入れ子関数を利用する動的負荷分散と高水準記述

八杉昌宏[†] 小宮常康[††] 湯淺太一[†]

本論文では，GNU C コンパイラなどが C 言語の拡張機能として提供する入れ子関数を利用して，呼出し元の変数にアクセスすることで，遅延タスク生成に基づく負荷分散を行うプログラムが書けることを示す．また，バックトラックに相当する動作の記述も可能であることを示す．ただし本記述方式では，オリジナルの遅延タスク生成より低水準な記述が可能であり，タスクは継続を処理するものとは限らず，明示された並列実行可能部分を処理するものとする．このため，クラスタなどの分散環境でも利用できる．一方，低水準な記述が望ましくない場合もあるので，高水準な記述についても考察する．また，GNU C コンパイラの入れ子関数の生成・維持コストが少なくなるよう実装を改良した．共有メモリ型並列計算機上での予備的性能評価により，理想的な台数効果と低い並列化オーバヘッドが確認できた．

Dynamic Load Balancing by Using Nested Functions and Its High-level Description

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

In this paper, we show that we can write a program with "*Lazy Task Creation*"-based load balancing where callers' variables are accessed by using *nested functions* provided as an extension to C by the GNU C compiler. We also show that we can describe a behavior corresponding to *backtracking*. Our scheme accepts a lower-level description than the original LTC. A task can be created not only to process a *continuation* but also to process an specified part for parallel execution. The low-level description can be used for the distributed computing such as cluster computing. Since the low-level description is sometimes undesirable, we also discuss its high-level description. We also enhanced GCC to reduce allocation overhead and maintenance overhead of nested functions. The results of preliminary performance measurements on various shared-memory parallel computers exhibit near-ideal speedups and quite low parallelization overhead.

1. はじめに

探索問題などの樹状再帰的な細粒度並列計算においてプロセッサ間でタスクを授受して効率よく負荷分散を行うには，実行中のタスクから lazy に並列実行可能なタスクを分割・生成し抜き取れるようにした遅延タスク生成[6] (Lazy Task Creation) などの負荷分散方式が有効である．

ある量の仕事 (work) を転送可能な 1 個の形にしたものを**タスク** (task) と呼ぶ．タスクの分割・抜き出しはできるだけ呼出しの根元から行うことで，ほぼ最小限のタスクの生成・授受で各プロセッサに仕事が充分に割り当てられる．しかし，そのためには関数の呼出し中に通常はアクセスできない呼出し元の変数にアクセスできる必要がある．

本論文では，GNU C コンパイラなどが C 言語の拡張機能として提供する入れ子関数[1),8)] を利用して，呼出し元の変数にアクセスすることで，遅延タスク生成に基づく負荷分散を行うプログラムが記述できることを示す．また，バックトラックに相当する動作の記述も可能であることを示す．

本記述方式はオリジナルの遅延タスク生成より低レベルであり，分散環境でも利用できる．このため，複数の計算機，並列計算機内の複数のプロセッサ，プロセッサ内のマルチスレッドユニットといったさまざまなレベルで負荷分散を行うことが可能である．我々の方式では，休眠状態の計算資源に応じてタスクが生成されるので，並列度の過不足の問題に対処できる．

† 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
†† 豊橋技術科学大学 情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

論文の構成は以下の通りである．2 章では，GNU C コンパイラ (GCC) での入れ子関数について述べる．3 章では，本論文で例題として取り上げる，単純な樹状再帰的計算である Fibonacci 数の計算と，より複雑でバックトラックが望まれるペントミノ全解探索問題について述べる．4 章では，関連研究として動的負荷分散を行う言語処理系についてその高水準マルチスレッド言語，目的言語，実装手法について述べる．5 章では，Fibonacci 数の計算を動的負荷分散するために，提案方式ではどのような目的言語のプログラムにするのかについて述べる．また，ペントミノ全解探索を用いて一時的なバックトラックを伴う動的負荷分散について同様に述べる．6 章では性能評価を行う．ここでは，我々が進めている GCC の入れ子関数の改良による効果も示す．7 章では高水準記述について議論する．

2. 背　　景

2.1 入れ子関数のクロージャの利用

我々は，並列処理の実現に適した並行協調拡張 C 言語 XC-cube を設計・開発している．その特徴の一つは，入れ子関数であり，クロージャ生成コストや維持コストを抑えた拡張 C 言語の実装が進んでいる．

従来の C 言語では，関数呼出しを行っている間，同じプロセッサからであっても呼出し元に眠っている変数にアクセスすることはできない．このため，一度関数呼出しを行ってしまうと，呼出し中の間ずっと，呼出し元に戻った後の残りの処理について外部と協調した調整を行うことができない．入れ子関数は関数内で入れ子に定義される関数であり，定義により，入れ子関数本体とその定義時の環境を合わせて組とした**クロージャ**（closure）が生成される．入れ子関数の関数ポインタで生成したクロージャを利用することで呼出し元にアクセスする手段が提供できる．

入れ子関数を利用することで，プログラムに並行性・協調性を追加することが容易となり，本論文で述べる動的負荷分散の他，マルチスレッド[15]，マイグレーション・チェックポインティング，例外処理，GC でのスタックスキャンなども実現できる．

2.2 GCC での入れ子関数の仕様と実装

GCC は拡張機能として入れ子関数の機能を提供している[1),8)]．ローカル変数が宣言可能な場所で定義可能であり，その名前はローカル変数と同様に定義されるブロック内で有効である．入れ子関数は定義時点の環境に従って外側の関数の仮引数やブロックのローカル変数やラベル，他の入れ子関数にアクセスすることもできる．例えば，**図 1** において，関数 `f` は，`x` 等にアクセスする入れ子関数 `g` を持つ．入れ子関数のポインタを取得して入れ子関数を定義した関数（環境）の外で入れ子関数を呼び出すことができる．例えば，図 1 において，関数 `h` は，間接的に入れ子関数 `g` を呼び出したり，さらに入れ子関数 `g` へのポインタを他に渡したりできる．ただし，ローカル変数へのポインタと同様に，ブロック内の入れ子関数へのポインタもブロックの実行完了後に使用してはならない．

```
int f(int x) {
  int y = x * x;
  int g(int z) { return x + y + z; }
  return h(g, 0);
}
```

図 1　入れ子関数
Fig. 1　Nested functions.

GCC の入れ子関数の実装ではスタティックリンクを用いている．入れ子関数名の有効範囲で入れ子関数を呼び出す場合には，スタティックリンクを特別な追加の引数として，入れ子関数本体の処理を行うコード（命令列）を呼び出せばよい．

関数ポインタの取得において通常の関数の関数ポインタの実体は，その関数本体の処理を行うコードの先頭のアドレスと考えて良いが☆，入れ子関数の関数ポインタの実体を単なるコードのアドレスとするのは簡単ではない．入れ子関数の関数ポインタにより，入れ子関数本体と定義された時の環境の組であるクロージャが利用できなくてはならず，単なるコードのアドレスでこの組を表す必要がある☆☆．

GCC ではこの問題をトランポリン[1)] と呼ばれる機構を使うことによって解決している，トランポリンとは入れ子関数を定義した際のフレームポインタの値をスタティックリンク用のレジスタ（特別な追加の引数）にセットしてから入れ子関数本体のコードのアドレスへジャンプする数命令の短いコードのことであり，スタック上に動的に生成される．そのスタック上のトランポリンのコードのアドレスを関数ポインタとすることによって入れ子関数のポインタでクロージャを利用することができ，ほとんどのアーキテクチャにおいて動作する．ただし，コードを動的に生成することや，アーキテクチャによってはプロセッサの持つ命令キャッシュを明示的にフラッシュすることなどのオーバヘッドがある．

☆ ただし，評価で述べるように，PowerPC では通常の関数ポインタであってもコードと環境の組になっている．

☆☆ 通常の関数の場合も定義時の環境でグローバル変数等にアクセスできるが，その環境はコンパイルとリンクの際にコードに埋めこまれる．

3. 例　　題

樹状再帰的で不規則な細粒度並列計算として，本論文でとりあげる例題について説明する．

1 つは，定番の

$\mathrm{fib}(1) = \mathrm{fib}(2) = 1$

$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$ for $n > 2$

で定義される Fibonacci 数列 $\mathrm{fib}(n)$ の第 n 項を再帰的な計算で求めるものである．実用上の意味はないが，樹状再帰的で不規則な細粒度並列計算を行うものであり，また各関数の純粋な仕事がほとんどないため呼出しや並列処理に関するオーバヘッドなどの測定に向いている．つまり，この問題について低いオーバヘッドで負荷分散が可能ならば大抵の問題についてそれ以下のオーバヘッドでの負荷分散が可能である．ここでは，fib(3) すらタスク分割の候補とする細粒度計算を行う．

もう 1 つはペントミノパズルとして知られている探索問題をとりあげる．ペントミノとは 1×1 の正方形 5 つを辺同士で連結した 12 種類のピースのことで，例えば，十字型やコ字型のピースがある．ペントミノパズルは，このピースを 6×10 の長方形のボードに全て納める解を見つけるという問題である．ここでは，すべての解を求めるペントミノ全解探索を例題として用いる．一般に，この種のパズルはバックトラック法を用いて解く．つまり，ピースがボードにうまく置けるかどうかを判定し，うまく置けるようなら，配置データを書き換えて，さらに再帰的に探索を続けるが，その置き方についての探索を終えたら，配置データを書き戻してピースを取り除き元に戻す．しかし，このやり方は，配置データを単体プロセッサのみが変更することを前提としており，並列処理で行うのは難しい．多くの並列処理では，より深く探索を続ける度に配置データをコピーする必要があった．配置データコピーについては差分のみを保持するなどコピーを仮想的に行うこともできるが，そうすると逆に配置データへのアクセスが遅くなる問題があった．

4. 関 連 研 究

4.1 高水準マルチスレッド言語

高水準マルチスレッド言語としては，Scheme 言語に `future` 構文を導入した Multilisp 言語[4]，C 言語に `spawn` 構文などのスレッド機能を導入した Cilk 言語[3]，Java 言語に `par` 構文などのスレッド機能を導入した OPA 言語[13] などがある．

4.2 目的 (中間) 言語

目的言語として直接アセンブリ言語を用いるコンパイラも考えられ，オリジナルの遅延タスク生成[6] で用いられている．Cilk や OPA のコンパイラでは C 言語を目的言語とし，アセンブリ言語のコードを出力するための中間言語として用いている．ただし，一部の機能の高速化のために GCC[8] の拡張機能を利用している．我々の拡張 C 言語 XC-cube は C 言語ではうまく記述できない協調機能を C 言語に追加した言語で GCC と同様の入れ子関数などが利用できる．

C--言語[5),7)] は，C 言語ではうまく記述できないという問題を，C 言語を拡張するのではなく，アセンブリ言語へと近づけるというアプローチをとっている．我々が入れ子関数で提供しようとしている並行性・協調性の機能と同様に，マイグレーション・チェックポインティング，例外処理[7]，GC でのスタックスキャン[5] などが実現できる．また，stack walking を用いれば本論文で述べる動的負荷分散，あいるはマルチスレッド[15] も実現できると考えられる．

4.3 実 装 手 法

もともとの遅延タスク生成[6](LTC) は，論理的なスレッド生成を意味する `future` 構文の実行において，`future` の対象となる式の評価実行をそのまま行い，`future` の後の継続をスタックに保存し，この継続を他の idle なプロセッサがタスクとして盗むこともできるようにすることで負荷分散を行う方式である．継続は LTQ という deque で管理され継続が盗まれなかったときは自分でそのまま実行する．ここで，自プロセッサは LTQ の tail 側に対して継続の push/pop を行い，他のプロセッサは LTQ の head 側から継続を取り出してタスクとして盗む．このような手法は，基本的にはアセンブリ言語のプログラムでスタックを直接操作する必要がある．Cilk の処理系[3] でも遅延タスク生成を用いているが，C 言語を目的言語としていることから，継続を C のスタックとヒープに二重に保存している．C のスタックは C 言語が管理するものなので，出力する C のプログラムでは，ヒープ上にも継続を置くようにする．継続は，自分でそのまま実行するか，idle なプロセッサがタスクとして盗むかどちらか一方なので，相互排他が必要である．それにはロックを用いるか[6]，Dekker のアルゴリズム等を利用する必要がある[3]．

一方，メッセージパッシング型 LTC[2] では，idle なプロセッサは継続（タスク）を直接盗むのではなく，まずタスクの要求を行い，要求されたプロセッサが，自分で継続からタスクを生成して要求したプロセッサに返す．この方式では，相互排他が不要となる上，callee save レジスタに保存されている継続の情報も利用して，

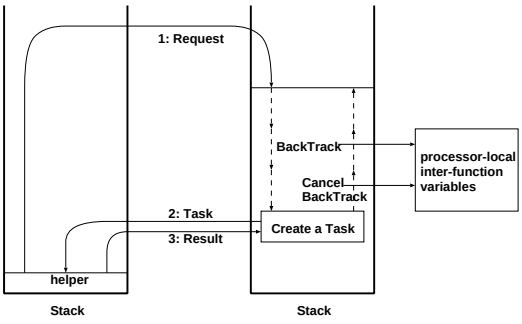

図 2 提案するタスクスティールプロトコル
Fig. 2 Proposed task steal protocol.

タスクの生成を行うことができる[10),14)]．また，OPA 言語の処理系における LTC[11)] のように，要求されてはじめて (より lazy に) 継続をヒープ上に作るようにすることもできる．

一方，継続ではなく，future 構文で論理的に生成されようとするスレッドを，lazy にタスクとして生成・授受する方法も考えられる．Leapfrogging[12)] はそのような方式である．この方式でもタスクは lazy に生成されるので，遅延タスク生成の一種と考えられる．ただし，未処理の継続ではなく未処理のスレッドからタスクを生成するという違いとなる☆．Leapfrogging では，未処理のスレッドをタスクとして盗まれたプロセッサはスレッドの結果を待つ状態になると，そのスレッドから派生したスレッドを盗み返す．これにより，逐次実行の場合より，スタックを余計に消費することはなくなる．

5. Lazy Backtracking Task Creation

入れ子関数を用いた負荷分散の実現方式を Lazy Backtracking Task Creation (LBTC) と呼ぶ．LBTC での記述は，マルチスレッド言語などと比べると低水準である．高水準記述については 7 章で議論する．

LBTC では，メッセージパッシング型[2)] であり，他の idle なプロセッサからのタスク要求をポーリングにて受け取る．タスク要求を受け取ると，**図 2** に示すように，呼び出し中 (実行中) の関数が持つ，入れ子関数を呼び出すことで，呼出し元に遡って，タスクが生成できる場所を見つける．つまり，各関数に入れ子関数を 1 つ持たせ，タスク要求を受け取ると，その要求を引数として入れ子関数を呼び出す．入れ子関数は，前回呼び出し元ではタスク生成されなかった場合を除き，まずは呼び出し元の入れ子関数を呼び出して，より root に近い位置でタスクが生成できないかを試みる．タスクが生成できると，図 2 のように送り返したあと，入れ子関数をリターンして，元の計算に戻る．

タスクの生成は将来する予定だった計算を一部切り取って渡すことにする．将来その部分を実行する際には，重複実行を避け，盗まれたタスクの結果が得られるまで待つことにする．ここで単に待つのは無駄であるので，他からタスクを盗むべきであるが，スタックサイズを考慮し，Leapfrogging のように待っている

☆ これらの中間的なものとして，Cilk などでは spawn の後から spawn の同期がとられるところ（sync）までの部分継続を盗むことも考えられる．

```
int fib(proc_env pr,
        int (*bkf)(req_buf_t *), int k){
  int s = 0;
  int restp = 1;
  int ret = 1;
  void **saved_req_port_p;
  int result_buf;
  int result_port;
  /* タスク生成のための入れ子関数 bk */
  int bk(req_buf_t *req_buf_p){
    if(ret) ret = bkf(req_buf_p);
    if(ret) return 1; /* 呼出し元で生成済 */
    if(restp){
      task_buf_t *tbp =req_buf_p->task_buf_p;
      int *tpp        =req_buf_p->task_port_p;
      restp = 0;
      /* save info for "request back" */
      saved_req_port_p =req_buf_p->req_port_p;
      /* result not available yet */
      result_port = 0;
      /* タスクを共有メモリに置く */
      tbp->f = tf_fib;
      tbp->sender = pr->myid;
      tbp->a.t1.result_buf_p = &result_buf;
      tbp->a.t1.result_port_p = &result_port;
      tbp->a.t1.k = k-2;
      /* そのタスクを転送する */
      finish_write_before_write();
      atomic_write_int(*tpp, 1);
      return 1;
    }
    return 0;
  }
  if(k <= 2) return 1;
  else {
    restp = 1;   /* 盗まれていない */
    POLL(pr->req_port, bk);
    s += fib(pr, bk, k-1);
    if(restp){   /* 盗まれたか */
      restp = 0; /* 盗ませない */
      s += fib(pr, bk, k-2);
    }else{
      /* 盗まれたタスクの完了を待つ */
      while(atomic_read_int(result_port) == 0)
        /* タスクを取り返して走らす */
        steal_run_task(pr,saved_req_port_p,1);
      start_read_after_read();
      s += result_buf;
    }
    return s;
  }
}
```

図 3 Fibonacci の動的負荷分散コード
Fig. 3 Load balancing code for Fibonacci.

```
int try_piece(proc_env pr, int (*bkf)(req_buf_t *),
              int k, int i){
  int s = 0;       /* the sum of solutions */
  int d;           /* direction */
  int n = ps[i].n; /* number of directions */
 /* タスク生成のための入れ子関数:
    一時的バックトラックしてから元の入れ子関数を
    呼び出し，その後，やり直す */
  int bk(req_buf_t *req_buf_p){
    int ret;
    int kk=k,l;
    int *pss = ps[i].pss[d];
    /* UNDO: remove the piece
       (temporary backtrack) */
    pr->a[i] = 0;            /* 使用可に戻す */
    pr->b[kk=k] = 0;   /* ボード b も元に戻す */
    for(l=0;l<4;l++) pr->b[kk += pss[l]] = 0;
    ret = bkf(req_buf_p);
    /* REDO: put the piece
       (cancel temporary backtrack)*/
    pr->b[kk=k] = i+'A';   /* ボード b に置く */
    for(l=0;l<4;l++) pr->b[kk += pss[l]] = i+'A';
    pr->a[i] = 1;            /* 使用済ピース */
    return ret;
  }
  for(d=0;d<n;d++){
    int kk=k,l;
    int *pss = ps[i].pss[d];
    /* room available? */
    for(l=0;l<4;l++)
      if((kk += pss[l]) >= 70 || pr->b[kk] !=0)
        goto Ln;
    /* DO: put the piece */
    pr->b[kk=k] = i+'A';   /* ボード b に置く */
    for(l=0;l<4;l++) pr->b[kk += pss[l]] = i+'A';
    pr->a[i] = 1;            /* 使用済ピース */
    /* find the first empty location */
    for(kk=k; kk<70; kk++)
      if( pr->b[kk] == 0 ) break;
    /* recursive search */
    s += search(pr, bk, kk);
    /* UNDO: remove the piece
       (backtrack) */
    pr->a[i] = 0;            /* 使用可に戻す */
    pr->b[kk=k] = 0;   /* ボード b も元に戻す */
    for(l=0;l<4;l++) pr->b[kk += pss[l]] = 0;
  Ln:
    continue;
  }
  return s;
}
```

図 4 Pentomino での一時的バックトラック
Fig. 4 Temporary backtracking code for Pentomino.

タスクを盗んだところから盗み返すようにしている．

タスク生成のための情報は，LTQ からではなく入れ子関数で呼出し元に遡って見つける．LTQ を使わず C のスタックのみで動作しており，タスクは lazy に，C のスタック内の情報から直接取り出される．これは，LTQ の管理コストの分だけ，LBTC が従来の LTC よりもさらに効率が良いことを示している．

プログラムの一部を**図 3** に示す．この `POLL` マクロでは，タスク要求があれば入れ子関数 `bk` を呼び出す．入れ子関数 `bk` は，関数呼び出し時に呼び出し先に引数で渡す．逆に呼び出し元の入れ子関数は `fib` 関数の引数 `bfk` で受け取っているが，入れ子関数 `bk` は，この入れ子関数ポインタ `bfk` を用いてより根本の呼び出し元でタスク生成できないかを試みている．入れ子関数 `bk` がタスク生成をすることになった場合は，将来計算する予定だった「`fib(k-2)`」をタスクとして生成し，要求元に回答することにしている．ここでは，共有メモリを仮定して同期変数への書き込みで回答を行っている．

また，ペントミノ全解探索問題のような問題に対しては，単純な樹状再帰的計算の動的負荷分散に加えて，タスクを生成するために遡る際に一時的にバックトラックする (配置情報に加えた変更を元に戻す) という機能が加えられる（図 2）．バックトラック探索の逐次プログラムでは，ピースをボードにある置き方で置くという選択をして探索を進める際にボードを表す配列を直接書き換え，バックトラック時には元に戻すことで，ボードを表す配列全体を探索を進める度にコピーするといったコストを避けられる．これを元に，一時的にバックトラック可能とするための入れ子関数を追加した並列実行用プログラムを**図 4** に示す．入れ子関数は呼出し元の入れ子関数を呼び出すが，その前に一時的に直接書き換えた効果を元に戻している．(ここではピースを取り除く) さらにリターン時には，元に戻した効果を打ち消すために，この場合，再度ピースを置きなおしている．これによってペントミノ全解探索問題では，配置情報のコピーをタスク生成時のみに限定することができる．

Cilk でも子の間で同じ領域を再利用可能とするための `SYNCHED` が用意されているが[9]，親子間はコピーしなくてはならない．また，SICStus Prolog では，並列実行時に束縛情報のコピーをさけるために，盗むプロセッサは盗まれるプロセッサに割り込みをかけて，スタック全体をコピーし，盗んだプロセッサ自身が必要なだけバックトラックをしてから盗んだ branch を実行するようになっている．

6. 評　　価

6.1 GCC の入れ子関数での評価

種々の共有メモリ型並列計算機上で，fib(36) の Fibonacci 数計算およびペントミノ全解探索について，負荷分散を行う並列プログラムを実行して測定した結果を示す．用いた共有メモリ型並列計算機の仕様/環境は以下のものである．

- Sun Ultra Enterprise 10000 （250MHz UltraSPARC-II, 1MB L2 Cache, 64CPUs）Solaris 7, gcc 2.95.2 `-O2 -mcpu=ultrasparc`

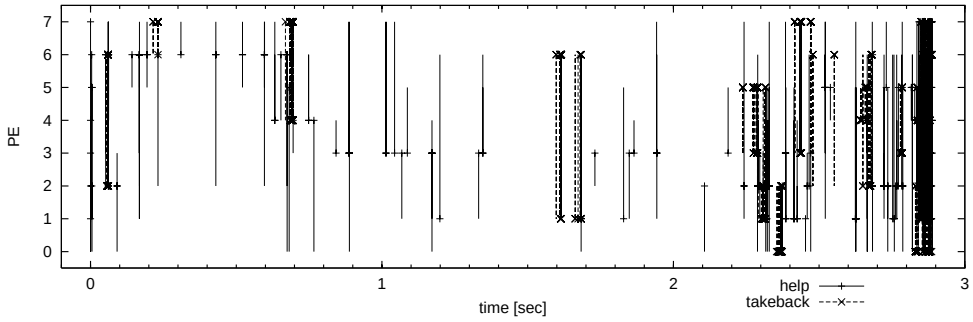

図 5 8 プロセッサ上のペントミノ全解探索でのタスク転送記録
Fig. 5 Record of task transfer by Pentomino search on 8 processors.

- IBM RS/6000 SP の 1 ノード（332MHz PowerPC 604e 命令 32KB/データ 32KB L1 cache, 0.25MB L2 cache, 4CPUs）AIX Version 4.3, gcc 2.95.2 `-O2 -mcpu=powerpc`
- PC（1GHz Pentium-III, 2CPUs）, Solaris 8, gcc 2.8.1 `-O2`

まず，並列プログラムの台数効果については，ほぼ理想的な台数効果が得られた．例えば Ultra Enterprise 10000 上では fib(36) は，48CPU の使用時に 47.8 倍の台数効果が得られた．これは，提案する負荷分散方式が十分に機能していることを示している．また，ペントミノ全解探索については，Ultra Enterprise 10000 上では 32CPU 使用時には，30.9 倍の台数効果，48CPU の使用時に 45.4 倍の台数効果が得られた．Fibonacci 数計算とくらべてやや落ちるのは，枝刈りの結果，せっかく授受したタスクの仕事量が少なすぎることもあるためと考えられる．ここで，**図 5** には，8 プロセッサ上のペントミノ全解探索でのタスク転送記録を示す．横軸が時刻で縦軸がプロセッサ番号である．図中 `help` はスタックが空になったプロセッサが他のプロセッサからタスクの供給を受けた場合を示し，`takeback` はタスクの完了待ちになったプロセッサが，タスクを盗んだプロセッサから仕事の一部をタスクとして取り返す場合を示す．実行の初期・中期ではときたま仕事の取り返しが繰り返されている以外は，タスクの生成・授受は最小限に抑えられているといえる．実行の終盤では，残り仕事量が減ってくるためタスクの生成・授受が活発になるが，残り仕事量も減っていくためそのうち実行が完了することになる．

以下では，負荷分散を可能としたことによるオーバヘッドとして，負荷分散を可能とした並列プログラムの 1CPU 上での実行時間を逐次プログラムの実行時間と比較する．fib(36) については**表 1** 上部のようになった．括弧内は，逐次 C プログラムを 1 とした相対的な実行時間を示している．SPARC のオーバヘッドが大きいのは，クロージャ生成のオーバヘッド，入れ子関数と元の関数で共有される変数のレジスタ割り付け阻害のオーバヘッドが高いためだと思われる．クロージャ生成のオーバヘッドについてはトランポリンに関する命令キャッシュのフラッシュを伴うことが挙げられる．PowerPC のオーバヘッドが比較的小さいのは，PowerPC ではもともと関数ポインタをコードのアドレスではなく，コードとその環境（定数ポインタのための）を表すデータ構造へのポインタとしているため，クロージャ生成時のスタック上への動的コード生成が不要となるためである．Pentium の場合にそれほど悪くないのは，もともとレジスタが少ないために，レジスタ割り付け阻害のオーバヘッドが目立たない点が考えられる．

表 1 逐次 C プログラムとの単体プロセッサ実行時間の比較 (秒)
Table 1 Single PE execution time compared to sequential C programs (sec).

Program	Processor	C	LBTC (/C)
Fibonacci	SPARC	2.36	7.88 (3.33)
	PowerPC	1.94	3.52 (1.81)
	Pentium	0.537	1.20 (2.24)
Pentomino	SPARC	14.4	23.0 (1.60)
	PowerPC	9.41	12.4 (1.32)
	Pentium	3.76	4.61 (1.23)

また，ペントミノ全解探索についても表 1 下部に示す．PowerPC と Pentium で逆転しているのは，一つの関数内での仕事の量が増えているため，クロージャ生成のオーバヘッドより，入れ子関数と元の関数で共有される変数のレジスタ割り付け阻害のオーバヘッドのほうが重要となるためだと考えられる．

6.2 XC-cube の入れ子関数での評価

XC-cube の入れ子関数は比較的新しい実装であり，XC-cube コンパイラは GCC 3.2 をベースにしている．以下では，SPARC については 750MHz

表 2 性能改善
Table 2 Performance Enhancement.

Program	Processor	C	LBTC	closure	L-closure	Cilk	Cilk-S
Fibonacci	SPARC	0.80	3.47 (4.34)	1.29 (1.61)	0.96 (1.20)	3.27 (4.08)	-
	Pentium	0.50	1.07 (2.16)	0.95 (1.91)	0.80 (1.62)	2.52 (5.09)	-
Pentomino	SPARC	4.47	8.09 (1.81)	6.58 (1.47)	4.85 (1.09)	18.2 (4.08)	8.96 (2.01)
	Pentium	3.68	4.36 (1.19)	4.28 (1.16)	3.90 (1.06)	9.94 (2.70)	7.19 (1.95)

UltraSPARC-III で評価を行っている．XC-cube コンパイラでは，入れ子関数のトランポリンの代わりに環境と関数ポインタを用いる “closure” と，入れ子関数がアクセスする各変数に場所を 2 つ準備し，入れ子関数が呼び出されるまでレジスタに値がおいておける可能性を高めた “L-closure” がある．詳細は別論文で発表する予定である．GCC の入れ子関数に加えて，XC-cube の入れ子関数も用いて評価を行い，また，Cilk 5.3.2[9] の結果を加えたものを**表 2** に示す．もともと GCC の入れ子関数でも Cilk より多くの場合で高速であるが，XC-cube を用いることで，入れ子関数を利用した負荷分散がさらに低いオーバヘッドで実現できることがわかる．また，提案方式では配置情報のコピーを避けることができるため，Cilk より高速である．表 2 の Cilk-S では，`SYNCHED` を用いて複数の子の間でのコピーを避けているが，それでも，提案方式の性能には及ばないことがわかる．

7. 議　　論

OPA 言語など高水準言語の実装で提案手法を利用したいという場合に言語仕様をどうするかという問題がある．オリジナルの LTC では，マルチスレッドとして，並列処理が記述されている場合の一部のスレッドの継続をタスクとした．今回のタスク分割では，スレッドを用いていない逐次プログラムが将来予定している仕事の一部を切り取ってタスクにしているため，「スレッド」という高水準記述には直接対応できない．特にマルチスレッドという高い水準の抽象化を行うと，一時的なバックトラックの記述は困難になる．つまり，OPA 言語のようなマルチスレッド言語では提案する低水準記述方式の能力すべてを発揮できない．

そこで，LBTC の負荷分散がうまくできてしかも一時的バックトラックもできるという性質は保ったままで，できるだけ高水準に近づけた言語を考えたい．それにはタスク要求に対する「ハンドラ」という方式が考えられる．これは，例えば，例外処理で

```
try { ... } catch (Exception e) {...}
```

が受け取った例外 `e` への対処 (ハンドラ) を catch 節として記述できるのに対応して，

```
POLL(bk1);
{
  int bk2(request *req){
    int r = 1;
    if (r) r = bk1 (req);
    if (r) return r;
    hand
    return 0;
  }
  body
}
```

図 6 タスク要求ハンドラの入れ子関数への翻訳
Fig. 6 Translating a task request handler into a nested function.

```
{ ini }
{
  int bk2(request *req){
    int r;
    { fin }
    r = bk1 (req);
    { ini }
    return r;
  }
  body
}
{ fin }
```

図 7 initially-finally 節の入れ子関数への翻訳
Fig. 7 Translating a initially-finally clause into a nested function.

`do {`*body*`} handle_spawn_req(`*req*`) {`*hand*`}`

のよう書けば，タスク要求 *req* への対処 (ハンドラ) を`handle_spawn_req` 節として記述できるといった方式である．ただし，例外処理と違い，ハンドラは非局所脱出には使わず実行後は元の計算に戻れるものとする．タスク要求ハンドラを入れ子関数による記述方式に翻訳すると，**図 6** のようになる．ただし，*body* 中では，入れ子関数として `bk1` ではなく `bk2` を使うことになる．また，*hand* 中で，タスクが生成できた場合は “`return 1;`” とすればよい．

また，バックトラック探索のように，一時的な変更を一時的に元に戻すためには，例えば，

```
try { ... } finally { ... }
```

が，`try` から出る場合に必ず `finally` 部を実行することになっているように，

`initially {`*ini*`} do {`*body*`} finally {`*fin*`}`

などとして，出る前には必ず `finally` 節，入る前には必ず `initially` 節を実行することにするというこ

とが考えられる．initially 節–finally 節を入れ子関数による記述方式に翻訳すると，**図 7** のようになる．ただし，*body* 中では，入れ子関数として `bk1` ではなく `bk2` を使うことになる．

なお図 6，7 は，それぞれ図 3，4 と対応している．

このようなマルチスレッドよりは低水準だが，LBTC の能力を生かしながらできるだけ高水準言語を目指す方式の利点には次のものが考えられる．

- 入れ子関数を直接用いるよりは高水準で，記述量が少なくて済む．
- 基本的には逐次実行しているので，トレースやバックトレースなどデバッグがマルチスレッド言語よりも容易である．
- 分割されない限り逐次的に実行される複数の処理で同じ作業メモリを使いまわすことが容易であり，メモリ使用量/コピー量の削減や，メモリアクセスの局所性の向上が期待できる．
- タスク要求ハンドラで必要なデータをパックして通信を行えば，クラスタ計算などの分散環境にも対応できる．また，パックした内容をバックアップしておけば，タスクを盗んだ先との通信が途切れたときなどに，バックアップを用いて計算が続けられる可能性がある．

8. おわりに

本論文では，入れ子関数を利用することで，バックトラックにも対応できる動的負荷分散が記述可能であることを示した．また，共有メモリ型並列計算機上で提案する記述方式のプログラムの評価を行い，並列プログラム実行においては理想的な台数効果が得られることや，入れ子関数の実装改善の効果について述べた．

今後は，分散環境での評価を行う予定である．Leapfrogging の制約の緩和すること，タスクをバッファリングすることで，タスク授受の遅延隠蔽が可能かを試す予定である．

参 考 文 献

1) Breuel, T. M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).
2) Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proc. of International Workshop on Parallel Symbolic Computing*, LNCS, Vol. 748, Springer-Verlag, pp. 94–107 (1993).
3) Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices (PLDI'98)*, Vol. 33, No. 5, pp. 212–223 (1998).
4) Halstead, Jr., R. H.: New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, *Parallel Lisp: Languages and Systems* (Ito, T. and Halstead, R.H.(eds.)), LNCS, Vol. 441, Springer-Verlag, pp. 2–57 (1990).
5) Jones, S. P., Ramsey, N. and Reig, F.: C--: a Portable Assembly Language that Supports Garbage Collection, *International Conference on Principles and Practice of Declarative Programming* (1999).
6) Mohr, E., Kranz, D. A. and Halstead, Jr., R. H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).
7) Ramsey, N. and Reig, F.: A Single Intermediate Language That Supports Multiple Implementations of Exceptions, *Proc. of PLDI2000*, pp. 285–298 (2000).
8) Stallman, R. M.: *Using and Porting GNU Compiler Collection*, Free Software Foundation, Inc., for gcc-2.95 edition (1999).
9) Supercomputing Technologies Group: *Cilk 5.3.2 Reference Manual*, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts (2001).
10) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proc. of PPoPP'99*, pp. 60–71 (1999).
11) Umatani, S., Yasugi, M., Komiya, T. and Yuasa, T.: Pursuing Laziness for Efficient Implementation of Modern Multithreaded Languages, *Proc. of ISHPC2003*, LNCS, Vol. 2858, pp. 174–188 (2003).
12) Wagner, D. B. and Calder, B. G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proc. of PPoPP'93*, pp. 208–217 (1993).
13) 八杉昌宏, 馬谷誠二, 鎌田十三郎, 田畑悠介, 伊藤智一, 小宮常康, 湯淺太一: オブジェクト指向並列言語 OPA のためのコード生成手法, 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG 11 (PRO 12), pp. 1–13 (2001).
14) 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, 情処研報 97-PRO-14(SWoPP'97), pp. 79–84 (1997).
15) 田畑悠介, 八杉昌宏, 小宮常康, 湯淺太一: 入れ子関数を利用したマルチスレッドの実現, 情報処理学会論文誌：プログラミング, Vol. 43, No. SIG 3 (PRO 14), pp. 26–40 (2002).