文字列照合のOpenMPによる並列化

氏名：南 扶友子

学籍番号：2260060107-9

指導教員：山崎 勝弘

提出日：平成22 年2 月23 日

立命館大学　理工学部　電子情報デザイン学科

内容梗概

分子生物学やコンピュータを利用した音楽解析の分野などで，繰り返しのある文字列に関する研究が進められているが，これらの分野では厳密な文字列マッチングより，誤りを考慮した最適な繰り返しに関する文字列マッチングが有用である．この最適な繰り返しに関するアルゴリズムは計算量が大きいため，並列処理などを用いて高速化することが求められる．

本研究では，文字列照合に関するパターンの探索問題を取り上げ，代表的な文字列照合のアルゴリズムを並列化した．並列化したアルゴリズムは，OpenMP を用いて SMP クラスタである Diplo(4 プロセッサ) と Score 型クラスタである Nycto(8 プロセッサ) 上でそれぞれ実装した．

計算速度の向上では，Diplo 上では，3 つのアルゴリズムすべてがプロセッサ数に従って，速度向上する結果になった。それぞれ 16 プロセッサでは 1 プロセッサで実行させたときに比べて，単純照合では 11.5 倍，KMP 法では 6.5 倍，BM 法では 1.6 倍の速度向上が得られることがわかった。Nycto 上では，単純照合はプロセッサの数が少ないと速度が遅くなってしまうが，8 プロセッサ以上に挙げると速度向上が望める．また，KMP 法においては，プロセッサ数以上の速度向上が望めることが実験によって得られた．一番効率の良いとされている，BM 法においては，今回の実験では 8 プロセッサで 3.5 倍の速度向上であり，KMP ほどではなかったが，どのアルゴリズムも速度向上が望めることが実験によって結果として得られた．

目 次

図 目 次

表 目 次

第1章　はじめに

文字列に関する研究は，分子生物学・データ圧縮・コンピュータを利用した音楽解析など，様々な分野で進められている．さらに，データベースやワープロソフトにおける検索，ネットワークにおけるコンテンツスイッチや侵入検知システム，そして，DNA の解析など，多様な分野で応用されている．文字列照合は応用範囲の広さから，古くからさまざまな高速か手法が提案されてきている．そのなかでも，BM(Boyer-Moore) 法が一般的に高速とされる．しかし，今日の情報化社会の急速な進展により，データベースに蓄積される情報やネットワークを介して転送される情報の増大，リアルタイム処理に対する要求の高まりに伴って，文字列照合の更なる高速化は求められている．

このような計算量が大きい問題に対して，並列処理は特に有効である．近年の PC クラスタなどの普及により，並列処理はより身近なものとなっている．PC クラスタは，既存の PC を組み合わせてネットワークを構成するだけで作ることができるため，一般の並列計算機に比べて導入が容易である．また，PC クラスタは従来，分散メモリ型のアーキテクチャであるが，SCore と呼ばれるクラスタシステムソフトを導入することにより，物理上では分散しているメモリを共有メモリのように扱うことができる．

最近では，各計算ノード内において，複数のプロセッサで 1 つの共有メモリを扱う，SMP(対象型マルチプロセッサ) を持つ PC を用いて構成する，SMP クラスタが注目を集めている，SMP は，近年普及しているマルチコアやメニーコアなどのプロセッサを搭載した PC に取り入れられているため，SMP クラスタの導入は一般の PC クラスタと同様に，容易である．この SMP クラスタを利用することにより，計算ノード内は共有メモリ型の並列処理，計算ノード間は分散メモリ型の並列処理をそれぞれ行う，ハイブリッド型の並列処理も可能である．

本研究では，高速化がさらに求められる文字列照合とその高速化に応えられるであろう並列化に着目した．文字列照合における 3 つのアルゴリズム (単純照合，KMP 法，BM 法) に着目し，それぞれ並列プログラムを OpenMP を用いて

作成し、SMP クラスタ上に実装して並列化の効果を評価した.

本論文の構成は次の通りである．第 2 章では文字列照合のアルゴリズム，単純照合，KMP 法，BM 法の 3 つのアルゴリズムを述べる．第 3 章では PC クラスタとその分類，PC クラスタを用いた並列処理に必要な並列プログラミング環境について述べた後，本研究に用いる PC クラスタ，今回考えた文字列照合の並列化手法について述べる．第 4 章では，並列化させた 3 つのアルゴリズムの並列化を行い，並列化したアルゴリズムの PC クラスタ上での実験結果と考察について述べる.

第2章　文字列照合のアルゴリズム

2.1　単純照合

単純照合は照合される文字列 (以下「テキスト」と呼ぶ) をはじめから順に照合文字列 (以下「パターン」と呼ぶ) と比較し，照合が失敗したら，1 文字ずつずらしながら処理を繰り返していくものである．例を図 2.1 に示す.

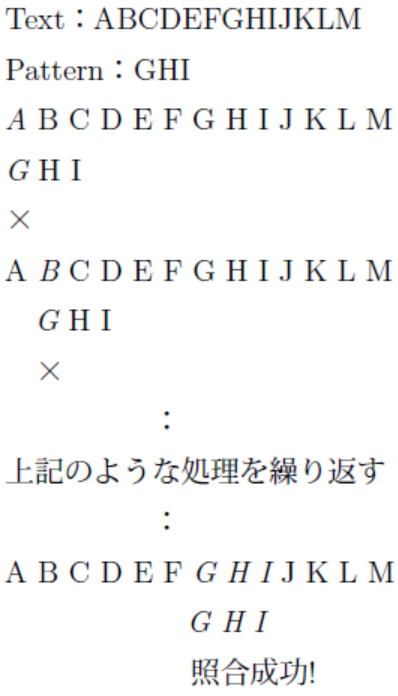


図 2.1: 単純照合の例

文字列比較の場合，処理が終わるまでに文字コードを比較する回数がアルゴリズムの性能を左右する．テキストとパターンのそれぞれの文字数が n,m で，かつ n が m に対して十分大きいとすると，上記のアルゴリズムの場合最悪 m × n 回近く文字を比較する必要がある．例えば，単一の文字が n 個並んだ文字

列から，同じ文字の並びの末尾に 1 文字だけ違う文字コードを付加した m 個の文字列を検索した場合，m-1 文字まで検索して最後の m 文字目で不一致を検出する処理を n 回近く繰り返すことになる．しかし，現実にはこのような「最悪の」場面というのはまずないと言える．仮に各文字コードがテキスト中に現れる確率が全て等しいとした場合，テキストと パターンの 1 文字目が一致する確率は「文字コードの種類の逆数」となり，さらに 2 文字目，3 文字目と連続して一致する確率はその 2 乗，3 乗とさらに小さく なっていく．実際には各文字コードの使用頻度がバラバラであるため正確さに欠けるとはいえ，現実の文字コードの種類は十分多いから，2 つの文字が連続して一致する確率より，一致しない確率の方がずっと高いということはまず間違いない．もし，常にパターンの先頭で不一致が見つかったとすれば，文字コードの比較回数は最大でも約 n 回となるので，パターンの先頭で不一致を検出する確率が非常 に高ければ実行速度はだいたい n に比例すると考えることができる．照合部分のプログラムを図 2.2 に示す．一般に，実行時間がデータの量に比例するアルゴリズムは「高速」，少なくとも「実用 的」な部類に属するので，上に示した「単純な」文字列比較は実用上十分「高速」であり，よって文字列照合に関しては複雑なアルゴリズムが入り込む余地があまりないことを意味する．そのため歴史的には，より高速な文字列照合のアルゴリズムが登場するのがかなり遅かったようである (クイックソートの論文が発表されたのは 1962 年，それに対して文字列照合アルゴリズムは 1977 年に発表されている)．文字列照合アルゴリズムを考案，改良する場合は，シンプルさを保たないと単純な方法には勝てないということを覚えておく必要がある．

2.2 KMP 法 (Knuth-Morris-Pratt algorithm)

単純な文字列照合アルゴリズムでは不一致が検出されたらパターンを 1 文字ずらして再び照合を繰り返すものであったが，より洗練されたアルゴリズムでは，パターンの移動量をなるべく大きくすることで効率を稼ぐようにしている．そのようなアルゴリズムの一つに Knuth と Pratt およびこの二人の他に Morris が同時期に考案した Knuth-Morris-Pratt(KMP) 法がある．KMP 法では，パターンとテキストの間で部分的に一致した場合に，その情報を元に大きくパターンを移動することによって効率化を図っている．そのために，不一致を起こした

```
1:  for(i=start; i<=end; i++)       /*テキストの位置を動かす
2:          for(j=0; j<Plen; j++)   /*パターンの位置を動かす
3:          {
4:           if (temp[i+j] != pattern[j])  /*テキストとパタ
5:                       break;                ーンが一致しな
                                              かったので 1:に戻る
6:           else if (j == Plen-1)          /*パターンが最後
                                              まで到達した

7:             //pattern の最後まで一致した
8:            //発見
9:            printf("fount at %d %c\n\n", i,temp[i]);
10:                num++;                /*一致をカウント
                                            num:初期値 0
        }
11:         if(num == 0)                /*num が 0 のままだったら
                                            一致なし
12:             printf("(null)\n");
```

図 2.2: 単純照合の照合部分プログラム

個所それぞれについて何文字までずらせばいいのかを事前に調べてテーブルにしておく．KMP 法の例を図 2.3，照合部分のプログラムを図 2.4 に示す．図 2.3 ではパターンを「KYOKAKYOKU」として照合している．例えば 4 文字目で不一致を検出したら，無条件に 3 文字分パターンを移動すればいいことになる．さらに，パターンを移動した後に文字の比較を開始する個所は，一文字目で不一致を起こした場合を除き，前に不一致を起こしたところから行えばいいこともわかる．というのも，パターンの移動量を調べた時点で，不一致を起こした個所より前の部分については一致していることがわかっているからである．

9 文字目では，前の 4 文字分はすでに一致していることがわかっているため，前に不一致を起こした個所，すなわちパターンの 5 文字目から比較処理を始めればいいことになる．このことからわかるように，事前に調べるパラメータとしては，パターンの移動量よりも再比較を始める個所 (上の例では 5 文字目) の方が都合がいいことになる．

KMP テーブルとは文字列照合を効率よく照合するために，パターンのどの文字で失敗したかをあらじめ入れておく配列である．

2.3 BM法(Boyer-Moore String Search algorithm)

Boyer と Moore，また両者とは別に Gosper が考案した Boyer-Moore(BM) 法の最大の特徴は，パターンを末尾側から逆方向に比較するということである．テキストとパターンの先頭をそろえた後，今までのアルゴリズムではパターン先頭とテキスト先頭を比較するので，BM 法ではパターン末尾 (先頭から m 文字目) の文字と，テキストの m 文字目の文字を比較する．もし一致していたら注目文字を 1 つ前にずらし，末尾側から逆方向に比較していく．もし不一致が検出されたら，不一致を引き起こしたテキスト側の文字に注目して，もしその文字がパターン中に含まれていたら，両者が重なる位置までパターンを右にずらす．そうしないと，同じ場所で再び不一致を検出してしまうからである．また同じ理由により，不一致を引き起こしたテキスト上の文字がパターン中に含まれていない場合，パターン先頭を不一致を起こしたテキスト上の文字より次の位置にまでずらすことができる．この様子を図 2.5，照合部分のプログラムを図 2.6 に示す．図 2.5 ではパターンを「KYOKU」として照合している．

KMP 法同様，BM 法でもパターンの移動量をあらかじめテーブルに登録する．KMP 法のテーブルは不一致を起こしたパターン上の位置をインデックスに

1 文字目で不一致の場合	?	?												
無条件に 1 文字ずらす		K												
2 文字目で不一致の場合	K	?												
1 文字ずらせば一致の可能性あり		K												
3 文字目で不一致の場合	K	Y	?	?										
1 文字ずらしても一致しない		K	Y											
2 文字ずらせば一致の可能性あり			K	Y										
4 文字目で不一致の場合	K	Y	O	?	?	?								
1 文字ずらしても一致しない		K	Y	O										
2 文字ずらしても一致しない			K	Y	O									
3 文字ずらせば一致の可能性あり				K	Y	O								
5 文字目で不一致の場合	K	Y	O	K	?	?	?							
1 文字ずらしても一致しない		K	Y	O	K									
2 文字ずらしても一致しない			K	Y	O	K								
3 文字ずらせば一致の可能性あり				K	Y	O	K							
9 文字目で不一致	K	Y	O	K	A	K	Y	O	K	?	?	?	?	?	?
5 文字ずらすと一致の可能性あり		K	Y	O	K	A	K	Y	O	K	U				

図 2.3: KMP 法の例

```
1:  for(i=start;i<=end;i++)
    {
        /* 文字コードが一致した場合、テキストとパターン
                                        の注目位置を進める */
2:    if ( temp[i] == pattern[j] )
      {
3:          i++;
        /* 全て一致した!! */
4:        if (pattern[++j] == '¥0' )
          {
5:          printf("fount at %d %c¥n¥n", (i-Plen),temp[i-Plen]);
6:              num++;
7:              break;
          }
        }
        /* パターン1文字目で不一致の場合はテキストの注目位置を進める */
8:    else if(j == 0)
9:            break;
      /* パターン1文字目以外で不一致の場合はKMP_TABLEから
                                        パターンの注目位置を取得する */
10:   else
11:       j = KMP_TABLE[j] ;
      }
12:   if(num == 0)
      printf("(null)¥n");
     }
```

図 2.4: KMP 法の照合部分プログラム

テキスト	T	O	K	K	Y	O	K	Y	O	K	A	K	Y	O	K	U
パターン	K	Y	O	K	U										
1.	T	O	K	K	*Y*	O	K	Y	O	K	A	K	Y	O	K	U
最後の文字を比較	K	Y	O	K	*U*										
2.	T	O	K	K	Y	O	K	*Y*	O	K	A	K	Y	O	K	U
違うので3つずらす				K	Y	O	K	*U*								
3.	T	O	K	K	Y	O	K	Y	O	K	*A*	K	Y	O	K	U
違うので5つずらす							K	Y	O	K	*U*					
4.	T	O	K	K	Y	O	K	Y	O	K	A	*K*	*Y*	*O*	*K*	*U*
								照合成功				*K*	*Y*	*O*	*K*	*U*

図 2.5: BM 法の例

していたが，BM 法では不一致を起こしたテキスト側の文字がインデックスとなる．このテーブルには，右端の文字位置を 0 として，各文字がパターンの右端から何文字目に現れるかを登録する．パターン中に同じ文字が複数ある場合には，その最も右側の位置を採用する．パターンが KYOKU という文字であった場合は，パターン長は 5 で，パターンの最後の文字からの長さは表 2.1 のようになる．したがって，照合を失敗したときに参照する BM 法のテーブルには表 2.2 のように入る．

表 2.1: パターンの配列

4	3	2	1	0
K	Y	O	K	U

表 2.2 のテーブルに沿ってテキストの文字を見てずらす量を変更する．
このように BM 法では，文字を 1 回比較するだけでパターンを最大 m 文字分ずらすことができる．したがって，うまくいけば O(n/m) 程度の計算量で 文字列の探索を行うことが可能である．パターンを末尾から先頭に向かって逆向きに

表 2.2: BM 法テーブル

K	1
Y	3
O	2
U	0
それ以外	5

チェックするというのが BM 法のミソで，種明かしをされてみれば，ごく当然の発想のように思える．こんな当たり前のことに，1977 年まで誰も 気がつかなかったとは，むしろ意外に感じられるほどである．

この BM アルゴリズムを改良した Boyer-Moor-Horspool のアルゴリズムは，英文の場合，ほぼ原文 1 文字あたり 0.24 回から 0.3 回の比較で検索できるという．

```
    /*アルゴリズムとは逆に後ろから見ている*/
      (プログラムの書きやすさから)
    /*テキストの最後からパターンの長さ引いたところからスタート*/
1:  for(i = end - Plen;i>=start;i--)
    {
        j = 0;
2:    while( temp[i] == pattern[j] )
      {
        /* 全て一致した!! */
3:        if ( j == Plen-1 )
          {
4:          printf("float at %d %c\n\n",i,temp[i]);
5:              num++;          /*一致をカウント
            /*今までiを増やしてきたから、パターンの2倍iを前にする
              ことで、今まで考えたところを飛ばして、初めと同じように
              検索*/
6:              i = i - 2*Plen;
7:              break;
          }
8:              i++;
9:              j++;
      }
      /*テキストとパターンが一致しなかった*/
      /*テキストの文字を見てテキストをずらす*/
10:     i -= BM_TABLE[temp[i]];
    }
11:   if(num == 0)
12:     printf("(null)\n");
```

図 2.6: BM 法照合部分プログラム

第3章　文字列照合の並列化手法

3.1　PCクラスタと並列処理

3.1.1　PC クラスタ

クラスタとは，LAN に接続された多数のコンピュータによって構成された並列計算機のことである．一般に使われているパーソナルコンピュータ (PC) を用いて構成されたクラスタを，PC クラスタという．PC クラスタにおいては，クラスタを構成する各マシンの独立性が高いので，システムの拡張が容易であり，各マシンに PC を用いているので低価格での構成が可能である．このような利点により，普及が進んでいる．

PC クラスタでは，クラスタ内の多くの異なったノード間でタスクを分割し，性能向上を図ることを主要な目的としているので，HPC クラスタと呼ばれる．HPC クラスタの中にも様々な種類が存在するが，ここでは Beowulf 型クラスタと SCore 型クラスタについて説明する．

(1)Beowulf型クラスタ

Beowulf とは，HPC クラスタを構成する方式の総称のことであり，Beowulf 方式で構成されたクラスタを Beowulf 型クラスタという．各ノードには Linux などのフリーの UNIX 系 OS がインストールされており，互いに高速なネットワークで接続されている．並列プログラミング環境として PVM や MPI などのライブラリを用いることが多い．クラスタシステムソフトウェアなどの特別なソフトウェアを使わずに構成できることが大きな特徴である．

(2)SCore 型クラスタ

SCore 型クラスタとは，PC Cluster Consortium[14] で配布されている，SCore Cluster System Software を用いて構成されたクラスタである．SCore Cluster

Systerm Software は，通信ライブラリである PMv2，グローバルオペレーティングシステムである SCore-D，ソフトウェア DSM(Distributed Shared Memory) システムである SCASH などで構成されているが，一番の特徴は，SCASH によって物理的に分散したメモリを共有メモリとして扱うことができることである．これにより，単一プロセッサの PC を繋げた PC クラスタでも，共有メモリ環境で並列プログラミングを行うことができる．

3.1.2 本研究で使用するクラスタ

(1)Nycto クラスタ

Nycto クラスタの構成を図 3.1 に示す．Nycto の計算ノードは, Nyctor00 と Nycto01 の計 2 台で, それぞれが Gigabie Ethernet Switch を介して, ホストサーバである Bronto と繋がっている．Nycto への操作は Bronto を介して行う．Nycto 上には, OS である CentOS4.4 の他に, PC クラスタ用の高性能並列プログラミング環境である SCore（Ver.6.0.2）が動作している．Nycto を使用するためにはまず Score を起動させなければならない．

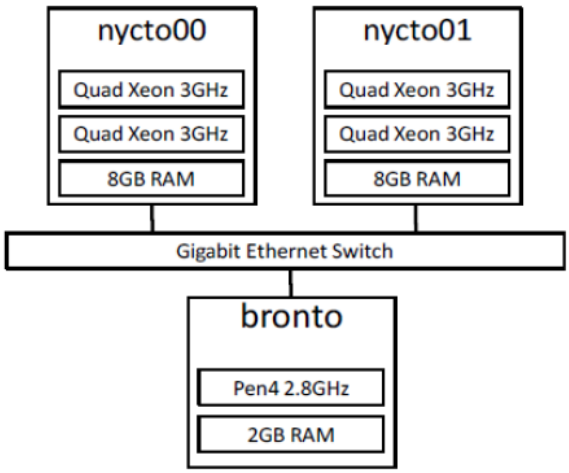


図 3.1: Nycto の構成

(2)Diplo クラスタ

Diplo クラスタの構成を図 3.2 に示す． Diplo の計算ノードは Diplo00 から Diplo03 までの計 4 台で, それぞれのノードに 2 台の Dual Intel Xeon プロセッサが搭載されている. つまり, 1 台の計算ノードに 4 台のプロセッサが搭載されている．これらのノードは Gigabit Ethernet Switch を介して, ホストサーバである Tarbo と繋がっている. Diplo クラスタへの操作は Tarbo を介して行う. Diplo クラスタは，Rocks Cluster Distribution と呼ばれるクラスタ構築ツールによって構築されている．この Rocks には仮想的に共有メモリを作る機能がないので, Diplo は Beowulf 型クラスタである.

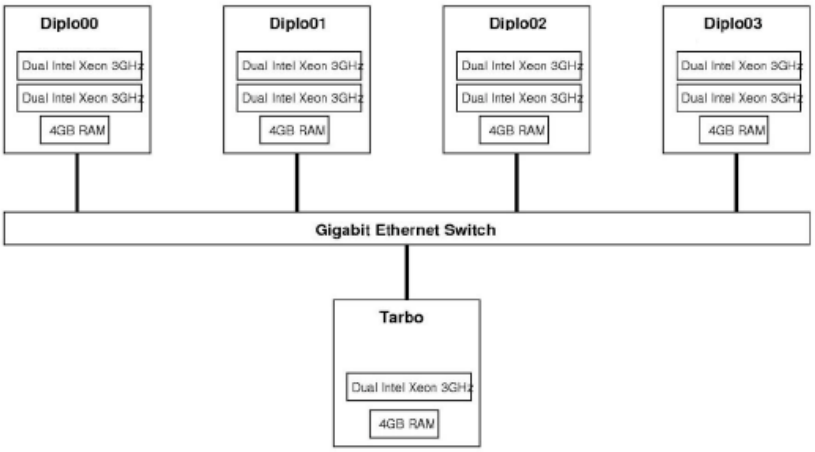


図 3.2: Diplo の構成

3.1.3 並列プログラミング環境

並列プログラミングは，一般的に並列処理ライブラリと呼ばれるライブラリを，C や Fortran などの逐次処理用のプログラムから呼び出すことにより実現される．代表的な並列処理ライブラリには，PVM・MPI・OpenMP がある.

(1)Parallel Virtual Machine(PVM)

PVM は，ネットワークで接続されたコンピュータを，仮想的な並列計算機として扱うためのソフトウェアツールである．動作するマシンの種類が多く，入

手方法が容易であるため広く利用されている．PVM ではメッセージパッシングによって並列処理を行うため，分散メモリ型の並列処理に適している．

(2)Message Passing Interface(MPI)

MPI は，並列処理用のメッセージパッシングの標準化された規格である．PVM と同様，分散メモリ型の並列処理に適しているが，ほとんどの言語でサポートされる高い移植性や，PVM には無い非同期通信のサポートなどの特徴により，現在は PVM より優勢である．
MPI においては，その都度メッセージパッシングを指定しなければならないのでプログラムの記述は難しいが，上手くプログラムを記述することによって大きな性能向上が期待できる．Raptor クラスタ上では MPICH-SCore，Diplo クラスタ上では MPICH という形で，MPI は実装されている．MPI プログラムのコンパイルには，どちらの環境でも mpicc を用いる．

(3)OpenMP

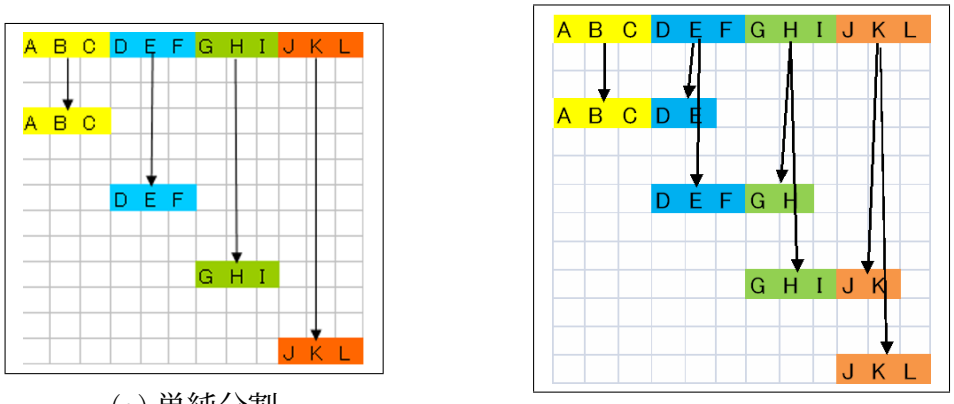
OpenMP は，並列処理用の標準化された基盤である．PVM や MPI と異なり，共有メモリ型の並列処理で使用することを前提としている．OpenMP の最大の利点は，並列化構文を容易に記述できることである．具体的には，逐次プログラムに#pragma omp で始まる，プラグマ指示文と呼ばれる文を挿入するだけで簡単に並列化が可能である．このプラグマ指示文は，OpenMP が実行できない環境では無視されるので，並列プログラムと逐次プログラムがほぼ同一のプログラムとなるため，デバッグが他の並列プログミング環境に比べて容易である，といった利点もある．しかし，共有メモリ環境以外の環境では実行できず，性能も MPI で効率的にメッセージパッシングを行った場合に劣るといった欠点もある．
Raptor クラスタ上での OpenMP プログラムのコンパイルには Omni Compiler を，Diplo クラスタ上での OpenMP プログラムのコンパイルには Intel C/C++ Compiler を，それぞれ用いる．

(4) ハイブリッド並列プログラミング

SMP クラスタにおいて，ノード間を MPI などの分散メモリ型で，ノード内を OpenMP などの共有メモリ型で，それぞれ並列化するプログラミング手法を，ハイブリッド並列プログラミングという．これにより SMP クラスタの特徴を十分生かしたプログラミングを行うことが可能となる．Diplo クラスタ上で，MPI と OpenMP で記述されたハイブリッド並列プログラムをコンパイルする場合は，MPI プログラムのコンパイラ mpicc から，Intel C/C++ Compiler を呼び出し，MPI の構文と OpenMP の構文に対応させる．

3.2 並列化手法

並列化の手法は簡単でテキストをしようするプロセッサ数でテキストを分割し，分割されたテキストを各プロセッサに送信する．このとき，単に分割するのではなく，分割されたテキストに (パターンの長さ-1) のテキストをつけ，重複させるようにする．上記をすることによって，各プロセッサに別々に処理されてしまったテキストの位置にパターンの文字列があっても発見することができる．



(a) 単純分割　　(b) 重複分割

図 3.3: テキストの分割処理

図 3.3 の (a) では分割された境目にパターンの文字が入っていた時に照合できないため，(b) にすることで境目で文字列が別れて照合できない可能性がない．そのため，本研究のプログラムでは，(b) のように記述を行った．

```
#pragma omp parallel private(Textthread,start,end) shared(text,pattern,Tlen)
{
Textthread = Tlen / omp_get_num_threads();      /*プロセッサ数でText
を分割*/
 if((Tlen % omp_get_num_threads())!= 0)
 Textthread++;
 start = Textthread * omp_get_thread_num();
 end   =(Textthread * (omp_get_thread_num()+1))-1 + (Plen-1);

                #pragma omp for
 for(i=start; i<=end; i++)
        以下文字列照合処理
```

図 3.4: テキストの分割処理部分

並列処理の部分は図 3.4 のように記述する．

pragma omp parallel の記述で並列処理がここから始まることが宣言されている．private の変数は各プロセッサごとに違う変数が入ることを示し，shared の変数はどのプロセッサも同じ変数になる．
本研究のプログラムでは，文字列照合を行う処理においてのみ用いた．Textthread は各プロセッサに送信される，start と end は各プロセッサに送られるテキストの配列のはじめを終わりを示す．さらに，OpenMP では for ループのみ可能な#pragma omp for を使用することにより，for ループを行っているところで並列処理をしている．上記の計算をすることにより，各プロセッサごとに文字列照合の処理が分割されたテキストの範囲のみで行うことができる．

第4章　PCクラスタ上での性能評価

4.1　実験内容

2章と3章で述べたアルゴリズムをOpenMPで実装し，Diploクラスタでは1,2,4プロセッサ，Nyctoクラスタでは1,2,4,6,8プロセッサで実行した．入力データとして，テキストを7,718,926バイト，パターンを5バイト(null:テキストにない文字列),8バイトで実験を行った．出力された処理時間の平均をとり，検証を行った．なお，1つのクラスタ上での実験はそれぞれ，テキスト，パターン共に同じものを扱った．

4.2　Diploクラスタ上での実装

実験結果は表4.1のようになる．また，速度向上比は図4.1になる．テキストは7,718,926バイト，パターンは5バイトと8バイトで検索したときの平均値の結果である．

表 4.1: Diploクラスタ上処理時間 (ms)

スレッド数	1	2	4	6	8	16
単純照合	77.2	32.9	17.2	11.1	8.8	6.7
KMP法	60.0	32.2	14.3	10.5	8.0	9.2
BM法	14.6	13.1	10.2	10.9	7.7	8.9

図4.1をみると，単純照合では16プロセッサの時，1プロセッサの時に比べて，11.5倍の速度向上が得られた．KMP法では6.5倍，BM法では1.6倍という速度向上が得られた．しかし，表4.1をみるとわかるように，単純照合では，プロセッサ数をあげると，速度向上は得られるが，処理時間としては，KMP法及びBM法での1プロセッサの時よりも単純照合での4プロセッサの時のほうが遅くなっている．

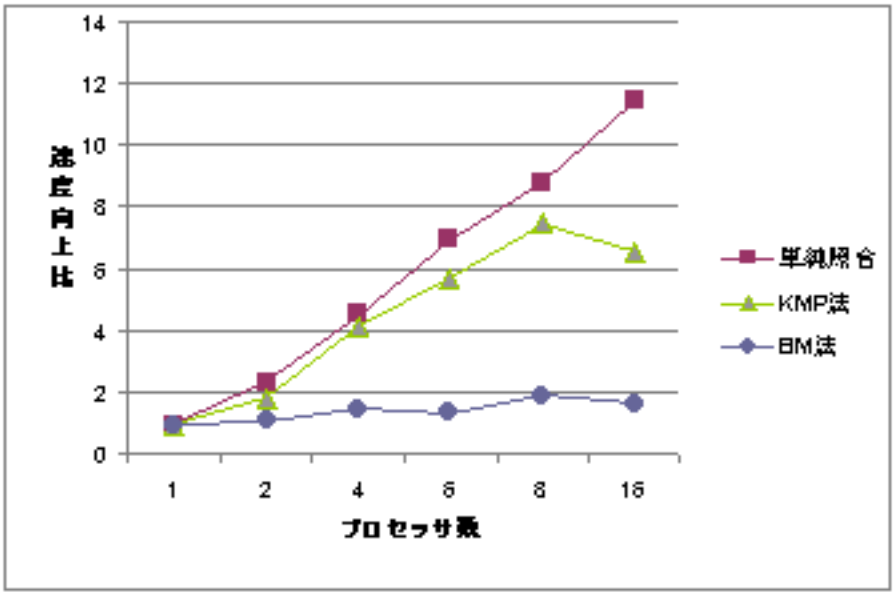


図 4.1: Diplo 速度向上比

4.3 Nycto クラスタ上での実装

実験結果は表 4.2 のようになる．テキストは 7,718,926 バイト，パターンは 5 バイトと 8 バイトで検索したときの平均値の結果である．

表 4.2: Nycto クラスタ上処理時間 (ms)

プロセッサ数	1	2	4	6	8
単純照合	104.6	98.8	74.9	50.7	21.7
KMP 法	96.9	43.5	24.2	16.5	9.6
BM 法	16.1	11.3	6.4	5.1	4.6

図 4.2 の Nycto 速度向上比から明らかであるが，単純照合では 1 プロセッサに比べて 8 プロセッサでは 4.3 倍の速度向上，6 プロセッサまでは速度向上は見られなかったが，8 以上のプロセッサ台数にすると速度向上は望めると考えられる．

KMP 法においては，かなりの速度向上が見られた．どのプロセッサもプロセッサ数以上の速度向上が望めた．また，プロセッサ数 8 においては，速度向上比が 10 倍近くあり，プロセッサ数を増やせば増やすほど，そのだけ，速度向上

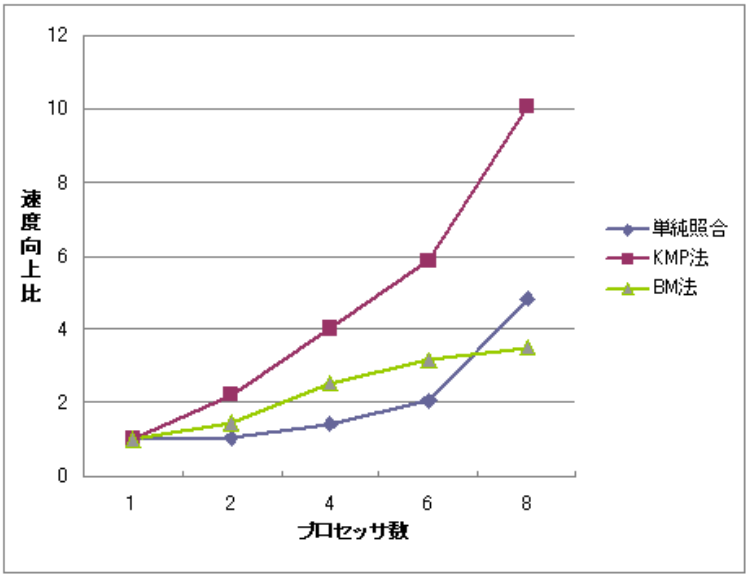


図 4.2: Nycto 速度向上比

が望めるという結果になった．BM 法においては，1 プロセッサに対して 8 プロセッサの時は 3.5 倍の速度向上が得られた．2,4,6 のプロセッサについても，比例する形で速度向上が望めることが分かった．Diplo クラスタ同様，単純照合において，8 プロセッサで 4.3 倍もの速度向上が望めるが，表 4.2 でもわかるように，単純照合の処理時間は KMP 法，BM 法に比べて非常に遅いことがわかる．

4.4 考察

4.2 章の Diplo クラスタと 4.3 章の Nycto クラスタの結果では，Nycto クラスタの時の方が，4 プロセッサの時を比べても，KMP 法と BM 法では Nycto クラスタで実行したときの方が少し速い．これは，この結果はテキストの量にも関係してくると思うが、通信速度が Diplo クラスタの方が遅いことが考えられる．

Diplo クラスタでは、どのアルゴリズムもプロセッサを増やすごとに速度が向上している。しかし、Nycto クラスタでは、単純照合では、6 プロセッサまでは速度向上はあまり見られなかったが 8 プロセッサで急に速度が上がった。逆に BM 法では、プロセッサ数を増やすと速度向上は収束していく．このような結果は同じ状況の下での実験のため、アルゴリズムによるためだと考えられる．単純に速度向上を増やすのであれば、分散メモリである、Diplo クラスタを利用

する方が良いと考えられる．

また，一番効率が良いとされている BM 法であるが，今回 KMP 法の方が並列処理に適している結果になった．BM 法については，今回作成したプログラムの改善が必要であるかもしれない．

第5章　おわりに

本研究では，文字列照合に関するパターンの探索問題を取り上げ，代表的な照合のアルゴリズムである単純照合，KMP 法，BM 法の 3 つを並列化した．並列化したアルゴリズムは，SMP クラスタである Diplo(4 プロセッサ) と Score 型クラスタである Nycto(8 プロセッサ) 上でそれぞれ OpenMP を用いて実装した．

計算速度の向上では，Diplo 上では，3 つのアルゴリズムすべてがプロセッサ数に従って，速度向上する結果になった。それぞれ 16 プロセッサでは 1 プロセッサで実行させたときに比べて，単純照合では 11.5 倍，KMP 法では 6.5 倍，BM 法では 1.6 倍の速度向上が得られることがわかった。Nycto 上では，単純照合はプロセッサの数が少ないと速度が遅くなってしまうが，8 プロセッサ以上に挙げると速度向上が望める．また，KMP 法においては，プロセッサ数以上の速度向上が望めることが実験によって得られた．一番効率の良いとされている，BM 法においては，今回の実験では 8 プロセッサで 3.5 倍の速度向上であり，KMP ほどではなかったが，どのアルゴリズムも速度向上が望めることが実験によって結果として得られた．

今後の研究課題として，まず，4 章の Diplo 上で速度向上が得られなったこと，BM 法の速度向上が KMP 法よりも望めなったことについて，検証することが挙げられる．また，今回取り上げた単純照合，KMP 法，BM 法の 3 つのアルゴリズム以外にも，近似文字列照合や BM 法の改良版 Boyer-Moor-Horspool アルゴリズム等のさらに高速なアルゴリズムを設計し，PC クラスタ上で並列化を行うことが挙げられる．

謝辞

本研究を遂行するにあたり全過程を通じて懇切丁寧なる御指導御鞭撻を賜わった立命館大学理工学部山崎勝弘教授に深く感謝致します.

高性能計算研究室の皆様には日頃より多くの御助言御協力戴いた種々の面でお世話になりました．ここに深謝の意を示します.

参考文献

[1] 北山 洋幸:, “ OpenMP 入門—マルチコア CPU 時代の並列プログラミング, ”秀和システム，2009.

[2] 検索アルゴリズム 単純照合アルゴリズム,KMP 法アルゴリズム <http://fussy.web.fc2.com/algo/algo7-3.htm>.

[3] 検索アルゴリズム BM 法アルゴリズム <http://fussy.web.fc2.com/algo/algo7-4.htm>.

[4] OpenMP 入門 <http://www.na.cse.nagoya-u.ac.jp/ reiji/lect/hpc02/OpenMPintro.html>.

[5] 三木光範他:PC クラスタ超入門 2000,“ PC クラスタ型並列計算機の構築と利用, ”超並列計算研究会，<http://mikikab.doshisha.ac.jp/dia/smpp/cluster2000/>,2000.

[6] 三木光範他:PC クラスタ超入門 1999,“ PC クラスタ型並列計算機の基礎と講習, ”超並列計算研究会，<http://www.is.doshisha.ac.jp/SMPP/report/1999/990910/9909-1lecture.pdf>,1999.

[7] 片山勝他：ハッシュ型オートマトンによる文字列検索の並列化手法, 電子情報通信学会 日本電信電話株式会社 NTT ねっと研究所 <http://ci.nii.ac.jp/naid/110003178180>.

[8] 田中秀宗: PC クラスタ上での文字列最適周期アルゴリズムの並列化, 立命館大学理工学部情報学科卒業論文, 2007.

[9] 池上広済: ハイブリッド並列プログラミングによる MPEG2 エンコーダの高速化, 立命館大学院理工学研究科修士論文, 2006.

[10] 加藤寛暁: SMP クラスタ上での OpenMP による MPEG2 エンコーダの並列化, 立命館大学院理工学部情報学科卒業論文, 2006.

[11] 山本毅雄他：”全文検索　-技術と応用,”丸善,1998.

付 録A　KMP法のソースコード

ckmpmatchfile_omp.c

```
#include <stdio.h>
#include <omp.h>
#include <string.h>
#include <sys/time.h>

#define MAXLENGTH  1000
int KMP_TABLE[MAXLENGTH];  /* 不一致時に再び比較を始めるパターン上の
位置 */

double second()
{
  struct timeval tv;
  gettimeofday(&tv,NULL);
  return tv.tv_sec + tv.tv_usec/1000000.0;
}

int make_kmp_table(const char *Pattern)
{

int txt_index = 1;
int pat_index = 0;

KMP_TABLE[txt_index] = 0;
```

```
printf("KMP_TABLE run.\n\n");

while ( Pattern[txt_index] != '\0' )
{
/* 一致したら、テーブルに次の文字位置を登録しつつ両者の位置を進める */
if ( Pattern[txt_index] == Pattern[pat_index] )
{
KMP_TABLE[++txt_index] = ++pat_index;
/* 一致しなかった場合、KMP 法を使ってパターンをずらす */
/* パターン 1 文字目で不一致の場合は、テーブルに 0 を登録してテキスト
                の注目位置を進める */
/* 次の文字も一致したら、再度テーブルから文字位置抽出して登録 */
if ( Pattern[txt_index] == Pattern[pat_index] )
KMP_TABLE[txt_index] = KMP_TABLE[pat_index];
}
else if ( pat_index == 0 )
{
KMP_TABLE[++txt_index] = 0;
/* パターン 1 文字目以外で不一致の場合は、テーブルから注目位置を抽出して
                パターンの注目位置とする */
}
else
{
pat_index = KMP_TABLE[pat_index];
}
}
return( 1 );
}

int main(void){
FILE *fp;
char text[1024];      /*入力文字列*/
char pattern[256];  /*入力検索文字列*/
```

```
char temp[10000000];
int Tlen,Plen=0;    /*Pattern の文字列長*/
int i,j,start,end;
int Textthread;
int num=0;
int c = 0;
        double time_start,time_end,time;

      fp =fopen("sample.txt","r");

//ファイルオープンに失敗したとき
if(fp == NULL)
{
//失敗を表示し終了
printf("File open ERROR");
return -1;
}
//fgets の戻り値が null になるまで続ける
//buff1 にファイルからバイト取得し格納

while((fgets(text,1024,fp))!=NULL)
{

if(c==0)
strcpy(temp,text);

strcat(temp,text);
c+=1;
}

printf("Length of temp is %d\n",strlen(temp));
```

```
printf("Type Find Letter : ");
scanf("%s",pattern);

Tlen = strlen(temp);
Plen = strlen(pattern);

   time_start = second();


            make_kmp_table(pattern);

#pragma omp parallel private(Textthread,start,end) shared(temp,pattern,Tlen)
         num_threads(8)
{


Textthread = Tlen / omp_get_num_threads();     /*プロセッサ数でText
を分割*/
if((Tlen % omp_get_num_threads())!= 0)
Textthread++;
start = Textthread * omp_get_thread_num();
end   =(Textthread * (omp_get_thread_num()+1))-1 + (Plen-1);

printf("\nstart %d %c end %d %c\n",start,temp[start],end,temp[end]);
printf("Plen %d \n",Plen);
printf("Textthread %d \n\n",Textthread);


j = 0;
i = start;
        #pragma omp for
         for(i=start; i<=end; i++)
         {
```

```
  /* 文字コードが一致した場合、テキストとパターンの注目位置を進める */
  if ( temp[i] == pattern[j] )
       {
    /* 全て一致した!! */
        if (pattern[++j] == '\0' )
{
  printf("fount at %d %c\n\n", (i-Plen),temp[i-Plen]);
        num++;
    j=0;
 }
  }
    /* パターン 1 文字目以外で不一致の場合は KMP_TABLE からパターンの注
目
                          位置を取得する */
    else   if(j != 1)
    {
j = KMP_TABLE[j];
                    }
  }
    if(num == 0)
      printf("(null)\n");
       }

//ファイルを閉じる
    fclose(fp);

    time_end = second();
    time = time_end -time_start;
    printf("time is %fsecond\n",time);


return 0;
}
```

付 録B　BM法のソースコード

cbmmatchfile_omp.c

```
#include <stdio.h>
#include <omp.h>
#include <string.h>
#include <sys/time.h>

#define UCHAR_MAX 1000
int BM_TABLE[UCHAR_MAX + 1];

double second()
{
  struct timeval tv;
  gettimeofday(&tv,NULL);
  return tv.tv_sec + tv.tv_usec/1000000.0;
}


int make_bm_table(const char *pattern )
{
int i;
int length;

printf("BM_TABLE run.\n\n");

/* テーブルをパターンの長さで初期化する */
```

```
length = strlen( pattern );
for ( i = 0 ; i <= UCHAR_MAX ; i++ )
BM_TABLE[i] = length;

/* パターンの先頭から、文字に対応する位置に末尾からの長さを登録する */
for (i = length-1 ; i > 0 ; i-- )
BM_TABLE[pattern[i]] = i;

}

int main(void){
FILE *fp;
char text[1024];  /*入力文字列*/
char pattern[256];  /*入力検索文字列*/
char temp[10000000];
int Tlen,Plen = 0;
int i,j,start,end;
int Textthread;
int num = 0;
int n;
int c = 0;
double time_start,time_end,time;



fp = fopen("sample.txt","r");

//ファイルオープンに失敗したとき
if(fp==NULL)
{
//失敗を表示し終了
printf("File open ERROR");
return -1;
}
```

```
//fgetsの戻り値がnullになるまで続ける
        //strにファイルからバイト取得し格納

    while((fgets(text,1024,fp))!=NULL)
    {



if(c==0)
        strcpy(temp, text);
strcat(temp, text);
c+=1;
    }

      printf("Length of temp is %d\n", strlen(temp));

printf("Type Find Letter : ");
scanf("%s",pattern);

      time_start = second();



Tlen = strlen(temp);
Plen = strlen(pattern);

         make_bm_table(pattern);



   #pragma omp parallel private(Textthread,start,end) shared(temp,pattern,Tlen)
      num_threads(8)
    {

Textthread = Tlen / omp_get_num_threads();      /*プロセッサ数でText
```

```
を分割*/
if((Tlen % omp_get_num_threads())!= 0)
Textthread++;

start = Textthread * omp_get_thread_num();
end   =(Textthread * (omp_get_thread_num()+1))-1 + (Plen-1);

printf("\nstart %d %c end %d %c\n",start,temp[start],end,temp[end]);
printf("Plen %d \n",Plen);
printf("Textthread %d \n\n",Textthread);

#pragma omp for
for(i = end - Plen;i>=start;i--)
        {
j = 0;
    while( temp[i] == pattern[j] )
    {
/* 全て一致した!! */
if ( j == Plen-1 )
{
    printf("float at %d %c\n\n",i,temp[i]);
num++;
i = i - 2*Plen;
break;
}
i++;
j++;
    }
i -= BM_TABLE[temp[i]];
}
if(num == 0)
printf("(null)\n");
    }
```

```
time_end = second();
time = time_end - time_start;
printf("time is %fsecond.\n",time);


    return 0;
}
```
