プログラミング 14−14
(1997. 8. 21)

Cプログラムにおける Lazy Task Creation

田端 邦男　　田浦 健次朗
米澤 明憲
{tabata,tau,yonezawa}@is.s.u-tokyo.ac.jp

東京大学大学院理学系研究科情報科学専攻

要旨

共有メモリマシンにおけるCプログラム上の関数呼び出しに対し、動的負荷分散を目的としたLazy Task Creationを実装した。細粒度並列処理において、負荷分散に要するオーバヘッドが処理全体の効率に大きく影響する。Lazy Task Creationという方法は、プロセッサがアイドル状態になった時のみに仕事を生成することで、負荷分散のオーバヘッドを小さく押えられる。基本的なアイデアは、スタックフレームにある情報から仕事を生成するということである。既存の実装ではスタックフレームのフォーマットやランタイムシステムなどコンパイラ全体を特定のプログラミングシステムのためにはじめから作り出しており、この方法では、C言語に対しLazy Task Creationを実装することができなかった。本研究では、標準的なスタックフレームのフォーマットに対し、既存のCコンパイラを変更することなく実装する方法を示す。この方法は、既存のコンパイラの逐次実行の最適化を最大限に生かすことができる。実験を通して効率の高い動的負荷分散を実現できることも確かめられた。

Lazy Task Creation for C Programs

Kunio Tabata　　Kenjiro Taura
Akinori Yonezawa
Department of Information Science,The University of Tokyo

abstract

We have implemented a Lazy Task Creation scheme for C programs on shared memory machines. On fine-grain pararell programs, overhead of dynamic load-balancing has a large impact on overall performance of applications. Lazy Task Creation (LTC) is a method that reduces this overhead by generating work on demand, only when some processors are idle. The key idea is to generate work from information present in stack frames. To make work generation as simple as possible, LTC or similar schemes have traditionally been implemented with stack frame formats, calling conventions, runtime systems, and code generators desigined from scratch for a particular programming system. This prevents language implementers from targetting their languages to C. This paper shows that an LTC-like mechanism can be implemented with unmodified sequential C compilers, with standard stack frame formats and calling conventions. Therefore, our method takes advantage of optimizations already implemented in C compilers. Moreover,because it isolates a machine-dependent part, it is easy to apply our implementation to various platforms. Through the experiments,we confirm our implementation has good performance.

1　はじめに

一般に並列処理の効率を決める要素は様々であるが、特に細粒度並列処理の場合、負荷分散に要するオーバヘッドが大きいと処理全体の効率が悪くなる。というのは、実際の仕事が小さいので負荷分散のための操作に要する時間は相対的に大きく見えるからである。よって、負荷分散に要するオーバヘッドを押えることが処理全体の効率を高めることになる。特に、全てのプロセッサにすでに仕事が割り当てられている状況のように、負荷分散の必要がない状況における、実際の仕事以外のオーバヘッドは明らかに無駄である。

Lazy Task Creation[4, 2] は、樹状再帰的に仕事を生成するプログラムに対して有効な動的負荷分散の方法である。これは、アイドル状態のプロセッサがいない状態のように、負荷分散の必要がない状態でのオーバヘッドを最小限に押えることで高い効率を得るため、細粒度の並列処理にも適する。また、これまでのLazy Task Creationの実装は独自の形式のスタックフレームを用いているため、開発が非常に大変であった。我々はLazy Task CreationをCの関数呼び出しに新しい方法で実装することで、少ない開発コストで、高効率かつ移植性の高い動的負荷分散を実現した。特に、並列計算のための計算機言語を実装する際、本研究の実装を伴ったC言語を中間言語とすることで、負荷分散を実現する手間がおおいに省けることになる。

以下では、2章でLazy Task Creationの概要を述べ、3章で関連研究について述べる。4章で本研究での実装法の概要を述べる。5章でその実装の詳細を述べる。6章で実験結果を述べ、7章で本実装の価値について結論し、今後の課題を述べる。

2 Lazy Task Creation

オリジナルのLazy Task Creation[4]は、Multilisp[6]で実装されている。この言語には、プログラマが明示的に並列実行を指示できる*future*というプリミティブが実装されている。これは、

$$(K\ (future\ X))$$

と書いてXとKを並列に実行することを意味する。素直な実装では、Xを他のプロセッサに割り当て、Kを自分で実行することになるだろう。もし、この時アイドル状態のプロセッサがいなく、結局、後で自分でXを実行することになる場合は、このような方法ではXを割り当てようとするオーバヘッドの分だけ効率が悪く、逐次実行の方が効率が良い。また、このように、KとXを逐次で実行したい場合、もともと並列で行なっても良い計算なので、その実行順に制限がない。従って、Xを先にKを後に実行しても良い。

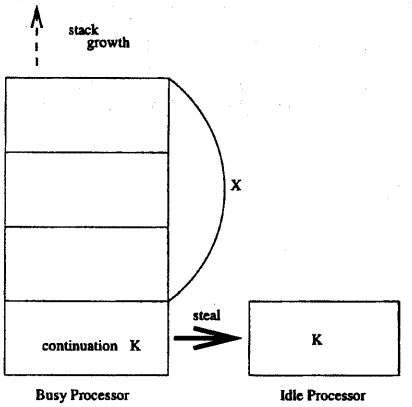


図1コンティニュエーションを盗みとる

Lazy Task Creationでは、この考えを発展させたものである。プロセッサPが$(K\ (future\ X))$を実行する場合、(1)Xを実行し、Kはスタックフレーム上に残しておく。(2)PがXを実行中にアイドル状態のプロセッサ(これをQとする)が現れたら、QはPのスタックフレーム上のコンティニュエーションKを盗みとり、実行する(図1)。もしアイドル状態のプロセッサが現れないままXの実行を終えたら、Pが続けてKを実行する。

ここで、(1)においてPがKをスタックフレーム中に保存する操作は、通常の逐次実行においても必要なものである。盗み取るプロセッサQは、この情報をそのまま利用する。こうすることで、アイドル状態のプロセッサがいない時に、Pが将来仕事を割り当てるのに備え特別な処理をする、というオーバヘッドが小さい。したがって、負荷の移動が起こらない時の、オーバヘッドを少なくすることができる[1]。

以上のようにLazy Task Creationを実現するためには、スタックフレームに関して、単なる関数呼び出しとそのリターン操作が可能なだけでは不十分で、現在実行中でない部分を突然盗むという操作が可能でなければならない。これは、一般に逐次計算を目的とした言語では不可能な操作である。我々の実装の対象となるC言語でもこの操作は不可能である。この操作を可能にする方法の一つは、コンパイラを新たに作ったり、既存のコンパイラを改造することである。しかしこの方法は、開発コストが高く、またコンパイラへの依存性が高いため移植性が低い。よって我々は、C言語のスッタクフレーム管理方式をそのまま利用し、アセンブルポストプロセッシングにより、フレーム管理に必要な情報を取り出すという方法をとる。

3 関連研究

Lazy Task Creationにおいて、プロセッサ間の仕事のやりとりの方法という視点から見ると、既存の実装法は2種類に分類できる。本研究では後者の方法を選択した。以下ではその2種類の方法と本研究との関連を述べ、後者を選択した理由について述べる。また、Cの関数呼び出しへ実装という本研究との共通点を持つ、動的負荷分散をしないLazy Task CreationであるStackThreads[8]との関連について述べる。

3.1 Task Stealing

Mohrらのオリジナルの Lazy Task Creation[4]では、アイドル状態になったプロセッサが、仕事が残っているフレームを見つけ、みずからその仕事を盗みに行く(task stealing)。この方法を実現するには、スタックフレームに関する詳細な情報を実行時に他のプロセッサから監視できることが必要である。具体的に言うと、フレーム境界や仕事の残っているフレームの場所などが分からなければ、仕事を盗むことはできない。このことは、Lazy Task Creationを実現するには単なる関数呼び出しで使われるフレームフォーマット管理方法では不十分なため、特別なフレーム管理機能が必要であるということを意味する。このためこれまでのLazy Task Creationやそれに類似した方式では、スタックフレームのフォーマットを新たに設計し、それに合わせたコード生成器を実装する必要があった。例えばLazy Threads[3]などではGCCコンパイラを改造することでこの方式を実現した。また、[7]などでは、アセンブラに非常に近いCプログラムを出力する形式でコンパイラを新たに開発した。これに対し、本研究では、既存のプログラミング言語の高効率に逐次実行できる部分をそのままに、並列負荷分散機能だけを実装したい。我々の実装の対象であるC言語では、上述したような他のプロセッサのスタックフレームの詳細な情報は参照できない。よって、この方法を使わないことにした。

3.2 Message Passing Lazy Task Creation

Message Passing Lazy Task Creation[1]とは、アイドル状態のプロセッサが忙しいプロセッサに信号を送ったり、忙しいプロセッサが定期的にアイドル状態のプロセッサを見つけにいく方法である。つまり、アイドル状態のプロセッサが生じた時には、忙しいプロセッサの方が自分のスタックフレームを遡り、残っている仕事を見つけ、アイドル状態のプロセッサにそれを割り当てる。忙しいプロセッサは、仕事を割り当てた後、自分の仕事に復帰する。この方法では、以前のフレームに一時的に戻ると言う処理が必要であり、

[1]粒度調節などの付加的な機能を実現するために、この部分でオーバヘッドが生じることがある。

これは、忙しい方のプロセッサへのオーバヘッドになる。本研究では、この方法に基づく実装を、既存のC言語のスッタクフレーム管理法をそのままに実装した。

3.3 StackThreads

動的負荷分散をしないものに限れば、StackThreads[8]は分散メモリ環境でのLazy Task CreationのCへの実装である。StackThreadsは、プロセッサ内のスレッドのみが効率良く動作する環境を提供し、プロセッサ間の負荷分散はユーザに任されていた。それに対し、本研究では、動的負荷分散を目的として、共有メモリマシン上においてプロセッサをまたいだスレッド間での仕事のやりとりを効率良く実現できる。さらに、アセンブリポストプロセッシングによってCコンパイラへの依存性をなくして、移植性を確保する。

4 実装方式の概略

我々の実装では、Message Passing Lazy Task Creationを基本とした方法をとるため、忙しいプロセッサが、アイドル状態のプロセッサを見つけると、自分のスタックフレームを遡り、やり残している仕事を見つけ、割り当てるという操作が必要である。

これら一連の処理は、しばしば新しい処理系プログラミング言語で目にする例外処理(exception)[2]とよく似ている[3]。特に、例外が発生したところに再び戻ることができる、resumeという機能をもつ、resumptiveな例外処理に類似している。フレーム巻き戻しは、ちょうど例外処理のtryブロック中の関数呼び出しの中でthrowを呼んだ状況と似ている。また、このあと他のプロセッサに仕事を与える処理をするには、例外処理で言うところのcatchブロックにその処理を書けば良い。仕事を与えたあと、元の処理に戻るにはresumeを実行すればよい。よって、説明を簡便にするため、以降でで示す具体例にはtry,catch, resumeなどの例外処理の語を、それと類似した処理を表す言葉として使用する。

我々の実装方式による、これらの処理の具体的なプログラム例を示す。$f(x)+g(x)$を、$f(x)$と$g(x)$を並列に実行し、その結果を足すという方法で計算するには、まず、tryブロックに書かれている$f(x)$を単に逐次実行する。この計算の途中で、アイドル状態のプロセッサが見つかった時点でthrowすることで、catchに処理が移る。すなわち、$f(x)$の仕事を一時中断して、この$g(x)$という仕事が残っているフレームに戻り、アイドル状態のプロセッサに$g(x)$を割り当てる(図2)。ただ、本実装方式では、throwの直前でアイドル状態のプロセッサが見つかったという情報を大域変数に代入し、catchの時点でその変数の値を判定しその後の処理に移る。アイドル状態のプロセッサが見つからなければ$f(x)$の実行を終えた後、$g(x)$をみずから逐次計算する。最後に結果を加算する。

```
b=0;
try{
    a=f(x);
}catch(found idle processor){
    b=1;
    c=give g(x);
    resume();
} yrt;
if(b==0)
    return  a+g(x);
else
    return  a+c;
```

図2 並列実行の例

このようなことを実現するには、一時的に以前のフレームに巻き戻るという処理が必要である。これを、C言語に対し実装するには様々な困難がある。1つは、リターンアドレスとフレームポインタの復帰の問題である。ある関数がリターンするアドレスは、その関数のスッタック中に格納されている。同様にリターンした後のフレームポインタの値も、その関数のスタック中に格納されている。例えば、多段に関数呼出ししている状態から元の状態に一度に戻りたいなら、呼び出された逆の順番にスタック中のフレームポインタとリターンアドレスを復帰しなければならない。もう1つは、callee-save-registerの復帰の問題である。callee-save-registerとは、各関数の呼ばれ側に退避する義務があるレジスタであり、このようなレジスタがあることで、変数をより多くのレジスタのに割り当てられるようになり、逐次実行の高速化につながる。ただ、これらも、各関数のスタックフレーム中に退避されることになるので、以前のフレームに巻き戻る際に順に復帰する必要がある。

それらの情報が保存してあるスタック上の位置を知るため、本研究ではアセンブラポストプロセッシングにより、必要な情報を取り出しておき、プログラム実行中に参照できるようにした。詳しくは次章で述べる。

5 実装の詳細

5.1 概観

我々の実装でモジュール化した、プラットフォームに強く依存する部分と汎用性の高い部分とは、フレーム巻き戻しという部分とそれを個々のアプリケーションに適用するという部分である。フレーム巻き戻し部分とは、忙しいプロセッサがアイドル状態のプロセッサを見つけた時に、仕事の残っているフレームまで戻る処理を指す。また、それを個々のアプリケーションに適用する部分とは、巻き戻った後に仕事を割り当てる部分や、割り当てた仕事の結果を処理する部分など、アプリケーションに強く依存する部分を指す。

以下では、まずフレーム巻き戻しという機能の必要性について論じ、その概要、詳細について述べ、適用法について簡単に触れる。なお本研究では、全ての実装をUltraSparc上のSolarisで行ない、GCCにLazy Task Creationを実装した。また、一般性を高

[2] C++やJavaのexceptionではthrowした場所に戻るresumeという機能は実装されていない。

[3] ただ本研究の実装はあくまでも例外処理に似ているだけで、完全にその機能を果たしているわけではない。例えば、tryブロックには負荷分散の対象のただ1つの関数呼び出ししか書けない。

めるため、GCC は -mflat で実行し、Sparc のレジスタウインドウは使わない。

5.2 フレーム巻き戻し処理の必要性

Lazy Task Creation を実現するには、現在実行中のフレームから、仕事が残っているフレームに戻る必要がある。また、そこで仕事を与えた後、元のフレームに戻れなければならない。これの実装法として簡単に考え付くのは、関数呼び出しの直前でスタックフレームやレジスタの状態を退避しておいて、そのフレームに戻りたい時にそれらを復帰するというやり方である。しかし、このやり方では実行効率が悪い。上述した通り、Lazy Task Creation は、他のプロセッサに仕事を割り当てたい時にのみオーバヘッドが生じるため効率が良い方法であった。すなわちここでは関数呼び出しの直前の状態に戻りたい時にのみオーバヘッドが生じるようにしたい。しかし、初めに挙げた実装法では、関数呼び出しの直前すなわち逐次処理の時点でスタックフレームやレジスタを退避するというオーバヘッドが生じてしまう。そこで、仕事が残っているフレームに戻る時にのみオーバヘッドが生じる、フレーム巻き戻し処理が必要になる。。

5.3 フレーム巻き戻し処理の概要

フレーム巻き戻し処理とは関数のエピローグコードをたどって呼び出した時の状態に遡ることである。エピローグコードとは、C の関数それぞれの最後にある命令列のことで、ここでスタックフレームや callee-save-register を関数呼び出しの直前の状態に戻す処理をしている (図 5 左)。従って、多段に関数呼び出ししている状態から、元の状態に一度に戻るには、呼び出した関数全てのエピローグコードを呼び出し逆順にたどればよいことになる (図 3)。このとき、巻き戻しの直前のレジスタやスタックフレームの状態は後で巻き戻しの直前の状態に戻れるように退避しておく。また、同様の理由で、この巻き戻しの過程や、巻き戻してから元に戻るまでの間の処理で、途中のスタックフレームを破壊しないようにする。さらに、巻き戻るべきフレームに達したあと、仕事を割り当てる処理の部分に制御を移す必要がある。よって、巻き戻って何か処理をしたあと元に戻る処理は、あらかじめ退避した巻き戻る直前のスタックポインタとフレームポインタ、callee-save-register を復帰することで実現する。

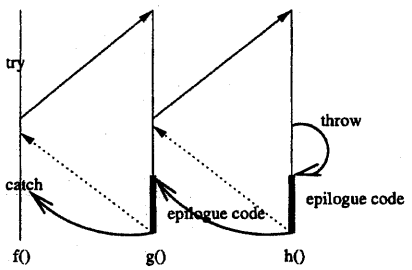


図 3 巻き戻し

3 つの関数 f,g,h があり、f() が g() を、g() が h() を呼び出す状態を例にとる (図 3)。h() の中で、throw が実行されると、まず h() のエピローグコードを実行し、次に g() のエピローグコードを実行し、あたかも g() を呼び出す直前の状態に戻る。ただ、仕事を与える処理をする部分に実行を移さなければならないので、g() の直後から処理を再開する代わりに、catch ブロックに実行が移るようにする。つまり、手続きとしては、try ブロック中に巻き戻ったかどうか判定しながら、エピローグコードを一つづつ実行する。

```
f(){              g(){        h(){
  try{              h();        throw();
    g();          }           }
  }catch{
    give task;
    resume();
  }yrt;
}
```

図 4 プログラム例

5.4 フレーム巻き戻し処理の詳細

エピローグコードのアドレスや、catch ブロックの先頭のアドレスを知る方法として、C のマクロと、C コンパイラが生成したアセンブラコードに対するポストプロセッシングにより必要な情報を取りだし、表を作るという方法を採った。この方法はコンパイラを改造する必要がないため、開発の労力が小さい。また、同じ理由で、コンパイラによる逐次部分の最適化を最大限生かすことができる。さらには、様々な種類の C コンパイラに適応させるのも比較的容易である。

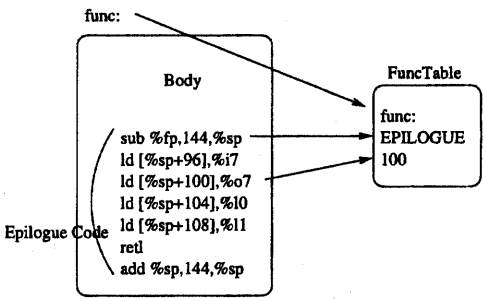


図 5 エピローグコードと FuncTable

図 5 は、図 4 をコンパイルしたアセンブラから FuncTable というスタックフレームに関する情報を管理する表を自動生成する様子を表している。フレーム巻き戻し処理の際、各関数の戻りアドレスが重要な役割を果たすので、関数表のエントリには、各関数の先頭アドレス、エピローグコードのアドレスの他に、戻りアドレスが格納されている場所のフレームポインタからの相対位置を加える。巻き戻し処理の後に適切な処理をするためにもう 1 つ trycatch 表が必要である。これも同様にアセンブラから自動生成する。巻き戻し表には、巻き戻るべきフレームの開始アドレス、終了アドレスを格納しておく。

関数 h() で、throw が呼ばれると、関数の戻りアドレスを見て、巻き戻し表の第 1 要素と第 2 要素との大小関係から catch ブロックに達したかどうかを判定する。この場合 catch ブロックは f() にしか存在しないので巻き戻し処理に移る。再び関数の戻りアドレスを見て、関数表の第 1 要素と第 2 要素との大小

関係から自分がどの関数から呼ばれたかを調べる。h() を特定できたら、今度は h() の戻りアドレスを格納しているスタック上のアドレスを関数表を使って調べ、そこに throw 内の次の処理のアドレスを入れる。そして、h() のエピローグコードにジャンプする。つまり、h() のエピローグコードだけ実行して戻ってくることになる。以上を繰り返し、g() のエピローグを実行した直後、try ブロックに達したので、trycatch 表を見て、catch ブロックのアドレスを知り、catch ブロックに処理を移す。

catch ブロックに処理を移す方法としては、C のマクロ展開を利用した。

- try を if(begin_try_block()==0) に
- catch(e) を else if(excpt==e) に
- throw(e) を excpt=e;throw_exception(); に
- yrt を else throw_exception(); に

展開する。ここで excpt は大域変数である。また、begin_try_block という関数は実体のない関数で、アセンブラポストプロセッシングにより、戻り値を 0 にする処理に置き換えられる。つまり、この if 節は必ず条件成立し、つまりは、try ブロックの内容は必ず実行されることになる。また、throw_exception という関数は、上で述べたような一連のスタックフレーム巻き戻し処理を行なう関数である。よって、巻き戻した後に大域変数の値を判定し条件成立すれば catch ブロックを実行し、しなければ、さらに外側の catch ブロックを探して巻き戻し処理を続けることになる。したがって、throw する値を調節することで、多段に関数呼出しされた状態から任意のフレームに戻ることができる。

遡って何か処理をしたあと元に戻る処理は、h() が呼ばれた直後にスタックフレームの位置やレジスタの内容を退避しておき、何か処理をしたあと resume() が呼ばれた時にそれらを単に元に戻すことで実現できる。ただ、この方法では、catch と resume の間で実行中のスタックの内容を壊してしまう可能性がある。よって、catch と resume の間では関数呼び出しができない。

5.5 フレーム巻き戻し処理の適用の方法

これらを樹状再帰呼び出しに適用するには、再帰呼び出しの枝の 1 つを try と catch の間で実行する。catch と resume の間に、他のプロセッサに残りの枝の 1 つを与える処理を実行させる。はじめの枝の実行を終え、まだやり残した枝があったらそれをやる。全ての枝を実行終了あるいは与え終えた時点で他のプロセッサの結果を待ち合わせる。まだ結果が揃っていなかったら、自分が仕事をしていないプロセッサとなり、他の仕事をもらう。結果が揃った時点で、今やっている途中の仕事を終えたらリターンする。

具体的なアプリケーションは次章で述べる。

6 実験結果

フィボナッチ数列を単純な樹状再帰呼び出しで計算する C プログラムと RNA の 2 次構造を予測する [5]C プログラムを実装した。これらは、前章のフレーム巻き戻し処理を使って動的な並列負荷分散する。これらの実装方法は非常に簡単なことは前章で述べた通りなので、アプリケーションに特化した部分の実装について以下で述べる。また、実験環境、測定項目について簡単に述べる。

```
fib(n){
    if(n<2)
        return n;
    try{
        fib(n-1);
    catch(found_idle_processor){
        give fib(n-2);
        resume();
    }yrt;
    if(gave)
        wait fib(n-2);
    else
        fib(n-2);
    return result;
}
```

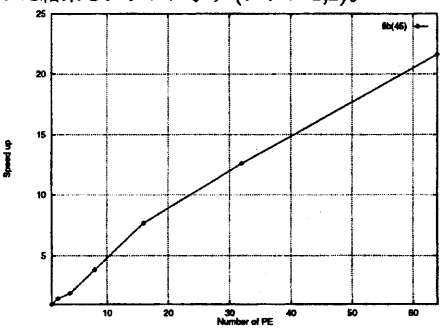
```
rna(d,p){
    if(n==maxn)
        return;
    try_find_answer();
    for(;p;p=p->next){
        try{
            rna(d+1,p)
        catch(found_idle_processor){
            p=p->next
            rna(d+1,p);
            resume();
        }yrt;
    }
    wait_all_result();
}
```

図 6 フィボナッチと RNA

フィボナッチ数列の第 n 項を計算するには、第 n-1 項と第 n-2 項を足せば良い。他のプロセッサから仕事の要求がない限り、第 n-1 項を再帰呼び出しで計算し続ける。仕事を要求されると第 n 項を計算するフレームに巻き戻り、第 n-2 項を計算するという仕事を与える。プログラム例を図 5 に示す。

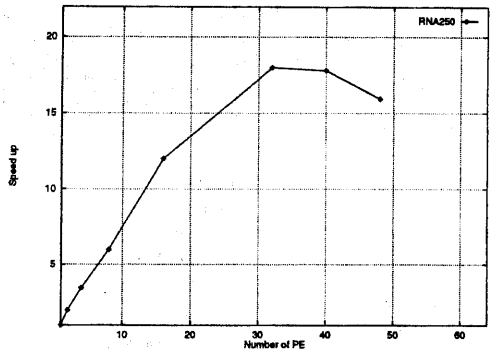
RNA2 次構造予測プログラムでは、解を探すために樹状再帰呼び出しをする。ただ、フィボナッチ数列との違いは一つのノードから出る枝が 2 本とは限らないことである。このプログラムを並列化は、全ての枝を探索し終えるまでループしながら逐次に再帰呼び出しをし、アイドルなプロセッサを見つけたら次の枝を渡すという方法をとった (図 6)。

これを Sun Ultra Enterprize 10000[4] 上で実行した。全体の実行時間をプロセッサ台数をパラメータとして測定することにより、逐次アルゴリズムで同様の計算をした場合に対する台数効果を調べた。フィボナッチ数列の第 45 項目を求めるプログラムの実行結果と、RNA の 2 次構造予測を 250 要素について行なった結果をグラフに示す (グラフ 1,2)。



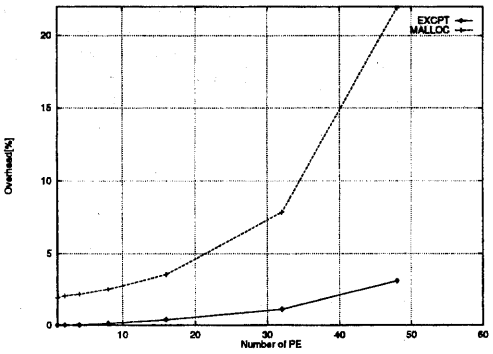
グラフ 1 フィボナッチ 45 の台数効果

[4]Ultra Sparc 255 MHz * 64



グラフ 2 RNA250 の台数効果

フィボナッチ数列は 64 台の並列負荷分散で逐次処理の約 30 倍の速度で実行できるのに対し、RNA は 32 台の並列負荷分散で逐次処理の約 18 倍の速度で実行できるのを最高に効率が上がらなくなる。この原因は、並列化による何らかのオーバヘッドが生じているためと考え、2 種類のオーバヘッドについて測定を行なった。プロセッサ全体の全処理時間の合計に対するオーバヘッドの割合の、プロセッサ台数による変化を示す (グラフ 3)。オーバヘッドの種類は、スタックフレーム巻き戻しに要する時間 (EXCPT) と、動的なメモリアロケーションにかかる時間 (MALLOC) である。プロセッサ台数を増やすことで動的なメモリアロケーションに要する時間がオーバヘッドになり、全体の処理速度の台数効果低下につながっているのが分かる。この極端なオーバヘッドの増加の原因はメモリアロケーションのやり方に原因がある。動的なメモリアロケーションは、共有メモリ上で複数プロセッサで共有する自由領域から新たな領域を確保することで実現している。よって、この自由領域に対し排他制御が必要になる。プロセッサ台数が増加することでこの自由領域に同時に複数のプロセッサがアクセスしようとする頻度が高くなる。結果的に自由領域に直ちにアクセスできず、待ち状態になるプロセッサが増え、オーバヘッド増加につながると考えられる。



グラフ 3 RNA250 のオーバヘッド

7 結論・今後の課題

実験を通して、樹状再帰呼び出しをするアプリケーションに対し効果的な動的負荷分散が実現できることが確かめられた。また、負荷分散に要するオーバヘッドがわずかであるため、大きな台数効果を得られることが確かめられた。さらに今回 2 つのアプリケーションに負荷分散機能を実装することで、フレーム巻き戻し処理さえ用意されていれば、個々のアプリケーションに負荷分散機能を実装するのは比較的容易であることも、定性的に確かめられた。

今後の課題としては、RNA の 2 次構造予測アプリケーションを並列化したことで明らかになったメモリアロケーションのオーバヘッドの問題を解決し、このようなアプリケーションがさらに大きな台数効果をあげられるようにすることが挙げられる。また、もう 1 つの課題としては逐次アルゴリズムで記述された C のプログラムにコンパイラが自動で負荷分散機能を付け加えることが挙げられる。つまり、本研究の特徴の、個々のアプリケーションに負荷分散機能を実装するのが容易であることを発展させ、この部分を自動化しようということである。

参考文献

[1] Marc Feeley. A message passing implementation of lazy task creation. In Robert H. Halstead, Jr. and Takayasu Ito, editors, *Proceedings of International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, No. 748 in Lecture Notes in Computer Science, pp. 94–107. Springer-Verlag, 1993.

[2] Mark Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.

[3] Seth Copen Goldstein and Klaus Erik Schauserand David Culler. Enabling primitives for compiling prallel languages. In *Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995.

[4] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A techinque for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280, July 1991.

[5] Akihiro Nakaya, Kenji Yamamoto, and Akinori Yonezawa. Rna secondary structure prediction using highly parallel computers. In *Comput.Applic.Biosci.(CABIOS)*, 1995.

[6] Jr. Robert H.Halstead. Multilisp:a language for concurrent symbolic computation. In *ACM transactions on Programming Languages and Systems*, pp. 7(4):501–538, 1985.

[7] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers.message passing implementation of lazy task creation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPOPP)*, pp. 218–228, 1993.

[8] Kenjiro Taura and Akinori Yonezawa. Fine-grain multithreading with minimal compiler support a cost effective approach to implementing efficient multithreading language. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.