Cerium による並列処理向け I/O の実装

古波倉 正隆[†1]　河 野 真 治[†2]

当研究室では、Task 単位で記述するフレームワーク、Cerium の開発を行っている。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 Blocked Read で実装を行った。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。さらに、Cerium 側にて I/O 専用スレッドを実装した。それらの結果、mmap ではキャッシュに残った状態からの測定だと速く、Blocked Read ではファイル読み込みからの測定だと速くなった。

1. 研究背景と目的

当研究室では、Task 単位で記述する並列プログラミングフレームワーク、Cerium の開発を行っている。

ファイルの読み込み等の 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非常に重くなる場合が多い。マルチコアでの並列処理を行ったとしても、I/O の動作の負担が大きければ、I/O を含めたプログラムの処理は高速にならない。

従来の実装のように、ファイルを mmap や read で読み込んでから並列処理をさせると、読み込んでいる時間、他の CPU が動いていないので、並列度が下がってしまう。

本研究では、並列処理時におけるファイル読み込みをどのように実装すれば最高速に動作するかを考慮し、なおかつ読み込みとそれらに対する処理をプログラム作成者が自由に書けるように設計・実装を行った。Cerium の例題にある Word Count[1)] のファイル読み込み部分を様々な実装方法で測定を行い、その結果、個々の Task のサイズが大きければ後述する Blocked Read のほうが mmap よりも速度が出た。しかし、Task のサイズが小さいと Blocked Read と mmap はほとんど同じ速度を計測した。

2. Cerium TaskManager

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ある。

図 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 毎に依存関係を表す wait_i と wait_me というリストがあり、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。さらに、Scheduler に転送する際には TaskList に変換を行ってから各 Scheduler に転送される。

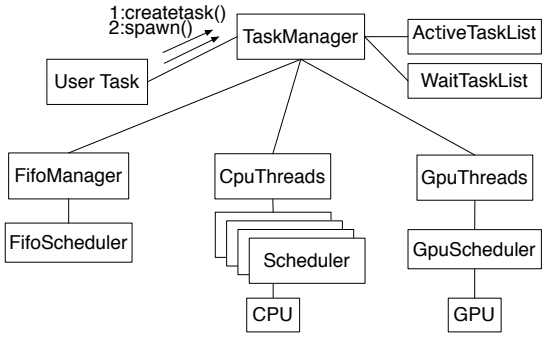

図 1　Cerium Task Manager

†1 琉球大学理工学研究科情報工学専攻
Interdisciplinary Infomation Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†2 琉球大学工学部情報工学科
Infomation Engineering, University of the Ryukyus.

2.1 Cerium Task Manager を使った例題

今回計測に使用した例題 WordCount を例にとり、以下に Task の生成部分を以下に示す。このプログラムは、WordCount Task と Print Task の 2 種類の Task から構成される。

input data とは、mmap や read で読み込んだファイルであり、このファイルを n KByte の大きさに分割して、WordCount Task にそれぞれ割り当てる。

WordCount Task は、input された data の単語数と行数をカウントし、それらを output に指定された data 領域に書きこむ。

以下に WordCount の Task 生成部分を示す。

```
exec = manager->create_task(TASK_EXEC);
exec->set_inData(0,
                file_mmap + i*division_size,
                size);
exec->set_outData(0,o_data + i*out_size,
                division_out_size);

exec->set_cpu(spe_cpu);
exec->spawn();
i++;
```

create_task	Task を生成する
set_inData	Task に入力データのアドレスを追加
set_outData	Task に出力データのアドレスを追加
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 1 Task 生成における API

set_inData には、WordCount Task が担当するテキストの範囲を設定し、set_outData にはその範囲の単語数や行数、範囲の先頭と末尾の状態を表すフラグを配列として格納してる。input data は分割されたデータが送られてくるため、分割された前後のテキストがどうなっているかはわからない。そのため、担当範囲であるデータの先頭と末尾のパラメータを単語数と行数の他に付け加える。後にそのデータを他の WordCount の結果と照らし合わせ、分割されたテキストを正しく整合する。

Task の記述は以下のようになる。

```
wordcount(SchedTask *s, void *rbuf, void *wbuf)
{
    char *i_data = (char *)s->get_input(0);
    unsigned long long *o_data =
        (unsigned long long *)s->get_output(0);
    unsigned long long *head_tail_flag =
                              o_data + 2;
    int length = (int)s->get_inputSize(0);
    head_tail_flag[0] =
        (i_data[0] != 0x20) &&
        (i_data[0] != 0x0A);

    word_num -= 1-head_tail_flag[0];

    for (; i < length; i++) {
        if (i_data[i] == 0x20) {
            word_flag = 1;
        } else if (i_data[i] == 0x0A) {
            line_num += 1;
            word_flag = 1;
        } else {
            word_num += word_flag;
            word_flag = 0;
        }
    }

    word_num += word_flag;
    head_tail_flag[1] =
        (i_data[i-1] != 0x20) &&
        (i_data[i-1] != 0x0A);

    o_data[0] = (unsigned long long)word_num;
    o_data[1] = (unsigned long long)line_num;

    return 0;
}
```

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

表 2 Task 側で使用する API

Print Task は WordCount Task によって書き出された単語数と行数を集計し、結果を出力する Task である。WordCount Task が全て走り終わったあとに、Print Task が走るように wait をかけている。(図 2)

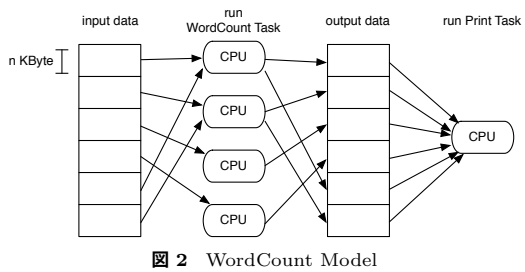

図 2 WordCount Model

3. 並列処理向け I/O の設計と実装

従来は mmap でファイルの読み込みの実装を行っていた。

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。ユーザメモリ空間にファイルのマッピングを行う UNIX のシステムコールであり、mmap した領域にアクセスされるときに、そのファイルが実メモリに展開される。

図 3 では、mmap で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。

1 個目の Task 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、その後 WordCount が行われる。そして Task 2 も同様に読み込みを行ってから WordCount 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。さらに、読み込みが 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できるようになれば高速で動くのではないかと考えた。

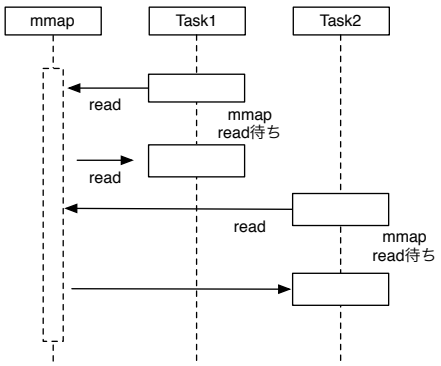

図 3 mmap Model

3.1 Blocked Read の設計と実装

Blocked Read とは、読み込みの Task と、WordCount を切り離すための実装方法である。それを実現するため、pread 関数にて実装した。pread 関数は、unistd.h に含まれている 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。(表 3)

従来の実装との違いは、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。Blocked Read は、読み込み専用の Read Task と、WordCount 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ってから読み込む。分割して読み込み終わったら、読み込んだ範囲内の WordCount が実行される。(図 5)

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 3 pread 関数の概要

Read Task が生成されて、その後 WordCount Task の実行となるので、Read Task は連続で走っている必要がある。

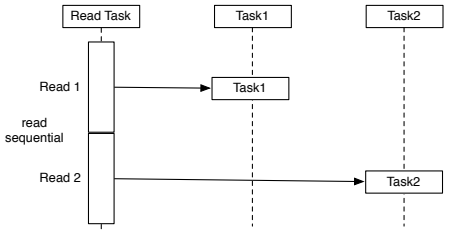

図 4 Read Task と WordCount の分離

この図では、Read Task 1 つに対して WordCount を 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 つ当たりがの 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。

Task Block が Blocked Read よりも先走ってしまうと、まだ読み込まれていない領域に対して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 Cerium の API である wait_for にて依存関係を設定する。(図 5)

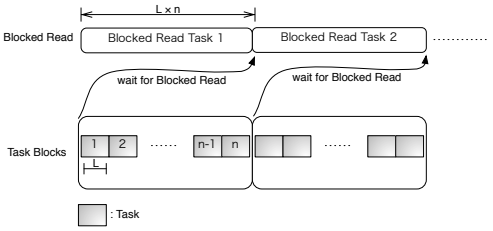

図 5 Blocked Read image

以下に、Blocked Read Task の生成部分を示す。

```
HTaskPtr t_read =
        manager->create_task(READ_TASK);
t_read->set_cpu(DEVICE_TYPE);
t_read->set_outData(0,
        file_mmap + task_num * division_size,
        task_blocks * division_size);
t_read->set_param(0,fd);
t_read->set_param(1,task_num*division_size);

run_tasks(manager,w, task_blocks, … );

t_read->set_param(2,task_num*division_size);
t_read->spawn();
```

set_cpu にて Read Task を担当するデバイスの設定を行う。set_outData(0) にファイルを読み込んだときの格納場所を指定し、set_param(0) にて読み込むファイルディスクリプタを設定している。set_param(1) 、set_param(2) にて Blocked Read Task 単体で読み込むファイルの範囲の先頭と末尾のポジションを設定する。

なお、run_tasks 内部で、task が生成されるたびに task_num のインクリメントを行っている。

Blocked Read Task の記述は以下のようになる。

```
static int
read_task(SchedTask *s, void *rbuf, void *wbuf)
{
    long fd = (long)s->get_param(0);
    long start_read_position =
            (long)s->get_param(1);
    long end_read_position =
            (long)s->get_param(2);
    char *read_text =
            (char*)s->get_output(wbuf,0);
    long read_size = end_read_position -
                     start_read_position;

    pread(fd, read_text,
          read_size , start_read_position);
    return 0;
}
```

Blocked Read Task の生成部分で設定したパラメータをそれぞれ受け取る。ファイル読み込みの先頭と末尾のポジションが渡されているので、どれだけファイルを読みこめばいいか求めることができる。

それらのパラメータを使用して、pread 関数に渡すことで Blocked Read によるファイル読み込みを実現している。

3.2 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。SPE_ANY という設定を使用すると、Task Manager で CPU の割り振りを自動的に行う。Blocked Read 、Task それぞれに SPE_ANY にてデバイスの設定を行うと、Task Manager 側で自動的に CPU を割り当てられ、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 6)

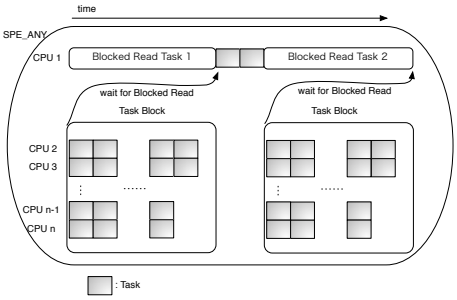

図 6 SPE_ANY での実装時

この問題を解決するため、Task Manager に新しく I/O 専用の thread 、 IO_0 の追加を行った。

IO_0 は、SPE_ANY よりも priority を高く設定しているので、IO_0 で設定された 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。Cerium では、並列処理を pthread で記述しており、pthread_getschedparam() で IO_0 の priority を設定することができる。(図 7)

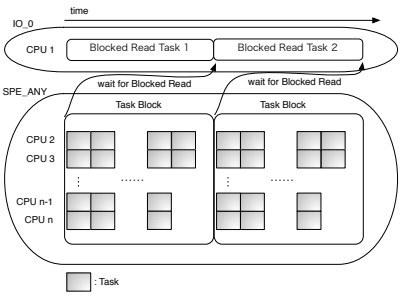

図 7 Blocked Read Task を IO_0 での実装時

4. Benchmark

例題で紹介した Word Count に Blocked Read を組み込み、1 GB のファイルで計測を行った。

実験環境

- Mac OS X 10.9.1
- 2*2.66 GHz 6-Core Intel Xeon
- Memory 16GB 1333MHz DDR3
- HHD 1TB
- CPU num 12

4.1 結　　果

以下の表に読み込み時間を含めた場合の実行結果を示す。

読み込み方法	実行速度 (s)
mmap	15.875
mmap (CPU num = 1)	15.294
一括 Read	12.520
一括 Read (CPU num = 1)	18.758
Blocked Read & SPE_ANY	14.028
Blocked Read & IO_0	10.295

表 4　読み込みを含めた実行結果

また、キャッシュに入った場合での実行結果を以下に示す。

読み込み方法	実行速度 (s)
mmap	0.878
一括 Read	1.469
Blocked Read & IO_0	0.866

表 5　キャッシュに入った時の実行結果

読み込みを含めた場合の実験結果より、Blocked Read & IO_0 の実行速度が mmap と比較して 1.55 倍向上した。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 1.36 倍向上した。Blocked Read で WordCount Task と Read Task を分離させた。Blocked Read Task がファイル読み込みを行っている間も、WordCount Task は読み込んだ範囲に対して並列処理を行うことができた。その結果、mmap より速く WordCount 処理を行うことができたと考える。

キャッシュに入った時は、mmap のほうが一括 Read と比較して 1.67 倍速くなる。そして、mmap と Blocked Read と mmap は、ほとんど同じ実行速度となった。

5. まとめと今後の課題

本研究では、Task と 読み込みが並列に動作するように Blocked Read の実装を行った。またそれだけだと、Blocked Read に Task が割り込まれるので、I/O 専用 thread の追加を行った。Blocked Read に I/O 専用 thread を割り当てると、さらに速くなった。

I/O が含まれるときの並列処理は、I/O のコントロールをプログラマが実装することで動作改善に繋がる。

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。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。それに対して、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たが、なぜこのようなことが発生するのか調査する。

さらに、pread による複数 read を実装したが、複数 mmap に関してはまだ実装・計測を行っていない。これらの計測を行って、どちらが最高速に動作するかどうか調べる必要がある。

また、Blocked Read のコードを記述するのは煩雑で、これらを毎回記述することは大変である。これを Cerium の API として落としこむことによって、簡単に利用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 TaskManager 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) (2011).
2) 渡真利勇飛, 河野真治：Cerium Task Manager の GPGPU への対応, 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 (2014).
3) 當眞大千, 河野真治：Cerium Task Manager におけるマルチコア上での並列実行機構の実装, 第 53 回プログラミング・シンポジウム (2012).