OpenMP* 4.5 による 新しいレベルの 並列プログラミング

- 2016 年 12 月

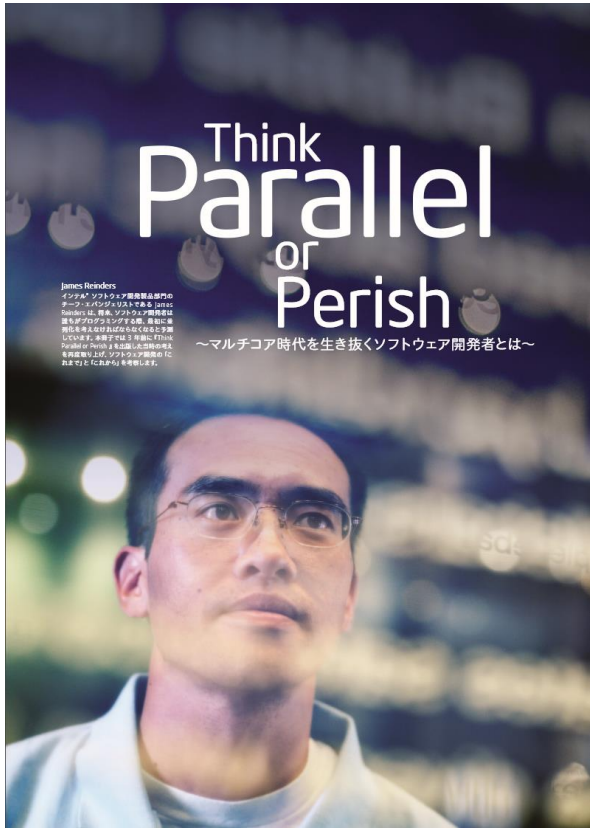
このセッションの目的

- 明示的な並列プログラミング手法として注目されてきた OpenMP* による並列プログラミングに加え、インテル® コンパイラーがサポートする OpenMP* 4.0 と 4.5 の機能を使用したベクトル・プログラミングとオフロード・プログラミングを紹介します

セッションの対象者

- OpenMP* 4.0 と 4.5 でサポートされるベクトル化機能とオフロード機能を導入し、アプリケーションのパフォーマンスをさらに向上したい開発者

Think Parallel or Perish



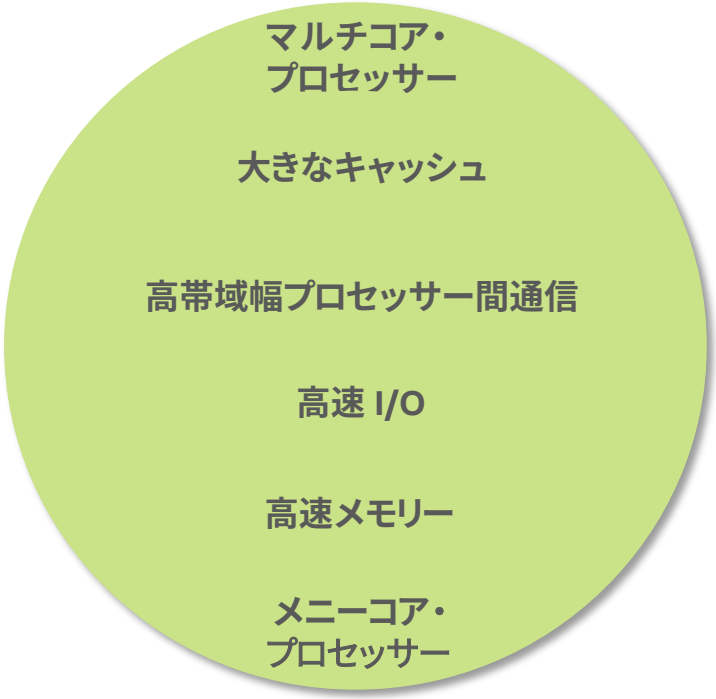
あれから 7 年 ...

Vectorize or Die

Think Parallel or Perish

Code Modernization

Code Modernization (コードのモダン化)



マルチレベルの並列性を実装する 5 つのステップ

ステージ 1: 最適化ツールとライブラリーの活用

ステージ 2: スカラーとシリアルの最適化

ステージ 3: ベクトル化

ステージ 4: 並列化

ステージ 5: マルチコアからメニーコアへスケール

http://www.isus.jp/article/article-parallel/what-is-code-modernization/

本日の内容

- 背景 (OpenMP* とは、歴史、各バージョンの機能概要) ー 30 分
- OpenMP* 4.0 と 4.5 の新しい機能 ー 90 分
- インテル® VTune™ Amplifier XE による OpenMP* のパフォーマンス解析 ー 30 分

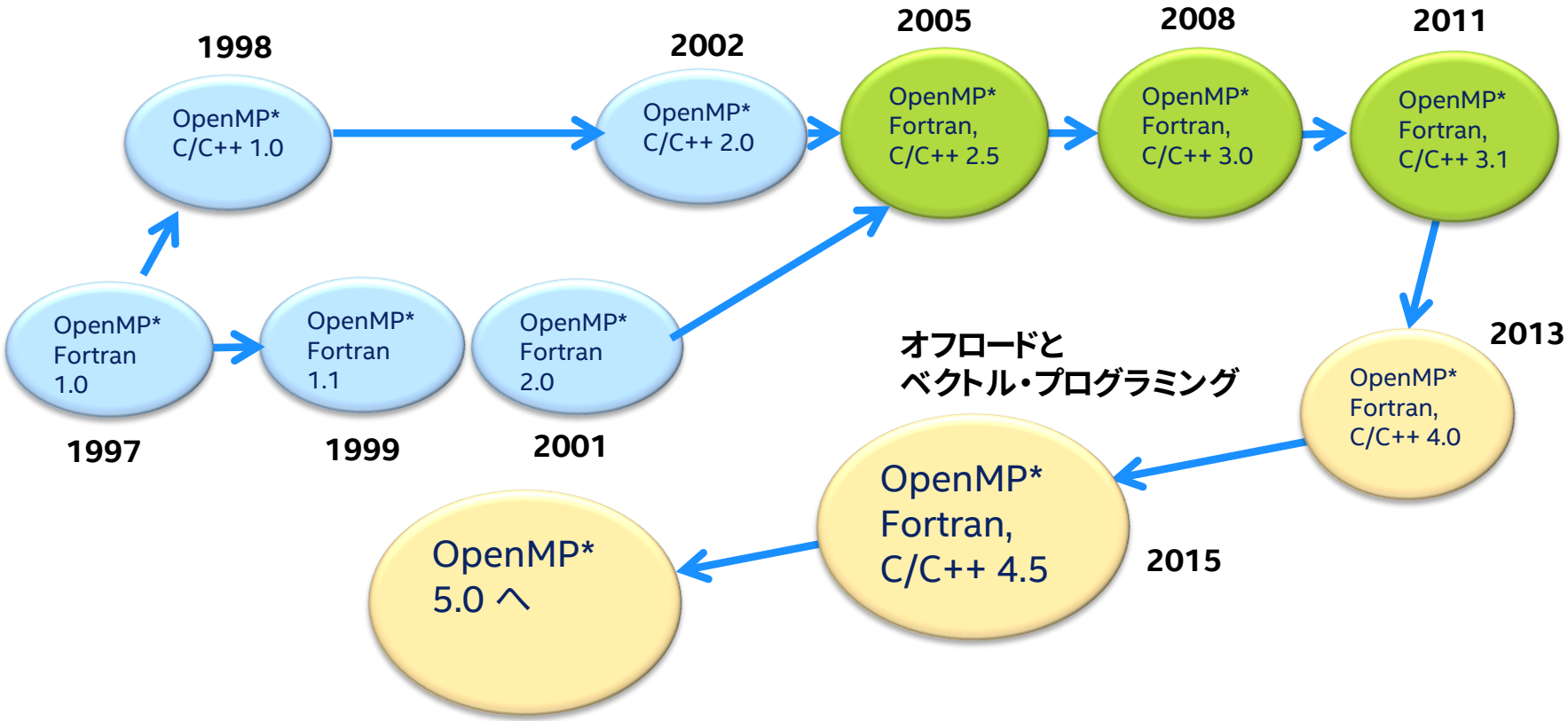
OpenMP* API

ほぼ 20 年来の技術計算/HPC における共有メモリー並列プログラミング (C/C++ と Fortran) 向けの業界標準;

- 最新の仕様: OpenMP* 4.5 (2015 年 11 月)
- ディレクティブ (C/C++ ではプラグマ) ベース
- ベンダーとプラットフォームにわたる移植性 - インテルは OpenMP* ARB (アーキテクチャー・レビュー委員会) のメンバーであり、OpenMP* のサポートを完全にコミット
- 各種並列性をサポート: スレッド、タスク、SIMD、オフロード
- インクリメンタルな並列処理をサポート

仕様ドキュメント、サンプル、および各種情報は www.openmp.org を参照してください

OpenMP* の歴史


1998
OpenMP* C/C++ 1.0
2002
OpenMP* C/C++ 2.0
2005
OpenMP* Fortran, C/C++ 2.5
2008
OpenMP* Fortran, C/C++ 3.0
2011
OpenMP* Fortran, C/C++ 3.1
OpenMP* Fortran 1.0
1997
OpenMP* Fortran 1.1
1999
OpenMP* Fortran 2.0
2001
オフロードと
ベクトル・プログラミング
2013
OpenMP* Fortran, C/C++ 4.0
OpenMP* Fortran, C/C++ 4.5
2015
OpenMP* 5.0 へ

バージョン 4.0 と 4.5 の違い

- Fortran 2003 のいくつかの機能のサポートが追加されました。
- ループ構文の **ordered** 句に引数が追加されました。これにより、**ordered** 構文にループ伝搬後方依存があるループで simd 構文を使用可能にするため doacross ループ入れ子がサポートされます。
- ループ構文に **linear** 句が追加されました。
- SIMD チャンクごとに必要な反復数の指定をサポートするため **simd** 構文に **simdlen** 句が追加されました。
- 明示的なタスクの実行優先順位に関するヒントの指定をサポートするため、**task** 構文に **priority** 句が追加されました。優先順位の値を取得するため **omp_get_max_task_priority** ルーチンが、そして許可する優先順位の最大値の値を制御するため **OMP_MAX_TASK_PRIORITY** 環境変数が追加されました。
- OpenMP* タスクを生成するネスト可能な並列ループをサポートするため、**taskloop** 構文が追加されました。
- ネイティブデバイスの実装との対話をサポートするため、**target data** 構文に **use_device_ptr** 句が追加され、**target** 構文に **is_device_ptr** 句が追加されました。
- **target** 領域の非同期実行のサポートを改善するため、**target** 構文に **nowait** と **depend** 句が追加されました。
- **target** 構文に **private**、**firstprivate** および **defaultmap** 句が追加されました。
- **declare target** ディレクティブが、特定のデバイスの実行に関連するグローバル変数の割り当てと、C/C++ での拡張リストの指定を可能にするため拡張されました。
- デバイスへの非構造化データ割り当てをサポートするため、**target enter data** と **target exit data** 構文が追加され、**map** 句が変更されました。

バージョン 4.0 と 4.5 の違い (続き)

- デバイス構文へのショートカットを充実されるため、**target parallel**、**target parallel loop**、**target parallel loop SIMD**、および **target simd** の結合構文が追加されました。
- 結合構文に適用できるように、**if** 句にデバイス名修飾子が指定できるようになりました。
- **critical** 構文に **hint** 句が追加されました。
- Doacross ループの入れ子をサポートするため、**depend** 句に **source** と **sink** 依存性タイプが追加されました。
- **target** 領域内のスカラー変数向けの明示的なデータ共有属性が、**firstprivate** に変更されました。
- いくつかのデータ共有属性句でいくつかの C++ 参照型の使用が許可されました。
- C/C++ 配列セクションでのリダクション向けのセマンティクスが追加され、リダクションにおける配列とポインターの使用に関する制限がなくなりました。
- **linear** 句に **ref**、**val** および **uval** 修飾子が追加されました。
- **map** 句に構造体要素を処理する機能のサポートが追加されました。
- OpenMP* スレッドのアフィニティーを問い合わせるクエリー関数が追加されました。
- ロック API が、アプリケーション・コードの意図するロックの利用を実装が選択できるように、ヒント付きのロックをサポートするためルーチンが拡張されました。
- 明示的な割り当て、再割り当て、メモリー転送、およびメモリーへの関連付けを可能にするため、デバイス・メモリー・ルーチンが追加されました。

バージョン 3.1 と 4.0 の違い

- Fortran 2003 の初期サポートを提供するため仕様全体を通してさまざまな変更が行われました。
- 配列セクションをサポートするため C/C++ 配列シンタックスが拡張されました。
- スレッドのアフィニティーをサポートするため、**proc_bind** 句 、**OMP_PLACES** 環境変数および **omp_get_proc_bind** ランタイムルーチンが追加されました。
- SIMD 並列をサポートするため **SIMD** 構文が追加されました。
- デバイス上での実行をサポートするため、**デバイス構文**、**OMP_DEFAULT_DEVICE** 環境変数、**omp_set_default_device**、**omp_get_default_device**、**omp_get_num_devices**、**omp_get_num_teams**、**omp_get_team_num** および **omp_is_initial_device** ランタイムルーチンがサポートされました。
- untied タスク向けのタスクのスケジューリング・ポイント実装定義が削除されました。
- タスクの依存性をサポートするため **depend** 句が追加されました。
- 柔軟性のある深いタスクの同期をサポートするため **taskgroup** 構文が追加されました。
- ユーザー定義リダクションをサポートするため、**reduction** 句が拡張され、**declare reduction** 構文が追加されました。
- **atomic** 構文が、**capture** 句で atomic スワップをサポートし、新しい atomic updtae と capture 形式を利用できるようにし、**seq_cst** 句でシーケンシャルで一貫した atomic 操作をサポートするように拡張されました。
- **cancel** 構文、**cancellation point** 構文、**omp_get_cancellation** ランタイムルーチンと **OMP_CANCELLATION** 環境変数がキャンセルのコンセプトをサポートするために追加されました。
- **OMP_DISPLAY_ENV** 環境変数が、OpenMP* 環境変数と関連する内部変数の値を表示するため追加されました。

バージョン 3.0 と 3.1 の違い

- **final** と **mergeable** 句が、タスクデータ環境の最適化をサポートする **task** 構文に追加されました。
- ユーザー定義のタスク・スケジュール・ポイントをサポートするため **taskyield** 句が追加されました。
- **atomic** 構文に **read**、**write**、**capture** 形式が追加され、既存の **atomic** 構文の形式に適用できる **update** 句が追加されました。
- データ環境の制限が変更され、**intent(in)** と **const** 修飾型が **firstprivate** 句で許可されるようになりました。
- データ環境の制限が変更され、**firstprivate** 句と **lastprivate** 句で Fortran ポインターが許可されました。
- C と C/C++ に新しいリダクション演算子 **min** と **max** が追加されました。
- **atomic** 領域内に密接した OpenMP* 領域を許可しないことが明確化されました。これにより、**atomic** 領域をほかの OpenMP* 領域と一貫性を持って定義し、すべてのコードを atomic 構文に含めることができます。
- final task 領域の同期をサポートするため **omp_in_final** ランタイムルーチンが追加されました。
- 内部変数 *nthreads-var* は、ネストされた並列領域レベルごとに使用するスレッド数のリストに変更されています。内部変数の値は、**OMP_NUM_THREADS** 環境変数を使用して設定できますが、アルゴリズムが並列領域で使用するスレッド数を決定する場合、リストを処理するように変更されています。
- 内部変数 *bind-var* が追加されました。これは、スレッドをプロセッサーにバインドされるかどうかを制御します。内部変数の値は、**OMP_PROC_BIND** 環境変数を使用して設定できます。
- Fortran インターフェイスの omp_integer_kind の誤った用法を select_int_kind に置き換えました。

バージョン 2.5 と 3.0 の違い

- OpenMP* 実行モデルにタスクの概念を導入しました。
- タスクを明示的に作成するメカニズムとして、**task** 構文が追加されました。
- タスクが生成した子タスクの完了を待機するため、**taskwait** 構文が追加されました。
- OpenMP* メモリーモデルが、アトミックなメモリーアクセスをカバーできるようになりました。**flush** 操作の項目から **volatile** の振る舞いの説明が削除されました。
- バージョン 2.5 では、プログラム全体で 1 つの *nest-var*、*dyn-var*、*nthreads-var* および *run-sched-var* 内部制御変数を持っていました。バージョン 3.0 では、タスクごとに 1 つのコピーを持ちます。その結果、**omp_set_num_threads**、**omp_set_nested** および **omp_set_dynamic** ランタイム・ライブラリー・ルーチンは、並列領域内から呼び出されても効果を持つようになりました。
- アクティブな **parallel** 領域の定義が変更されました: OpenMP* 3.0 では、2 つ以上のスレッドで構成されるチームによって実行される場合 **parallel** 領域はアクティブです。
- **parallel** 領域で使用されるスレッド数を決定する規則が変更されました。
- バージョン 3.0 では、ループ構造における反復のスレッドへの割り当ては、**static** スケジュールがデフォルトです。
- バージョン 3.0 では、ループ構造は完全に入れ子になった複数のループに関連付けられることがあります。関連するループの数は、**collapse** 句で制御できます。
- ランダム・アクセス・イテレーターと符号なし整数型変数が、ループ構造に関連付けられたループ内のループ・イテレーターとして使用できます。
- スケジュールの種別に **auto** が追加されました。これにより実装でループ構造をスレッドのチームへループ反復をマッピングする自由度が高まります。
- Fortran の形状引き継ぎ配列が、事前定義されたデータ共有属性を持ちます。
- Fortran において、**default** 句への引数として **firstprivate** 節が指定できるようになりました。
- **private** 句のリスト項目で、実装では元のリスト項目のストレージを使用して、マスタースレッド上に新しいリスト項目を保持することを許可しなくなりました。**parallel** 領域内で元のリスト項目を参照しない場合、その値は **parallel** 領域から出ても定義されています。
- バージョン 3.0 では、Fortran の割り当て可能配列を **private**、**firstprivate**、**lastprivate**、**reduction**、**copyin** そして **copyprivate** 句に指定できるようになりました。

バージョン 2.5 と 3.0 の違い (続き)

- バージョン 3.0 では、**threadprivate** ディレクティブに静的クラスメンバー変数を指定できるようになりました。
- バージョン 3.0 では、private と threadprivate クラス型変数のコンストラクターとデストラクターの呼び出し場所と引数が明確にされました。
- ランタイム・ライブラリー・ルーチン **omp_set_schedule** と **omp_get_schedule** が追加されました。このルーチンは、内部制御変数 *run-sched-var* の値を設定および取得します。
- OpenMP* プログラムが使用する最大スレッド数を制御する内部制御変数 *thread-limit-var* が追加されました。この内部制御変数の値は、**OMP_THREAD_LIMIT** 環境変数で設定し、**omp_get_thread_limit** ランタイム・ライブラリー・ルーチンで取得できます。
- アクティブな **parallel** 領域の入れ子数を制御する内部制御変数、*max-active-levels-var* が追加されました。この内部制御変数は、**OMP_MAX_ACTIVE_LEVELS** 環境変数と **omp_set_max_active_levels** ランタイム・ライブラリー・ルーチンを使用して設定し、**omp_get_max_active_levels** ランタイム・ライブラリー・ルーチンで取得できます。
- OpenMP* 実装が作成するスレッドのスタックサイズを制御する内部制御変数 *stacksize-var* が追加されました。内部制御変数の値は、**OMP_STACKSIZE** 環境変数を使用して設定できます。
- 待機中のスレッドの振る舞いを制御する内部制御変数 *wait-policy-var* が追加されました。内部制御変数の値は、**OMP_WAIT_POLICY** 環境変数を使用して設定できます。
- この呼び出しを含むタスクの parallel 領域の入れ子レベルを返す **omp_get_level** ランタイム・ライブラリー・ルーチンが追加されました。
- 呼び出しを行ったスレッドの入れ子レベルの祖先のスレッド番号を返す、**omp_get_ancestor_thread_num** ランタイム・ライブラリー・ルーチンが追加されました。
- 呼び出しを行ったスレッドの入れ子レベルの祖先に関連するスレッドチームのサイズを返す、**omp_get_team_size** ランタイム・ライブラリー・ルーチンが追加されました。
- この呼び出しを含むタスクの入れ子になったアクティブな **parallel** 領域の数を返す **omp_get_active_level** ランタイム・ライブラリー・ルーチンが追加されました。
- バージョン 3.0 では、ロックはスレッドではなくタスクによって保持されます。

コンパイラーが OpenMP* をサポートするか?

OpenMP* は最も簡単なマルチスレッド・プログラミング・モデルであるが、コンパイラーが OpenMP* をサポートしている必要がある

OpenMP* をサポートするコンパイラーは、_OPENMP マクロに仕様が公開された年月を数値で返す

仕様のバージョン	値	代表的なコンパイラー
OpenMP* 4.5	201511	インテル® コンパイラー 17.0 、gcc 5.?
OpenMP* 4.0	201307	インテル® コンパイラー 14.0 、gcc 4.9 (offload 5.1)
OpenMP* 3.1	201107	インテル® コンパイラー 12.1 、gcc 4.7
OpenMP* 3.0	200805	インテル® コンパイラー 11.0 、gcc 4.4
OpenMP* 2.5	200505	インテル® コンパイラー 9.0 、gcc 4.2

S

記述している構文が OpenMP* のどの仕様か?

OpenMP API 4.5 C/C++ Page 1

OpenMP

C/C++ 日本語翻訳版

OpenMP 4.5 API C/C++ シンタックス・リファレンス・カード

OpenMP アプリケーション・プログラム・インターフェイス (API) は、スケーラブルでポータブルな並列処理を実現します。ポータブルな並列アプリケーションを開発するためのシンプルで柔軟性のあるインターフェイスです。

OpenMP は、Unix プラットフォームと Windows プラットフォームを含む、すべてのアーキテクチャー上での C/C++ と Fortran による共有メモリー型並列プログラミングをサポートします。 仕様に関しては www.openmp.org をご覧ください。

このテキストの色 は、OpenMP 4.5 仕様の新機能もしくは変更点を指します。
[n.n.n] は、OpenMP API 4.5 仕様書のセクションを示します。
[n.n.n] は、OpenMP API 4.0 仕様書のセクションを示します。

C/C++ のディレクティブと構文

OpenMP の実行ディレクティブは、後続の構造化ブロックや OpenMP 構文に適用されます。各ディレクティブは #pragma omp で始まります。ディレクティブの残りの部分は、C と C++ コンパイラーの標準規約に準拠します。構造化ブロックは、単一のステートメントか出入り口がそれぞれ 1 つの複合文です。

parallel [2.5] [2.5]
スレッドのチームを形成し、並列実行を開始します。

#pragma omp parallel *[節[[,]節] ...]*
構造化ブロック

節:
if(*[parallel:]スカラー式*)
num_threads(*整数式*)

sections [2.7.2] [2.7.2]
一連の構造化ブロックを含む非反復型のワークシェアリング構文は、スレッドのチームに分散され実行されます。

#pragma omp sections *[節[[,] 節] ...]*
{
[#pragma omp section]
構造化ブロック

for simd [2.8.3] [2.8.3]
ループが SIMD 命令を使用して命令レベルの並列性を持つと同時に、それらの反復がスレッドのチームで並列に実行できることを指示します。

#pragma omp for simd *[節[[,]節] ...]*
for ループ

節

http://openmp.org/wp/openmp-specifications/ (英語)

インテルの OpenMP* ライブラリーのよくある質問

- ほかのコンパイラーとのソースとオブジェクトの互換性
- インテル® コンパイラーの異なるバージョンを使用するガイドライン
- インテル® コンパイラーとその他のコンパイラーを併用するガイドライン
- OpenMP* ライブラリーをその他のコンパイラーで使用する際の制限
- Visual Studio® でインテルの OpenMP* ライブラリーを使用する際の制限
- OS X* で OpenMP* を使用する際の注意
- Fortran と C/C++ の混在
- インテルの OpenMP* のサポートライブラリー

ほかのコンパイラーとのソースとオブジェクトの互換性

- インテル® C++ コンパイラーには、次のコンパイラーの OpenMP* サポートとソースおよびオブジェクト・レベルの互換性があります:
  - Microsoft® Visual C++® 2012 以降のコンパイラー (Windows®)
  - GCC 4.4.2 以降 (Linux*)
  - インテル® C++ コンパイラー 10.0 以降

コンパイラーが異なるとサポートされる OpenMP* 仕様も異なります。
アプリケーションが使用する OpenMP* 機能に基づいて、どのバージョンの OpenMP* 仕様が必要かを判断してください

複数のソースコードの一部のみで OpenMP* を使用する

- もしくは異なるバージョンのコンパイラーでプロジェクトをビルドする場合

```
main() {
    printf("OpenMP version %d ¥n", _OPENMP);
    printf("OpenMP version %d ¥n", omp());
}
```

```
int omp(void){
return _OPENMP;
}
```

インテル® コンパイラー 17.0 でコンパイル

インテル® コンパイラー 16.0 でコンパイル

```
OpenMP version 201511
OpenMP version 201307
```

Visual C++® 2015 のコンパイラーでコンパイル

```
OpenMP version 201511
OpenMP version 200203
```

S

インテル® コンパイラーの異なるバージョンを使用するガイドライン

リンク時または実行時の問題を回避するには、次の点に注意してください:

- 現行のインテル® C++ コンパイラーで生成されたオブジェクト・モジュールは、インテル® コンパイラー 10.0 より前のバージョンでコンパイルされたオブジェクト・モジュールとは互換性がありません
- OpenMP* スタティック・ライブラリーではなく、ダイナミック・ライブラリーを使用して、ライブラリーの複数のコピーが 1 つのプログラムにリンクされないようにします (Linux*、OS X*)

インテル® C++ コンパイラーとその他のコンパイラーを併用するガイドライン

リンク時または実行時の問題を回避するには、次の点に注意してください:

- 常にインテル® C++ コンパイラーの OpenMP* ライブラリーを使用してリンクします。これにより、異なるコンパイラーから OpenMP* ランタイム・ライブラリーの複数のコピーがリンクされるのを防ぎます
- 可能であれば、すべての OpenMP* ソースを同じコンパイラーでコンパイルします
- OpenMP* スタティック・ライブラリーではなく、ダイナミック・ライブラリーを使用して、ライブラリーの複数のコピーが 1 つのプログラムにリンクされないようにします

OpenMP* ライブラリーをその他のコンパイラーで使用する際の制限

オブジェクト・レベルの互換性には、threadprivate オブジェクトに次の制限があります:

- インテル® C++ コンパイラーは threadprivate データを参照する際、デフォルト (/Qopenmp-threadprivate:legacy) で GCC や Microsoft® Visual C++® コンパイラーとは異なる方法を用います
- コードで変数を threadprivate と宣言し、インテル® C++ コンパイラーと GCC/Visual C++® コンパイラーの両方でコンパイルする場合、インテル® C++ コンパイラーでコンパイルされたコードと GCC/Visual C++® コンパイラーでコンパイルされたコードでは、同じスレッドによって参照されていたとしても、異なる場所の変数が参照されます
- GCC/Visual C++® コンパイラーでコンパイルされたコードと同じ threadprivate 場所を参照させるには、インテル® C++ コンパイラーでのコンパイル時に、/Qopenmp-threadprivate:compat オプションを使用します

Visual Studio® でインテルの OpenMP* ライブラリーを使用する際の制限

Windows® システムで、Visual Studio® の Visual C++® 開発環境を一部変更すると、インテル® C++ コンパイラーと Visual C++® を使用して、インテルの OpenMP* ライブラリーを使用するアプリケーションを作成することができます

Microsoft® Visual C++® では、**_OPENMP_NOFORCE_MANIFEST** シンボルが定義されていなければなりません。定義されていない場合は、vcomp90 dll のマニフェストがインクルードされます。これにより、ビルドシステムでは問題ありませんが、この DLL がインストールされていない別のシステムにアプリケーションを移動すると問題が発生します

```
cl /MD /openmp /c f1.c f2.c
icl /MD /Qopenmp /c f3.c f4.c
link f1.obj f2.obj f3.obj f4.obj /out:app.exe /nodefaultlib:vcomp libiomp5md.lib
```

OS X* で OpenMP* を使用する際の注意

- 古いバージョンの OS X* プラットフォームでは、インテル® C++ コンパイラーと GCC コンパイラーを併用してコンパイルできます
- OS X* 10.9 以降の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
- OS X* 10.9 以降 (Xcode* 5.x 以降) には GCC コンパイラーが含まれていませんが、インテル® C++ 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

Fortran と C/C++ の混在

- gcc コンパイラー、インテル® C++ コンパイラー、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
- 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
- インテル® C++ 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とができますが、-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

C ソースと
Fortran ソースの混在

```
gfortran -fopenmp -c foo.f
icc -qopenmp -c ibar.c
icc -qopenmp foo.o bar.o -lgfortranbegin -lgfortran
```

OpenMP* のサポート・ライブラリー

OS	ダイナミック・リンク	スタティック・リンク
Linux*	libiomp5.so	libiomp5.a
OS X*	libiomp5.dylib	libiomp5.a
Windows®	libiomp5md.lib libiomp5md.dll	なし

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* 構造および機能は、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、バインドです

実行環境を確認する便利な機能

OpenMP* 4.0 以降のライブラリーでは、環境変数 OMP_DISPLAY_ENV が利用できます。true に設定すると、実行時に情報を表示できます

```
Intel Compiler 17.0 Update 1 Intel(R) 64 Visual Studio 2015
Z:¥CodesC¥OpenMP¥openmp>pi

OPENMP 表示環境開始
  _OPENMP='201511'
  [ホスト] OMP_CANCELLATION='FALSE'
  [ホスト] OMP_DEFAULT_DEVICE='0'
  [ホスト] OMP_DISPLAY_ENV='TRUE'
  [ホスト] OMP_DYNAMIC='FALSE'
  [ホスト] OMP_MAX_ACTIVE_LEVELS='2147483647'
  [ホスト] OMP_MAX_TASK_PRIORITY='0'
  [ホスト] OMP_NESTED='FALSE'
  [ホスト] OMP_NUM_THREADS: 値が定義されていません。
  [ホスト] OMP_PLACES: 値が定義されていません。
  [ホスト] OMP_PROC_BIND='false'
  [ホスト] OMP_SCHEDULE='static'
  [ホスト] OMP_STACKSIZE='4M'
  [ホスト] OMP_THREAD_LIMIT='2147483647'
  [ホスト] OMP_WAIT_POLICY='PASSIVE'
OPENMP 表示環境終了


Pi = 3.141593 Time = 0.452000
 OpenMP version = 201511

Z:¥CodesC¥OpenMP¥openmp>_
```

true、false (デフォルト)、verbose が設定できます

先に進む前に少し思い出しておきましょう

インテル® コンパイラーの OpenMP* 関連のオプション

/Qopenmp ; OpenMP* ディレクティブに基づいてコンパイラーがマルチスレッド・コードを生成するようにします。無効にするには /Qopenmp- を使用します。

/Qopenmp-stubs ; シーケンシャル・モードで OpenMP* プログラムをコンパイルします。OpenMP* ディレクティブは無視され、OpenMP* スタブ・ライブラリーがリンクされます (シーケンシャル)。

/Qopenmp-lib:<ver> ; リンクする OpenMP* ライブラリーのバージョンを選択します。

compat - Microsoft® 互換の OpenMP* ランタイム・ライブラリーを使用します (デフォルト)。

/Qopenmp-task:<arg> ; サポートする OpenMP* タスクモデルを選択します。

omp - OpenMP* 3.0 タスクをサポートします (デフォルト)。intel - インテルのタスク・キューイングをサポートします。

/Qopenmp-threadprivate:<ver> ; 使用する threadprivate 実装を選択します。

compat - GCC/Microsoft® 互換のスレッド・ローカル・ストレージを使用します。legacy - インテル互換の実装を使用します (デフォルト)。

/Qopenmp-simd ; OpenMP* SIMD コンパイルを有効にします。/Qopenmp を指定するとデフォルトで有効になります。無効にするには /Qopenmp-simd- を使用します。

/Qopenmp-offload[:<kind>] ; target プラグマの OpenMP* オフロードコンパイルを有効にします。このオプションは、インテル® MIC アーキテクチャーおよびインテル® グラフィックス・テクノロジーにのみ適用されます。/Qopenmp を指定するとデフォルトで有効になります。無効にするには /Qopenmp-offload- を使用します。target プラグマのデフォルトのデバイスを指定します。

host - オフロードの準備はしますが、ターゲットコードをホストシステムで実行します。

mic - インテル® MIC アーキテクチャー、gfx - インテル® グラフィックス・テクノロジー

OpenMP* のコンポーネント

ディレクティブ

- ワークシェア
- タスク処理
- アフィニティー
- オフロード
- キャンセル
- 同期
- SIMD

環境変数

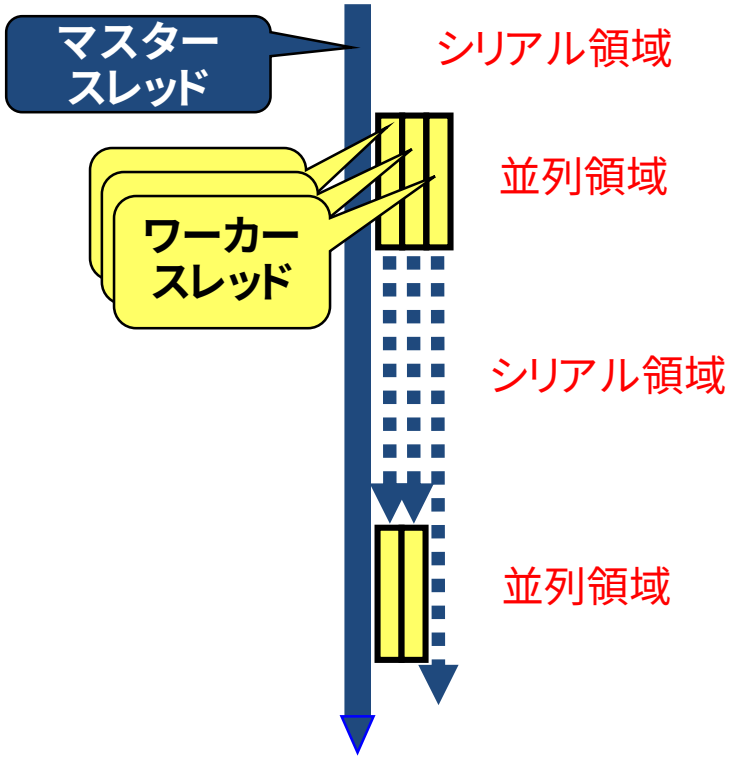
- スレッドの設定
- スレッドの制御
- ワークシェア
- アフィニティー
- アクセラレーター
- キャンセル
- 操作可能

ランタイム

- スレッド管理
- ワークシェア
- タスク処理
- アフィニティー
- アクセラレーター
- デバイスメモリー
- キャンセル
- ロック

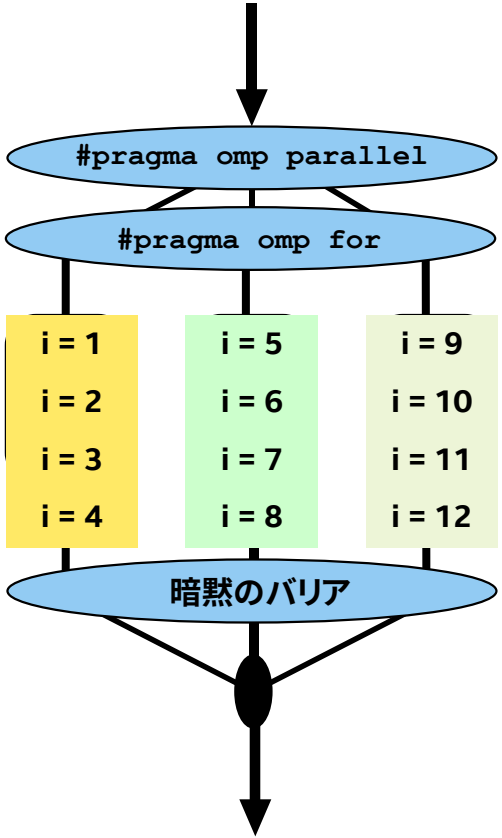
OpenMP* の実行モデル

- OpenMP* プログラムはシングルスレッドで処理を開始: マスタースレッド
- ワーカースレッドは並列領域でスポーンされ、マスターとともにスレッドのチームを形成
- 並列領域の間ではワーカースレッドはスリープ状態になる。OpenMP* ランタイムがすべてのスレッドの動作を管理
- コンセプト: フォークジョイン
  - インクリメンタルな並列処理を許可



OpenMP* 並列領域: for ワークシェアの例

```
// N=12 を想定
#pragma omp parallel
#pragma omp for
   for(i = 1, i < N+1, i++)
      c[i] = a[i] + b[i];
```



- スレッドには独立したループ反復が割り当てられる
- スレッドはワークシェア構文の最後で待機

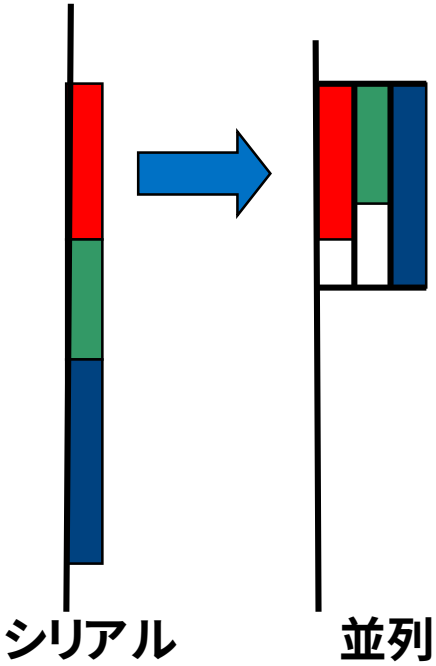
OpenMP* 並列領域: sections ワークシェアの例

独立したセクションのコードを同時に実行

– 実行時間を短縮

“非同期実行” で利用される (I/O、オフロードなど)

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



OpenMP* のリダクション

```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
```

各スレッドに *sum* のローカルコピーを作成

sum のすべてのローカルコピーはマージされ、“グローバル” 変数にストアされる

OpenMP* 4.0 と 4.5 でリダクションの概念が拡張された:

- ユーザー定義リダクション
- リダクション変数は、もやはスカラーの制限がない

```
!$OMP SIMD reduction(+:A)
do I=1,25,4
  do J=1,8
    A(J) = A(J) +
B(I,J)*B(I,J)
  end do
end do
```

OpenMP* 並列領域: single ワークシェアとタスク処理

```
#pragma omp parallel
// 8 スレッドを想定
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

ここで 8 スレッドのプールを作成

1 つのスレッドが while ループを実行

“while ループ” を実行するスレッドは、processwork() の各インスタンス向けにタスクを生成

OpenMP* の同期

- データのアクセス同期
  atomic 構文、critical 構文、ロックルーチン
- 実行制御
  barrier 句、master 句、single 句、flush 句、暗黙の同期、nowait 節

```
#pragma omp atomic
   x +=  tmp;

#pragma omp critical
     x += func(B);
```

```
#pragma omp parallel
{
    int id=omp_get_thread_num();
#pragma omp master
    A[id] = big_calc1(id);
#pragma omp barrier
    B[id] = big_calc2(id, A);
}
```

OpenMP* のデータ属性

- データ環境のデフォルト属性を変更:
  default(private | none | shared) // private は Fortran のみ

- 構文内のストレージの属性を変更:
  shared, firstprivate, private

- 並列ループ内のプライベート変数の最後の値をループ外の共有変数に転送:
  lastprivate

まとめ: ループとリダクションによる pi プログラム

```
#include <omp.h>
static long num_steps = 100000;  double x,step;
void main ()
{     int i;        double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
        #pragma omp for private(x) reduction(+:sum)
        for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
        }
      }
      pi = step * sum;
}
```

スレッドのチームを生成 … parallel 構文がないと 1 スレッド以上にはなりません

ループを分割してスレッドに割り当てます … sum へリダクション演算を行うよう設定します。

注意 … ループ・インデックスはデフォルトでスレッドローカルです

並列領域内でテンポラリー値を保持するため、各スレッドでプライベートのスカラー変数を使用します

本日の内容

- 背景 (OpenMP* とは、歴史、機能概要)
- OpenMP* 4.0 と 4.5 の新しい機能
  - タスク

    task 句の depend 節、taskloop 句、cancel 句、cancellation point 句、taskguoup 句、task 句の priority 節
  - omp SIMD
  - オフロード
  - OpenMP* 4.5 のサポート状況
- インテル® VTune™ Amplifier XE による OpenMP* のパフォーマンス解析

新しい機能を説明する前に

- Combine Construct (結合)

シーケンス内の複数のプラグマのショートカットとして使用します。結合された構文は、別の構文内で入れ子になった、もう一方の構文を指定するショートカットとなります。結合された構文は、意味的には 2 番目の構文を含んでいますが、ほかのステートメントを含まない最初の構文を指定するのと同じです

例: #pragma omp parallel for

- Composite Construct (複合)

複合構文は、2 つの構文で構成されますが、入れ子になった構文のいずれかを指定する同一の意味を持ちません。複合構文では、指定した構文が別々の意味を持ちます

例: #pragma omp for simd

OpenMP* タスクに関する拡張

従来の OpenMP* ワークシェアの問題

OpenMP* ワークシェア構文が上手く構成されていない
問題の例: 並列化された領域からインテル® MKL の dgemm を呼び出す

```
void example()
{
  #pragma omp parallel
  {
    compute_in_parallel(A);
    compute_in_parallel_too(B);
    // dgemm はパラレルもしくはシリアル
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, A, k, B, n, beta, C, n);
  }
}
```

次のいずれかの状態となる:

- オーバーサブスクライブ (dgemm が内部で並列化されている場合)
- OpenMP* のオーバーヘッドによるパフォーマンスの低下、または
- 部分行列に dgemm を適用するため、周辺コードが必要になる

OpenMP* のタスク処理: 簡単な例

```
#pragma omp parallel
// 8 スレッドを想定
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

ここで 8 スレッドのプールを作成

1 つのスレッドが while ループを実行

“while ループ” を実行するスレッドは、processwork() の各インスタンス向けにタスクを生成

タスクによるフィボナッチ数列

```
int fib ( int n )
{

  int x,y;
  if ( n < 2 ) return n;
   #pragma omp task shared(x)
    x = fib(n-1);
   #pragma omp task shared(y)
    y = fib(n-2);
   #pragma omp taskwait
    return x+y;
}
```

n は両方のタスクで
firstprivate

x と y は共有
最良の解決策
sum を計算するため x と y
両方の値が必要

task 間の変数の依存関係

生成されたタスクの実行順序は不定、参照する変数に依存関係がある場合、意図する結果が得られないことがある

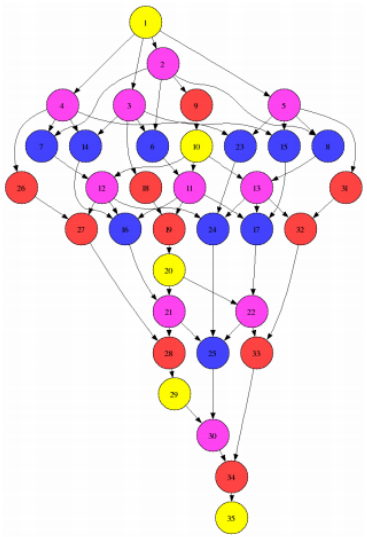
タスクが使用する変数を depend 節で依存関係 in、out、inout を指定できるようになりました

フロー依存(out,in)、アンチ依存(in,out)、出力依存(out,out) を制御

```
int val=0;

int main(){
#pragma omp parallel num_threads(3)
  {
  #pragma omp single
    {
  #pragma omp task depend(out:val)
    val = 100;
  #pragma omp task depend(in:val)
    val += 1000;
    }
  }
  printf("Value is %d¥n", val);
}
```

タスク依存関係 -「OMP フローグラフ」

```
void blocked_cholesky( int NB, float *A[NB][NB] ) {
  int i, j, k;
    for (k=0; k<NB; k++) {
        #pragma omp task depend(inout:A[k][k])
        spotrf (A[k][k]) ;
       for (i=k+1; i<NB; i++)
        #pragma omp task depend(in:A[k][k])
                                depend(inout:A[k][i])
        strsm (A[k][k], A[k][i]);
       //  末端の部分行列を更新
       for (i=k+1; i<NB; i++) {
         for (j=k+1; j<i; j++)
           #pragma omp task depend(in:A[k][i],A[k][j])
                                depend(inout:A[j][i])
             sgemm( A[k][i], A[k][j], A[j][i]);
           #pragma omp task depend(in:A[k][i])
                                depend(inout:A[i][i])
             ssyrk (A[k][i], A[i][i]);
       }
    }
}
```



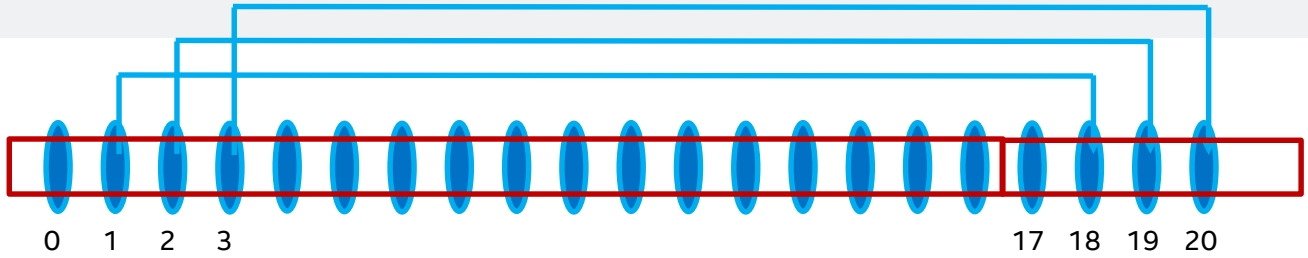
* イメージのソース: BSC

Do-across ループ並列

依存性はループ反復間で生じる

以下のコードは、ループ伝搬後方依存がある

```
void lcd_ex(float* a, float* b, size_t n, int m, float c1, float c2) {
    size_t K;
  #pragma omp parallel for ordered(1)
    for (K = 17; K < n; K++) {
      #pragma omp ordered depend(sink: K-17)
      a[K] = c1 * a[K - 17] + c2 * b[K];
      #pragma omp ordered depend(source)
    }
}
```



depend について

Task dependences are derived from the *dependence-type* of a **depend** clause and its list items when *dependence-type* is **in**, **out**, or **inout**.

- For the **in** *dependence-type*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in an **out** or **inout** *dependence-type* list of a **task** construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

- For the **out** and **inout** *dependence-types*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in an **in**, **out**, or **inout** *dependence-type* list of a **task** construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

OpenMP* 4.5 taskloop

OpenMP* task を使用してループを並列化 (Fortran と C/C++)

#pragma omp taskloop [simd] [節]
for ループ

- ループをチャンクへ分割
- オプションの **grainsize** と **num_tasks** 節でタスクの生成を制御
- インテル® Cilk™ Plus の “cilk_for” に類似
- それぞれのループチャンク向けにタスクを生成

まだ気軽に taskloop は使えない …

```
void CG_mat(Matrix *A, double *x, double *y)
{
    // ...
#pragma omp taskloop
    private(j,is,ie,j0,y0) ¥
                grainsize(500)
 for (i = 0; i < A->n; i++) {
   y0 = 0;
   is = A->ptr[i];
   ie = A->ptr[i + 1];
   for (j = is; j < ie; j++) {
     j0 = index[j];
     y0 += value[j] * x[j0];
   }
   y[i] = y0;
    // ...

}
    // ...
```

OpenMP* のタスク処理: さらに ...

キャンセル

- OpenMP* 4.0 以前では、並列実行を途中でキャンセルできなかった
  - コード領域は常に最後まで実行された (もしくはすべて実行しないか)
- OpenMP* 4.0 の cancel 句は OpenMP* 領域の中断を可能にする

タスクグループ

- 次のような処理のため、タスクを論理的にグループ化する
  - 同期
  - キャンセル

cancel 句

指定された構文タイプの最も内側の並列領域の要求をキャンセルします。if 文、while 文、do 文、switch 文とラベルの後には指定できません:

#pragma omp cancel [構文タイプ] [[,] if 節]

構文タイプ: parallel、sections、for、taskgroup

if 節: if(スカラー式)

注:
構文に到達したとき、デッドロックを引き起こす可能性があるロックやその他のデータ構造を解放する必要があります。ブロックされたスレッドは取り消すことができません

cancellation point 句

指定された構文タイプの最も内側の並列領域のキャンセルが要求された場合に、キャンセルをチェックする位置を指定:

#pragma omp cancellation point [構文タイプ]

構文タイプ: parallel、sections、for、taskgroup

制約事項:

- この構文は、実行文が許可されている場所にのみ追加できます
- if 文のアクション文として使用したり、プログラムで参照されるラベルの実行文として使用することはできません

cancel 句の例

```
#define N 10000

void example() {
    std::exception *ex = NULL;
#pragma omp parallel shared(ex)
    {
#pragma omp for
        for (int i = 0; i < N; i++) {
            // no 'if' that prevents compiler optimizations
            try {
                causes_an_exception();
            }
            catch (std::exception *e) {
             // 後で例外を処理するため ex にステータスをストア
#pragma omp atomic write
                ex = e;
#pragma omp cancel for // for ワークシェアリングをキャンセル
            }
        }
       if (ex) // 例外が発生したら parallel 構文をキャンセル
      #pragma omp cancel parallel
        phase_1();
       #pragma omp barrier
        phase_2();
    }
    // 例外がワークシェア・ループ内でスローされている場合は継続
    if (ex)
        // ex の例外処理
    } // parallel のおわり
```

例外をキャッチしたら for ワークシェア並列処理をキャンセル

例外をキャッチしたら parallel 並列処理をキャンセル

cancellation point 句の例

```
subroutine example(n, dim)
  integer, intent(in) :: n, dim(n)
  integer :: i, s, err
  real, allocatable :: B(:)
  err = 0
!$omp parallel shared(err)
! ...
!$omp do private(s, B)
  do i=1, n
!$omp cancellation point do
    allocate(B(dim(i)), stat=s)
    if (s .gt. 0) then
   !$omp atomic write
      err = s
   !$omp cancel do
    endif
!   ...
! deallocate private array B
    if (allocated(B)) then
      deallocate(B)
    endif
  enddo
!$omp end parallel
end subroutine
```

ほかのスレッドはこの位置でキャンセルをチェック

allocate 文からエラーが返されたときに
cancel do をアクティブにします

taskgroup 構文

taskgroup 構文は、現在のタスクの子タスク (孫タスク) の完了を待機することを指示できます

```
int main()
{
  int i; tree_type tree;
  init_tree(tree);
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task
        start_background_work();
    for (i = 0; i < max_steps; i++) {
        #pragma omp taskgroup
        {
           #pragma omp task
             compute_tree(tree);
        } // このステップの tree トラバースを待機
        check_step();
    }
  } // ここで start_background__work の完了を待機
  print_results();
  return 0;
}
```

この 2 つのタスクは同時に実行され、start_background_work() は compute_tree() の同期の影響を受けない

なぜ taskwait が使用できないのか?

task 句の priority 節

priotity 節は生成されたタスクの優先度に関するヒントです。実行準備が整っているタスクの中で、優先度の高いものより優先度の高いタスク (数値が大きいもの) を実行します:

#pragma omp task priority[優先順位]

優先順位: タスクの実行順序のヒントを提供する負でない数値のスカラー式です

この機能はバージョン 17.0.1.143 ではまだ未サポートです

明示的な SIMD プログラミング・モデル

なぜ SIMD 拡張? OpenMP* 4.0 以前

ベンダー固有の拡張機能をサポート

- プログラミング・モデル (例えば、インテル® Cilk™ Plus)
- コンパイラー・プラグマ (例えば、#pragma vector)
- 低レベルの構文 (例えば、__mm_add_pd())

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

コンパイラーが "正しい" ことをするのを信頼する必要がある

ベクトル化に影響するプログラムの要素

ループ伝搬依存

```
DO I = 2, N
   A(I) = A(I-1) + B(I)
ENDDO
```

関数呼び出し

```
for (i = 1; i < nx; i++) {
  x = x0 + i * h;
  sumx = sumx + func(x, y, xp);
}
```

ポインター・エイリアシング

```
void scale(int *a, int *b)
{
   for (int i = 0; i < 1000; i++)
        b[i] = z * a[i];
}
```

[さらに ....]

不明なループカウント

```
struct _x { int d; int bound; };
void doit(int *a, struct _x *x)
{
   for(int i = 0; i < x->bound; i++)
    a[i] = 0;
}
```

間接メモリーアクセス

```
DO i=1, N
  A(B(i)) = A(B(i)) + C(i)*D(i)
ENDDO
```

外部ループ

```
DO I = 1, MAX
  DO J = I, MAX
    D(I,J) = D(I,J) + 1;
  ENDDO
ENDDO
```

自動ベクトル化: シリアル・セマンティクスによる制限

コンパイラーは以下をチェックする:

```
for(i = 0; i < *p; i++) {
  A[i] = B[i] * C[i];
  sum = sum + A[i];
}
```

- *p はループ不変か?
- A[]、B[]、C[] はオーバーラップしているか?
- sum は、B[] および/または C[] とエイリアスされているか?
- 演算操作の順番は重要か?
- ターゲット上のベクトル演算はスカラー演算よりも高速であるか? (ヒューリスティックの評価)

自動ベクトル化は言語規則によって制限される: 意図することを表現できない

SIMD プラグマ/宣言子による明示的なベクトル・プログラミング

```
#pragma omp simd reduction(+:sum)
for(i = 0; i < *p; i++) {
  A[i] = B[i] * C[i];
  sum = sum + A[i];
}
```

プログラマーの主張:

- *p はループ不変
- A[] は、B[] および C[] とオーバーラップしない
- sum は、B[] および C[] とエイリアスされていない
- sum はリダクションされる
- コンパイラーが効率良いベクトル化のため順番を入れ替えることを許容する
- ヒューリスティックの評価が利点をもたらさなくても、ベクトル化されたコードを生成する

明示的ベクトル・プログラミングにより何を意図するかを表現できる!

プログラマーの意図: ベクトルループ中のデータ

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd
for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

- += 操作を行う 2 つの行は、互いに異なる意味を持つ
- プログラマーは、この違いを表現する必要がある
- コンパイラーは、異なるコードを生成する必要がある
- 変数 i、p、そして step は、それぞれ異なる意味を持つ

プログラマーの意図: ベクトルループ中のデータ

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

- += 操作を行う 2 つの行は、互いに異なる意味を持つ
- プログラマーは、この違いを表現する必要がある
- コンパイラーは、異なるコードを生成する必要がある
- 変数 i、p、そして step は、それぞれ異なる意味を持つ

OpenMP* SIMD ディレクティブ

Pragma SIMD:
simd 構文は、ループを SIMD ループに変換することを明示的に指示 (それぞれのループ反復は、SIMD 命令を使用して同時に実行される)

シンタックス:

#pragma omp simd [節 [,節]...]
for ループ

for ループは「標準ループ形式」でなければいけない

リダクション変数には、ランダム・アクセス・イテレーターが必要 (C++ の整数型やポインター型)

- インダクション変数のテストとデクリメントの制限
- ループを実行する前に反復回数が判明していること
- ...

OpenMP* SIMD ディレクティブの節

- **safelen(n1[,n2] ...)**
  n1, **n2**, ... 2 の累乗であること: コンパイラーは、**n1**, **n2**, ... のベクトル長で安全なベクトル化を想定できる
- **private(v1, v2, ...)**: 変数は各ループ反復でプライベート
  - **lastprivate(...)**: 最後反復の値がグローバル変数にコピーされる
- **linear(v1: ステップ1, v2: ステップ2, ...)**
  このスカラーループの各反復では、**v1** は**ステップ 1** でインクリメントされる
  そのため、ベクトルループでは**ステップ 1 * ベクトル長**になる
- **reduction(演算子: v1, v2, ...)**
  変数 **v1**、**v2**、... は、**演算子**によるリダクション変数
- **collapse(n)**: 入れ子になったループを崩して 1 つの大きなループに再構成する
- **aligned(v1:ベース, v2:ベース, ...)**: 変数 **v1**、**v2**、... がアライメントされていることを通知
  (デフォルトはアーキテクチャー固有のアライメント)

OpenMP* SIMD の例

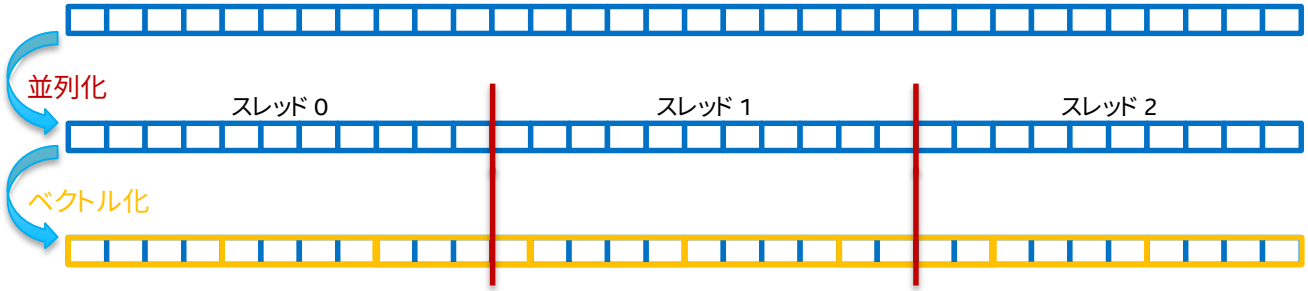
データの依存性と間接的な制御フローの依存性がないことを明示してアライメントを指示

```
void vec1(float *a, float *b, int off, int len)
{
#pragma omp simd safelen(32) aligned(a:64, b:64)
    for(int i = 0; i < len; i++)
    {
        a[i] = (a[i] > 1.0) ?
            a[i] : b[i];
            a[i + off] * b[i];
    }
}
```

```
LOOP BEGIN at simd.cpp(4,5)
   remark #15388: ベクトル化のサポート: 参照 a にアラインされたアクセスが含まれています。[ simd.cpp(6,9) ]
   remark #15388: ベクトル化のサポート: 参照 b にアラインされたアクセスが含まれています。[ simd.cpp(6,9) ]
   …
   remark #15301: OpenMP SIMD LOOP がベクトル化されました。
   …
LOOP END
```

ループ・スケジュールの SIMD 修飾子

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp parallel for simd reduction(+:sum) schedule(simd:static,5)
for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



新しい SIMD 修飾子は、コンパイラーとランタイムが SIMD レジスターのレングスにチャンクサイズを合わせることを可能にする

- 新しいチャンクサイズは、⌈*chunk_size/simdlen*⌉ * *simdlen*
- インテル® AVX2: 新しいチャンクサイズは、8 以上の 2 の累乗
- インテル® SSE: 新しいチャンクサイズは、4 以上の 2 の累乗

SIMD 対応関数

SIMD 対応関数 (以前は declare simd 構文と呼ばれていた):
SIMD ループから呼び出される関数が、SIMD命令を使用した処理を行う複数のバージョンを生成することを有効にすることを指示。[OpenMP* 4.0 の API: 2.8.2]

シンタックス:

```
#pragma omp declare simd [節 [,節]...]
  関数定義または宣言
```

目的:
スカラー計算 (カーネル) としてワークを表現し、コンパイラーにベクトルバージョンを生成させる。ベクトルサイズは移植性を考慮してコンパイル時に指定できる (インテル® SSE、インテル® AVX、インテル® AVX-512)

注意:
関数定義と関数宣言 (ヘッダーファイル) の両方で同じように指定する必要がある

SIMD 対応関数の節

- **simdlen(len)**
  len は 2 の累乗: 引数ごとに多くの要素を渡すことを可能にする (デフォルトは実装依存)
- **linear(v1: ステップ 1, v2: ステップ 2, ...)**
  引数 **v1**、**v2**、... を SIMD レーンにプライベートに定義し、ループのコンテキストで使用される場合リニアな関係を持ちます (**ステップ 1**、**ステップ 2**、...)
- **uniform(a1, a2, ...)**
  引数 **a1**、**a2**、... は、ベクトルとして扱われません (SIMD レーンに定数がブロードキャストされる)
- **inbranch, notinbranch**: SIMD 対応関数は分岐から呼び出される、または呼び出されない
- **aligned(a1:ベース, a2:ベース, ...)**: 引数 **a1**、**a2**、... がアライメントされていることを通知 (デフォルトはアーキテクチャー固有のアライメント)

OpenMP*: SIMD 対応関数のベクトル化

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {

#pragma omp parallel for simd

    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

```
vec8 min_v(vec8 a,vec8 b){
  return a < b ? a : b;
}
```

```
vec8 distsq_v(vec8 x,vec8 y) {
  return (x - y) * (x - y);
}
```

```
vd = min_v(distsq_v(va, vb), vc)
```

SIMD 対応関数: Linear/Uniform

- なぜそれらが必要なのか?
- uniform もしくは linear が省略されると、関数への引数はベクトルとして扱われる

```
#pragma omp declare simd uniform(a) linear(i:1)
void foo(float *a, int i):
```

a は、ポインター
i は、int *[i, i+1, i+2, ...]* のシーケンス
a[i] は、ユニットストライドなロード/ストア (**[v]movups**)

dec_simd2.c

```
#pragma omp declare simd
void foo(float *a, int i):
```

a は、ポインターのベクトル
i は、int のベクトル
a[i] は、スキャッター/ギャザーとなる

参考文献: http://software.intel.com/en-us/articles/usage-of-linear-and-uniform-clause-in-elemental-function-simd-enabled-function-clause

SIMD 対応関数: 呼び出しの依存性

呼ばれる側

dec_simd3.c

```
#pragma omp declare simd uniform(a),linear(i:1),simdlen(4)
void foo(int *a, int i){
     std::cout<<a[i]<<"¥n";
}
```

呼び出し側

```
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++)
        foo(a, i);
```

ベクトル化レポート

```
testmain.cc(5):(col. 13) remark: OpenMP SIMD LOOP がベクトル化されました
header.cc(3):(col. 24) remark: FUNCTION がベクトル化されました
header.cc(3):(col. 24) remark: FUNCTION がベクトル化されました
header.cc(3):(col. 24) remark: FUNCTION がベクトル化されました
header.cc(3):(col. 24) remark: FUNCTION がベクトル化されました
```

参考文献: http://software.intel.com/en-us/articles/call-site-dependence-for-elemental-functions-simd-enabled-functions-in-c

SIMD 対応関数: 呼び出しの依存性

呼ばれる側

```
#pragma omp declare simd uniform(a),linear(i:1),simdlen(4)
void foo(int *a, int i){
     std::cout<<a[i]<<"¥n";
}
```

呼び出し側

```
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++)  foo(a, i);
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++){
        k = b[i];  // k はリニアでない
        foo(a, k);
}
```

ベクトル化レポート

```
testmain.cc(14):(col. 13) remark: OpenMP SIMD LOOP がベクトル化されました
testmain.cc(21):(col. 9) remark: 関数 '?foo@@YAXPEAHH@Z' の適切なベクトルバージョンが見つかりません
testmain.cc(18):(col. 1) remark: OpenMP SIMD LOOP がベクトル化されました
header.cc(3):(col. 24) remark: FUNCTION がベクトル化されました
```

SIMD 対応関数: 複数のベクトル定義

呼ばれる側

```
#pragma omp declare simd uniform(a),linear(i:1),simdlen(4)
#pragma omp declare simd uniform(a),simdlen(4)
void foo(int *a, int i){
     std::cout<<a[i]<<"¥n";
}
```

呼び出し側

```
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++) foo(a, i);
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++){
        k = b[i];  // k はリニアでない
        foo(a, k);
}
```

ベクトル化レポート

```
testmain.cc(14):(col. 13) remark: OpenMP SIMD LOOP がベクトル化されました
testmain.cc(18):(col. 1) remark: OpenMP SIMD LOOP がベクトル化されました
header.cc(3):(col. 24) remark: FUNCTION がベクトル化されました
```

SIMD 対応関数を使用する際の制限事項

- 引数は 1 つの **uniform** または **linear** 句に記述できる
- **linear** 句に *constant-linear-step* 式が指定される場合、正の整数式でなければならない
- 関数やサブルーチンは、構造化ブロックでなければならない
- SIMD ループから呼び出される関数やサブルーチンは、OpenMP* 構造を実行することはできない
- 関数やサブルーチンの実行では、SIMD チャンクの同時反復の実行を変更する副作用があってはならない
- 関数の内側から外側へ、または外側から内側へ分岐するプログラムは不適合である
- C/C++: 関数は、*longjmp* や *setjmp* を呼び出してはならない

ボルテックス・コード: 外部ループのベクトル化

```
#pragma omp  simd   // SIMD 関数の呼び出し側での外部ループのための simd pragma
for (int i = beg*16; i < end*16; ++i)
      particleVelocity_block(px[i], py[i], pz[i],
                             destvx + i, destvy + i, destvz + i, vel_block_start, vel_block_end);

#pragama  omp declare simd linear(velx,vely,velz) uniform(start,end) aligned(velx:64, vely:64, velz:64)
static  void  particleVelocity_block(const  float  posx, const  float posy, const  float  posz,
                                     float *velx,  float *vely,  float *velz,  int start, int end) {
    for (int j = start; j < end; ++j) {
        const  float  del_p_x  = posx - px[j];
        const  float  del_p_y  = posy - py[j];
        const  float  del_p_z  = posz - pz[j];
        const  float  dxn= del_p_x * del_p_x + del_p_y * del_p_y + del_p_z * del_p_z +pa[j]* pa[j];
        const  float  dxctaui  = del_p_y * tz[j] - ty[j] * del_p_z;
        const  float  dyctaui  = del_p_z * tx[j] - tz[j] * del_p_x;
        const  float  dzctaui  = del_p_x * ty[j] - tx[j] * del_p_y;
        const  float  dst      = 1.0f/std::sqrt(dxn);
        const  float  dst3     = dst*dst*dst;
        *velx                -= dxctaui * dst3;
        *vely                -= dyctaui * dst3;
        *velz                -= dzctaui * dst3;
    }
}
```

KNC† のパフォーマンス改善
2 倍以上
内部ループから外部ループのベクトル化

†開発コード名

ベクトル化の効率を評価する

- 完全な最適化オプションでビルドして実行
- 同じオプションに以下を追加してビルド:
  /Qopenmp-simd- (-qopenmp-simd-)
- 2 つの結果を比較する

スピードアップ(S) = 実行時間(no-vec)/実行時間(vec)

- スピードアップは 1.0 以上であること。スピードアップの上限:
  - 単精度: インテル® SSE では S <= 4、インテル® AVX では S <= 8、インテル® AVX-512 では S <= 16
  - 倍精度: インテル® SSE では S <= 2、インテル® AVX では S <= 4、インテル® AVX-512 では S <= 8
  - 高い値が良い、上限を目指す

- 例外: インテル® MKL を呼び出しているコード領域は、効率良くベクトル化され、将来にわたって有効!

ベクトル化の成功を評価 1

- **アセンブラー・コードの調査:**
  アセンブリーを表示: Linux* および OS X*: **-S**、Windows®: **/Fa**
  最も信頼できる方法であり、詳細を知ることができる。
  スカラー(s)/パックド(p) もしくは VEX エンコードされた命令をチェックする: アセンブラー・リストは、対応するソースコードの行番号を含んでいる
- **VEC フェーズの最適化レポート:**
  Linux*、OS X*: **-qopt-report<n> -qopt-report-phase=vec**
  Windows®: **/Qopt-report:<n> /Qopt-report-phase:vec**
  n: **0**...**5** 詳細レベルを指定; デフォルトは **2**

ベクトル化の成功を評価 2

- **インテル® VTune™ Amplifier XE の使用**

  例えばイベントを使用

  第 2 世代、第 3 世代および第 4 世代インテル® Core™ プロセッサーでは、**FP_COMP_OPS_EXE.SSE_PACKED_[SINGLE|DOUBLE]** イベントを収集して SIMD 命令の実行を確認

ベクトル化レポート

例:

```
4 : void fd(float *y)
5 :{
6 :     int i;
7 :     for (i=2; i<10; i++) {
8 :       y[i] = y[i-1] + 1;
9 :     }
10:}
```

```
icc novec.c -opt-report3 -vec-report-phase=vec
icl novec.c /Qopt-report:3 /Qopt-report-phase:vec

novec.c(7):(col. 4) remark: ループはベクトル化されませんでした: ベクトル依存関係がベクトル化を妨げています。
詳細については、レベル 5 のレポートを使用してください。
novec.c(8):(col. 6) remark: ベクトル依存関係: FLOW の依存関係が y 行 5 と y 行 5 の間に仮定されました。
```

注意:

プロシージャー間の最適化 (-ipo や /Qipo) が有効である場合、リンクにレポートオプションを追加する

インテル® コンパイラーの最適化レポートの改善

変数名とメモリー参照レポートが大幅に改善された

- 16.0: リマーク #15346: ベクトル依存関係: ANTI の依存関係が 行 108 と 行 116 の間に仮定されました
- 17.0: リマーク #15356: ベクトル依存関係: ANTI の依存関係が *(s1)(108:2) と (r+4)(116:2) の間に仮定されました

ベクトル化できない理由をより詳細に

- 例:「例外処理の関数呼び出しはベクトル化を妨げます」

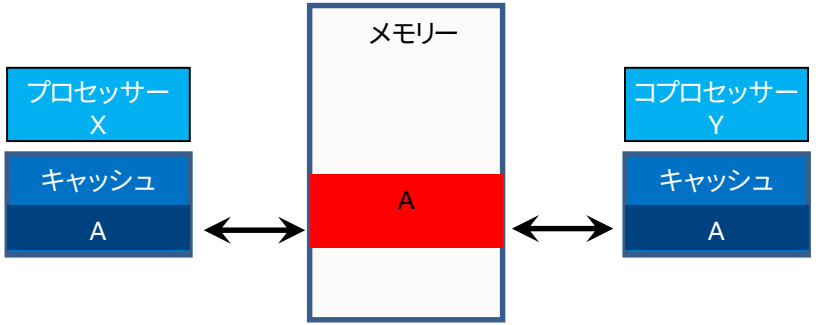
ギャザーと部分的なスカラー化の理由をレポート (-qopt-report:5)

- 16.0: リマーク #15328: ベクトル化のサポート: 変数 xyBase のギャザーはエミュレートされました: 間接アクセス [scalar_dslash_fused.cpp(334,27)]
- 17.0: リマーク #15328: ベクトル化のサポート: 変数 <xyBase[xbOffset][c][s][1]> のギャザーはエミュレートされました: 間接アクセス。インデックスの部分的な条件 [scalar_dslash_fused.cpp(334,27)]
- その他の理由:
  - メモリー読み込み
  - 非線形計算
  - 関数呼び出しの結果
  - linear だが、オーバーフローする可能性がある ← 符号なしのインデックスやアドレス計算
  - private ← 明示的なベクトル化でのメモリーのプライベート化やシリアル化された計算

GPU、コプロセッサーおよび SoC 向けの Target (または Offload) 拡張

データ共有/マッピング: 共有もしくは分散メモリー

共有メモリー



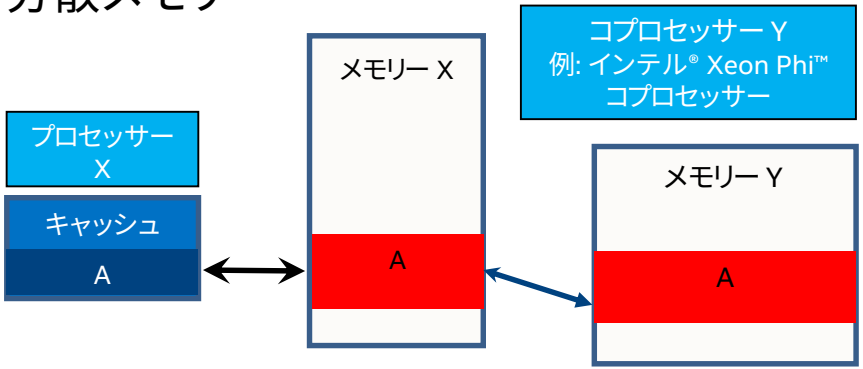
スレッドは共有メモリーへアクセスできる

- 共有データ向けに
- 各スレッドは、同期バリア間の共有メモリー (レジスター、キャッシュなど) の一時的なビューを保持できる

スレッドはプライベート・メモリーを持つ

- プライベート・データ向けに
- 各スレッドは、実行される各タスクのローカル・データ・スタックを保持できる

分散メモリー



デバイスデータ環境に対応する変数は、元の変数とストレージを共有

対応する変数への書き込みは、元の変数の値を更新する

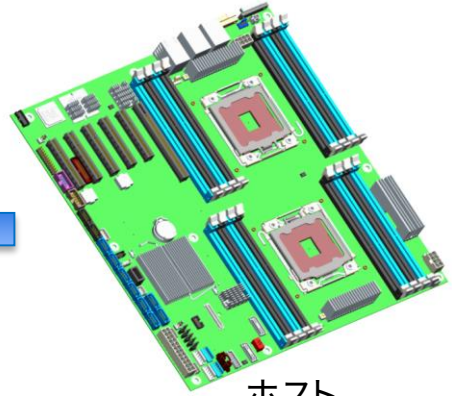
OpenMP* デバイスモデル

OpenMP* は、アクセラレーターとコプロセッサーをサポート

デバイスモデル:

- 1 つのホスト
- 同種の複数のアクセラレーター/コプロセッサー

プロセッサー/コプロセッサー/GFX



ホスト

インテル® Xeon Phi™ コプロセッサーへオフロードする際の注意点

要件:

1. インテル® MPSS を入手する
   http://www.isus.jp/hpc/software-stack-mpss/
2. インテル® コンパイラー14.0 以降を入手する
3. インテル® MPSS 環境を設定する

Windows® 環境でビルドのみ行う場合、Intel® Xeon Phi™ coprocessor essentials のみをインストール

赤字が OpenMP* 4.5 での拡張

OpenMP* 4.0/4.5 Target 拡張

ターゲットデバイス上で実行するためコードをオフロード

- omp target [節[[,] 節],...] [nowait]
  構造化ブロック
- omp declare target
  [関数定義または宣言]

ターゲットデバイスへ変数をマップ

- map ([マップタイプ修飾子][マップタイプ:] リスト)
  - マップタイプ := alloc | tofrom | to | from | release | delete
  - マップタイプ修飾子: always
- omp target [enter | exit] data [節[[,] 節],...]
  構造化ブロック
- omp target update [節[[,] 節],...]
- omp declare target
  [関数定義または宣言]

アクセラレーション向けのワークシェア

- omp teams [節[[,] 節],...]
  構造化ブロック
- omp distribute [節[[,] 節],...]
  for ループ

ランタイム・サポート・ルーチン

- void omp_set_default_device(int dev_num)
- int omp_get_default_device(void)
- int omp_get_num_devices(void);
- int omp_get_num_teams(void)
- int omp_get_team_num(void);
- Int omp_is_initial_device(void);

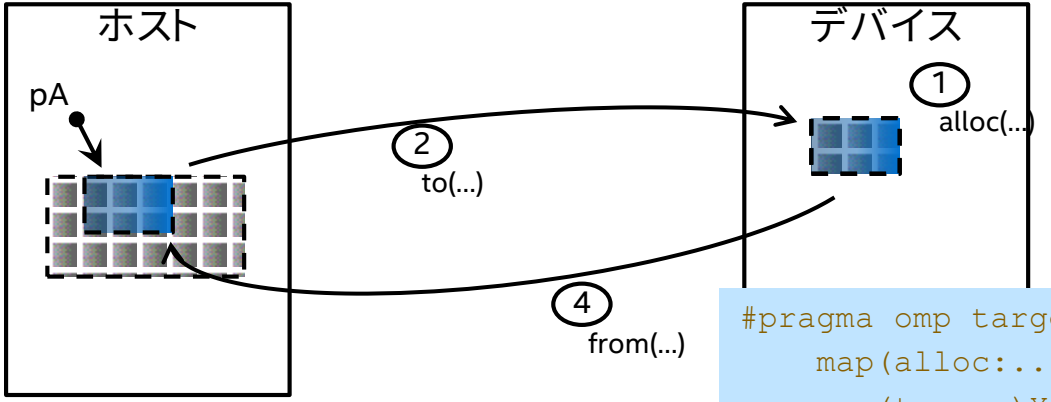
環境変数

- OMP_DEFAULT_DEVICE を介してデフォルトデバイスを制御
- 負ではない整数値

オフロードとデバイスデータのマッピング

target 構文を使用して

- ホストからターゲットデバイスへ制御を転送
- ホストとターゲットデバイスのデータ環境間で変数をマップ




```
#pragma omp target ¥
    map(alloc:...)¥
    map(to:...)¥
    map(from:...)
{ ...}
```

3

ホストスレッドはターゲット (オフロードされた) タスクをスポーン

- 同期オフロード (スレッドはターゲットタスクを待機)
- 非同期オフロード (スレッドはターゲットタスクを待機することなく継続)

map 節は、データ環境の元の変数をデバイスデータ環境の対応する変数にどのようにマップするかを決定する

例: target + map

```
#define N 1000
#pragma omp declare target
float p[N], v1[N], v2[N];
#pragma omp end declare target
extern void init(float *, float *, int);
extern void output(float *, int);
void vec_mult()
{
   int  i;
   init(v1, v2, N);
   #pragma omp target update to(v1, v2)
   #pragma omp target
   #pragma omp parallel for simd
   for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

   #pragma omp target update from(p)
   output(p, N);
}
```

グローバル変数がプログラム全体でデバイスデータ環境にマップされることを示す

ホストとデバイス間で一貫性を保つため target update を使用する

parallel for simd ループがターゲットへオフロードされることを示す

例: OpenMP* 4.0 での非同期オフロード実装

OpenMP* 4.0 の target 構文は、非同期オフロードをサポートするため既存の OpenMP* の機能 (task) を活用できる

```
#pragma omp parallel sections
{
  #pragma omp task
  {
    #pragma omp target map(in:input[:N]) map(out:result[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++) {
        result[i] = some_computation(input[i], i);
      }
  }
  #pragma omp task
  {
    do_something_important_on_host();
  }
}
```

例: OpenMP* 4.5 での非同期オフロード実装

非同期オフロードをサポートするため target 構文に nowait 節が追加された。taskwait で待機

```
#pragma omp parallel sections
{
#pragma omp target map(in:input[:N]) map(out:result[:N]) nowait
  #pragma omp parallel for
      for (i=0; i<N; i++) {
        result[i] = some_computation(input[i], i);
      }
     // 以下をホストで非同期に実行
   do_something_important_on_host();
#pragma omp taskwait
}
```

例: teams+parallel for (SAXPY – アクセラレーター向けコード)

```
int main(int argc, const char* argv[]) {
   float *x = (float*) malloc(n * sizeof(float));
   float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y
#pragma omp target data map(to:x[0:n])
  {
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) thread_limit(nthreads)
```

```
   for (int i = 0; i < n; i += num_blocks){
      for (int j = i; j < i + num_blocks; j++) {
            y[j] = a*x[j] + y[j];
  } }
  }
   free(x); free(y); return 0;
}
```

1 つ以上のループの反復を実行するスレッドチームを生成。
マスタースレッドで実行を開始

例: teams+parallel for (SAXPY – アクセラレーター向けコード)

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
   // Define scalars n, a, b & initialize x, y
#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) thread_limit(nthreads)
```

```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```

```
#pragma omp parallel for
      for (int j = i; j < i + num_blocks; j++) {
```

```
        y[j] = a*x[j] + y[j];
  } }
} free(x); free(y); return 0; }
```

1 つ以上のループの反復をすべてのスレッドチームのマスタースレッド間で共有するかどうかを指定

distribute されたスレッドのチームで、ループ反復をワークシェア

OpenMP* と OpenACC* の比較

OpenMP* 4.0/4.5 – チームとスレッド間で parallel for ループをアクセラレート

```
#pragma omp target teams map(X[0:N]) num_teams(numblocks)
#pragma omp distribute parallel for
  for (i=0; i<N; ++1) {
    X[i] += sin(X[i]);
  }
```

OpenACC* 2.0/2.5 – ギャングとワーカー間で for ループをアクセラレート

```
#pragma acc parallel copy(X[0:N]) num_gangs(numblocks)
#pragma acc loop gang worker
  for (i=0; i<N; ++i) {
    X[i] += sin(X[i]);
  }
```

GFX へオフロードする際の注意点 (Windows®)

要件:

1. インテル® C++ コンパイラー 15.0 以降を入手する
2. 最新のインテル® HD グラフィックス・ドライバーを入手する
   http://downloadcenter.intel.com
3. Binutils パッケージを入手する
   http://software.intel.com/en-us/articles/open-source-downloads
4. サンプルフォルダーにある gfx_samples.tar.gz を任意のディレクトリーに展開します。この zip ファイルは、プロセッサー・グラフィックスへの計算オフロードを行ういくつかのサンプルコードを含んでいます。それらのコードをビルドして実行できれば、環境は正しく設定されていると言えます。

http://www.isus.jp/products/psxe/getting-started-with-compute-offload-to-gfx/

GFX へオフロードする際の注意点 (Linux*)

要件:

1. インテル® C++ コンパイラー 15.0 以降を入手する
2. インテル® HD グラフィックス・ドライバー、またはオープンソース・メディア・カーネル・ランタイムを入手する
3. 「video」グループの権限を持つアカウントで、Linux* マシンにログインする
4. /usr/lib/x86_64-linux-gnu を LD_LIBRARY_PATH に追加する:
   export LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu:$LD_LIBRARY_PATH
5. リンカーがライブラリーを検索できるようにするため、/etc/ld.so.conf.d/x86_64-linux-gnu.conf ファイルに /usr/lib/x86_64-linux-gnu パスを追加する

Linux* ドライバーのバージョンによる OS サポート:
http://www.isus.jp/products/psxe/getting-started-with-compute-offload-to-gfx/

GFX へのオフロード: OpenMP* 4.0 offload への追加機能

```
bool Sobel::execute_offload()
{
        int w = COLOR_CHANNEL_NUM * image_width;
        float *outp = this->output;
        float *img = this->image;
        int iw = image_width;
        int ih = image_height;
#pragma omp target map(to: ih, iw, w) ¥
                map(tofrom: img[0:iw*ih*COLOR_CHANNEL_NUM], ¥
                            outp[0:iw*ih*COLOR_CHANNEL_NUM])
#pragma omp parallel for collapse(2)
     for (int i = 1; I < ih - 1; i++) {
          for (int k = COLOR_CHANNEL_NUM; k < (iw - 1) * COLOR_CHANNEL_NUM; k++) {
                float gx = 1 * img[k + (i - 1) * w -1 * 4]
                        + 2 * img[k + (i - 1) * w +0 * 4]
                        + 1 * img[k + (i - 1) * w +1 * 4]
                        - 1 * img[k + (i + 1) * w -1 * 4]
                        - 2 * img[k + (i + 1) * w +0 * 4]
                        - 1 * img[k + (i + 1) * w +1 * 4];
                float gy = 1 * img[k + (i - 1) * w -1 * 4]
                        - 1 * img[k + (i - 1) * w +1 * 4]
                        + 2 * img[k + (i + 0) * w -1 * 4]
                        - 2 * img[k + (i + 0) * w +1 * 4]
                        + 1 * img[k + (i + 1) * w -1 * 4]
                        - 1 * img[k + (i + 1) * w +1 * 4];
                outp[i * w + k] = sqrtf(gx * gx + gy * gy) / 2.0;
          }
     }
     return true;
}
```

利用方法:

- OpenMP* の一部の機能のみをサポート
- 配列データはポインター渡し “tofrom” を “pin” へマップ
- GFX 向けのコンパイルを指示 “-qopenmp-offload=gfx” を指定

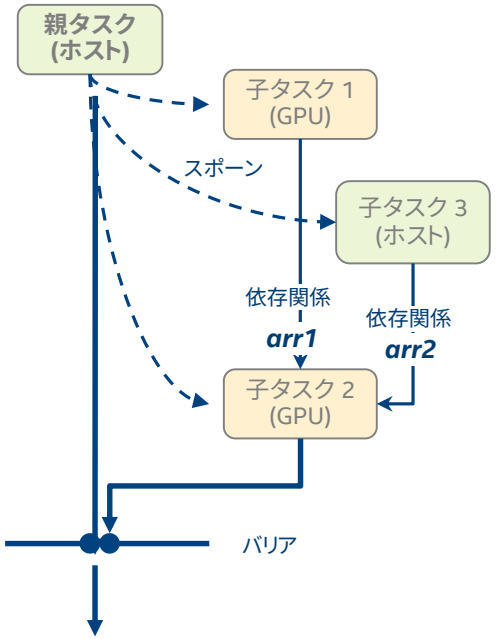
target ディレクティブで depend 節を使用して GPU へ非同期オフロードする例

```
// arr1 を初期化 – ターゲットへオフロード
#pragma omp target map(from: arr1[0:SIZE]) depend(out:arr1) nowait
#pragma omp parallel for
    for (int i = 0; i < SIZE; i++) { arr1[i] +=  i; }
```

```
// arr2 の初期化
#pragma omp task depend(out:arr2)
#pragma omp parallel for
    for (int i = 0; i < SIZE; i++) { arr2[i] += -i; }
```

```
// ターゲット上で中間結果を計算
#pragma omp target ¥
  map(to: arr1[0:SIZE], arr2[0:SIZE]) ¥
  map(from:arr3[0:SIZE]) ¥
  nowait depend(in:arr1, arr2)
#pragma omp parallel for
    for (int i = 0; i < SIZE; i++) { arr3[i] = arr1[i] + arr2[i]; }
```

```
#pragma omp taskwait
#pragma omp parallel for
    for (int i = 0; i < SIZE; i++) { res[i] += arr3[i]; }
```



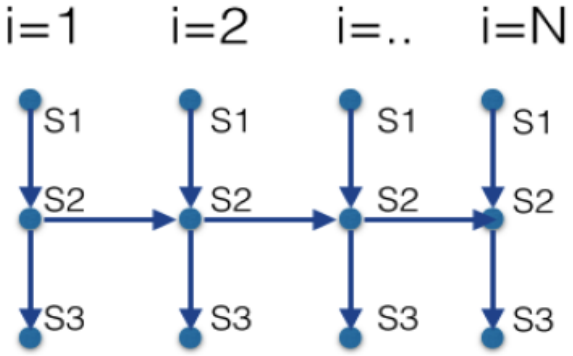
OpenMP* 4.5 “doacross” ループ

source と **sink** 依存型は、**ordered** 句を伴う入れ子のループ構造で「doacross」をサポートするため、**depend** 節に追加された

depend(source) と **depend(sink:<反復ベクトル>)**

うまく構造化された依存性を持つループが並列化可能になった

```
#pragma omp parallel for ordered
for (i=1, …; i<=N) {
  S1;
  #pragma omp ordered depend(sink:i-1)
  S2;
  #pragma omp ordered depend(source)
  S3;
}
```



OpenMP* 4.5 Offload 拡張 (pragma omp target)

OpenMP* 4.5 によるオフロード拡張への追加機能

- 主要機能は、インテル® コンパイラー 16.0 で実装済み

新機能:

- target 構文と API への新しいデバイスポインター節の追加

  #pragma omp target data ... use_device_ptr(list) ...

  void omp_target_free(void * *device_ptr*, int *device_num*);

- link 節による遅延マップ

  #pragma omp declare target ... [to] (extended-list) link (list)

- target 構文の節の多様性

  #pragma omp declare target ... private(list) firstprivate(list) if (...)

オフロードレポートの活用

このトピックは、インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーにのみ適用されます。コードのオフロード領域の実行時間と実行中のデータ転送量を測定することができます

ホストとターゲットで実行が進むと、オフロードに関する情報を含むオフロードレポートを取得することができます:

- コードのオフロード領域の実行時間
- ホストとターゲット間のデータ転送量
- デバイスの初期化と個々の変数の転送を含む詳細

次のメカニズムを使用してオフロードレポートを有効または無効にすることができます:

- **OFFLOAD_REPORT** 環境変数
- **_Offload_report** API

オフロードレポートの内容

行マーカー	説明
[状態]	オフロードの一部として実行されている処理。
[変数]	転送された変数名および転送の向き。
[CPU 時間]	ホスト上の offload プラグマの実行時間。
[MIC 時間]	ターゲット上のオフロード実行時間。 ホストとターゲット間のデータ転送時間を除く、ターゲット上の実行時間のみカウントされます。
[CPU->MIC データ]	ホストからターゲットへ転送されたデータのバイト数。
[MIC->CPU データ]	ターゲットからホストへ転送されたデータのバイト数。

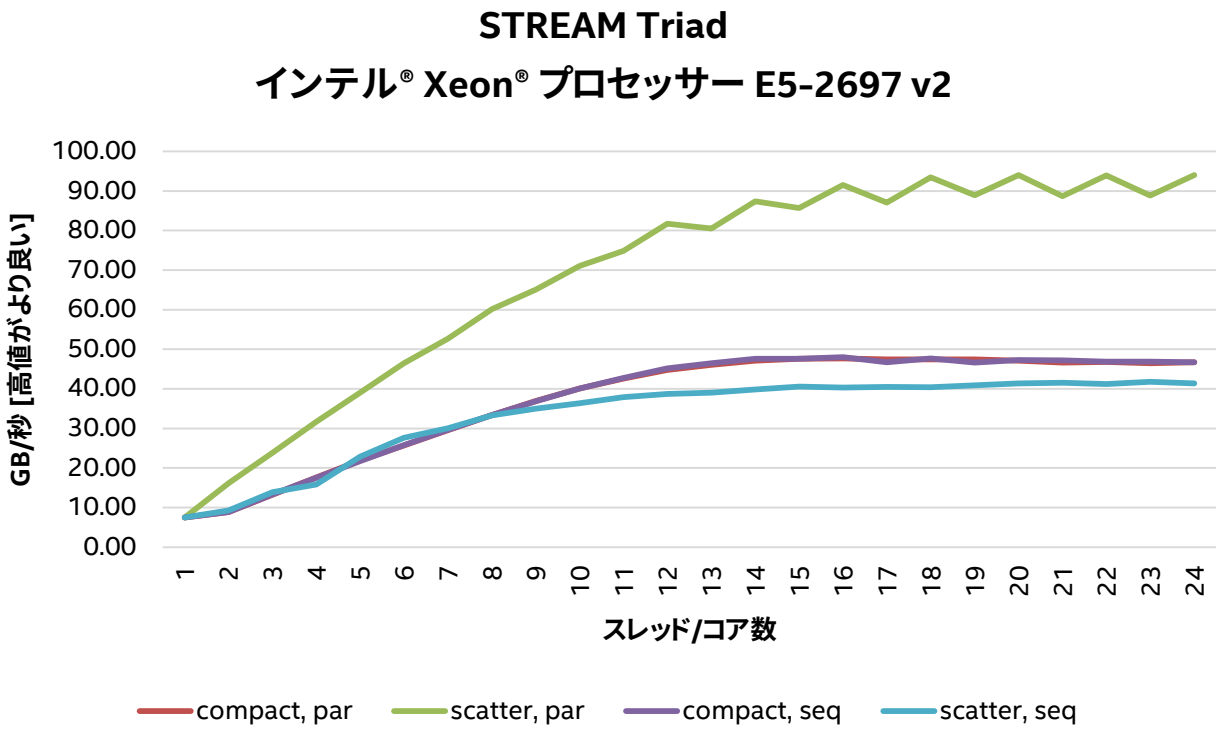
```
Intel Parallel Studio XE 2015 Composer Edition Intel(R) 64 Visual Studio 2012
C:¥Users¥kiyo¥Desktop¥code¥mkl¥mic_offload¥blasc¥source>sgemm 23000

Matrix dimension is being set to 23000

Initialize Arrays ..
Start Calc ..
[MKL] [MIC --] [AO 関数]¥tSGEMM
[MKL] [MIC --] [AO SGEMM ワークディビジョン]¥t0.60 0.40
[MKL] [MIC 00] [AO SGEMM CPU 時間]¥t26.087971 秒
[MKL] [MIC 00] [AO SGEMM MIC 時間]¥t11.141352 秒
Resulting matrix C: size = 23000
Elapsed Time 26.190581  Result = 793511488.000
C:¥Users¥kiyo¥Desktop¥code¥mkl¥mic_offload¥blasc¥source>
```

アフィニティー拡張

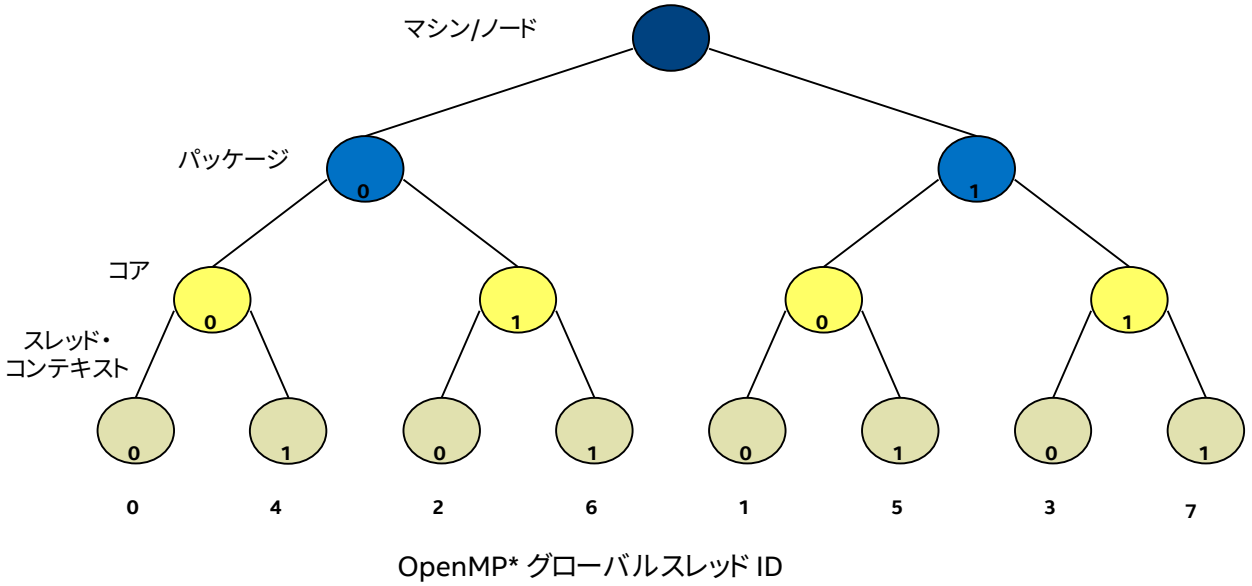
NUMA はここに属する …

- (ほとんど) すべてのマルチソケット計算サーバーは、NUMA システムである
  - 異なるメモリー位置へのアクセス・レイテンシーは一定ではない
  - 異なるメモリー位置の帯域幅が異なる可能性がある
- 例: インテル® Xeon® プロセッサー E5-2600 v2 製品ファミリー

スレッド・アフィニティー - なぜ重要なのか?



KMP_AFFINITY: 柔軟性のある OMP スレッドの割り当て
(インテル固有の実装)



サンプル: 2x2x2 マシントポロジー (2 パッケージ/ソケット、2 コア、2 論理スレッド); 次の設定で OpenMP* スレッドをバインド

KMP_AFFINITY=”granularity=fine,scatter”

KMP_PLACE_THREADS (インテル固有の実装)

- スレッドの分配を制御
  - ソケット数
  - ソケットあたりのコア数
  - コアあたりのスレッド数
- KMP_PLACE_THREADS 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。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
- 例えば、インテル® Xeon Phi™ コプロセッサーでは、各コプロセッサーの最大スレッド数は 4 であるが、コアあたり 3 スレッド以下にしてもパフォーマンスが向上することがある
- サンプル: KMP_PLACE_THREADS=2s,4c,2t

OpenMP* 4.0: 標準化されたアフィニティー

parallel 領域向けの追加された節: proc_bind (アフィニティー・タイプ)

環境変数によるアフィニティー設定制御:

- OMP_PROC_BIND
  例えば: export OMP_PROC_BIND="spread,spread,close"
- OMP_PLACES
  例えば: export OMP_PLACES="{0,1,2,3},{4,5,6,7},{8:4},{12:4}"

OMP_PLACES は実装依存で、OpenMP* では定義されない

OpenMP* の今後

インテル® コンパイラーの OpenMP* サポート状況

インテル® コンパイラー 16.0 は、ほとんどすべての OpenMP* 4.0 といくつかの OpenMP* 4.5 仕様をサポート

- OpenMP* 4.0 で確認中な物: ユーザー定義のリダクションの完全なサポート
  - 作業中: OMP PARALLEL における C/C++ の POD 型 (「plain old data」)
  - 未実装: Fortran、SIMD、および非 POD 型 (C++)

インテル® コンパイラー 17.0 は大部分の OpenMP* 4.5 仕様をサポート

- ロックとクリティカル領域のトランザクショナル・メモリー・サポートを除く
- 2016 年 9 月リリース

インテル® コンパイラー 18.0 : 計画中 .....

- 完全な OpenMP* 4.5 仕様のサポート
- リダクションでインテル® Cilk™ Plus の配列セクションを許可
- ...

OpenMP* 4.5 ロックのヒント

現代のプロセッサー・アーキテクチャーの中には、トランザクショナル・メモリーをサポートするものがある。例: インテル® TSX (インテル® トランザクショナル・シンクロナイゼーション・エクステンション)

- この機能が同期 (ロック) を最適に実行するかどうかは、競合条件に依存
- 開発者は、コンパイラーの実装へこの情報を渡す意味を知っている必要がある

OpenMP* 4.5 では、これに相当する拡張を提供:

- 追加された OpenMP* API/ランタイムルーチン:

```
omp_init_lock_with_hint(omp_lock_t *lock,
                        omp_lock_hint_t hint)
omp_init_nest_lock_with_hint(omp_nest_lock_t *lock,
                        omp_lock_hint_t hint)
```

サポートされるヒント:

```
omp_lock_hint_none
omp_lock_hint_uncontended
omp_lock_hint_contended
omp_lock_hint_nonspeculative
omp_lock_hint_speculative
```

- critical 構文の新たな節 **hint(type)**: “type” は、新しいロック API と同じ値を指定可能

OpenMP* 4.x 機能のまとめ

OpenMP* 4.x 機能のまとめ

- OpenMP* 4.5 はバグフィックスのためのリリースではない
- OpenMP* 4.0 で実装されていなかった重要な機能を追加

OpenMP* 5.0 は仕様開発作業を開始

- Supercomputing 2018 でリリースされる見込み
- インテルでは Supercomputing 2017 までに実装を検討中
- 計画されている機能

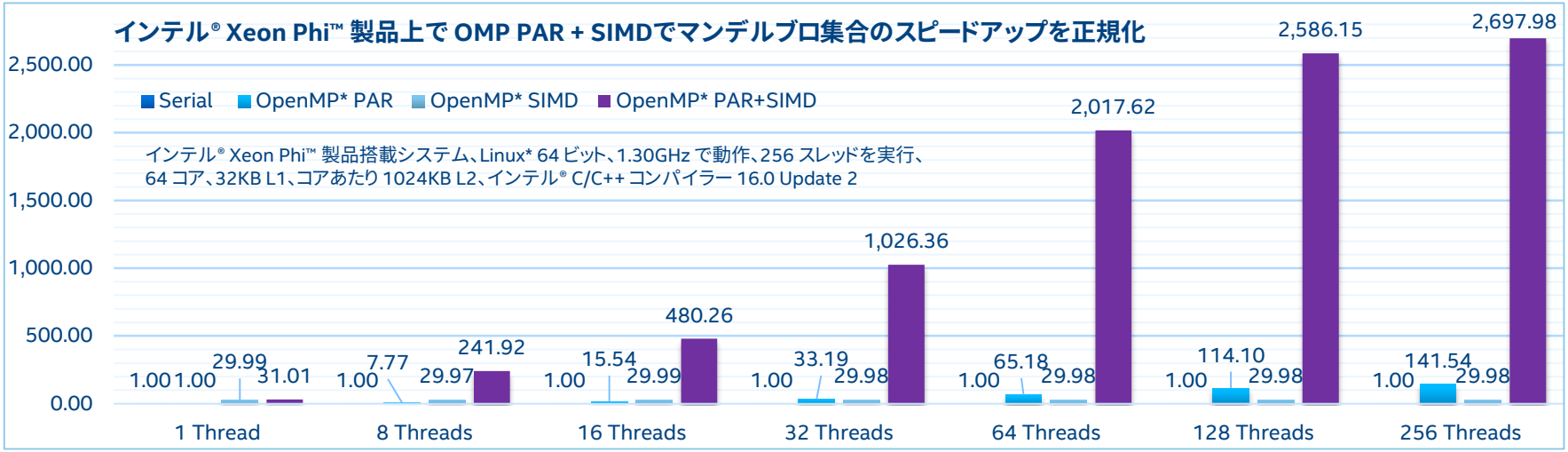
→ バグフィックス ☺
→ フューチャー
→ エラー処理
→ トランザクショナル・メモリー
→ タスク処理の拡張
→ Fortran 2008 サポート
→ C++1x のサポート
→ データの局所性とアフィニティー

OpenMP* の明示的な Target、Parellel および SIMD プログラミングは、「ハードウェアのパフォーマンス」を最大限に引き出す道を開く

マンデルブロ: インテル® Xeon Phi™ 製品上で ~2698x スピードアップ 素晴らしくないですか?

```
#pragma omp declare simd uniform(max_iter),
simdlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{   uint32_t count = 1; fcomplex z = c;
    while ((cabsf(z) < 2.0f) && (count < max_iter)) {
       z = z * z + c; count++;
    }
    return count;
}
```

```
#pragma omp parallel for schedule(guided)
for (int32_t y = 0; y < ImageHeight; ++y) {
  float c_im = max_imag - y * imag_factor;
  #pragma omp simd simdlen(32)
  for (int32_t x = 0; x < ImageWidth; ++x) {
    fcomplex in_vals_tmp = (min_real + x * real_factor) + (c_im
* 1.0iF);
    count[y][x] = mandel(in_vals_tmp, max_iter);
  }
}
```



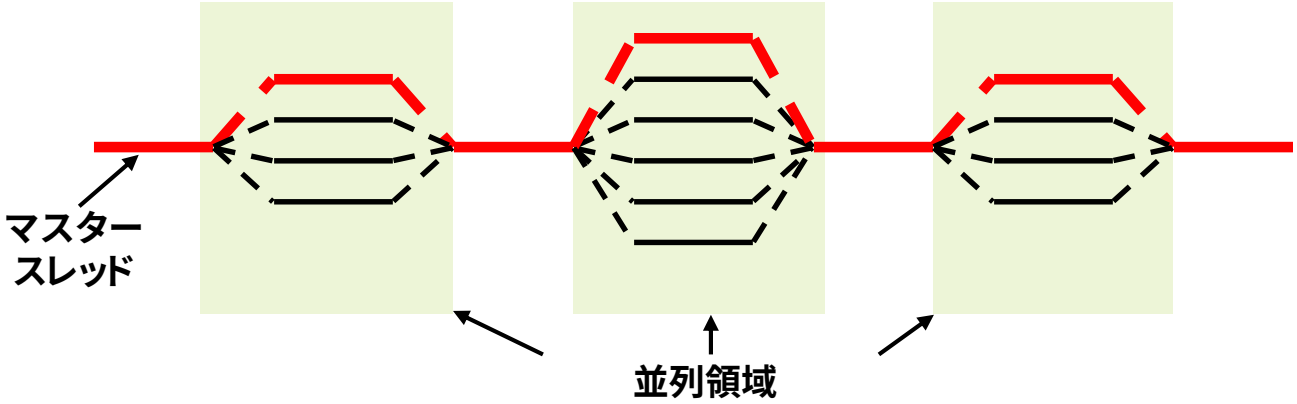
本日の内容

- 背景 (OpenMP* とは、歴史、機能概要)
- OpenMP* 4.0 と 4.5 の新しい機能
- インテル® VTune™ Amplifier XE による OpenMP* のパフォーマンス解析

OpenMP* プログラミング・モデル

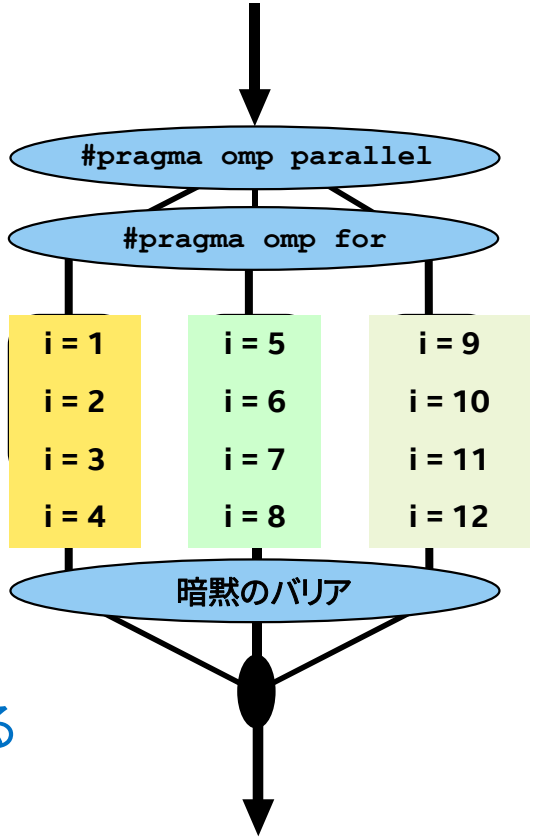
フォークジョイン並列:

- マスタースレッドが必要に応じてスレッドのチームをスポーンする
- OpenMP* チーム := マスター + ワーカー
- 並列処理を段階的に追加: シリアルプログラムが並列プログラムに発展する



OpenMP* 並列領域の例

```
// N=12 を想定
#pragma omp parallel
#pragma omp for
   for(i = 1, i < N+1, i++)
      c[i] = a[i] + b[i];
```



- スレッドには独立したループ反復が割り当てられる
- スレッドはワークシェア構文の最後で待機

インテル® VTune™ Amplifier XE/OpenMP* 解析

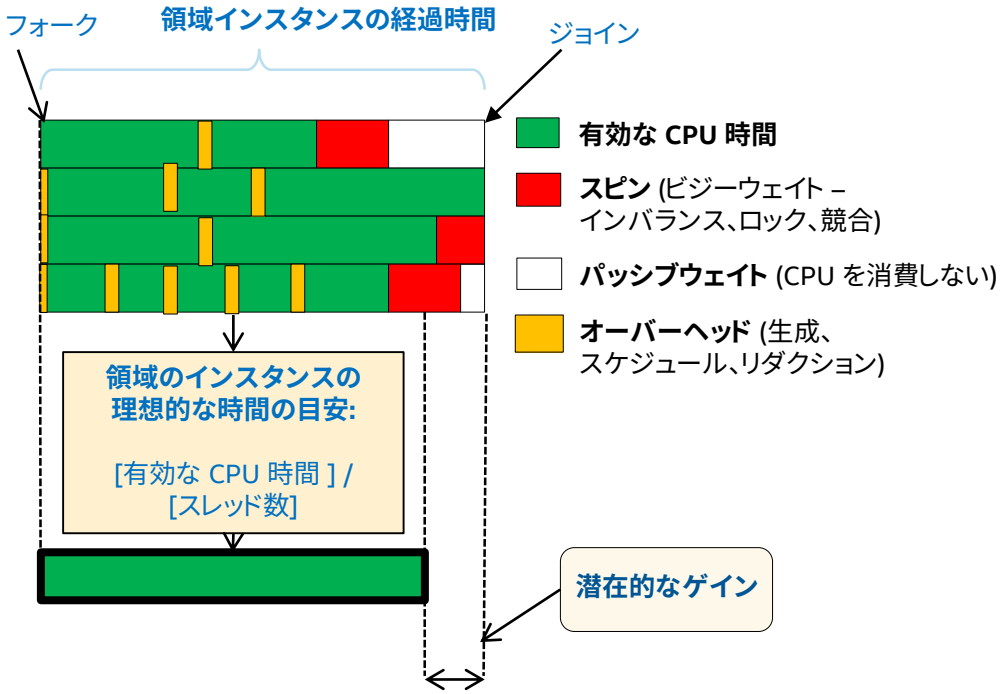
次の問いに答えるメトリックと OpenMP* 解析の強化:

- **アプリケーションのシリアル時間はスケーリングに影響するほど長いか?**
- **OpenMP* 並列領域の効率は?**
- **ロード・インバランスとオーバーヘッドを軽減したらどれくらいのゲインを得られるか?**
- **投資効果がより見込める領域は?**

メトリックは経過時間に基づく ➡ アプリケーション・ウォールクロック時間に直結する改善の可能性

インテル® VTune™ Amplifier XE パフォーマンス解析

- シリアル時間: パフォーマンス・データの収集中にマスタースレッドが OpenMP* 並列領域外でアプリケーションによって費やされた時間:
  経過時間 - ∑[すべての並列領域の経過時間]
- **並列領域のインスタンスの有効な CPU 時間:**
  ([CPU 時間] – [スピン時間] – [オーバーヘッド時間])
  領域のインスタンスのスレッドで集約された、CPU、スピン、およびオーバーヘッド時間
- **領域のインスタンスの理想的な時間の目安:**
  [有効な CPU 時間] / [スレッド数]
- **並列領域のインスタンスの潜在的なゲイン:**
  [領域インスタンスの経過時間] – [領域インスタンスの理想的な時間の目安]
- **領域の潜在的なゲイン:** ∑[領域の全インスタンスの潜在的なゲイン]
- **プログラムの潜在的なゲイン:** ∑[全領域の潜在的なゲイン]

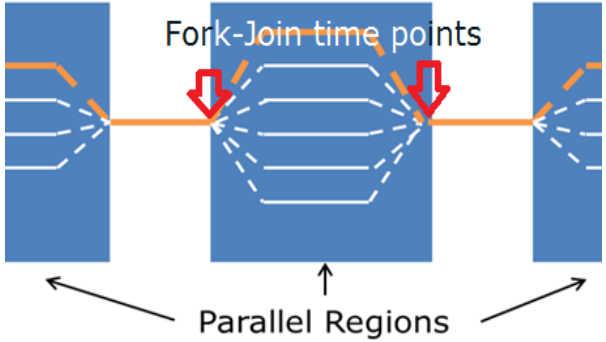


インテル® VTune™ Amplifier XE/OpenMP* 解析

- OpenMP* のトレースは領域/ワークシェア・コンテキストを解析するために使用される

  - インテル® OpenMP* ランタイムからインテル® VTune™ Amplifier XE に提供される:
    - **並列領域とワークスレッドのフォークジョイン時間ポイント**

  - トレースのオーバーヘッドは、領域のフォークジョイン・ポイントで領域のインスタンスごとに生じる



- サンプリングはオーバーヘッド、同期スピンなどの種類を特定

  インテル® VTune™ Amplifier XE の解析タイプは、CPU 時間の計算をサポート (hotspot、スタックトレースあり/なしの高度な hotspot)

インテル® VTune™ Amplifier XE/OpenMP* 解析

メトリックのサマリー

Advanced Hotspots Hotspots viewpoint (change) Intel VTune Amplifier XE 2015

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

CPU Frequency Ratio:	1.111
Paused Time:	0s
Overhead Time:	0.005s
Spin Time:	371.709s

A significant portion of CPU time is spent waiting. Use this metric to discover which synchronizations are spinning. Consider adjusting spin wait parameters, changing the lock implementation (for example, by backing off then descheduling), or adjusting the synchronization granularity.

OpenMP Analysis. Application Elapsed Time: 69.073

Serial Time (outside any parallel region):	0.038s
Parallel Region Elapsed Time:	**69.035s**
Estimated Ideal Time:	53.116s
Potential Gain (Elapsed Time):	15.919s

アプリケーションのシリアル時間はスケーリングを制限するか?

理想的な並列実行に対してアプリケーションはどれくらい効率的か?

インバランス/オーバーヘッドのチューニングへの投資からどれくらい潜在的なゲインを得られるか?

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	Potential Gain (Elapsed Time)	Elapsed Time	Instance Count
y_solve_ompparallel:24@unknown:42:395	4.584s	16.885s	201
z_solve_ompparallel:24@unknown:42:408	4.385s	17.333s	201
x_solve_ompparallel:24@unknown:44:396	4.050s	14.723s	201
compute_rhs_ompparallel:24@unknown:17:426	2.675s	18.278s	202
add_ompparallel:24@unknown:18:27	0.147s	1.558s	201
[Others]	0.079s	0.257s	6

どの領域が投資に対してよりゲインがあるか?

リンクをクリックしてグリッドビューでさらに詳しい情報を表示

インテル® VTune™ Amplifier XE/OpenMP* 解析

メトリックのグリッド

CPU 時間階層の改善

Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	Potential Gain	Potential Gain (% of Collection Time)	Elaps... Time	Number of OpenMP threads	Inst... Cou.	CPU Time: Effective Time by Utilizati... (Idle, Poor, Ok, Idea)	Spin Time: Imbalance...	Spin Time: Lock Cont...	Spin Time: Other	Overhead Time: Creation ...	Overhead Time: Scheduling...	Overhead Time: Red.	Overhead Time: Other	Instructions Retired	CPI Rate
_mainompparallel:4@unknown:89:96	5.649s	31.2%	7.139s	4	1	5.961s	5.400s	6.220s	2.338s	0s	0s	0s	0.027s	29,302,460,000	1.90
_mainompparallel:4@unknown:53:56	1.268s	7.0%	5.950s	4	1	18.729s	0.787s	0s	0.152s	0s	0s	0s	0.030s	45,147,850,000	1.21
_mainompparallel:4@unknown:36:39	0.276s	1.5%	1.674s	4	1	5.591s	0.012s	0s	0.005s	0s	0s	0s	0.013s	12,156,650,000	1.28
[Serial - outside any region]		0.0%	2.647s			3.915s	0.614s	0s	0.287s	0s	0s	0s	0.003s	12,989,020,000	1.05
_mainompparallel:4@unknown:71:74	0.164s	0.9%	0.690s	4	1	2.106s	0s	0s	0s	0s	0.513s	0s	0s	5,135,900,000	1.42

potential gain (潜在的なゲイン) メトリックは、時間を消費する領域の上位に焦点を当てず、**チューニングから最大限の結果を得られる領域に焦点を当てているため**、CPU や経過時間よりも重要

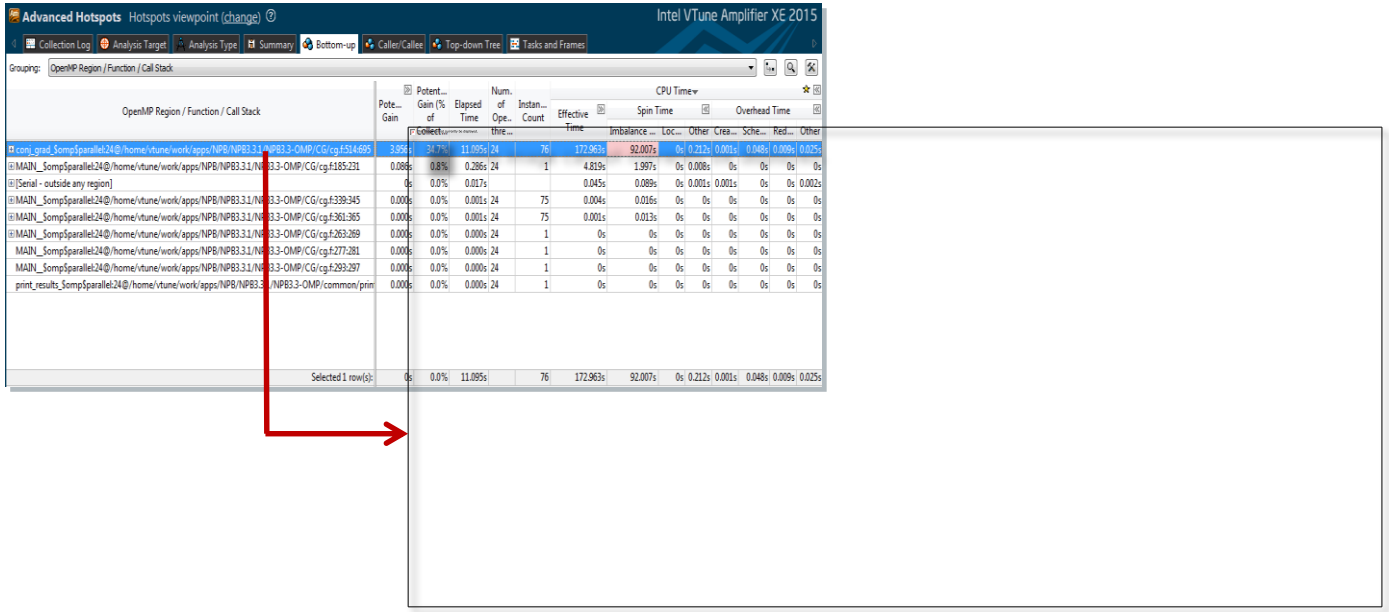
OpenMP* 領域はタイムライン・ペインに示される

Thread: OMP Worker Thre, OMP Worker Thre, OMP Master Threa, OMP Worker Thre

Region Instance
Start: 3.679s Duration: 5.95s
OpenMP Region: _mainompparallel:4@unknown:53:56
OpenMP Region Type: Good

インテル® VTune™ Amplifier XE/OpenMP* 解析

「Grouping」グリッドの各行から領域のソースへドリルダウン



インテル® VTune™ Amplifier XE/OpenMP* 解析

OpenMP* 解析データの解釈

シリアルコードのシナリオ

Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: OpenMP Region / Thread / Function / Call Stack

OpenMP Region / Thread / Function / Call Stack	Potential Gain	Potential Gain (% of Collection Time)	Elapsed Time	Number of OpenMP threads	Instance Count	CPU Time: Effective Time	Spin Time: Imbalance...	Spin Time: Lock Contention...	Spin Time: Other	Overhead Time: Creation ...	Overhead Time: Scheduling...	Overhead Time: Red.	Overhead Time: Ot.
_mainompparallel:4@unknown:89:96	5.461s	33.1%	6.809s	4	1	5.389s	5.044s	5.835s	2.188s	0s	0s	0s	0.0.
_mainompparallel:4@unknown:71:74	0.150s	0.9%	0.538s	4	1	1.551s	0s	0s	0s	0s	0.523s	0s	0s
_mainompparallel:4@unknown:53:56	1.061s	6.4%	5.561s	4	1	18.003s	0.628s	0s	0.117s	0s	0s	0s	0.0.
_mainompparallel:4@unknown:36:39	0.121s	0.7%	1.445s	4	1	5.299s	0s	0s	0.001s	0s	0s	0s	0.0.
[Serial - outside any region]		0.0%	2.144s			3.292s	0.554s	0s	0.239s	0s	0s	0s	0s
OMP Worker Thread #3 (TID: 4892)		0.0%				0.482s	0.216s	0s	0.080s	0s	0s	0s	0s
OMP Worker Thread #2 (TID: 7508)		0.0%				0.374s	0.160s	0s	0.075s	0s	0s	0s	0s
OMP Worker Thread #1 (TID: 5332)		0.0%				0.393s	0.178s	0s	0.084s	0s	0s	0s	0s
OMP Master Thread #0 (TID: 5212)		0.0%						0s	0s	0s	0s	0s	0s
Wow64SystemServiceEx		0.0%						0s	0s	0s	0s	0s	0s
RtlSetEnvironmentVar		0.0%						0s	0s	0s	0s	0s	0s
RtlCompareUnicodeStrings		0.0%						0s	0s	0s	0s	0s	0s
Selected 1 row(s):		0.0%						0s	0.239s	0s	0s	0s	0s

マスタースレッドが計算中に、ほかの OMP ワーカースレッドはスピンして待機

Thread: OMP Worker Thre / OMP Worker Thre / OMP Master Threa / OMP Worker Thre

- より並列性を高める、またはシリアル実行が避けられないように見えるアルゴリズムやマイクロアーキテクチャーをチューニングすることで、シリアル実行領域を最小化する方法を探す
- コア数の多いマシンのシリアル領域は、潜在的なスケーリングに多大な影響を与えるため、可能な限り最小化すべき

インテル® VTune™ Amplifier XE/OpenMP* 解析

OpenMP* 解析データの解釈

同期オブジェクトと待機時間のシナリオ

omptest.cpp r002ah

Advanced Hotspots Hotspots viewpoint (change) Intel VTune Amplifier XE 2015

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: OpenMP Region / Thread / Function / Call Stack

OpenMP Region / Thread / Function / Call Stack	Potential Gain	Potential Gain (% of Collection Time)	Elaps... Time	Number of OpenMP threads	Instance Count	CPU Time: Effective Time	Spin Time: Imbalance or Serial Spinning (OpenMP)	Spin Time: Lock Contention (OpenMP)	Spin Time: Other	Overhead Time: Creation ...	Overhead Time: Scheduling ...	Overhead Time: Red.
_mainompparallel:4@unknown:53:56	1.061s	6.4%	5.561s	4	1	18.003s	0.628s	0s	0.117s	0s	0s	0s
_mainompparallel:4@unknown:89:96	5.461s	33.1%	6.809s	4	1	5.389s	5.044s	5.835s	2.188s	0s	0s	0s
_mainompparallel:4@unknown:36:39	0.121s	0.7%	1.445s	4	1	5.299s	0s	0s	0.001s	0s	0s	0s
[Serial - outside any region]		0.0%	2.144s			3.292s	0.554s	0s	0.239s	0s	0s	0s
_mainompparallel:4@unknown:71:74	0.150s	0.9%	0.538s	4	1	1.551s	0s	0s	0s	0s	0.523s	0s
Selected 1 row(s):		0.0%	6.809s		1	5.389s	5.044s	5.835s	2.188s	0s	0s	0s

大きな潜在的なゲイン

スピン時間: ロード・インバランスとロック競合の両方

OMP Worker Thread #3 (TID: 4892)
OMP Worker Thread #2 (TID: 7508)
OMP Master Thread #0 (TID: 5212)
OMP Worker Thread #1 (TID: 5332)

Region Instance
Start: 9.498s Duration: 6.809s
OpenMP Region: _mainompparallel:4@unknown:89:96
OpenMP Region Type: Good

Ruler Area | Region In... | Thread | Running | CPU Time

- 可能であれば OpenMP* のリダクション、omp atomic 構文、またはスレッド・ローカル・ストレージを使用して、領域内での同期を排除することを検討する
- 問題の原因となる特定の同期オブジェクトを検出するには、“Locks and Waits (ロックとウェイト)” 解析を行う

インテル® VTune™ Amplifier XE/OpenMP* 解析

OpenMP* 解析データの解釈

ロード・インバランスのシナリオ

Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: OpenMP Region / Thread / Function / Call Stack

OpenMP Region / Thread / Function / Call Stack	Potential Gain	Potential Gain (% of Collection Time)	Elapsed Time	Number of OpenMP threads	Instance Count	CPU Time: Effective Time	Spin Time: Imbalance...	Spin Time: Lock Contention...	Spin Time: Other	Overhead Time: Creation ...	Overhead Time: Scheduling...	Overhead Time: Red.	Overhead Time: Ot.	Instructions Retired
_mainompparallel:4@unknown:89:96	5.461s	33.1%	6.809s	4	1	5.389s	5.044s	5.835s	2.188s	0s	0s	0s	0.0.	27,367,010,000
_mainompparallel:4@unknown:71:74	0.150s	0.9%	0.538s	4	1	1.551s	0s	0s	0s	0s	0.523s	0s	0s	3,726,690,000
_mainompparallel:4@unknown:53:56	1.061s	6.4%	5.561s	4	1	18.003s	0.628s	0s	0.117s	0s	0s	0s	0.0.	28,115,890,000
_mainompparallel:4@unknown:36:39	0.121s	0.7%	1.445s	4	1	5.299s	0s	0s	0.001s	0s	0s	0s	0.0.	7,253,510,000
[Serial - outside any region]		0.0%	2.144s			3.292s	0.554s	0s	0.239s	0s	0s	0s	0s	8,903,070,000
Selected 1 row(s):		0.0%	5.561s		1	18.003s	0.628s	0s	0.117s	0s	0s	0s	0.0.	28,115,890,000

重要な潜在的ゲインに注目 - 全体で 5.561 秒から 1.061 秒のゲイン!!
改善の余地があることを意味する

Thread: OMP Worker Thre, OMP Worker Thre, OMP Master Threa, OMP Worker Thre

Region Instance
Start: 3.092s Duration: 5.561s
OpenMP Region: _mainompparallel:4@unknown:53:56
OpenMP Region Type: Good

CPU Time

インテル® VTune™ Amplifier XE/OpenMP* 解析

OpenMP* 解析データの解釈

バランスのとれた並列領域のシナリオ

Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: OpenMP Region / Thread / Function / Call Stack

OpenMP Region / Thread / Function / Call Stack	Potential Gain	Potential Gain (% of Collection Time)	Elaps... Time	Number of OpenMP threads	Instance Count	CPU Time: Effective Time	Spin Time: Imbalance...	Spin Time: Lock Contention...	Spin Time: Other	Overhead Time: Creation ...	Overhead Time: Scheduling...	Overhead Time: Red.	Overhead Time: Ot.	Instructions Retired	CPI Rate	CPU Freq Rati
_mainompparallel:4@unknown:53:56	1.061s	6.4%	5.561s	4	1	18.003s	0.628s	0s	0.117s	0s	0s	0s	0.0.	28,115,890,000	1.863	1.10
_mainompparallel:4@unknown:89:96	5.461s	33.1%	6.809s	4	1	5.389s	5.044s	5.835s	2.188s	0s	0s	0s	0.0.	27,367,010,000	1.874	1.09
_mainompparallel:4@unknown:36:39	0.121s	0.7%	1.445s	4	1	5.299s	0s	0s	0.001s	0s	0s	0s	0.0.	7,253,510,000	2.047	1.10
[Serial - outside any region]		0.0%	2.144s			3.292s	0.554s	0s	0.239s	0s	0s	0s	0s	8,903,070,000	1.304	1.12
_mainompparallel:4@unknown:71:74	0.150s	0.9%	0.538s	4	1	1.551s	0s	0s	0s	0s	0.523s	0s	0s	3,726,690,000	1.555	1.10
s):		0.0%	1.445s		1	5.299s	0s	0s	0.001s	0s	0s	0s	0.0.	7,253,510,000	2.047	1.10

すべてのスレッドがビジーで、オーバーヘッドやスピンがない(赤色以外)

潜在的なゲインは小さい

CPU 時間のほとんどは効果的

1.257s 2s 3s 4s 5s 6s 7s 8s 9s 10s 11s 12s 13s 14s 15s 16s

OMP Worker Thread #3 (TID: 4892)
OMP Worker Thread #2 (TID: 7508)
OMP Master Thread #0 (TID: 5212)
OMP Worker Thread #1 (TID: 5332)

Region Instance
Start: 0.171s Duration: 1.445s
OpenMP Region: _mainompparallel:4@unknown:36:39
OpenMP Region Type: Good

これはすべてが完璧であることを意味するものではない。OpenMP* スレッドベースの並列処理とロードバランスに問題はないが、マイクロアーキテクチャーの問題などは潜在する可能性がある

関連情報

OpenMP* に関する情報

OpenMP* SIMD によるベクトル・プログラミング

– http://jp.xlsoft.com/documents/intel/catalog/Intel_ParallelUniverse_Issue22_JPN.pdf

OpenMP* API バージョン 4.5: 標準化の進化

– http://www.isus.jp/products/c-compilers/pu24-openmp-api-version-4-5/

Knights Corner† から Knights Landing† へ:
次世代のインテル® Xeon Phi™ プロセッサー/コプロセッサーに備える

– http://jp.xlsoft.com/documents/intel/catalog/Intel_ParallelUniverse_Issue20_JPN.pdf

オンライン・トレーニング

– http://www.isus.jp/online-training/

コードモダン化のリンク

モダンコード開発者コミュニティー

– http://www.isus.jp/modern-code/

インテル® Code Modernization Enablement Program

– software.intel.com/code-modernization-enablement (英語)

インテル® Parallel Computing Centers

– software.intel.com/ipcc (英語)

技術ウェビナーシリーズ

– http://bit.ly/spring16-tech-webinars (英語)

インテル® Parallel Universe マガジン

– http://www.isus.jp/products/psxe/issue/

† 開発コード名

参考資料

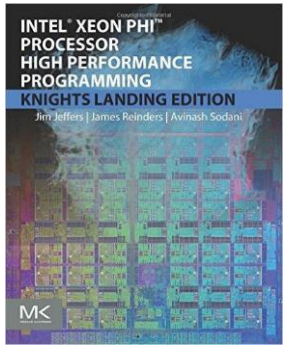
- iSUS の OpenMP* ページ
- インテルのベクトル化ツール
- マルチスレッド・アプリケーション開発のためのガイド
- インテル® C/C++ コンパイラー
- インテル® Fortran コンパイラー
- OpenMP.org

オンライン・トレーニング・コース:

- インテル® コンパイラーによる OpenMP* 入門 (8 回)
- OpenMP* 4.x による新しいレベルの並列化 (2 回)

関連書籍

アプリケーションの並列化がさまざまな分野で求められる昨今、開発者に求められる並列プログラミングのスキルは科学技術計算の分野のみならず、一般のコンシューマー・アプリケーションや自社開発のアプリケーションでも求められるようになっています。
Windows® や Linux* をはじめとする近年のオペレーティング・システムは、マルチプロセスおよびマルチスレッドのプログラミング環境をサポートしていますが、残念ながらそれらは統一されたものではありません。開発者はそれぞれの環境ごとに API などの使い方を学習し、OS ごとに異なるプログラムを開発しなければいけません。マルチスレッド環境以前にも皆さんは GUI のプログラミングなどで同じことを経験していることでしょう。プログラムがさらに複雑になるのは大変なことです。
異なる OS 環境で同じプログラミング手法が利用できればそれに越したことはありません。OpenMP* はそのような開発環境を提供する優れたプログラミング手法であり、スレッド化に際しプログラムの構造を変えることなく容易にマルチスレッド機能を実装することができます。本書は C/C++ プログラマー向けの OpenMP* 入門書として書かれています。皆さんが利用しているコンパイラーはすでに OpenMP* をサポートしているかもしれません。ぜひこの機会に OpenMP* プログラミングを始めてください。
2009 年 7 月に本書初版が出版された後、2011 年 7 月に OpenMP* 3.1 の仕様が公開されました。本書では第 2 版 (増補版) として付録 F に新しい仕様の説明を加えるとともに、一部改訂しています。



本書は、Knights Landing† に含まれる多くの拡張機能を最大限に活用する並列プログラミング手法に焦点を当てています。以前の書籍では、第 1 世代のインテル® Xeon Phi™ 製品 (開発コード名: Knights Corner) について紹介しましたが、その後、多くのことが起こっています。Knights Corner† で得た多くの経験を、利用可能になった第 2 世代インテル® Xeon Phi™ 製品に活用できることに興奮を覚えます。これらの経験は、一貫してすべてのインテル® Xeon Phi™ 製品の心臓部であるインテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーの「デュアル・チューニング (二重のチューニング)」という価値を高めるのに役立ちました。インテル® Xeon Phi™ 製品のプログラミング向けの「デュアル・チューニング (二重のチューニング)」の特徴は、ソフトウェア開発者が「インテル® Xeon Phi™ 製品が登場するまで、私たちは、インテル® Xeon® プロセッサーの素晴らしさを実感していませんでした」、そして「インテル® Xeon Phi™ 製品向けのチューニングは、常にインテル® Xeon® プロセッサーでも実質的な利益をもたらします」とコメントしたように、彼らを奮い立たせました。これこそが、マルチコア・プログラミングにおいて並列性を活用するため、コードを変更するモチベーションとなっていることを示しています。このような作業は、メニーコアとマルチコア・プロセッサーで同じコードを使用してパフォーマンスを向上することで恩恵を受けます。そのため、「デュアル・チューニング(二重のチューニング)」と呼びます。この一連の作業は、「コード・モダニゼーション (コードの現代化)」と呼ばれています。

絶賛翻訳中!

† 開発コード名

法務上の注意書きと最適化に関する注意事項

本資料の情報は、現状のまま提供され、本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスも許諾するものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責任を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的財産権の侵害への保証を含む) をするものではありません。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。

© 2016 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Core、Xeon、Intel Xeon Phi、Cilk、VTune は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。
Microsoft、Visual C++、Visual Studio、および Windows は、米国 Microsoft Corporation の、米国およびその他の国における登録商標または商標です。
*その他の社名、製品名などは、一般に各社の商標または登録商標です。

最適化に関する注意事項

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

改訂 #20110804

バックアップ

OpenMP* 4.5 拡張

OpenMP* 4.5 の機能

- 非構造化データ割り当て
  omp target [enter | exit] data
- 非同期オフロード
  omp task nowaint
- 依存性 (シグナル)
  omp task depend
- map 指示節の拡張
  always と delete 修飾子

OpenMP* 4.5 拡張
1. 非構造化データ割り当て

***#pragma omp target enter data** map(...)*

- ターゲットのデータ領域を開始します。map 節は、ターゲットデバイスに割り当てを行い、ホストから移動するデータを指定します
- map タイプには、alloc もしくは to を指定できます

***#pragma omp target exit data** map(...)*

- ターゲットのデータ領域を終了します。map 節は、ターゲットから廃棄し、ホストへ移動するデータを指定します
- map タイプには、from、release および delete を指定できます

```
#pragma omp target enter data map( alloc : x, a[0:1] ) map( to : b) map( always, to : c )
…
foo(); // omp target を含む
…
#pragma omp target exit data map(release : x, a[0:1] ) map( from : b ) map( always, from : c )
```

OpenMP* 4.5 拡張
2. 非同期オフロード

- 非同期オフロードは、既存のタスクモデルと nowait 節で使用する
- 既存のタスクを表現するプラグマ:

  #pragma omp task [節 [[,] 節] ...]
  　構造化ブロック

  節: nowait

- 利用例:
  - #pragma omp target device map(...)　nowait (リスト) if(...)
  - #pragma omp target enter data map(...)　nowait (リスト) if(...)
  - #pragma omp target exit data map(...)　nowait (リスト) if(...)
  - #pragma omp target update to/from　nowait (リスト) if(...)

OpenMP* 4.5 拡張
2. 非同期オフロード

- target と parallel の統合

```
#pragma omp target nowait map( tofrom : b )
    #pragma omp parallel for
        for (…)
```

以下に相当:

```
#pragma omp task
    #pragma omp target map ( tofrom : b )
        #pragma omp parallel for
            for (…)
```

OpenMP* 4.5 拡張
3. 依存性を伴う非同期オフロード

- 依存性を伴うオフロード
- 既存のタスクモデルで利用

  #pragma omp task depend(依存性タイプ:リスト) [節 [[,] 節] ...]
  構造化ブロック

  節: depend (in/out/inout)

- 利用例:
  - #pragma omp target device map(...) depend(リスト) if(..)
  - #pragma omp target enter data map(...) depend(リスト) if(..)
  - #pragma omp target exit data map(...) depend(リスト) if(..)
  - #pragma omp target update to/from depend(リスト) if(..)

OpenMP* 4.5 拡張
3. 依存性を伴う非同期オフロード

- 依存性を伴うオフロード

```
#pragma omp target map( tofrom : b ) ¥
          depend( in : x ) depend( out : y )
   #pragma omp parallel for
       for (…)
```

以下に相当：

```
#pragma omp task depend( in : x ) depend( out : y )
   #pragma omp target map( tofrom : b )
       #pragma omp parallel for
          for (…)
```

OpenMP* 4.5 拡張

4. map 節の拡張

- map 節の拡張

  節:　　map ([マップタイプ修飾子 ,] マップタイプ:リスト)
  マップタイプ修飾子: always
  マップタイプ: alloc、release、to、from、tofrom、delete

  - マップタイプ修飾子: always
    - マップタイプが to/tofrom でデバイス上にメモリーが存在する場合、データはホストからデバイスへ転送されます
    - マップタイプが from/alloc でデバイス上に存在しない場合、未定義の動作となります
  - マップタイプ: delete
    - デバイス上のメモリーは解放されます

```
01 #pragma omp target data device(0) map(to:a) map(to:b)
02 #pragma omp target data device(0) map(to:a) map(to:b)
03 #pragma omp target data device(0) map(always, to:a) map(to:b)
04 #pragma omp target data device(0) map(to:a) map(to:b)
```