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。

Implement asynchronous read of Cerium

Masataka KOHAGURA[†1] and Shinji KONO [†2]

We are developing a Parallel task manager Cerium. With I/O, special care is necessary to work with parallel processing. It is easy to use mmap system call to read from file parallelly, but current implementation of mmap sometimes does work well. So we implement asynchronous read thread with high priority. If the file is in a kernel file system cache, mmap and asynchronous read has no difference. In real read situation, asynchronous read sometimes gives good result on word count example. We gives the result and analysis.

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。

†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. 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 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。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 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。

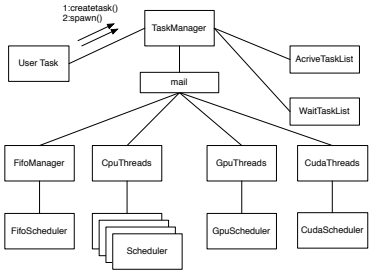


図 1　Cerium Task Manager

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 の input data には、WordCount Task が担当するテキストの範囲を設定し、output data には単語数や行数、そしてその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する。

以下に WordCount Task の生成部分を示す。

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

wc->set_cpu(spe_cpu);
wc->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 する

表 1　Task 生成における API

状態を表すフラグを格納している理由は、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。空白か改行が読み込まれたときに単語として認識するので、単語の終わりで分割されたときにその単語がカウントされない。

単語数 3 の例文「I'm so happy.」の o 直後に分割される場合を図 2 に示す。

wc Task 1 は “I'm” “so” の 単語に見えるが、“so” は単語の途中で分割された可能性がある。1 つの単語かどうかわからないので、wc Task 1 の単語数は 1 となる。

wc Task 2 は “happy.” だけなので、単語数は 1 となり、 合計単語数が 2 となってしまう。

wc Task 1 の “so” を単語として認識させるには、wc Task 2 の先頭が空白か改行である必要がある。しかし、wc Task 内では隣接した Task の情報がわからないので、各 Task の先頭と末尾の状態を持たせて、最後の集計のときに整合させる必要がある。

このように単語の終わりでファイル分割されたとき、その分割ファイルの末尾が空白か改行以外の文字列であり、そして、隣接した分割ファイルの先頭が空白か改行であれば、単語として認識される。

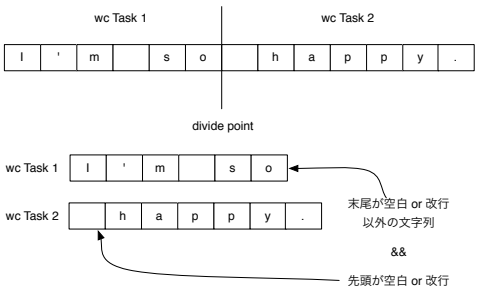


図 2　単語の終わりで分割されたテキスト

そのため、出力結果に分割ファイルの先頭と末尾の状態のフラグを head_tail_flag として単語数と行数に付け加える。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。

WordCount 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);
    int word_flag = 0;
    int word_num = 0;
    int line_num = 0;
    int i = 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

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、行数、`head_tail_flag` 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 が走るように依存関係を設定している。(図 3)

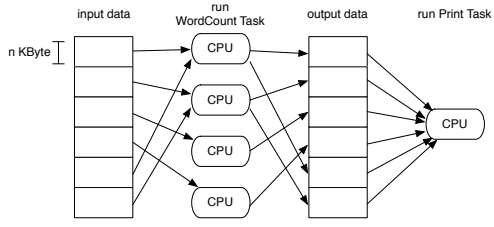


図 **3** WordCount Model

WordCount Task と Print Task の生成と依存関係の設定を以下に示す。

```
static void
run_start(TaskManager *m, char *filename)
{
    HTaskPtr print;
    WordCountPtr w;

    print = m->create_task(TASK_PRINT,
        (memaddr)&w->self,sizeof(memaddr),0,0);

    w->t_print = print;
    exec_num = 4;

    for(int i=0;i<exec_num;i++) {
        wcb = m->create_task(RUN_WORDCOUNT_BLOCKS,
        (memaddr)&w->self,sizeof(memaddr),0,0);

        print->wait_for(wcb);
        wcb->spawn();
    }
    print->spawn();
}
```

`WordCountPtr w` は、ファイルサイズや読み込んだファイルの先頭アドレスなど、Task 間で共有する情報をまとめた構造体である。`RUN_WORDCOUNT_BLOCKS` は WordCount Task をある程度まとめた単位で生成しているが、3.1 Blocked Read の設計と実装にて後述する。

`create_task`	Task を生成する。`w->self` は input data、`sizeof(memaddr)` は input data のサイズ
`wait_for`	Task の依存関係を設定する。ここでは、`print` が `wcb` を待つように設定

表 **3** run_start で使用した API

WordCount Task がデータを集計し終わったあとに、 Print Task にて最終集計を行う。

Print Task の記述を以下に示す。

```
static int
run_print(SchedTask *s, void *rbuf, void *wbuf)
{
    WordCount *w = (WordCount*)s->get_input(0);
    unsigned long long *idata = w->o_data;
    long status_num = w->status_num;
    int out_task_num = w->out_task_num;

    unsigned long long word_data[2];

    int flag_cal_sum = 0;
```

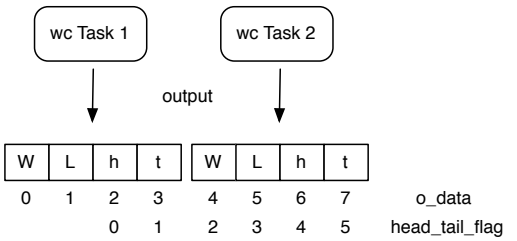
```
    for (int i = 0; i < out_task_num ; i++) {
        word_data[0] += idata[i*w->out_size+0];
        word_data[1] += idata[i*w->out_size+1];
        unsigned long long *head_tail_flag =
            &idata[i*w->out_size+2];
        if((i!=out_task_num-1)&&
           (head_tail_flag[1] == 1) &&
           (head_tail_flag[4] == 0)) {
            flag_cal_sum++;
        }
    }

    word_data[0] += flag_cal_sum;

    for (int i = status_num-1; i >=0; i--) {
        s->printf("%llu ",word_data[i]);
    }
    return 0;
}
```

word_count[0] にそれぞれの単語数の合計を格納して、word_count[1] にそれぞれの行数の合計を格納する。WordCount Task で集計したデータは配列として保存され、それらの配列を 1 つの大きな配列 w->o_data として Print Task に渡している。

また、o_data[2] を head_tail_flag[0] に置換して、末尾と先頭の処理を行う。(図 4) 末尾に文字列があり、先頭が空白か改行の場合は単語がカウントされないので、この状態を見て単語数を整合させる。



W : 単語数
L : 行数
h : 先頭の状態
t: 末尾の状態

図 4 集計したデータの配列図

3. 並列処理向け I/O の設計と実装

I/O を平行に行うには、従来の read loop を行うのでは read がプログラム全体を止めてしまう。もっとも簡単に read を並行して行うには、file open の代わりに mmap を使う方法がある。mmap は直ぐにファイルを読みに行くのではなく、仮想メモリ空間にファイルの中身を対応させる。そのメモリ空間にアクセスに行くと、それに対応した部分のファイルを OS が見に行く。ファイルが読み込まれるまでは、アクセスしたスレッド/プロセスは、その場で待たされる。しかし、mmap は逐次アクセスを仮定しているので、OS 内部で自動的にファイルの先読みを行うと期待される。

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。mmap の read は、Task と並列に実行されるべきであるが、それは OS の実装に依存する。実際、mmap のフラグのほとんどは実装されていない。読み込みの実行が、うまく並列に実行されなければ、Task が読み込み待ちを起こしてしまう。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、mmap では細かく制御することができない。

そこで、mmap を使わずに read を独立したスレッドで行い、読み込まれた部分に対して、並列タスクを起動する方法があるが、プログラミングは煩雑になる。今回は、この部分を実装し、mmap に対して、どれだけ性能が向上するかを調べる

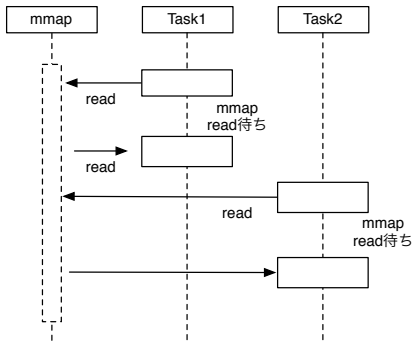


図 5 mmap Model

3.1 Blocked Read の設計と実装

Blocked Read とは、読み込みの Task と、それらに対して何らかの処理を行う Task 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 4)

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

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　pread 関数の概要

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 7)

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。

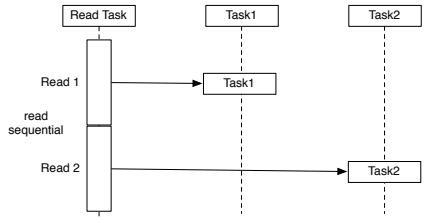


図 6　Read Task と WordCount の分離

図 7 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。

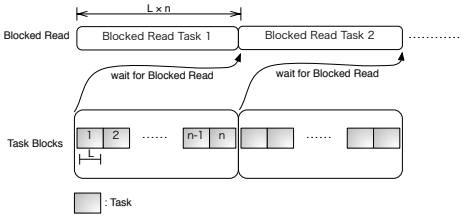


図 7　Blocked Read image

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。

Cerium の Task は、最初に WordCount に必要な Task を全部起動してしまうと、その Task を表すデータ構造自体がメモリを消費してしまう。そこで、既にある程度の量の Task を起動し、それが終了してから (正確には、終了する前に) 次の Task を生成するようになっている。これが run_tasks である。そこの部分にファイルを読み込む Task との待ち合わせを入れれば良いので、実装は比較的容易にできる。

run_tasks の中で、READ_TASK に対する待ち合わせを wair_for() を使って実現している。

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 8)

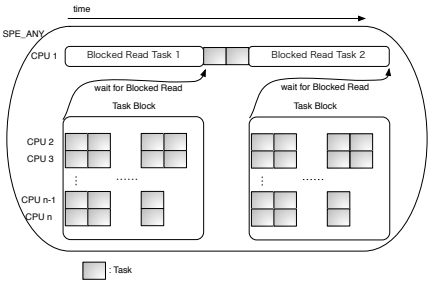


図 **8** SPE_ANY での実装時

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` とは、別なスレッド動いているスケジューラーで動くので、`SPE_ANY` で動いている WordCount に割り込まれることはない。しかし、読み込みが完了した時に、その完了を通知し、次の read を行う時に、他の計算スレッドにスレッドレベルで割り込まれてしまうと、全体の計算が待たされてしまう。そこで、 `pthread_getschedparam()` で `IO_0` の priority を設定している。I/O thread は計算はほとんどしないので、高い優先順位を割り当てても他の計算を行うスレッドには影響しない。I/O を含む処理では I/O を行うスレッドの優先順位を高くする必要があるということである。(図 9)

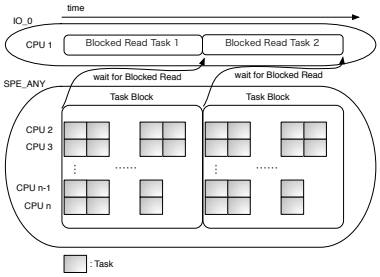


図 **9** Blocked Read Task を IO_0 での実装時

4. Benchmark

例題で紹介した WordCount 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
- WordCount 1 つの読み込み量 (divide size)128KB
- ブロック数 48
- filesize 1GB

4.1 結　　果

CPU 数を変化させたときの結果を以下に示す。以下の数値の単位は全て秒である。

読み込み方法	CPU 1	CPU 4	CPU 8	CPU 12
mmap	20.179	22.861	22.789	22.713
read	21.351	15.737	14.785	12.520
bread & SPE_ANY	18.531	15.646	15.287	14.028
bread & IO_0	13.930	14.634	14.774	10.295

表 **5** 読み込みを含めた実行結果 (divide size 128KB)

WordCount 1 つの読み込み量だけ変更してみた。16KB に変更して、 CPU 数を変化させたときを計測してみた。結果は以下に示す。

読み込み方法	CPU 1	CPU 4	CPU 8	CPU 12
mmap	15.353	11.287	11.707	11.137
read	16.846	11.730	11.487	11.437
bread & SPE_ANY	13.297	11.984	10.887	11.146
bread & IO_0	11.503	11.437	11.365	11.412

表 **6** 読み込みを含めた実行結果 (divide size 16KB)

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
Read	1.469
Blocked Read & IO_0	0.866

表 7　キャッシュに入った時の実行結果 (divide size 128KB)

divide size が 128KB のときの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。

divide size が 16 KB のとき、128KB よりも全体的に速くなり、CPU 12 のときはほとんど同じ結果を示した。また、CPU の数を 4 以上に設定してもほとんど変化が見られなかった。

これより、読み込みを様々な実装で試してみたが、最適な実装を行えば mmap でも充分に速くなる。さらに、Blocked Read のような実装を行うと、それ以上に速く動作させることができる。

キャッシュに入った時は、mmap のほうが 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).