並列処理のための協調機能を持つ手続き型言語

八杉　昌宏

■ 研究のねらい

プログラミング言語とは人工的な「ことば」であり、計算機に計算をさせたいとき、計算機への指示をプログラムとして記述するために用いられる。どのようにプログラムを書くのか(構文)、プログラムがどのような計算を表すのか(意味)を定めたのがプログラミング言語の仕様である。信頼性・再利用性・実行効率の高いプログラムを簡単に作成できるようなものが良い言語仕様といえ、本研究も、より良い言語仕様を設計しようとするものである。

計算機ハードウェアにとってみれば単純で実行し易い言語で書かれた詳細なプログラムで指示してもらったほうが高速に実行できる一方で、人にとってみれば分かりやすく自然な言語でプログラムを書きたい。そのため一般に使われている抽象化手法の一つは、人が理解しやすい言語のプログラムが実行できる仮想的な計算機を考え、ソフトウェアを含むシステム(言語処理系)として実現するというものである。このような階層的なシステムでは、「人にとって分かりやすい」高水準言語と「計算機ハードウェアにとって実行しやすい」低水準言語が使われており、また、両者の中間には、段階的に様々な水準の言語があってもよい。なお、高水準言語はただ一つあるわけではなく、用途に合わせて様々な高水準言語がある。(本研究では、高水準言語として、並列論理型言語、並列関数型言語、オブジェクトベース並列言語、マルチスレッド言語など、不規則で細粒度の並列処理（例えば、探索問題、最適化問題など）を記述することができる言語を主に対象としている。もちろん規則的で粗粒度の並列処理を行なうデータ並列言語なども対象として構わないが、そのような自明な並列処理には既存のライブラリを用いるなどの既存の技術で十分といえる。)

高水準言語のプログラムを言語仕様に従って実行する仮想的な計算機を実現する言語処理系は、プログラムをそのまま解釈実行してもよいが、まず、高水準言語のプログラムを低水準言語のプログラムに翻訳してから、実際のハードウェア（＋オペレーティングシステム）としての計算機で実行するほうが、効率よく実行できる。このように翻訳することをコンパイルするといい、翻訳を行うシステムをコンパイラという。実際のプログラムの実行効率はハードウェアの性能のほかにコンパイラを含む言語処理系の実装技術にも左右される。また実際的な観点から、高水準言語のコンパイラの開発において、共通の中間言語へとまず翻訳し、中間言語から低水準言語へと翻訳するという形をとると、さまざまな高水準言語の処理系の実装において中間言語から低水準言語への翻訳部分を共通化できるという利点がある。このような実装用の中間言語として、Ｃ言語という言語が使われることが多い。計算機ハードウェアが直接実行できるプログラムを記述するための機械語の仕様はプロセッサの種類に依存するが、Ｃ言語のプログラムとすることで特定のプロセッサの種類に依存しないようにできるからである。

一般に使われている抽象化手法には別の次元のものもある。プログラムにおいてまとまった手続きに名前を付けて、名前で手続きを何回でも呼び出せるようにするという手法である。再帰的な計算をうまく表現できるほか、同じような手続きの内容を何度も書かずにすむ。手続きを呼び出すと、現在処理中の手続きの実行をいったん中断し、呼び出した手続きの実行を開始する(call)。呼び出した手

続きの処理が完了すると、自動的に途中まで処理した手続きの実行に戻る(return)。C言語もこのような手続き型言語の一種である。

本研究では、特に並列処理のためのプログラミング言語に着目している。計算機の高性能化は、複雑な問題の解決、大規模シミュレーション、マルチメディア情報処理などの新しい能力を人類にもたらしているが、そのような計算機の高性能化のために並列計算機は魅力的であるからである。並列処理というのは、二つ以上の計算実行部を持つ計算機に計算させることであるが、並列アーキテクチャは、共有メモリ型アーキテクチャ、分散メモリ型アーキテクチャなど様々であり、その違いを吸収しつつ信頼性・再利用性・実行効率の高いソフトウェアを開発するには、並列処理のための高水準言語の役割は重要である。このような並列計算機のための高水準言語の処理系の実現には、上で述べたように、言語階層を用いると便利である。特にC言語を実装用の中間言語として用いることで処理系の共通に利用できる部分を増やし、処理系実装の労力を減らすことができる。(図1)

高水準言語であれば並列計算機であることを意識させないものも考えられるが、コンパイル後のC言語のレベルでは、複数の実行主体を意識し、それぞれが処理するためのプログラムを作成することにする。(C言語のレベルでも並列計算機であることを意識させないことも考えられるがここでは扱わない)。実行主体がひとつではないので、その間で協調する必要がある。逐次処理では、指示された仕事(計算)に「集中して」実行をしていたらよかったが、並列処理では周りと協調する必要がある。しかし、従来のC言語では、並列処理に必要となる協調機能に乏しい。C言語は本来逐次処理用の言語であり、他の実行主体との通信・同期やスケジュール調整といった点が考慮されていないためである。そこで、本研究では、C言語を改良・補強した、より良い言語仕様をもつ並列処理に適した実装

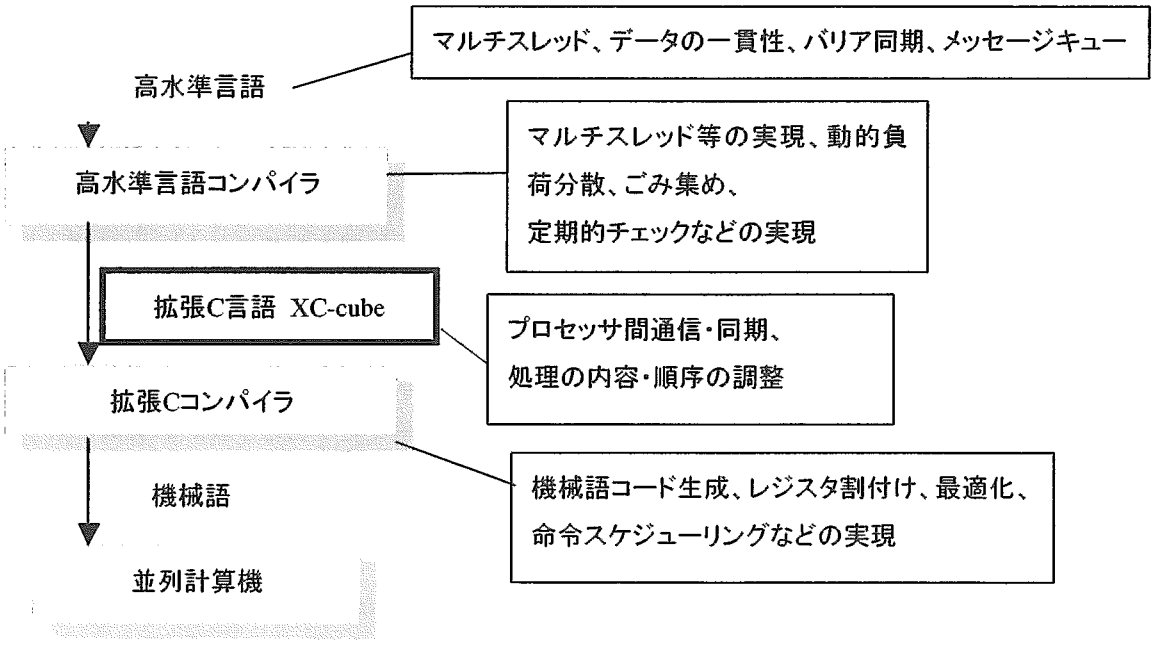

図1. 言語階層を用いた並列計算機上の高水準言語の実装

用言語を目指して並行協調拡張Ｃ言語 XC-cube （an eXtended C language for Concurrency and Cooperation = XC[3]）を設計した。具体的には、そのような協調のための機能として：

・実行主体間で通信・同期をするための機能
（共通の空間でデータを共有するのに必要な機能）
・仕事（計算）の実行手順の調整機能
（手続き呼び出しの枠組みを超える機能）

をどのように言語仕様に加えていけばよいかの研究を行った。

また、

・XC-cube 言語の実装 （XC-cube 言語から機械語へのコンパイル手法）
・XC-cube 言語の利用 （高水準言語から XC-cube 言語へのコンパイル手法）

についても研究を進めた。XC-cube 言語の実装では、さまざまな種類のプロセッサの並列計算機の仕様を調査し、XC-cube で追加した機能（実行主体間で通信・同期をするための機能や、仕事（計算）の実行手順の調整機能）を効率よく実現する手法について研究した。また XC-cube 言語の利用では、仕事（計算）の実行手順の調整機能を利用して、遅延タスク生成に似た負荷分散、マルチスレッドなど、高水準言語が提供すべき機能を実現する手法について研究を進めた。

■ 研究成果

並列処理に適した実装用言語を目指した並行協調拡張Ｃ言語 XC-cube の設計・利用・実装では、効率の良いプログラムを簡単に作成するために一般に使われている２つの抽象化手法を改良した。１つは計算機より人が理解しやすい言語でプログラムを書くというもので、実行主体間での通信・同期が失敗しないような言語設計・翻訳方式を提案した。もう１つはまとまった手続きに名前を付けて呼び出せるようにするもので、従来、呼び出し中に呼び出し元は見えなかったが、並列する他者との協調のために、一旦、呼び出し元にさかのぼって自身のやりかけの処理さえも変更可能とする言語機能と、その実現・利用について提案した。

並列処理では、実行主体がひとつではないので、その間で通信・同期（話し合ったり、通知したり、待ち合わせたりすること）が必要となるが、本研究では、共有メモリの概念に基づいて XC-cube 言語レベルでこれを記述する方式を提案した。また、単に通信・同期をするだけでは表面的に協調しているにすぎないので、各実行主体が、並列する他者との協調のために、一旦、呼び出し元にさかのぼって自身の（元々設定した目標に従って）やりかけの処理さえも変更可能とする。本研究では、入れ子関数の概念に基づいて XC-cube 言語レベルでこれを記述する方式を提案した。

１．実行主体間で通信・同期をするための機能

実行主体間で通信・同期をするための機能として、共通の空間でデータを共有するのに必要な機能について、簡単な例を用いて考えてみる。実行主体が計算をするには、計算の途中結果などをデータとして覚えたりできなくてはならない。例えば、人がソロバンで計算することを考えると、ソロバンの珠の位置で、計算途中のデータや計算結果を表すことになる。このとき、普通は、一つのソロバンを一人が弾いて計算を進める。

さて、ソロバンの上級者となると、計算結果が正しければ、計算途中に忠実に珠を弾かなくても計算できるようになる。頭の中のソロバンで計算しても良いわけで、いわゆる暗算である。ここで、一人で計算しているときは、頭の中に覚えきれないデータだけを実際のソロバン上に記録しておいたり、頭の中でソロバンを弾いた順序とは別の順序で実際のソロバン上に記録したりしてもよい。あるいは、データを予測して、プログラムの指示より先に実際のソロバン上に記録を行ったり、プログラムの指示より後に実際のソロバンから読み取りを行ったりすることも考えられる。いろいろ速く計算するための工夫が考えられるが、計算結果が正しければそれでよい。

ところが、一つのソロバンを複数の人が共通して使い、そのデータを共有するとなると、各人が頭の中だけで保持しているソロバンのデータは勝手なものなのでまずい。つまり、通信・同期のためには、実際のソロバンに対するデータの読み書きなどのアクセス順に関する制約を与える機能などが必要となり、それに従って、必要な部分について頭の中のソロバンの状態と実際のソロバンの状態を一致させたりできなくてはならない。(頭の中のソロバンを使わないで、常に実際のソロバンを使わなくてはならないという方法も考えられるが、さまざまな速く計算するための工夫が許されなくなる場合が増え、効率的ではない。) この他にも、区切りのついたところで、計算がある段階まで終わったことを伝え合ったり、共通のデータについて自分の担当分に限って計算を進めたり、ソロバンの同じ部分を使いたいときは他人がその部分を同時に使わないようにしたりするための機能が必要となる。本研究では、こういった機能を持たせるため、その記述方法と意味という形でその仕様を策定した。ここでは、「頭の中」で高速に計算してよい部分をできるだけ残せる形で、明確にしてやる必要があった。

従来のC言語では、実行主体がメモリに対して行なったロード/ストアなどが実際にどの順序で共有メモリ上で完了すべきかを記述することができず、順序を保証するにはライブラリルーチンなど言語外の機能を用いる必要があった。このため、Dekker のアルゴリズムによる排他制御等が効率よく記述できないばかりでなく、データ領域とその領域への書き込み完了を表すフラグを用いるような単純な同期でさえ効率よく記述できない。また、共有メモリ上での新たな演算(不可分に更新されるリングカウンタなど)を言語そのものでは記述することができなかった。このため、C言語のレベルで提供されるべき仮想的な計算機のモデルが直接提供されておらず、逐次計算機用のCコンパイラが翻訳したコードが並列計算機上で動いているという実装そのものがもたらす変なモデルをCプログラマレベルで考えなくてはならなかった。(図2)

実行主体間で通信・同期をするための機能として、共有メモリ向けプリミティブを一通り設計した。(1)メモリ操作完了順序を指定するメモリバリア用プリミティブ、(2)粒度保証同期変数アクセスプリミティブ、を提供する。さらによく使う機能としてロックのプリミティブ(mutex ロック、reader/writer ロック) を提供する。ここで、粒度保証同期変数アクセスプリミティブには、不可分な読み出し、不可分な書き込み、不可分な読み出し＋書き込み、不可分な比較＋書き込みを用意する。ただし、compare and swap 命令も、予約付き load 命令／条件付き store 命令も持たないプロセッサへの対応としては、一部のプリミティブは使用するとエラーとする。つまり、プロセッサの種類によっては、不可分に読み出し＋書き込みや、不可分な比較＋書き込みに対応できないこともある。

最初のステップとしてあるいは普及の戦略として、これを組み込んだ XC-cube コンパイラを開発する前に、GNU C コンパイラの拡張機能を利用したプリミティブの実装を行った。実装にあたっては、Alpha、MIPS、Pentium、PowerPC、SPARC-V9、SPARC-V8 といった現在代表的なプロセッサアーキテクチャ/マルチプロセッサシステムの仕様の詳細なサーベイを行い、対象とした。

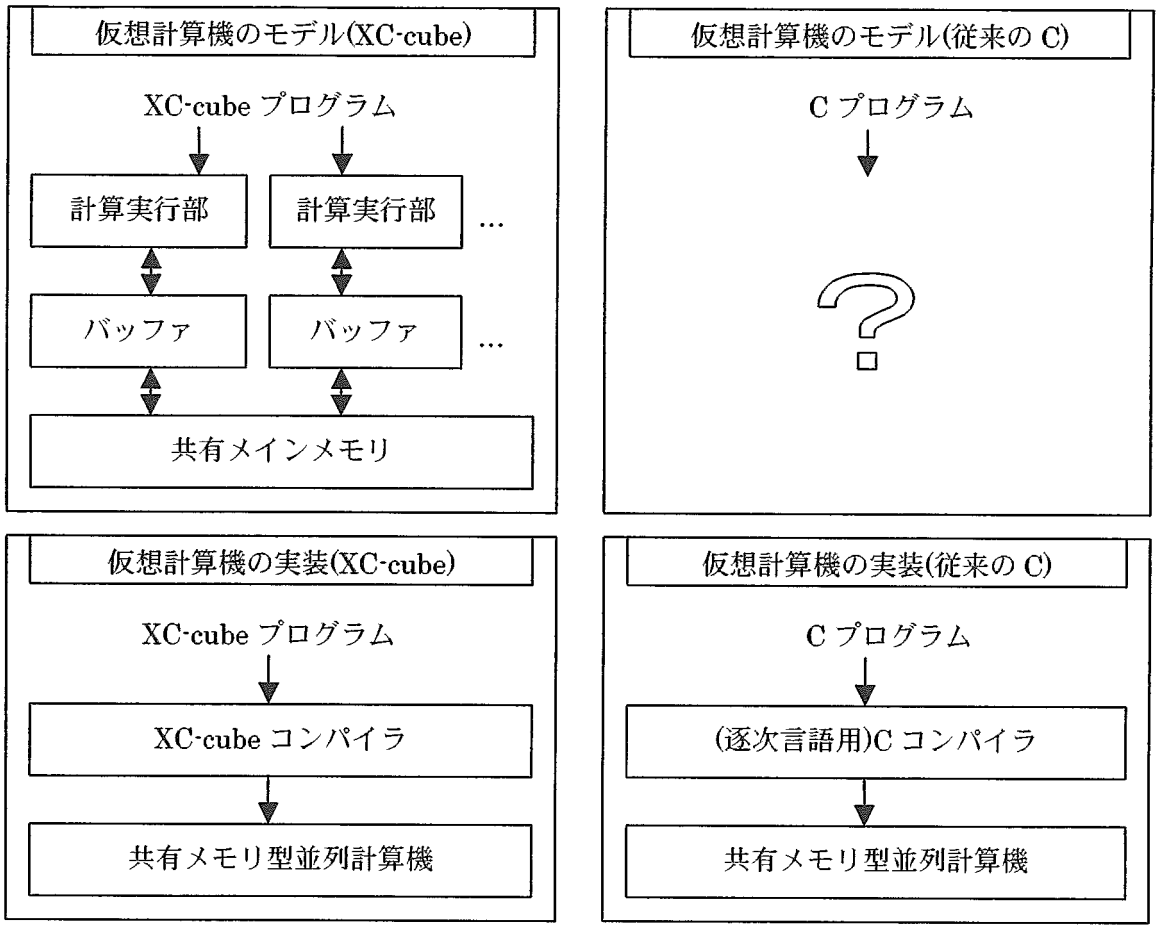

図２．XC-cube 言語／C 言語レベルの仮想計算機のモデルと実装

２．仕事（計算）の実行手順の調整機能

一般に、次に実行できることが複数あったとしても、どれから先にやるのかを決めておくことで、高速な実行が可能となる。つまり、得意なやり方で集中して実行できるように、実行手順を固定化しておきたい。実行手順の固定化は、一つの実行主体のみで計算をするのであれば問題ない。最終結果をさえ求められればよいので、実行手順を実行中に柔軟に変更可能としても得るものはなく、逆にいえば、実行手順を固定化しても失うものはないためである。(厳密にいえば、実行手順の固定化には、計算結果を得るのに必要のない部分の実行は完了しなくて良いNon-Strict計算を行いにくいという欠点はある。)

ところが、並列処理のために複数の実行主体間で協調をする場合は、事情が異なる。他の並列する実行主体からの連絡を待つ間に別の仕事を進めておけるようにできることや、他の実行主体から協力の申し出に対して残りの仕事の一部を分け与えたりできることなどが重要となるためである。そのために、ここでしたいことは、「普段は(半)固定化した通りの実行手順で高速に実行するが、いざというときには実行手順を見直すための仕掛けを準備する」というものである。例えば公共工事の「見直し」のように、一度計画通りはじめた仕事であっても完全には手順を固定化せず周りと協調して途中で手

順を見直せるようにするものである。そのように、元々設定した目標を変更できるようなプログラムを記述できるような言語設計を行う。ただし、これはプログラムをあらかじめそのように書くのであって、プログラムを実行時に書き換えることを意味しているわけではないので注意して欲しい。

C言語は、まとまった手続きに名前を付けて名前で手続きを何回でも呼び出せるようにする手続き型言語の一種であることは既に述べた。「手続き呼出し」では、やりかけの手続きの続け方を覚えておいて、呼び出された手続きを先に実行する。呼び出された手続きの実行が完了したら、やりかけの手続きの続きの実行に戻るとしている。これだと、一度、「手続き呼出し」をすると、やりかけの手続きについては、ただ、続きの実行に戻るのを待つことしかできない。

例えば、関数 f は 関数 g と 関数 h を用いて：

```
f(x) = g(x) + h(x)
```

のように計算するということを表すプログラムがあった場合、f(7)を計算するような呼出しがあったときに、関数 f に関する手続きとしては、x=7 なので、g(7)をまず計算し、次に h(7)を計算して、最後にそれらの結果を足し合わせて f に関する呼出しの結果とするという実行手順で計算すれば、計算結果は正しく求まる。(図３) しかし、複数の実行主体間で協調をする場合は、このように実行手順を

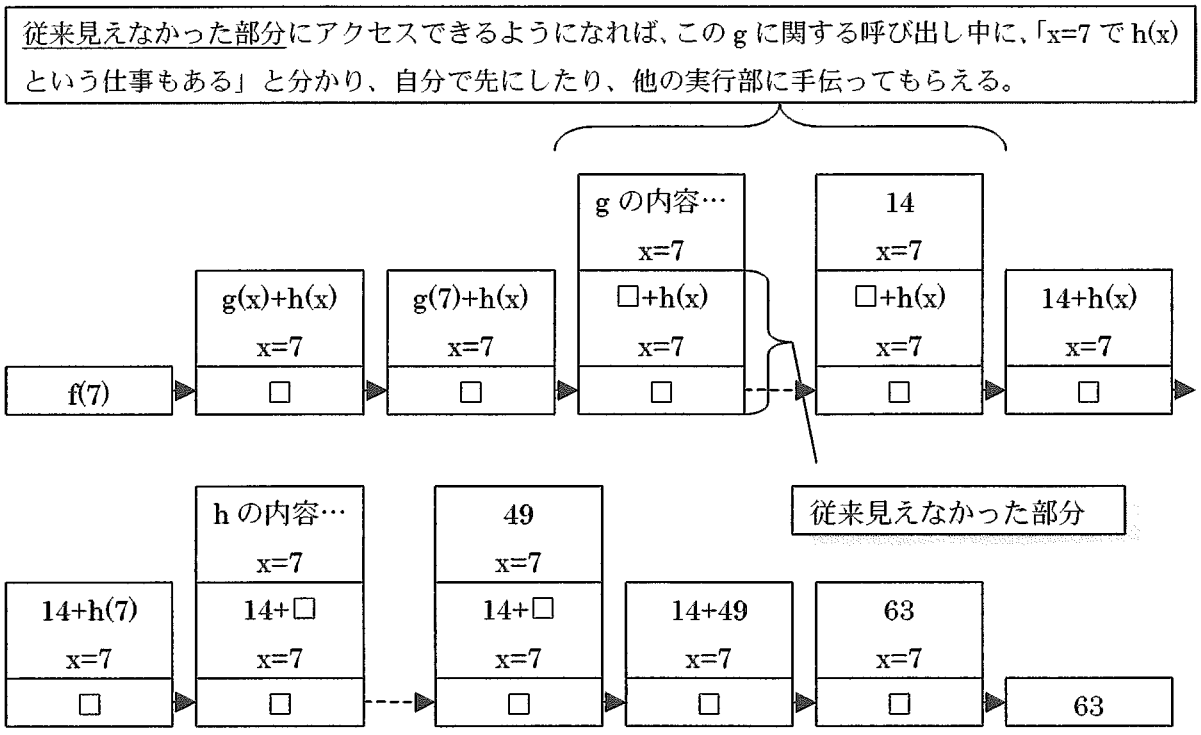

図３．計算時の手続き呼出しとスタック。f(7)を計算。f(x)=g(x)+h(x)

固定化するのではなく、次のように変更できると良い。一つは、g(7)を計算するために関数 g に関する手続きを呼び出している最中に、他の実行主体からの連絡を待たないと g(7)の計算が先に進めなくなったとする。このときに、先に h(7)の計算をやっておけば待ち時間が無駄にならない。つまり、g に関する呼び出しが終わってから実行するとしていた仕事を g の呼び出し中にもかかわらず先にできるようにしたい。もう一つは、他の実行主体からの計算を手伝うと持ちかけられた場合である。このとき、h(7)の計算を手伝ってもらえば、並列に計算が進んで実行時間を短くすることができる。つまり、g に関する呼び出しが終わってから自分がする予定だった仕事を自分では実行しないで他の実行主体に渡せるようにしたい。

これが実現できるためには、g に関する呼び出し中に、一旦、f に関する呼び出しのところまで戻ってきて、x の値や、次は h(x) を計算するのだといったことが分かるようになっていなくてはならない。そこで、各手続きには、半固定化された高速な実行手順以外に「普段は使わないが、いざというときのための手続き」を含ませておく。この手続き中の手続き（入れ子手続き）が呼び出されたときに「元手続きがどういうふうにやりかけであったか」が分かるようになっていればよい。これで、普段は半固定された手順で高速に実行することを可能とするが、いざというときは、この入れ子手続きを呼び出すことで、やりかけの手続きでさえ見直しを可能とできる。

高水準言語には、上で例でいえば g(7)と h(7)を並列に実行できるようにするようなマルチスレッドの機能を持つものが多い。一つの実行主体上に複数のスレッドを持たせておき、あるスレッドが他の実行主体との兼ね合いですぐに処理を続けられなくなっても他のスレッドに切り替えて実行主体としては別の処理を続けることが期待される。このようなスレッドは POSIX threads などの既存のスレッドライブラリを用いても実現できるが、スレッドの生成や切り替えのオーバヘッドが大きく、細粒度の並列言語での利用では使いものにならなかった。

入れ子手続きとしては、GNU C コンパイラ（GCC）の入れ子関数に相当する機能、を提供することにした。マルチスレッドの実現などは高水準言語から GCC の拡張言語、あるいは、XC-cube 言語へと変換する際にこれを実現する。まず、マルチスレッドの実現より比較的容易な遅延タスク生成の実現も GCC の入れ子関数を利用して行った。GCC の入れ子関数はそのままではオーバヘッドが大きいが、フィボナッチ数の再帰計算をそれぞれ並列に行うような極端に細粒度の並列処理においても 6 倍程度のオーバヘッドで並列処理が可能であることが分かった。

入れ子関数を利用することで、従来のＣ言語では記述できなかった「他の実行主体と協調した自身の実行順序の調整」が可能となる理由は次のようなものである。従来のＣ言語では、関数呼出しを行っている間、実行主体自身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。GCC の入れ子関数は関数内で入れ子に定義された関数であり、その関数ポインタを一種のクロージャとして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。

遅延タスク生成の実現より課題の多いマルチスレッドの実現については、以前から研究していた高水準言語であるオブジェクト指向並列言語 OPA のためのコード生成手法が変換手法の基礎として利用できた。このコード生成手法はＣ言語への変換を行うものであるが、Ｃのスタック領域から退避するためのヒープ領域を必要としたり、関数フレームの変換を必要としたり、局所変数のポインタを呼出しの引数にできないといった問題点がある。入れ子関数の機能を用いれば「(呼出し中の)まだできな

い仕事を後回しにする」のではなく「(呼出し後の)先にできる仕事を先取りする」ことで、マルチスレッドが実現でき、問題点も解消された。

GCC の入れ子関数は、その関数ポインタを一種のクロージャとして利用できるが、関数ポインタとの互換性のためにいわゆるトランポリンという技法を用いていたり、他のプロセッサからも呼び出せるようにするために効率的なレジスタの使用が妨げられるなどオーバヘッドも大きい。XC-cube では、同等の機能を効率よい軽量クロージャとして実現する技法を開発している。これは、関数ポインタとは別扱いすることでトランポリンなどを不要とするとともに、はじめて呼び出されたときに変数の場所をレジスタからスタックへ移動させるというものである。

■ 今後の展開

既存のC言語に対する不満があってなんとかできないかという動機で始めた研究だが、研究開始した時点では具体的なアイデアがあったわけではなく、まずは詳細な調査を行うということから始めた。研究の蓄積があったわけでもなく、先の見えない中での手探りのスタートであった。

最終的に、得られた成果の一つは、共有メモリのモデルをC言語のレベルで与えるにはどうするかということだった。この分野は、元々ハードウェアのレベルで議論されることが多かったが、モデルができあがってから、改めて調べてみると、例えば Java 言語においても Java 言語の提供する共有メモリモデルは何かということで研究がされている。ある意味では、似たようなモデルになっているが、逆にいえば、他の言語でもやはりモデルが必要であり、他の言語にも本研究の成果を活かせる可能性があることがわかる。

もう一つ得られた成果の一つは、入れ子手続きをC言語の機能として追加しておけば、C言語のプログラムへうまく翻訳することでマルチスレッドや自動負荷分散を提供する高水準言語の実装ができることに気が付いたことだった。これは、研究期間も終盤にさしかり、2000 年秋の「情報と知」領域会議が終わって帰ろうとしたときに、ふと思いついたアイデアだった。このアイデアは非常に面白く、この研究で行った並列処理以外にも、ごみ集め処理や Non-strict 言語の実装などにも応用できそうである。

今後は、XC-cube 言語処理系を完成させていくとともに、性能測定などを行っていきたい。また、XC-cube 言語を実際に高水準言語の実装に用いて、アプリケーションの記述や評価を行っていきたい。

■ 成果リスト

雑誌論文：

- 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), November 2001.
- 八杉 昌宏、田畑 悠介、小宮 常康、湯淺 太一、"共有メモリ向けプリミティブとその GCC を使った実現"、情報処理学会論文誌:プログラミング、Vol. 43, No. (PRO 13), January 2002. (採録予定)
- 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. (PRO 14), March 2002. (採録予定)

その他の論文・口頭発表 :

- 八杉昌宏、馬谷誠二、小宮常康、湯淺太一、"マルチコンテキスト管理をサポートする実装用言語"、情報処理学会プログラミング研究会 (SWoPP'99), August 1999.
- 八杉昌宏、田畑悠介、小宮常康、湯淺太一、"共有メモリ関連命令を生成可能な実装用言語の設計"、情報処理学会プログラミング研究会 (SWoPP2000), August 2000.
- 八杉昌宏、"入れ子関数を利用した準遅延タスク生成の検討"、日本ソフトウェア科学会第 4 回プログラミングおよび応用のシステムに関するワークショップ (SPA2001), March 2001.