インテル® コンパイラーによる並列プログラミング: OpenMP* 5.0 プレビュー

2018 年 3 月
iSUS 編集部

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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・プログラミングの概要をリフレッシュし、インテル® コンパイラー V18.0 でサポートされる OpenMP* 5.0 TR4 および TR6 の機能と実装を紹介します

セッションの対象者

- 既に OpenMP* でマルチスレッド・プログラミングを開発し、4.0 以降でサポートされる新たなベクトル化とオフロードを導入し、アプリケーションのパフォーマンス向上を計画する開発者

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

内容

- はじめに (OpenMP* が必要とされる背景)
- 概要 (OpenMP* とは、歴史、各バージョンの機能概要)
- OpenMP* 4.0 と 4.5 および 5.0 の新機能

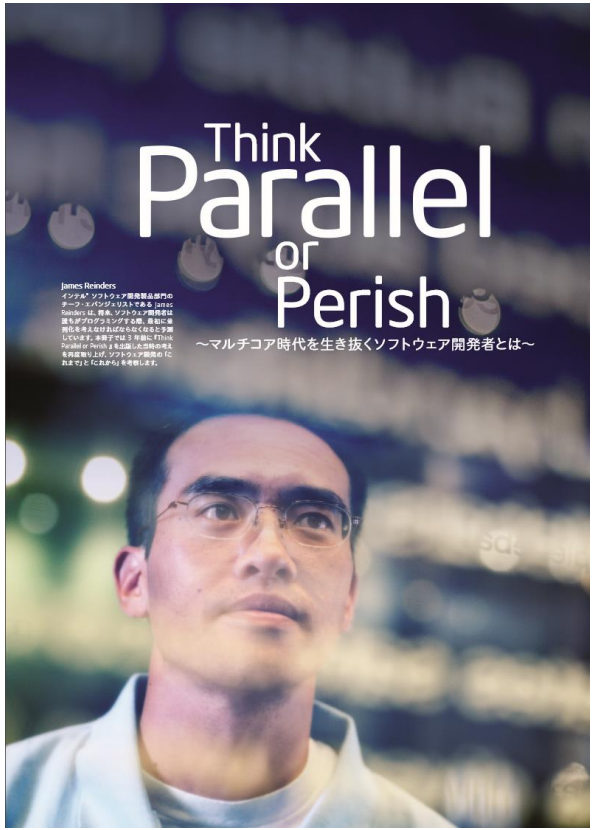
© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

はじめに

- コードの現代化 (モダニゼーション)
- OpenMP* が求められる背景
  - マルチスレッド化
  - ベクトル化
  - ノード間の並列処理

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

Think Parallel or Perish



あれから 8 年 …

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

Vectorize or Die

The Parallel Universe 3

編集者からのメッセージ

James Reinders インテル コーポレーションの並列プログラミング・エバンジェリスト兼アーキテクチャー。新しい書籍 「High Performance Parallel Programming Pearls Volume Two」 の共著者で、このほかに、「High Performance Parallel Programming Pearls [Volume One]」 (2014)、「Multithreading for Visual Effects」 (2014)、「Intel® Xeon Phi™ Coprocessor High Performance Programming」 (2013、日本語 「インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング」)、「Structured Parallel Programming」 (2012、日本語 「構造化並列プログラミング」)、「Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism」 (2007、日本語 「インテル® スレッディング・ビルディング・ブロック ― マルチコア時代の C++ 並列プログラミング」、中嶋謙一訳 [近刊]) の著書があります。「VTune™ Performance Analyzer Essentials」 [2006] などの文献を発表しています。

ベクトル化で生き残る

ベクトル化というキーワードは、最近、私の中で大きなウェイトを占めています。それには、「ベクトル化アドバイザー」と呼ばれる新しいツールが登場したこと、OpenMP* 「SIMD」 宣言子をより深く理解したこと、**実世界のベクトル化例を含む新しい書籍**を執筆したことなど、さまざまな理由があります。

ベクトル化は、この春に 「**Vectorize or Die (ベクトル化か死か)**」 という名前で開催された 2 つのインテル® ソフトウェア・テクニカル Web セミナーのトピックでした。この言葉の意味は、簡単な計算を行うことですぐに理解できます。例えば、最新の並列 x86 製品であるインテル® Xeon Phi™ コプロセッサー (最大 244 スレッド、512 ビット幅ベクトル) で考えてみましょう。単純な計算では、244 スレッドを使用するスレッド化プログラムのスピードアップは 244 倍を超えることはありません。[1] また、完璧にベクトル化されたプログラムのスピードアップは 16 倍を超えることはありません (単精度、512 ビット SIMD)。しかし、これらを組み合わせて使用すると、「超えられない」 スピードアップは 3,904 倍に跳ね上がります。つまり、スレッド化とベクトル化を組み合わせることが、パフォーマンス向上の鍵なのです。ベクトル化が最近話題になっている理由は、まさにここにあります。

Dr. Fortran (Steve Lionel 氏) は、Fortran ユーザーの調査に熱心に取り組み、ブログで紹介しています (**ページ 24 を参照**)。私はかねてからの Dr. Fortran のファンであり、彼の最新のブログは一見の価値があります。

この号の注目記事 「OpenMP* SIMD によるベクトル・プログラミング ― インテル® Xeon® プロセッサー、インテル® Xeon Phi™ コプロセッサー、インテル® GPU 向け」 では、OpenMP* で利用可能になった明示的なベクトル・プログラミング用の SIMD 宣言子を詳しく説明します。OpenMP* を使用しない場合でも、この記事の**明示的な**ベクトル・プログラミングに関する考察に目を通してみてください。明示的なベクトル化は、シーケンシャル・コードに基づいてベクトル化を行うため、これまでコンパイラーの判断によるベクトル化機能と、コンパイラーに与えることができる「ヒント」に依存していました。しかし、ベクトル化できなかったときにどのようなヒントやコンパイラー・オプションを追加すべきかは悩みの種でした。明示的なベクトル化によりベクトル化を直接プログラムすることができますが、コンパイラーの判断を覆す操作を行うことはコンパイラーによる保証を放棄することになります。詳細は記事をご覧ください。

「ベクトル化アドバイザー : ベクトル化を支援する新しいツール」 では、ベクトル化を行う方法に関する質問への回答を提供する新しいツールを取り上げます。ベクトル化アドバイザーは、最新のインテル® Advisor XE の新機能です。この記事をお読みになった後、評価版をダウンロードしてコードを適切にベクトル化できるかどうか確認してみてください。

マルチスレッド・アプリケーションをベクトル化することで、ハイパフォーマンスを達成できるチャンスが大きく広がります。この号が効率的なベクトル化を達成する実践的な方法を見つけるのに役立つことを期待しています。

James Reinders
2015 年 9 月

1. ここでは、倍形 (ニア) スピードアップを上回る、キャッシュ効果による超線形 (スーパーリニア) スピードアップは考慮していません。考慮しても数字が複雑に異なるだけで結論が異なることはないため、たやすく計算できる数値を取り上げています。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。

arallel or Perish

lodernization

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

Code Modernization (コードのモダン化)

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

マルチレベルの並列性を実装する 5 つのステップ

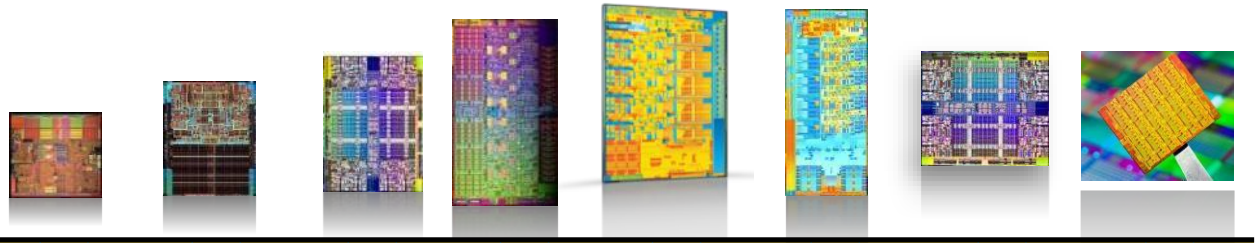
ステージ 1: 最適化ツールとライブラリーの活用

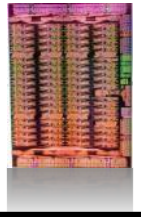
http://www.isus.jp/article/article-parallel/what-is-code-modernization/

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

Parallel + SIMD は前進への鍵

インテル® Xeon® プロセッサーとインテル® Xeon Phi™ 製品ファミリーは、両者とも並列性を高めています





	インテル® Xeon® プロセッサー 64 ビット	インテル® Xeon® プロセッサー コード名 Woodcrest EP	インテル® Xeon® プロセッサー コード名 Nehalem EP	インテル® Xeon® プロセッサー コード名 Westmere EP	インテル® Xeon® プロセッサー コード名 Sandy Bridge EP	インテル® Xeon® プロセッサー コード名 Ivy Bridge EP	インテル® Xeon® プロセッサー コード名 Haswell EP	インテル® Xeon® プロセッサー コード名 Skylake Server[1]	インテル® Xeon Phi™ x100 コプロセッサー (コード名: Knights Corner)	インテル® Xeon Phi™ x200 プロセッサー & コプロセッサー (コード名: Knights Landing)
コア	1	2	4	6	8	12	18	28	61	72
スレッド	2	2	8	12	16	24	36	56	244	288
SIMD 幅	128	128	128	128	256	256	256	512	512	512

さらに多いコア　さらに多くのスレッド　より広いベクトル

OpenMP* は parallel + SIMD を前進させる最も重要な機能の 1 つ

* ark.intel.com で公開されている出荷済の製品仕様　1.出荷前もしくは計画中。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

内容

- はじめに (OpenMP* が必要とされる背景)
- 概要 (OpenMP* とは、歴史、各バージョンの機能概要)
- OpenMP* 4.0 と 4.5 および 5.0 の新機能

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* API

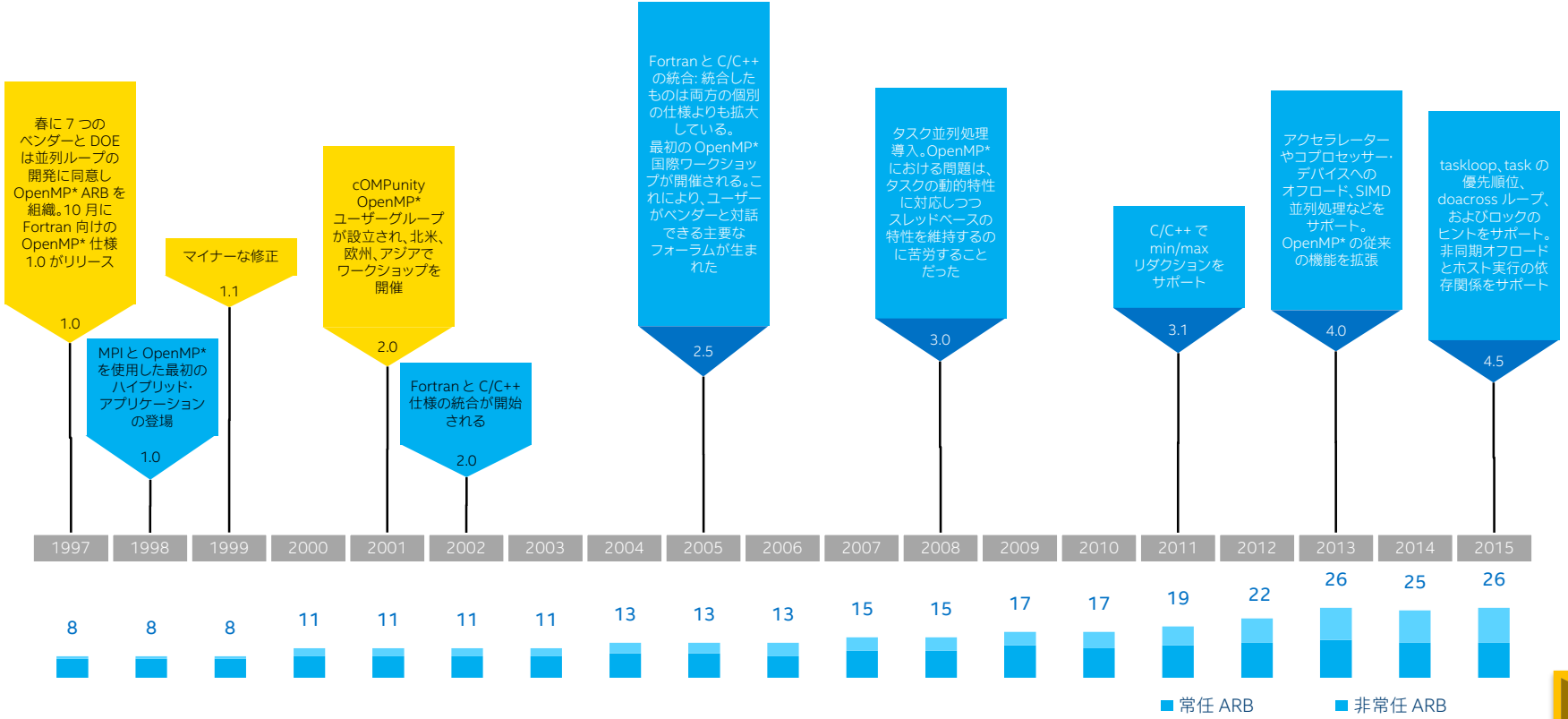
ほぼ 20 年来の技術計算/HPC における共有メモリー並列プログラミング (C/C++ と Fortran) 向けの業界標準:

- 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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* の歴史



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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* 5.0 TR4	201611	インテル® コンパイラー V18.0
OpenMP* 4.5	201511	インテル® コンパイラー V17.0 、gcc 6.1
OpenMP* 4.0	201307	インテル® コンパイラー V14.0 、gcc 4.9 (offload 5.1)
OpenMP* 3.1	201107	インテル® コンパイラー V12.1 、gcc 4.7
OpenMP* 3.0	200805	インテル® コンパイラー V11.0 、gcc 4.4
OpenMP* 2.5	200505	インテル® コンパイラー V9.0 、gcc 4.2

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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)

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

ソースとコンパイラーのバージョンの不一致

ソースコードで使用する OpenMP* の仕様よりもコンパイラーがサポートする OpenMP* 仕様が古い場合、コンパイルエラーとなります

Intel Compiler 17.0 Update 4 Intel(R) 64 Visual Studio 2015

```
C:¥Users¥kiyo¥Desktop¥OpenMP2018¥code¥task>
C:¥Users¥kiyo¥Desktop¥OpenMP2018¥code¥task>icl pi.c /Qopenmp
インテル(R) 64 対応インテル(R) C++ コンパイラー (インテル(R) 64 対応アプリケーション用) バー
ジョン 17.0.4.210 ビルド 20170411
(C) 1985-2017 Intel Corporation.  無断での引用、転載を禁じます。

pi.c
pi.c(21): エラー: reduction 句はディレクティブと互換性がありません。
  #pragma omp taskloop reduction(+:sum) private(x)
                       ^

コンパイルは pi.c で異常終了しました (コード 2)。
```

_OPENMP マクロでコンパイルを制御:

```
#if _OPENMP >= 201611
#pragma omp taskloop reduction(+:sum) private(x)
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

インテル® 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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

S

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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 18.0 Update 1 Intel(R) 64 Visual Studio 2017
C:¥Users¥kiyo¥Desktop¥OpenMP2018¥code¥task>pi

OPENMP 表示環境開始
   _OPENMP='201611'
  [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 = 3.141592653589793
Pi = 3.141593 Time = 0.452000

C:¥Users¥kiyo¥Desktop¥OpenMP2018¥code¥task>_
```

true、false (デフォルト)、verbose が設定できます

S

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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*)

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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　(廃止予定)

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

先に進む前に少し思い出しておきましょう

OpenMP* 3.1 までの機能

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

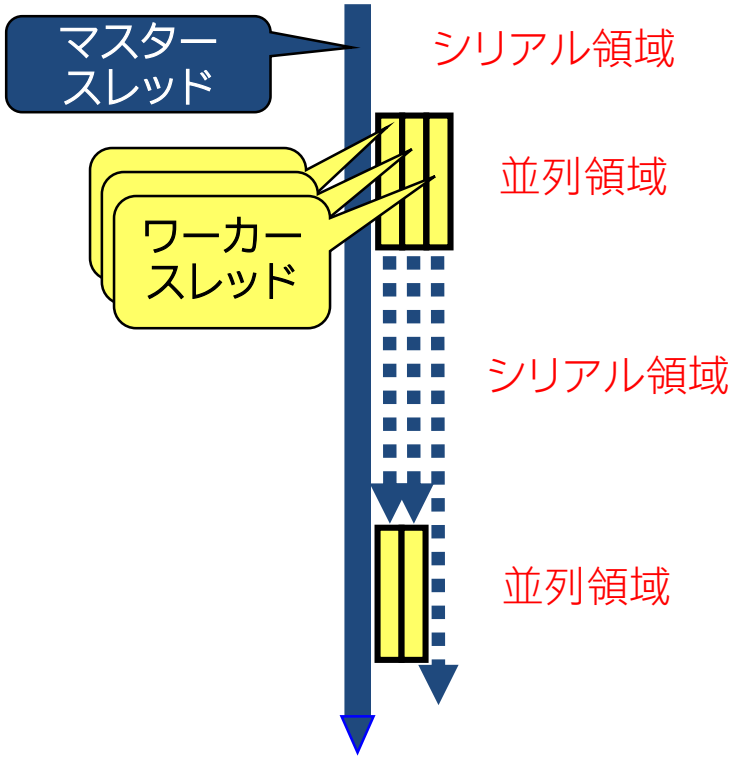
ランタイム

- 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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* の実行モデル

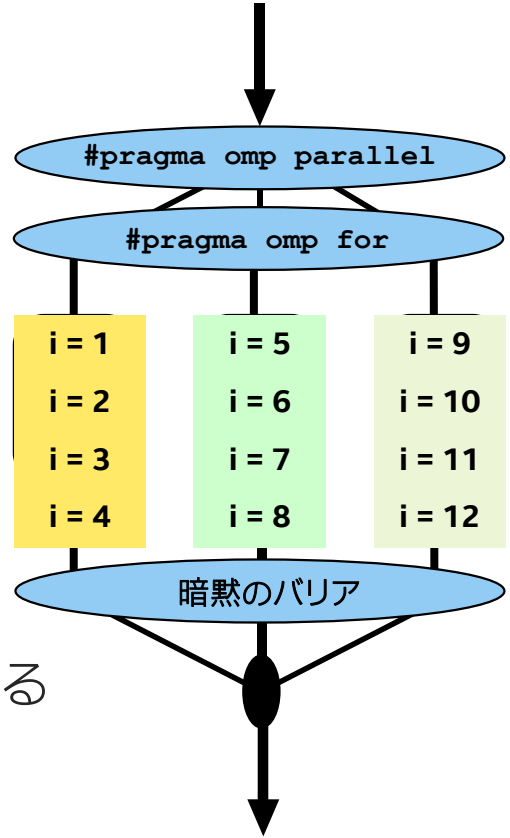
- OpenMP* プログラムはシングルスレッドで処理を開始: マスタースレッド
- ワーカースレッドは並列領域でスポーンされ、マスターとともにスレッドのチームを形成
- 並列領域の間ではワーカースレッドはスリープ状態になる。OpenMP* ランタイムがすべてのスレッドの動作を管理
- コンセプト: フォークジョイン
  - インクリメンタルな並列処理を許可



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

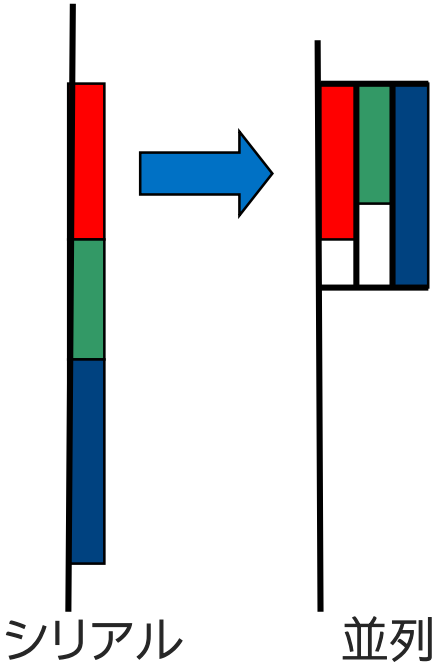
OpenMP* 並列領域: sections ワークシェアの例

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();
}
```



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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);
}
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

本日の内容

- 概要 (OpenMP* とは、歴史、各バージョンの機能概要)
- OpenMP* 4.0 と 4.5 および 5.0 の新機能
  - タスク
  - omp simd
  - オフロード

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 TR6 の機能

- **複数レベルのメモリーシステムのサポート:** OpenMP* 5.0 では、異なる種類のメモリーにデータを配置する直観的なメモリー割り当てメカニズムがサポートされます
- **複雑なデータ構造のディープコピーをサポート:** TR6 ではユーザー定義のマッパーが追加されます。オブジェクト指向のデータ構造を、アクセラレーター・デバイスに正しくコピーする構成可能なメカニズムを提供します
- **OpenMP* device 構文の追加拡張:** その他、アクセラレーター向けの OpenMP* サポートの主要機能として、ユニファイド共有メモリーのサポート、デバイス固有の機能実装を使用する能力、明示的なデータマッピングの制御、および実行時のデバイスオフロードのオーバーライドなどが考えられています
- **記述的なループの最適化サポート:** 並行性構造は、関連するループ入れ子のすべての反復が任意の順番で同時実行されることを示唆します。これは、コンパイラーによる実装固有の最適化により実現されます
- **OpenMP* アプリケーションの改善されたデバッグ機能をサポート:** TR6 では OMPD を追加します。これはサードパーティー・ツール・インターフェイスによる直観的な OpenMP* コードのデバッグを可能にし、OpenMP* パフォーマンスの詳細な解析をサポートするため TR4 で追加されたファーストパーティー・ツール・インターフェイスである OMPT を補完します
- **タスク依存の新しい形式:** 新しいメカニズムは、(1) セット中で相互排他を必要とするタスクセットと、他のタスクに関連する順序付け、および (2) 動的に発生する依存性のセットとタスクをサポートします
- **大きなメモリーモデルの柔軟性:** OpenMP* 5.0 は、低レベルのメモリー同期の最適化を可能にする acquire (取得)/release (解放) セマンティクスをサポートします

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* タスクに関する拡張

task、taskwait、taskyield

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

タスクを使用しない 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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* の task 構文

task [2.9.1] [2.11.1]

明示的にタスクを定義します。タスクのデータ環境は、task 構文のデータ共有属性節と適用されるデフォルトに従って作成されます。

#pragma omp task *[節[[,]節] ...]*

構造化ブロック

節:

- if(*スカラー式*)
- final(*スカラー式*)
- untied
- default(shared | none)
- mergeable
- private(*リスト*)
- firstprivate(*リスト*)
- shared(*リスト*)

Depend 節 [2.13.9]

タスクまたはループ反復のスケジュールに追加の制約を適用します。これらの制約は、兄弟タスクまたはループ反復間のみの依存関係を設定します。

depend(*依存性タイプ*: *リスト*)

*依存性タイプ*は、in、out もしくは inout です:

in: 生成されるタスクは、out または inout *依存性タイプ*リストの 1 つ以上のリスト項目を参照する、以前に生成されたすべての兄弟タスクに依存します。

out と inout: 生成されるタスクは、in、out、または inout *依存性タイプ*リストの 1 つ以上のリスト項目を参照する、以前に生成されたすべての兄弟タスクに依存します。

---

depend(*依存性タイプ*)

*依存性タイプ*は source です。

---

depend(*依存性タイプ*[: *vec*])

*依存性タイプ*は sink で、*vec* 反復は次の形式です:

$x_1 [\pm d_1], x_2 [\pm d_2], \ldots, x_n [\pm d_n]$

priority(*優先値*)

*優先値*は、生成されたタスクに優先順位を付けるためのヒントを指定する負ではない数値スカラー式です。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

タスク動作を制御する例

```
#pragma omp task
{
    …
    while( (status = omp_test_lock(lock)) != true){ // ロックが取得できるまでループ
#pragma omp taskyield
    }
    …
    omp_unset_lock(lock);
}
```

while ループで OpenMP ロックの取得を待機していますが、タスクは消費されます

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

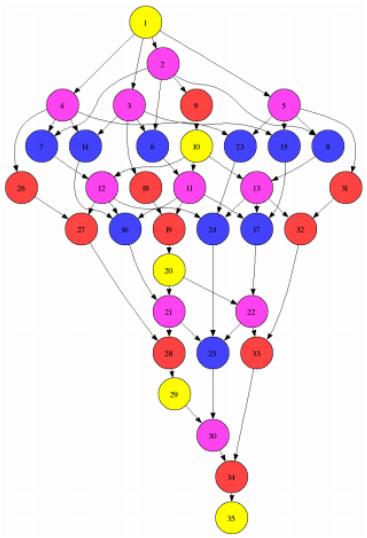
```
int val=0;

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);
}
```

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

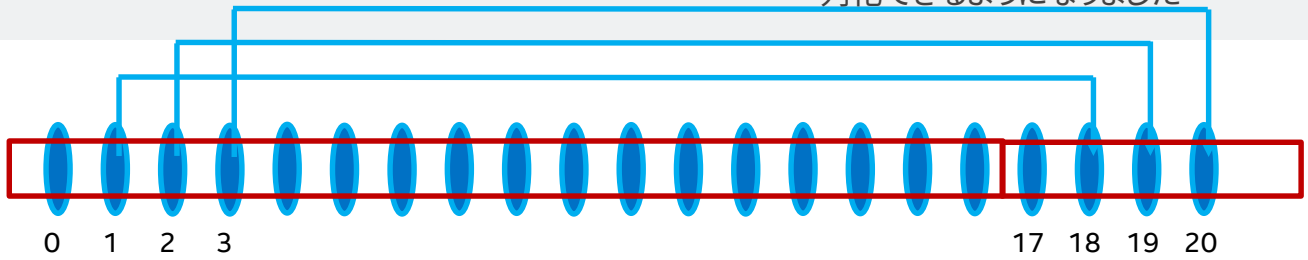
Do-across ループ並列

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)
    }
}
```

ループを並列化するとき、ループ伝搬依存がないときは、Doallループとして並列化し、ループ伝搬依存があるときは、同期操作がある Doacrossループとして並列化します
そのため、ordered 句を伴う入れ子になった Do Across ループをサポートするため、depend 節の source と sink 依存性タイプがサポートされました。
これにより、構造化された依存性を持つループを並列化できるようになりました



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

taskloop 句

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

#pragma omp taskloop [simd] [節]
for/do 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

```
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;
    // ...

}
    // ...
```

4.5 ではまだ気軽に taskloop は使えない …

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* のタスク処理: さらに ...

キャンセル (cancel)

- 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 (taskgroup)

- 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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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?

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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 では未サポートです

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 の task 機能強化

- task、taskgroup、taskloop 句でのリダクション演算のサポート
- taskloop 構文で collapse 節をサポート
- taskwait 句に depend 節が指定できるようになりました
- depend 節に排他 inout セットをサポートする mutexinoutset タイプが追加されました

https://www.isus.jp/products/c-compilers/openmp-50-support-in-180/

https://www.isus.jp/products/c-compilers/openmp_tr6_new_features/

https://www.isus.jp/hpc/pu27-01-present-and-future-of-openmp/

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

taskloop でのリダクション演算

OpenMP* 5.0 の機能

OpenMP 4.5 までは次のようなリダクションを含むワークシェア構文を taskloop に書き換えることは困難でした

```
#pragma omp parallel
#pgarma omp single
#pragma omp taskloop reduction(+:sum) private(x)
   for (i=0;i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
  pi = step * sum;
```

OpenMP* 5.0 では、taskloop でリダクション演算が利用できます

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 の機能

taskgroup を適用して次のように記述できます

task_reduction と in_reduction を使用

```
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup task_reduction(+:sum)
#pragma omp taskloop in_reduction (+:sum) private(x)
   for (i=0;i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
  pi = step * sum;
```

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

task_reduction と in_reduction

OpenMP* 5.0 の機能

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

in_reduction が指定されないタスクの変数はリダクションに参加しません

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

taskloop 構文中での collapse 節

OpenMP* 5.0 の機能

```
void taskloop_example() {
#pragma omp taskgroup
    {
#pragma omp task
        long_running_task() // 同時に実行可能
#pragma omp taskloop collapse(2) grainsize(500) nogroup
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                loop_body();
```

この構文は、ループの反復空間をチャンクに分割し、それぞれのチャンクに対して 1 つのタスクを生成します。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

taskwait 句での depend 節

OpenMP* 5.0 の機能

V18 では未実装

```
main ( ){
int x = 2;
#pragma omp task // shared(x) mergeable
    {
    x++;
    }
#pragma omp taskwait // depend(in:x)
   printf("%d¥n",x); // prints 3
}
```

2 つのタスクの
データスコープ
は異なります

taskwait 構文に depend 節が指定される場合、マージ可能なインクルード･タスクを生成する空の構造化ブロックを持つ task 構文であるかのように動作します

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

相互排他を必要とするタスクセット mutexinoutset 依存関係タイプ

OpenMP* 5.0 の機能

V18 では未実装

```
int val=0;

int main(){
#pragma omp parallel
  {
  #pragma omp single
    {
  #pragma omp task depend(mutexinoutset:val)
    …
  #pragma omp task depend(mutexinoutset:val)
    …
    }
  }
  printf("Value is %d¥n", val);
}
```

タスク生成構文に mutexinoutset 依存関係タイプを持つ depend 節のリスト項目がある場合、異なるタスク生成構文の mutexinoutset 依存関係タイプを持つ depend 節のリスト項目と同じストレージである必要があります。両方の構文は兄弟タスクを生成し、兄弟タスクは排他タスクとなります

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

明示的な SIMD プログラミング・モデル

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

なぜ SIMD 拡張? OpenMP* 4.0 以前

コンパイラー･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() 組込み関数)

```
#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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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];
}
```

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
```

さらに ....

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

SIMD プラグマ/ディレクティブ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!

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
- …

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* SIMD ディレクティブの節

OpenMP* 4.0 の機能
OpenMP* 4.5 の機能

- safelen(レングス):
  SIMD 命令によって同時に 2 つの反復が実行できない場合、この値でより大きな論理的反復空間を指定します
- 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)
- simdlen (レングス): 正の整数式で関数の同時引数の数を指定します

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

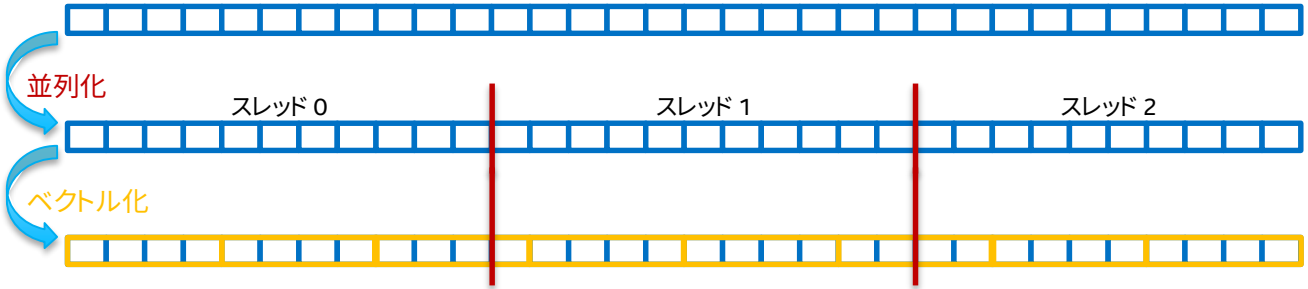
```
void vec1(float *a, float *b, int off, int len){
#pragma omp simd safelen(32) aligned(a:64, b:64)
    for(int i = 0; i < len; i++){
        a[i] = (a[i] > 1.0) ?
            a[i] : b[i];
            a[i + off] * b[i];
    }
}
```

```
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
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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)

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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)
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

SIMD 対応関数:Linear/ Uniform

- なぜこれらが必要なのか?
- 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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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 句に記述できます
- 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りません
- 関数やサブルーチンは、構造化ブロックでなければなりません
- 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ません
- 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りません
- 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す
- C/C++: 関数は、*longjmp* や *setjmp* を呼び出してはなりません

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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!

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 の simd 機能強化

OpenMP* 5.0 の機能

V18 では未実装

- simd 構文に nontemporal 節が追加されました
  - nontemporal 節は、リスト項目のトレージ位置へのアクセスが、ループ反復間で低い時間的な局所性を持つことを指定します
- simd 構文で collapse 節をサポートしました
  - simd 構文でも入れ子のループを 1 つのループに展開できるようになりました
- simd 構文で atomic 節が利用できるようになりました

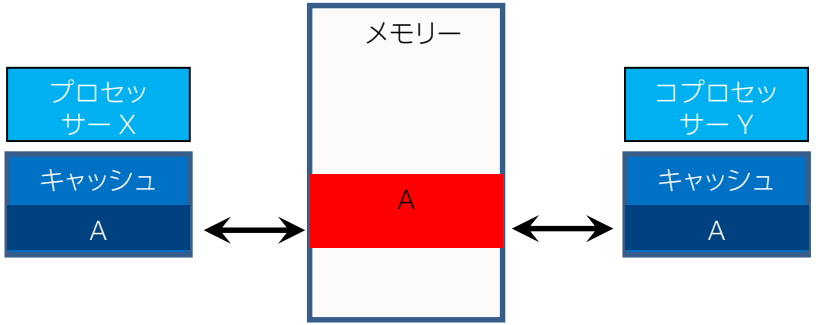
© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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



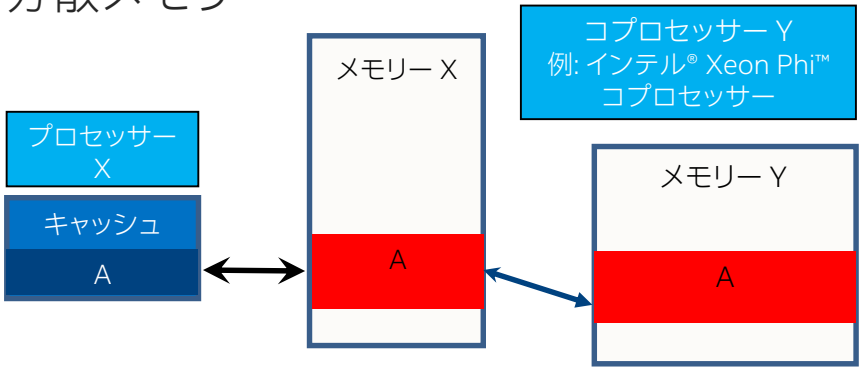
スレッドは共有メモリーへアクセスできる

- 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

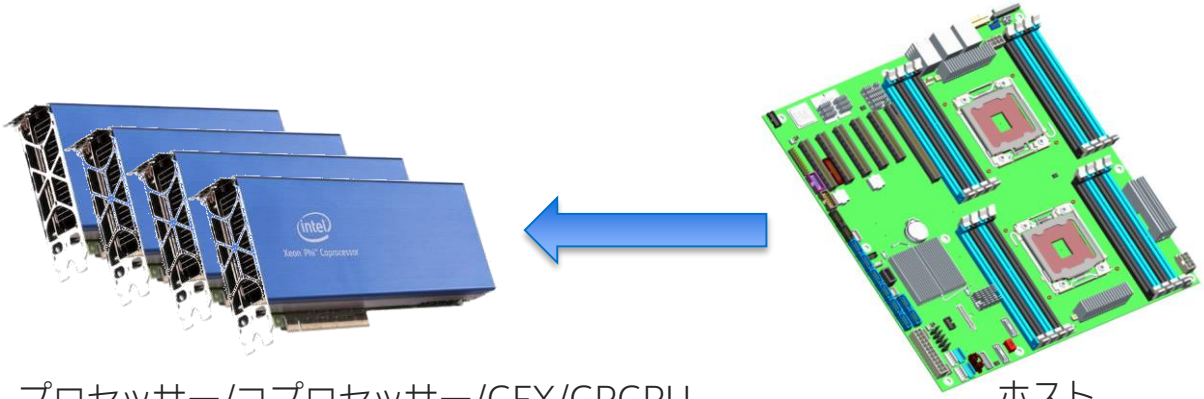
© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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/GPGPU



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

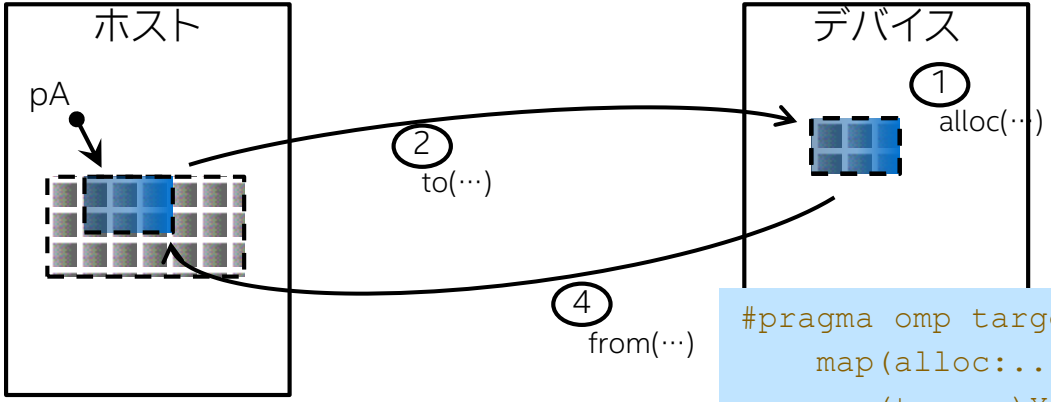
- OMP_DEFAULT_DEVICE を介してデフォルトデバイスを制御
- 負ではない整数値

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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();
  }
}
```

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
}
```

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* と OpenACC* の比較例

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]);
  }
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* と OpenACC* 対応

機能	OpenACC*	OpenMP*
オフロード	parallel または kernels	target
データ環境	data	target data
データ転送、変数定義	copy(変数)/ create(変数)	map(tofrom:変数) map(alloc:変数)
並列処理(チーム)	loop gang num_gangs(n)	teams distribute num_teams(n) thread_limit(n)
並列処理	loop vector vector_length(n)	parallel for

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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/

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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/

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

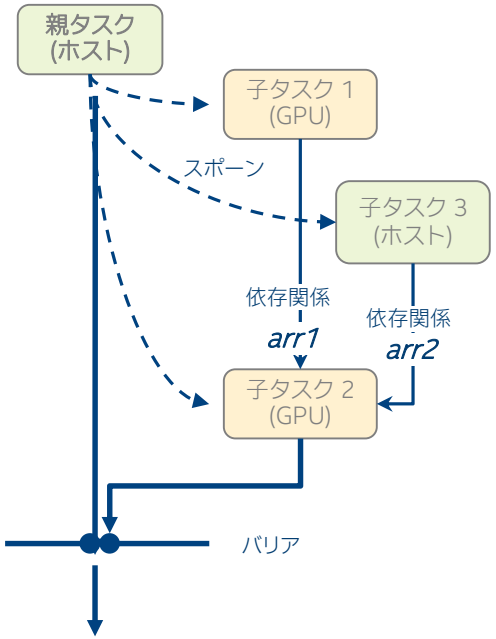
target ディレクティブで depend 節を使用して GPU へ非同期オフロードする例

```
// 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]; }
```



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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 (…)

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

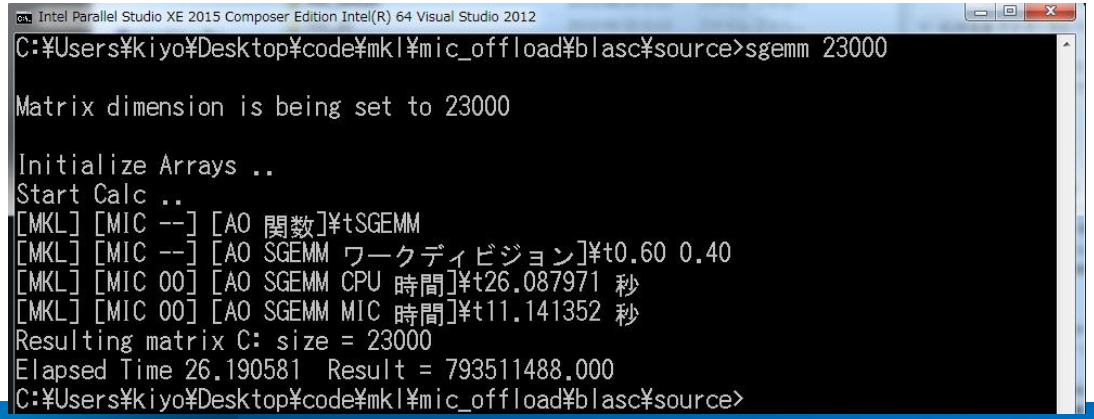
次のメカニズムを使用してオフロードレポートを有効または無効にすることができます:

- OFFLOAD_REPORT 環境変数
- _Offload_report API

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 の offload 拡張

- プログラマーの負担を軽減するため、いくつかの関数 (C、C++、Fortran) とサブルーチン (Fortran) で暗黙の declare target ディレクティブが追加されました [2.15.5、2.15.7]
- 入れ子になった declare target ディレクティブのサポートが追加されました [2.15.7]
- デバイス固有の関数実装をサポートするため、declare target ディレクティブに implements 節が追加されました [2.15.7]
- 複雑なデータタイプのマッピングをサポートするため、declare mapper ディレクティブが追加されました [2.15.8]
- 配列セクションへのマップで、ポインター変数へのマップ (C/C++) とデバイスメモリーのアドレス割り当てが追加されました [2.20.6.1]
- スレッドがどのデバイスで実行されているかを特定するため、omp_get_device_num ランタイムルーチン [3.2.36] が追加されました
- オフロード動作の制御をサポートするため、OMP_TARGET_OFFLOAD 環境変数が追加されました [5.17]

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

暗黙の declare target ディレクティブ

- オフロード領域で呼び出されている関数を自動的に検出して、それらの関数が declare target ディレクティブで指定されているかのように扱います。以前は、オフロード領域で呼び出されるすべての関数は、declare target ディレクティブによって明示的にタグ付けされている必要がありました

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

OpenMP* バージョン 4.5 のスタイル

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

OpenMP* バージョン 5.0 のスタイル

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

静的記憶域を含む変数の自動検出

- OpenMP* 5.0 では、静的記憶域を含む変数も自動的に検出できます。これにより、次の 2 つの例は等価となります

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

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


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

配列セクションへのマップで、ポインター変数へのマップ (C/C++) とデバイスメモリーのアドレス割り当て

- OpenMP* バージョン 4.5 では、use_device_ptr 節が追加されましたが、use_device_ptr の変数は、使用する前にマップする必要があります。変数は 1 つのデータ節にのみ記述できるため、プログラマーは個別の #pragma target data 節を記述する必要がありました:

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

- OpenMP* 5.0 では、単一構文で変数を map 節と use_device_ptr 節の両方に記述できる例外が追加されました:

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


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

つづき

- OpenMP* 4.5 では、最初の構造がターゲットである結合構造の reduction 節または lastprivate 節で使用されるスカラー変数は、ターゲット構造の firstprivate として扱われます。ホストの変数が更新されることはありません。ホストの変数を更新するには、プログラマーは結合構造から omp target ディレクティブを分離してスカラー変数を明示的にマップする必要があります
- OpenMP* 5.0 では、これらの変数は自動的にmap(tofrom:variable) が適用されているかのように扱われます

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

omp declare target 内の静的データメンバー

- OpenMP* 5.0 では、スタティック･データ･メンバーが omp declare target 構文内のクラスで使用できるようになりました。
- また、スタティック･メンバーを含むクラス･オブジェクトは map 節でも使用できます

```
#pragma omp declare target
class C {
    static int x;
    int y;
}
class C myclass;
#pragma omp end declare target
void bar() {
#pragma omp target map(myclass)
    {
    myclass.x = 10
    }
}
```


© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

入れ子になった declare target のサポート

- 外側の omp target data 構文内で構造体変数のフィールドをマップして、内側の入れ子の omp target 構文内で構造体変数のアドレスを使用すると、構造体の一部がすでにマップされている場合、構造体変数全体をマップしようとします

```
struct {int x,y,z} st;
int A[100];
#pragma omp target data map(s.x A[10:50])
{
#pragma omp target
    {
    A[20] = ; // OpenMP* 4.5 ではエラー、5.0 では OK
    foo(&st); // OpenMP* 4.5 ではエラー、5.0 では OK
    }
#pragma omp target map(s.x, A[10:50])
    {
    A[20] = ; // OpenMP* 4.5 と 5.0 の両方で OK
    foo(&st); // OpenMP* 4.5 と 5.0 の両方で OK
    }
}
```

OpenMP 5.0 では、プログラマーが想定した動作になるように、これらのケースが修正されました

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

ヘテロジニアス・プログラミングの向上

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

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

アフィニティー拡張

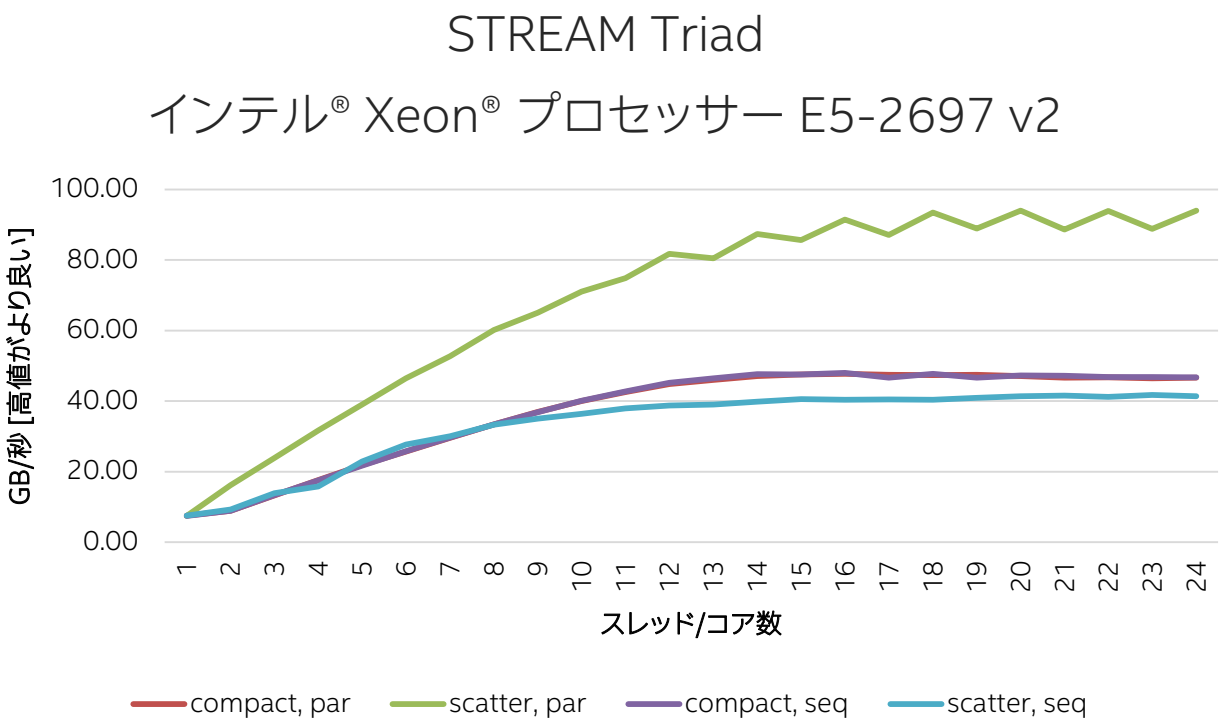
© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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
  - 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

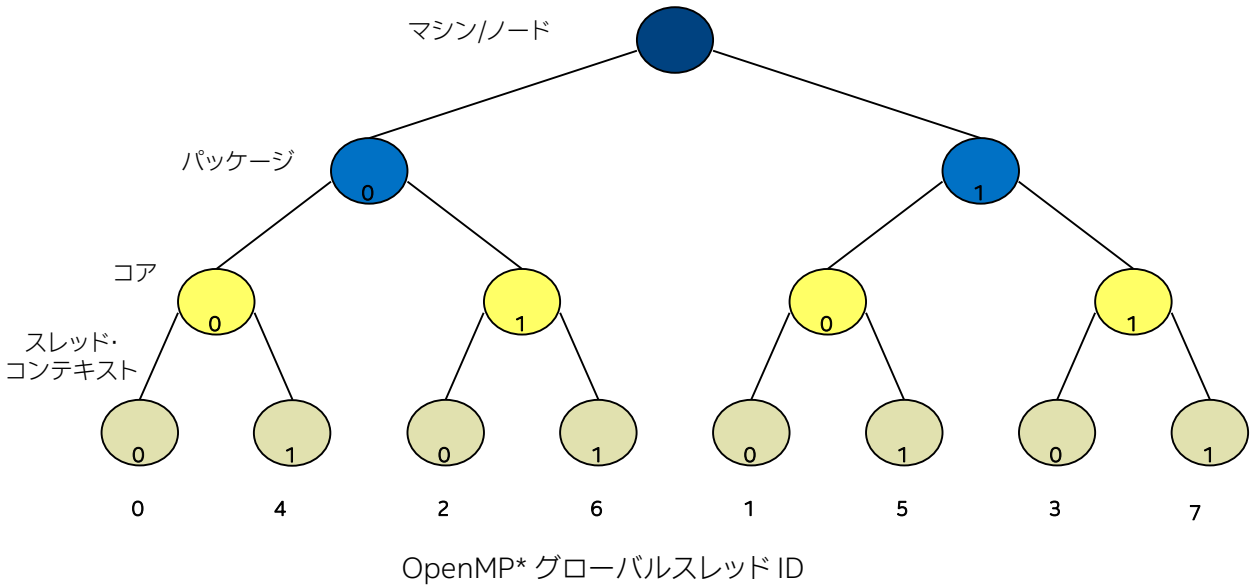
© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

スレッド･アフィニティー - なぜ重要なのか？



© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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"

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 のメモリーと同期管理

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

新しい allocate ディレクティブ

- 新しい allocate ディレクティブは、API 呼び出しで割り当てられない変数の割り当てを制御するために提案されたものです (自動変数または静的変数など)
- OpenMP* ディレクティブによる割り当て操作に使用できます (変数のプライベート・コピーなど)

```
int a[N], b[M];
#pragma omp allocate(a,b) memtraits(bandwidth=highest, pagesize=2*1024*1024)
void example() {
#pragma omp parallel private(b) allocate(memtraits(latency=lowest):b)
    {
    // ...
    }
}
```

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 4.5 でロックのヒントを定義

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。例えば、インテル® TSX (インテル® Transactional Synchronization Extensions)

- 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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 で同期のヒントを定義

ロックヒント (Lock hint) の名前が同期ヒント (synchronization hint) に変更されました。古い名前は廃止予定です。[2.18.11]

- OpenMP* ロック機能の一部
  - C/C++ の omp_lock_hint_t と Fortran の omp_lock_hint_kind タイプを廃止予定
  - および、ロックのヒント定数 omp_lock_hint_none、omp_lock_hint_uncontended、omp_lock_hint_contended、omp_lock_hint_nonspeculative、および omp_lock_hint_speculative を廃止

lock → sync

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* の今後

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 4.x と 5.0 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 TR6 の実装作業中

- Supercomputing 2018 でリリースされる見込み
- インテル® コンパイラーは、V18 で TR4 を実装

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

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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 は OpenMP* 5.0 TR4 をサポート

- 2017 年 9 月リリース

インテル® コンパイラー次期バージョンで OpenMP* 5.0 TR6 をサポート予定

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

参考資料

- OpenMP* TR6 (テクニカル･レポート 6) に含まれる新機能
- インテル® コンパイラー 18.0 の OpenMP* 5.0 サポート
- OpenMP* 誕生から 20 年 Part1
- OpenMP* 誕生から 20 年 Part2
- 現在と将来の OpenMP* API 仕様
- OpenMP* バージョン 4.5: 標準化の進化
- OpenMP サポートのまとめ

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)
- インテル® Parallel Studio XE によりアプリケーション･パフォーマンスを大幅に向上

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。

†開発コード名

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

バックアップ

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

付録: OpenMP* の仕様の違い

- 4.5 -> 5.0
- 4.0 -> 4.5
- 3.1 -> 4.0
- 3.0 -> 3.1
- 2.5 -> 3.0

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenMP* 5.0 で廃止予定の機能

- ICV nest-var の廃止に伴う廃止予定の機能
  - OMP_NESTED 環境変数
  - omp_set_nested および omp_get_nested API

- OpenMP* ロック機能の一部
  - C/C++ の omp_lock_hint_t と Fortran の omp_hint_hint_kind タイプを廃止予定
  - および、ロックのヒント定数 omp_lock_hint_none、omp_lock_hint_uncontended、omp_lock_hint_contended、omp_lock_hint_nonspeculative、および omp_lock_hint_speculative

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

バージョン 4.5 と 5.0 の違い

1. 指定されたフラッシュ特性に従って異なるタイプのフラッシュ操作を区別するためメモリーモデルが拡張され [1.4.4] 、同期フラッシュ操作の前に行われるように定義されました。[1.4.5]
2. C11、C++11、および C++14 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。[1.7]
3. Fortran 2003 のいくつかの機能をのサポートが追加されました。[未サポートの機能については 1.7 を参照]
4. タスク生成構文の depend 節に指定できるリスト項目 (C/C++ のすべての *lvalue* 表現を含む) が拡張されました。[2.1、2.18.10]
5. デバイス間の一般及び共有メモリー実装固有の機能を必要とするアプリケーションをサポートするため、require ディレクティブ [2.3] が追加されました。
6. デバイス構文の実行をランタイム制御するため、*target-offload-var* 内部制御変数 (ICV) [2.4] と OMP_TARGET_OFFLOAD 環境変数 [5.17] が追加されました。
7. *nest-var* ICV のデフォルト値が、実装依存から *false* に変更されました。[2.4.2]*nest-var* ICV [2.4.1]、OMP_NESTED 環境変数 [5.6]、および omp_set_nested/omp_get_nested ランタイムが廃止予定となりました。[3.2.10、3.2.11]
8. リスト表現が複数の式に拡張される可能性を表現できるようイテレーターが追加されました。[2.6]
9. 事前定義メモリー・アロケーター [2.7] とディレクティブ、節 [2.14]、および API ルーチン [3.6] が、異なる種類のメモリーをサポートするため追加されました。
10. C/C++ の正規表現ループ形式の *relational-op* が、!= を含むように拡張されました。[2.9]

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

バージョン 4.5 と 5.0 の違い (続き)

11. 不完全な入れ子ループに関連する collapse が、ループ [2.10.1]、simd [2.11.1]、taskloop [2.13.2]、distribute [2.15.11] 構文で定義されました。
12. SIMD 構文 [2.11] 内で atomic 構文が使用できるよう拡張されました。
13. simd 構文に nontemporal 節が追加されました。[2.11.1]
14. 任意の順番で同時実行されるループ反復をコンパイラーが最適化できるよう concurrent 構文が追加されました。[2.12]
15. タスクのリダクションをサポートするため、task [2.13.1] と target [2.15.5] 構文が in_reduction 節 [2.20.4.6] を受け入れるように拡張され、taskgroup 構文 [2.18.6] が task_reduction 節 [2.20.4.5] を受け入れるように拡張されました。
16. taskloop リダクションをサポートするため、taskloop [2.13.2] と taskloop simd [2.13.3] 構文が reduction 節 [2.20.4.4] と in_reduction 節 [2.20.4.6] を受け入れるように拡張されました。
17. 排他 inout セットをサポートするため、depend 節 [2.13.6、2.18.10] に mutexinoutset *dependence-type* が追加されました。
18. プログラマーの負担を軽減するため、いくつかの関数 (C、C++、Fortran) とサブルーチン (Fortran) で暗黙の declare target ディレクティブが追加されました。[2.15.5、2.15.7]
19. 入れ子になった declare target ディレクティブのサポートが追加されました。[2.15.7]
20. デバイス固有の関数実装をサポートするため、declare target ディレクティブに implements 節が追加されました。[2.15.7]

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

バージョン 4.5 と 5.0 の違い (続き)

21. 複雑なデータタイプのマッピングをサポートするため、declare mapper ディレクティブが追加されました。[2.15.8]
22. taskwait 構文に depend 節が追加されました。[2.18.5]
23. Weak メモリー順序付けのメモリー要求と解放セマンティクスをサポートするため、atomic 構文 [2.18.7] と flush 構文 [2.18.8] に acq_rel、acquire、および release 節が追加されました。
24. atomic 構文に hint 節が拡張されました。[2.18.7]
25. イテレーターをサポートするため depend 節 [2.18.10] が拡張されました。
26. ロックヒント (Lock hint) の名前が同期ヒント (synchronization hint) に変更されました。古い名前は廃止予定です。[2.18.11]
27. lastprivate 変数に条件付きの割り当てをサポートするため、lastprivate 節に conditional 修飾子が追加されました。[2.20.3.5]
28. 構造体メンバーのマッピングを明確に説明するため、map 節の説明が変更されました。[2.20.6.1]
29. 配列セクションへのマップで、ポインター変数へのマップ (C/C++) とデバイスメモリーのアドレス割り当てが追加されました。[2.20.6.1]
30. 構文が明示的にマップする必要がある、すべての変数、すべての変数参照の要求をサポートするため、defaultmap 節 [2.20.6.2] が none パラメーターを許容するように拡張されました。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

バージョン 4.5 と 5.0 の違い (続き)

31. OpenMP* スレッド･アフィニティーの情報を提供するため、ランタイムルーチン [3.2.29、3.2.30、3.2.31、3.2.32] と環境変数 [5.13、5.14] が追加されました。
32. スレッドがどのデバイスで実行されているかを特定するため、omp_get_device_num ランタイムルーチン [3.2.36] が追加されました。
33. ファーストパーティーのツール･インターフェイス [4.1] のサポートが追加されました。
34. サードパーティーのツール･インターフェイス [4.2] のサポートが追加されました。
35. オフロード動作の制御をサポートするため、OMP_TARGET_OFFLOAD 環境変数が追加されました。[5.17]

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

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。

© 2018 iSUS *その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

iSUS