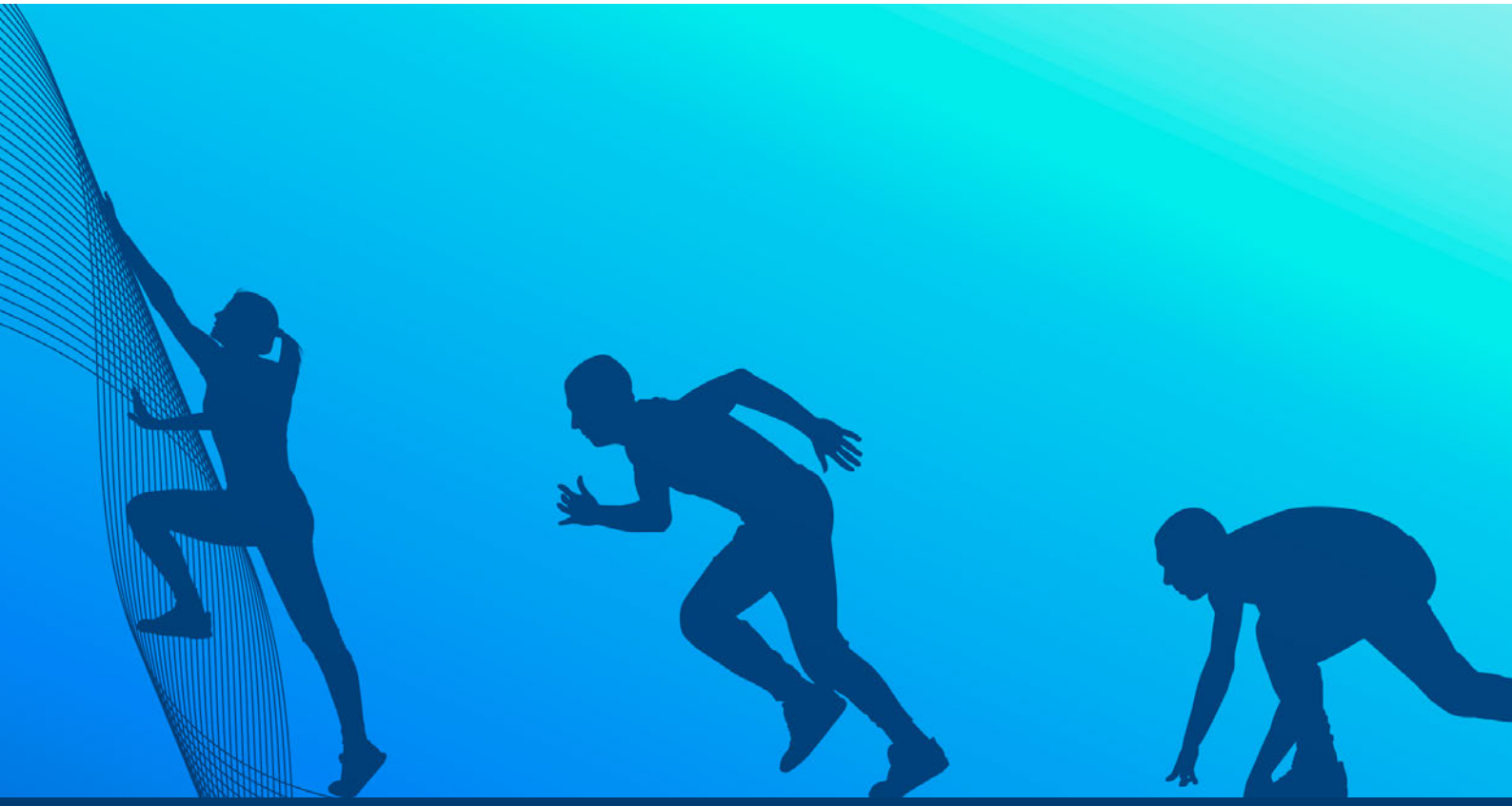


OpenMP* バージョン 4.5: 標準化の進化

ループを容易に並列実行し異種プログラミングをサポート

Michael Klemm インテル コーポレーション シニア・アプリケーション・エンジニア
Christian Terboven アーヘン工科大学 HPC グループ

OpenMP* API 仕様はよく知られるようになり、共有メモリーシステムにおけるマルチスレッド化で広く利用されています。バージョン 4.5 はこの標準化の次のステップへの進化であり、オフロード・プログラミング向けの機能とともに新たな並列プログラミングのコンセプトを導入しています。

歴史

以前のバージョンの OpenMP* API 仕様 (4.0) は、2013 年 7 月に公開されました。このバージョンの主な機能追加は、ホストからコプロセッサーへのオフロード計算による異種プログラミングの基本機能がサポートされたことです。その後、ハイパフォーマンス・コンピューティングの分野で計算能力を高めるため、コプロセス (共同処理) に多くの関心が集まりました。これにより、OpenMP* 4.0 では異種コンピューティングをサポートする上で重要な機能を欠いていることが明らかになりました。また、プログラマーは常に新しい並列化の可能性とより優れた並列処理を記述することを求めています。

OpenMP* 4.5 では、コプロセッサーへのオフロード機能を改善するだけでなく、**並列プログラミング**を容易にする便利な機能を追加することに努めました。この記事では、OpenMP* 4.5 における 3 つの新機能について説明します。

- ループを簡単に並列実行するため、ループでタスクを生成します。
- ロックを使用する際に、ロックにヒントを追加することで並列性を高めます。
- システム上のコプロセッサーを使用するためオフロードを改善しました。

この記事の**ドイツ語版**は、*Heise Developer* オンラインマガジンに掲載されました。

タスク生成ループ

並列ループは、OpenMP* アプリケーションにおける最も重要な領域の 1 つです。これまでのワークシェア構文 (C/C++ の for 文、Fortran の do 文) は、ループ反復をチャンクに分割して並列チーム内の各スレッドにそれらを分配する、スレッド並列ループを表現する非常にシンプルな方式でした。しかしワークシェア構文には、大規模なアプリケーションのコードに対処する際に、並列プログラマーの作業を複雑にするいくつかの制約があります。最も重大な問題の 1 つは、ワークシェア構文の中に入れ子のワークシェア構文を記述することが OpenMP* で禁止されていることです。そのため、並列ループ内では、入れ子になった内部の並列ループを実行するには、新たにスレッドのチームを生成しなければいけません。

```
void taskloop_example() {
#pragma omp taskgroup
    {
#pragma omp task
        long_running_task()  // 同時に実行可能

#pragma omp taskloop collapse(2) grainsize(500) nogroup
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                loop_body();
   }
}
```

1 新しい taskloop 構文で OpenMP* タスクを利用するコード例

新しい **taskloop** 構文は、ループのチャンクを直接ワーカースレッドに割り当てて実行する代わりに、OpenMP* タスクを使用することでこの問題を解決します。OpenMP* タスクは任意の階層に入れ子にできるため、ワークシェア構文の入れ子の制限は適用されません。**図 1** は、OpenMP* タスクと **taskloop** 構文を併用する方法を示します。**long_running_task()** 関数を OpenMP* タスクとして呼び出すことで、その長い実行時間を後続のループと同時に実行することができます。**taskloop** 構文は、ループ反復空間をチャンクに分割し、それぞれのチャンクに対して 1 つの OpenMP* タスクを生成します。処理を終えた任意のワーカースレッドが、それらのタスクをピックアップして実行します。長い時間を要するタスクが早期に終了した場合、その実行スレッドも **taskloop** によって生成されたタスクをピックアップすることができます。タスクを任意の階層に入れ子にできるため、実行中の各タスクで新しいタスクを生成して、アプリケーションの並行性をさらに高めることが可能です。OpenMP* ランタイムシステムは、すべてのタスクが実行を終えるまでワーカー全体のタスクの負荷バランスを調整します。

```
template<class K, class V>
struct hash_map {

  hash_map() {
    omp_init_lock(&lock);
  }

  ~hash_map() {
    omp_destroy_lock(&lock);
   }

  V& find(const K& key) const {
    V* ret = 0;
    omp_set_lock(lock);
    ret = internal_find(key);
    omp_unset_lock(lock);
    return *ret;
  }

  void insert(const K& key, const V& value)
{
    omp_set_lock(lock);
    internal_insert(key, value);
    omp_unset_lock(lock);
  }
  //...

private:
  mutable omp_lock_t lock;
  hash_buckets *buckets;
  // ...
};
```

2 キーを値にマップするハッシュマップの例

taskloop 構文は、ワークシェア構文とタスク構文の両方からシンタックスを継承します。データの属性 (**shared**、**private**、**firstprivate**、および **lastprivate**) を制御する通常の節に加えて、タスク節 (**final** と **mergeable**) をサポートしています。さらに、nogroup 節もサポートします。この節は、構文によって生成されたすべてのタスクを自動的に同期する、暗黙的なタスクのグループを非アクティブ化します。例では、長時間実行されるタスクと **taskloop** 構文の実行を同期するため、明示的な **taskgroup** 構文を使用しています。

生成されるタスクのサイズは、**grainsize** 節で制御できます。これは、生成されるタスクにループ反復をいくつ割り当てるか定義します。プログラマーがタスク数の指定を望む場合、代わりに **num_tasks** 節を使用できます。さらに、完全に入れ子になったループを collapse 節を使用して 1 つのループにまとめることができます。これは、ワークシェア構文における collapse 節の作用と同じです。

OpenMP* 4.5 では、OpenMP* タスク構文 **task** と **taskloop** に **priority** 節が追加されています。この節は、ランタイムシステムによるタスクの実行順序に影響します。priority 節には正の整数値を指定します。値が大きいほど生成されるタスクの優先順位は高くなります。低い優先順位のタスクの前に高い順位のタスクを実行する必要があることを、ランタイムシステムへ知らせるヒントとして使用されます。しかし、タスクの実行に際して特定の順番を完全に保証することはできないため、OpenMP* の実装ではヒントに固執する必要はありません。priority 節が省略されると、デフォルトでゼロ (0) が仮定されます。

Locks、Locks、Locks

ロックの取得と解除による相互排他は、大部分の並列プログラムで避けられない必要悪です。競合状態やデータ衝突を回避するため、共有リソースにアクセスするコード領域に入る前にロックを取得しなければいけません。ロックによるリソースの保護には、実行をシリアル化するため並列処理を制限するという代償が伴います。アプリケーションによっては、共有リソースへの同時アクセスが衝突する可能性が非常に低い (ゼロではない) にもかかわらず、安全性の理由からロックが利用されることがあります。

```
template<class K, class V>
struct hash_map {

  hash_map() {
    omp_init_lock_with_hint(&lock,
      kmp_lock_hint_hle);
  }

  ~hash_map() {
    omp_destroy_lock(&lock);
  }

  V& find(const K& key) const {
    V* ret = 0;
    omp_set_lock(lock);
    ret = internal_find(key);
    omp_unset_lock(lock);
    return *ret;
  }

  void insert(const K& key, const V& value)
{
    omp_set_lock(lock);
    internal_insert(key, value);
     omp_unset_lock(lock);
  }

private:
    mutable omp_lock_t lock;
    hash_buckets *buckets;
};
```

3 クリティカル領域を楽観的に実行するため、投機的なロックを使用

図 2 のコード例は、型 **V** の値に型 **K** のキーをマッピングするハッシュマップを単純化した非効率な実装です。エラー処理やほかの最適化を実装しない簡単な例ですが、このコードはロックに関する問題を抱えています。排他制御は、**hash_map** クラスの各メソッドに入った後に直ちにロックを取得することで有効になります。すると、個々のハッシュバケットの潜在的な競合状態を避けるため、実行がシリアル化されます。ハッシュ関数とデータ構造は、できるだけアクセス競合を回避するように設計されますが、これはときには不要なこともあります。各スレッドが異なるハッシュバケットや要素上で動作している場合、複数のスレッドがハッシュマップを操作するコード領域に入るのは安全であると考えられます。一般的な実装では、個々のバケットや要素を保護するロックを使用するか、ロックなしのデータ構造が採用されます。いずれの場合でも、スケーラブルかつ効率良いソリューションを見つけるため、プログラマーはかなりの作業を強いられます。

OpenMP* 4.5 は、この負担を軽減する新機能を提供します。プログラマーは、新しい API を使用して OpenMP* ランタイムにロックのヒントを渡すことで、アプリケーション・コード内のロックの使用目的を知らせることができます。ロックを初期化する 2 つの新しい関数、**`omp_init_lock_with_hint`** と **`omp_init_nest_lock_with_hint`** が定義されています。これらの関数は、型 **`omp_lock_hint_t`** の追加の引数を受け付けます (ロックのヒントについては、**表 1** をご覧ください)。ほかの OpenMP* のヒントと同様に、意図するロックの実装を最適化するためヒントを使用することができます。例えば、OpenMP* の実装は、テストとセットロックを futex (fast userspace mutex: 高速ユーザー空間 mutex の略) ベースのロックで置き換えることができます。また、特別なハードウェア命令 (例えば、インテル® トランザクショナル・シンクロナイゼーション・エクステンション: インテル® TSX) を利用することもできます。すべての状況において、プログラムの動作に影響を与えないようロックのセマンティクスは保持されます。

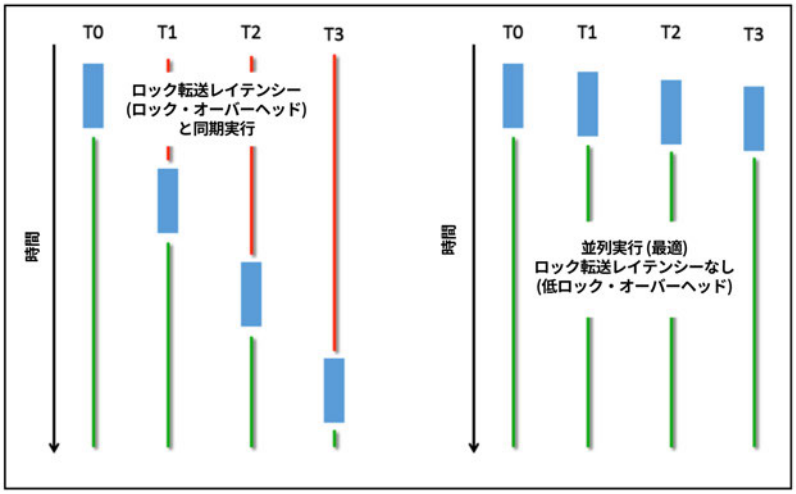
ヒントは整数式で表します。C/C++ では "|" オペレーターで、Fortran では "+" オペレーターで組み合わせることができます。これは、特定のロックを使用する際に、状況に応じた表現が可能となるため、プログラマーの柔軟性が高まります。OpenMP* 規格は、実装において事前定義されたヒントを追加拡張することを許可しています。インテルの OpenMP* ランタイムでは、**表 2** の追加のヒントが定義されています。

ヒント	セマンティクス
omp_lock_hint_none	OpenMP* ランタイムは、自由にロックの実装を選択できます。
omp_lock_hint_uncontended	スレッドが同時にロックにアクセスすることは少なく、競合の可能性は低いです。
omp_lock_hint_contended	複数のスレッドが同時にロックを取得しようとするため、頻繁に競合するロックを最適化します。
omp_lock_hint_ nonspeculative	楽観的なロックを使用しないことを指示します。取得するスレッドのワーキングセットがオーバーラップするため、かなりの競合があります。
omp_lock_hint_speculative	楽観的なロックを使用することを指示します。スレッドのワーキングセットは、ほとんど競合しないことが期待できます。

表 1. OpenMP* 4.5 でサポートされるロックのヒント

ヒント	セマンティクス
kmp_lock_hint_hle	インテル® TSX のハードウェア・ロック省略機能を使用します。
kmp_lock_hint_rtm	インテル® TSX の RTM 機能を実装したロックを使用します。
kmp_lock_hint_adaptive	競合をチェックしてあまりにも多くの衝突が発生する場合は、伝統的なテスト & セットロックへフォールバックする投機的な適応ロックを使用します。

表 2. インテルの OpenMP* ランタイムで定義される追加のロックのヒント



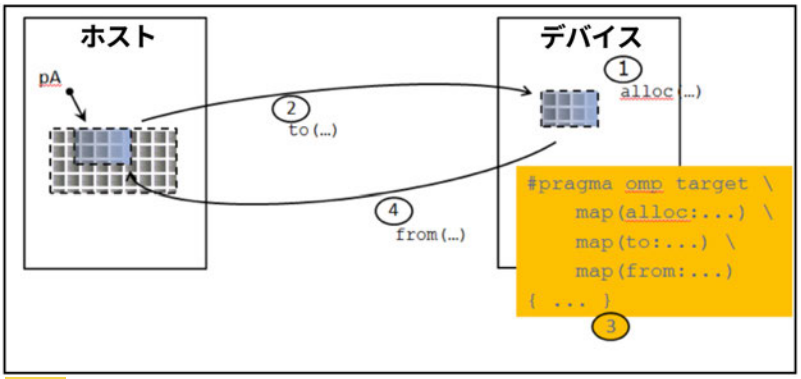
4 伝統的な相互排他 (左) と楽観的な相互排他 (右)

図 3 のコードでは、プログラマーはスレッドがハッシュマップにアクセスする際に競合しないと想定し、ロックを投機的に実行することを OpenMP* ランタイムに伝えるため新しい機能を使用しています。インテル® TSX が利用可能なプロセッサーでは、ハードウェアはロックを取得/解放する代わりに楽観的にロックを無視します (**図 4** を参照)。ハードウェアがロックのセマンティクスの違反を検出した場合にのみ (例えば、あるスレッドがハッシュバケットを変更している間に、ほかのスレッドがそれを読み取った場合)、プロセッサーは遡って排他制御のため伝統的なロックによるコードを再実行します。

投機的なロックを使用する目安として、2 つの基本的なケースが考えられます。1 つは、ロックが競合せず (ハッシュテーブルの例のように) 多くの衝突が発生しない場合に、インテル® TSX と投機的なロックを使用するケースです。2 つ目は、読み取りが多く、書き込みのロック競合がわずかであり、競合が発生しても衝突がほとんどない場合に、投機的なロックによりロックのオーバーヘッドが軽減できるケースです。

オフロード

OpenMP* 4.0 では、コプロセッサーなどの接続された計算デバイスへのオフロードにより、OpenMP* によるマルチスレッド化の範囲が拡がりました。**`target`** 構文は、制御フローをホストスレッドからコプロセッサーへ切り替えます。プログラマーは、**`map`** 節を使用して転送するデータ・オブジェクトと、転送の方向を指定できます (**図 5** を参照)。通常、オフロードされるコード領域はカーネルと呼ばれ、ターゲットデバイスの特性を活用する大規模な並列処理の断片です。



5 データ転送を含むオフロードされたコードの実行モデル

OpenMP* 4.0 では、ホストとターゲット間で無駄なデータ転送を避けるため、異なるオフロードでデータを保持できるデバイスデータ環境を導入しました。**図 6** は、**kernel1** と **kernel2** の呼び出し中に、ターゲットデバイス上の配列 **var1** を継続して利用するデータ領域の例を示しています。データ領域の存続期間は、**target data** 構文に関連付けられたスコープの構造化ブロックに依存します。コードが波括弧 "{" に到達すると、データ環境が作成されデータが転送されます。波括弧 "}" に到達すると、データ環境は破棄されます。これは、複数のカーネルに渡ってデータ環境を維持するための実装を可能にしますが、新たなデータのマッピングや非構造化データのマッピング (例えば、C++ クラスのコンストラクターやデストラクターでの) を禁止します。

```
double var1[N];

void offload_example() {
#pragma omp target data map(tofrom:var1[:N])
    {

        C *c = new C();

#pragma omp target
        c.kernel1();    // var1 と C のメンバーを使用する

#pragma omp target
        c.kernel2();    // var1 と C のメンバーを使用する

        delete c;
    }
}
```

6 **target data** 構文のスコープにバインドされたデバイスデータ環境

OpenMP* 4.5 では、この制限を解除するため新たな構文が追加されました。target enter data と target exit data ディレクティブは、ターゲットデバイス上のデータマッピングを作成および廃棄します。

```
#pragma omp target enter data map(map-type: var-list) [clauses]
#pragma omp target exit data map(map-type: var-list) [clauses]
```

図 7 は、新しいディレクティブを使用して、C++ オブジェクトのコンストラクターとデストラクターでデータ環境を作成および破棄する方法を示しています。**C** クラスのインスタンスが構築されると、メンバーの値がターゲットデータ環境にマッピングされます。オブジェクトが破棄されると、デストラクターは値に関連付けられたデータ環境を破棄します。

その他の拡張は、構造体データ型の要素のマッピングに関するものです。OpenMP* 4.0 では、スカラー変数、配列、またはビット単位でコピー可能なデータ構造体のマッピングのみ許可していました。OpenMP* 4.5 では、構造体データ型のメンバーの部分的なマッピングに対応するため、データ転送が拡張されました。**図 8** にいくつかの新しい可能性を示します。

ついに、OpenMP* 4.5 では非同期オフロードがサポートされました。OpenMP* 4.0 の実装では、ホストスレッドは処理を継続する前に、オフロードされたコード領域の実行完了を待機していました。OpenMP* 4.5 では、**nowait** 節を指定して、ホストのスレッドと **target** 構文の OpenMP* タスクを同時に実行することができます。**target enter data** と **target exit data** ディレクティブも同様に、非同期データ転送を可能にする新しい節をサポートします。

非同期オフロードとデータ転送は、通常の OpenMP* タスクであるため、この新しい機能はホスト上で実行されているほかの OpenMP* タスクと非同期実行を同期する **depend** 節を継承します。**図 9** では、ホストの実行と非同期データ転送がオーバーラップしています。**depend** 節は、データ転送が完了するまでカーネルの実行を待機させます。

```
class C {

 public:
    C() {
#pragma omp target enter data map(alloc:values[M])
    }

    ~C() {
#pragma omp target exit data map(delete:values[M])
    }
 private:
    double *values;
};
```

7 C++ オブジェクトにおけるデバイスデータ環境の作成と破棄

```
struct A {
   int field;
   double array [N];
} a;

#pragma omp target map(a.field)
#pragma omp target map(a.array[23:42])
```

8 構造体データ型の要素のマッピング

```
double data[N];

void synchronization_example() {
#pragma omp target enter data map(to:data[N]) \
            depend(out:data[0]) nowait

    do_something_on_the_host_1();

#pragma omp target depend(inout:data[0]) nowait
    perform_kernel_on_device();

    do_something_on_the_host_2();

#pragma omp target exit data map(from:data[N]) \
            depend(inout:data[0])

#pragma omp task depend(in:data[0])
    task_on_the_host(data);

    do_something_on_the_host_3();
}
```

9 非同期オフロードとデータ転送、ホストスレッドとの同期

また、カーネルの実行が完了する前に、デバイスからホストにデータ転送が行われないようにします。最後に、ホストスレッドは、`do_something_on_the_host_3()` でホストコードを同時実行する前に、最後のデータ転送完了を待機するタスクを生成します。

まとめ

OpenMP* 4.5 は、いくつかの改良に加えて、プログラマーがより良い並列処理を表現し、ホスト・アプリケーションとオフロードコードの両方のパフォーマンス向上を可能にする機能を提供します。タスクを生成するループは、並列ループに起因する負荷バランスと構成の容易性の問題を解決し、プログラマーの負担を軽減します。ヒント付きのロックのサポートは、プログラマーがアプリケーションのロックの動作を最適化し、ハードウェア・トランザクショナル・メモリーに対応した現代のプロセッサーを活用できるようにします。最後に、オフロードの拡張機能は、ホストとコプロセッサーの両方で、計算と通信をオーバーラップする非同期実行を可能にします。