

修士(工学)学位論文
Master's Thesis of Engineering
Cerium による文字列の並列処理
Parallel processing of strings using Cerium

2016年3月

March 2016

古波倉 正隆

Masataka Kohagura



琉球大学

大学院理工学研究科

情報工学専攻

Information Engineering Course
Graduate School of Engineering and Science
University of the Ryukyus

指導教員：教授 和田 知久

Supervisor: Prof. Tomohisa WADA

要 旨

Cerium は当研究室で開発している並列プログラミングフレームワークである。

従来はファイル読み込みを `mmap` で実装していたが、本論文では並列処理向け I/O の Blocked Read を実装を行った。Blocked Read とは、ファイルを一度に読み込まずに、あるサイズに分割して読み込む手法である。

Cerium にはファイルを読み込んで文字列処理を行う例題があり、Word Count、Boyer-Moore String Search、正規表現を実装し測定した。それぞれの例題によって結果の整合性を取る必要があるが、どのように整合性を取るかは問題によって考慮する必要がある。

本研究で実装した正規表現は、正規表現から正規表現木を生成し、その正規表現木に状態を割り振りながら NFA を生成する。もし NFA が生成した場合は Subset Construction で DFA に変換する。そして、DFA の生成後、ファイルとマッチングさせる。

それぞれの例題と Mac に built-in されている `wc`、`egrep` と比較し測定を行なった。

Abstract

Cerium is a parallel programming framework, which developed by our laboratory. We implemented file read with mmap in parallel string search. It is easy to use mmap system call to read from file, but current implementation of mmap sometimes does work well. So we implement 'Blocked Read', that is reading file separated by a certain size.

Examples of parallel string search are included in Cerium. They are Word Count, Boyer-Moore String Search and Regular Expression.

It needs to check for correct result in divided file every examples.

Implemented Regular Expression sequential execute. At first, convert Regular Expression to binary tree. Second, allocated state to binary tree. Third, convert state with Subset Construction. Finally, implement parallel regular expression.

We evaluate examples of Cerium, wc of Mac built-in, and egrep.

目次

第1章	文字列処理の並列処理	2
第2章	Cerium	3
2.1	Cerium TaskManager	3
第3章	並列処理向け I/O	7
3.1	mmap	7
3.2	Blocked Read	8
3.3	I/O 専用 thread の追加	9
第4章	Cerium による文字列処理の例題	11
4.1	文字列処理の並列処理	11
4.2	Word Count	12
4.3	Boyer-Moore String Search	15
4.4	正規表現	20
4.4.1	正規表現木の生成	21
4.4.2	正規表現木への状態の割当	26
4.4.3	Subset Construction による状態の変換	27
4.4.4	正規表現マッチャの並列処理の実装	28
第5章	ベンチマーク	36
5.1	Word Count	36
5.2	Boyer-Moore String Search	38
5.3	正規表現	38
第6章	結論	41
6.1	今後の課題	41
	参考文献	43

目次

2.1	Task Manager	4
3.1	mmap Model	8
3.2	BlockedRead Model	9
3.3	BlockedRead と Task を同じ thread で動かした場合	9
3.4	IO Thread による BlockedRead	10
4.1	File 読み込みから処理までの流れ	11
4.2	ファイル分割無しの Word Count	13
4.3	ファイル分割有りの Word Count	13
4.4	力まかせ法	15
4.5	pattern に含まれていない文字で不一致になった場合	16
4.6	pattern に含まれている文字で不一致になった場合	17
4.7	pattern に同じ文字が複数入り、その文字で不一致になった場合	17
4.8	分割周りの処理	18
4.9	3つの表記ゆれの文字列を1つの正規表現にまとめる	20
4.10	正規表現から正規表現木への変換の例	22
4.11	文字の接続	23
4.12	文字列の接続	23
4.13	選択	24
4.14	繰返し	24
4.15	グループ	25
4.16	正規表現の接続	25
4.17	与えられた正規表現木に状態を割り振る	26
4.18	b が入力されるとどちらに遷移すればいいのかわからない	27
4.19	b が入力されたときは新しい状態を作り、新しい状態に遷移させる	27
4.20	Subset Construction による新しい状態の生成	29
4.21	複数の文字クラスを Merge するときの全パターン	30
4.22	StateArray による状態遷移	31
4.23	分割された部分に正規表現がマッチングする場合の処理	34

6.1	文字単位の状態割り振りを文字列単位での状態割り振りに変更	42
-----	----------------------------------------	----

表 目 次

2.1	Task 生成における API	5
2.2	Task 側で使用する API	5
4.1	サポートしているメタ文字一覧	21
4.2	メタ文字の結合順位	21
5.1	ファイル読み込みを含む Word Count	37
5.2	ファイル読み込み無しの Word Count	37
5.3	力任せ法と Boyer-Moore String Search の比較	38
5.4	ファイル読み込み有りと無しを変化させた各 grep の結果	39
5.5	ファイルサイズを変化させた各 grep の結果	39
5.6	正規表現の状態数を増やした Grep の結果	40
5.7	全くマッチングしないパターンを grep した結果	40

第1章 文字列処理の並列処理

世界中のサーバには様々な情報や Log が保管されており、それらのテキストファイル全体のデータサイズを合計すると TB 単位ととても大きなサイズになると予想される。それらの中から特定の文字列や正規表現によるパターンマッチングを探すなどの文字列処理には膨大な時間がかかる。検索時間を短縮するためには、ファイルの読み込み時間を軽減し、プログラムの並列度をあげる必要がある。

Cerium は並列プログラミングフレームワークであり当研究室で開発している。文字列処理を Cerium に実装するにあたり、ファイルの読み込み方法について改良を行なった。ファイルの読み込みを行なったあとに文字列処理が走っていたが、ファイルの読み込みと文字列処理が同時に走るように改良した。

文字列の並列処理の例題として、Word Count、正規表現の実装を行なった。文字列処理を並列実行する際、ファイルを一定の大きさに分割して、それぞれに対して文字列処理を行う。それぞれの分割されたファイルに対して処理を行なったあと結果が出力される。

それぞれの結果を合計する際に、分割された部分に対して各例題工夫をすることによって整合性を取る必要がある。

本論文では文字列処理だけでなくファイルの読み込みまでを含む文字列処理を考慮した並列処理を実装し、処理全体の速度を上げるような実装を行なった。

第2章 Cerium

Cerium は、本研究室で開発している並列プログラミングフレームワークで Cell 向けに開発されており、C/C++ で実装されている。Cell は Sony Computer Entertainment 社が販売した PlayStation3 に搭載されているヘテロジニアスマルチコア・プロセッサである。現在では Linux、MacOS X、CUDA 上で動作する並列プログラミングフレームワークである。Cerium は TaskManager、SceneGraph、Rendering Engine の3要素から構成されており、本研究では汎用計算フレームワークである TaskManager を利用して文字列の並列計算を行なっている。本章では Cerium TaskManager の構成と Cerium TaskManager を利用したプログラムの実装例について説明する。

2.1 Cerium TaskManager

Cerium Task Manager は、User が並列処理を Task 単位で記述し、関数やサブルーチンを Task として扱い、その Task に対して Input Data、Output Data を設定する。Input Data、Output Data とは関数でいう引数に相当する。そして Task が複数存在する場合、それらに依存関係を設定することができる。そして、それに基づいた設定の元で Task Manager にて管理し実行される。

図2.1 は Cerium が Task を作成・実行する場合のクラスの構成となる。User が createtask を行い、input data や Task の依存関係の設定を行うと、TaskManager で Task が生成される。Task Manager で依存関係が解消されて、実行可能になった Task は ActiveTaskList に移される。ActiveTaskList に移された Task は依存関係が存在しないのでどのような順序で実行されてもよい。Task は Scheduler に転送しやすい TaskList に変換してからデバイスに対応する Scheduler に Synchronized Queue である mail を通して転送される。

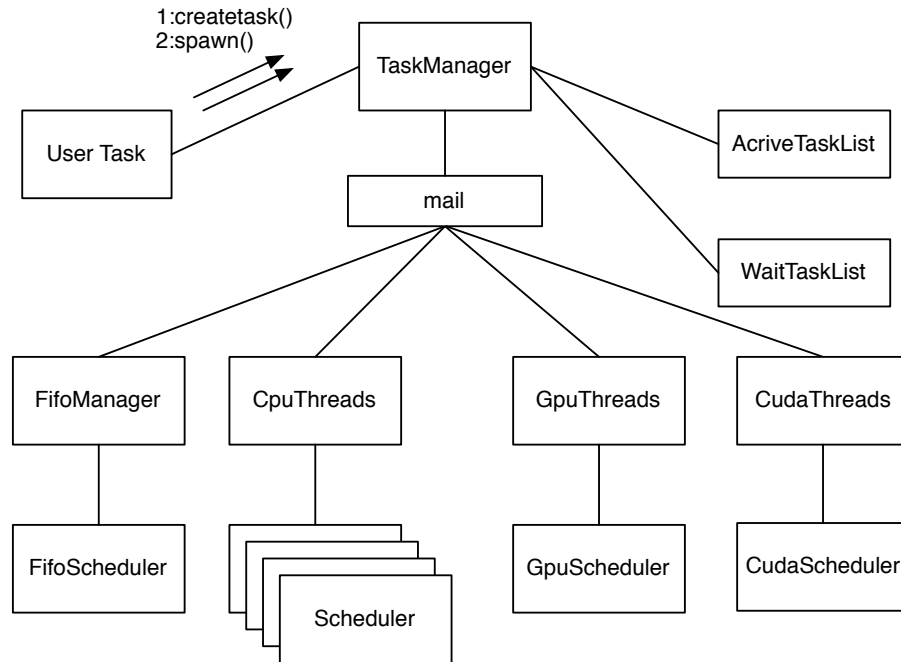


図 2.1: Task Manager

Input Data で格納した 2 つの数を乗算し、Output Data に演算結果を格納する multiply という例題のソースコード 2.1 を以下に示す。

また、Task の生成時に用いる API 一覧を表 2.1 に示す。

ソースコード 2.1: Task の生成

```

1 multi_init(TaskManager *manager)
2 {
3     float *A, *B, *C;
4
5     // create Task
6     HTaskPtr multiply = manager->create_task(MULTIPLY_TASK);
7
8     // set device
9     multiply->set_cpu(SPE_ANY);
10
11    // set inData
12    multiply->set_inData(0, (memaddr)A, sizeof(float)*length);
13    multiply->set_inData(1, (memaddr)B, sizeof(float)*length);
14
15    // set outData
16    multiply->set_outData(0, (memaddr)C, sizeof(float)*length);
17
18    // set parameter
19    multiply->set_param(0, (long)length);

```

```

20|
21| // spawn task
22| multiply->spawn();
23| }

```

create_task	Task を生成する
set_inData	Task への入力データのアドレスを追加
set_outData	Task への出力データのアドレスを追加
set_param	Task へ値を一つ渡す。ここでは length
set_cpu	Task を実行するデバイスの設定
spawn	生成した Task を TaskList に set

表 2.1: Task 生成における API

次に、デバイス側で実行される Task のソースコードを 4.9 に示す。

ソースコード 2.2: Task

```

1 static int
2 run(SchedTask *s) {
3     // get input
4     float *i_data1 = (float*)s->get_input(0);
5     float *i_data2 = (float*)s->get_input(1);
6
7     // get output
8     float *o_data = (float*)s->get_output(0);
9
10    // get parameter
11    long length = (long)s->get_param(0);
12
13    // calculate
14    for (int i=0; i<length; i++) {
15        o_data[i] = i_data1[i] * i_data2[i];
16    }
17    return 0;
18 }

```

また表 2.2 は Task 側で利用する API である。Task 生成時に設定した Input Data や parameter を取得することができる。

表 2.2: Task 側で使用する API

get_input	Scheduler から input data を取得
get_output	Scheduler から output data を取得
get_param	set_param した値を取得

Task 生成時に設定できる要素を以下に列挙する。

- Input Data
- Output Data
- Parameter
- CpuType
- Dependency

Input/Output Data、Parameter は関数の引数に相当する。Cpu Type は Task を動作させるデバイスを設定することができ、Dependency は他の Task との依存関係を設定することができる。

第3章 並列処理向け I/O

ファイル読み込みなどの I/O を含むプログラムは、読み込み時間が Task の処理時間と比較して大きくなってしまいます。計算処理の並列化を図ったとしても I/O がボトルネックになるので、読み込み時間の長さだけプログラム全体の処理速度が遅くなってしまいます。従来の例題のファイル読み込み部分では mmap を利用していたが、読み込みと Task が並列に動くような実装を行ない、プログラム全体の高速化を図った。

本章では mmap による読み込みと並列処理向け I/O について述べる。

3.1 mmap

Cerium の従来の例題ではファイル読み込みを mmap にて実装していた。mmap は function call 後にすぐにファイルを読みに行くのではなく、仮想メモリ領域にファイルの中身を対応させ、その後メモリ空間にアクセスされたときに OS が対応したファイルを読み込む。そのため、mmap によるファイルを読み込みは読み込み後に Task を実行するので、その間は他の CPU が動作せず並列度が落ちる。

また、読み込む方法が OS 依存となってしまうため環境に左右されやすく、プログラムの書き手が読み込みの制御をすることが難しい。

図 3.1 は mmap で読み込んだファイルに対して Task1、Task2 が並列で動作し、それぞれの Task がファイルにアクセスしてそれぞれの処理を行うときのモデルである。

Task1 が実行されると仮想メモリ上に対応したファイルが読み込まれ、読み込み後 Task1 の処理が行われる。それと同時に Task2 も Task1 と同様の処理が行われるが、Task1 が読み込みをしている間 Task2 が読み込みを行わない。

また、Task1 が実行されるときに初めてそれらの領域にファイルが読み込まれるので、Task1 の文字列処理を実行しない限り Task2 がさらに待たされてしまう。

mmap によるファイルの読み込みは Task と並列に実行されるべきであるが、OS の実装に依存してしまう。

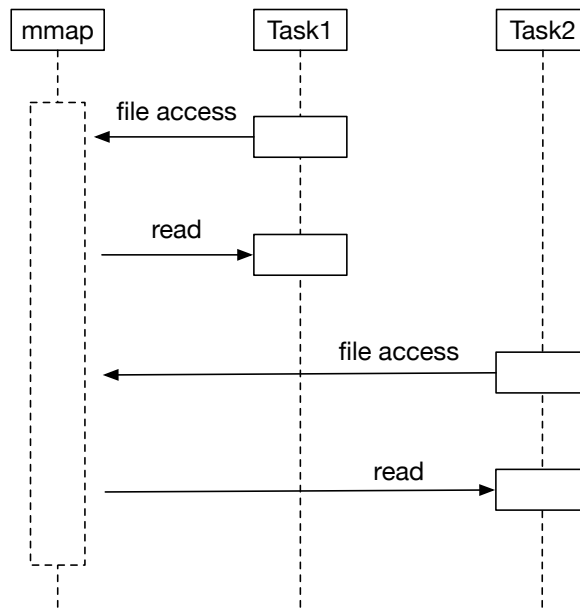


図 3.1: mmap Model

3.2 Blocked Read

mmap ではファイル読み込みを細かく設定することができないので、読み込みを制御できるように実装した。さらに、読み込みと Task が並列に動作するようにした。

読み込みを独立した Thread で行ない、ファイルをある程度の大きさ (Block) ごとに読み込む。そして、読み込まれた部分に対して並列に Task を起動する。これを Blocked Read と呼び、I/O の読み込みと Task の並列化を図った。

ファイルを読み込む Task (以下、Blocked Read) と、読み込んだファイルに対して計算を行う Task を別々に生成する。Blocked Read は一度にファイル全体を読み込むのではなく、ある程度の大きさで分割してから読み込みを行う。分割して読み込んだ範囲に対して Task を実行する。

図 3.2 では、Task を n 個単位でまとめた Task Block を生成する。1つの Task はファイルの長さ L の部分を処理する。したがって、1つの Task Block は $L \times n$ の部分を処理する。

Task Block で計算される領域が Blocked Read で読み込む領域を追い越して実行してし

まうと、まだ読み込まれていない領域に対して計算されてしまう。Blocked Read による読み込みが終わってから TaskBlock が起動されるように、Cerium の API である wait_for を使って待ち合わせを行う。

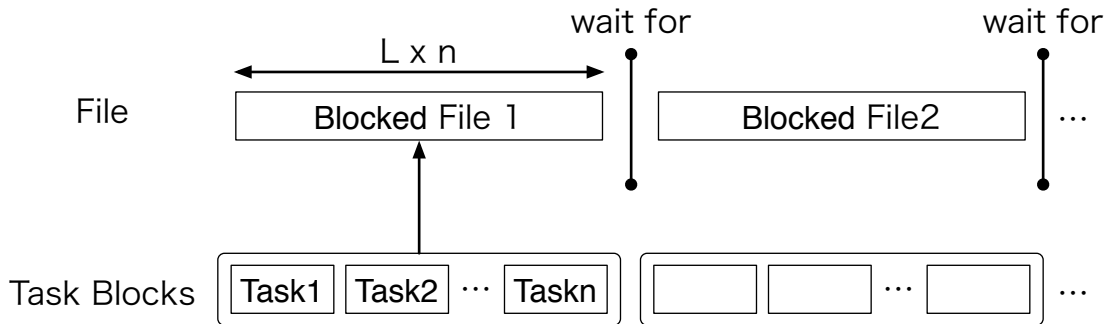


図 3.2: BlockedRead Model

3.3 I/O 専用 thread の追加

Blocked Read は読み込みを含む処理なので、Blocked Read 1 つあたりの処理時間は大きくなる。Blocked Read がファイルを読み込む前提で Task がその領域に対して計算を行うので、Blocked Read の処理によってプログラム全体の処理速度が左右されてしまう。

Cerium Task Manager では、それぞれの Task に対してデバイスを設定することができる。SPE_ANY 設定をすると、Task Manager が CPU の割り振りを自動的に行う。しかし、自動的に割り振りを行ってしまうと、Blocked Read Task 間に Task が割り込まれてしまい、読み込みが遅延してしまう可能性がある。(図 3.3)

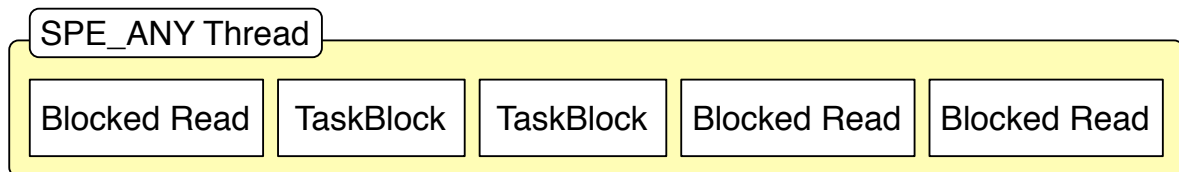


図 3.3: BlockedRead と Task を同じ thread で動かした場合

そこで、Task が Blocked Read Task 間に割り込まれないようにするため、I/O 専用 thread である iO_0 の設定を追加した。

IO_0 は SPE_ANY とは別 thread の scheduler で動作するので、SPE_ANY で動作している Task に割り込むことはない。しかし、読み込みの終了を通知し、次の read を行う時に他の Task がスレッドレベルで割り込んでしまう事があるため、pthread_getschedparam() で IO_0 の priority の設定を行う必要がある (図:3.4)。

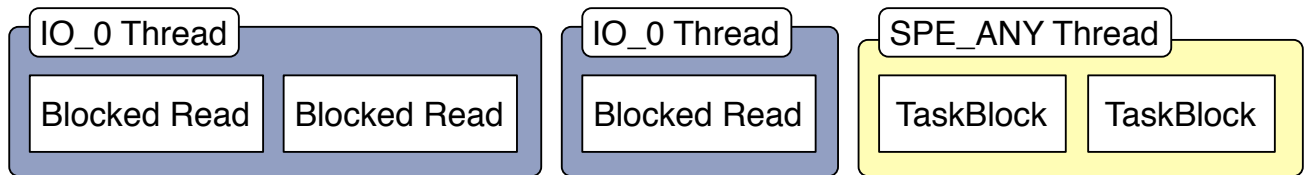


図 3.4: IO Thread による BlockedRead

第4章 Cerium による文字列処理の例題

本項ではファイルを読み込んで文字列処理を並列処理をする流れと例題を記述する。

Cerium に実装している例題として、Word Count、Boyer-Moore String Search、正規表現を紹介する。

4.1 文字列処理の並列処理

文字列処理を並列で処理する場合を考える。まずファイルを読み込み、ファイルのある一定の大きさで分割する。そして、分割されたファイル (Input Data) に対して文字列処理 (Task) をおこない、それぞれの分割単位で結果を出力する (Output Data)。それらの Output Data の結果が出力されたあとに、結果をまとめる処理を行う (Print Task)。(図 4.1)

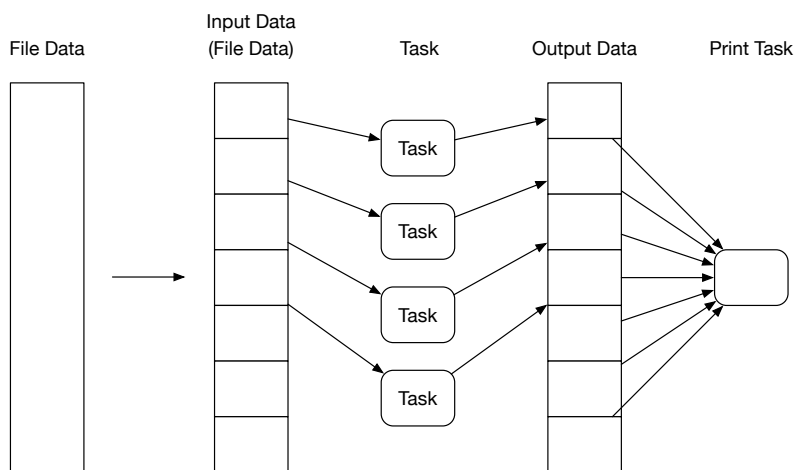


図 4.1: File 読み込みから処理までの流れ

ファイルを読み込んで文字列処理をする流れを 1 つのクラスとして Cerium 内に組み込んだ。Cerium で文字列処理の並列処理を記述する際にこのクラスを利用すれば、自動的にファイルのある程度のサイズに分割し、文字列処理の Task と結果を表示する Print

Task の依存関係も設定される。このクラスは、ファイルをマッピングし処理をすることで小さいデータの集合を出力することから FileMapReduce と名付けた。

Cerium のプログラムを記述する際には TMmain 関数が C の main 関数に相当する。TMmain 関数内に Task の生成や依存関係を設定する。これまではファイルを分割して読み込む Task や文字列処理の Task を生成し、依存関係を設定するプログラムを記述していた。FileMapReduce を実装したことにより、FileMapReduce を呼び出すだけでそれらの設定を行うようになった。

ソースコード 4.1: FileMapReduce を利用する際の TMmain の記述

```

1 int
2 TMmain(TaskManager *manager, int argc, char *argv[])
3 {
4     char *filename = 0;
5     FileMapReduce *fmp = new FileMapReduce(manager, TASK_EXEC,
6         TASK_EXEC_DATA_PARALLEL, TASK_PRINT);
7     filename = fmp->init(argc, argv);
8     if (filename < 0) {
9         return -1;
10    }
11    /*
12     * 文字列処理後に出力されるデータの数を設定する。
13     */
14    fmp->division_out_size = sizeof(unsigned long long)*4;
15    task_init();
16    fmp->run_start(manager, filename);
17    return 0;
18 }

```

TASK_EXEC TASK_EXEC_DATA_PARALLEL GPU TASK_PRINT 集計 Task

文字列処理の Task と結果の整合や表示を行う Print Task をそれぞれ決められたフォーマットに沿って記述すればよい。ファイルを分割して文字列処理を行なった際、分割された部分でそれぞれの例題の整合性が取れなくなってしまうことがある。

文字列処理や結果の整合性の取り方についてはそれぞれの例題にて述べる。

4.2 Word Count

Word Count は読み込んだテキストに対して単語数を数える処理である。Input Data には分割されたテキストが対応しており、Output Data には単語数と行数を出力する。

読み込んだテキストを先頭から見ていき、単語の末端に空白文字か改行文字があれば単語数、改行文字があれば行数を数えることができる。

分割された部分に単語が含まれた場合、単語数や行数について整合性を取る必要がある。図 4.2 ではファイル分割無しの Word Count である。

分割しない状態では単語数 (Word Num) 3、行数 (Line Num) 2 となる。

w	o	r	d		w	o	r	d	\n	w	o	r	d	\n	\0
---	---	---	---	--	---	---	---	---	----	---	---	---	---	----	----

Word Num : 3

Line Num : 2

図 4.2: ファイル分割無しの Word Count

図 4.3 では単語で分割された場合である。分割された 1 つ目のファイルは単語数 1 となる。単語と認識されるためには空白か改行が必要である。しかし 1 つ目のファイルには空白または改行が 1 つしか含まれていないため、単語数は 1 となってしまう。2 つ目のファイルは改行が 2 つあるにも関わらず、単語数は 1 となる。ファイルの先頭に空白、改行が含まれていた場合は単語が現れていないにも関わらず単語数 1 とカウントされてしまう。この場合は単語数をカウントしないようにする。

分割されたファイルそれぞれの結果を合計すると単語数 2、行数 2 となり、分割されていない時と結果が変わってしまう。

w	o	r	d		w	o	r	d	\n	w	o	r	d	\n	\0
---	---	---	---	--	---	---	---	---	----	---	---	---	---	----	----

Word Num : 1

Word Num : 1

Line Num : 0

Line Num : 2

図 4.3: ファイル分割有りの Word Count

この問題の解決方法として、分割されたファイルの一つ目が文字で終わり、二つ目のファイルの先頭が改行または空白で始まった場合はそれぞれの単語数の合計数から 1 足すことにより整合性を取ることができる。ソースコード 4.3 の 18-22 行目にてその処理を行なっている。

ソースコード 4.2 は Word Count のメインルーチンである。

ソースコード 4.2: Word Count の記述

```

1 SchedDefineTask1(Exec,wordcount);
2
3 static int
4 wordcount(SchedTask *s, void *rbuf, void *wbuf)
5 {
6     unsigned long long *head_tail_flag = o_data + 2;
7     // 先頭が空白か改行かのチェック
8     head_tail_flag[0] = (i_data[0] != 0x20) && (i_data[0] != 0x0A);
9     word_num -= 1-head_tail_flag[0];

```

```

10|     for (; i < length; i++) {
11|         if (i_data[i] == 0x20) { // 空白
12|             word_flag = 1;
13|         } else if (i_data[i] == 0x0A) { // 改行
14|             line_num += 1;
15|             word_flag = 1;
16|         } else {
17|             word_num += word_flag;
18|             word_flag = 0;
19|         }
20|     }
21|     word_num += word_flag;
22|     // ファイルの末尾が空白か改行かのチェック
23|     head_tail_flag[1] = (i_data[i-1] != 0x20) && (i_data[i-1] != 0x0A);
24|     // Output Data の設定
25|     o_data[0] = (unsigned long long)word_num;
26|     o_data[1] = (unsigned long long)line_num;
27|     return 0;
28| }

```

ソースコード 4.3 は、それぞれの Task で出力された結果をまとめる処理がまとめられている。Task それぞれの単語数と行数を集計し、分割された部分に関して整合性をとり単語数を調整する。単語数、行数がそれぞれ決定されたのちに結果が Print される。

ソースコード 4.3: Word Count の Print Task

```

1| #define STATUS_NUM 2
2|
3| static int
4| run_print(SchedTask *s, void *rbuf, void *wbuf)
5| {
6|     long status_num = STATUS_NUM;
7|     unsigned long long word_data[STATUS_NUM];
8|     int flag_cal_sum = 0;
9|     for (int i = 0; i < STATUS_NUM; i++) {
10|         word_data[i] = 0;
11|     }
12|     int out_size = w->division_out_size / sizeof(unsigned long long);
13|     // 結果の整合性を取りながら、行数と単語数をカウントする。
14|     for (int i = 0; i < out_task_num; i++) {
15|         word_data[0] += idata[i*out_size+0]; // 単語数の集計
16|         word_data[1] += idata[i*out_size+1]; // 行数の集計
17|         // 前のファイルの末尾と次のファイルの先頭を比較し単語数の集計を微調整
18|         unsigned long long *head_tail_flag = &idata[i*out_size+2];
19|         if((i!=out_task_num-1)&&
20|            (head_tail_flag[1] == 1) && (head_tail_flag[4] == 0)) {
21|             flag_cal_sum++;
22|         }
23|     }
24|     word_data[0] += flag_cal_sum;
25| }

```

4.3 Boyer-Moore String Search

読み込んだテキストファイルに対してある特定の文字列検索を行う例題として、Boyer-Moore String Search が挙げられる。Boyer-Moore String Search は 1977 年に Robert S. Boyer と J Strother Moore が開発した効率的なアルゴリズムである。[1]

以下、テキストファイルに含まれている文字列を `text`、検索する文字列を `pattern` と定義する。

原始的な検索アルゴリズムとして力任せ法が挙げられる。力任せ法は `text` と `pattern` を先頭から比較していき、`pattern` と一致しなければ `pattern` を 1 文字分だけ後ろにずらして再度比較をしていくアルゴリズムである。`text` の先頭から `pattern` の先頭を比較していき、文字の不一致が起きた場合は `pattern` を後ろに 1 つだけずらして再比較を行う。(図 4.4)

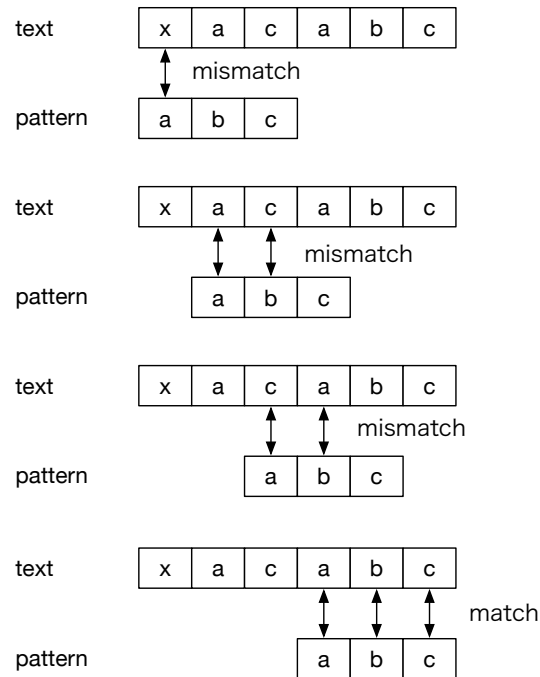


図 4.4: 力まかせ法

このアルゴリズムは実装が容易であるが、`text` と `pattern` の文字数が大きくなるにつれて、比較回数も膨大になる恐れがある。`text` の長さを n 、`pattern` の長さを m とすると、力任せ法の最悪計算時間は $O(nm)$ となる。

力任せ法の比較回数を改善したアルゴリズムが Boyer-Moore String Search である。力任せ法との大きな違いとして、`text` と `pattern` を先頭から比較するのではなく、`pattern`

の末尾から比較していくことである。さらに不一致が起こった場合は、その不一致が起こった text の文字で再度比較する場所が決まる。

図 4.5 は、text と pattern の末尾が不一致を起こして、そのときの text が pattern に含まれていない場合である。不一致した text の文字が pattern に含まれていない場合は、pattern を比較する場所に match することはないので、pattern の長さ分だけ後ろにずらすことができる。

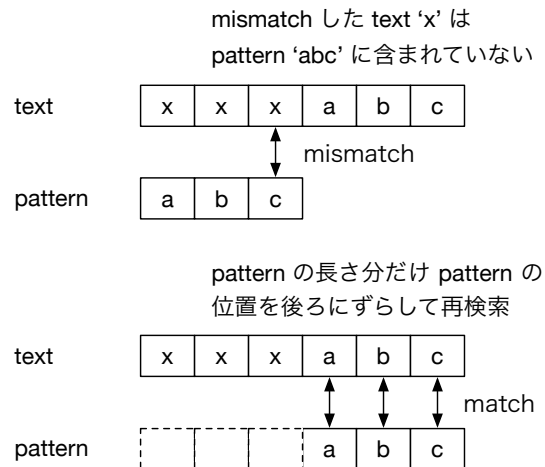


図 4.5: pattern に含まれていない文字で不一致になった場合

図 4.6 は不一致が起こったときの text の文字が pattern に含まれている場合である。この場合は pattern を後ろに 2 つずらすと text と pattern が一致する。

不一致したときの text の文字が pattern に含まれていた場合の後ろにずらす量は、pattern の長さから含まれていた文字が pattern の何文字目に含まれているかを引いた値となる。この場合、pattern の文字列の長さは 3 で text で不一致を起こした文字 'a' が pattern の 1 文字目に含まれているので、2 文字分だけ後ろにずらすことができる。

図 4.7 は不一致が起こったときの text の文字が pattern に含まれ、その不一致文字が pattern に複数含まれている場合である。

pattern の長さは 4 で、不一致を起こした時の text の文字 'a' は pattern の 1 番目と 3 番目に含まれている。pattern を後ろにずらす量は 1 か 3 となる。ずらす量を 3 にすると、pattern が含まれている text を見逃す可能性があるため、この場合 'a' で不一致したときは最小の値 1 をとる。

pattern と text と不一致時の処理をまとめると、

- pattern に含まれていない文字で不一致した場合は、pattern の長さだけ後ろにずらす。

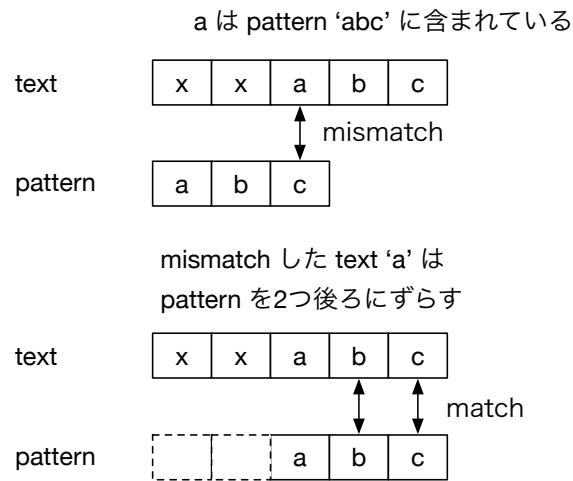


図 4.6: pattern に含まれている文字で不一致になった場合

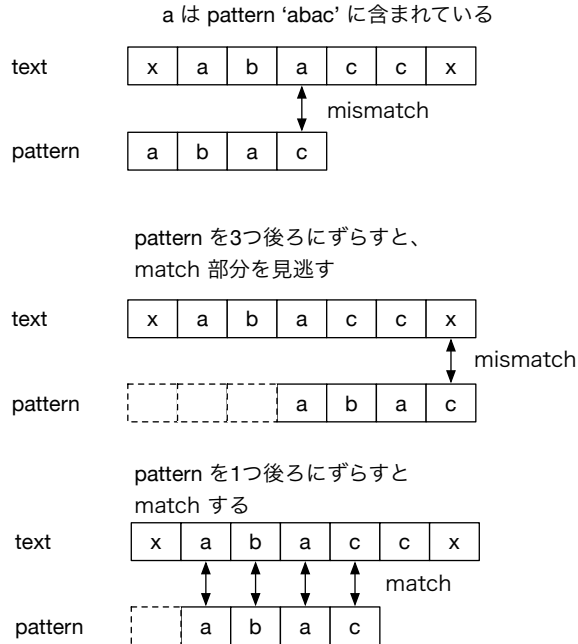


図 4.7: pattern に同じ文字が複数入り、その文字で不一致になった場合

- pattern に含まれている文字の場合は、pattern の長さから pattern に含まれている文字の位置を引いた数だけ後ろにずらす。
- pattern に含まれている文字でその文字が pattern に複数含まれている場合は後ろにずらす量も複数現れる。その中の最小の値だけ後ろにずらす。

不一致が起きたときの文字によって数文字分だけ後ろにずらす skip table を、文字列探索する前に生成する必要がある。

text 分割時に、分割部分で pattern が含まれる場合が存在する。その場合は、本来の読み込み部分の text の長さ L に加えて、pattern の長さ s から 1 引いた数だけ多く読みこむように設計することで、正しく結果を算出することができる。(図 4.8)

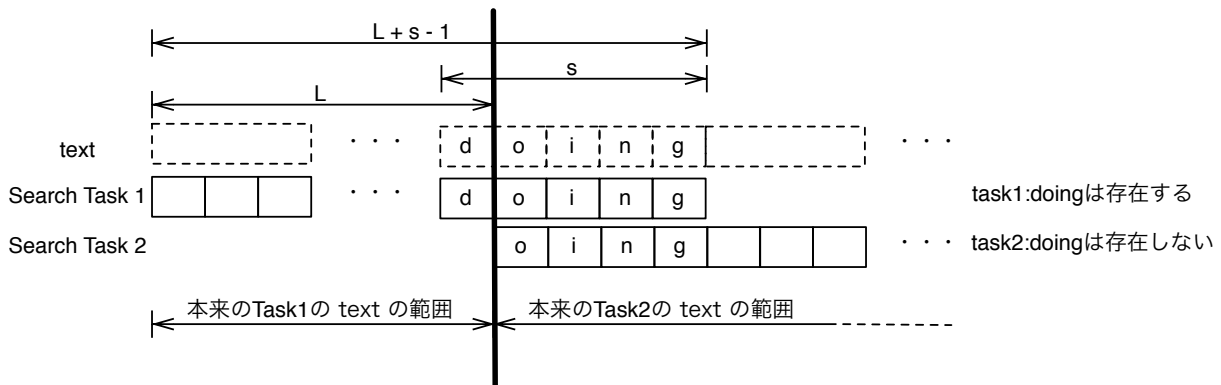


図 4.8: 分割周りの処理

Boyer-Moore String Search は文字列処理の Task を生成する前に、Skip Table を生成している。そして Boyer-Moore String Search に必要な情報を bm という構造体にまとめ、この構造体をそれぞれの Task に送信している。(ソースコード 4.4)

ソースコード 4.4: Boyer-Moore String Search の TMmain

```

1 typedef struct bm {
2     int* skip_table;
3     unsigned char *search_word;
4     int search_word_len;
5 } BM, *BMPtr;
6
7 //Boyer Moore String Search で使用する skip table を作成
8 static void
9 create_BMskiptable(BMPtr bm)
10 {
11     bm->skip_table = (int*)malloc(sizeof(int)*256);
12     for (int i = 0; i < 256; ++i) {
13         bm->skip_table[i] = bm->search_word_len;
14     }

```



```

15|
16|   for (int j = 0; j < bm->search_word_len - 1; ++j) {
17|       bm->skip_table[bm->search_word[j]] = bm->search_word_len - j - 1;
18|   }
19| }
20|
21| int
22| TMmain(TaskManager *manager, int argc, char *argv[])
23| {
24|     BMPtr bm = new BM;
25|     bm->skip_table = (int*)malloc(sizeof(int)*256);
26|     for (int i = 1; i < argc; i++) {
27|         if (strcmp(argv[i], "-sw") == 0) {
28|             bm->search_word = (unsigned char*)argv[i+1]; i++;
29|             bm->search_word_len = strlen((const char*)bm->search_word);
30|             create_BMskiptable(bm);
31|         }
32|     }
33|     fmp->w->global = (void*)bmp;
34|     fmp->overrap = bmp->search_word_len - 1;
35|     fmp->division_out_size = sizeof(unsigned long long);
36|     task_init();
37|     fmp->run_start(manager, filename);
38|     return 0;
39| }

```

ソースコード 4.5: Boyer-Moore String Search の記述

```

1| static int BM_method(unsigned char *text,int text_len,
2|                    unsigned char *pattern,int sw_len,int *skip)
3| {
4|     int i = sw_len - 1;
5|     int match_counter = 0;
6|
7|     while ( i < text_len){
8|         int j = sw_len - 1;
9|         // pattern の末尾から比較していく
10|        while (text[i] == pattern[j]){
11|            if (j == 0){
12|                match_counter++;
13|            }
14|            --i;
15|            --j;
16|        }
17|        i = i + max(skip[text[i]],sw_len - j);
18|    }
19|    return match_counter;
20| }
21|
22| static int
23| task_exec(SchedTask *s, void *rbuf, void *wbuf)
24| {
25|     unsigned char *i_data = (unsigned char *)s->get_input(0);
26|     int length = (int)s->get_inputSize(0);
27|     MapReduce *w = (MapReduce*)s->get_param(4);
28|     BMPtr bm = (BMPtr)w->global;
29|
30|     unsigned long long *o_data = (unsigned long long*)s->get_output(0);

```

```

31|
32|     o_data[0] = BM_method(i_data,length,bm->search_word,bm->search_word_len,bm->skip_table
33|         );
34|     return 0;

```

ソースコード 4.6: Boyer-Moore String Search の Print Task

```

1| static int
2| print_task(SchedTask *s, void *rbuf, void *wbuf)
3| {
4|     MapReduce *w = (MapReduce*)s->get_input(0);
5|     unsigned int idata_task_num = w->task_num;
6|     int match_counter = 0;
7|     for (int i = 0; i < idata_task_num; i++) {
8|         match_counter += idata[i];
9|     }
10|     return 0;
11| }

```

4.4 正規表現

正規表現は文字列のパターンを表現するための方法である。

BOSE という文字列をファイルから検索する場合を例にとる。BOSE という文字列は、そのファイルに Bose もしくは bose と記述されているかもしれない。もし、BOSE で検索すると小文字が含まれている Bose、bose は検索の対象外となってしまう、それら一つ一つを検索するのは手間が掛かってしまう。

このようなあるパターンで文字列を表現できる場合は、正規表現を利用すればこの問題は簡単に解決することができる。正規表現にはメタ文字と呼ばれる正規表現内での特殊記号があり、それらを利用することによって BOSE、Bose、bose の 3 つの文字列を一つの正規表現で表現することができる。(図 4.9)

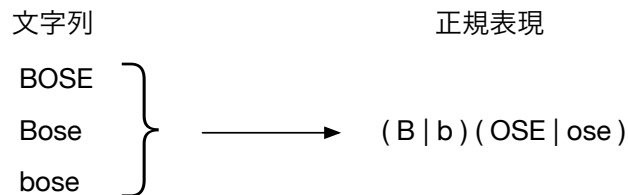


図 4.9: 3 つの表記ゆれの文字列を 1 つの正規表現にまとめる

本実装でサポートするメタ文字は、正規表現の基本三演算子 (接続、繰返し、選択)[2] に文字クラスとグループを加えている。(表 5.4)

AB	連続した文字 (接続)
A*	直前の文字の 0 回以上の繰返し
A B	A または B(選択)
[A-Z]	A-Z の範囲の任意の一文字 (文字クラス)
()	演算の優先度の明示 (グループ)

表 4.1: サポートしているメタ文字一覧

また、これらのメタ文字は数式の四則演算のように結合順位を持っている。それぞれのメタ文字の結合順位は表 4.2 のようになる。

結合順位	メタ文字
高	() (グループ化)
	[] (文字クラス)
	* 繰返し
	接続
低	選択

表 4.2: メタ文字の結合順位

例題として実装した正規表現マッチャのアルゴリズムは、

1. 与えられた正規表現を構文解析し、正規表現木に変換
2. 正規表現木への状態の割当
3. Subset Construction による状態の変換
4. 正規表現マッチャの並列処理の実装

となる。本項はそれぞれのアルゴリズムについて述べていく。

4.4.1 正規表現木の生成

まずはじめに、図 4.10 のように与えられた正規表現から正規表現木に変換する。正規表現木は二分木になるように生成していく。与えられた正規表現を頭から一文字ずつ読み

込み、読み込んだ文字やメタ文字と呼ばれる正規表現での特殊記号を元に木を構成していく。

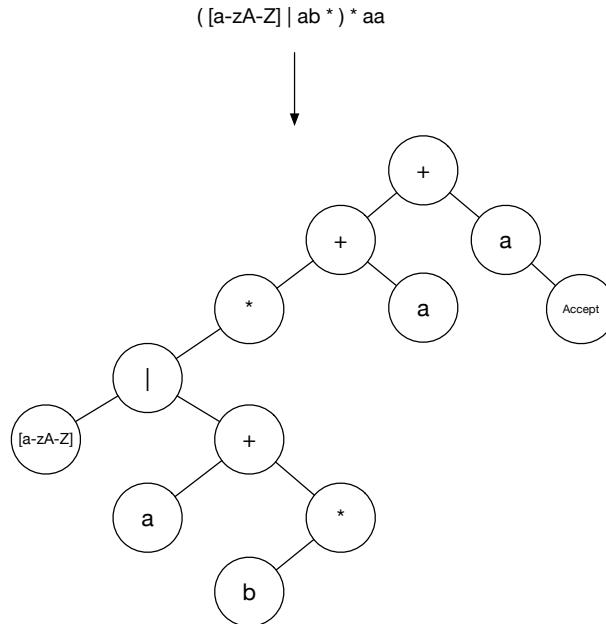


図 4.10: 正規表現から正規表現木への変換の例

正規表現木は与えられた正規表現を先頭から一文字ずつ読み込み、読み込んだ文字やメタ文字を一定のルールに従って生成していく。文字やメタ文字、文字クラスは正規表現木のノードとして表現され、メタ文字が現れた時に親子関係が決定される。

文字または文字クラスが読み込まれた場合はノードを生成する。それらが接続されたい場合は '+' ノードを親ノードとして、左ノードに前の文字、右ノードに後ろの文字が接続される。(図 4.11)

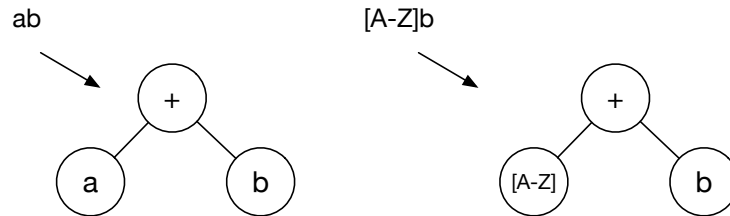


図 4.11: 文字の接続

また、文字列のように接続が連続した場合、読み込まれた順番に接続していく。接続済みの '+' ノードを左の子ノードとしてさらに '+' ノードで結合していく。(図 4.12)

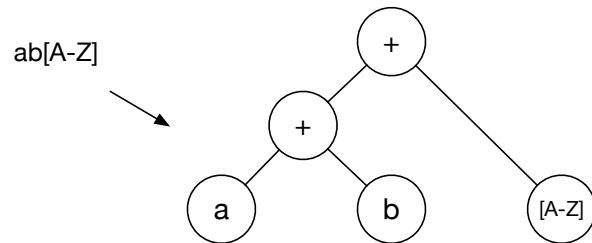


図 4.12: 文字列の接続

選択 ‘|’ が読み込まれた場合、親ノードを ‘|’ として木を構成していく。‘|’ の直前の文字、文字クラスまたは正規表現は左ノード、直後の文字、文字クラスまたは正規表現は右ノードとした木が構成される。‘|’ は直前と直後の正規表現の関係を表しているので、左右のノードに正規表現の要素を持ったノードとなる。(図 4.13)

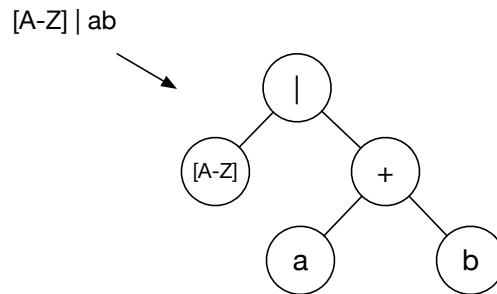


図 4.13: 選択

繰り返し ‘*’ が読み込まれた場合、‘*’ の直前の正規表現を左の子ノードとした木が生成される。また ‘*’ は、‘*’ の直前の文字や文字クラス、正規表現だけに結合するので、右の子ノードに何かしらのノードが生成されることはない。(図 4.14)

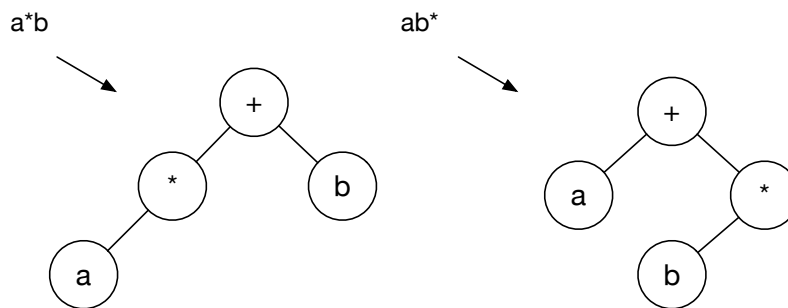


図 4.14: 繰り返し

グループ化 ‘()’ が読み込まれた場合、‘()’ 内をひとかたまりの正規表現として木を構成する。構成後さらに文字列が読み込まれれば、上記のルールにしたがって木が構成される。(図 4.15)

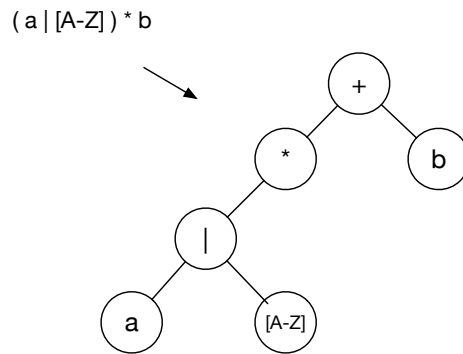


図 4.15: グループ

正規表現が接続した場合も文字の接続と同様に ‘+’ を親ノードとして接続していく。(図 4.16)

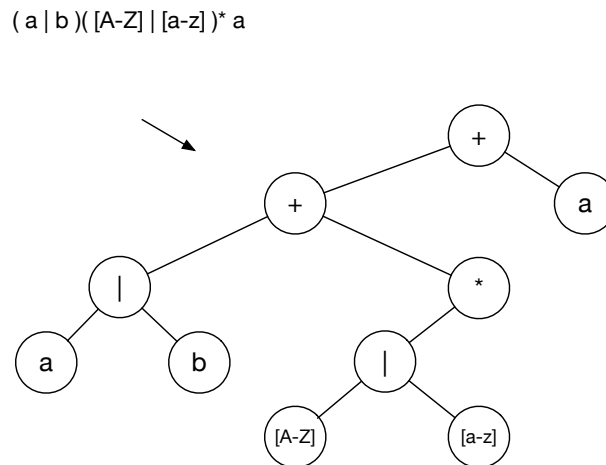


図 4.16: 正規表現の接続

これらのルールに則って正規表現木を構成し、それを元に DFA・NFA を生成していく。

4.4.2 正規表現木への状態の割当

次に正規表現木に状態を割り当てて、非決定性有限オートマトン (NFA) を生成する。正規表現木のノードに対して一定のルールに沿って状態を割り当てていく。

- 左子ノードが '*' でない '+' は新しい状態を作る
- '|' ノードの場合、左右の子ノードの先頭の状態は同じ状態を割り振る。
- '*' があれば、次の状態は '*' に接続されている木の先頭の状態と同じ。次の状態が受理状態なら先頭の状態と受理状態の組み合わせになる。

ノードに状態を割り振りながら次の状態の遷移先を設定することによって、オートマトンによる状態遷移を表現することができる。

$([a-zA-Z] \mid ab^*)^*aa$

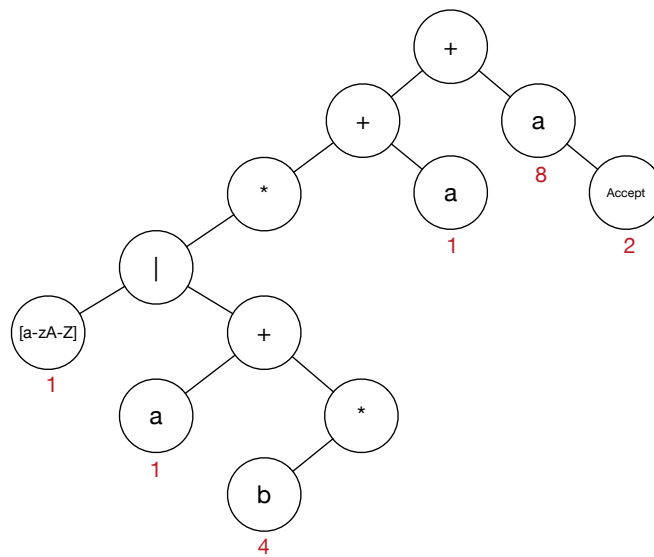


図 4.17: 与えられた正規表現木に状態を割り振る

1 入力に対して遷移先が複数存在している場合は、文字によって場合分けをする必要がある。図?? の状態 4 は $[a-z]$ が入力されると状態 4 に遷移し、 b が入力されると状態 2 に遷移する。このとき、 b が入力されると状態 2 か状態 4 のどちらかに遷移することになる。(図 4.18)

このとき、状態 2 と 4 を組み合わせて一つの状態を新しく作り、その状態に遷移させる。新しく作られる状態の数は状態の組み合わせなので、その状態の組み合わせの和を

NFA

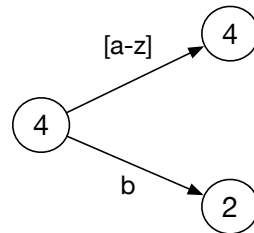


図 4.18: b が入力されるとどちらに遷移すればいいのかわからない

とっている。これより、状態 4 に a か [c-z] を入力すると状態 4 に遷移し、b が入力されると新しい状態 2,4 に遷移する。このような変換をすることによって、入力によって遷移先が一意に決定されるようになる。(図 4.19)

DFA

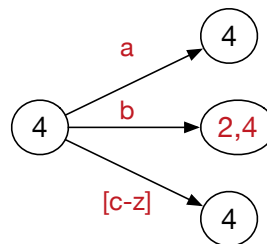


図 4.19: b が入力されたときは新しい状態を作り、新しい状態に遷移させる

4.4.3 Subset Construction による状態の変換

組み合わせられた状態からそれぞれの状態の場合分けをたどって、さらに別な組み合わせられた状態が生成される。新しい状態の組み合わせが出てこなくなるまでこれを繰り返す。これを Subset Construction と呼ぶ。ここでは状態の表現に BitVector を用いる。1つの Bit が正規表現に割り振られた Base 状態に対応する。組み合わせられた状態は、複数の Bit が立っていることにより表される。状態の組み合わせは、BitVector の論理和によって簡単に計算される。

ソースコード 4.7 は文字クラスの構造体である。正規表現木の各ノードそれぞれに、この構造体を持っている。文字クラスは二分木で構築されており、それぞれの二分木に文字

の範囲である Condition を持っている。その Condition の範囲内の文字の入力があれば、次の状態 nextState に遷移する。

ソースコード 4.7: 文字クラス (charClass) の構造体

```

1 typedef struct utf8Range {
2     unsigned long begin;
3     unsigned long end;
4 } RangeList , *RangeListPtr;
5
6 typedef struct condition {
7     RangeList range;
8     Word w;
9 } Condition, *ConditionList;
10
11 typedef struct charClass {
12     struct charClass *left;
13     struct charClass *right;
14     Condition cond;
15     int stateNum;
16     BitVector nextState;
17 } CharClass, *CharClassPtr;

```

図 4.20 に 2 つの状態があり、それぞれに入力と遷移先がある。これら 2 つの状態を一つの状態として表現し、新しく状態遷移図を生成する。もし、文字クラスの範囲が重なっている場合は、遷移先の状態を組み合わせる新しい状態を生成する。

ある文字クラスの範囲に別の文字クラスの範囲が重なり、さらにそれぞれの文字クラスの遷移先が異なる場合、図 4.21 のような 13 パターンの場合分けを考慮しなければならない。

4.4.4 正規表現マッチャの並列処理の実装

正規表現から正規表現木を生成し、その木に対して状態を割り振りを行ない、Subset Construction による新しい状態の生成が終わったあとにマッチングを行う。

今回の実装では、状態それぞれを Bit Pattern で表現している。Bit Pattern で状態を持つことで、Subset Construction による状態の組み合わせも Bit Pattern の和を取ること容易に表現できる。それらの状態は StateArray と呼ばれる配列で格納されており、状態遷移が行われるたびに StateArray を見に行く。あらかじめ配列の大きさのメモリを確保する必要があるが、Subset Construction による新しい状態生成も考慮しなければならない。正規表現木に状態を割り振った時に最大の状態の Bit は分かっているので、その状態の Bit の 2 倍だけ配列を確保しておけば、Bit Pattern が全て 1 までの状態を表現することができる。

StateArray を見れば、ある状態に対して入力をすると状態の遷移先をチェックすることができる。状態の遷移先は文字クラスごとの List 構造になっており、入力された文字をチェックしながらリストを辿って状態の遷移先を判断する。(図 4.22)

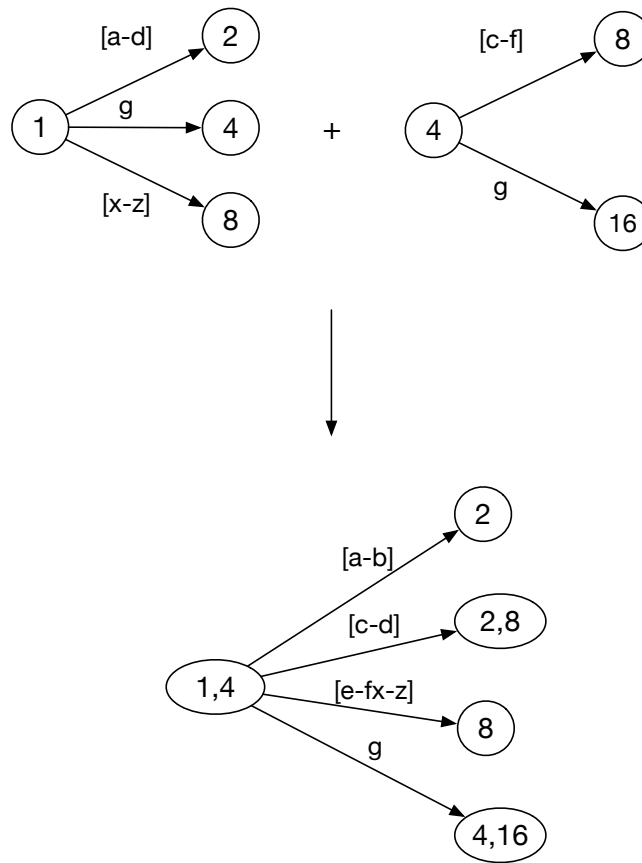


図 4.20: Subset Construction による新しい状態の生成

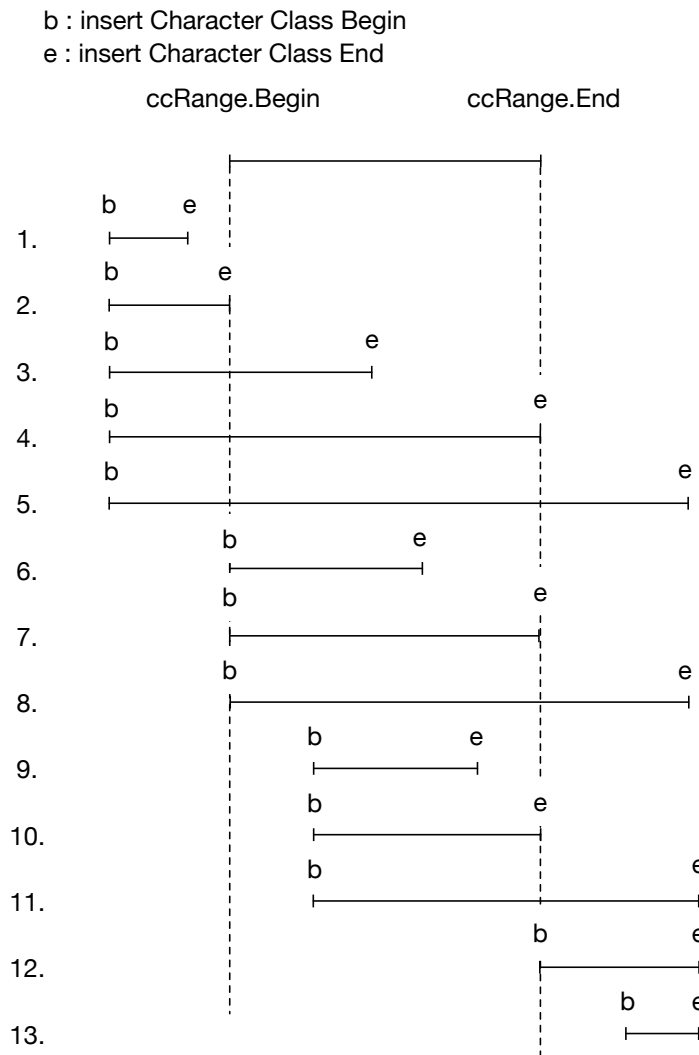


図 4.21: 複数の文字クラスを Merge するときの全パターン

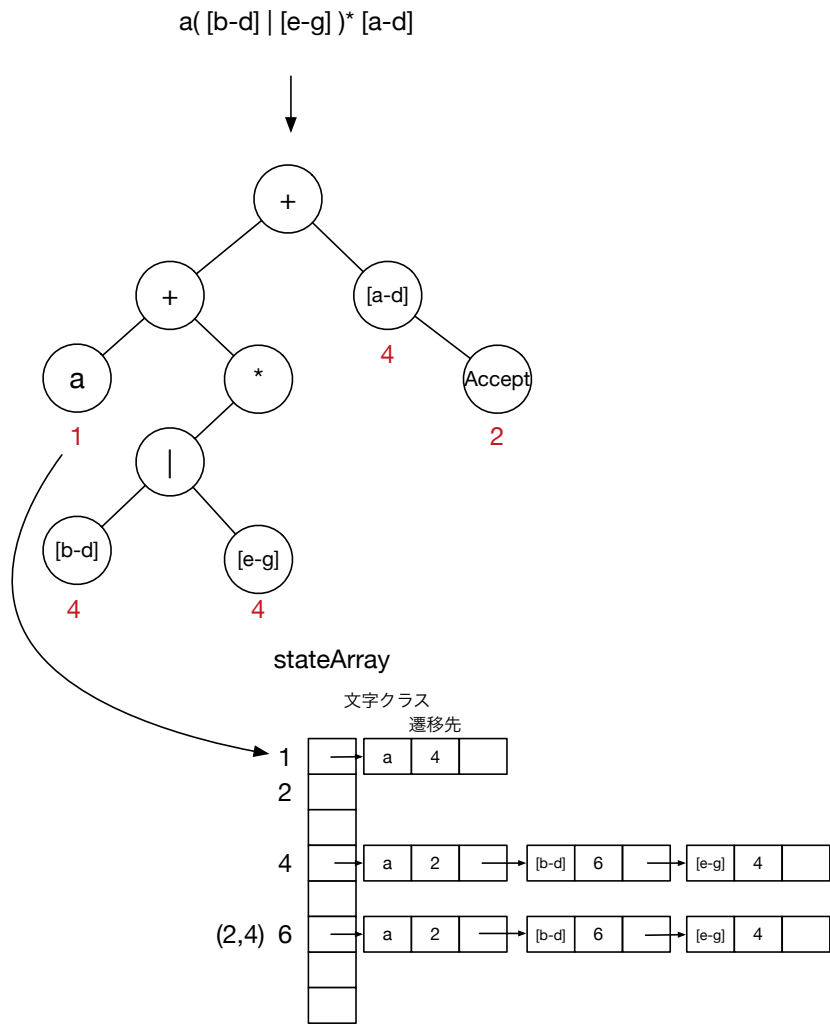


図 4.22: StateArray による状態遷移

ソースコード 4.8: ceriumGrep の TMmain

```

1 int
2 TMmain(TaskManager *manager, int argc, char *argv[])
3 {
4     char *filename = 0;
5     Search s = grep(argc,argv,true); // 正規表現木の生成、木の状態割り振りを行う
6     FileMapReduce *fmp = new FileMapReduce(manager,TASK_EXEC,
7         TASK_EXEC_DATA_PARALLEL,TASK_PRINT);
8     filename = fmp->init(argc, argv);
9     fmp->w->global = (void*)s.tg;
10    if (filename < 0) {
11        return -1;
12    }
13    fmp->division_out_size = sizeof(void*)*3; // *Result,*blockBegin,*blockEnd
14    task_init();
15    fmp->run_start(manager, filename);
16    return 0;
17 }

```

ソースコード 4.9: ceriumGrep のマッチング部分

```

1 TSValue blockSearch(TSValue tsv,Buffer buff,int task_spawned) {
2     tsv.current = tsv.tg->stateStart->tState;
3     tsv.blk->result = NULL;
4     ResultPtr result = NULL;
5     tsv.blk->resultEnd = &result;
6     tsv.blk->blockBegin = tsv.current;
7     addResult(tsv,true,buff.buff,buff.buffend); // entire buffer
8     tsv = tSearch(tsv); // Matching
9     tsv.blk->blockEnd = tsv.current;
10    if (tsv.blk->blockEnd->state->bitState.bitContainer != 1) {
11        if (tsv.matchBegin != tsv.buff.buffptr) {
12            // partial match case at block end.
13            addResult(tsv,true,tsv.matchBegin,tsv.matchEnd);
14        }
15    }
16    tsv.blk->result = result;
17    return tsv;
18 }
19
20 static int
21 blockedGrep(SchedTask *s, void *rbuf, void *wbuf)
22 {
23     TransitionGeneratorPtr tg = (TransitionGeneratorPtr)w->global;
24     Buffer buff;
25     buff.buff = buff.buffptr = i_data;
26     buff.buffend = buff.buff + length;
27     BlockOutput blk;
28     TSValue tsv = createTSValue(tg,buff);
29     tsv.blk = &blk;
30     tsv = blockSearch(tsv,buff,task_spawned);
31     o_data[0] = (unsigned long)tsv.blk->result;
32     o_data[1] = (unsigned long)tsv.blk->blockBegin->state; // never used now
33     o_data[2] = (unsigned long)tsv.blk->blockEnd->state;
34     return 0;
35 }

```

分割されたファイルに対して正規表現によるマッチングを行う。マッチングすると、マッ

チングの始まりのファイルの場所と終わりのファイルの場所を Result という構造体に格納する。マッチングの数だけこの構造体が生成され、これらは List 構造として結果をまとめていく。(ソースコード 4.10) もし、分割されたファイル

ソースコード 4.10: Result の構造体

```

1 typedef struct result {
2     unsigned char *begin;
3     unsigned char *end;
4     bool continued;
5     struct result *next;
6 } Result, *ResultPtr;

```

全ての分割されたファイルに対してマッチングが終了すると、Print Task にてまとめてマッチした部分を表示する。

正規表現をファイル分割して並列処理をする際、本来マッチングする文章がファイル分割によってマッチングしない場合がある。図 4.23 はその一例である。並列処理時、分割されたファイルに対してパターンマッチさせるので、分割された 1 つ目のファイルの末尾の abb、2 つ目のファイルの先頭に bbc はマッチングしない。本来分割される前はマッチングする文字列だが、この場合見逃してしまう。それを解決するために、正規表現にマッチングし始めたファイルの場所を覚えておく。そして、1 つ目のファイルの末尾が状態遷移の途中で終わっていた場合 (状態 1 でない場合) は、結果を集計する際に再度マッチングし始めた場所から正規表現をマッチングさせる。

このマッチングは結果を集計する Print Task にて実行されるので、single thread で処理される。そのため、分割された部分で状態遷移の途中で終わっている場合が多ければそれだけ集計時の処理が重くなる。

ソースコード 4.11: マッチした結果を Print する

```

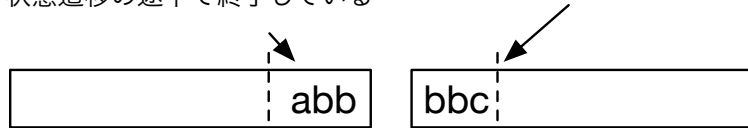
1 static
2 TSValue stateSkipOnce(TSValue tsv) {
3     if (tsv.matchEnd) {
4         addResult(tsv, false, tsv.matchBegin, tsv.matchEnd);
5     }
6     tsv.buff.buffend = tsv.buff.buffptr; // end search
7     return tsv;
8 }
9
10 static int
11 run_print(SchedTask *s, void *rbuf, void *wbuf)
12 {
13     ResultPtr prev = NULL;
14     for (int i = 0; i < out.task_num; i++) {
15         ResultPtr r = (ResultPtr)w->o_data[i*out_size+0];
16         // first reply contains block begin and block end
17         unsigned char *begin = r->begin;
18         unsigned char *end = r->end;
19         r = r->next;
20         if (r == NULL) {
21             prev = NULL;

```

正規表現 : ab^*c

abb しか読めない
ので、正規表現には
match しないが、
状態遷移の途中
で終了している

bbc しか読めない
ので、正規表現には
match しない



no match ←————→ no match

マッチする部分が
分割されている



状態遷移の途中で終了している場合
は、match し始めたときの先頭を
覚えて再度マッチするかどうか確認
する。

abbbbc

match

図 4.23: 分割された部分に正規表現がマッチングする場合の処理


```
22|         continue;
23|     }
24|     if (prev) {
25|         if (i >= out_task_num) break;
26|         // 最後のブロックでなく、前の prevBlockEnd が state 1 でない場合)
27|         StatePtr prevBlockEnd = (StatePtr)w->o_data[i*out_size-1];
28|         if (prevBlockEnd->bitState.bitContainer !=1) {
29|             // そこから最初の stateSkip までやり直し。マッチしたら表示。
30|             TransitionGeneratorPtr tg = (TransitionGeneratorPtr)w->global;
31|             tg->stateSkip = stateSkipOnce;
32|             Buffer buff;
33|             buff.buff = buff.buffptr = begin;
34|             buff.buffend = end;
35|             TSValue tsv = createTSValue(tg,buff);
36|             BlockOutput blk;
37|             tsv.blk = &blk;
38|             tsv.current = prevBlockEnd->tState;
39|             tsv.blk->result = NULL;
40|             ResultPtr result = NULL;
41|             tsv.blk->resultEnd = &result;
42|             tsv.matchBegin = prev->begin;
43|             tsv.matchEnd = prev->end;
44|             tsv = tSearch(tsv);
45|             if (result) {
46|                 resultPrint(result,"Print");
47|             }
48|         }
49|     }
50|     prev = resultPrint(r,"Print");
51| }
52| return 0;
53| }
```

第5章 ベンチマーク

本項で行なった実験の環境は以下の通りである。

- Mac OS X 10.10.5
- 2*2.66 GHz 6-Core Intel Xeon
- Memory 16GB 1333MHz DDR3
- 1TB HDD

Cerium で実装した Word Count と Mac の `wc` の比較と、実装した正規表現と Mac の `egrep` の比較を行なった。また、それぞれの結果に実装した並列処理向け I/O の結果も含む。

5.1 Word Count

ファイルの大きさは約 500MByte で、このファイルには約 650 万行、約 8300 万単語が含まれている。

表 5.1 は、ファイル読み込みを含めた Word Count の結果である。Mac の `wc` ではこのファイルを処理するのに 10.59 秒かかる。それに対して、Cerium Word Count は mmap Blocked Read 全ての状況で Mac の `wc` よりも速いことを示している。Cerium Word Count 12 CPU のとき、7.83 秒で処理をしており、Mac の `wc` の 1.4 倍ほど速くなっている。

mmap は読み込みを OS が制御しており、書き手が制御できない。また Word Count が走る際ファイルアクセスはランダムアクセスとなる。mmap はランダムアクセスを想定していなくて結果がばらつきが起きていると考えられる。Blocked Read では読み込みをプログラムの書き手が制御しており、ファイルの読み込みもファイルの先頭から順次読み込みを行なっている。そのため、読み込みを含めた結果にばらつきが起りにくくなっていると予想される。

ファイルサイズ：500MB(単語数約 8500 万)
 ファイル読み込みも含む

CPU Num / 実行方式	Mac(wc)	mmap	Blocked Read
1	10.59	9.96	9.33
4	—	8.63	8.52
8	—	10.35	8.04
12	—	9.26	7.82

表 5.1: ファイル読み込みを含む Word Count

表 5.2 はファイル読み込みを含まない Word Count の結果である。

Mac の wc ではこのファイル进行处理するのに 4.08 秒かかる。それに対して、Cerium Word Count は 1 CPU で 3.70 秒、12 CPU だと 0.40 秒で処理できる。

1 CPU で動作させると Mac の wc よりも 1.1 倍ほど速くなり、12 CPU で動作させると wc よりも 10.2 倍ほど速くなった。1 CPU と 12 CPU で比較すると、9.25 倍ほど速くなった。

ファイルを読み込んだ結果と比較すると、ファイルを読み込まないで実行したほうが 6,7 秒ほど速くなる。これよりファイルを読み込んだ文字列処理の場合、処理時間の 60%から 90% はファイルの読み込みであることがわかる。

ファイルサイズ：500MB(単語数約 8500 万)
 ファイル読み込みは含まない

実行方式	実行速度(秒)
Mac(wc)	4.08
Cerium Word Count(CPU 1)	3.70
Cerium Word Count(CPU 4)	1.00
Cerium Word Count(CPU 8)	0.52
Cerium Word Count(CPU 12)	0.40

表 5.2: ファイル読み込み無しの Word Count

5.2 Boyer-Moore String Search

ファイルの大きさは約 500MByte で、このファイルには、約 8300 万単語が含まれている。このファイルに 'Pakistan' という文字列がいくつマッチするかカウントしている。(マッチ数 約 400 万)

表 5.3 はファイル読み込みを含まないで計測している。力任せ法と比較すると、Boyer-Moore String Search が最大 63 % ほど速くなる。

ファイルサイズ : 500MB(単語数約 8500 万)

検索文字列 : Pakistan

マッチ数約 400 万

ファイル読み込みは含まない

CPU 数	力任せ法	Boyer-Moore String Search
1	3.17	1.70
4	0.87	0.49
8	0.47	0.27
12	0.33	0.21

表 5.3: 力任せ法と Boyer-Moore String Search の比較

5.3 正規表現

当実験では、Mac の egrep、C で実装した逐次に DFA の状態遷移と照らし合わせマッチングをする single thread grep、Cerium で並列処理をする CeriumGrep を比較している。

表 5.4 は 500MB(単語数約 8500 万) のファイルに対して正規表現 '[A-Z][A-Za-z0-9]*s' をマッチングした結果である。これはファイル読み込みを含めた結果と読み込みを含まない結果の比較である。

single thread grep や CeriumGrep は繰り返し実行をすると実行速度が短くなる。これは、読み込んだファイルがキャッシュに残っており、ファイル読み込みが省略されるためである。同じファイルに対して違う正規表現でマッチングさせたとしてもファイル読み込みが省略されるため、高速に結果が返ってくる。

しかし egrep は繰り返し実行しても同じ速度で実行されるため、毎回ファイルを読み込む。そのため、同じファイルに違う正規表現をマッチさせようとする毎回は読み込みが行われる。

正規表現 '[A-Z][A-Za-z0-9]*s'
 ファイルサイズ：500MB(単語数約 8500 万)

実行方式	ファイル読み込み有	ファイル読み込み無
single thread grep	21.17	16.15
CeriumGrep(CPU 12) mmap	18.00	5.12
CeriumGrep(CPU 12) bread	15.76	5.18
egrep	59.51	59.51

表 5.4: ファイル読み込み有りと無しを変化させた各 grep の結果

表 5.5 は正規表現 '[A-Z][A-Za-z0-9]*s' を 500MB(単語数約 8500 万)、1GB(単語数約 1.7 億語) のファイルに対してマッチングを行なった。

ファイルサイズに比例して処理時間も長くなっていく。

正規表現 '[A-Z][A-Za-z0-9]*s'
 ファイルサイズ：500MB(単語数約 8500 万)
 ファイル読み込みを含む

実行方式/File Size(Match Num)	50MB(54 万)	100MB(107 万)	500MB(536 万)	1GB(1072 万)
single thread grep	4.51	9.42	20.62	40.10
CeriumGrep(CPU 12) mmap	8.97	10.79	18.00	29.16
CeriumGrep(CPU 12) bread	7.75	10.49	15.76	26.83
egrep	6.42	12.80	59.51	119.23

表 5.5: ファイルサイズを変化させた各 grep の結果

表 5.6 は a と b が多く含まれている約 500MB(単語数約 2400 万) のファイルに対して、正規表現の状態数を変化させてみた。これは読み込みを含んでいる結果で、CeriumGrep のファイル読み込みは Blocked Read、CPU 数 12 にて実行した。

この例題で使用した正規表現は '(a |b)' を '*a' の直後に付け加えていくと、それだけ状態数が指数関数的に増えていく。CeriumGrep は状態数が指数関数的に増加しても 1 秒ほど増加するが、egrep では 5,6 秒ほど増加していく。

ファイルサイズ：500MB(単語数約 2400 万)
 状態数の内訳：状態割当時の状態数/subset 後の状態数
 ファイル読み込みを含む

正規表現	状態数	マッチ数	CeriumGrep (CPU 12) bread	egrep
'(a b)*a(a b)(a b)z'	5/12	約 10 万	26.58	70.11
'(a b)*a(a b)(a b)(a b)z'	6/21	約 8 万	27.89	76.78
'(a b)*a(a b)(a b)(a b)(a b)z'	7/38	約 4 万	28.86	81.88
'(a b)*a(a b)(a b)(a b)(a b)(a b)z'	8/71	約 2 万	29.15	86.93

表 5.6: 正規表現の状態数を増やした Grep の結果

表 5.7 ab の文字列がならんでいるところに (W |w)ord の正規表現 a と b が多く含まれている約 500MB(単語数約 2300 万) のファイルに対して、全くマッチしない正規表現を与えてマッチングさせてみた。

マッチングしない場合でも egrep と比較して CeriumGrep bread のほうが 30 % ほど速くなる。

ファイルサイズ：500MB(単語数約 2400 万)
 正規表現：(W |w)ord
 ファイル読み込みを含む

実行方式	time (s)
single thread grep	27.13
CeriumGrep(CPU 12) mmap	21.58
CeriumGrep(CPU 12) bread	19.99
egrep	28.33

表 5.7: 全くマッチングしないパターンを grep した結果

第6章 結論

本研究室で開発している Cerium を利用して文字列処理の並列処理に関する研究を行なった。並列処理時のファイルの読み込みについて改良を行なった結果、最大 13%速くなる

本研究では文字列処理の並列処理の例題として、WordCount、Boyer-Moore String Search、正規表現を実装した。それぞれの例題によって結果の整合性を取る必要があるが、どのように整合性を取るかは問題によって考慮する必要がある。

本研究で実装した正規表現は、正規表現から正規表現木を生成し、その正規表現木に状態を割り振りながら NFA や DFA を生成する。もし NFA が存在した場合は Subset Construction で DFA に変換する。DFA に変換後、その DFA を元に与えられたファイルに対してマッチングさせる。その結果、ファイル読み込みを含め egrep と比較して最大 66 % 速度がでる。

6.1 今後の課題

これまでの正規表現は一文字ずつ参照して状態を割り振っていった。この状態割り振りの問題として文字列の長さの分だけ状態ができてしまう。状態が長くなればなるほど、ファイルと正規表現のマッチング時の状態遷移数もそれだけ多くなってしまふ。状態遷移数が多くなると、それだけ状態と入力を見て次の状態に遷移するという動作を何度も繰り返すことになってしまうので、処理的にも重くなってしまふ。同じ正規表現でも状態を少なくすればそのような繰り返し処理も減っていくので、状態数を減らせばマッチングするまでの処理を軽減することができる。状態数を減らす工夫として、文字列を一つの状態として見ることによって状態数を減らす。

図 6.1 は、‘word’ という文字列の正規表現の正規表現木、DFA 及び状態遷移テーブルである。一文字ずつそれぞれに状態を割り振った場合、状態数 5 のオートマトンが構成される。これを一つの文字列に対して状態を割り振った場合、状態数 2 のオートマトンが構成され、状態数を削減することができる。

実際にファイルに対して文字列を検索するときは、Boyer-Moore String Search を採用することにより、さらに高速になると期待される。

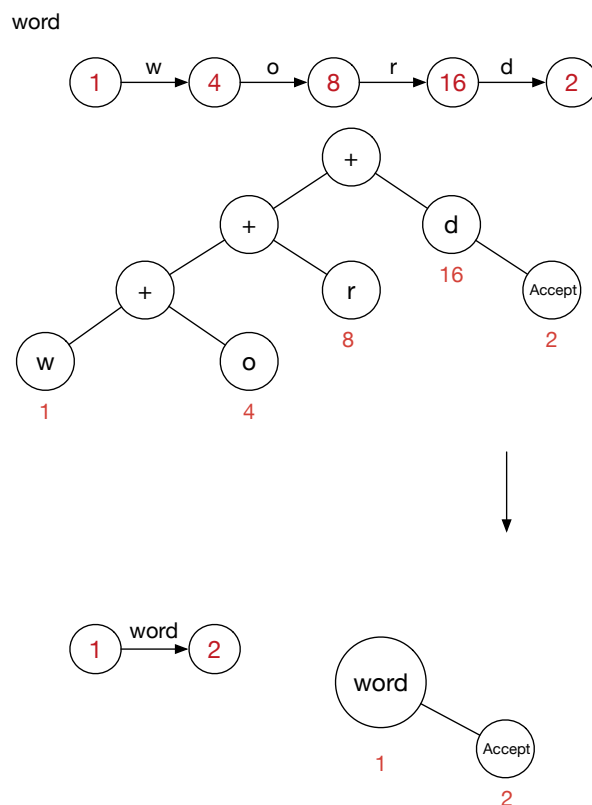


図 6.1: 文字単位の状態割り振りを文字列単位での状態割り振りに変更

参考文献

- [1] R.S. Boyer. J.S.Moore. *A Fast String Searching Algorithm*, 1977.
- [2] 新屋 良磨, 鈴木 勇介, 高田 謙. 正規表現技術入門 (技術論評社), 2015.
- [3] 金城裕. 並列プログラミングフレームワーク cerium の改良. 琉球大学工学部情報工学科平成 24 年度学位論文 (修士), March 2012.
- [4] 渡真利勇飛. マルチプラットフォーム対応並列プログラミングフレームワーク. 琉球大学大学院理工学研究科情報工学専攻平成 26 年度学位論文 (修士), 2013.
- [5] 新屋 良磨 河野真治. 動的なコード生成を用いた正規表現マッチャの実装. 第 52 回プログラミング・シンポジウム, January 2011.
- [6] Michael Sipser ・阿部 正幸・植田 広樹・藤岡 淳・渡辺 治著・太田 和夫・田中圭介監訳. 計算理論の基礎 [原著第 2 版](共立出版), May 2008.