卒業論文

OpenMP による共有メモリ型並列プログラミング[II]

氏名：土屋 悠輝
学籍番号：2210960144-0
指導教員：山崎 勝弘 教授
提出日：2000 年 2 月 18 日

立命館大学 理工学部 情報学科

内容梗概

共有メモリマルチプロセッサシステムの普及を背景に、いままで移植性のなかった共有メモリマルチプロセッサシステムの並列化指示文に共通化を持たせたOpenMPが提案され、注目を集めている。本研究では、RWC OpenMPコンパイラと実行時ライブラリを用いて、符号化問題（ランレングス圧縮）と文字列照合問題（KMP法、BM法）を解くためのCプログラムの並列化を最大4 CPUで行い、OpenMPにおける評価を行った。

目 次

付録

1　はじめに

近年多くの計算機が導入され、その性能向上はとどまるところを知らない。さらにこれらは非常に安価なものとなり、手に入れやすくなった。さらに長時間かかっていた計算も、コスト、柔軟性、および信頼性に優れたシステムにより、短時間で処理することができるようになった。このように諸問題への性能向上は今度とも衰えることはないかもしれない。その理由は非常にシンプルで、この現世には解決しなければならない問題を抱えた人たち（国家や地域自治体なども）とその供給のバランスが保てているからである。そして、それらの勢いはいずれ衰える、衰えないの如何にかかわらず素速く処理することが強く求めらる。例えば、数ある諸問題から人口問題にスポットを当て論ずるならば、情報産業をはじめとする各産業の人材不足という問題があるが、それ以上に人口の爆発的増加が各地で多くの問題を訴えていることの方が大きい。まず、そのために医療、環境、食糧問題、そして教育問題を積極的に解決していかねばならない。つまり、このことは産業界の人材不足も無視できないどころか、そこへ帰属する問題でもある。なぜならば、現時点では医療・福祉、情報系を中心に雇用の拡大が広がっていることが事実であり、その人口問題への帰属問題として、情報産業界に多くの教育者を投入する必要性があることが理解できるだろう。例えば、マイクロソフト社によるMOT（マイクロソフト・オフィシャル・トレーナー）などの新しい教育資格者の適用がそれを証明していることだろう。これらのことより、情報化社会への研究が急がれている。私はこんな時代背景の中でいつしかプログラムの高速化への夢がふくらんでいった。今はそれを実現するための第一歩である。

さて、プログラムを高速化するといってもどうするのか。この研究目的の底をなすものに教育支援があるのだとしたら、いきなり難解な問題に取りくみ、さらにそれ以上に難解に説明するわけにはいかない。しかし、多くの高速化の問題はこういったものが多いようである。そこで私が研究の題材としたのは並列処理である。複雑なアルゴリズムを小手先のテクニックでそこそこの速度向上を得るよりも、単純なアルゴリズムを大胆に2倍以上の速度向上を得てしまおうというのである。ではどうしたら、そんな夢みたいなことが可能なのか。その性能を大きく向上させる手っ取り早い方法が並列処理である。

しかし、その一方で並列プログラミングは難解であるといわれる。確に、並列プログラムは高度で専門的でもある。しかし、それだけが並列計算のすべてではない。もっと手を伸ばせば、手が届くところにある。並列計算機は先端的な分野で活用され、大きな成果を上げている。しかし、科学・工学の多くの研究者、技術者が日常的に用いる計算機になっていないのが事実としてある。その原因として、並列計算機の方式は多様性に富んでいて、そのプログラムは機種依存性が高く、一方で、完全自動並列化コンパイラは現状では難しいこと、ワークステーションやパーソナルコンピュータはありふれたもので、手軽に使用できる一方、本格的な並列計算機は高価なもので、現状では利用できる環境にある人の数が限られていること、また、並列計算機に関する教育や教科書は専門教育用であるのが現状であることがあげられる。しかし、プログラミングインタフェースについて言えば、ＭＰＩ（Message Passing Interface）やＰＶＭ（Parallel Virtual Machine）の出現および高水準言語の標準化の進展により、事態が改善されつつある。当研究で扱っている題材に含まれるOpenMPはそうした高水準言語が標準化したうちの一つである。また、ワークステー

ションやパーソナルコンピュータをＬＡＮで結合したクラスタも高性能計算環境として現実的なものとなりつつあり、多くの人が並列計算を行えるようになってきているのである[1]。

この研究は OpenMP という共有メモリマルチプロセッサの並列プログラミングモデルを用いて並列プログラミングをする研究である。現在は上述のＭＰＩ、ＰＶＭのような分散メモリ並列プログラムが主流となり、共有メモリプログラムにおける並列化が積極的に開発されなかった。それは共有メモリ計算機ごとに独自に言語を拡張して並列化を行わなければならなかったためである。しかし、共有メモリマルチプロセッサの普及にともない、OpenMP という Fortran、C の拡張（文法の拡張ではない）で並列実行を行えるソフトウェアが提案され注目を受けている。OpenMP による並列化はメッセージパッシングモデルや分散システムの拡張ソフトウェアのように、全体を分割したり、プログラムを全面的に書き換えることはない。ユーザーの指示によって、逐次から部分、部分で明示的に並列化が行える点に OpenMP の期待がよせられている。つまり、OpenMP は比較的易しくプログラミングでき、依存性がないプログラムインターフェースである。この研究では逐次 C 言語プログラムを対象として OpenMP による並列化作業をおこなった。

主に、本研究では OpenMP による符号化問題と文字列照合問題の並列化を行い、その評価を行った。２章に並列処理についての背景や、並列処理の種類について説明した。３章に OpenMP の解説を示した。４章では今回扱った問題（ランレングス圧縮、KMP 法、BM 法）を簡単に説明し、実際に OpenMP にて並列処理させて、測定結果をとった。５章ではその考察を行った。これらの本研究により一人でも多くの方が並列プログラミングに挑戦し、プログラミング高速化への楽しみを享受できれば幸いである。

2　並列処理

LSI や超 LSI 技術の進歩により、CPU やメモリなどハードウェア価格が安くなり、計算機システムが大規模かつ高速になってきた。しかし、計算機の適用される分野も拡大され、従来は不可能視されていた問題にも適用されるようになると、より高速な処理が必要となった。ところが、ハードウェアの高速化もそろそろ限界に近づいてきたため、どうしても並列処理による高速化が必要不可欠になっている。

並列処理と言っても様々な形態があるが、逐次処理も含めた処理方法を分類すると、次の 3 つの形態に大別される。

1. SISD (Single Instruciton Single Date)
2. SIMD (Single Instruciton Multiple Data)
3. MIMD (Multiple Instruciton Multiple Data)

SISD 方式は 1 つのデータに 1 つの命令を作用させるもので、従来の逐次処理のことである。SIMD 方式は 1 つの命令を同時に複数のデータに対して作用させるものであり、一番簡単な並列処理方式である。これを実現したハードウェアとしては、アレイ・プロセッサなどがある。ベクトル計算や微分方程式などの数値計算や、画像処理の分野で利用されている。

MIMD 方式は複数の命令を複数のデータに同時に作用させるものであり、一般的に並列処理といった場合にはこれを指す。この方式は実現が難しく、実現方法として次の 2 つに分類できる。第 1 の方法は 1 つの処理装置内で並列処理を行うものである。第 2 の方法は複数の処理装置を利用し、それらがそれぞれ別々の処理を行うものである。

2.1　並列処理の必要性

MIMD 方式の実現のための研究・開発されるようになった理由には、ハードウェア上の問題とソフトウェア上の問題がある。

ハードウェア上の問題としては、前述したように計算機アーキテクチャの速度の目覚ましい向上を背景に、アーキテクチャが処理する問題は大規模化しているので、並列処理する必要が高まっている。

第 2 に、VLSI 技術の発達により、大量生産のできる汎用処理装置の低価格化が実現し、そのために、超大型計算機をつくるよりも、処理装置を複数台接続した並列処理システムの方が価格/性能比が良くなる傾向にあるので、必然的に並列化の必要性が高まってる [2]。

ソフトウェア上の問題としては、ソフトウェア危機ということがあげられる [2]。これは数年前から言われていることであるが、作られるソフトウェアが大規模なものになり、その作成・保守・修正が困難になっていることである。この問題を解決するために、プログラムのモジュール化、データの抽象化、情報隠蔽、オブジェクト指向プログラミングなどの概念が提案されている。これらの概念を実現するには、従来の逐次処理言語に比べ、並列処理言語の方が適している [2]。

モジュール化に関しては、逐次処理言語では処理の流れに重点を置いてモジュール分割せざるを得ないのに対し、並列処理言語では処理の流れとは無関係に、論理的機能を中心に分割できるため、プログラムの作成、理解が容易になる。データの抽象化、情報隠蔽についても、１つの情報を１つのモジュールが管理し、その情報へのアクセスの方法だけを外部に見せることにより、自然な形で言語の枠内に組み込むことができる。オブジェクト指向プログラミングについては、モジュールをオブジェクトと考えれば、その間のメッセージ交換によって全体の処理が記述できるので、並列処理言語内で実現することは容易である。

2.2 並列計算機システム

並列計算機アーキテクチャから見て、並列処理に影響を与える因子のうちメモリモデルが挙げられる。主なメモリモデルとしては、共有メモリ、分散メモリ、分散共有メモリがある。それぞれの主なシステムに下の２つの手段がある。

1つは共有メモリ型のSMP(Symmetric Multi-Processor)システムである。SMPシステムは複数のプロセッサ間で共用するメモリを用いて単一のOS下でシステムが動作しているものである。SMPシステムのプロセッサ間結合技術として従来は「バス接続」が一般的であった。この技術はバックパネル上に張ったデータバス、アドレスバスを複数のプロセッサが共有する方式で、現状では接続プロセッサ数が30台程度、また、転送周波数が40MHz～50MHz程度が限界となっている。それ以上のプロセッサ数の接続とデータ転送のスループットを実現するために、最近はバックパネル上にLSIを搭載し、データバスを「クロスバス接続」する技術が用いられたが、この方式でも，現状は64プロセッサ、80MHz～100MHz程度の転送周波数が限界となっている。いずれの方式を採用しても、繋ぐことができるプロセッサの数がそれ程拡大出来ず、また、データ転送のスループットも飛躍的な拡大が望めないという問題がある[1]。

もう1つのシステムとして、分散（共有）メモリ型のクラスタシステムである。これは、それぞれ独立したOSとメモリを持つサーバを連結する方式であり、いままでは10Base-Tや100Base-T等のLANでサーバ間を連結して大規模システムを構築してきた。現在、これらのLANの回線速度はギガビットクラスに到達しており、この回線速度ではスループット的にも、また、進化し続けるプロセッサの性能からみれば、ますます差が開いていく状況にあるといえる[1]。

つまり、高性能なプロセッサの能力を最大限に活用するには、接続部分のデータ転送速度の高速化、すなわち、インタコネクト技術が非常に重要となるといえる。

3　OpenMP

前章では、近年の大規模な問題に取り組むためには、ハードウェアとソフトウェアの両サイドからアプローチする必要があることを述べた。ハードウェアからのアプローチをSMPシステムとして捉えたとき、ソフトウェアからのアプローチはどうしたらよいのだろうか。この章では、OpenMPの是非とその内容について探ってみる。

3.1　OpenMPとは

OpenMPは共有メモリ計算機のSMPを対象にした、逐次プログラムの並列化に関する指示を行うための並列化指示インタフェースである。OpenMPは1997年のSC'97においてFortran版の仕様が公開され、さらに翌年のSC'98でC/C++版の仕様が公開された[6]。

OpenMPの指示は、データ並列を利用した並列実行を行うループの指定と並列実行を制御する指示から成っている。この他、ユーザが定義したプログラム部分の並列実行の指定も可能である。これらの指示は、FortranやCの文法の拡張ではなく、コメントあるいはプログマの一部として定義されているため、並列化指示を加えた後でも、逐次実行するようにコンパイル可能である[10]。コンパイラに対する並列化指示以外に、実行時ライブラリと環境変数に関して定義されている。

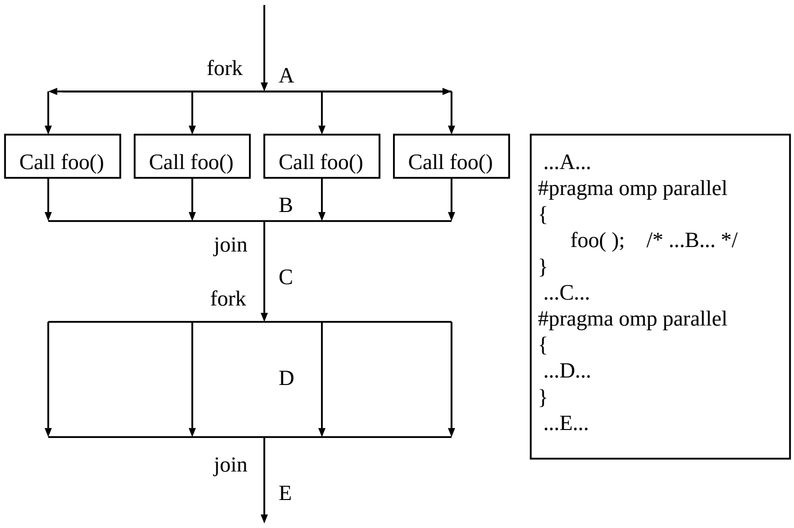
OpenMPと同様の並列化指示の仕様として、分散メモリマシンを対象にプログラムの並列化を行うために提案されたHPF(High Performance Fortran)がある。HPFはプログラム中のデータを分散メモリ上に配置する方法を指定し、並列ループの検出などは処理系にまかされている。一方、OpenMPは共有メモリを対象としていることもあり、データ配置に関する指定はなく、並列ループなどの並列実行の指定が中心となっている。HPFはFortranのみの仕様であり、逐次プログラムをそのまま使えるわけではなく、対象とする並列計算機のアーキテクチャや、プログラムの制御フローなどを考慮しなければならないのに対して、OpenMPはFortranに加えC/C++にも対応していて、さらに逐次プログラムをそのまま使い、段階的な並列かが可能である。

OpenMPの並列実行は図1が示すようにfork-join型の実行モデルを採用している。このため、プログラム実行中に並列実行部分の指定があった時点で、複数のスレッドを生成し、配列実行部分が終了した時点でマスタ以外のスレッドは消滅して元の1スレッドとなる。

OpenMPでは、プログラムの整合性や並列化可能性などは、全てユーザーが保証しなければならない。このため、OpenMPコンパイラはループの依存関係やデッドロック等のチェックを行う必要がなく、プログラム中に指示した通りにプログラムの並列化を行うことが可能となっている。OpenMPの指示は、コンパイラに対するヒントとして扱われていた並列化指示とは異なり、並列実行を記述するプログラミング言語であるとも言える[10]。

3.2　OpenMP API

OpenMP APIは言語の拡張であり、プログラマは並列構文から呼び出されるルーチン内でdirectiveを使うことができる。Cでの構文では、♯ pragma ompから始まるpragma行



```
 ...A...
#pragma omp parallel
{
     foo( );    /* ...B... */
}
 ...C...
#pragma omp parallel
{
 ...D...
}
 ...E...
```

図 1: fork-join 実行モデル

を用いて記述する。なお、pragma 行は構文要素として扱われるので注意が必要である。

♯ pragma omp directive 名 指示節,...

OpenMP の並列構文には、以下の要素がある。

- parallel region 構文

  複数のスレッド（term）によって、並列に実行される parallel region を定義する。

- Work sharing 構文

  for 構文（データ並列）、sections 構文（タスク並列）、single 構文（単スレッド実行）があり、parallel region 内において、ループなど、現在のスレッド（team）で分割して実行する部分を指定。また、Work sharing 構文は、nowait 指示節を指定しない限り、最後でバリア同期が行われる。

- 同期構文

  master 構文、critical 構文、barrier 指示文、atomic 構文、flush 指示文、ordered 構文がり、同期を取るための構文である。

例えばループレベルのプログラムは次のように記述する。

```
#pragma omp parallel for

         for(i = 1; i < n; i++)

                   b[i] = a[i] + a[i-1]
```

3.3 RWCP Omni OpenMP コンパイラ

Omni Compiler Software は、プログラムの解析や変換を行うための、パーサーなどのプログラム、Java のライブラリを含むプログラム群である [6]。Omni OpenMP コンパイラは、これらのソフトウエアの一部として開発したものである。このコンパイラは、OpenMP の C および Fortran77 プログラムをプログラムを SMP 向けのマルチスレッドの C プログラムに変換、バックエンドとして既存の コンパイラによりコンパイルし、Omni OpenMP 実行時ルーチンとリンクして並列プログラムを生成するコンパイラである [6]。

新情報処理開発機構つくば研究センター（RWC）では OpenMP を並列化インタフェースとした並列処理環境として、SMP マシンをターゲットにした OpenMP コンパイラとその実行時ライブラリを発している [10]。OpenMP コンパイラは、移植性を高くすること、また将来並列化に関する機能拡張を容易に行うことができることを考慮して開発された [10]。さらにサポート内容として、C 言語 parser、JAVA によるコード変換ツールキット、Omni tlogview（Omni OpenMP コンパイラの directive イベントのトレースを表示するツール）などを含んでいて、現在は RWC のホームページにて公開されている。

4 OpenMP による並列プログラミング

本実験では符号化と文字列照合の問題について扱った。符号化問題ではランレングス圧縮をを、文字列照合問題では有名な KMP 法と BM 法を扱った。

4.1 動作環境

現在のプラットフォームとして、2 種類のマルチプロセッサマシンを使用した。一つは SUN Enterprize450 Solaris2.7、CPU は UltraSPARC-II (400MHz) が 4 つ、メモリ容量は 512MB で HD は SCSI 接続である。もう一つは DELL Precision の SMP(Pentium3 450MHz が 2 つ) 上の Vine Linux1.1、CPU は Pentium3 450MHz が 2 つ、メモリ容量は 512MB、HD は IDE 接続である。並列実行のスレッド生成は、実行時ライブラリのオプションで Solaris スレッドおよび POSIX スレッドの両方に対応可能としている。

4.2 ランレングス圧縮

ランレングス圧縮法はランレングス符号化 (run-length encoding)[5] とも呼ぶ、符号化法のひとつである。ファイルの中で同じ文字が繰り返される長い連 (レン、run) は最も単純な冗長性である。例えば次の場合を考えてみる。

AAACCCCCCNNIIIIIIOOOLLLLLLLAAAAD

この文字列は各連を 1 文字と反復回数 (連の長さ) の組で置き換えることによってもっとコンパクトに符号化できる。つまりこの文字列は 3 つの A の後に 6 つの C が続き、その後に 2 つの N が、6 つの I が、……と続いたものと言い表せる。もし文字列が英字しか含まないとわかっているならば、これを反復回数の英字で区切られた数字列として符号化することができる。例えば上の文字列は次のように符号化される。

A3C6N2I6O3L7A4D

長い連が含まれている場合は劇的な圧縮が可能となる。ランダムなテキストファイルや 2 値ファイルには適してないが、通常のテキストファイルでは 20 %から 30 %、2 値ファイルでは 50 %から 90 %の圧縮が可能である [5]。

コンピュータのメモリの価格が劇的に低下したのでファイル圧縮技術の重要性はかつてほどではなくなったとする意見んがある。しかし、メモリの使用量が飛躍的に増加すれば節約できる量も増大するから、ファイル圧縮技術はこれまで以上に重要になると考えられる。

4.3 KMP アルゴリズム

D.E.Knuth (クヌース)、J.H.Morris (モリス)、V.R.Pratt (プラット) による文字列照合アルゴリズム [4] を KMP 法という。n 文字のテキストから m 文字の照合文字列を探すとき、単純照合 (力まかせ探索) 法とでは最悪の場合 O(nm) の手間がかかるが、この方法では O(n+m) の手間ですむ。

$$\begin{array}{l}A_0 B_1 C_2 A_3 B_4 C_5 A_6 B_7 A_8 \\ A_0 B_1 C_2 A_3 B_4 A_5 \\ \qquad\qquad\quad\uparrow\end{array}$$

まず、単純照合法と同様に左端を合わせて、左から 1 文字ずつ比較する。上のでは 5 番目で照合失敗になる。単純照合法ならば、照合文字列を右に 1 文字ずらして照合を再開するが、KMP 法は照合文字列の A_3B_4 が A_0B_1 とが照合することを利用して、テキスト A_3 と照合文字列の A_0の位置に合わせてから比較を続けることができる。上のようにテキスト

$$\begin{array}{l}A_0 B_1 C_2 A_3 B_4 C_5 A_6 B_7 A_8 \\ \qquad\quad A_0 B_1 C_2 A_3 B_4 A_5 \\ \qquad\qquad\quad\uparrow\end{array}$$

の現在位置と照合文字列の C_2とを揃え、そこから調べ直せばよい。これを効率よく行うために $next$ 関数を用いる。例えば、照合の失敗した照合文字列の位置が 5 番目であるとき、照合文字列の 2 番目と比較を始めるという情報を $next[5] = 2$ という風に表すことができる。この $next[j]$ 関数は照合文字列の先頭の j文字について、その頭と尾が何文字一致するかを調べたらよい。これによってスムーズに照合が行える。

4.4 BM アルゴリズム

R.S.Boyer (ボイヤー)、J.S.Moore (ムーア) による文字列照合アルゴリズムのことを BM 法という [4]。下の例でテキスト' PQRSTUABBA... ' の中から'AABBA' を探し出すための基本的なアルゴリズムを示す。

あらかじめすべての文字 A-Z について、それが照合文字列の右端以外の場所に現れるか、現れるなら右から何文字目に初めて現れるかを調べ、表にしておく。この場合、A が右から 4 文字目、B が左から 2 文字目に現れる。次に左端を揃える。

PQRSTUABBA......
AABBA
    ↑

照合文字列の最後の文字 A とその上のテキストの T とを比べる。これらは一致しない。最初に作った表を引けば T が照合文字列中に現れないことが分かるので、照合文字列を一挙に 5 文字ずらす。

PQRSTUABBA......
AABBA
↑

ここでまた照合文字列の左端から調べていく。今度は一致するので、左に向かって順に１文字ずつ調べていくと、照合文字列の左端で一致しなくなる。この場合、照合文字列の右端と比べたテキストのAが照合文字列の右から４文字目にもあることを表で調べ、この両者が重なるように照合文字列を３文字分ずらす。以下同様に続ける。

PQRSTUABBA......
AABBA

このように、たった１文字の比較で何文字も読み飛ばすことができるのがBM法の特徴である。

4.5　並列プログラミング

前述した３つをC言語によって作成し、それらの逐次プログラムにそれぞれOpenMPの指示文を加えて並列実行を行った。

4.5.1　ランレングス圧縮の並列化

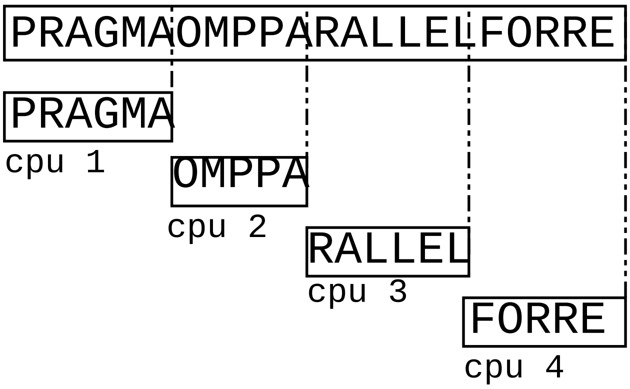


図2: ランレングス圧縮のスレッド分割

これは、文字列照合問題と同様に文字列を扱う場合を想定して、符号化されるべき文字列をスレッド数に分割した。これは符号化の問題なので文字列をどこで分割しても問題はない。つまり、この問題では文字列をスレッド数分に等分割した。

4.5.2 並列効果

ランレングス圧縮の実験に用いるデータ数をそれぞれ、10 万文字、1,00 万文字、10,000 万文字として、SunOS で最大スレッド数を 4 つずつ、Linux で最大スレッド数を 2 つずつで計測を行った。以下、表 1 に並列実行時間（秒）を示す。計測をした個所は pragma 行からはじまる、parallel 実行間でバリア同期をとるまでを計った。逐次プログラムで parallel 実行間は全体の実行時間の約 9 %を占める。

表 1 OpenMP によるランレングス圧縮の並列実行時間（秒）

	SunOS(1)	SunOS(2)	SunOS(3)	Sun0S(4)	Linux(1)	Linux(2)
10 万文字	0.032	0.017	0.012	0.040	0.021	0.040
1,00 万文字	0.107	0.058	0.038	0.29	0.133	0.068
10,000 万文字	10.936	5.543	3.691	2.792	13.206	6.752

以下に、表 1 の実験データに基づいて、SunOS と Linux のそれぞれのスレッド数が 1 のときにおける速度向上を表 2 で表す。() 内はスレッド数を表す。

表 2 OpenMP によるランレングス圧縮の速度向上

	SunOS(1)	SunOS(2)	SunOS(3)	Sun0S(4)	Linux(1)	Linux(2)
10 万文字	1.00	1.88	2.67	3.56	1.00	0.50
1,00 万文字	1.00	1.84	2.82	3.69	1.00	0.20
10,000 万文字	1.00	1.97	2.92	3.92	1.00	1.92

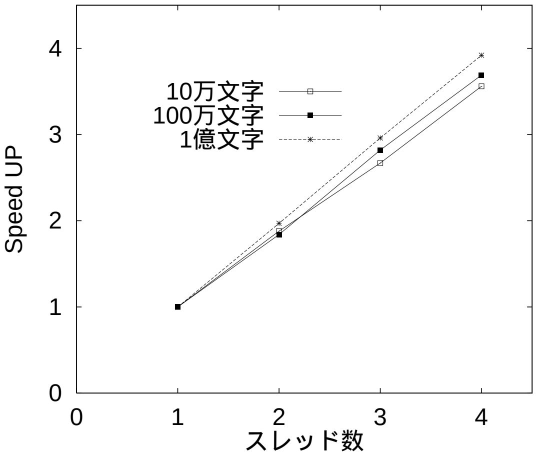


図 3: OpenMP によるランレングス圧縮の並列実行時間 (秒)

4.5.3 KMP 法と BM 法の並列化

KMP 法と BM 法はアルゴリズムは違うため、ソースコードも違うが、並列手法は同じである。これらの文字列照合問題は莫大なテキスト文字列を使用する場合を考え、テキスト文字列をスレッド数分にブロック分割する手法をとった。この問題は単純に等分割を行えば、正確な照合が行えないことを考慮して、つまり、隣接するブロックの境界に照合文字列が存在する場合もあるため、分割する際に照合文字列分のデータを重複して割りふる方法で行った。

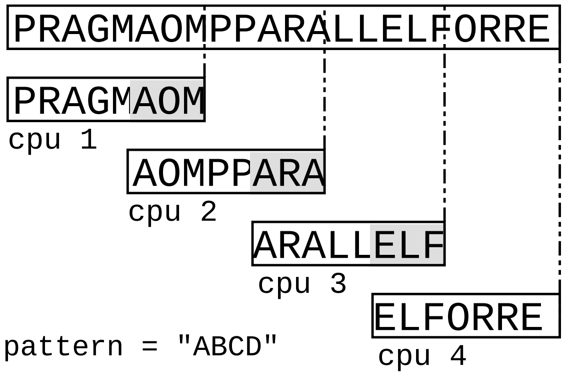


図 4: テキスト分割

4.5.4 並列効果

各文字列照合問題(KMP法BM法)の実験に用いるデータ数をそれぞれ、1万文字、10万文字、100万文字として、照合文字列を10文字として、SunOSで最大スレッド数を4つずつ、Linuxで最大スレッド数を2つずつで計測を行った。以下、表3に並列実行時間(秒)を示す。計測をした個所はpragma行からはじまる、parallel実行間でバリア同期をとるまでを計った。逐次プログラムでparallel実行間は全体の実行時間の約１１％~１８％を占める。また、照合文字列を最大50で変化させて実験をおこなったが、データの格差は見られなかった。

表3 OpenMPによる文字列照合(KMP法とBM法)の並列実行時間(秒)

	SunOS(1)	SunOS(2)	SunOS(3)	Sun0S(4)	Linux(1)	Linux(2)
1万文字(KMP)	0.0021	0.0013	0.0012	0.0011	0.0009	0.0006
10万文字(KMP)	0.0209	0.0113	0.0078	0.0063	0.0091	0.0049
100万文字(KMP)	0.2188	0.1176	0.0858	0.0736	0.0961	0.0497
1万文字(BM)	0.0017	0.0014	0.0012	0.0011	0.0007	0.0007
10万文字(BM)	0.0133	0.0076	0.0057	0.0048	0.0048	0.0029
100万文字(BM)	0.1255	0.0659	0.0467	0.0392	0.0519	0.0278

以下に、表1の実験データに基づいて、SunOSとLinuxのそれぞれのスレッド数が1のときにおける速度向上を表2で表す。()内はスレッド数を表す。

表4 OpenMPによる文字列照合問題(KMP法とBM法)の速度向上

	SunOS(1)	SunOS(2)	SunOS(3)	Sun0S(4)	Linux(1)	Linux(2)
1万文字(KMP)	1.00	1.62	1.75	1.91	1.00	1.50
10万文字(KMP)	1.00	1.85	2.68	3.32	1.00	1.83
100万文字(KMP)	1.00	1.86	2.55	2.97	1.00	1.93
1万文字(BM)	1.00	1.21	1.42	1.54	1.00	1.01
10万文字(BM)	1.00	1.75	2.33	2.77	1.00	1.66
100万文字(BM)	1.00	1.90	2.69	3.20	1.00	1.87

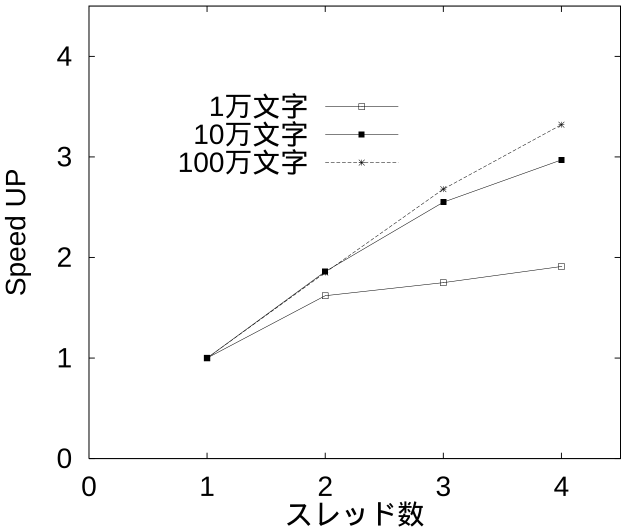


図 5: OpenMP による KMP 法の速度向上

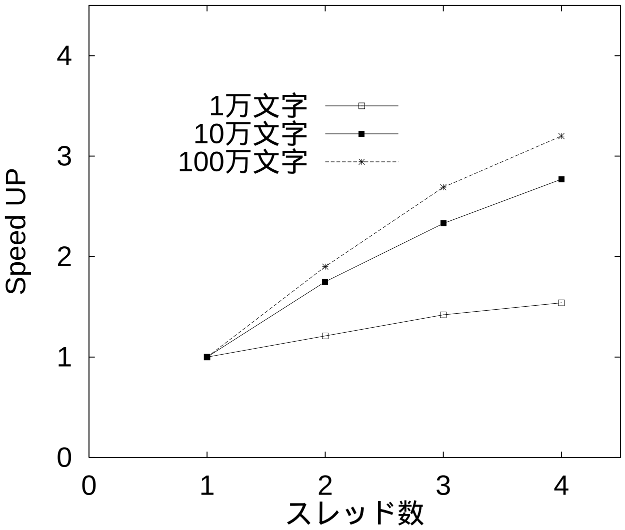


図 6: OpenMP による BM 法の速度向上

4.6　トレースで見る実験結果

さらに、参考までに Linux の 2 スレッドで並列化たランレングス圧縮プログラム、KMP プログラムの実行時間の様子をトレースで示す。

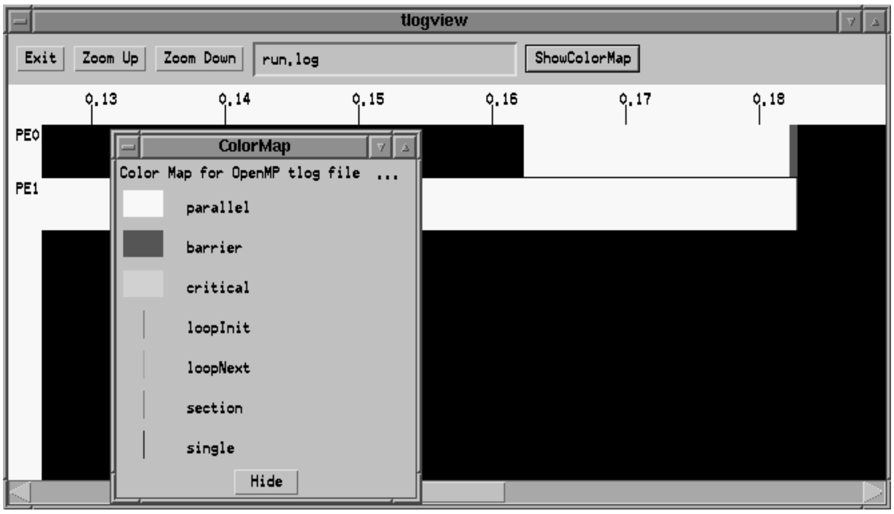


図 7: tlogview で見るランレングス圧縮の並列実行

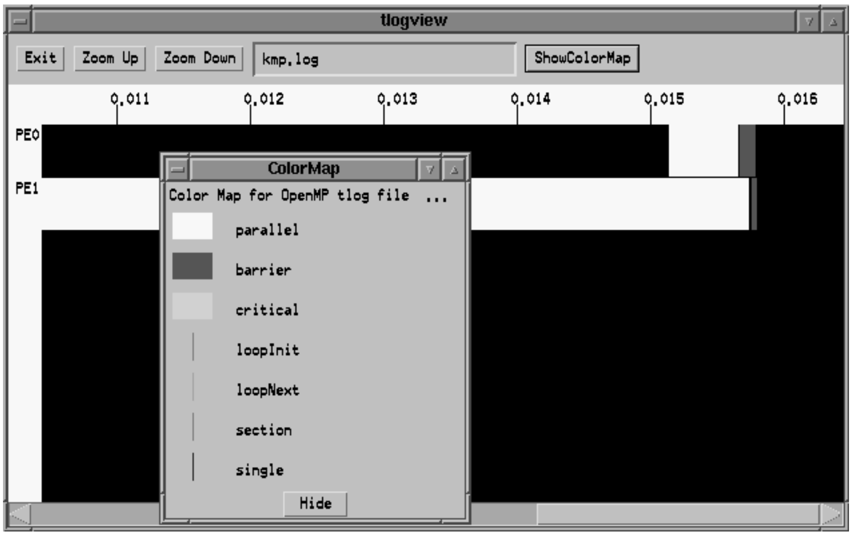


図 8: tlogview で見る KMP 法の並列実行

図７はランレングス圧縮を２つのスレッドで並列実行した場合で、プログラムが時間軸にそって進んでる様子が図の白い部分で表されている。0.016 秒を過ぎたところで PE0 が並列化しているのがわかる。0.018 秒を過ぎたところの同期をとっている様子が灰色部分でしめされている。図８は KMP を２つのスレッドで同様に並列実行させた様をしめしている。実行条件はランレングス圧縮は１０万文字の場合、KMP 法は１万文字の場合である。

5　性能と考察

実験より、基本的にはスレーブ数を多くすれば、それなりの速度向上は得られるということが確認できた。さらに、扱う文字列を多くすれば、より並列効果がでた。

符号化問題と文字列照合問題を比較してみると、両問題は文字列を扱ういう点では共通しているが、分割方法が異なるため、速度向上に違いがある。これは図３、５、６より明らかで、ランレングス圧縮問題は文字列を変えても並列効果に差は少ないが、文字列照合問題は、文字列数によって並列効果が異なり、文字列が1万や10万の時は理想的な並列効果はでていない。これは文字列照合問題のほうは単純なブロック分割ではないため、そのための作業が遅れをとる原因だと考えられる。

次に、KMP法とBM法の違いを比べてみる。分割法は同じであるが、実行速度はBM法のほうが早い。これは、逐次アルゴリズムより総計算量の少ない並列アルゴリズムはないからであって、並列効果に大きな差はないところを考えても、これは並列アルゴリズムの問題ではない。

さらに、２種類のSMPについて。この２種類の実験データを慎重にとったが、他の計算機とネットワークでつながっている以上、プロセッサになんらかの負荷がかかった状態のまま計測したため、多少のばらつきがある。同じプログラムをおなじ条件で計測したが、Linuxのデータのほうが早い結果となった。これは並列プログラムは並列計算機のアーキテクチャに依存するためであると考えられる。

そして、これはOpenMPでparallel動作させたアルゴリズム部分の評価であって、実用的なプログラムとして見た場合は、疑問的が残る。今回評価した部分は全体プログラムの１０から２０％しか占めないため、全体の速度向上はあまりえられていない。全体のほとんどを占めているのはデータの読み書き部分がほどんどで、今回はこの並列化を対象外とした。また、読み書きが逐次で占めているため、文字列数に比例して実行時間が多くなってしまった。これは図７、８でも確認できよう。とくに符号化プログラムでは９０％程度が読み書きを占めているため、本当の意味で速度向上が得られたとはいえない。

6 おわりに

以上、本稿ではOpenMPの概要とその性能評価を文字列問題の観点からアプローチしてきた。プログラムの難しさを考えると、PVMのようなメッセージパッシングモデルより、OpenMPによる並列化作業は容易であった。それは共有メモリモデルであるということ、さらに、FortranまたはCを拡張することでアーキテクチャに依存しない移植性を保証し、および逐次プログラムからインクリメンタルに並列化していくことが可能であるためである。しかし、プログラムによっては並列化ができない問題もあり、たとえ並列化されても速度向上が得られない場合もあり、性能のチューニングがむずかしい。

さらに、今回扱ったプログラムはすでに我々が研究した並列アルゴリズムを用いて、それをOpenMPで移植を行ったわけだが、我々の研究室は多くの並列プログラミングを作成し、それらから共通のカテゴリーで分類しプログラミングを支援する事例ベース並列システムのために、さらなる新しい事例への着手が求められている。今後、OpenMPについても研究を深めていく必要がある。本稿ではスレッド数が最大数4つまでの実験、評価を行ったが、これからの課題としてスレッド数をもっと増やすことなど検討すべきだろう。そこで我々の研究室では、それを可能にするためにPCクラスタで実行させるOpenMP並列プログラミングの検討をしている。これによって、より多くのスレッド生成を可能とし、それに合わせて速度向上が望める。実際、OpenMPはSMPクラスタシステムへの移行など、OpenMPの分散共有システムの研究、実用化がなされている。OpenMPは、このようにユーザーごとに、ターゲットマシンごとに、より最適な並列プログラムを可能とし、並列化におけるソフトウェアの位置付けが高まってこようと思われる。

謝辞

本研究の機会を与えてくださり、数々の助言を頂きました山崎勝弘教授、西村俊和助教授、ならびに、Sunのサーバーを利用させていただきました京都産業大学の新実治男教授のお二人に心より感謝致します。また、本研究にあたり、様々なご意見や、質疑に答えていただくなどした本研究室の皆様にも心より感謝します。

参考文献

[1] 湯浅太一、安村通晃、中田登志之 編 :『はじめての並列プログラミング』、共立出版、1999.

[2] 宮村勲 著 :『並列処理言語』、マグロウヒル、1986.

[3] 村岡洋一 著 :『並列処理』、昭晃堂、1986.

[4] 奥村晴彦 著 :『C 言語による最新アルゴリズム事典』、技術評論社、1998.

[5] Robert Sedgewick 著、野下浩平、星守、佐藤創、田口東 共訳 :『アルゴリズム C 第 2 章 探索 ·文字列 ·計算幾何』、近代科学社、1996.

[6] OpenMP のホームページ :http://www.openmp.org/

[7] 佐藤三久 :『JSPP'99 OpenMP チュートリアル資料』、RWCP、2000. http://pdplab.trc.rwcp.or.jp/pdperf/Omni/openmp-jspp99/index.htm

[8] RWCP : 『OpenMP C language API Specification』 http://pdplab.trc.rwcp.or.jp/pdperf/Omni/htdocs.ja/OMP-C-API.html

[9] Leonardo Dagum and Ramesh Menon :『OpenMP:A Proposed Industry Standard API for Shared Memory Programming October 1997』、1998.

[10] 草野和寛、佐藤三久、佐藤茂久 :『OpenMP コンパイラの試作と評価』, 情報処理学会ハイパフォーマンスコンピューティング研究会,1999.

[11] 内田大介 :『OpenMP による並列プログラミング [I]』、立命館大学理工学部情報学科卒業論文、2000.