第1章 コア間で整合性を取る方法から その代償フォルス・シェアリングまで

マルチスレッド環境における CPUキャッシュの仕組み

高野 修一

現代のCPUではマルチコアが当たり前です．PC向けのCPUはもちろんシングルボード・コンピュータ用CPUやマイコンに至るまで複数のコアを搭載するのが普通になっています．

それらのハードウェアの性能を引き出すためには，マルチスレッド・プログラミングを行う必要があります．単にマルチスレッドにしただけで性能が上がる，と考えがちですが，性能を引き出すにはCPUのキャッシュ・システムに対する理解が必要不可欠です．

本稿では，キャッシュ・システムの仕組みを知ることで，C++のコードの振る舞いについて説明します．主にPCに使用されるCPUをターゲットにしていますが，シングルボード・コンピュータやスマートフォンなどのCPUも基本的な仕組みは同じです．

マルチスレッドでのCPUの不思議なふるまい

複数のスレッドで独立して何かしらの処理を行う最も簡単な例として**リスト1**のようなC＋＋で書かれたプログラムを考えてみます．本稿のプログラムは次のURLでも参照できます．

`https://github.com/shuichitakano/false_sharing_sample`

配列中の一部の値をインクリメントするだけの簡単な処理を，2つのスレッドで行います．1つ目のスレッドでは配列の最初の要素を，2つ目のスレッドでは引数で指定した要素を処理します．

このプログラムの引数を変えながら実行すると興味深い結果が得られます．Ryzen 7 2700X（AMD）を搭載する筆者のPCでは，引数1で1092ms，引数32のときに389msという結果が得られました．完全に同一の処理を行っているにもかかわらず，なぜこれほどの差が出るのでしょうか．

この現象を理解するにはCPUが持つキャッシュ・メモリの仕組みを理解する必要があります．

リスト1　2つのスレッドで独立した配列要素を更新するサンプル（fs.cpp）

```
int main(int, char **v) {
  using namespace std::chrono;
  using namespace std::chrono_literals;
  alignas(128) std::array<volatile int, 64> a{};
  int idx = std::stoi(v[1]) & 63;
  auto w = [&](int i) {
    for (auto k = 200'000'000ULL; k > 0; --k)
      ++a[i];
  };
  auto s = steady_clock::now();
  std::thread t0(w, 0), t1(w, idx);
  t0.join(); t1.join();
  std::cout << (steady_clock::now() - s) /
      1ms << "ms " << a[0] << ' ' << a[idx] << '\n';
}
```

プログラムの局所性とキャッシュの相性

現代の多くのCPUにはメモリ・アクセスを高速化するためにキャッシュ機構が組み込まれています．キャッシュ・メモリの仕組みを理解するために，メモリとプログラムの特性を簡単に説明します．

● 主として使われる2種類のRAM

メモリには，大きくSRAM（Static RAM）とDRAM（Dynamic RAM）の2種類があります．

▶ランダム・アクセスにも強いSRAM

SRAMは1ビットを6トランジスタのフリップフロップで保持します．ランダム・アクセスが非常に高速で扱いやすいのですが，構成するために必要な回路面積が大きいため高価です．

▶ランダム・アクセスの苦手なDRAM

1ビットを1トランジスタ＋1キャパシタで保持します．セルが極めて小さいため高密度に実装でき，低コストです．しかし，セル電荷が自然に漏れるため定期的なリフレッシュ（記憶保持動作）が必須です．

ランダム・アクセスのレイテンシはSRAMより非常に大きいです．DRAMはセルが2次元配列で並び，アドレスを行（Row）と列（Column）に分けて扱いま