オブジェクト指向並列言語OPAのためのコード生成手法

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[†]

MIMD 型並列計算機における効率の良い並列処理のための，メソッドの実行時置換と構造化されたスレッドによる並列処理を特徴とするオブジェクト指向並列言語 OPA を開発している．本論文では，その共有メモリ型並列計算機用のコード生成手法について述べる．コンパイル時には，オブジェクトへのメモリアクセスやコンテキストスイッチ時のメモリアクセスを削減するための解析を行う．また実装方式としては，プロセッサ間通信とスレッドスケジューリングには lock-free バッファ管理方式，スレッド内スケジューリングには関数フレーム二重表現方式と値ベースサスペンドチェック方式，同期処理には重み付きカウント方式などを用いている．値ベースのチェックにより，1 呼出しあたり 1 分岐命令追加程度のオーバヘッドで高速なコンテキストスイッチを可能とした．

Code Generation Techniques for an Object-oriented Parallel Language OPA

Masahiro Yasugi,[†,††] Seiji Umatani,[†] Tomio Kamada,[†††]
Yusuke Tabata,[†] Tomokazu Ito,[†] Tsuneyasu Komiya[†]
and Taiichi Yuasa[†]

We are developing an object-oriented parallel language OPA for efficient parallel processing on MIMD computers, which features dynamic method replacement and structured parallel processing using multiple threads. In this paper, the code generation techniques for shared-memory parallel computers are presented. The compiler performs analyses to reduce the memory access to objects and the memory access on context switches. The implementation techniques include a lock-free buffering method for inter-processor communication and thread scheduling, a double function-frame representation method and a value-based suspension check method for intra-thread scheduling, and an weighted counting method for synchronization. The value-based check enables fast context switches with small overhead for an additional branch instruction per call.

1. はじめに

近年，MIMD 型並列計算機を用いた並列処理システムは，高い計算能力を従来のベクトル計算機より安価に提供するものとして注目されてきた．しかし，MIMD 型並列計算機には，いくつもの異なった並列アーキテクチャがあり，現状では主流となる特定の並列アーキテクチャが存在しないという問題がある．このような現状においては，プログラミング言語により並列アーキテクチャの違いを吸収すること（抽象化）は重要となる．

そこで本研究では，並列処理，特に，不規則な並列性を含む問題の記述を容易とするプログラミング言語の研究，及びその効率良い実行を可能とする言語処理系の実装技術の研究を行っている．近年注目されているオブジェクト指向言語である Java 言語からスレッドに関する仕様を取り除き，代わりに，構造化された並列構文を含んだスレッドに関する仕様を追加することで並列プログラミングの記述を容易とした，オブジェクト指向言語 OPA（an Object-oriented language for PArallel processing）を設計している[19]．また，並列

† 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
†† 科学技術振興事業団さきがけ研究 21「情報と知」領域グループ
"Information and Human Activity", PRESTO, Japan Science and Technology Corporation(JST)
††† 神戸大学工学部情報知能工学科
Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University

処理に必要な，一貫性制御，スレッド管理に関する実装技術を研究し，共有メモリ型並列計算機における評価を行っている[18]．具体的には次の通りである．(1) Java 言語からオブジェクト指向の機能を引き継ぐとともに，より自然なスレッド利用の機能やオブジェクトデータの一貫性制御の機能を実装・評価し，並列処理における本言語の使い易さを示す．(2) スレッド管理方式，一貫性制御実装方式，実行時メソッド置換機能を実装・評価し，高い処理性能が得られることを実証する．

本研究では上記のような目的で OPA コンパイラを含む処理系を開発している．OPA コンパイラは OPA のプログラムから C 言語のプログラムへのトランスレータとして実装されている．本研究では性能も強く追求しており，固定的なランタイムは最小限とし，重要な処理はコンパイラが生成する C のコード中で行なうようにしている．本論文では，そのコード生成手法を中心に述べる．

以下，2 節でオブジェクト指向並列言語 OPA の言語設計についてまとめた後，3 節で共有メモリ型並列計算機における実装技術，4 節で今回中心となるコード生成手法，5 節で評価結果を示し，6 節で関連研究などを議論し，7 節で結論を述べる．

2. オブジェクト指向並列言語 OPA

OPA は Java 言語を並列処理用に改良したオブジェクト指向言語であり，プログラム例は図 **1** のようになる．オブジェクトとスレッドを分離させ，オブジェクトに対する複数スレッドからの並行アクセスを排他制御しオブジェクトのデータの一貫性を保つというモデルによって並列処理を行なう．

並列処理には様々な分類方法が存在するが，OPA ではスレッド間の関係に応じて並列処理を，(1) 分担型並列処理，(2) 協調型並列処理，(3) 排他型並列処理，の三つの型に分類し，それぞれを直接的に記述できるようにする．(1) 分担型並列処理は，図 **2** のようにスレッド間に fork と join の関係があり，並列処理の構造化された記述が可能である．分担型並列処理で捉えられない (2) 協調型並列処理では，図 **3** のようにスレッド間では単に通信/同期がとられ，通信/同期をした後もそれぞれのスレッドは実行を続ける．通信/同期はオブジェクトを介して行うこととし，そのためのクラスを用意する．(3) 排他型並列処理では，図 **4** のように他のスレッドを排除して実行できる区間を設けられるようにする．

(3) 排他型並列処理については，instant メソッドと

```
class BinTree{
 internal int key, value;
 internal BinTree left = null,
                  right = null;
 public BinTree(int k, int val){
   key = k; value = val;
 }
 public instant readonly
  int search(int k, int v){ ... }
 public replaceable instant
  void insert(int k, int val){
    if(k < key){
      if(left != null)left.insert(k, val);
      else{
        left = new BinTree(k, val);
        if(right != null)
          setmethod(insert, insertF);
      }
    }else{
      if(right != null)right.insert(k, val);
      else{
        right = new BinTree(k, val);
        if(left != null)
          setmethod(insert, insertF);
      }
    }
  }
 substitutive instant readonly
  void insertF(int k, int val){
    if(k < key)left.insert(k, val);
    else right.insert(k, val);
  }
}
```

図 1 OPA のプログラム例 (二分木辞書)
Fig. 1 An example of OPA programs (binary tree dictionary)

いうメソッドを処理単位の 1 つとして，オブジェクトデータの一貫性制御のための排他制御を最低限に抑える．instant メソッドはさらに，読み出し専用 (RO) メソッドと読み書き両用 (RW) メソッドに分類され，また，instant メソッドによるオブジェクトデータに関する読み出しや書き込みをそれぞれメソッド開始点や更新の可能性が消失する点で一括して行うものとする．これにより，RO メソッドの並列実行を許すとともに，RO メソッドの non-blocking な実行が可能となる．

オブジェクトには，適切な書き込みの後に更新されなくなるという性質をもつものが多くあり，更新を行なわなくなったメソッドを RO メソッドに置換することで，処理の高速化が図れる．OPA では，メソッドの実行時置換によりこのようなプログラミングが可能である．

2.1 分担型並列処理

処理しようとする問題がいくつかの部分問題に不規則に分割可能である場合に，それぞれの部分問題を分担して並列に処理するようにプログラムを記述す

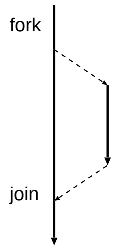

図 **2** 分担型並列
Fig. 2 Fork-Join parallel

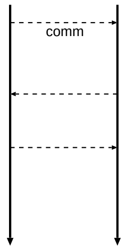

図 **3** 協調型並列
Fig. 3 Cooperative parallel

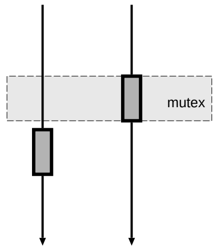

図 **4** 排他型並列
Fig. 4 Mutually exclusive parallel

ることは，従来の言語においては必ずしも容易とはいえない．OPA では，そのような分担型並列処理を，fork-join を用いて容易に誤りなく記述できるようにする．

問題を処理しているとき，ある部分で並列に実行できる部分問題が複数ある場合は，必要個数のスレッドを生成（fork）し，それに部分問題を解かせることで，並列処理を行なう．新たに生成されたスレッドも，さらにスレッドを生成することで，与えられた部分問題をより並列に処理でき．入れ子構造で並列処理を記述できる．join ブロック内で生成された全てのスレッドは，ブロックの終了地点で同期（join）がとられる．（図 **5**）．

スレッドの fork と join には，キーワード `par`，`join` を以下のように用いる（図 5 に対応）

```
void f0(){
  join{
      par f1();     // 並列実行
      par f2();     // 並列実行
        ...
  }                 // ここで同期
}
void f1(){
   par f1_1();      // これも join する
   ...
}
void f2(){
  join{            // 入れ子の join
    par f2_1();
    ...
  }
}
```

fork させたいメソッド呼出しに `par` を付加すると，新たにスレッドが生成され，そのメソッドを新しいスレッドで実行することができる．また，join ブロック内で fork されたスレッドは，その join ブロック内で終了するため，処理が終了した後もスレッドが生き残ることはない．ここで，`par` は実行時のスレッド生成操作であり，スレッドが問題のどの部分を担当するかを動的に決められるが，`join` については，実行時にスレッドを指定して個々のスレッドの完了を待つための同期操作なのではなく，その内部で生成された全スレッドの完了を待つための同期構文となっている．また，スレッドが生成されたときの join 先は動的スコープで参照される[10),16)]．

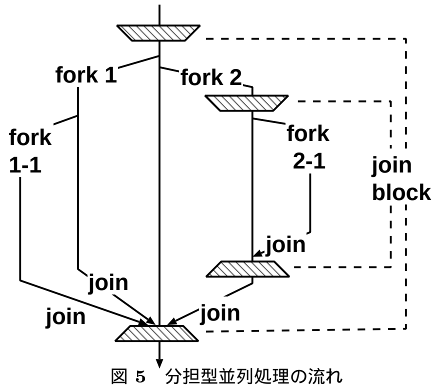

図 **5** 分担型並列処理の流れ
Fig. 5 Threads of fork-join parallel processing.

また，スレッドの計算結果を受け取る必要があれば次のように書く．

```
{
  int x = par obj1.m();     // 並列実行
  int y = par obj2.m();     // 並列実行
        ..
join:                       // ここで同期
  r = x + y;
}
```

このように複文を「`join:`」で区切ることで，区切り以前が join ブロック内であるとして実行され，またその実行中に生成したスレッドの計算結果を区切り以降利用することもできる．

OPA では，以上のように分担型並列処理を構文でサポートすることにより，ある大きな問題に含まれる並列処理可能な部分問題を記述しやすく設計されているが，これにより並列処理中の例外処理も自然にできるようになる．つまり，部分問題を分担する各スレッド毎に例外が発生したときの処理を記述する必要は無く，並列に部分問題を解く上で発生した例外を一括して処理するように記述できたほうが自然である．そこ

で，OPA の例外処理は，join ブロック内の並列処理で発生する例外を join 先で受け止めることができるようにする．詳細は文献 10)，16) で述べられている．

2.2 一貫性制御とメソッド呼出し

オブジェクトのメソッドを分類して，それぞれに適した手順で呼び出すことで，メソッド呼出しを高速化することができる．例えば，オブジェクトのもはや変化しなくなった変数の値を読み出すだけのメソッドを，free なメソッドと呼ぶことにすると，free なメソッドがアクセスする変数の値をコピーしておくことで（特に分散環境では）高速なアクセスが可能になる．また，free なメソッドは全くの排他制御なしで起動できる．

OPA では，メソッドの定義にキーワード `instant` を付加することで，オブジェクトへのアクセスを一括して行うことができる．instant メソッドはさらに `readonly` キーワードの有無によって読み出し専用メソッド（RO メソッド）と読み書き両用メソッド（RW メソッド）に分類される☆．instant メソッドによる一括アクセスは文面上で読み書きされる変数に対して行い，RO メソッドについては読み出しのみであることをコンパイラが確認する．

RO メソッドについては，一貫性のとれた状態のオブジェクトのデータを読み出すことができれば，読み出した後のオブジェクトの排他制御は必要ない．RW メソッドについては，オブジェクトのデータの更新を，インスタンス変数への書き込みがある度に行なうのではなく，文献 9) のようにメソッドのある特定の点で一括更新することによって，排他制御区間を最小限に抑えることができる．instant メソッドでは実行に際してオブジェクトの必要なデータを一時変数に一括読み出し（コピー）し，メソッドの実行はこの複製（一時変数）☆☆を利用して行なう．オブジェクトへの書き込みもまずは複製（一時変数）に対して行ない，それ以降の書き込みの可能性が消えた点（OPA コンパイラが解析）でオブジェクト本体のデータを一括更新する．その際，値が更新された可能性のある一時変数（OPA コンパイラが解析）に関してのみオブジェクト本体の該当する部分のデータを更新する．更新の可能性を解析することで，すべての変数に関して更新する場合，

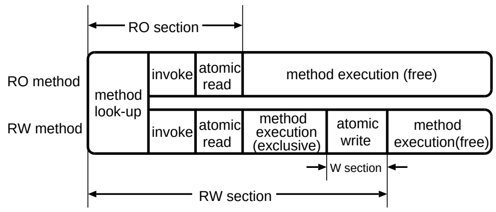

図 6　メソッド呼出しの手順
Fig. 6　Method invocation steps.

あるいは実行中に更新の有無を記録する場合と比較してオーバヘッドを削減している．これらの解析については 4.1.1 節，4.1.2 節で述べる．

次の一括更新までの間，RO メソッドは最後に更新された状態のデータが読み出せるものとすると，RO メソッドは必ずデッドロックなしで実行できることになり，RO メソッドと free なメソッドは性能上の違いはあれ，計算モデル上の違いはなくなる．free なメソッドは実装では高速化のために用いるが，言語として採用しない．

RO メソッドと RW メソッドにつき，メソッド呼出し手順（図 **6**）と排他制御する区間を整理すると次のようになる．

RO メソッド： (RO1) メソッドを取り出す．(RO2) メソッド本体の実行開始．(RO3) 一括してオブジェクトの必要なデータを読み出す．(RO4) メソッドの続きを実行する．

RW メソッド： (RW1) メソッドを取り出す．(RW2) メソッド本体の実行開始．(RW3) 一括してオブジェクトの必要なデータを読み出す．(RW4) 一括更新までのメソッドの処理を行う．(RW5) 一括更新を行う．(RW6) 一括更新後のメソッドの処理を行う．

以下では，RW1 から RW5 までの処理の区間を RW 区間，RO1 から RO3 までを RO 区間，RW5 を W 区間と呼ぶ．排他制御は，データを読み出し加工して書き戻すという（一貫性のとれた状態への遷移するための）一連の処理が持つ効果の不可分性・分離性を保証するため RW 区間の実行と RW 区間の他の実行の間に必要であり，また，オブジェクトから一貫性のとれた状態のデータを読み出すためには，W 区間実行と RO 区間実行の間にも必要となる．ここで，RO 区間の実行と RO 区間の他の実行の間の排他制御は必要ない．また，W 区間は RW 区間に含まれるため RW 区間に関する排他制御により自動的に W 区間の実行は排他的となる．ここで，メソッドを取り出す部分も区間に含まれるのは，後述の実行時メソッド置換のため

☆ 従来は `readonly` を用いずにコンパイラが自動的に分類するとしていたが，次の問題があった：(1) 継承時の上書きが許されない `final` メソッドの呼出しのメソッド検索などを省くといった最適化の際にその判定に手間がかかる．(2) 自動的に分類することでプログラマとコンパイラの認識が異なることにプログラマが気づきにくい．(3) `native` メソッドの解析ができない．

☆☆ 実装としては C コンパイラによりできるだけレジスタ上に複製（一時変数）が作られるように注意する．

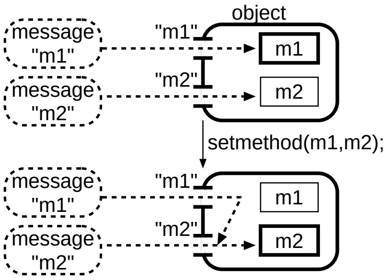

図 7 実行時メソッド置換
Fig. 7 Dynamic method replacement.

である．

これにより，メソッド呼出し全体を単位として排他制御する場合に比べ，排他制御の頻度や区間が短縮され（RW 区間実行の間の排他制御，RO 区間実行と W 区間実行の間の排他制御のみに短縮），排他制御の不要な部分については単一オブジェクトに関して同時に（並列に）複数のスレッドが動作することが可能になる．

2.3 実行時メソッド置換

オブジェクトには適切な更新の後，変化しなくなるという特性を持つものが多く見られる．また，これはプログラマに把握されていることが多い．これを利用し，更新を行なわなくなった RW メソッドを RO メソッドに置換するよう記述することで排他制御が必要な区間を縮小し，並列性を向上させることができる．これは，メソッド内部で処理を分岐させることでは得られない効果である．

実行時にメソッド置換を行うことで，排他制御を減らしたり，変化しない変数を増やすことができる．図 1 の二分木辞書の例では，左右の子ノードを生成した後は，登録時にインスタンス変数の更新を行わなくなるため，`setmethod(insert, insertF)` の部分でメソッド `insert` を RO メソッドの `insertF` に置換している．置換可能なメソッド `insert` は，キーワード `replaceable` を付けて定義される．なお，図 1 において，`substitutive` キーワードはメソッドが置換用メソッドであるので，`setmethod` をしないで直接呼び出すことはできないことを示し，文献 11)，12)，14)，15) で述べた free メソッド解析において解析精度を高めるために利用できる☆．また，`internal` キーワードはそのクラスのメンバにアクセスできるのはオブジェクトそれ自身（`this`）だけであることを示す．Java の修飾子 `private` では，他のクラスからのアクセスは禁止できるが，同じクラスの他オブジェクトからのアクセスは禁止できない．これにより，変数の値が更新される可能性についての解析を容易にしている．

☆ 従来は `substitutive` を用いずに普通のメソッドを用いていたが，`setmethod` されていない状態で呼び出されないことを直接示すことができなかった．間接的に示すには `internal` かつ `this` からの呼出しがないことを解析する必要があった．

多重定義されているメソッドは対応する置換可能メソッド，置換用メソッドの双方が存在するものを実行時メソッド置換の対象とする．また，継承時における上書き（override）は，置換可能メソッド，置換用メソッドそれぞれについて通常のメソッドと同様に行うことができる．ただし，継承時にサブクラスで多重定義として追加されたメソッドについては，実行時メソッド置換の対象としない．

OPA では，スレッド間での通信，同期のためにメソッド置換を利用できる．メソッド置換において，置換先のメソッド名にキーワード `suspends` を指定すると，そのメソッドを呼び出したスレッドはその場でサスペンドする．他のスレッドによってメソッドを通常のメソッドに置換することで，そのスレッドは実行を再開する．

オブジェクト指向の立場からメソッド置換をみると，これはオブジェクトに送られたメッセージの届き先をスイッチするという考え方ができる．メッセージを送る側から起動されるメソッドが特定できない点は通常の遅延束縛と同様に捉えることができる．この立場から見た `setmethod(m1, m2)` のメソッド置換の様子を図 7 に示す．

メソッド置換ではメソッドとインスタンス変数は合わせて一括更新するものとする．また，メソッド取り出しも 2.2 節で定義した排他制御に関する区間で行う．注意が必要な点は，メソッド置換の影響により，実行しようとしているメソッドが RO メソッドであるか RW メソッドであるかは実際にメソッドを取り出すときまでわからないにもかかわらず，メソッドを取り出す操作自身も排他制御が必要な点である．

3. OPA の実装手法

OPA のコンパイラは，OPA のソースコードを C のコードに変換するものである．また OPA のランタイムは，OS のシステムコール等により，各プロセッサに 1 つずつ，仮想メモリ空間を共有するプロセス（またはシステムレベルのスレッド）を生成する．OPA の言語レベルのスレッドのサスペンドや実行再開，他プロセッサへの移動は，OPA のランタイムに管理されるが，その大部分の処理は生成された C コードが行うようになっている．

3.1 join の実現

効率良く join の完了を判定するには，join フレームに重み付き参照カウンタ[1] を設け，重み付き参照カウンタが 0 になっているかどうかを調べれば良い．join フレームには，スレッドの計算結果を格納する場所や join の完了後の処理を続けるスレッドが待ち合わせる（サスペンドする）ための場所も準備する．実行時に新たなスレッドの fork を行うときには，そのスレッドはその親スレッドから join 先（join フレーム）の参照を重み付きで引き継ぐ．こうすることで join するスレッドの数が実行時にしか分からなくても効率的に join 操作ができる．重みが 1 となり再分割できなくなった場合は，間接参照カウント[4] の考え方と同様，新たに同期を中継するための join フレームを作り対処している．

また，新たに fork するスレッドが全て同一のプロセッサに作られる可能性を考えると，文献 7) で reply box に対して用いられているような lazy なアロケーションを，join フレームに対しても用いることができる．これについては現在実装中である．

3.2 スケジューリング

スレッドを他のプロセッサに生成したり，移動させたりするための通信バッファ（スケジューリングキューを兼ねる）の管理には，送信元・受信先のペア毎にキューを設けてロックを排した lock-free 方式[17] を用いる．本研究では，文献 17) の方式を改良し，送信側が必要なバッファをリンクしていくとともに，すでに受信されたバッファの回収も行う．また，バッファ内のどこまでを通信に使用したのかを示す変数をバッファと同じ構造体に置くことで，書き込みの局所性が高められた．

以降はスケジューラがスレッドをどのようにスケジュールするか，また，各スレッドの実行においてその関数フレームをどのように管理するかについて述べる．

3.2.1 スレッドのスケジューリング

基本的なスレッドのスケジューリングは，スケジューラが各スレッドのコードを呼び出すことで行う（実際には後述の関数フレームスケジューラを動かす）．この際，各スレッドのコードはスケジューラの C のスタック上で動作する．つまり，文献 7) と同様，各プロセッサのスケジューラは，その 1 つのスタックを OPA 言語レベルの多数のスレッドのために用いる．

スレッド生成時には他のプロセッサにスレッド生成することができる．スレッドの実行を開始するためのフレームを作り，通信バッファを兼ねたスケジューリングキューに入れることで実現される．なお，新たに fork するスレッドが全て同一のプロセッサに作られる場合，文献 7) のような lazy なヒープ上のアロケーションを，スレッドのフレームに対しても用いることができる．これについても join フレームの lazy なアロケーションと同様，現在実装中であり，別に発表する予定である．

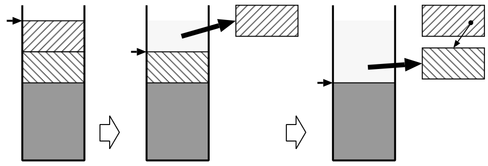

図 8 スレッドサスペンド時における関数フレームのリスト表現への変換
Fig. 8 Function frame conversion to a list representation on thread suspension.

3.2.2 関数フレームの管理

スレッドの関数フレームには文献 5)，7) のように 2 種類の表現を用いる．1 つは通常のように C のスタックの上にとる（C コンパイラが表現を決定する）．もう 1 つはフリーリストから割り当て，ヒープ上で子の関数フレームから親フレームへのリスト表現となるようにする．OPA では文献 5) と同様にそれぞれの表現に合わせて 2 つのバージョンのコード（関数）を生成する．スタック表現は新たなメソッド呼出しに用いられ，サスペンドしない限りそのまま用いられる．一方，リスト表現は呼出しのサスペンド/再開のために用いられる．

スレッド内で（新たに）OPA の関数（メソッド）を呼び出す場合は，単純にスタック用の C の関数を呼び出す．その結果として C のスタックの上に関数フレームが普通に確保される．できるだけスタック上でスレッドが動作するようにしてやることで，その性能を C 言語で普通に書いたプログラムに近づけることが可能となる．

スタックを用いている関数がサスペンドしなくてはならなくなった場合は，その関数はそのスタック表現に対応するリスト表現の関数フレームを作り C の返り値として親に返す．サスペンドせずにそのまま関数呼出しが完了した場合は C の返り値として 0 を返す．（0 でない）子の関数フレームが返されるのはサスペンドしなくてはならない場合なので，呼出し元（親）もスタック上で動作している場合はリスト表現の関数フレームを新たに作り，図 8 のように子の関数フレームからリンクされるようにすることで，リスト表現へ変換する．

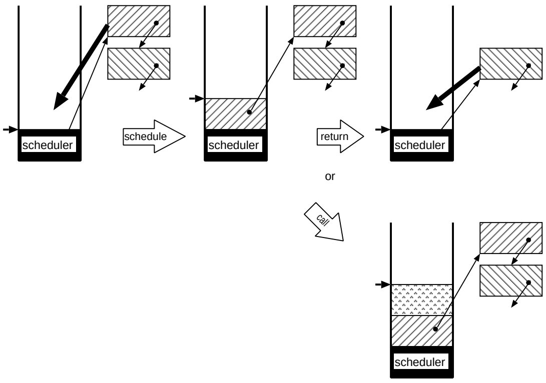

図 9　リスト表現の関数フレームを使ったスレッド実行再開とその後のリターンまたは呼出し
Fig. 9　Thread resumption with a list of function frames and subsequent return or call.

また，関数呼出しが完了して C の返り値として 0 を返す場合，OPA の返り値のほうについてはプロセッサ毎に決まった（メモリ上の）場所を使って返す．決まった場所としているため，文献 5) のように別途，返り値を格納する場所のアドレスを引数として渡してやる必要は無い．しかし，以上の方式では，OPA の返り値の型が `void` でなければ，返り値の授受にメモリアクセスが発生することになる．これについては 4.2 節で提案するように「値に基づくチェック」を用いて，できるだけこれを省くことを行っている．

リスト表現の場合，次のような単純な関数フレームスケジューラを用いてスレッドを実行する（**図 9**）．

```
while(fr){ fr = (fr->f)(pr, fr); }
```

子の関数は C の返り値として，図 9 上段のように次にスケジュールしてほしい親の関数フレーム（リストの次の要素）を返す（あるいは新たに関数フレームを作って返すことで新たな呼出しとすることもできる．）ここで，`fr` は OPA のリスト表現のヒープ上の関数フレームで再開（あるいは開始）を担う C の関数を `f` にとる．`pr` は各プロセッサ毎に必要なデータを保持する構造体で，OPA の返り値のための場所もここに確保される．`fr` として 0 が返されたときは，そのスレッドの実行は完了または中断した場合なので，次の実行可能なスレッドがスケジュールされるようにする．C の返り値の役割が，スタック表現の場合と異なっているが，別のバージョンのコードを用いるので問題ない．また，OPA の返り値のほうについてはプロセッサ毎に決まった場所を使って返し，次にスケジュールされる親の関数フレームの実行の際に読み出されることになる．

一度サスペンドしてリスト表現になったものをスタック表現に戻すことはできないが，リスト表現での実行中に新たな関数呼出しがあった場合は図 9 下段のようにスタック表現を用いるので，実際には，1 つのスレッドが，子のほうはスタック表現，親のほうはリスト表現となり得る．親がリスト表現で関数フレームスケジューラ上で子の関数呼出しの完了を待つ間に，子の関数呼出しがサスペンドした場合，親は自分のリスト表現の関数フレームの内容を更新し，かつ，その関数フレームが子の関数フレームからリンクされるようにした後，C の返り値として 0 を返す．この際，リスト表現で動作することになった子の呼出しが（関数フレームスケジューラ上で）完了したときに，適切に OPA の返り値を受け取って処理が続けられるように更新する必要がある．

3.3　一貫性管理と実行時メソッド置換の実装

オブジェクトのデータの一貫性のため，RW メソッドでは RW 区間–RW 区間の排他制御，RO メソッドでは RO 区間–W 区間の排他制御が必要である．RO 区間–W 区間の排他制御の実装には，ロックを必要としないバージョン番号方式を利用している．

排他制御の実装と実行時メソッド置換の実装については，文献 11)，12)，14)，15) に詳しく述べられているので省略する．今回，コード生成に必要な解析については次節で述べる．

4.　コード生成手法

4.1　OPA コンパイラの構成

OPA コンパイラは，構文解析部（字句解析部），意味解析部，中間コード生成部，中間コード解析部，C コード生成部で構成されている．構文解析部（字句解析部）は flex + bison を用いて開発されており，OPA コンパイラの残りの部分は Common Lisp により開発されている．

構文解析部（字句解析部）は構文木を S 式として出力し，意味解析部はそれを受け取り，名前のスコープの処理，型付け等を行う．中間コード生成部は一時変数等を導入して制御フローが明示された命令列形式の中間コードを生成する．中間コード解析部では，更新可能性消失点の解析，一括更新変数の解析，生きている変数解析を行い，コード生成に必要な情報を収集する．C コード生成部では 2 つのバージョンの C の関数を用いる C コードの生成を行う．

4.1.1　更新可能性消失点の解析

2.2 節で述べたように OPA の instant メソッドの読み書き両用メソッド（RW メソッド）では，オブジェクトへの書き込みをまず複製（一時変数）に対して行

ない，その以降の書き込みの可能性が消えた点でオブジェクト本体のデータを一括更新する．この更新可能性消失点はコンパイラにより解析する．

更新可能性消失点はいわゆるデータフロー解析で求めることができる．データフロー解析では，中間コードから基本ブロックの集合と基本ブロック間の制御フローを表す辺からなるフローグラフを作成する．基本ブロックの先頭・末尾における情報間の関係をデータフロー方程式として表し，これを反復法などで解くことで求める情報が得られる．

更新可能性消失点の解析では，フローグラフを用いて，RW メソッドの各実行経路において，インスタンス変数が最後に書き込まれる点，もしくは，それ以降インスタンス変数への書き込みが行われない区間の最初の点を求める．データフロー方程式は「以降の更新がある」という情報について連立させればよい．つまり，基本ブロックの末尾においては後続の基本ブロックの先頭の情報の論理和であるとし，基本ブロックの先頭の情報はブロック末尾の情報とブロック内更新有無との論理和として方程式を連立させればよい．

4.1.2 一括更新変数の解析

2.2 節で述べたように RW メソッドの一括更新の際には，値が更新された可能性がある複製の一時変数（OPA コンパイラが解析）に関してのみオブジェクト本体の該当する部分のデータを更新する．

一時変数の更新の可能性は「以前に更新の可能性がある複製の一時変数の集合」という情報についてデータフロー方程式を連立させ，これを解くことで求められる．つまり，基本ブロックの先頭においては先行する基本ブロックの末尾の情報の集合和であるとし，基本ブロックの末尾の情報はブロック先頭の情報とブロック内更新変数集合との集合和として方程式を連立させればよい．

また，先に求めた更新可能性消失点で求められた更新変数集合について一括書き込みを行えるように中間コードを更新する．この一括書き込みでの一時変数の値を用いたオブジェクト本体への書き込みは，一時変数に関しては読み出しであるので，次に述べる生きている変数解析ではその一時変数についてはその直前で生きているものとして扱われる．

4.1.3 生きている変数解析

生きている変数解析についてはよく知られた解析であるのでくわしくは述べないが「以後で（更新前に）使用の可能性がある変数の集合」という情報についてデータフロー方程式を連立させ，これを解くことで求められる．

生きている変数の情報を用いると，2.2 節で述べたメソッドの先頭における一括読み出しにおいて，読み出すべき変数を最小限に抑えることができる．つまり，メソッドの先頭で生きている変数の集合に含まれる複製の一時変数についてのみ一括読み出しを行えばよい．また，生きている変数の情報を用いることにより，サスペンド時に値を退避（実行再開時に復帰）する変数を最小限にとどめることが可能となる．

4.2 値に基づくサスペンドチェック

3.2.2 節で述べたように，スレッド内の関数スケジューリングでは，スタック表現とリスト表現の 2 種類の関数フレームが使い分けられ，それぞれに対応して OPA の関数を担う C のコード（関数）も 2 つのバージョンが生成される．ここで，関数呼出し間での OPA の返り値の授受や，サスペンド時の処理については原則的には 3.2.2 節に述べた通りであるが，スタック表現の関数から呼出し元に戻る際には，値に基づくチェックを使うことで，メモリアクセスを減らすことが可能となる．

スタック表現の関数から戻る際には，呼出しの親側に対して，メソッド本体の実行が完了して OPA の返り値を伴って return したのか，サスペンドし子の関数フレームを伴って return したのかを知らせる必要がある．これらの情報の渡し方を，OPA のメソッドの返り値の型に応じて次のように工夫することができる（表 **1**）．

- `int`，`long` などの場合や，参照型の場合
  関数実行完了時には C の返り値として OPA の返り値が返され，また，サスペンド時には C の返り値として特別なフラグ値（例：-5）が返されるようにする．さらに，サスペンド時にはリスト表現の関数フレーム（のアドレス）を`pr->child_fr`に入れて呼出し元に返すようにする．呼出し元では，C の返り値がフラグ値であった場合は，続けて`pr->child_fr`に関数フレームが設定されているか調べることで，本当にサスペンドしたのか，あるいは，OPA の返り値がたまたまフラグ値であったのかが区別できる．こうすることで，関数実行完了であった場合のサスペンドチェックでは，メモリアクセスを行なう必要が生ずるのがごく稀な場合だけとなる．もちろん，OPA の返り値には C の返り値（が使うレジスタ）を用いるので，メモリアクセスは必要ない．また，`pr->child_fr`の値は通常 0 に保つようにし「サスペンド中」の間のみ関数フレームを保持するようにする．このため，関数実行完了時に return する際に，`pr->child_fr`

表 1 関数からの，OPA の返り値と（サスペンドした）関数フレームの返し方
Table 1 How to return the return value for OPA and the (suspended) function frame from the called function.

関数フレーム /C コードバージョン	スタック表現 /高速バージョン		リスト表現（ヒープ） /低速バージョン
OPA の返り値の型	int, long 等, 参照型	void, float, double	すべての型
関数呼出し完了時	C の返り値 = OPA の返り値 pr->child_fr = 0	C の返り値 = 0 pr->ret = OPA の返り値	C の返り値 = 次の関数フレーム pr->ret = OPA の返り値
サスペンド時	C の返り値 = フラグ値 (例: −5) pr->child_fr = 子の関数フレーム	C の返り値 = 子の関数フレーム	C の返り値 = 0

に 0 を書き込む必要はない．
フラグ値には高速に判定できる（命令の即値に収まる）稀な値を用いればよい．つまり，この方法では，1 つの C の返り値を通常の OPA の返り値とサスペンドを表すフラグの両方に利用しており，通常の OPA の返り値がフラグ値と重なるのは稀であるという性質を利用している．

- void，double，float の場合
OPA メソッドの返り値がない（void）場合は OPA の返り値に関する処理は必要なく，このため，OPA の返り値をレジスタに乗せる必要もないので，原則通り C の返り値として子の関数フレームが返されるようにすればよい．
float，double の場合は，整数型と違い，一般的でかつ実際には稀なフラグ値を準備するのが難しいうえ，浮動小数点数の比較には，メモリアクセスと同程度の遅延を要することが多いため，値に基づくチェックには向かない．また，浮動小数点数はメモリ上に置かれることが多いと想定した命令セットになっていることも多い．よって，これらについては原則通り C の返り値として子の関数フレームが返されるようにし，OPA の返り値が pr->ret というメモリ上の場所に格納されるものとする．

4.3 スタック表現用のコード生成

OPA の以下のプログラムを考える．

```
class C {
  static int c(int x){ ... }
  static int p(int n){
    return c(n+1) + n;
  }
}
```

ここで，クラス（static）メソッド p のスタック表現用コードは図 **10** のように生成される．

引数 n（C の上では p__n）以外に，プロセッサ固有のデータ領域を指す pr も引数とし，int 型の結果を返す関数となっている．関数名 f__p_II_C は，高速バージョンであることと，メソッドの signature（引数と返り値の型）とクラス C により定めている．t0__，t1__ はコンパイラが準備した一時変数である．OPA におけるクラスメソッド c の呼出しに対応して，生成されたコードではそのスタック表現用関数を呼び出し，その直後の if 文でサスペンドの有無を確認して必要な処理を行っている．サスペンドの確認は，フラグ値（SUSPEND）との比較と，一致する場合には続けて子の関数フレーム（pr->child_fr から child_fr に読み出している）が 0 でないことを確認することで行っている．サスペンド処理としてはリスト表現の関数フレームを生成してコンテキストを格納し，子の関数フレーム（child_fr）からのリンクを設定した後，表 1 のスタック表現の int 型に従って親に情報を返している．サスペンドがなかった場合は，続きの処理を行った後普通に値を返せばよい．

```
int f__p_II_C(proc_env *pr, int p__n){
 f_frame *child_fr;
 int t0__, t1__;
 t1__ = p__n + 1;
 t0__ = f__c_II_C(pr, t1__); /* 呼出し */
 if((t0__ == (int)SUSPEND) &&
    (child_fr = pr->child_fr)){
  /* pr->child_fr = 0 は省略可 */
  /* サスペンド処理開始 */
  // ここでフリーリストから関数フレーム fr を
  // 割り当て，リスト表現用コードで実行再開
  // できるようにコンテキストを格納する
  child_fr->parent_fr = fr; /* リンク */
  pr->child_fr = fr;        /* 親へ返す */
  return (int)SUSPEND;      /* 親へ返す */
 }
 return t0__ + p__n;        /* 親へ返す */
 /* pr->child_fr は通常 0 なので設定不要 */
}
```

図 **10** スタック表現（スタックフレーム）用コード
Fig. 10 Code for stack representations (stack frames).

4.4 リスト表現用のコード生成

図 **11** には，クラスメソッド p のリスト表現用コードを示す．リスト表現用コードでは，メソッドの引数 n は C の auto 変数となり，リスト表現の関数フレーム自体が C の関数の引数となる．リスト表現用コードは，呼出しのサスペンドだけでなく実行再開も担うの

```
f_frame *c__p_II_C(proc_env *pr, f_frame *fr){
 f_frame *child_fr, *parent_fr;
 int t0__, t1__;
 int p__n;
 switch (fr->ln) {
  case 1: goto CONT_1;
 }
CONT_0:
 // ここで開始のため fr から p__n を読み出す
 t1__ = p__n + 1;
 t0__ = f__c_II_C(pr, t1__);
 if((t0__ == (int)SUSPEND) &&
    (child_fr = pr->child_fr)){
  pr->child_fr = 0;
  /* サスペンド処理開始 */
  // ここで関数フレーム fr が ln = 1 で
  // 再開するようにコンテキストを設定する
  child_fr->parent_fr = fr;  /* リンク */
  return 0;          /* スケジューラへ */
CONT_1:
  // ここで再開のため fr から p__n を読み出す
  t0__  = pr->ret.i; /* 子からの返り値 */
 }
 pr->ret.i = t0__ + p__n;  /* 親へ返す */
 parent_fr = fr->parent_fr; /* 親フレーム */
 // ここで fr をフリーリストに戻す
 return parent_fr; /* 次にスケジュール */
}
```

図 **11**　リスト表現（ヒープフレーム）用コード
Fig. 11　Code for list representations (heap frames).

で，関数の途中からの処理の再開に対応する．このため関数の先頭の switch 文で，`fr->ln` が保持するラベル番号（コンパイラが`CONT_1` 等のラベルにつけた番号）に従って再開点を決定している．`CONT_0` は，呼出しを最初から実行する場合のラベルである．クラスメソッド `c` の呼出しに対応して生成される部分のコードはスタック表現の場合と同様であるが，サスペンド時の return の方法と再開のためのコードを含む点が異なっている．再開のための処理はサスペンド時のコンテキストの保存に対応した読み出しを行っている．なお，生きている変数解析により再開時点で，`t0__`，`t1__`は生きていないことが解析されるので，この場合，`p__n` の情報のみ回復させればよい．スタック表現用コードのサスペンド部分が保存するコンテキストはリスト表現用コードのサスペンド部分が保存するコンテキストと同じになるようにコード生成を行っており，この再開部分はスタック表現用コードでサスペンドしたものを再開するのにも用いられる．

return の方法は表 1 の通りであり，サスペンド時は 0 を返し，完了時は`pr->ret` に OPA の返り値を設定して，次にスケジュールして欲しい親のフレームを返している．

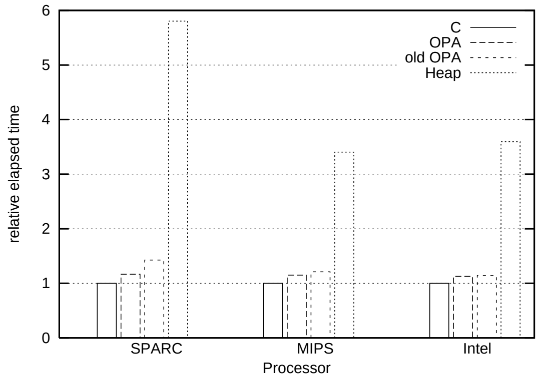

図 **12**　スレッド内スケジューリングのオーバヘッド
Fig. 12　Overheads of within-thread scheduling.

5. 評　　価

ここではスレッド内の関数スケジューリング方式の比較を行なう．再帰呼出しでフィボナッチ数を求めるプログラムを用い，C で書いたサスペンドのできないプログラムと，OPA コンパイラが生成するサスペンド可能な C のプログラムの性能を比較する．ここでは，サスペンドが実際には発生しない場合の単一プロセッサ上の性能を比較し，図 **12** には，C で書いたプログラムの実行時間を 1 としたときの相対的な実行時間を示す．

評価は，SPARC，MIPS，Intel のプロセッサを持つ計算機で行なった．SPARC では，Sun Ultra 30（UltraSPARC-II 296MHz，オンチップキャッシュ 32KB，外部キャッシュ 2MB，Solaris 2.6）の計算機で C コンパイラに GCC 2.8.1 `-O2` を用いた．また，MIPS では，SGI POWEROnyx（MIPS R10000 196 MHz，32KB 命令/32KB データ 1 次キャッシュ，2 次キャッシュ 2MB，Irix 6.5）の計算機で，C コンパイラに MIPSpro Compilers: Version 7.2.1 `-O2 -n32` を用いた．また，Intel では，PC（PentiumIII 500MHz，1 次キャッシュ 32KB（命令用 16KB/データ用 16KB），2 次キャッシュ512KB，Solaris 7）で，GCC 2.8.1 `-O2` を用いた．また，アドレス空間は，すべて 32bit のものを利用した．

ここで「OPA」とあるのが値ベースサスペンドチェックを使った OPA の実行時間「old OPA」とあるのが値ベースサスペンドチェックを使わず OPA の返り値を常に`pr->ret` を使って授受する場合の実行時間「Heap」とあるのがスタック表現を使わずに，常に関数フレームをヒープにとるリスト表現を用いた場合の実行時間である．その場合でも，関数フレームはフリーリストで LIFO 管理されているため，キャッシュミスが頻発

するということはなく，コード量の違い程度の実行時間の差といえる．

フィボナッチ関数において，OPA の生成するコードの性能は，C のプログラムと比較して，14~43%程度のオーバヘッドであったものが，値ベースサスペンドチェックを使うことで，12~17%程度のオーバヘッドに抑えることができている．サスペンドしない場合に実際に実行する命令列の C のプログラムとの違いは，呼出しから戻ってきたときにサスペンドしているかのチェックをレジスタの値に基づいた分岐命令にて行なっている点だけである（callee-save レジスタの処理を除く）．ここで，フィボナッチ関数は呼出しの処理が実行時間の大部分を占める関数であるので，他の一般的な関数の場合は，さらにオーバヘッドは小さくなる．

6. 議 論

OPA における RO メソッド，RW メソッドは Schematic[9) の method，method! という 2 種類のメソッドに相当している．method は読み出し専用のメソッドで，method! はオブジェクトの状態を変化させることのできる読み書き両用メソッドである．method! に関しては，メソッドの実行を before-stage と after-stage という 2 つの部分に分けて行なう．after-stage は他の after-stage と同時に実行することができるとしている． method! の before-stage, after-stage は RW メソッドの一括更新点以前と一括更新点以降に対応するが，OPA ではこれを更新可能性消失点として自動的に解析していおり，また一括読み出し，一括更新の変数も自動的に解析している．

サスペンド/実行再開のために，2 つのバージョンのコードを用いるものとしてはチェックポインティングを可能とする処理系や実行中のスレッドのマイグレーションを可能とする処理系[20) がある．これらのシステムでは，通常，スレッドのスタック全体についてコンテキストの保存/復帰が行われ，呼出し単位での保存/復帰が行われない．このため，実行再開時においては保存したスタック全体の回復がなされるまで実行再開ができないのでコンテキストの切換えが遅くなってしまう．OPA では呼出しを単位とし，残りの呼出しの関数フレームをそのままに実行を再開できるため，高速にコンテキストが切り換えられる．

2 種類の関数フレーム表現と 2 つのバージョンのコードを用いるものとしては Concert システム[5) の他に，Cilk[2) などもある．Cilk は OPA の分担型並列処理に似た並列処理が行なえるが，このシステムでは，Lazy Task Creation[3) に基づいたタスクスティールが行なわれるようになっている．そのため，2 種類の関数フレームの両方にコンテキストを保存しておき，タスクをスティールしたプロセッサは，ヒープ上に表現されたフレームに関する（遅い）コードのほうを用いて，スティールしたタスクを処理するようになっている．

2 種類の関数フレーム表現と 1 つのバージョンのコードを用いるものとしては，StackThreads[7) がある．文献 7) で述べられているコード生成にはヒープ表現の関数フレームを再開する部分でのスタック領域の確保に若干問題があるが，OPA の手法では 2 つのバージョンのコードを用いることでその問題は解決されている．また StackThreads をライブラリとして実現したものには文献 13) があるが，C 言語のみに依存するものではない．一方，StackThreads/MP[8) では，1 種類の関数フレーム表現と 1 つのバージョンのコードを用いており，再開時にはスタックポインタの部分は十分確保した上でフレームポインタが元の関数フレームを指すようになっているが，その実現のために C コンパイラが生成するアセンブリコードを後処理している．

値に基づくチェックは，ソフトウェア分散共有メモリシステム Shasta[6) でも用いられている．Shasta では，キャッシュからのロードミスを検出するのに，キャッシュから値をロードした後，その値が特別なフラグ値でなければロードミスは発生しなかったものとして処理を続けるようになっている．

ポインタが 32bit で表現されている場合，64bit の整数を用いることで関数の返り値とサスペンド情報を同時に返すといった実装法も可能である．しかし，その場合，C 言語のみに依存するという移植性を保つのは難しくなる上，近い将来（あるいは現在も）メモリ容量が増えて，64bit のアドレス空間を用いるようになった場合には利用できない．

7. お わ り に

本研究では，不規則な並列性を含む問題の記述を容易とするオブジェクト指向に基づく並列言語の言語設計，及びその効率良い実行のための並列処理に必要な，一貫性制御，スレッド管理に関する実装技術など開発している．本論文では，その共有メモリ型並列計算機用のコード生成手法について述べた．今後は，スレッド間のスケジューリングにおいて，新しく生成したスレッドに関する様々なフレームのアロケーションを lazy に行なう方法の開発，スケジューラを再帰的

に呼び出すことで無駄なリスト表現への変換を行なわないようにする方式の開発を進めていく予定である．

参　考　文　献

1) Bevan, D. I.: Distributed Garbage Collection Using Reference Counting, *Proceedings of PARLE, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, Vol. 259, Eindhoven, The Netherlands, June 15–19, Springer, Berlin, pp. 176–187 (1987).
2) Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices*, Vol. 33, No. 5, pp. 212–223 (1998).
3) 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).
4) Piquer, J. M.: Indirect Reference Counting: a Distributed Garbage Collection Algorithm, *PARLE'91—Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, No. 505, Eindhoven (the Netherlands), Springer-Verlag, pp. 150–165 (1991).
5) Plevyak, J., Karamcheti, V., Zhang, X. and Chien, A. A.: A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers, *SUPERCOMPUTING'95* (1995).
6) Scales, D. J., Gharachorloo, K. and Thekkath, C. A.: Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory, *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 174–185 (1996).
7) Taura, K., Matsuoka, S. and Yonezawa, A.: *StackThreads*: An Abstract Machine for Scheduling Fine-Grain Threads on Stock CPUs, *JSPP'94*, pp. 25–32 (1994).
8) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pp. 60–71 (1999).
9) Taura, K. and Yonezawa, A.: Schematic: A Concurrent Object-Oriented Extension to Scheme, Technical Report 95-11, Department of Information Science, University of Tokyo (1995).
10) Yasugi, M.: Hierarchically Structured Synchronization and Exception Handling in Parallel Languages Using Dynamic Scope, *Parallel and Distributed Computing for Symbolic and Irregular Applications* (Ito, T. and Yuasa, T.(eds.)), World Scientific, pp. 122–148 (2000).
11) Yasugi, M., Eguchi, S. and Taki, K.: Eliminating Bottlenecks on Parallel Systems using Adaptive Objects, *Proc. of International Conference on Parallel Architectures and Compilation Techniques, Paris, France*, pp. 80–87 (1998).
12) Yasugi, M., Eguchi, S. and Taki, K.: Adaptive Objects for Concurrent Accesses, *Object-Oriented Parallel and Distributed Programming* (Bahsoun, J.-P., Baba, T., Briot, J.-P. and Yonezawa, A.(eds.)), HERMES Science Publications, pp. 187–206 (2000).
13) 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).
14) 江口重行, 可児亜祐美, 八杉昌宏, 瀧和男: 適応的オブジェクトによる並列処理のボトルネック解消, 並列処理シンポジウム (JSPP'98) 論文集, pp. 111–118 (1998).
15) 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).
16) 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).
17) 八杉昌宏, 瀧和男: 分散共有メモリ型並列計算機 KSR 上の細粒度並列処理用実行方式の評価, 信学技報 CPSY-95-59(SWoPP'95), Vol. 95, No. 210, pp. 71–78 (1995).
18) 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, 情処研報 96-PRO-8(SWoPP'96), Vol.96, No.82, pp.157–162 (1996).
19) 八杉昌宏, 瀧和男: 実用的な並列処理のためのオブジェクト指向言語 OPA の設計, 第 13 回オブジェクト指向計算ワークショップ (WOOC'97) (1997).
20) 多賀奈由太, 関口龍郎, 米澤明憲: 通常の実行効率をほとんど損わないスレッドマイグレーションが可能な C++, 情報処理学会論文誌：プログラミング, Vol. 41, No. SIG 2 (PRO 6), pp. 41–53 (2000).

(平成 13 年 1 月 04 日受付)
(平成 13 年 4 月 16 日採録)

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．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）．1998 年より科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．

馬谷　誠二

1974 年生．1999 年京都大学工学部情報学科卒．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．

鎌田十三郎（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒．1995 年同大学大学院理学系研究科情報科学専攻修士課程修了．1998 年同博士課程単位修得退学．1996～1998 年日本学術振興会特別研究員（東京大学）．1998 年より神戸大学工学部助手．修士（理学）．並列・分散処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．

田畑　悠介

1976 年生．2000 年京都大学工学部情報学科卒．同年より同大学大学院情報学研究科修士課程に在学中．並列処理と言語処理系に興味を持つ．

伊藤　智一

1977 年生．2001 年京都大学工学部情報学科卒．同年より同大学大学院情報学研究科修士課程に在学中．言語処理系などに興味を持つ．

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」他．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．