修士論文

Burrows-Wheeler変換の省メモリな並列計算手法

A Parallel and Memory-Efficient Burrows-Wheeler Transform

平成26年2月6日提出

指導教員　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-126435 林 伸也

概 要

全文検索はテキストを扱うアプリケーションの基礎を成す重要な技術である．テキストの任意の部分文字列の検索を可能にするインデックスとして代表的なものに接尾辞配列があるが, 元のテキストに比べてサイズが非常に大きくなるという問題がある．そこで近年, 接尾辞配列と同様の機能を持ちながら, より小さいメモリで動作する圧縮全文検索インデックスに注目が集まっている．圧縮全文検索インデックスの中でも特に FM-Index と呼ばれるものは, 圧縮した状態での検索が可能である上に, 元となるテキストが必要なくなるという性質を持っており, 有用性の高いインデックスである．

通常 FM-Index を構築するためには, まずテキストに対して Burrows-Wheeler 変換と呼ばれる変換を行う必要がある．しかし, Burrows-Wheeler 変換を定義通りに実行すると, その過程で一旦接尾辞配列を構築する必要があり, これではメモリ使用量を小さくするという圧縮全文検索インデックスの目的が達成されない．

そこで, テキスト全体に対して接尾辞配列を構築することなく Burrows-Wheeler 変換を行う研究が行われている．中でも高速なものは, 非常に小さいメモリのみを使用し, Burrows-Wheeler 変換を線形時間で計算することに成功している．しかし, 逐次アルゴリズムでは線形時間を下回ることはできないため, さらなる高速化のためには並列アルゴリズムを考案する他にない．

以上を鑑みて, 本研究では FM-Index を構築する際に必要となる Burrows-Wheeler 変換を, テキスト全体に対して接尾辞配列を構築することなく, かつ並列に計算するアルゴリズムを提案する．提案手法では分割統治法により細かく分割したテキストに対して部分的に接尾辞配列を構築し, そこから Burrows-Wheeler 変換を計算することで, メモリ使用量の増加を抑える．そして, 各部分文字列の Burrows-Wheeler 変換の出力をマージし, テキスト全体に対する Burrows-Wheeler 変換の計算結果を得る．さらに, タスク並列処理系を用いた並列化を行うことで, 動的に負荷分散を行う．これにより, テキスト全体の接尾辞配列を作ることなく, 線形時間を下回るクリティカルパスを達成している．

実験により, 人間のゲノム配列や英文テキストを入力とした実用的な場合において, メモリ使用量をある程度削減しながら, 既存手法より高速に動作することが確かめられた．

目 次

図 目 次

表 目 次

第1章　序論

1.1　圧縮全文検索インデックス

インターネット上で利用できるテキストデータの量は爆発的に増加している．テキストから有用な情報を抽出するときにはさまざまな技術が必要となるが, 特に全文検索はそれらのアプリケーションの基礎を成す重要な技術である．全文検索とは, 複数のテキストの中から短い文字列のパターンが出現する箇所を探しだす処理である．大量のテキストデータの中から検索を行う際には, 主に二通りの方法がある．一つは, テキストの先頭からパターンに一致する部分を探すシーケンシャルサーチである．シーケンシャルサーチは余計なメモリを使わないという利点があるが, サイズ n のテキストの検索に平均して $O(n)$ の時間がかかるため, 大きなテキストに対して適用するのは現実的ではない．もう一つは, テキストに対してあらかじめインデックスと呼ばれる, 検索を高速に行うためのデータ構造を用意する方法である．こちらはインデックスを構築するための時間が必要であるが, 一旦構築できてしまえば高速に計算を行うことが可能である．

全文検索のための最も代表的なインデックスの 1 つに転置インデックスがある．転置インデックスとは, テキスト中に出現する単語の種類を調べ, それらを辞書順に並べ, 各単語ごとにそれらがテキスト中で出現する箇所を列挙したものである．これを用いると, ある単語を検索するためには, まず辞書順に並べた単語リストに対して二分探索を行い, 続いてその単語の出現箇所を記したリストを先頭から確認していくことになる．そのため, 単語の出現箇所を高速に検索することができる．しかし, 転置インデックスは単語の区切りが明確であるテキストしか扱えないという短所がある．そのため, ゲノムやタンパク質の配列, 音声や映像等のシグナルデータを扱うことが難しかったり, 日本語や中国語等いくつかのアジア圏の言語などを扱う際には, 言語毎に形態素解析が必要になる．

転置インデックスとは異なるインデックスとして, 接尾辞配列 [27] が代表的である．接尾辞配列とは, テキストの全ての suffix をソートし, それらのテキスト中での出現位置を記録したインデックスである．接尾辞配列は転置インデックスとは異なり, テキスト中の任意の部分文字列を検索することが可能である．しかし, 接尾辞配列は元のテキストデータに比べて大きな領域を使用するという欠点があり, 大規模なテキストを扱うことが難しい．

この問題の解決策として, インデックスの圧縮が有効である．圧縮全文検索インデックスの中でも, FM-Index[14] と呼ばれるものはインデックスが元のテキストを復元するのに十分な情報を含んだ形で圧縮されており, かつ圧縮された状態での検索が可能であることから, 多くの研究が行われている [16][20][32][36]．また, 近年では圧縮全文検索インデックスを広く情報検索に用いようという研究も行われており [12][29][31][33], 今後ますますその重要性は大きくなるものと思われる．

このように非常に有用な性質を持った FM-Index であるが, その効率的な構築方法が問題となる．FM-Index を構築するときには, まずテキストに対して Burrows-Wheeler 変換 (BWT)[8] という変換を行う必要がある．BWT を定義通りに行うためには接尾辞配列を構築する必要がある

が, これでメモリ使用量を小さくするという圧縮の目的が達成されない.

BWT の省メモリな実現方法はいくつか研究が行われている. その中でも岡野原らは $O(n)$ という時間計算量を達成しており, これは非常に高速である [36]. しかし, 逐次アルゴリズムでは $o(n)$ の時間計算量を達成することはできないため, さらなる高速化のためには並列アルゴリズムを考案する他にないと考えられる.

1.2　本研究の目的と貢献

本研究では, 共有メモリ環境で, 小さいメモリ領域のみを使用し, かつ並列に Burrows-Wheeler 変換 (BWT) の計算を行うことを目的とし, タスク並列モデルによる BWT の並列計算アルゴリズムを提案する.

提案手法では, 分割統治法により再帰的に分割したテキストの各部分文字列に対して接尾辞配列を構築し, それを元に BWT を計算する. そして, 各部分文字列に対する BWT の出力を再帰的にマージしていく. こうすることで, テキスト全体に対して接尾辞配列を構築することがなくなるため, メモリ使用量を削減することが可能となる. また, 分割統治法によって計算を行っているためタスク並列モデルとの相性がよく, 並列計算における負荷分散を効率よく行うことが可能となる. 本研究で提案するアルゴリズムは, テキスト全体の接尾辞配列を構築することなく, かつ計算量のクリティカルパスとして $o(n)$ を達成した, 我々の知る限り最初のアルゴリズムである.

本研究では様々な種類の入力テキストに対して実験を行い, 既存の逐次アルゴリズムに比べて, 並列計算を行うことで高速に BWT を計算できることを確認した. また, 接尾辞配列を構築する場合に比べてメモリ使用量を削減できていることを確認し, 提案手法の有用性を示した.

1.3　本稿の構成

本稿の構成は以下のようになっている. 2 章では接尾辞配列の基本的性質と代表的な構築方法について触れ, 3 章では本研究の主眼となる BWT の説明をする. 4 章では簡潔データ構造について述べ, 5 章ではそれらを利用した圧縮全文検索インデックスについて説明する. 6 章では関連研究を紹介する. 7 章では提案手法の逐次アルゴリズムを説明し, 8 章でそれらの並列化について述べる. そして 9 章で提案手法の評価を行い, 10 章で結論を述べる.

1.4　記号の定義

テキスト T は文字の並びから成る. 各文字はアルファベットと呼ばれる集合 $\mathbf{\Sigma}$ の要素である. $\mathbf{\Sigma}$ の要素数は σ である. ある配列 A の i 番目の要素を $A[i]$ と表す. 例えば T の i 番目の文字は $T[i]$ である. 添字は 0 から始まることとする. また, テキスト T のサイズは自然数 n を用いて $|T| = n + 1$ と表されるとする. ただし, $T[n] =$ '\$' である. '\$' は番兵で, アルファベットのどの文字よりも辞書順で小さい文字であるとする. テキストサイズの最後で 1 を足しているのはこの番兵のためである. T の i 文字目から j 文字目までの部分文字列を $T[i, j]$ と書く. なお, 対数の底は特に断らない限り 2 であるとする.

第2章　接尾辞配列

2.1　接尾辞配列

接尾辞配列 [27] はテキストの全ての suffix を辞書順に並び替え, 各 suffix がテキスト中で出現する位置を格納した配列である. テキスト T の suffix とは, T の途中から始まり, 最後まで続く部分文字列を意味する. 具体的には T, $T[1, n]$, $\cdots$, $T[n-1, n]$, $T[n]$ である. 接尾辞配列は任意の部分文字列に対するインデックスであるため, ゲノム配列やバイナリデータなど, 区切りが明確でないデータに対しても検索を行えるという特徴がある.

例として, $T =$“mississippi\$” というテキストを考える. 1.4 節で述べた通り, テキストの最後は \$になることに注意しなければならない. T の suffix は後ろから順に “\$”, “i\$”, “pi\$”, “ppi\$”, $\cdots$, “mississippi\$” である. これを辞書順に並び替えると, 表 2.1 のようになる. このようにして得られる配列 $\mathrm{SA} = [11,\ 10,\ 7,\ \cdots,\ 2]$ が接尾辞配列である.

2.2　接尾辞配列の規則性

接尾辞配列は一般には $[0, n]$ に含まれる整数の並び替えになっている (番兵まで含めたテキストサイズを $n+1$ としていることに注意). しかし, もし $\sigma \ll n$ のとき, 可能なテキストのパターンは σ^{n+1} 通りであり, これは $[0, n]$ の並び替えの数 $(n+1)!$ に比べてずっと少ない. つまり, $[0, n]$

表 2.1. 接尾辞配列と Ψ 関数

	suffixes	sorted suffixes	接尾辞配列	Ψ
0	mississippi\$	\$	11	5
1	ississippi\$	i\$	10	0
2	ssissippi\$	ippi\$	7	7
3	sissippi\$	issippi\$	4	10
4	issippi\$	ississippi\$	1	11
5	ssippi\$	mississippi\$	0	4
6	sippi\$	pi\$	9	1
7	ippi\$	ppi\$	8	6
8	ppi\$	sippi\$	6	2
9	pi\$	sissippi\$	3	3
10	i\$	ssippi\$	5	8
11	\$	ssissippi\$	2	9

の並び替えとして可能な数列の全てがあるテキストの接尾辞配列になり得るわけではない．この意味で, 接尾辞配列にはある種の制約があると言える．以下ではそのような制約の詳細について説明する．

表 2.1 をもう一度見てみよう．この表の接尾辞配列の値を見てみると, 例えば $[3, 4]$ の範囲の値が 4, 1 となっている．また, $[10, 11]$ の範囲の値が 5, 2 となっており, さらに $[8, 9]$ の範囲の値が 6, 3 となっており, $[2, 3]$ の範囲の値が 7, 4 となっている．つまり, 範囲 $[3, 4]$ の接尾辞配列の値に対して, それらに 1 足したもの, 2 足したもの, 及び 3 足したものがそれぞれ隣接して現れている箇所が存在している．$[3, 4]$ の suffix は “issippi$”, “ississippi$” であり, $[10, 11]$ の suffix はこれらの先頭一文字を取り除いた “ssippi$”, “ssissippi$” となっている．さらに, $[8, 9]$ の suffix は先頭をもう一文字を取り除いた “sippi$”, “sissippi$” である．$[2, 3]$ の suffix も同様である．つまり先ほどの現象は, 同じ文字から始まるいくつかの suffix について, それらの先頭一文字を取り除いてもそれらの辞書的な順序は変化しないことに起因している．以上のことから, 接尾辞配列の自己反復 (self repetition)[24][25] の概念を定義できる．

定義 1 (自己反復)**.** あるテキストの接尾辞配列を SA とする．このとき, 範囲 $[i, i + l]$ について, もし $[j, j + l]$ が存在し, $0 \leq r \leq l$ を満たす全ての r について $SA[j + r] = SA[i + r] + 1$ が成り立つとき, 範囲 $[i, i + l]$ を自己反復と呼ぶ．

自己反復は Ψ 関数と呼ばれる関数と密接な関係がある．以下に Ψ 関数の定義を述べる．

定義 2 (Ψ 関数)**.** Ψ 関数とは, $0 \leq i \leq n$ に対して $SA[\Psi(i)] = SA[i] + 1$ を満たす関数である．

Φ 関数は直感的には, 辞書順で i 番目の suffix よりテキスト中で 1 つ後ろの位置の suffix の辞書的な順位を表している．Ψ 関数の例を表 2.1 に示す．Ψ 関数にはいくつかの性質があるが, 特に重要なのは, 接尾辞配列の自己反復の中においては, Ψ 関数の値はちょうど 1 ずつ単調増加するということである．このような範囲の数を数えることで, 接尾辞配列を被覆するのに必要な自己反復の数 n_{sr} が分かる．n_{sr} は後の章で説明する Burrows-Wheeler 変換と密接な関係を持つ重要な値である．

2.3　構築アルゴリズム

2.3.1　ソートアルゴリズムを利用する方法

最も単純な接尾辞配列の構築方法はクイックソートを用いて全ての suffix をソートする方法である．クイックソートの計算量は $O(n \log n)$ であるが, 文字列ソートの場合は 2 つの文字列を比較するのにどれだけの時間を要するかが問題になる．ソートアルゴリズムを用いた接尾辞配列の計算量を考えるために, 識別 prefix (distinguishing prefix) という概念を定義する．

定義 3 (識別 prefix)**.** 文字列集合の中で, 他のどの文字列の *prefix* にもなっていない *prefix* のうち, 長さが最小のものを識別 *prefix* という．

例えば {“apple”, “apart”, “appeal”} という文字列集合があったときに, “apple” の識別 prefix は “appl”, “apart” の識別 prefix は “apa”, そして “appeal” の識別 prefix は “appe” となる．

ソート対象の文字列の識別 prefix 長の総和を D とする．このとき, クイックソートによるソートの計算量は $O(Dn \log n)$ となる．これはソート対象の文字列の prefix が長く一致する場合に, 極めて性能が悪くなる．そこで, 文字列ソートのためにクイックソートを改良したのがマルチキー・クイックソート [4] である．マルチキー・クイックソートではピボットを決めた後, まず文字列の一文字目だけを順に見ていき, ピボットより小さいもの, 同じもの, そして大きいものの 3 つに分ける．そして, 小さいものと大きいものについては再帰的に処理を行う．ピボットと一文字目が同じものは, 最早一文字目を比較する必要がないため, ニ文字目について同様の操作をする．このように, ピボットと同じものは比較する文字の位置を 1 つ進めていくことで, 同じ文字を何度も比較する必要がなくなる．マルチキー・クイックソートの計算量は $O(D + n \log n)$ となる．

マルチキー・クイックソートは文字列ソートとして優れているが, 一般の文字列をソートするための手法であり, suffix 間に存在する強い依存関係を利用できない．そのため, 実際に接尾辞配列を構築する際には, suffix 間の依存関係をうまく利用した, 時間計算量が $O(n)$ であるアルゴリズムが使われている．

2.3.2　DC3

DC3 (difference cover modulo 3)[21] は, テキスト全体の suffix のうちのいくつかの順序を先に計算し, それによって残りの suffix の順位を決定することで接尾辞配列を構築する手法である．実は後で述べる SA-IS でも同様の手順により接尾辞配列を構築するのだが, 先に計算するいくつかの suffix の選び方, 及びそれを元に残りの suffix の順位を計算する方法が異なる．

DC3 の説明をするために, まずは数学的な背景となる difference cover について述べる．$D \subseteq \mathbb{Z} \cap [0, v)$ という集合について, D に含まれる全ての要素の差をある整数 $v(v > 0)$ で割った値の集合が $\mathbb{Z} \cap [0, v)$ に等しいとき, D を v を法とする difference cover という．Difference cover の定義は以下のようになる．

定義 4 (Difference cover[21])**.** 集合 $D \subseteq \mathbb{Z} \cap [0, v)$ が以下の条件を満たしているとする．

$$\{(i - j) \bmod v \mid i, j \in D\} = \mathbb{Z} \cap [0, v) \tag{2.1}$$

このとき, D を v を法とする *difference cover* という．

Difference cover の直感的な意味は, 要素の差の modulo を全て集めるとそれが区間 $[0, v)$ の整数を被覆するということである．すなわち, 2 つの任意の 0 以上の整数 a, b に対して, $a + r \bmod v \in D$ かつ $a + r \bmod v \in D$ となるような整数 r が必ず存在する．

例として, 5 を法とする difference cover を示す．例えば $D = \{1, 2, 4\}$ とすると, これは difference cover となっている．表 2.2 は, D の各要素の差を 5 で割った余りを示したものである．この表の中には確かに $[0, 5)$ の全ての数が現れていることが分かる．

DC3 では, 3 を法とする difference cover を利用している．これは具体的には $D = \{1, 2\}$ となる．これを元に $D' = \{x | x \bmod 3 \in D\}$ という集合を構成する．そして, $i \in D'$ となるような場所 i についてのみ先に接尾辞配列を計算する．これを SA_{12} とする．これらの順位を元に, $i \notin D'$, つまり $i \bmod 3 = 0$ の接尾辞配列を求める．これを SA_0 とする．SA_{12} の逆関数 ISA_{12} をあらかじめ求めておけば, $i \bmod 3 = 0$ の位置の suffix の先頭文字と $i \bmod 3 = 1$ の suffix の順位を元に radix sort を行うことで, SA_0 を $O(n)$ で求めることができる．

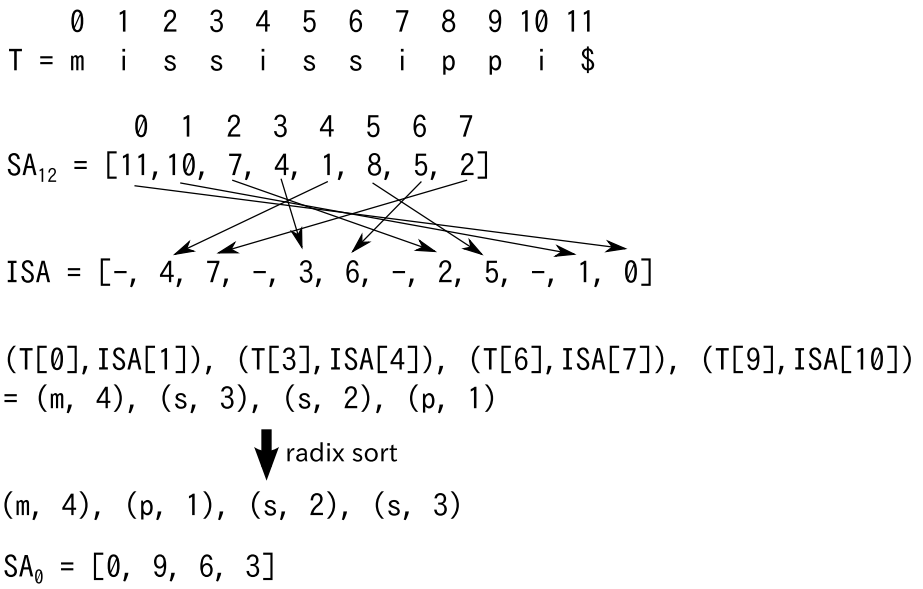

図 2.1. DC3 における SA_0 の計算例

ここまでの例を図 2.1 に示す. 例では $T =$“mississippi\$” としている. SA_{12} が求まったら, まずはその逆関数を計算する. 接尾辞配列の逆関数の各要素は, その位置から始まる suffix の順位を表す. そして, $i \bmod 3 = 0$ の位置の suffix の先頭文字と, その次の位置の接尾辞配列の逆関数の要素の組を作り, それをソートする. これにより, SA_0 を構築することができる.

最後に, SA_0 と SA_{12} をマージする. SA_0 と SA_{12} が指す suffix を先頭から順に比較していき, 小さい方を順に別の領域に格納していく. 比較の際には, $i \bmod 3 = 0$ と $i \bmod 3 = 1$ の suffix を比較する場合と, $i \bmod 3 = 0$ と $i \bmod 3 = 2$ の suffix を比較する場合の 2 通りがある. 前者の場合, それぞれ次の位置の suffix の順位はあらかじめ計算済みであるため, (SA_0 or SA_{12} が指す場所の文字, 次の位置の suffix の順位) という 2 つ組を比較することで順位が求まる. つまり, suffix 自体を直接比較する必要がない. 後者の場合, それぞれ 2 つ先の位置の suffix の順位はあらかじめ計算済みであるため, (SA_0 or SA_{12} が指す場所の文字, SA_0 or SA_{12} が指す場所の次の文字, 2 つ先の位置の suffix の順位) という 3 つ組を比較することで順位が求まる. このようにして, SA_0 と SA_{12} のマージを $O(n)$ の時間で行うことができる.

ここまでの話は SA_{12} をあらかじめ計算し終えていることを前提としていた. SA_{12} の構築は, 一見すると $O(n \log n)$ の計算量を避けられないように思われるが, 実際には $O(n)$ で構築すること

表 2.2. 5 を法とする difference cover の例

	1	2	4
1	0	1	3
2	4	0	2
4	2	3	0

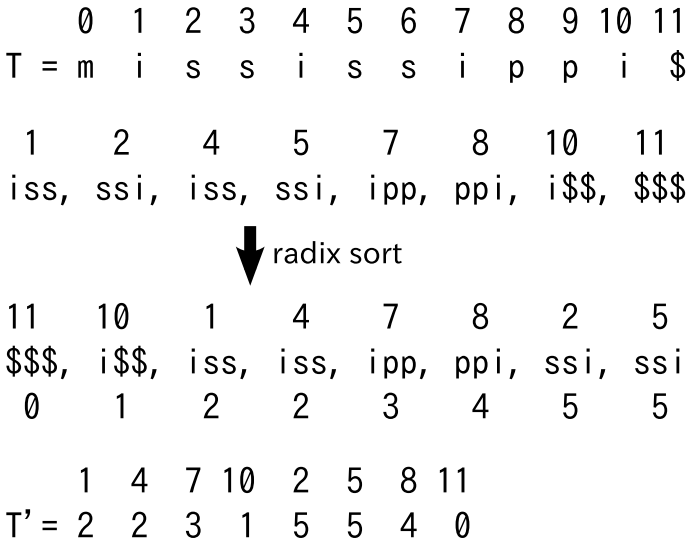

図 2.2. DC3 における SA_0 の計算例

が可能である. そのためには, まず $i \bmod 3 = 1$ と $i \bmod 3 = 2$ の suffix を先頭 3 文字で radix sort する. 文字列の最後に到達してしまった場合は, それより先には\$が入っているものとする. もし 3 文字の radix sort で全ての suffix の順位が一意に決まれば, 計算はこれで終わりである. もしこれでは順位が決定できない場合は, 上で比較に用いた 3 文字を, radix sort によって得られた順位で置き換えた新たな文字列 T' を構築し, これに対して再帰的に DC3 の手続きを適用する. T' は元の文字列 T に比べて, 3 分の 2 程度の大きさになるため, 再帰的に DC3 の計算を行っていった場合に, 最悪の計算量は以下のようになる.

$$\begin{aligned} \mathrm{DC3}(n) &= \mathrm{DC3}\left(\frac{2}{3}n\right) + O(n) \\ &= \sum_{k=0}^{\infty} O\left(\left(\frac{2}{3}\right)^k n\right) \\ &= O(n) \end{aligned} \tag{2.2}$$

SA_{12} の計算例を図 2.2 に示す. この例ではまず $i \bmod 3 = 1$, $i \bmod 3 = 2$ の位置である 1, 2, 4, 5, 7, 8, 10, 11 について, そこから始まる suffix の先頭三文字を radix sort によって並び替えている. すると, iss と ssi がそれぞれ二回ずつ出現しているため, この段階ではまだそれぞれの順位が一意に定まらない. そこで, $i \bmod 3 = 1$ の suffix の順位を 1, 4, 7, 10 と順に並べ, その後ろに $i \bmod 3 = 2$ の suffix の順位を 2, 5, 8, 11 と順に並べた新たな入力文字列 T' を構築する. T' に対して DC3 の手続きを再帰的に適用することで, 各 suffix の順位を一意に定めることができる.

以上により, DC3 は $O(n)$ で接尾辞配列を構築することが可能である.

2.3.3 SA-IS

SA-IS[34] は DC3 と同じく $O(n)$ で接尾辞配列を構築できるアルゴリズムである．SA-IS でも DC3 と同じく一部の suffix の順位を先に決定し, それを元に残りの suffix の順位を計算する．先に計算する suffix というのも, DC3 と同様に再帰的に副問題を生成していくことで $O(n)$ の時間で計算が可能である．後に説明するが, この先に計算する一部の suffix の割合が DC3 よりも小さいため, SA-IS は DC3 よりも高速に動作する．

SA-IS では, 各 suffix をそれぞれ S 型と L 型という 2 つの組に分ける．もしある位置の suffix がテキスト中の次の位置の suffix よりも辞書順で小さい場合, その suffix は S 型であり, そうでなければ L 型であるとする．さらに, S 型の suffix の中でも L 型の直後に位置するものを LMS (left most S) 型と呼ぶ．また, テキストの最後の終端文字 '$' も LMS 型の suffix であるとする．

SA-IS では, まずこの LMS 型の suffix をソートする．今, LMS 型の suffix がソートされていると仮定する．残りの suffix をソートするために, まずテキストを構成する各文字の種類毎にバケツを用意する．LMS 型の suffix を, その先頭文字に応じて対応するバケツに入れる．各文字のバケツは更に L 型の領域と S 型の領域に分けることができる．これは, 同じ文字から始まる suffix については, L 型より S 型の suffix の方が辞書順で必ず大きくなるためである．よって, LMS 型 suffix は, 各文字のバケツの後半部分である S 型領域に入れる．次に, このバケツを先頭から順に見ていく．もし要素が入っているバケツに遭遇したら, その suffix よりテキスト中で 1 つ前の位置の suffix の型を調べる．もしそれが L 型であれば, この suffix を対応するバケツの L 型領域の先頭に入れる．そして, L 型領域の先頭を指すポインタを 1 つ後ろに進める．もし S 型であれば何もしない．LMS 型は L 型の直後にある S 型 suffix 及びテキストの終端文字であると定義したため, LMS 型 suffix の直前の L 型 suffix は確実にバケツに入れることができる．また, L 型 suffix が連続する領域においては, 手前のものほど辞書順で大きいため, L 型が連続する領域の最後の suffix がバケツに入っていれば, 順次手前の suffix をバケツに入れることができる．よって以上の操作で L 型の suffix を全て正しい辞書順でバケツに入れることが可能である．

最後に, L 型 suffix の並びを元に S 型 suffix の順序を決定する．まず, 先ほどバケツに入れた LMS 型 suffix は破棄しておく．すると, バケツには L 型 suffix のみが入っていることになる．ここで, 今度はバケツの要素を後ろから順に見ていく．もしある要素のテキスト中の 1 つ前の suffix が S 型であれば, これを対応するバケツの S 型領域の一番後ろに入れる．そして, 一番後ろを指すポインタを 1 つ手前に進める．このようにすることで, 今度は全ての S 型 suffix を正しい辞書順でバケツに入れることができ, 結果としてテキスト T の接尾辞配列を構築できたことになる．このように, まずソートされた LMS 型の suffix から L 型の順序を計算し, そこからさらに S 型の suffix の順序を計算する方法は induced sorting と呼ばれている．

Induced sorting の例を図 2.3 に示す．図では LMS 型 suffix を S*で表している．この例では, まず LMS 型の suffix に対する接尾辞配列は 11, 7, 4, 1 となる．これらの値を, suffix の先頭文字に応じて分ける．先頭から順に要素を見ていくと, まず 11 に遭遇する．テキスト中で 11 の 1 つ前の位置は 10 であり, この位置は L 型である．よってこの位置の suffix をバケツに入れる．10 の位置の文字が 'p' であるため, 'p' のバケツに入れる．次に遭遇するのは 7 である．7 の 1 つ前の位置である 6 も L 型であるため, 6 をバケツに入れる．このようなことを繰り返していくことで, L 型の suffix の順位を求めることができる．図には L 型を induce する部分しか示していないが, S 型も同様である．

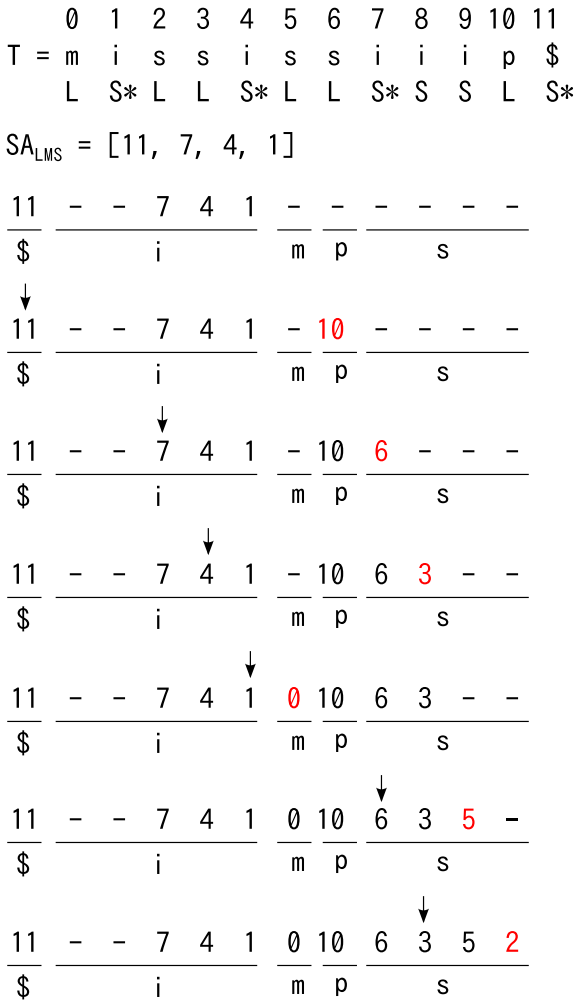

図 2.3. Induced sorting の例

以上の議論では LMS 型の suffix の順位があらかじめ計算してあることを前提としていた．実際, LMS 型の suffix は $O(n)$ で計算が可能である．このために, LMS 部分文字列という概念を導入する．LMS 部分文字列とは, 2 つの LMS 型 suffix の先頭位置に挟まれた部分文字列である．LMS 部分文字列の両端は, LMS 型の文字自体を含むが, LMS 部分文字列の内部には他の LMS 型の文字は存在しないとする．この LMS 部分文字列をソートし, もしそれらの順位が一意に決まれば, それはつまり LMS 型の suffix の順位が決まることになる．もし LMS 部分文字列の比較だけでは順位が決定できなければ, 各 LMS 部分文字列を, 先のソートによって得られた順位に置き換えた新たな文字列 T' を構築し, これに対して再帰的に SA-IS の手続きを適用する．

問題は LMS 部分文字列のソートであるが, これも $O(n)$ で行うことが可能である．このためにはさらに LMS suffix という概念を導入する．LMS suffix とは, テキスト中のある位置から始まって, それ以降にある最も近い LMS 型の位置で終わる (その場所自身も含む) 部分文字列である．LMS 型の位置そのものは, その位置の文字のみからなる LMS suffix であると考える．実はこの LMS suffix の順序は, わずかに変更を加えた induced sorting により計算することができる．その

ために, まず一文字の LMS suffix をバケツに入れる. このとき, 同じ文字の LMS suffix については, その順序は問わず, 任意の順序でバケツに配置すればよい. 次に, バケツを先頭から見ていき, induced sorting と同様に, もしある LMS suffix のテキスト中での 1 つ前の位置が L 型であれば, それを対応するバケツの L 型領域の先頭に入れ, 先頭を指すポインタを 1 つ先に進める. この操作が終わったら, やはり induced sorting と同様に, 今度はバケツを後ろから順に見る. そして, ある要素のテキスト中での 1 つ前の位置が S 型であれば, それを対応するバケツの S 型領域の最後に入れ, 最後を指すポインタを 1 つ手前に進める.

ここまでの操作で, 一意な LMS 部分文字列は正しくソートされており, 同じ LMS 部分文字列が複数あるような場合については, 少なくともそれらは隣接する位置に配置される. よって最後にそれらを直接比較することで現状での順位を計算することができる. 隣接する suffix しか比較しないので, これは最悪でも各 LMS 部分文字列を 2 回しか走査せず, 結果としてテキスト全体を 2 回走査するのと同じだけの文字列比較しか行わないので, $O(n)$ で計算可能である. 以上により, LMS 型部分文字列の順位を決定でき, 結果として LMS 型の suffix の順位を計算できる.

SA-IS において始めにソートする LMS 型 suffix は, 最大でもテキスト全体の 1/2 の割合でしか存在しない. よって SA-IS の最悪計算量は以下のようになる.

$$\begin{aligned} \text{SA-IS}(n) &= \text{SA-IS}\left(\frac{1}{2}n\right) + O(n) \\ &= \sum_{k=0}^{\infty} O\left(\left(\frac{1}{2}\right)^k n\right) \\ &= O(n) \end{aligned} \tag{2.3}$$

このように, 再帰呼び出しする副問題のサイズが DC3 より小さいため, 一般的に SA-IS は DC3 より高速である.

2.4　空間計算量の問題

接尾辞配列の空間計算量は $O(n \log n)$ ビットである. これは元のテキストに比べて $\log n / \log \sigma$ 倍大きい. これは特にゲノムデータのように σ が小さい場合に問題となるが, 例えば英文のテキスト等であっても σ は高々128 程度であり, 4GB 程度のテキストの場合だと接尾辞配列はテキスト自体の 4 倍以上ものメモリを必要とすることになる.

第3章　Burrows-Wheeler変換

3.1　Burrows-Wheeler変換

Burrows-Wheeler 変換 (BWT)[8] はテキストに対する可逆変換の一種である. BWT は, テキスト中の各文字を, その文字に後続する suffix をキーとして並び替えたものである. 以下に BWT の定義を示す.

定義 5 (Burrows-Wheeler 変換)**.**

$$T^{BWT}[i] = \begin{cases} T[n] & (SA[i] = 0) \\ T[SA[i] - 1] & (otherwise) \end{cases} \tag{3.1}$$

BWT により変換されたテキストには同じ文字が連続して現れやすいという性質がある. 例えば英語には “The” という単語が数多く出現するため, “he” から始まる suffix の前に “T” という文字が現れる確率が高い. その結果, BWT の出力中で “he” から始まる suffix に対応する範囲には “T” が連続して並びやすくなる [38]. この性質により, BWT を行ったテキストは圧縮しやすくなる. 例として, T =“mississippi\$” の BWT を表 3.1 に示す. この例では T^{BWT} =“ipssm\$pissii” となる.

表 3.1. BWT の例

	suffixes	sorted suffixes	接尾辞配列	T^{BWT}
0	mississippi$	$	11	i
1	ississippi$	i$	10	p
2	ssissippi$	ippi$	7	s
3	sissippi$	issippi$	4	s
4	issippi$	ississippi$	1	m
5	ssippi$	mississippi$	0	$
6	sippi$	pi$	9	p
7	ippi$	ppi$	8	i
8	ppi$	sippi$	6	s
9	pi$	sissippi$	3	s
10	i$	ssippi$	5	i
11	$	ssissippi$	2	i

T^{BWT} に同じ文字が連続して現れる範囲の個数 n_{bw} は, 2.2 節で説明した n_{sr} によって評価できる. すなわち, 以下の不等式が成立する [32].

$$n_{sr} \leq n_{bw} \leq n_{sr} + \sigma \tag{3.2}$$

証明の概略を述べる. まず右側の不等式について説明する. もし自己反復中の suffix が全て同じ文字から始まっていたとき, T^{BWT} の中にその文字が連続して現れる範囲が存在する. すなわち, $0 \leq r \leq l$ の範囲において $SA[j+r] = SA[i+r]+1$ かつ $T[SA[i+r]] = c$ であるとすると, $T^{BWT}[j+r] = T[SA[j+r]-1] = T[SA[i+r]] = c$ となる. 一方で, ソートされた suffix の先頭文字を順に見ていくと最大で σ 回変わるので, 自己反復の中で suffix の先頭文字は σ 回変わる可能性がある. よって $n_{bw} \leq n_{sr} + \sigma$ となる.

次に左側の不等式について説明する. $T^{BWT}[j+1] = T^{BWT}[j]$ となっているとき, $T[SA[j], n] = X$ の後ろには $T[SA[j+1], n] = Y$ が続き, それらの一文字前の文字は共に c である. よって 2 つの suffix cX と cY もソートされた suffix の列において連続した位置 i, i+1 に現れ, $\Psi(i) = j$, $\Psi(i{+}1) = j+1$ となる. すなわち, $T^{BWT}[j+1] = T^{BWT}[j]$ となっているときは必ず $\Psi(i+1) - \Psi(i) = 1$ となる. 以上により $n_{sr} \leq n_{bw}$ が成立する.

3.2　BWT の逆変換

BWT は可逆変換であるから, 変換された文字列から元のテキストを復元することができる. 以下ではまずそのために必要である LF-mapping[8] と呼ばれる変換について説明する.

ソートされた suffix の先頭文字を集めた配列を F, それらの 1 つ前の文字を集めた配列を L とする (L は T^{BWT} そのものである). このとき F と L は T の並び替えの一種であるから, F 中の文字と L 中の文字の間には一対一の対応関係がある. L 中のある文字が F 中に現れる位置を求める操作のことを LF-mapping という. LF-mapping を求めるためには, 同じ種類の文字の並びは F, L において同じであるという性質を利用する. 例を図 3.1 に示す. この図では, 説明の便宜上 suffix の代わりに, テキストを一文字ずつ回転させて出来る文字列であるサイクリックシフトをソートしたものを示している. しかし, テキストの最後に番兵があるため, 結局 suffix の並び替えとサイクリックシフトの並び替えは同じである. さて, この図では 'i' から始まる suffix が 4 つあるが, これらの suffix から 'i' を取り去っても, suffix 間の順序は変わらない. そのような suffix はサイクリックシフトの中で L が 'i' になっているものに相当する. よって L の値が 'i' になっている suffix 間の順序は, 'i' から始まる suffix の順序と同じになる. 結果として F, L のある文字に注目すると, それらは上から順に一対一に対応する.

以下に LF-mapping の定義を示す.

定義 6 (LF-mapping)**.**

$$LF(i) = C(L[i]) + \mathrm{rank}_{L[i]}(L, i) - 1 \tag{3.3}$$

ただし $C(c)$ は T に現れる, 文字 c よりも小さい文字の数を返す関数, $\mathrm{rank}_c(L, i)$ は L の i 番目までに現れる文字 c の数を返す関数を表す. この式では, まず $L[i]$ より小さい文字の数を数え, 次に $L[i]$ と同じ種類の文字の中で $L[i]$ 自身は何番目になるのかを計算し, F 中での位置を求めている.

$T^{BWT}[i] =$'\$' となる場所 i から, $T^{BWT}[i]$, $T^{BWT}[LF(i)]$, $T^{BWT}[LF^2(i)] \cdots$ というように LF-mapping を繰り返すと, テキストを逆向きに復元することができる.

図 3.1. LF-mapping

3.3　BWT の計算

式 (3.1) を見ると分かるように, BWT の計算には通常接尾辞配列が必要である. 後に説明するように, BWT は圧縮全文検索インデックスの構築に用いられるが, これは接尾辞配列のメモリ使用量の問題を解消するための手法であるにも関わらず, その構築のために一度接尾辞配列を構築しなければならないというのは, あまり意味がないように思われる. そこで BWT を小さいメモリ領域しか使わずに構築する方法が考案されている. 詳細は 6 章で述べる.

第4章　簡潔データ構造

4.1　簡潔データ構造

簡潔データ構造とは, データ構造を情報理論的下限に近い領域に収めつつ, 索引を利用することで効率的な検索を行うことができるデータ構造である. ここでは特にデータ構造に対して access, rank, select と呼ばれる 3 つの操作を効率的に行えるものを考える.

4.2　完備辞書

完備辞書とは, ビット列に対する簡潔データ構造である. B を任意のビット列とすると, access, rank, select の定義は以下のようになる.

- $\mathrm{access}(B, i)$: $B[i]$ を返す.
- $\mathrm{rank}_b(B, i)$: i 番目までに存在するビット $b \in \{0, 1\}$ の数を返す.
- $\mathrm{select}_b(B, i)$: i 個目のビット $b \in \{0, 1\}$ の位置を返す.

ビット列のサイズを n とする. 以下ではビット列を $O(n)$ ビットの領域に収めつつ, これらの演算を定数時間で行うことができる方法について説明する. この方法では, ビット列自身はそのまま保持し, さらに補助的なデータ構造を用意することで rank, select を定数時間で行うことが可能になる. access については, ビット列自身はそのまま保持されているので, 定数時間での実現は容易である. また, select は本研究では使用しないのに加え, rank が定数時間で実現できれば二分探索によって $O(\log n)$ の時間で実現することも可能なので割愛する. よってここでは rank を定数時間で実現する方法 [9] について述べる.

i 番目までに現れる 1 の数を数える方法について述べる. 0 の数については, $i - \mathrm{rank}_B(1, i)$ で実現することができる. まず, ビット列を $\log^2 n$ 個毎にスーパーブロックと呼ばれる単位に分割する. そして各スーパーブロック間の境界について, そこまでに出現する 1 の数を数え, それを保持しておく. 各要素には $\log n$ ビットの領域が必要であるため, これには $n/\log^2 n \times \log n = n/\log n$ ビットの領域が必要である. 次に, 各スーバーブロックをさらにサイズ $\log n/2$ のブロックに分割する. そして各ブロックの境界について, 直前のスーパーブロックの境界から数えてそこまでに 1 がいくつ出現するかカウントする. 各要素には $\log(\log^2 n) = 2 \log \log n$ ビットの領域が必要であるため, これには $n/\log n \times 2 \log \log n = n \log \log n/\log n$ ビットの領域が必要である. 最後に, ブロックの全てのビットのパターンについて, 予め rank を計算したテーブルを用意する. 各要素には $\log(\log n/2)$ ビットの領域が必要であり, またビットのパターンは全部で $2^{\log n/2}$ 個あるため, これには $2^{\log n/2} \times \log(\log n/2) = \sqrt{n} \log(\log n/2)$ ビットの領域が必要である. よってこれらの補助的なデータ構造のサイズは $o(n)$ に抑えられることが分かる. rank を求めるためには, これらのデータ構造に 3 回アクセスすれば良いので, rank は定数時間で計算することができる.

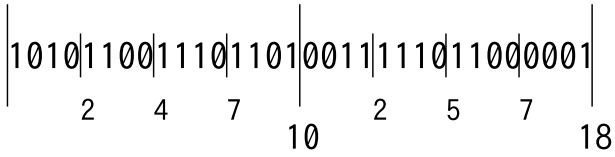

図 4.1. 完備辞書の例

例を図 4.1 に示す. 図ではブロックサイズを 4, スーパーブロックサイズを 16 としている. (それぞれのサイズが $\log n/2$, 及び $\log^2 n$ になっていないが, 説明の便宜上このようにする.) この図において $\mathrm{rank}_1(B, 21)$ を計算することを考える. まず, スーパーブロックサイズが 16 なので, $\lfloor 21/16 \rfloor = 1$ となり, 21 は (0 始まりで数えて)1 つ目のスーパーブロックに存在する. また, ブロックサイズが 4 なので, $\lfloor 21/4 \rfloor = 5$ となり, 21 は 5 番目のブロックに存在することが分かる. 0 番目までのスーパーブロックに現れる 1 の数は 10, さらに 1 番目のスーパーブロックの最初から数えて, 4 番目までのブロックに現れる 1 の数は 2 である. 5 番目のブロックの中身は 1110 となっている. 0000, 0001, 0010, $\cdots$, 1111 というビットに現れる 1 の数を記録した配列 P を用いて, このビット列の 1 の数を取得したいのだが, 21 はこの 1 つ目の位置に相当するため, 2 つ目以降のビットは数えてはならない. よって $(1110\&1100) = 1100$ というようにマスクしたビット列を用いて値を取得する. 結局, 1110 の 1 番目のビットまでに現れる 1 の数は 2 となる. 以上を全て足し合わせると, 21 番目までに現れる 1 の数は $10 + 2 + 2 = 14$ と分かる.

以上の方法は, 空間計算量が $n + o(n)$ となっており, 圧縮されているわけではない. ビット列を圧縮した状態で保持し、かつ access, rank, select を定数時間で行うことができるというより洗練された手法 [37] が Raman らによって提案されているが, 本研究の範囲を外れるので割愛する.

4.3　Wavelet 行列

4.2 節ではビット列に対する簡潔データ構造について述べたが, 一般の文字列に対しても access, rank, select を行うことができるような簡潔データ構造が存在する. ここで, access, rank, select は以下のように定義される.

- $\mathrm{access}(T, i)$: $T[i]$ を返す.
- $\mathrm{rank}_c(T, i)$: i 番目までに存在する文字 $c \in \mathbf{\Sigma}$ の数を返す.
- $\mathrm{select}_c(T, i)$: i 個目の文字 $c \in \mathbf{\Sigma}$ の位置を返す.

このようなデータ構造のうち, 最も基本的なものが wavelet tree[17] である. Wavelet tree は, 文字列に対して構成されたいくつかのビット列を tree 上に繋げたデータ構造であり, これを用いることで access, rank, select がそれぞれ $O(\log \sigma)$ で行うことが可能となる. しかし, wavelet tree では木構造を構成するために各ノードに対応するビット列間の関係を表すポインタを用意する必要がある. このような wavelet tree の複雑さを解消するため, よりシンプルな wavelet 行列 [10] という新しいデータ構造が提案された. 以下では wavelet 行列の構築方法, 及び access, rank の計算方法について述べる. Select は本研究では用いないので割愛する.

構築について説明する. まず, 文字列 T を構成する文字の種類に対して, 0 から順に 1 つずつ整数値を割り当てていく. そして, T の各文字に対して, その文字に対して割り当てられた数値を二

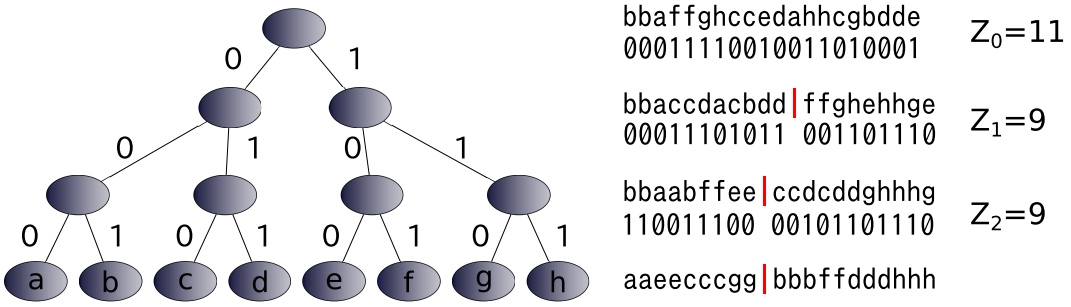

図 4.2. Wavelet matrix

進数として捉えたときの 0 番目のビットを割り当てていく．その後, 0 が割り当てられた文字は, それらの要素間での順番を保ったまま左詰めにし, その後ろに 1 が割り当てられた文字を, 同じく順序を保存したまま移動させる．このようにして得られたビット列 B_0 を保存する．次に, 先の操作で得られた文字列に対して, 各文字に割り当てられた数値の先頭から 1 番目のビットを割り当てていく．その後, 割り当てられたビットを元に, 再び 0 を割り当てられたものを左へ詰め, 1 を割り当てられたものをその後ろに置く．これによって得られたビット列 B_1 を保存する．このような操作を繰り返していき, 各レベルで得られたビット列 $B_0, B_1, \cdots, B_{\log\sigma-1}$ が wavelet 行列である．

wavelet 行列の例を図 4.2 に示す．この例では “bbaffghccedahhcgbdde” という文字列に対して wavelet 行列を構築している．アルファベットの各要素に割り当てられる数値は図の左側のアルファベット木をルートから辿ることで分かる．例えば, ‘e’ はルートから木を辿ると 100_2 である (下付き文字の 2 は二進数であることを表す)．まず, 入力文字列の各文字対して, 0 番目のビットに応じてそれぞれ 0, 1 を割り当てる．ここでは ‘a’, ‘b’, ‘c’, ‘d’ に 0 が割り当てられ, それ以外には 1 が割り当てられる．次に, 0 が割り当てられた文字を, 順番を保ったまま左詰めにし, その直後に 1 が割り当てられた文字を置く．これらに対して, 今度は 1 番目のビットを見て同様の操作を行う．結果として, 図のように 3 つのビット列が得られる．これが wavelet 行列である．

次に, wavelet 行列を用いた access の計算方法について説明する．まず, B_0 の i 番目のビットについて, もしこれが 0 であれば $rank_0(B_0, i) - 1$ を計算し, もし 1 であれば $rank_1(B_0, i) + Z_0 - 1$ を計算する．ここで, Z_i は B_i における 0 の個数である．1 を引いているのは要素のカウントを 0 始まりにするためである．この計算結果を i_1 とする．次に, B_1 の i_1 番目のビットについて同様の計算を行い, i_2 を得る．このような操作を繰り返していき, 最終的に $i_{\log\sigma}$ を得る．その後, $i_{\log\sigma}$ がどのアルファベットの範囲に相当するか調べる．Wavelet 行列において, 最下位レベルでは各文字がビット逆順に並ぶ．そのため, 予め各文字がどういう順番でそれぞれいくつ出現するかを計算しておけば, 二分探索によって $i_{\log\sigma}$ が属するアルファベットの範囲が計算できる．結果として, ビット列に対する rank を $\log\sigma$ 回行い, さらに σ 個のアルファベットの中から二分探索によって該当する文字を見つけるため, access の計算量は $O(\log\sigma)$ となる．

最後に rank の計算方法について説明する．まず, 文字 c に割り当てられている数値の 0 番目のビットを調べる．これが 0 であれば, $rank_0(B_0, i) - 1$ を計算し, そうでなければ $rank_1(B_0, i) + Z_0 - 1$ を計算する．この結果を i_1 とする．またこれとは別に, 文字 c に割り当てられている数値の 0 番目のビットに応じて, $rank_0(B_0, -1) - 1$ または $rank_1(B_0, -1) + Z_0 - 1$ を計算する．この結果を j_1 とする．次に, 文字 c に割り当てられている数値の 1 番目のビットを調べる．先ほどと同様に, も

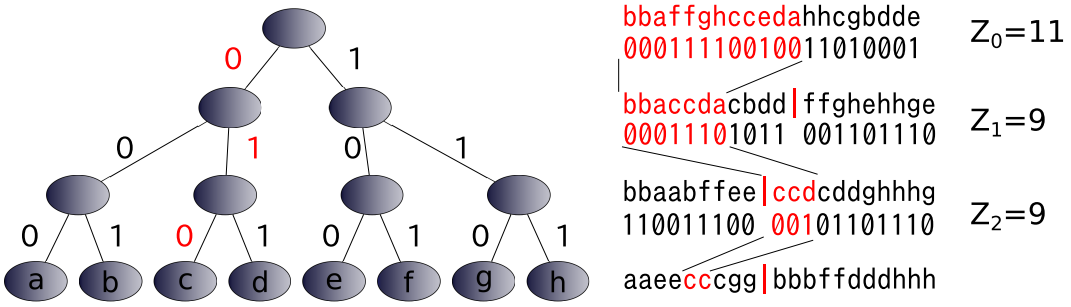

図 4.3. Wavelet 行列の rank 計算例

しこれが 0 であれば $rank_0(B_1, i_1) - 1$ を計算し, そうでなければ $rank_1(B_1, i_1) + Z_1 - 1$ を計算する. この結果を i_2 とする. また, やはり文字 c に割り当てられている数値の 1 番目のビットに応じて, $rank_0(B_1, j_1) - 1$ または $rank_1(B_1, j_1) + Z_1 - 1$ を計算する. この結果を j_2 とする. このような操作を繰り返すことで最終的に得られる $i_{\log\sigma}$, $j_{\log\sigma}$ に対して, $i_{\log\sigma} - j_{\log\sigma}$ が $\mathrm{rank}_c(T, i)$ の結果となる.

Wavelet 行列による rank の計算例を図 4.3 に示す. この例では $\mathrm{rank}_{\text{`c'}}(T, 11)$ を計算している. 'c' に割り当てられている数値は 010_2 である. まず 'c' の 0 番目のビットは 0 なので, B_0 に対して $\mathrm{rank}_0(B_0, -1)$, 及び $\mathrm{rank}_0(B_0, 11)$ を計算する. それぞれの結果は -1, 6 である. 次に, 'c' の 1 番目のビットは 1 なので, $\mathrm{rank}_1(B_1, -1) + Z_1 - 1$, 及び $\mathrm{rank}_1(B_1, 6) + Z_1 - 1$ を計算する. それぞれの結果は 8, 11 である. 最後に, 'c' の 2 番目のビットは 0 なので $\mathrm{rank}_1(B_1, 8) - 1$, 及び $\mathrm{rank}_1(B_1, 11) - 1$ を計算する. それぞれの結果は 3, 5 である. よって $\mathrm{rank}_{\text{`c'}}(T, 11)$ の結果は $5 - 3 = 2$ となる.

Wavelet 行列の空間計算量は, $\log\sigma$ 個の長さ n のビット列とそれに付随する補助的なデータ構造から成るため, $O(n\log\sigma) + o(n\log\sigma)$ ビットである. 長さ n のテキストの空間計算量が $O(n\log\sigma)$ であることを考えると, これは圧縮されていない. Wavelet 木については圧縮された状態で access, rank, select を行うための研究 [15][26] が成されているが, wavelet 行列は比較的新しい概念であるため, まだそのような研究は行われていない. しかし, 実際には wavelet tree と同様の圧縮手法が適用できる. 詳細は 7.6 節で述べる.

第5章　圧縮全文検索インデックス

5.1 FM-Index

接尾辞配列の空間計算量の問題を解決する方法として, 近年圧縮全文検索インデックスが注目されている. これは, 接尾辞配列とほぼ同等の機能を持ちながら, より小さい領域に収めることができるインデックスである. 圧縮全文検索インデックスは compressed 接尾辞配列 (CSA) 系 [18] と FM-Index 系 [14] の 2 つに大別されるが, 特に FM-Index 系は性能が高く, 注目を集めている. FM-Index 系では検索を行う際に二分探索ではなく backward search[14] と呼ばれる操作を行う. FM-Index 系のインデックスは全てこの backward search を省メモリかつ高速に実現することを目指すものである.

5.2 Backward Search

Backward search では, 検索対象の文字列を後ろから順に読んでいくことで検索を行う. Backward search では, 利用するのは主にテキスト T に BWT を適用した結果の文字列 T^{BWT} である. テキスト T からパターン P を探すという操作を考える. P は T を構成するのと同じアルファベット集合に含まれる文字から構成される, 長さ $m+1$ の文字列であるとする. ここで 1 を足しているのは説明の便宜のためであり, 本質的な要請ではない. まず, ソートされた suffix が $P[m]$ から始まる範囲 $[sp_m, ep_m]$ を求める. Backward search を行うときには接尾辞配列は構築しないが, 関数 C があれば以下のようにしてこの範囲を求めることが可能である.

$$[sp_m, ep_m] = [C(P[m]), C(P[m]+1) - 1] \tag{5.1}$$

ここで, 文字 c に対して $c+1$ はアルファベット集合 $\mathbf{\Sigma}$ において c の次に大きい文字であるとする. 次に, 上で得られた情報を元に $P[m-1, m]$ から始まる suffix の範囲 $[sp_{m-1}, ep_{m-1}]$ を求める. これには LF-mapping を用いる. 3.2 節と同様に F, L を定義すると, T^{BWT} の範囲 $[sp_m, ep_m]$ に現れる文字 $P[m-1]$ の順番は, F 中においても変わらない. b, e を $[sp_m, ep_m]$ において文字 $P[m-1]$ が現れる最初と最後の場所であると定義する. このとき, $LF(b)$, $LF(e)$ は $P[m-1, m]$ から始まる suffix の範囲を表すことになり, これは $[sp_{m-1}, ep_{m-1}]$ に等しい. 実際には b, e を求めることはできないが, $LF(b)$, $LF(e)$ は求めることができる. 実際, $rank_L(L[b], b)$, $rank_L(L[e], e)$ は以下のように計算される.

$$
\begin{aligned}
rank_L(L[b], b) &= rank_L(P[m-1], b) \\
&= rank_L(P[m-1], sp[m]-1) + 1 \qquad (5.2) \\
rank_L(L[e], e) &= rank_L(P[m-1], e) \\
&= rank_L(P[m-1], ep_m) \qquad (5.3)
\end{aligned}
$$

このような操作を繰り返すことで, 一般に $[sp_{i+1}, ep_{i+1}]$ から $[sp_i, ep_i]$ を求めることができ, 最終的に $[sp_0, ep_0]$ を求めることができる. これはまさに P から始まる suffix の範囲を表しており, 検索が完了したことになる. これが backward search である.

例を図 5.1 に示す. この例では, $T =$ “mississippi\$”, $P =$ “iss” である. 図には説明のために接尾辞配列や suffix 自身も表示しているが, 実際にはこれらは使用できない点に注意する必要がある. まず, “s” から始まる suffix の範囲を式 (5.1) で求める. すると, $[sp_2, ep_2] = [8, 11]$ であることがわかる. 次に, “ss” から始まる範囲を求める. これは $LF(b)$, $LF(e)$ を計算することで分かる.

$$
\begin{aligned}
LF(b) &= C(L_b) + rank_L(L_b, b) - 1 \\
&= C(P_1) + rank_L(P_1, sp_2 - 1) + 1 - 1 \\
&= C(\text{‘s’}) + rank_L(\text{‘s’}, 7) \\
&= 8 + 2 \\
&= 10 \\
LF(e) &= C(L_e) + rank_L(L_e, e) - 1 \\
&= C(P_1) + rank_L(P_1, ep_2) - 1 \\
&= C(\text{‘s’}) + rank_L(\text{‘s’}, 11) - 1 \\
&= 8 + 4 - 1 \\
&= 11
\end{aligned}
$$

よって $[sp_1, ep_1] = [10, 11]$ だと分かる. このような操作をもう一度行うことで “iss” から始まる suffix の範囲を求めることができ, 結果は $[sp_0, ep_0] = [3, 4]$ となる.

ここで, 式 (3.3) に注目してみよう. 関数 C は $\sigma \log n$ ビットの領域があれば容易に実現可能であるが, rank を高速かつ空間効率よく実現する方法は自明ではない. このような rank の計算を実現する方法の数々が, FM-Index 系のインデックスである. 最初の方法は Ferragina らによって提案されたが, 現在では wavelet tree や wavelet 行列がよく用いられている.

backward search の計算量について述べる. 接尾辞配列を用いた二分探索は, テキストサイズを n とすると, 文字列比較に最悪で $O(n)$ の時間が必要であるため, $O(n \log n)$ の最悪計算量となる. 一方, もし rank が定数時間で実現できれば, backward search の計算量はパターン P の長さ m に対して $O(m)$ となる. これは検索がテキストサイズによらないことを意味し, backward search の大きな特徴の 1 つであると言える.

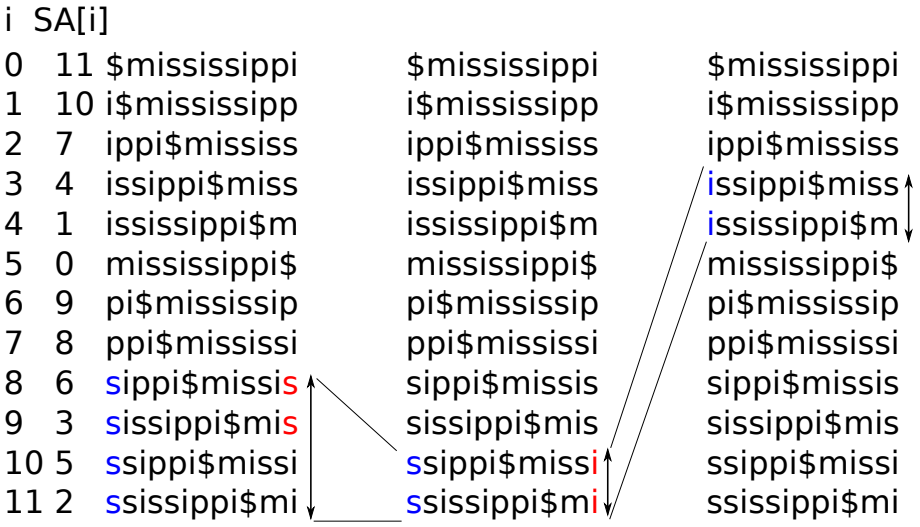

図 5.1. Backward search の例

5.3　パターンの出現位置特定

前節で説明した backward search により, 検索したいパターンが, 接尾辞配列の何番目から何番目に相当するかを求めることができることが分かった. しかし, これによりそのパターンの出現回数を知ることはできるが, 出現位置や, その周辺の文字列を取得するなどといったことはできない. これを行うためには, サンプリングされた接尾辞配列を利用する. つまり, テキストを一定間隔でサンプリングし, その場所の接尾辞配列のみを集めたもの, より具体的には, s をサンプリング間隔としたとき, $SA[i] = s \cdot j (j \leq n/s)$ となるような $SA[i]$ だけを集めた配列 SA' を利用する.

Backward search によって, パターンが現れる位置の接尾辞配列の範囲 $[i, j]$ が得られたとする. この範囲の接尾辞配列の値を取得するには, まず $SA[i]$ がサンプリングされているかどうかを調べる. もしサンプリングされていればその値を返す. そうでなければ $LF(i)$ を計算し, 先ほどと同様に $SA[LF(i)]$ がサンプリングされているかどうかを調べる. このようなことを繰り返し, $SA^r[LF(i)]$ がサンプリングされているような最小の $r \geq 0$ を求める. このとき, $SA[i] = SA[LF^r(i)] + r$ となる. LF-mapping は 3.2 節で説明したように, テキスト中で 1 つ前の位置から始まる suffix の順位を返す操作である. そのため, もし r 回の LF-mapping でサンプリングされた値を知ることができたら, もともとの $SA[i]$ はテキスト中でそれより r 個後ろの位置から始まる suffix であるため, 上で述べたような方法で計算できることになる.

問題はどの suffix がサンプリングされているかを, どのようにして知るのかということである. これには mark という長さ n のビット列を用いる. もし $SA[i]$ がサンプリングされていれば $\mathrm{mark}[i] = 1$ とし, そうでなければ $\mathrm{mark}[i] = 0$ とすれば, 接尾辞配列の各要素がサンプリングされているかどうかが瞬時に分かる. 実際の値を取り出す際には, $SA'[\mathrm{rank}_1(\mathrm{mark}, i)]$ を計算すればよい. 以上のような計算を順次 $SA[i], SA[i+1], \cdots, SA[j]$ に対して行うことで, パタンーンの出現位置を全て特定することができる.

接尾辞配列のサンプリングの例を図 5.2 に示す. この例では $T =$“mississippi$” に対して, 3 つ

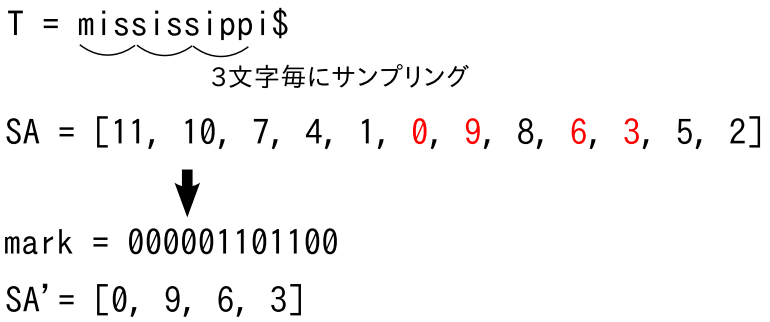

図 5.2. 接尾辞配列サンプリングの例

毎にテキストの位置をサンプリングしている．すると, 0, 3, 6, 9 という値が得られる．この値が接尾辞配列の中に出現する場所を探し, この位置のみ mark の値を 1 にする．その後, 0, 3, 6, 9 を接尾辞配列で出現する順番に格納し, それ以外の要素を破棄した SA' を構築する．このようにして接尾辞配列のサンプリングを行う．

5.4　Self-index としての性質

前節で説明した方法では, 接尾辞配列の値を取得するのに元となるテキストは一切使用していない．また, T^{BWT} を wavelet 行列のようなデータ構造に変換しておけば, テキスト中の任意の部分文字列を取得するのにすら元のテキストは必要ない．このように, FM-Index 系のインデックスは, 一旦構築してしまえば, 検索の際に元となるテキストは必要としない．このような性質を持つインデックスを self-index と呼ぶ．FM-Index 系のインデックスは元のテキスト以下のサイズでありながらパターンの検索が可能であるという, 非常に優れたインデックスであると言える．

第6章　関連研究

BWT を定義通りに計算しようとすると, 一度テキストに対して接尾辞配列を構築する必要がある. しかし, これでは圧縮全文検索インデックスの目的が達成されない. そこで現在までに, テキスト全体の接尾辞配列を構築することなく BWT を計算するための手法が数多く提案されてきた [16][19][20] [23][36]. 以下ではそのようなアルゴリズムをいくつか説明する.

6.1　メモリ上での BWT

6.1.1　サンプリングによる方法

Kärkkäinen はサンプリングによって suffix をいくつかのバケツに分類し, 分類したバケツを順にソートするという手法を提案した [20]. この手法では, まずいくつかの suffix をサンプリングし, ソートする. そして全ての suffix をサンプリングされた各 suffix と比較し, どの 2 つのサンプリングされた suffix の間に入るかで分類する. その後, 分類された各バケツに対して順に接尾辞配列の構築, BWT の計算を行う. こうすることで, テキスト全体に対する接尾辞配列の構築を防ぎ, メモリ使用量を削減できる.

この手法では, 文字列のソートにマルチキー・クイックソートを用いている. 文字列の比較で注意すべきは, 2 つの文字列の prefix が非常に長く一致すると, 比較に多くの時間を要してしまうという点である. これを防ぐために, この手法では difference cover sample (DCS) を利用している. DCS とは, 2.3.2 節で説明した DC3 の考え方を利用し, 2 つの suffix の順位を決定するまでに必要な文字の比較回数を制限する手法である. DC3 では 3 を法とする difference cover D について, $i \bmod 3 \in D$ となるような位置の suffix の順位を先に計算することで, $i \bmod 3 \notin D$ の suffix の順位を高速に計算できるのであった. これを利用し, 一般に v を法とする difference cover D に対して, $i \bmod v \in D$ となるような位置の suffix の接尾辞配列を構築したものが DCS である. DCS を構築すると, 任意の 2 つの suffix の順位を計算するために, 最悪でも $O(v)$ 回の文字比較を行えば, 双方がすでに順位の決定されている suffix の先頭位置に到達することが保証される. pp この手法の最悪計算量は $O(n(\log n + \log_\sigma^2 n))$, 平均計算量は $O(n \log n)$ である. また空間計算量は $O(n \log \sigma)$ である.

6.1.2　BWT-IS

岡野原らは, 接尾辞配列の構築アルゴリズムである SA-IS を BWT の構築に応用した BWT-IS[36] を考案した. この手法では, SA-IS において使用される L 型, S 型, 及び, LMS 型の概念を BWT に対しても適用し, まず LMS 型の BWT を構築し, そこから L 型, S 型の BWT を順に計算する. このようにすることで, BWT を $O(n)$ の時間計算量で構築できる.

岡野原らの手法では, L 型, S 型の BWT を導出する際に, 各イテレーションの計算がそれ以前のイテレーションに依存する形になっており, 並列化を行うことは難しい.

この手法の時間計算量は $O(n)$, 空間計算量は $O(n \log \log n)$ となる.

6.2　ディスク上での BWT

ディスク上での BWT の計算手法として, Ferragina らの手法 [16] がある. この手法では, テキストをメモリサイズに応じてブロックに切り分け, それらを後ろから順に処理し, 結果をマージしていく. また, 2 つの suffix を比較する際に, それらの prefix が長く一致してしまうと比較に無視できない時間を要する問題があるが, 予め一部の suffix 間の順位を計算しておくことで, これを回避している.

より具体的には, まずテキストをサイズ m のブロックに分割する. すなわち, $T = T_{n/m-1}$, $T_{n/m-2}, \cdots, T_0$ とする. そしてまず T_0 をメモリにロードし, BWT を計算する. その後, これをディスクに書き戻す. 次に T_1 を読み出し, BWT を計算する. そして先ほど作った T_0 の BWT とマージを行う. マージの結果は処理の途中で随時ディスクに書き戻していく. このようにして, 一般にブロック T_{h+1} をメモリから読み出し, その BWT を計算後, $T_h T_{h-1} \cdots T_0$ の BWT とマージを行うことで, T の BWT を得ることができる.

この手法では, ディスク上にあるテキストや途中結果の BWT を読みだす時にシーケンシャルアクセスしか行わないため, 非常に効率的である. またそれにより, 途中状態の BWT を圧縮して扱うことが可能になるため, これによりディスク I/O を削減することができる. この手法の時間計算量は $O(n^2)$ となる.

この手法は BWT をディスク上で計算する方法として優れている. しかし, BWT を使って行われる検索手法である backward search はランダムアクセスが非常に多いという性質があり, ディスク上での利用にはまだ困難があると言える.

6.3　分散環境での BWT

分散環境での BWT の計算として, MapReduce[13] を用いた Menon らの方法 [28] が挙げられる. MapReduce とは, Google によって大規模データ処理のために考案されたプログラミングモデルである. MapReduce では全体の処理が map, shuffle, reduce の 3 つのフェーズから成っており, これらの処理をユーザが記述するだけで比較的容易に並列計算を行うことが可能となる. MapReduce のオープンソースの実装として Hadoop があり, Menon らはこれを利用して BWT を分散環境で計算している.

分散環境での利点として, 複数台のコンピュータを使うことで使用できるメモリ量が増大することが挙げられる. Menon らの手法では接尾辞配列を分散環境で構築し, 各ノードで BWT を定義から計算する. こうすることで, 1 つのマシンのメモリ上で接尾辞配列を構築せずに済むので, 結果としてメモリが問題でなくなる.

Menon らの手法では, まず suffix の中からいくつかのサンプルを選び, それをソートする. そして, map フェーズで全ての suffix が辞書的にそれらのどの 2 つの間に位置するかを計算する. map フェーズでの計算結果を元に, 各 suffix を shuffle フェーズで適切なノードに割り振り, 最後

に reduce フェーズでソートを行う. 基本的な動作は以上の通りであるが, 同じ文字が連続して現れる領域を run-length 符号化するなどの工夫が行われている.

この手法ではテキストを Hadoop 付属のファイルシステムである HDFS (Hadoop Distributed File System) に格納している. 分散ファイルシステムとは, ファイルを細かいチャンクと呼ばれる単位に分割し, 各チャンクをそれぞれのノードに分けて配置するというものであり, ファイルアクセスの際にはその領域のチャンクを持っているノードと通信を行う必要がある. そのため, reduce フェーズで suffix のソートを行う際には, 全てのノードが全てのノードと通信する可能性があり, スケーラビリティに限界があるものと思われる. 一般的に, 接尾辞配列を分散環境で構築する際には元となるテキストにある程度ランダムにアクセスすることになるため, 効率的な計算が難しい.

6.4　既存手法の性能比較

ここまで説明した既存手法の性能の一覧を表 6.1 に示す. ここで, p はコア数を表している.

表 6.1. 既存手法の性能比較

アルゴリズム	時間計算量	空間計算量
サンプリング (6.1.1 節)	$O(n \log n)$	$O(n \log \sigma)$
BWT-IS (6.1.2 節)	$O(n)$	$O(n \log \log n)$
ディスク上での計算 (6.2 節)	$O(n^2)$	$O(n \log \sigma)$
MapReduce (6.3 節)	$O(n \log n/p)$	$O(n \log n)$

第7章　分割統治法によるBWTの計算

7.1　概要

本研究では分割統治法によるメモリ内での BWT の並列計算法を提案する．提案手法ではまずテキストを再帰的に分割し, ある程度のサイズになったところでその部分文字列の BWT を計算する．続いてそれらを再帰的にマージしていく．アルゴリズムの概要を図 7.1 に示す．

各再帰呼び出しは範囲 $[i, j]$ を受け取り, (擬似コードでは begin, end で示されている) その範囲の接尾辞配列を構築する．つまり, 範囲 $[i, j]$ が与えられたら, $T[i, n]$, $T[i+1, n]$, ..., $T[j, n]$ の接尾辞配列を構築し, そこから BWT を計算する．そのため, T^{BWT} の i 番目の文字は辞書順で i 番目の suffix の, テキスト中で 1 つ前の文字ということになり, 得られる T^{BWT} は $T[i-1], T[i], \ldots, T[j-1]$ の並び替えということになる．

このアルゴリズムの重要な点は, 2 つの隣接した部分文字列に対する BWT を小さいメモリのみを使ってどのようにマージするかということである．各部分文字列の接尾辞配列を全て保持し, それを利用してマージするのが最も単純な方法であるが, これでは $O(n \log n)$ という空間計算量を避けることができない．そのため, ここではサンプリングされた接尾辞配列を利用する．

もう一つ重要な点として, 2 つの suffix を比較する際に最悪の場合 $O(n)$ の時間が必要になるという問題がある．この問題を解決するために, 提案手法では difference cover sample (DCS)[20] を利用する．DCS を用いることで, 2 つの suffix の大小を決定するために必要な最悪の場合での文字比較の回数を制限することが可能になる．

以下ではまず DCS について改めて説明し, BWT が再帰呼び出しのリーフでどのように計算するかを述べた後, それらのマージ方法について説明する．

7.2　Difference Cover Sample

Difference cover sample (DCS) とは, v を法とする difference cover D に対して, $i \bmod v \in D$ となるような位置の suffix の接尾辞配列を構築したものである．DCS で実際に選ばれる位置の数は, $O(n/\sqrt{v})$ となる．$v \propto \log^2 n$ となるように v を選べばこれは $O(n/\log n)$ であり, 一要素あたり $\log n$ ビットのメモリが必要であるため, 結局 DCS が必要とするメモリは $O(n)$ ビットとなる．

v の値が大きい場合, difference cover を構築する方法が問題となる．実際, 大きい v について v を法とする最小の difference cover を求める方法は知られていない．しかし, 理論的に最小な difference cover の定数倍の要素数を持つものを構築する方法 [11] は知られている．

DCS の構築は, DC3 において $i \bmod 3 = \{1, 2\}$ となる suffix に対して接尾辞配列を構築する方法とほぼ同様に行うことができ, その計算量は $O(vn)$ であることが知られている [21]．しかし, こ

```
1  void ComputeBWT(int begin, int end, Node& parent){
2    if(inputSize<=LEAF_SIZE){
3      ComputeLeaf(begin, end, globalBegin, globalEnd, parent);
4      return;
5    }
6    Node left, right;
7    int mid=inputSize/2;
8    // 関数の再帰呼び出し
9    ComputeBWT(begin, begin+mid, left);
10   ComputeBWT(begin+mid, end, right);
11   // 左側と右側の子ノードをマージ
12   MergeNode(left, right, parent, left.begin(), left.end(), right.begin(), right.end());
13 }
```

図 7.1. アルゴリズムの概要

こでは後に DCS の構築も並列化することを考え, 一般的に知られている方法を一部変更したものについて説明する.

まず, $i \bmod v \in D$ となる位置の suffix を先頭 v 文字でソートする. これは DC3 の場合と同様に radix sort によって $O(vn)$ で行うことができる. しかし, 今は $v \propto \log^2 n$ としているため, これでは $O(n \log^2 n)$ の計算量となってしまう. そのため, ここでは radix sort は使わず, マルチキー・クイックソートによってソートを行う. もしこれによって順位が一意に決まれば, DCS の構築はこれで終了である.

マルチキー・クイックソートで順位が一意に決まらなかった場合, 再帰的に処理を行うことにより順位を決定する. まず, D の要素を小さい方から順に $d_0, d_1, \cdots, d_{v-1}$ とし, これらを順に見ていく. そして, $i \bmod v \in D$ となる位置の suffix について, $i \bmod v$ が d_0 と一致するものを, i の値が小さい順に並べたものを考える. そして, それらを先ほどの先頭 v 文字のソートで得られた順位で置き換える. さらにその後ろに $i \bmod v$ が d_1 と一致するものを, i の値が小さい順に並べたものを付加する. そして, それらを先ほどと同様に順位で置き換える. 同様のことを d_{v-1} まで行うことで出来上がる数値列を, 新たな入力文字列とする. この新たな入力文字列に対して, 通常なら再帰的に計算を行うのだが, v を法とする difference cover を利用する場合, $i \bmod v \in D$ の suffix と $i \bmod v \notin D$ の suffix をマージする処理が, 3 を法とする場合と比べて複雑になる. また, この新たな入力文字列については, 単に接尾辞配列が計算できればよい. そのため, ここでは DCS 構築の処理を再帰的に呼び出すのではなく, DC3 の処理を再帰的に呼び出すことで簡略化する. このようにして, DCS を構築することができる.

以上のような方法で DCS を構築した場合, 時間計算量は最初のソートの部分がボトルネックとなり, 平均で $O((n/\log n)\log(n/\log n)) = O(n)$ となる. 最悪の場合では, $O((n/\log n)\log^2(n/\log n)) = O(n \log n)$ である.

```
1  void ComputeLeaf(int begin, int end,
2    int globalBegin, int globalEnd, Node& leaf){
3    vector<int> sa;
4    // 接尾辞配列を構築
5    leaf.StringSort(begin, end, globalEnd, sa);
6    // BWT を計算
7    for(int i=0; i<leaf.bwt.size(); i++){
8      if(globalBegin==begin && sa[i]==0)
9        leaf.bwt[i]=$;
10     else leaf.bwt[i]=T[begin+sa[i]−1];
11   }
12   if(!leaf.isRootNode() && leaf.isLeftNode())
13     // 左側子ノードに対してのみwavelet 行列を構築
14     leaf.ConstructWaveletMatrix();
15   SampleSA(sa, leaf.sampledSA, leaf.mark);
16 }
```

図 7.2. 部分文字列への BWT 計算

7.3　リーフに対する BWT

提案手法では, 入力テキスト T の範囲 $[0, n]$ をある閾値以下の大きさになるまで再帰的に分割する. そして, そのようにして得られた部分文字列に対して接尾辞配列を構築し, そこから BWT を計算した後, 接尾辞配列のサンプリングを行う. 図 7.2 に擬似コードを示す.

各 suffix は実際にはテキストの最後まで続いている. これにより suffix の比較が部分文字列内で完結しないため, 接尾辞配列の構築に通常用いられる効率的なアルゴリズムをそのまま用いることが難しい. そこで, マルチキー・クイックソートなどの文字列ソートアルゴリズムを用いることが考えられる. しかし, 単なる文字列ソートアルゴリズムでは, 隣接する suffix 間に存在する強い依存関係を利用することができない. そこで, 既存の高速な接尾辞配列構築アルゴリズムのアイディアを用いて, 可能な限りこの依存関係を利用することを考える.

既存の高速な接尾辞配列構築アルゴリズムである DC3 や SA-IS では, まずテキストの一部分に対して接尾辞配列を構築し, その情報を元に残りの部分の suffix の順位を効率的に計算するという流れで行われる. このうち, テキストの一部に対する接尾辞配列を構築する段階においては, 再帰的に副問題を生成し, それを解くことで $O(n)$ の計算量を達成している. しかし, 部分文字列に対しては, 再帰することでアルファベットが変化する上に, その後ろの文字列まで考慮する必要があるため, 副問題の生成が煩雑になる. そこで, この部分には通常の文字列ソートアルゴリズムを用い, 残りの部分はこれら高速なアルゴリズムのアイディアを用いて計算することにする. このような方法では理論的な計算量は依然として文字列ソートを用いる場合と変わらないが, 文字列ソートを行わなければならない suffix の割合が減少するため, 定数倍の高速化が期待できる. 今回はより高速である SA-IS を元にした部分文字列のソートについて考える.

まず, SA-IS に従い suffix を S, L, LMS 型に分類する. そして, LMS 型の suffix をマルチキー・クイックソートによってソートする. 次に, ソートされた LMS 型の suffix の位置情報を, その先頭アルファベット毎にバケツに分類する. SA-IS の場合と同様, バケツはさらに L 型, S 型の領域に

二分できるので, S 型領域に入れる.

次に, バケツの要素を先頭から順に見ていく. 通常の SA-IS では, もしある要素のテキスト中での一文字前の位置の suffix が L 型だった場合, それをバケツに入れるという処理を行うが, 現在考えている状況では, 部分文字列の最後の LMS 型 suffix よりもテキスト中で後ろに位置する suffix が考慮されなくなるという問題がある. L 型の suffix の順位を計算する際にこのような問題が起こるのは, 部分文字列の最後の suffix が L 型の場合のみなので, そのような場合にのみ, 末尾に並ぶ L 型 suffix の順位を別途計算する必要がある. そのための方法として, まず末尾に並ぶ L 型の suffix がそれぞれどの文字から始まっているか調べる. また, 連続した L 型 suffix の領域においては, 同じ種類の文字は連続して現れるため, それぞれの文字が一番最後に出現する場所を記録しておく. そして, 通常の SA-IS のステップに従ってバケツ内の要素を見ていき, もしある要素のテキスト中での一文字前の位置の suffix が L 型であり, かつその先頭文字が末尾の L 型領域に含まれていたとする. 仮にこの文字を c とする. このとき, 当該の suffix と c が最後に現れる場所から始まる suffix を比較し, 後者が小さい間は後者をバケツに入れ, その位置を記録したポインタを 1 つ手前に移動させるということを繰り返す. 前者の方が大きくなったところで, 前者をバケツにいれ, c については処理を終了したとして, 処理済みのフラグを立てる. もし後者が一回もバケツに入らなかった場合, 再び c から始まる suffix が現れたときに同様の処理を行う. また, もしその後一度も後者の suffix がバケツに入らなかった場合, バケツの要素を全て走査した後にこれらを後ろから順にバケツに入れる. このようにすることで, L 型の suffix の順位を全て計算することができる.

最後にバケツの要素を後ろから順に見ていき, S 型の suffix を処理する. この場合も同様に, もし末尾の suffix が S 型である場合のみ, 別途これらの順位を計算する必要がある. 方法は L 型の場合とほぼ同様である.

上記の方法による接尾辞配列の計算例を図 7.3 に示す. この例では対象となる部分文字列は T' =“AGTTAATG” であり, その後ろには “GA$\cdots$” が続いているとする. まず, LMS 型の suffix をマルチキー・クイックソートによりソートする. 図で LMS 型は S*として表記している. この例では LMS 型は 4 のみである. 次に, ソートした LMS 型 suffix の開始位置を, 先頭文字に応じてバケツに入れる. バケツの要素を先頭から順に見ていくと, 4 の 1 つ前の位置である 3 は L 型であることが分かる. 通常の SA-IS であればここで 3 をバケツに入れるのだが, 今回は部分文字列の末尾に L 型の領域が存在するため, ここに存在する suffix とも比較を行う必要がある. 末尾の L 型領域において, 3 番目の位置の suffix と同じ ‘T’ から始まる suffix のうち, 一番後ろにあるものと比較をする. 例では 6 番目の suffix1 つのみである. これを比較すると, 3 番目の suffix の方が辞書順で小さいので, 3 をバケツに入れる. 再びバケツの要素を順位見ていくと, 次は先ほどバケツに入れた 3 に遭遇する. 3 の 1 つ前の位置は 2 である. 2 番目の位置の suffix は先ほどと同じく L 型かつ ‘T’ で始まっており, 末尾の L 型領域で ‘T’ から始まるものはまだ未処理であるため, 2 番目の suffix と 6 番目の suffix を比較する. すると, 今度は 6 番目の suffix の方が辞書順で小さいため, 6 をバケツに入れる. これにより, 末尾の L 型領域のうち ‘T’ で始まる suffix の処理が完了したため, 本来入れるべき 2 をバケツに入れる. この例では’T’ から始まる末尾の L 型 suffix は 1 つしかなかったが, もしこれらが複数存在した場合は, それらを後ろから順に 2 番目の suffix と比較し, 2 番目の方が小さくなるか, 要素がなくなるまで比較を続けることになる. バケツを最後まで走査したら, まだ未処理である末尾の L 型領域の suffix をバケツに入れる. この例では 7 番目の suffix が未処理であったため, これをバケツに入れる. 複数存在する場合は後ろのものから順に先に入れていくことになる. 以上で L 型の suffix のソートが完了する. S 型についても同様にソートを行うこ

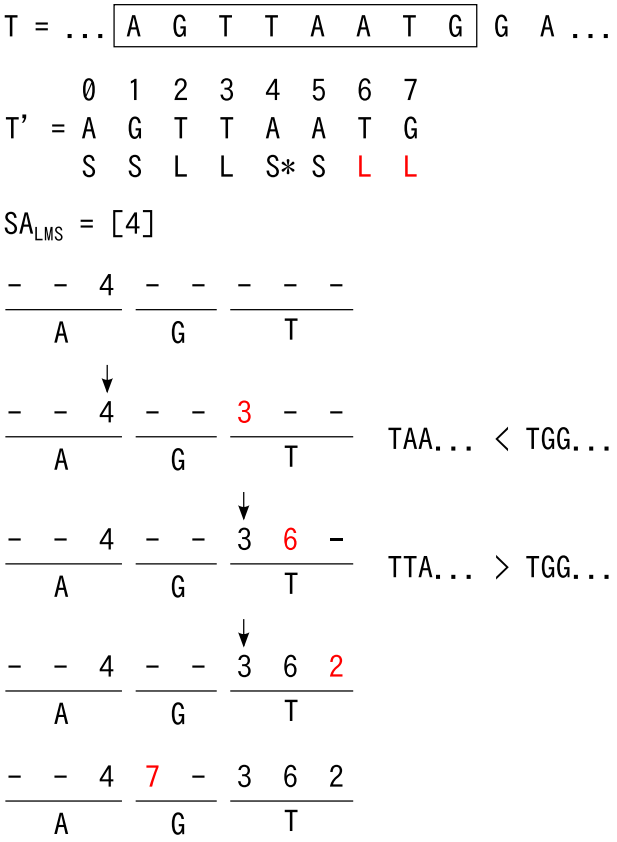

図 7.3. リーフにおける接尾辞配列の計算例

とができる.

1 つ注意しなければならないのは, 通常の文字列の場合と違い, 部分文字列の場合は LMS 型の suffix が 1 つも存在しない場合も存在するということである. そのような場合は以下の 3 つに分類できる.

1. 全て L 型である.
2. 全て S 型である.
3. S 型が 1 つ以上並んだ後, L 型が並ぶ.

1 の場合, suffix の辞書順は単調減少であるため, 逆順に並べればよい. 2 の場合, suffix の辞書順は単調増加であるため, そのままの順番でよい. 3 の場合, まず L 型の suffix を逆順にバケツに入れる. その後, 先ほど説明したような方法で S 型 suffix の順位を計算する. このようにして, 部分文字列の接尾辞配列を構築することができる.

上記の方法の計算量について考える. 計算量において支配的となるのは, LMS 型 suffix のソートと, 末尾の L 型, または S 型領域との比較である. LMS 型 suffix のソートの最悪計算量は, 部分文字列の長さを m とすると $O(m \log^2 n)$ である. $\log^2 n$ は DCS によって比較回数を抑えていることによる項である. また, 末尾の L 型, または S 型領域との比較比較では, 最大でテキストのほぼ全体が末尾までの連続領域となる可能性があるため, 最悪計算量は $O(m \log^2 n)$ である. よって, こ

の方法での最悪計算量は $O(m \log^2 n)$ となる．そのため, 最悪計算量はマルチキー・クイックソートと変わらない．しかし, 冒頭にも述べた通り実際にはこちらの方法の方が, 最悪でないケースにおいては定数倍高速であると思われる．

部分文字列に対する接尾辞配列が構築できたら, 次は BWT を計算する．これは BWT の定義式 (3.1) により線形時間で求めることができる．

最後に, 接尾辞配列を 5.3 節で述べた方法によってサンプリングする．後に BWT をマージする際に SA が必要となるが, SA は $O(n \log n)$ ビットのメモリを必要とする．サンプリングはこのメモリ使用量を抑えるために行う．サンプリング間隔によって, メモリ使用量と接尾辞配列の要素の復元速度の間にトレードオフが生じる．サンプリング間隔を s としたとき, 接尾辞配列の要素の値を復元するのに平均して必要な時間は $O(s \log \sigma)$ であり, サンプリングされた接尾辞配列と mark ビット列に必要なメモリはそれぞれ $O(n \log n/s)$ ビット, $O(n)$ ビットである．$o(n \log n)$ の空間計算量を達成するため, 提案手法では $s \propto \log n$ とする．

7.4　部分文字列における LF-mapping

前節で述べたとおり, 後のマージ処理において接尾辞配列が必要であるが, 空間計算量を小さくするために, それらはサンプリングされる．サンプリングされた接尾辞配列から元の値を復元するためには LF-mapping を行う必要があるが, 部分文字列においては, そのままでは LF-mapping を計算することができない．これは, LF-mapping において部分文字列の最後から始まる suffix に対応する T^{BWT} の要素が存在しないことに起因する．表 7.1 に例を示す．この例では “isssissizz$” という文字列のうち, “isssiss” という部分文字列に注目している．この例だと, 例えば $LF(2) = 3$ となっているが, 2 番目の suffix である “sissizz$” を一文字前に伸ばした suffix は “ssissizz$” であるはずなのに, 3 番目の suffix は “sizz$” となっており, これは誤りである．このような現象が起こる理由は, 部分文字列の最後から始まる suffix である “sizz$” の先頭文字 ‘s’ に対応する文字が L 中に存在しないためである．

部分文字列に対する LF-mapping が必要になるのは, サンプリングされた接尾辞配列から元の接尾辞配列の値を復元するときだけである．そこで, 接尾辞配列の先頭要素は必ずサンプルとして選択するようにしておけば, 通常の LF-mapping のように文字列を巡回する必要はない．つまり, 後ろから前へ部分文字列を順に辿ることができればよいので, 部分文字列の最後から始まる suffix に対応する番号に LF-mapping によって到達することはない．よって, 部分文字列の最後の位置については, LF-mapping の rank 計算から除外する．こうすることで, 部分文字列に対しても正しく LF-mapping を行うことができるようになる．

修正された LF-mapping の例を表 7.2 に示す．この例では, 0 番目, 及び 2 から 4 番目の L の値が ‘s’ である．このうち $L[0]$ については対応する要素が 2 番目の suffix の先頭文字で正しいが, $L[2]$ に対応する文字は 4 番目の suffix の先頭文字である．3 番目の suffix は注目している部分文字列の最後の文字を先頭とする suffix であり, LF-mapping でこの suffix に到達する必要はない．そこで, 3 番目の suffix を無視するために $LF(2)$ の値を 1 つ進める．$LF(3)$, 及び $LF(4)$ についても, この影響により LF-mapping の値が 1 つずれることになる．

7.5　部分文字列に対する BWT のマージ

再帰呼び出しの内部のノードでは, 1 つ下の深さの 2 つの再帰呼び出しの結果をマージする. 図 7.4 に擬似コードを示す. T^{BWT} をマージするためには, サンプリングされた接尾辞配列と mark ビット列も必要となる. 再帰的にマージを行うためには, これらに対してもマージを行う必要がある. これら 3 つの配列はほぼ同様にマージできるため, ここでは T^{BWT} のマージについてのみ説明する.

以下で扱う隣接する部分文字列をそれぞれ T_l, T_r, それらに対する BWT をそれぞれ T_l^{BWT}, T_r^{BWT} とする. 長さはそれぞれ m_l, m_r であるとする. 最も簡単な T_l^{BWT}, T_r^{BWT} のマージ方法は, それぞれの接尾辞配列の要素を先頭から復元し, 小さい方を順番に出力用の配列に格納していくことである. しかし, 全ての要素を復元するためには $O((m_l + m_r) \log \sigma \log n)$ の時間が必要である. さらに suffix の比較の度に最悪で $O(\log^2 n)$ の時間がかかるため (DCS により比較回数を抑えていることに注意), 全体として $O((m_l + m_r) \log \sigma \log^3 n)$ の時間を要することになる. これを削減するために, 提案手法では Ferragina らの定理 [16] を利用する.

定理 1. $T_l[i, n]$(または $T_r[i, n]$) を, 部分文字列 T_l(または T_r) の i 番目の文字から始まり, テキストの最後まで続く suffix であるとする.

$$T_l[SA[i], n] < T_r[t, n] < T_l[SA[i + 1], n] \tag{7.1}$$

表 7.1. LF-mapping が計算できない例

i	sorted suffixes	L	LF(i)
0	issizz$	s	2+1-1=2
1	isssissizz$	-	-
2	sissizz$	s	2+2-1=3
3	sizz$	s	2+3-1=4
4	ssissizz$	s	2+4-1=5
5	ssizz$	i	0+1-1=0
6	sssissizz$	i	0+2-1=1

表 7.2. 部分文字列のために変更された LF-mapping

i	sorted suffixes	L	LF(i)
0	issizz$	s	2+1-1=2
1	isssissizz$	-	-
2	sissizz$	s	2+2-1+1=4
3	sizz$	s	2+3-1+1=5
4	ssissizz$	s	2+4-1+1=6
5	ssizz$	i	0+1-1=0
6	sssissizz$	i	0+2-1=1

```
1  void ConstructGap(DynamicSizeVec &gap, Node &left, Node &right,
2      int lBegin, int lEnd, int rBegin, int rEnd){
3    int c, idx;
4    // T[idx] < T[rEnd − 1] < T[idx + 1] となる idx を探索
5    idx=StringBinSearch(left, lBegin, rEnd−1);
6    gap[idx+1]++;
7    for(int i=right.size()−2; i>=0; i−−){
8      c=T[rBegin+i];
9      idx=left.C(c)+left.rank(c, idx)−1;
10     // 式7.4
11     if(c==T[lEnd−1] && StringLessThan(rBegin, rBegin+i+1)) idx++;
12     gap[idx+1]++;
13   }
14 }
15
16 void MergeBWT(DynamicSizeVec &gap, Node &left, Node &right, Node &parent){
17   int pBWTIdx=0, rIdx=0;
18   for(int j=0; j<gap[0]; j++){
19     if(right.isSA0(rIdx)) parent.bwt[pBWTIdx++]=T[lEnd];
20     else parent.bwt[pBWTIdx++]=right.bwt[rIdx++];
21   }
22   for(int i=1; i<gap.size(); i++){
23     parent.bwt[pBWTIdx++]=left.bwt[i];
24     for(int j=0; j<gap[i]; j++){
25       if(right.isSA0(rIdx)) parent.bwt[pBWTIdx++]=T[lEnd];
26       else parent.bwt[pBWTIdx++]=right.bwt[rIdx++];
27     }
28   }
29   if(!parent.isRoot() && parent.isLeftNode())
30     // 左側子ノードに対してのみ wavelet 行列を構築
31     parent.ConstructWaveletMatrix();
32 }
33
34 void MergeNode(Node &left, Node &right, Node &parent,
35   int lBegin, int lEnd, int rBegin, int rEnd){
36   // gap 構築
37   DynamicSizeVec gap;
38   ConstructGap(gap, left, right, lBegin, lEnd, rBegin, rEnd);
39   // 3つの配列をマージ
40   MergeBWT(gap, left, right, parent);
41   MergeSampledSA(gap, left, right, parent);
42   MergeMark(gap, left, right, parent);
43 }
```

図 7.4. BWT のマージ

が成立しているとき, $T_r[t-1,n] = cT_r[t,n]$ の挿入位置が以下で表されるとする.

$$T_l[SA[j],n] < T_r[t-1,n] < T_l[SA[j+1],n] \tag{7.2}$$

このとき j は以下の式で求められる.

$$j = \begin{cases} C[c] + \mathrm{rank}_c(T_l^{BWT}, i) + b & (\text{if } c = T_l[m_l - 1]) \\ C[c] + \mathrm{rank}_c(T_l^{BWT}, i) & (\text{otherwise}) \end{cases} \tag{7.3}$$

ただし p は以下のように定める.

$$b = \begin{cases} 1 & (\text{if } T_r[0, n] < T_r[t, n]) \\ 0 & (\text{otherwise}) \end{cases} \tag{7.4}$$

証明の概略を述べる. $T_l[SA[i], n] < T_r[t, n] < T_l[SA[i+1], n]$ を満たすような suffix $T_r[t, n]$ について, $T_r[t, n]$ を一文字前に伸ばした $T_r[t-1, n]$ の挿入位置を考える. $T_r[t-1, n]$ の先頭文字を c とすると, $T_r[t-1, n]$ は c より小さい文字から始まる suffix よりは辞書順で大きくなる. よって $T_r[t-1, n]$ の挿入位置 j は $C[c]$ よりも大きい. 次に, 同じ文字 c から始まる suffix の中で, $T_r[t-1, n]$ が何番目に大きいのかを考える必要がある. ここで, ソートされた suffix の先頭文字を集めた配列と T^{BWT} において, 同じ種類の文字は同じ順番に一対一対応することを考えると, c から始まり $T_r[t-1, n] = cT_r[t, n]$ よりも辞書順で小さい suffix の数は, T^{BWT} 中で i よりも前に存在する c の数に等しくなる. ただし, T_l の最後の文字が c である場合には注意が必要である. T^{BWT} はテキスト中の文字を, それに後続する suffix をキーとして並び替えたものであり, T_l の後ろには suffix が存在しない. そのため, 最後の文字が c の場合, 最後から始まる suffix のみ直接的に文字列比較で大小を決める必要がある.

図 7.5 に例を示す. この例では T_l =“isiis”, T_r=“sip$\cdots$” である. もし “p$\cdots$ の挿入位置が 2 だと分かったとする. このとき, それより一文字前から始まる suffix である “ip$\cdots$” の挿入位置は, 図中段右に示すような計算式により求まる. さらに一文字前から始まる suffix である “sip$\cdots$” の挿入位置は, 同様に図下段右に示すような計算式により求まる. ただし, この suffix の先頭文字は ‘s’ であり, さらに T_l の最後の文字も ‘s’ であるため, この suffix と T_l の最後から始まる suffix の大小は直接比較しなければ分からない. ただし, どちらも同じ文字から始まることは分かっているので, 実際には一文字後ろから文字の比較を始めれば十分である.

この定理によって, $T_r[m_r - 1]$ から始まる suffix の挿入位置さえ分かれば, T_r^{BWT} の各要素がそれぞれ T_l^{BWT} のどこに挿入されるのかが帰納的に分かる. 各位置に挿入される要素数をカウントしたサイズ $(m_l + 1)$ の配列を, Ferragina らの論文に習い, 以下では gap と呼ぶ. Ferragina らの定理を利用すれば, gap を構築することが可能となり, それを用いて T_l^{BWT}, T_r^{BWT} をマージすることができるようになる. しかし, gap にはそのままだと $O(m_l \log m_r)$ ビットのメモリが必要である. gap の要素は非負の整数であること, そして gap の要素の総和は m_r に等しいことを利用すると, gap をより空間効率のよい形に圧縮することが可能だが, 一般的に用いられるような圧縮手法だと, 動的に更新することが難しい. そこで, メモリ使用量をある程度削減しながら, 動的な更新も可能であるような gap の実現方法を考える.

まず, gap の各要素に p (> 0) ビットを割り当てる. $2^p - 1$ よりも小さい値は gap 配列に直接格納する. もしある要素が $2^p - 1$ 以上になった場合, gap 配列には $(2^p - 1)$ という値を格納し, 要素番号を値へとマッピングする key-value ペアを作成する. これらの key-value ペアはハッシュテーブル上に保持される. そのようなペアを作成するためには $2 \log m_r$ ビットのメモリが必要である. メモリ使用量を推定するために, 最悪の場合でどれだけの key-value ペアが作成されるかを考える.

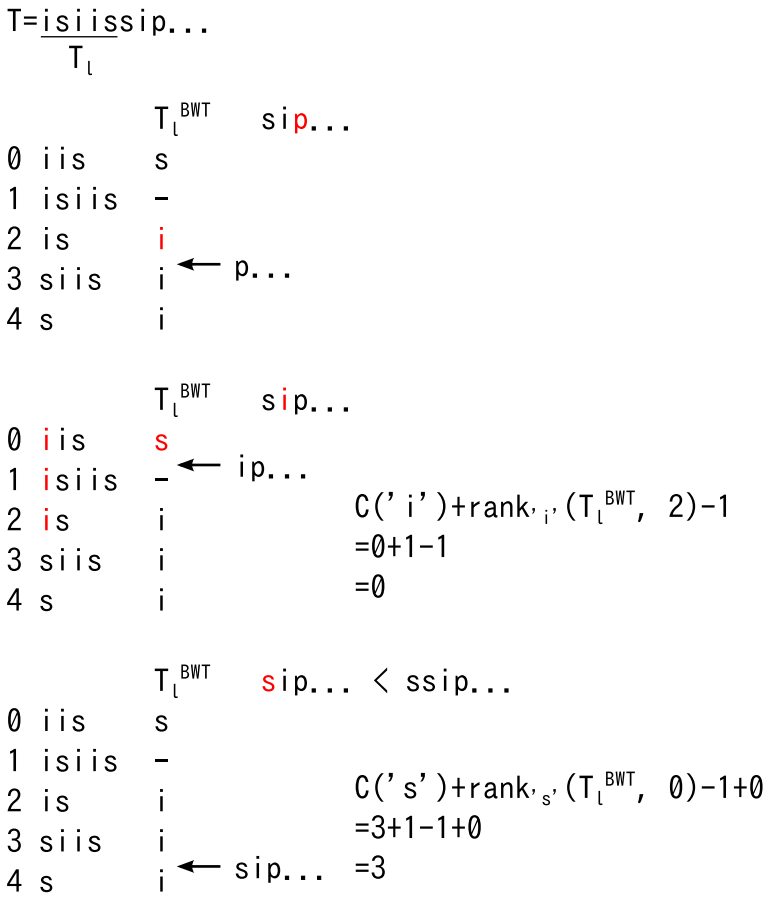

図 7.5. Ferragina らの定理の適用例

gap の全要素の和は m_r に等しいため, 最悪の場合に作成される key-value ペアの数は $m_r/(2^p-1)$ である. この場合, メモリ使用量は $m_l p + 2m_r \log m_r/(2^p-1)$ ビットとなる. ここで $p \propto \log\log m_r$ とすれば, メモリ使用量は $O(m_l \log\log m_r)$ となる. 擬似コードでは, このような gap のデータ構造を表す型を DynamicSizeVec として表している.

gap の例を図 7.6 に示す. この例では, まず一つ一つの要素を 2 ビットで表しているとする. つまり, 各要素は 0 から 3 までの値を保持することができ, 値が 3 以上になったときは key-value ペアをハッシュテーブルに追加する. まず始めに $[1,0,2,0,0,2,0,1]$ という数値配列があるとする. 5 番目の要素に 1 を足すと, 値が 3 になる. 2 ビットで表した場合, 3 は最大の数値となるため, ここでハッシュテーブルに要素番号と値を追加する. 次に, 2 番目の要素に 1 を足す. すると, やはり値が 3 になるため, ハッシュテーブルに要素番号と値を追加する. ここで, さらに 2 番目の要素に 1 を足すことを考える. 2 番目の要素はすでに値が最大の 3 になっているので, 値がハッシュテーブルに登録されていることが分かる. よってハッシュテーブルから key-value ペアを取得し, value を更新する.

$T_r[m_r-1]$ から始まる suffix の挿入位置を求めるには二分探索を行う. 通常の文字列二分探索の最悪計算量は, 文字列比較に最悪で $O(m_l)$ の時間が必要であることを考えると $O(m_l \log m_l)$ であるが, DCS を使うことでこれを $O(\log m_l \log^2 n)$ に抑えることができる. しかし, 提案手法では接

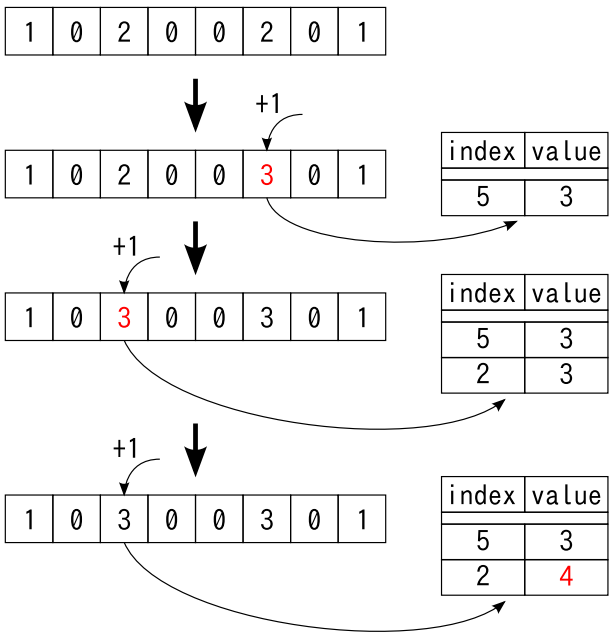

図 7.6. gap の構築例

尾辞配列がサンプリングされているため, 各要素を復元する必要がある. これは, T_l^{BWT} を wavelet 行列に変換することで $O(\log\sigma \log m_l)$ の時間で行えるようになる. それゆえ, ここでの二分探索の最悪計算量は $O(\log\sigma \log^2 m_l \log^2 n)$ となる. 一旦 T_r の最後から始まる suffix の挿入位置が求まれば, それより手前から始まる suffix の挿入位置は式 (7.3) と式 (7.4) を用いることで帰納的に求めることができる. Suffix を一文字手前に辿るためには, 式 (7.3) の下の場合 (b の項が不要な場合) では $O(\log\sigma)$ の時間が必要となる. 追加の b の項は, 一文字前に延長された文字 c と T_l の最後の文字が同じ場合に必要となる. この場合でも, 文字列比較の回数は DCS によって $O(\log^2 n)$ に抑えられる.

以上をまとめると, gap 構築の最悪計算量は $O(\log\sigma \log^2 m_l \log^2 n + m_r \log\sigma + m_r \log^2 n)$ となる. 最初の項は二分探索, 二番目, 三番目の項は suffix を一文字前に延長したものを帰納的に計算していくのに必要な時間である. 多くの場合 $\sigma \leq n$ であるため, 結局のところ最悪計算量は $O(m_r \log^2 n)$ となる. gap により, T_l^{BWT} と T_r^{BWT} は $O(m_l \log\sigma + m_r)$ の時間でマージすることができる. ここで, T_l^{BWT} は wavelet 行列に変換されており, 各要素を access 関数によって順に取り出し, それらを gap の要素に従って配置していくため, $\log\sigma$ の項がかかっている. 最終的に, T_l^{BWT}, T_r^{BWT} のマージは gap 構築の部分が支配的となり, 最悪計算量は $O(m_r \log^2 n)$ となる. サンプリングされた接尾辞配列と mark ビット列のマージも同様である.

ここまでの計算はそれぞれ長さ m_l, m_r の部分文字列の BWT の出力をマージすることを考えていたが, m_l, m_r の値が最大になるのは, 再帰的なマージの最後のステップ, すなわちテキストを 2 つに分けたものの BWT の出力をマージするときである. このとき, $m_l \simeq n/2$, $m_r \simeq n/2$ であるため, どちらも $O(n)$ であると考えれば, gap が使用するメモリは $O(n \log\log n)$ ビットとなり, マージの全体的な計算量は $O(n \log^2 n)$ となる.

マージの例を図 7.7 に示す. 文字 '-' は文字が不定であることを表している. ここでは接尾辞配

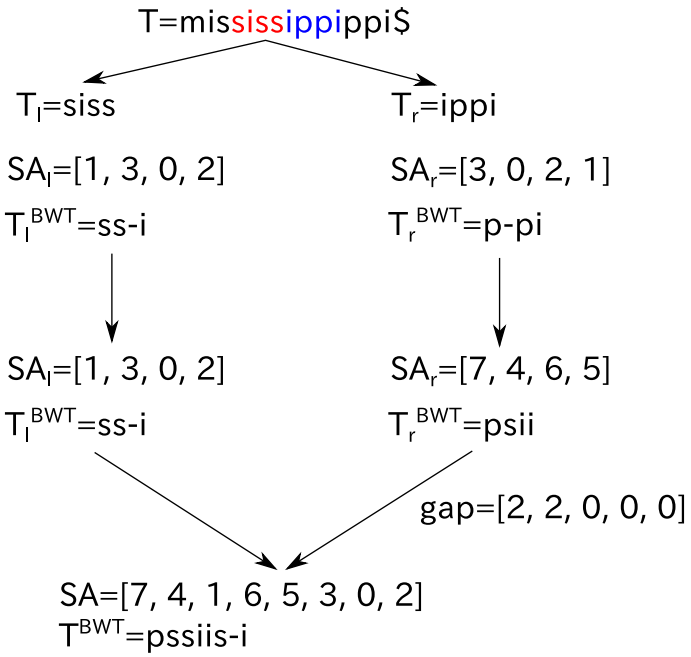

図 7.7. BWT のマージ例

列全体を示しているが, 実際にはこれらをサンプリングしたもの及び mark ビット列を保持していることに注意が必要である.

7.6　ハフマン符号を利用した wavelet 行列

Wavelet 行列はそのままの状態では圧縮が行われておらず, wavelet 行列を保持するために必要なメモリは wavelet 行列に変換する前の文字列を保持するために必要なメモリとほぼ同じである. Wavelet tree においては様々な圧縮手法が考案されているが, wavelet matrix は比較的新しい概念であるため, 圧縮手法についてはまだ十分に研究が行われていない. しかし, wavelet 行列は wavelet tree と似たデータ構造であるため, 実際には wavelet 行列にも同様の圧縮手法を適用することが可能であると考えられる. そこで提案手法ではハフマン符号を利用した圧縮手法を wavelet 行列にも適用する.

まず, wavelet 行列に変換したい文字列に出現する文字の数を種類毎にカウントする. 次に各文字に対して, その出現頻度を元にハフマン符号を割り当てる. 通常の wavelet 行列の構築ではアルファベットの各文字に対して順に整数を割り当て, それらのビットを先頭から順に見ていくことで文字の振り分けを行ったが, 今回は各文字に割り当てられたハフマン符号を先頭ビットから見ていき, 0 が割り当てられている文字は左に順番を保ったまま移動し, 1 が割り当てられているものはその後ろに, やはり順番を保ったまま移動させる. このようなことを通常の wavelet 行列の構築の時と同じように行っていく. この方法の場合, 文字の種類毎に符号の長さが違うのに加え, リーフでの各文字の並び方に規則性がない. そのため, どのアルファベットがどの深さで終了し, またそのときの範囲はどこからどこまでなのかを, 別の領域に記録しておく必要がある.

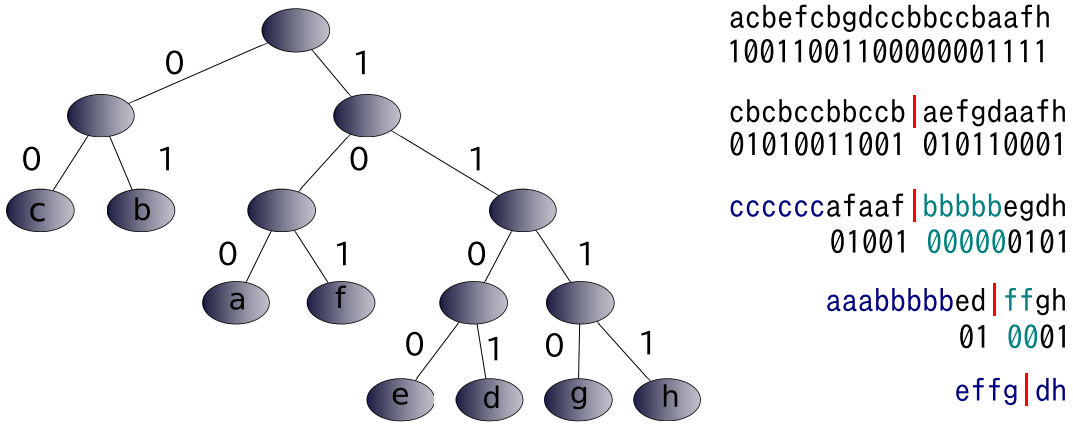

図 7.8. ハフマン符号を利用した wavelet 行列の例

この方法を用いることで, テキストに通常のハフマン符号化を施して圧縮を行った場合とほぼ同等の圧縮率を保ちながら, access や rank といった演算の計算も高速に行えるようになる. これは, よく出現する文字ほど浅い深さで access や rank の計算を終えることができるためである.

ハフマン符号を利用した wavelet 行列の例を図 7.8 に示す. この例では $T =$“acbefcbgdccbbccbaafh” に対して wavelet 行列を構築している. まず, 各文字の出現頻度に応じてハフマン木を構築し, アルファベットの各文字にハフマン符号を割り当てる. 各文字に割り当てられるハフマン符号は, 図左側の木をルートから辿ったときに途中で通過するエッジに割り当てられた数値を結合したものである. 例えば ‘a’ であれば 100_2 となる. 次に, 各文字のハフマン符号の先頭ビットと同じビットをテキストの各文字に割り当てる. そして, 通常の wavelet 行列と同様に, 0 が割り当てられた文字を前半に, 1 が割り当てられた文字を後半に移動させる. このような操作をもう一度繰り返すと, ‘b’, ‘c’ には 2 ビットの符号が割り当てられているので, これ以上操作を行うことができなくなる. 今, ‘c’ は並び替えられた文字列の一番左側にいるため, これ以降の操作ではこれを無視して進めることができるが, ‘b’ が並ぶ範囲の左側にはまだ処理が終了していない文字列が並んでいる. そのため単純に ‘b’ を無視することはできない. そこで, これ以降の操作では ‘b’ に 0 が割り当てられていると仮定し, ‘b’ が並ぶ範囲の左側の文字列が全て処理済みになった段階で, ‘b’ もそれ以降の操作で無視するようにする. 例では 3 度目の移動のあと, 左側には 3 ビットの符号が割り当てらた ‘a’ のみが存在するので, 以降の操作では ‘a’ と ‘b’ を消去して考える. このようにすることで, ハフマン符号に基づいた wavelet 行列を構築することができる.

上記のような wavelet 行列の改良により, wavelet 行列のサイズ, 及び access や rank の計算時間を削減することが可能である. しかし, すでにビットが終了している文字についても, 左側にまだ処理済みでない文字の並びが存在していた場合, その後もビットを割り当てなければならないため, ハフマン符号による圧縮を完全に行えていないという問題があり, まだ改良の余地があると考えられる.

7.7　逐次アルゴリズムの計算量

再帰呼び出しの末端においては $O(n \log^2 n)$ の時間が必要であり, マージにも $O(n \log^2 n)$ の時間がかかることを述べた. 提案手法の時間計算量を $\mathrm{PM}(n)$ とすると, これは以下のような漸化式で表すことができる.

$$\mathrm{PM}(n) = \begin{cases} O(n \log^2 n) & (\text{if } n < \text{threshold}) \\ 2\mathrm{PM}(n/2) + O(n \log^2 n) & (\text{otherwise}) \end{cases} \tag{7.5}$$

再帰の深さが $O(\log n)$ であることを考えると, これより全体の最悪計算量は $O(n \log^3 n)$ であることが分かる. また, 空間計算量は gap が支配的となり, $O(n \log \log n)$ である.

第8章　BWT計算の並列化

8.1　概要

提案手法は分割統治法に基づいており, タスク並列処理と相性がよい. すなわち, 提案手法では手続きを再帰的に呼び出しており, その再帰呼び出しを 1 つのタスクとみなすことで, 数多くのタスクを生み出すことができる. それらのタスクを各スレッド間で動的に分配することで, 並列化を実現できる. これによって, クリティカルパスは $O(n \log^2 n)$ にまで減少する. 既存手法には逐次性能で $O(n)$ を達成しているものが存在するため, これだけでは実用的なアルゴリズムとは言えない. よって, マージ処理も並列化する必要がある. これらについて以下で順に述べる.

8.2　タスク並列モデル

タスク並列モデルとは, プログラム上の任意の場所でタスクと呼ばれる計算単位を生成し, runtime システムによってそれらが動的に各ワーカーに分配され, 並列実行されるというプログラミングモデルである. タスク並列モデルにおいてプログラマがすべきことはタスクの生成とそれらの同期を指示する文を追加することのみであり, pthread などの低レベルな並列ライブラリに比べて, より容易に並列性の記述が可能であると言える.

タスク並列モデルは, 大きな問題を再帰的に分割し, それより小さい副問題を生成していくという分割統治法と相性が良い. 分割統治法は通常再帰によって記述され, そのような再帰呼び出しをタスクとみなすことでタスク並列モデルによって自然に実装が可能である.

分割統治法で記述されたプログラムをタスク並列モデルで並列化する例を図 8.1 に示す. この例ではクイックソートの並列化を行っている. 例では配列をピボットより小さい要素とピボット以上の要素に分けた後, それぞれの範囲に対して再帰的にクイックソートの手続きを呼び出しているが, そこで spawn という文によってタスクを生成している. その後, 生成された 2 つのタスクを同期するために, sync という文を用いている.

タスク並列モデルを用いると, for 文で書かれるようなループ処理に対しても, 分割統治法で記述し直すことによって並列化できる. これにより, for 文の各イテレーションで計算の量が異なるような場合でも, 動的に負荷分散を行うことで効率良く並列化することが可能である.

タスク並列モデルにより, 提案手法はまず図 8.2 のように並列化することができる. これにより, クリティカルパスは以下のように表される.

$$\mathrm{PM}(n) = \begin{cases} O(n \log^2 n) & (\text{if } n < \text{threshold}) \\ \mathrm{PM}(n/2) + O(n \log^2 n) & (\text{otherwise}) \end{cases} \tag{8.1}$$

```
1  void parallel_qsort(int begin, int end, vector<int> &T){
2    if(end − begin < THREASH){
3      // タスクが閾値以下のサイズになったら逐次実行
4      small_sort(begin, end, T);
5    }else{
6      int pivot = T[begin];
7      int i = begin + 1, j = end − 1;
8      // ピボットより小さい要素とピボット以上の要素に分類
9      while(true){
10       while(T[i] < pivot && i < end) i++;
11       while(T[j] >= pivot && j >= begin) j−−;
12       if(j < i) break;
13       // T の i 番目の要素と j 番目の要素を交換
14       swap(T, i, j);
15     }
16     swap(T, 0, j);
17     // 再帰呼び出し時にタスクを生成
18     spawn parallel_qsort(begin, j, T);
19     spawn parallel_qsort(j, end, T);
20     // タスクの終了同期
21     sync;
22   }
23 }
```

図 8.1. タスク並列モデルによるクイックソートの並列化

これを解くと, 結局クリティカルパスは $O(n \log^2 n)$ となる. $o(n)$ のクリティカルパスを達成するためには, さらにマージの並列化を行う必要がある他, difference cover sample の構築等も並列化する必要がある.

8.3　Difference Cover Sample 構築の並列化

Difference cover sample (DCS)[21] は, 基本的には DC3 とほぼ同様に構築することができる. DC3 には pDC3[22] という並列化手法がすでに存在しているので, DCS の構築も pDC3 のアイディアを借りて行うことができる. ここではその方法について説明する.

先頭 v 文字のソートについては, 並列ソートを行う. pDC3 では, サンプルソートなどの通常の並列ソートを行えばよかったが, DCS の場合は先頭 v 文字でのソートであるため, 文字列に適合した並列ソートを行う必要がある.

並列文字列ソートの最新の研究は Bingmann らによる並列文字列サンプルソート [5] だと思われる. Bingmann らの方法は実用的に高速であることが示されている. Bingmann らの手法ではソート対象の文字列をバケツに分類する際に, CPU のビット幅に収まる数の文字 (例えば 64 ビットマシンの場合は 8 文字) を同時に比較しながら二分探索を行うのだが, 同一サンプルが数多く存在してしまうとバケツの分類がうまくいかない. それを避けるためにユニークなサンプルを選ぶための方法も説明されているが, その場合, サンプルが実際の分布と異なった分布になりやすく, 分類

```
1  void ComputeBWT(int begin, int end, Node& parent){
2    if(inputSize<=LEAF_SIZE){
3      ComputeLeaf(begin, end, globalBegin, globalEnd, parent);
4      return;
5    }
6    Node left, right;
7    int mid=inputSize/2;
8    // 再帰呼び出し時にタスクを生成
9    spawn ComputeBWT(begin, begin+mid, left);
10   spawn ComputeBWT(begin+mid, end, right);
11   // タスクの終了同期
12   sync;
13   // 左側と右側の子ノードをマージ
14   MergeNode(left, right, parent, left.begin(), left.end(), right.begin(), right.end());
15 }
```

図 8.2. 並列アルゴリズムの概要

後の各バケツのサイズに偏りが生じやすくなると考えられる．もし可能であれば, 分類の際には先頭数文字だけでなく, 文字列全体を比較できるとよい．

そのため, 本研究ではバケツへの分類方法を変更した新たな並列文字列ソート手法である BinLCP ソートを考案した．この手法では, longest common prefix (LCP)[27] と呼ばれるデータ構造を改変して使用する．LCP とは, ソートされた suffix の列において, 隣接する 2 つの suffix の prefix が何文字一致するかを記録したものである．より具体的には, $T[SA[i-1], n]$ と $T[SA[i], n]$ の共通する prefix の長さが LCP_i ということである．

まず, ソートされたサンプルに対して LCP を構築することを考える．すると, 全ての suffix をバケツに分ける段階で, ソートされたサンプルの先頭要素とだけ文字列比較をすれば, それ以降のサンプルとは一文字の比較だけで大小が判別できる．例えばソートされたサンプルの i 番目の要素とバケツに分類したい suffix を比較し, (0 始まりで数えて)a 文字目で初めて異なる文字が現れ, 後者の方が大きいと分かったとする．つまり, 両者の共通 prefix 長は a である．

次に, ソートされたサンプルの $i+1$ 番目の要素とこの suffix を比較する際には, $\min(a, \mathrm{LCP}[i+1])$ 文字目のみを比較すればよいことになる．もし $a < \mathrm{LCP}[i+1]$ の場合, a 文字目の段階では i 番目のサンプルと $i+1$ 番目のサンプルは一致している．よって, 先ほどと同じく a 文字目で最初に異なる文字に遭遇するはずである．また $a > \mathrm{LCP}[i+1]$ の場合, $\mathrm{LCP}[i+1]$ 文字目において i 番目のサンプルと $i+1$ 番目のサンプルは異なる文字であるため, 先ほどとは違い, a 文字目に到達することなく $\mathrm{LCP}[i+1]$ 文字目で初めて異なる文字が現れることになる．以上により, LCP が構築できれば suffix のバケツへの分類は比較的高速に行うことができる．しかし, この方法だとソートされたサンプル suffix を先頭から順に見ていく必要があり, 平均してサンプル数に比例した回数の比較をしなければならない．通常サンプルはコア数 p に比例した数だけ選ぶので, 例え分類操作を p 並列で行ったとしても, $O(p)$ の比較を p 並列で n 個の要素に対して行うため, クリティカルパスは $O(n)$ となってしまう．

そこで, LCP を二分木状に構築することを考える．つまり, まずサンプルの中央の要素と 1/4, 及び 3/4 の位置の suffix の共通 prefix 長を計算する．次に, サンプルを左右に分け, それぞれに対

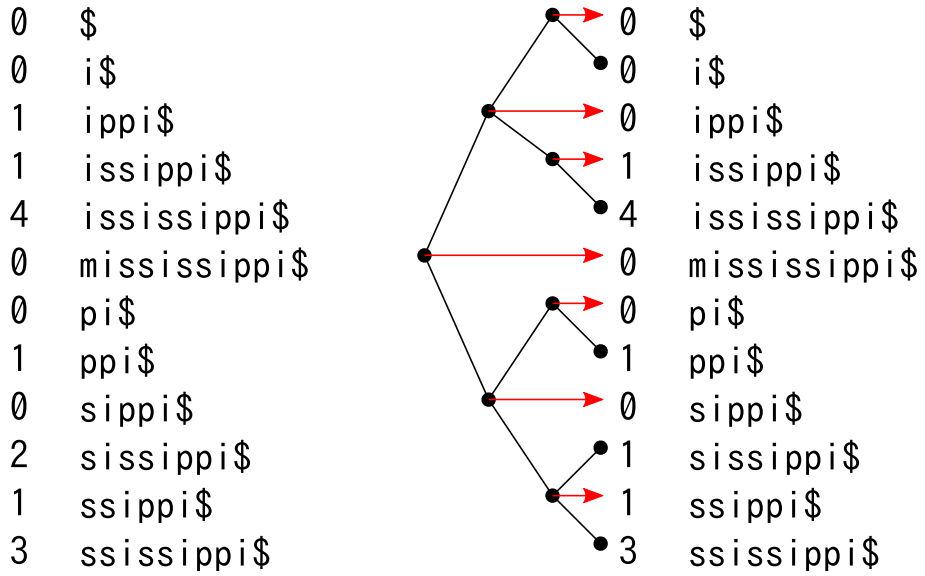

図 8.3. (左)LCP, (右)BinLCP

して再帰的に中央の要素と 1/4, 及び 3/4 の位置の suffix の共通 prefix 長を計算していく. このようなデータ構造を用いることで, suffix を分類する際に高速な二分探索を用いながら, 二回目以降の文字列比較は一文字のみで済むという LCP の利点を残すことができる. このように二分木状に構築した LCP を, 以下では BinLCP と呼ぶ. BinLCP による suffix 分類の計算量について考える. 最初に一回の文字列比較があり, その後は一文字のみの比較で二分探索を行えばよいので, サンプル数が p に比例するようにした場合, 計算量は $O((n/p)(\log^2 n + \log p))$ となる. クリティカルパスは $p = n$ となるときであり, $O(\log^2 n)$ となる.

BinLCP の例を図 8.3 に示す. 図左側には通常の LCP を示している. 例えば, 3 番目と 4 番目の suffix はそれぞれ “issip···”, “issis···” であるため, LCP[4] = 4 であり, 5 番目の suffix は “missi···” であるため, LCP[5] = 0 である. それに対し, 図右側の BinLCP では, まず中央にある 5 番目の suffix と, 1/4, 及び 3/4 の位置にある 2 番目, 8 番目の suffix を比較する. それぞれ “miss···”, “ippi···”, 及び “sipp···” であるため, BinLCP[2] = 0, BinLCP[8] = 0 である. 次に, 5 番目の位置を境に前半部分である $[0, 4]$, 及び後半部分である $[6, 11]$ に対して再帰的に処理を繰り返す. 前半に注目すると, 中央にある 2 番目の suffix と, 1/4, 及び 3/4 の位置にある 0 番目, 3 番目の suffix を比較することになる. これらの共通 prefix 長を求めることで, BinLCP[0] = 0, BinLCP[3] = 1 となる. このような処理を全ての要素を網羅するまで再帰的に行うことで, BinLCP を構築することができる.

DCS の構築に話を戻す. 残るは再帰計算の部分だけであるが, これは DC3 をそのまま利用しているだけなので, これを pDC3 に置き換えれば並列化が可能である.

8.4　簡潔データ構造の並列構築

提案手法では, マージの際に wavelet 行列を構築する. そのため, wavelet 行列も並列に構築できなければ全体のクリティカルパスを下げることはできない. Wavelet 行列の各深さでは, さらに完備辞書を作成するため, これらも並列に構築する必要がある.

まず, 完備辞書の並列構築について説明する. Access については補助的なデータ構造は不要なため, ここでは rank を定数時間で行うための補助的なデータ構造の構築について述べる. 最初に, ビット列の各ブロック内にある 1 の数をブロック毎に数える. 各ブロックに対する処理は独立であるため, これは容易に並列化できる. 次に, スーパーブロック内において先ほど数えた各ブロック毎の 1 の数の prefix sum を計算する. これにより, 各ブロックの境界について, 直近のスーパーブロックの境界から数えていくつ 1 が現れるかが分かり, さらに各スーパーブロックの中にいくつ 1 が存在するかが分かる. 最後に, 各スーパーブロックに現れる 1 の数について prefix sum を計算する. これにより, ビット列の先頭から数えて各スーパーブロックの境界までに現れる 1 の数が分かる. このようにしてブロックとスーパーブロックの構築は並列に行うことができる. ブロック内に現れるビットパターンのテーブルについては, 各ビット毎に独立であるため, やはり容易に並列化可能である.

計算量について考える. ブロック内の 1 の数を数えるのは, ブロックサイズが $O(\log n)$ なのでクリティカルパス $O(\log n)$ で計算できる. スーパーブロック内において各ブロックの 1 の数の和に対して prefix sum を計算するのは, n 個の要素に対する prefix sum のクリティカルパスが $O(\log n)$ であること, そしてスーパーブロック内に存在するブロック数が $O(\log^2 n/\log n) = O(\log n)$ 個であることから, クリティカルパス $O(\log \log n)$ となる. さらにスーパーブロック毎の 1 の数の和に対して prefix sum を計算するのは, スーパーブロックの数が $O(n/\log^2 n)$ であるため, クリティカルパス $O(\log(n/\log^2 n)) = O(\log n)$ で計算できる. ビットパターンのテーブルは, 各要素のビットの長さは $O(\log n)$ であり, また $O(\sqrt{n})$ 個の要素について独立に計算できるため, クリティカルパスは $O(\log n)$ である. 以上をまとめると, rank のための補助的なデータ構造を構築するためのクリティカルパスは $O(\log n)$ となる.

Rank のための補助的データ構造の並列構築例を図 8.4 に示す. この例では, ブロックサイズは 4, スーパーブロックサイズは 16 としている. まずブロックごとに 1 の数を数える. そして, 得られた結果に対してスーパーブロック内で prefix sum を計算する. すると, スーパーブロック内での最後の値はスーパーブロック内の 1 の数を表すことになる. これが図中段の赤い四角で囲まれた 10, 8 という数値である. 最後に, このスーパーブロック内の 1 の数を表す数値に対して prefix sum を計算する. この例では, 最初のスーパーブロックには 10 個の 1 があるので 10, 二番目のスーパーブロックには 8 個の 1 があるので, $10 + 8 = 18$ という値が, 二番目のスーパーブロックの最後までに存在する 1 の数ということになる. このようにして, ブロックとスーパーブロックの 1 の数を数えることができる.

次に, wavelet 行列の並列構築について述べる. wavelet 行列において並列化すべき処理は, 各文字へのビットの割り当て, 完備辞書の構築, そしてビットに応じた各文字の移動である. 完備辞書の並列構築については上で述べた通りである. 各文字へのビット列の割り当ては文字ごとに独立であるため, 容易に並列化可能である. 文字の移動については, もしある文字に 0 が割り当てられていた場合, そこまでに出現する 0 の数を rank によって数える. これにより, その文字は結果を格納する配列のどこに書き込めばいいか分かる. また, もしある文字に 1 が割り当てられていた場

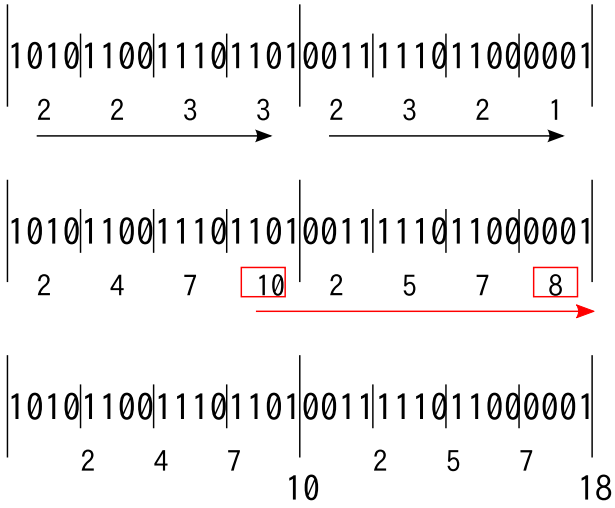

図 8.4. rank のための補助的データ構造の並列構築

合, ビット列の全ての 0 の数を rank で数え, さらにそこまでに出現する 1 の数を rank によって数える. これらを足すことで, その文字は結果を格納する配列のどこに書き込めばいいか分かる.

wavelet 行列を構築するためのクリティカルパスを考える. 各文字へのビット割り当ては $O(1)$, 完備辞書の構築は $O(\log n)$ である. 文字の移動については, やはり全ての要素について独立に行うことができるので $O(1)$ となる. これらの操作が $\log \sigma$ の深さまで行われるので, 全体のクリティカルパスは $O(\log \sigma \log n)$ である.

8.5　マージの並列化

マージの並列化では, 主に gap 配列の構築と, それを利用した実際のマージ処理の 2 つの処理について考える必要がある.

8.5.1　gap の並列構築

まず, gap の並列構築法について説明する. 概要を図 8.5 に示す. 7.5 節で, gap の構築では T_r の末尾の suffix が T_l の接尾辞配列のどこに挿入されるかが分かれば, あとは帰納的にその挿入位置を決定できることを説明した. gap の構築を並列化するためには, T_r を大きさ B のブロックに分け, それらの末尾の suffix の挿入位置を二分探索によって決定する. そして, 各ブロックのその他の suffix に対して, そのブロックの末尾の suffix から順に帰納的に挿入位置を求めていく.

このような並列化による計算量を考える. $\lceil n/B \rceil$ 個のブロックに対してそれぞれ独立に計算が可能なので, 計算量は以下のようになる.

$$
\begin{aligned}
& O\left(\left\lceil \frac{\lceil n/B \rceil}{p} \right\rceil \log \sigma \log^4 n + \left\lceil \frac{\lceil n/B \rceil}{p} \right\rceil (B-1) \log \sigma + \left\lceil \frac{\lceil n/B \rceil}{p} \right\rceil (B-1) \log^2 n \right) \\
= \quad & O\left(\left\lceil \frac{\lceil n/B \rceil}{p} \right\rceil \log \sigma \log^4 n + \left\lceil \frac{\lceil n/B \rceil}{p} \right\rceil (B-1)(\log \sigma + \log^2 n) \right) \qquad (8.2)
\end{aligned}
$$

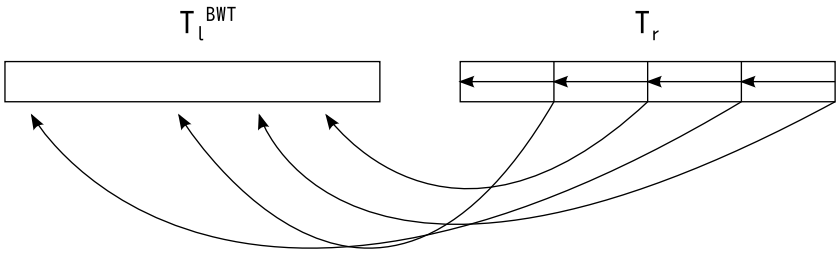

図 8.5. gap の並列構築

```
1  ParallelConstructGap(int begin, int end, DynamicSizeVec &gap, Node &left, Node &right,
2      int lBegin, int lEnd, int rBegin, int rEnd){
3    if(begin − end <= THREASH){
4      int c, idx;
5      // 各ブロックの最後の suffix の挿入位置を計算
6      idx=StringBinSearch(left, lBegin, end−1);
7      gap[idx+1]++;
8      // ブロック内の要素の挿入位置を帰納的に計算
9      for(int i = end − 2; i >= begin; i−−){
10       c=T[rBegin+i];
11       idx=left.C(c)+left.rank(c, idx)−1;
12       if(c==T[lEnd−1] && StringLessThan(rBegin, rBegin+i+1)) idx++;
13       gap[idx+1]++;
14     }
15   }else{
16     int mid =(begin + end)/2;
17     spawn ParallelConstructGap(begin, mid, gap, left, right, lBegin, lEnd, rBegin, rEnd);
18     spawn ParallelConstructGap(mid, end, gap, left, right, lBegin, lEnd, rBegin, rEnd);
19     sync;
20   }
21 }
```

図 8.6. gap 構築の並列化

B が n や p によらない定数であるとすると, 結局クリティカルパスは $O(\log \sigma \log^4 n)$ となる.

以上のような並列化を分割統治法によって記述し, さらにタスク並列モデルによって並列化した擬似コードを図 8.6 に示す.

8.5.2　gap への並列アクセス

Gap の構築において, 挿入位置は 0 から m_l までの全ての値になり得るため, 各スレッド間で gap への書き込みの競合が発生する. 書き込みの競合を防ぐための最も基本的な手法として mutex を利用した排他制御が考えられる. しかし, 実際の実装を考えると, gap の要素数だけ mutex のためのメモリを確保するというのは現実的ではない. 例えば pthread の pthread_mutex_t 型の変数は 40 バイトものサイズがあり, これを要素数分だけ確保すると, 元のテキストの 40 倍ものメモリを

消費することになる．

この問題の解決策として, mutex 変数を M 個用意し, gap の i 番目の要素に変更を加えるときは $i \bmod M$ 番目の mutex 変数に対してロックを取得するという方法がある．これにより, mutex を gap の要素数分だけ用意する必要がなくなる．しかし, M がコア数 p に依存してしまうという点と, 本来排他制御する必要がない要素に対しても排他制御を行ってしまうという問題がある．例えば 0, M, $2M$, $\cdots$ 番目の要素に対して書き込みを行うとき, それぞれに対しては独立に書き込みが行えるにも関わらず, 排他制御を行ってしまう．

そこで, compare and swap を用いたアトミックな更新を行う．Compare and swap とは, あるポインタが指す変数とある値を比較し, もし両者が等しければそのポインタが指す位置に別の指定した値を書きこむという処理をアトミックに行う CPU 命令である．Compare and swap を使うと, まず他のスレッドの挙動を考慮せずに変数の変更処理を行い, それが正しく行われていれば確定し, 途中で他のスレッドと競合して不正な値になった場合は変更処理を破棄するという操作を行うことができる．よって, mutex のように特別な変数を使用することなく, gap の各要素に対してアトミックに変更を行うことができる．実際には compare and swap が利用できるのは, gap の要素が最初に割り当てられた p ビットに収まっているときだけであり, この範囲に収まらず, key-value ペアをハッシュテーブルに登録した後は, mutex による排他制御を行わなければならない．

8.5.3　マージの並列化

次に, gap を用いたマージ処理について説明する．gap には T_r^{BWT} の要素がそれぞれ T_l^{BWT} のどこにいくつ挿入されるかが記されている．そこで, gap を大きさ B' のブロックに分け, 各ブロックの境界までの prefix sum を計算する．そうすることで, 現在注目しているブロックより前の段階で挿入処理が行われる T_r^{BWT} の要素数が分かる．さらに, そのブロックの手前に存在する gap の要素の個数を数えれば, 手前のブロックにおいて扱われる T_l^{BWT} の要素数が分かる．それゆえ, それよりも 1 つ後の要素から順にそのブロックでのマージ処理を行えばよいことが分かる．

しかし実際には, gap の prefix sum を計算することはできない．なぜなら, prefix sum を行うことで, gap 配列の要素に大きい数値が入るようになり, 7.5 節で説明したようなエンコーディングが行えなくなるためである．すると, gap の prefix sum の結果を格納するためには $O(n \log n)$ ビットのメモリが必要となり, 接尾辞配列をテキスト全体に対して構築する場合と変わらなくなってしまう．

そこで, prefix sum の計算結果を直接保持するのではなく, gap に補助的なデータ構造を用意することで, prefix sum の値をある程度高速に取得できるようにすることを考える．方法は完備辞書の rank のための補助的なデータ構造とほぼ同様で, テキストをブロックとスーパーブロックに分ける．そして, 各スーパーブロックの境界までの gap の要素の総和を計算し, さらに直近のスーパーブロックの境界から各ブロックの境界までの gap の要素の総和を計算する．ただし, 完備辞書の rank の場合と異なり, ブロック境界からは直接要素を足し合わせる．このようにすることで, 配列アクセス 2 回とブロック内での要素の足し合わせによって, prefix sum の値を得ることができる．ブロックサイズ, スーパーブロックサイズをそれぞれ $\log n$, $\log^2 n$ としておけば, prefix sum の各要素を平均して $O(\log n)$ の時間で求めることができる．このような方法は four Russians technique[3] と呼ばれる．

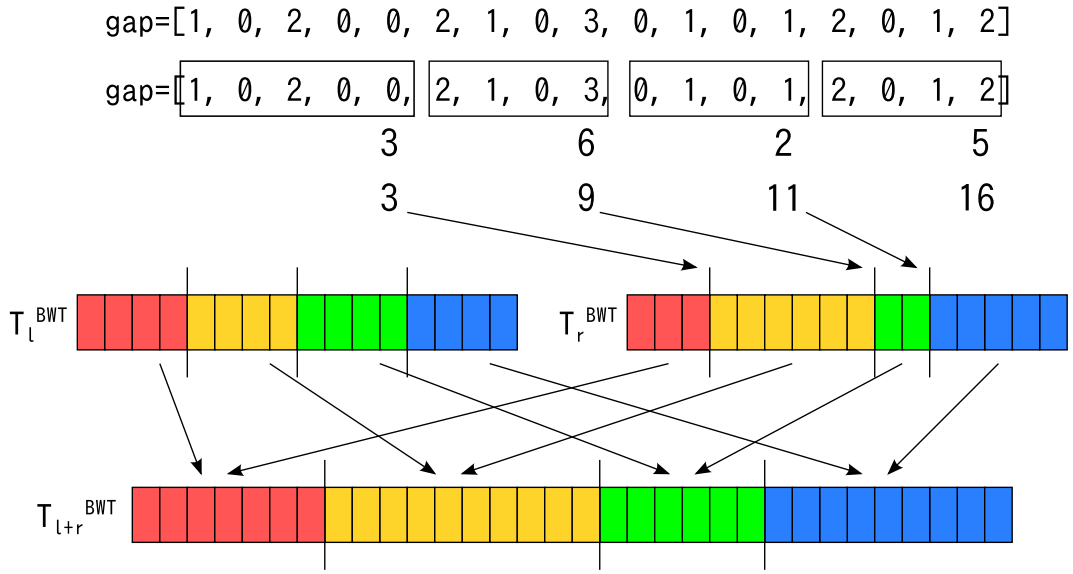

図 8.7. 並列マージの例

1つ注意しなければならないのは, マージの際に T_r^{BWT} の要素が全て T_l^{BWT} のある一箇所に入ってしまう可能性があるということである. この場合, ここまで説明した並列化の方法だと, T_r^{BWT} の全ての要素が挿入される T_l^{BWT} のブロックのマージにおいて, T_r^{BWT} の要素数分の計算, つまり $O(n)$ の計算が発生することになる. よって, T_r^{BWT} の要素を gap の要素の値に応じて結果格納用の配列に書きこむという処理も並列化しなければならない. これは単なる配列の要素のコピーであり, 容易に並列化が可能である.

マージの例を図 8.7 に示す. この例ではまず gap を均等に分割している. (gap の要素数が 17 であるため, 最初のブロックだけ大きさが 1 つ大きい.) そして, 各ブロックの境界までの要素の排他的 prefix sum を計算する. 例では各ブロックの境界までの要素の合計はそれぞれ 3, 9, 11, 16 となっている. この値に対して排他的 prefix sum を計算することで, T_r^{BWT} は 0, 3, 9, 11 番目の要素からマージを開始すればよいことが分かる. T_l^{BWT} は単純に四等分すればよい. こうすることで, 各スレッドが T_l^{BWT}, 及び T_r^{BWT} を読み始める位置が分かり, また結果を格納する配列に書きこみ始める位置も分かる.

以上をまとめると, 各ブロックの処理をする度に $O(\log n)$ の時間で prefix sum を計算しなければならない代わりに, マージ処理を並列化することができるようになる. 今, ブロックサイズは B' であるとしているので, 計算量は以下のようになる.

$$O\left(\left\lceil \frac{\lceil n/B' \rceil}{p} \right\rceil \log n + B' \left\lceil \frac{\lceil n/B' \rceil}{p} \right\rceil\right) \tag{8.3}$$

B' が n や p によらない定数であるとすると, 結局クリティカルパスは $O(\log n)$ となる.

以上のような並列化を分割統治法によって記述し, さらにタスク並列モデルによって並列化した擬似コードを図 8.8 に示す.

```
1  void ParallelMergeBWT(int begin, int end, DynamicSizeVec &gap,
2       Node &left, Node &right, Node &parent){
3    if(begin − end <= THREASH){
4      // 結果を格納する配列に書き始める位置を計算
5      int pBWTIdx=gap.GetPrefixSum(begin)+begin
6      // 右側のBWTを読み始める位置を計算
7      int rIdx=gap.GetPrefixSum(begin);
8      for(int j=0; j<gap[0]; j++){
9       if(right.isSA0(rIdx)) parent.bwt[pBWTIdx++]=T[lEnd];
10      else parent.bwt[pBWTIdx++]=right.bwt[rIdx++];
11     }
12     for(int i=1; i<gap.size(); i++){
13       parent.bwt[pBWTIdx++]=left.bwt[i];
14       for(int j=0; j<gap[i]; j++){
15         if(right.isSA0(rIdx)) parent.bwt[pBWTIdx++]=T[lEnd];
16         else parent.bwt[pBWTIdx++]=right.bwt[rIdx++];
17       }
18     }
19   }else{
20     int mid =(begin + end)/2;
21     spawn ParallelMergeBWT(begin, mid, gap, left, right, parent);
22     spawn ParallelMergeBWT(mid, end, gap, left, right, parent);
23     sync;
24   }
25 }
```

図 8.8. gap を用いたマージの並列化

8.6　並列アルゴリズムのクリティカルパス

以上をまとめると, 再帰呼び出しのある深さにおけるマージのための各計算のクリティカルパスは以下のようになる.

- wavelet 行列構築: $O(\log \sigma \log n)$
- gap 構築: $O(\log \sigma \log^4 n)$
- マージ: $O(\log n)$

このように, マージにおいては gap の並列構築が支配的となるため, クリティカルパスは $O(\log \sigma \log^4 n)$ となる. 再帰呼び出しの深さが $O(\log n)$ であることを考えると, アルゴリズム全体の最悪の場合でのクリティカルパスは $O(\log \sigma \log^5 n)$ となる.

第9章　評価

9.1　評価環境

提案手法はC++で実装した．提案手法の並列化には, 当研究室が開発しているMassiveThreads[30]というタスク並列ライブラリを使用した．MassiveThreadsは, Cilk[6] やOpenMP, Intel TBBなどに見られるものと類似したタスク並列のためのprimitiveを持ち, さらにワークスチーリングを行うためのスケジューラが実装されている [7]．提案手法を実行する際, 以下の全ての場合において, 再帰的な分割から部分文字列のBWT計算に移る閾値は100,000とし, DCSは400を法として構築した．

提案手法の評価として実行時間とメモリ使用量を調べた．実行時間はgettimeofdayシステムコールによって, メモリ使用量はgetrusageシステムコールによって測定した．比較対象として, 岡野原らの手法であるBWT-IS, 及びテキスト全体の接尾辞配列をSA-ISによって構築してからBWTを計算する方法 (以下では単にSA-ISと呼ぶ) を使用した．BWT-ISの実装はRead & Researchmapのウェブページ [1] で公開されているものを使用した．また, SA-ISは考案者であるNongらが公開しているテクニカルレポート [35] に書かれているものを利用した．入力ファイルとして表9.1に示すようなデータを利用した．このうち, genomeというのは人間の人間のゲノム配列を記したデータであるHG19と呼ばれるものであり, UCSCのウェブサイト [2] で公開されているものを使用した．

評価は全て単一ノード上で行った．以降で示すデータは全て三回計測を行ったものの平均値である．以下では提案手法をPM-BWT (Parallel and Memory-efficient BWT) と呼ぶことにする．

9.2　データサイズに対する性能変化

データサイズを変更したときに, 実行時間とメモリ使用量がどのように変化するかを調べた．この評価はIntel (R) Xeon (R) E7540 2.00GHz 24コアのCPU, メモリ256GBのマシンで行った．この評価では, PM-BWTは24コア全てを使用したときの性能を計測している．BWT-ISとSA-ISは並列化が不可能なアルゴリズムであるため, これらについては1コアで実行したときの性能を計測した．

それぞれの入力データに対する実行時間を図9.1, 9.2, 9.3, 9.4に示す．これを見ると, genome, wikipedia, 及びrandomについてはPM-BWTが最も高速に計算を終えていることが分かる．特に, randomに対してはBWT-IS, 及びSA-ISの性能が著しく悪いため (縦軸の数値に注意), 相対的に高速になっている．しかし, sameにおいてはPM-BWTの性能は極めて悪く, 逆にBWT-ISとSA-ISが非常に高速であることが分かる．BWT-ISとSA-ISは, どちらも入力データの冗長性を利用することで高速に動作するアルゴリズムであるため, 入力テキストのエントロピーが小さ

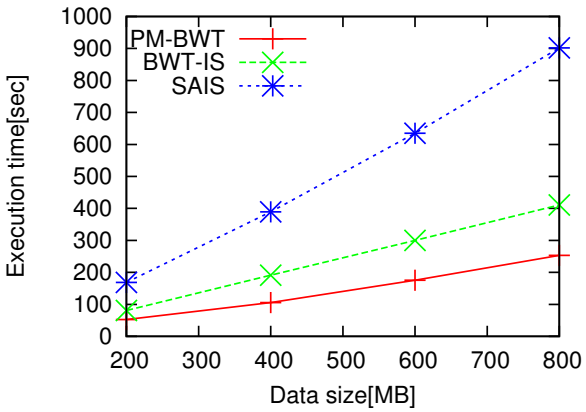

図 9.1. genome を入力としたときのデータサイズに対する実行時間の変化

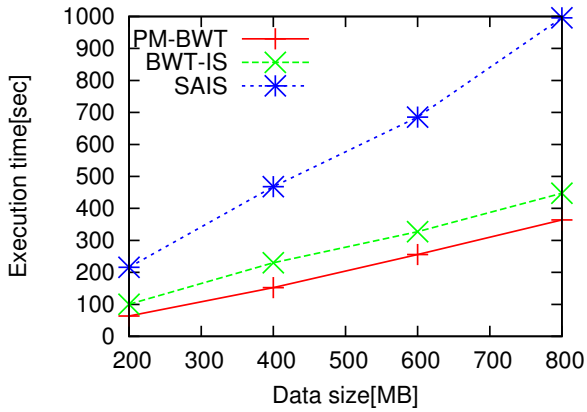

図 9.2. wikipedia を入力としたときのデータサイズに対する実行時間の変化

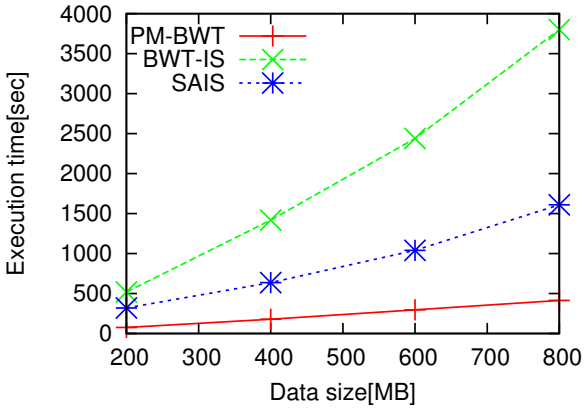

図 9.3. random を入力としたときのデータサイズに対する実行時間の変化

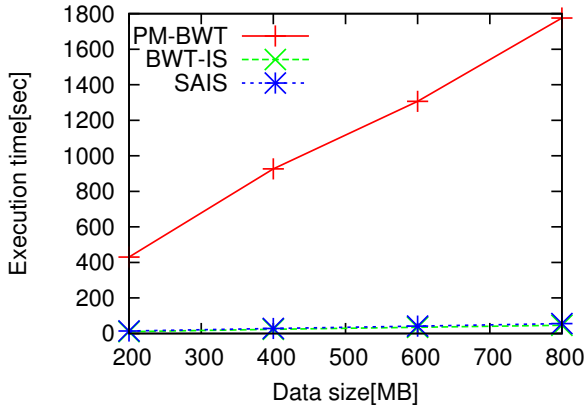

図 9.4. same を入力としたときのデータサイズに対する実行時間の変化

いほど高速に計算を行うことができる．それに対して, PM-BWT はリーフでの処理は SA-IS と類似した処理を行っているものの, マージ処理において文字列比較を多数行うため, 入力データがランダムであるほど高速に計算ができる傾向にある．それがこのような結果に繋がっていると考えられる．

それぞれの入力データに対するメモリ使用量を図 9.5, 9.6, 9.7, 9.8 に示す．これを見ると,

表 9.1. 入力テキスト

名前	説明	エントロピー
genome	人間のゲノム配列	3.23
wikipedia	wikipedia の記事	4.83
random	ランダム英数字	5.95
same	全て同じ文字からなるテキスト	0.00

BWT-IS のメモリ使用量が極めて小さいことが分かる．BWT-IS, PM-BWT 共に空間計算量は $O(n \log \log n)$ であるが, PM-BWT ではマージの際に T^{BWT}, サンプリングされた接尾辞配列, 及び mark ビット列という 3 つの配列を扱う必要があり, そのマージのためにさらに gap を構築するため, より多くのメモリを使用しているものと考えられる．

SA-IS の空間計算量は $O(n \log n)$ ビットであるが, グラフを見るとどれもデータ数に対してメモリ使用量が線形に増加している．SA-IS の空間計算量は, 接尾辞配列の 1 つの要素を表すのに $\log n$ ビット必要であり, それが n 個あることによるものであった．しかし, 実際の実装では数値は全て int 型で表現しており, 今回の実験では int 型で収まる大きさの入力データしか扱っていない．そのため, この実験においてはデータサイズに対して線形にメモリ使用量が増加するのが正しい挙動であると言える．

データの種類によるメモリ使用量の変化は BWT-IS では大きく, SA-IS と PM-BWT では小さい．SA-IS では入力テキストのエントロピーが大きいほど再帰処理の深さが大きくなる傾向にあるが, 今回利用した実装では再帰処理の際に接尾辞配列を格納する配列を使いまわすようになっており, 入力テキストの種類に対するに対する変化が小さい．PM-BWT については, これらのグラフではあまり変化がないように見えるが, コア数を変化させるとデータの種類ごとに開きが現れる．詳細は 9.3 節で述べる．

9.3　コア数に対する性能変化

コア数を変更したときに, 実行時間とメモリ使用量がどのように変化するかを調べた．この評価は AMD Opteron 6172 2.1GHz 48 コアの CPU, メモリ 64GB のマシンで行った．この評価ではデータサイズはすべて 200MB としている．

それぞれの入力データに対する実行時間を表 9.2, 9.3, 9.4, 9.5 に示す．表で w/o DCS と書かれているカラムは, DCS 構築以外の時間を表している．また, 各入力データに対するスケーラビリティを図 9.9 に示す．ここでは各入力データに対する 1 コアでの実行時間をそれぞれ 1 としてスケーラビリティを計算している．

まずスケーラビリティのグラフを見ると, same においては全くスケールしておらず, 8 コア以降では性能が悪化している．これは, マージ処理において gap を構築するときに, T_l^{BWT} の各要素を T_r^{BWT} に挿入する位置が全て同じになってしまい, 排他制御に時間がかかっているためであると思われる．

genome については, 32 コア以降で大きく性能が低下している．今回使用した人間のゲノムデータには, 途中で塩基が何であるか不明であることを表す ‘N’ という文字が約 20,000,000 個連続して現れる箇所があり, ここで same と同様, マージ時の排他制御に時間がかかっているものと思われる．それに比べて wikipedia, 及び random はよくスケールしている．random は当然として, wikipedia も同じ文字のパターンが繰り返すことが少ないのが原因だと思われる．

次に, 表の方に注目する．1 コアでの実行時間を比べると, same, genome, wikipedia, random の順に性能がよくなっていることが分かる．これは, リーフにおいて SA-IS の考え方を利用した BWT の計算を行っているためであると思われる．例えば same の場合だと, ほぼ全ての部分文字列において, LMS 型の suffix が存在せず, 結果としてリーフでの BWT の計算を $O(m)$ (m はリーフサイズ) で行うことができる．

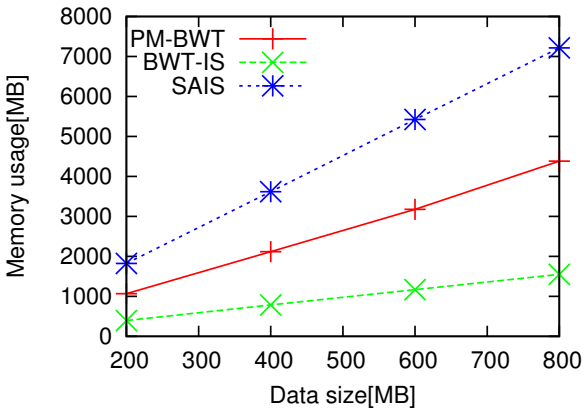

図 9.5. genome を入力としたときのデータサイズに対するメモリ使用量の変化

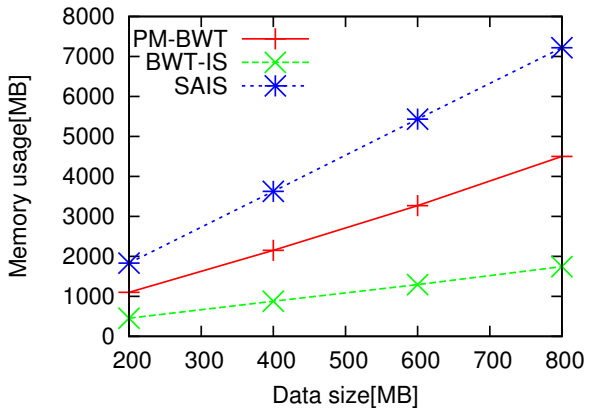

図 9.6. wikipedia を入力としたときのデータサイズに対するメモリ使用量の変化

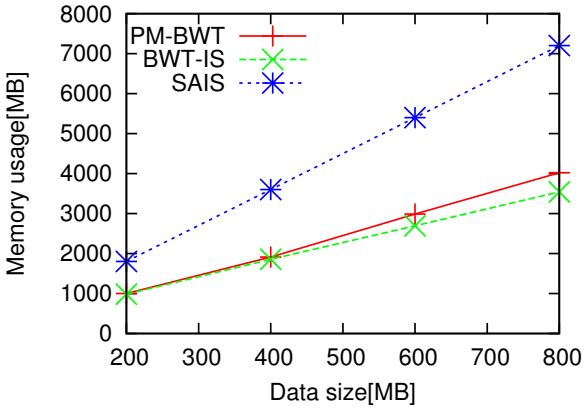

図 9.7. random を入力としたときのデータサイズに対するメモリ使用量の変化

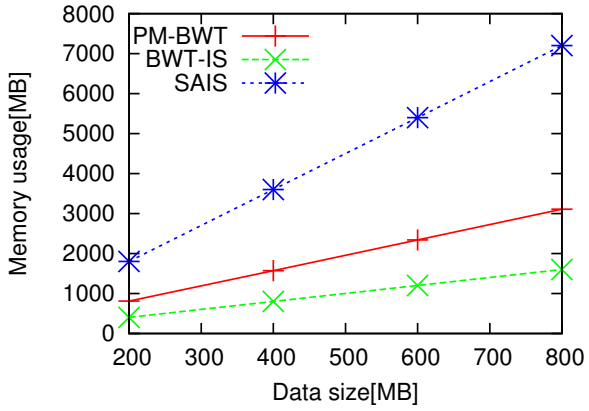

図 9.8. same を入力としたときのデータサイズに対するメモリ使用量の変化

DCS 構築の時間を比べてみると, まず random, wikipedia, 及び genome については, この順で性能がよくなることが分かる. 特に random については非常に高速に処理を行えていることが分かる. DCS 構築においては, まず最初の v 文字をソートする必要があったり, 内部で用いている DC3 では 3 文字のソートで順位が一意に決まらなければ再帰をしなければならないなど, テキストの冗長性に大きな影響を受ける. そのため, 特に random の場合だとほぼ通常の数値列のソートと変わらないため, 他と比べて高速に DCS を構築できることになる. same については, 1 コアと 2 コアで実行時間が大きく異なっていることが分かる. 通常, 並列プログラムにおいては並列化したプログラムを 1 コアで実行する場合と, そもそも並列化が行われていないプログラムを比べると, 並列化のための処理が含まれない分, 後者の方が高速に実行できるという傾向がある. そこで, 今回の実験においても, 1 コアで実行するときは suffix の先頭 v 文字をソートするのに並列文字列ソートではなくマルチキー・クイックソートを使っている. しかし, same の場合にマルチキー・クイックソートを行うと, suffix をピボットとの大小によって振り分ける際に, 毎回ほぼ全ての suffix が真ん中のバケツに入ることになる. 結果としてソート対象の suffix 数 $\times v$ 文字を触るこ

とになる. しかも, i 番目の再帰では全ての要素の i 文字目を順に見ていくことになるため, キャッシュヒット率が非常に悪くなる. 一方, 2 コア以上の場合に用いている BinLCP ソートでは, ランダムに選んでソートしたピボットが重複している場合, それを排除するようにしている. つまり, same の場合ではピボットはほとんどの場合で 1 つだけとなる. そのため, 通常のクイックソートのように要素を 2 つのバケツに分けることになる. ここで, ピボットと各 suffix を比較すると, やはりほぼすべての場合で v 文字全て同じ文字であるため, 結局 v 文字全てを比較しなければならない. しかし, ここでは文字列を先頭から順に走査していくため, キャッシュヒット率が高くなる. また, 結果としてほぼすべての suffix がピボットと一致するため, その後に各バケツをソートするという処理が不要となる. 結局触る文字数はマルチキー・クイックソートの場合と変わらないが, キャッシュヒット率が高い分だけこちらが高速になるのだと考えられる.

perf コマンドによって実際にキャッシュミス回数を測定したものを表 9.6 に示す. これを見ると, マルチキー・クイックソートの L1 キャッシュのロードミス回数は, BinLCP ソートに比べて約 51 倍, ラストレベルキャッシュのロードミス回数は約 88 倍となっており, 確かに BinLCP ソートの方がキャッシュミス回数が少ないことが分かる.

試しに 1 コアの場合でも DCS における先頭 v 文字のソートを BinLCP ソートで行うように変更したところ, DCS 構築の時間は 38.08[sec] となり, 2 コア以上の場合の実行時間と照らし合わせて整合性のある結果となった.

表 9.2. genome を入力としたときのコア数に対する実行時間の変化

コア数	PM-BWT		
	DCS[sec]	w/o DCS[sec]	total[sec]
1	182.95	918.52	1101.47
2	92.72	466.45	559.17
4	48.79	241.19	289.98
8	25.38	125.69	151.07
16	14.99	67.53	82.52
32	9.87	52.04	61.91
48	8.89	63.17	72.06

表 9.3. wikipedia を入力としたときのコア数に対する実行時間の変化

コア数	PM-BWT		
	DCS[sec]	w/o DCS[sec]	total[sec]
1	152.66	1473.00	1625.66
2	76.66	744.19	820.85
4	39.84	385.50	425.34
8	20.95	200.25	221.20
16	11.92	104.44	116.36
32	8.10	55.68	63.78
48	7.39	39.40	46.79

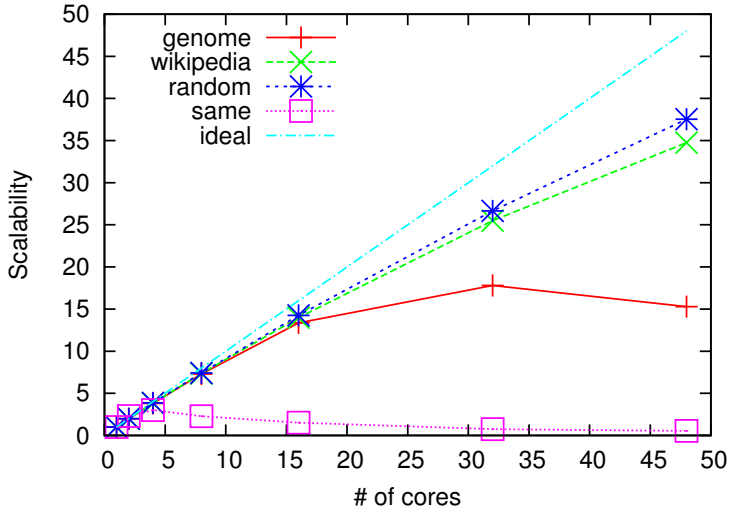

図 9.9. 各入力データに対するスケーラビリティ

表 9.4. random を入力としたときのコア数に対する実行時間の変化

コア数	PM-BWT		
	DCS[sec]	w/o DCS[sec]	total[sec]
1	22.47	1813.70	1836.17
2	11.47	913.23	924.70
4	6.05	470.34	476.39
8	3.15	244.74	247.89
16	1.91	126.87	128.78
32	1.64	67.25	68.89
48	1.74	47.16	48.90

表 9.5. same を入力としたときのコア数に対する実行時間の変化

コア数	PM-BWT		
	DCS[sec]	w/o DCS[sec]	total[sec]
1	260.93	740.46	1001.39
2	23.37	410.04	433.41
4	14.00	317.54	331.54
8	11.19	427.28	438.47
16	10.01	646.16	656.17
32	9.76	1323.33	1333.09
48	9.83	1867.29	1877.12

それぞれの入力データに対するメモリ使用量を図 9.10 に示す．これを見ると, genome と wikipedia ではコア数に応じてメモリ使用量が大きく増加していることが分かる．一方, random では 8 コアまではメモリ使用量が増加するものの, それより先ではほぼ変化しない．same に至ってはコア数に対してほぼ変化がない．

PM-BWT は大きく分けて 3 つのフェーズから成る．すなわち, DCS の構築, リーフでの BWT 計算, そしてマージである．このうちどのフェーズがメモリ使用量のボトルネックになっているのかを調べるため, DCS 構築を行うところまでのメモリ使用量を計測した．結果を図 9.11 に示す．これを見ると, genome, wikipedia, 及び same は図 9.10 の結果とほぼ同様の形状をしており, DCS 構築が最もメモリ使用量の多いフェーズであることが分かる．

DCS 構築において genome, wikipedia でコア数に応じてメモリ使用量が増えており, また random, same ではそうなっていない理由について考える．random については, 単純に最初の v 文字でのソートで全ての suffix がユニークになる確率が非常に高く, 再帰処理がほとんど行われないことが原因である．実際, 今回の実験条件では, random に対しては一度も再帰が行われない一方, 例えば same の場合は 12 回の再帰が行われることが分かっている．

再帰の回数のみで言えば genome は 10 回, wikipedia は 3 回であり, same より少ないが, same はこれらと比べてメモリ使用量が小さい．これは, 再帰の際のアルファベットサイズに関係がある．内部で使用している DC3 では, 先頭 3 文字でソートを行い, それによって得られる順位がユニークでない場合に, 現段階での順位として現れている数を新たなアルファベットとして再帰が行われる．same の場合, ほとんど全ての suffix の先頭 3 文字が一致してしまうため, 新たなアルファベットのサイズが小さくなる傾向にある．DC3 の内部でマージを並列に行っている箇所があり, そこでは入力を再帰的に分割し, ある程度小さくなったところで radix sort を行い, それらをマージしていくという処理を行っている．この radix sort において, アルファベットサイズに応じた配列を確保している箇所があり, これがメモリのボトルネックとなっている．実際には配列ではなく, gap を構築したのと同様のエンコーディング方法によってメモリ使用量を削減しているが, それでもアルファベットサイズが大きくなると, ここがボトルネックになってしまうようである．

9.4　v の値を変更したときの性能変化

9.3 節で述べたことを確かめるために, DCS 構築時のパラメータ v を変化させたときに, 実行時間とメモリ使用量がどのように変化するかを調べた．評価には 9.3 節と同じマシンを用いた．この評価では, 9.3 節でコア数に応じてメモリ使用量が大幅に増加した genome, 及び wikipedia のみ使用した．評価は 48 コア全てを使用して行った．実行時間の変化を表 9.7, 9.8 に示す．DCS の構築時間は genome, wikipedia ともに v と共に減少している．v の値に対して DCS の要素数は $O(n/\sqrt{v})$

表 9.6. same を入力として 1 コアで実行したときのマルチキー・クイックソートと BinLCP ソートのキャッシュミス数の比較

	マルチキー・クイックソート	BinLCP ソート
L1-dcache-load-misses	1.21×10^{10}	2.38×10^{8}
LLC-load-misses	1.20×10^{10}	1.36×10^{8}

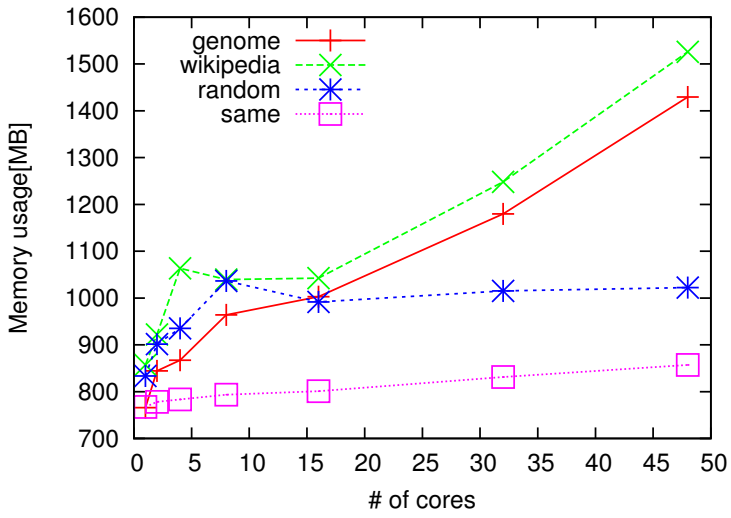

図 9.10. 各入力データにおけるコア数に対するメモリ使用量の変化

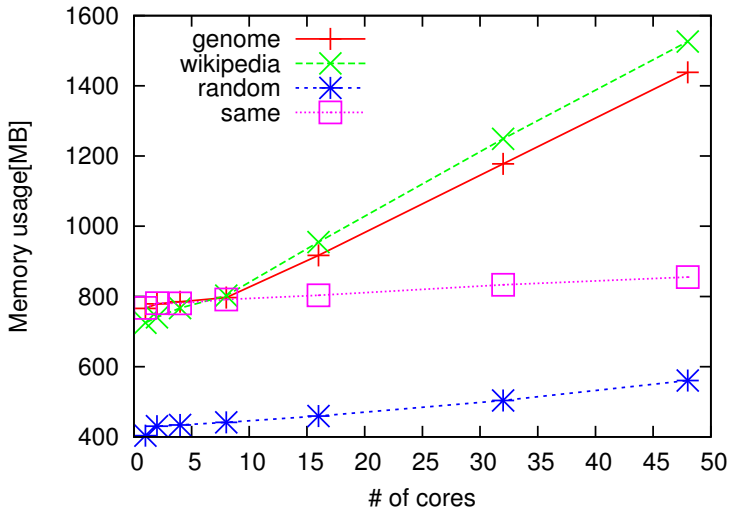

図 9.11. 各入力データにおけるコア数に対するメモリ使用量の変化 (DCS 構築まで測定)

となっているため, v の値が 200 から 400 へ変わるところで大きく実行時間が変化している. DCS 以外の実行時間を見ると, genome では 200 から 400 にかけて大きく増加したが, 600 から 800 にかけて減少している. また wikipedia については, v の値に対してほとんど変化がない. v の値が大きいということは, 最悪の場合にそれだけ多くの文字を比較しなければならないことを意味するが, 特に wikipedia については最悪の場合というのは頻繁に発生しないため, このような結果になっているものと思われる.

また, ここには示していないが, 実際にはこの実験結果には大きなばらつきがあり, 例えば genome で $v = 800$ とした実験における 3 回の測定結果の標準偏差は 6.27 となっている. つまり, genome の 600 から 800 の間での減少は誤差の範囲内であり, 実際にはそれほど大きな差が生じていない

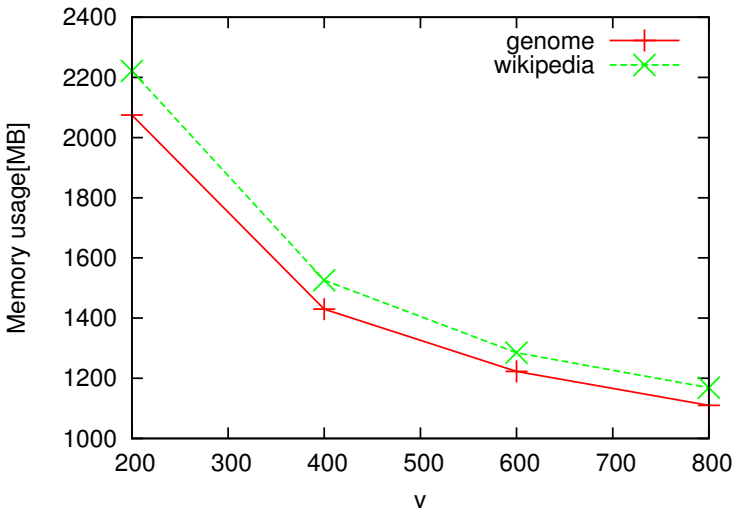

図 9.12. v を変化させたときのメモリ使用量の変化

ものと思われる．しかし，もし DCS を構築しない場合，genome などにおいては実行時間が非常に増加することが確認されており，DCS の構築が不要なわけではない．

次に，メモリ使用量の変化を図 9.12 に示す．これを見ると，どちらの場合も v の値に応じてメモリ使用量が減少していることが分かる．先の節でも述べた通り，genome と wikipedia では DCS がメモリ使用量のボトルネックとなっているため，v の値を増加させ，DCS の要素数を少くすることでメモリ使用量が減少したものと思われる．

表 9.7. genome を入力として v を変化させたときの実行時間の変化

v	PM-BWT		
	DCS[sec]	w/o DCS[sec]	total[sec]
200	15.95	51.63	67.58
400	8.89	63.17	72.06
600	6.76	63.78	70.54
800	6.05	60.10	66.15

表 9.8. wikipedia を入力として v を変化させたときの実行時間の変化

v	PM-BWT		
	DCS[sec]	w/o DCS[sec]	total[sec]
200	15.16	38.94	54.11
400	7.39	39.40	46.79
600	5.66	39.37	45.03
800	4.45	39.38	43.83

第10章　結論

10.1　まとめ

本研究ではテキストの任意の部分文字列を検索するための圧縮全文検索インデックスを構築する過程で必要な Burrows-Wheeler 変換を省メモリかつ並列に計算する手法を提案した. 提案手法では分割統治法により入力テキストを再帰的に分割し, それによって得られる部分文字列に対して接尾辞配列を構築し, BWT を計算することにより, 多くのメモリを消費する接尾辞配列をテキスト全体に対して構築することを防いだ. 部分文字列に対する接尾辞配列の構築に対しても, 既存の高速な接尾辞配列構築アルゴリズムのアイディアを適用した. また, BWT の性質を利用したマージ手法を用いることで, マージの計算量を小さくした. さらに, 2 つの suffix 同士の辞書的な大小を比較する際に必要な文字比較の回数を小さく抑えるための補助的なデータ構造を利用することで, 逐次実行時の最悪計算量は $O(n \log^3 n)$ となり, さらにタスク並列モデルによる並列化を行うことで, 最悪クリティカルパスは $O(\log \sigma \log^5 n)$ を達成した. テキスト全体の接尾辞配列を構築することなく $o(n)$ の時間計算量で BWT を計算した例は我々の知る限り他に存在せず, 提案手法は有用だと言える.

本研究では提案手法をタスク並列処理系 MassiveThreads を使って実装し, 評価を行った. 評価の結果, 人間のゲノムデータや英文テキストのような実用的な入力データを用いた場合などに, メモリ使用量をある程度削減しながら, 既存手法より高速に BWT を計算可能であることが分かった.

10.2　今後の展望

提案手法では, 人間のゲノムデータや 1 種類の文字だけで構成されるようなテキストを入力とした場合に, スケーラビリティが良くないことが分かった. これの主な原因は, マージの際に同一要素を並列に更新するための排他制御にある. よりスケーラブルなマージを達成する方法として, 1 つには gap 構築時に各スレッドがスレッドローカルなメモリ領域に自身の担当範囲の計算結果を格納し, それを最後に統合するという方法が考えられる. しかし, これだとコア数に比例してメモリ使用量が増加してしまう. そのため, このトレードオフを解決する方法を考える必要がある.

また, 提案手法は既存の高速な手法に速度では優っているものの, メモリ使用量でやや劣っている. 一般的に並列アルゴリズムは逐次の高速なアルゴリズムに比べて多くのメモリを使用する傾向にあり, メモリ使用量で洗練された既存手法を上回るのは難しいと考えられる. しかし, マージの際に使用する mark ビット列を圧縮した形で直接構築するアルゴリズムを考案するなど, 改善の余地は存在するものと考えられる.

謝辞

まず, 本研究を行うにあたり, 研究テーマ決めから始まり, 研究の方向性, 実装, 論文執筆に至るまで, 様々なお力添えを頂きました田浦健次朗准教授に深く感謝します. 研究に行き詰まったときに先生の居室に伺うと, お忙しい時であってもいつも手を止めて私の相談に親身に乗って頂けたことや, 学会のために論文を執筆していたときに, 大変長時間に渡りみっちりと指導して頂いたことなど, 大変嬉しく思っております. 先生の存在なくして私の充実した修士生活はあり得ませんでした.

また, 博士 3 年の中島潤先輩には, 実装面で悩んでいたときにいつも快く相談に乗って頂き, プログラムの性能を向上させるためのノウハウなど多くのことを教えて頂きました. 博士 2 の秋山茂樹先輩には, 学会発表の練習で私の拙い発表に何度もアドバイスを頂き, 少なからず発表の質を向上させることができました. 同期の中澤君には, 良き友人として, ある時には真剣に研究の議論をし, ある時には修士生活の辛さを分かち合い, またある時にはけん玉で互いを高め合いました. 中澤君のおかげで楽しい修士生活を送ることができました. 同期の中谷君には, 技術的なことで困った際に度々助けて頂き, また研究室のサーバ管理なども率先してやって頂きました. 同期の Amgaa 君には, ぶれない生き方とはどういうものかという事を身を持って見せて頂き, 今後の人生の参考になりました. 修士 1 年の菊地君には, 研究室サーバーの管理において大変尽力して頂きました. 修士 1 年の早水君には, 電気系計算機管理 TA として一緒に仕事をして頂いた他, 良き話し相手として, 日々面白い話題を提供して頂きました. 修士 1 年の西岡君には, 研究室の掃除当番決めやスポーツ大会の連絡係など, 円滑な研究室運営に貢献して頂きました. その他にも研究生活を送る中で本当に多くの方々にお世話になりました. この場を借りて厚くお礼申し上げます.

最後に, 2 年間の修士生活の間, 金銭面も然る事ながら, 精神面で支えてくれた家族に感謝します. 私自身が修士生活を無事に送ることができたのも, 故郷で家族が応援してくれていたおかげだと思っています. ありがとうございました.

発表文献

[1] 林伸也，田浦健次朗．文字列検索における圧縮インデックス構築の省メモリな並列化手法．**先進的計算基盤システムシンポジウム** *(SACSIS2013)*, 仙台, 2013/5.

[2] Shinya Hayashi, Kenjiro Taura. Parallel and Memory-efficient Burrows-Wheeler Transform. *BigData In Bioinformatics and Health Care Informatics*, Santa Clara, 2013/10.

参考文献

[1] Read & researchmap. `http://researchmap.jp/sada/csalib/`.

[2] UCSC Genome Bioinformatics. `http://genome.ucsc.edu/`.

[3] V Arlazarov, E Dinic, M Konrod, and I Faradzev. On economic construction of the transitive closure of a directed graph. *Sov.Math. Dokl*, pages 1209–1210, 1975.

[4] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. *In Proceedings of the 8th ACM—SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.

[5] Timo Bingmann and Peter Sanders. Parallel string sample sort. *arXiv preprint arXiv:1305.1157*, 2013.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. of PPoPP '95*, pages 207–216, 1995.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, September 1999.

[8] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical report, 1994.

[9] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 383–391, 1999.

[10] Francisco Claude and Gonzalo Navarro. The wavelet matrix. *Proc. SPIRE'12*, pages 167–179, 2012.

[11] Charles J. Colbourn and Alan C.H. Ling. Quorums from difference covers. *Information Processing Letters*, 75:9–12, 2000.

[12] J. Shane Culpepper, Matthias Petri, and Falk Scholer. Efficient in-memory top-k document retrieval. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 225–234, New York, NY, USA, 2012. ACM.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[14] P. Ferragina and G. Manzini. Opportunistic data structures with applications. *FOCS '00*, pages 390–398, 2000.

[15] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly fm-index. In Alberto Apostolico and Massimo Melucci, editors, *String Processing and Information Retrieval*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160. Springer Berlin Heidelberg, 2004.

[16] Paolo Ferragina Travis Gagie and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[17] Roberto Grossi and Ankur Gupta. High-order entropy-compressed text indexes. *SODA '03 Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, 2003.

[18] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *STOC '00 Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 397–406, 2000.

[19] Wing-Kai Hon, Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, pages 23–36, 2007.

[20] Juha Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

[21] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linearwork suffix array construction. *Journal of the ACM*, 53:918–936, 2006.

[22] Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33:605–612, 2007.

[23] Ross A. Lippert, Clark M. Mobarry, and Brian P. Walenz. A space-efficient construction of the burrows—wheeler transform for genomic data. *Computational Biology*, 2005.

[24] Veli Mäkinen. Compact suffix array. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, COM '00, pages 305–319, London, UK, UK, 2000. Springer-Verlag.

[25] Veli Mäkinen. Compact suffix array — a space-efficient full-text index. *Fundam. Inf.*, 56(1-2):191–210, January 2003.

[26] Veli Mäkinen and Gonzalo Navarro. Run-length fm-index. In *In Proc. DIMACS Workshop: “ The Burrows-Wheeler Transform: Ten Years Later*, pages 17–19, 2004.

[27] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SODA '90*, pages 319–327, 1990.

[28] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz. Rapid parallel genome indexing with mapreduce. *MapReduce '11 Proceedings of the second international workshop on MapReduce and its applications*, pages 51–58, 2011.

[29] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 657–666, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[30] Jun Nakashima. MassiveThreads: A Lightweight Thread Library for High Productivity Languages, 2012.

[31] Gonzalo Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *CoRR*, abs/1304.6023, 2013.

[32] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[33] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1066–1077. SIAM, 2012.

[34] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. *Data Compression Conference*, pages 193–202, 2009.

[35] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear suffix array construction. *IEEE Transactions on Computers*, 60(10), 2011.

[36] Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. *SPIRE '09*, pages 90–101, 2009.

[37] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[38] 岡野原 大輔. **高速文字列解析の世界**. 岩波書店, 2012.