インテル® コンパイラー V19.0 による並列プログラミング: OpenMP* 5.0 プレビュー Part 1

2018 年 9 月

iSUS 編集部

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 V19.0 でサポートされる OpenMP* 5.0 の機能と実装を紹介します

セッションの対象者

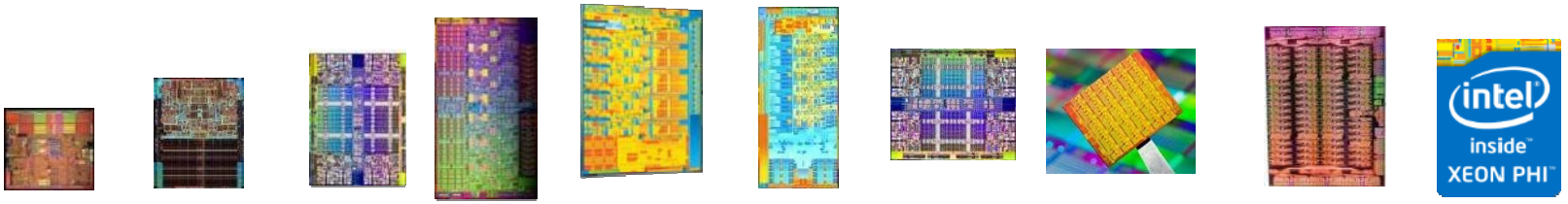
- 既に OpenMP* でマルチスレッド・プログラミングを開発し、4.0 以降でサポートされる新たなベクトル化とオフロードを導入し、アプリケーションのパフォーマンス向上を計画する開発者

内容

- 概要 (OpenMP* とは、歴史、各バージョンの機能概要)
- OpenMP* 4.0、4.5 の機能と5.0 の新機能

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。

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

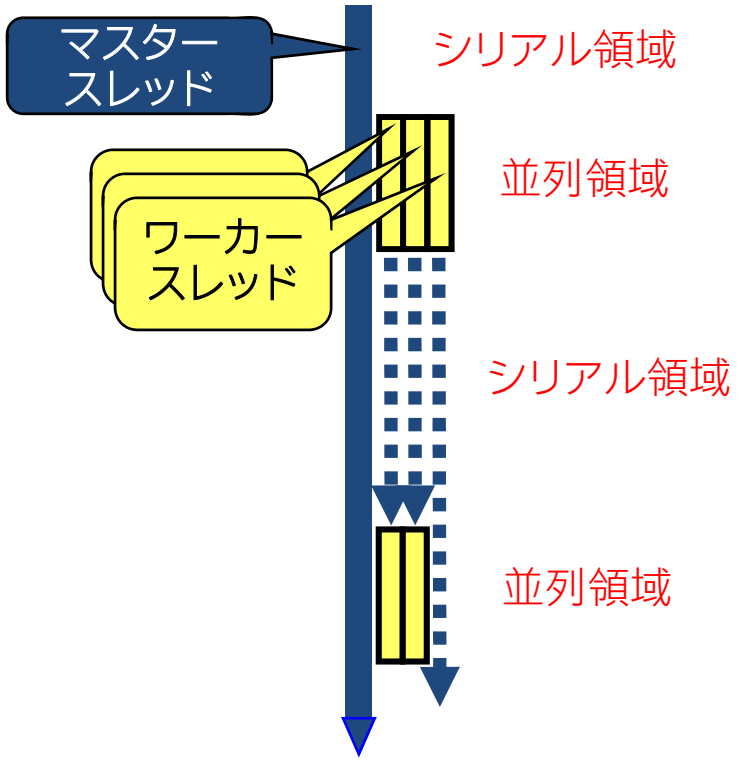
- スレッドの設定
- スレッドの制御
- ワークシェア
- アフィニティー
- アクセラレーター
- キャンセル
- 操作可能

ランタイム

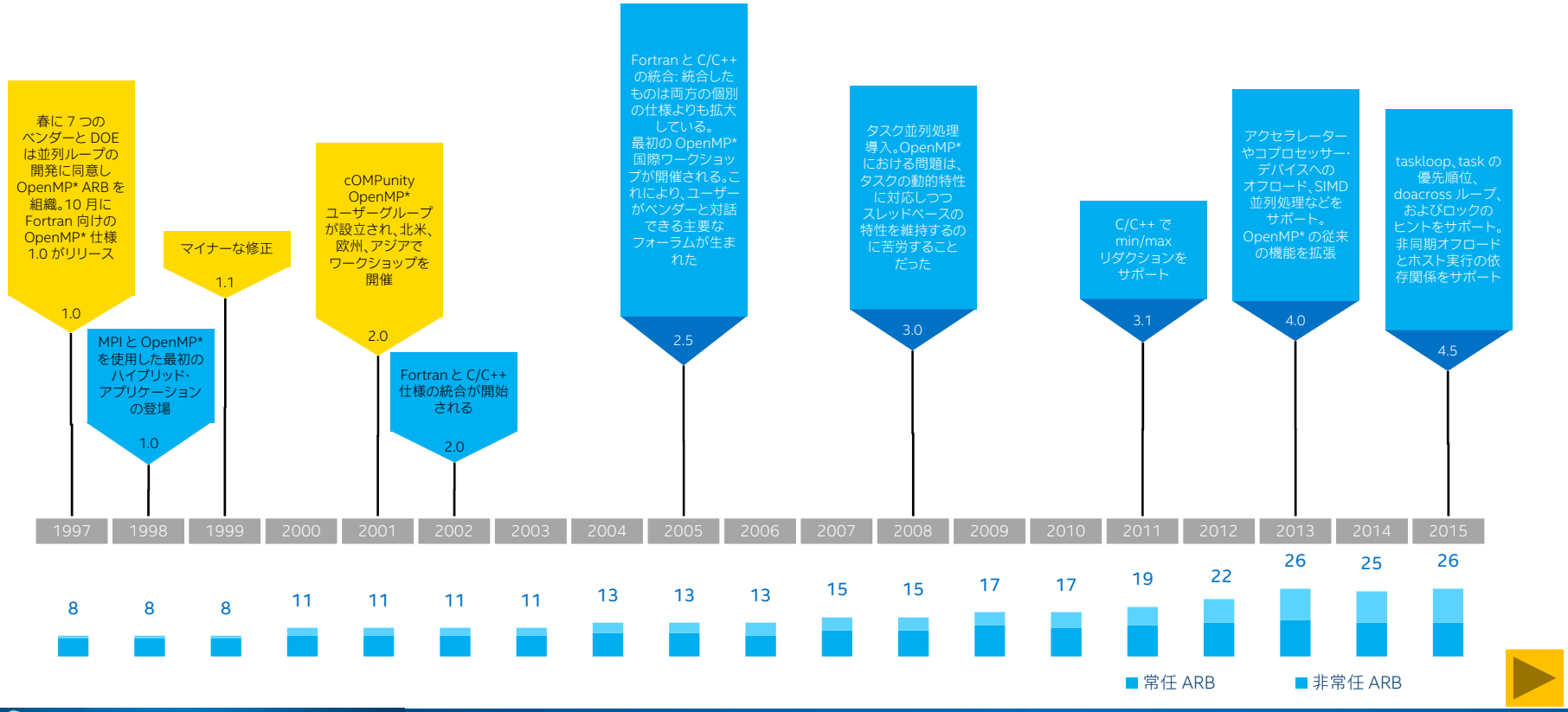
- スレッド管理
- ワークシェア
- タスク処理
- アフィニティー
- アクセラレーター
- デバイスメモリー
- キャンセル
- ロック

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* の歴史



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 TR6	201611	インテル® コンパイラー V19.0
OpenMP* 5.0 TR6	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 V15.0 & V16.0 、gcc 4.9 (offload 5.1)
OpenMP* 3.1	201107	インテル® コンパイラー V12.0、V13.0、V14.0、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

https://www.openmp.org/resources/openmp-compilers-tools/

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。

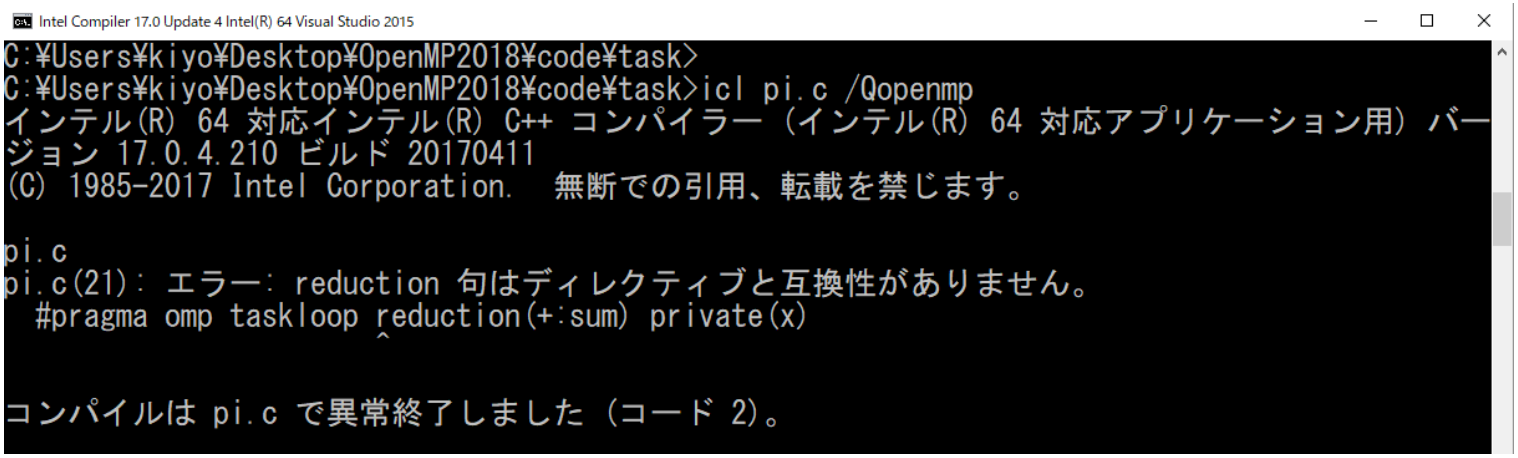
#pragma omp for simd [節[[,]節] ...]
for ループ

節

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

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

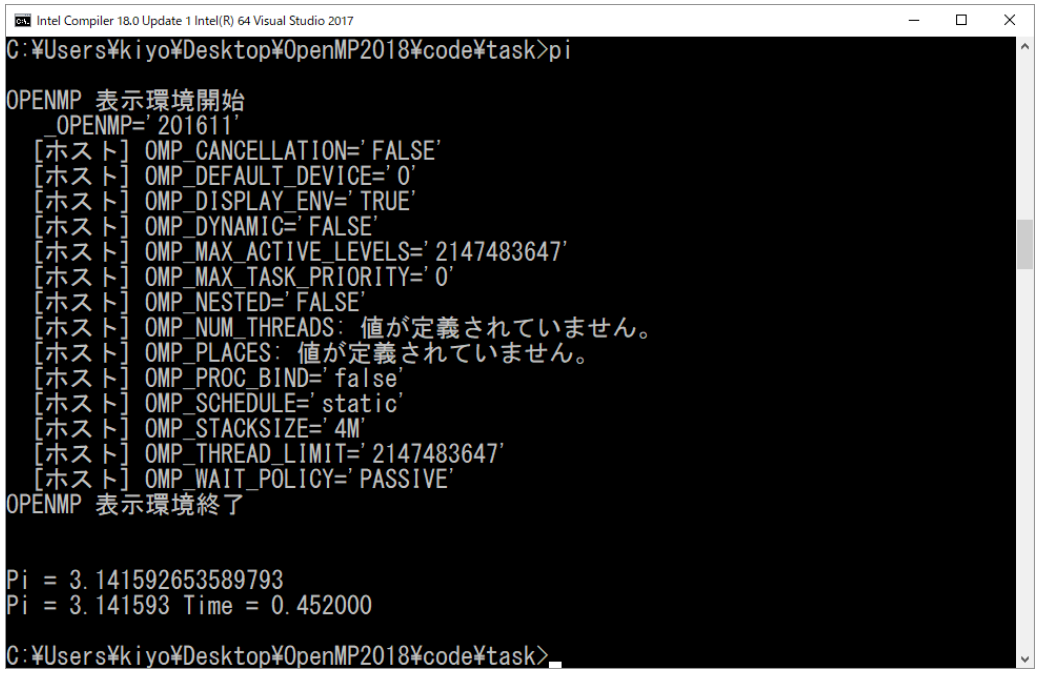


_OPENMP マクロでコンパイルを制御:

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

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



true、false (デフォルト)、verbose が設定できます

S

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

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、4.5 の機能と5.0 の新機能
  - タスク
  - omp simd
  - オフロード
  - メモリー操作
  - アフィニティー

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

task、taskloop、taskgroup

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

OpenMP* 4.5 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。

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

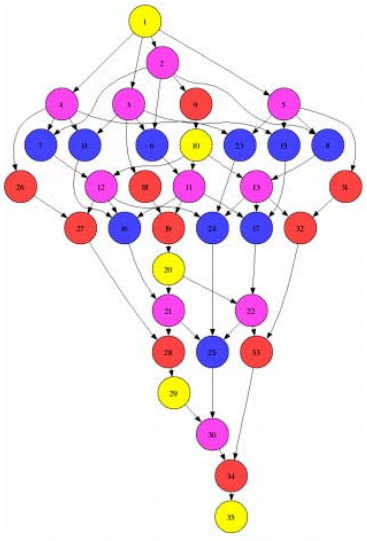
```
int val=0;

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」

OpenMP* 4.0 の機能

```
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

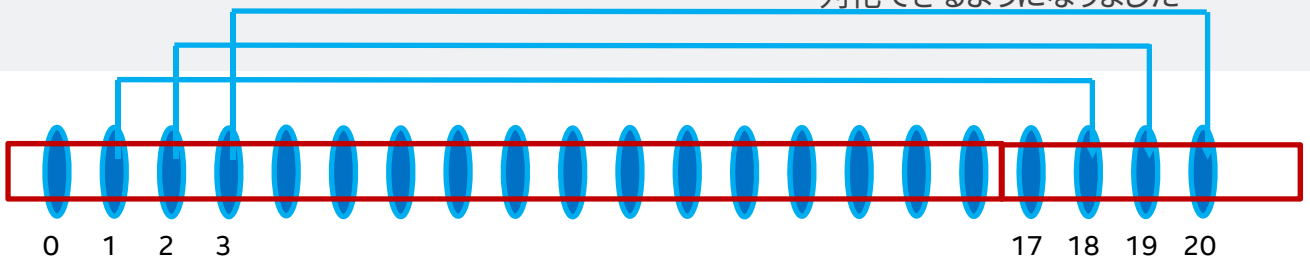
Do-across ループ並列

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)
    }
}
```

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



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

4.5 ではまだ気軽に 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 (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

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

OpenMP* 4.0 の機能

```
#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?

OpenMP* 4.5 の機能

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

taskloop 構文中での collapse 節

OpenMP* 4.5 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。

OpenMP* 5.0 の task/taskloop/taskwait 機能強化

- task、taskgroup、taskloop 句でのリダクション演算のサポート
- task、taskgroup、taskloop 句に allocate 節が指定できるようになりました
- task 句に detach 節にが追加されました
- taskwait 句に depend 節が追加されました

OpenMP* 5.0 TR7 の仕様抜粋訳
インテル® コンパイラー 18.0 の OpenMP* 5.0 サポート
OpenMP* 5.0 TR6 (テクニカル・レポート 6) に含まれる新機能
現在と将来の OpenMP* API 仕様

OpenMP* 5.0 の task 構文

#pragma omp task [clause[[,] clause] …]
　　構造化ブロック

if([task :] 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(依存性タイプ :リスト)
priority(優先値)

in_reduction(リダクション識別子:リスト)
allocate([アロケーター:]リスト)
affinity([アフィニティー修飾子 :] ロケーターリスト)
detach(イベントハンドラー)

- アフィニティー修飾子 は iterator(*iterators-definition*) です
- イベントハンドラーは omp_event_t * 型です

allocate、affinity、detach は、V19 では未サポート

OpenMP* 5.0 の taskloop 構文

#pragma omp taskloop [clause[[,] clause] …]
構造化ブロック

if ([taskloop :] スカラー式)
shared (リスト)
private (リスト)
firstprivate (リスト)
lastprivate (リスト)
default (shared | none)
num_tasks (タスク数)
collapse (n)
final (スカラー式)
priority (優先値)
untied
mergeable
nogroup

reduction (リダクション識別子:リスト)
in_reduction (リダクション識別子 :リスト)
allocate ([アロケーター:]リスト)
nogroup

allocate は、V19 では未サポート

OpenMP* 5.0 の taskgroup/taskwait 構文

#pragma omp taskgroup [clause[[,] clause] …]
　　構造化ブロック

4.5 ではオプションの節は無し	task_reduction (リダクション識別子:リスト) allocate ([アロケーター:]リスト)

allocate は、V19 では未サポート

#pragma omp taskwait [clause[[,] clause] …]

4.5 ではオプションの節は無し	depend ([depend 修飾子:][依存関係タイプ] 位置リスト)

depend は、V19 では未サポート

OpenMP* 5.0 TR7 の detach 節

OpenMP* 5.0 の機能

- detach 節が task 構文に指定されると、omp_event_t 型の新しいイベント *allow-completion-event* が作成されます。*allow-completion-event* は、関連するタスク領域の完了に関連付けられます。元のイベントハンドラーは、タスクデータ環境が作成される前に *allow-completion-event* を指すように更新されます。イベントハンドラーは、firstprivate 節で指定されたものとみなされます。task 構文の detach 節の式で使用された変数は、その task 構文内のすべての構文でその変数が暗黙的に参照されます。

- detach 節が task 構文に指定されていない場合、関連する構造化ブロックの実行が完了すると生成されたタスクも完了します。detach 節が task 構文に存在する場合、タスクは関連する構造化ブロックが完了し、*allow-completion-event* が満たされた時に完了します。

関連するイベントルーチン:
void omp_fulfill_event(omp_event_t *event_handler);

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に taskloop を適用するのは困難でした

```
#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

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;
```

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;
}
```

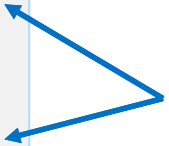
in_reduction が指定されないタスクの変数はリダクションに参加しません

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

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
V19 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

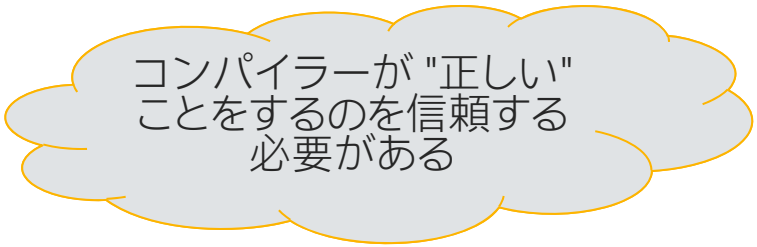
明示的な SIMD プログラミング･モデル

なぜ SIMD 拡張? OpenMP* 4.0 以前

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] + ...;
}
```



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/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!

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

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

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

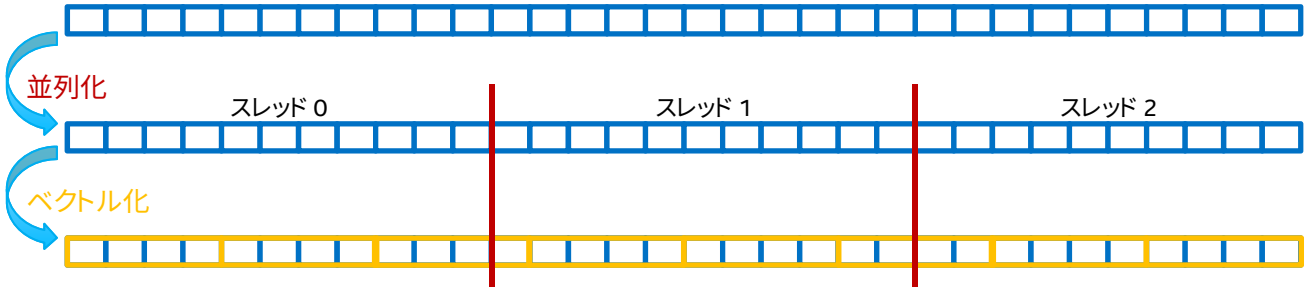
```
void vec1(float *a, float *b, int off, int len){
#pragma omp simd safelen(32) aligned(a:64, b:64)
    for(int i = 0; i < len; i++){
        a[i] = (a[i] > 1.0) ?
            a[i] : b[i];
            a[i + off] * b[i];
    }
}
```

```
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

OpenMP* 4.5 の機能

```
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

OpenMP* 4.0 の機能

```
#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)
```

OpenMP* 4.0 の機能

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

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

OpenMP* 4.0 の機能

- 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

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;
    }
}
```

内部ループから外部ループのベクトル化

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!

OpenMP* 5.0 の simd 構文

OpenMP* 5.0 の機能

#pragma omp simd *[clause[[,] clause] …] 改行*
　　構造化ブロック

safelen (*レングス*)
simdlen (*レングス*)
linear (リスト*[: リニアステップ]*)
aligned (リスト*[: アライメント]*)
private (リスト)
lastprivate (リスト)
reduction (リダクション識別子: リスト)
collapse (*n*)

if (*[*simd :*] スカラー式*)
nontemporal (リスト)
order (concurrent)

- nontemporal 節は、リスト項目のス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
- order (concurrent) 節が存在する場合、order(concurrent) 節を持つループ構文すべての制限も適用されます。

- simd 構文内で atomic 構文ができるようになりました

OpenMP* 5.0 の atomic 構文の拡張

OpenMP* 5.0 の機能

#pragma omp atomic [clause[[[,] clause] ...] [,]] atomic-clause
[[,] clause [[[,] clause] ...]] 改行
#pragma omp atomic [clause[[,] clause] ...]改行
#pragma omp atomic [clause[[[,] clause] ...] [,]] capture
[[,] clause [[[,] clause] ...]] 改行

atomic-clause: read、write、update、capture
memory order clause: seq_cst、acq_rel、release、acquire、relaxed
clause: hint (ヒント式)

loop 構文の追加

OpenMP* 5.0 の機能
V19.0 では未サポート

#pragma omp loop *[clause[[,] clause] …] 改行*
　　do/for ループ

bind (*binding*)
collapse (n)
order (concurrent)
private (list)
reduction (reduction-identifier : list)

binding には、teams、parallel、thread、binding のいずれかを指定します

loop 構文は、ディレクティブに続く 1 つ、または複数のループから成る入れ子のループに関連付けられます

入れ子構造のループ

```
#pragma omp parallel for
for (int i=0; i<MAX; i++) {
#pragma omp parallel for
      for (int j=0; j<MAX; j++){
        ……
      }
}
```

parallel 構文内の parallel 構文は、デフォルトではシングルスレッドで実行。OMP_NESTED=1 でマルチスレッドで実行されるが、オーバーサブスクライブとなります

OpenMP* 5.0 では、OMP_NESTED が非推奨となり入れ子構造の並列ループを制御するため、loop 構文が追加されました。

OpenMP* 5.0 の機能

scan ディレクティブの追加

```
#pragma omp for / for simd / simd ディレクティブ
for ループ･ヘッダー {
        構造化ブロック
#pragma omp scan [inclusive, exclusive] 改行
        構造化ブロック
}
```

scan ディレクティブは、ワークシェア･ループ、ワークシェア･ループ SIMD または simd 構造に関連するループ、または入れ子のループ本体で指定でき、ループで 1 つ以上のスキャン計算が行われることを指定できます

OpenMP* 5.0 で追加された節

#pragma omp parallel #pragma omp sections #pragma omp single	allocate ([アロケーター:] リスト)
#pragma for	allocate ([アロケーター:] リスト) linear (リスト [: リニアステップ]) order (concurrent)

OpenMP* 5.0 機能のまとめ

OpenMP* 5.0 は TR7 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、V19 で OpenMP* TR6: プレビュー 2 を実装

Part 2 では、オフロード、メモリー操作、アフィニティーなどについて説明します

参考資料

- OpenMP* 5.0 TR7 の仕様抜粋訳 (4.5 から 5.0 仕様への追加機能)
- 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


iSUS