オブジェクト指向並列言語OPAのための遅延正規化手法

馬谷誠二[†] 八杉昌宏[†]
小宮常康[†,☆] 湯淺太一[†]

本論文では，Java 言語を拡張したオブジェクト指向並列言語 OPA のためのいくつかの実装手法を提案する．Java 等のオブジェクト指向並列計算においては，各スレッドは，`synchronized` 構文を実現するため同一性（スレッド ID）を維持する必要がある．また，共有オブジェクトへの排他的アクセスをサポートするため一般的な同期を可能とする機能（中断および再開）が必要である．洗練された例外処理のため，OPA は動的スコープによる join 構文を採用しており，例外ハンドラは，並列実行中，任意の子スレッドにより投げられた例外を捕まえられる．マルチスレッド言語の効率良い実装において,「遅延」は重要な概念である．たとえば，遅延タスク生成（Lazy Task Creation: LTC）は，負荷分散を効率良く行うことができる．本論文では，スレッド同一性維持，一般的な同期，動的スコープによる join といった OPA の現代的な言語機能に遅延性を利用する方法を提案する．さらに，OPA 処理系は移植性のため C コードを生成するが，これにより LTC を用いるのが難しい．Cilk 言語の実装は，制限された（うまく構造化された）マルチスレッド計算において，すでにこの問題を解決しているが，我々の実装は，LTC を採用するだけでなく，OPA の持つ現代的な言語機能をサポートしており，さらに Cilk を上回る性能を達成している．

Lazy Normalization Techniques for an Object-oriented Parallel Language OPA

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

This paper describes various techniques for implementing a modern multithreaded language OPA, which is an extended Java programming language that supports object-oriented programming and exception handling. For object-oriented parallel computing as in Java, each thread needs to keep its identity to implement the `synchronized` construct and each thread should have ability in general synchronization (suspension and resumption) to support mutually-exclusive access to a shared object. For elegant exception handling, OPA employs a join construct with dynamic scope which enables an exception handler to catch an exception thrown by any of child threads during parallel execution. For efficient implementation of multithreaded languages, *laziness* is an important idea; for example, Lazy Task Creation (LTC) is a well known technique for good load balancing. In this paper, we pursue laziness for the modern language features, including thread identity preservation, general synchronization, and dynamically-scoped join. In addition, the OPA system generates C code for good portability; this makes the adoption of LTC difficult. Although the implementation of the Cilk language has already overcome this difficulty in a limited (well-structured) multithreaded computations, our implementation not only adopts LTC but also supports the modern language features and furthermore achieves better performance than Cilk.

1. はじめに

並列処理の記述のためのマルチスレッド言語が数多く存在する．それらの言語は動的にスレッドを生成し，スレッド間で互いに協調するための簡潔な構文を備えている．我々は，Java 言語をシンプルだが強力なマルチスレッド構文により拡張したオブジェクト指向並列言語 OPA（Object-oriented language for PArallel processing）[1)~5)] の設計・実装を行っている．OPA では fork/join 型マルチスレッド構文を用意し，分割統治型の並列処理の記述を可能としている．さらに，OPA では，オブジェクト指向パラダイムにより，たとえば，

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

共有オブジェクトを介した通信（同期）や相互排他的なメソッド起動といった，そのほかの同期処理も表現できる．またマルチスレッド構文に合わせて拡張した例外処理もサポートする．OPA の詳細については，2 章で説明する．

マルチスレッド構文，およびその実装は，次のようなな性質を満たすべきである．まず，それらは効率良くなければいけない．いいかえると，スレッド生成・同期処理のオーバヘッドを減らす必要がある．我々は，マルチスレッド構文を使用して，細粒度マルチスレッドプログラムを記述したいのであって，もし許容できないほどのオーバヘッドを含むようであれば，多くのプログラマは，それを使おうとはせず，かわりに，粗粒度スレッドを生成し，細粒度な仕事単位を明示的に管理するだろう．そこで，我々は，OPA のための効率良い実装（コンパイル）手法を提案する．次に，言語システムは移植性が良くなければいけない．これを実現するために，OPA コンパイラは標準の C コードを生成している．これとは対照的に，他のシステムのいくつかは，効率のためアセンブリ言語レベルの手法を用いている．最後に，マルチスレッド構文は，十分な表現力を備えていなければならない．他のシステムのいくつかは，効率のため表現力に制限を加えている．たとえば，Cilk[6),7)] の fork/join 構文では，「静的スコープ」を採用しており，また，そのほかの同期処理を備えていない．Cilk の構文は簡潔ではあるが，柔軟性に欠け，さまざまな不規則なプログラムを記述するのには向いていない．

本論文では，fork/join 構文のオーバヘッドを減らすための 3 つの技法を提案する．それらに共通する基本的なアイディアは**遅延**（laziness）である．遅延とは，ある操作の実行を，その結果（あるいは効果）が本当に必要になるまで遅らせることである．1 つめに，関数フレームをヒープ領域に確保する処理を遅延することができる．C 言語上で細粒度マルチスレッド言語を実装する場合，各プロセッサは，たくさんのスレッドを実行するにもかかわらず，単一の（あるいは，限られた個数の）スタックしか持たないため，複数スレッドのためのフレームは，一般的にヒープに確保される．効率のため，我々は，まず最初にスタックにフレームを確保し，そのフレームが他のスレッドの実行の妨げとならない限り，そのままそこに居続けるようにする．2 つめに，**タスク**（task）の生成を遅延できる．本論文において，タスクとはスケジューラブルな能動実体（のデータ構造）のことであり，単一の（言語レベルの）スレッドには対応しない．ブロックされたスレッドは，使用していたプロセッサとスタックを解放するためタスクとなる．また，負荷分散を目的として，各スレッドはプロセッサ間の移動のためタスクとなるかもしれない．我々の方法では，うまく負荷分散しながら，実際に生成されるタスクの数を減らすことができる．さらに，我々は，スレッド生成に関連する他のいくつかの操作も遅延させることで，結果的に，スレッド生成コストをゼロに近づけることができた．3 つめに，同期処理のためのデータ構造の構築を遅延させることができる．OPA では，fork/join 構文におけるスレッドの同期先が動的スコープにより決められ[4)]，また各スレッドはブロックされるかもしれないので，そのようなデータ構造は不可欠である．

本論文では，OPA の高効率実装を，C 言語の並列拡張である Cilk 言語の実装と比較する．Cilk 言語も fork/join 構文を持ち，その処理系は共有メモリ型マルチプロセッサ上においてうまく負荷分散する機能を備えている．Cilk は，OPA における共有オブジェクトを介した通信（同期）や相互排他的なメソッドの実行のような，他の同期処理機能を提供しないので，Cilk の実装手法をそのまま OPA に適用することはできない．しかしながら，遅延を利用することで，我々の OPA 実装は，OPA のより豊かな表現力を持ちながら，Cilk 実装よりも優れた性能を得ている．

本論文の残りは，以下のように構成されている．2 章では，OPA 言語について概説する．3 章では，従来の OPA の実装について述べる．4 章において，OPA の実装における遅延正規化手法を提案する．遅延正規化といういいまわしは，正規バージョン（重量バージョン）の実体（たとえば，ヒープに確保されるフレーム）は，本当に必要なときだけ，一時的バージョン（軽量バージョン）の実体（たとえば，スタックに確保されるフレーム）から生成される，という意味である．5 章では，関連研究について触れ，それらの表現力，効率，移植性について議論する．6 章では，OPA と Cilk で書かれたいくつかのプログラムの性能評価の結果を示す．最後に，7 章で結論を述べる．

2. OPA 言語の概要

先に述べたように，OPA は，構造化された分割統治型の並列アルゴリズムを直接的に表現するため，fork/join 構文を持つ．この構文は，とてもポピュラーなものであり，多くの並列言語[7)~9)] で採用されている（それらの言語はシンタックスおよびセマンティクスの詳細において多種多様である．それらについては，5 章において後で議論する）．

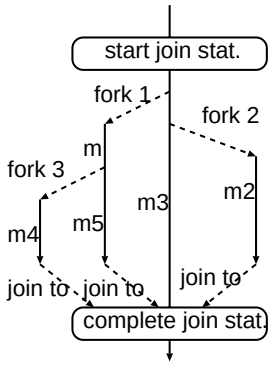

図 1 fork/join 型並列処理
Fig. 1 Fork/join parallelism.

```
 1 class Fib{
 2   private static int fib(int n){
 3     if(n < 2) return n;
 4     else{
 5       int x = par fib(n-1);
 6       int y = fib(n-2);
 7     join:
 8       return x+y;
 9     }
10   }
11   public static void main(String[] args){
12     int r = fib(36);
13   }
14 }
```

図 2 OPA 言語による `fib` コード
Fig. 2 `fib` code in OPA language.

OPA では，複数スレッドを同期させるため，“`join` 〈文〉” のような構文の join 文を導入する．〈文〉 は，カレントスレッドにより実行され，〈文〉 の実行中に新しく生成されたスレッドは，join 文の実行完了に同期する．新たなスレッドを fork するため，“`par` 〈文〉” のような構文の par 文を使用する．スレッドは，〈文〉 を実行するために fork される．たとえば，次の文：

```
join { par o.m(); par o2.m2(); o3.m3(); };
```

は，2 つのスレッドを，それぞれ `o.m()` と `o2.m2()` を実行するため fork し，カレントスレッドは `o3.m3()` を実行し，それから fork したスレッドの実行完了を待つ（**図 1** 参照）．さらに，fork されたスレッドが別の新たなスレッドを fork した場合，そのスレッドもまた同じ場所に同期する．たとえば，上記の例において 1 つめに fork されたスレッドの実行するメソッド `m()` が次のように定義されていると：

```
m(){ par o4.m4(); o5.m5(); };
```

`o4.m4()` を実行するスレッドもまた join 文の最後に同期する．

スレッドが fork すると，自動的に，そのスレッドを fork した par 文を含む，一番内側の join 文が，そのスレッドの join 先となる．いいかえると，OPA は join ターゲットが動的スコープの fork/join 構文を持つ．動的スコープのより詳細に興味があるようであれば，文献 4) を参照されたい．

図 2 は，OPA で書かれた単純なフィボナッチプログラムである．5 行目の `par fib(n-1)` により，そのメソッド起動は新しく生成されるスレッドにより並行に実行される．`fib` が通常のメソッドであることに注意されたい．つまり，`fib` を逐次に呼び出すこともできる（6 行目）．これは親スレッドを計算に参加させるためには望ましく，またスレッド生成数も減らすことができる．このプログラムでは，子スレッドから返される値を安全に使用するために，join 文の変種である `join` ラベルを用いている．この `join` ラベルを文 `return x+y` の前に置くことで，`x` が計算される前に `y` と足されるのを防いでいる．

fork/join 構文は，分割統治型のプログラムを表現するのに十分なだけ簡潔で強力ではある．しかしながら，ときには，より複雑で不規則なプログラムを書きたいことがある．そういうプログラムでは，スレッドはオブジェクトに排他的にアクセスしたり，別のスレッドにあるイベントが起こったことを知らせたりするといった機能が必要である．紙面の都合により詳細にはあまり触れられないが，OPA はそれらの機能をオブジェクト指向スタイルで提供する[2),3)]．大事なことは，そういった機能を使用するプログラムにおけるスレッドは，一般的な状況でブロックされる可能性があり，そのことが，OPA 処理系を他の fork/join 構文のみをサポートする言語のシステムに比べてより複雑にしている．

3. OPA のための実装手法

我々は，OPA 処理系を共有メモリ型並列計算機上に実装している[5)]．OPA 処理系は，OPA から C へのコンパイラおよび C で書かれた実行時システムからなる．

この章では，従来の OPA の実装について述べる．特に，本研究に関連する 3 つの事柄，(1) 関数フレームの確保，(2) スレッド生成およびスケジューリング，(3) 入れ子になった fork/join の管理，について説明する．

次章において，これらを改良するための遅延正規化手法について説明する．

3.1 関数フレームの確保

各プロセッサは，数多くのスレッドを管理するのに，

たった 1 つ（あるいは限られた個数）の C の実行スタック（以降，単にスタックと呼ぶ）を持っている．そのため，単純な実装においては，スレッドのための関数フレームはヒープに確保されることが多く，スタック上の確保に比べ性能面で劣る．多くの研究が，この問題に取り組んでいる[7),8),10)~12)]．

Hybrid Execution Model[12)] と OPA 処理系は，フレームの確保にほとんど同じ方法を用いている．どちらも，2 種類のフレーム（スタックに確保されるフレームと，ヒープに確保されるフレーム），2 つのバージョンの C コード（fast バージョンと slow バージョン）を使用する．

各プロセッサは，fast バージョンの C 関数を使って，とりあえず，スレッドをそのスタック上で実行する．OPA では，メソッドを逐次的にも並行的にも呼び出せるため，スレッドは複数の関数フレームからなるかもしれない．何らかの理由（たとえば，相互排他処理）によりスタック上で実行中のスレッドがブロックされるとき，そのスレッドの一部であるスタックフレームごとに，ヒープに関数フレームが確保され，それぞれの状態（**継続**）が保存される．ヒープフレームは，（スタック上では暗黙の）caller（呼び出し側）/callee（呼び出される側）間の関係を保たなければいけない．そこで，callee のフレームは，その caller のフレームを指すフィールドを持っており，1 つのスレッドはヒープにおいては，フレームのリストとして表現される．ブロックされたスレッドの実行を再開する際は，slow バージョンの C コードによりヒープフレームから状態がスタックに戻される．

OPA コンパイラが出力した，fib プログラムの fast バージョンの C コードを**図 3** に示す（コード中，`//` で始まるコメントは，そこにあるコードの詳細が省略されていることを意味する）．OPA のメソッドに対応する C の関数は，通常の引数に加え，プロセッサ固有のデータ領域を指す `pr` も引数にとる．21–28 行目が逐次メソッド呼び出しに対応する．OPA のメソッド呼び出しは，C の関数呼び出しとそれに付随するコードへとコンパイルされる（混乱を避けるため，OPA における起動をメソッド起動，C における呼び出しを関数呼び出しと呼ぶことにする）．関数呼び出しにおいて caller へ制御が戻ってくると，まず callee がサスペンドしているかどうかがチェックされる（22–23 行目，チェック方法の詳細は後述）．サスペンドしていた場合，caller も，callee と同じスレッドに属するので，自身の継続を保存する（24–25 行目）．`f_frame` はヒープフレームを表す構造体であり，親フレームへのポインタを入れるフィールド `caller_fr` 等を含む．その後，自分のフレームを callee のフレームにつなぎ（26 行目），さらに caller の呼び出し側へ自分がサスペンドしたことを報せる（27 行目）．

```
 1 int f__fib(private_env *pr, int n) {
 2  f_frame *callee_fr; thread_t *nt;
 3  f_frame *nfr; int x,y; f_frame *x_pms;
 4  if(n < 2) return n;
 5  else{
 6   /* joinブロックの開始 */
 7   frame *njf = ALLOC_JF(pr);
 8   njf->bjf = pr->jf;    /* 外側へリンク */
 9   njf->bjw = pr->jw;    /* 外側へリンク */
10   pr->jf = njf; pr->jw = njf->w;
11   /* 新しいスレッドの生成 */
12   nt = ALLOC_OBJ(pr,sizeof(thread_t));
13   nt->jf = pr->jf;    /* 重みの切り分け */
14   nt->jw = SPLIT_JW(pr, pr->jw);
15   nt->cont =nfr=MAKE_CONT(pr,c__fib);
16   // 継続を保存（n-1 等）
17   x_pms = MAKE_CONT(pr, join_to);
18   nfr->caller_fr = x_pms;
19   enqueue(pr, nt);  /* 実行可能キューへ */
20   /* 逐次メソッド呼び出し */
21   y = f__fib(pr, n-2);
22   if((y==SUSPEND) /* サスペンドチェック */
23      && (callee_fr=pr->callee_fr)){
24    f_frame *fr=MAKE_CONT(pr,c__fib);
25    // 継続を保存
26    callee_fr->caller_fr = fr;
27    pr->callee_fr=fr; return SUSPEND;
28   }
29   WAIT_FOR_ZERO(pr->jf);      /* 同期 */
30   pr->jf = pr->jf->bjf;
31   pr->jw = pr->jf->bjw;
32   x = x_pms->ret.i; /* 返り値の取り出し */
33   return x+y;
34  }
35 }
```

図 3 `fib` からコンパイルされた擬似 C コード
Fig. 3 Compiled C code for `fib` (pseudo).

callee は，メソッドの返り値に加えて，自身のヒープフレームへのポインタも返さなくてはいけない（もしフレームが存在すれば．存在しない場合には，0 が返される）．Hybrid Execution Model では，ヒープフレームへのポインタが C の関数呼び出しの返り値として返され，メソッドの返り値は，caller によってあらかじめ確保された（メモリ上の）別の場所に入れられる．しかし，この方法では，呼び出しごとにその返り値を取り出すというメモリアクセスが生じてしまう．このオーバヘッドを減らすため，OPA 処理系ではメソッドの返り値が C の関数呼び出しの返り値として返される．その返り値として特別な値 `SUSPEND`（たとえば `-5` のようなあまり使われない値から選択される）が返されると，それは callee がサスペンドしてい

る可能性があることを意味する．そうした場合にだけ，処理系はさらに，callee のフレームへのポインタがあるプロセッサ固有の場所（pr->callee_fr）に入れられていないかをチェックする（入っていなければ，メソッドの返り値が SUSPEND に重なっただけだと判断できる）．

3.2 スレッド生成およびスケジューリング

OPA の実行時システムは，各スレッドを,（プログラマからは見えない）メタな**スレッドオブジェクト**により管理する．スレッドオブジェクトは次のようなフィールドを含む：(a) 継続,(b) join ブロック情報．継続フィールドには（もし存在すれば）対応するヒープフレームのリストへのポインタが入る．join ブロック情報については後述する．さらに，OPA 処理系は，スレッドの同一性の管理にもスレッドオブジェクトを利用する．たとえば，ロックを獲得した同一のスレッドは，何回でもそのロックを獲得することができるという Java の synchronized メソッドを実現するのに用いられる．

従来の OPA 実装においては，スレッド生成は以下のようにして処理される（図 3 の 12–19 行目）：

(1) スレッドオブジェクト（型 thread_t）を生成する（12 行目），
(2) join フレーム情報に関連する操作（13–14 行目）（後述），
(3) fork したメソッド本体の先頭から実行を開始するという継続を保存（15–16 行目），
(4) 新規スレッドの完了後，join 同期処理を行う関数 join_to を実行するフレーム x_pms をつなげる（17–18 行目）．フレームはプレースホルダとしても用いられ，スレッドの返り値として任意の型の値を返せるように共用型で定義されたフィールドを持つ（たとえば，int 型なら ret.i），
(5) 生成したスレッドオブジェクトをプロセッサのローカルなタスクキューに入れる（19 行目）．

タスクキューは，両頭キュー（deque）となっており，スレッドオブジェクトはその末尾に入れられる．プロセッサはスタックが空になると，ローカルなタスクキューの末尾からタスクを取り出して実行する．アイドルなプロセッサは，他のプロセッサのタスクキューの先頭から（最も仕事量が大きいと思われる）タスクを盗みだすことができ，これにより，動的負荷分散を可能としている．すべてのスレッドが，キューに入れられる前にタスク（プロセッサ間での授受が可能な表現）へと変換されていることに注意してほしい．

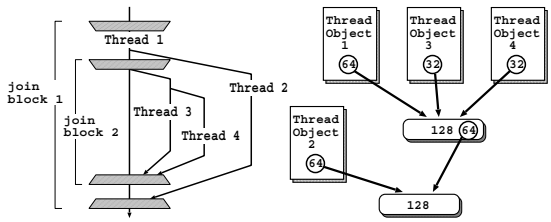

図 4 join フレームおよびスレッドオブジェクトの構造
Fig. 4 Structure of join frames with thread objects.

3.3 入れ子になった fork/join の管理

最後に，入れ子になった fork/join 構文の管理について説明する．join ブロックから抜け出すとき，親スレッドは当然，その join ブロック中で生成されたすべての子スレッドの実行完了を待たなければならない．子スレッドの個数は，一般的に実行時にしか分からない．そのため，OPA 処理系では，join ブロックごとにカウンタを用意して，まだ同期していない子スレッドの（最大）数を記録する．

OPA の同期処理は **join フレーム**により実装される．join フレームは，**図 4** のようにスレッドオブジェクトとともに用いる．各 join フレームは，内部のカウンタにより，まだ同期していないスレッドの最大数を記録し，その値が 0 になれば，同期全体が完了したと分かる．つまり，我々は重み付き参照カウント法[13] を採用している．各スレッドは，そのスレッドオブジェクト中に，join フレームへの重み付き参照（ポインタと重み）を持つ．この方法の利点は，子スレッドの生成には親スレッドの持つ重みを親子間で切り分けるだけで，そのほかの操作や同期がまったく必要ない点である．各 join フレームは，join ブロックの入れ子構造を保つため，1 レベル外側の join フレームへのポインタ（重み付き参照）を持つ．また，同期のためにサスペンドした親スレッドが待ち合わせる場所もその中にある．

図 3 のコードだと，親スレッドが join ブロックに入るとき（7–10 行目），新しい join フレーム njf を生成しカレントスレッドの持つ 1 レベル外側の join フレームへのポインタ（pr->jf，pr->jw）を入れる．カレントスレッドには，あらたに内側への重み付き参照を持たせる．13–14 行目では，新しく生成するスレッドに重みを切り分ける．その後，join ブロックから抜けるとき（29–31 行目），同期先の join フレームの重みをチェックして同期をとり，1 レベル外側の join フレームへの重み付き参照を戻している．同期処理の完了後，32 行目でプレースホルダから返り値を取り出す．

まとめてみると，従来の OPA の実装手法は，細粒

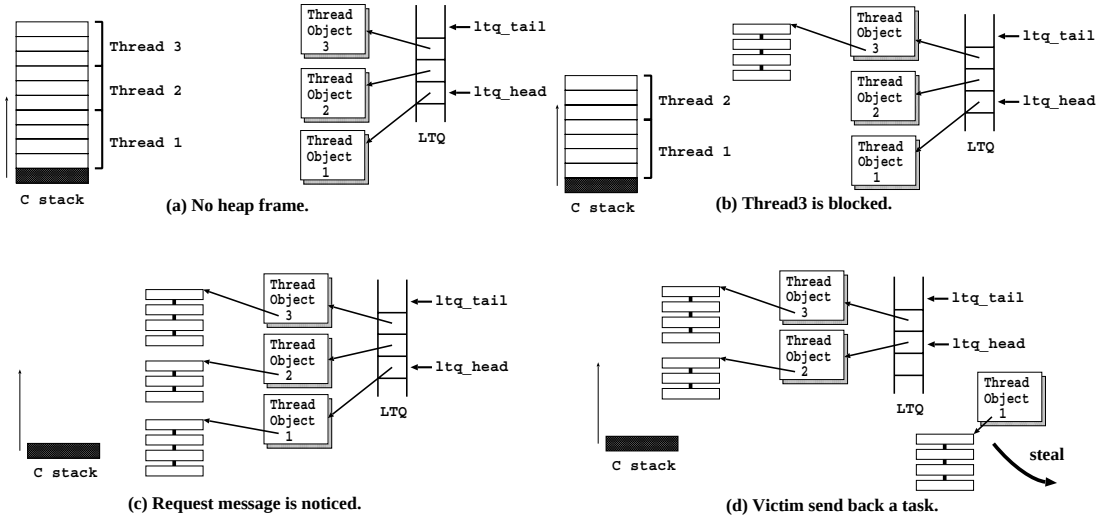

図 5 タスクスティール処理中，victim のスタックにあるスレッドはタスクに変換される
Fig. 5 Each thread in victim's stack is converted to a task during a task steal.

度マルチスレッドプログラムに対して，許容できないほどのオーバヘッドを課しているといえる．

4. 遅延正規化手法

前の章で説明した実装手法のうち，いくらかの処理は，スレッドのスケジューリング方針を変更することで遅延させることが可能である．

4.1 節では，スレッドを生成する際の，ヒープフレームの確保およびタスク生成を遅延させる方法を提案する．メソッド呼び出しにおけるヒープフレームの確保は，3.1 節ですでに，そのメソッドを含むスレッドがサスペンドするまで遅延されているが，この章で提案する手法を用いると，さらにスレッド生成処理に関して，本当にタスク要求のメッセージが届くまで，(親スレッドと子スレッドのどちらについても）ヒープフレームの確保を遅延させることが可能となる．

4.2 節では，join フレームの確保を遅延させる方法を提案する．

4.1 遅延タスク生成

先に説明したように，従来の OPA 処理系は，スレッド生成時にタスク，すなわち完全なスレッドオブジェクトを生成する．

遅延タスク生成（Lazy Task Creation: LTC）[14] は，もともと Multilisp [15] の高効率実装に用いられた手法である．基本的なアイディアを簡潔にまとめると次のようになる：

(1) スレッド生成時，プロセッサは子スレッドを親スレッドの継続より先に実行する．親スレッドの継続は（自動的に）スタックの中に保存される．

(2) アイドルなプロセッサは他プロセッサのスタックの底からスレッド（の継続）を抜き取り，それをもとにタスクを生成して実行することにより仕事を盗む．

タスクを盗むプロセッサを **thief**，盗まれるプロセッサを **victim** と呼ぶことにする．LTC の利点は，主に，(a) 親スレッドの継続の保存にかかるコストは逐次呼び出しのそれと変わらない，(b) thief が victim のスタックからスレッドを抜き取れることで動的負荷分散を可能にしている，の 2 つである．

この 2 つの処理を C の上で実現することを考慮してみる．まず，親スレッドの継続は，C の関数呼び出しでもスタックに保存されるので，何も考えなくてよい．次に，C のスタックから継続を抜き取ることを考えるとき，それを移植性を保ったまま実現するのは簡単ではない．Multilisp 処理系の場合だと，スタックをアセンブリ言語レベルで操作する．OPA と同様に C で実装されている Cilk 処理系の場合，親スレッドの継続を，前もって，ヒープに確保されたフレームに保存する．この方法だと，thief が victim のスタックを操作することなしに抜き取ることが可能ではあるが，すべてのスレッド生成にある程度のオーバヘッドをかけてしまう．

この問題を解決するため，我々は，メッセージパッシングによる LTC の実装 [16] を採用した．その特徴は，thief が，スタックを含めた他のプロセッサの局所データにアクセスしない点にある．thief は，単に，ランダムに選択された victim へタスク要求メッセージを送信し，応答を待つだけである．victim は，その

要求に気づくと，自分のスタックから継続を抜き取りタスクにして thief へと送り返す．これを行うにあたり，OPA 処理系では，ヒープフレーム確保を遅延する機構を利用する（**図 5**）．C スタックの底から継続を抜き取るのに，処理系は，その上にあるすべての継続（スレッド）を一時的に（内部的に）サスペンドする．

遅延を利用して `fib` から生成された（擬似）C コード**図 6** を用いて詳細について説明する．スレッド生成部は，14–30 行目にある．15 行目の関数呼び出しが子スレッドの実行に対応する．16–17 行目のサスペンションチェックは，従来のものと変わらない．つまり，底の継続を抜き取る間，スタック上のすべての関数はすべてのスレッドがブロックされたかのように動作する．従来との差異は 20 行目にあり，そこでプロセッサへのタスク要求メッセージ（`thief_req` に入っている）がチェックされる．これが 0 以外の値（thief のプロセッサ ID を意味する）であれば底の継続を渡すため，親スレッドのサスペンドを開始する（21–27 行目）．`ltq_ptr` は，タスクキュー中の対応するスレッドオブジェクトを指す．重みを切り分ける方法が従来のコードと異なるが，それについては次節で述べる（要求をチェックするのに，ポーリング手法を用いている．効率良いポーリングの方法については文献 17) 等に詳しい）．

以上の操作により，底の継続はタスクへと変換されタスクキューの先頭に入っているので，thief へと送り返すことができる．この方法の短所は，底の継続だけでなく，スタック上にあるすべての継続を変換してしまう点にある．しかしながら，LTC では，うまくバランスのとられた分割統治型のプログラムを想定しており，そのようなプログラムの実行においては，わずかな回数しかスティールは起こらないと期待できる．また，そうではないプログラムについても，我々は，次のように対処することでオーバヘッドを回避している．victim はスティール処理の後，実行再開のためにタスクキュー中のすべてのタスクをスタックに戻す（すなわち，C スタックを完全に元どおりにする）のではなく，victim は slow バージョンコードを用いて，キューの末尾から 1 つだけタスクを取り出してスタックに戻す．キューの残りのタスクは次のタスク要求の際，変換にかかるオーバヘッドなしに thief に渡すことができる．

これまで，C スタックから底の継続をいかにして抜き取るかについて述べてきた．しかしながら，今のところ，3.2 節でリストしたタスク生成に関する 5 つの操作のうち 2 つ，継続の保存とキューへの追加，しか

```
 1 int f__fib(private_env *pr, int n) {
 2  f_frame *callee_fr;
 3  thread_t **ltq_ptr = pr->ltq_tail;
 4  int x, y; f_frame *x_pms = NULL;
 5  if(n < 2) return n;
 6  else{
 7   /* join ブロックの開始 */
 8   pr->js_top++;   /* join スタック操作 */
 9   /* polling */
10   if(pr->thief_req){ /* thief からの要求 */
11    // サスペンドコード
12   }
13   /* 新しいスレッドの生成 */
14   pr->ltq_tail = ltq_ptr+1;  /* LTQ 操作 */
15   x = f__fib(pr, n-1);   /* 逐次呼び出し */
16   if((x==SUSPEND)   /* サスペンドチェック */
17      && (callee_fr=pr->callee_fr)){
18    f_frame *x_pms=MAKE_CONT(pr,join_to);
19    callee_fr->caller_fr = x_pms;
20    if(pr->thief_req){ /* スティール処理中 */
21     f_frame *fr=MAKE_CONT(pr,c__fib);
22     (*ltq_ptr)->jf=*(pr->js_top);
23     (*ltq_ptr)->jw=SPLIT_JW(pr,pr->jw);
24     (*ltq_ptr)->cont = fr;
25     // 継続を保存
26     SUSPEND_IN_JOIN_BLOCK(pr);
27     pr->callee_fr=fr; return SUSPEND;
28    }
29   }
30   pr->ltq_tail = ltq_ptr;   /* LTQ 操作*/
31   /* 逐次メソッド呼び出し */
32   y = f__fib(pr, n-2);
33   if((y==SUSPEND)   /* サスペンドチェック */
34      && (callee_fr=pr->callee_fr)){
35    f_frame *fr = MAKE_CONT(pr,c__fib);
36    // 継続を保存
37    callee_fr->caller_fr = fr;
38    SUSPEND_IN_JOIN_BLOCK(pr);
39    pr->callee_fr = fr; return SUSPEND;
40   }
41   /* join ブロック終了時の処理 */
42   if(*(pr->js_top)){
43    WAIT_FOR_ZERO(pr->jf);    /* 同期 */
44    if(x_pms) x = x_pms->ret.i;
45   }
46   pr->js_top--;   /* join スタック操作 */
47   return x+y;
48  }
49 }
```

図 6 `fib` からコンパイルされた遅延を利用した擬似 C コード
Fig. 6 Compiled C code for `fib` with laziness (pseudo).

遅延できていない．スレッド生成コストを逐次呼び出しのそれに限りなく近づけるには，残り 3 つの操作についても適切な修正が必要である．

重要な点は，子スレッドが親スレッドから通常の関数呼び出しにより呼び出されていることである．もし子スレッドが，ブロックせずに（スタック上で）最後まで実行されるなら,(a) スレッドの返り値は，プレースホルダではなく，関数の返り値によって渡せる,(b)

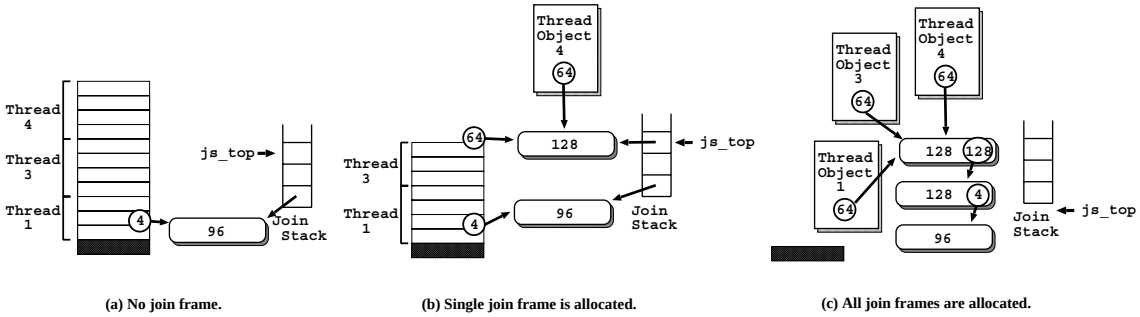

図 7 join フレーム確保の遅延
Fig. 7 Lazy join frame allocation.

親スレッドが join ブロックの終わりに到達する前に，必ず子スレッドはその実行を完了するので，重みを切り分ける必要がない．図 6 では，15 行目で，スレッドの返り値は直接 x にセットされる．子スレッドがブロックした場合にだけ，同期処理およびプレースホルダのためのフレームがつなげられ（18 行目），かつその場合だけ親スレッドはプレースホルダへのポインタをたどり返り値を取り出す（44 行目）（重みは，ブロックした時点（すなわちポーリングの直後）で切り分けられ，子スレッドのスレッドオブジェクトにセットされている）．

こうしてみると，スレッド生成ごとに生成されるスレッドオブジェクトの中身は（少なくとも現実装において）まったく使用されておらず，スレッド ID としてだけ存在していることに気付く．スレッドオブジェクトは，対応するスレッドがスタックに生存している間だけ一意的であればよい．スレッド生成のたびに，何度もタスクキューの同じ場所にスレッドオブジェクトを確保するのは冗長である．そこで，スレッド終了時にはスレッドオブジェクトを解放しないことにし，タスクキューの中に保持しておいて，次のスレッド生成のために再利用することにする．スレッドオブジェクトはタスクキューの初期化時に確保され，スレッドがスタックから出るときにスレッドオブジェクトも一緒に持っていくので代わりのスレッドオブジェクトが確保される．プロセッサがタスクをスタックに戻すとき，タスクキューの先頭に元からあったスレッドオブジェクトは解放してしまえばよい．

最終的に，サスペンドが起こらない場合，スレッド生成コードは 14–16 行および 30 行目へと減らすことができた．いいかえると，逐次呼び出しと比べたオーバヘッドは，タスクキューポインタのインクリメント/デクリメントとサスペンションチェックだけである．

4.2 join フレーム確保の遅延

前節では，重みを必ず切り分けることにはならないという遅延正規化手法について触れた．本節では，この遅延正規化手法を join ブロックの開始/終了処理にも拡張する．

前に述べたように，スタック上の親スレッドと子スレッドの間には暗黙の同期が存在している．そのため，ある join ブロックで同期するすべてのスレッドが同じスタック上に存在する限り，この join ブロックにはカウンタを使う必要がない．そうした場合，対応する join フレームは join ブロックの入れ子の深さを維持するためにしか使われていない（join フレームが外側 join フレームへのポインタフィールドを持つことを思い出してほしい）．そこで，join フレームを確保する代わりに，join ブロックの入れ子の深さを維持するためのスタックをプロセッサごとに用意することにする．深さを表す単純なカウンタではなく，スタック（**join スタック**と呼ぶことにする）を用いる理由は，上記の条件が満たされなくなったときに join フレームを確保し，それを join スタックに入れたいからである．

図 4 の fork/join 構造は，**図 7** へと改良される．最初，join スタックは空である．何度か join ブロックに入ると，必要な処理は top ポインタをインクリメントするだけであり（図 6 8 行目），状態は図 7 (a) のようになる（join スタックの底に 1 つ join フレームが存在するが，それはスタックに戻されたタスクがすでに持っていたものである）．join フレームを生成せずに join ブロックを出る場合，42 行目のチェックを通過し，top ポインタをデクリメントするだけである（46 行目）．あるスレッドがブロックし，かつ join スタックの top に join フレームへのポインタがなかった場合，新しく join フレームを確保し，スタック top にセットし，重みを切り分ける（図 7 (b)）スティールのため継続を抜き取るときは，すべての join フレームが

確保され，すべてのスレッドのために重みが切り分けられる（22–23 行目）．さらに，自身のローカルな join ブロックの中で実行中だった関数は，ヒープ中の入れ子 join ブロック構造を保つために join フレーム間をポインタでつなげる必要がある（26，38 行目）．最終的な join ブロックの状態は，図 7 (c) のようになる．

5. 関連研究

低コストなスレッド生成/同期処理を，自動的な負荷分散とともに実現しているマルチスレッド言語あるいはマルチスレッディング・フレームワークは数多く存在する．ここでは，それらの言語/フレームワークを大まかに 2 つのカテゴリに分類する．1 つは，制限された並列性のみをサポートするものであり，もう 1 つは，任意の並列性をサポートするものである．

WorkCrews [9] は，fork/join 並列処理を効率良く，かつ移植性の高い方法で制御するためのモデルである．WorkCrews では，スレッド生成時，スティール可能な実体（つまりタスク）を生成し，親スレッドの実行を続ける．親スレッドが同期ポイントに到達したとき，まだそのタスクが盗まれていないなら，親スレッドがそれを逐次的に呼び出す．タスクが盗まれていたら，盗まれたタスクが同期するのを待って親スレッドはブロックする．このモデルでは，いったん親スレッドが子スレッドを逐次に呼び出してしまうと，たとえ子スレッドがブロックしても親スレッドへコンテキストスィッチできないことに注意してほしい．そのため，このモデルはうまく構造化された fork/joiin 並列処理にのみ適応可能である．

Lazy RPC [8] は C 言語をベースにした並列関数呼び出しの実装手法である．Lazy RPC は WorkCrews と同じような手法を用いており，並列呼び出しに同様の制限がある．

FJTask [18),19)] は細粒度 fork/join マルチスレッドのための Java ライブラリであり，その手法は Lazy RPC と似ている．

WorkCrews，Lazy RPC，FJTask はどれも制限された（うまく構造化された）fork/join 並列処理を扱うため，他のタスクにすでに使用されているスタックを使ってタスクの実行を開始できる．したがって，タスク生成コストは，オブジェクトアロケーションのコストと同程度である．

Cilk [6),7)] は fork/join 構文による拡張 C 言語であり，OPA 同様，その実装は Cilk から C へのコンパイラ（とランタイム）である．その実装手法もまた，LTC ライクなスティール処理に基づいている．しかしながら，いくつかの点で OPA とは異なる．まず，join 構文はレキシカルスコープであり，また，他の種類の同期処理をサポートしない．これらにより子スレッドの管理が簡単化される．2 つめに，ベース言語が C であるため例外処理機能を提供しない．3 つめに，`synchronized` 構文を持たないためスレッド ID を管理する必要がない．4 つめに，Cilk ではスティール処理のため，すべてのスレッド生成時に親スレッドの継続をヒープに確保されたフレームへ保存する．Indolent Closure Creation [20)] は Cilk の実装の変形で，LTC を行うのに OPA と同様のポーリング手法を用いている．OPA と異なるのは，victim が実行を続ける際，盗まれたもの以外のすべてのタスクを使ってスタックを再構築する点である．

Cilk のようにレキシカルスコープにより join 先が決まるのと比較して，OPA の動的スコープにより join 先が決まる方が，より応用が利くと我々は考えている．なぜならスレッドの join 先というのは，本来，関数呼び出しの caller（リターン先）や，さらには例外ハンドラ（throw 先）等と同じように動的に求まるものだからである．さらに，Cilk では負荷分散の性能低下を避けるため，いくつかスレッドを fork するような関数は必ず並行に呼び出さなければならない．なぜなら，逐次に呼び出すと，callee から fork した全スレッドが完了するのを待たずに caller が実行を続けられなくなるからである．

レキシカルスコープによる join 先の決定は，プログラミングに制限を設ける代わりに，Cilk の実装を単純で効率良いものとしている．それに比べて，OPA では遅延を利用することにより動的スコープによる join 先の決定を実現しており，プログラマに何の制限も課さない．また，OPA でも「レキシカルスコープな」スタイルのプログラムを記述することは可能であり，コンパイラにより，プログラムがレキシカルスコープなスタイルに従っているかを確かめられる．

LTC [14)]（および，メッセージパッシング LTC [16)]）は，Multilisp のための効率良い実装手法である．Multilisp は，future（と暗黙の touch）構文により動的なスレッド生成と一般的な同期処理の機能を提供するが，fork/join 構文を持たないため，正しいプログラムを書くためにはプログラマにある程度のスキルが必要となる．スティール処理のためのスタックの操作はアセンブリ言語レベルで実装されており，移植性の面で制限がある．LTC を実現するのにポーリングを利用するのは，もともとメッセージパッシング LTC で提案された．

表 1 計算機環境
Table 1 Computer settings.

マシン	Sun Fire 3800
CPU	Ultra SPARC III 750 MHz, 8 MB L2 キャッシュ
メインメモリ	6 GB
CPU 数	6
コンパイラ	gcc 3.0.3（`-O3 -mcpu=ultrasparc` オプション）

StackThreads/MP[10] もまた，スタックベースのマルチスレッド手法である．StackThreads/MP では，あるプロセッサがスティール処理のために他プロセッサのスタックにあるフレームを使用することを可能にしている．また，ヒープに確保されたフレームなしで，一般的な同期処理を実現している．しかし，これを実現するため，実装は共有メモリ型マルチプロセッサ上のみに限り，またスタック/フレームポインタのアセンブリ言語レベルでの操作を行う．

これらの言語と比較して，OPA は両方のカテゴリの利点を備えているといえる．すなわち，簡潔な fork/join 構文，高い移植性，一般的な同期処理である．さらに，例外処理や `synchronized` メソッドといった，その他の先進的な機能もサポートしている．

6. 性能評価

この章では，OPA の実装の性能評価を行い，マルチプロセッサ上の細粒度マルチスレッド言語の優れた実装として知られる Cilk 5.3.2[6] との比較を行う．測定に用いた共有メモリ型並列計算機の構成は，**表 1** である．

性能評価には，Cilk ディストリビューションに含まれる Cilk ベンチマークプログラムのうち，5 種類のプログラム（`fib`，`knapsack`，`cilksort`，`matmul`，`heat`）を OPA に移植して使用した．

表 2 に，SMP 上での実行結果を示す．また，**表 3** には，OPA 処理系で実行した際のプログラムの実行全体におけるスレッド生成回数，タスク生成回数，およびスティール回数を数えた結果も示す．回数の測定は，OPA コンパイラが生成する C コードに対し，各イベントに応じてカウンタを増やすコードを追加することにより行った．すべてのプログラムにおいて，カウントするコードを追加したことによる実行時間に対するオーバヘッドは 2–5%に抑えられており，そのプローブ効果は無視できる程度である．プログラム全体でのスレッドの個数は，プロセッサの台数には関係なく一定である．`heat` プログラムにおいてスティールの回数（やタスク生成回数）が多いのは，`heat` が，プログラムの一番外側でバリア同期をとりながら少しずつ計算を進めるため，その反復回数（40 回程度）に比例するからと思われる．

図 8 に，各プログラムの逐次 C バージョン（Cilk プログラムから `spawn`，`sync` 等の Cilk 拡張を取り除いたもの）の実行時間を 1 としたときの，OPA プロ

表 2 絶対実行時間（括弧内は C との相対時間）
Table 2 Absolute execution time (and relative time to C in parentheses).

(sec)

# of PEs		1	2	3	4	5	6
fib(38)	C	3.36	-	-	-	-	-
	Cilk	12.1 (3.6)	6.06	4.17	3.03	2.44	2.02
	OPA	6.80 (2.02)	3.39	2.26	1.70	1.36	1.13
knapsack	C	5.03	-	-	-	-	-
	Cilk	9.05 (1.80)	4.64	3.17	2.36	1.86	1.55
	OPA	7.42 (1.48)	3.84	2.63	1.79	1.35	1.17
cilksort	C	2.29	-	-	-	-	-
	Cilk	3.42 (1.49)	1.72	1.16	0.89	0.72	0.62
	OPA	2.91 (1.27)	1.42	0.99	0.72	0.61	0.52
matmul	C	5.02	-	-	-	-	-
	Cilk	7.74 (1.54)	3.83	2.62	1.93	1.56	1.35
	OPA	7.12 (1.42)	3.48	2.34	1.77	1.42	1.19
heat	C	6.49	-	-	-	-	-
	Cilk	6.58 (1.01)	3.35	2.32	1.77	1.48	1.27
	OPA	6.63 (1.02)	3.31	2.05	1.61	1.38	1.20

表 3 スレッド生成，タスク生成，およびスティール回数
Table 3 The number of counts of thread creation, task creation and steal.

# of PEs		1	2	3	4	5	6
fib(38)	thread	63,245,985					
	task	0	91	160	253	423	548
	steal	0	9	19	37	56	79
knapsack	thread	26,839,428					
	task	0	99	223	294	379	608
	steal	0	14	25	39	46	77
cilksort	thread	7,072,482					
	task	0	104	774	360	1,177	1,434
	steal	0	21	176	84	265	321
matmul	thread	23,967,450					
	task	0	82	964	850	1,872	2,646
	steal	0	25	249	163	334	522
heat	thread	171,990					
	task	0	82	450	730	965	1,252
	steal	0	82	489	835	1,113	1,471

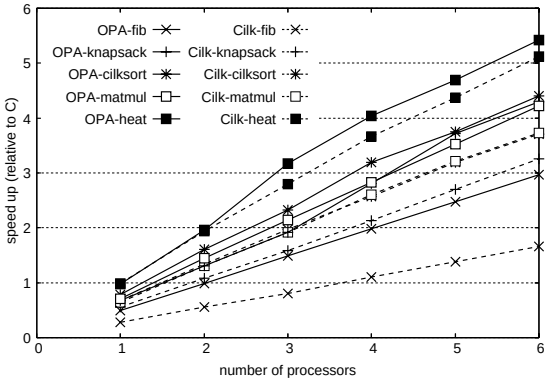

図 8 （C の実行時間を 1 とした）台数効果
Fig. 8 Speedup (relative to sequential C code).

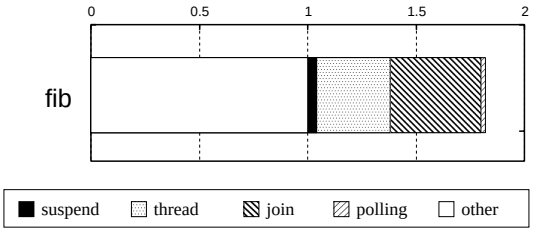

図 9 単一プロセッサ上での `fib` の実行におけるオーバヘッドの詳細
Fig. 9 Breakdown of overhead for `fib` on 1 processor.

グラムおよび Cilk プログラムの台数効果を示す．

OPA，Cilk ともに，すべてのプログラムにおいて優れた台数効果を得ていることが分かる（6CPU で 5–6）．ただし，表 2 からも分かるように，OPA 処理系の方が絶対的な実行時間において，Cilk 処理系の性能を上回っていることが分かる（`heat` プログラムについては，スレッドの粒度があまり細かくないため Cilk，OPA ともに絶対的な実行時間は C とほとんど変わらない）．`cilksort`，`matmul` プログラムにおける両者の性能差は，`fib`，`knapsack` に比べ，あまり顕著ではない．これは，`cilksort` と `matmul`（と `heat`）は，プログラムにおいて配列を使用していることによるものと思われる．具体的には，もとの Cilk プログラムにおいて：

```
int A[N];
f(&A[N/2]);
```

等として部分配列（へのポインタ）を引数として渡せるのに対し，OPA は Java の拡張であるため：

```
int[] A = new int[N];
f(A, N/2);
```

のように，配列オブジェクト全体（への参照）と部分配列のインデックスの 2 つを引数として渡さなければならない．これらの関数本体での配列要素へのアクセスでも余分なコストが必要となる．これらのマルチスレッドとは関係のないオーバヘッドが，OPA 処理系と Cilk 処理系の性能差を埋めているものと思われる．

Cilk の論文[7] では，C と比較した `fib` の相対実行時間は 3.63 となっており，本論文での Cilk の結果とほぼ同じである．Cilk の論文[7] によると，そのうちヒープフレームの確保にかかるオーバヘッドがおよそ 1.0，THE プロトコルに 1.3 となっている．THE プロトコルは，タスクキューにおける thief のスティールと victim のタスクの取り出し間での競合を解決するための，Cilk 処理系が用いる lock-free なプロトコルである．

OPA の場合，C と比較した `fib` の相対実行時間は 1.82 である．オーバヘッドの詳細を**図 9** に示す．全体の実行時間が Cilk よりも短い理由は，主に，ヒープフレームの確保を遅延していることと，thief と victim の間の競合をポーリング解決していることによる．つまり，OPA 処理系では，ヒープフレーム確保（盗むことのできる継続の生成）および THE プロトコルのオーバヘッドがなく，代わりに，メソッド（スレッド）呼び出しごとのサスペンドチェック（suspend）におよそ 0.04，ポーリング（polling）におよそ 0.02 とわずかなオーバヘッドしか必要としない．StackThreads/MP も OPA 処理系と同様の手法を用いているが，Cilk 処理系と同程度の性能しか達成できていない．これは，StackThreads/MP では，スレッドが直接その親スレッドに値を返したり，親スレッドへ同期したりできないことによるものと思われる．それらの操作は明示的に記述しなくてはならず，余分なオーバヘッドが加わる．

実際には，OPA と Cilk の性能を比べる場合，OPA が Cilk よりも豊かな表現力を持つことも考慮する必要がある．まず，OPA における `fib` の 2 つめの再帰呼び出しは逐次呼び出しとして表現でき，スレッド生成にかかるコストを避けられる．次に，Java スタイルのロック，動的スコープによる同期，スレッドのサスペンドといった，先進的な機能をサポートするため，OPA 処理系には余分なオーバヘッドが必要である．具体的には，タスクキューの操作（thread）におよそ 0.34，join スタックの操作と同期のためのカウンタチェック（join）におよそ 0.42 のオーバヘッドがかかる．これらの余分なオーバヘッドにもかかわらず，我々の OPA 実装は，遅延を利用することで，全体として Cilk よりも少ないオーバヘッドですんでいる．

最後に，遅延を利用しない従来の OPA 処理系の上で `fib` を実行し，その実行時間について，今回の遅

表 4 従来の OPA 処理系との比較
Table 4 Comparison with the conventional OPA implementation.

(sec)

	A	B	C	D	Cilk
fib(32)	16.3	0.86	0.72	0.37	0.67
fib(38)	292（推定値）	15.4	12.9	6.8	12.1

延を利用する OPA 処理系との比較を行った．比較にあたり，本論文が提案する遅延可能な処理，すなわち，ヒープフレームの確保，タスク生成，スレッドオブジェクトの生成，join フレームの確保について，A）メソッド呼び出しにおけるヒープフレームの確保のみ遅延，B）さらにスレッド生成におけるヒープフレームの確保とタスク生成を遅延，C）さらにスレッドオブジェクトの生成を遅延，D）さらに join フレームの確保を遅延，と段階的に提案手法を適用させたバージョンを用意することで，各手法の有効性を検証した．結果は**表 4** のとおりである（バージョン A では，メモリ不足のため `fib(38)` を計算できず，そのため `fib(32)` でも測定した．バージョン A の `fib(38)` にある値は，他のバージョン（と Cilk）の `fib(38)`/`fib(32)`（ほぼ同じ値 18）から求めた推定値である）．なお，単一プロセッサ上での結果のみを示してあるが，マルチプロセッサ上での実行においてすべてのバージョンでほぼ線型な台数効果を得られた．この表から，まずタスク生成処理の遅延の効果が非常に大きいことが分かる．これは，Cilk 処理系や遅延を利用する OPA 処理系と異なり，従来の OPA 処理系では親子スレッドの実行順序を逆転させていないため（つまりプロセッサは親スレッドの実行を続ける），スレッド生成ごとに子スレッドに対応するタスクを生成しており，さらに，そのスケジューリングが `fib` のような fork/join 型プログラムに適さないため，頻繁にコンテキストスイッチが生じるせいである．次に，バージョン B，C から，先進的な機能を遅延を利用せずにサポートする際のオーバヘッドが比較的大きく，実際，これらを上記の遅延した場合の操作（図 9 の thread，join）に変えなければ，Cilk の単一プロセッサでの性能にかなわない．

7. おわりに

本論文では，OPA の fork/join 構文のための高効率で移植性の高い実装手法を提案した．OPA 言語は，相互排他，Java の `synchronized` メソッド，例外処理と結び付いた動的スコープによる同期といった先進的な機能をサポートする．

これらの機能をサポートすることは，たとえば遅延タスク生成のような，細粒度 fork/join マルチスレッド言語のこれまでの高効率な実装手法の効果を低下させると考えられてきた．我々の提案する実装手法は，スティール可能な継続の生成，スレッドオブジェクトの確保，join フレームの確保といった操作に「遅延」を利用することで，性能の低下を防いでいる．

OPA の実装を Cilk の実装と比較し，OPA プログラムの性能が Cilk プログラムの性能を上回ることを確認し，我々の手法の有効性を示せた．

謝辞 本研究は，一部，研究拠点形成費補助金（課題番号：14213）の助成を受けている．

参考文献

1) 八杉昌宏，瀧 和男：並列処理のためのオブジェクト指向言語 OPA の設計と実装，情報処理学会研究報告 96-PRO-8(SWoPP '96)，Vol.96, No.82, pp.157–162 (1996).
2) Yasugi, M., Eguchi, S. and Taki, K.: Eliminating Bottlenecks on Parallel Systems using Adaptive Objects, *Proc. International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, pp.80–87 (1998).
3) 江口重行，八杉昌宏，鎌田十三郎，瀧 和男：適応的オブジェクトによる排他制御の実行時緩和，情報処理学会論文誌，Vol.40, No.5, pp.2084–2092 (1999).
4) 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.40, No.SIG 4 (PRO 3), pp.44–57 (1999).
5) 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).
6) Supercomputing Technologies Group: *Cilk 5.3.2 Reference Manual*, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, USA (2001).
7) 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).
8) Feeley, M.: Lazy Remote Procedure Call and its Implementation in a Parallel Variant of C, *Proc. International Workshop on Parallel Symbolic Languages and Systems*, LNCS, No.1068, pp.3–21, Springer-Verlag (1995).
9) Vandevoorde., M.T. and Roberts, E.S.: WorkCrews: An Abstraction for Controlling Parallelism, *International Journal of Parallel*

Programming, Vol.17, No.4, pp.347–366 (1988).
10) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPoPP*), pp.60–71 (1999).
11) Wagner, D.B. and Calder, B.G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proc. Principles and Practice of Parallel Programming* (*PPoPP'93*), pp.208–217 (1993).
12) Plevyak, J., Karamcheti, V., Zhang, X. and Chien, A.A.: A hybrid execution model for fine-grained languages on distributed memory multicomputers, *Proc. 1995 Conference on Supercomputing* (*CD-ROM*), p.41, ACM Press (1995).
13) Bevan, D.I.: Distributed garbage collection using reference counting, *PARLE: Parallel Architectures and Languages Europe*, LNCS, No.259, pp.176–187, Springer-Verlag (1987).
14) 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).
15) Halstead, Jr., R.H.: MULTILISP: a language for concurrent symbolic computation, *ACM Trans. Programming Languages and Systems* (*TOPLAS*), Vol.7, No.4, pp.501–538 (1985).
16) Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proc. International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, LNCS, No.748, pp.94–107, Springer-Verlag (1993).
17) Feeley, M.: Polling Efficiently on Stock Hardware, *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp.179–190 (1993).
18) Lea, D.: A Java Fork/Join Framework, *Proc. ACM 2000 Conference on Java Grande*, pp.36–43, ACM Press (2000).
19) Lea, D.: *Concurrent Programming in Java: Design Principles and Patterns*, 2nd edition, Addison Wesley (1999).
20) Strumpen, V.: Indolent Closure Creation, Technical Report MIT-LCS-TM-580, MIT (1998).

(平成 15 年 9 月 22 日受付)
(平成 15 年 12 月 2 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 年同大学大学院情報学研究科修士課程修了．同年より同大学院情報学研究科博士後期課程に在籍．並列/分散処理，プログラミング言語処理系に興味を持つ．

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 年〜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．2003 年より同大学助教授．博士（理学）．1998 年〜2001 年科学技術振興事業団さきがけ研究 21 研究員．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 年同大学院情報学研究科通信情報システム専攻助手．2003 年より豊橋技術科学大学情報工学系講師．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（フェロー）

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，並列処理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．