



インテル・テクノロジー講座 ソフトウェア最適化方法論(2)

インテル株式会社

2017 年度HAL版

インテル・ソフトウェア教育カリキュラム

コースの内容

- SIMD 命令とベクトル化
- 並列化とOpenMP
- 最新プロセッサでのデモ
 - 最内ループをベクトル化して最外ループを並列化

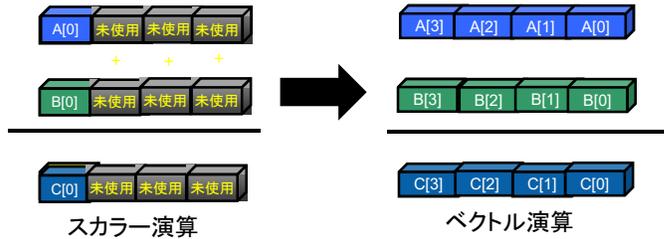


ベクトル化

ベクトル化とはインテル® マイクロプロセッサがサポートする、SIMD命令 (SSEやAVX命令等)を利用して、1つの命令で複数のデータを処理するようなコードを生成する最適化技術であり、多くの場合ループ内のデータ演算や操作に適用できる

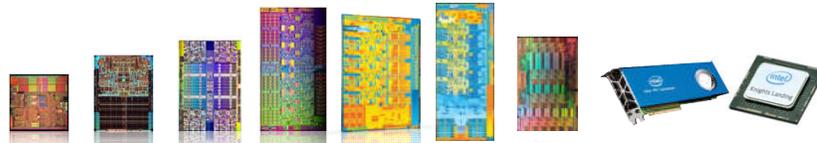
```
for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
```

ベクトル化は数値演算固有のものではない



より多くのコア、広いベクトル、コプロセッサ

パフォーマンスを得るには、ツールは 並列性を考慮しなければならない



イメージの大きさは実際のサイズとは異なります

	インテル® Xeon® プロセッサ 64 ビット	インテル® Xeon® プロセッサ 5100 シリーズ	インテル® Xeon® プロセッサ 5500 シリーズ	インテル® Xeon® プロセッサ 5600 シリーズ	Intel® Xeon® プロセッサ E5-2600 (Sandy Bridge)	Intel® Xeon® プロセッサ E5-2600 v2 (Ivy Bridge)	Intel® Xeon® プロセッサ E5-2600 v3 (Haswell)	Intel® Xeon Phi™ コプロセッサ 7100A	Knights Landing
コア数	1	2	4	6	8	12	14	61	>61
スレッド数	2	2	8	12	16	24	28	244	>240
SIMD 幅	128	128	128	128	256	256	256	512	512
	SSE2	SSSE3	SSE4.2	SSE4.2	AVX	AVX	AVX2 FMA	IMCI	AVX512

ソフトウェアの挑戦: スケーラブルなソフトウェアを開発する



ベクトル命令の概要

ベクトル命令のフォーマット（ここではMASMの形式で表記する）

- 明示的に結果を送るレジスタを指定する 3 オペランド形式

instruction destination, source1, source2

- 入力レジスタの内容は破壊されない
- コードをコンパクトにできる

- (大概の) MIC 命令はマスクすることができる

instruction destination {mask}, source1, source2

→ マスクされた部分は非破壊的である、つまり、結果を送る先の値は保持される

- 例:

vaddps zmm1{k1},zmm2,zmm3



© 2013 Intel Corporation. 無断での引用、転載を禁じます。*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。

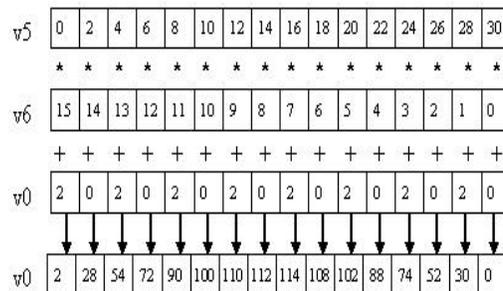
Optimization
Notice 123



Fused Multiply Add (乗加算)

Multiply-Add (デスティネーションは最初のソース)

- *Vfmadd231ps v0, v5, v6 ; v0=v5*v6+v0*
- オペランド 2 にオペランド 3 を掛けて、オペランド 1 に加算



© 2013 Intel Corporation. 無断での引用、転載を禁じます。*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。

Optimization
Notice 123



SIMD 命令によるベクトル化

ソフトウェアがプロセッサのリソースを効果的に使用しているかどうかを判断する

1. コンパイラーのレポートを解析し、最適化を妨げる要因を見つける
2. 適切なオプション、プラグマ、およびループ変換を利用する

一般的なループのベクトル化の問題

- 非依存性
 - ループの反復は依存性があってはならない
- ループに関する制限:
 - ループ内の変数は明確でなければならない
 - 一部の依存ループはベクトル化が可能
 - 多くの関数呼び出しはベクトル化できない
 - 一部の条件分岐はベクトル化を妨げる
 - ループはカウント可能でなければならない
 - ネストするループの外部ループはベクトル化できない
 - 混在データ型はベクトル化できない
 - その他

定義

- ループの非依存性(independency)とは:
ループの反復 i は反復 $i+1(i-1)$ に依存しない

```
int a[MAX], b[MAX];
for (i = 0; i < MAX; i++) {
    a[i] = b[i];
}
```

コンパイラによるベクトル化や並列化に影響

重要度

OpenMP*: 並列処理
自動並列化: 並列処理
インテル® Cilk™ Plus: 並列処理

} 影響大

SIMD: ベクトル化
(MMX、SSE、SSE2、SSE3、SSE4、AVX)
SWP: ソフトウェアのパイプライン化

OOO: アウトオブオーダー・コア

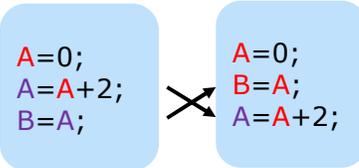
ILP: 命令レベルの並列処理

O3: ループ変換による最適化

} 影響あり

依存性とは何か

- 依存性は手続き型（インパラティヴ）言語の不可欠な要素
- 実行順序に結果が依存する！！
- 実行文間の関係
- ループに適用できる



11

インテル・ソフトウェア教育カリキュラム



フロー依存

- 書き込み後の読み取り (RAW)
- ループ間のフロー依存:
- 変数を書き込んでから別のループで読み取る

```
for (j=1; j<MAX; j++) {
    A[j]=A[j-1];
}
```

```
A[1]=A[0];
A[2]=A[1];
```

sample1.c

書き込みが完了しないと、依存性のある読み込みは開始できない

12

インテル・ソフトウェア教育カリキュラム



逆依存

- 読み取り後の書き込み (WAR)
- ループ間の逆依存:
- 変数を読み取ってから別のループで書き込む

```
for (j=1; j<MAX; j++) {
    A[j]=A[j+1];
}
```

```
A[1]=A[2];
```

```
A[2]=A[3];
```

sample2.c

読み込みが完了しないと、依存性のある次の書き込みは開始できない

出力依存

- 書き込み後の書き込み (WAW)
- ループ間の出力依存:
- 変数を書き込んでから別のループで同じ場所に再度書き込む

```
for (j=1; j<MAX; j++) {
    A[j]=B[j];
    A[j+1]=C[j];
}
```

```
A[1]=B[1];
```

```
A[2]=C[1];
```

```
↓
```

```
A[2]=B[2];
```

```
A[3]=C[2];
```

sample3.c

書き込みが完了しないと、依存性のある次の書き込みは開始できない

単純なテスト

ループの順序を反転して
再実行する

結果が同じ場合、ループは
独立している[§]

[§] 例外: 帰納変数を含むループ

```
for(j=0;j<MAX;j++){
  <...>
  compute(j,...)
  <...>
}
```



```
for(j=MAX-1;j>=0;j--){
  <...>
  compute(j,...)
  <...>
}
```

sample4.c

依存関係の排除

依存関係の排除は他の最適化
を改善するため、最良の選択

並列処理では必須

すべての依存関係を排除
できるわけではない

```
for (j=1; j<MAX; j++) {
  A[j]=A[j-1] + 1;
}
```



```
for (j=1; j<MAX; j++) {
  A[j]= A[0] + j;
}
```

sample5.c

リダクション

リダクションは、結合演算により配列データをスカラーデータに変換する

リダクションを利用して、プライベート領域の部分和を計算する

```
for (j=0; j<MAX; j++)
    sum = sum + c[j];
```

sample11.c

リダクション変数はループ内でフロー依存が発生するループを並列化するには排除しなければならない問題

次に、アクセスが同期するように注意しながら、部分的な結果を共有結果と組み合わせる

17

インテル・ソフトウェア教育カリキュラム



データのあいまいさとコンパイラー

- ループの反復は非依存か?
- 通常コンパイラーはこれを判別できない
- 最適化の余地はない
エラーなしで実行するために、コンパイラーは a と b のオーバーラップを仮定する

```
void func(int *a, int *b) {
    for (j=0; j<MAX; j++) {
        a[j] = b[j];
    }
}
```

18

インテル・ソフトウェア教育カリキュラム



関数呼び出し

- 一般的に関数呼び出しは
ILP 妨げる

呼び出される関数のデータスコープ
内の依存性を見つけることが困難

```
for (j=0;j<MAX;j++) {
    compute(&a[j],&b[j]);
    a[j]=sin(b[j]);
}
```

sample12.c

- 例外:
 - 超越関数
 - IPO 最適化コンパイル

SVML ライブラリーを使用した数学関数

SVML (Short vector math library) により
効率的なソフトウェア実装を提供:

- sin/cos/tan
- asin/acos/atan
- sinh/cosh/tanh
- asinh/acosh/atanh
- log10/ln
- exp/pow
- ... その他

状態依存の関数呼び出し

多くのルーチンは呼び出しの状態を維持する

- メモリー割り当て
- 擬似乱数生成
- I/O ルーチン
- グラフィック・ライブラリー
- サードパーティー・ライブラリー

これらのルーチンの並列アクセスは同期されていない限り安全ではない

スレッドの安全性を確実にするため特定の関数におけるリソースのアクセスを確認する

以下のループの依存性を識別する

```
for (i=0; j<MAX-2, i++) {
  A[i+2]=A[i] + 1;
}
```

```
for (i=0; j<MAX-1, i+=2) {
  A[i+1]=A[i] + 1;
}
```

```
for (i=1; j<MAX, i++) {
  A[i]=A[i-1] * 2;
  B = A[i-1];
}
```

```
for (i=0; j<MAX, i++) {
  A[j][i+1] = A[k][i] + B
}
```

以下のループの依存性を識別する

```
for (i=0; i<MAX-2, i++) {
  A[i+2]=A[i] + 1;
}
```

```
for (i=0; i<MAX-1, i+=2) {
  A[i+1]=A[i] + 1;
}
```

$A[2] = A[0] + 1$
 $A[3] = A[1] + 1$
 $A[4] = A[2] + 1$

$A[1] = A[0] + 1$
 $A[3] = A[2] + 1$
 $A[5] = A[4] + 1$
 奇数 偶数

以下のループの依存性を識別する

```
for (i=1; i<MAX, i++) {
  A[i]=A[i-1] * 2;
  B = A[i-1];
}
```

```
for (i=0; i<MAX, i++) {
  A[j][i+1] = A[k][i] + B
}
```

$A[1] = A[0] * 2$
 $B = A[0]$
 $A[2] = A[1] * 2$
 $B = A[1]$

$A[j][1] = A[k][0] + 1$
 $A[j][3] = A[k][2] + 1$
 $A[j][5] = A[k][4] + 1$

ベクトル化可能なループとは

1. 単純な「if 文」構造ブロックを含むループ
2. 一部の組み込み関数(SVML) を含むループ
3. スカラー演算(リダクション)を行うループ
4. 非ユニットのストライド(非効率的)を行うループ

ループをベクトル化できる可能性は多い

ループがベクトル化されなかった理由: "ループ回数が少なすぎる"

ループで SSE 命令を使用すると、オーバーヘッドが発生する
反復回数が少ないループをベクトル化すべきではない
コンパイラーにループの大きさを知らせる

```
for (i = 0; i < MAX1; ++i) {
    a1[i] = b1[i] * c1[i] + d1[i];
}
```

```
for (i = 0; i < n; ++i) {
    a2[i] = b2[i] * c2[i] + d2[i];
}
```

ループがベクトル化されなかった理由: "ループ回数が少なすぎる"

ループで SSE 命令を使用すると、オーバーヘッドが発生する
反復回数が少ないループをベクトル化すべきではない
コンパイラにループの大きさを知らせる

```
#pragma loop count(5)
for (i = 0; i < MAX1; ++i) {
    a1[i] = b1[i] * c1[i] + d1[i];
}
```

```
#pragma loop count(1000)
for (i = 0; i < MAX2; ++i) {
    a2[i] = b2[i] * c2[i] + d2[i];
}
```

ループがベクトル化されなかった理由: "ベクトル化しても非効率"

ループが大きいと、非常に多くのレジスターが必要になる

```
for (i = 0; i < N; i++) {
    A[i] = B[i] * max(C[i], D[i]) + F[i] - E[i];
    G[i] = H[i] * sin(I[i]) + J[i] / K[i];
}
sample15.c
```

他の問題によってもこの警告が出力されることがある

ループがベクトル化されなかった理由: "ベクトル化しても非効率"

ループが大きいと、非常に多くのレジスターが必要になる

- ソリューション: ループの分割

- **#pragma distribute point** - ループの分割を指示

```
for (i = 0; i < N; i++) {
    A[i] = B[i] * max(C[i], D[i]) + F[i] - E[i];
    #pragma distribute point
    G[i] = H[i] * sin(I[i]) + J[i] / K[i];
}
```

sample15.c

- 2つのループに分割可能であるヒントをコンパイラへ指示
他の問題によってもこの警告が出力されることがある

ループがベクトル化されなかった理由: "サポートされないループ構造"

```
struct _xx { int data; int bound; };
```

```
doit1(int *a, struct _xx *x) {
    for (int i = 0; i < x->bound; i++)
        a[i] = 0;
}
```

サポートされないループ構造とは、カウントできないループ、またはコンパイラがループ回数をランタイム時に判定できないループを意味する

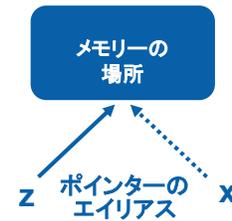
ループがベクトル化されなかった理由: "ベクトルの依存関係が存在する"

ループの反復に依存関係がないか?

または

コンパイラーが 2 つのポインターがエイリアスではないと仮定できない (メモリー内でオーバーラップする可能性あり)

```
void scale(float *z, float *x) {
    for (i = 0; i < 100; i++)
        z[i] = A * x[i];
}
```



31

インテル・ソフトウェア教育カリキュラム



ベクトルレポート問題に対するソリューション

後続くループのベクトル化を制御する

- **#pragma novector** - ループのベクトル化が有効な場合でもループをベクトル化しない
- **#pragma vector always** - メリットと例外検出に関するヒューリスティックを無視して常にループをベクトル化する

```
#pragma vector always
for (i = 0; i < MAX; ++i) {
    a[i] = b[i] * c[i] + d[i];
}
```

32

インテル・ソフトウェア教育カリキュラム



推奨する明確化スイッチ

Linux*

Windows*

-ipo

/Qipo

IPO によるグローバルなスタティック解析は
ポインターを明確化できる

-restrict

/Qrestrict

restrict オプションはポインターの明確化を有効にする

明確化: "restrict" キーワード

ISO C99 規格の一部

次のコマンドライン・スイッチが必要:

Linux*

Windows*

-restrict

/Qrestrict

-c99

/Qstd=c99

```
void foo(int *x, int *y, int * restrict z) {
    int i;
    for (i = 0; i < 100; i++)
        z[i] = A * x[i] + y[i];
}
```

sample18.c

// 2次元配列の例

```
void mult(int a[][NUM], int b[restrict][NUM]);
```

明確化: プラグマ

ベクトルの依存関係をコンパイラーに無視させる

```
#pragma ivdep
  for (i = 0; i < 100; i++)
    z[i] = A * x[i] + y[i];
```

```
#pragma simd
  for (i = 0; i < 100; i++)
    z[i] = A * x[i] + y[i];
```

明確化のテスト

Linux*

-fno-alias

ソースファイルのすべてのポインタはエイリアスされないと仮定する

-fno-fnalias

-alias_args[-]

関数内にエイリアスが存在しないと仮定する
(つまり、ポインタ引数は一意)

-ansi_alias[-] デフォルト

異なるタイプのポインタ間のエイリアスが存在しないと仮定する

Windows*

/Oa

/Ow

/Qalias_args[-]

/Qansi_alias[-]

ベクトル化レポート

```
Intel(R) C++ Intel(R) 64 Compiler XE for applications running on Intel(R)
64, Version 12.0.0.104 Build 20101006
Copyright (C) 1985-2010 Intel Corporation. All rights reserved.
```

```
novvec.cpp(59) (col. 19): remark: LOOP WAS VECTORIZED.
```

デフォルトではベクトル化されたかどうかは通知されない

/Qvec-report (-vec-report) は、
コンパイラーがベクトル化できた/できなかった場所を通知する

ベクトル化レポートのオプション

ベクトル化に関するレポートを表示
/Qvec-report[:n] (-vec-report=n)

ベクトライザーの診断レベルを設定する

- n=0: 診断情報を表示しない
- n=1: ベクトル化されたループのみを表示する(デフォルト)
- n=2: ベクトル化されなかったループとその理由も表示する
- n=3: 依存情報も表示する
- n=4: ベクトル化されなかったループのみを表示する
- n=5: ベクトル化されなかったループのみを表示し、依存情報も表示する

ベクトル化レポート

Intel(R) C++ Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 12.0.0.104 Build 20101006
Copyright (C) 1985-2010 Intel Corporation. All rights reserved.

novect.cpp(40) (col. 3): remark: loop was not vectorized: not inner loop.
novect.cpp(41) (col. 5): remark: loop was not vectorized: not inner loop.
novect.cpp(43) (col. 7): remark: loop was not vectorized: existence of vector dependence.
novect.cpp(50) (col. 3): remark: loop was not vectorized: existence of vector dependence.
novect.cpp(67) (col. 4): remark: loop was not vectorized: vectorization possible but seems inefficient.

/Qvec-report4はベクトル化できなかった場所を通知する

ガイドレポート

バージョン 12 のインテルコンパイラーでは、新たにガイド付き最適化（/Qguide オプション）がサポートされています。この機能とレポート機能の違いは、レポートでは問題点は報告してくれますが、問題は開発者自身が考えて直さなければならない。ガイドによる最適化では、どのように問題に対処すべきか、コンパイラーがアシストメッセージを出力してくれる

```
// サンプルコード: sample20.c
#include "mul.h"
void matvec(float a[][COLWIDTH], float b[], float x[]){
int i, j, sz1 = size1; sz2 = size2;
    for (i = 0; i < sz1; i++) {           // 5 行目
        b[i] = 0;
        for (j = 0; j < sz2; j++) {     // 7 行目
            b[i] += a[i][j] * x[j];     // 8 行目
        }
    }
}
```

icl sample20.c /Qguide (icc sample20.c -guide)

ガイドレポート

remark #30761: コンパイラーに自動並列化を向上させるアドバイスを生成させる場合は -Qparallel オプションを追加します。

\sample20.c(9): remark #30536: (LOOP) 必要に応じて、-Qno-alias-args オプションを追加してコンパイラーによる型ベースの一義化解析を向上できます (オプションはコンパイル全体に適用されます)。これにより、行 9 のループがベクトル化され最適化が向上します。

[確認] コンパイル全体でこのオプションのセマンティクスに沿っていることを確認してください。[別の方法] ルーチン "matvec" のすべてのポインター型の引数に "restrict" キーワードを追加することで同様の効果が得られます。これにより、行 9 のループがベクトル化され最適化が向上します。

[確認] "restrict" ポインター修飾子のセマンティクスに沿っていることを確認してください。ルーチンにおいて、そのポインターによってアクセスされるすべてのデータは、ほかのポインターからはアクセスできません。

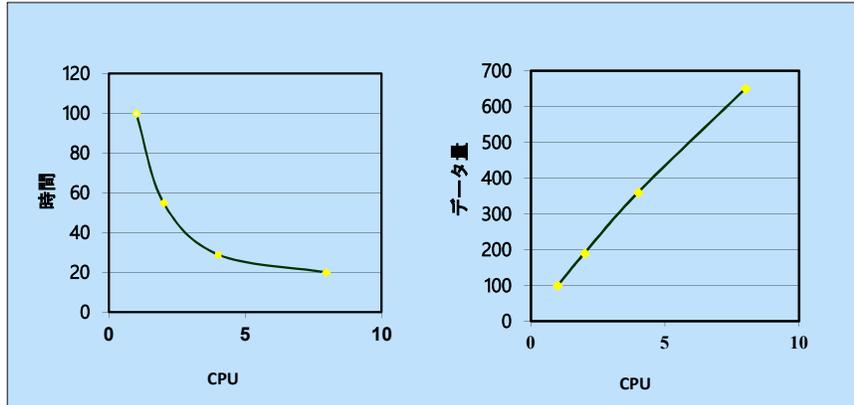
このコンパイルセッションで出力されたアドバイス・メッセージの数: 1。

コースの内容

- SIMD 命令とベクトル化
- 並列化とOpenMP
- 最新プロセッサでのデモ

なぜ並列処理を使用するのか?

- 計算をより短い時間で処理

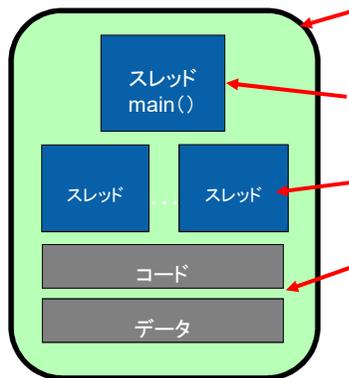


43

インテル・ソフトウェア教育カリキュラム



スレッドとプロセス



- 現在のオペレーティング・システムではプログラムをプロセスとしてロードする
- プロセスはエントリーポイントでスレッドとして実行を開始する
- スレッドはプロセス内で他のスレッドを作成できる
- プロセス内のすべてのスレッドはコードとデータ領域を共有する
- スレッドは 2 つのスケジュール状態 (アクティブ、インアクティブ) を持つ

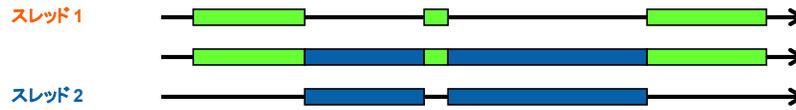
44

インテル・ソフトウェア教育カリキュラム

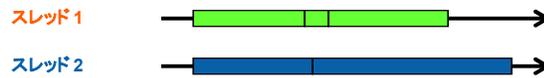


コンカレンシー(並行性)とパラレリズム(並列性)

- 並行性: 2 つもしくはそれ以上のスレッドが同時に進行する



- 並列性: 2 つもしくはそれ以上のスレッドが同時に実行される



なぜスレッドを利用するのか？

- 利点
 - パフォーマンスの向上
 - リソースの利用率を改善
- 欠点
 - アプリケーションがより複雑になる
 - デバッグが困難(データ競合、デッドロック、その他)

主要な 5 つの並列化手法

• いずれの手法を導入するか？

- メッセージ・パッシング
 - MPI、PVM
- 明示的なスレーディング
 - Windows スレッド API、Pthreads、Solaris スレッド、Java スレッド・クラス
- コンパイラーのサポート
 - 自動並列化、OpenMP、インテル・スレーディング・ビルディング・ブロック(TBB)
- 並列プログラミング言語
 - HPF、CAF、UPC、CxC、Cilk Plus
- 並列マス・ライブラリー
 - ScaLAPACK、PARDISO、PLAPACK、PETsc

並列化プログラミングで直面する 3 つの課題

- スケーラビリティ
- 正当性
- 容易なプログラミングと保守



スレッド化の例

Hello World

```

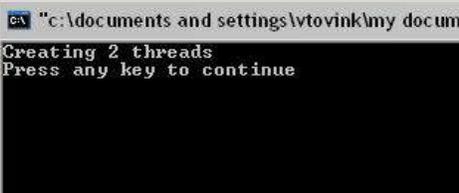
DWORD WINAPI HelloFunc (LPVOID)
{
    cout << "Hello Thread\n";
    return 0;
}

HANDLE hThread[numThreads];
for (int i = 0; i < numThreads; i++)
    hThread[i] = CreateThread(
        NULL, 0, HelloFunc,
        NULL, 0, NULL );

// Clean up thread handles
for (int i = 0; i < numThreads; i++)
    CloseHandle (hThread[i]);

return 0;

```



49

インテル・ソフトウェア教育カリキュラム



スレッド化の例

Hello World(変更後)

```

for (int i = 0; i < numThreads; i++)
    hThread[i] = CreateThread (NULL, 0, HelloFunc,
        NULL, 0, NULL );

getchar();

// Clean up thread handles
for (int i = 0; i < numThreads; i++)
    CloseHandle (hThread[i]);

return 0;

```



サンプル・コードの変更

“Hello World Thread <スレッド番号>” メッセージに変更するには？

```

Hello World Thread #1
Hello World Thread #0

```

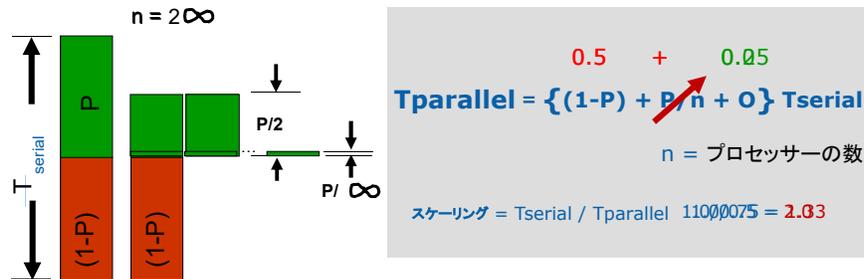
50

インテル・ソフトウェア教育カリキュラム



アムダールの法則

- 並列化のスピードアップ(スケーリング)の上限を説明
- オーバーヘッドの影響を考慮する際に役立つ



シリアルコードがスケーリングを制限する

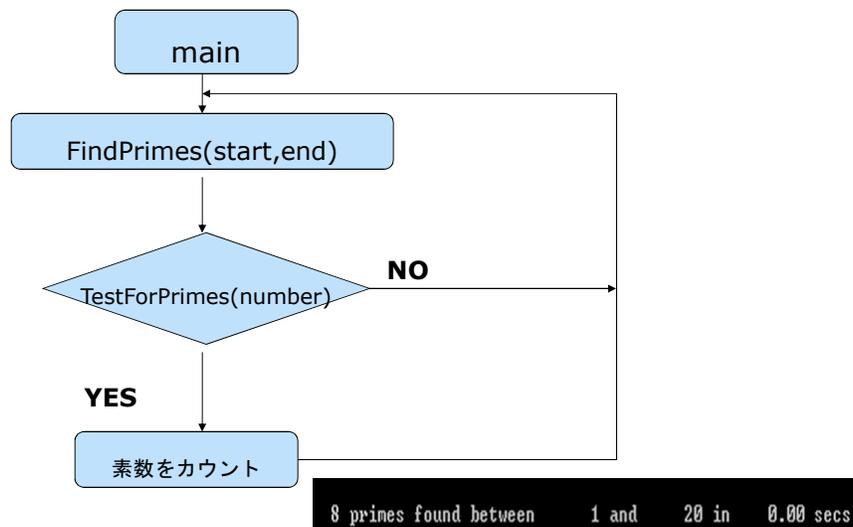
並列プログラミング・モデル

- 機能分割
 - タスクの並列処理
 - 同じ問題の独立したタスク
 - UI タスク、UI 更新、生産と消費のパラダイム
 - 並列演算と次の表示タスク
- データ分割
 - 異なるデータで実行される同じ演算
 - 例: 行列の乗算

アプリケーションをスレッド化する際によくある質問

- どこをスレッド化すればいいのか?
- スレッド化に必要な時間は?
- 必要な再設計の回数は?
- 選択した領域をスレッド化する価値はあるか?
- どの程度のスピードアップを期待すべきか?
- パフォーマンスは期待値を満たすことができるか?
- プロセッサを追加すれば性能が向上するか?
- どのスレッドモデルを使用すべきか?

素数をカウントするサンプルコード



ケーススタディー

素数をカウントするサンプルコード

8 primes found between 1 and 20 in 0.00 secs

2
3
5
7
9
11
13
15
17

3
5
7

11

```

        limit = (long)(sqrtf((float)val)+0.5f);
        while( (factor <= limit) &&
              (val % factor)
              factor ++;

        return (factor > limit);
    }

    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
            gPrimesFound++;
    }
}

```

55

インテル・ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめるとどこをスレッド化すればいいのか?

• シングルスレッド "Primes" の実行を VTune™ Amplifier X

Function / Call Stack	Time (ns)
TestForPrime	296.188
GetConsoleMode	233.962
WriteFile	31.454

```

bool TestForPrime(int val)
{
    // let's start checking from 3
    int limit, factor = 3;
    limit = (long)(sqrtf((float)val)+0.5f);
    while( (factor <= limit) &&
          (val % factor)
          factor ++;

    return (factor > limit);
}

void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;
    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
            gPrimesFound++;
    }
}

```

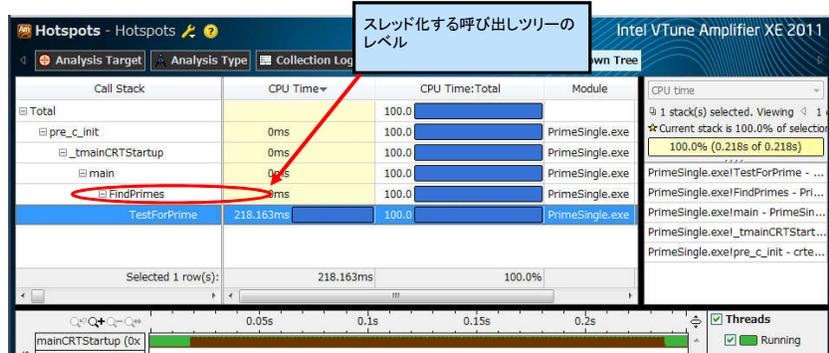
時間のかかる領域を特定する

56

インテル・ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめる どこをスレッド化すればいいのか?



スレッド化する呼び出しツリーのレベルを特定する

57

インテル・ソフトウェア教育カリキュラム



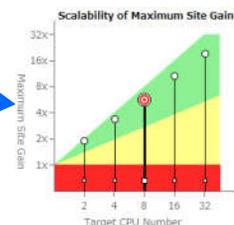
OpenMP* プログラミングをはじめる 選択した領域をスレッド化する価値はあるか?

コードのどの位置で並列化を実装するか、インテル®
Parallel Advisor を使って性能を見積もる

```

ANNOTATE_SITE_BEGIN(prime);
for( int i = start; i <= end; i+=2 ){
  ANNOTATE_TASK_BEGIN(openmp);
  if( TestForPrime(i) )
    gPrimesFound++;
  ANNOTATE_TASK_END(openmp);
}
ANNOTATE_SITE_END(prime);

```



スピードアップする場所が見つかったら、
OpenMP で並列化を実装する

58

インテル・ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめる データの整合性をチェックする

プログラム・コード中の競合の可能性のある変数を見つける
インテル® コンパイラーの診断機能を利用すると;
>icl PrimeSingle.cpp /Qdiag-enable:thread
スタティック変数の利用、代入、参照をチェックできる

```
Primesingle.cpp
Primesingle.cpp(15): 警告 #1711: 静的に割り当てられた変数 "gProgress" への代入
gProgress++;
^
Primesingle.cpp(16): 警告 #1710: 静的に割り当てられた変数 "gProgress" への参照
percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);
^
Primesingle.cpp(39): 警告 #1711: 静的に割り当てられた変数 "gPrimesFound" への代入
gPrimesFound++;
^
Primesingle.cpp(49): 警告 #1711: 静的に割り当てられた変数 "gPrimesFound" への代入
gPrimesFound = 1; // 2 は特殊ケースなので1つカウント
^
Primesingle.cpp(56): 警告 #1710: 静的に割り当てられた変数 "gPrimesFound" への参照
start, end, gPrimesFound,(float)(after - before)/ CLOCKS_PER_SEC);
^
```

59

インテル®ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめる データの整合性をチェックする

インテル® Parallel Advisor を使って性能を見積もるとともに、データの整合性をチェックできる

The screenshot shows the Intel Parallel Advisor interface with the 'Correctness Report' tab selected. The 'Problems and Messages' table lists three issues:

ID	Problem	Sources	Modules	State	Severity
P1	Data communication	PrimeSingle.cpp	PrimeSingle.exe	Not fixed	Error
P2	Data communication	PrimeSingle.cpp	PrimeSingle.exe	Not fixed	Remark
P3	Parallel site information	PrimeSingle.cpp	PrimeSingle.exe	Information	Problem

The 'Data communication: Code Locations' section shows the following code snippets:

```
X3 Parallel site PrimeSingle.cpp:38 FindPrimes PrimeSingle.exe Information
36 // 開始位置から開始
37 int range = end - start + 1;
38 ANNOTATE_TASK_BEGIN(primes);
39 for( int i = start; i <= end; i+=2 ){
40 ANNOTATE_TASK_BEGIN(opeomp);
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

60

インテル®ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめる どこをスレッド化すればいいのか?

- どこをスレッド化すればいいのか?
 - FindPrimes()
- 選択した領域をスレッド化する価値はあるか?
 - 依存関係が少ない
 - データを並列化できる
 - 実行時間の多く (90% 以上) を占める



OpenMP* プログラミングをはじめる 選択した領域をスレッド化する価値はあるか? どの程度のスピードアップが期待できるか?

- 予想されるメリットは? 並列化対象のコードが全体の 100% を占める場合

$$\text{スケーリング (8P)} = 100 / (100 / 8) = \sim 8 \times$$

(8コアシステムでの実行を想定)

- 最小限の作業で予想されるメリットを判断する方法は?

OpenMP* を使用した素早いプロトタイプ作成

- スレッド化に必要な時間は?
- 必要な再設計の回数は?

OpenMP* プログラミングをはじめる どのようにスレッドに負荷を分散するか?

```
#pragma omp parallel for
for( int i = start; i <= end; i += 2 ){
    isPrime = isPrimeForPrime(i)
    gPrimesFound += 1
    ShowPrimes(i)
}
```

OpenMP*

for ループでワークシェアを定義する

この並列領域のスレッドをここで作成する

```
Intel Composer XE 2011 IA-32 Visual Studio 2010
Z:\sample>prime
1から10000000までの範囲の素数の数は 77730個。計算時間 1.90秒
```

OpenMP* プログラミングをはじめる どの程度のスピードアップが期待できるか?

- 予想されるメリットは?
- 最小限の作業で予想されるメリットを判断する方法は?

$6.36 / 1.9 = 3.34 \times$ のスケーリング

- スレッド化に必要な時間は?
- 必要な再設計の回数?
- 可能な最良のスケーリングか?

OpenMP* プログラミングをはじめる 正当性のデバッグ

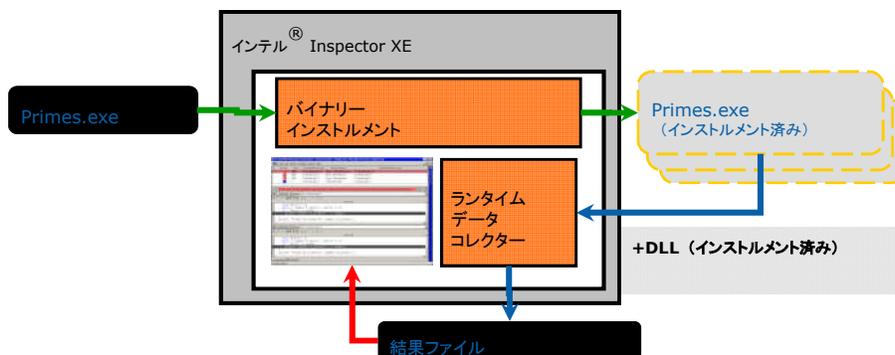
```

Intel Composer XE 2013: IA-32 Visual Studio 2010
Z:\sample>prime
1から10000000までの範囲の素数の数は 78590個 計算時間 1.73秒
Z:\sample>prime
1から10000000までの範囲の素数の数は 77730個 計算時間 1.83秒
  
```

- 毎回答えが同じであるか？
このスレッドの実装は正しいか？

OpenMP* プログラミングをはじめる 正当性のデバッグ

- インテル® Inspector XE は、データの競合、ストール、およびデッドロックなどのスレッド化の問題を正確に検出する



インテル[®] Inspector XE で問題を検出

The screenshot displays the Intel Inspector XE interface. The 'Problems' window shows a 'Data race' issue in 'PrimeSingle.cpp' at line 28. The 'Code Locations' window shows three instances (X1, X2, X3) of the 'gPrimesFound++' statement being accessed simultaneously in a parallel loop. The code snippet is as follows:

```

26 for( int i = start; i <= end; i+=2 ){
27     if( TestForPrime(i) )
28         gPrimesFound++;
29 }
30 }

```

8 スレッド動作中

67

インテル・ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめる 正当性のデバッグ

```

#pragma omp parallel for
for( int i = start; i <= end; i+= 2 ){
    if( TestForPrime(i) )
    #pragma omp atomic
        gPrimesFound++;
}

```

アトミック演算で
インクリメントする

この同期は最適か？

```

#pragma omp parallel for reduction(+:gPrimesFound)
for( int i = start; i <= end; i+= 2 ){
    if( TestForPrime(i) )
        gPrimesFound++;
}

```

68

インテル・ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめる パフォーマンス

- $6.36 / 1.78 = 3.57 \times$ のパフォーマンスは最適か?
- このアルゴリズムから期待できる最良のパフォーマンスか?

```

Intel Composer XE 2011. 1A-32 Visual Studio 2010
Z:\sample>prime
1から10000000までの範囲の素数の数は 664580個。計算時間 1.78秒
  
```

OpenMP* プログラミングをはじめる パフォーマンスの検証



明白なロード・インバランス。8つのスレッドで負荷が異なっている

設計段階に戻る

OpenMP* プログラミングをはじめ パフォーマンスの検証

- OpenMP における for ワークシェア構文のデフォルトのスケジューリングは?
- ループのブロック分割

```
#pragma omp parallel for reduction(+: gPrimesFound)
for( int i = start; i <= end; i+= 2 ){
    if( TestForPrime(i) )
        gPrimesFound++;
}
```

TestForPrimeは
値が大きいほど
判定に時間がか
かる

このループは、i = 1 から 10000000 までを、8 つのブロックに分割される

つまり最後のブロックを処理するスレッドに負荷が大きくなる

71

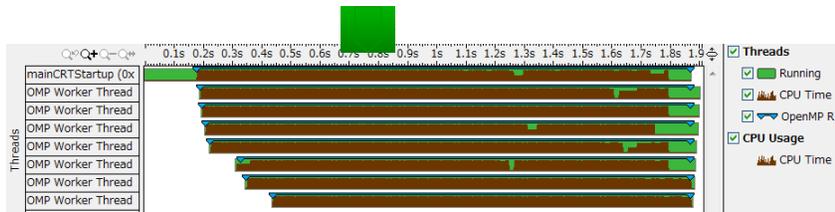
インテル・ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめ パフォーマンスの改善

```
#pragma omp parallel for reduction(+: gPrimesFound) schedule(auto)
for( int i = start; i <= end; i+= 2 ){
    if( TestForPrime(i) )
        gPrimesFound++;
}
```

for ワークシェア構文に `schedule(auto)` 節を追加してスケジューリングを変更



72

インテル・ソフトウェア教育カリキュラム



OpenMP* プログラミングをはじめる パフォーマンスの再確認

$6.36 / 1.59 = 4.0 \times$ のパフォーマンスは最適か？

このアルゴリズムはプロセッサコアが増加すると
適切にスケールする



```
Intel Composer XE 2011 IA-32 Visual Studio 2010
Z:\sample>prime
1から10000000までの範囲の素数の数は 664580個。計算時間 1.59秒
```

質問の再確認

アプリケーションをスレッド化する際によくある質問

- どこをスレッド化すればいいのか？
- 選択した領域をスレッド化する価値はあるか？
- どの程度のスピードアップが期待できるか？
- どのようにスレッドに負荷を分散するか？
- プロセッサを追加すれば性能が向上するか？
- どこでスレッド化と最適化を終了できるか？

セクションによる簡単な並列化

```
main()
{
  #pragma omp parallel sections
  {
    /* task 1 */
    #pragma omp section
    {
      printf("I am sleeping 1\n");
      sleep(5);
    }

    /* task 2 */
    #pragma omp section
    {
      printf("I am sleeping 2\n");
      sleep(5);
    }
  }
}
```

75

インテル・ソフトウェア教育カリキュラム



コースの内容

- SIMD 命令とベクトル化
- 並列化とOpenMP
- 最新プロセッサでのデモ
 - 際内ループをベクトル化して最外ループを並列化

76

インテル・ソフトウェア教育カリキュラム



まとめ

- SIMD命令を利用したベクトル化の為には依存性等のベクトル化を妨げる要因を見つけて排除する
- 最新のプロセッサではOpenMP等の並列化を行ってすべてのコアを有効利用
- 最内側ループをベクトル化し、最外側ループを並列化することにより最大限の性能を実現



本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。製品に付属の売買契約書『Intel's Terms and conditions of Sales』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証（特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的所有権を侵害していないことへの保証を含む）に関しても一切責任を負わないものとします。

インテル製品は、予告なく仕様が変更されることがあります。

* 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。

© 2017 Intel Corporation.

