インテル® コンパイラーを使用した OpenMP* による並列プログラミング

セッション 3: OpenMP* の SIMD 機能

IA Software User Society (iSUS)
編集長　すがわら　きよふみ

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 V19.1 でサポートされる OpenMP* 5.0 の機能と実装を紹介します。さらに新たなアクセラレーター・デバイスへのオフロードについて考えます

セッションの対象者

すでに 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

セッションリスト

セッション	説明
はじめに (13:35 – 14:20)	異なるバージョンのインテル® コンパイラーや異なるコンパイラー間で OpenMP* を使用する注意点や制限について説明します
OpenMP* のタスク機能 (14:30 – 15:30)	OpenMP* 3.1 で追加されたタスク機能が 4.0 から 4.5 でどのように進化したかを例を使用して説明し、最新の OpenMP* 5.0 で強化された新機能を紹介します
OpenMP* の SIMD 機能 (13:30 – 14:30)	OpenMP* のスレッド化機能を使用してプログラマーがマルチスレッドの動作をプログラミングしたように、OpenMP* 4.0 からは omp simd を使用してプログラマーが明示的にベクトル化もできるようになりました。OpenMP* simd に関連する機能を 4.0 から 5.0 までの進化を追って紹介します
OpenMP* のオフロード機能 (14:30 – 15:30)	OpenMP* 4.0 で追加されたオフロード機能を利用することで、これまで共有メモリー型並列処理に加え分散メモリー型の並列処理を表現できるようになりました。このセッションでは、注目されるヘテロジニアス・プログラミング環境での OpenMP* オフロード機能について説明します
OpenMP* 5.0 の注目する機能	セッション2、3、4でカバーされなかった OpenMP* 5.0 のそのほかの機能について紹介します
インテル® C++/Fortran コンパイラーのバージョン 19.1 を使用して GPU オフロードに備えましょう	oneAPI 向けのデータ並列 C++ (DPC++) へ移行する前に、現行のインテル® C++/Fortran コンパイラー V19.1 やインテル® oneAPI HPC ツールキットに含まれるベータ版インテル® C++/Fortran コンパイラー 2021 を使用して簡単にインテル® グラフィックスへのオフロードを行うソフトウェアを開発および検証方法を紹介します

内容

- はじめに (OpenMP* が必要とされる背景) と概要 (OpenMP* とは、歴史、各バージョンの機能概要)
- OpenMP* の各バージョンの機能 (4.0、4.5 および 5.0 の注目される新機能)
- 次世代インテル® コンパイラー (nextgen) の機能

OpenMP* 5.0 API シンタックス・クイック・リファレンス・カードの日本語訳を公開しました:
https://www.isus.jp/products/c-compilers/openmp-ref-5-0-0519-released/

OMP SIMD の補足セッション:
https://www.gotostage.com/channel/f4ff4cee166845b8acf8364e833ab925/recording/5527731d1dfd4a86aeff8f43c153a03a/watch

内容

- OpenMP* の各バージョンの機能
- OpenMP* 4.0 と 4.5、および 5.0 の新機能
  - タスク
  - SIMD
  - オフロード
  - OpenMP* 5.0 の注目する新機能

omp simd と自動ベクトル化の関係

omp simd を指定しなくても自動ベクトル化が適用されることがあります

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

/Qopt-report3 /Qopt-report-phase:vec,openmp で確認

オプション	ベクトル化
なし	SIMD ループ がベクトル化されました
/Qopenmp	SIMD ループ がベクトル化されました
/Qopenmp /Qopenmp-simd-	ループがベクトル化されました
/Qopenmp-simd-	ループがベクトル化されました
/Qvec-	SIMD ループ がベクトル化されました
/Qvec- /Qopenmp-simd-	ループ はベクトル化されませんでした
/Qopenmp /O1	ルーチンはスキップされました: ループの最適化が無効になりました

- インテル® コンパイラーでは、/Qopenmp が指定されなくても omp simd はデフォルトでオンになります
- /O2 以上でベクトル化が有効になります

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 “期待する” ことを行うのを信頼する必要がある

ベクトル化に影響するプログラムの要素

ループ伝搬依存

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

関数呼び出し

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

ポインター・エイリアシング

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

さらに....

不明なループカウント

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

間接メモリーアクセス

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

外部ループ

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

自動ベクトル化: シリアル・セマンティクスによる制限

コンパイラーは以下をチェックする:

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

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

自動ベクトル化は言語規則によって制限されます:
意図することを表現できません

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

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];
}
```

- *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ます!

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

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

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)

構文:

```
#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
- SIMD 構造内には並列構造を記述できません

例えば …

OK

```
#pragma omp parallel for
#pragma omp simd
      for ループ
```

```
#pragma omp parallel for simd
      for ループ
```

```
#pragma omp simd
      for ループ
```

エラー

```
#pragma omp parallel simd
#pragma omp for
      for ループ
```

```
#pragma omp parallel simd for
      for ループ
```

```
#pragma omp simd
      for ループ構造以外
```

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

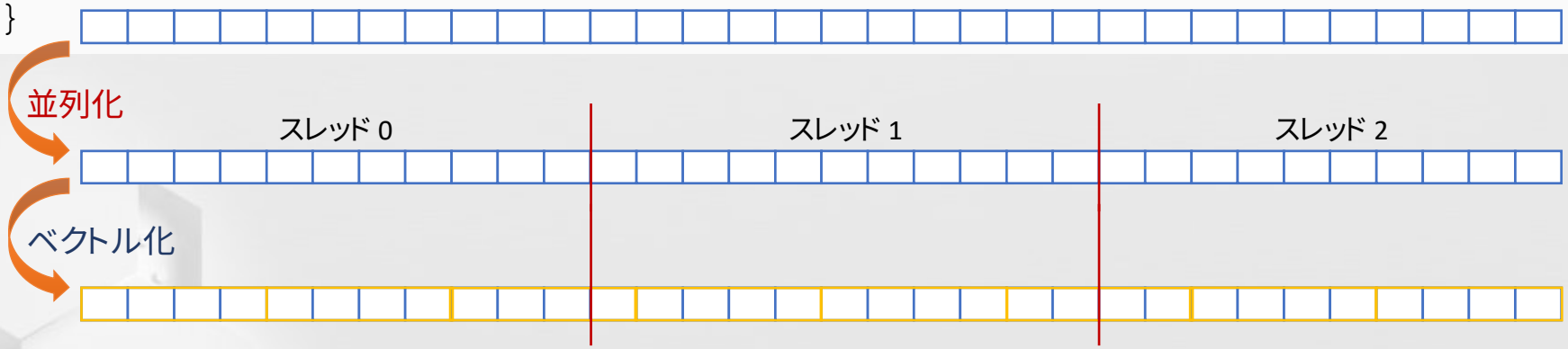
データの依存性と間接的な制御フローの依存性がないことを明示してアライメントを指示

```
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

```
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します

- 新しいチャンクサイズは、*[chunk_size/simdlen]* simdlen*

17

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します [OpenMP* 4.0 の API: 2.8.2]

構文:

```
#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することを指示します。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ます (インテル® SSE、インテル® AVX、インテル® AVX-512)

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

SIMD 対応関数の節

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

OpenMP*: SIMD 対応関数のベクトル化

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

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

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
    }
}
```

```
vec8 min_v(vec8 a,vec8 b){

  return a < b ? a : b;
}
```

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

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

SIMD 対応関数: Linear/Uniform

- なぜ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**)

```
#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 (英語)

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 (英語)

SIMD 対応関数: 呼び出しの依存性

呼ばれる側

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

呼び出し側

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

ベクトル化レポート

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

SIMD 対応関数: 複数のベクトル定義

呼ばれる側

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

呼び出し側

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

ベクトル化レポート

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

SIMD 対応関数を使用する際の制限事項

- 引数は 1 つの uniform または linear 句に記述できます
- 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;
    }
}
```

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

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 機能強化

V18 では未実装

- simd 構文に nontemporal 節が追加されました
  - nontemporal 節は、リスト項目のストレージ位置へのアクセスが、ループ反復間で時間的な局所性が低いことを指定します
- 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

OpenMP* 5.0 の declare Variant

- OpenMP* 4.5 では、複数バージョンの関数を作成できます

```
#pragma omp declare simd simdlen(4)
#pragma omp declare simd simdlen(8)
double important_stuff(double x)
{
  // 関数 'important_stuff'のコード
}
```

- declare variant はユーザー定義バージョンの関数を挿入できます

```
#pragma omp declare variant(int important_stuff(int x)) ¥
              match( context={simd(simdlen(4))}, device={isa(avx2)} )
__m256 _mm256_important_stuff(__m256 x);
{
     /*
    関数 'important_stuff' の特殊化されたコード
    インテル® AVX2 を実行するプロセッサーで呼び出されます
       */
}
```

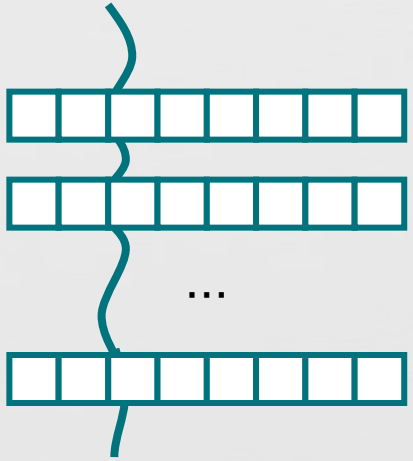
例: 実際の名前は異なることがあります

OpenMP* 5.0 の loop 構造

- 既存のループ構造は、それらの実行モデルと密接に関係します:

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


```
#pragma omp simd
for (i=0; i<N;++i) {…}
```




```
#pragma omp taskloop
for (i=0; i<N;++i) {…}
```

loop 構造は OpenMP* 実装が並列ループに適した並列化スキームを選択することを可能にします

OpenMP* 5.0 の loop 構造

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));

 // スカラー n、a、b を定義し、x、y を初期化

#pragma omp parallel
  {
#pragma omp loop
    for (int i = 0; i < n; ++i){
      y[i] = a * x[i] + y[i];
    }
  }
}
```

OpenMP* SIMD のまとめ

- OpenMP* のスレッド化機能を使用してプログラマーがマルチスレッドの動作をプログラミングしたように、omp simd を使用してプログラマーがベクトル化も行うことができます
- OpenMP* はもはやスレッド化のための業界標準ではありません
- これまで、コンパイラーが単独ではベクトル化できなかったアルゴリズムをベクトル化できるようになりました

iSUS

ソフトウェア・セミナー