# インテル® Optane™ DC パーシステント・メモリー導入への道: その 3 - パーシステント・メモリーで C++ アプリケーションをブースト (簡単な grep の例)

この記事は、インテル® デベロッパー・ゾーンに公開されている「Code Sample: Boost Your C++ Applications with Persistent Memory - A Simple grep Example」の日本語参考訳です。

---

サンプルコードをダウンロード

## はじめに

この記事では、パーシステント・メモリー (PMEM) を活用するため、簡単な C++ プログラム (UNIX* の grep コマンドの簡易バージョン) を改良する方法を考えます。最初に、grep プログラムの通常バージョンのコードを詳しく見ていきましょう。次に、検索結果を保持するパーシステント・キャッシュを追加して grep を改良する方法を説明します。キャッシュは、[1] フォールトトレランス (FT) 機能を追加し、[2] すでに表示されている検索パターンの照会を高速化することで grep を改善します。この記事では、パーシステント・メモリー開発キット (PMDK) (英語) のコア・ライブラリーである libpmemobj の C++ バインディングを使用した、grep のパーシステント・バージョンについて考えます。最後に、スレッドと PMEM に対応した同期を使用して、並列処理 (ファイル単位で) を実装します。

## 通常の grep

### 説明

GNU*/Linux* など UNIX* に類似したオペレーティング・システムをよく知る開発者は、コマンドライン・ユーティリティーである grep (Globally search a Regular Expression and Print の略) にも精通していることでしょう。基本的に、grep は 2 つの引数 (残りはオプション) を持ちます。引数には、正規表現形式の文字列パターンと、入力ファイル (標準入力を含む) を指定します。grep の目的は、入力を 1 行ずつ検索して、指定されたパターンに一致する行を表示することです。詳細については、grep のマニュアルページを参照してください (ターミナルで man grep を入力するか、オンラインで Linux* マニュアルページの grep に関する説明 (英語) を参照してください)。

ここで使用する grep の簡易バージョンでは、2 つの引数 (パターンと入力) のみを受け付けます。また、入力は単一ファイル、またはディレクトリーのいずれかでなければなりません。ディレクトリーが指定されると、入力ファイルを見つけるためそのディレクトリーがスキャンされます (サブディレクトリーは常に再帰的にスキャンされます)。これがどのように動作するか確認するため、入力としてソースコードを、パターンとして「int」を指定して実行してみます。

コードは、GitHub* からダウンロード (英語) できます。pmdk-examples リポジトリーのルートからコードをコンパイルするには、**make simple-grep** と入力します。システムに libpmemobj (英語) と C++ コンパイラーがインストールされている必要があります。Windows* オペレーティング・システムとの互換性を維持するため、コードでは Linux* 固有の関数は使用していません。代わりに、(基本的なファイルシステムの入出力を処理するため) Boost C++ library collection (英語) を使用します。Linux* を使用する場合、Boost C++ はディ

ストリビューション向けにパッケージが提供されています。例えば、Ubuntu* 16.04 では次のようにインストールします。

```
# sudo apt-get install libboost-all-dev
```

プログラムが正しくコンパイルされると、次のように実行できます。

```
$ ./grep int grep.cpp
FILE = grep.cpp
44: int
54:     int ret = 0;
77: int
100: int
115: int
135: int
136: main (int argc, char *argv[])
```

ご覧の通り、grep は「int」という単語が含まれる 7 つの行 (行 44、54、77、100、115、135、および 136) を検出しました。確認のため、システムが提供する grep を使用して同様の操作を行います。

```
$ grep int -n grep.cpp
44:int
54:     int ret = 0;
77:int
100:int
115:int
135:int
136:main (int argc, char *argv[])
```

同じ結果が得られました。以下にコードを示します (注: コードのフォーマットが元のソースファイルとは異なるため、上記の行番号は以下のソースとは一致していません)。

```
#include <boost/filesystem.hpp>
#include <boost/foreach.hpp>
#include <fstream>
#include <iostream>
#include <regex>
#include <string.h>
#include <string>
#include <vector>

using namespace std;
using namespace boost::filesystem;
/* 補助関数 */
int
process_reg_file (const char *pattern, const char *filename)
{
    ifstream fd (filename);
    string line;
    string patternstr ("(.*)(");
    patternstr += string (pattern) + string (")(.*)");
    regex exp (patternstr);

    int ret = 0;
    if (fd.is_open ()) {
        size_t linenum = 0;
        bool first_line = true;
        while (getline (fd, line)) {
            ++linenum;
            if (regex_match (line, exp)) {
                if (first_line) {
                    cout << "FILE = " << string (filename);
```

```
                    cout << endl << flush;
                    first_line = false;
                }
                cout << linenum << ": " << line << endl;
                cout << flush;
            }
        }
    } else {
        cout << "unable to open file " + string (filename) << endl;
        ret = -1;
    }
    return ret;
}

int
process_directory_recursive (const char *dirname, vector<string> &files)
{
    path dir_path (dirname);
    directory_iterator it (dir_path), eod;

    BOOST_FOREACH (path const &pa, make_pair (it, eod)) {
        /* フルパス名 */
        string fpname = pa.string ();
        if (is_regular_file (pa)) {
            files.push_back (fpname);
        } else if (is_directory (pa) && pa.filename () != "."
                   && pa.filename () != "..") {
            if (process_directory_recursive (fpname.c_str (), files)
                < 0)
                return -1;
        }
    }
    return 0;
}

int
process_directory (const char *pattern, const char *dirname)
{
    vector<string> files;
    if (process_directory_recursive (dirname, files) < 0)
        return -1;
    for (vector<string>::iterator it = files.begin (); it != files.end ();
         ++it) {
        if (process_reg_file (pattern, it->c_str ()) < 0)
            cout << "problems processing file " << *it << endl;
    }
    return 0;
}

int
process_input (const char *pattern, const char *input)
{
    /* 入力タイプをチェック */
    path pa (input);
    if (is_regular_file (pa))
        return process_reg_file (pattern, input);
    else if (is_directory (pa))
        return process_directory (pattern, input);
    else {
        cout << string (input);
        cout << " is not a valid input" << endl;
    }
    return -1;
}

/* MAIN */
int
main (int argc, char *argv[])
```

```
{
    /* パラメーター読み込み */
    if (argc < 3) {
        cout << "USE " << string (argv[0]) << " pattern input ";
        cout << endl << flush;
        return 1;
    }
    return process_input (argv[1], argv[2]);
}
```

長いコードですが、それほど難解なコードではありません。ここで行うことは、`process_input()` で入力がファイルであるかディレクトリーであるかを確認するだけです。ファイルの場合、`process_reg_file()` で直接処理します。ディレクトリーの場合、`process_directory_recursive()` でファイルをスキャンし、`process_directory()` を呼び出すことで、スキャンされたファイルを `process_reg_file()` で 1 つずつ処理します。ファイルを処理する際に、各行がパターンと一致するか確認します。一致すると、行は標準出力に表示されます。

### パーシステント・キャッシュ

次に grep を改善する方法を考えてみましょう。最初に気付いたことは、grep が状態をまったく保持していないことです。入力が解析され、出力が生成されると、プログラムは単純に終了します。例えば、毎週、特定のパターンを検索するため (数十万のファイルがある) ディレクトリーをスキャンすることを想定してみましょう。そして、このディレクトリー内のファイルは、時間経過とともに変化する可能性があるとします (すべて同時ではない可能性が大です)。また、ファイルが追加される可能性もあります。このタスクに従来の grep を使用すると、一部のファイルを何度もスキャンする可能性があり、貴重な CPU サイクルを無駄にします。この問題はキャッシュを追加することで解決できます。ファイルが特定のパターンですでにスキャンされていることが判明した場合 (または、最後にスキャンされてから内容が変更されていない場合)、grep はファイルを再スキャンする代わりにキャッシュされた結果を返します。

キャッシュを実装するにはいくつかの方法があります。例えば、特定のデータベース (DB) を作成して、スキャンされた各ファイルとパターンの結果を保持する (ファイルの変更を検出するためタイムスタンプも追加) 方法があります。これは、確実に機能しますが、ファイルが解析されるたびに DB クエリーを実行する必要があり、DB エンジンも必要になります (ネットワークと入出力のオーバーヘッドが生じる可能性もあります)。複雑性も増すため、より単純なソリューションのほうが良いでしょう。別の方法として、キャッシュを通常のファイルとして保存し、最初にパーシステント・メモリーにロードして、実行の最後または新しいファイルが解析されるたびにパーシステント・メモリーを更新することが考えられます。これは、最初のものよりも良いアプローチであるように思われますが、2 つのデータモデル (揮発性 RAM と第 2 のパーシステント・ストレージ (ファイル)) を作成する必要があります。この追加のコーディング作業は避けた方が良いでしょう。

## パーシステント grep

### 設計上の考慮事項

libpmemobj を使用して PMEM 対応のコードを作成するには、最初に永続化されるデータ・オブジェクトのタイプを設計する必要があります。定義すべき最初のタイプは、ルート・オブジェクトのタイプです。このオブジェクトは必須であり、PMEM プールで作成されるほかのすべてのオブジェクトを維持するために使用されます (プールを PMEM デバイス内のファイルと考えます)。この grep のサンプルでは、次の永続的なデータ構造を使用します。

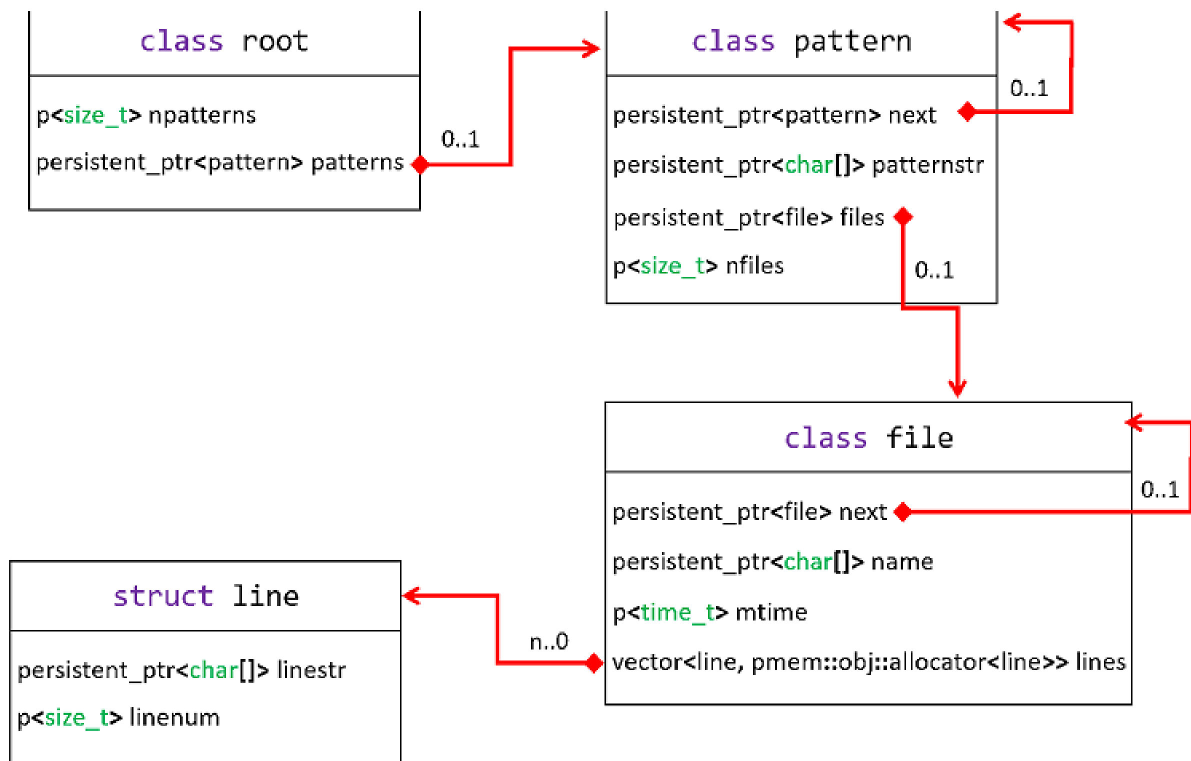

図 1. PMEM 対応 grep のデータ構造

キャッシュされるデータは、root クラス配下に patterns のリンクリストを作成することで構成されます。新しいパターンが検索されるたびに、pattern クラスの新しいオブジェクトが作成されます。検索するパターンが以前にも検索されている場合、オブジェクトは作成されません (パターン文字列が patternstr に保存されます)。pattern クラスから、スキャンされたファイルのリンクリストをつなげます。ファイルは、名前 (ファイルシステムのパスと同一)、変更時間 (ファイル更新の確認に使用)、およびパターンに一致する行のベクトルで構成されます。以前にスキャンされていないファイルに対してのみ、file クラスの新しいオブジェクトを作成します。

ここで最初に気付くことは、特殊クラス `p<>` (基本型用) と `persistent_ptr<>` (複合型へのポインター用) です。これらのクラスは、トランザクション中にメモリー領域をライブラリーに通知するために使用されます (オブジェクトに対する更新はログに記録され、障害が発生するとロールバックされます)。また、仮想メモリーの特性により、PMEM にあるポインターには常に `persistent_ptr<>` を使用する必要があります。プロセスによってプールが開かれ、仮想アドレス空間にマップされると、プールの位置は同じプロセス (または、同じプールにアクセスするほかのプロセス) が使用する以前の位置と異なる可能性があります。PMDK を使用する場合、パーシステント・ポインターはファットポインターとして実装されます。これは、プール ID (変換テーブルから現在のテーブル仮想アドレスへのアクセスに使用) + オフセット (プールの先頭から) で構成されます。PMDK の詳細については、「libpmemobj のタイプ・セーフ・マクロ」(英語) および「libpmemobj の C++ バインディング (パート 2) - パーシステント・スマート・ポインター」(英語) を参照してください。

行のベクトル (`std::vector`) が、パーシステント・ポインターとして宣言されていないことに疑問を持つかもしれません。その理由は、必要ないからです。ベクトルを表すオブジェクトである line は、(file クラスのオブジェクトの構築中に) 一度作成されると変更されることがないため、トランザクション中にオブジェクトを追跡する必要はありません。それでも、ベクトル自体はオブジェクトを内部的に割り当ておよび削除します。そのため、std::vector (揮発性メモリーのみを識別し、ヒープ内にすべてのオブジェクトを割り当てる) のデフォルト・アロケーターのみをあてにするわけにはいきません。libpmemobj が提供する PMEM 対応のカスタマイズされた

バージョン、`pmem::obj::allocator<line>` を使用する必要があります。ベクトルを宣言して通常の揮発性コードと同じように使用することができます。この方法で標準のコンテナークラスを使用できます。

## コードの修正

では、コードを見ていきましょう。重複を避けるため、新しいコードのみを示します (完全なコードは、`pmemgrep/pmemgrep.cpp` にあります)。新しいヘッダー、マクロ、名前空間、グローバル変数、およびクラスの定義から始めます。

```
...
#include <libpmemobj++/allocator.hpp>
#include <libpmemobj++/make_persistent.hpp>
#include <libpmemobj++/make_persistent_array.hpp>
#include <libpmemobj++/persistent_ptr.hpp>
#include <libpmemobj++/transaction.hpp>
...
#define POOLSIZE ((size_t) (1024 * 1024 * 256)) /* 256MB */
...
using namespace pmem;
using namespace pmem::obj;

/* グローバル */
class root;
pool<root> pop;

/* パーシステント・データ構造 */
struct line {
    persistent_ptr<char[]> linestr;
    p<size_t> linenum;
};

class file
{
    private:

    persistent_ptr<file> next;
    persistent_ptr<char[]> name;
    p<time_t> mtime;
    vector<line, pmem::obj::allocator<line>> lines;

    public:

    file (const char *filename)
    {
        name = make_persistent<char[]> (strlen (filename) + 1);
        strcpy (name.get (), filename);
        mtime = 0;
    }

    char * get_name (void) { return name.get (); }

    size_t get_nlines (void) { return lines.size (); /* nlines; */ }

    struct line * get_line (size_t index) { return &(lines[index]); }

    persistent_ptr<file> get_next (void) { return next; }

    void set_next (persistent_ptr<file> n) { next = n; }

    time_t get_mtime (void) { return mtime; }

    void set_mtime (time_t mt) { mtime = mt; }
```

```
    void
    create_new_line (string linestr, size_t linenum)
    {
        transaction::exec_tx (pop, [&] {
            struct line new_line;
            /* 新しい行を作成 */
            new_line.linestr
            = make_persistent<char[]> (linestr.length () + 1);
            strcpy (new_line.linestr.get (), linestr.c_str ());
            new_line.linenum = linenum;
            lines.insert (lines.cbegin (), new_line);
        });
    }

    int
    process_pattern (const char *str)
    {
        ifstream fd (name.get ());
        string line;
        string patternstr ("(.*)(");
        patternstr += string (str) + string (")(.*)");
        regex exp (patternstr);
        int ret = 0;
        transaction::exec_tx (
        pop, [&] { /* 処理中のファイルをそのままにしない */
              if (fd.is_open ()) {
                  size_t linenum = 0;
                  while (getline (fd, line)) {
                      ++linenum;
                      if (regex_match (line, exp))
                          /* adding this line... */
                          create_new_line (line, linenum);
                  }
              } else {
                  cout
                  << "unable to open file " + string (name.get ())
                  << endl;
                  ret = -1;
              }
        });
        return ret;
    }

    void remove_lines () { lines.clear (); }
};

class pattern
{
    private:

    persistent_ptr<pattern> next;
    persistent_ptr<char[]> patternstr;
    persistent_ptr<file> files;
    p<size_t> nfiles;

    public:

    pattern (const char *str)
    {
        patternstr = make_persistent<char[]> (strlen (str) + 1);
        strcpy (patternstr.get (), str);
        files = nullptr;
        nfiles = 0;
    }
```

```
    file *
    get_file (size_t index)
    {
        persistent_ptr<file> ptr = files;
        size_t i = 0;
        while (i < index && ptr != nullptr) {
            ptr = ptr->get_next ();
            i++;
        }
        return ptr.get ();
    }

    persistent_ptr<pattern> get_next (void)    { return next; }

    void set_next (persistent_ptr<pattern> n) { next = n; }

    char * get_str (void) { return patternstr.get (); }

    file *
       find_file (const char *filename) {
        persistent_ptr<file> ptr = files;
        while (ptr != nullptr) {
            if (strcmp (filename, ptr->get_name ()) == 0)
                return ptr.get ();
            ptr = ptr->get_next ();
        }
        return nullptr;
    }

    file *
       create_new_file (const char *filename) {
        file *new_file;
        transaction::exec_tx (pop, [&] {
            /* 新しいファイルヘッドを割り当て */
            persistent_ptr<file> new_files
            = make_persistent<file> (filename);
            /* 新しい割り当てを実際のヘッドにする */
            new_files->set_next (files);
            files = new_files;
            nfiles = nfiles + 1;
            new_file = files.get ();
        });
        return new_file;
    }

    void
    print (void)
    {
        cout << "PATTERN = " << patternstr.get () << endl;
        cout << "\tpattern present in " << nfiles;
        cout << " files" << endl;
        for (size_t i = 0; i < nfiles; i++) {
            file *f = get_file (i);
            cout << "###############" << endl;
            cout << "FILE = " << f->get_name () << endl;
            cout << "###############" << endl;
            cout << "*** pattern present in " << f->get_nlines ();
            cout << " lines ***" << endl;
            for (size_t j = f->get_nlines (); j > 0; j--) {
                cout << f->get_line (j - 1)->linenum << ": ";
                cout
                << string (f->get_line (j - 1)->linestr.get ());
                cout << endl;
            }
        }
    }
};
```

```
class root
{
    private:

    p<size_t> npatterns;
    persistent_ptr<pattern> patterns;

    public:

    pattern *
    get_pattern (size_t index)
    {
        persistent_ptr<pattern> ptr = patterns;
        size_t i = 0;
        while (i < index && ptr != nullptr) {
            ptr = ptr->get_next ();
            i++;
        }
        return ptr.get ();
    }

    pattern *
    find_pattern (const char *patternstr)
    {
        persistent_ptr<pattern> ptr = patterns;
        while (ptr != nullptr) {
            if (strcmp (patternstr, ptr->get_str ()) == 0)
                return ptr.get ();
            ptr = ptr->get_next ();
        }
        return nullptr;
    }

    pattern *
    create_new_pattern (const char *patternstr)
    {
        pattern *new_pattern;
        transaction::exec_tx (pop, [&] {
            /* 新しいパターン配列を割り当て */
            persistent_ptr<pattern> new_patterns
            = make_persistent<pattern> (patternstr);
            /* 新しい割り当てを実際のヘッドにする */
            new_patterns->set_next (patterns);
            patterns = new_patterns;
            npatterns = npatterns + 1;
            new_pattern = patterns.get ();
        });
        return new_pattern;
    }

    void
    print_patterns (void)
    {
        cout << npatterns << " PATTERNS PROCESSED" << endl;
        for (size_t i = 0; i < npatterns; i++)
            cout << string (get_pattern (i)->get_str ()) << endl;
    }
}
...
```

これは図 1 のダイアグラムに対応する C++ コードです。libpmemobj のヘッダー、プールのサイズを定義するマクロ (POOLSIZE)、および開いているプールを格納するグローバル変数 (pop) などを確認できます。pop は特殊なファイル記述子であると考えられます。`root::create_new_pattern()`、`pattern::create_new_file()`、および `file::create_new_line()` データ構造に対するすべての変更が、トランザクションにより保護されていることに注意してください。libpmemobj の C++ バインディング

では、トランザクションはラムダ関数によって実装されています。そのため、コンパイルには C++11 互換のコンパイラーが必要です。何らかの理由でラムダ関数を使用できない (しない) 場合は、別の方法 (英語) もあります。

また、通常の malloc() や C++ の new 演算子の代わりに、make_persistent<>() を使用してすべてのメモリーが割り当てられることに注意してください。

古い process_reg_file() 関数は、file::process_pattern() メソッドに移動します。新しい process_reg_file() は、処理中のファイルが (与えられたパターンに存在し、前回から変更されていない場合) パターン検索済みか確認するロジックを実装します。

```
int
process_reg_file (pattern *p, const char *filename, const time_t mtime)
{
    file *f = p->find_file (filename);
    if (f != nullptr && difftime (mtime, f->get_mtime ()) == 0) /* ファイルは存在する */
        return 0;
    if (f == nullptr) /* ファイルは存在しない */
        f = p->create_new_file (filename);
    else /* ファイルは存在するがタイムスタンプが古い (変更) */
        f->remove_lines ();
    if (f->process_pattern (p->get_str ()) < 0) {
        cout << "problems processing file " << filename << endl;
        return -1;
    }
    f->set_mtime (mtime);
    return 0;
}
```

ほかの関数への変更は、変更時間の追加のみです。例えば、process_directory_recursive() は、vector<string> ではなく tuple<string, time_t> のベクトルを返します。

```
int
process_directory_recursive (const char *dirname,
                    vector<tuple<string, time_t>> &files)
{
    path dir_path (dirname);
    directory_iterator it (dir_path), eod;
    BOOST_FOREACH (path const &pa, make_pair (it, eod)) {
        /* フルパス名 */
        string fpname = pa.string ();
        if (is_regular_file (pa)) {
            files.push_back (
            tuple<string, time_t> (fpname, last_write_time (pa)));
        } else if (is_directory (pa) && pa.filename () != "."
               && pa.filename () != "..") {
            if (process_directory_recursive (fpname.c_str (), files)
                < 0)
                return -1;
        }
    }
    return 0;
}
```

**サンプルを実行**

このコードを「int」と「void」の 2 つのパターンで実行してみます。ここでは、PMEM デバイス (実際の PMEM、または RAM を使用してエミュレート (英語)) が、/mnt/mem にマウントされていることを前提とします。

```
$ ./pmemgrep /mnt/mem/grep.pool int pmemgrep.cpp
$ ./pmemgrep /mnt/mem/grep.pool void pmemgrep.cpp
$
```

引数なしで実行すると、キャッシュされたパターンが表示されます。

```
$ ./pmemgrep /mnt/mem/grep.pool
2 PATTERNS PROCESSED
void
int
```

パターンのみを指定すると、キャッシュされている結果が表示されます。

```
$ ./pmemgrep /mnt/mem/grep.pool void
PATTERN = void
    1 file(s) scanned
###############
FILE = pmemgrep.cpp
###############
*** pattern present in 15 lines ***
80:     get_name (void)
86:     get_nlines (void)
98:     get_next (void)
103:    void
110:    get_mtime (void)
115:    void
121:    void
170:    void
207:    get_next (void)
212:    void
219:    get_str (void)
254:    void
255:    print (void)
326:    void
327:    print_patterns (void)
$
$ ./pmemgrep /mnt/mem/grep.pool int
PATTERN = int
    1 file(s) scanned
###############
FILE = pmemgrep.cpp
###############
*** pattern present in 14 lines ***
137:    int
147:        int ret = 0;
255:    print (void)
327:    print_patterns (void)
337: int
356: int
381: int
395: int
416: int
417: main (int argc, char *argv[])
436:    if (argc == 2) /* パターン指定なし。保存されたパターンを表示して終了 */
438:        proot->print_patterns ();
444:        if (argc == 3) /* 入力なし。データを表示して終了 */
445:            p->print ();
$
```

既存のパターンにファイルを追加することもできます。

```
$ ./pmemgrep /mnt/mem/grep.pool void Makefile
$ ./pmemgrep /mnt/mem/grep.pool void
PATTERN = void
    2 file(s) scanned
###############
FILE = Makefile
###############
*** pattern present in 0 lines ***
###############
FILE = pmemgrep.cpp
###############
*** pattern present in 15 lines ***
80:     get_name (void)
86:     get_nlines (void)
98:     get_next (void)
103:    void
110:    get_mtime (void)
115:    void
121:    void
170:    void
207:    get_next (void)
212:    void
219:    get_str (void)
254:    void
255:    print (void)
326:    void
327:    print_patterns (void)
```

## 並列パーシステント grep

ここまで改良を行ってきたので、マルチスレッドのサポートも考えてみましょう。これには、若干の追加コードが必要です (完全なコードは `pmemgrep_thx/pmemgrep.cpp` にあります)。

最初に pthread と パーシステントな mutex を使用するのに必要なヘッダーを追加します (これについては後述します)。

```
...
#include <libpmemobj++/mutex.hpp>
...
#include <thread>
```

プログラム内でスレッド数を設定するため新しいグローバル変数を追加し、スレッド数を指定するコマンドライン・オプション (`-nt=number_of_threads`) を受け入れるようにします。`-nt` が指定されない場合、シングルスレッドで実行されます。

```
int num_threads = 1;
```

次に、パーシステント mutex を pattern クラスに追加します。この mutex は、ファイルのリンクリストへの書き込みを同期するために使用されます (並列処理がファイル単位で行われるため)。

```
class pattern
{
    private:

    persistent_ptr<pattern> next;
    persistent_ptr<char[]> patternstr;
```

```
    persistent_ptr<file> files;
    p<size_t> nfiles;
    pmem::obj::mutex pmutex;
    ...
```

なぜ mutex の `pmem::obj` バージョンが必要であるか、疑問に思うかもしれません。これは、mutex が PMEM に保存されており、クラッシュした場合に libpmemobj がリセットする必要があるためです。正しくリカバーできないと、破損した mutex によってデッドロックが発生する可能性があります。詳細については、「libpmemobj と同期」(英語) の記事をご覧ください。

PMEM に mutex を格納することは、特定のパーシステント・データ・オブジェクトに mutex を関連付ける場合には有用ですが、すべての状況で必須ではありません。実際、このサンプルでは、単一の標準 mutex 変数 (揮発性メモリーに格納) で十分です (すべてのスレッドが一度に 1 つのパターンのみを処理するため)。パーシステントな mutex を使用する理由は、その存在を紹介するためです。つまり、必須ではありません。

パーシステントであるか否かにかかわらず、mutex を取得したら単純に `transaction::exec_tx()` (最後のパラメーター) に渡すだけで、`pattern::create_new_file()` への書き込みを同期できます。

```
transaction::exec_tx (pop,
    [&] { /* ロックされたトランザクション */
              /* 新しいファイルヘッドを割り当て */
        persistent_ptr<file> new_files
            = make_persistent<file> (filename);
              /* 新しい割り当てを実際のヘッドにする */
        new_files->set_next (files);
        files = new_files;
        nfiles = nfiles + 1;
        new_file = files.get ();
    },
    pmutex); /* ロックされたトランザクションの最後 */
```

最後のステップは、`process_directory()` を実行してスレッドを生成およびジョインすることです。スレッドを制御するため、新しい関数 `process_directory_thread()` を作成します (スレッド ID でワークを分割)。

```
void
process_directory_thread (int id, pattern *p,
    const vector<tuple<string, time_t>> &files)
{
    size_t files_len = files.size ();
    size_t start = id * (files_len / num_threads);
    size_t end = start + (files_len / num_threads);
    if (id == num_threads - 1)
        end = files_len;
    for (size_t i = start; i < end; i++)
        process_reg_file (p, get<0> (files[i]).c_str (),
            get<1> (files[i]));
}

int
process_directory (pattern *p, const char *dirname)
{
    vector<tuple<string, time_t>> files;
    if (process_directory_recursive (dirname, files) < 0)
        return -1;
    /* スレッドを開始し、ワークを分割 */
    thread threads[num_threads];
    for (int i = 0; i < num_threads; i++)
```

```
        threads[i] = thread (process_directory_thread, i, p, files);
    /* スレッドのジョイン */
    for (int i = 0; i < num_threads; i++)
        threads[i].join ();
    return 0;
}
```

## まとめ

この記事では、パーシステント・メモリー (PMEM) を活用するため、簡単な C++ プログラム (UNIX* の grep コマンドの簡易バージョン) を改良する方法を紹介しました。最初に、grep プログラムの通常バージョンのコード調査し、その後、PMDK のコア・ライブラリーである libpmemobj の C++ バインディングを使用して、PMEM キャッシュを追加することでプログラムを改善しました。最後に、スレッドと PMEM に対応した同期を使用して、並列処理 (ファイル単位で) を実装しました。

## 著者紹介

Eduardo Berrocal は、イリノイ州シカゴのイリノイ工科大学 (IIT) でコンピューター・サイエンスの博士号を取得した後、2017 年 7 月にクラウド・ソフトウェア・エンジニアとしてインテルに入社しました。彼の博士課程での研究は、ハイパフォーマンス・コンピューティングのデータ解析とフォールトトレランスでした。過去には、ベル研究所 (Nokia*) で夏のインターン、アルゴンヌ国立研究所の研究助手、シカゴ大学でサイエンス・プログラマーおよびウェブ開発者、そしてスペインの CESVIMA 研究所でインターンとして勤務していました。

## 関連情報


1. パーシステント・メモリー開発キット (PMDK)
   http://pmem.io/pmdk/ (英語)
2. grep コマンドのマニュアルページ
   https://linux.die.net/man/1/grep (英語)
3. Boost C++ Library collection
   http://www.boost.org/ (英語)
4. libpmemobj のタイプ・セーフ・マクロ
   http://pmem.io/2015/06/11/type-safety-macros.html (英語)
5. libpmemobj の C++ バインディング (パート 2) - パーシステント・スマート・ポインター
   http://pmem.io/2016/01/12/cpp-03.html (英語)
6. libpmemobj の C++ バインディング (パート 6) - トランザクション
   http://pmem.io/2016/05/25/cpp-07.html (英語)
7. 7. パーシステント・メモリーのエミュレーション
   http://pmem.io/2016/02/22/pm-emulation.html (英語)
8. GitHub* のサンプルコードへのリンク (英語)


コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。