OpenMP* 4.x における拡張

ベクトル化

2016 年 1 月

sugatech

内容

- 配列セクション (配列表記)
- SIMD 構文による明示的なベクトル化

内容

- 配列セクション

配列セクション

配列セクションでは配列内の部分要素を指定でき、配列セクションの利用が許される句で利用できます。インテル® Cilk™ Plus の配列表記のサブセット

文法: [下限：レングス]、[下限：]、[：レングス]、[：]

- 配列セクションは、元の配列の一部でなければいけません
- 多次元配列でも利用できます
- ベース言語の配列添字式を適用できます
- 下限とレングスは整数式で次のように評価されます:
  [下限, 下限 + 1, 下限 + 2, …, 下限 + レングス -1]
- 下限とレングスは + の整数でなければいけません
- 配列の次元サイズが不明である場合、レングスを指定する必要があります
- レングスが省略された場合、(配列サイズ – 下限) と評価されます
- 下限が省略されると、デフォルトで 0 と評価されます

配列セクション

インテル® Cilk™ Plus の配列表記との違い:

インテル® Cilk™ Plus の配列表記	OpenMP* 4.0 の配列セクション
[下限： レングス：ストライド]	[下限： レングス]
サポート関数あり (__sec_reduce_add(a[:]) など)	サポート関数なし

Array.c

使い方の例:

e[:] = f[:] + g[:]　　(配列、ヒープ、またはスタック全体)

r[:] = s[i[:]]、r[i[:]] = s[:]　　(gather, scatter)

func(a[:])　　(scalar/simd 対応関数への引数)

if(5 == a[:]) result[:] = 0　　(条件式で利用可能)

内容

- 配列セクション
- SIMD 構文による明示的なベクトル化
  - 概要と宣言文

SIMD 構文による明示的なベクトル化

- ベクトル化として知られる
- Speed Kills
  - 90 年代に Cray のベクトルの歴史を変えたのは、マイクロプロセッサーのスピードだった
  - 私たちは、ベクトルを利用する歴史を再び繰り返します
  - マイクロプロセッサーのベクトルは、長い間 2 DP (倍精度)
- 私たちは、いまパラレルワールドに暮らしている
  - それは、SIMD による並列性だけでなく、スレッドと MPI タスクによる並列処理も含む

これらをどう判断すべきか？

SIMD レジスター幅は広まっているが、そのほかに考慮すべき事がある

- キャッシュ: おそらくコヒーレントではない、 数階層先のメモリーの可能性
- アライメント: キャッシュからレジスターへの非効率な転送を避ける
- プリフェッチ: MIC はプログラマーの介入を必要とする
- データの再配置 AoS と SoA、gather、scatter、permute
- マスク: 条件付き実行を許すが、非効率になる恐れがある
- ストライド: 1 が最良

コンパイラーは支援を必要とする

SIMD 宣言文

コンパイラーにベクトル化のための情報を提供する

- 演算の独立性を保証
- 「指示どおりにしなさい」 という強制宣言

インテル® Cilk™ Plus

- SIMD 宣言文
  - ループ
  - C 配列
  - 関数
  - C/C++ のみ

OpenMP*

- SIMD 宣言文
  - ループ
  - C 配列なし
  - 関数
  - C/C++ と Fortran

OpenMP* SIMD の進化

- 2013 年に OpenMP 4.0 で初めて実装
- 宣言文
  - SIMD
  - SIMD do/for
  - declare SIMD
- OpenMP 4.5 (2015 年 11 月) において、一部を改良

内容

- 配列セクション
- SIMD 構文による明示的なベクトル化
  - 概要と宣言文
  - なぜ必要なのか?
  - SIMD は OpenMP* の進化形
  - インテル® Cilk™ Plus と OpenMP*
  - SIMD 構文の使い方と例

SIMD プラグマが必要な理由

独立性のないループはベクトル化されない

- ベクトル化が失敗する原因: 多くのポインター、複雑なインデックス ... (ivdep はヒント)
- SIMD pragma は、コンパイラーにループ反復を SIMD 操作するように強制

#pragma なしで、/Qopt-report=2 (Windows*)、 -qopt-report=2 (Linux*) でレポートを取得:
「remark #15541: 外部ループは自動ベクトル化されませんでした: SIMD 宣言文の使用を検討してください」

```
void do2(double a[n][n], double b[n][n], int end){
#pragma SIMD または #pragma omp SIMD
    for (int i=0 ; i<end ; i++) {
       a[i][0] = (b[i][0] - b[i+1][0]);
       a[i][1] = (b[i][1] - b[i+1][1]);
   }
}
```

ivdep と vector always は、ここでは作用しない

インテル® Cilk™ Plus → OpenMP* 対応機能

インテル® Cilk™ Plus

- SIMD (ループ)
  - reduction
  - vectorlength
  - linear (インクリメント)
  - private、lastprivate

OpenMP*

- SIMD (ループ)
  - reduction
  - safelen
  - linear (インクリメント)
  - private、lastprivate

例 (Fortran)

```
!dir$ simd reduction(+:mysum) linear(j:1) vectorlength(4)
      do…; mysum=mysum+j; j=fun(…); enddo
```

```
!$omp simd reduction(+:mysum) linear(j:1) safelen(4)
      do…; mysum=mysum+j; j=fun(…); enddo
```

インテル® Cilk™ Plus – OpenMP* SIMD 違い

インテル® Cilk™ Plus

- SIMD (ループ)
  - firstprivate
  - vectorlengthfor
  - [NO]VECREMAINDER
  - [no]assert
- #pragma cilk grainsize

OpenMP

- SIMD (ループ)
  - aligned(var_list,bsize)
  - collapse
  - schedule(kind, chunk)
- #pragma taskloop simd*

*インテル® コンパイラー V16 では未サポート

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];
}
```

add_sum.cpp

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

自動ベクトル化は言語規則によって制限される： 意図することを表現できない

SIMD プラグマ/宣言文による
明示的なベクトル・プログラミング

プログラマーの主張：

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

add_sum.cpp

- *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

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！

SIMD 構文の表記

OpenMP* 4.0

#pragma omp simd [節[[,] 節]…] **C/C++**

!$OMP SIMD [節[[,] 節]…] **Fortran**

ループ指定

- 内側か外側のループを指定できる

結果は開発者が保証しなければならない

- 開発者は、ループが SIMD に適していることを明示
  - ループ伝搬依存が無く、反復は並列に評価できること
- SIMD プラグマ/宣言子の振る舞いを変更するため、節を選択できる
- 開発者は結果を評価しなければならない

OMP SIMD の節

reduction(operator:v1, v2, …)

- v1、v2 … は、 operator で操作するリダクション変数
- 配列の平均値や総和を単一のスカラー変数に求める場合など： *reduction (+:sum)*

linear(v1:step1, v2:step2, …)

- SIMD レーンでプライベートにする 1 つ以上の項目とループ反復空間に対してリニアな関係を持つことを宣言：*linear (i:2)*

safelen(*length*)

- 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する
- 一般的な値は、 2、4、8、もしくは 16

OpenMP 4.5 仕様を参照：http://www.openmp.org/mp-documents/openmp-4.5.pdf

OMP SIMD の節（続き）

aligned(v1:alignment, v2:alignment)

- 各リスト項目（v1、v2) が、オプションのパラメーター（alignment) で指定されたバイト数でアライメントされていることを宣言する

collapse(n)

- 入れ子になった n 個のループ構造を、より大きな 1 つのループに畳み込むことを指示

private(v1, v2, …), lastprivate (v1, v2, …)

- 暗黙のタスクもしくは SIMD レーンでプライベートにする 1 つ以上の項目を宣言する。lastprivate 節では、領域終了後に指定された項目が更新される

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;
}
```

add_sum2.cpp

- += 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

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;
}
```

add_sum2.cpp

- += 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

OMP SIMD 構文の制限事項

omp simd 構文の制限事項（すべてが記載されていない）：

- for / do ループにのみ適用
- インダクション変数は、符号あり/なしの int のみ
- 関連するループは、構造化ブロックであること
- SIMD 領域の内側から外側へ、または外側から内側へ分岐するプログラムはサポートされない
- OpenMP* 構文は SIMD 領域内に記述できない
- ループ本体は、C++ 例外と Windows* 構造化例外処理、**setjmp(…)** & **longjmp(…)** が在ってはならない

演習

- SIMD1
- MATMUL

内容

- 配列セクション
- SIMD 構文による明示的なベクトル化
  - 概要と宣言文
  - なぜ必要なのか?
  - SIMD は OpenMP* の進化形
  - インテル® Cilk™ Plus と OpenMP*
  - SIMD 構文の使い方と例
  - SIMD 対応関数
  - SIMD 対応関数の使い方と例

SIMD 対応関数の概念

- スカラー構文は単一要素の操作を記述できる
- 開発者は：
  - スカラー値を操作する標準的な関数を記述
  - 関数にベクトル属性と修飾子を注釈
    #pragma omp declare simd や **!$OMP DECLARE SIMD**
    - ベクトル属性を示すため適切な修飾子を利用する
  - スカラー引数よりも、引数の配列を操作する関数呼び出しを行う

- コンパイラーは：
  - スカラーとベクトルバーションのコードを生成
  - ベクトル化されたループからベクトル関数を呼び出すことができる
  - スカラーループからスカラー関数を呼び出すことができる

SIMD 対応関数

- SIMD 化可能な関数

```
double fun1(double r, double s, double t);
double fun2(double r, double s, double t);

…

void driver (double R[N], double S[N], double T[N]){
  for (int i=0; i<N; i++){
    A[i] = fun1(R[i],S[i],T[i]);
    B[i] = fun2(R[i],S[i],T[i]);
  }
}
```

インテル® Cilk™ Plus の SIMD 対応関数

- スカラーまたはベクトル引数で呼び出し可能
- SIMD バージョンと共に配列表記を使用 (ベクトル幅に最適化)

```
                    **
__declspec(vector) double fun1(double r, double s, double t);
__declspec(vector) double fun2(double r, double s, double t);

    // 要素を処理する関数;
…   // 並列コンテキスト (Cilk Plus) では、ベクトル版の配列を提供
void driver (double R[N], double S[N], double T[N]){
    A[:]= fun1(R[:],S[:],T[:]);
    B[:]= fun2(R[:],S[:],T[:]);
}
```

配列表記 (アレイ・ノーテーション)

** または __attribute__((vector))

インテル® Cilk™ Plus で有効、omp declare 違い

OpenMP* :	#pragma omp declare simd <節>	C/C++
	!$OMP DECLARE SIMD <節>	Fortran
インテル® Cilk™ Plus :	__declspec(vector) <節>	

インテル® Cilk™ Plus

- vector 節
  - vectorlength
  - linear
  - uniform
  - [no]mask
  - processor(cpuid)
  - vectorlengthfor

OpenMP*

- declare simd 文
  - simdlen
  - linear
  - uniform
  - inbranch/notinbranch
  - aligned

SIMD 対応関数

- 1 要素を処理する関数を記述し、次のように **pragma** を記述する

```
#pragma omp declare simd
float foo(float a, float b, float c, float d)
{
  return a * b + c * d;
}
```

- スカラーバージョンの呼び出し：

```
e = foo(a, b, c, d);
```

- ベクトル・バージョンを SIMD ループから呼び出す：

```
#pragma omp simd
for(i = 0; i < n; i++) {
  A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

- インテル® Cilk™ Plus 配列表記から呼び出す：

```
A[:]= foo(B[:], C[:], D[:], E[:]);
```

dec_simd.c

SIMD 対応関数の概念

- スカラー構文は、単一要素の操作を記述できる
- 開発者は：
  - スカラー値を操作する標準的な関数を記述
  - 関数にベクトル属性と修飾子を注釈
    #pragma omp declare simd
    - ベクトル属性を示すため適切な修飾子を利用する
  - スカラー引数よりも、引数の配列を操作する関数呼び出しを行う

- コンパイラー：
  - スカラーとベクトルバーションのコードを生成
  - ベクトル化されたループからベクトル関数を呼び出すことができる
  - スカラーループからスカラー関数を呼び出すことができる

SIMD 対応関数：構文

OpenMP 4.0

#pragma omp declare simd [節[[,] 節]…]　　C/C++
!$OMP DECLARE SIMD [節[[,] 節]…]　　Fortran

- ベクトル句は、関数のすべての引数がベクトルとして扱われ、戻り値もベクトルとして扱われることを意味する。これは、デフォルト動作
- 開発者は、修飾子を指定することでデフォルトの動作を変えることができる
- 関数プロトタイプやヘッダーファイルにも simd 対応宣言子を追加することを推奨

omp declare simd の節

オプションの節：

***uniform**(param1[, param2]…)*
共有、スカラー引数は全ての反復にブロードキャストされる

***linear**(param1:step1[, param2:step2]…)*
シリアル実行中、指定するパラメーターは step 分だけインクリメントされる。例としては、一定の間隔を持つ誘導（インダクション）変数

***simdlen**(num)*
コンパイラーが判断して使用できるベクトルの最大サイズ。
通常は、2、4、8、もしくは 16

aligned(argument-list[:alignment])
argument-list 中のすべての argument は、少なくとも指定されたアライメントで配置される

OpenMP 4.5 仕様を参照：http://www.openmp.org/mp-documents/openmp-4.5.pdf

SIMD 対応関数：Linear/ Uniform の必要性

- なぜそれらが必要なのか?
- uniform もしくはlinear が省略されると、各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

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);
#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

dec_simd3.c

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

呼び出し側

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

ベクトル化レポート

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

SIMD 対応関数を使用する際の制限事項

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

内容

- 配列セクション
- SIMD 構文による明示的なベクトル化
  - 概要と宣言文
  - なぜ必要なのか?
  - SIMD は OpenMP* の進化形
  - インテル® Cilk™ Plus と OpenMP*
  - SIMD 構文の使い方と例
  - SIMD 対応関数
  - SIMD 対応関数の使い方と例
  - SIMD 構文とスレッド

SIMD とスレッド – インテル® Cilk™ Plus

- Cilk の “los tres amigos (3人の友達)”
  - cilk_for
  - cilk_spawn
  - cilk_sync
- Cilk Plus ループは SIMD 化され、複数のスレッドで実行される
- Cilk Plus ループ内では SIMD 対応関数を呼び出す

SIMD とスレッド – OpenMP* ワークシェア

OMP 宣言文　ワークシェアと SIMD ループ

- ベクトルサイズでインクリメントするチャンクの SIMD ループを生成
- 余剰ループは一貫性を持つように処理される
- スケジュールの詳細は与えられない

シンタックス：　　複合宣言文

!$OMP DO	SIMD <節>	F90
#pragma omp for	simd <節>	C/C++

節: 任意の do/for 文のデータ共有属性、nowait など、任意の SIMD 句

!$OMP TASKLOOP	SIMD <節>
#pragma omp taskloop	simd <節>

タスクによる新しいスケジュール

SIMD のまとめ

インテル® Cilk™ Plus

- C 配列
- タスクスレッドで動作 (Fortran ループにはない)
- SIMD ループと関数 (より適した節がある)
- prefetch 命令がある

OpenMP*

- C 配列なし
- ワークシェアスレッド (C/C++, Fortran) で動作
- SIMD ループと関数 (より適した節がある)
- prefetch 命令はない (インテル® コンパイラーの命令を利用可能)

参考サイト

インテル® ソフトウェア・フォーラム、ナレッジベース、記事、ツールのサポート
(http://software.intel.com 参照、http://isus.jp 翻訳版)

記事の例:
- http://www.isus.jp/article/parallel-special/requirements-for-vectorizable-loops/
  (ループをベクトル化するための条件)

OpenMP* 4.5 Specification
- http://www.openmp.org 5 /mp-documents/openmp-4.5.pdf

OpenMP* 3.1 仕様をカバーするオンライン・トレーニング
- http://www.isus.jp/online-training/

関連書籍

Structured Parallel Programming: Patterns for Efficient Computation

著者 Michael McCool, James Reinders ,Arch Robison 出版日: 2012 年 7 月 9 日 | ISBN: 978-0-124159938

『構造化並列プログラミング: 効率良い計算を行うためのパターン』

著者　マイケル・マックール／アーク・D・ロビソン／ジェームス・レインダース（共著）

訳者　菅原 清文／エクセルソフト株式会社（共訳） | ISBN　978-4-87783-305-3

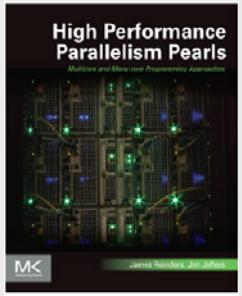
Intel® Xeon Phi™ Coprocessor High Performance Programming

著者 Jim Jeffers , James Reinders 出版日: 2013 年 3 月 | ISBN: 978-0-124104143

『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』

著者　ジェームス・レインダース/アーク・D・ロビソン（共著）

訳者　菅原 清文／エクセルソフト株式会社（共訳） | ISBN　978-4-87783-332-9



High Performance Parallelism Pearls

著者 Jim Jeffers , James Reinders　出版日: 2014 年 11 月

簡単にインテル® Xeon Phi™ コプロセッサー・ファミリーの優れた並列性を利用してコードを実行できるため、最適化に集中し、ハイパフォーマンスを実現することが可能です。並列処理を細かくチューニングすることで、正しいアプリケーションを正しく効率良いアプリケーションにすることができます。インテル コーポレーションの並列プログラミング・エバンジェリストである James Reinders とインテル コーポレーションのエンジニアである Jim Jeffers により執筆された最新の書籍は、69 人の専門家の実際の経験を基に、インテルのマルチコアおよびメニーコア・プロセッサーを最大限に利用するための創意工夫を紹介しています。

sugatech

sugatech