インテル® Fortran コンパイラー・クラシック 2021.1 およびベータ版インテル® Fortran コンパイラーの OpenMP* サポート

この資料は、インテル® デベロッパー・ゾーンで公開されている『インテル® Fortran コンパイラー・クラシック 2021.1 およびベータ版インテル® Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス』の「OpenMP* Support」セクションの日本語参考訳です。

OpenMP* サポート

インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーは、OpenMP* アプリケーション・プログラミング・インターフェイスのバージョン 4.5 を完全にサポートしており、OpenMP* バージョン 5.0 の一部の機能もサポートしています。OpenMP* 仕様の詳細は、OpenMP* ウェブサイトで公開されている仕様(http://www.openmp.org(英語)の OpenMP* Specifications)を参照してください。このドキュメントでは、OpenMP* の言語特性の説明において、通常この OpenMP* 仕様で定義されている用語を使用しています。

OpenMP* API は、次の主な機能を持った対称型マルチプロセシング(SMP) を提供します。

- 反復のパーティショニング、データの共用、スレッドの生成、スケジュール、および同期化に関する下位レベルの詳細を処理して、ユーザーの負担を軽減します。
- サポートされるすべてのインテル® アーキテクチャー上のインテル® ハイパースレッディング・テクノロジー(インテル® HT テクノロジー)対応プロセッサーを含む共有メモリーのマルチプロセッサー・システムおよびマルチコア・プロセッサー・システムから優れたパフォーマンスを引き出します。

コンパイラーは、ソースプログラムの OpenMP* ディレクティブの指定に従ってコード変換を実行し、マルチスレッド・コードを生成して、既存のソフトウェアへスレッドをより簡単に実装します。コンパイラーは、業界標準の OpenMP* ディレクティブに対応し、並列プログラムをコンパイルします。

コンパイラーは、ランタイム・ライブラリー・ルーチンおよび環境変数(英語)を含む OpenMP* 仕様にインテル独自の拡張機能を提供します。コンパイラー・オプションの概要は、「OpenMP* オプションのクイック・リファレンス」(英語)を参照してください。

OpenMP* による並列処理

OpenMP* API を使用してコンパイルするには、コードに Fortran プログラムコメント形式のディレクティブを追加します。インテル® コンパイラーはコードを処理し、マルチスレッド・バージョンを内部的に生成してから、コンパイルして並列領域または構造を実行するスレッドによって実装される並列処理の実行プログラムを生成します。

その他のコンパイラーの使用

OpenMP* 仕様は複数の実装の互換性について定義していません。このため、他のコンパイラーでサポートされている OpenMP* 実装とインテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーでサポートされている OpenMP* 実装は互換性がないことがあります。アプリケーション全体を 1 つのコンパイラーでコンパイルしてビルドしたとしても、異なるコンパイラーでは OpenMP* のソース互換性(異なるコンパイラーで同じアプリケーション・ソース・セットをリンクして、コンパイルし、期待する並列実行結果を得られること)が提供されないことに注意してください。

OpenMP* サポートをアプリケーションに追加する

OpenMP* サポートをアプリケーションに追加するには、次の操作を行います。

1. 適切な OpenMP* ディレクティブをソースコードに追加します。
2. `/Qopenmp`(Windows*)または `-qopenmp`(Linux* および macOS*)オプションを指定してアプリケーションをコンパイルします。
3. 大規模なローカル配列や一時配列を持つアプリケーションの場合、実行時に利用可能なスタック領域を増やす必要があります。また、`OMP_STACKSIZE` 環境変数や対応するライブラリー・ルーチンを設定して、個々のスレッドに割り当てられるスタックを増やします。

マルチスレッド・コードの実行を制御するその他の環境変数を設定できます。

OpenMP* ディレクティブ構文

OpenMP* サポートをアプリケーションに追加するには、最初に適切な OpenMP* ディレクティブをソースコードに追加します。

OpenMP* ディレクティブは、特定の形式と構文を使用します。「インテルの OpenMP* 拡張ルーチン」では、インテル® Fortran コンパイラーに追加された OpenMP* の拡張機能について説明されています。

次の構文は、ソースでディレクティブを使用する例を示します。

例
`<prefix> <directive> [<clause>[[,]<clause>...]]`

説明:

- `<prefix>` - すべての OpenMP* ディレクティブに必須。自由形式のソース入力の場合、プリフィクスは `!$OMP` のみです。固定形式のソース入力の場合、プリフィクスは `!$OMP` または `C$OMP` です。
- `<directive>` - 有効な OpenMP* ディレクティブ。プリフィクスの直後に必ず指定する必要があります(例:`!$OMP PARALLEL`)。
- [`<clause>`] - オプション。`clause` は順番に関係なく指定でき、制限されていない限り必要に応じて繰り返すことができます。
- [`<newline>`] - ディレクティブ構文の必須コンポーネント。このディレクティブで囲まれた構造化ブロックに先行します。

- [,]: オプション。`<clause>` 間のカンマはオプションです。

`/Qopenmp`(Windows*)または `-qopenmp`(Linux* および macOS*)オプションを指定しない場合、ディレクティブはコメントとして解釈されます。

並列領域を定義している OpenMP* 構造の構文形式は、次のいずれかです。

例

```
!$OMP <directive>
  <structured block of code>
!$OMP END <directive>
  # OR
!$OMP <directive>
  <structured block of code>
  # OR
!$OMP <directive>
```

次の例は、OpenMP* ディレクティブを使用してループを並列化する 1 つの方法を説明します。

例

```
subroutine simple_omp(a, N)
  use omp_lib
  integer :: N, a(N)
!$OMP PARALLEL DO
  do i = 1, N
a(i) = i*2
  end do
end subroutine simple_omp
```

アプリケーションのコンパイル

`/Qopenmp`(Windows*)または `-qopenmp`(Linux* および macOS*)オプションを指定すると、並列化機能はソースの OpenMP* ディレクティブに基づいてマルチスレッド・コードを生成します。このコードはシングル・プロセッサー・システム、マルチプロセッサー・システム、マルチコア・プロセッサー・システムで並列実行が可能です。

`/Qopenmp`(Windows*)または `-qopenmp`(Linux* および macOS*)オプションは、`-O0`(Linux* および macOS*)または `/Od`(Windows*)、`O1`、`O2`、`O3` の最適化レベルで動作します。

`-O0`(Linux* および and macOS*)または `/Od`(Windows*)オプションと `/Qopenmp`(Windows)または `-qopenmp`(Linux* および macOS*)オプションを指定すると、OpenMP* アプリケーションのデバッグに役立ちます。

次のようなコマンドを使用してアプリケーションをコンパイルします。

オペレーティング・システム	構文の例
Linux*	`ifort -qopenmp source_file`
macOS*	`ifort -qopenmp source_file`
Windows*	`ifort /Qopenmp source_file`

次のようなコマンドを使用して、上記の例をコンパイルすると仮定します。`c` オプションは、実行ファイルを生成しないでコードをコンパイルするようにコンパイラーに指示します。

オペレーティング・システム	拡張構文の例
Linux*	`ifort -qopenmp -c parallel.f90`
macOS*	`ifort -qopenmp -c parallel.f90`
Windows*	`ifort /Qopenmp /c parallel.f90`

OpenMP* 環境の設定

マルチスレッド・コードを起動する前に、OpenMP* 環境変数 `OMP_NUM_THREADS` で使用するスレッドの数を設定できます。

関連情報

c(英語)
O(英語)
OpenMP* の例
qopenmp、Qopenmp(英語)

並列処理モデル

OpenMP* API コンパイラー・ディレクティブを含むプログラムは、実行の初期スレッドと呼ばれる単一のスレッドとして実行を開始します。最初の並列構造が検出されるまで、初期スレッドは、シーケンシャルに実行します。

OpenMP* API では、`PARALLEL` と `END PARALLEL` ディレクティブは並列構造の範囲を定義します。初期スレッドが並列構造を検出すると、その初期スレッドがチームのマスターとなり、スレッドのチームを作成します。並列構造に囲まれたプログラム文はすべて(そこから呼び出されるすべてのルーチンを含む)、チーム内の各スレッドで並列に実行されます。

構造内で字句的に囲まれた文は、構造の静的範囲を定義します。動的範囲には、スレッドによる構造の実行中に実行されるすべての文(呼び出される全ルーチンを含む)が含まれます。

1 つのスレッドが並列構造によって囲まれた構造化ブロックの終わりに到達すると、そのスレッドは、チーム内のすべてのスレッドが到着するまで待機します。すべてのスレッドが到着すると、チームは消滅し、マスタースレッドのみが次の並列構造のコードの実行を続けます。チーム内の他のスレッドは、別のチームの形成で必要になるまで待機状態になります。単一プログラム内で並列構造は何回でも指定できます。結果として、プログラム実行中に、スレッドのチームは何度も生成され消滅します。

次の例は、上位レベルから OpenMP* 構造の実行モデルを示します。コードのコメントは、各構造やセクションの説明です。

例

```
PROGRAM MAIN            ! Begin serial execution.
  ...                   ! Only the initial thread executes.
 !$OMP PARALLEL         ! Begin a Parallel construct, form a team.
   ...                  ! This code is executed by each team member.
  !$OMP SECTIONS        ! Begin a worksharing construct.
  !$OMP SECTION         ! One unit of work.
   ...                  !
  !$OMP SECTION         ! Another unit of work.
   ...                  !
   !$OMP END SECTIONS   ! Wait until both units of work complete.
   ...                  ! More Replicated Code.
  !$OMP DO              ! Begin a worksharing construct,
     DO                 !   each iteration is a unit of work.
     ...                ! Work is distributed among the team.
     END DO             !
  !$OMP END DO NOWAIT   ! End of worksharing construct, NOWAIT
                        !   is specified (threads need not wait).
                        ! This code is executed by each team member.
  !$OMP CRITICAL        ! Begin critical construct.
   ...                  ! One thread executes at a time.
  !$OMP END CRITICAL    ! End the critical construct.
   ...                  ! This code is executed by each team member.
  !$OMP BARRIER         ! Wait for all team members to arrive.
   ...                  ! This code is executed by each team member.
 !$OMP END PARALLEL     ! End of parallel construct, disband team
                        !   and continue with serial execution.
  ...                   ! Possibly more parallel constructs.
END PROGRAM MAIN        ! End serial execution.
```

親なしディレクティブの使用

並列構造内で呼び出されたルーチンで、ディレクティブを使用することができます。並列構造の静的範囲ではなく、動的範囲のディレクティブは、親なしディレクティブと呼ばれます。親なしディレクティブは、プログラムのシーケンシャル・バージョンに最小限の変更を行うだけで、プログラム部分を並列に実行できます。この機能を使用すると、プログラム・コール・ツリーの最上位レベルで並列構造をコーディングでき、ディレクティブを使用して呼び出されるすべてのルーチンの実行を制御することができます。次に例を示します。

例

```
subroutine F
...
!$OMP PARALLEL...
   call G
...
subroutine G
!$OMP DO... ! This is an orphaned directive.
...
```

並列領域がサブルーチン G で字句的に指定されていないので、これが親なし DO ディレクティブとなります。

データ環境制御

並列およびワークシェア構造内でデータ環境を制御できます。ディレクティブとデータ環境節を使用して、THREADPRIVATE ディレクティブで名前付きグローバル・ライフタイム・オブジェクトをプライベート化したり、ディレクティブがサポートする場合はデータ環境節でデータスコープ属性を制御できます。

データスコープ属性節:

- DEFAULT
- FIRSTPRIVATE
- IN_REDUCTION
- LASTPRIVATE
- LINEAR
- PRIVATE
- REDUCTION
- SHARED

データコピー節:

- COPYIN
- COPYPRIVATE

データモーション節:

- MAP

その他の節:

- ALIGNED
- COLLAPSE
- DEPEND
- DEVICE
- FINAL
- IF
- MERGEABLE
- NOWAIT
- PRIORITY
- UNTIED

複数のディレクティブを使用して、変数のデータスコープ属性を指定した構造の継続期間中にその属性を制御することができます。ただし、ディレクティブでデータスコープ属性節を指定しない場合、ディレクティブに影響を受ける変数の動作はデフォルトのスコープ規則によって決まります。これは、OpenMP* API 仕様で説明されています。

使用するスレッド数の決定

ワークロードがアプリケーションの(一定ではない)入力に依存する場合、実行時に入力サイズが確認されるまで、使用するスレッド数の決定を保留します。スレッド数に影響を与えるワークロードの入力引数には、行列のサイズ、データベースのサイズ、イメージ/ビデオのサイズおよび解像度、ツリーベースの構造体の階層の深さ/幅/種類、およびリストベースの構造体のサイズなどが含まれます。同様に、プロセッサー数が一定ではないシステムで実行するように設計されたアプリケーションも、実行時にマシンサイズが確認されるまで、使用するスレッド数の決定を保留します。

入力データから処理の量を予測できないアプリケーションでは、適切なスレッド数を選択するため、測定ステップを取り入れて、ワークロードおよびシステムの特性を確認することを推奨します。測定結果は、ファイルシステムなど永久保管場所に格納することによって、(測定ステップのオーバーヘッドを抑え)永続的に使用することができます。

システムの処理ユニット数を超えるスレッドを同時に使用しないでください。同時に使用すると、オペレーティング・システムによりプロセッサー上でスレッドが多重化され、パフォーマンスが最適化されません。

ライブラリーを開発する場合(アプリケーション全体を開発する場合とは異なり)、ライブラリーによって使用されるスレッド数を簡単に選択できるメカニズムを使用してください。これは、より高度な並列化を使用している場合、ライブラリー内の並列化が必要ないためです。

並列領域で NUM_THREADS 節を使用して、使用するスレッド数を制御し、並列領域で IF 節を使用してマルチスレッドを使用するかどうかを決定します。OMP_SET_NUM_THREADS ルーチンも使用することができますが、呼び出し側のスレッドにより作成される並列領域にも影響します。NUM_THREADS 節の効果はローカルであるため、ほかの並列領域には影響しません。明示的なスレッド数の設定には、次のような短所があります。

1. 多数のプロセッサーを搭載したシステムで、アプリケーションはすべてのプロセッサーではなく、その一部を使用します。
2. 少数のプロセッサーを搭載したシステムで、アプリケーションはパフォーマンスの低下につながるオーバーサブスクリプションを強制する可能性があります。

インテルの OpenMP* ランタイムは、OMP_SET_NUM_THREADS ルーチンを使用しない限り、利用可能な論理プロセッサー数と同じ数のスレッドを生成します。実際の制限を特定するには、OMP_GET_THREAD_LIMIT() と OMP_GET_MAX_ACTIVE_LEVELS() を使用します。システムがオーバーロードしないように、開発者はスレッドの使用と並列処理の入れ子構造に注意する必要があります。OMP_THREAD_LIMIT 環境変数は、OpenMP* プログラム全体に使用する OpenMP* スレッドの数を制限します。OMP_MAX_ACTIVE_LEVELS 環境変数は、アクティブな入れ子構造の並列領域の数を制限します。

バインド設定

OpenMP* 構文を別の OpenMP* 構文で入れ子にしたり、その動作については、さまざまなバインド設定で指定されます。

OpenMP* 構文のバインド領域は、実行コンテキストとディレクティブの有効範囲を決定する領域です。

- ORDERED 構文のバインド領域は、最内の DO ループ領域です。
- TASKWAIT 構文のバインド領域は、最内の TASK 領域です。
- 現在のチームがバインドスレッドであるか、現在のチームタスクがバインドタスクであるその他すべての構文のバインド領域は、最内の PARALLEL 領域です。
- バインドタスクがタスク生成である構文のバインド領域は、生成されるタスクの領域です。
- PARALLEL 構文は、アクティブでなくても、明示的でなくても、バインド領域になります。
- TASK 構文は、明示的でなくても、バインド領域になります。
- 領域は、最内の囲まれた並列領域外のどの領域ともバインドしません。

OpenMP* 構文のバインドタスクは、領域の実行により影響を受ける、または領域の実行にコンテキストを与えるタスクのセットです。特定の構文のバインドタスクは、すべてのタスク、現在のチームタスク、または生成されるタスクです。

OpenMP* 構文のバインドスレッドは、領域の実行により影響を受ける、または領域の実行にコンテキストを与えるスレッドのセットです。特定の構文のバインドスレッドは、デバイス上のすべてのスレッド、グループのすべてのスレッド、現在のチーム、または到達スレッドです。

スレッド割り当ての制御

KMP_HW_SUBSET および KMP_AFFINITY 環境変数は、プロセッサーにおける OpenMP* ランタイムのハードウェア・スレッドの使用を制御します。これらの環境変数を使用することで、プロセッサーのコアで異なるスレッドの分配を試したり、スレッドをどのようにコアにバインドするかを決定して、アプリケーションに最適な設定を見つけることができます。

KMP_HW_SUBSET 変数はハードウェア・リソースの割り当てを制御し、KMP_AFFINITY 変数はそれらのリソースへ OpenMP* スレッドをバインドする方法を制御します。

スレッドの分配の制御

KMP_HW_SUBSET 変数は、プログラムで使用されるハードウェア・リソースを制御します。この変数は、使用するソケット数、各ソケットで使用するコア数、および各コアに割り当てるスレッド数を指定します。コアあたり 1 スレッドの場合よりも、コアあたり 2 スレッドのほうがパフォーマンスは向上しますが、コアあたり 3 または 4 スレッドにすると、パフォーマンスは向上することもしないこともあります。この変数でコアあたり 4 スレッドまでのパフォーマンスを測定できます。

例えば、次のように変数を設定することで、24 コアを搭載したシステムで 24、48、72、または最大 96 の OpenMP* スレッドを割り当てた場合の影響が分かります。

割り当てるスレッド数	...設定
24	KMP_HW_SUBSET=24c,1t
48	KMP_HW_SUBSET=24c,2t
72	KMP_HW_SUBSET=24c,3t
96	KMP_HW_SUBSET=24c,4t

注:
この変数と OMP_NUM_THREADS 変数を一緒に使用する場合は注意が必要です。OMP_NUM_THREADS 変数を使用すると、オーバーサブスクリプション/アンダーサブスクリプションが発生します。

スレッドのバインドの制御

KMP_AFFINITY 変数は、KMP_HW_SUBSET 変数によって割り当てられたハードウェア・リソースに OpenMP* スレッドをバインドする方法を制御します。この変数は、いくつかのバインドまたはアフィニティー・タイプに設定できますが、プロセッサーで OpenMP* スレッドを実行する場合、推奨するアフィニティー・タイプは次のとおりです。

- compact: キャッシュを共有するコア間でシーケンシャルにスレッドを分配します。
- scatter: キャッシュに関係なくコア間でスレッドを分配します。

次の表は、KMP_HW_SUBSET=2c,3t を指定して、2 コアでコアあたり 3 スレッドを使用する場合のスレッドのコアへのバインド方法を示します。

アフィニティー	コア 0 の OpenMP* スレッド	コア 1 の OpenMP* スレッド
KMP_AFFINITY=compact	0、1、2	3、4、5
KMP_AFFINITY=scatter	0、2、4	1、3、5

最適な設定の特定

これらの変数を使用して最適なスレッドの分配とバインドを見つけるには、次の手順を実行します。

1. これらの環境変数を使用する前に、OpenMP* コードが正しく動作することを確認します。
2. 現在の OpenMP* コードを使用してベースラインを確定し、プロセッサーにスレッドを割り当てた場合のパフォーマンスと比較します。
3. KMP_HW_SUBSET 変数により、コアあたり 1、2、3、または 4 スレッドを割り当てた場合のパフォーマンスを測定します。
4. KMP_AFFINITY 変数により、スレッドをコアにバインドした場合のパフォーマンスを測定します。

関連情報

スレッド・アフィニティー・インターフェイス
サポートされる環境変数(英語)

OpenMP* ディレクティブの概要

ここでは、インテル® Fortran コンパイラーでサポートされている OpenMP* ディレクティブの概要を説明します。OpenMP* API の詳細については、OpenMP* ウェブサイトの OpenMP* Application Program Interface Version Technical Report 4: Version 5.0 仕様を参照してください。

次のリストで、ほかでも同じ名前が使用されている OpenMP* ディレクティブには**ディレクティブ**と明記しています。例えば、`FLUSH` はディレクティブ、文、サブルーチンを指します。

並列ディレクティブ

このディレクティブを使用してスレッドのチームを形成しスレッドを並列に実行します。

ディレクティブ	説明
PARALLEL (OpenMP*) (英語)	並列実行領域を定義します。

タスク・ディレクティブ

このディレクティブを使用して遅延実行を行います。

ディレクティブ	説明
TASK (英語)	タスク領域を定義します。
TASKLOOP (英語)	1 つ以上の関連する `DO` ループの反復を OpenMP* タスクを使用して並列に実行するように指定します。反復は、構文によって作成されるタスクに分配され、実行がスケジュールされます。

ワークシェア・ディレクティブ

これらのディレクティブを使用してスレッドのチーム間のワークを共有します。

ディレクティブ	説明
DO (英語)	関連付けられたループの反復がチーム内のスレッド間で分割される、反復的なワークシェアの構造を識別します。
LOOP (英語)	関連する `DO` ループの反復が任意の順序または同時実行可能であることを指定します。この機能は `ifx` でのみ利用できます。
SECTIONS (英語)	囲まれた `SECTION` ディレクティブがチームのスレッド間に分割されるコードのブロックを定義することを指定します。各セクションは、チーム内のスレッドにより 1 回だけ実行されます。
SINGLE (英語)	コードのスレッドが一度に 1 つのスレッドでのみ実行されることを指定します。
WORKSHARE (英語)	文または構造のブロックを実行する作業を個別のユニットに分割します。また、実行単位の作業をチームのスレッドに分配して、各作業単位が 1 回だけ実行されるようにします。

同期ディレクティブ

これらのディレクティブを使用してスレッド間を同期します。

ディレクティブ	説明
ATOMIC(英語)	特定のメモリー位置をアトミックに更新し、複数のスレッドが同時に読み取り/書き込みを行う危険性を回避します。
BARRIER(英語)	チーム内のすべてのスレッドを同期化します。各スレッドは、チーム内のほかのすべてのスレッドがバリアに到達するまで待機します。
CRITICAL(英語)	コードのブロックへのアクセスを一度に 1 つのスレッドのみに制限します。
FLUSH(英語)	チームのスレッドでメモリーの状態の整合性が保たれる同期ポイントを定義します。
MASTER(英語)	チームのマスタースレッドで実行されるコードブロックを指定します。
ORDERED(英語)	チームのスレッドがループの反復順に実行するスレッドのコードブロックを指定します。
TASKGROUP(英語)	現在のタスクの子タスクと派生タスクがすべて完了するまで待機するように指定します。
TASKWAIT(英語)	現在のタスクが開始してから、生成された子タスクの完了まで待機するように指定します。
TASKYIELD(英語)	現在のタスクを中断し、別のタスクの実行を優先することを許可します。

データ環境ディレクティブ

このディレクティブを使用してグローバルなプライベート・データをスレッドに割り当てます。

ディレクティブ	説明
THREADPRIVATE (英語)	各スレッドにプライベート(ローカル)な名前付き共通ブロック(スレッド内ではグローバル)を指定します。

オフロードターゲットを制御するディレクティブ

これらのディレクティブを使用して 1 つ以上のオフロードターゲット上の実行を制御します。オフロードは、Windows* では利用できません。

ディレクティブ	説明
DECLARE TARGET (英語)	デバイス向けに作成、またはマップする名前付きルーチンと変数を指定します。
DECLARE VARIANT(英語)	ベース・プロシージャーのバリアントを識別し、このバリアントが使用されるコンテキストを指定します。この機能は `ifx` でのみ利用できます。
DISTRIBUTE (英語)	`TEAMS` 構文により生成されるすべてのスレッドチームのマスタースレッド間で、ループ反復を分配するように指定します。
TARGET(英語)	デバイスデータ環境を作成してそのデバイスで構文を実行します。

TARGET DATA (英語)	領域の範囲のデバイスデータ環境へ変数をマップします。
TARGET ENTER DATA (英語)	デバイスのデータ環境へ変数をマップします。
TARGET EXIT DATA (英語)	デバイスのデータ環境から変数をアンマップ(解放)します。
TEAMS (英語)	ターゲット領域内でスレッドチームを複数作成し、各チームのマスタースレッドの構造化ブロックを実行します。
TARGET UPDATE (英語)	デバイスデータ環境のリスト項目と対応するオリジナルのリスト項目の整合性を保持します。

ベクトル化ディレクティブ

これらのディレクティブを使用してベクトル・ハードウェア上の実行を制御します。

ディレクティブ	説明
SIMD (OpenMP*) (英語)	ループを SIMD (Single Instruction, Multiple Data) 命令を使用して同時に実行されるループにベクトル化します。 `EARLY_EXIT` 節は、インテル独自の OpenMP* 仕様の拡張です。 `EARLY_EXIT` 複数の終了ポイントを持つループのベクトル化を許可します。この節を指定すると、プログラムは次のように振る舞います。 • ループの最後の途中終了文の前にある各操作は、SIMD チャンク内で途中終了がトリガーされなかったかのように実行されることがあります。 • ループの最後の途中終了文の後、すべての操作はループの最後の反復が見つかったかのように実行されます。 • `LINEAR` 節で指定された各リスト項目は、ループ終了時の最後の反復数に基づいて計算されます。 • `LINEAR` の最後の値と条件付き `LASTPRIVATE` は、スカラー実行では保持されます。 • `REDUCTION` の最後の値は、ループ終了時に最後の SIMD チャンクの最後の反復が実行されたかのように計算されます。 • スカラー実行の共有メモリーの状態は保持されない可能性があります。 • 例外は許可されていません。
DECLARE SIMD (英語)	SIMD プロシージャーを生成します。

キャンセル構文

ディレクティブ	説明
CANCEL(英語)	指定した構文の最内領域の取り消し要求を行います。このプラグマに到達したタスクは、取り消された構文の最後に進みます。
CANCELLATION POINT(英語)	暗黙的または明示的なタスクが、指定された節の最内領域で取り消し要求があったかどうかをチェックするポイントを定義します。

メモリー空間割り当てディレクティブ(ifx のみ)

このディレクティブを使用してメモリー空間を割り当てます。この機能は `ifx` でのみ利用できます。

ディレクティブ	説明
ALLOCATE(英語)	非割り付けかつ非ポインターのリスト項目(変数と共通ブロック)を指定したメモリー空間アロケーターで割り当て/解放することを指定する宣言ディレクティブです。

結合ディレクティブ

これらのディレクティブは、連続する複数のディレクティブのショートカットとして使用します。結合構造は、ある構造内に別の構造を入れ子するためのショートカット形式です。内部に別の構造を 1 つだけ含み、ほかの文を含まない構造を明示的に指定することと同じです。

複合構造は、2 つの構造から成り、構造内に別の構造を入れ子する場合とセマンティクスが異なります。複合構造は、その構成要素である 2 つの構造には含まれないセマンティクスを追加したり、構造内に別の構造を入れ子することが不適合な場合に使用します。

ディレクティブ	説明
DISTRIBUTE PARALLEL DO[1](英語)	複数のチームのメンバーである複数のスレッドによって並列に実行できるループを指定します。
DISTRIBUTE PARALLEL DO SIMD[1](英語)	複数のチームのメンバーである複数のスレッドによって並列に実行されるループを指定します。また、ループは SIMD 命令を使用して同時に実行されます。
DISTRIBUTE SIMD[1](英語)	チーム領域のマスタースレッド間で分配されるループを指定します。また、ループは SIMD 命令を使用して同時に実行されます。
DO SIMD[1](英語)	SIMD 命令による同時実行も適用されます。
PARALLEL DO(英語)	1 つの `DO` ディレクティブを含む並列領域を簡潔に指定する方法を提供します。
PARALLEL DO SIMD(英語)	SIMD 命令による同時実行も適用されます。1 つの SIMD ループ構造だけを含み、その他の文を含まない `PARALLEL` 構造を簡潔に指定する方法を提供します。

PARALLEL LOOP (英語)	1 つの LOOP 構造を含む並列領域を簡潔に指定する方法を提供します。この機能は ifx でのみ利用できます。
PARALLEL SECTIONS (英語)	1 つの SECTIONS ディレクティブを含む並列領域を簡潔に指定する方法を提供します。セマンティクスは SECTIONS ディレクティブが直後に続く PARALLEL ディレクティブを明示的に指定することと同じです。
PARALLEL WORKSHARE (英語)	1 つの WORKSHARE (英語) ディレクティブを含む並列領域を簡潔に指定する方法を提供します。
TARGET PARALLEL (英語)	並列領域でデバイスデータ環境を作成して、そのデバイスで構文を実行します。
TARGET PARALLEL DO (英語)	1 つの PARALLEL DO 構造だけを含み、その他の文を含まない TARGET 構造を簡潔に指定する方法を提供します。
TARGET PARALLEL DO SIMD (英語)	1 つの PARALLEL DO SIMD 構造だけを含み、その他の文を含まない TARGET 構造を簡潔に指定する方法を提供します。
TARGET PARALLEL LOOP (英語)	1 つの PARALLEL LOOP 構造だけを含み、その他の文を含まない TARGET 構造を簡潔に指定する方法を提供します。この機能は ifx でのみ利用できます。
TARGET SIMD (英語)	1 つの SIMD 構造だけを含み、その他の文を含まない TARGET 構造を簡潔に指定する方法を提供します。
TARGET TEAMS (英語)	デバイスデータ環境を作成して同じデバイスで構文を実行します。また、スレッドチームを複数作成し、各チームのマスタースレッドが構造化ブロックを実行します。このプラグマは、インテル® MIC アーキテクチャーにのみ適用されます。
TARGET TEAMS DISTRIBUTE (英語)	デバイスデータ環境を作成してそのデバイスで構文を実行します。また、TEAMS により生成されるすべてのスレッドチームのマスタースレッド間で、ループ反復を分配するように指定します。
TARGET TEAMS DISTRIBUTE PARALLEL DO (英語)	デバイスデータ環境を作成してそのデバイスで構文を実行します。また、TEAMS 構造により生成される複数のチームのメンバーである複数のスレッド間で、ループが並列に実行されるように指定します。
TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD (英語)	デバイスデータ環境を作成してそのデバイスで構文を実行します。また、TEAMS 構造により生成される複数のチームのメンバーである複数のスレッド間で、ループが並列に実行されるように指定します。ループはチーム全体に分配され、SIMD 命令を使用して同時に実行されます。
TARGET TEAMS DISTRIBUTE SIMD (英語)	デバイスデータ環境を作成してそのデバイスで構文を実行します。また、TEAMS 構文により生成されるすべてのスレッドチームのマスタースレッド間で、ループ反復を分配するように指定します。また、ループは SIMD 命令を使用して同時に実行されます。
TARGET TEAMS LOOP (英語)	1 つの TEAMS LOOP 構造だけを含み、その他の文を含まない TARGET 構造を簡潔に指定する方法を提供します。この機能は ifx でのみ利用できます。
TASKLOOP SIMD[1] (英語)	SIMD 命令を使用して同時に実行可能で、反復が OpenMP* タスクを使用して並列に実行されるループを指定します。

TEAMS DISTRIBUTE (英語)	スレッドチームを複数作成し、各チームのマスタースレッドの構造化ブロックを実行します。また、TEAMS により生成されるすべてのスレッドチームのマスタースレッド間で、ループ反復を分配するように指定します。
TEAMS DISTRIBUTE PARALLEL DO (英語)	スレッドチームを複数作成し、各チームのマスタースレッドの構造化ブロックを実行します。また、複数のチームのメンバーである複数のスレッドによって並列に実行できるループを指定します。
TEAMS DISTRIBUTE PARALLEL DO SIMD (英語)	スレッドチームを複数作成し、各チームのマスタースレッドの構造化ブロックを実行します。また、複数のチームのメンバーである複数のスレッドによって並列に実行できるループを指定します。ループはチーム領域のマスタースレッド間で分配され、SIMD 命令を使用して命令レベルで同時に実行されます。
TEAMS DISTRIBUTE SIMD (英語)	スレッドチームを複数作成し、各チームのマスタースレッドの構造化ブロックを実行します。また、チーム領域のマスタースレッド間で分配されるループを指定します。
TEAMS LOOP (英語)	1 つの LOOP 構造だけを含み、その他の文を含まない TEAMS 構造を簡潔に指定する方法を提供します。この機能は ifx でのみ利用できます。

脚注:

[1] このディレクティブは、複合構造を指定します。

OpenMP* ライブラリー・サポート

OpenMP* ランタイム・ライブラリー・ルーチン

OpenMP* は、並列モードでプログラムを管理しやすくするために、ランタイム・ライブラリー・ルーチンを提供します。これらのランタイム・ライブラリー・ルーチンの多くには、デフォルトとして設定可能な環境変数が対応付けられています。ランタイム・ライブラリー・ルーチンを使用すれば、これらの変数を動的に変更でき、プログラムを簡単に制御できます。いずれの場合も、ランタイム・ライブラリー・ルーチンを呼び出すと、それに対応する環境変数は無効になります。これらのルーチンはすべて外部プロシージャーです。

注:
OpenMP* ランタイム・ライブラリー・ルーチンを実行することで OpenMP*ランタイム環境が初期化され、OpenMP* 環境変数のそれ以降のプログラムコードによる設定の効果がなくなることがあります。この問題を回避するには、インテルの拡張ルーチン `kmp_set_defaults()` を使用して OpenMP* 環境変数を設定します。

インテル® コンパイラーは、すべての OpenMP* ランタイム・ライブラリー・ルーチンをサポートしています。これらのルーチンの使用に関する詳細は、OpenMP* API 仕様を参照してください。

ルーチンを含むプログラム単位に次のような文を追加して適切なルーチンの宣言をインクルードします。

例
`use omp_lib`

モジュールファイルは、コンパイラーをインストールした `../include`(Linux* および macOS*)または `..\include`(Windows*)ディレクトリーにあります。

`omp_lib.mod`(および `omp_lib.mod` のインクルード・バージョンである `omp_lib.h`)の整数引数 `openmp_version` は、10 進数 `yyyymm` です。`yyyy` と `mm` は、現在のバージョンのコンパイラーとライブラリーでサポートされる OpenMP* API のバージョンを示します。詳細は、定義済みプリプロセッサー・シンボル `_OPENMP` を参照してください。

注:
ルーチンのインターフェイスの一部には、同等のオフロード・インターフェイスがあります。同等のオフロード・インターフェイスには、ターゲットの種類と番号を指定する 2 つの追加の引数があります。詳細は、「CPU の関数を呼び出してコプロセッサーの実行環境を変更する」を参照してください。

次の表は、各ルーチンの仮引数のデータ型を指定するキーのリストです。

キー	OMP_LIB Kind	BIND(C) Kind	KIND=
INTEGER (int)	OMP_INTEGER_KIND	C_INT	4
LOGICAL (log)	OMP_LOGICAL_KIND		4
REAL (dp)	DOUBLE PRECISION	C_DOUBLE	8
INTEGER(OMP_LOCK_KIND)		C_INTPTR_T	intptr_t <<1>>
INTEGER(OMP_NEST_LOCK_KIND)		C_INTPTR_T	intptr_t <<1>>
INTEGER (OMP_SCHED_KIND)	OMP_INTEGER_KIND	C_INT	4
INTEGER(OMP_PROC_BIND_KIND)	OMP_INTEGER_KIND	C_INT	4
INTEGER(OMP_LOCK_HINT_KIND)		C_INTPTR_T	intptr_t <<1>>

intptr_t はポインター(アドレス)を保持できる大きな整数です。インテル® Fortran コンパイラーでは、32 ビット・アプリケーションをビルドする場合は INTEGER(4)、64 ビット・アプリケーションをビルドする場合は INTEGER(8) です。これは、Fortran 組込み関数 INT_PTR_KIND() によって返される値です。

実行環境ルーチン

実行環境ルーチンを使用して、スレッドおよび並列環境の監視と操作を行います。

ルーチン	説明
SUBROUTINE OMP_SET_NUM_THREADS(num_threads) INTEGER(int) num_threads	呼び出し側のスレッドにより作成される後続の並列領域に使用するスレッド数を設定します。
SUBROUTINE OMP_SET_DYNAMIC(dynamic_threads) LOGICAL dynamic_threads	並列領域の実行に使用するスレッド数の動的な調整を有効または無効にします。dynamic_threads が .TRUE. の場合は、動的スレッドは有効です。dynamic_threads が .FALSE. の場合は、動的スレッドは無効です。動的スレッドはデフォルトでは無効です。
SUBROUTINE OMP_SET_NESTED(nested) LOGICAL(log) nested	入れ子構造の並列処理を有効または無効にします。nested が .TRUE. の場合は、入れ子構造の並列処理は有効です。nested が .FALSE. の場合は、入れ子構造の並列処理は無効です。入れ子構造の並列処理はデフォルトでは無効です。
INTEGER(int) FUNCTION OMP_GET_NUM_THREADS()	現在の並列領域に使用されているスレッドの数を返します。 この関数は、OMP_SET_NUM_THREADS() 関数からの呼び出し側スレッドによって継承される値は返しません。
INTEGER(int) FUNCTION OMP_GET_MAX_THREADS()	呼び出し側のスレッドにより作成される後続の並列領域に利用可能なスレッド数を返します。
INTEGER(int) FUNCTION OMP_GET_THREAD_NUM()	現在の並列領域のコンテキスト内の呼び出し側スレッド数を返します。
INTEGER(int) FUNCTION OMP_GET_NUM_PROCS()	プログラムで利用できるプロセッサー数を決定します。

LOGICAL(log) FUNCTION OMP_IN_PARALLEL()	並列で実行されている並列領域の動的な範囲内で呼ばれた場合、.TRUE. を返します。そうでない場合は、.FALSE. を返します。
LOGICAL(log) FUNCTION OMP_IN_FINAL()	final タスク領域内で呼ばれた場合、.TRUE. を返します。そうでない場合は、.FALSE. を返します。
LOGICAL(log) FUNCTION OMP_GET_DYNAMIC()	動的なスレッド調整が有効の場合は、.TRUE. を返します。そうでない場合は、.FALSE. を返します。
LOGICAL FUNCTION OMP_GET_NESTED()	入れ子構造の並列処理が有効な場合は、.TRUE. を返します。そうでない場合は、.FALSE. を返します。
INTEGER FUNCTION OMP_GET_THREAD_LIMIT()	OpenMP* プログラムで同時に実行するスレッド数の最大値を返します。
SUBROUTINE OMP_SET_MAX_ACTIVE_LEVELS (max_active_levels) INTEGER max_active_levels	入れ子構造のアクティブな並列領域の数を制限します。負の max_active_levels が指定された場合、呼び出しは無視されます。
INTEGER FUNCTION OMP_GET_MAX_ACTIVE_LEVELS()	入れ子構造のアクティブな並列領域の最大数を返します。
INTEGER FUNCTION OMP_GET_LEVEL()	呼び出しが含まれたタスクを囲む、入れ子構造の並列領域(アクティブ、非アクティブにかかわらず)の数を返します。暗黙的な並列領域は含まれません。
INTEGER FUNCTION OMP_GET_ACTIVE_LEVEL()	呼び出しが含まれたタスクを囲む、入れ子構造のアクティブな並列領域の数を返します。
INTEGER FUNCTION OMP_GET_ANCESTOR_THREAD_NUM(level) INTEGER level	指定された現在のスレッドの入れ子レベルに対する先祖のスレッド番号を返します。
INTEGER FUNCTION OMP_GET_TEAM_SIZE(level) INTEGER level	現在のスレッドの指定された入れ子レベルの先祖または現在のスレッドが所属するスレッドチームのサイズを返します。
SUBROUTINE OMP_SET_SCHEDULE(kind,chunk_size) INTEGER(KIND=omp_sched_kind) kind INTEGER(int) chunk_size	'runtime' がスケジュール種別として使用されている場合に適用されるワークシェア・ループのスケジュールを決定します。
SUBROUTINE OMP_GET_SCHEDULE(kind,chunk_size) INTEGER(KIND=omp_sched_chunk_size) kind INTEGER(int) chunk_size	'runtime' スケジュールが使用されている場合に適用されるワークシェア・ループのスケジュールを返します。
INTEGER(KIND=OMP_PROC_BIND_KIND) OMP_GET_PROC_BIND()	OMP_PROC_BIND 環境変数で設定される現在アクティブなスレッド・アフィニティー・ポリシーを返します。 このポリシーは、後続の入れ子構造の並列領域で使用されます。
INTEGER(int) FUNCTION OMP_GET_NUM_PLACES()	初期タスクのプレースリスト(通常、スレッド、コア、またはソケット)で、実行環境が利用可能なプレースの数を返します。

INTEGER(int) FUNCTION OMP_GET_PLACE_NUM_PROCS(place_num) INTEGER(int) place_num	プレース番号 place_num に関連付けられたプロセッサー数を返します。place_num が負または OMP_GET_NUM_PLACES() 以上の場合はゼロを返します。
SUBROUTINE OMP_GET_PLACE_PROC_IDS(place_num,ids) INTEGER(int) place_num INTEGER(int) ids(*)	プレース番号 place_num に関連付けられた各プロセッサーの ID 番号を返します。ID 番号は負以外で、その意味は実装定義に依存します。ID 番号は配列 ids で返され、配列内の順序は実装定義に依存します。ids には、少なくとも OMP_GET_PLACE_NUM_PROCS(place_num) 要素が含まれていないければなりません。place_num が OMP_GET_NUM_PLACES() 以上の場合、ルーチンは何も返しません。
INTEGER(int) FUNCTION OMP_GET_PLACE_NUM()	到達スレッドがバインドされているプレースの番号を返します。プレース番号は、0 から OMP_GET_NUM_PLACES() - 1 の範囲の値です。到達スレッドがプレースにバインドされていない場合は、-1 を返します。
INTEGER(int) FUNCTION OMP_GET_DEFAULT_DEVICE()	デフォルトのデバイス番号を返します。
SUBROUTINE OMP_SET_DEFAULT_DEVICE(device_number) INTEGER(int) device_number	デフォルトのデバイス番号を設定します。
INTEGER(int) FUNCTION OMP_GET_NUM_DEVICES()	ターゲットデバイス数を取得します。
INTEGER(int) FUNCTION OMP_GET_NUM_TEAMS()	現在のチーム領域のチーム数を取得します。
INTEGER(int) FUNCTION OMP_GET_TEAM_NUM()	呼び出し側のスレッドのチーム番号を取得します。
LOGICAL(log) FUNCTION OMP_GET_CANCELLATION()	キャンセルが有効な場合は .TRUE. を返します。そうでない場合は、.FALSE. を返します。 このルーチンは、環境変数 OMP_CANCELLATION の設定の影響を受けます。
LOGICAL(log) FUNCTION OMP_IS_INITIAL_DEVICE()	現在のタスクがホストデバイスで実行している場合は .TRUE. を返します。そうでない場合は、.FALSE. を返します。
INTEGER(int) FUNCTION OMP_GET_INITIAL_DEVICE()	ホストデバイスのデバイス番号を返します。デバイス番号は実装定義に依存します。0 から OMP_GET_NUM_DEVICES() - 1 の範囲の場合、すべてのデバイス構造/ルーチンで有効です。そうでない場合、デバイス・メモリー・ルーチンでのみ有効で、DEVICE 節では無効です。
INTEGER(int) FUNCTION OMP_GET_MAX_TASK_PRIORITY()	PRIORITY 節で指定可能な最大値を返します。

ロックルーチン

ロックルーチンを使用して OpenMP* ロックを操作します。

関数	説明
SUBROUTINE OMP_INIT_LOCK(svar) INTEGER (KIND=OMP_LOCK_KIND)svar	後続の呼び出しに使用する単純なロック変数 svar に関連付けられたロックを初期化します。
SUBROUTINE OMP_INIT_LOCK_WITH_HINT(svar, hint) INTEGER (KIND=OMP_LOCK_KIND)svar INTEGER (KIND=OMP_LOCK_HINT_KIND)hint	svar に関連付けられたロックを解除状態に初期化します。オプションで hint を基に特定のロック実装を選択できます。
SUBROUTINE OMP_DESTROY_LOCK(svar) INTEGER(KIND=OMP_LOCK_KIND)svar	svar により指定されたロックを未定義または未初期化にします。ロックは初期化されており、ロックされていてはなりません。
SUBROUTINE OMP_SET_LOCK(svar) INTEGER(KIND=OMP_LOCK_KIND)svar	svar に関連付けられているロックが使用可能な状態になるまで実行中のスレッドを強制的に待機させます。ロックが使用可能になると、スレッドにはそのロックの所有権が与えられます。ロックは初期化されていなければなりません。
SUBROUTINE OMP_UNSET_LOCK(svar) INTEGER(KIND=OMP_LOCK_KIND)svar	svar に関連付けられているロックの所有権から実行スレッドを解放します。svar に関連付けられたロックを実行中のスレッドが所有していない場合の動作は不定です。
LOGICAL(log) OMP_TEST_LOCK(svar) INTEGER(KIND=OMP_LOCK_KIND)svar	svar に関連付けられているロックを設定しようと試みます。成功した場合は、.TRUE. を返します。そうでない場合は、.FALSE. を返します。ロックは初期化されていなければなりません。
SUBROUTINE OMP_INIT_NEST_LOCK(nvar) INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar	後続の呼び出しに使用する入れ子されたロック変数 nvar に関連付けられた入れ子されたロックを初期化します。
SUBROUTINE OMP_INIT_NEST_LOCK_WITH_HINT(nvar, hint) INTEGER (KIND=OMP_NEST_LOCK_KIND)nvar INTEGER (KIND=OMP_LOCK_HINT_KIND)hint	nvar に関連付けられた入れ子されたロックを解除状態に初期化します。オプションで hint を基に特定のロック実装を選択できます。nvar の入れ子カウントはゼロに設定されます。
SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar) INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar	nvar に関連付けられている入れ子されたロックを未定義または未初期化にします。ロックは初期化されており、ロックされていてはなりません。
SUBROUTINE OMP_SET_NEST_LOCK(nvar) INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar	nvar に関連付けられている入れ子されたロックが使用可能な状態になるまで実行中のスレッドを強制的に待機させます。スレッドがすでにロックを所有している場合は、ロックの入れ子カウント数は増えます。ロックは初期化されていなければなりません。

SUBROUTINE OMP_UNSET_NEST_LOCK(nvar) INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar	入れ子カウント数がゼロの場合は、nvar に関連付けられた入れ子されたロックの所有権から実行中のスレッドを解放します。解放されない場合、入れ子カウント数は減ります。nvar に関連付けられた入れ子のロックを、実行中のスレッドが所有していない場合の動作は不定です。
INTEGER(int) OMP_TEST_NEST_LOCK(nvar) INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar	nvar により指定されている入れ子されたロックを設定しようと試みます。成功した場合は入れ子カウント数を返し、失敗した場合は 0 を返します。

タイミングルーチン

関数	説明
REAL (dp) FUNCTION OMP_GET_WTIME()	任意の参照時間から経過したウォールクロック時間(秒)に等しい倍精度値を返します。参照時間は、プログラム実行中には変更されません。
REAL (dp) FUNCTION OMP_GET_WTICK()	連続するクロック刻みの間隔の秒数に等しい倍精度値を返します。

次の PARAMETER 定数は OMP_LIB.MOD で定義され、OMP_SET_SCHEDULE と OMP_GET_SCHEDULE の KIND 仮引数で設定または返すことができます。

```
integer(omp_sched_kind), parameter :: omp_sched_static  = 1
integer(omp_sched_kind), parameter :: omp_sched_dynamic = 2
integer(omp_sched_kind), parameter :: omp_sched_guided  = 3
integer(omp_sched_kind), parameter :: omp_sched_auto    = 4
```

次の PARAMETER 定数は OMP_LIB.MOD で定義され、OMP_GET_PROC_BIND によって返される値を表します。

```
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_false  = 0
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_true   = 1
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_master = 2
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_close  = 3
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_spread = 4
```

次の PARAMETER 定数は OMP_LIB.MOD で定義され、OMP_INIT_LOCK_WITH_HINT と OMP_INIT_NEST_LOCK_WITH_HINT の HINT 仮引数で設定できます。

```
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_none           = 0
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_uncontended    = 1
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_contended      = 2
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_nonspeculative = 4
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_speculative    = 8
```

Fortran では、+ 演算子を使用してヒントを組み合わせることができます。ヒントを組み合わせた場合の動作は、実装定義に依存します。omp_lock_hint_none とヒントを組み合わせた場合、そのヒントを指定した場合と同じ効果があります。次の制約事項は、ヒントを組み合わせる場合に適用されます。

- omp_lock_hint_uncontended と omp_lock_hint_contended を組み合わせることはできません。
- omp_lock_hint_nonspeculative と omp_lock_hint_nonspeculative を組み合わせることはできません。

メモリー管理ルーチン

この機能は ifx でのみ利用できます。

関数	説明
FUNCTION omp_init_allocator(memspace, ntraits, traits) INTEGER (KIND=omp_allocator_handle_kind) omp_init_allocator INTEGER (KIND=omp_memspace_handle_kind), INTENT(IN) :: memspace INTEGER,INTENT(IN) :: ntraits TYPE(omp_alloctrait), INTENT(IN) :: traits(*)	アロケーターを初期化してメモリー空間に関連付けて、アロケーター・ハンドルを返します。memspace 引数は、「OpenMP* メモリー空間アロケーター」で定義されている事前定義されたメモリー空間のいずれかでなければなりません。ntraits 引数は、特性配列引数に含まれる特性の数を指定します。特性が ntraits で指定された数よりも少ない場合、動作は不定です。dynamic_allocators を指定する REQUIRES ディレクティブは、この関数を呼び出すターゲット領域の同じコンパイル単位になければなりません。
SUBROUTINE omp_destroy_allocator (allocator) INTEGER(KIND=omp_allocator_handle_kind), INTENT(IN) :: allocator	アロケーター・ハンドル引数によって使用されるすべてのリソースを解放します。アロケーター引数は、事前定義されたアロケーターのハンドルで会ってはなりません。dynamic_allocators を指定する REQUIRES ディレクティブは、この関数を呼び出すターゲット領域の同じコンパイル単位になければなりません。
SUBROUTINE omp_set_default_allocator (allocator) INTEGER(KIND=omp_allocator_handle_kind) :: allocator	アロケーターを指定しない ALLOCATE 節やディレクティブによるメモリー割り当てに使用されるデフォルトのメモリー・アロケーターを設定します。アロケーター引数は有効なアロケーターでなければなりません。
FUNCTION omp_get_default_allocator () INTEGER(KIND=omp_allocator_handle_kind) :: omp_get_default_allocator	アロケーターを指定しない ALLOCATE 節やディレクティブによる割り当てに使用されるアロケーターのアロケーター・ハンドルを返します。
TYPE(c_ptr) FUNCTION omp_target_alloc (size, device_num) USE, INTRINSIC :: ISO_C_BINDING INTEGER(c_size_t) :: size INTEGER(c_int) :: device_num	サイズが size バイトのストレージのデバイスアドレスを返します。割り当て要求を満たせない場合、null アドレスを返します。device_num で指定されたデバイス上に割り当てます。

SUBROUTINE omp_target_free (device_ptr, device_num) USE, INTRINISIC :: ISO_C_BINDING TYPE (c_ptr) :: device_ptr INTEGER(c_int) :: device_num	device_ptr で指定されたデバイス上の device_ptr が指す位置にある、以前に割り当てられたメモリー割り当てを解放します。
TYPE(c_ptr) FUNCTION omp_target_alloc_host (size, device_num) USE, INTRINSIC :: ISO_C_BINDING INTEGER(c_size_t) :: size INTEGER(c_int) :: device_num	ホストメモリー上に割り当てられたサイズが size バイトのストレージのアドレスを返します。同じポインターを使用して、ホストとすべてのサポートされているデバイスのメモリーにアクセスできます。割り当て要求に失敗した場合、null ポインターを返します。
TYPE(c_ptr) FUNCTION omp_target_alloc_device (size, device_num) USE, INTRINSIC :: ISO_C_BINDING INTEGER(c_size_t) :: size INTEGER(c_int) :: device_num	サイズが size バイトのストレージのアドレスを返します。デバイス割り当ては、デバイス・ローカル・メモリーに device_ptr が存在する場合、device_ptr で指定されたデバイスによって所有されます。一般に、そのデバイスのみが割り当てにアクセスできますが、割り当てを他のデバイスやホストの割り当てメモリーにコピーすることもできます。戻り値が null ポインターの場合、割り当てに失敗したことを示します。
TYPE(c_ptr) FUNCTION omp_target_alloc_shared (size, device_num) USE, INTRINSIC :: ISO_C_BINDING INTEGER(c_size_t) :: size INTEGER(c_int) :: device_num	サイズが size バイトのストレージのアドレスを返します。共有割り当ては、ホストと 1 つ以上の関連デバイスによって共有され、ホストと 1 つ以上のデバイス間で移行されます。割り当てに失敗すると null ポインターが返されます。

派生型 omp_allocator は OMP_LIB.MOD で定義されており、omp_init_allocator 関数の特性引数の型として使用されます。

```
type omp_alloctrait
  integer(kind=omp_alloctrait_key_kind) :: key
  integer(kind=omp_alloctrait_val_kind) :: value
end type omp_alloctrait
```

以下のパラメーター定数は OMP_LIB.MOD で定義されており、omp_alloctrait 型の派生型で割り当て特性キーの指定に使用できます。

```
integer(omp_alloctrait_key_kind), parameter :: omp_atk_sync_hint  = 1
integer(omp_alloctrait_key_kind), parameter :: omp_atk_alignment  = 2
integer(omp_alloctrait_key_kind), parameter :: omp_atk_access     = 3
integer(omp_alloctrait_key_kind), parameter :: omp_atk_pool_size  = 4
integer(omp_alloctrait_key_kind), parameter :: omp_atk_fallback   = 5
integer(omp_alloctrait_key_kind), parameter :: omp_atk_fb_data    = 6
integer(omp_alloctrait_key_kind), parameter :: omp_atk_atk_pinned = 7
integer(omp_alloctrait_key_kind), parameter :: omp_atk_partition  = 8
```

以下のパラメーター定数は OMP_LIB.MOD で定義されており、omp_alloctrait 型の派生型で割り当て特性値の設定に使用できます。

```
integer(omp_alloctrait_val_kind), parameter :: omp_atv_false            =  0
integer(omp_alloctrait_val_kind), parameter :: omp_atv_true             =  1
integer(omp_alloctrait_val_kind), parameter :: omp_atv_default          =  2
integer(omp_alloctrait_val_kind), parameter :: omp_atv_contended        =  3
integer(omp_alloctrait_val_kind), parameter :: omp_atv_uncontended      =  4
integer(omp_alloctrait_val_kind), parameter :: omp_atv_sequential       =  5
integer(omp_alloctrait_val_kind), parameter :: omp_atv_private          =  6
integer(omp_alloctrait_val_kind), parameter :: omp_atv_all              =  7
integer(omp_alloctrait_val_kind), parameter :: omp_atv_thread           =  8
integer(omp_alloctrait_val_kind), parameter :: omp_atv_pteam            =  9
integer(omp_alloctrait_val_kind), parameter :: omp_atv_cgroup           = 10
integer(omp_alloctrait_val_kind), parameter :: omp_atv_default_mem_fb   = 11
integer(omp_alloctrait_val_kind), parameter :: omp_atv_null_fb          = 12
integer(omp_alloctrait_val_kind), parameter :: omp_atv_abort_fb         = 13
integer(omp_alloctrait_val_kind), parameter :: omp_atv_allocator_fb     = 14
integer(omp_alloctrait_val_kind), parameter :: omp_atv_environment      = 15
integer(omp_alloctrait_val_kind), parameter :: omp_atv_nearest          = 16
integer(omp_alloctrait_val_kind), parameter :: omp_atv_blocked          = 17
integer(omp_alloctrait_val_kind), parameter :: omp_atv_interleaved      = 18
```

関連情報

インテルの OpenMP* 拡張ルーチン
定義済みプリプロセッサー・シンボル(英語)

インテルの OpenMP* 拡張ルーチン

インテル® コンパイラーでは、OpenMP* ランタイム・ライブラリーへの拡張機能として、次のルーチングループをサポートしています。

- 実行環境の取得と設定
- 並列スレッドのスタックサイズの取得と設定
- メモリー割り当て
- スループット実行モードにおけるスレッドのスリープ時間の取得と設定

ここで説明するインテル拡張ルーチンは、ライブラリー・コードとアプリケーションが目的どおりに機能することを確認する低レベルのチューニングに使用できます。これらのルーチンは、一般にその他の OpenMP* 互換コンパイラーで認識されず、別のコンパイラーではリンクの段階で失敗することがあります。これらの OpenMP* ルーチンを実行するには、/Qopenmp-stubs(Windows*)または -qopenmp-stubs(Linux* および macOS*)オプションを使用します。

多くの場合、環境変数は拡張ライブラリー・ルーチンの代わりに使用されます。例えば、並列スレッドのスタックサイズは、KMP_SET_STACKSIZE_S() ライブラリー・ルーチンではなく、OMP_STACKSIZE 環境変数を使用して設定できます。

注:
インテル拡張ルーチンへのランタイムの呼び出しは、対応する環境変数の設定よりも優先します。

実行環境

関数	説明
SUBROUTINE KMP_SET_DEFAULTS(STRING) CHARACTER*(*) STRING	引数が "**|**" で区切られた変数のリストとして定義される OpenMP* 環境変数を設定します。
SUBROUTINE KMP_SET_LIBRARY_THROUGHPUT()	実行モードをスループットに設定します(デフォルト)。アプリケーションによりランタイム環境を特定できます。マルチユーザー環境で使用します。
SUBROUTINE KMP_SET_LIBRARY_TURNAROUND()	実行モードをターンアラウンドに設定します。専用並列(シングルユーザー)環境で使用します。
SUBROUTINE KMP_SET_LIBRARY_SERIAL()	実行モードをシリアルに設定します。
SUBROUTINE KMP_SET_LIBRARY(LIBNUM) INTEGER (KIND=OMP_INTEGER_KIND) LIBNUM	関数に渡された値による実行モードに設定します。有効な値は以下のとおりです。 • **1** - シリアルモード • **2** - ターンアラウンド・モード • **3** - スループット・モード 最初の並列領域が実行される前にこのルーチンを呼び出します。
FUNCTION KMP_GET_LIBRARY() INTEGER (KIND=OMP_INTEGER_KIND) KMP_GET_LIBRARY	現在の実行モードに対応する値を返します。 • **1** - シリアル • **2** - ターンアラウンド • **3** - スループット

スタックサイズ

関数	説明
FUNCTION KMP_GET_STACKSIZE_S() INTEGER(KIND=KMP_SIZE_T_KIND) & KMP_GET_STACKSIZE_S	各並列スレッドがプライベート・スタックとして使用するバイト数を返します。この値は、最初の並列領域の前に KMP_SET_STACKSIZE_S() ルーチンで変更するか、または KMP_STACKSIZE 環境変数で変更できます。
FUNCTION KMP_GET_STACKSIZE() INTEGER KMP_GET_STACKSIZE	下位互換性のみ提供します。異なるインテル® プロセッサーとの互換性には KMP_GET_STACKSIZE_S() ルーチンを使用します。
SUBROUTINE KMP_SET_STACKSIZE_S(size) INTEGER (KIND=KMP_SIZE_T_KIND)size	各並列スレッドがプライベート・スタックとして使用するバイト数を size に設定します。この値は、KMP_STACKSIZE 環境変数で設定することもできます。KMP_SET_STACKSIZE_S() 有効にするには、プログラムの最初の(動的に実行された)並列領域の先頭の前に呼び出す必要があります。
SUBROUTINE KMP_SET_STACKSIZE_S(size) INTEGER size	下位互換性のみ提供します。異なるインテル® プロセッサーとの互換性には KMP_SET_STACKSIZE_S(size) を使用します。

メモリー割り当て

インテル® コンパイラーでは、OpenMP* ランタイム・ライブラリーに対する拡張機能として、メモリー割り当てルーチンを実装しています。そのため、スレッドは各スレッドにローカルなヒープからメモリーを割り当てることが可能です。これらのルーチンは、KMP_MALLOC()、KMP_CALLOC()、および KMP_REALLOC() です。

これらのルーチンによって割り当てられたメモリーは、KMP_FREE() ルーチンによって解放しなければなりません。あるスレッドでメモリーを割り当て、別のスレッドでメモリーを解放することは可能ですが、このような処理はパフォーマンスを多少低下させます。同期化の必要ないため、ローカルなヒープで作業するとアプリケーション・パフォーマンスが向上する場合があります。

関数	説明
FUNCTION KMP_MALLOC(size) INTEGER(KIND=KMP_POINTER_KIND)KMP_MALLOC INTEGER(KIND=KMP_SIZE_T_KIND)size	スレッド・ローカル・ヒープから size バイトのメモリーブロックを割り当てます。
FUNCTION KMP_CALLOC(nelem,elsize) INTEGER(KIND=KMP_POINTER_KIND)KMP_CALLOC INTEGER(KIND=KMP_SIZE_T_KIND)nelem INTEGER(KIND=KMP_SIZE_T_KIND)elsize	スレッド・ローカル・ヒープからサイズ elsize の nelem 要素の配列を割り当てます。
FUNCTION KMP_REALLOC(ptr, size) INTEGER(KIND=KMP_POINTER_KIND)KMP_REALLOC INTEGER(KIND=KMP_POINTER_KIND) ptr INTEGER(KIND=KMP_SIZE_T_KIND)size	スレッド・ローカル・ヒープからアドレス ptr および size バイトにメモリーブロックを再割り当てします。
SUBROUTINE KMP_FREE(ptr) INTEGER (KIND=KMP_POINTER_KIND)ptr	スレッド・ローカル・ヒープからアドレス ptr のメモリーブロックを解放します。 メモリーは、事前に KMP_MALLOC()、KMP_CALLOC()、または KMP_REALLOC() で割り当てられている必要があります。

スレッドのスリープ時間

スループット実行モードでは、スレッドは、新しい並列作業を並列領域の終わりで待機し、一定期間が経過するとスリープ状態に移行します。この待機期間を設定するには、KMP_BLOCKTIME 環境変数または KMP_SET_BLOCKTIME() 関数を使用します。

関数	説明
FUNCTION KMP_GET_BLOCKTIME(INTEGER KMP_GET_BLOCKTIME	並列領域の実行が終了した後、スレッドがスリープ状態になるまで待機する時間(ミリ秒単位)を返します。この時間は、KMP_BLOCKTIME 環境変数または KMP_SET_BLOCKTIME() 関数によって設定された値です。
FUNCTION KMP_SET_BLOCKTIME(msec) INTEGER msec	並列領域の実行が終了した後、スレッドがスリープ状態になるまで待機する時間(ミリ秒単位)を設定します。このルーチンは、呼び出し側のスレッドおよび呼び出し側のスレッドにより形成される OpenMP* チームのスレッドのブロック時間の設定に影響します。その他のスレッドのブロック時間には影響しません。

関連情報

openmp-stubs、Qopenmp-stubs(英語)
OpenMP* ランタイム・ライブラリー・ルーチン
OpenMP* サポート・ライブラリー

OpenMP* のサポート・ライブラリー

インテル® Fortran コンパイラーでは、OpenMP* のサポート・ライブラリーが提供されています。ライブラリーには次のような種類があります。

- **パフォーマンス・ライブラリー:** OpenMP* 並列実行をサポートします。
- **スタブ・ライブラリー:** OpenMP* アプリケーションのシリアル実行をサポートします。

Linux* および macOS* システムでは、各ライブラリーでダイナミック・リンクとスタティック・リンクの両方が利用できます。Windows* システムでは、ダイナミック・リンクのみ利用できます。

パフォーマンス・ライブラリー

これらのライブラリーを使用するには、`/Qopenmp`(Windows*)または `-qopenmp`(Linux* および macOS*)オプションを指定します。

OpenMP* を使用するオプションは、インテル製マイクロプロセッサーおよび互換マイクロプロセッサーの両方で利用可能ですが、両者では結果が異なります。両者の結果が異なる可能性のある OpenMP* 構造および機能の主なリストは次のとおりです:ロック(内部的なものおよびユーザーが利用可能なもの)、`SINGLE` 構造、バリア(暗黙的および明示的)、並列ループ・スケジュール、リダクション、メモリーの割り当て、スレッド・アフィニティー、バインド。

オペレーティング・システム	ダイナミック・リンク	スタティック・リンク
Linux*	`libiomp5.so`	`libiomp5.a`
macOS*	`libiomp5.dylib`	`libiomp5.a`
Windows*	`libiomp5md.lib` `libiomp5md.dll`	なし

OpenMP* サポート・ライブラリーの多くのルーチンは、互換マイクロプロセッサーよりもインテル製マイクロプロセッサーでより最適化されます。

スタブ・ライブラリー

これらのライブラリーを使用するには、`/Qopenmp-stubs`(Windows*)または `-qopenmp-stubs`(Linux* および macOS*)オプションを指定します。これにより、OpenMP* アプリケーションをシリアルモードでコンパイルし、OpenMP* ルーチンとインテル固有の拡張ルーチンのスタブを提供することができます。

オペレーティング・システム	ダイナミック・リンク	スタティック・リンク
Linux*	`libiompstubs5.so`	`libiompstubs5.a`
macOS*	`libiompstubs5.dylib`	`libiompstubs5.a`
Windows*	`libiompstubs5md.lib` `libiompstubs5md.dll`	なし

実行モード

コンパイラーは、実行時に指定した実行モードでアプリケーションを実行することができます。ライブラリーは、ターンアラウンド・モード(`turnaround`)、スループット・モード(`throughput`)、シリアルモード(`serial`)をサポートしています。`KMP_LIBRARY` 環境変数(英語)を使用して、実行時にモードを選択します。

モード	説明
throughput (デフォルト)	スループット・モードでは、プログラムは他の実行プログラムへ作業を渡し、リソースの使用を調整することで、動的環境において効率良く実行することができます。 並列マシン上の負荷が一定ではない、またはジョブストリームが予測できないマルチユーザー環境下では、スループット用に設計しチューニングするほうが良い場合もあります。これにより、複数のジョブを同時に実行した際の合計時間を最小限に抑えることができます。このモードでは、ワーカースレッドは追加の並行作業の待機中、他のスレッドの作業を手伝います。 1 つの並列領域の実行が完了すると、スレッドは新しい並列作業が使用可能になるまで待機します。その後一定期間が経過すると、スレッドが待機状態からスリープ状態に移行します。スリープ状態では、次の並列作業が使用できるようになるまでの間、プロセッサーとリソースは、並列領域間で実行される非 OpenMP* のスレッドコードや他のアプリケーションによる別の作業に使用することができます。 スリープ状態に移行するまでの待機時間を設定するには、`KMP_BLOCKTIME` 環境変数または `KMP_SET_BLOCKTIME()` 関数を使用します。ブロック時間の値を小さくすると、並列領域間で実行される非 OpenMP* スレッドコードを含むアプリケーションの場合には、全体的なパフォーマンスが向上します。ブロック時間の値を大きくすると、スレッドが OpenMP* 実行専用に予約されている場合には適していますが、他の同時実行 OpenMP* やスレッド・アプリケーションに悪影響を与える可能性があります。
turnaround	ターンアラウンド・モードは、並列計算を行うすべてのプロセッサーをアクティブな状態で維持して、単一ジョブの実行時間を最小限に抑えるよう設計されています。ワーカースレッドは、他のスレッドの作業を手伝うことなく、アクティブな状態で追加の並列作業を待機します(ただし、ワーカースレッドはまだ `KMP_BLOCKTIME` に制御されます)。すべてのプロセッサーが、プログラムの全実行に対し排他的に割り当てられる専用(バッチまたはシングルユーザー)並列環境では、常にすべてのプロセッサーを効果的に使用することが最も重要です。 **注:** 過剰なシステムリソースの割り当てを避けてください。過剰なシステムリソースの割り当ては、スレッドが多すぎるか、または実行時に利用可能なプロセッサーが少なすぎる場合に発生します。システムリソースが過剰に割り当てられると、このモードはパフォーマンスの低下を起こします。この問題が発生した場合、スループット・モードを使用してください。
serial	シリアルモードは、並列アプリケーションをシングルスレッドとして強制的に実行します。

関連情報

openmp、Qopenmp(英語)
openmp-stubs、Qopenmp-stubs(英語)
サポートされる環境変数(英語)

OpenMP* ライブラリーの使用

このセクションでは、コマンドラインで OpenMP* ライブラリーを設定して使用するのに必要なステップを説明します。Windows* システムでは、Microsoft* Visual Studio* 開発環境で OpenMP* ライブラリーを使用してコンパイルされたアプリケーションをビルドすることもできます。

OpenMP* ライブラリーにより使用されるオプションとライブラリーのリストは、「OpenMP* のサポート・ライブラリー」を参照してください。

インテル® Fortran コンパイラーへのアクセスを確立するために環境を設定し、リンクで適切な OpenMP* ライブラリーが利用できるようにします。Windows* システムでは、適切なバッチ(`.bat`)ファイルを実行するか、またはすでに環境設定されているコマンドライン・ウィンドウ(コンパイラーのプログラムメニューから起動可能)を使用します。Linux* および macOS* システムでは、適切なスクリプトファイル(`setvars` ファイル)をソース(読み込み)します。

gfortran コンパイラーを、OpenMP* API 関数を含むインテルの OpenMP* ライブラリーとともに使用するには、次の操作を行います。

1. `use omp_lib` 文を使用して、`include` ディレクトリーにある `omp_lib.f90` ソースファイルをコンパイルします。
2. 生成されたモジュールファイルへの適切なパスとともに、コマンドラインで `-I` オプションを追加します。

コンパイルで、コンパイル時に使用される `omp_lib.h` または `omp_lib.mod` のバージョンがコンパイラーで提供されているバージョンであることを確認します。

注:
gcc コンパイラーまたは Microsoft* コンパイラーの使用時は、不適切なヘッダー/モジュールファイルを誤って使用することがあるので注意してください。これを避けるには、別のディレクトリーにヘッダー/モジュールファイルをコピーし、`-I` オプションを使用して、正しい `include` パスを指定します。

プログラムでデータ構造やクラスが使用されており、それらが `omp_lib.h` ファイルで定義されるデータ型を持つメンバーを含む場合、そのようなデータ構造を使用するソースファイルはすべて同じ `omp_lib.h` ファイルでコンパイルする必要があります。

インテル® Fortran コンパイラーのコマンドは `ifort` です。

コンパイラーで使用される OpenMP* ライブラリーとオプションに関する情報は、「OpenMP* サポート・ライブラリー」を参照してください。

コマンドラインの例(Windows*)

互換ライブラリーを使用して、1 つのコマンドでアプリケーション全体をコンパイルおよびリンク(ビルド)するには、次のコマンドを指定します。

ファイルの種類	コマンド
Fortran ソース、ダイナミック・リンク	`ifort /MD /Qopenmp hello.f90`

Microsoft* Visual C++* コンパイラーを使用する場合は、インテルの OpenMP* 互換ライブラリーとリンクします。Microsoft* OpenMP* ランタイム・ライブラリー(`vcomp`)にリンクされないように、リンカーオプションを使用して(`/link` の後で)、インテルの OpenMP* 互換ライブラリー名を明示的に渡す必要があります。

ファイルの種類	コマンド
C ソース、ダイナミック・リンク	`cl /MD /openmp hello.c /link /nodefaultlib:vcomp libiomp5md.lib`

また、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーと Visual C++* コンパイラーの両方を使用して、アプリケーションの一部をコンパイルし、オブジェクト・ファイルを作成することができます(オブジェクト・レベルの互換性)。この例では、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーはアプリケーション全体をコンパイルしてリンクします。

ファイルの種類	コマンド

C ソースと Fortran ソースの混在、ダイナミック・リンク	`cl /MD /openmp /c f1.c f2.c`
	`ifort /MD /Qopenmp /c f3.f f4.f`
	`ifort /MD /Qopenmp f1.obj f2.obj f3.obj f4.obj /Feapp /link /nodefaultlib:vcomp`

最初のコマンドは、Visual C++* コンパイラーでコンパイルされた 2 つのオブジェクト・ファイルを生成します。2 番目のコマンドはインテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーによってコンパイルされたさらに 2 つのオブジェクト・ファイルを生成します。最後のコマンドは 4 つのオブジェクト・ファイルをアプリケーションにリンクします。

また、下記の 3 行目は、Visual C++* リンカーを使用してアプリケーションをリンクし、互換ライブラリーの `libiomp5md.lib` を 3 番目のコマンドの終わりに指定しています。

ファイルの種類	コマンド

C ソースと Fortran ソースの混在、ダイナミック・リンク	`cl /MD /openmp /c f1.c f2.c`
	`ifort /MD /Qopenmp /c f3.f f4.f`
	`link f1.obj f2.obj f3.obj f4.obj /out:app.exe /nodefaultlib:vcomp libiomp5md.lib`

次の例は、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーで複数のファイルのプロシージャー間の最適化を使用し、Visual C++* コンパイラーで複数のファイルをコンパイルし、Visual C++* リンカーでオブジェクト・ファイルをリンクして、実行ファイルを作成する例を示しています。

ファイルの種類	コマンド

C ソースと Fortran ソースの混在、ダイナミック・リンク	`ifort /MD /Qopenmp /O3 /Qipo /Qipo-c f1.f f2.f f3.f`
	`cl /MD /openmp /O2 /c f4.c f5.c`
	`cl /MD /openmp /O2 ipo_out.obj f4.obj f5.obj /Feapp /link /nodefaultlib:vcomp libiomp5md.lib`

最初のコマンドは、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーを使用して、デフォルトで、`ipo_out.obj` という名前の最適化された複数ファイル情報を持つ単一のオブジェクト・ファイルを生成します(`/Fe` オプションは必要ありません)。2 番目のコマンドは、Visual C++* コンパイラーを使用してさらに 2 つのオブジェクト・ファイルを生成します。3 番目のコマンドは、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーの OpenMP* ライブラリーを使用して、Visual C++* の `cl` コマンドで 3 つすべてのオブジェクト・ファイルをリンクします。

コマンドラインの例(Linux*)

macOS* 上でインテルの OpenMP* ライブラリーを使用して、1 つのコマンドでアプリケーション全体をコンパイルおよびリンク(ビルド)するには、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーの次のコマンドを指定します。

ファイルの種類	コマンド
Fortran ソース	`ifort -qopenmp hello.f90`

デフォルトでは、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーは OpenMP* ライブラリーのダイナミック・リンクを行います。スタティック・リンク(非推奨)を行うには、`-qopenmp-link=static` オプションを追加します。オプション `-qopenmp-link`(英語)は、Linux* および macOS* システムでリンカーが使用する OpenMP* ライブラリー(スタティックまたはダイナミック)を制御します(デフォルトは `-qopenmp-link=dynamic`)。

また、`icx`/`icpx` と `gcc`/`g++` コンパイラーの両方を使用して、アプリケーションの一部をコンパイルし、オブジェクト・ファイルを作成することができます(オブジェクト・レベルの互換性)。

この例では、`gcc` コンパイラーは、C ファイルの `foo.c`(`gcc` オプション `-fopenmp` で OpenMP* サポートは有効)をコンパイルし、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーは、インテルの OpenMP* ライブラリーを使用してアプリケーションをリンクしています。

ファイルの種類	コマンド
C ソース	`gcc -fopenmp -c foo.c` `ifort -qopenmp foo.o`

インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーの OpenMP* 互換ライブラリーを使用して、GNU* `gcc` または `g++` コンパイラーでアプリケーションをリンクする場合は、`-l` オプションでインテルの OpenMP* 互換ライブラリー名、`-l` オプションで Linux* `pthread` ライブラリー、`-L` オプションでインテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーがインストールされている場所にあるインテルのライブラリーへのパスを明示的に渡す必要があります。

ファイルの種類	コマンド
C ソース	`gcc -fopenmp -c foo.c bar.c` `gcc foo.o bar.o -liomp5 -lpthread -L<icx_dir>/lib`

オブジェクト・ファイルを混在させることもできますが、`gcc` の `-l` オプション、`-L` オプション、`-lpthread` オプションを指定しなくても済むように、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーでアプリケーションをリンクするほうが簡単です。

ファイルの種類	コマンド

C ソース	`gcc -fopenmp -c foo.c`
	`icx -qopenmp -c bar.c (Linux および macOS*)`
	`ifort -qopenmp foo.o bar.o (Linux および macOS*)`

gcc コンパイラー、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーでコンパイルされた OpenMP* オブジェクト・ファイルを混在させることができます。

注:
インテル® Fortran コンパイラーと gfortran コンパイラーでコンパイルされたオブジェクト・ファイルは混在させることはできません。

次の表は、インテル® Fortran コンパイラーを使用して、すべてのオブジェクトをリンクする例を示します。

ファイルの種類	コマンド

C ソースと Fortran ソースの混在	`icx -qopenmp -c ibar.c`
	`gcc -fopenmp -c gbar.c`
	`ifort -qopenmp -c foo.f`
	`ifort -qopenmp foo.o ibar.o gbar.o`

インテル® Fortran コンパイラーを使用する際、インテル® Fortran コンパイラー(`ifort`)でコンパイルされたメインプログラムが Fortran オブジェクト・ファイルにない場合は、リンク時に `-nofor-main` オプションを `ifort` コマンドラインに指定します。

注:
GNU* Fortran コンパイラー(`gfortran`)でインテル® Fortran コンパイラー(`ifort`)により作成されたオブジェクトを混在させないでください。代わりに、すべての Fortran ソースを同じ Fortran コンパイラーで再コンパイルしてください。GNU* Fortran コンパイラーは、Linux* オペレーティング・システムでのみ利用できます。

同様に、GNU* Fortran コンパイラー(`gfortran`)でリンクすると、インテル® C++ コンパイラー、GNU* C/C++ コンパイラー、GNU* Fortran コンパイラー(`gfortran`)でコンパイルされたオブジェクト・ファイルを混在させることができます。インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーの OpenMP* 互換ライブラリーを使用して、GNU* `gfortran` コンパイラーでアプリケーションを

リンクする場合は、-l オプションでインテルの OpenMP* 互換ライブラリー名とインテルの irc ライブラリー、-l オプションで Linux* pthread ライブラリー、-L オプションでインテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーがインストールされている場所にあるインテルのライブラリーへのパスを明示的に渡す必要があります。リンク行で -fopenmp オプションを指定する必要はありません。

ファイルの種類	コマンド
C ソースと GNU* Fortran ソースの混在	icx -qopenmp -c ibar.c gcc -fopenmp -c gbar.c gfortran -fopenmp -c foo.f gfortran foo.o ibar.o gbar.o -lirc -liomp5 -lpthread -lc -L<icx_dir>/lib

また、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーを使用してアプリケーションをリンクするこができますが、-l オプションを使用して、リンク行で複数の gfortran ライブラリーを渡す必要があります。

ファイルの種類	コマンド
C ソースと Fortran ソースの混在	gfortran -fopenmp -c foo.f icx -qopenmp -c ibar.c ifort -qopenmp foo.o ibar.o -lgfortranbegin -lgfortran

コマンドラインの例(macOS*)

macOS* 上でインテルの OpenMP* ライブラリーを使用して、1 つのコマンドでアプリケーション全体をコンパイルおよびリンク(ビルド)するには、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーの次のコマンドを指定します。

ファイルの種類	コマンド
Fortran ソース	ifort -qopenmp hello.f90

デフォルトでは、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーは OpenMP* ライブラリーのダイナミック・リンクを行います。スタティック・リンク (非推奨) を行うには、-qopenmp-link=static オプションを追加します。オプション -qopenmp-link(英語)は、Linux* および macOS* システムでリンカーが使用する OpenMP* ライブラリー(スタティックまたはダイナミック)を制御します(デフォルトは -qopenmp-link=dynamic)。

また、icx/icpx と gcc/g++ コンパイラーの両方を使用して、アプリケーションの一部をコンパイルし、オブジェクト・ファイルを作成することができます (オブジェクト・レベルの互換性)。

注:
古いバージョンの macOS* プラットフォームでは、GCC コンパイラーを併用してコンパイルできます。最新の macOS* 10.x プラットフォームには、GCC コンパイラーの代わりに Clang コンパイラーが含まれていますが、このコンパイラーは OpenMP* 実装をサポートしていません。Clang コンパイラーの将来のバージョンでは OpenMP* 実装がサポートされる可能性があります。

この例では、icx は C ファイルの foo.c をコンパイルし、icpx は C++ ファイルの ifoo.cpp をコンパイルします。Qopenmp(Windows*) または qopenmp(Linux* および macOS*)は OpenMP* サポートを有効にし

て、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーが、インテルの OpenMP* ライブラリーを使用してアプリケーションをリンクするようにします。

ファイルの種類	コマンド
C ソース	`icx -qopenmp -c foo.c`
C++ ソース	`icpx -qopenmp -c ifoo.cpp`
Fortran ソース	`ifort -qopenmp foo.o ifoo.o`

注:
macOS* 10.9 以降には GCC コンパイラーが含まれていません。ただし、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーと一緒に GCC コンパイラーをインストールできます。

オブジェクト・ファイルを混在させることもできますが、`gcc` の `-l` オプション、`-L` オプション、`-lpthread` オプションを指定しなくても済むように、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーでアプリケーションをリンクするほうが簡単です。

ファイルの種類	コマンド

C ソース	`icpx -qopenmp -c bar.c`
	`ifort -qopenmp -c foo.f90`
	`ifort -qopenmp foo.o bar.o`

インテル® Fortran コンパイラーを使用する際、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラー (`ifort`) でコンパイルされたメインプログラムが Fortran オブジェクト・ファイルにない場合は、リンク時に `-nofor-main` オプションを `ifort` コマンドラインに指定します。

また、インテル® Fortran コンパイラー・クラシックおよびベータ版インテル® Fortran コンパイラーを使用してアプリケーションをリンクするこができますが、`-l` オプションを使用して、リンク行で複数の `gfortran` ライブラリーを渡す必要があります。

ファイルの種類	コマンド

C ソースと Fortran ソースの混在	`gfortran -fopenmp -c foo.f`
	`icx -qopenmp -c ibar.c`
	`ifort -qopenmp foo.o ibar.o -lgfortranbegin -lgfortran`

関連情報

openmp、Qopenmp (英語)
IPO の使用 (英語)
OpenMP* サポート・ライブラリー
qopenmp-link、Qopenmp-link (英語)

スレッド・アフィニティー・インターフェイス

インテルのランタイム・ライブラリーには、OpenMP* スレッドを物理処理ユニットにバインドする機能があります。このインターフェイスは、KMP_AFFINITY 環境変数により制御されます。システム(マシン)のトポロジー、アプリケーション、オペレーティング・システムによってスレッド・アフィニティーはアプリケーションの速度に大きく影響します。

スレッド・アフィニティーは、特定のスレッド(仮想実行ユニット)をマルチプロセッサー・コンピューターの物理処理ユニットのサブセットに限定します。マシンのトポロジーにより、スレッド・アフィニティーはプログラムの実行速度に大きな影響を与えます。

スレッド・アフィニティーは、Windows* システムとスレッド・アフィニティー対応カーネルを持つ Linux* システムのバージョンでサポートされていますが、macOS* ではサポートされていません。

インテルの OpenMP* ランタイム・ライブラリーには、OpenMP* スレッドを物理処理ユニットにバインドする機能があります。このバインド機能は、3 種類のインターフェイスで使用できます。これらのインターフェイスは総称して、インテルの OpenMP* スレッド・アフィニティー・インターフェイスと呼ばれます。

- 高レベルのアフィニティー・インターフェイスでは、環境変数を使用してマシントポロジーを特定し、マシンの物理的位置に基づいて OpenMP* スレッドをプロセッサーに割り当てます。このインターフェイスは、KMP_AFFINITY 環境変数により全体が制御されます。
- 中レベルのアフィニティー・インターフェイスでは、環境変数を使用してどのプロセッサー(整数 ID)が OpenMP* スレッドにバインドされるかを明示的に指定します。このインターフェイスは、gcc の GOMP_AFFINITY 環境変数と互換性があり、KMP_AFFINITY 環境変数を使用して起動することもできます。GOMP_AFFINITY 環境変数は、Linux* システムでのみサポートされています。ただし、Windows* または Linux* のユーザーにも KMP_AFFINITY 環境変数によって同様の機能が提供されます。
- 低レベルのアフィニティー・インターフェイスでは、API を使用して OpenMP* スレッドに OpenMP* ランタイム・ライブラリーを呼び出しさせ、実行するプロセッサー・セットを明示的に指定することができます。このインターフェイスは、Linux* システムの sched_setaffinity および関連関数や Windows* システムの SetThreadAffinityMask および関連関数と本質的に似ています。また、KMP_AFFINITY 環境変数の特定のオプションを指定して、低レベル API インターフェイスの動作を指定できます。例えば、KMP_AFFINITY アフィニティー・タイプを無効にすると、低レベルのアフィニティー・インターフェイスが無効になります。また、KMP_AFFINITY や GOMP_AFFINITY 環境変数を使用して、初期アフィニティー・マスクを設定してから、低レベル API インターフェイスでマスクを取得することができます。

次の用語がこのセクションで使用されています。

- マシン上のプロセシング要素の合計数は、OS スレッド・コンテキストの数と呼ばれます。
- 各プロセシング要素は、オペレーティング・システム・プロセッサーまたは OS proc と呼ばれます。
- 各 OS プロセッサーには、OS proc ID と呼ばれる一意の整数の識別子が関連付けられます。
- パッケージとは、シングルまたはマルチコアのプロセッサー・チップを指します。
- OpenMP* グローバルスレッド ID(GTID)は、インテルの OpenMP* ランタイム・ライブラリーにより認識されるすべてのスレッドを一意に特定する整数です。ライブラリーを最初に初期化するスレッドは、GTID 0 です。その他すべてのスレッドがライブラリーによって作成される通常の場合で、入れ子構造の

並列性がないときは、n-threads-var – 1(範囲は 1 から ntheads-var - 1 まで)の新しいスレッドが GTID により作成されます。各スレッドの GTID は omp_get_thread_num() 関数によって返される OpenMP* スレッド番号と同じです。高レベルおよび中レベルのインターフェイスは、この概念に大きく基づいています。したがって、有用性は入れ子構造の並列性を含むプログラムに限定されます。低レベルのインターフェイスでは、GTID の概念を利用していないため、任意の並列化レベルを含むプログラムで使用できます。

一部の環境変数はインテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、インテル製マイクロプロセッサーにおいてより多くの最適化が行われる場合があります。

KMP_AFFINITY 環境変数

注:
KMP_AFFINITY 環境変数は、後述の「低レベルのアフィニティー API」で説明されているように、最初の並列領域の前、または omp_get_max_threads()、omp_get_num_procs()、アフィニティー API 呼び出しを含む特定の API 呼び出しの前に設定する必要があります。

KMP_AFFINITY 環境変数には、次の一般的な構文が使用されます。

構文
KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]

例えば、マシン・トポロジー・マップをリストするには、KMP_AFFINITY=verbose,none を指定し、modifier に verbose、type に none を使用します。

次の表は、サポートされる引数のリストです。

引数	デフォルト	説明
modifier	noverbose respect granularity=core	オプション。キーワードと指定子から構成されます。 • granularity=<specifier> 設定可能な指定子は fine、thread、core、および tile です。 • norespect • noverbose • nowarnings • proclist={<proc-list>} • respect • verbose • warnings <proc-list> の構文は、中レベルのアフィニティー・インターフェイスで説明されています。 **注:** 複数のプロセッサー・グループがある Windows* では、プロセス・アフィニティー・マスクが単一のプロセッサー・グループに等しい場合、norespect アフィニティー修飾子が仮定されます(これは Windows* 上でのデフォルトです)。そうでない場合は、respect アフィニティー修飾子が使用されます。

type	none	必須です。使用するスレッド・アフィニティーを示します。 • balanced • compact • disabled • explicit • none • scatter • logical (廃止予定。代わりに compact を指定します。ただし、permute value 値は省略します。) • physical (廃止予定。代わりに scatter を指定します。場合により、offset 値も指定します。) logical と physical は推奨されていないタイプですが、下位互換性のためにサポートされています。
permute	0	オプション。正の整数値です。type 値の explicit、none、disabled との使用は無効です。
offset	0	オプション。正の整数値です。type 値の explicit、none、disabled との使用は無効です。

アフィニティー・タイプ

"type" は、唯一必須の引数です。

type = none (デフォルト)

OpenMP* スレッドは特定のスレッド・コンテキストにバインドされません。ただし、オペレーティング・システムでアフィニティーがサポートされる場合は、コンパイラーは OpenMP* スレッド・アフィニティー・インターフェイスを使用してマシンのトポロジーを特定します。KMP_AFFINITY=verbose,none を指定して、マシンのトポロジーマップをリストします。

type = balanced

scatter と同様に、すべてのコアに少なくとも 1 つのスレッドが配置されるまで、別のコアにスレッドを配置します。ただし、ランタイムが同じコアで複数のハードウェア・スレッド・コンテキストを使用する際、balanced では OpenMP* スレッドの番号が互いに隣接していることが保証されるのに対して、scatter では保証されません。シングルソケットのシステムの場合のみ CPU 上でサポートされます。

注:
OpenMP* 環境変数 OMP_PROC_BIND=spread は、KMP_AFFINITY=balanced と似ており、マルチソケット CPU システムを含むすべてのプラットフォームで利用できます。

type = compact

compact を指定すると、フリー・スレッド・コンテキストの OpenMP* スレッド <n>+1 は、OpenMP* スレッド <n> が割り当てられたスレッド・コンテキストにできる限り近いスレッド・コンテキストに割り当てられます。例えば、トポロジーマップで、ルートにより近いノードほど、スレッドをソートしたときに上位になります。

type = disabled

disabled を指定すると、スレッド・アフィニティー・インターフェイスを完全に無効にします。これにより、OpenMP* ランタイム・ライブラリーはアフィニティー・インターフェイスがオペレーティング・システムでサポートされていないかのように動作します。これには、kmp_set_affinity や kmp_get_affinity などのような、効果がなく、非ゼロのエラーコードが返される低レベル API インターフェイスが含まれます。

type = explicit

explicit を指定すると、proclist= modifier (このアフィニティー・タイプには必須) を使用して明示的に指定された OS proc ID のリストに OpenMP* スレッドが割り当てられます。「OS プロセッサー ID (GOMP_CPU_AFFINITY)を明示的に指定する」を参照してください。

type = scatter

scatter を指定すると、システム全体にわたってスレッドが均等に分配されます。scatter は、compact の逆です。そのため、マシンのトポロジーマップをソートするとノードリーフは最上位になります。

推奨されていないタイプ: logical と physical

logical と physical は廃止予定で、将来のリリースではサポートされなくなります。両方とも、下位互換性のためにサポートされています。

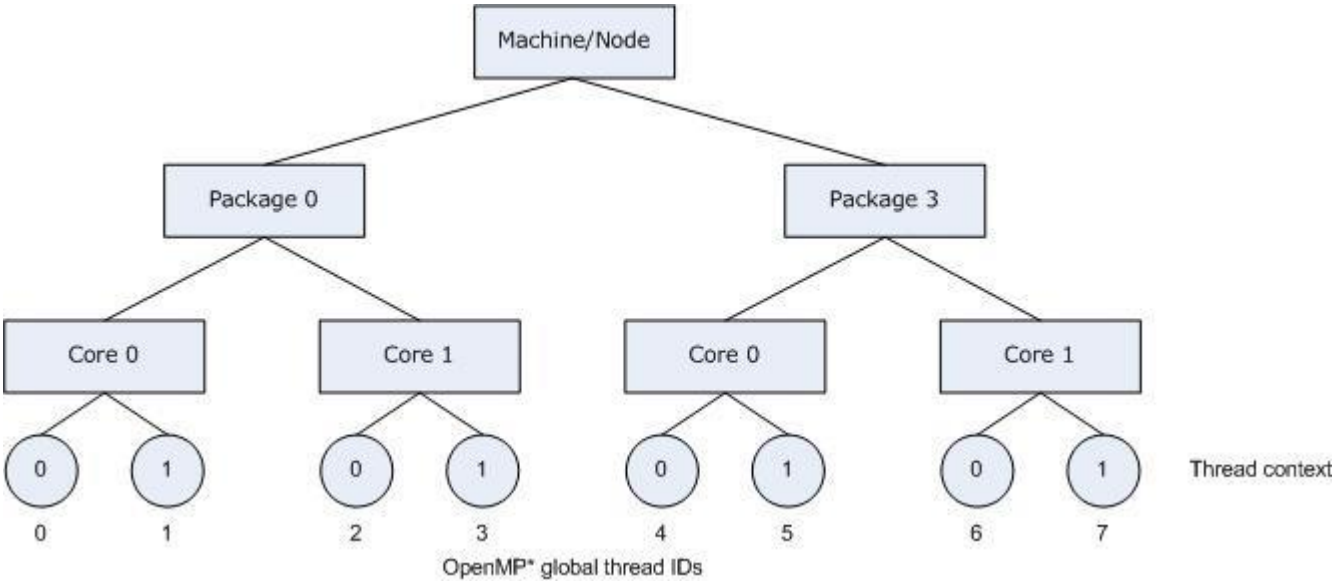
logical および physical では、整数値が 1 つしかない場合は、permute 指定子ではなく、offset 指定子と解釈されます。これに対し、compact と scatter では、整数値が 1 つのみの場合は permute 指定子と解釈されます。

- logical を指定すると、OpenMP* スレッドを連続した論理プロセッサー (ハードウェア・スレッド・コンテキストとも呼ぶ) に割り当てます。このタイプは、permute 指定子が使えないことを除いては、compact と同等です。そのため、KMP_AFFINITY=logical,n は KMP_AFFINITY=compact,0,n と (granularity=fine 修飾子の有無にかかわらず) 同等です。
- physical を指定すると、OpenMP* スレッドを連続した物理プロセッサー (コア) に割り当てます。コアごとに 1 つのスレッド・コンテキストしかないシステムの場合、logical と同じです。コアごとに複数のスレッド・コンテキストがあるシステムの場合、physical は permute 指定子が 1 に指定された compact と同じです。つまり、KMP_AFFINITY=physical,n は、KMP_AFFINITY=compact,1,n と (granularity=fine 修飾子の有無にかかわらず) 同等です。これは、コンパイラーがマップをソートしたときにマシンのトポロジーマップの最も内側のレベルを最も外側 (おそらくスレッド・コンテキスト・レベル) に入れ替えることを意味します。このタイプでは permute 指定子はサポートされません。

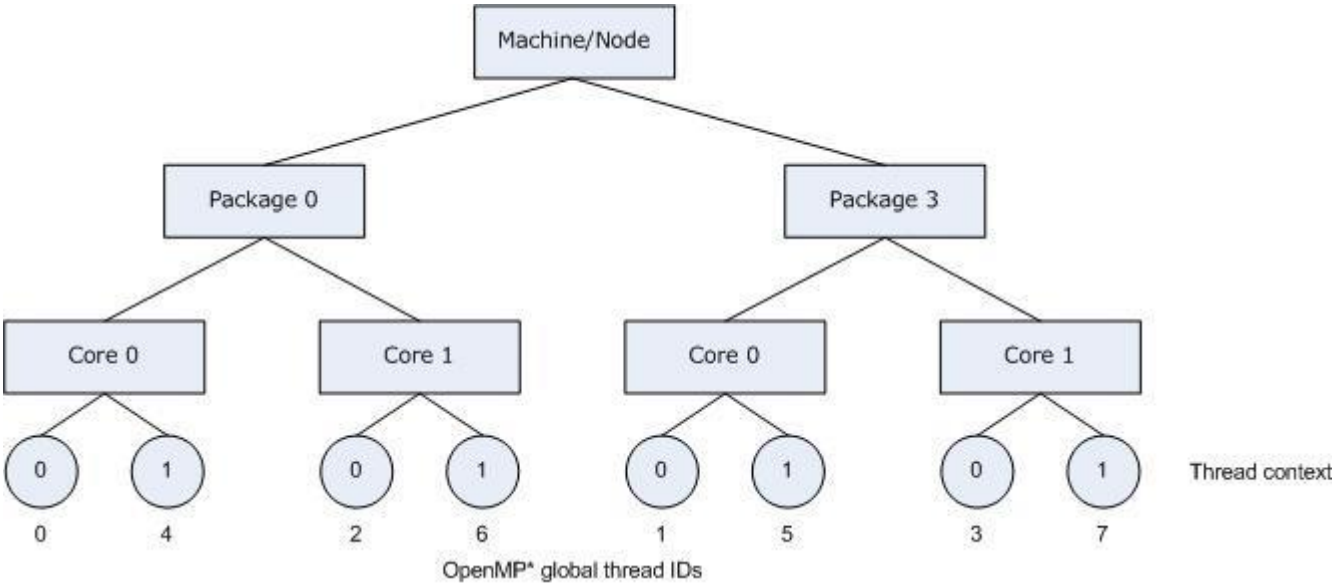
compact と scatter の例

次の図は、2 つのプロセッサーを搭載しているマシンを示しています。それぞれのプロセッサーには 2 つのコアがあります。各コアはインテル® ハイパースレッディング・テクノロジー (インテル® HT テクノロジー) 対応です。

また、次の図は、KMP_AFFINITY=granularity=fine,compact を指定したときに、OpenMP* スレッドがハードウェア・スレッド・コンテキストにバインドされる様子も示しています。



上図のシステムで `scatter` を指定すると、次の図のように、OpenMP* スレッドがスレッド・コンテキストに割り当てられます。これは、`KMP_AFFINITY=granularity=fine,scatter` を指定した結果です。



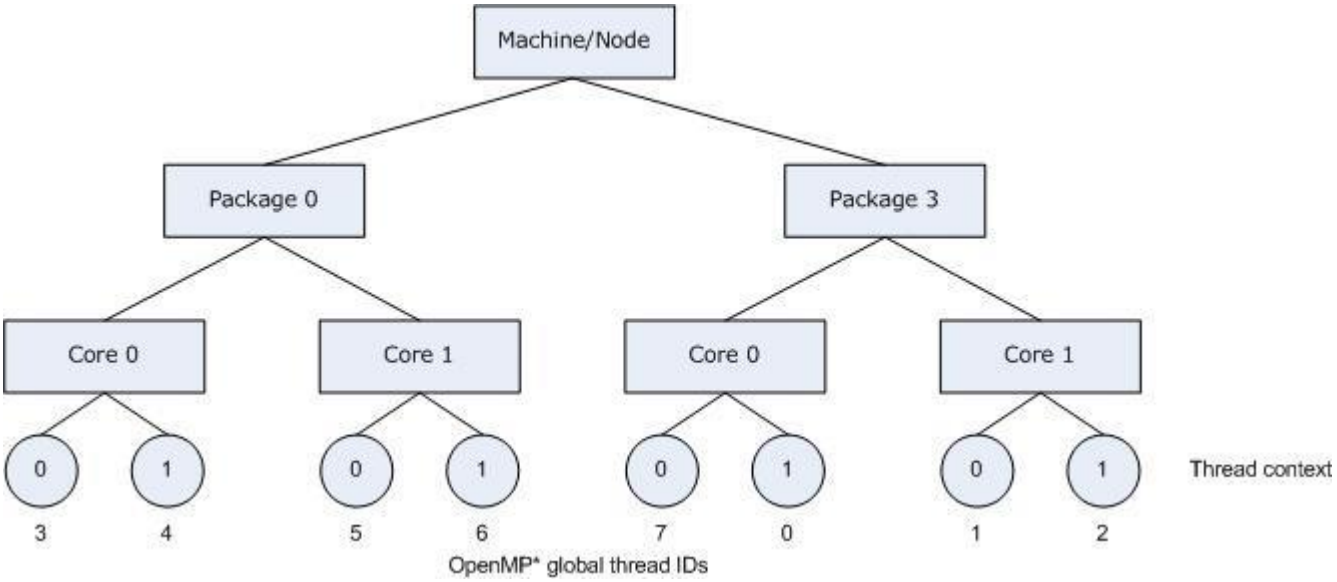
permute と offset の組み合わせ

`compact` と `scatter` の両方とも `permute` と `offset` が指定できます。ただし、1 つの整数だけを指定した場合は、コンパイラーは値を `permute` 指定子として解釈します。`permute` と `offset` の両方ともデフォルトは 0 です。

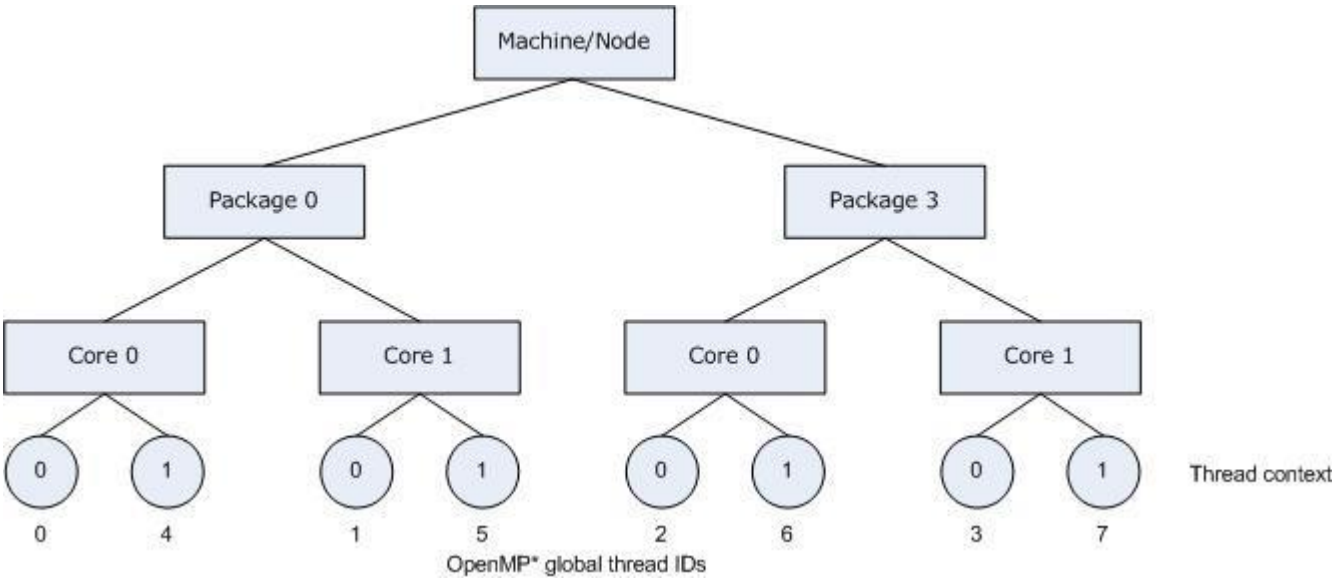
`permute` 指定子はマシンのトポロジーマップをソートしたときに最上位にするレベルを制御します。`permute` の値により、指定された番号の最上位レベルは最下位にマッピングされ、順位が入れ替わります。ツリーのルートノードは、ソート操作では個別のレベルとは見なされません。

`offset` 指定子は、スレッド割り当ての開始地点を示します。

次の図は、`KMP_AFFINITY=granularity=fine,compact,0,3` を指定した結果を示しています。



前述の例、連続したループ反復間でデータ共有を示す OpenMP* アプリケーションを実行するハードウェア構成について考えてみます。`KMP_AFFINITY=compact` を指定した際に行われるように、通信オーバーヘッド、キャッシュラインの無効オーバーヘッド、ページ・スラッシングが最小限になるように連続したスレッドを隣接してバインドさせます。ここで、利用可能な OpenMP* スレッドを活用していないアプリケーションに多くの並列領域があるとします。通常、スレッドは、同じコアにある別のアクティブなスレッドとの間にリソースの競合がない場合に実行速度が速くなるため、他のコアを使用せずに複数のスレッドを同じコアにバインドする状態は避けたほうが良いでしょう。通常、スレッドは、同じコアにある別のアクティブなスレッドとの間にリソースの競合がない場合に実行速度が速くなるため、他のコアを使用せずに複数のスレッドを同じコアにバインドする状態は避けたほうが良いでしょう。次の図は、`KMP_AFFINITY=granularity=fine,compact,1,0` を設定してこの方法を示したものです。



OpenMP* スレッド `n`+1 は、別のコアの OpenMP* スレッド `n` にできるだけ近いスレッド・コンテキストにバインドされます。いったん、それぞれのコアに 1 つの OpenMP* スレッドが割り当てられると、後続の OpenMP* スレッドは利用可能なコアに同じ順番で異なるスレッド・コンテキストに割り当てられます。

アフィニティー・タイプの修飾子の値

タイプの前に付ける修飾子はオプションです。修飾子を指定しない場合は、noverbose、respect、granularity=core が自動で使用されます。

修飾子は、左から右の順に解釈され、互いに無効にすることができます。例えば、KMP_AFFINITY=verbose,noverbose,scatter は、KMP_AFFINITY=noverbose,scatter または単に KMP_AFFINITY=scatter を指定するのと同じです。

modifier = noverbose(デフォルト)

詳細なメッセージは出力しません。

modifier = verbose

サポートされるアフィニティーに関するメッセージを出力します。メッセージには、パッケージ数、各パッケージにあるコア数、各コアにあるスレッド・コンテキスト数、物理スレッド・コンテキストにバインドされた OpenMP* スレッドについての情報が含まれます。

物理スレッド・コンテキストにバインドされた OpenMP* スレッドについての情報は、ハードウェア・スレッド・コンテキストとオペレーティング・システム(OS)プロセッサー(proc)ID 間のマッピングの形式で間接的に示されます。各 OpenMP* スレッドのアフィニティー・マスクは、OS プロセッサー ID セットとして出力されます。

例えば、インテル® ハイパースレッディング・テクノロジーを無効にした 2 つのプロセッサーを搭載したデュアルコア・システムで KMP_AFFINITY=verbose,scatter を指定すると、プログラムの実行時に次のようなメッセージが出力されます。

verbose, scatter を指定した場合のメッセージ

```
...
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {1}
```

verbose 修飾子を指定すると、いくつかの標準的、一般的なメッセージが生成されます。次の表は、メッセージの説明です。

メッセージ	説明
"affinity capable (アフィニティー可)"	すべてのコンポーネント(コンパイラー、オペレーティング・システム、ハードウェア)でスレッドのバインドが可能なようにアフィニティーがサポートされていることを示します。
"using global cpuid info (グローバル cpuid 情報使用)"	スレッドを各オペレーティング・システムのプロセッサーにバインドし、cpuid 命令の出力をデコードすることにより、マシントポロジーが特定されたことを示します。
"using local cpuid info (ローカル cpuid 情報使用)"	コンパイラーは初期スレッドのみから発行される cpuid 命令の出力をデコードし、マシントポロジーでオペレーティング・システムのプロセッサー数が使用されていることを想定しています。
"using /proc/cpuinfo (/proc/cpuinfo 使用)"	Linux* のみ。cpuinfo がマシントポロジーを特定するのに使用されることを示します。
"using flat(flat 使用)"	オペレーティング・システムのプロセッサー ID は、物理パッケージ ID と同じと見なされています。マシントポロジーを特定するこの方法は、その他の方法がどれも使用できず、また実際のマシントポロジーを正しく検出しない可能性がある場合に使用されます。
"uniform topology of (一様トポロジー)"	マシントポロジーのマップは、どの階層においてもリーフがそろった完全なツリーです。

次に、オペレーティング・システムのプロセッサーからスレッド・コンテキスト ID へのマッピングが出力されます。OpenMP* スレッドのスレッド・コンテキスト ID へのバインドは、アフィニティー・タイプが none でない限り、次に出力されます。スレッドレベルは、角括弧の中に示されます(上のリストを参照)。これは、マシントポロジーのマップにはそのスレッド・コンテキスト・レベルが示されないことを意味します。詳細は、「マシントポロジーの特定」を参照してください。

modifier = granularity

OpenMP* スレッドを特定のパッケージやコアにバインドすると、インテル® ハイパースレッディング・テクノロジー(インテル® HT テクノロジー)対応のインテル® プロセッサーを搭載したシステムでパフォーマンス・ゲインを期待できます。しかし、一般的に各 OpenMP* スレッドを特定のコアにある特定のスレッド・コンテキストにバインドしても効果は期待できません。粒度は、トポロジーマップ内で OpenMP* スレッドのフロートが許可される最下位レベルを示します。

この修飾子は、次の指定子をサポートします。

指定子	説明
core	デフォルト。異なるスレッド・コンテキスト間でフロートするため、すべての OpenMP* スレッドをコアにバインドできます。
fine または thread	最も細かい粒度レベルです。各 OpenMP* スレッドが 1 つのスレッド・コンテキストにバインドされます。この 2 つの指定子は、機能的に同じです。
tile	タイルを構成するコアの異なるスレッド・コンテキスト間でフロートするため、すべての OpenMP* スレッドをタイルにバインドできます。

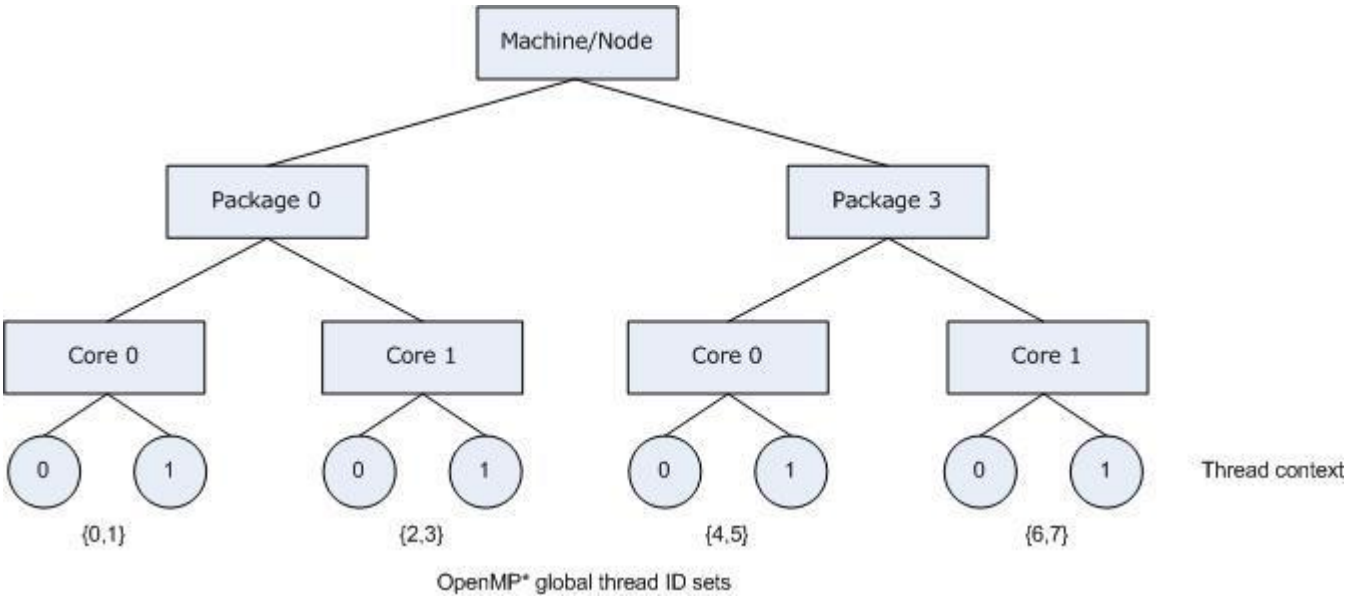
2 つのプロセッサーを搭載したデュアルコア・システムで、インテル® ハイパースレッディング・テクノロジー (インテル® HT テクノロジー)が有効な場合に、`KMP_AFFINITY=verbose,granularity=core,compact` を指定すると、プログラムの実行時に次のようなメッセージが出力されます。

verbose, granularity=core,compact を指定した場合のメッセージ

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP_AFFINITY: 8 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,4}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {0,4}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2,6}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {2,6}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {1,5}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {1,5}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {3,7}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {3,7}
```

各 OpenMP* スレッドのアフィニティー・マスクは、OpenMP* スレッドがバインドされたオペレーティング・システムのプロセッサー・セットとしてリストされます(上を参照)。

次の図は、OpenMP* スレッドがバインドされた上記のリストのマシントポロジーを示しています。



これに対し、KMP_AFFINITY=verbose,granularity=fine,compact または KMP_AFFINITY=verbose,granularity=thread,compact を指定すると、プログラムの実行時に各 OpenMP* スレッドが 1 つのハードウェア・スレッド・コンテキストにバインドされます。

verbose, granularity=fine,compact を指定した場合のメッセージ

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP_AFFINITY: 8 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {1}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

OpenMP* をハードウェア・コンテキストにバインドする例は、最初の例に示されています。

granularity=fine を指定すると、各 OpenMP* スレッドが 1 つの OS プロセッサーにバインドされます。これは、granularity=thread と等価で、現在、最も微細な粒度レベルです。

modifier = respect (デフォルト)

プロセスの元のアフィニティー・マスク、厳密には、OpenMP* ランタイム・ライブラリーを初期化するスレッドに指定されたアフィニティー・マスクが順守されます。動作は、Linux* と Windows* で異なります。

- Windows*:プロセスの元のアフィニティー・マスクが順守されます。
- Linux*:OpenMP* ランタイム・ライブラリーを初期化するスレッドのアフィニティー・マスクが順守されます。

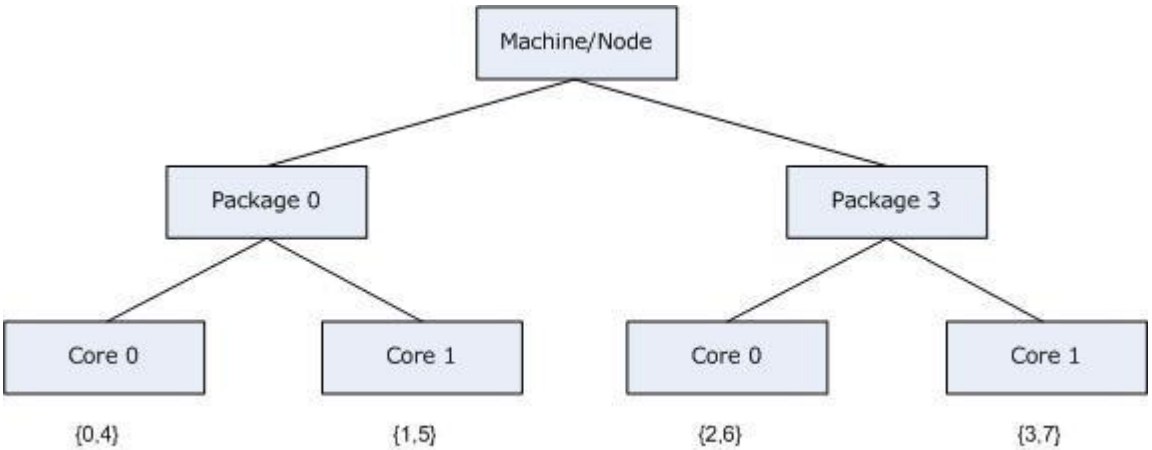
前の例と同じ、インテル® ハイパースレッディング・テクノロジー(インテル® HT テクノロジー)が有効なシステムで、KMP_AFFINITY=verbose,compact を指定して、最初のアフィニティー・マスク {4,5,6,7}(各コアでスレッド・コンテキストは 1)を起動すると、コンパイラーでは、インテル® ハイパースレッディング・テクノロジー(インテル® HT テクノロジー)が無効の 2 つのプロセッサーを搭載したデュアルコアのマシンであると判断されます。

verbose,compact を指定した場合のメッセージ

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{4,5,6,7}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 [thread 1]
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 [thread 1]
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 [thread 1]
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 [thread 1]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {7}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

マシンには、8 個のスレッド・コンテキストがあるため、デフォルトではコンパイラーにより OpenMP* `parallel` 構造で 8 個のスレッドが作成されています。

"スレッド 1" の角括弧はスレッド・コンテキストのレベルが無視され、トポロジーマップには示されていないことを意味します。次の図は、対応するマシンのトポロジーマップです。



OpenMP* global thread ID sets

ローカルの `cpuid` 情報を使用してマシントポロジーを特定する場合、インテル® ハイパースレッディング・テクノロジー(インテル® HT テクノロジー)をサポートしていないマシンとサポートしているけれども無効になっているマシンとの区別がつきません。そのため、そのレベルの要素(ノード)に兄弟がいない場合は、コンパイラーはそのレベルをマップに含めません(ただし、パッケージレベルは常に含まれます)。前述したように、パッケージレベルは、マシンに 1 つのパッケージしかない場合でも、常にトポロジーマップに表示されます。

modifier = norespect

プロセスのアフィニティー・マスクは順守されません。OpenMP* スレッドはすべてのオペレーティング・システムのプロセッサーにバインドされます。

physical と logical のみがアフィニティー・タイプとしてサポートされていた以前のバージョンの OpenMP* ランタイム・ライブラリーでは、norespect がデフォルトで、修飾子としては認識されませんでした。

compact と scatter がアフィニティー・タイプに追加されてから、デフォルトは respect に変更されました。そのため、アプリケーションが不完全な初期スレッド・アフィニティー・マスクを指定した状態では、logical と physical のスレッドのバインドは、新しいコンパイラー・バージョンで変更されることがあります。

modifier = nowarnings

アフィニティー・インターフェイスからの警告メッセージを出力しません。

modifier = warnings (デフォルト)

アフィニティー・インターフェイスからの警告メッセージを出力します(デフォルト)。

マシントポロジーの特定

IA-32 およびインテル® 64 アーキテクチャー・システムでは、パッケージに APIC(Advanced Programmable Interrupt Controller)がある場合、コンパイラーは cpuid 命令を使用して package id、core id、および thread context id を入手します。通常の状態では、システム上の各スレッド・コンテキストには起動時に固有の APIC ID が割り当てられます。コンパイラーは、OS スレッド・コンテキスト数(マシン上のプロセシング要素の合計数)とともに cpuid 命令によって取得されるその他の情報を取得し、APIC ID から package ID、core ID、および thread context ID を識別する方法を特定します。

cpuid 命令で APIC ID を指定する方法は 2 つあります:リーフ 4 で指定する以前の方法とリーフ 11 で指定するより新しい方法。リーフ 4 で利用可能な一意の APIC ID は 256 のみです。リーフ 11 ではそのような制限はありません。

通常、パッケージ上のすべての core id とコア上のすべての thread context id は連続しています。ただし、上の図で示されるように、package id の番号付けでは連続していないことも一般的です。

コンパイラーがいずれの方法を使用してもトポロジーを特定できないが、アフィニティーがオペレーティング・システムでサポートされている場合は、警告メッセージが出力され、トポロジーは flat と想定されます。例えば、ある flat トポロジーでは、オペレーティング・システムがプロセス N をパッケージ N にマップし、1 コアにつき 1 スレッドのみ、そして各パッケージには 1 コアがあると想定されます。

マシントポロジーが上記のように特定できない場合は、手動で /proc/cpuinfo を一時ファイルにコピーして、エラーを修正し、「KMP_CPUINFO_FILE と /proc/cpuinfo」のセクションで説明されているように、KMP_CPUINFO_FILE=<temp_filename> 環境変数を介して、マシントポロジーを OpenMP* ランタイム・ライブラリーに指定します。

マシントポロジーの特定に使用する方法にかかわらず、マシン上の各コアで 1 コアにつき 1 スレッドしかない場合は、スレッド・コンテキスト・レベルはトポロジーマップには表示されません。マシン上の各パッケージで 1 パッケージにつき 1 コアしかない場合は、コアレベルはトポロジーマップには表示されません。各パッケージには異なる数のコアが含まれている可能性があり、各コアは異なる数のスレッド・コンテキストをサポートしている可能性があるため、トポロジーマップは完全なツリー状とは限りません。

パッケージレベルは、マシンに 1 つのパッケージしかない場合でも、常にトポロジーマップに表示されます。

KMP_CPUINFO_FILE と /proc/cpuinfo

OpenMP* ランタイム・ライブラリーが Linux* システム上のマシントポロジーを検出する方法の 1 つに、/proc/cpuinfo の内容を解析する方法があります。このファイル(または Linux* ファイルシステムにマップされているデバイス)の内容が不十分であったり、エラーを含む場合は、内容を書き込み可能な一時ファイル <temp_file> にコピーし、修正するか必要な情報を追加して、さらに KMP_CPUINFO_FILE=<temp_file> を設定します。

これを行うことにより、OpenMP* ランタイム・ライブラリーは、/proc/cpuinfo に含まれる情報を読み取る代わりに、また APIC ID をデコードしてマシントポロジーを検出する代わりに、KMP_CPUINFO_FILE が示す <temp_file> を読み取ります。つまり、<temp_file> に含まれる情報が、その他の方法よりも優先されます。/proc/cpuinfo を持たない Windows* システムでは、KMP_CPUINFO_FILE インターフェイスを使用することができます。

/proc/cpuinfo または <temp_file> にはマシン上の各プロセシング要素のエントリーのリストが含まれています。各プロセッサー要素にはエントリー(分かりやすい名前と各行の値)のリストが含まれています。空白行は各プロセッサー要素を区切ります。次のフィールドのみが、<temp_file> または /proc/cpuinfo にある各エントリーからマシントポロジーを特定するのに使用されます。

フィールド	説明
processor:	プロセシング要素の OS ID を指定します。OS ID は一意でなければなりません。processor フィールドと physical id フィールドは、このインターフェイスの使用に唯一必須のものです。
physical id:	物理チップ ID であるパッケージ ID を指定します。各パッケージには複数のコアが含まれています。パッケージレベルは常にインテル® コンパイラーの OpenMP* ランタイム・ライブラリーのマシン・トポロジー・モデルにあります。
core id:	コア ID を指定します。コア ID がない場合は、デフォルトの 0 に設定されます。マシン上の各パッケージがシングルコアしかない場合は、(コア ID フィールドのいくつかが非ゼロの場合でも)コアレベルはマシンのトポロジーマップにはありません。
thread id:	スレッド ID を指定します。コア ID がない場合は、デフォルトの 0 に設定されます。マシン上の各コアでシングルスレッドしかない場合は、(スレッド ID フィールドのいくつかが非ゼロの場合でも)スレッドレベルはマシンのトポロジーマップにはありません。
node_n id:	これは、/proc/cpuinfo の拡張です。不均等メモリーアクセス(NUMA: Non-Uniform Memory Access)システムにおける異なるレベルのメモリー相互接続でノードを指定するのに使用できます。任意のレベル n がサポートされています。node_0 レベルがパッケージレベルに最も近いレベルで、複数のパッケージがレベル 0 で 1 つのノードを構成します。レベル 0 の複数のノードは、レベル 1 で 1 つのノードを構成します。

各エントリーは、示されているとおりに正確に小文字で、オプションのスペース、コロン (:)、さらにオプションのスペース、そして整数 ID を記述します。これ以外のフィールドは無視されます。

注:
多くの Linux* バリアントで thread id フィールドが /proc/cpuinfo に含まれておらず、siblings とラベル付けされたフィールドがノードごとのスレッド数またはパッケージごとのノード数を指定することはよくあ

ることです。ただし、インテルの OpenMP* ランタイム・ライブラリーは、siblings とラベルされたフィールドを無視するため、thread id フィールドと siblings フィールドを区別できます。このような状況が発生した場合は、警告メッセージの「物理ノード/パッケージ/コア/スレッド ID が一意ではありません。」が表示されます(指定されたタイプが nowarnings の場合を除く)。

Windows* プロセッサー・グループ

64 ビットの Windows* オペレーティング・システムでは、複数のプロセッサー・グループが 64 を超えるプロセッサーに対応できます。各グループのサイズは、64 プロセッサーの最大値までに制限されます。

複数のプロセッサー・グループが検出されると、デフォルトでは 2 レベルのツリー (レベル 0 はグループ内のプロセッサー、レベル 1 は異なるグループ) であると判断されます。そして、グループにバインドされた OpenMP* スレッド数がグループ内のプロセッサー数と等しくなるまで、スレッドがグループに割り当てられ、後続のスレッドは次のグループに割り当てられます。

デフォルトでは、スレッドはグループ内のすべてのプロセッサー間を移動することができ、粒度はグループの数と同じになります [granularity=group]。このバインドを無効にし、compact、scatter など、別のアフィニティー・タイプを明示的に使用することができます。その場合は、スレッドが異なるグループの複数のプロセッサーにバインドされないように、十分に細かい粒度を使用する必要があります。

特定のマシン・トポロジー・モデル・メソッドの使用(KMP_TOPOLOGY_METHOD)

KMP_TOPOLOGY_METHOD 環境変数を設定して、特定のマシン・トポロジー・モデル・メソッドを OpenMP* で使用するように強制できます。

値	説明
cpuid_leaf11	cpuid 命令のリーフ 11 により指定された APIC ID をデコードします。
cpuid_leaf4	cpuid 命令のリーフ 4 により指定された APIC ID をデコードします。
cpuinfo	KMP_CPUINFO_FILE を指定しない場合は、/proc/cpuinfo を解析してトポロジーを特定するように強制できます(Linux* のみ)。 前述のように、KMP_CPUINFO_FILE を指定する場合は、それを使用します(Windows* または Linux*)。
group	2 レベルのマップ(レベル 0 はグループ内の異なるプロセッサーを指定、レベル 1 は異なるグループを指定)として判断します(64 ビットの Windows* のみ)。
flat	プロセッサーの flat(線形)リストとして判断します。
hwloc	Portable Hardware Locality(hwloc) ライブラリーと同様に判断します。 最も詳細なモデルで、NUMA ノード、パッケージ、コア、ハードウェア・スレッド、キャッシュ、Windows* プロセッサー・グループなどを含みます。

OS プロセッサー ID(GOMP_CPU_AFFINITY)を明示的に指定する

注:
GOMP_CPU_AFFINITY 環境変数は、後述の「低レベルのアフィニティー API」で説明されているように、最初の並列領域の前、または omp_get_max_threads()、omp_get_num_procs()、アフィニティー API 呼び出しを含む特定の API 呼び出しの前に設定する必要があります。

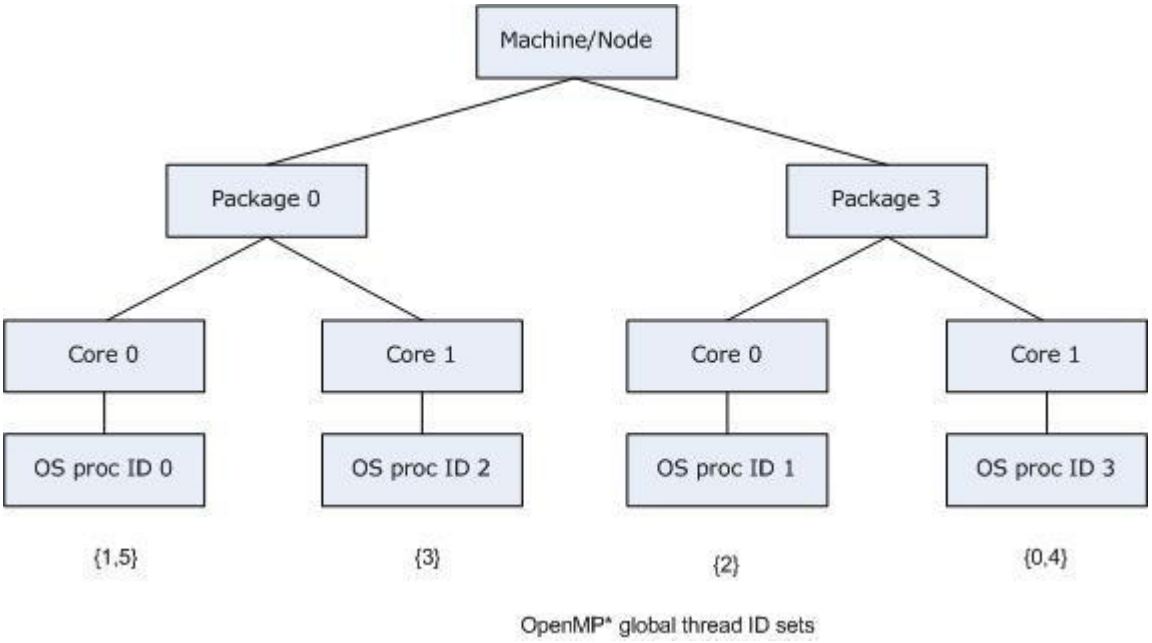
ライブラリーにハードウェア・トポロジーを検出させて自動的に OpenMP* スレッドをプロセシング要素に割り当てる代わりに、オペレーティング・システム(OS)プロセッサー(proc) ID のリストを使用して、明示的に割り当てを指定することができます。ただし、これには OS proc ID がどのプロセシング要素を表現しているかについての知識が必要です。

-qopenmp-lib compat コンパイラー・オプションで有効にされたインテルの OpenMP* 互換ライブラリーを使用する際、Linux* システムでは GOMP_AFFINITY 環境変数を使用して OS プロセッサー ID を指定します。その構文は、libgomp のものと同じです(<proc_list> が GOMP_AFFINITY 環境文字列全体を生成すると想定)。

値	説明
<proc_list> :=	<entry> | <elem> , <list> | <elem> <whitespace> <list>
<elem> :=	<proc_spec> | <range>
<proc_spec> :=	<proc_id>
<range> :=	<proc_id> - <proc_id> | <proc_id> - <proc_id> : <int>
<proc_id> :=	<positive_int>

このリストに指定されている OS プロセッサーは、その後 OpenMP* グローバルスレッド ID の順に OpenMP* スレッドに割り当てられます。リストにある要素よりも多くの OpenMP* スレッドが作成された場合は、リストサイズを基にする割り当てが行われます。つまり、OpenMP* グローバルスレッド ID n は、リスト要素 n mod <list_size> にバインドされます。

前の図の例と同じように OS proc ID が割り当てられた前述のマシン、インテル® ハイパースレッディング・テクノロジー(インテル® HT テクノロジー)が有効になっていないデュアルコア、デュアルパッケージ・マシンについて考えてみましょう。アプリケーションが 4(デフォルト)ではなく、マシンを超過してしまう 6 つの OpenMP* スレッドを生成するとします。GOMP_AFFINITY=3,0-2, の場合、次の図に示されているように、gcc でコンパイルして、libgomp でリンクしたときと同じように OpenMP* スレッドがバインドされます。



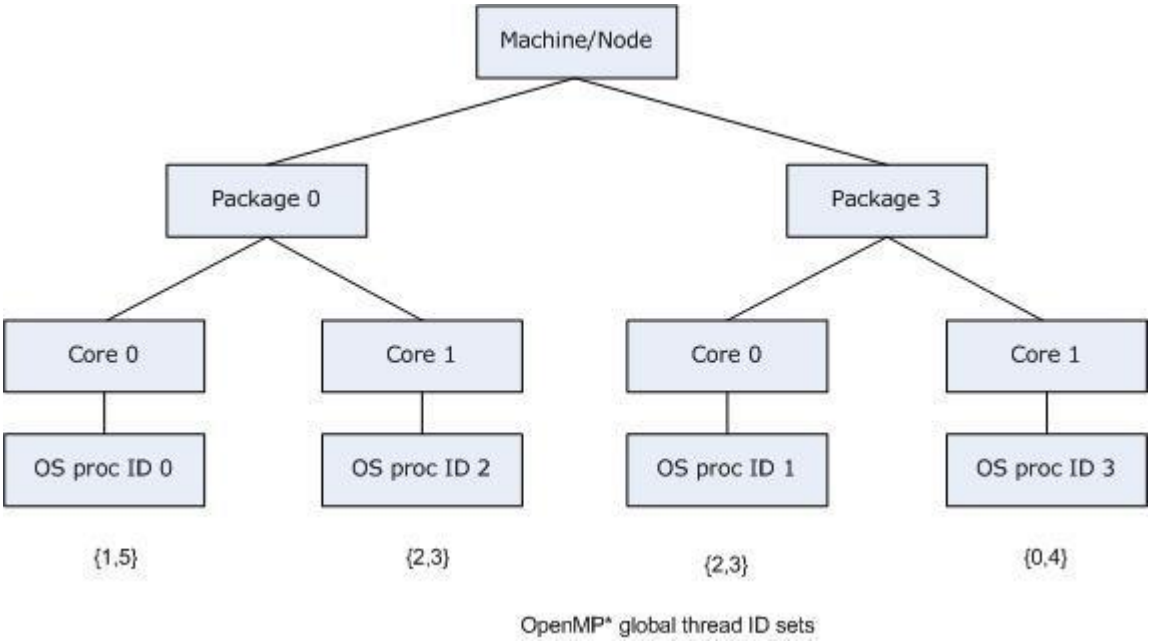
OpenMP* global thread ID sets

同じ構文が、OS proc ID リストを KMP_AFFINITY 環境変数文字列の proclist=[<proc_list>] 修飾子に指定するのに使用できます。若干の差異として、gcc の OpenMP* ランタイム・ライブラリー libgomp と全く同じセマンティクスを持つために、GOMP_AFFINITY 環境変数は granularity=fine を示唆します。KMP_AFFINITY 環境変数で granularity= 指定子を使用せずに OS proc リストを指定すると、デフォルトの granularity は変更されません。つまり、OpenMP* スレッドはシングルコアの異なるスレッド・コンテキスト間でフロートすることができます。このように、GOMP_AFFINITY=<proc_list> は、KMP_AFFINITY="granularity=fine,proclist=[<proc_list>],explicit" のエイリアスです。

KMP_AFFINITY 環境変数文字列では、構文はオペレーティング・システムのプロセッサー ID セットを処理するために拡張されます。ユーザーは OpenMP* スレッドが実行("フロート")できるオペレーティング・システムのプロセッサー ID セットを角括弧で指定できます。

値	説明
<proc_list> :=	<proc_id> | { <float_list> }
<float_list> :=	<proc_id> | <proc_id> , <float_list>

これにより、機能的には granularity= specifier と同様で、より柔軟性を発揮します。OpenMP* スレッドが実行される OS プロセッサーには、マシントポロジーで隣接している OS プロセッサーは除外されますが、その他の離れた OS プロセッサーは含まれます。次の図に示すように KMP_AFFINITY="granularity=fine, proclist=[3,0,{1,2},{1,2}],explicit" を使用して、OS プロセッサー 1 と OS プロセッサー 2 の間で OpenMP* スレッド 2 と 3 を "フロート" させることができます。



OpenMP* global thread ID sets

`verbose` も指定された場合、アプリケーションが実行された出力結果には以下が含まれます。

```
KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"

KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {1,2}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {0}}
```

低レベルのアフィニティー API

ユーザーがプログラムの実行開始前に環境変数を設定して(または並列領域に到達する前に `kmp_settings` インターフェイスを使用して)OpenMP* スレッドを OS proc にバインドさせなくても、それぞれの OpenMP* スレッドでは、実行し、`kmp_set_affinity` API 呼び出しでバインドさせる適切な OS proc セットを決定できます。

注:
このアフィニティー・インターフェイスを使用することで、スレッドを実行するハードウェア・リソースを完全に制御することができます。そのためには、論理 CPU(OS によって制御されるハードウェア・スレッドの列挙子)がどのように実行マシンの物理ハードウェアへマップされるかを詳しく理解する必要があります。マップ方法はマシンによって異なるため、マシン固有の情報をコードに記述すると、別のマシンで実行するときに不適切なアフィニティーを強制することになります。この詳細レベルで最適化を検討している場合、コードを別のマシンに移動する可能性が高いでしょう。

このインターフェイスを使用することで、メッセージ・パッシング・インターフェイス(MPI)などのプログラムの起動メカニズムによって設定されるリソースの制約も無視できます。特に、同じノード上の複数の OpenMP* プロセスが同じハードウェア・スレッドを使用しないようにできます。これも、パフォーマンスの低下を引き起こすアフィニティーを強制する可能性があります。OpenMP* ランタイムはこの問題の発生を防ぐことも、問題が発生した際に警告を出力することもしません。このインターフェイスはエキスパート向けです。注意して使用してください。

移植性に優れ、低レベルの知識を必要としない、できるだけ高いレベルのアフィニティー設定を利用することを推奨します。

タイプ名 `kmp_affinity_mask_t` が omp_lib.h または omp_lib.mod で定義される Fortran API インターフェイス:

注:
これらのインターフェイスの一部には、同等のオフロード・インターフェイスがあります。同等のオフロード・インターフェイスには、ターゲットの種類と番号を指定する 2 つの追加の引数があります。詳細は、「CPU の関数を呼び出してコプロセッサーの実行環境を変更する」を参照してください。オフロードは、Windows* では利用できません。

構文	説明
`integer function kmp_set_affinity(mask) integer (kind=kmp_affinity_mask_kind) mask`	現在の OpenMP* スレッドのアフィニティー・マスクを `mask` に設定します。ここで、`mask` は以下にリストされた API 呼び出しで作成された OS proc ID のセットで、スレッドは、そのセットの OS proc でのみ実行します。成功した場合はゼロ(0)、エラーコードの場合は非ゼロを返します。
`integer kmp_get_affinity(mask) integer (kind=kmp_affinity_mask_kind) mask`	現在の OpenMP* スレッドのアフィニティー・マスクを取得し、`kmp_create_affinity_mask()` への呼び出しで前もって初期化されているはずの `mask` に格納します。成功した場合はゼロ(0)、エラーコードの場合は非ゼロを返します。
`integer function kmp_get_affinity_max_proc()`	マシン上の最大 OS proc ID に 1 を足した値が返されます。すべての OS proc ID が 0(包括的)と `kmp_get_affinity_max_proc()` (排他的)の間で保証されます。
`subroutine kmp_create_affinity_mask(mask) integer (kind=kmp_affinity_mask_kind) mask`	新しい OpenMP* スレッド・アフィニティー・マスクを割り当て、`mask` を OS proc の空のセットに初期化します。セット自身として、実際のセットへのポインターとして、またはセットを示すテーブルへのインデックスとしてなど、`kmp_affinity_mask_kind` のオブジェクトの使用は自由です。実際の表現が何であるかといった仮定をしてはなりません。
`subroutine kmp_destroy_affinity_mask(mask) integer (kind=kmp_affinity_mask_kind) mask`	OpenMP* スレッド・アフィニティー・マスクの割り当てを解除する `kmp_create_affinity_mask()` への各呼び出しで、対応する `kmp_destroy_affinity_mask()` への呼び出しがなければなりません。

`integer function kmp_set_affinity_mask_proc(proc, mask)`
`integer proc`
`integer (kind=kmp_affinity_mask_kind) mask`
	OS proc ID proc をセット `mask` に追加します(追加されていない場合)。成功した場合はゼロ(0)、エラーコードの場合は非ゼロを返します。

`integer function kmp_unset_affinity_mask_proc(proc, mask)`
`integer proc`
`integer (kind=kmp_affinity_mask_kind) mask`
	OS proc ID proc がセット `mask` にある場合、それを削除します。成功した場合はゼロ(0)、エラーコードの場合は非ゼロを返します。

`integer function kmp_get_affinity_mask_proc(proc, mask)integer proc` `integer (kind=kmp_affinity_mask_kind) mask`	OS proc ID proc がセット `mask` にある場合は 1 を返します。ない場合は、0 を返します。

OpenMP* スレッドが `kmp_set_affinity()` への呼び出しに成功して自身のアフィニティー・マスクを設定したら、後続の `kmp_set_affinity()` への呼び出しでリセットされない限り、少なくとも並列領域が終わるまでは、そのスレッドは対応する OS proc セットにバインドされたままです。

並列領域間では、アフィニティー・マスク(および OS proc バインド機能に対応する OpenMP* スレッド)はスレッド・プライベートのデータ・オブジェクトと見なされ、「OpenMP* アプリケーション・プログラム・インターフェイス」で説明されているように、同様の永続性を持ちます。詳細は、OpenMP* API 仕様(http://www.openmp.org)を参照してください。該当部分の一部を以下に抜粋(参考訳)します。

アフィニティー・マスクとスレッドバインド機能が 2 つの連続したアクティブな並列領域間で永続するには、次の 3 つの条件がすべて満たされなければなりません。

- 別の明示的な並列領域の内側に入れ子構造の並列領域がないこと。
- 両方の並列領域の実行に使用されているスレッド数が同じであること。
- 囲まれたタスク領域にある `dyn-var` 内部制御変数の値が、両方の並列領域の入口で `false` であること。

そのため、プログラムの実行が上記の OpenMP* 仕様の 3 つの規則に従っており、唯一の目的が各スレッドのアフィニティー・マスクを設定することであるプログラムの開始時点で並列領域を作成すると、低レベルのアフィニティー API 呼び出しにより、ユーザーは `KMP_AFFINITY` 環境変数の動作と同じことができます。

次に、低レベルのインターフェイスの使用例を示します。このコードは、実行スレッドを特定の論理 CPU にバインドします。

例

```
! Force the executing thread to execute on logical CPU i
! Returns .TRUE. on success, .FALSE. on failure

function forceAffinity (i)
    use omp_lib
    logical forceAffinity
    integer, intent(in) :: i

    integer(kmp_affinity_mask_kind) :: mask

    call kmp_create_affinity_mask(mask)
    forceAffinity = (kmp_set_affinity_mask_proc(i, mask) == 0)
    if (.not. forceAffinity) return
    forceAffinity = (kmp_set_affinity_mask(mask) == 0)
    return
    end function forceAffinity
```

このプログラムは、OS proc ID のターゲットマシンの物理プロセシング要素へのマッピングに関する知識に基づいて記述されたものです。別のマシンや異なる OS がインストールされている場合、プログラムは実行されますが、OpenMP* スレッドの物理プロセシング要素へのバインド動作は異なります。また、明示的に効率の悪い分配を強制してしまう可能性があります。

OpenMP* メモリー空間とアロケーター

この機能は `ifx` でのみ利用できます。

変数の保存と取得のために OpenMP* はメモリー空間と呼ばれるメモリーを提供します。異なるメモリー空間は異なる特性を持っています。変数の使用方法とアクセス方法に応じて、適切なメモリー空間が決定されます。各メモリー空間には固有のアロケーターがあり、その空間内のメモリーを割り当てたり、解放するために使用されます。アロケーターは、メモリー空間内の他の割り当てと重複しない連続した空間に変数を割り当てます。異なる特性を持つ複数のメモリー空間を単一のメモリーリソースに割り当てることも可能です。

アロケーターの動作は、ユーザーが指定したアロケーターの特性によって影響を受けます。アロケーターの特性、指定可能な値、およびデフォルト値を次の表に示します。

アロケーターの特性	指定可能な値	デフォルト値
access	• all • cgroup • pteam • thread	all
alignment	バイト数を指定する 2 の累乗の正の整数値	1 バイト
fallback	• default_mem_fb • null_fb • abort_fb • allocator_fb	default_mem_fb
fb_data	アロケーター・ハンドル	なし
partition	• environment • nearest • blocked • interleaved	environment
pinned	• true • false	false
pool_size	正の整数値	実装定義
sync_hint	• contended • uncontended • serialized • private	contended

access 特性は、割り当てられるメモリーのアクセシビリティーを指定します。access 特性値 all は、割り当てられるメモリーが、メモリー割り当てが行われるデバイス内のすべてのスレッドからアクセス可能でなければならないことを示します。access 特性値 cgroup は、割り当てられるメモリーが、割り当てを要求したスレッドと同じ競合グループのすべてのスレッドからアクセス可能でなければならないことを示します。access 特性値 pteam は、割り当てられるメモリーが、割り当てを要求するスレッドと同じ並列領域にバインドしているすべてのスレッドからアクセス可能であることを示します。メモリーを割り当てたスレッドと同じ並列領域にバインドしていないスレッドによるメモリーへのアクセスは、未定義の動作となります。access 特性値 thread は、割り当てられるメモリーが、メモリーを割り当てたスレッドからのみアクセス可能であることを示します。他のスレッドがメモリーを割り当てようとすると、定義されていない動作になります。

alignment 特性は、割り当てられる変数のアライメントを指定します。変数は、少なくとも指定された値(バイト)でアライメントされます。一部のディレクティブと OpenMP* ランタイム・アロケーター・ランタイム・ルーチンは、ここでは説明していない追加のアラインメント要件を指定することができます。

fallback 特性は、アロケーターが割り当て要求を満たすことができない場合の動作を示します。値 abort_fb は、割り当て要求に失敗した場合にプログラムが終了することを示します。fallback 特性が allocator_fb に設定されている場合、割り当て要求に失敗すると、fb_data trait で指定されたアロケーターによって割り当てが試行されます。値 null_fb は、アロケーターが割り当て要求に失敗した場合にゼロの値を返すことを示します。値 default_mem_fb は、失敗した割り当て要求が omp_default_mem_space メモリー空間で試行されることを示します。omp_default_mem_space アロ

ケーターのすべての特性は、デフォルトの特性値に設定する必要がありますが、fallback 特性は null_fb に設定する必要があります。

fb_data 特性は、要求されたアロケーターが割り当て要求を満たせなかった場合に使用するフォールバック・アロケーターを指定することを可能にします。fb_data 特性で指定されたアロケーターを使用するには、失敗したアロケーターの fallback 特性が allocator_fb に設定されていなければなりません。

partition 特性は、アロケーターのメモリー空間によって表されるストレージリソース上で、割り当てられるメモリーがどのように分割されるかを示します。値 blocked は、割り当てられるメモリーが、ストレージリソースごとに 1 つの、ほぼ同じサイズのメモリーブロックに分割されることを示します。値 environment は、ランタイム実行環境によって割り当てられるメモリーの配置が決定されることを示します。値 interleaved は、割り当てられるメモリーが、ストレージリソース間でラウンドロビン方式で分散されることを示します。値 nearest は、割り当てられるメモリーが、割り当てを要求したスレッドに最も近いストレージリソースに配置されることを示します。

pinned 特性の値が true の場合、アロケーターによって行われた各割り当ては、解放されるまでストレージの同じ位置に残ります。

pool_size の値は、割り当てが行われていない場合にアロケーターが利用可能なストレージの総バイト数です。access 特性の値が all の場合、pool_size の値は、アロケーターにアクセスするすべてのスレッドのすべての割り当ての上限値です。アロケーターの access 特性の値が cgroup の場合、pool_size の値は、同じ競合グループ内のスレッドから行われる割り当ての上限値です。access 特性値 pteam の割り当てでは、pool_size の値は、同じ並列チーム内で行われる割り当ての上限値です。access 特性の値が thread の場合、pool_size の値は、アロケーターを使用して各スレッドから行われる割り当ての上限値です。pool_size の値を超える割り当てを要求すると、アロケーターは割り当て要求を満たすことができません。

sync_hint 特性は、複数のスレッドがアロケーターにアクセスする方法を示します。値 contended は、多くのスレッドが同時に割り当てを要求することが想定されることを示し、uncontended は、少数のスレッドが同時に割り当てを要求することが想定されることを示します。値 private は、すべての割り当て要求が同じスレッドから行われることを示します。private が指定されている場合、2 つ以上のスレッドが同じアロケーターによる割り当て要求を行うと未定義の動作になります。値 serialized は、複数のスレッドが同時に割り当てを要求しないことを示します。sync_hint 値が serialized のアロケーターによって 2 つ以上のスレッドが同時に割り当てを要求した場合の動作は未定義です。

あらかじめ定義された 5 つのメモリー空間があります。

- omp_default_mem_space は、システムのデフォルトのメモリーです。
- omp_large_cap_mem_space は、大容量メモリーです。
- omp_high_bw_mem_space は、高帯域幅メモリーです。
- omp_low_lat_mem_space は、低レイテンシーのメモリーです。
- omp_const_mem_space は、定数値の格納に最適なメモリーです。
  コンパイル時の定数式で初期化するか、firstprivate 節を使用して初期化できます。
  omp_const_mem_space 内の変数への書き込みは未定義の動作になります。

以下の表は、OpenMP* の事前定義済みメモリー・アロケーター、関連するメモリー空間、およびデフォルト以外のメモリー特性値を示しています。

アロケーター名	関連するメモリー空間	デフォルト以外の特性値
omp_default_mem_alloc	omp_default_mem_space	なし
omp_large_cap_mem_alloc	omp_large_cap_mem_space	なし
omp_low_lat_mem_alloc	omp_low_lat_mem_space	なし
omp_high_bw_mem_alloc	omp_high_bw_mem_space	なし
omp_const_mem_alloc	omp_const_mem_space	なし
omp_cgroup_mem_alloc	実装/システム定義	access=cgroup
omp_pteam_mem_alloc	実装/システム定義	access=pteam
omp_thread_mem_alloc	実装/システム定義	access=thread

関連情報

ALLOCATE 節(英語)
ALLOCATE ディレクティブ(英語)
OpenMP* ランタイム・ライブラリー・ルーチン

OpenMP* の高度な問題

ここでは、OpenMP* ライブラリー関数と環境変数の使用方法について説明し、OpenMP* を使用してパフォーマンスを向上するためのいくつかのガイドラインを示します。

OpenMP* は、特定の関数呼び出しおよび環境変数を提供します。ここで使用する主要な関数および環境変数については、次のトピックを参照してください。

- OpenMP* ランタイム・ライブラリー・ルーチン
- OpenMP* 環境変数(英語)

この関数呼び出しを使用するには、`omp_lib.h` ヘッダーファイルをインクルードするか、または `use omp_lib` を指定して Fortran90 モジュールファイルを使用します。これらのファイルは、コンパイラーのインストール時に `INCLUDE` ディレクトリーにインストールされ、`/Qopenmp`(Windows*)または `-qopenmp`(Linux* および macOS*)オプションを使用してアプリケーションをコンパイルします。

次の例では、OpenMP* 関数を使用してアルファベットを出力する方法といくつかの重要な概念について説明します。

1. ディレクティブの代わりに関数を使用するには、コードを書き換える必要があります。コードの書き換えには、追加のデバッグ、テスト、メンテナンスが伴います。
2. OpenMP* サポートなしにコンパイルすることは困難です。
3. 次のループではスレッド数が 26 の倍数でない場合、アルファベットの文字はすべて出力されません。このように、単純なバグは簡単に引き起こされます。
4. ワークキューのアルゴリズムを独自に作成しない限り、ループ・スケジュールの調整ができなくなります。ワークキューのアルゴリズムを独自に作成する場合、一般的には例に示すような `STATIC` スケジュールが多く、自らのスケジュールによって制限されることになります。

例

```
include "omp_lib.h"
integer i
integer LettersPerThread, ThisThreadNum, StartLetter, EndLetter

call omp_set_num_threads(4)
    !$OMP PARALLEL PRIVATE(i)

        ! OMP_NUM_THREADS is not a multiple of 26,
        ! which can be considered a bug in this code.
        LettersPerThread = 26 / omp_get_num_threads()
        ThisThreadNum = omp_get_thread_num()
        StartLetter = 'a'+ThisThreadNum*LettersPerThread
        EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread

        DO i = StartLetter, EndLetter - 1
             write( *,FMT='(A)',ADVANCE='NO') char(i)
        END DO

     !$OMP END PARALLEL
write(*,*)
end
```

スレッド・アプリケーションのデバッグには細心の注意が必要です。これは、デバッガーによってランタイム時のパフォーマンスが左右され、競合状態が表面化しないことがあるためです。PRINT 文でさえも、問題を発見しにくくすることがあります。これは、PRINT 文が、同期およびオペレーティング・システム関数を使用するためです。OpenMP* 自体も、プライベート変数と共有変数を区別するために追加の構造を挿入するため、さらに問題を複雑にします。OpenMP* をサポートするデバッガーを使用することにより、変数を検証しステップ実行することが可能になります。また、インテル® Inspector を使用して、発見が困難なスレッド化エラーを分析して検出することができます。高度なデバッグツールを使用しなくても、排除処理が問題の特定に役立つこともあります。

誤りの多くは競合状態です。ほとんどの競合状態は、本来ならばプライベート変数として宣言されるべき共有変数によって引き起こされます。最初に、並列領域内の変数から検証し、必要に応じて変数がプライベートとして宣言されていることを確認します。次に、並列領域内の関数呼び出しを確認します。

次に示す DEFAULT(NONE) 節は、発見が困難な変数を探すのに役立ちます。DEFAULT(NONE) を指定する場合、各変数はデータ共有属性節とともに宣言する必要があります。宣言がないとエラーになるため、発見が容易です。

例
`!$OMP PARALLEL DO DEFAULT(NONE) PRIVATE(x,y) SHARED(a,b)`

その他のよくある誤りは、初期化されていない変数の使用です。プライベート変数は、並列構造の入口では初期値を持っていません。FIRSTPRIVATE 節および LASTPRIVATE 節を使用して初期化してください(これには余分なオーバーヘッドが伴うため、必要な場合のみ実行します)。

ここまで試してもバグが見つからない場合は、スコープの縮小について考慮します。バイナリーハントを試してください。別の方法として、大きな並列領域をクリティカル・セクションと見なします。バグが含まれている疑いのあるコード領域を選択して、クリティカル・セクションに配置します。クリティカル・セクション内では動作し、クリティカル・セクション外では失敗するコードのセクションを探します。そして、変数を調べて、バグが明白であるかどうかを検証します。それでも動作しない場合は、コンパイラー固有の環境変数 KMP_LIBRARY=serial を設定して、プログラム全体をシリアルで実行します。

この時点でコードがまだ動作せず、OpenMP* API 関数呼び出しを使用していない場合は、/Qopenmp (Windows*)または -qopenmp(Linux* および macOS*)オプションを指定しないでコンパイルして、シリアルバージョンが動作することを確認してください。OpenMP* API 関数呼び出しを使用している場合は、/Qopenmp-stubs(Windows*)または -qopenmp-stubs(Linux* および macOS*)オプションを使用します。

パフォーマンス

OpenMP* スレッド・アプリケーションのパフォーマンスは、次の要因に大きく依存します。

- 基本となるシングルスレッド・コードのパフォーマンス。
- CPU 稼働率、アイドルスレッド、負荷のバランス。
- 複数のスレッドにより並列実行されるアプリケーションの比率。
- スレッド間における同期と通信の量。

- スレッドを生成、管理、破棄、および同期するのに必要なオーバーヘッド(fork-join と呼ばれるシングルから並列への切り替え(`single-to-parallel`)、または並列からシングルへの切り替え(`parallel-to-single`)によって増加します)。
- メモリー、バスの帯域幅、CPU 実行ユニットなど、共有リソースのパフォーマンス制約。
- 共有メモリーまたは偽の共有メモリーによって生じるメモリーの競合。

パフォーマンスの解析は常に、適切に構成された並列化アルゴリズムまたはアプリケーションから始めます。例えば、バブルソートの並列化は、手動で最適化されたアセンブリー言語であっても、良い開始位置とはいえません。スケーラビリティーに注意してください。2 個の CPU で実行するプログラムの作成は、`n` 個の CPU で実行するプログラムの作成よりも効率的ではありません。OpenMP* では、スレッド数はコンパイラーによって選択されます。このため、スレッド数に関係なく動作するプログラムが非常に望ましいといえます。生産/消費構造は、2 つのスレッド用に作成されているため、効率的ではありません。

アルゴリズムが決定したら、対象のインテル® アーキテクチャーでコード(シングルスレッド・バージョンが望ましい)が効率的に実行されることを確認します。`/Qopenmp`(Windows*)または `-qopenmp`(Linux* および macOS*)オプションをオフにしてシングルスレッド・バージョンを生成するか、`/Qopenmp-stubs`(Windows*)または `-qopenmp-stubs`(Linux* and macOS*)オプションでビルドして、通常の最適化でシングルスレッド・バージョンを実行します。

シングルスレッドのパフォーマンスを確認したら、マルチスレッド・バージョンを生成して、解析を始めます。

最適化を行うには、忍耐力、経験、実践が必要です。最適化するアプリケーションと同じようにコンピューターのリソースを使用する小規模なテストプログラムを作成して、何をすると速くなるのか試してみてください。コードの並列セクションで異なる `scheduling` 節を試すことも忘れないでください。並列領域のオーバーヘッドが実行時間に対して大きい場合、IF 節を使用してセクションをシリアルに実行すると良いかもしれません。

C/C++ における OpenMP* との互換性

インテル® Fortran コンパイラーは、呼び出し元と呼び出し先の両方で OpenMP* 構造が使用されている場合、Fortran での C/C++ の呼び出しまたは C/C++ での Fortran の呼び出しをサポートしていません。

OpenMP* 構造を使用する Fortran コードは、呼び出し先の C/C++ コードで OpenMP* 構造が使用されていない限り、C/C++ を呼び出すことができます。

関連情報

OpenMP* ランタイム・ライブラリー・ルーチン
サポートされる環境変数(英語)
openmp、Qopenmp(英語)
openmp-stubs、Qopenmp-stubs(英語)

OpenMP* 実装定義に依存する動作

ここでは、OpenMP* API 仕様で実装定義とされている動作について説明します。

注
内部制御変数 (ICV) については、OpenMP* API 仕様を参照してください。

名前	説明
single 構文	single 構文に最初に到達したスレッドが構造化ブロックを実行します。
teams 構文	num_teams 節を省略すると、生成されるチーム数は 1 になります。
dist_schedule 節、distribute 構文	dist_schedule 節を省略すると、distribute 構文のスケジュールは static になります。
omp_set_num_threads ルーチン	引数が正の整数でない場合、インテルの OpenMP* 実装は、現在のタスクの nthreads-var(ICV)の最初の要素の値を 1 に設定します。
omp_set_max_active_levels ルーチン	引数が負の整数の場合、この呼び出しは無視され、最後に設定された有効な値が使用されます。
omp_get_max_active_levels ルーチン	明示的な並列領域内から呼び出されると、omp_get_max_active_levels 領域のバインドスレッドとバインド領域(必要な場合)は現在のタスク領域になります。
OMP_SCHEDULE 環境変数	変数の値が指定された形式に適合していない場合、run-sched-var (ICV) の値は static に設定され、チャンクサイズは 1 に設定されます。
OMP_NUM_THREADS 環境変数	環境変数で指定したリストのいずれかの値が負の場合、リスト全体が無視されます。リストのいずれかの値が 0 の場合、1 に設定されます。
OMP_PROC_BIND 環境変数	値が true、false、または master、close、spread のカンマ区切りのリストでない場合、インテルの OpenMP* 実装は bind-var(ICV)を false に設定します。
OMP_DYNAMIC 環境変数	値が true または false でない場合、インテルの OpenMP* 実装は dyn-var(ICV)を false に設定します。
OMP_NESTED 環境変数	値が true または false でない場合、インテルの OpenMP* 実装は nest-var(ICV)を false に設定します。
OMP_STACKSIZE 環境変数	値が指定された形式に適合しない場合、または実装が指定されたサイズのスタックを提供できない場合、インテルの OpenMP* 実装は stacksize-var(ICV)をデフォルトのサイズ(アーキテクチャーに応じて 1MB ~ 4MB)に設定します。
OMP_MAX_ACTIVE_LEVELS 環境変数	値が負の整数の場合、または実装でサポート可能な並列領域のレベル数よりも大きい場合、インテルの OpenMP* 実装は max-active-levels-var(ICV)を特定のプラットフォームでサポートされる並列領域の最大レベル数に設定します。

OMP_THREAD_LIMIT 環境変数	値が負の整数の場合、または値が実装でサポート可能なスレッド数よりも大きい場合、インテルの OpenMP* 実装は thread-limit-var(ICV)を特定のプラットフォームでサポートされる最大スレッド数に設定します。要求された値が 0 の場合、インテルの OpenMP* 実装は thread-limit-var(ICV)を 1 に設定します。
ランタイム・ライブラリーの定義	インテルの OpenMP* 実装は、インクルード・ファイル omp_lib.h とモジュール omp_lib の両方を提供します。

OpenMP* の例

次の例では、いくつかの OpenMP* 機能の使用方法を示します。

単純な差分演算子

この例は、各繰り返しごとにワーク量が異なる単純な並列ループを示したものです。負荷のバランスを向上させるため、動的スケジュールを使用しています。

並列領域の最後に暗黙的な barrier があるため、`END DO` に `NOWAIT` が含まれています。

例

```
subroutine do_1(a,b,n)
 real a(n,n), b(n,n)
 !$OMP PARALLEL SHARED(A,B,N)
 !$OMP DO SCHEDULE(DYNAMIC,1) PRIVATE(I,J)
 do i = 2, n
 do j = 1, i
 b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
 end do
 end do
 !$OMP END DO NOWAIT
!$OMP END PARALLEL
end
```

2 つの差分演算子：DO ループバージョン

例では、fork/join のオーバーヘッドを減らすために融合される 2 つの並列ループを使用します。2 番目のループで使用するすべてのデータは最初のループで使用されるすべてのデータと異なるため、最初の `END DO` ディレクティブには、`NOWAIT` 節が含まれています。

例

```
subroutine do_2(a,b,c,d,m,n)
 real a(n,n), b(n,n), c(m,m), d(m,m)
 !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
 !$OMP DO SCHEDULE(DYNAMIC,1)
 do i = 2, n
 do j = 1, i
 b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
 end do
 end do
 !$OMP END DO NOWAIT
 !$OMP DO SCHEDULE(DYNAMIC,1)
 do i = 2, m
 do j = 1, i
 d(j,i) = ( c(j,i) + c(j,i-1) ) / 2.0
 end do
 end do
 !$OMP END DO NOWAIT
 !$OMP END PARALLEL
end
```

2 つの差分演算子：SECTIONS バージョン

例では、SECTIONS ディレクティブの使用方法を示します。ロジックは、前述の DO の例と同じですが、DO の代わりに SECTIONS を使用します。ここでは、2 つの作業単位しかないため、スピードアップは、2 が限度です。前述の例では、作業単位は (n-1) + (m-1) です。

例

```
subroutine sections_1(a,b,c,d,m,n)
 real a(n,n), b(n,n), c(m,m), d(m,m)
 !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
 !$OMP SECTIONS
 !$OMP SECTION
 do i = 2, n
 do j = 1, i
 b(j,i)=( a(j,i) + a(j,i-1) ) / 2.0
 end do
 end do
 !$OMP SECTION
 do i = 2, m
 do j = 1, i
 d(j,i)=( c(j,i) + c(j,i-1) ) / 2.0
 end do
 end do
 !$OMP END SECTIONS NOWAIT
 !$OMP END PARALLEL
end
```

共有スカラーの更新

この例では、共有配列 a の要素を更新する SINGLE 構造の使用方法を示します。最初のループの後にくるオプションの NOWAIT 節は取り除かれています。これは、SINGLE 構造に進む前にループの最後で待機する必要があるためです。

例

```
subroutine sp_1a(a,b,n)
 real a(n), b(n)
 !$OMP PARALLEL SHARED(A,B,N) PRIVATE(I)
 !$OMP DO
 do i = 1, n
 a(i) = 1.0 / a(i)
 end do
 !$OMP SINGLE
 a(1) = min( a(1), 1.0 )
 !$OMP END SINGLE
 !$OMP DO
 do i = 1, n
 b(i) = b(i) / a(i)
 end do
 !$OMP END DO NOWAIT
 !$OMP END PARALLEL
end
```

製品とパフォーマンス情報

[1] 実際の性能は利用法、構成、その他の要因によって異なります。詳細については、www.Intel.com/PerformanceIndex（英語）を参照してください。