obj.g()` を実行するスレッドがつくられる．

従来の OPA における実装では，スレッド生成時にタスクを生成し指定されたプロセッサに割り当てている．この方法だと，プロセッサ間でタスクを授受するコストを払いすぎないで，アイドルなプロセッサが生じないように割り当てるのは難しい．

そこで「スレッド生成時にタスクを生成し，自プロセッサに貯めておき，アイドルなプロセッサが生じたら古い（仕事量が多い）タスクから盗ませる」ように改良することはできる．その場合，タスクを授受する回数は最小限に近くできるが，やはりスレッド生成ごとに（他のプロセッサがそのまま利用できる形の）タスクの生成が必要で，新しいスレッドの仕事を同じプロセッサ上の逐次の呼び出しで処理する場合と比べて，タスク生成のオーバヘッドがある．

これに対し，LTC[4]を用いるとスレッド生成時には以下のように動作する．

(1) `f()` を実行していたプロセッサは，現在実行中のスレッドの par 呼び出し以降の実行（継続）をスタックに積み，Lazy Task Queue（LTQ）に継続へのポインタを push してから自分は `obj.g()` 本体を実行する．ここで LTQ への push 以外は逐次呼び出しでも必要な動作である．

(2) アイドル状態になったプロセッサは，（何らかの基準で選んだ他のプロセッサの）LTQ に継続があれば，一番底の継続を盗み取る．

(3) `obj.g()` の実行中に push しておいた継続を盗まれたプロセッサは，`obj.g()` 終了後に自分がアイドルとなり他のプロセッサへ継続を盗みにいく．

(4) 継続が盗まれないまま `obj.g()` の実行が終了した場合には，自分で par 呼び出し以降の継続を LTQ から pop して実行を続ける．

この場合，LTQ の底から（仕事量の多い）継続を盗みタスク生成させることで各プロセッサに十分な仕事を割り当てており，プロセッサ間でのタスク授受の回数を減らしている．それに加えて，タスク生成もタスク授受が必要になってはじめて行われているので，オーバヘッドが少ない．

2.2 メッセージパッシング LTC

前節において説明したオリジナルの LTC は共有メモリ型並列計算機における実装を前提としており，アイドルなプロセッサは他のプロセッサの LTQ の底から勝手に継続を盗み取る．そのため，継続を盗むプロセッサと盗まれるプロセッサ間での LTQ へのアクセスの排他制御が必要になる．

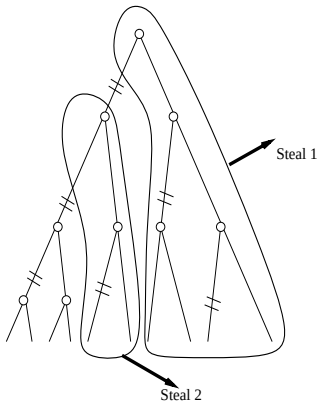
これに対し，メッセージパッシング LTC[1]は，分散メモリ型においても実現可能な LTC の実装である．メッセージパッシング LTC においては，アイドルなプロセッサは，他のプロセッサに対してタスクの要求メッセージだけを出す．盗まれてもよい継続を持っている各プロセッサは，定期的に polling[5] してタスクの要求を出しているプロセッサがいるかどうかを調べる．アイドルなプロセッサが見つかれば，自分で実行中のタスクから継続を抜き出したうえでそのタスクを生成し要求を出しているプロセッサに渡してやり，その後自分が実行中であったタスクを再開する．

この方式だと，LTQ の管理においてプロセッサ間での排他制御の必要がなく，また，関数フレームなどの表現についても自分以外のプロセッサには分からないようにしてもよい分，制約が少ない．さらに，スタックおよび LTQ をプロセッサにローカルなキャッシュ上におくことができるので，共有メモリ型においても性能の向上が図れる．

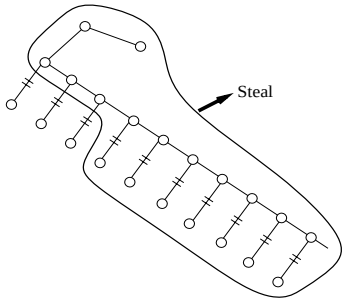
2.3 ループ内 par 呼び出し時のオーバヘッド

LTC による負荷分散は，本来，樹状再帰的なコードに対して効率良く動作する方式である．樹状再帰的なコードとして，たとえば，探索問題などで使われる

```
f(...){
  if(...){ ... }
```



(a) Tree recursive



(b) Loop

図 1 タスクスティール
Fig. 1 Task steal.

```
  else{
    par f(...);
    f(...);
  }
}
```

のような関数が考えられる．

par 呼び出し先と継続との間の仕事（work）の量に著しい差がないことを前提としており，その場合には盗みにきたプロセッサに渡す継続の持つ仕事の量は適度であるといえる（**図 1** (a)）．○は，1 つのメソッドの実行，あるいは，1 回の反復実行に対応している．○をつなぐ枝に横棒（=）がついているものは，par 呼び出しされることを表す．

一方，プログラミング言語 OPA には，Java や C と同様のループ構文が存在する．以下のようなコードを書いたとする．

```
for(int i = 0; i < N; i++)
  par g(i);
```

LTC でこのコード（for-par ループと呼ぶことにする）を動作させた場合に問題となるのは，par で呼び出す処理とその後ろの継続とでは仕事量が大幅に不均等になるということである（図 1 (b)）．たとえば，各反復において par 呼び出しされる処理が同程度の大きさであるとするなら，反復 $[i = 0]$ での par 呼び出しにおける残りの継続は $N - 1$ 回の反復であり，仕事の大きさの比率は約 $1 : N - 1$ となる．

切り分けたタスクの持つ仕事の量に差がありすぎると，仕事量が小さいほうのタスクの実行が大きいほうのタスクの実行に比べてすぐに終了してしまい，再びタスクスティールを起こす．この場合だと，反復回数だけのタスクスティールを引き起こしてしまいオーバヘッドが大きい．

本研究ではこのようなループ構文に対しても効率の良い負荷分散を行える方式を用意することで性能低下を防ぐことを図る．もともと LTC が実装されていた Scheme 処理系においても末尾再帰的なコードに対して同様の理由で性能が上がらないが，本研究で提案する方式は原理的にはそれらにも適用可能である．

3. 並列言語 OPA 処理系の概要

現在の OPA の実装においては，各メソッドは C の関数により実現されている．一般にあるプロセッサ上で無数の（細粒度）スレッドを動作させようとするときには，C のスタックをそのまま用いることはできず，関数の駆動フレームはヒープ領域などに確保される．しかし，その方式では，スタック上での実行に比べ実行効率が悪くなってしまう．

そこで OPA では可能な限りスレッドをスタック上で実行し，サスペンドしたスレッド☆が邪魔になった時点で，そのスレッドをヒープに退避させるという方法[3)]を用いている[11)]．

そのため OPA の場合，各メソッドに対してスタック上で実行中の場合とヒープに退避している場合とで，それぞれ異なった関数フレーム表現が用いられる．

あるスレッドがサスペンドしてヒープに退避する様子を**図 2** に示す．

ヒープ上のスレッドは呼び出し先から呼び出し元へのリンクでつながれた関数フレームのリストとして表現されることになる．実行再開可能なヒープ上のスレッドは，スケジューラによってリンクをたどりながら呼び出し先から順に実行される．

4. OPA における LTC の実装

3 章で述べたように，OPA では各スレッドは C の

☆ たとえば，オブジェクトへのアクセスを排他的にするために，他のスレッドによる使用が終わるのを待つ場合などにサスペンドする．

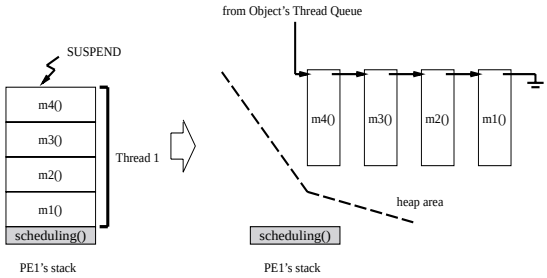


図 2 ヒープ上に退避しているスレッド
Fig. 2 Thread on heap.

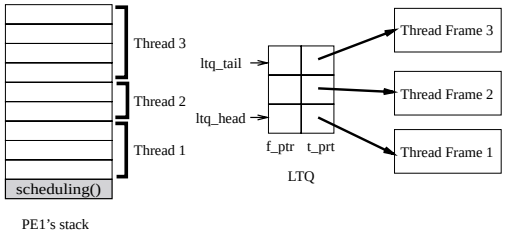


図 3 LTC における par 呼び出し
Fig. 3 par call with LTC.

スタック上で実行されている．そのため，盗まれてもよい継続がスタックの底のほうに存在していたとしても，他のプロセッサが勝手にその継続をスタックから抜き出して盗んでしまったのでは C のスタックが壊されてしまい，実行を続けることが不可能となってしまう．

そこで，C の上で LTC を実装するにあたり 2.2 節のメッセージパッシング LTC を用いれば，自分でスタックから継続を抜き出せばよく，他のプロセッサに壊される心配がない[2)]．

OPA での実装にあたり，スタックから継続を抜き出す作業，および自分が実行中であったタスクを再開する作業については，3 章のサスペンド機構が利用できることが分かる．以下，OPA での LTC の実装の詳細を説明する．

LTQ にはスレッドフレーム（スレッドを識別するために，スレッドごとにヒープに確保されており，スレッド固有の情報を格納している）を入れる場所が用意され，カレントスレッドのスレッドフレームは LTQ の末尾（`ltq_tail`）から参照される．

par 呼び出しにおいて，まず呼び出し元から呼び出し先へとカレントスレッドの切替えを行うために，`ltq_tail` を 1 増やす．

その後のメソッド呼び出しは逐次の呼び出しと同様にして呼ばれる（**図 3**）．

図 3 において，LTQ 中の `f_ptr` は，後述するタスクスティール時の動作においてスタック上のスレッドの巻き戻しを行う際，関数フレームのリスト表現へと変換されたスレッドへのポインタを格納する．図 3 では，まだ巻き戻されていないので，f_ptr の中身は `NULL` となっている．

その後，タスクスティールが起こらないまま呼び出し先の実行から return してくれば，`ltq_tail` を 1 減らして（カレントスレッドの切替え）呼び出し元メソッドの実行を続ける．

なお，起動される逐次呼び出し用コードにはスレッドの実行完了時の同期処理など☆は含まれていないが，呼び出し先がサスペンドすることなく実行完了することで，親スレッドよりも先に終了していることは保証されており同期処理の必要がない．

par 呼び出し先実行中，プロセッサはときどき `polling()` を実行することで，他にアイドルなプロセッサがいないか監視する．盗まれてもよい継続が LTQ にある場合にはタスクを欲しがっているアイドルなプロセッサがいるか調べ，もしいればタスクの抜き出し作業に入る．ただし，抜き出し作業の前にアイドルなプロセッサを獲得しておくことで，タスクを渡すのが自分だけであることを保証しておく．

タスクの抜き出し作業自体は，次のようにして行われる．

(1) タスクスティール中であることを表すフラグをセットし，これからサスペンドしなければならないという指示のもと `polling()` から return する．
(2) スタックを巻き戻しながら，各メソッドの関数フレームはヒープ上に退避されていく．
(3) par 呼び出しを行った箇所まで巻き戻ってきたら，スタックから退避したスレッドの実行再開の順序については何も保証されないので，呼び出し先スレッドを表す関数フレームのリスト表現の末尾に同期処理を行う `join_to` を付加する．そして，引き続き下のフレームを退避させるためサスペンドの指示のもとで return する．
(4) `scheduling()` まで制御が戻ってきたら，LTQ からスタックの底にあった継続を取り出して盗みたがっているプロセッサに渡す．
(5) (1) でサスペンドさせたスレッドから実行を再開する．

☆ OPA で生成したスレッドは動的スコープを用いて同期される[12)]．

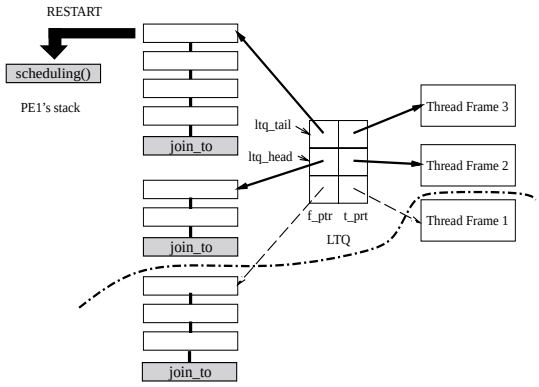


図 4 OPA 処理系におけるタスクスティールの実現
Fig. 4 Task steal implementation in OPA.

タスク抜き出し後の様子を**図 4** に示す．タスクスティール時の巻き戻し作業直後には，スタック上に存在した各スレッドはリスト表現に変換され LTQ の `f_ptr` から指される．その中から，`ltq_head` にあるリスト表現のスレッド（とスレッドフレーム）を他プロセッサに渡し（点線下部），`ltq_head` を 1 つずらす．

この方法では，スタックの巻き戻し作業自体はオーバヘッドとなる．ただし，一度でもヒープに退避させてしまえば，次に継続を盗ませる際には，ヒープにある関数フレームリストからの切り離しだけですむ．

5. ループ中の par 呼び出しへの対処

2.3 節で説明したように処理系全体が LTC のもとで動作している場合，for-par ループを実行中のプロセッサから継続をそのまま盗んでしまったのでは，仕事（work）を過剰に取りすぎることになる．

そこで，本研究における OPA の LTC 実装では，このような状況での継続の抜き出し作業については別の方法を用いることにする．

5.1 FORALL 型ループの分割

次のコードについて考える．

```
for(int i = 0; i < N; i++)
  par g(i);
```

この for-par ループだと，ループ制御変数およびループ回数についてはコンパイル時に解析可能である．また，ループ本体には par 呼び出ししかなく，かつ呼び出しに必要な情報に反復間での依存がない．すなわち反復間に並列性の存在する FORALL 型ループであるといえる．

FORALL 型の for-par ループを実行中，アイドルなプロセッサからタスク要求が来た場合，残りのループのうち半分は自分が続ける処理として残しておき，

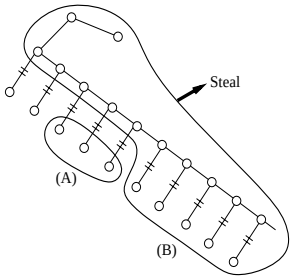


図 5 for-par ループ実行中のタスクスティール
Fig. 5 Task steal in for-par loop execution.

その先からの継続を渡してやればよい（**図 5** の A と B の大きさが同じ）．あるいは，複数のプロセッサからのタスク要求が来ていた場合，自分自身を含めたプロセッサ数でループを等分に分けることも可能である．

4 章で述べた LTC 時の動作に対する修正は次のようになる．

盗ませたい継続を抜き出すとき，先に制御変数 `i` の値を自分で処理する分だけスキップしてから抜き出す．元々その継続があった場所には，自分用に残しておいた範囲の `i` で for-par ループを実行するだけの新しいタスクをつなげておく．

その後，さらにアイドルなプロセッサからのタスク要求が来たときには，範囲 `i` を切り分けて，範囲が異なる以外は同じ処理を行う新しいタスクを生成して渡せばよい．FORALL 型ループにも各反復の仕事量に偏りのあるような不規則な処理は存在するが（たとえば，マンデルブロウ集合の計算），ここで述べている負荷分散では，ループの再分割を可能とすることでそのような処理を行うプログラムに対しても，適切な負荷の均等化を行うことができる．

なお，for-par ループ実行中は，（通常の LTC のように）`par` で呼ばれる先を実行中でなくてもループの切り分けが可能である．そのため，for-par ループ実行中であることを処理系に知らせるためのフラグが用意されている．

5.2 ストックモード実行

一般に for-par ループの本体には，任意の文が入っているかもしれない．次のコードを考えてみる．

```
for(int i = 0; i < N; i++){
  j = h(i, j);
  par g(j);
}
```

この場合，$i = I_1$ 回目の par 呼び出しに用いられる引数 `j` はループを I_1 回まわしてやらないと分からないし，`h()` の（逐次）呼び出しも $i = I_1$ 回目までは

先に実行が終了していなければならない．

また，次のようなコード

```
for(List xs = li;xs != null;xs = xs.cdr)
  par xs.car.f();
```

においては，逐次部分により次の要素を見つけてからでなくては par 呼び出しは不可能である．

一般に，for-par ループ中に par 呼び出し以外の文が存在する場合には，それらが引き起す副作用のためそれらの実行に先立って，par 呼び出しだけを取り出すことができない．また，制御変数に対する不規則な更新が行われている可能性もあり残りループ回数も定かではない．

そこで，このような場合における for-par ループの切り分けは以下のようにして行うことにする．

- 残りループ回数が分からないので，自分用に残す分（図 5 の A）はタスクスティールの回数を十分に少なくできる程度に多く，蓄える際にキャッシュからのあふれが問題にならない程度に小さくしておく．
- 他のプロセッサに渡す継続（図 5 の B）を抜き出す前に，A の逐次コードの実行を完了させておく．

そこで，for-par ループ本体を実行するタスクについて，

- par 呼び出し以外の文についてはそのまま実行．
- par 呼び出し文については，そこで実際に呼び出すのではなく，呼び出し自身を専用のテーブルにストックする．

という通常実行時とは異なる動作を行うモード（ストックモード）を用意する．タスクスティール時には，継続の抜き出しの前にストックモードでテーブルに par 呼び出しを貯めていけば，直前までに実行されていなければならない逐次コードをすませたうえでの A の par 呼び出しを貯めることができる．テーブルに貯め終われば通常モードに戻し，そこからの継続を B とすればよい．

なお，par 呼び出しに必要な情報に関して，単にテーブルに貯めるのではなく，共通する情報については何度も保存しないようにするなどの手法を用いることで par 呼び出しを蓄える処理の最適化が図れると考えられ，今後の課題としている．

その後，さらにアイドルなプロセッサからのタスク要求が来たときには，単に貯めてある par 呼び出しの半分を分けるだけですむ．

6. 性能評価

6.1 性能モデル

実際に計算機を用いて性能評価を行う前に，まず本節において，FORALL 型でない（すなわちコンパイル時に反復回数などの解析ができない）ような for-par ループであっても，通常の LTC で実行するよりストックモードで動作させる方が効率が良いことを，処理系の動作をモデル化して考察する．

樹状再帰コードを LTC で実行した場合，for-par ループを通常の LTC で実行した場合，および for-par ループをストックモードで実行した場合の実行時間を解析的に求めてみると次のようになる．

- 樹状再帰コードを LTC で実行した場合．
  仕事が十分に分割可能で，ほぼ均等に分割できるとすると，タスクの授受の回数（≒ タスクの生成の回数）は次のようになる．P をプロセッサ数，T_a を全仕事量（1PE での処理時間），T_1 を最小のタスクの仕事量とし，C を 1 回のタスク授受に要する時間とすると，タスクの授受の回数はおよそ $(P-1)\log_2((T_a/P)/(T_1+C))$ 回となる（この式を求める計算については，本節の主題からは少しはずれることもあり，ページ数の都合により割愛する）．
  たとえば，$P = 10$PE，$T_a = 10$ 秒，$T_1 = 0.001$ 秒，$C = 0.001$ 秒なら 80 回程度，1PE あたりの授受のオーバヘッドは (80 回 * 2PE * 0.001 秒 / 10PE) $= 0.016$ 秒程度で無視できる量である．
- for-par ループを通常の LTC で実行した場合．
  たとえば
  `for(`s_0`;`c`;`s_2`){`s_1`;par` p`;}`
  なら，ループの実行の最初と最後を除いては，s_2，c，s_1 と逐次処理してから，並列実行可能な p のスレッド生成をするというパターンが繰り返されている．
  まず，単一 PE 上で通常の LTC で実行した場合を考える．T_q を逐次部分（s_2，c，s_1）の処理時間，T_p を並列実行可能な部分（`par` p）の処理時間とすると，1 反復あたりの処理時間は $T_q + T_p$ となる．
  次に，複数 PE 上で通常の LTC で実行した場合を考える．T_s をタスクを盗むための処理時間，T_g をタスクを盗ませるための処理時間とすると，1 反復あたりの処理時間は $T_s + T_q + T_p + T_g$ となる．つまり，処理時間は $(T_s + T_g)/(T_q + T_p)$ の割合だけ増えることになる．T_g にはサスペン

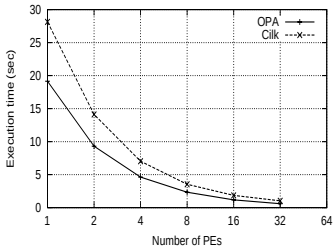


図 6 樹状再帰コードの実行時間
Fig. 6 Elapsed time for tree recursive.

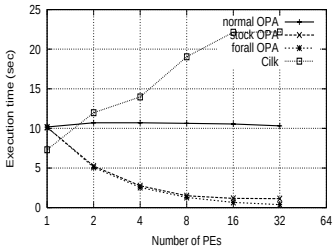


図 7 FORALL 型 for-par ループの実行時間
Fig. 7 Elapsed time for FORALL for-par loop.

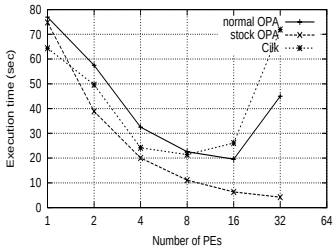


図 8 FORALL 型ではない for-par ループの実行時間
Fig. 8 Elapsed time for non-FORALL for-par loop.

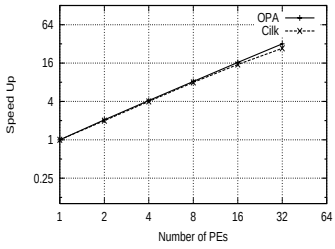


図 9 樹状再帰コードの台数効果
Fig. 9 Speed up for tree recursive.

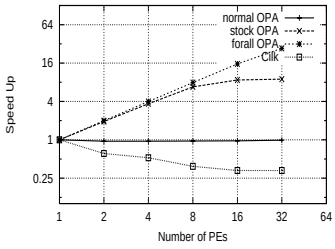


図 10 FORALL 型 for-par ループの台数効果
Fig. 10 Speed up for FORALL for-par loop.

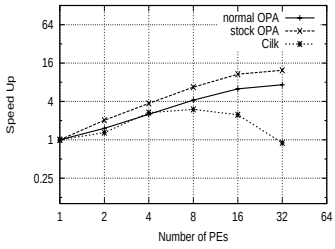


図 11 FORALL 型ではない for-par ループの台数効果
Fig. 11 Speed up for non-FORALL for-par loop.

ド・タスク生成・授受・実行再開などの時間が含まれる．さらに T_s には T_g の待ち時間に加えて，盗める継続が存在する状態でポーリングされるのを待つ時間も含まれる．その時間には，他のプロセッサがタスクを盗んだ場合の待ち時間も含まれるため，PE 数が増えると待ち時間も増えて，台数効果が得られない傾向があることが分かる．

- for-par ループをストックモード実行した場合．m 回の反復についてストックするものとし，T_o を par 呼び出しをストックするための処理時間，T_r をストックから取り出すための処理時間とすると，m 反復あたりの処理時間は $T_s+m(T_q+T_o)+T_g+m(T_r+T_p)$ となる．ここで，T_s には $m(T_q+T_o)$ というストックの完了を待つ時間も含まれる．m が十分大きく，また，「T_s に含まれる，盗める継続が存在する状態でポーリングされるのを待つ時間」を m で割ったものが無視できるとすると，1 反復あたりの処理時間は $2(T_q+T_o)+(T_r+T_p)$ となる．よって，$T_s+T_g > T_q+2T_o+T_r$ であれば，通常の LTC と比較し，ストックモードが有利と考えられる．ここで上記不等式の右辺は，すべて基本的には PE 間ではなく PE 内の処理に関するものなので，比較的小さいと考えられる．ここで，ストックしたものを他 PE に分けることも考慮する場合は，ある程度まとめて分けるときの 1 反復あたりのコストを上記不等式の右辺に加えればよい．

6.2 測定結果

本研究で実装した LTC により，いくつかのコードを Sun Ultra Enterprise 10000（Ultra SPARC II 250 MHz，10 GB Main Memory，1 MB L2 Cache，64 CPUs）上で実行し，実行時間を測定した．

また，それぞれ，同様のプログラムを Cilk[7] 言語でも作成し，その性能を比較検討した．

まず，樹状再帰的なコードとして Fibonacci 数列を再帰的に計算するプログラムに対し，LTC で動作させた場合の測定結果は，**図 6**，**図 9** のようになった．

この結果から，どちらの言語（処理系）でも LTC による負荷分散を用いて十分な台数効果が得られていることが分かる．また，1 台での実行において，Cilk では par 呼び出し時に，毎回，タスクスティール用の関数フレームをヒープ上に作成するため，1 台での実行において，そのオーバヘッドの分，OPA に較べて時間がかかっている．そのかわり，Cilk では，タスクスティール時に，OPA が行うようなスタックの巻き戻し作業を必要としない．

次に，$0 \leq x, y < 1000$ の範囲でマンデルブロウ集合を求めるプログラムの測定結果は**図 7**，**図 10** のようになった．このプログラムは，二重ループの最内側で par 呼び出しを行う FORALL 型 for-par ループとなっている．本論文で提案した方法を用いることで，仕事の分割量の大幅な不均等をなくすとともに，一般に行われている FORALL ループの単純なブロック分割とは異なり，一度分割したループの再分割も許しているので，マンデルブロウ集合のように反復ごとの仕事量に偏りがある場合でも，全体として負荷を均等化できている．

一方，通常の LTC 実装や Cilk においては，タスクスティール回数を抑えることができず，台数効果はまったく現れない．

次に，$N(= 1024)$ 個の粒子の集合に対して多体問題をシミュレートするプログラムの測定結果は**図 8**，**図 11** のようになった．このプログラムは各粒子について，他の粒子から受ける力の和を求めるフェーズと，そこから次の粒子の位置を求めるフェーズからなる．どちらも N 個の粒子への処理をループで記述しているが，粒子の集合をリスト表現により実現してあるので，リストの次の要素を見つける操作が必要となり，FORALL 型のように単純に分割できない for-par ループとなっている．

この場合にも，ストックモードで動かした方が，通常の LTC（OPA，Cilk ともに）よりも台数効果を得られることが分かる．

以上の結果から，通常の LTC では台数効果の上がらない for-par ループに対しても本論文の提案する手法により効率の良い負荷分散の行われていることが確認できた．

7. 議　　論

本研究ではループ中のスレッド生成を対象にして，効率の良い負荷分散を行える方法を提案した．

これは，並列化コンパイラなどが対象とする規則的なループだけでなく，ループ間での依存関係について解析時に対処できない場合，また，各反復実行の内容が著しく異なる場合のような不規則なループであっても対処できる方法である．

2.3 節で説明したように，

(a) `f(...){ par g(); f();}`

のような末尾再帰コードに対してもこの方法が適用できる．一方，

(b) `f(...){ par f(); g();}`

のようなコードでは，LTQ に残りの仕事が `g()` の実行だけの継続を次々と貯めていく．通常の LTC では盗まれる継続が小さすぎるため負荷の均等化が図れない．これは，切り分ける仕事（work）の量の大小関係が逆転しただけで for-par ループと同じ問題である．これについては，for-par ループに対しストックモードで自分のための仕事（work）を確保したのと同様に，LTQ にある程度（小さな）継続が貯まったところで，次の par 呼び出しを他のプロセッサに渡してしまえばよい．

par で呼び出す側を貯めておいて動的負荷分散を行うものとして SHE [8),9)]があげられるが，ループ内 par 呼び出しでは（仕事量の小さな）par 呼び出しを貯めすぎることになり LTC で (b) を実行した場合と同じ状況となる．

C 言語における LTC の実装を行っている他の言語としては Cilk[7)]がある．OPA との違いとして，Cilk ではスレッドのサスペンド時のヒープへの退避は行われておらず，また，LTQ には C のスタック上の継続と同じ継続をヒープの別の領域にも保存して直接盗めるようにしているので，余分なメモリアクセスのコストは必要だが，スタックを巻き戻したり polling したりしなくてもよい．また，Cilk 処理系では，LTC のスティール用にあらかじめ，par 呼び出し時にヒープにもスレッドがつくられる．

OpenMP [6)]では，for ループに対するプロセッサ間でのループの分割は動的なものも含めて行われている．しかし，各反復での仕事量に著しい違いのあるような不規則な問題については考慮されておらず，分割した部分ループに対する再分割は行われていない．再分割の行われているものとしては，文献 10) がある．ただし，文献 10) は LTC は行っておらず，スタックの一番上で実行中のループについての分割を行うものである．

8. ま と め

並列言語 OPA における従来の実装において，par 呼び出しによる実行時スレッド生成は，呼び出しと同時にタスクの生成・割当てを行うことで実現されていた．

これに対し，本研究では LTC 方式を OPA に実装することで必要のないタスク生成を抑え，プロセッサ間の負荷を均等に分けることを可能にした．

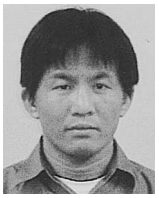
また，通常の LTC ではタスク生成数を抑えることのできないループ内での par 呼び出しについても，呼び出し以降の継続を全部渡すのではなく，その中から自分が実行するための仕事（work）を残しておくことで適切な負荷分散を図ることができる方法を用い，十分な性能向上が得られることを確認した．

謝辞 本研究の性能評価にあたり，並列計算機の使用を快く許可していただきました東京大学の平木先生に，つつしんで感謝の意を表します．

参 考 文 献

1) Feeley, M.: A message passing implementation of lazy task creation, *Proc. International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Halstead, Jr. R.H. and Ito, T. (Eds.), Lecture Notes in Computer Science, No.748, pp.94–107, Springer-Verlag (1993).
2) 田端邦男，田浦健次朗，米澤明憲：Cプログラムにおける Lazy Task Creation，情報処理学会研究報告 97-PRO-14（SWoPP'97），pp.79–84 (1997).
3) 田浦健次朗，米澤明憲：最小限のコンパイラサポートによる細粒度マルチスレッディング—効率的なマルチスレッド言語を実装するためのコスト効率の良い方法，情報処理学会論文誌，Vol.41, No.5, pp.1459–1469 (2000).
4) Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy task creation: A technique for increasing the granularity of parallel programs, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (1991).
5) Feeley, M.: Polling Efficiently on Stock Hardware, *Proc. 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, pp.179–187 (1993).
6) OpenMP: Simple, Portable, Scalable SMP Programming. `http://www.openmp.org/`
7) Frigo, M., Leiserson, C.E. and Randall, K.H.: The implementation of the Cilk-5 multithreaded language, *Proc. ACM SIGPLAN'98 Conference on Programming Language Design and Implementation* (*PLDI*), *SIGPLAN Notices*, Vol.33, No.5, pp.212–223 (1998).
8) Ito, T.: Efficient evaluation strategies for structured concurrency constructs in parallel Scheme systems, *Lecture Notes in Computer Science 1068*, pp.22–52, Springer-Verlag (1996).
9) Ito, T.: A Sound Parallelization Framework for Parallel Scheme Programming, *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, pp.3–40, World Scientific (2000).
10) Saito, H., Stavrakos, N. and Polychronopoulos, C.: Multithreading Runtime Support for Loop and Functional Parallelism, *Lecture Notes in Computer Science 1615, High Performance Computing, Second International Symposium* (*ISHPC'99*), pp.133–144, Springer-Verlag (1999).
11) 八杉昌宏，瀧 和夫：並列処理のためのオブジェクト指向言語OPAの設計とその実装，情報処理学会研究報告，Vol.96, No.82, pp.157–162 (1996).
12) 八杉昌宏：動的スコープの利用による並列言語の同期・例外処理の階層的構造化，情報処理学会論文誌：プログラミング，Vol.96, No.SIG 4(PRO 3), pp.44–57 (1999).

(平成 13 年 9 月 5 日受付)
(平成 14 年 2 月 13 日採録)



馬谷 誠二

1974 年生．1999 年京都大学工学部情報学科卒業．2001 年 3 月同大学大学院情報学科通信情報システム専攻修士課程修了．2001 年 4 月より同研究科同専攻博士課程に在籍中．並列/分散処理，プログラミング言語に興味を持つ．

八杉 昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年より京都大学大学院情報学研究科通信情報システム専攻講師．博士（理学）．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 会員．

小宮 常康（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年より同大学院情報学研究科通信情報システム専攻助手．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．平成 8 年度情報処理学会論文賞受賞．

湯淺 太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門」（共著),「C 言語によるプログラミング入門」,「コンパイラ」ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．