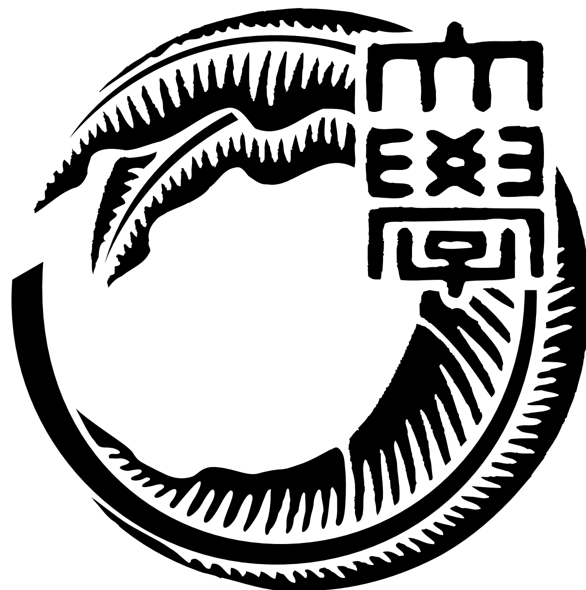
平成25年度 卒業論文

Ceriumによる 並列処理向けI/Oの設計と実装



琉球大学工学部情報工学科

085726C 古波倉 正隆

指導教員 河野 真治

目 次

第1章　はじめに **1**

1.1 研究背景 1

1.2 研究目的 1

第2章　Cerium **2**

2.1 Cerium Task Manager の概要 2

2.2 Cerium Task Manager の利用方法 3

第3章　例題 **5**

3.1 ファイルの読み込みに関する例題 5

3.2 ファイルに対して処理を行う例題 7

3.2.1 力任せ法 7

3.2.2 Boyer-Moore String Search Algorithm 8

3.2.3 ファイル分割時の処理の注意点 15

第4章　並列処理向け I/O の設計と実装 **18**

4.1 map reduce の設計 18

4.2 mmap での実装の問題点 19

4.3 Bloked Read の設計と実装 20

4.4 I/O 専用 thread の実装 22

第5章　ベンチマーク **26**

5.1 実験環境 26

5.2 結果 26

5.3 考察 27

第6章　結論 **28**

6.1 まとめ 28

6.2 今後の課題 28

6.2.1 実メモリ以上のファイルの取り扱い 28

6.2.2 Blocked Read で読み込んだファイルがキャッシュに残らない 29

6.2.3 Blocked Read を Cerium API 化 29

図 目 次

2.1 Cerium Task Manager 2

3.1 力まかせ法 7
3.2 pattern に含まれていない文字で不一致になった場合 8
3.3 pattern に含まれている文字で不一致になった場合 9
3.4 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 10
3.5 Boyer-Moore Search String 11
3.6 BMsearch skip table 12
3.7 skip table の 生成時の様子 13
3.8 IO を含む Task 15
3.9 分割周りの処理・失敗時 (例:doing の検索) 16
3.10 分割周りの処理・成功時 (例:doing の検索) 16

4.1 map reduce image 18
4.2 mmap image 20
4.3 Blocked Read image 21
4.4 Blocked Read image 22
4.5 SPE_ANY での実装時 23
4.6 IO_0 の追加 23
4.7 Blocked Read Task を IO_0 での実装時 24

表 目 次

2.1 Task 生成における API 3
2.2 Task 側で使用する API 4

3.1 pread 関数の概要 5
3.2 file read の実行結果 6
3.3 文字列検索アルゴリズムの比較 17

4.1 mmap 関数の概要 19
4.2 pread 関数の概要 20

5.1 実行結果 26

第1章　はじめに

1.1　研究背景

近年、CPU 1 コア当たりのクロック数が頭打ちとなっているので、シングルコアでの処理能力はほとんど上がっていない。それを解決した結果、シングルコアからマルチコアへの移行によって CPU 性能が向上している。しかし、マルチコア CPU を最大限に活かすためには、プログラムの並列度を向上させなければならない。そこで当研究室では Cerium Library を提供することによって並列プログラミングを容易にしている。

1.2　研究目的

先行研究による Task の並列化によって、プログラム全体の処理速度は飛躍的に向上しているが [1] 、ファイル読み込み等の I/O と Task が並列で動作するようには実装されていない。ファイル読み込みと Task を並列化させることにより、さらなる処理速度の向上が見込まれる。I/O と Task が並列に動作し、高速かつ容易に記述できるような API を Cerium Library が提供することにより、様々な人が容易に並列プログラミングが記述できるようになるであろうと考えている。

本研究では、 I/O と Task の並列化の設計・実装によって既存の正規表現の処理速度、処理効率を上げることを目指す。

第2章　Cerium

2.1　Cerium Task Manager の概要

Cerium Task Manager は並列プログラミングフレームワークであり、内部では C や C++ で実装されている。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 及び依存関係を設定する。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。Cerium Task Manager は PlayStation 3/Cell、Mac OS X 及び Linux 上で利用することが可能で、近年では GPU へ の利用も可能となった。

図 2.1 では、 User が Task を生成して、CPU や GPU の各デバイスに Task が割り振られる様子を表している。User が設定を行った Task は Task Manager にて生成される。その生成した Task に HTask にて Input Data 、Output Data や依存関係などを設定して Task の集合体である TaskList に Set する。そして TaskList を各デバイスに割り振って、各 Scheduler に管理をさせたあとにそれぞれの Task を起動する。

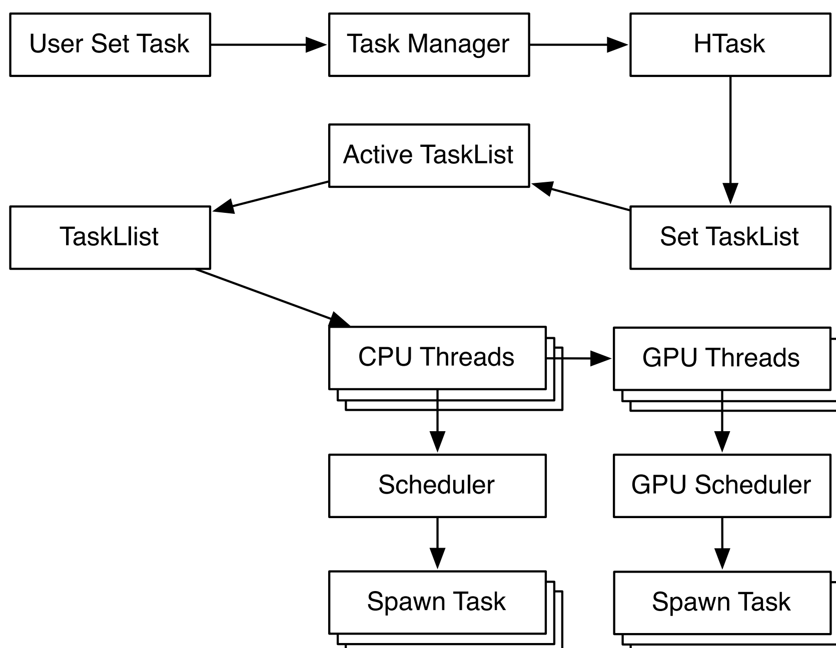


図 2.1: Cerium Task Manager

2.2 Cerium Task Manager の利用方法

input Data で格納して 2 つの数を乗算し、output data に格納する multiply という例題がある。その例題の Task 生成部分を以下に示す。

```
multi_init(TaskManager *manager)
{
    float *A, *B, *C;
    HTaskPtr multiply = manager->create_task(MULTIPLY_TASK);
    multiply->set_cpu(SPE_ANY);
    multiply->set_inData(0, (memaddr)A, sizeof(float)*length);
    multiply->set_inData(1, (memaddr)B, sizeof(float)*length);
    multiply->set_outData(0, (memaddr)C, sizeof(float)*length);
    multiply->set_param(0,(long)length);
    multiply->spawn();
}
```

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

Task の記述は以下のようになる。

```
static int
run(SchedTask *s,void *rbuf, void *wbuf)
{
    float *A, *B, *C;
    A = (float*)s->get_input(rbuf,0);
    B = (float*)s->get_input(rbuf,1);
    C = (float*)s->get_output(wbuf,0);
    long  length=(long)s->get_param(0);
    for (int i=0;i<length;i++) {
        C[i]=A[i]*B[i];
    }
    return 0;
}
```

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

第3章　例題

3.1　ファイルの読み込みに関する例題

テキストファイルをある一定のサイズに分割して読み込むプログラムである。このプログラムでは、pread という関数で実装した。pread 関数は UNIX 標準に関するヘッダファイル、unistd.h に含まれている関数である。(表 4.3) 読み込んだテキストファイルはバッファに格納されるが、その格納先は TaskManager の API でメモリを確保する。

ssize_t pread(int fd, void *buf, size_t nbyte, off_t offset);

int fd	読み込むファイルディスクリプタ
void *buf	予め用意したバッファへの書き込み
size_t nbyte	読み込むサイズ
off_t offset	ファイルの先頭からのオフセット

表 3.1: pread 関数の概要

この例題の Task 生成部分を以下に示す。

```
HTaskPtr read = manager->create_task(Read_task);
read->set_cpu(SPE_ANY);

read->set_param(0,(long)task_number);
read->set_param(1,(long)division_size);
if(read_left_size <= division_size){
    read->set_param(2,(long)left_size);
}else{
    read->set_param(2,(long)division_size);
}
read->set_param(3,(long)fd);

read->set_outData(0,read_text + task_number*division_size,
                                              division_size);
read->spawn();
```

```
read_left_size -= division_size;
task_number++;
```

read という Task を宣言し、Task に CPU Type、パラメータとして生成した Task 番号の task_number 1 つの Task の読み込み量 division_size、読み込むファイルのファイルディスクリプタ fd を設定する。読み込んだデータの格納先を set_outData にて設定を行い、そして spawn にて Task を生成する。

Task が生成されると、division_size 分の読み込みを行ったということで、残りの読み込み量 read_left_size から division_size を引き、そして task_number を増加させる。この Task 生成部分は ファイルサイズ を division_size で割った数だけ生成され、もし余りがあれば更に 1 加えた数になる。その数が Read Task の数となり、その数だけループ処理を行う。中盤にある if 文は、最後の Read Task かどうかで実際に読み込む量が決定される。

read Task の記述を以下に示す。

```
static int
read_task(SchedTask *s, void *rbuf, void *wbuf)
{
    long task_number = (long)s->get_param(0);
    long division_size = (long)s->get_param(1);
    long read_size = (long)s->get_param(2);
    long fd = (long)s->get_param(3);

    char *read_text = (char*)s->get_output(wbuf,0);

    pread(fd, read_text, (long)read_size , division_size*task_number);
    return 0;
}
```

生成時に設定したデータ群を受け取り、それらのデータを pread の引数に渡す。読み込まれたファイルは read_text に格納される。

ハードディスクに保存されている 10GB のテキストファイルを分割して読み込み終わるまでの時間を表 3.1 に示す。分割サイズとは、1 回の読み込み量である。

分割サイズ	読み込み速度 (s)
16KB	391.7
16MB	123.6

表 3.2: file read の実行結果

分割サイズを大きくすると、pread の呼ばれる回数が少なくなるので読み込むことが速くなる。

3.2 ファイルに対して処理を行う例題

読み込んだテキストファイルに対して文字列検索を行う例題として、力任せ法と Boyer-Moore String Search を実装した。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。なお、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。

3.2.1 力任せ法

力任せ法 (総当り法とも呼ばれる) 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 を右に 1 つだけずらして、再度 text と pattern の先頭を比較していく。(図 3.1)

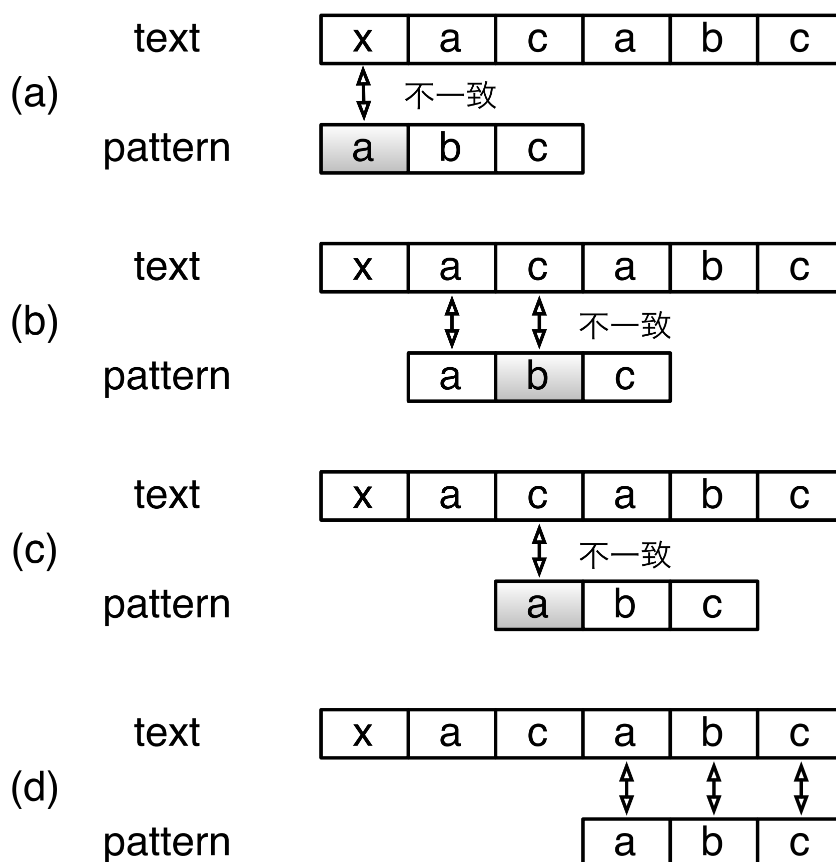


図 3.1: 力まかせ法

このアルゴリズムは実装が簡単であり、pattern が text に含まれていれば必ず探しだすことができる。しかし、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 の文字数を n、pattern の文字数を m とすると、力任せ法の最悪計算時間は $O(nm)$ となる。

3.2.2 Boyer-Moore String Search Algorithm

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。そして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。

まず始めに、比較する場所を着目点とおく。図 3.2 の場合、最初に比較する pattern の末尾 と、それに対応する text を着目点とする。(a) ではその着目点で不一致が起こっているので、それ以上比較しなくてもよいことがわかる。不一致が起こった場合は (b) のように着目点をずらしていく。着目点を 1 つ右にずらして再度 pattern の末尾から比較していく。これを繰り返しをして、(d) のときに初めて一致することがわかる。

(a) のときに不一致を起こした text の文字に注目する。その文字が pattern に含まれていない文字であるならば、着目点を 1 つずらしても、2 つずらしても一致することはない。pattern に含まれていない文字で不一致になった場合は、pattern の文字数分だけ着目点をずらすことができる。

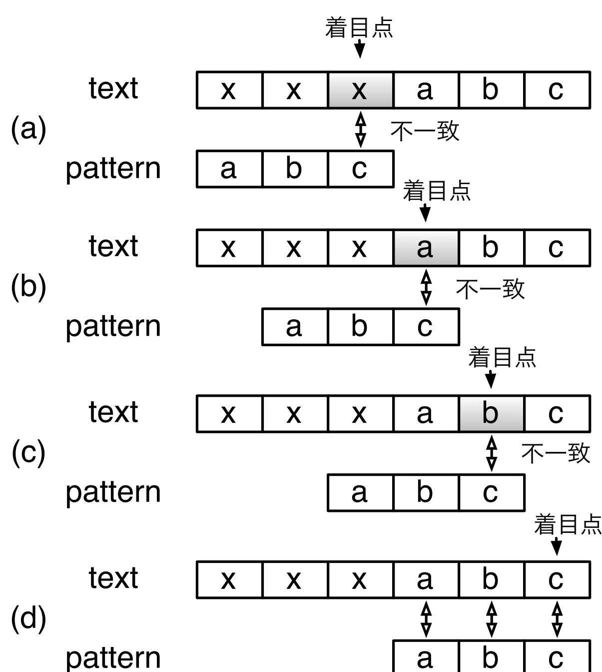


図 3.2: pattern に含まれていない文字で不一致になった場合

次に、pattern に含まれている文字で不一致になった場合を紹介する。図 3.2 と同様に、文字を比較していく。図3.3(a) のときに不一致が起こる。その時の text の文字列は pattern に含まれている。この場合は着目点を右に 2 つずらすと text と pattern が一致する。もし、pattern に含まれている文字で不一致になった場合は、その text の文字に注目する。その文字を pattern 内から探し、その文字が pattern の末尾からどれだけ離れているかで着目点を右にずらす量が決定される。図 3.3 の場合であれば、不一致時の文字が a であれば右に 2 つ、b であれば右に 1 つずらすことができる。

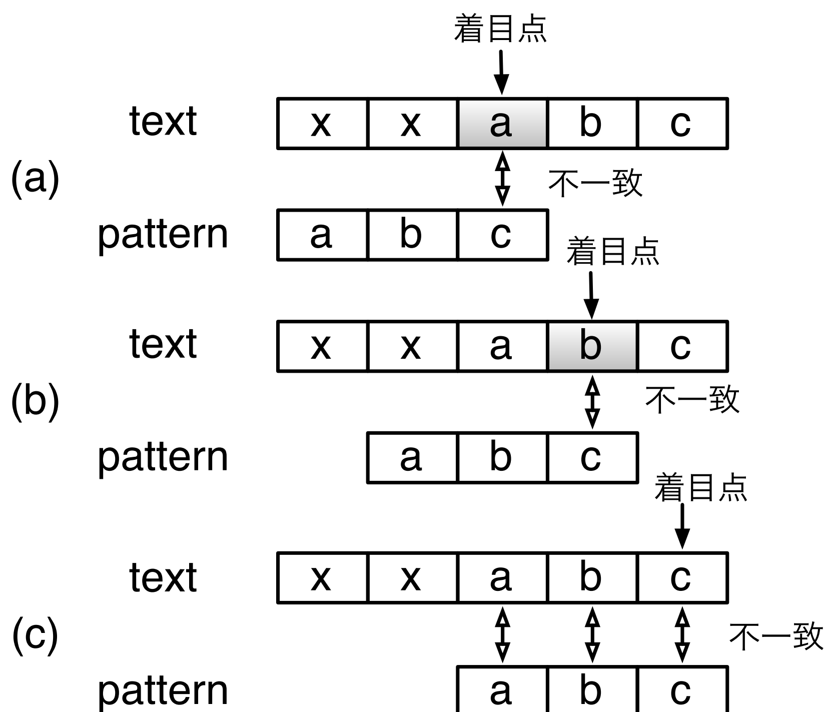


図 3.3: pattern に含まれている文字で不一致になった場合

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は図 3.4 のようになる。この場合 a という文字が pattern の末尾から 1 つ離れている箇所と 3 つ離れている箇所が存在する。(a) のように、a で不一致が起こった場合は、着目点を右に 1 つか 3 つ移動できる。しかし、着目点を右に 3 つずらしてしまうと、(b-1) のように text の途中にある "abac" という文字列を見逃してしまう。着目点を右に 1 つずらせば、(b-2) のように検索漏れを起こすことはなくなる。

このように、pattern に同じ文字が複数入っている場合は、末尾から一番近いほうを適用する。よって、図 3.3 では、不一致時の文字が a であれば右に 1 つ、b であれば右に 2 つ着目点をずらすことができる。

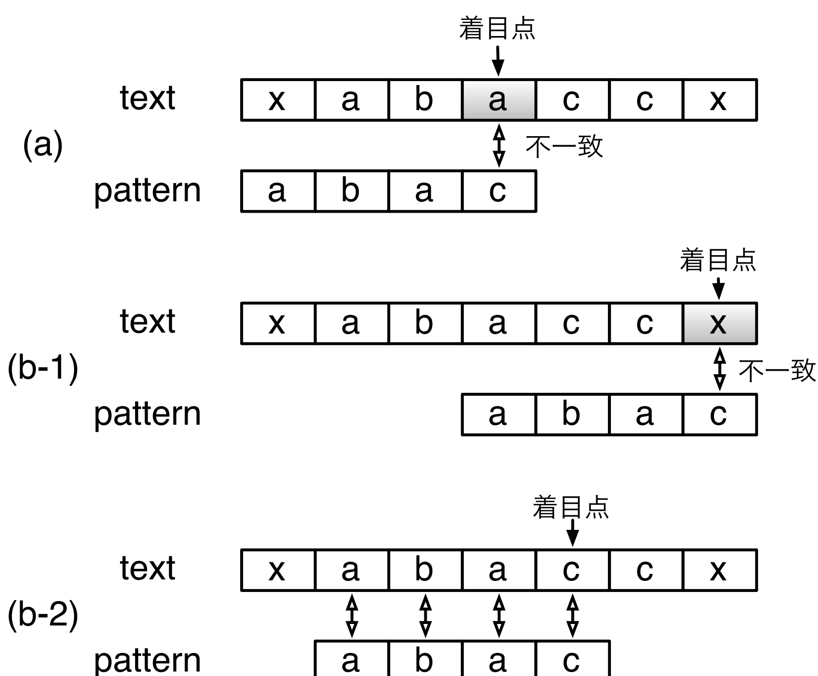


図 3.4: pattern に同じ文字が複数入り、その文字で不一致になった場合

pattern 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の場合は、 pattern の長さだけ着目点を右にずらす
- pattern に含まれている文字の場合は、 その文字が pattern の末尾から離れている分だけ着目点を右にずらす
- pattern に含まれている文字かつ、その文字が pattern に複数含まれている場合は、pattern の末尾から一番近い分だけ着目点を右にずらす

となる。 図 3.5 の例であれば、不一致字の text の文字が a であれば着目点を 2 つ、 b であれば 1 つ、それ以外の文字列は 3 つずらすことができる。

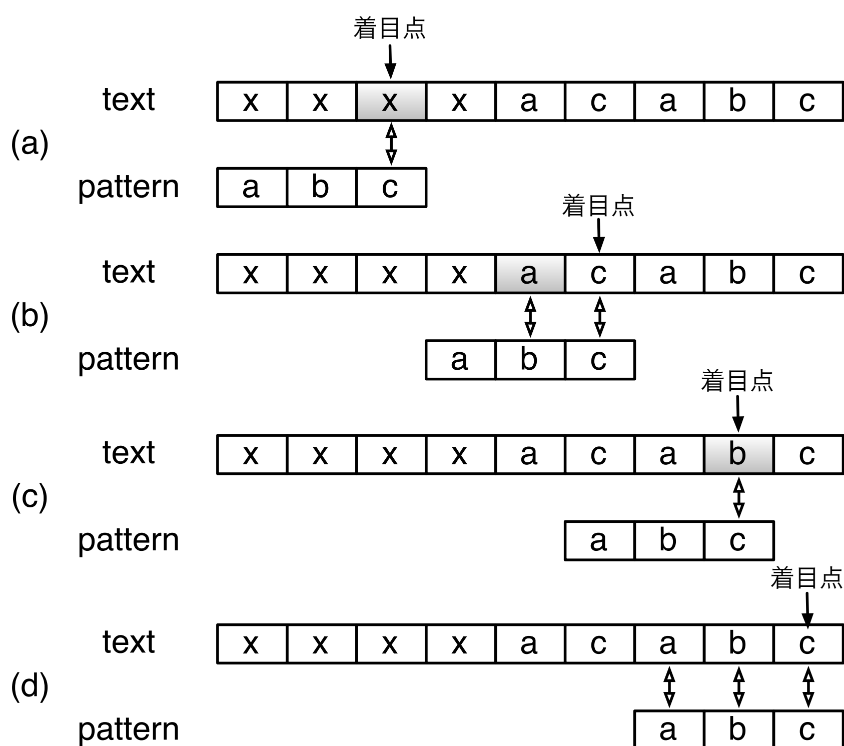


図 3.5: Boyer-Moore Search String

このように、Boyer-Moore Search String は、不一致が起こった時の text の文字によって着目点をずらすということが起こるので、文字列検索を行う前に、着目点をずらす量を参照するための BMsearch skip table を作成する。”doing” という pattern であれば、そのテーブルは図 3.6 となる。

文字	a	b	c	d	e	f	g	h
移動量	5	5	5	4	5	5	5	5

文字	i	j	k	l	m	n	o	・・・
移動量	2	5	5	5	5	1	3	・・・

図 3.6: BMsearch skip table

このようなテーブルを、文字列検索を行う前に生成する必要がある。そのテーブル生成プログラムは以下のようになる。

```
static int*
create_BMskiptable(unsigned char *pattern,
                    int pattern_len,int *skip_table)
{
    for(int i = 0; i < 128; ++i){
        skip_table[i] = pattern_len;
    }

    for(int j = 0; j < pattern_len - 1; ++j){
        skip_table[pattern[j]] = pattern_len - j - 1;
    }

    return skip_table;
}
```

このソースでの 128 とは ASCII コード表における最大値である。それぞれの文字に対して pattern の長さである pattern_len を格納する。pattern が ”doing” という文字列だと仮定すると、pattern_len = 5 となる。次に pattern の先頭から文字を調べる。先頭の文字は d であり、d に対応する table に着目点をずらすための移動量を格納する。移動量を格納したら、pattern の次の文字を調べ、そして移動量を格納していく。(図 3.7)

(a)

10進数	97	98	99	100	101	102	103	104
文字	a	b	c	d	e	f	g	h
移動量	5	5	5	5	5	5	5	5

10進数	105	106	107	108	109	110	111	
文字	i	j	k	l	m	n	o	・・・
移動量	5	5	5	5	5	5	5	・・・

(b)

10進数	97	98	99	100	101	102	103	104
文字	a	b	c	d	e	f	g	h
移動量	5	5	5	4	5	5	5	5

10進数	105	106	107	108	109	110	111	
文字	i	j	k	l	m	n	o	・・・
移動量	5	5	5	5	5	5	5	・・・

(c)

10進数	97	98	99	100	101	102	103	104
文字	a	b	c	d	e	f	g	h
移動量	5	5	5	4	5	5	5	5

10進数	105	106	107	108	109	110	111	
文字	i	j	k	l	m	n	o	・・・
移動量	5	5	5	5	5	5	3	・・・

(d)

10進数	97	98	99	100	101	102	103	104
文字	a	b	c	d	e	f	g	h
移動量	5	5	5	4	5	5	5	5

10進数	105	106	107	108	109	110	111	
文字	i	j	k	l	m	n	o	・・・
移動量	2	5	5	5	5	1	3	・・・

図 3.7: skip table の 生成時の様子

生成したテーブルを利用して文字列検索を行うアルゴリズムを以下に示す。

```
int i = pattern_len - 1;
int match_counter = 0;

while ( i < text_len){
    int j = pattern_len - 1;
    while (text[i] == pattern[j]){
        if (j == 0){
            match_counter++;
        }
        --i;
        --j;
    }
    i = i + max(skip_table[text[i]],pattern_len - j);
}
```

読み込まれたテキストファイル text[i] と、検索文字列 pattern[j] を末尾から比較していく。一致していれば参照する場所を先頭方向にずらしていき、先頭まで完全一致した場合は、検索文字列を発見したということで、ここではマッチした数 match_counter を増やしている。

もし途中で不一致が起こった場合は、その不一致が起こった時の text[i] の文字列か pattern の長さから j 引いたものの最大値の分だけ text の参照先をずらす。

3.2.3 ファイル分割時の処理の注意点

この例題ではファイルを読み込んで一定の大きさでファイルを分割する。分割したものにそれぞれ文字列検索を行う。それぞれの結果は pattern が含まれている個数が返ってくるので、最後に集計をして個数を表示する。このような一つ一つの処理を Task と呼ぶ。図 3.10 では、ファイルの読み込みが File Read、分割したものに文字列検索を行うことが Run Tasks、返した結果が Output Data、それぞれの結果の集計が Run resultTask に相当する。

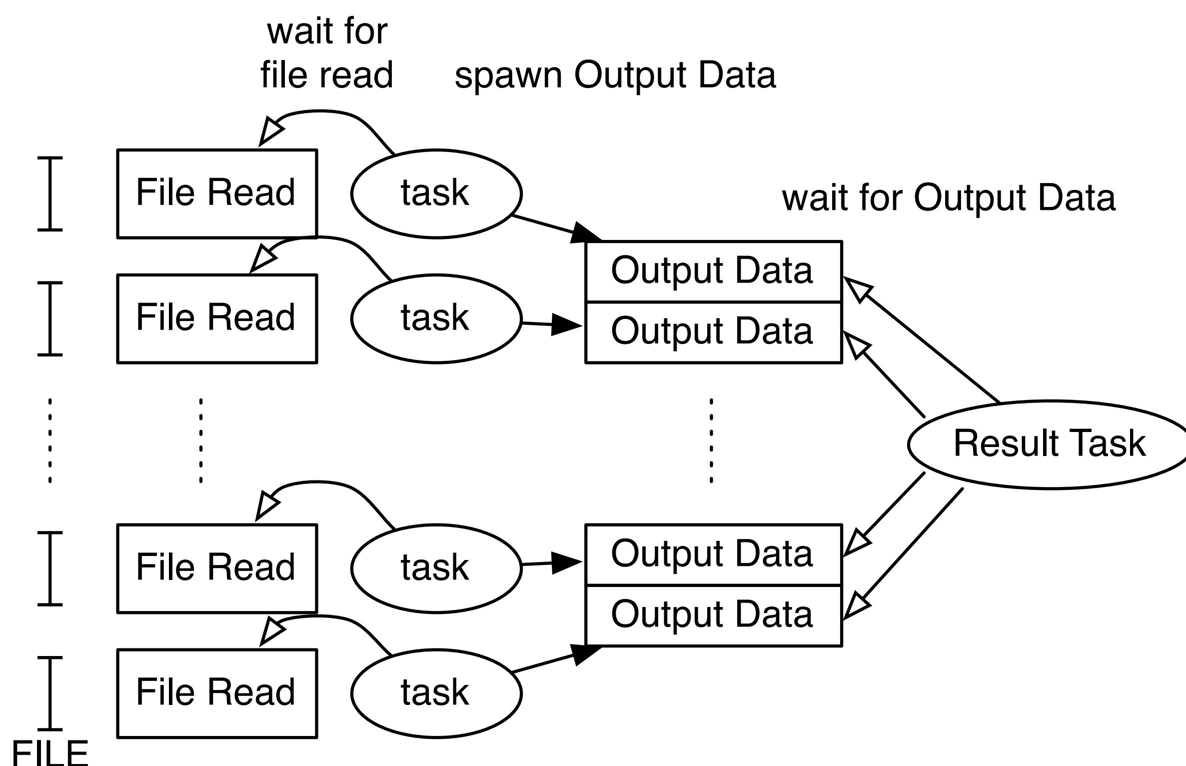


図 3.8: IO を含む Task

ファイルを分割したときに、分割される部分で 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 だけ多く読みこむように設計することでこの問題は解決できる。しかし、1 つの Search Task の text の長さが $L+s$ の場合だと、pattern が Search Task 1 に含まれ、Search Task 2 にも含まれてしまう。(図 3.10)

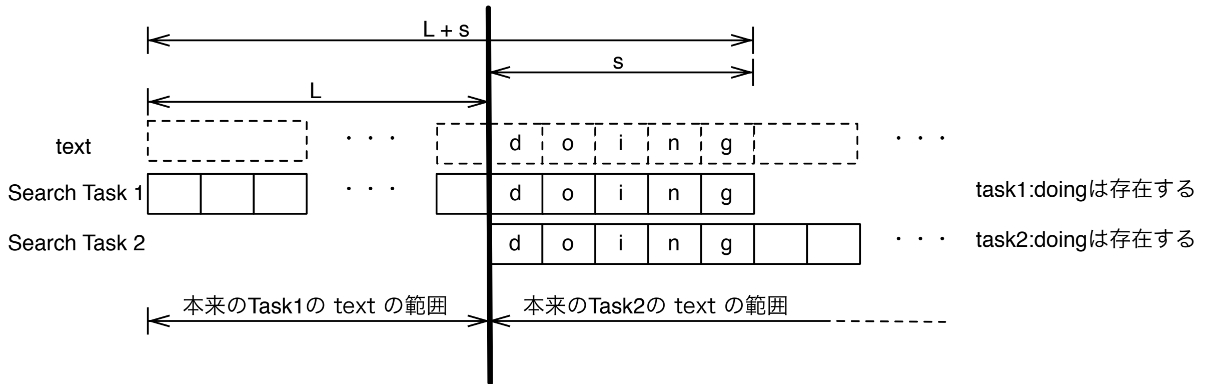


図 3.9: 分割周りの処理・失敗時 (例:doing の検索)

それを解決するために、1 つの Search task の text の長さに pattern の長さを加えてから 1 引いた数だけ読み込むようにすればそのような問題は起こらない。よって、読み込むデータ量は $L+s-1$ となる。

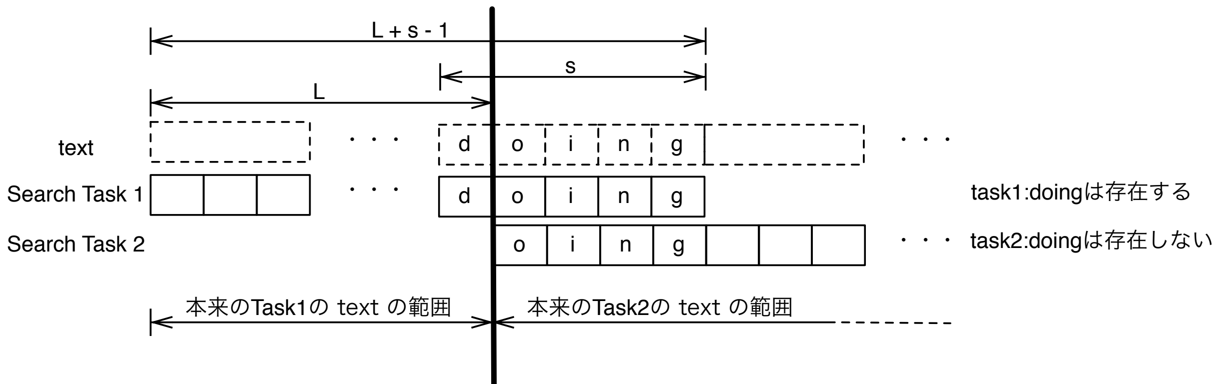


図 3.10: 分割周りの処理・成功時 (例:doing の検索)

力任せ法と Boyer-Moore String Search を比較してみた。以下に実験環境と結果を示す。(表 3.2.3)

- Mac OS X 10.9.1
- 2*2.66 GHz 6-Core Intel Xeon
- File Size 10GB

文字列検索アルゴリズム	処理速度 (s)
力任せ法	11.792
Boyer-Moore String Search	6.508

表 3.3: 文字列検索アルゴリズムの比較

Boyer-Moore String Search によって 44% 改善した。

第4章　並列処理向け I/O の設計と実装

4.1　map reduce の設計

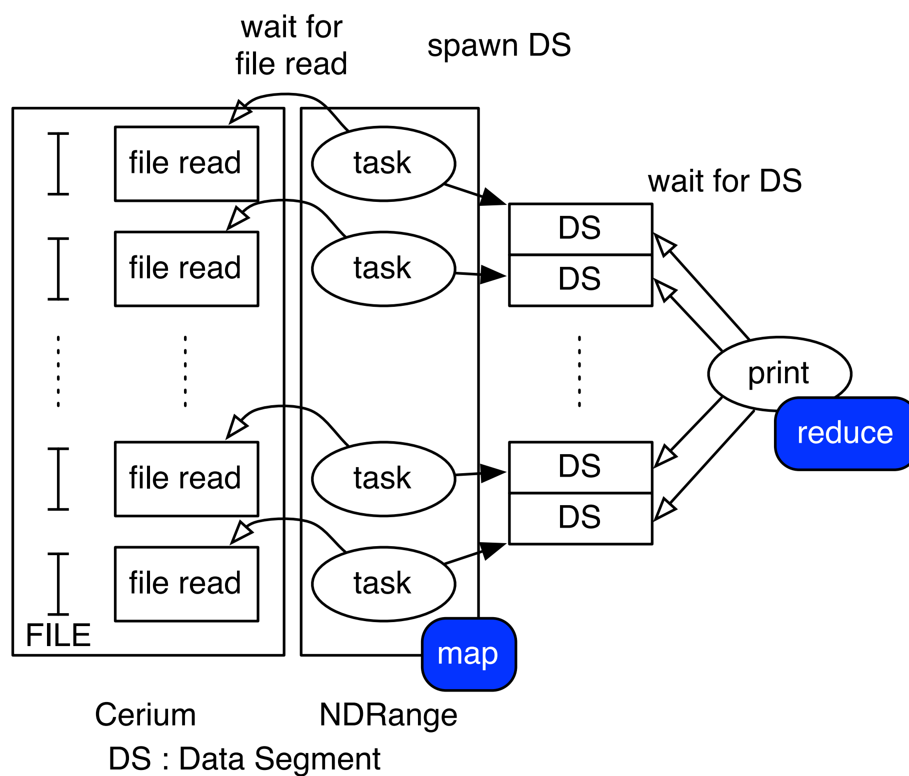


図 4.1: map reduce image

4.2 mmap での実装の問題点

mmap とは、sys/mman.h に含まれている関数で、ファイルの読み込み等に使用される関数である。ファイルディスクリプタで指定したファイルを offset から len バイトの範囲を読み込む。この時にアドレス addr からメモリを確保するようにする。prot には、PROT_READ によるページの読み込み、PROT_WRITE によるページへの書き込みなどを指定でき、flags にはメモリ確保する際のオプションを指定することができる。(表 4.2)

void * mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);

void *addr	メモリに確保するときの先頭のアドレス
size_t len	メモリを確保するサイズ
int prot	ファイルモード選択
int flags	確保するときのオプション指定
int fd	読み込むファイルのファイルディスクリプタ
off_t offset	ファイル先頭からの読み込み開始位置

表 4.1: mmap 関数の概要

mmap でファイルを読み込むタイミングは、mmap 関数が呼ばれたときではなく、mmap した領域に対して何らかのアクセスをしたときに初めてファイルが読み込まれる。

図 4.2 では、読み込んだファイルを分割して、それらの領域に何らかの処理を加えるときの図である。これらの処理を Task と呼ぶ。Task 1 という 1 個目の Task が実行される。実行されたときに初めてそれらの領域にファイルが読み込まれ、その後何らかの処理が行われ、そして Task 2 も同様に読み込みを行ってから処理が行われる。これら Task は並列に実行されるべきであるが、ファイル読み込みの I/O 部分がネックとなり、本来並列実行される Task が読み込み待ちを起こしてしまう恐れがある。その上、読み込み方法が OS 依存となるために環境によって左右されやすく、プログラムの書き手が読み込みに関して制御しにくい。

それらを解決するためには、ファイル読み込みと Task を分離し、ファイルの読み込みも制御しやすくでき、なおかつ高速で動くのではないかと考えた。

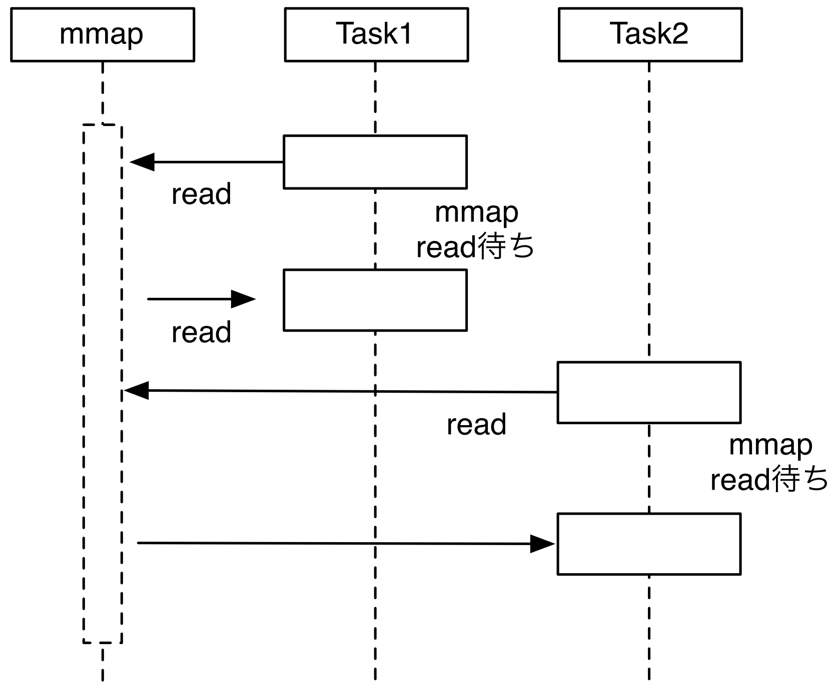


図 4.2: mmap image

4.3 Bloked Read の設計と実装

Blocked Read とは、読み込みの Task と、それに対する何らかの処理の Task を切り離すための実装方法で、pread 関数にて実装した。pread 関数は、unistd.h に含まれているので、UNIX 専用の関数である。ファイルディスクリプタで指定したファイルの先頭 から offset 分ずれた場所を基準として、その基準から count バイトを読み込み、それを buf に格納する。4.3

ssize_t pread(int d, void *buf, size_t nbyte, off_t offset);

int d	読み込むファイルのファイルディスクリプタ
void *buf	読み込んだファイルの格納場所
size_t nbyte	読み込むファイル量
off_t offset	ファイル先頭からの読み込み開始位置

表 4.2: pread 関数の概要

mmap での実装との違いは、ファイルの読み込みがどのタイミングで起こるかである。mmap で実装したときは、Task 1 つ 1 つが読み込みを行ってから処理を行う。それに対して、Blocked Read は、読み込み専用の Read Task と、処理専用の Task を別々に生成する。Read Task はファイル全体を一度に読み込むのではなく、ある程度の大きさで分割を行う。分割して読み込み終わったら、それぞれの Task が実行される。(図 4.4) Read Task が生成されて、その後 Task の生成となるので、Read Task は常に走っている必要がある。

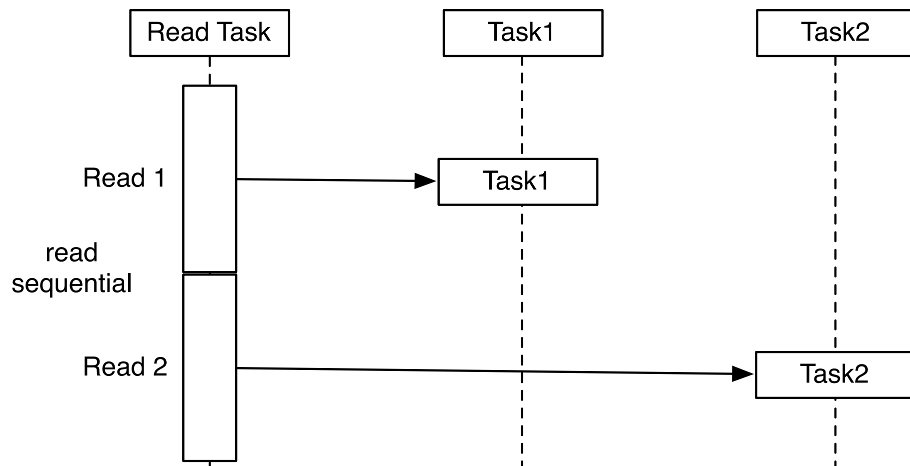


図 4.3: Blocked Read image

図 4.4 では、Read Task 1 つに対して Task 1 つ起動しているが、このように 1 つ 1 つ生成、起動をすると Task 生成でメモリを圧迫してしまい、全体的な動作に影響を与えてしまう。実際には Task をある一定数まとめた単位で生成し、起動を行っている。この単位を Task Block と定義する。

Task Block 1 つ当たりの Task 量を n とおく。Task 1 つ当たりの読み込む量を L とすると、Task Block 1 つ当たりの読み込む量は $L \times n$ となる。Blocked Read が読み込み終わってから、Task Block が起動するようにするので、Blocked Read 1 つ当たりの読み込み量も $L \times n$ となる。

もし、Task Block が Blocked Read よりも先走ってしまうとどうなるであろうか。まだ読み込まれていない領域に対して何らかの処理を行ってしまうので、正しい結果が返ってこなくなってしまう。それを防止するために、Blocked Read が読み込み終わってから Task Block が起動されるように wait をかけている。

(図 4.4)

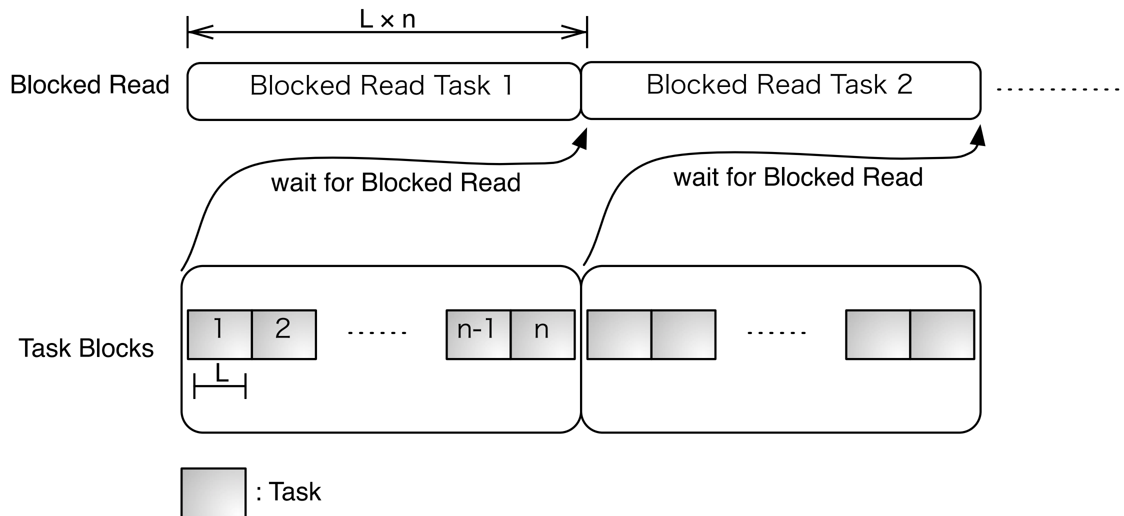


図 4.4: Blocked Read image

4.4 I/O 専用 thread の実装

Cerium Task Manager では、各種 Task にデバイスを設定することができる。デバイスとは、GPU や CPU であり、GPU を利用するときは GPU_ANY、CPU を利用するときは SPE_ANY と設定することによってデバイスを利用できる。

SPE_ANY を使用すると、Task Manager で CPU の割り振りを自動的に行う。しかし、この機能を使用すると、Blocked Read に影響を与えてしまう。

Blocked Read 、Task それぞれに SPE_ANY で CPU を自動的に割り振ると、Task Manager 側で自動的に CPU を割り当てる。このように CPU を割り当ててしまうと、本来 Blocked Read は連続で読み込むはずが、他の Task を割り当てられてしまう。(図 4.5)

この問題を解決するために、Task Manager に新しく I/O 専用の thread を用意した。(図 4.6)

この設定は他のデバイス設定よりも priority を高く設定している。

SPE_ANY で使用する CPU の設定よりも高く設定しているので、IO_0 で設定を行う Read Task に SPE_ANY で設定した 文字列検索 Task に割り込まれることがなくなる。(図 4.7)

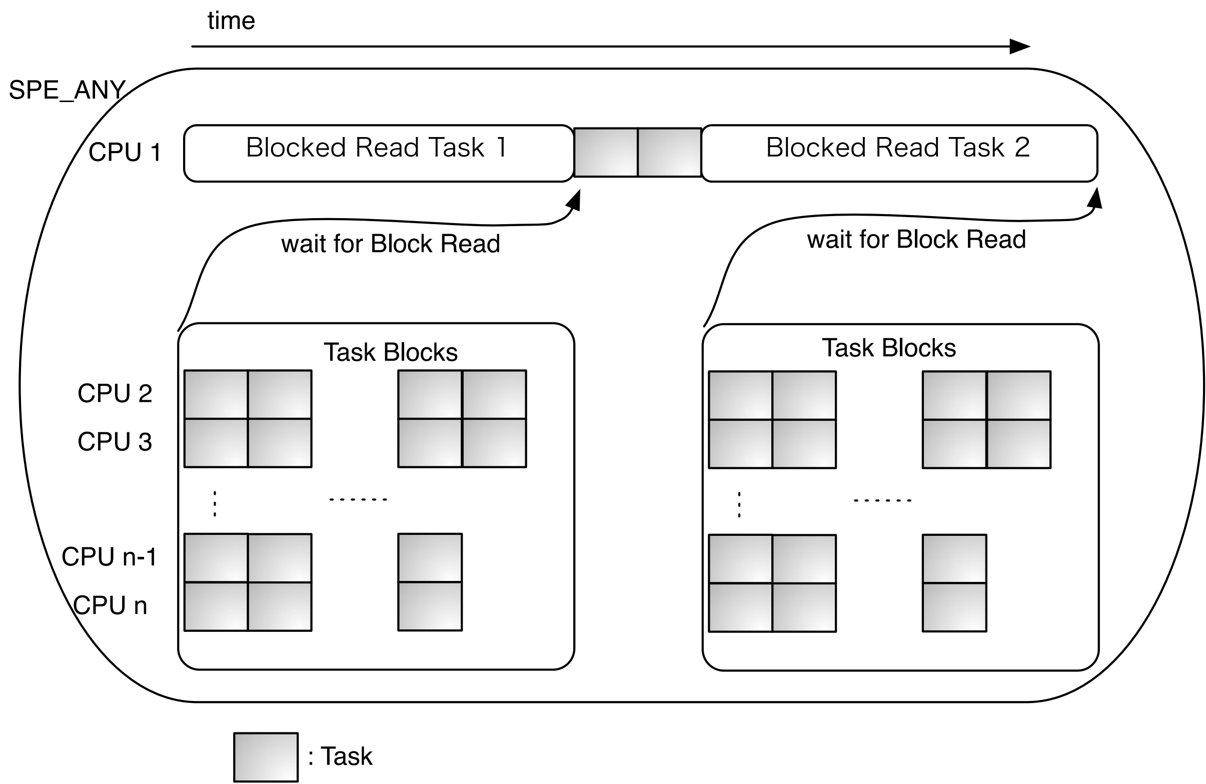


図 4.5: SPE_ANY での実装時

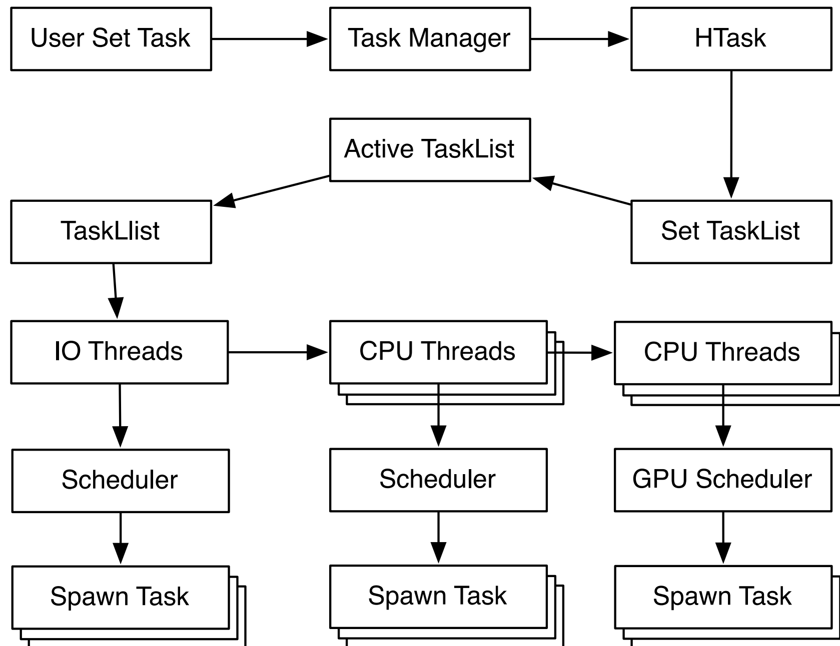


図 4.6: IO_0 の追加

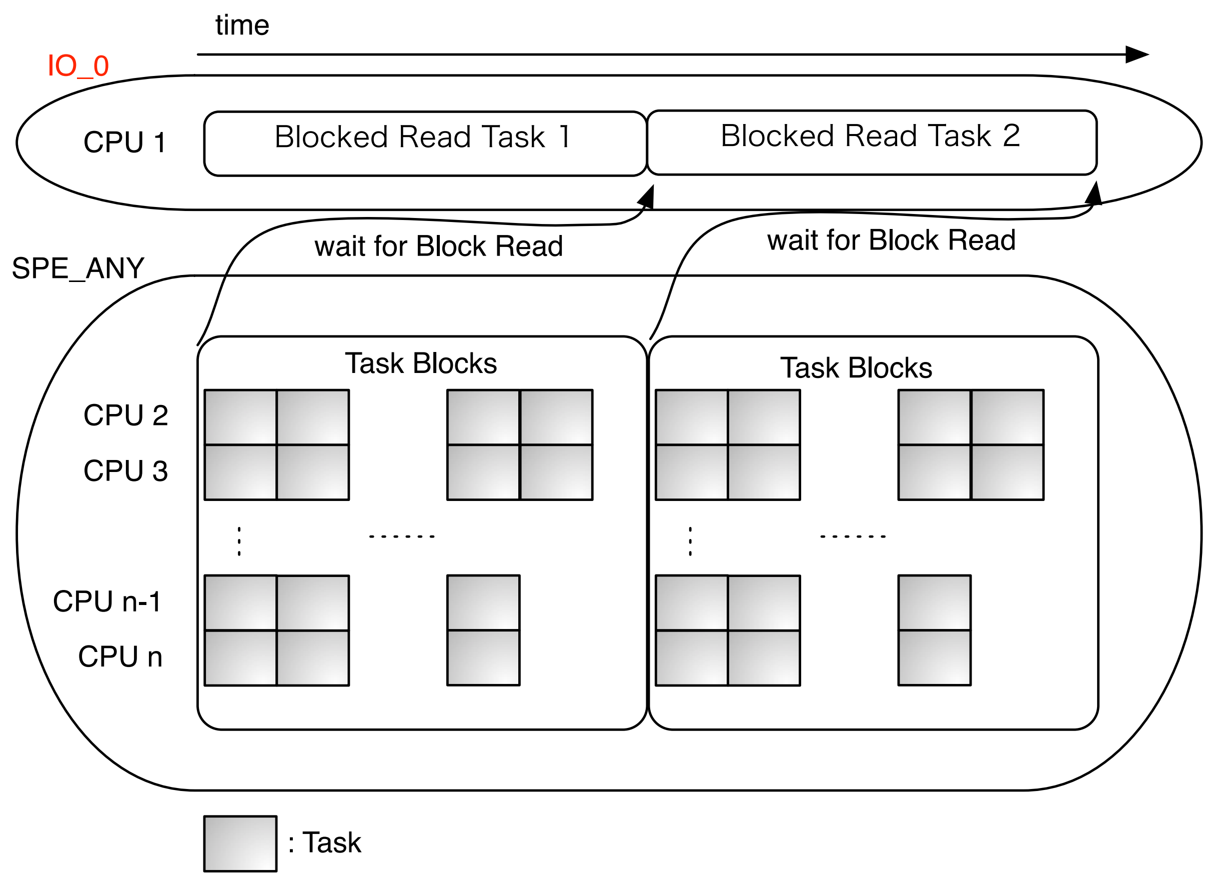


図 4.7: Blocked Read Task を IO_0 での実装時

IO_0 の priority を高く実装したソースコードは以下のようになる。

```
void *
CpuThreads::cpu_thread_run(void *args)
{
    cpu_thread_arg_t *argt = (cpu_thread_arg_t *) args;

        ...

    if (argt->cpuid >= argt->cpu_num) {
        // set IO thread priory maximum
        int policy;
        struct sched_param param;
        pthread_getschedparam(pthread_self(), &policy, &param);
        param.sched_priority = 1;
        pthread_setschedparam(pthread_self(), policy, &param);
    }

    return NULL;
}
```

(ソース説明)

第5章　ベンチマーク

5.1　実験環境

- Mac OS X 10.9.1
- 2*2.66 GHz 6-Core Intel Xeon
- Memory 16GB 1333MHz DDR3
- HHD 1TB
- file size 10 GB
- CPU num 12
- Boyer-Moore String Search で pattern がいくつ含まれているか検索
- ファイルを読み込みから結果が返ってくるまでを測定

5.2　結果

以下の表に実行結果を示す。

読み込み方法	平均実行速度 (s)
mmap	154.6
Blocked Read & SPE_ANY	106.0
Blocked Read & IO_0	99.2

表 5.1: 実行結果

実験結果より、mmap より Blocked Read & IO_0 の実行速度が 36 % 改善された。また、Blocked Read の CPU Type も SPE_ANY から IO_0 に変更することによって更に 4 % の改善が見られた。

5.3 考察

mmap より Blocked Read で実装したほうが速くなったが、これは mmap の読み込み方法が問題であると考える。

I/O を含む例題の場合、シングルコアでの逐次実行であれば、mmap や pread で実装しても、Task は 読み込みを行って文字列検索を行うというシンプルな動作になる。しかし、マルチコアの並列実行であれば、mmap で実装してしまうと、Task それぞれで読み込みを行ってしまうので競合が発生してしまう。

読み込みの競合が起こらないように Blocked Read にて読み込み部分と文字列検索部分を分けた結果、こちらのほうが速度が向上した。

第6章　結論

6.1　まとめ

本研究では、I/O を含む Task の並列処理の動作の改善を行った。ファイルを mmap でメモリを確保すると、文字列検索を行う Task が読み込みを行い、それが終了後に検索が行われる。読み込みが各 Task それぞれに割り当てられてしまうので、すべての Task が読み込み待ちとなってしまう。それを解決する方法として、読み込みを行う Task と文字列検索を行う Task を分けるように Blocked Read の設計と実装を行った。

Blocked Read である程度の大きさを読み込んだら Task が順次起動するように実装したが、それだけだと順次読み込んでいる Blocked Read に Task が割り込まれてしまう。そのようなことが起こらないように、Cerium Task Manager に新しいデバイスの設定 IO_0 というタイプを追加した。このデバイスは、他のデバイス設定よりも priority を高く設定しているので、このタイプ以外で起動する Task に割り込まれることが起こらなくなる。

これらを実装した結果、本研究では mmap で実装したときよりも 36 % の動作改善が見られた。本研究を通して、I/O を含む Task の並列化の問題において、I/O の動作を改善する余地があると考える。

6.2　今後の課題

6.2.1　実メモリ以上のファイルの取り扱い

本研究での実験では、実メモリ以上のファイルを取り扱ってはいない。Blocked Read のメリットとして、実メモリ以上のファイルサイズを取り扱うことができることである。実メモリ以上のファイルを読み込む際には一旦分割して読み込み、その読み込み領域を担当する Task が終了したら、そのメモリ領域を解放する。そして、その解放した部分に読み込みを行うことで、メモリの節約にも利用できる。

現段階での実装は、Task Manager 側で予めテキストファイルの大きさをメモリに確保して、その部分に読み込んだファイルを格納していくようになっている。この実装方法ではメモリ以上のファイルを読み込めないので、この問題を解決する必要がある。

6.2.2 Blocked Read で読み込んだファイルがキャッシュに残らない

本来読み込みを行ったファイルは、一度プログラムを実行したあとでもキャッシュとしてメモリ上にテキストがそのまま残っている。キャッシュとは、使用頻度の高いデータを高速なデバイスに蓄えておくことによって読み込みのオーバーヘッドを少なくするための機能である。

ハードディスクはメモリと比較すると読み込みが遅いので、ハードディスクからファイル読み込みを行うと、読み込みが速いメモリのほうに格納される。読み込んだファイルが再利用されるとき、ハードディスクからメモリに格納するという時間が無くなるので、2 回目以降の実行結果は速くなる。

mmap で実装を行うと、同じファイルに対して複数回検索を行うときに 2 回目以降のプログラムの処理は速くなる。本研究では mmap で 10GB のファイルに対して文字列検索を行うと約 150 秒かかるが、2 回目以降の実行速度に関しては、約 7 秒でプログラムが終了する。

Blocked Read も 2 回目以降の実行速度は mmap と同様に速くなるのだが、ある一定のファイルサイズを越えてしまうとキャッシュが無効となってしまう。10GB のファイルではそのようなことが発生することは確認しているが、どれくらいの大きさからキャッシュが無効になるのか不明である。

キャッシュが無効になってしまうと、Blocked Read で実装した文字列検索は複数回実行するときに不利となる。なぜこのようなことが起こるのか調査して、それが起こらないように実装していきたい。

6.2.3 Blocked Read を Cerium API 化

現段階での Blocked Read の実装は、複雑な書き方で実装しなければならない。この実装を Cerium の API に落としこむことによって、簡単に実装できるようにしたい。

参考文献

[1] 金城裕、河野真治、多賀野海人、小林佑亮 (琉球大学)
ゲームフレームワーク Cerium Task Manager の改良
情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), April 2011

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました hoge 助教授に深く感謝したします。

また、本研究の遂行及び本論文の作成にあたり、日頃より終始懇切なる御教授と御指導を賜わりました hoge 教授に心より深く感謝致します。

数々の貴重な御助言と細かな御配慮を戴いた hoge 研究室の hoge 氏に深く感謝致します。

また一年間共に研究を行い、暖かな気遣いと励ましをもって支えてくれた hoge 研究室の hoge 君、hoge 君、hoge さん並びに hoge 研究室の hoge、hoge 君、hoge 君、hoge 君、hoge 君に感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2010 年 3 月
hoge