動的なスレッド生成をサポートする言語のコンパイル技法

大山 恵弘　　田浦 健次朗　　米澤 明憲
{oyama,tau,yonezawa}@is.s.u-tokyo.ac.jp
〒113 東京都文京区本郷 7-3-1
東京大学大学院理学系研究科

概要

並列オブジェクト指向言語の実装において、マルチスレッディングの効率は全体のパフォーマンスに対して大きく影響を及ぼす。しかしライブラリによるスレッドの実装は効率の面で大きな問題がある。そこで我々は、スレッド生成を積極的に支援する言語を効率的にコンパイルする枠組を提案し、その有効性を示す。我々はその枠組をもとに、Scheme に並列オブジェクト指向拡張を加えた言語 Schematic を設計、実装した。また、その中間言語として単純だが十分な表現力をもっているプロセス記述言語 HACL のサブセットを用い、その上でいくつかの最適化を行なった。ベンチマークを行ない、高速な実行を達成したことを確認した。

Abstract

The efficiency of multithreading is quite essential to the overall performance of concurrent object-oriented languages. It is very inefficient to implement such languages by using thread libraries. In this paper, we propose the framework that efficiently compiles languages which supports aggressive thread creation. In the framework, we designed and implemented the programming language Schematic, which is a concurrent object-oriented extension to Scheme. As an intermediate language of Schematic, we are using the subset of the process calculus HACL, which is simple but has enough power to express higher-level constructs. To the intermediate language, we applied some optimization techniques and achieved very high performance in benchmark.

1　はじめに

並列計算機やワークステーションクラスタの普及にともない、高性能な並列言語の重要性がますます高まっている。中でも並列オブジェクト指向言語は並列性を自然に記述できる強力なモデルであり、今後さらに注目されていくと思われる。

しかし、並列オブジェクト指向言語はこれまで実行性能の面で問題があることが多かった。その原因の１つに、ノード内並列実行のオーバヘッドが大きいというものがある。

普通の言語実装においては、非同期メソッド起動ごとにスレッドが生成されるため、少なくともその数以上はスレッド間コンテキストスイッチが起こることになる。一般に、その個数は非常に多く、また、細粒度なスレッドが大半である。よって、多数の細粒度スレッドを処理する方法は並列オブジェクト指向言語の全体の性能に対し支配的な影響を及ぼす。

スレッドを実現する１つの方法はライブラリを使うことである [5]。しかし、このやり方は、１スレッドがスタック１つを消費するため、メモリ容量の面で同時に起動できるスレッド数に制限がつく上、スレッドスイッチの際にはレジスタ全体の退避、回復が必要なので、速度が著しく落ちる。このため、細かいスレッドを多数生成して頻繁にコンテキストスイッチを行なう言語を高効率に処理することは、ライブラリでは非常に難しい。

そこで、我々はこの問題を解消するため、本論文で細粒度マルチスレッディングをコンパイラによって実現する技法を提案する。我々が設計、実装した並列オブジェクト指向言語 Schematic[17] は、Scheme に並列オブジェクト指向拡張を施したものである。コンパイラがスレッドを制御することで新たに最適化の機会が生まれ、高速な処理が可能になった。現在 Schematic は WS および並列計算機 AP1000[10] 上で稼働しており、グローバル GC[19] などの研究も同時に進行している。

第 2章で我々の表面言語 Schematic を説明し、第 3章でコンパイラの構成を概観する。第 4章で中間言語 Venezia について述べ、その実行モデルを第 5章に示す。第 6章でコード生成方法と最適化を、第 7章でコンパイル技法に関する議論を行なう。第 8章にはベンチマークを示す。第 9章に関連研究を述べ、結論を第 10章に記す。

2　並列オブジェクト指向言語 Schematic

2.1　概要

Schematic は Scheme の call/return の枠組を保持しながら並列オブジェクト指向の拡張機能をスムーズに統合した言語である。

関数型言語が λ 計算を基礎にしているのと同様に、Schematic はプロセス計算を基礎にしている。Scheme における関数を非同期的なプロセスのひな形とみなし、関数適用をプロセス生成とみなす。さらに、同期および通信のための基本 data 型として channel を導入する。

channel は値を保持する箱のようなものであり、プロセスは値を送信すること、受信することができる。channel に複数の値が送信された場合、それらはキューイングされる。同時に複数の受信者、送信者があった時、どのペアが通信を行なうかについては非決定的である。Schematic は channel を一階の値として扱うことができるので flexible な同期機構を記述できる。

2.2　基本 primitives

簡単に Schematic の基本 primitive を紹介する。
Channel への送信は

```
(reply v c)
```

で行なう。channel c に値 v が送られる。

一方、受信は

```
(touch c)
```

とする。この式は c に値が送信されていればそれを返り値とし、そうでなければブロックして値の到着を待つ。

channel を明示的に作成するには、

```
(make-channel)
```

を実行する。返り値は空の channel である。

プロセスの起動は Scheme の関数呼びだしと同じく、

```
(f x y)
```

などと書けばよい。プロセス f が同期的に実行される。

非同期な呼び出しは、future を用いる。

```
(future (f x y))
```

を実行すると、

- 起動プロセスの返り値を入れるための channel(reply channel と呼ぶ) の生成
- f の非同期的実行 (暗黙のうちに reply channnel を f に渡す)
- reply channel を future 式全体の値として受けとり、caller の処理を継続

が行なわれる。reply channel を変数にバインドしておけば、セルに入れたり、touch によって送信された値を取り出すことができる。最初に future が提案された Multilisp [7] とは違い、我々は値と channel を明確に区別している。

並列計算機上の実行環境においては、式を実行するノードをプログラマが指定することができる。

```
(future (f x y) :on pe)
```

とすると (f x y) の計算はノード *pe* で行なわれる。単 CPU 上の実行では :on によるノード指定は無視される。

Schematic では他にも、並列にバインド節の実行を行なう plet や並列に各節を実行する pbegin などの高レベルな construct がマクロのような形で基本 construct の上に実装されている。

2.3 並列オブジェクト指向拡張

Scheme では set-car! などを用いればデータ構造を更新可能である。しかし、そのデータが複数の並列プロセスによって共有され、2 つ以上の操作が interleave しながら実行されると、望ましくない結果を生むことがある。

そこで Schematic では並列オブジェクトという排他実行の単位となる構造体を導入した。プログラマが mutable なデータをすべて並列オブジェクトとして定義すれば、それらの状態変化はメソッドを単位として行なわれ、中間状態が外に観測されることはない。

多くの Actor[1] に基づいた言語 [18] ではメソッドの実行は serialize される。しかし Schematic においては、更新を行なうメソッドとそうでないものをプログラマが区別して記述することで、メソッドをできるだけ並列に実行して効率を上げている。

オブジェクト指向の様々な primitive は、前述の future, touch などの並列 construct とレコードを組み合わせて実現している。例えば下の図は並列オブジェクトへの update 操作を Schematic で記述したものである。

```
(define (update instance)
  (let ((lock (get-lock instance)))
    (touch lock)
    update instance
    (reply 0 lock)))
```

排他処理に入る前に touch によって lock を獲得し、終了したら channel に値を送って lock を解放する。

並列オブジェクトの詳細については [17] を参照されたい。

2.4 Schematic のプログラム例

下は Schematic で記述されたフィボナッチ関数を計算するプログラムである。

```
(define (fib x)
  (if (< x 2)
      1
      (let ((c1 (future (fib (- x 1))))
            (c2 (future (fib (- x 2)))))
        (+ (touch c1) (touch c2)))))
```

(future (fib (- x 1))) では非同期に (fib (- x 1)) を起動し、その結果を入れる channel を c1 にバインドする。c2 の方も同様である。(touch c1) では c1 を見て結果が返っていればそれを返し、そうでなければブロックする。また、fib 全体はプログラムに現れていない reply channel に計算結果を reply する。

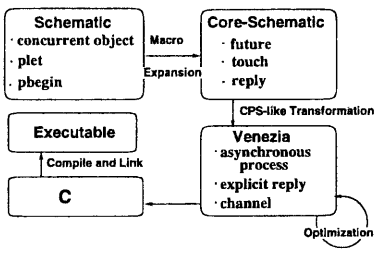
2.5 まとめ

Schematic ではプロセス起動や同期処理が非常に簡潔に表現されている。C 言語などで同様のプログラミングをしようとすると、はるかに繁雑な作業が必要になる。並列性が簡単かつ自然に記述でき、小さな負担で排他処理もできるので、Schematic は GUI[11] や分散並列アプリケーションに適している。

また、Schematic では高度な construct は少数の基本 construct を組み合わせて実現しているため言語カーネルが非常に小さい。これによって実装が極めて単純になるとともに、仕様の変更に対する柔軟性も獲得している。

3 コンパイラの概要

コンパイラの処理の流れと中間言語を次の図に示す。



まず Schematic は並列オブジェクト指向などのマクロを展開することで Core-Schematic になる。Core-Schematic の各 primitive は CPS 変換 [2] に類似した方法を利用して、かなり単純な操作で中間言語 Venezia に変換される。最後に Venezia 上で様々な最適化を適用したのち、C 言語のプログラムを生成する。

C にコンパイルする方式はポータビリティ、実装の簡単さ、高性能な C コンパイラが使用可能、というアドバンテージを生む。

以降混乱のない限り Core-Schematic を Schematic と呼ぶ。

4 中間言語 Venezia

Schematic から直接 C コードを生成するのは、一般性がなく、見通しが悪いことから、避けた方がよい。そこで我々は単純かつ十分強力な表現力をもつ中間言語 Venezia を設計、実装した。

4.1 Veneziaの解説

Schematicのレベルではプロセス呼びだしは同期的なものと非同期的なものの2つが存在したが、Veneziaではあらゆるプロセス呼び出しは非同期であるとし、同期呼びだしは「非同期呼びだし＋channelによる同期」で表現する。それにはSchematicのプロセスに対して次の操作を施す。

- Schematicにおいて非明示的であったreply channelをすべて明示的にする
- 同様にプロセスが新たに生まれる場所(fork, reply)をすべて明示的にする

例えばSchematicで `(define (f x) (+ x 1))` という定義があったら、これを `(define (f c x) (reply (+ x 1) c))` のように展開する。通信channelを引数に加え、返り値を明示的にreplyで返していることに注意されたい。

同期的なプロセス起動 `(f x)` はパーザが `(touch (future (f x)))` であると解釈する[1]。同期呼び出しをなくしたことでHACL[13]やπ-calculus[14]などのプロセス計算に直接対応づけができるとともに、最適化なども含めた言語実装が高度に単純化される。

Veneziaは ML[3]-like な文法を持つ。以下で定義する。

$P(proc)$	::=	$P_1 \mid P_2$	並列実行
		$\mid \$x.P$	channel生成
		$\mid e_1\ e_2$	プロセス起動
		$\mid e(x)\texttt{=>}P$	channelからの受信
		$\mid e_1\texttt{<=}e_2$	channelへの送信
		$\mid$ `if` e `then` P_1 `else` P_2	条件分岐
		$\mid$ `fix` Γ `in` P	局所プロセス定義
$\Gamma(defs)$	::=	$\{f_1(x_1,\ldots) = P_1,\ \ldots,\ f_n(x_1,\ldots) = P_n\}$	
$e(expr)$	::=	var $\mid$ const $\mid$ $op(e) \mid (e_1, \ldots, e_n)$	

$P_1 \mid P_2$ は P_1 と P_2 を並列に実行する。$\$x.P$ は新しくchannelを生成し、それを x にバインドして P を実行する。受信を示す $e(x)\texttt{=>}P$ はchannel e から値を取り出し、それを x にバインドして P を実行する。値がなければ来るまで待つ。$e_1\texttt{<=}e_2$ はchannel e_1 に e_2 を送信する。$e_1\ e_2$ は引数 e_2 を与えてプロセス e_1 を起動する。

我々の研究室はこれまでに、非同期プロセスと通信channelの概念を積極的に使用した言語HACL[13]を提案してきた。VeneziaはHACLに類似しているが、Choiceや関数(λ式)がない、型がないなどの点においてサブセットであると言える。HACL上での研究成果はVeneziaでも参考にすることができる[8]。

4.2 Veneziaのプログラム例

下はフィボナッチ数列を求めるVeneziaプログラムである。

```
fib m x =
 if (x < 2)
  then m <= 1
  else $c1.$c2.(fib c1 x-1 |
                (fib c2 x-2 |
                 c1(v1)=>c2(v2)=>(m<=v1+v2)))
```

mはfibが計算した値を返すためのreply channelである。else節では左右の子のために$c1.$c2.で2つchannelを生成し、それぞれをforkする。そしてc1(v1)=>c2(v2)=>(m<=v1+v2)でc1およびc2から受信し、値をそれぞれv1、v2にバインドする。最後に足し算の結果をmに送信する。

[1] `(f x)` のセマンティクスは `(touch (future (f x)))` で与えられるので、厳密にはSchemeのそれとは異なる。例えば `(f x)` を呼んでも制御が戻ってこないことがありうる。

4.3 SchematicからVeneziaへの変換

基本的にCPS変換[2]に準じた操作を行なう。特殊なprimitiveについて以下で解説を加える。

reply `(reply v c)` は `c<=v` という送信を意味するVeneziaコードに変換される。reply式自体の返り値は不定なので、`(cons (reply v c) 10)` のように出現する場合は、送信と並列に `(cons` *undefined* `10)` の処理を行なう。

touch touchは受信に変換される。例えば、`(reply (+ (touch c1) 1) c2)` という式は `c1(v)=>(c2<=(v+1))` と変換される。c1から受信をしてその値をvとし、1を足した結果をc2に送信する。

future futureが実行されたときの具体的な流れは2.2に述べた。`(future (f x))` という式は、`$c.(f c x)` のように通信channelの生成と、そのchannelを引数に加えたプロセス呼びだしに変換される。それに加えて、replyと同様に、channel cを返り値として得たfutureの外側の式を並列に実行する。

5 Veneziaの実行時モデル

5.1 Channelの表現

各channelは値を保持するvalue queueと値の到着を待つthreadのclosureを保持するthread queueの2つを持つ。

Cレベルにおいてはchannelに対し

```
THREAD_QUEUE_EMPTY(m)
ENQUEUE_VALUE(v, m)
DEQUEUE_THREAD(m)
```

などの関数を用意する。例えば`ENQUEUE_VALUE(v, m)`ではchannel m のvalue queueに v をenqueueする。

5.2 実行

実行時には1つのノードは1つのglobal scheduling queueを持つ。Cレベルでは`ENQUEUE_SCHEDULING_QUEUE(t)`などの関数を用意している。

Veneziaではプロセスは非同期に他のプロセスを起動しながら処理を進めるので、呼びだしの構造はstackではなくtreeを形成する。具体的にはこのtreeの実行は片方の子をscheduling queueに入れ、もう片方を実行していくことで実現される。そのtreeをたどって行き着く先は「他のプロセスのfork」もしくは「channelへの送信」のどちらかである。「fork」の場合はそのプロセスを直接実行する。「送信」で待ちプロセスが存在する場合はforkと同様にそのプロセスを直接実行し、存在しない場合は値をvalue queueに入れた後、scheduling queueから他のプロセスを取り出して実行する。

6 Cコード生成

VeneziaからCを生成する枠組と最適化について述べる。

6.1 基本的な枠組

コード生成器はVeneziaの一つのプロセス定義を一つのC関数へと変換する．つまり，あるプロセス定義:

```
f c x y = P
```

に対して, P を以下の章で述べる方法で変換した C コードを全て含む, 一つの C 関数を生成する.

このように実現した場合, P の実行の中断 (suspend) および復帰をどう実現するかが自明でない問題となる. 例えば, P が channel からの値の受信 $m(x)$ => Q を実行した際に, m が空だと, Q の実行は後に m が値を受信した時まで延期されなくてはならず, その時点では f が実行されているとは限らない. このような関数間に跨る制御の移動を, 唯一の entry point を持つ C 関数で実現するのは困難である.

我々の現在の解決方法は, GNU C コンパイラに実装されている値つきラベル (labels as values) を使うことである. GNU C コンパイラでは C のラベルのアドレスを値として取り出し, 後に任意の地点からその場所に goto によって制御を移すことができる. あるラベルに goto する際に, 引き続く計算が必要とする状態 (レジスタおよびスタック) を正しい状態に戻すことは, コード生成器の責任である. これを実現するために我々は再び, GNU C コンパイラの「明示的なレジスタ変数 (Explicit Reg Vars) を用いた. GNU C コンパイラでは大域変数を含め, 任意の変数を指定したレジスタに割り当て, かつそのレジスタが他の目的に使われないように指示することができる. 我々は, レジスタを表すための C の大域変数を多数設け, そのうちの小さなレジスタ番号を持つもの 8 個を明示的にレジスタに割り当てた. コード生成器は明示的に Venezia 変数をそれらの大域変数に割り当てることによって, 計算状態を復帰させるためのコード列を C プログラムとして生成することができる.

6.2 Closure 変換

Venezia を C にコンパイルする途中で、closure 変換を行なう。Closure はコードアドレスと自由変数からなるレコードである。詳しくは [2] を参照してほしい。

6.3 各 primitive の変換

Venezia の各 primitive について変換手順を非形式的に説明する。

並列実行 $P_1 | P_2$ は、

```
ENQUEUE_SCHEDULING_QUEUE(closure of P2);
P1';

label_P2:
P2';
```

のように並列節の片方を queue に入れ、残りをそのまま実行する。ここで P_1'、P_2' はそれぞれ P_1、P_2 を C に変換したものである。P_1' の末尾にはジャンプ命令があるので P_2' には直接たどり着かない。P_2 の closure がのちに global scheduling queue から取り出され、goto で *label_P2* に直接制御が移る。どちらの節を queue に入れるかの判断については 6.4.1節で議論する。

プロセス呼びだし プロセス呼びだしは「レジスタ並べかえとジャンプ」で表現される。Venezia のプロセス呼びだし
f 10 20 は C では APPLY(f,10,20) と変換される。この APPLY は最終的には

```
r1 = 10;
r2 = 20;
JUMP(CODE_ADDRESS(f));
```

のようにコンパイルされる。r1、r2 は前述のレジスタに割り当てられたグローバル変数である。JUMP(CODE_ADDRESS(f)) は関数 f の先頭のラベルにジャンプする命令に展開される。

送信 Channel m へ値 v を送信する部分の Venezia コード m<=v は次のようにコンパイルされる。

```
if (THREAD_QUEUE_EMPTY(m)) {
    ENQUEUE_VALUE(v, m);
    SCHEDULER();
} else {
    c = DEQUEUE_THREAD(m);
    APPLY(c,v);
}
```

もし待ちスレッドがなければ、value queue に値を入れたのちSCHEDULER() によって scheduling queue にあるスレッドの実行に移る。待ちスレッドがあればそのスレッドを取り出し、アドレスを得たのち、送信された値を引数に加えてジャンプを行なう。

受信 Channel からの受信 $m(x)$=>P は次のようにコンパイルされる。

```
c = closure of P
if (VALUE_QUEUE_EMPTY(m) == NO) {
    v = DEQUEUE_VALUE(m)
    APPLY(c, v);
} else {
    ENQUEUE_THREAD(c, m);
    SCHEDULER();
}
label_P:
load freevars;
P';
```

受信に成功した場合は直接 P に実行を移す。失敗した場合、P の closure c を channel の thread queue に入れたのち global scheduling queue から他のプロセスを取り出して実行する。P の closure には resume した時にジャンプするべきラベルのアドレスを格納しておく (ここでは *label_P*)。

6.4 最適化

6.4.1 セーブ変数の解析

並列実行節がある場合、どの節を続けて実行し、どの節を scheduling queue に入れるかを決定しなければならない。このとき、多くの自由変数をもつ節をそのまま実行すれば、ロードセーブのオーバヘッドを省くことができる。

そこで我々はメモリにセーブする自由変数の数を各並列節について数え、その情報を利用してどの節をそのまま実行し、どの節をセーブするのかを決定する最適化を実装した。

```
f m n a b c d e f g h
  =  bar m a
   | m(v) => n<=(a+b+c+d+e+f+g+h)
   | baz n a b c d
   | n<=10
```

例えば上のプログラムでは、各節について次の値をセーブしなければならない。

節	セーブ変数 (アドレス除く)	個数
第 1 節	m,a	2
第 2 節	m,n,a,b,c,d,e,f,g,h	10
第 3 節	n,a,b,c,d	5
第 4 節	n	1

コンパイラは、第 1,3,4 節を scheduling queue の中に入れ、自由変数の個数の最も多い第 2 節をそのまま実行する判断を下す。

6.4.2 無駄なランタイムチェックの削除

上のセーブ変数解析をそのまま適用すると

```
f n a b c d e f =
  $m.(f(m, a) | m(v) => n<=(v+a+b+c+d+e+f))
```

というプログラムは

1. f(m, a) をセーブ
2. m(v)=>... を実行

と scheduling される。しかしこれは m(v)=> において受信に失敗し、ブロックののち n<=(v+a+b+c+d+e+f) を結局セーブすることになり、上の節を実行するよりも非効率的な scheduling を与えてしまう。

そこで我々は空の channel を記憶しながらコンパイルを行なうことで、この問題の解決をした。空 channel からの受信節がある場合にはランタイムチェックを省き、受信失敗のコードに置き換えてしまう。具体的には、上のコードでいえば、

1. n<=(v+a+b+c+d+e+f) の部分の closure を channel m の thread queue に入れる
2. 直接 f(m, a) を実行

とコンパイルする。これによってランタイムチェック、無駄な変数のロードとセーブばかりでなく、global scheduling queue の操作をも省いた。Channel への送信の際にそのまま待ちスレッドへ制御を移すことが可能である。

現在空 channel か否かの判定は「新しく作られ、何も操作を受けていない channel は空である」という簡単な基準を用いている。

6.4.3 inlining

小さいサイズのプロセス呼びだしについて inlining を行なった。

```
foo c x = c<=x+1
bar n y = $m.(foo m y | m(v)=>(n<=v+y))
```

におけるプロセス bar は、

```
bar n y = $m.(m<=y+1  | m(v)=>(n<=v+y))
```

と変換できる。

6.4.4 静的な通信

関数の inlining に伴い、静的に通信できる channel が頻繁に出現する。そこで我々は

$$\$n.(\ n<=e\ |\ n(v)\ =>E)$$

のようなパターンが現れた際には、channel n に関して静的な通信を行ない、この節全体を

$$\$n.E[e/v]$$

にする最適化を行なった。上で E 中に n が出現しなければ、$n という channel 生成自体も省くことができる。

6.4.5 Linear Channel

一回しか通信が行なわれない channel (linear channel[12] と呼ばれる) については、channel の中に queue を作る必要がなく、値を入れる領域およびタグのみを用意しておけばよい。また、この他にも様々な最適化を施せることがわかっている。

最近、言語 HACL 上で channel が linear であるかないかを推論するアルゴリズムが提案された [9]。我々のコンパイラでもこの枠組を用いて自動的に linear channel を検出できるようになるが、現時点では、プログラマが明示的に linear channel だと指定するようにしている。

7 議論

7.1 スレッドの実装方式

我々はラベルを用いてジャンプを実装しているが、関数を分割し、C の「関数へのポインタ」で suspension point を表現する方法もある。この方式では、fork や resume は通常の関数呼び出しで行なわれる。しかし、これは関数呼び出しのオーバヘッドを伴う上、スタックが非常に長く伸びてキャッシュミスを頻発する可能性があるので極めて不利である。

Scheme の拡張であることを考えると、Scheme 処理系の上にマクロのように実装するのが合理的であるという見方もある。事実、call/cc を用いれば Scheme レベルでスレッドを実現可能である。しかし call/cc は continuation をコピーするという形で実装されていることがある。スレッド生成のたびに膨大なコピーが行なわれるため、大幅な速度の低下と無駄なメモリ使用を招く。

7.2 並列計算機環境での実行

これまで 1CPU 上での実行を中心に述べてきたが、我々のコンパイルの枠組は少数の追加で並列計算機上にそのまま適用できる。

リモートノードへの fork は、単に enqueue する scheduling queue が変わるだけであり、リモートな channel への操作も、送信や受信を行なうランタイムの初めに簡単なチェックをはさむだけで容易に多ノード実行のそれになる。よって、並列実行をしても逐次の枠組の部分はほとんど影響を受けず、効率を失うことはない。

我々のコンパイル技法は

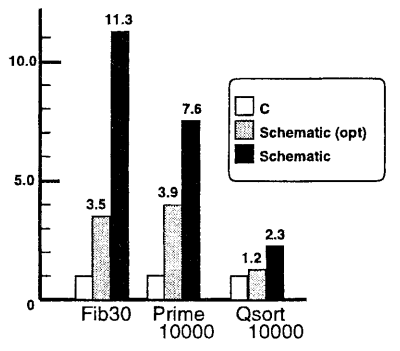
- 高効率なノード内実行
- 高効率な並列実行

の 2 つを同時に実現しており、広い一般性を持つ極めて強力な枠組であると言える。

例えば分散 tree の summation は我々のモデルが特に力を発揮するアプリケーションである。子の計算をすべて future で呼び出してその結果を touch で待つようにする。子がリモートであるなどの理由でブロックした時でも親はブロックせずに他の子の処理が継続できる。ブロックしない時には、少ないオーバヘッドで逐次実行が行なえる。

8 性能測定

Schematic コンパイラのベンチマークを行なった。結果を図に示す。



C と比較して、channel(メモリ) による値の授受、closure への変数の無駄な出し入れがオーバヘッドになっている。chan-

nel をレジスタに置くことや、closure のシェアを行なえばさらに高速化が可能である。

9 関連研究

ABCL/f 我々の研究室では言語 ABCL/f[16] がすでに設計、実装されている。future-touch による明示的並列処理や、並列オブジェクトの概念を用いている点で Schematic に非常に似ている。

Schematic が Scheme をベースにしているのに対し、ABCL/fは Common Lisp をベースにしている。また、ABCL/fは ML-like なパラメタ型多相システムを持っている。

HACL HACL[13] は線形論理 [6] に基づいて設計されたプロセス記述言語である。HACL は高レベルな並列言語のための汎用核言語の役割を果たすだけではなく、言語一般の解析のためのプラットフォームとして利用されている。Venezia の各 primitive は HACL に counterpart を持つ。

KL1 KL1[4] は第五世代プロジェクトの核言語として設計された並列論理型言語である。もともとは特殊なプロセッサ用に開発された言語であるが、KLIC という C へ translate する処理系も実装されている。通信 channel の概念、非同期なプロセスの概念など、Venezia と共通点が多い言語である。

TAM TAM は non-strict 言語 Id の中間言語である [15]。コンパイルはまず Id から dataflow graph を作成し、さらにそれを TAM 上のスレッドに変換することで行なう。我々の Venezia を C に変換する部分は並列構造を逐次化する作業だと見ることができるが、彼らの dataflow graph から TAM への変換の部分も類似したことをしている。かなり高度な最適化が研究されており、参考にする部分が多い。

10 結論および今後の課題

我々は細粒度スレッドを効率良くコンパイルする枠組について述べた。また、その枠組の上に並列オブジェクト指向言語 Schematic の最適化コンパイラを実装した。そして、ベンチマークを行なうことにより、我々の方式の正しさを確認した。

今後はコンパイラを信頼に足るものに完成させ、複雑で巨大なアプリケーションを構築するとともに、より洗練された一般性ある最適化技法を研究していきたい。

11 謝辞

本研究を進めるにあたり、細谷晴夫氏には Schematic の性能測定をしていただくとともに、性能向上のための様々なアドバイスをいただきました。また、研究室の方々との議論が大変役に立ちました。ここに感謝致します。

参考文献

[1] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.

[2] A. W. Appel. *Compiling with Continuations* . Cambridge University Press, 1992.

[3] A. W. Appel and D. B. MacQueen. A Standard ML Compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[4] T. Chikayama, T. Fujise, and D. Sekita. A Portable and Efficient Implementation of KL1. In *Proceedings of PLILP'94*. Springer-Verlag, 1994.

[5] E. C. Cooper and R. P. Draves. C Threads, 1987.

[6] J.-Y. Girard. Linear Logic . *Theoretical Computer Science*, 50, 1987.

[7] R. H. Halsted. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7(4)*, pages 501–538, Apr 1985.

[8] H. Hosoya, N. Kobayashi, and A. Yonezawa. Partial Evaluation Scheme for Concurrent Languages and its Correctness . In *Euro-Par 96*, 1996.

[9] Atsushi Igarashi and Naoki Kobayashi. Type-based analysis of usage of communication channels for concurrent programming languages. Technical report, Department of Information Science, University of Tokyo, 1996. to appear.

[10] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. An Architecture of Highly Parallel Computer AP1000. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 13–16, May 1991.

[11] R. Kakimoto. SchematicTk: A concurrent object-oriented GUI toolkit (Senior Thesis), February 1996.

[12] N. Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus . In *Proceedings 23'rd Symposium on Principles of Programming Languages*, 1996.

[13] N. Kobayashi and A. Yonezawa. Typed Higher-Order Concurrent Linear Logic Programming. Technical report, Department of Information Science, University of Tokyo, December 1994.

[14] R. Milner. The polyadic π-calculus: A tutorial. Technical report, University of Edinburgh, 1991.

[15] K. E. Schauser, D. E. Culler, and S. C. GoldStein. Separation constraint partitioning – a new algorithm for partitioning non-strict programs into sequential threads. In *POPL ' 95*, 1995.

[16] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f : A future-based polymorphic typed concurrent object-oriented language — its design and implementation —. *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[17] K. Taura and A. Yonezawa. Schematic: A Concurrent Object-Oriented Extension to Scheme. Technical report, Department of Information Science, University of Tokyo, 1996.

[18] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

[19] 遠藤敏夫・田浦健次朗・米澤明憲. Portable で robust な global garbage collector の構築について. In *SWoPP '96*, 1996.