卒　業　論　文

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

氏名　　：　田中 秀宗
学籍番号　：　2210030228-8
指導教員　：　山崎 勝弘 教授
提出日　　：　2007 年 2 月 19 日

立命館大学　理工学部　情報学科

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，最適な繰り返しの概念の 1 つである，最適周期を取り上げた．最適周期の概念を用いたとき，与えられた文字列とパターンがどれだけ離れているかを調べる誤り測定問題と，与えられた文字列はどのようなパターンの繰り返しになっているかを調べるパターン探索問題が考えられる．それぞれの問題に対するアルゴリズムを並列化し，PC クラスタ上で実装した．実装した PC クラスタは，分散共有メモリ環境を持つ Raptor(8 プロセッサ) と，各計算ノードが SMP(対称型マルチプロセッシング) 構造を持つ Diplo(16 プロセッサ) である．Diplo クラスタへの実装では，ノード間を分散メモリ環境，ノード内を共有メモリ環境でそれぞれ並列処理を行う，ハイブリッド並列化も行った．

並列処理を行うと，誤り測定問題では，Raptor クラスタで最大約 4.8 倍，Diplo クラスタで最大約 11.2 倍の速度向上を得た．また，パターン探索問題では，Raptor クラスタで最大約 6.2 倍，Diplo クラスタで最大約 9.0 倍，Diplo クラスタでハイブリッド並列化を行った場合には最大約 6.8 倍の速度向上を得た．

目次

図目次

表目次

1 はじめに

繰り返しのある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．文字列が繰り返しのある構造をしているとき，その繰り返し (パターン) を正規性 (regularity) という．正規性には様々なものがあり，周期 (period)・カバー (cover)・シード (seed)・反復 (repetition) などの重要な正規性については，文字列の中からこれらの正規性を取り出すアルゴリズムなどの研究が活発に行われてきた．

正規性の概念を拡張したものとして，誤りを含んだ繰り返しが考えられる．誤りを含んだ繰り返しを，最適な (approximate) 繰り返しという．特に，分子生物学やコンピュータを利用した音楽解析の分野では，厳密な文字列マッチングよりも，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 [1]．また，最適な繰り返しにおいては，マッチングを行う本来の繰り返しと，マッチングが行われる文字列中の繰り返しとの距離が重要となる．文字列間の距離を測るための代表的な距離関数として，ハミング距離・編集距離 (レーベンシュタイン距離)・重み付き編集距離などが挙げられる．この最適な繰り返しに関するアルゴリズムは，入力された文字列の全ての部分列に対して距離を計算する必要があるため，計算量が大きくなるという特徴がある．

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) 構造を持つ PC を用いて構成する，SMP クラスタが注目を集めている．SMP 構造は，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，周期の概念を誤りを考慮するよう拡張した，最適周期と呼ばれる繰り返しの概念を取り上げる．最適周期の概念を用いたとき，与えられた文字列とパターンはどれだけ離れているのか．また，与えられた文字列はどのようなパターンの繰り返しになっているのか．前者を誤り測定問題，後者をパターン探索問題と名付け，これらの問題に対するアルゴリズム [1] を並列化し，PC クラスタ上に実装して検証を行った．計算に用いたクラスタは，SCore を導入した Raptor クラスタと，SMP クラスタである Diplo クラスタである．実装は，両方の問題で MPI(Message Passing Interface) を用いたが，Diplo クラスタ上でパターン探索問題を計算するときのみ，ノード間は MPI・ノード内は OpenMP で並列計算を行う，ハイブリッド並列化の手法で実装を行った．

本論文の構成は次の通りである．2 章では，PC クラスタとその分類，また本研究で用いる 2 つの

PC クラスタについて述べた後，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る．3 章では，文字列間の距離として採用する重み付き編集距離と，文字列の最適周期，及び文字列の最適周期に関する 2 つの問題の定義を行い，その問題を解く逐次アルゴリズムについて述べる．4 章・5 章では，これら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 クラスタ上での検証結果と考察について述べる．

2 PC クラスタと並列処理

2.1 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．

2.1.1 HPC クラスタ

HPC クラスタとは，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たクラスタである．多くの PC クラスタは，この HPC クラスタに分類される．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．

2.1.2 本研究で使用するクラスタ

本研究では，高性能計算研究室が所有している，Raptor・Diplo と呼ばれる 2 つの PC クラスタを使用し，検証を行った．以下でこれらのクラスタについて説明する．

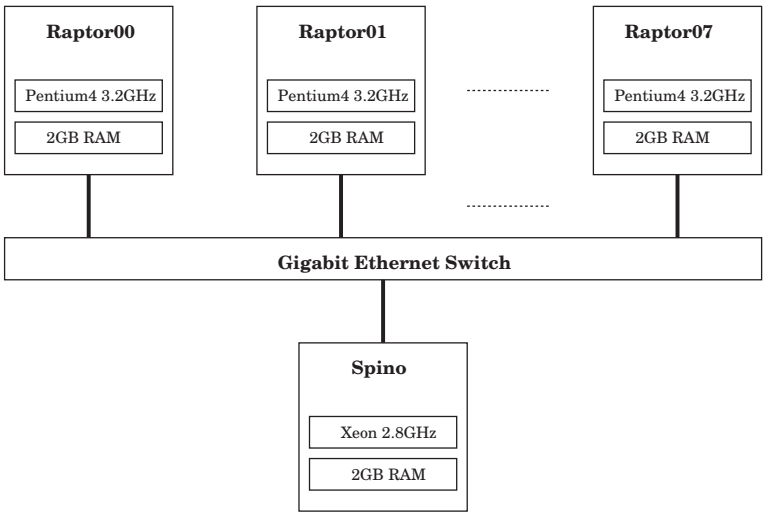

図 1　Raptor クラスタの構成

(1)　Raptor クラスタ

図 1 に Raptor の構成を示す．Raptor の計算ノードは，Raptor00 から Raptor07 までの計 8 台で，それぞれが Gigabie Ethernet Switch を介して，ホストサーバである Spino と繋がっている．Raptor は，SCore Cluster System Software がインストールされているため，SCore 型クラスタであり，共有メモリ型の並列処理が可能である．

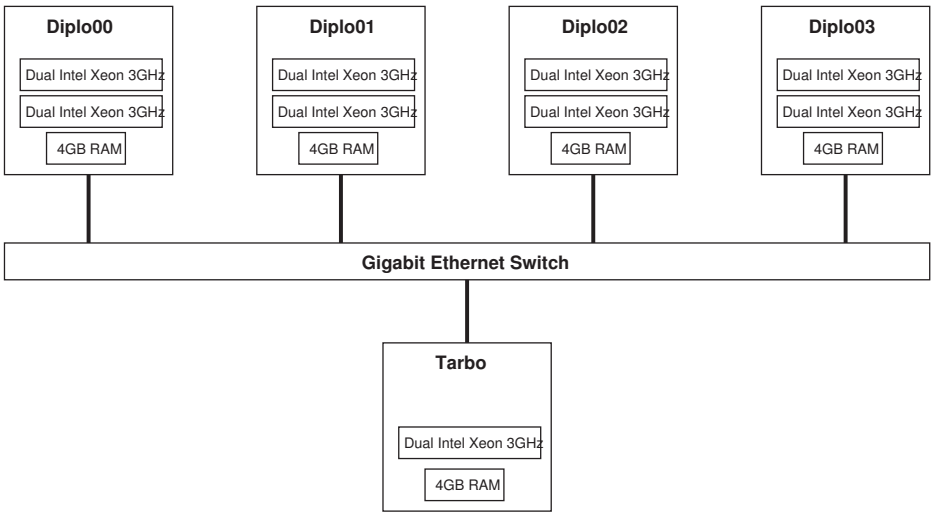

図 2　Diplo クラスタの構成

(2)　Diplo クラスタ

図 2 に Diplo の構成を示す．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 個の 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 個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は無いので，Diplo は Beowulf 型クラスタであり，計算ノードを跨いでメモリを共有メモリとして扱うことは不可能である．

2.2 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　文字列の最適周期アルゴリズム

最適周期 (approximate period) とは，代表的な正規性の一つである，周期を誤りを認めるよう拡張した概念である．ここでは，最適周期とそれに関する問題を定義し，その問題を解くための逐次アルゴリズム [1] について述べる．

3.1　最適周期の定義とその問題

3.1.1　文字列間の距離

誤りを認めた文字列マッチングを行うためには，文字列間の距離を定義する必要がある．本研究では，文字列間の距離として重み付き編集距離を採用した．まず，一般的な編集距離 (レーベンシュタイン距離) の定義を述べたのち，重み付き編集距離の定義を述べる．

定義 1(編集距離)

ある 2 つの文字列に対して，一方の文字列をもう一方の文字列に置き換えるときに必要な文字の挿入・削除・置換の最小回数を，**編集距離**(edit distance) という．

例 1. “GCA” と “AGT” との編集距離は 3 である．
これは，次の 3 つの操作で “CGA” が “AGT” に変換できるためである．

1. 1 文字目の G の前に A を挿入
   GCA → AGCA
2. 4 文字目の A を削除
   AGCA → AGC
3. 3 文字目の C を T に置換
   AGC → AGT

定義 2(重み付き編集距離)

挿入・削除・置換にかかるコストを全ての文字に対して定義し，計算した編集距離を，**重み付き編集距離**(weighted edit distance) という．

各コストは，表 1 に示すようなペナルティ行列によって与えられる．ここで，Δ は空白文字を表し，行列の各要素は行の文字を列の文字で置換するコストを表している．また，挿入にかかるコストを “Δ を置換するコスト” で，削除にかかるコストを “Δ で置換するコスト” でそれぞれ表す．

以下では，文字 c を文字 c' で置換するコストを $\delta(c, c')$ と表す．例えば，ペナルティ行列が表 1 で与えられたとき，$\delta(\mathtt{A}, \mathtt{C}) = 2$ である．

例 2. ペナルティ行列が表 1 で与えられたとき，”GCA” と “AGT” との重み付き編集距離は 5 である．
これは，次の 3 つの操作にかかるコストの合計が，以下のように 5 となるためである．

表 1　ペナルティ行列の例

	Δ	A	C	G	T
Δ	0	3	3	3	3
A	3	0	2	1	2
C	3	2	0	2	1
G	3	1	2	0	2
T	3	2	1	2	0

1. 1 文字目の G を A に置換 ($\delta(\mathtt{G},\mathtt{A}) = 1$)
   GCA → ACA
2. 2 文字目の C を G に置換 ($\delta(\mathtt{C},\mathtt{G}) = 2$)
   ACA → AGA
3. 3 文字目の A を T に置換 ($\delta(\mathtt{A},\mathtt{T}) = 2$)
   AGA → AGT

3.1.2　最適周期の定義

本研究の中核を成す概念である最適周期を定義する．以下では，文字列 x と文字列 y との距離を $d(x,y)$ で表すものとする．

定義 3(最適周期)

x を文字列，p をパターンとする．x を $p_1, p_2, \ldots, p_r$ に分割したとき，

$$\begin{cases} d(p,p_i) \leq t & (1 \leq i < r) \\ d(p',p_r) \leq t & (p' \text{は } p \text{ のプレフィックス}) \end{cases}$$

を満たすなら，p は x の t-**最適周期**(approximate period) といい，このときの t を**誤り**(error) という．

大まかに言うと，文字列 x をいくつかの部分列に分割し，分割した部分列とパターン p との距離を測る．測った距離の最大値が誤り t となる．ただし，x を分割した最後の部分列 p_r は，p のプレフィックス (p の先頭からの一部)p' との距離を測る．

例 3. 距離 d に編集距離を用いると，AGGATCACTAG は AGT の 2-最適周期となる．
($x =$)AGGATCACTAG を，($p_1 =$)AGG，($p_2 =$)ATC，($p_3 =$)ACT，($p_4 =$)AG に分割し，パターン ($p =$)AGT と各部分列との編集距離を求める．

- AGT と AGG との編集距離は 1.　($d(\mathtt{AGT},\mathtt{AGG}) = 1$)
- AGT と ATC との編集距離は 2.　($d(\mathtt{AGT},\mathtt{ATC}) = 2$)
- AGT と ACT との編集距離は 1.　($d(\mathtt{AGT},\mathtt{ACT}) = 1$)

- $(p' =)$AG と AG との編集距離は 0. $(d(\texttt{AG}, \texttt{AG}) = 0)$

となり，各距離は 2 以下なので，誤りは 2 となる．

例 4. 距離 d に重み付き編集距離 (ペナルティ行列は表 1) を用いると，AGGTAGACT は ACT の 5-最適周期となる．
$(x =)$AGGTAGACT を，$(p_1 =)$AGGT，$(p_2 =)$AG，$(p_3 =)$ACT に分割し，パターン $(p =)$ACT と各部分列との重み付き編集距離を求める．

- ACT と AGGT との重み付き編集距離は 5. $(d(\texttt{ACT}, \texttt{AGGT}) = 5)$
- ACT と AG との重み付き編集距離は 5. $(d(\texttt{ACT}, \texttt{AG}) = 5)$
- ACT と ACT との重み付き編集距離は 0. $(d(\texttt{ACT}, \texttt{ACT}) = 0)$

となり，各距離は 5 以下なので，誤りは 5 となる．

3.1.3 問題定義

最適周期に関する問題として，次の 2 つの問題が考えられる．なお，距離関数には重み付き編集距離を用いるので，いずれの問題でもペナルティ行列が入力として与えられているものとする．

(1) 誤り測定問題

誤り測定問題は，文字列とパターンが与えられたとき，それらの間の誤りを計算する問題である．この問題の入出力を，以下で定義する．

定義 4(誤り測定問題)

入力: 文字列 x，パターン p
出力: p が x の t-最適周期となるような誤り t

例 5. 文字列 AGGATCACTAG と パターン AGT を入力したとき，誤り 2 を出力する．(例 3)
例 3 では (重み付けされていない) 一般の編集距離を用いているが，これはペナルティ行列を，対角成分を 0，それ以外の要素を 1 と設定した場合の重み付き編集距離と等価である．

(2) パターン探索問題

パターン探索問題は，文字列が与えられたとき，誤りが最小となるようなパターンを求める問題である．しかし，この問題は NP-完全であることが知られている [1] ため，パターンが与えられた文字列の一部である，といった制限を加える．この問題の入出力を，以下で定義する．

定義 5(パターン探索問題)

入力: 文字列 x
出力: p が x の最小の t で t-最適周期となるような パターン p
(ただし，p は x の一部)

3.2 誤り測定アルゴリズム

疑似言語で記述した誤り測定アルゴリズム全体を表す手続き Measure_Error をアルゴリズム 1 に示す.

アルゴリズム 1 p が x の t-最適周期となる t を求める Measure_Error

```
Measure_Error (x, p)
1: w ← Compute_All_Distance(x, p)
2: t ← Minimum_Error(w, |x|)
3: return t
```

アルゴリズム 1 に示した通り，誤り測定アルゴリズムは，大まかに次の 2 つのステップに分かれる．

Step1. x の全ての部分列と p との距離を計算する
Step2. p が $x[1..i]$ の $t[i]$-最適周期となるような最小の $t[i]$ を $i = 1$ から n まで順に求める

なお，$x[1..i]$ は x の 1 文字目から i 文字目までの部分列を表す．また，後に述べるが，アルゴリズム中の w は 2 次元配列である．以下では，この 2 つのステップについて詳しく説明する．

3.2.1 距離計算

Step1 の計算を行う手続き Compute_All_Distance をアルゴリズム 2 に示す．ここで，$n := |x|$(x の長さ)，$m := |p|$(p の長さ) とする．このアルゴリズムの返値である w は，n 行 n 列の 2 次元配列で，i 行 j 列目の要素 $w[i, j]$ は $x[i..j]$ と p との距離を格納する．また，D は D テーブルと呼ばれる 2 次元配列で，文字列間の距離を格納する．

アルゴリズム 2 の中で，実際に距離の計算を行っているのは 3 行目の手続き Create_Dtable である．まず，手続き Create_Dtable をアルゴリズム 3 に示し，これについて説明する．

(1) D テーブルの構成

重み付き編集距離の計算は，D テーブル D を構成することによって行われる．D テーブルの構成は，アルゴリズム 3 を実行することにより成される．アルゴリズムの 2 行目から 8 行目は D テーブルの初期化のための処理であり，9 行目から 13 行目までの

$$D[i,j] = \min \left\{ \begin{array}{lcl} D[i-1,j] & + & \delta(x[i], \Delta) \\ D[i,j-1] & + & \delta(\Delta, y[j]) \\ D[i-1,j-1] & + & \delta(x[i], y[j]) \end{array} \right\} \ (1 \leq \forall i \leq n,\ 1 \leq \forall j \leq m) \qquad (1)$$

を計算している部分が中心となる処理である．これは，各位置での削除・挿入・置換のコストを求め，それらの最小値を要素とする処理である．

例 2 の計算を行う D テーブルは，表 2 のようになる．ここでは，D テーブルの最終行・最終列の値 $D[4, 3] = 5$ が求める距離 5 となる．D テーブルの各要素は，文字列の部分列間の距離を表して

アルゴリズム 2　x の全ての部分列と p との距離を計算する `Compute_All_Distance`

`Compute_All_Distance`(x, p)

1: $n \leftarrow |x|$, $m \leftarrow |p|$
2: **for** $i = 1$ to n **do**
3:　$D \leftarrow$ `Create_Dtable`($x[i..n]$, p)
4:　**for** $j = 1$ to n **do**
5:　　**if** $j = n$ **then**
6:　　$w[i, j] \leftarrow \min_{0 \le h \le m} (D[j, h])$
7:　　**else**
8:　　$w[i, j] \leftarrow D[j, m]$
9:　　**end if**
10:　**end for**
11: **end for**
12: **return** w

アルゴリズム 3　x と y との重み付き編集距離を計算する `Create_Dtable`

`Create_Dtable`(x, y)

1: $n \leftarrow |x|$, $m \leftarrow |y|$
2: $D[0, 0] \leftarrow 0$
3: **for** $i = 1$ to n **do**
4:　$D[i, 0] \leftarrow D[i-1, 0] + \delta(x[i], \Delta)$
5: **end for**
6: **for** $j = 1$ to m **do**
7:　$D[0, j] \leftarrow D[0, j-1] + \delta(\Delta, y[j])$
8: **end for**
9: **for** $i = 1$ to n **do**
10:　**for** $j = 1$ to m **do**
11:　$$D[i, j] \leftarrow \min \left\{ \begin{array}{lcl} D[i-1, j] & + & \delta(x[i], \Delta) \\ D[i, j-1] & + & \delta(\Delta, y[j]) \\ D[i-1, j-1] & + & \delta(x[i], y[j]) \end{array} \right\}$$
12:　**end for**
13: **end for**
14: **return** D

表 2　例 2 の場合の D テーブル

	Δ	G	C	A
Δ	0	3	6	9
A	3	1	4	6
G	6	3	3	5
T	9	6	4	5

いる．例えば，表 2 の $D[3,3](=3)$ は，文字列 GC と AG との重み付き編集距離を表している．この特徴はとても有効で，ある文字列間の距離を求めるとその文字列の部分列も同時に求められるため，計算量の節約となる．この例だと，GCT と AGT との重み付き編集距離を求めると，同時に GC と AGT との重み付き編集距離 $(D[2,3]=4)$ も求まるのである．

D テーブルの構成が完了すると，Create_Dtable は構成した D テーブルそのものを返す．

(2)　D テープルから距離を取り出す

Compute_All_Distance(アルゴリズム 2) は，文字列 x の開始位置を i によって指定し，この i から始まる x の部分列 $x[i..n]$ を Create_Dtable に送り，パターン p との距離を計算する．Create_Dtable は D テーブルそのものを返してくるので，そこから距離を取り出す必要がある．

基本的には，D テーブルの最小列の要素を順に取り出して行けば，x の i から始まる部分列と p との距離が取り出せる (8 行目)．ただし，D テーブルの最小行の要素だけは，その行の最小値を取り出す必要がある (6 行目)．これは，最適周期の定義において，“文字列 x を分割した最後の部分列 p_r はパターンの一部 p' との距離を測る”，としたことによる．

例えば，$x[i..n]$ が “GCA”，p が “AGT” のとき (すなわち，表 2 のとき)，$w[i,j]$ を $1 \leq \forall j \leq n$ について書き出すと，$(6,4,5)$ となる．

Compute_All_Distance は，以上の手順で構成した w をそのまま返す．

3.2.2　誤り最小化

Step2 の計算を行う手続き Minimum_Error をアルゴリズム 4 に示す．ここでの $t[i]$ は，$x[1..i]$ が p の $t[i]$-最適周期となるような誤り $t[i]$ のことである．

アルゴリズム 4　距離 w を用いて誤り t を求める Minimum_Error

Minimum_Error $(w,\ n)$

1: $t[0] \leftarrow 0$
2: **for** $i = 1$ to n **do**
3: $\quad t[i] \leftarrow \min_{0 \leq h < i} (\max\{t[h], w[h+1, i]\})$
4: **end for**
5: **return** $t[n]$

このアルゴリズム中の 2 行目から 4 行目では,

$$t[i] = \min_{0 \le h < i} (\max\{t[h], w[h+1, i]\}) \ (1 \le \forall i \le n) \tag{2}$$

を計算している．h 文字目までの，誤りになり得る値の候補を順に求めていき，その中で最小のものを $t[i]$ として採用する手法である．

Minimum_Error は $t[n]$ を返す．$t[n]$ は p が $x[1..n](= x)$ の $t[n]$-最適周期となる誤りであるため，所望の t となる．

3.2.3 計算量

$n := |x|$(文字列の長さ)，$m := |p|$(パターン長) とすると，このアルゴリズム全体の時間計算量は $O(mn^2)$ である．まず，D テーブルの大きさは最大 $(n+1) \times (m+1)$ であるので，1 つの D テーブルを構成するために $O(mn)$ の時間計算量が掛かる．D テーブルは合計 n 回構成されるので，距離計算に掛かる時間計算量は $O(mn^2)$ である．誤り最小化に掛かる時間計算量は，アルゴリズム 4 の 3 行目で $O(n)$，それが n 回繰り返されているので全体で $O(n^2)$ である．よって，アルゴリズム全体の時間計算量は $O(mn^2)$ である．

3.3 パターン探索アルゴリズム

入力文字列 x の中からパターンを探索するには，x から部分列をひとつ取り出し，それをパターン候補 p_c として x との誤り測定を行い，全てのパターン候補の中から誤りが最小のものをパターン p として採用すればよい．疑似言語で記述したパターン探索アルゴリズム全体を表す手続き Search_Pattern をアルゴリズム 5 に示す.

まず，パターン候補 p_c を x の部分列の中から設定する．i はパターン候補の開始位置であり，m はパターン候補の長さである．つまり，p_c は $x[i..i+m-1]$ に設定される (5 行目)．T は最小の誤りである．文字列 x とパターン候補 p_c との距離 t を計算し，t が T を下回っていれば，パターン候補 p_c をパターンとして採用する (8 行目から 11 行目).

パターン候補を設定し，誤りとパターンを更新する以外は，誤り測定アルゴリズムとほとんど変わらないが，時間計算量を節約するため少し工夫がなされている．それを以下で説明する．

3.3.1 距離計算

アルゴリズム 2 の Compute_All_Distance をこの場合でもそのまま使うと，$n \times n$ 回 D テーブルを構成する必要がある．しかし，パターン候補 p_c が前の内容に 1 文字加わっただけなら，新たに D テーブルを構成し直す必要は無い．例えば，表 2 において Δ，A，G の列の計算が終わっていたとしよう．ここで，列に T が加わったとしても D テーブルを一から構成し直す必要は無く，元の D テーブルに T の列だけ計算をし付け加えればいいのである．ただし，この方法を用いると以前に計算した D テーブルの内容を保持しておく必要があるため，空間計算量が多くなってしまうが，時間計算量は n の次数が 1 つ減るので大変有効である．この方法を用いて記述した手続き Compute_All_Distance をアルゴリズム 6 に示す.

アルゴリズム 5 文字列 x からパターン p を探索する `Search_Pattern`

`Search_Pattern` (x)

1: $T \leftarrow \infty$
2: $n \leftarrow |x|$
3: **for** $i = 1$ to n **do**
4: **for** $m = 1$ to $n - i + 1$ **do**
5: $p_c \leftarrow x[i..i + m - 1]$
6: $w \leftarrow$ `Compute_All_Distance`(x, p_c)
7: $t \leftarrow$ `Minimum_Error`$(w, |x|)$
8: **if** $t < T$ **then**
9: $T \leftarrow t$
10: $p \leftarrow p_c$
11: **end if**
12: **end for**
13: **end for**
14: **return** p

アルゴリズム 6 x の全ての部分列と p との距離を計算する `Compute_All_Distance`

`Compute_All_Distance`(x, p)

1: $n \leftarrow |x|, m \leftarrow |p|$
2: **for** $i = 1$ to n **do**
3: **if** $m = 1$ **then**
4: $D \leftarrow$ `Initialize_Dtable`$(D, x[i..n], i)$
5: **end if**
6: $D \leftarrow$ `Add_Dtable`$(D, x[i..n], p, i)$
7: **for** $j = 1$ to n **do**
8: **if** $j = n$ **then**
9: $w[i, j] \leftarrow \min_{0 \le h \le m} (D[i, j, h])$
10: **else**
11: $w[i, j] \leftarrow D[i, j, m]$
12: **end if**
13: **end for**
14: **end for**
15: **return** w

誤り測定アルゴリズムと異なり，D テーブルは 3 次元配列となっている．これは先程計算した D テーブルの内容を保持するためのもので，この 3 次元配列は処理が他の手続きに移っても内容を保持しておかなければいけない．なお，3 次元配列の要素 $D[i,j,k]$ には，i を先頭とする x の部分列 $x[i..n]$ とパターン候補 p_c との距離を計算するための D テーブル $D[j,k]$ が格納されている．

次に，D テーブルを操作する手続きである，`Initialize_Dtable` と `Add_Dtable` について説明する．

(1) D テーブルの初期化

D テーブルを初期化するための手続き `Initialize_Dtable` をアルゴリズム 7 に示す．

アルゴリズム 7　D テーブル D を初期化する `Initialize_Dtable`

`Initialize_Dtable`(D, x, i)

1: $n \leftarrow |x|$
2: $D[i,0,0] \leftarrow 0$
3: **for** $k = 1$ to n **do**
4: $\quad D[i,k,0] \leftarrow D[i,k-1,0] + \delta(x[k], \Delta)$
5: **end for**
6: **return** D

$D[i,0,0]$ から $D[i,n,0]$ までの初期化を行っている．以降の列の初期化は，現時点でのパターン長が不明なので行わない．

(2) D テーブルの追加

D テーブルに 1 列を追加するための手続き `Add_Dtable` をアルゴリズム 8 に示す．

アルゴリズム 8　D テーブル D に 1 列を追加する `Add_Dtable`

`Add_Dtable`(D, x, y, i)

1: $n \leftarrow |x|$, $m \leftarrow |y|$
2: $D[i,0,m] \leftarrow D[i,0,m-1] + \delta(\Delta, y[m])$
3: **for** $k = 1$ to n **do**
4: $\quad D[i,k,m] \leftarrow \min \left\{ \begin{array}{lcl} D[i,k-1,m] & + & \delta(x[k], \Delta) \\ D[i,k,m-1] & + & \delta(\Delta, y[m]) \\ D[i,k-1,m-1] & + & \delta(x[k], y[m]) \end{array} \right\}$
5: **end for**
6: **return** D

$D[i,0,m]$ から $D[i,n,m]$ までの計算を行っている．計算には，式 (1) を用いている．

3.3.2 誤り最小化

この手続きは，誤り測定アルゴリズムのときと変わらない．アルゴリズム 4 に示した手続き `Minimum_Error` をそのまま用いればよい．

3.3.3 計算量

$n := |x|$(文字列の長さ) とすると，このアルゴリズム全体の時間計算量は $\mathrm{O}(n^4)$ である．まず，上に示した手法を用いることにより，D テーブルを構成する回数は n 回となった．D テーブルの大きさは最大 $(n+1) \times (n+1)$ であるので，1 つの D テーブルを構成するために $\mathrm{O}(n^2)$ の時間計算量が掛かる．つまり，距離計算に掛かる時間計算量は $\mathrm{O}(n^3)$ である．誤り最小化に掛かる時間計算量は，誤り測定アルゴリズムのときと同じく $\mathrm{O}(n^2)$ である．これに，パターン候補 p_c の n 通りの開始位置が掛かるので，アルゴリズム全体の時間計算量は $\mathrm{O}(n^4)$ である．

4　誤り測定アルゴリズムの並列化と検証

4.1　並列化アルゴリズム

3.2 節で説明した誤り測定アルゴリズムを並列化する．各プロセッサには，誤り計算において取り出す文字列 x の部分列を割り当て，並列に距離を計算する．

4.1.1　並列化手法

Compute_All_Distance(アルゴリズム 2) の 2 行目の i を各プロセッサに割り当てる．この i は，文字列 x の部分列 $x[i..n]$ の先頭位置である．この手法で並列化を行った Compute_All_Distance をアルゴリズム 9 に示す．

アルゴリズム 9　x の全ての部分列と p との距離を計算する Compute_All_Distance

Compute_All_Distance(x, p)
(w は全プロセッサで共有)

```
1:  n ← |x|, m ← |p|
2:  par i: 1 ≤ i ≤ n do
3:    D ← Create_Dtable(x[i..n], p)
4:    for j = 1 to n do
5:      if j = n then
6:        w[i, j] ← min_{0≤h≤m} (D[j, h])
7:      else
8:        w[i, j] ← D[j, m]
9:      end if
10:   end for
11: end par
12: return w
```

赤字で表した 2 行目と 11 行目との間が並列に計算を行う部分である．1 から n までの i を割り当てられた各プロセッサは，$x[i..n]$ とパターン p との距離を計算する．その距離を共有メモリにある w に格納する．MPI などの分散メモリ環境で実装する場合は，部分列の開始位置 i を各プロセッサで割り出し，距離計算を行った後，計算した距離をマスターのプロセッサに送信すればよい．

各プロセッサには，最大 $\lfloor n/N \rfloor$ 個の i が割り当てられる (N はプロセッサ数)．負荷が均衡するように，プロセッサへの割り当てはサイクリックに行う．例えば，x が AGGTAGACT，プロセッサ数が 4 のとき，各プロセッサに割り当てられる i と x の部分列 ($x[i..9]$) は表 3 のようになる．

表 3　x = AGGTAGACCT，プロセッサ数 4 のときの i と $x[i..9]$ の割り当て

プロセッサ番号	割り当てる i(部分列 $x[i..9]$)		
0	1(AGGTAGACT)	8(CT)	9(T)
1	2(GGTAGACT)	7(ACT)	
2	3(GTAGACT)	6(GACT)	
3	4(TAGACT)	5(AGACT)	

4.1.2　計算量

逐次アルゴリズムの場合と同様，$n := |x|$(文字列の長さ)，$m := |p|$(パターン長) とする．各プロセッサにアルゴリズム 9 の i を 1 個ずつ割り当てたとすると，必要なプロセッサ数は $\mathrm{O}(n)$ である．各プロセッサに割り当てる i は $\mathrm{O}(\log n)$ 個であるから，実際に必要なプロセッサ数は $\mathrm{O}(n/\log n)$ である．$\mathrm{O}(n/\log n)$ 個のプロセッサを用いると，距離計算 (Compute_All_Distance，アルゴリズム 9) の時間計算量は，1 つの D テーブル構成に掛かる時間計算量が $\mathrm{O}(mn)$ であるので，$\mathrm{O}(mn\log n)$ となる．誤り最小化 (Minimum_Error，アルゴリズム 4) に掛かる時間計算量は $\mathrm{O}(n^2)$ であるので，アルゴリズム全体の時間計算量は $\mathrm{O}(mn\log n)$ である．

4.2　実験

(1)　実験条件

4.1 節で述べたアルゴリズムを，MPI で実装し，Raptor クラスタ・Diplo クラスタ上でそれぞれ実験を行った．入力データとして，アルファベット $\{\Delta, \mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}\}$ 上の文字列 x と，A が任意回繰り返されたパターン p を用いた．文字列 x の長さ n は $\{1024, 2048, 4096, 8192, 16384\}$ の 5 通り，パターン p の長さ m は $\{30, 60\}$ の 2 通り，合計 10 通りとして実験を行った．

(2)　実験結果

Raptor クラスタで行った実験においてのプログラムの実行時間を，$m = 30$ の場合は表 4 に，$m = 60$ の場合は表 5 に，プロセッサ数 8 のときの速度向上比を表 8 にそれぞれ示す．同様に，Diplo クラスタで行った実験においてのプログラムの実行時間を，$m = 30$ の場合は表 6 に，$m = 60$ の場合は表 7 に，プロセッサ数 16 のときの速度向上比を表 9 にそれぞれ示す．なお，速度向上比の算出に当たっては，プロセッサ数 1 のときの実行時間を 1 とした．

また，

- $n = 4096$, $m = 30$
- $n = 8192$, $m = 30$
- $n = 4096$, $m = 60$
- $n = 8192$, $m = 60$

表 4　実行時間 (誤り測定問題 ($m = 30$)，Raptor クラスタ)

プロセッサ数	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$
1	0.48	1.92	8.21	37.98	158.36
2	0.35	1.47	6.14	27.81	112.33
4	0.22	0.81	3.50	19.47	78.63
8	0.15	0.54	2.30	15.38	61.40

(単位：秒)

表 5　実行時間 (誤り測定問題 ($m = 60$)，Raptor クラスタ)

プロセッサ数	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$
1	0.93	3.74	15.63	68.67	279.77
2	0.59	2.42	9.99	43.03	172.03
4	0.32	1.30	5.41	27.15	107.53
8	0.21	0.78	3.25	19.31	76.28

(単位：秒)

表 6　実行時間 (誤り測定問題 ($m = 30$)，Diplo クラスタ)

プロセッサ数	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$
1	0.56	2.25	8.98	36.19	152.29
2	0.31	1.22	4.80	19.51	85.74
4	0.18	0.69	2.67	10.95	50.42
8	0.17	0.45	1.65	6.82	33.18
16	0.10	0.34	1.15	3.83	24.62

(単位：秒)

の 4 通りの入力データにおける速度向上を，Raptor クラスタの場合は図 3 に，Diplo クラスタの場合は図 4 にそれぞれ示す.

4.3　考察

全体として，Raptor クラスタ (8 プロセッサ) で最大約 4.8 倍 ($n = 2048$ 及び $n = 4096$，$m = 60$)，Diplo クラスタ (16 プロセッサ) で最大約 11.2 倍 ($n = 8192$，$m = 60$) の速度向上を得ることができた.

表 7　実行時間 (誤り測定問題 ($m = 60$)，Diplo クラスタ)

プロセッサ数	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$
1	1.10	4.37	17.41	70.44	294.97
2	0.58	2.27	9.00	36.82	161.15
4	0.32	1.22	4.79	19.68	88.25
8	0.22	0.71	2.70	10.04	52.47
16	0.20	0.48	1.69	6.27	34.27

(単位：秒)

表 8　プロセッサ数 8 のときの速度向上比 (誤り測定問題，Raptor クラスタ)

	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$
$m = 30$	3.21	3.54	3.57	2.47	2.58
$m = 60$	4.36	4.82	4.81	3.56	3.67

(プロセッサ数 1 のとき 1)

表 9　プロセッサ数 16 のときの速度向上比 (誤り測定問題，Diplo クラスタ)

	$n = 1024$	$n = 2048$	$n = 4096$	$n = 8192$	$n = 16384$
$m = 30$	5.81	6.40	7.79	9.45	6.19
$m = 60$	5.60	9.06	10.32	11.23	8.61

(プロセッサ数 1 のとき 1)

(1)　Raptor クラスタにおける実行結果

まず，n が同じであれば，m が大きいほど速度向上が成されている．これは，m の大きさは並列化に直接関係せず，そのために m が大きくなると各プロセッサに割り当てられる処理の量が大きくなり，m が小さいときより負荷が均衡したためと考えられる．

また，$m = 30$ のときと $m = 60$ のときの両方で，$n = 2048$，$n = 4096$ で速度向上が最大となっており，$n = 8192$ で速度向上が落ちている．ここで，$n = 2048$ と $n = 4096$ を比較すると，速度向上はほぼ横ばいである．つまり，$n = 4096$ 近辺で速度向上が止まっていると考えるのが妥当である．この $n = 4092$ 近辺で各プロセッサに割り当てられる処理が大きくなることにより起こる，負荷均衡による速度向上に対して，通信に掛かるオーバーヘッドが無視できない程大きくなっていることが考えられる．

このことは，図 3 からも読み取れる．当然，プロセッサ数が増えるごとに通信量は増大し，そのオーバーヘッドによって速度向上は低下する．ここで，“$n = 4096$，$m = 30$”(赤線) と “$n = 8192$，$m = 30$”(緑線) を比べると，プロセッサ数が 4 から 8 へと変化した時点で，その差が大きく離れていることがわかる．この現象は，“$m = 4096$，$m = 60$”(青線) と “$m = 8192$，$m = 60$”(ピンク色

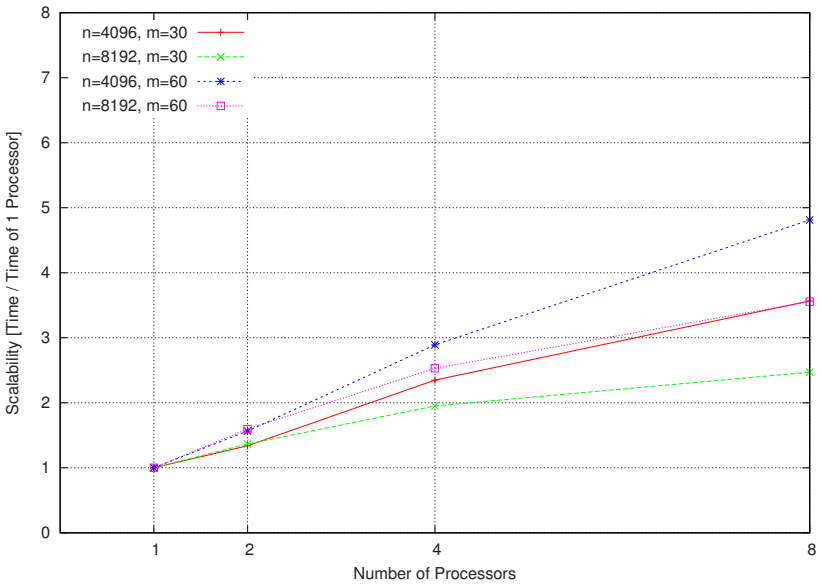

図 3　速度向上 (誤り測定問題，Raptor クラスタ)

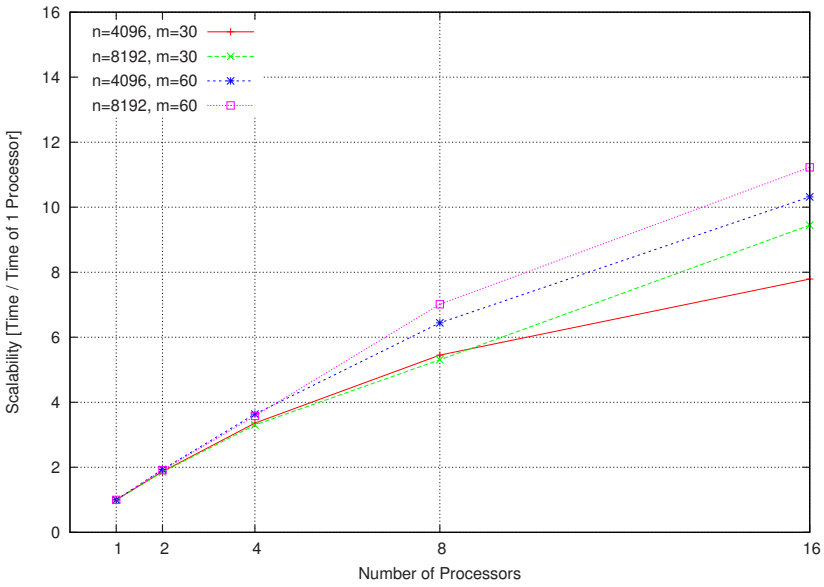

図 4　速度向上 (誤り測定問題，Diplo クラスタ)

の線) との間にも見られる．これは，$n = 4096$ のときの通信量より，$n = 8192$ のときの通信量が増大していることを示しており，プロセッサ数が増える際に発生する，これら 2 つの入力データの速度向上の差が通信によるオーバーヘッドによるものだと考えられる．

(2) Diplo クラスタにおける実行結果

Raptor クラスタと同じく，n が同じであれば，m が大きいほど速度向上が成されている．その原因も，Raptor クラスタの場合と同じで，負荷が均衡したためと考えられる．

また，$m = 30$，$m = 60$ の両方で，$n = 8192$ で速度向上が最大となっており，$n = 16384$ では速度向上が落ちている．これも，Raptor クラスタの場合と同じで，$n = 8192$ 又は $n = 16384$ 近辺で通信に掛かるオーバーヘッドが大きくなったためと考えられる．Raptor クラスタと Diplo クラスタで，速度向上が止まる地点，すなわち通信量が無視できなくなるほど大きくなる地点が異なるのは，それぞれのクラスタを構成するネットワークの速度が異なるためだと考えられる．つまり，Diplo クラスタを構成するネットワークは，Raptor クラスタを構成するネットワークより高速なので，速度向上が止まるときの n が大きくなったと考えられる．

5　パターン探索アルゴリズムの並列化と検証

5.1　単純な並列化アルゴリズム

3.3 節で説明したパターン探索アルゴリズムを並列化する．各プロセッサには異なったパターン候補 p_c を割り当て，文字列 x と割り当てられた p_c との誤りを並列に計算する．

5.1.1　並列化手法

Search_Pattern(アルゴリズム 5) の 3 行目の i を各プロセッサに割り当てる．この i は，パターン候補 $p_c = x[i..m]$ の先頭位置である．この手法で並列化を行った Search_Pattern をアルゴリズム 10 に示す．

アルゴリズム 10　文字列 x からパターン p を探索する Search_Pattern

```
Search_Pattern (x)
(T, p は全プロセッサで共有)
 1: T ← ∞
 2: n ← |x|
 3: par i: 1 ≤ i ≤ n do
 4:    for m = 1 to n − i + 1 do
 5:       p_c ← x[i..i + m − 1]
 6:       w ← Compute_All_Distance(x, p_c)
 7:       t ← Minimum_Error(w, |x|)
 8:       if t < T then
 9:          T ← t
10:          p ← p_c
11:       end if
12:    end for
13: end par
14: return p
```

赤字で表した 3 行目と 13 行目との間が並列に計算を行う部分である．1 から n までの i を割り当てられた各プロセッサは，x とパターン候補 $p_c(= x[i..m])$ との誤り t を計算する．計算した t は，共有メモリにある T と比較され，$t < T$ なら共有メモリの T と p が更新される．MPI などの分散メモリ環境で実装する場合は，パターン候補の開始位置 i を各プロセッサで割り出し，距離計算を行った後，計算した最小の誤りとパターン候補をマスターのプロセッサに送信すればよい．

各プロセッサには，最大 $\lfloor n/N \rfloor$ 個の i が割り当てられる (N はプロセッサ数)．負荷が均衡するように，誤り測定アルゴリズムのときと同様，プロセッサへの割り当てはサイクリックに行う．例えば，x が AGGTAGACT，プロセッサ数が 4 のとき，各プロセッサに割り当てられる i と p_c は表 10 の

表 10　x = AGGTAGACCT，プロセッサ数 4 のときの i と p_c の割り当て

プロセッサ番号	割り当てる i(パターン候補 p_c)		
0	1(A, AG, ..., AGGTAGACT)	8(C, CT)	9(T)
1	2(G, GG, ..., GGTAGACT)	7(A, AC, ACT)	
2	3(G, GT, ..., GTAGACT)	6(G, GA, ..., GACT)	
3	4(T, TA, ..., TAGACT)	5(A, AG, ..., AGACT)	

ようになる．

5.1.2　計算量

逐次アルゴリズムの場合と同様，$n := |x|$(文字列の長さ) とする．各プロセッサにアルゴリズム 10 の i を 1 個ずつ割り当てたとすると，必要なプロセッサ数は $\mathrm{O}(n)$ である．各プロセッサに割り当てる i は $\mathrm{O}(\log n)$ 個であるから，実際に必要なプロセッサ数は $\mathrm{O}(n/\log n)$ である．$\mathrm{O}(n/\log n)$ 個のプロセッサを用いると，アルゴリズム全体 (Search_Pattern，アルゴリズム 10) の時間計算量は，1 つのパターン候補 p_c に掛かる時間計算量が $\mathrm{O}(n^3)$ であるので，$\mathrm{O}(n^3 \log n)$ である．

5.2　ハイブリッド並列化アルゴリズム

ハイブリッド並列化など 2 段階で並列化を行う場合には，単純に並列化を行うときと同様，最上位の各プロセッサにパターン候補 p_c を割り当てた後，次の階層の各プロセッサに誤り計算において取り出す文字列 x の部分列を割り当てる．

5.2.1　並列化手法

5.1 節で示したパターン候補 p_c による並列化を行った上で，誤り測定の並列化も行うことができる．手法は 4.1.1 節で示したものとほぼ同じで，Compute_All_Distance(アルゴリズム 6) の 2 行目の i を各プロセッサに割り当てる．この i は，文字列 x の部分列 $x[i..n]$ の先頭位置である．この手法で並列化を行った Compute_All_Distance をアルゴリズム 11 に示す．

赤字で表した 3 行目と 14 行目との間が並列に計算を行う部分である．基本的な並列化手法はアルゴリズム 9 と同じであるが，D テーブル D を全プロセッサで共有するのが大きな違いである．これは，手続き Add_Dtable で以前の D テーブルの結果を用いるので，全プロセッサで統一した D テーブルの内容を保持しておく必要があるためである．

なお，プロセッサへの割り当ては表 3 のようにサイクリックに行う．

5.2.2　計算量

4.1.2 節と 5.1.2 節で行った議論を組み合わせればよい．必要なプロセッサ数は，最上位のプロセッサが $\mathrm{O}(n/\log n)$ 個，次の階層のプロセッサが $\mathrm{O}(n/\log n)$ 個である．距離計算 (Compute_All_Distance，アルゴリズム 11) に掛かる時間計算量は $\mathrm{O}(n^2 \log n)$ である．これ

アルゴリズム 11　x の全ての部分列と p との距離を計算する Compute_All_Distance

```
Compute_All_Distance(x, p)
(D, w は全プロセッサで共有)
 1: n ← |x|, m ← |p|
 2: par i: 1 ≤ i ≤ n do
 3:   if m = 1 then
 4:     D ← Initialize_Dtable(D, x[i..n], i)
 5:   end if
 6:   D ← Add_Dtable(D, x[i..n], p, i)
 7:   for j = 1 to n do
 8:     if j = n then
 9:       w[i, j] ← min_{0≤h≤m} (D[i, j, h])
10:     else
11:       w[i, j] ← D[i, j, m]
12:     end if
13:   end for
14: end par
15: return w
```

に，パターン候補 p_c の $\log n$ 通りの開始位置が掛かるので，アルゴリズム全体の時間計算量は $\mathrm{O}(n^2 \log^2 n)$ である．

5.3　実験

(1)　実験条件

5.1 節で述べたアルゴリズムを MPI で実装し，Raptor クラスタ・Diplo クラスタ上でそれぞれ実験を行った．なお，Diplo クラスタ上で実装する際は，5.2 節で述べたアルゴリズムにより，ハイブリッド並列化も行った．また，ハイブリッド並列化を行う際には，OpenMP のスレッド数を 4 で固定した．実験データとして，アルファベット $\{\Delta, \mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}\}$ 上の文字列 x を用いた．文字列 x の長さ n は $\{128, 256, 512\}$ の 3 通りとして実験を行った．

(2)　実験結果

Raptor クラスタで行った実験においてのプログラムの実行時間を表 11 に，プロセッサ数 8 のときの速度向上比を表 13 にそれぞれ示す．同様に，Diplo クラスタで行った実験においてのプログラムの実行時間を表 12 に，プロセッサ数 16 のときの速度向上比を図 14 にそれぞれ示す．なお，速度向上比の算出に当たっては，プロセッサ数 1 のときの実行時間を 1 とした．ハイブリッド並列化の場合は，MPI による並列化でプロセッサ数 1 のときの実行時間を 1 としている．

また，この入力データにおける速度向上を，Raptor クラスタの場合は図 5 に，Diplo クラスタの

場合は図 6 にそれぞれ示す.

表 11　実行時間 (パターン探索問題，Raptor クラスタ)

プロセッサ数	$n = 128$	$n = 256$	$n = 512$
1	6.66	110.56	1975.84
2	4.22	73.58	1146.80
4	2.32	45.05	631.99
8	1.26	22.62	316.58

(単位：秒)

表 12　実行時間 (パターン探索問題，Diplo クラスタ)

	MPI による並列化			ハイブリッド並列化			
プロセッサ数	$n = 128$	$n = 256$	$n = 512$	MPI スレッド数	$n = 128$	$n = 256$	$n = 512$
1	3.32	110.74	1850.91				
2	1.85	82.32	1359.07				
4	0.97	41.45	683.40	1	1.72	64.60	1082.80
8	0.60	26.77	408.81	2	0.91	32.37	542.76
16	0.39	13.50	205.43	4	0.51	16.23	271.08

(単位：秒)

表 13　プロセッサ数 8 のときの速度向上比 (パターン探索問題，Raptor クラスタ)

$n = 128$	$n = 256$	$n = 512$
5.28	4.89	6.24

(プロセッサ数 1 のとき 1)

5.4　考察

全体として，Raptor クラスタ (8 プロセッサ) で最大約 6.2 倍 ($n = 512$)，Diplo クラスタ (16 プロセッサ) で，MPI による並列化を行った場合は最大約 9.0 倍 ($n = 512$)，ハイブリッド並列化を行った場合は最大約 6.8 倍 ($n = 256$ 及び $n = 512$) の速度向上を得ることができた.

(1)　Raptor クラスタにおける実験結果

全体として，ノード数が変わってもほとんど速度向上の伸び率は変わらず，線型的な速度向上が得られていることが，図 5 から読み取れる．$n = 128$ のときに比べ，$n = 256$ のときの速度向上は

表 14　プロセッサ数 16 のときの速度向上比 (パターン探索問題，Diplo クラスタ)

	$n = 128$	$n = 256$	$n = 512$
MPI による並列化	8.55	8.20	9.01
ハイブリッド並列化	6.52	6.82	6.83

(プロセッサ数 1 のとき 1)

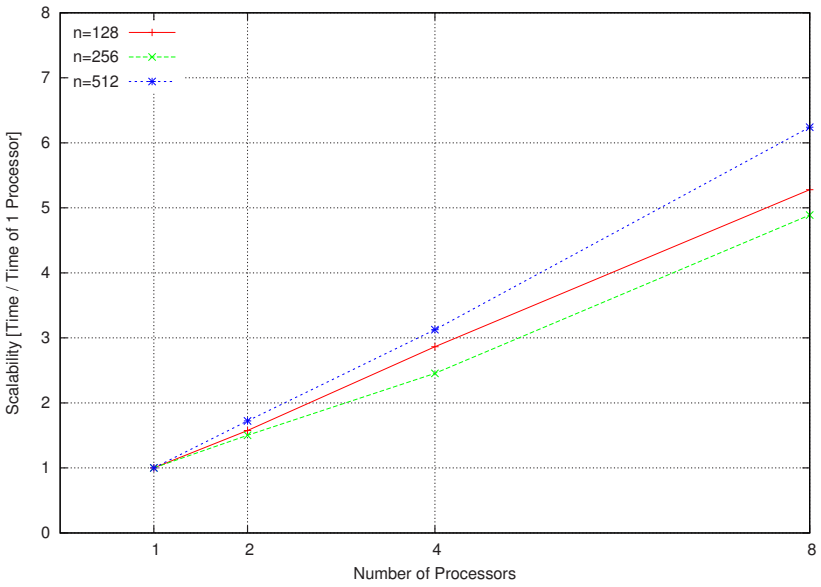

図 5　速度向上 (パターン探索問題，Raptor クラスタ)

伸び悩んでいるが，$n = 512$ になると最大の速度向上が得られている．

$n = 512$ のとき最大の速度向上が得られているのは，これは各プロセッサに割り当てられる処理の量が大きくなり，負荷が均衡し，なおかつ通信によるオーバーヘッドが大きくならないためだと考えられる．逆に，$n = 256$ のとき速度向上が伸び悩んでいるのは，各プロセッサに割り当てられる処理の量は大きいが，負荷が均衡する程の大きさでは無かった，ということが考えられる．

(2)　Diplo クラスタにおける実験結果 (MPI による並列化)

Raptor クラスタの場合より多少歪であるが，全体としては線型的な速度向上が得られていることが，図 6 から読み取れる．16 プロセッサの場合は，Raptor クラスタと同じく，$n = 256$，$n = 128$，$n = 512$ の順でよりよい速度向上が得られている．これは Raptor クラスタの場合と同じく，負荷均衡が原因となっていると考えられる．

Diplo クラスタの場合では，8 プロセッサまで $n = 128$ が最大の速度向上となっている．これは，通信によるオーバーヘッドが Raptor クラスタよりも小さいことが原因だと考えられる．

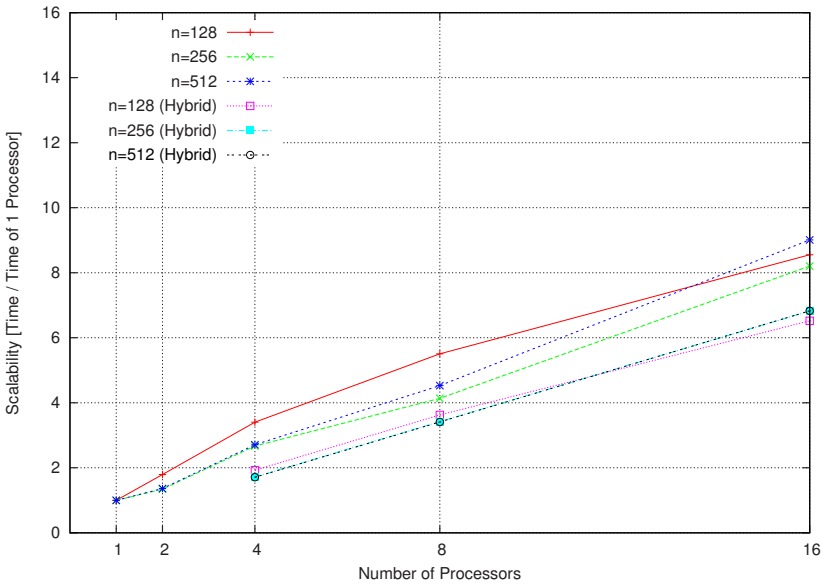

図 6　速度向上 (パターン探索問題，Diplo クラスタ)

(3)　Diplo クラスタにおける実験結果 (ハイブリッド並列化)

ハイブリッド並列化を行ったプログラムは，実行時間が MPI による並列化を行ったプログラムよりも遅く，速度向上も MPI によるプログラムには及ばなかった．これは，パターン探索問題とハイブリッド並列化との相性が悪かったためと考えられる．一般に，並列化を行い高い効果を得られる問題の中にも，ハイブリッド並列化と相性が悪く，ハイブリッド並列化によって逆に性能が下がってしまう問題は存在する [12]．今回の実験により，この問題もそのような問題の 1 つであることが判明した．

また，ハイブリッド並列化を行った場合，入力データの違いによって速度向上があまり変化しなかった．図 6 の $n = 128$(ピンク色の線) と $n = 256$(スカイブルーの線) と $n = 512$(黒線) が，ほとんど重なっていることからもこのことが読み取れる．これは，OpenMP による並列化で負荷が均等に配分されたことにより，ハイブリッド並列化における速度向上が，どのような入力データに対しても最大となるためであると考えられる．

6 おわりに

本研究では，文字列の最適周期に関する誤り測定問題とパターン探索問題を取り上げ，それらに対するアルゴリズムを並列化した．並列化したアルゴリズムは，SCore 型クラスタである Raptor(8 プロセッサ) と Beowulf 型の SMP クラスタである Diplo(16 プロセッサ) 上でそれぞれ実装した．実装には MPI を用い，特に Diplo クラスタ上でパターン探索問題の実装には，ノード間を MPI，ノード内を OpenMP でそれぞれ並列化する，ハイブリッド並列化の手法を用いた．

誤り測定問題では，Raptor クラスタで最大約 4.8 倍，Diplo クラスタで最大約 11.2 倍の速度向上を得た．また，パターン探索問題では，Raptor クラスタで最大約 6.2 倍，Diplo クラスタで最大約 9.0 倍，Diplo クラスタでハイブリッド並列化を行った場合には最大約 6.8 倍の速度向上を得た．MPI による並列化の効果に比べて，ハイブリッド並列化の効果が上がらなかったことから，パターン探索問題はハイブリッド並列処理に適さない問題であることが確認できた．

今後の研究課題として，まず，4 章・5 章で考察した内容を検証することが挙げられる．また，今回取り上げたパターン探索問題では，パターンは入力文字列の一部であるという制限を設けている．もう 1 つの研究課題として，この制限を外した一般的なパターン探索問題に対して，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．一般的なパターン探索問題は，NP-完全であることが証明されているため，その解決に当たっては最適化アルゴリズムや乱択アルゴリズムなどの手法を用いることが不可欠となる．

謝辞

本研究の機会を与えて下さり，貴重な助言，ご指導を頂きました山崎勝弘教授に深く感謝致します．また，主に PC クラスタの使用法について，貴重な助言を頂きました Truong Vinh Truong Duy 氏に深く感謝致します．

最後に，様々な貴重な助言を与えて下さった，松崎裕樹氏，中谷嵩之氏，井ノ口春寿氏，梅原直人氏をはじめとする高性能計算研究室の皆様方に心より感謝致します．

参考文献

[1] J. S. Sim, C. S. Ilipoulos, K. Park and W. F. Smyth: Approximate periods of strings, Theoretical Computer Science, Vol.262, No.1, p.p.557-568, 2001.

[2] M. Christodoulakis, C. S. Iliopoulos, K. Park and J. S. Sim: Implementing Approximate Regularities, Mathematical and Computer Modelling, Vol.42, No.7-8, p.p.855-866, 2005.

[3] 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.

[4] 三木 光範他: PC クラスタ超入門 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.

[5] 吉川 茂洋他: SMP-PC クラスタにおける OpenMP+MPI の性能評価, 情報処理学会 研究報告, Vol.2000, No.023, p.p.155-160, 2000.

[6] 宮野 悟: 並列アルゴリズム -理論と設計-, 近代科学社, 1993.

[7] P. S. Pacheco, 秋葉 博訳: MPI 並列プログラミング, 培風館, 2001.

[8] OpenMP Architecture Review Board, 技術研究組合 新情報処理開発機構訳: OpenMP C/C++ アプリケーション プログラム インタフェース バージョン 1.0, <http://phase.hpcc.jp/Omni/spec.ja/omp-C-1.0.pdf>, 2000.

[9] T. H. Cormen, R. L. Rivest and C. E. Leiserso, 浅野 哲夫他訳: アルゴリズムイントロダクション, 近代科学社, 1995.

[10] D. A. Patterson and J. L. Hennessy, 成田 光彰訳: コンピュータの構成と設計 ハードウエアとソフトウエアのインタフェース 第 2 版 下, 1999.

[11] 石川 祐, 佐藤 三久, 堀 敦史, 住元 信司, 原田 浩, 高橋 俊行: Linux で並列処理をしよう -SCore で作るスーパーコンピュータ-, 共立出版, 2002.

[12] 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, 立命館大学理工学研究科修士論文, 2006.

[13] 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, 立命館大学理工学部卒業論文, 2006.

[14] PC Cluster Consortium, <http://www.pccluster.org/>.