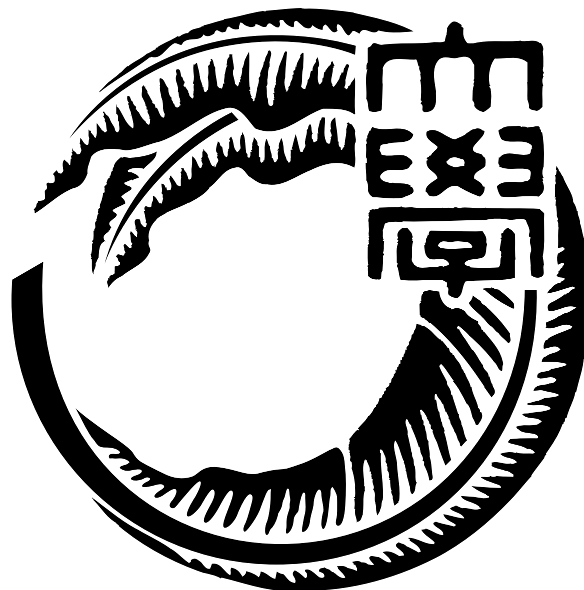
平成25年度 卒業論文

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 の概要 2
2.2　Cerium Task Manager 2

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 6

第4章　並列処理向け I/O の設計と実装 **12**
4.1　map reduce 12
4.2　mmap での実装の問題点 13
4.3　Bloked Read の設計と実装 13
4.4　I/O 専用 thread の実装 15

第5章　ベンチマーク **18**
5.1　実験環境 18
5.2　結果 18
5.3　考察 18

第6章　結論 **19**
6.1　まとめ 19
6.2　今後の課題 19

図 目 次

2.1 Cerium Task Manager 3

3.1 力まかせ法 6
3.2 pattern に含まれていない文字で不一致になった場合 7
3.3 pattern に含まれている文字で不一致になった場合 8
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 8
3.5 Boyer-Moore Search String 9
3.6 BMsearch skip table 9
3.7 IO を含む Task 10
3.8 分割周りの処理・失敗時 (例:doing の検索) 10
3.9 分割周りの処理・成功時 (例:doing の検索) 11

4.1 map reduce image 12
4.2 mmap image 14
4.3 Blocked Read image 14
4.4 Blocked Read image 15
4.5 SPE_ANY での実装時の image 16
4.6 [image]priority を上げたときの image 図 17

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 5
3.3 文字列検索アルゴリズムの比較 11

4.1 mmap 関数の概要 13

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 の概要

・PS3 の Cell 向けに開発されていた。
・Cerium は C 及び C++ で記述されている。
・Mac OS X や Linux でも動作可能。
・マルチコア CPU だけでなく、近年では GPGPU もサポートした。
・並列プログラミングをサポートしているのは Cerium Task Manager。

2.2　Cerium Task Manager

Cerium Task Manager では、並列処理を Task 単位で記述する。関数やサブルーチンをそれぞれ Task として扱い、 Task には Input Data 、 Output Data 及び依存関係を設定することができる。
・User が Task の設定を行い、それを spawn。
・spawn すると 設定された Task が Task Manager で管理される。
・(いろいろとヤバい)
図 2.1
・Task の生成方法の説明
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();
}
```

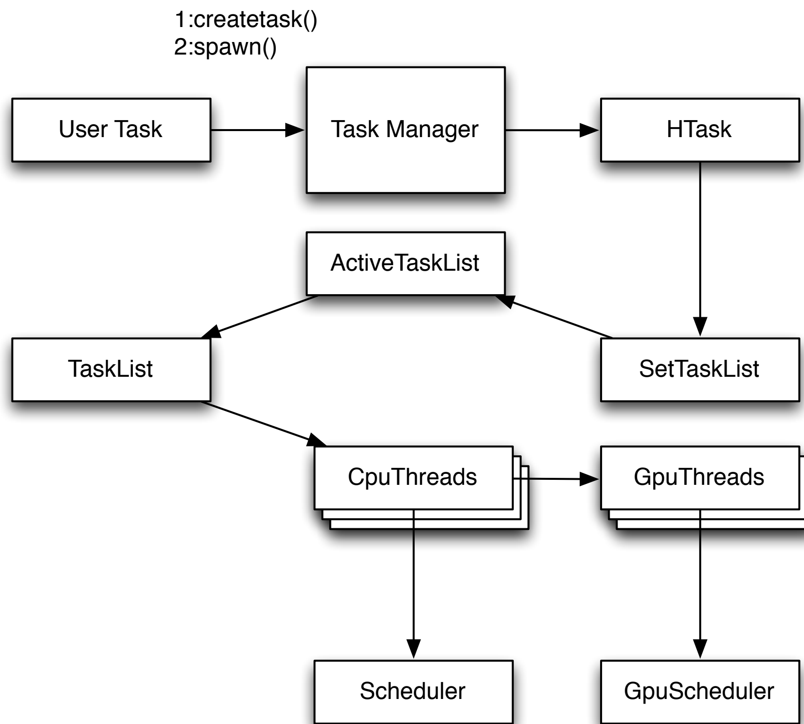


図 2.1: Cerium Task Manager

表 2.1: Task 生成における API

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 への入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

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 3.1) 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

1GB 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	読み込み速度 (s)
16KB	XX.XXX
16MB	XX.XXX
256MB	XX.XXX

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。しかし、ある一定以上の大きさになると I/O ネックが勝ってしまい、読み込み速度が変わらない。

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、Boyer-Moore String Search を実装した。このアルゴリズムは 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。

Boyer-Moore String Search を紹介する前に、文字列検索で比較的単純なアルゴリズムである力任せ法を紹介する。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。

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)

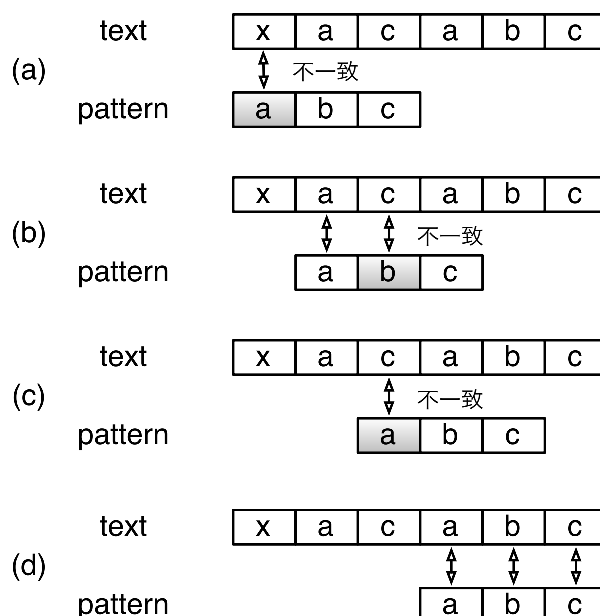


図 3.1: 力まかせ法

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。

こ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。

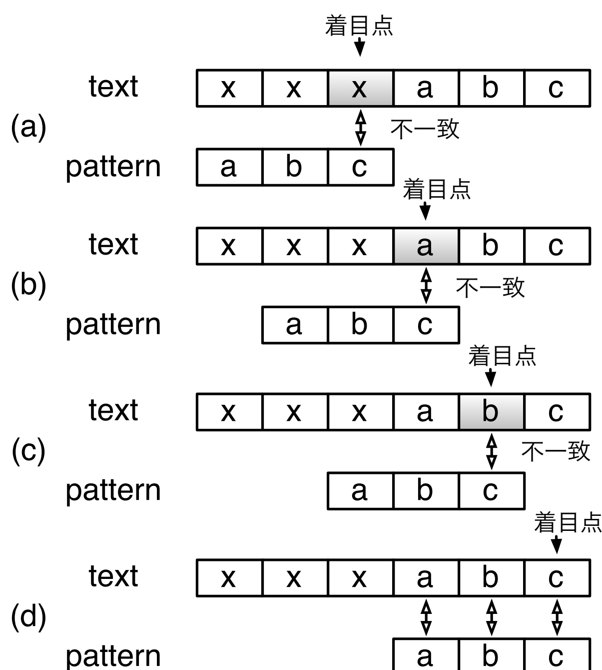


図 3.2: pattern に含まれていない文字で不一致になった場合

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。

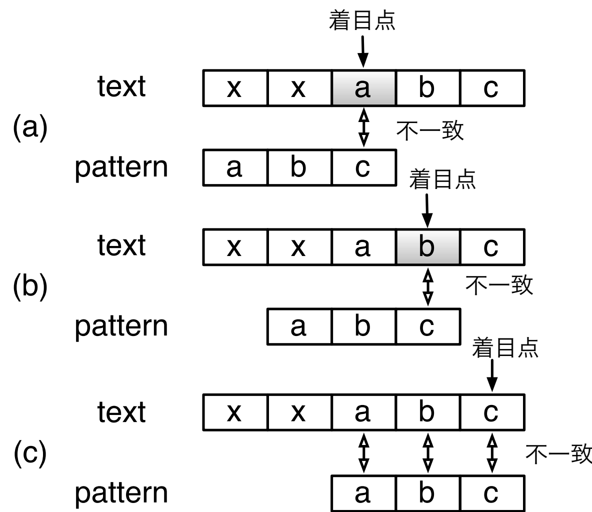


図 3.3: pattern に含まれている文字で不一致になった場合

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。

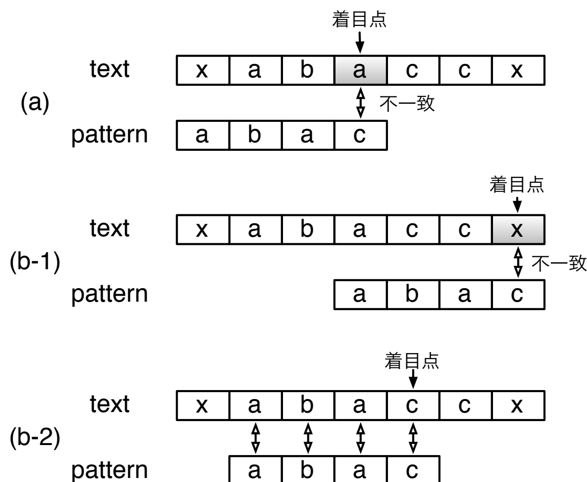


図 3.4: pattern に同じ文字が複数入り、その文字で不一致になった場合

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。

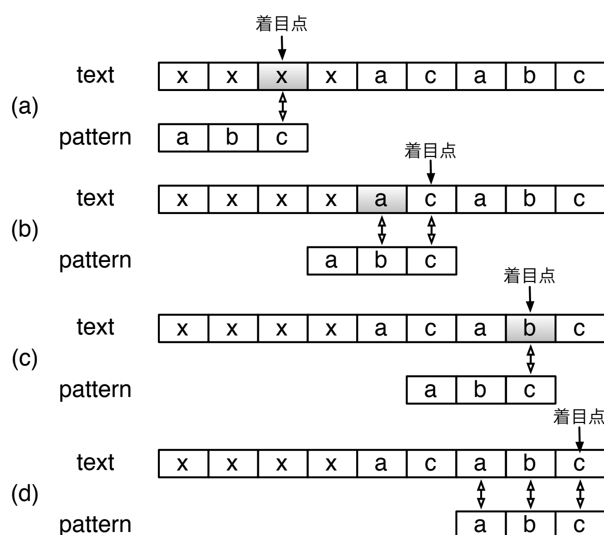


図 3.5: Boyer-Moore Search String

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 Boyer-Moore String Search 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 3.9 では、ファイルの読み込みが File Read、分割したものに Boyer-Moore String Search することが 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。

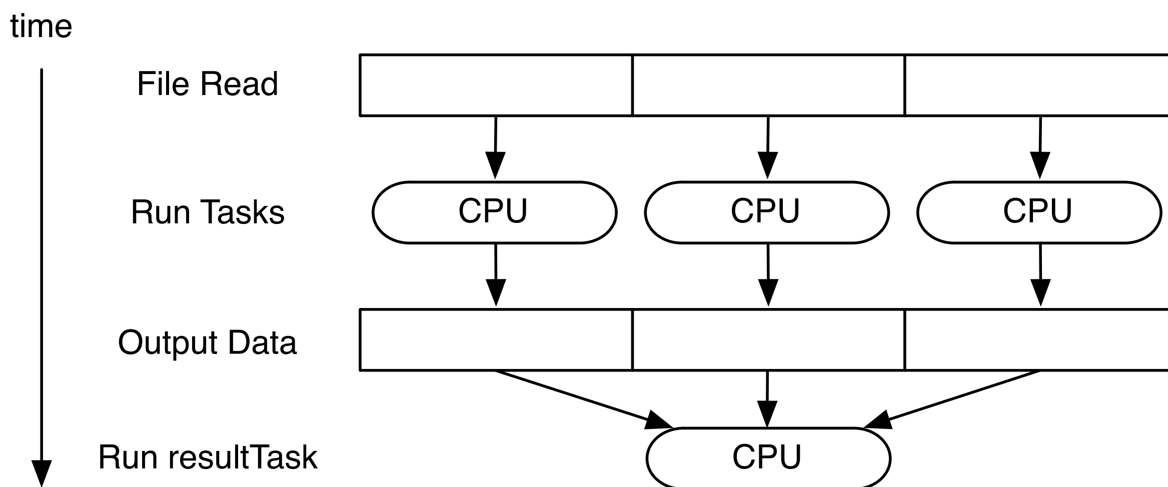


図 3.7: IO を含む Task

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 3.9)

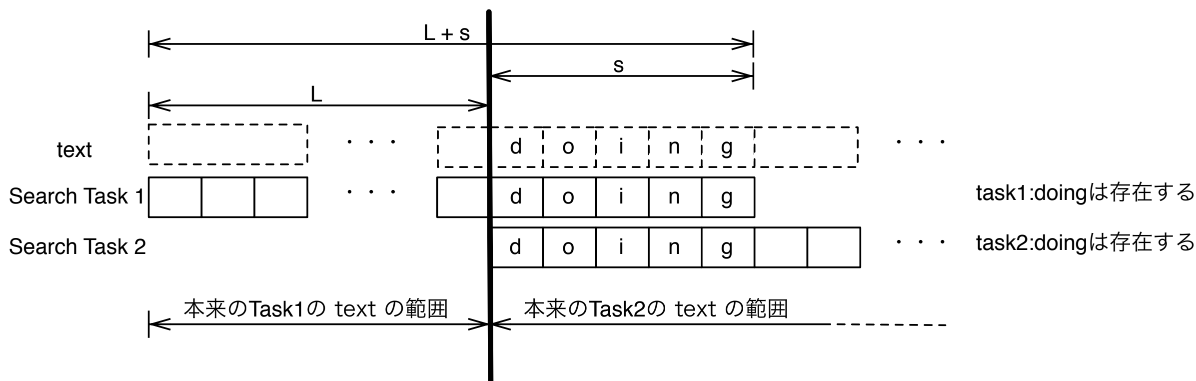


図 3.8: 分割周りの処理・失敗時 (例:doing の検索)

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。

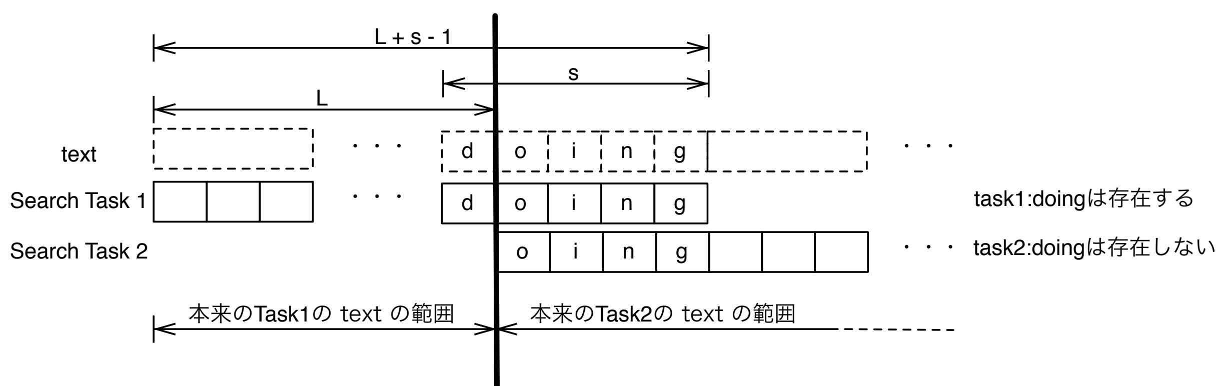


図 3.9: 分割周りの処理・成功時 (例:doing の検索)

力任せ法と Boyer-Moore String Search では以下の表のようになる。実験環境は Mac OS X 10.9、memory 16GB で実験して、ファイルの大きさは 10GB である。

mode	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

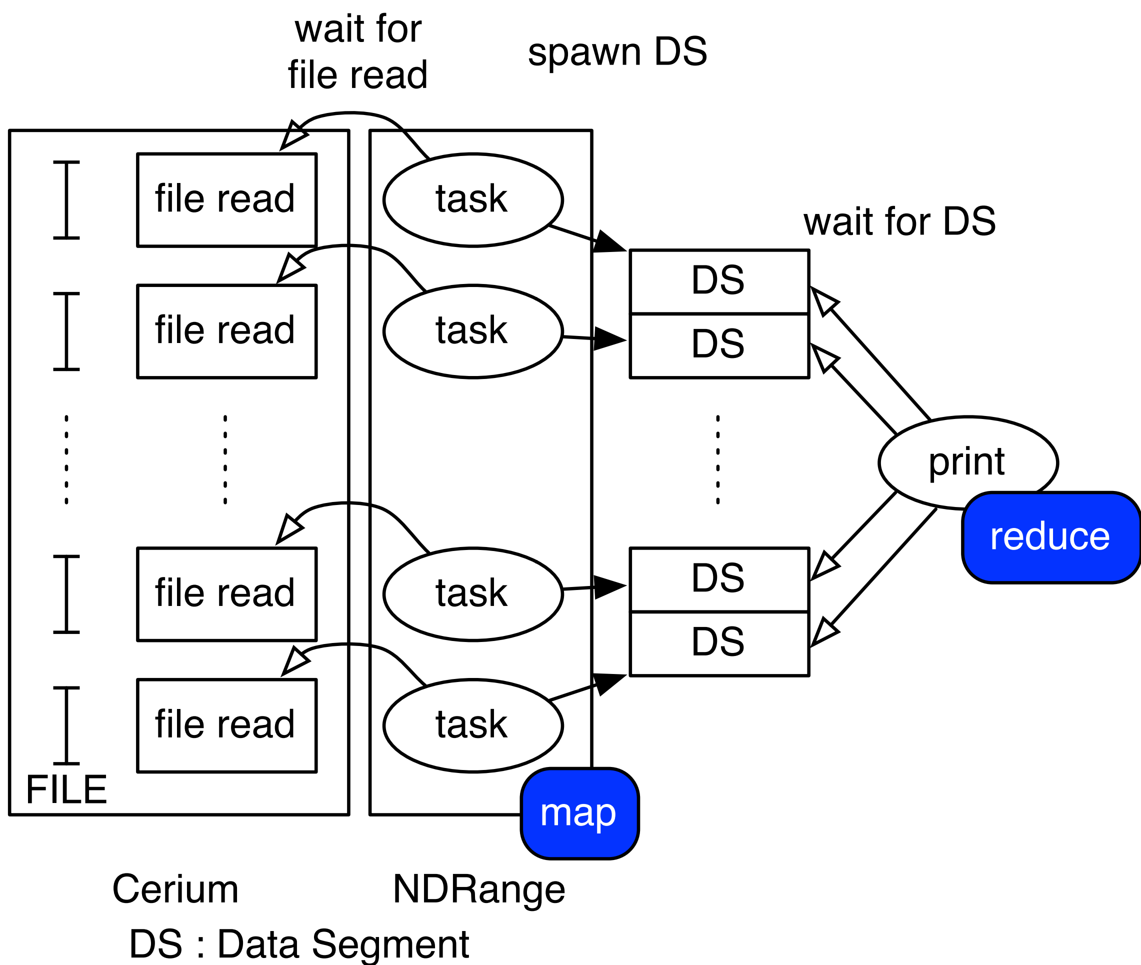


図 4.1: map reduce image

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	ファイル読み込みの先頭からの開始位置

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。

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 関数で実装した。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。

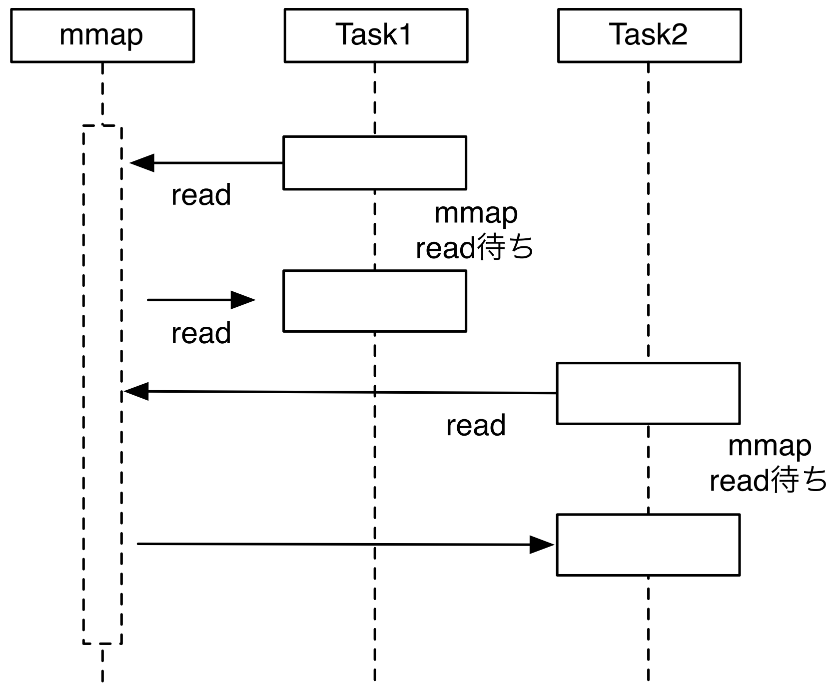


図 4.2: mmap image

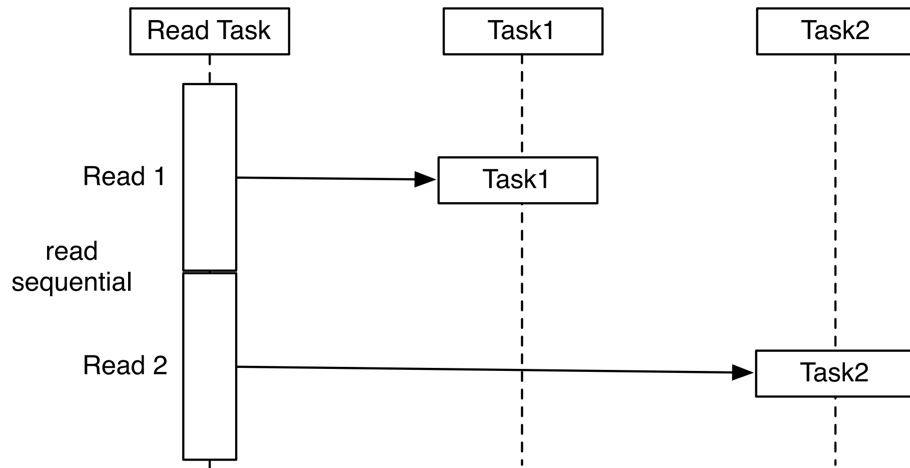


図 4.3: Blocked Read image

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)

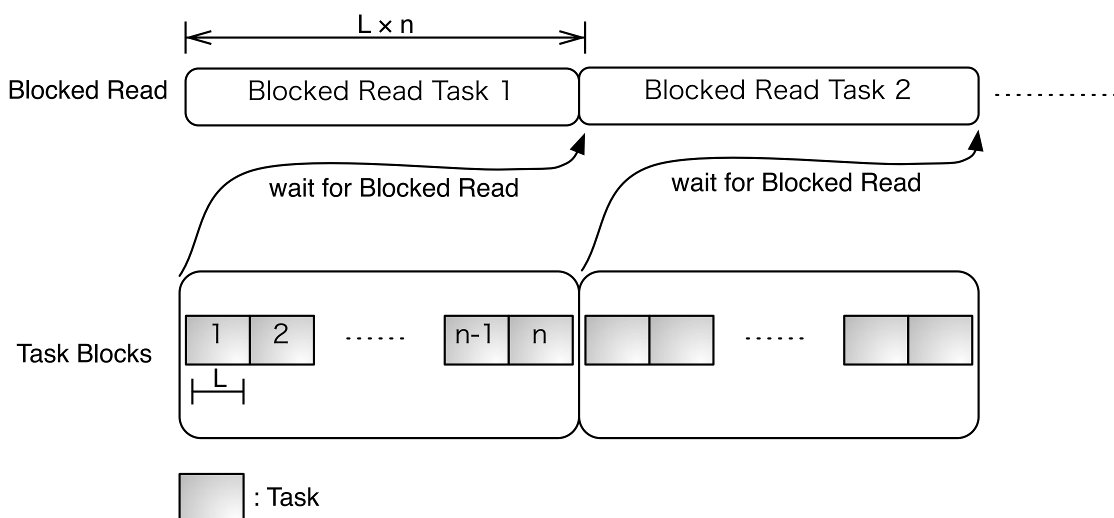


図 4.4: Blocked Read image

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ってくれる便利な機能である。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。

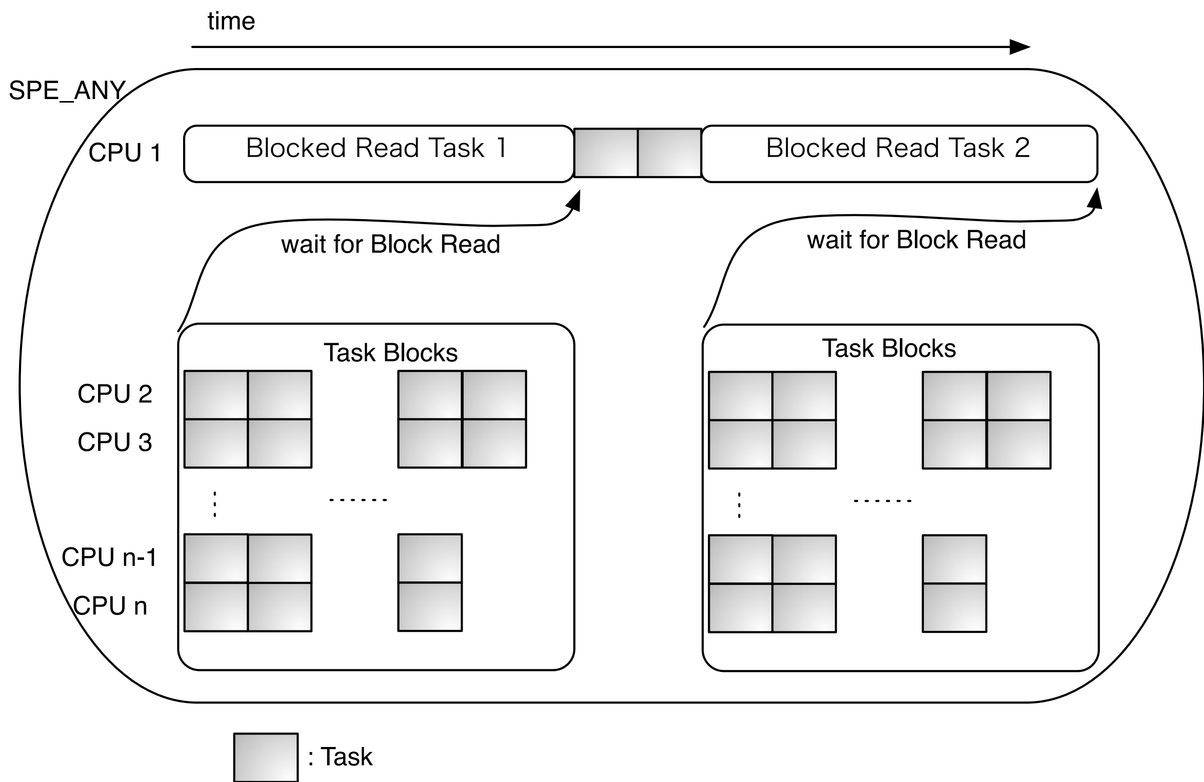


図 4.5: SPE_ANY での実装時の image

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)

・ Cerium の Task に CPU Type を設定することができる。しかし、同じ CPU Type を使用すると、IO を担当している CPU に Task が割り振られて、read 全体の速度が遅くなってしまう。・ Cerium では ptherad で並列処理を記述している

・ SPY_ANY という CPU Type は、Cerium 側が自動的に CPU 割り当てを行う便利なマクロ

・ SPE_ANY を使用すると、IO の部分にも割り込まれてしまうので、これをどうにかしたい。

・ IO_0 という新しい CPU Type を追加

・ pthread の API で CPU の priority をあげることができる。

図 4.6

・ これで IO 部分に割り込みがおこらないよね!!

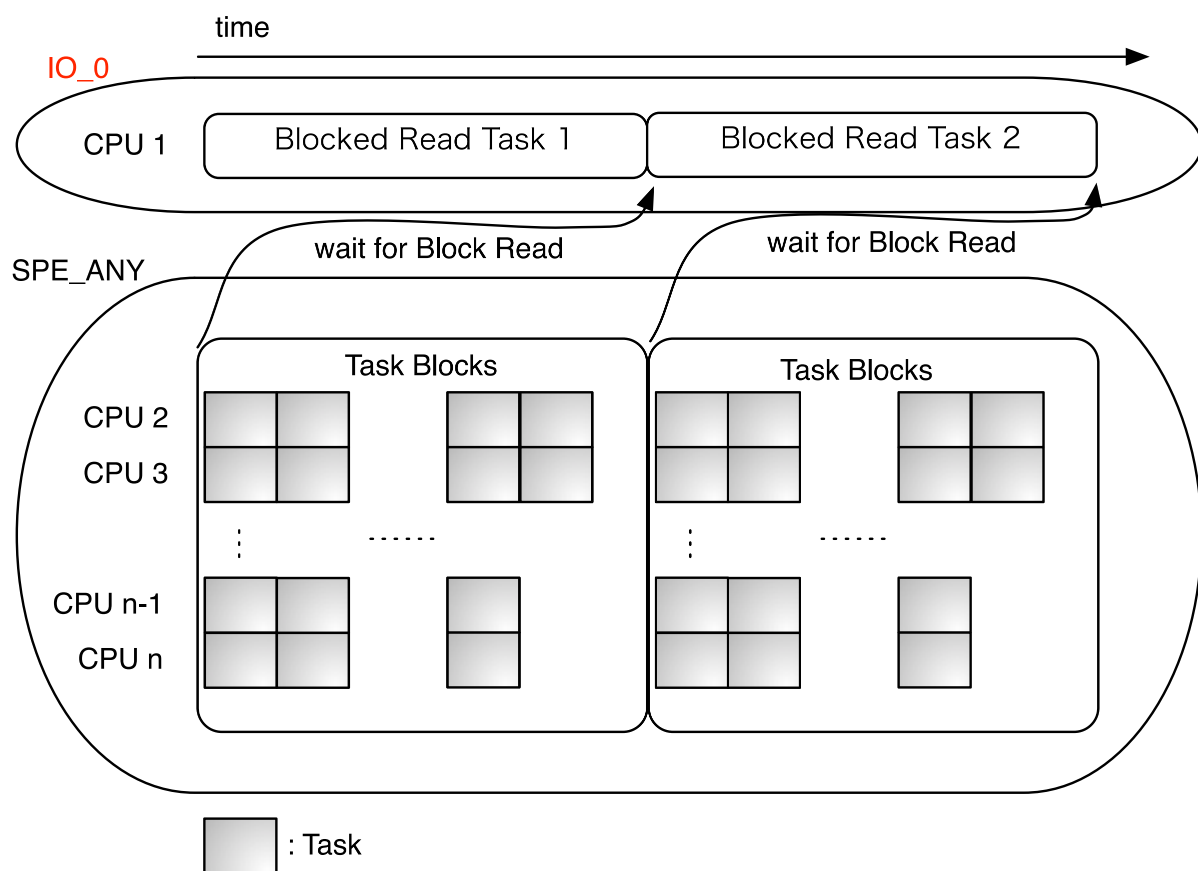


図 4.6: [image]priority を上げたときの image 図

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 約10 GB
BM search で文字列を検索

5.2　結果

・ mmap での計測
・ Blocked Read が SPE_ANY のとき
・ Blocked Read が IO_0 のとき
・初回実行時には Broked Read のほうが速くなったが、キャッシュに残らない
・サイズによってはキャッシュに入らないので、同じファイル2回検索するとそうとうなコストに
・ 2回目以降での測定 (ファイルがキャッシュにのこったときのもの)

5.3　考察

第6章　結論

6.1　まとめ

・ 本研究では、ファイル読み込みを含む並列処理の動作の改善をおこなった
・ ファイルを mmap で読み込むと、Task 1 つ 1 つを起動するごとに読み込み、Task の計算を動かすことになるので、IO の 回数がとても多くなる。
・ そこで、mmap を使用せずにファイルを読み込み、かつ、IO と Task が同時に動作するような処理を記述
・ Broked Read を実装して、IO と Task の分離
・ IO 専用の CPU Type を追加して、その CPU を使用することで、読み込みに専念させることができる。
・ その結果、初回での計測は 約 33 パーの動作改善につながり、I/O と mmap に並列度が加わった。

6.2　今後の課題

・ 実メモリ以上のファイルを取り扱えるようにする
・ 2 回目以降の実行時にも、キャッシュから読み込まれるようにする
・ Cerium の API としてまとめる

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 助教授に深く感謝いたします。

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