アスキー形式のデータ読み込みを速くする方法について

2012.11.05 武田隆顕

概要

シミュレーションデータを読み書きする場合は、バイナリ形式で読み書きすると効率的だと知られているが、簡便性などからアスキー形式での読み書きも良く行われる。アスキーデータの文字の羅列としてのデータファイルを読み込む作業は、バイナリーのファイルを直接読んでメモリーにコピーする作業と比較してずいぶんと遅いため、速くアスキーデーターを読み込みたいという需要が存在する。ここでは、文字として数字が記述されたファイルを読んで、メモリー上の配列に数値としてデータを配置するまでを高速化する方法を検討する。

結論としては、大きなデータを読み込む際にファイル読み込みをブロックごとに分け、次のブロックを読み込んでいる間に以前に読み込んだデータの数値化を別スレッドで行うことで、高速化が行える。また、既にキャッシュ上にファイルのデータが存在する場合には、HDD 等からの読み込みに律速されないために、数値化を複数のスレッドで同時に行うことで、さらなる高速化が可能である。

セクション 1 では、数字から文字列への変換速度の測定と、とくに使われると思われる `sscanf` と `atof` の速度の比較を比較した結果を述べる。セクション 2 では、データの読み込み部分と数値変換部分をスレッド別に処理した場合の処理速度の測定を行い、セクション 3 では、数値変換部分をさらに複数のスレッドで同時に処理した場合の結果を述べる

1. コンバート速度の計測 (atof と sscanf の比較)

まず、数字データから数値への変換にどれぐらいの時間がかかるのか、メモリ上に書かれた文字列から数値への変換を繰り返して行い実行時間の計測を行った。想定しているデータは、N 体シミュレーションなどの結果で、多数の粒子の xyz 情報を読み込みたいものとする。今回は `float` 型で xyz の位置情報を持つ、約一千万個の粒子を模したプログラムになっている。バイナリーなら 120MB 相当。アスキーの場合は、桁数や表記によって違いは生じるが、300MB 程のデータに相当すると思われる。ファイル I/O にかかる時間を無視するために、テキスト情報はあらかじめメモリー上に展開されているとして、アスキー情報を `float` 型に変換する作業のみ計測している。与えたデータ自体はあまり意味のない適当な値の繰り返しになっている。本来は桁数が予想できないので、バッファーサイズなどを大きめに取るべきだが、そうしたマージンは考慮していない。

以下すべてのセクションで、環境はCore i7 975(3.33GHz) で、OSはWindows7。cygwin上の gcc 4.5.3 で -O3 付きコンパイルをして実行している。このセクションでの値は3回実行した実行時間の平均値を出している。

1.1. atof のみを利用する場合

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// 1024*1024 = 1048576 data size 120MB
#define PNUM 10485760
int main(int argc, char* argv[]){
    static char  bufx[PNUM][10], bufy[PNUM][10], bufz[PNUM][10];
    static float valuexyz[PNUM][3];
    for (int i = 0; i < PNUM; i++){
        sprintf(bufx[i], "1.337e-4");
        sprintf(bufy[i], "12.057732");
        sprintf(bufz[i], "3.273112");
    }
    clock_t t1, t2;
    t1 = clock();
    printf("10 million xyz info::atof converting start\n");
    for (int i = 0; i < PNUM; i++){
        valuexyz[i][0] = atof(bufx[i]);
        valuexyz[i][1] = atof(bufy[i]);
        valuexyz[i][2] = atof(bufz[i]);
    }
    t2 = clock();
    printf("time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);
    return 0;
}
```

結果：3.5673秒

ディスクからの読み込み効率を100MB/秒として、読み込み時間も3秒程度だと考えると、読み込みと同程度の時間がデータの変換にかかっている。

1.2. sscanf を利用する場合

xyz 成分ごとに一行で書かれているデータを読みこむ場合には、sscanf を使うことが多いと思われる。sscanf だと空白文字の判定などが必要になるため、遅くなると考えられる。実際に使っている文字列は最後の'¥0' を含めて 28 文字だが、一行のバッファサイズ (30) は、エラーにならない切の良い数として適当に与えている。データ量として 300MB 相当。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// 1024*1024 = 1048576 data size 120MB
#define PNUM 10485760
int main(int argc, char* argv[]){
    static char bufxyz[PNUM][30];
    static float valuexyz[PNUM][3];
    for (int i = 0; i < PNUM; i++){
        sprintf(bufxyz[i], "1.337e-4 0.057732 3.273112");
    }
    clock_t t1, t2;
    t1 = clock();
    printf("10 million xyz info::sscanf converting start\n");
    for (int i = 0; i < PNUM; i++){
        sscanf(bufxyz[i], "%f %f %f",
               &valuexyz[i][0], &valuexyz[i][1], &valuexyz[i][2]);
    }
    t2 = clock();
    printf("time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);
    return 0;
}
```

結果：6.999 秒

atof のみを使った場合に比べておよそ倍遅くなったので、速度最優先であればできれば sscanf は使わない方が良い。

1.3. sscanf と改行判定

実際のアスキーデータは、粒子ごとに改行されていることが多いと思われる。ファイルから直接一行読み込む時などは fgets() 関数を使うと思われるが、今回は fgets 関数が速いか遅いかに関しては無視して、大きな文字列データから自分で改行記号を探し、順次 1 行分の情報を取り出すようにして実験している。Windows を想定して、改行記号は¥r¥n。一行あたりに利用する文字数はちょうど 30。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
// 1024*1024 = 1048576 data size 120MB
#define PNUM 10485760
int main(int argc, char* argv[]){
    static char bufxyz[PNUM*30+1];
    static float valuexyz[PNUM][3];
    char *ptr = bufxyz;
    for (int i = 0; i < PNUM; i++){
        ptr += sprintf(ptr, "1.337e-4 12.057732 3.273112\r\n");
    }
    clock_t t1, t2;
    t1 = clock();
    printf("10 million xyz info::sscanf converting start\n");
    ptr = bufxyz;
    for (int i = 0; i < PNUM; i++){
        char buf[30];
        memcpy(buf, ptr, 30);
        sscanf(buf, "%f %f %f",
               &valuexyz[i][0], &valuexyz[i][1], &valuexyz[i][2]);
        while (*ptr != '\n') ptr++;
        ptr++;
    }
    t2 = clock();
    printf("time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);
    return 0;
}
```

結果：5.823 秒

単純に sscanf を使った場合よりなぜか少し速かった。配列の位置を探す時間がいらなかったりとか、バッファーがキャッシュの上にあって効率が良いとか、そうした些細な違いが効いているのか？

1.4. atof と改行判定

xyz 成分ごとに一行で書かれているデータでも、atof を使った方が速いので、改行や空白文字の判定を自分で行っても sscanf より効率が良いかもしれない。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
// 1024*1024 = 1048576 data size 120MB
#define PNUM 10485760
int main(int argc, char* argv[]){
    static char bufxyz[PNUM*35];
    static float valuexyz[PNUM][3];
    char *ptr = bufxyz;
    for (int i = 0; i < PNUM; i++){
        ptr += sprintf(ptr, "1.337e-4 12.057732 3.273112\r\n");
    }
    clock_t t1, t2;
    t1 = clock();
    printf("10 million xyz info::sscanf converting start\n");
    ptr = bufxyz;
    for (int i = 0; i < PNUM; i++){
        valuexyz[i][0] = atof(ptr);
        while (*ptr != ' ') ptr++;
        ptr++;
        valuexyz[i][1] = atof(ptr);
        while (*ptr != ' ') ptr++;
        ptr++;
        valuexyz[i][2] = atof(ptr);
        while (*ptr != '\n') ptr++;
```

```
        ptr++;
    }
    t2 = clock();
    printf("time = %f\n", (double)(t2 - t1) / CLOCKS_PER_SEC);
    return 0;
}
```

結果：3.713 秒

速度が優先で、フォーマットが決まっているのであれば sscanf よりも atof の方が速いようなので、自分で空白文字などを判定しながらでも atof を使った方が速度が出る。

1.5. 16 進数アスキーを利用した場合

アスキー表記を行った場合でも 16 進数で浮動小数点定数を記述すると、10 進数から 2 進数への変換で生じる誤差の問題が解決する。また、アルゴリズム上も 10 進数からの変換が必要ないので速くなるかもしれない。16 進数表記した場合での変換速度の比較を行った。以下の 16 進数表記は、printf 構文の%a 表記で得ている。ただし、全ての精度を%a 表記すると倍精度表記になり文字数が相当大きくなるので、できるだけ文字数が少なくなるように桁数を落としている。前のサブセクション (atof と改行判定付) のプログラムの

```
ptr += sprintf(ptr, "1.337e-4 12.057732 3.273112\r\n");
```

の部分を

```
ptr += sprintf(ptr, "0x1.18p-13 0x1.819p+3 0x1.a2fp+1\r\n");
```

と変形し、それに応じてバッファーサイズも大きしくている。atof 変換は、16 進数表記にも対応しているのでそのまま使っている。

結果：4.596 秒

atof の内部での処理方法を調べていないので原因は分からないが、残念なことに 10 進数表記の場合よりも速度は低下してしまった。

2. データ読み込みと数値への変換を別スレッドで行う。

メモリ上にある文字列から数値に変換する作業について、計測した結果では sscanf よりも atof の方が速く、フォーマットが決まっているのであれば自分で空白文字などを判定

しながらでも atof を使った方が速いことが分かった。

テストを行った環境では HDD からアスキーデータを読み込む速度と、atof を使ってそのデータを数値化する速度は同じ程度であった。そのため、この 2 つの作業を同時に行えれば倍程度の高速化が期待できる。

1. ファイルを読んで、ある大きさのブロックごとにメモリーに展開する
2. それを別のスレッドで数値データに変換して、配列としてメモリー上に配置する。

という作業をスレッド別に行うことにより、2 の作業を行っている間に次のブロックに対する 1 の作業を平行して行い、高速化するルーチンを作成する。

もしキャッシュメモリ上にデータがある場合は、HDD からの読み込みにかかる時間は無視できるようになる。その際にはスレッドごとに行っている数値化を、複数のスレッドで同時に行えば、さらに高速化が期待できる。数値化をマルチスレッドで高速化することは、次セクションで行う。

2.1. 結果

1. ファイルを読んで、ある大きさのブロックごとにメモリーに展開する
2. それを別のスレッドで数値データに変換して、配列としてメモリー上に配置する。

の手順において、2 の作業が終わるまで 1 の作業を待つ場合を SERIAL、2 の作業と並行して次のブロックの 1 の作業を開始する場合を PARALLEL とする。比較として、2 の作業をコメントアウトした純粋なデータ読み込みのみの時間も計測した。

読み込むファイルは 2000 万体分の id,x,y,z,vx,vy,vz についてのアスキーデータで、1.2GB。時間計測は clock() と、time() で行った。clock() は cpu 時間を返し time() は実時間を返すので time() の方が待ち時間の計測には良いが、1 秒単位でしか値が得られない。Table 1, 2 にキャッシュ上にファイルのデータがある場合、無い場合についてのそれぞれの結果を示している。

結果として、キャッシュメモリにデータが無い場合、読み込みと解釈をパラレルに行うことで、確かに高速化が期待できる。既にファイルデータがキャッシュに乗っている場合は、データ読み込みの速度は無視できるので、データの解釈部分をさらに複数のスレッドに分けることによって、さらなる高速化が期待できる。

-	clock	time
データ読み込みのみ	1.2 秒	12 秒
SRIAL	17.9 秒	30 秒
PARALLEL	15.6 秒	16 秒

Table 1: キャッシュメモリ上にデータ無しの場合 (起動後初めての実行)

-	clock	time
データ読み込みのみ	0.7 秒	1 秒
SRIAL	15.5 秒	16 秒
PARALLEL	15.5 秒	15 秒

Table 2: キャッシュメモリ上にデータ有りの場合 (2 回目以降)

2.2. サンプルプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

#ifndef TRUE
#define TRUE  true
#define FALSE false
#endif

#define _PARALLEL 1
//#define _SKIP_ATOF
#define _SIZE_MB_ 10

void *ThreadFunc(void *arg);

int  ReadFile2Memory(FILE **pfp, char **ppAllBuffer);

int  ReadFile2Memory(FILE **pfp, long endPoint, long startPoint, long offset,
                     char **ppAllBuffer){

    bool bFinal = FALSE;
    if (endPoint - startPoint > 1024*1024*_SIZE_MB_)
        endPoint = startPoint + 1024*1024*_SIZE_MB_;
    else
        bFinal = TRUE;

    int contentsSize = endPoint - startPoint;

    fread(*ppAllBuffer, sizeof(char), contentsSize, *pfp);

    // 後ろから改行記号を探す
    int i;
    for (i = contentsSize; i > 0; i--){
        if ((*ppAllBuffer)[i] == '\n') break;
    }

    if (bFinal == TRUE){
        fclose(*pfp);
        return -1;
    }
    else
        fseek (*pfp, startPoint+i+offset, SEEK_SET);
    return startPoint + i + 1;
```

```
}
struct structInformation{
    int particleIndex;
    char  *pBuffer;
    long   endPoint;
    float *pX;
    float *pY;
    float *pZ;
    float *pVX;
    float *pVY;
    float *pVZ;
    int particleNumber;
};
int main(int argc, char* argv[]){

    FILE *fp;
    fp = fopen(argv[1], "rb");

    int PNum;
    fscanf(fp, "%d\n", &PNum);

    float *pX = new float [PNum];
    float *pY = new float [PNum];
    float *pZ = new float [PNum];
    float *pVX = new float [PNum];
    float *pVY = new float [PNum];
    float *pVZ = new float [PNum];

    // データ本体部分のサイズを調査
    fpos_t fsize_struct_start = 0;
    fpos_t fsize_struct_end = 0;
    fgetpos(fp, &fsize_struct_start);
    fseek  (fp, 0, SEEK_END);
    fgetpos(fp, &fsize_struct_end);
    long fsize = fsize_struct_end - fsize_struct_start;

    fseek  (fp, fsize_struct_start, SEEK_SET);

    time_t timeStart, timeEnd;
    clock_t start, end;
    start = clock();
    time (&timeStart);

    int   particleIndex = 0;
    long  startPoint = 0;
    char *pAllBuffer = NULL;
    while(TRUE){
        int  nextBufferSize;
        bool bThread = FALSE;
        pthread_t Cthread;
        structInformation structInfo;
        if (pAllBuffer != NULL){
            //スレッドの宣言
            structInfo.particleIndex = particleIndex;
            structInfo.pBuffer  = pAllBuffer;
            structInfo.endPoint = nextBufferSize;
            structInfo.pX = pX;
            structInfo.pY = pY;
            structInfo.pZ = pZ;
```

```
            structInfo.pVX = pVX;
            structInfo.pVY = pVY;
            structInfo.pVZ = pVZ;
            pthread_create(&Cthread, NULL, ThreadFunc, &structInfo);
            bThread = TRUE;
        }

        if (startPoint == -1){
            pthread_join(Cthread, NULL);
            particleIndex += structInfo.particleNumber;
            break;
        }

#ifdef _SERIAL
        if (bThread == TRUE){
            //threadの終了を待つ
            pthread_join(Cthread, NULL);
            particleIndex += structInfo.particleNumber;
        }
#endif

        pAllBuffer = new char [1024*1024*_SIZE_MB_];
        long startPointOrg = startPoint;
        startPoint = ReadFile2Memory(&fp, fsize, startPoint, fsize_struct_start,
                                     &pAllBuffer);
        if (startPoint == -1)
            nextBufferSize = fsize - startPointOrg;
        else
            nextBufferSize = startPoint - startPointOrg;

#ifdef _PARALLEL
        if (bThread == TRUE){
            //threadの終了を待つ
            pthread_join(Cthread, NULL);
            particleIndex += structInfo.particleNumber;
        }
#endif
    }

    fclose(fp);

#ifdef _WRITE_COPY
    fp = fopen("sampleData_Copy.txt", "wt");
    fprintf(fp, "%d\n", PNum);

    for (int i = 0; i < PNum; i++)
        fprintf(fp, "%d %f %f %f %f %f %f\n", i,
                pX[i], pY[i], pZ[i],
                pVX[i], pVY[i], pVZ[i]);

    fclose(fp);
#endif

    end = clock();
    time(&timeEnd);

    printf("Finished time=%f!!\n", (double)(end-start)/CLOCKS_PER_SEC);
    printf("Start Time: %s\n", ctime(&timeStart));
    printf("End   Time: %s\n", ctime(&timeEnd));

    return 1;
}
```

```
void *ThreadFunc(void *arg){
    //新しく作成した子スレッドの処理
    structInformation *pStruct = (structInformation*)arg;
    int    particleIndex = pStruct->particleIndex;
    char  *pBuffer  = pStruct->pBuffer;
    int    endPoint = pStruct->endPoint;
    float *pX = pStruct->pX;
    float *pY = pStruct->pY;
    float *pZ = pStruct->pZ;
    float *pVX = pStruct->pVX;
    float *pVY = pStruct->pVY;
    float *pVZ = pStruct->pVZ;
    int particleNumber = 0;
    char  *ptr = pBuffer;
    int i;
#ifdef _SKIP_ATOF
#else
    while (TRUE){
        i = atoi(ptr);
        int index = particleIndex + particleNumber;
        while(*ptr != ' ') ptr++;
        ptr++;
        pX[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pY[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pZ[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pVX[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pVY[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pVZ[index] = atof(ptr);
        while(*ptr != '\n') ptr++;
        ptr++;
        particleNumber++;
        if (ptr >= pBuffer+endPoint) break;
    }
#endif
    pStruct->particleNumber = particleNumber;
    pthread_exit(NULL); //スレッド終了
}
```

2.2.1. サンプルプログラム中の関数の解説

```
    int ReadFile2Memory(......)
```

ファイルを一定サイズ読んで、バッファーにアスキーデータをコピーする。通常は行の合間でデータが切れるために、後ろから改行記号を探して FILE **pfp の位置をそこに持って行き、次のファイル読み込みに備える。最後までデータを読んだなら-1 を返し、そうでなければ次の読み込みの位置を数値で返す。サイズの計算をしたいので、FILE 構造体の位置を変える以外にも数字を返している。

struct structInformation

スレッドに渡す構造体。アスキーデータの保存してあるバッファーや、数値化した結果を書き込むべき配列のアドレスなどを保持する。

int main(....)

１．バッファー上にあるアスキーデータを数値化するスレッドを作り実行。２．その間に別のバッファーにファイルから次のブロックのアスキーデータ読み込む。３．数値化スレッドの終了を待つ。というループを、終了まで繰り返す。

void *ThreadFunc(...)

数値化するスレッドで実行する関数。渡された構造体を読んで、アスキーデータのバッファーのアドレス、結果をしまうべき配列のアドレスを得る。あとはバッファーが終了するまで atof を使って配列に数値を記入していく。

3. 複数スレッドで数値への変換を行う

前セクションではデータ読み込みと数値化の処理を分割して別スレッドで行い、時間差で同時に処理することでアスキーデータ読み込みの高速化を実現した。一度使用したファイルはキャッシュとしてメモリ上に保持されるので、読み込み時間は大幅に短縮される。この場合には、アスキーデータから数値化の処理を高速化できればより高速に読み込みが行えることが期待される。

1. ファイルを読んで、ある大きさのブロックごとにメモリーに展開する
2. それを別のスレッドで数値データに変換して、配列としてメモリー上に配置する。

という作業をスレッド別に行うことは前セクションと同様だが、数値データ変換のスレッドは同期は取らずに、ファイルを読み込んだ順に次々にスレッドを作成し、最後にすべての作業が終了した時点でデータの統合を行う。

ブロックの大きさなどをデータとコア数に応じて適切に設定しなければ、スレッドを作りすぎて効率が落ちる可能性もあるが、今回はそうした最適化は行っていない。その状態で

も前節の結果に比べ 4 倍ほどの高速化を達成することができ、相応の効果があったといえる。問題点として、ブロックごとに切り分けた段階ではそのブロックにいくつの粒子分のデータが入っているかが事前に分からないために、あらかじめ用意したデータ保持用の配列にデータを直接保存することができないことがある。そのため、スレッドごとに余裕をもったバッファーを作成し、一度そこに情報を溜めて最後に全体の統合を行っている。データのコピーにかかる若干の時間のロスと、アスキーデータを保持するバッファ、スレッドごとに使うバッファーなど、一時的に使うメモリの使用量が多いことが欠点になる。

3.1. 結果

1. ファイルを読んで、ある大きさのブロックごとにメモリーに展開する

2. それを別のスレッドで数値データに変換して、配列としてメモリー上に配置する。

の手順を次々に行い、最後にすべてのスレッドの終了を待ち、各スレッドで得たデータを一つの配列へコピーを行った。読み込むファイルは前セクション同様に 2000 万体分の id,x,y,z,vx,vy,vz についてのアスキーデータで、1.2GB。一度に読み込むブロックの大きさとして 10MB(スレッド数 123) と 100MB(スレッド数 12) の２通りを行った。(10 回の平均)

3.2. 結果

結果として、CPU タイムは 5 割増することになったが、待ち時間としては 4 分の１程となった。コア数４ (Hyper Threading 有り) の CPU なので、期待通りの高速化といえる。100MB 毎の分割の方は粒度が少し粗すぎたのか、CPU タイムは少し短かったが実待ち時間は少し長かった。ただし、大きな差ではないのでカリカリにチューニングしたい場合でなければ、さほど頑張って最適化する必要はなさそうである。キャッシュにデータが無い場合は、当然ながら前セクションと同様に HDD 読み込み速度で律速されていると思われる。

-	clock	time
スレッド１つ（前セクション。キャッシュ有）	15.5 秒	15 秒
キャッシュ上にデータ有り（10MB ブロック）	24.4 秒	3.4 秒
キャッシュ上にデータ有り（100MB ブロック）	22.0 秒	3.9 秒
キャッシュ上にデータ無し（初回）	16.1 秒	15 秒

Table 3: 複数スレッドで数値化の処理を行った場合

3.3. サンプルプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <string.h>

#include <vector>

#ifndef TRUE
#define TRUE  true
#define FALSE false
#endif

#define _PARALLEL 1
//#define _WRITE_COPY 1
#define _SIZE_MB_ 10

class DataBlock_cls{
public:
    DataBlock_cls();
    ~DataBlock_cls();

    int   mPNum;
    float *mpX;
    float *mpY;
    float *mpZ;
    float *mpVX;
    float *mpVY;
    float *mpVZ;
};

DataBlock_cls::
DataBlock_cls(){
    // 1粒子あたり30byte以上であることを仮定する。
    int maxPNum = 1024 * 1024 * _SIZE_MB_ / 30;
    mpX = new float [maxPNum];
    mpY = new float [maxPNum];
    mpZ = new float [maxPNum];
    mpVX = new float [maxPNum];
    mpVY = new float [maxPNum];
    mpVZ = new float [maxPNum];
    mPNum = 0;
}

DataBlock_cls::
~DataBlock_cls(){
    delete [] mpX;
    delete [] mpY;
    delete [] mpZ;
    delete [] mpVX;
    delete [] mpVY;
    delete [] mpVZ;
}

void *ThreadFunc(void *arg);

int  ReadFile2Memory(FILE **pfp, char **ppAllBuffer);

int  ReadFile2Memory(FILE **pfp, long endPoint, long startPoint, long offset,
                     char **ppAllBuffer){
```

```
    bool bFinal = FALSE;
    if (endPoint - startPoint > 1024*1024*_SIZE_MB_)
        endPoint = startPoint + 1024*1024*_SIZE_MB_;
    else
        bFinal = TRUE;

    int contentsSize = endPoint - startPoint;

    fread(*ppAllBuffer, sizeof(char), contentsSize, *pfp);

    // 後ろから改行記号を探す
    int i;
    for (i = contentsSize; i > 0; i--){
        if ((*ppAllBuffer)[i] == '\n') break;
    }

    if (bFinal == TRUE){
        fclose(*pfp);
        return -1;
    }
    else
        fseek (*pfp, startPoint+i+offset, SEEK_SET);
    return startPoint + i + 1;
}

struct structInformation{
    int            mThreadID;
    long           endPoint;
    DataBlock_cls *mpCDataBlockLink;
    char          *pBuffer;
};

int main(int argc, char* argv[]){

    FILE *fp;
    fp = fopen(argv[1], "rb");

    int PNum;
    fscanf(fp, "%d\n", &PNum);

    float *pX  = new float [PNum];
    float *pY  = new float [PNum];
    float *pZ  = new float [PNum];
    float *pVX = new float [PNum];
    float *pVY = new float [PNum];
    float *pVZ = new float [PNum];

    // データ本体部分のサイズを調査
    fpos_t fsize_struct_start = 0;
    fpos_t fsize_struct_end = 0;
    fgetpos(fp, &fsize_struct_start);
    fseek  (fp, 0, SEEK_END);
    fgetpos(fp, &fsize_struct_end);
    long fsize = fsize_struct_end - fsize_struct_start;

    fseek  (fp, fsize_struct_start, SEEK_SET);

    std::vector < DataBlock_cls* >     VpCDataBlock;
    std::vector < pthread_t* >         VpCPThread;
    std::vector < structInformation* > VpCStructInfo;

    time_t timeStart, timeEnd;
    clock_t start, end;
    start = clock();
```

```
time (&timeStart);

int   threadID = 0;
long  startPoint = 0;
char *pAllBuffer = NULL;
while(TRUE){
    int  nextBufferSize;
    bool bThread = FALSE;
    if (pAllBuffer != NULL){ // 既に読み込み開始したなら
        //スレッドの宣言
        structInformation *pCStructInfo = new structInformation;
        pthread_t     *pCThread    = new pthread_t;
        DataBlock_cls *pCDataBlock = new DataBlock_cls;
        VpCDataBlock .push_back (pCDataBlock);
        VpCPThread   .push_back (pCThread);
        VpCStructInfo.push_back (pCStructInfo);

        pCStructInfo->mThreadID        = threadID;
        pCStructInfo->pBuffer          = pAllBuffer;
        pCStructInfo->endPoint         = nextBufferSize;
        pCStructInfo->mpCDataBlockLink = pCDataBlock;

        pthread_create(pCThread, NULL, ThreadFunc, pCStructInfo);
        bThread = TRUE;
        threadID++;
    }

    if (startPoint == -1){
        break; // ここからループを脱出
    }

    pAllBuffer = new char [1024*1024*_SIZE_MB_];
    long startPointOrg = startPoint;
    startPoint = ReadFile2Memory(&fp, fsize_struct_end, startPoint,
                                 fsize_struct_start,
                                 &pAllBuffer);
    if (startPoint == -1)
        nextBufferSize = fsize - startPointOrg;
    else
        nextBufferSize = startPoint - startPointOrg;
}

fclose(fp);

for (int i = 0; i < VpCPThread.size(); i++){
    //thread の終了を待つ
    pthread_join(*(VpCPThread[i]), NULL);
    delete VpCPThread[i];
    delete VpCStructInfo[i];
}

// データを統合する
int particleNumber = 0;
for (int i = 0; i < VpCDataBlock.size(); i++){
    DataBlock_cls *pCDBlock = VpCDataBlock[i];
    int PNumBlock = pCDBlock->mPNum;
    memcpy(&pX[particleNumber], pCDBlock->mpX, sizeof(float)*PNumBlock);
    memcpy(&pY[particleNumber], pCDBlock->mpY, sizeof(float)*PNumBlock);
    memcpy(&pZ[particleNumber], pCDBlock->mpZ, sizeof(float)*PNumBlock);
    memcpy(&pVX[particleNumber], pCDBlock->mpVX, sizeof(float)*PNumBlock);
```

```
        memcpy(&pVY[particleNumber], pCDBlock->mpVY, sizeof(float)*PNumBlock);
        memcpy(&pVZ[particleNumber], pCDBlock->mpVZ, sizeof(float)*PNumBlock);
        particleNumber += PNumBlock;
        delete pCDBlock;
    }
#ifdef _WRITE_COPY
    fp = fopen("sampleData_Copy.txt", "wt");
    fprintf(fp, "%d\n", PNum);
    for (int i = 0; i < PNum; i++)
        fprintf(fp, "%d %f %f %f %f %f %f\n", i,
                pX[i], pY[i], pZ[i],
                pVX[i], pVY[i], pVZ[i]);
    fclose(fp);
#endif
    end = clock();
    time(&timeEnd);
    printf("Finished time=%f!!\n", (double)(end-start)/CLOCKS_PER_SEC);
    printf("Start Time: %s\n", ctime(&timeStart));
    printf("End   Time: %s\n", ctime(&timeEnd));
    return 1;
}
void *ThreadFunc(void *arg){
    //新しく作成した子スレッドの処理
    structInformation *pStruct = (structInformation*)arg;
    DataBlock_cls* pCDataBlockLink = pStruct->mpCDataBlockLink;
    int    threadID = pStruct->mThreadID;
    char  *pBuffer  = pStruct->pBuffer;
    int    endPoint = pStruct->endPoint;
    float *pX = pCDataBlockLink->mpX;
    float *pY = pCDataBlockLink->mpY;
    float *pZ = pCDataBlockLink->mpZ;
    float *pVX = pCDataBlockLink->mpVX;
    float *pVY = pCDataBlockLink->mpVY;
    float *pVZ = pCDataBlockLink->mpVZ;
    int   particleNumber = 0;
    char  *ptr = pBuffer;
    int i;
    while (TRUE){
        int index = particleNumber;
        i = atoi(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pX[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pY[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pZ[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pVX[index] = atof(ptr);
```

```
        while(*ptr != ' ') ptr++;
        ptr++;
        pVY[index] = atof(ptr);
        while(*ptr != ' ') ptr++;
        ptr++;
        pVZ[index] = atof(ptr);
        while(*ptr != '\n'){
            ptr++;
        }
        ptr++;
        particleNumber++;
        if (ptr >= pBuffer+endPoint) break;
    }
    pCDataBlockLink->mPNum = particleNumber;
    pthread_exit(NULL); //スレッド終了
}
```

3.4. サンプルプログラム中の関数とクラスの解説

今回はバッファの確保にコンストラクタとデストラクタを使いたかったので、C++を用いている。

`class DataBlock_cls(......)`

スレッドごとに用いるバッファを管理するクラス。

`int ReadFile2Memory(......)`

ファイルを一定サイズ読んで、バッファーにアスキーデータをコピーする。（前セクションと同等）

`struct structInformation`

スレッドに渡す構造体。アスキーデータの保存してあるバッファーや、数値化した結果を書き込むべき配列のアドレスなどを保持する。（前セクションと同等）

`int main(....)`

１．バッファー上にあるアスキーデータを数値化するスレッドを作り実行。２．その間に別のバッファーにファイルから次のブロックのアスキーデータ読み込む。というループを繰り返し、最後にデータを統合する。

`void *ThreadFunc(...)`

数値化するスレッドで実行する関数。（前セクションと同等）

4. 結論

アスキーデータの読み込みの高速化を行いたい場合、大きなデータを読み込む際にファイル読み込みをブロックごとに分け、次のブロックを読み込んでいる間に以前に読み込んだデータの数値化を別スレッドで行うことで高速化が行える。`sscanf` を用いるより `atof` を用いた方が高速に処理が行える。ただし、改行の判定も自力で行うなど、プログラミングの手間は増える。また、既にキャッシュ上にファイルが存在する場合には、HDD 等からの読み込みに律速されないために、数値化を複数のスレッドで同時に行うことで、さらなる高速化が可能である。