現在と将来の OPENMP* API 仕様

代表的な並列プログラミング言語の各バージョンにおける進化

**Michael Klemm Intel Deutschland GmbH シニア・スタッフ・アプリケーション・エンジニア、
Alejandro Duran Intel Corporation Iberia アプリケーション・エンジニア、
Ravi Narayanaswamy インテル コーポレーション シニアスタッフ開発エンジニア、
Xinmin Tian 同シニア主任エンジニア、Terry Wilmarth 同シニア開発エンジニア**

OpenMP* API には 20 年の歴史があります。その登場以来、ハードウェアとソフトウェアの進歩とともに、新しいハードウェアのプログラミングに対応するさまざまな機能が追加されてきました。2013 年にリリースされたバージョン 4.0 から、OpenMP* 言語はヘテロジニアス・プログラミングと SIMD プログラミングをサポートしました。同様に、2008 年には、タスク構文の追加により、不規則な並列処理を含むプログラムのサポートが向上しました。OpenMP* Technical Report 4: Version 5.0 Preview 1 (略称 TR4) は、OpenMP* 言語の進化における次のステップです。タスク・リダクションの追加、SIMD 並列処理の拡張が行われ、ヘテロジニアス・プログラミングの生産性が大幅に向上します。この記事では、既存の OpenMP* 機能を確認した後、TR4 をサポートする実装で追加される機能を紹介します。

タスクベースのプログラミング : タスクで表現する

タスクベースのプログラミングは、不規則な並列処理 (再帰アルゴリズム、グラフのトラバース、非構造化データを操作するアルゴリズムなど) が必要なアプリケーションにとって重要な概念です。OpenMP* バージョン 3.0 から、OpenMP* ランタイムシステムのスケジューラーに渡される小さな単位のワークの並列実行を容易に表現する方法として、タスク構文が提供されました。

```
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* タスクを利用する例

図 1 は、long_running_task 関数を実行する OpenMP* タスクを生成し、その後 **taskloop** 構文を使用してループを並列化する例を示しています。この構文は OpenMP* 4.5 で追加されたもので、プログラマーが OpenMP* タスクを使用してループを簡単に並列化できる糖衣構文を提供します。この構文は、ループの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 つのタスクを生成します。細かい制御ができるように、いくつかの節 (例えば、タスクあたりのワークの量を制御する **grainsize** や i ループと j ループを 1 つのループにまとめる **collapse**) をサポートしています。TR4 では、**taskgroup**、**task**、**taskloop** 構文に生成されたタスク間でリダクションを行う新しい節を定義することにより、OpenMP* タスクの表現が拡張されます。

図 2 は、リンクリストを処理してリストの全要素の最小値を調べるタスクの例を示しています。**parallel** 構文は、ワーカースレッドがタスク実行で利用できるように並列領域を作成します。**single** 構文は、リンクリストのトラバースを 1 つのスレッドの実行に制限し、**omp** タスクにより各リスト項目に 1 つのタスクを生成します。これは OpenMP* で生産と消費パターンを実装する一般的な方法です。

TR4 のタスク・リダクションは、OpenMP* バージョン 4.0 で追加された **taskgroup** 構文を使用します。この構文は、タスクを論理的にグループ化し、グループのすべてのタスクの完了を待機する方法を提供します。TR4 では、**図 2** に示すように、**task_reduction** 節によりリダクションを行うように、**taskgroup** 構文が拡張されます。この節が構文に追加されると、**taskgroup** 領域の最後で個々のタスクにより収集された部分結果がすべて集計され、最終的な結果が生成されます。リダクション操作に関係するタスクには、**taskgroup** の **reduction** 節と一致する **in_reduction** 節が必要です。

TR4 から、**taskloop** 構文はタスク・リダクション・セマンティクスによる **reduction** 節および **in_reduction** 節をサポートします。**reduction** 節が **taskloop** 構文に含まれていると、ループの最後で要求されたリダクション操作を実行する、暗黙的なタスクグループが作成されます。**in_reduction** 節が追加されると、**taskloop** 構文により生成されたタスクは外側の **taskgroup** 領域のリダクションに関係します。

```
int find_minimum(list_t * list) {
    int minimum = INT_MAX;
    list_t * ptr = list;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup task_reduction(min:minimum)
    {
        for (ptr = list; ptr ; ptr = ptr->next) {
#pragma omp task firstprivate(ptr) in_reduction(min:minimum)
            {
                int element = ptr->element;
                minimum = (element < minimum) ? element : minimum;
            }
        }
    }
    return minimum;
}
```

2 タスク・リダクションを使用したリンクリストのトラバースと最小値の計算

オフロード : コプロセッサーを最大限に利用する

OpenMP* API は、ユーザーのフィードバックに基づいてオフロードプラグマが使いやすくなるように努めています。これにより、新機能が TR4 に追加され、いくつかの既存の機能が強化されました。重要な新機能の 1 つは、オフロード領域で呼び出されている関数を自動的に検出して、それらの関数が **declare target** ディレクティブで指定されているかのように扱います。以前は、オフロード領域で呼び出されるすべての関数は、**declare target** ディレクティブによって明示的にタグ付けされている必要がありました。特にプログラマーが関係していないヘッダーファイルにルーチンが含まれている場合 (標準テンプレート・ライブラリーなど)、ヘッダーファイル自体に **declare target** ディレクティブが必要になるため (その結果、一部の関数がオフロード領域で使用されていない場合でもデバイスのヘッダーファイルにすべての関数のコピーを作成する必要があり)、これは大変な作業でした。

```
#pragma omp declare target
void foo() {
    // ...
}
#pragma omp end declare target

void bar()  {
#pragma omp target
    {
        foo();
    }
}
```

```
void foo() {
    // ...
}


void bar() {
#pragma omp target
    {
        foo();
    }
}
```

3 オフロード領域で使用される関数を自動的に検出する

OpenMP* バージョン 4.5 では、**図 3** の左側のようにコードを記述する必要があります。TR4 では、右側に示すコードでデバイス関数を暗黙的に検出して作成できます。

TR4 では、静的記憶域を含む変数も自動的に検出できます。これにより、**図 4** の 2 つの例は等価となります。

```
int x;
#pragma omp declare target to (x)

void bar() {
    #pragma omp target
    {
        x = 5;
    }
}
```

```
int x;


void bar() {
    #pragma omp target
    {
        x = 5;
    }
}
```

4 静的記憶域を含む変数を自動的に検出する

OpenMP* バージョン 4.5 では、**use_device_ptr** 節が追加されました。**use_device_ptr** の変数は、使用する前にマップする必要があります。変数は 1 つのデータ節にのみ記述できるため、プログラマーは個別の **#pragma target data** 節を記述する必要があります。そのため、**図 5** の OpenMP* ディレクティブが必要になります。

```
#pragma omp target data map(buf)
#pragma omp target data use_device_ptr(buf)
```

5 変数のマップ

TR4 では、**図 6** に示すように、単一構文で変数を **map** 節と **use_device_ptr** 節の両方に記述できる例外が追加されました。

```
#pragma omp target data map(buf) use_device_ptr(buf)
```

6 変数を **map** 節と **use_device_ptr** 節の両方に記述できる

スタティック・データ・メンバーが **omp declare target** 構文内のクラスで使用できるようになりました。また、スタティック・メンバーを含むクラス・オブジェクトは map 節でも使用できます **(図 7)**。

```
#pragma omp declare target
class C {
    static int x;
    int y;
}
class C  myclass;
#pragma omp end declare target

void bar() {
#pragma omp target map(myclass)
    {
        myclass.x = 10
    }
}
```

7 **map** 節で使用されるスタティック・メンバーを含むクラス・オブジェクト

さらに、仮想メンバー関数を **omp declare target** 構文内のクラスまたは **map** 節で使用されているオブジェクトで使用できるようになりました。唯一の注意点は、オブジェクトが同じデバイスで作成される場合、仮想メンバー関数はそのデバイス上でのみ呼び出すことができる点です。

OpenMP* 4.5 では、最初の構造がターゲットである結合構造の **reduction** 節または **lastprivate** 節で使用されるスカラー変数は、ターゲット構造の **firstprivate** として扱われます。ホストの変数が更新されることはありません。ホストの変数を更新するには、プログラマーは結合構造から **omp target** ディレクティブを分離してスカラー変数を明示的にマップする必要があります。TR4 では、これらの変数は自動的に **map(tofrom:variable)** が適用されているかのように扱われます。

名前付き配列のセクションを **omp target** データを使用してマップする場合、配列を参照する **omp target data** 構文内の入れ子の **omp target** は、同じセクションまたは外部の **omp target data map** 節で使用される配列のサブセクションに暗黙的にマップする必要があります。内側の **omp target** 領域で明示的なマップが省略されると、暗黙的なマップ規則が適用され、配列全体が OpenMP* バージョン 4.5 の仕様に従ってマップされます。これにより、配列のサブセクションがすでにマップされている場合、大きなサイズの配列をマップするとランタイムエラーが発生します。同様に、外側の **omp target data** 構文内で構造体変数のフィールドをマップして、内側の入れ子の **omp target** 構文内で構造体変数のアドレスを使用すると、構造体の一部がすでにマップされている場合、構造体変数全体をマップしようとします。TR4 では、プログラマーが想定した動作になるように、これらのケースが修正されました **(図 8)**。

```
struct {int x,y,z} st;
int A[100];
#pragma omp target data map(s.x A[10:50])
{
#pragma omp target
    {
        A[20] = ;       // OpenMP* 4.5 ではエラー、TR4 では OK
        foo(&st);       // OpenMP* 4.5 ではエラー、TR4 では OK
    }
#pragma omp target map(s.x, A[10:50])
    {
        A[20] = ;       // OpenMP* 4.5 と TR4 の両方で OK
        foo(&st);       // OpenMP* 4.5 と TR4 の両方で OK
    }
}
```

8 改良されたマップ処理

TR4 の新機能により、OpenMP* を使用したオフロードのプログラミングの容易性が向上し、アプリケーションに要求される変更点が少なくなります。**target** 領域で使用される変数と関数は自動的に検出されるため、明示的に指定する必要はなくなります。同様に、入れ子の領域の内側で **map** 節を繰り返す必要性がなくなります。**map** 節と **use_device_ptr** 節の両方に変数を記述できるため、必要な OpenMP* ディレクティブの数が減ります。リダクション変数の動作は、プログラマーの想定に沿うように変更されます。全体的に、セマンティクスが簡潔になり、OpenMP* アプリケーション内でより簡単に、さらに直感的にオフロードデバイスを利用できるようになります。

効率的な SIMD プログラミング

繰り返し間の依存関係がある SIMD ループ

OpenMP* バージョン 4.5 では、**ordered** 構文に新しく **simd** 節が追加され機能が拡張されました。**ordered simd** 構造は、SIMD ループまたは SIMD 関数の構造化ブロックを反復順または関数呼び出しの順で実行することを宣言します。**図 9** の左のコード例は、**ordered simd** ブロックを使用して各反復内および反復間の読み取り / 書き込み、書き込み / 読み取り、書き込み / 書き込みの順序を保存する方法を示しています。すべてのループは SIMD 命令を使用して同時に実行できます。最初の **ordered simd** ブロックでは、配列のインデックス **ind[i]** の書き込み / 書き込みが重複する可能性があるため (例えば、**ind[0] = 2, ind[2] = 2**)、すべてのループが**ベクトル化**できるように **ordered simd** でシリアル化する必要があります。2 つ目の **ordered simd** ブロックでは、**myLock(L)** 操作と **myUnlock(L)** 操作を 1 つの **ordered simd** ブロックで行う必要があります。そうしないと、ループベクトル化の一部として (例えば、ベクトル長 2 の場合)、**myLock(L)** と **myUnlock(L)** の呼び出しが **{myLock(L); myLock(L); …; myUnlock(L); myUnlock(L);}** のように 2 つの呼び出しになります。一般に、ロック関数を入れ子にするとデッドロックが発生します。例で示している **ordered simd** 構文は、適切なシーケンス **{myLock(L); …; myUnlock(L); …; myLock(L); myUnlock(L);}** を作成します。

```
#pragma omp simd
for (i = 0; i < N; i++) {
    // ...
#pragma omp ordered simd
    {
        // 書き込み/書き込みの重複
        a[ind[i]] += b[i];
    }
    // ...
#pragma omp ordered simd
    {
       // アトミック更新
       myLock(L)
       if (x > 10) x = 0;
       myUnlock(L)
    }
    // ...
}
```

```
#pragma omp simd
for (i = 0; i < N; i++) {
    // ...
#pragma omp ordered simd
    {
        if (c[i] > 0) q[j++] = b[i];
    }
    // ...
#pragma omp ordered simd
    {
        if (c[i] > 0) q[j++] = d[i];
    }
    // ...
}
```

9 各反復内および反復間の読み取り / 書き込み、書き込み / 読み取り、書き込み / 書き込みの順序を保存する

ordered 構文で **simd** 節を使用する場合、2 つの **ordered simd** ブロック間の固有の依存関係に違反しないように注意してください。**図 9** の右のコード例は、**#pragma omp ordered simd** の誤った使い方 (シリアル実行時の格納順が SIMD 実行で変更される) を示しています。**c[0] = true** および **c[1] = true** と仮定します。上記のループがシリアルに実行された場合、格納順は **q[0] = b[0], q[1] = d[0], q[2] = b[1], q[3] = d[1], ...** になります。しかし、ループがベクトル長 2 で同時に実行された場合、格納順は **q[0] = b[0], q[1] = b[1], q[2] = d[0], q[3] = d[1], ...** になります。格納順の違いは、ループの 2 つの **ordered simd** ブロック間の変数 **j** の書き込み / 読み取り依存関係の違反によるものです。正しい使い方は、2 つの **ordered simd** ブロックを 1 つの **ordered simd** ブロックにマージすることです。

ref/uval/val 修飾子を linear 節に追加

linear 節は **private** 節で指定される機能のスーパーセットを提供します。**linear** 節が構文で指定されると、関連するループの各反復の新しいリスト項目の値は、構文に入る前のオリジナルのリスト項目の値に対応し、反復の論理数と linear ステップを掛けた値が加えられます。そして、関連するループの連続する最後の反復に対応する値がオリジナルのリスト項目に割り当てられます。**linear** 節が宣言型のディレクティブで指定された場合、すべてのリスト項目は各 SIMD レーンで同時に呼び出される関数の仮引数になります。

ref/uval/val 修飾子を **linear** 節に追加した理由は、コンパイラーがギャザー / スキャッターの代わりにユニットストライド方式のロード / ストアを使用して効率的な SIMD コードを生成できるように、アドレスとデータ値に関してメモリー参照の **linear** または **uniform** プロパティーを正確に指定する方法をプログラマーに提供するためです。基本的に、暗黙的に参照される linear 引数は、linear として参照するほうが良いでしょう。**uval/val/ref** のセマンティクスは次のようになります。

- **linear(val(var):[step])** は、**var** が参照で渡されている場合でも値が linear であることを示します。アドレスのベクトルは参照渡しで渡されます。この場合、コンパイラーはギャザーまたはスキャッターを生成する必要があります。
- **linear(uval(var):[step])** は、参照で渡される値は linear で、参照自体は uniform であることを示します。そのため、最初のレーンの参照は渡されますが、ほかの値は step を使用して構築します。コンパイラーは、汎用レジスターを使用してベースアドレスを渡し、その linear 値を計算します。
- **linear(ref(var):step)** は、パラメーターは参照で渡され、根本的な参照は linear で、メモリーアクセスは step の値に応じて linear ユニットストライドまたは非ストライドであることを示します。コンパイラーは、汎用レジスターを使用してベースアドレスを渡し、その linear アドレスを計算します。

図 10 は、関数 **FOO** と引数 X および Y (Fortran で参照で渡される) を示しています。“**VALUE**” 属性はこの動作を変更しません。Fortran 2008 言語仕様では更新された値が呼び出し元に見えないことを示すだけです。X と Y の参照は linear として示されていないため、コンパイラーは、ベクトル長が 4 であると仮定して、(X0, X1, X2, X3) および (Y0, Y1, Y2, Y3) をロードするギャザー命令を生成する必要があります。**図 11** では、X と Y の参照が linear として示されているため、コンパイラーはパフォーマンスに優れたユニットストライド方式の SIMD ロードを生成することができます。

```
    REAL FUNCTION FOO(X, Y)
!$omp declare simd(FOO)
      REAL, VALUE :: Y     !! 参照渡し
      REAL, VALUE :: X     !! 参照渡し
      FOO = X + Y          !! アドレスのベクトルに
                           !! 基づいて生成された
                           !! ギャザー
      END FUNCTION FOO
      ! ...
!omp$ simd private(X,Y)
      DO I= 0, N
        Y = B(I)
        X = A(I)
        C(I) += FOO(X, Y)
      ENDDO
```

10 参照 X と Y の直線性がコンパイル時に不明

```
      REAL FUNCTION FOO(X, Y)
!$omp declare simd(FOO) linear(ref(X), ref(Y))
      REAL, VALUE :: Y     !! 参照渡し
      REAL, VALUE :: X     !! 参照渡し
      FOO = X + Y          !! ユニットストライド
                           !! SIMD ロード
      END FUNCTION FOO
      ! ...
!omp$ simd private(X,Y)
      DO I= 0, N
        Y = B(I)
        X = A(I)
        C(I) += FOO(X, Y)
      ENDDO
```

11 X と Y の参照が linear として示されている

図 12 で、関数 **add_one** は SIMD 関数として示され、C++ の参照引数 **const int &p** を含んでいます。**p** が **linear(ref(p))** として示されている場合、コンパイラーはベクトル長を 4 と仮定し、**rax** レジスターのベースアドレス **p** でユニットストライド方式のロード命令を生成して、**p[0]**、**p[1]**、**p[2]**、**p[3]** を **xmm0** レジスターにロードします。 この場合、**add_one** 関数で必要な命令は 3 つのみです。

```
#pragma omp declare simd notinbranch // linear(ref(p))
__declspec(noinline)
int add_one(const int& p) {
    return (p + 1);
}
```

12 **linear(ref(p))** アノテーションあり / なしの SIMD コードの比較

しかし、**p** が **linear(ref(p))** として示されていない場合、コンパイラーは 4 つの異なるアドレス **p0**、**p1**、**p2**、**p3** が 2 つの **xmm** レジスターで渡され、ギャザー操作がスカラーロードとパック命令のシーケンスでエミュレートされていると仮定します。その結果、**add_one** 関数で必要な命令は 3 つではなく 16 になります。

全体的に、OpenMP* バージョン 4.5 で追加された SIMD 機能を利用すると、ユーザーは、より詳細な情報をコンパイラーに提供できます。その結果、コンパイラーは、多くの状況で、より多くのループをベクトル化し、より優れたベクトルコードを生成します。

アフィニティー : スレッドの配置が簡単に

OpenMP* バージョン 4.0 仕様では、スレッド・アフィニティーを制御する標準的な方法が初めて提供され、言語に 2 つの新しい概念をもたらしました。

1. バインドポリシー
2. プレース・パーティション

バインドポリシー (**bind-var** 内部制御変数 (ICV) で指定) は、チームのスレッドがどこで親スレッドのプレースにバインドされるか決定します。プレース・パーティション (**place-partition-var** ICV で指定) は、スレッドをバインドできる場所のセットです。スレッドは、いったん指定されたチームのプレースにバインドされたら、そのプレースから移動すべきではありません。

仕様では、**master**、**close**、**spread** の 3 つのバインドポリシーが定義されています。これらのポリシーの説明では、4 つのプレース (それぞれ 1 コアと 2 スレッド) のセットについて考えます。これらのプレースに 3 スレッドおよび 6 スレッドを配置する例を示します。親スレッドは常に 3 つ目のプレースに配置されると仮定します。**master** ポリシーでは、マスタースレッドは親スレッドのプレースにバインドされ、チームの残りのスレッドはマスタースレッドと同じプレースに割り当てられます (**表 1**)。

	プレース 1: {0,1}	プレース 2: {2,3}	プレース 3: {4,5} (親)	プレース 4: {6,7}
3 スレッド			0、1、2	
6 スレッド			0、1、2、3、4、5	

表 1. **master** ポリシーのスレッド配置

close ポリシーは、親スレッドのプレースにマスタースレッドを配置して開始した後、チームの残りのスレッドをラウンドロビン方式で処理します。**P** 個のプレースに **T** スレッドを配置するには、マスターのプレースで最初の **T/P** スレッドを取得した後、プレース・パーティションの次のプレースで次の **T/P** スレッドを取得し (以後同様、必要に応じてラップアラウンドされます)、スレッドを分散します (**表 2**)。

	プレース 1: {0,1}	プレース 2: {2,3}	プレース 3: {4,5} (親)	プレース 4: {6,7}
3 スレッド	2		0	1
6 スレッド	4	5	0,1	2,3

表 2. **close** ポリシーのスレッド配置

spread ポリシーでは、非常に興味深い処理になります。スレッドの配置は利用可能なプレースに拡散されます。プレース・パーティションのサブパーティション (**T >= P** の場合は **P** パーティション) で **T** がほぼ均等になるようにします。**T <= P** の場合、マスタースレッドから順に各スレッドは独自のサブパーティションを取得し、親スレッドがバインドされるプレースを含むサブパーティションを取得します。各後続スレッドは、各後続サブパーティションの最初のプレースにバインドされます (必要に応じてラップアラウンドされます)。**T > P** の場合、連続するスレッドのセットが同じサブパーティション (このケースでは 1 つのプレース) を取得します。そのため、セットのすべてのスレッドは同じプレースにバインドされます。中括弧形式のサブパーティションを**表 3** に示します。入れ子の並列処理が使用される場合、各入れ子の並列領域で使用される利用可能なリソースに影響するため、これらは重要です。

	プレース 1: {0,1}	プレース 2: {2,3}	プレース 3: {4,5} (親)	プレース 4: {6,7}
3 スレッド	1 {{0,1}}	2 {{2,3}}	0 {{4,5},{6,7}} 注: 0 は {4,5} にバインドされる	
6 スレッド	4 {{0,1}}	5 {{2,3}}	0,1 {{4,5}}	2,3 {{6,7}}

表 3. **spread** ポリシーのスレッド配置とサブパーティション

OpenMP* バージョン 4.0 では、スレッド・アフィニティーのバインドポリシーのクエリー関数 **omp_proc_bind_t omp_get_proc_bind()** も提供されました。この関数は、次の **parallel** 領域で使用されるバインドポリシーを返します (その領域で **proc_bind** 節が指定されないと仮定します)。

spread ポリシーの興味深い点はサブパーティションで起こることです。**master** および **close** ポリシーでは、暗黙的なタスクはそれぞれ、親の暗黙的なタスクのプレース・パーティションを継承します。しかし、**spread** ポリシーでは、暗黙的なタスクは代わりにサブパーティションの **place-partition-var** ICV セットを取得します。これは、入れ子の **parallel** 構文ではすべてのスレッドが親のサブパーティション内に配置されることを意味します。

bind-var ICV の値は **OMP_PROC_BIND** 環境変数で初期化することができます。**bind-var** ICV の値は **proc_bind** 節を **parallel** 構文に追加してオーバーライドすることもできます。**place-partition-var** ICV の値は **OMP_PLACES** 環境変数で指定します。プレースは、ハードウェア・スレッド、コア、ソケット (特定数を含む) になります。明示的なプロセッサー・リストの場合もあります。詳細は、OpenMP* API 仕様を参照してください。

OpenMP* 4.5 仕様では、プレース・パーティションを照会して現在のスレッドのプレースをバインドする機能のセットが提供され、言語のアフィニティー機能が強化されました。これらの新しい API 関数は、希望するスレッド・アフィニティーを達成するため、設定の正確さを確認するのに役立ちます。コードが複雑で入れ子の並列処理が **spread** バインドポリシーとともに使用され、低レベルキャッシュを共有する入れ子の並列領域にスレッドを配置する場合は特に重要です。次の API 関数があります。

- **int omp_get_num_places():** 初期タスクの実行環境で **place-partition-var** のプレースの数を返します。
- **int omp_get_place_num_procs(int place_num):** プレース・パーティションの **place_num** で指定されたプレースの実行環境で利用可能なプロセッサーの数を返します。
- **void omp_get_place_proc_ids(int place_num, int *ids):** プレース・パーティションの **place_num** で指定されたプレースの実行環境で利用可能なプロセッサーを取得し、それらを保持する配列を割り当てて、その配列を **ids** に格納します。
- **int omp_get_place_num(void):** 到達スレッドがバインドされるプレース・パーティションのプレースの数を返します。
- **int omp_get_partition_num_places(void):** 最も内側の暗黙的なタスクのプレース・パーティションのプレースの数を返します。常にオリジナルのプレース・パーティションを示す **omp_get_num_places()** とは異なり、プレース・パーティションをサブパーティションに分割する **spread** バインドポリシーの効果を示すことに注意してください。
- **void omp_get_partition_place_nums(int *place_nums):** 最も内側の暗黙的なタスクのプレース・パーティションに対応するプレース番号のリストを取得し、**place_nums** の配列を割り当てて格納します。プレース番号はオリジナルのプレース・パーティションのプレースの番号であることに注意してください。この関数は、spread バインドポリシーで生成されるサブパーティションに含まれるオリジナルのプレース・パーティションのプレースを確認する際に特に役立ちます。

OpenMP* API 仕様バージョン 5.0 の概要

OpenMP* ARB (Architecture Review Board) では、OpenMP* 仕様のバージョン 5.0 で追加される可能性のある機能について議論しています。最も可能性の高い候補をこのセクションで紹介します。

メモリー管理のサポート

OpenMP* アプリケーション内でますます複雑になるメモリー階層をサポートする方法は、活発に議論されています。この複雑さは、異なる特性の新しいメモリー (インテル® Xeon Phi™ プロセッサーの MCDRAM やインテル® 3D XPoint™ メモリーなど)、優れたパフォーマンスを保証するため割り当てメモリーの特性に関する要求 (特定のアライメントやページサイズなど)、いくつかのメモリーに対する特別なコンパイラー・サポート、NUMA による影響など、さまざまな要因によるものです。さらに、多くの新しいメモリー・テクノロジーが調査中であるため、将来のテクノロジーに対応できるように拡張可能なようにする必要があります。

現在取り組んでいる方法は、異なるテクノロジーと操作をモデル化する 2 つの重要な概念 (メモリー空間およびアロケーター) に基づいています。メモリー空間は、プログラマーがプログラムで使用するメモリーを見つけるために指定できる、システムメモリーとメモリー特性のセット (ページサイズ、容量、帯域幅など) を表します。アロケーターは、メモリー空間からメモリーを割り当てるオブジェクトで、動作 (アロケーターのアライメントなど) を変更する特性を含むこともできます。

新しい API は、メモリー空間とアロケーターの操作、およびメモリーの割り当てと割り当て解除を行うように定義されています。アロケーターの作成とメモリーの割り当ての呼び出しを分けることにより、メモリーを割り当てる場所に関する決定が共通の「決定」モジュールで得られる、管理しやすいインターフェイスを構築できます。**図 13** は、2MB ページを使用してメモリーから異なる 2 つのアロケーター (割り当てが 64 バイトでアライメントされることを保証するアロケーターと保証しないアロケーター) を定義するシステムで最も高い帯域幅のメモリーを選択する際に、どのように提案を使用するかを示しています。

```
omp_memtrait_set_t  trait_set;
omp_memtrait_t  traits[] = {{OMP_MTK_BANDWIDTH,OMP_MTK_HIGHEST},
                            {OMP_MTK_PAGESIZE, 2*1024*1024}};
omp_init_memtrait_set(&trait_set,2,traits);

omp_memspace_t *amemspace = omp_init_memspace(&trait_set);

omp_alloctrait_t trait = {{OMP_ATK_ALIGNMENT},{64}};
omp_alloctrait_set_t trait_set;
omp_init_alloctrait_set(&trait_set,1,&trait);

omp_allocator_t *aligned_allocator = omp_init_allocator(amemspace,
                                                        &trait_set);
omp_allocator_t *unaligned_allocator = omp_init_allocator(amemspace, NULL);

double *a = (double *)omp_alloc( aligned_allocator, N * sizeof(double) );
double *b = (double *)omp_alloc( unaligned_allocator, N * sizeof(double) );
```

13 最も高い帯域幅のメモリーを選択

新しい **allocate** ディレクティブは、API 呼び出しで割り当てられない変数の割り当てを制御するために提案されたものです (自動変数または静的変数など)。新しい allocate 節は、OpenMP* ディレクティブによる割り当て操作に使用できます (変数のプライベート・コピーなど)。**図 14** は、ディレクティブを使用して 2MB ページを使用する最も高い帯域幅のメモリーに変数 **a** と **b** の割り当てを変更する方法を示しています。並列領域で **b** のプライベート・コピーは、レイテンシーが最も低いメモリーに割り当てられます。

```
int a[N], b[M];
#pragma omp allocate(a,b) memtraits(bandwidth=highest, pagesize=2*1024*1024)

void example() {
#pragma omp parallel private(b) allocate(memtraits(latency=lowest):b)
    {
        // ...
    }
}
```

14 2MB ページを使用する最も高い帯域幅のメモリーに変数 **a** と **b** の割り当てを変更

ヘテロジニアス・プログラミングの向上

OpenMP* のデバイスサポートを向上するために次のような機能が検討されています。

- 現在、**map** 節の構造は、構造のポインターフィールドを含めて、ビット単位でコピーされます。ポインターフィールドが有効なデバイスメモリーを指すようにプログラマーが要求した場合、デバイス上にメモリーを確保してデバイスのポインターフィールドを明示的に更新する必要があります。委員会は、構造のポインターフィールドのサポートを拡張することにより、プログラマーが **map** 節を使用して構造のポインターフィールドの自動割り当て / 割り当て解除を指定できるようにする拡張について議論しています。
- ARB は、関数ポインターを **target** 領域で使用できるようにすること、および関数ポインターを **declare target** に記述できるようにすることを検討しています。
- 非同期に実行できる新しいデバイス **memcpy** ルーチンのサポート。
- **target** 構文の「デバイスで実行または失敗」セマンティクスのサポート。現在、デバイスが利用できない場合、ターゲット領域はホストで実行されます。
- デバイスのみに存在し、ホストベースのコピーでない変数や関数のサポート。
- 単一アプリケーションでの複数のデバイスタイプのサポート。

タスクの改良

この記事で説明した機能以外に、OpenMP* 5.0 では次の機能が検討されています。

- **taskloop** 構文間のデータ依存関係の有効化。
- **task** 構文と **taskloop** 構文の両方でのデータ依存関係の有効化。1 つの **depend** 節からより多くの依存関係を生成する複数の値に拡張できる式を含む。
- **OMP_PROC_BIND** に似たパターンのスレッド・アフィニティーでのタスク表現のサポート。

その他の変更

OpenMP* 5.0 の追加機能として、次のような機能が議論されています。

- OpenMP* のベース言語の仕様を C11、C++11、C++14、Fortran 2008 にアップグレードする。
- 非矩形のループ形状、入れ子のループ間のコード記述がそれぞれ可能になるように、**`collapse`** 節の制限を緩和する。
- ワークシェアリング構造に関連付けられていない並列領域の内部でのリダクションを可能にする。

OpenMP* API は、ハイパフォーマンス・コンピューティングの C、C++、Fortran アプリケーションの共有メモリー並列化に適した、可搬性のあるベンダー・ニュートラルな並列プログラミング言語として不動の地位を確立しています。バージョン 5.0 の今後の開発では、開発者が最近のプロセッサーの機能を最大限に活用できるように、さらに多くの機能が提供される予定です。

BLOG HIGHLIGHTS

コードの現代化 : CERN における科学的発見と全体的な革新の促進 (パート 2)

RUSS BEUTLER (INTEL) ›

インテルは、CERN openlab CTO の Dr. Maria Girone と、CERN とインテルが協力して処理速度を向上する方法や CERN で行われている物質の基本成分の研究に与える影響について対談を行いました。この対談は、「モダンコード」アプローチが研究の進歩やブレークスルーにどのように役立つか、分かりやすくプログラマーに説明することが目的でした。

対談の第 2 部で、Dr. Maria は、コードの現代化戦略に関する開発者へのアドバイスを述べ、開発者と学生がスキルを伸ばし、キャリアをアップさせて、取り組んでいるアプリケーションに大規模な改良をもたらすことができるプログラムについて紹介しました。

コードの現代化戦略を構築し、アプリケーションが最近のサーバー・ハードウェアを確実に活用することを求めている開発者と企業に対するアドバイスを聞かせてください。

まず、古いコードを利用している場合、大幅な改良の余地があると認識することが重要です。これは、古いコードの更新が得意な開発者が職を得られる機会があることを意味します。

具体的には、レガシー・ソフトウェアと現在のパフォーマンス標準との差、および効率的な並列化とベクトル化により達成可能なゲインを理解する必要があります。次に、コードの現代化に取り組んだ場合に達成できるパフォーマンスの向上を分かりやすく示すことが重要です。

この記事の続きはこちら (英語) でご覧になれます。›