CodeZine
開発者のための実装系Webマガジン

特集記事

C++11：スレッド･ライブラリひとめぐり

Visual Studio 2012 RC であそんでみたよ

C++ Visual Studio 開発／設計／テスト

WEB用を表示

επιστημη[著]

2012/06/29 14:00

ダウンロード サンプルファイル (36.4 KB)

C++11が提供するスレッド･ライブラリの使い心地を、Visual Studio 2012 RCでの試運転を兼ねてざっくりと体感してみましょう。

5月の末、Visual Studio 2012 RCがリリースされました。僕は根っからのC++屋ですから、興味の対象は第一にVisual C++ 2012（以下、VC11）です。現Visual C++ 2010（VC10）にはlambdaなどの国際標準C++11の一部をサポートしていますけど、VC11ではさらにC++のサポート範囲が広がっています。今回はVC++11で新たに追加された標準スレッド･ライブラリのご紹介です。

十万未満の素数はいくつある？

「1とそれ自身以外の約数を持たない数」が素数です。言い換えれば「2以上n未満のすべての数でnを割り切ることができなければ、nは素数」ですわね。だからnが素数であるか否かを判定する関数はこんなカンジ。

リスト1

```
// nは素数?
bool is_prime(int n) {
  for ( int i = 2; i < n; ++i ) {
    if ( n % i == 0 ) {
      return false;
    }
  }
  return true;
}
```

このis_primeを使って「lo以上hi未満の範囲にある素数の数」を求める関数count_primeは

リスト2

```
// lo以上hi未満の範囲に素数はいくつある?
int count_prime(int lo, int_hi) {
  int result = 0;
  for ( int i = lo; i < hi; ++i ) {
    if ( is_prime(i) ) {
      ++result;
    }
  }
  return result;
}
```

10万未満の素数を勘定してみましょう。

リスト3

```
/*
 * M未満の素数はいくつある?
 */
void single(int M) {
  chrono::system_clock::time_point start = chrono::system_clock::now();
  int result = count_prime(2,M);
  chrono::duration<double> sec = chrono::system_clock::now() - start;
  cout << result << ' ' << sec.count() << "[sec]" << endl;
}

int main() {
  const int M = 100000;
  single(M);
}

/* 実行結果:
9592 1.64009[sec]
*/
```

10万までの整数には1割弱の素数があるんですねぇ。僕のマシン（i7-2600K、Windows7-64bit）では1.6秒ほどで答えを出してくれました。

0:Windows API

この計算をマルチスレッドによる高速化を試みます。戦略はいたって単純、素数か否かの判定範囲、[2,M) (2以上M未満,M=100000)をスレッド数で等分します。たとえばスレッドふたつなら [2,M/2) と [M/2,M)に。んでもって各スレッドに分割された範囲での素数の勘定を分担させ、それぞれの結果を全部足し合わせればよかろう、と。

まずはWindows APIで実装します。範囲[lo,hi)をn等分するクラスdiv_rangeを用意します。

リスト4　div_range.h

```
#ifndef DIV_RANGE_H_
#define DIV_RANGE_H_

// [lo,hi) を n等分する
template<typename T =int>
class div_range {
private:
  T lo_;
  T hi_;
  T stride_;
  int n_;
public:
  div_range(T lo, T hi, int n)
    : lo_(lo), hi_(hi), n_(n) { stride_ = (hi-lo)/n; }
  T lo(int n) const { return lo_ + stride_ * n; }
  T hi(int n) const { return (++n < n_) ? lo_ + stride_*n : hi_; }
};
#endif
```

分割された範囲ごとにスレッドを起こし、全スレッドが完了したところで積算します。

リスト5

```
/* Win32 thread のためのwrapper */

// <0>:loi, <1>:hi , <1>:result
typedef tuple<int,int,int> thread_io;

DWORD WINAPI thread_entry(LPVOID argv) {
  thread_io& io = *static_cast<thread_io*>(argv);
  get<2>(io) = count_prime(get<0>(io), get<1>(io));
  return 0;
}

/*
 * M未満の素数はいくつある?
 */
void multi(int M, int nthr) {
  vector<HANDLE> handle(nthr);
  vector<thread_io> io(nthr);
  div_range<> rng(2,M,nthr);

  for ( int i = 0; i< nthr; ++i ) {
    io[i] = thread_io(rng.lo(i), rng.hi(i), 0);
  }

  chrono::system_clock::time_point start = chrono::system_clock::now();
  for ( int i = 0; i< nthr; ++i ) {
    handle[i] = CreateThread(NULL, 0, &thread_entry, &io[i], 0, NULL);
  }
  WaitForMultipleObjects(nthr, &handle[0], TRUE, INFINITE);
  chrono::duration<double> sec = chrono::system_clock::now() - start;

  int result = 0;
  for ( int i = 0; i < nthr; ++i ) {
    CloseHandle(handle[i]);
    result += get<2>(io[i]);
  }
  cout << result << ' ' << sec.count() << "[sec] : " << nthr << endl;
}

int main() {
  const int M = 100000;
  for ( int i = 1; i < 10; ++i ) multi(M, i);
}

/* 実行結果:
9592 1.6651[sec] : 1
9592 1.21607[sec] : 2
9592 0.970055[sec] : 3
9592 0.752043[sec] : 4
9592 0.631036[sec] : 5
9592 0.52903[sec] : 6
9592 0.549031[sec] : 7
9592 0.497028[sec] : 8
9592 0.470027[sec] : 9
*/
```

スレッド数1～9で実行すると、8個の論理コアを持つi7では当然のことながらスレッド8個あたりでスピード頭打ちとなります。3倍ちょっと速くなってますね。

Window-APIで実装する場合、スレッドの本体であるcount_primeをスレッドに乗っけるためのwrapper（このサンプルではthread_ioとthread_entry）が必要になり

ます。

1：thread

それでは、C++11のスレッド・ライブラリを使った版をご覧ください。

リスト6

```
void multi(int M, int nthr) {
  vector<thread> thr(nthr);
  vector<int> count(nthr);
  div_range<> rng(2,M,nthr);

  chrono::system_clock::time_point start = chrono::system_clock::now();
  for ( int i = 0; i < nthr; ++i ) {
    thr[i] = thread([&,i](int lo, int hi) { count[i] = count_prime(lo,hi); },
                    rng.lo(i), rng.hi(i));
  }
  int result = 0;
  for ( int i = 0; i < nthr; ++i ) {
    thr[i].join();
    result += count[i];
  }
  chrono::duration<double> sec = chrono::system_clock::now() - start;

  cout << result << ' ' << sec.count() << "[sec] : " << nthr << endl;
}

int main() {
  const int M = 100000;
  for ( int i = 1; i < 10; ++i ) multi(M, i);
}
```

いかがです？ Windows APIによる実装と比べるとあっけないほどに簡単、std::threadのコンストラクタにスレッドのエントリとそれに与える引数とを食わせるだけでスレッドが生成されて動き出します。あとは join() で完了を待つだけ。

コンストラクタに引き渡すスレッド・エントリは()できるもの、つまりファンクタならなんでもOKです。

リスト7

```
 void count_prime_function(int lo, int hi, int& result) {
   result = count_prime(lo, hi);
 }

class count_prime_class {
   int& result_;
public:
  count_prime_class(int& result) : result_(result) {}
  void operator()(int lo, int hi) { result_ = count_prime(lo, hi); }
};

void multi(int M) {
  thread thr[3];
  int    count[3];
  div_range<> rng(2,M,3);

  auto count_prime_lambda = [&](int lo, int hi) { count[2] = count_prime(lo,hi); };

  chrono::system_clock::time_point start = chrono::system_clock::now();
```

```
  // 関数ポインタ, クラスインスタンス, lambda式からスレッドを作る
  thr[0] = thread(count_prime_function,        rng.lo(0), rng.hi(0), ref(count[0]));
  thr[1] = thread(count_prime_class(count[1]), rng.lo(1), rng.hi(1));
  thr[2] = thread(count_prime_lambda,          rng.lo(2), rng.hi(2));
  for ( thread& t : thr ) t.join();
  chrono::duration<double> sec = chrono::system_clock::now() - start;

  cout << count[0] + count[1] + count[2] << ' '
       << sec.count() << "[sec]" << endl;
}
```

2：async／future

threadを使った場合、join()による完了待ちの後、結果の参照（読み取り）を行うのですが、async／futureを使えば完了待ちと結果の参照が簡略化されます。戻り値（結果）を返すファンクタと引数をasyncに与えるとfutureが返ってきます。そのfutureに対するget()ひとつで、完了待ちと結果の参照が行えます。

リスト8

```
class count_prime_class {
public:
  int operator()(int lo, int hi) { return count_prime(lo, hi); }
};

void multi(int M) {
  future<int> fut[3];
  div_range<> rng(2,M,3);

  auto count_prime_lambda = [&](int lo, int hi) { return count_prime(lo,hi); };

  chrono::system_clock::time_point start = chrono::system_clock::now();
  // 関数ポインタ, クラスインスタンス, lambda式からfutureを作る
  fut[0] = async(count_prime,         rng.lo(0), rng.hi(0));
  fut[1] = async(count_prime_class(), rng.lo(1), rng.hi(1));
  fut[2] = async(count_prime_lambda,  rng.lo(2), rng.hi(2));
  int result = fut[0].get() + fut[1].get() + fut[2].get();
  chrono::duration<double> sec = chrono::system_clock::now() - start;

  cout << result << sec.count() << "[sec]" << endl;
}
```

3：mutex/atomic

1:threadで示したコードを少しばかりいじくります。lo以上hi未満の素数を勘定するcoutn_primeをばっさり削り、lambda内で直接やらせます。

リスト9

```
void multi(int M, int nthr) {
  vector<thread> thr(nthr);
  div_range<> rng(2,M,nthr);

  int result = 0;

  chrono::system_clock::time_point start = chrono::system_clock::now();
  for ( int t = 0; t < nthr; ++t ) {
    thr[t] = thread([&](int lo, int hi) {
                      for ( int n = lo; n < hi; ++n ) {
                        // nが素数なら resultに1を加える
                        if ( is_prime(n) ) {
                          ++result;
```

```
                        }
                      }
                    },
                    rng.lo(t), rng.hi(t));
  }
  for ( thread& th : thr ) { th.join(); }
  chrono::duration<double> sec = chrono::system_clock::now() - start;

  cout << result << ' ' << sec.count() << "[sec] : " << nthr <<  endl;
}
```

うまいこと動いてくれそう...ですがコレ大きな穴が空いています。nが素数の時に++resultしてますが、++resultはresultを読んで「1を加えて／書き戻す処理」が行われます。**読んで**から**書き戻す**までの間に他のスレッドが割り込むとresultの結果が狂ってしまう。data-race（データ競合）と呼ばれ、きわどいタイミングで発生するためになかなか再現しない厄介なバグです。こんなときに「こっからここまで、他のスレッドは入ってくるな（外で待ってろ）！」を実現するのがmutex（mutual exclusion：相互排他）。mutexをlock()してからunlock()までの間、他のスレッドはlock()地点でブロックされます。上の例でちゃんと動かすには、

リスト10

```
  ...
  mutex mtx;
  int result = 0;
  ...
     // nが素数なら resultに1を加える
     if ( is_prime(n) ) {
       mtx.lock();
       ++result;
       mtx.unlock();
     }
  ...
```

あるいは

リスト11

```
  ...
  mutex mtx;
  int result = 0;
  ...
     // nが素数なら resultに1を加える
     if ( is_prime(n) ) {
       lock_guard<mutex> guard(mtx);
       ++result;
     }
  ...
```

lock_guardはコンストラクト時にlock/デストラクト時にunlock()してくれるのでunlock()忘れがなくてお手軽です。途中で例外が発生しても確実にunlock()してくれますし。

また、int,longなどのビルトイン型およびポインタ型に対して++,--,+=,-=,&=,|=などの演算を行う（極めて短い）間だけ、他スレッドの割り込みを抑止する際は高速／軽量なatomicがオススメです。上の例であれば、

リスト12

```
  ...
  atomic<int> result = 0;
  ...
     // nが素数なら resultに1を加える
```

```
    if ( is_prime(n) ) {
      ++result;
    }
  ...
```

でOK。

4：condition_variable

condition_variableを使って、あるスレッドから他のスレッドへイベントを投げることができます。イベントを待ち受けるスレッドでは、

リスト13

```
mutex mtx;
condition_variable cv;

unique_lock<mutex> lock(mtx);
...
cv.wait(lock);
...
```

condition_bariablでwait()するとlockされたmutexをいったん解いて（unlockして）待ち状態となります。イベントを送出するスレッドでは、

リスト14

```
unique_lock<mutex> lock(mtx);
...
cv.notify_all();
...
```

condition_bariablにnotify_all（またはnotify_one）すると待ち受け側のwait()が解けると同時にmutexが再度lockされるというカラクリ。

これを使って素数を勘定している全スレッドの完了を待ち受けてみます。

リスト15

```
void multi(int M, int nthr) {
  vector<thread> thr(nthr);
  div_range<> rng(2,M,nthr);
  condition_variable cond;
  int finished = 0;
  atomic<int> result = 0;
  mutex mtx;

  chrono::system_clock::time_point start = chrono::system_clock::now();
  for ( int t = 0; t < nthr; ++t ) {
    thr[t] = thread([&](int lo, int hi) {
                      for ( int n = lo; n < hi; ++n ) {
                        if ( is_prime(n) ) ++result;
                      }
                      lock_guard<mutex> guard(mtx);
                      ++finished;
                      cond.notify_one();
                    },
                    rng.lo(t), rng.hi(t));
  }
  unique_lock<mutex> lock(mtx);
  // 全スレッドが++finishedすることでfinished==nthrとなるのを待つ
```

```
  cond.wait(lock, [&]() { return finished == nthr;});
  chrono::duration<double> sec = chrono::system_clock::now() - start;

  cout << result << ' ' << sec.count() << "[sec] : " << nthr << endl;
  for ( thread& th : thr ) { th.join(); }

}
```

condition_variableを使った例をもうひとつ。ランデブー（rendezvous）あるいはバリア（barrier）と呼ばれる「待ち合わせ」のからくり。

リスト16

```
class rendezvous {
public:
  rendezvous(unsigned int count)
    : threshold_(count), count_(count), generation_(0) {
    if (count == 0) { throw std::invalid_argument("count cannot be zero."); }
  }

  bool wait() {
    std::unique_lock<std::mutex> lock(mutex_);
    unsigned int gen = generation_;
    if ( --count_ == 0) {
      generation_++;
      count_ = threshold_;
      condition_.notify_all();
      return true;
    }
    condition_.wait(lock, [&](){return gen != generation_;});
    return false;
  }

private:
  std::mutex mutex_;
  std::condition_variable condition_;
  unsigned int threshold_;
  unsigned int count_;
  unsigned int generation_;
};
```

rendezvous r(5);のように、待ち合わせる人数（スレッド数）を引数としてコンストラクトしておきます。各スレッドがr.wait()すると全員が揃うまで待ち状態となり、最後のスレッドがr.wait()した途端全員のブロックが解けて一斉に動き出します。

先ほどの全スレッドの完了待ちをrendezvousで実装すると、

リスト17

```
void multi(int M, int nthr) {
  vector<thread> thr(nthr);
  div_range<> rng(2,M,nthr);
  atomic<int> result = 0;
  rendezvous quit(nthr+1);

  chrono::system_clock::time_point start = chrono::system_clock::now();
  for ( int t = 0; t < nthr; ++t ) {
    thr[t] = thread([&](int lo, int hi) {
                      for ( int n = lo; n < hi; ++n ) {
                        if ( is_prime(n) ) ++result;
```

```
                    }
                    quit.wait();
                  },
                  rng.lo(t), rng.hi(t));
  }
  quit.wait(); // 全スレッドが wait するまで待つ
  chrono::duration<double> sec = chrono::system_clock::now() - start;

  cout << result << ' ' << sec.count() << "[sec] : " << nthr << endl;
  for ( thread& th : thr ) { th.join(); }

}
```

C++11が提供するスレッド･ライブラリをざっくりと紹介しました。いかがでしょう、マルチスレッド･アプリケーションがずっとお手軽に書けるようになりました。CPUメーターが全コア振り切るHigh Performance Computimgのカイカンをお楽しみください。

バックナンバー WEB用を表示

著者プロフィール

επιστημη（エピステーメー）

C++に首まで浸かったプログラマ。Microsoft MVP, Visual C++ (2004.01〜) だったり わんくま同盟でたまにセッションスピーカやったり 中国茶淹れてにわか茶人を気取ってたり、あと Facebook とか。著書: - STL標準講座 (監修) -...

※プロフィールは、執筆時点、または直近の記事の寄稿時点での内容です

ページトップへ