Orchestrating a brighter world

NEC

Vector Engine向け C/C++コンパイラの使い方

第2版 2019年 11月発行

Orchestrating a brighter world

未来に向かい、人が生きる、豊かに生きるために欠かせないもの。
それは「安全」「安心」「効率」「公平」という価値が実現された社会です。

NECは、ネットワーク技術とコンピューティング技術をあわせ持つ
類のないインテグレーターとしてリーダーシップを発揮し、
卓越した技術とさまざまな知見やアイデアを融合することで、
世界の国々や地域の人々と協奏しながら、
明るく希望に満ちた暮らしと社会を実現し、未来につなげていきます。

目次

なお、本書で「並列処理」と記述したとき、コンパイラの自動並列化機能、または、OpenMP C/C++機能を使用した共有メモリ型並列処理を指します。

Vector Engine向けC/C++コンパイラ

製品名:NEC C/C++ Compiler for Vector Engine

- 対応する言語仕様
  - ISO/IEC 9899:2011 Programming languages – C
  - ISO/IEC 14882:2014 Programming languages – C++
  - OpenMP Version 4.5
- 主な機能
  - 自動ベクトル化機能
  - 自動並列化機能・OpenMP C/C++
  - 自動インライン展開機能

C/C++コンパイラの使い方

C/C++コンパイラの利用

```
$ ncc -mparallel -O3 a.c b.c … Cプログラムのコンパイル、リンク
$ nc++ -O4 x.cpp y.cpp       … C++プログラムのコンパイル、リンク
```

-O4 … 最大レベルの自動ベクトル化を適用
-O3 … 高度なレベルの自動ベクトル化を適用
-O2 … 既定レベルの自動ベクトル化を適用
-O1 … 副作用のない自動ベクトル化を適用
-O0 … ベクトル化、最適化を行わない

高
低

これらは、コンパイラの自動ベクトル化、最適化レベルをコントロールする。

-fopenmp … OpenMP C/C++機能を利用
-mparallel … 自動並列化機能を利用

これらは、コンパイラの並列処理機能をコントロールする。
並列処理機能を使用しないときは指定しなくてよい。

代表的なコンパイラオプションの指定例

`$ ncc a.c b.c`	既定レベルの自動ベクトル化を適用し、コンパイル、リンク
`$ nc++ -O4 a.C b.C`	最大レベルの自動ベクトル化を適用し、コンパイル、リンク
`$ ncc -mparallel -O3 a.c b.c`	自動並列化、および、高度なレベルの自動ベクトル化を適用し、コンパイル、リンク
`$ nc++ -O4 -finline-functions a.cpp b.cpp`	自動インライン展開、および、最大レベルの自動ベクトル化を適用し、コンパイル、リンク
`$ ncc -O0 -g a.c b.c`	ベクトル化を止めて、シンボリックデバッグするコンパイル、リンク
`$ ncc -g a.c b.c`	ベクトル化を止めずに、シンボリックデバッグするコンパイル、リンク
`$ ncc -E a.c b.c`	プリプロセスのみ実行。プリプロセス結果は標準出力に出力する
`$ nc++ -fsyntax-only a.cpp b.cpp`	シンタックスチェックのみ実行

プログラムの実行

```
$ ncc a.c b.c
$ ./a.out
```

実行ファイルを指定

```
$ ./b.out data1.in
```

実行ファイルに入力ファイルやパラメータを渡すとき、実行ファイル名に続けてそれらを指定

```
$ ./c.out < data2.in
```

実行ファイルに入力ファイルをリダイレクト

```
$ ncc -mparallel -O3 a.c b.c
$ export OMP_NUM_THREADS=4
$ ./a.out
```

自動並列化したプログラムのスレッド数は環境変数**OMP_NUM_THREADS**で指定できる

```
$ env VE_NODE_NUMBER=1 ./a.out
```

利用するVEノードは
環境変数**VE_NODE_NUMBER**で指定する

実行性能の測定方法

Vector Engineでの性能測定機能

PROGINF(プログインフ)

- プログラム全体の性能情報
- 性能情報取得のためのオーバーヘッドは極小

FTRACE(エフトレース)

- 関数ごとの性能情報
- プログラムの再コンパイル、再リンクが必要
- 関数の呼び出し回数が多いと、性能情報取得のためのオーバーヘッドが大きくなり、実行時間が長くなることがある

PROGINF

プログラム全体の性能情報

```
$ ncc -O4  a.c b.c c.c
$ ls a.out
a.out
$ export VE_PROGINF=DETAIL
$ ./a.out
            ********  Program  Information  ********
  Real Time (sec)                          :           11.329254
  User Time (sec)                          :           11.323691
  Vector Time (sec)                        :           11.012581
  Inst. Count                              :          6206113403
  V. Inst. Count                           :          2653887022
  V. Element Count                         :        619700067996
  V. Load Element Count                    :         53789940198
  FLOP count                               :        576929115066
  MOPS                                     :        73492.138481
  MOPS (Real)                              :        73417.293683
  MFLOPS                                   :        50976.512081
  MFLOPS (Real)                            :        50924.597321
  A. V. Length                             :          233.506575
  V. Op. Ratio (%)                         :           99.572922
  L1 Cache Miss (sec)                      :            0.010847
  CPU Port Conf. (sec)                     :            0.000000
  V. Arith. Exec. (sec)                    :            8.406444
  V. Load Exec. (sec)                      :            1.384491
  VLD LLC Hit Element Ratio (%)            :          100.000000
  Power Throttling (sec)                   :            0.000000
  Thermal Throttling (sec)                 :            0.000000
  Max Active Threads                       :                   1
  Available CPU Cores                      :                   8
  Average CPU Cores Used                   :            0.999509
  Memory Size Used (MB)                    :          204.000000
```

実行時に環境変数**VE_PROGINF**に
以下のどちらかの値をセット
“**YES**” … 基本情報
“**DETAIL**” … 基本情報+メモリ情報

時間情報

命令実行回数情報

ベクトル化情報・メモリ情報・
並列化情報

FTRACE

```
$ ncc -ftrace a.c b.c c.c
$ ./a.out
$ ls ftrace.out
ftrace.out
$ ftrace
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Thu Mar 22 17:32:54 2018 JST
Total CPU Time : 0:00'11"163 (11.163 sec.)


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS     CONF HIT E.%

    15000     4.762( 42.7)     0.317  77117.2  62034.6  99.45 251.0     4.605   0.002    0.000  100.00 funcA
    15000     3.541( 31.7)     0.236  73510.3  56944.5  99.46 216.0     3.554   0.000    0.000  100.00 funcB
    15000     2.726( 24.4)     0.182  71930.2  27556.5  99.43 230.8     2.725   0.000    0.000  100.00 funcC
        1     0.134(  1.2)   133.700  60368.8  35641.2  98.53 214.9     0.118   0.000    0.000    0.00 main
-------------------------------------------------------------------------------------------------------------
--
    45001    11.163(100.0)     0.248  74505.7  51683.9  99.44 233.5    11.002   0.002    0.000  100.00 total
```

(コンパイル、リンク時に**-ftrace**コンパイラオプションを指定)

(実行終了時、性能情報が格納された**ftrace.out**ファイルが出力)

(**ftrace**コマンドで解析結果を表示)

MPIプログラムのとき、性能情報が格納されたファイルが複数出力される。それらを**-f**オプションで指定する。

```
$ ls ftrace.out.*
ftrace.out.0.0   ftrace.out.0.1   ftrace.out.0.2   ftrace.out.0.3
$ ftrace -f ftrace.out.0.0 ftrace.out.0.1 ftrace.out.0.2 ftrace.out.0.3
```

性能測定時の注意

- FTRACEでは、関数の入口/出口で性能情報を採取するため、関数の呼び出し回数が多いプログラムでプログラム全体の実行時間が増加してしまう。

```
$ nc++ -ftrace -c a.cpp
$ nc++ -c main.cpp b.cpp c.cpp
$ nc++ -ftrace a.o main.o b.o c.o
$ ./a.out
```

- 目的の関数が含まれているファイルのみ**-ftrace**付きでコンパイルする
- リンク時にも**-ftrace**を指定する

- **-ftrace**なしでコンパイルされたファイル中の関数の性能情報は、それらを呼び出した関数の性能情報に含めて表示される。
- FTRACEでは、インライン展開された関数の性能情報は、それを呼び出した関数の性能情報に含めて表示される。
- システムライブラリ関数に関する性能情報
  - PROGINFで表示される性能情報には、プログラムから呼び出しているシステムライブラリ関数の性能情報も含まれる。
  - FTRACEで表示される性能情報には、プログラムから呼び出しているシステムライブラリ関数の性能情報も含まれる。それらは、呼び出した関数の性能情報に含めて表示される。

プログラムのデバッグ

トレースバック機能

トレースバック機能を利用する場合、
コンパイル、リンク時に**-traceback**を指定し、
実行時に環境変数**VE_TRACEBACK**に“**FULL**”をセット

演算例外を発生させるには
環境変数**VE_FPE_ENABLE**に、
以下の値のいずれかをセット
“**DIV**” … ゼロ除算例外
“**INV**” … 無効演算例外

```
#include <stdio.h>
int main(void) {
    printf("%f¥n",1.0/0.0);
}
```

ゼロ除算が発生

```
$ ncc -traceback main.c
$ export VE_TRACEBACK=FULL
$ export VE_ADVANCEOFF=YES
$ export VE_FPE_ENABLE=DIV
$ ./a.out
Runtime Error: Divide by zero at 0x600000000cc0
[ 1] Called from 0x7f5ca0062f60
[ 2] Called from 0x600000000b70
Floating point exception
$ naddr2line -e a.out -a 0x600000000cc0
0x0000600000000cc0
/.../main.c:3
```

コンパイル、リンク時に**-traceback**を指定
トレースバック機能を使用
先行命令制御機構をOFF
ゼロ除算時に例外発生
トレースバック情報
ソースコード中の例外発生箇所を特定
main.cファイル内の3行目でゼロ除算が発生していると分かる

※ **VE_FPE_ENABLE**は上記以外の値を設定できるが、トレースバックでは基本的に上記二つを使用する

デバッガ(gdb)の利用

実行時間の長いプログラムでは、事前に問題のある関数を突き止めておき、その関数が含まれるファイルのコンパイル時のみ**-g**オプションを使用する

```
$ ncc -O0 -g -c a.c
$ ncc -O4 -c b.c c.c
$ ncc a.o b.o c.o
$ gdb a.out
(gdb) break func
Breakpoint 1 at func
(gdb) run
Breakpoint 1 at func
(gdb) continue
...
```

a.cのみ**-O0 -g**でコンパイル(性能ダウンを避ける)

それ以外のファイルは**-g**なしで最適化も適用

gdbを起動

注意事項

- **-O0**を指定せずにデバッグするとき、コンパイラの最適化によりコードや変数が削除、移動されるため、デバッガで変数が参照できなかったり、ブレークポイントが設定できないことがある。
- HWによる命令の先行制御によって例外発生個所が正しく表示されないことがある。環境変数**VE_ADVANCEOFF**に “**YES**”を設定することで先行制御を無効にできる。ただし、先行制御を無効にすることでプログラムの実行時間が大幅に長くなることがあるため注意すること。

システムコールのトレース:strace

```
$ /opt/nec/ve/bin/strace ./a.out
...
write(2, "delt=0.0251953, TSTEP".., 27)           = 27
open(“MULNET.DAT", O_WRONLY|O_CREAT|O_TRUNC, 0666)= 5
ioctl(5, TCGETA, 0x8000000CC0)                    Err#25 ENOTTY
fxstat(5, 0x8000000AB0)                           = 0
write(5, "1 2 66 65", 4095)                       = 4095
write(5, "343 342", 4096)                         = 4096
write(5, "603 602", 4096)                         = 4096
write(5, "863 862", 4094)                         = 4094
write(5, "1105 1104", 4095)                       = 4095
write(5, "1249 1313 1312", 4095)                  = 4095
write(5, "1456 1457 1521 1520", 4095)            = 4095
write(5, "1727", 4095)                            = 4095
...
```

システムコールの引数　　システムコールの返却値

- システムコールの引数、返却値のトレース情報の表示
  - システムライブラリの呼び出しが適切に行われたか? などが確認できる。
  - 出力が大量になるので、**strace**コマンドの**-e**オプションでトレースするシステムコールを厳選するとよい。

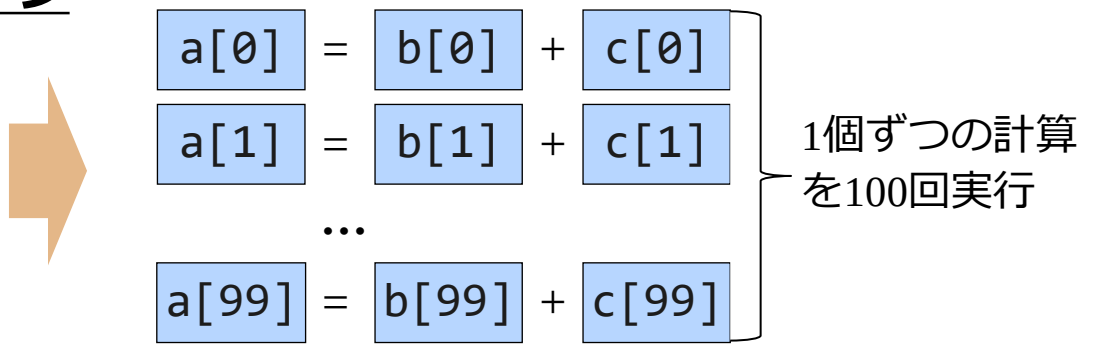
自動ベクトル化機能

ベクトル化とは?

規則的に並んだデータ列をベクトルデータと呼び、ベクトルデータを処理するスカラ命令列を、等価な処理を行うベクトル命令で置き換えることをベクトル化という

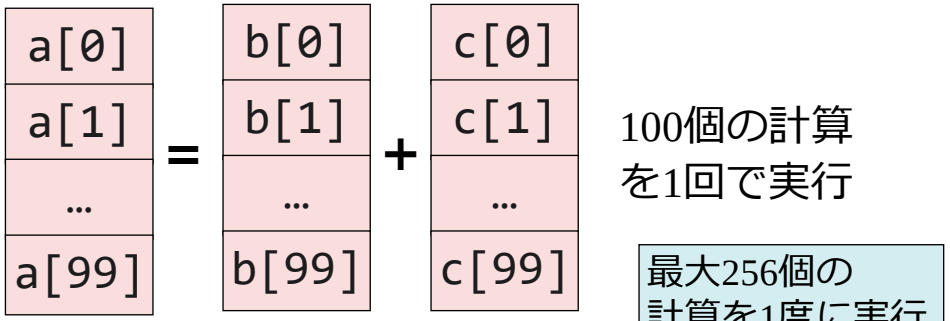
スカラ命令の実行イメージ

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
…
a[99]=b[99] + c[99];
```



ベクトル命令の実行イメージ

```
for (i=0; i<100; i++)
  a[i] = b[i] + c[i];
```



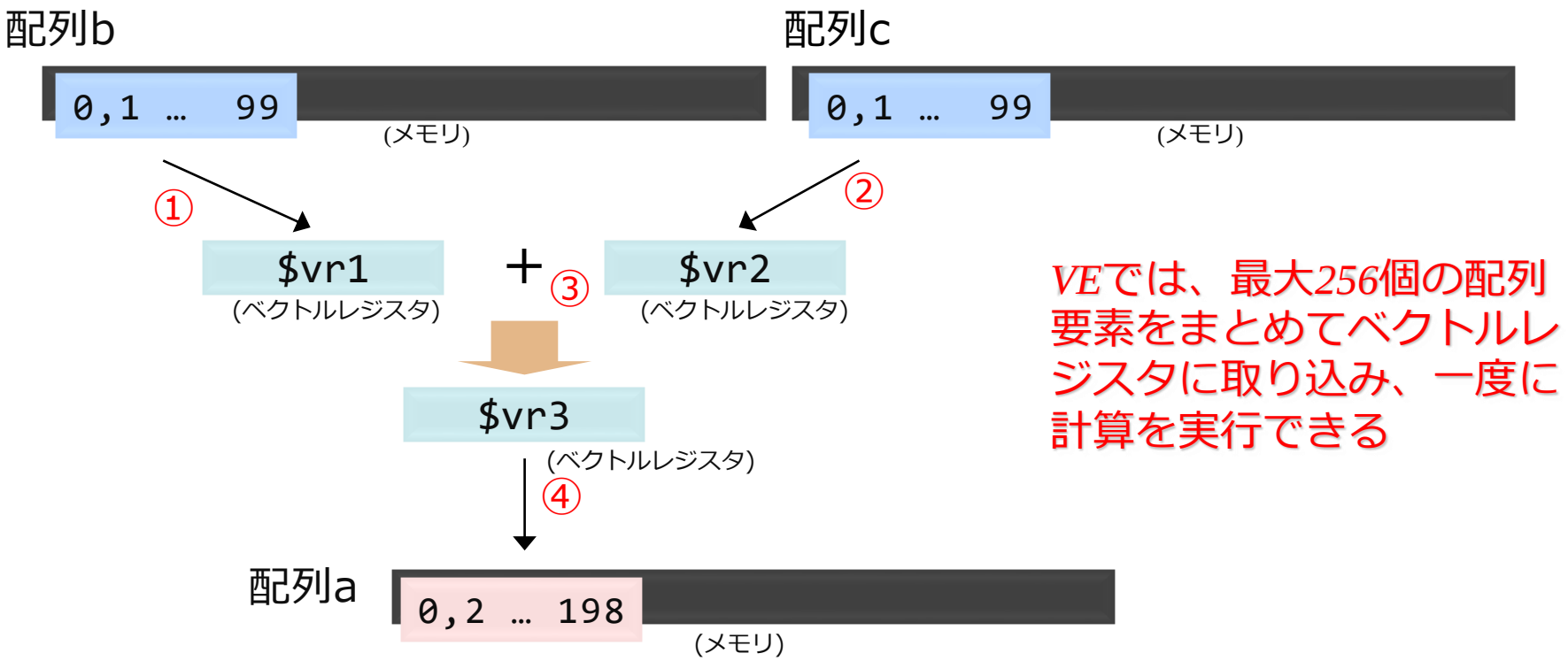
HW命令との対応

スカラ機のときこの四つの命令列を
100回繰り返さなければならない

```
a[0]  = b[0]  + c[0];
a[1]  = b[1]  + c[1];
…
a[99] = b[99] + c[99];
```

④ ① ③ ②

```
① VLoad  $vr1, b[0:99]
② VLoad  $vr2, c[0:99]
③ VAdd   $vr3, $vr1, $vr2
④ VStore $vr3, a[0:99]
```

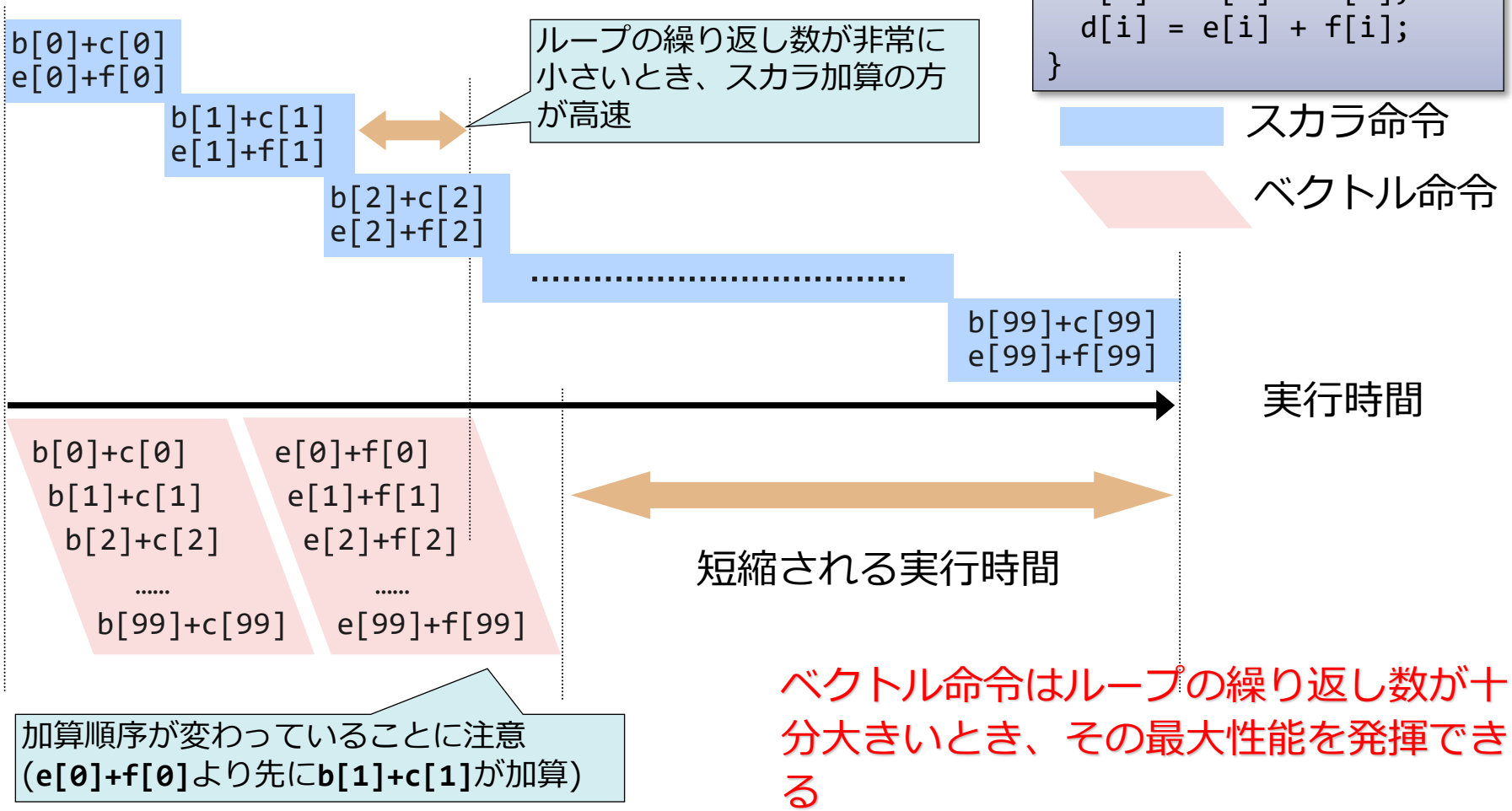


*VE*では、最大256個の配列要素をまとめてベクトルレジスタに取り込み、一度に計算を実行できる

命令実行時間の比較

スカラ加算命令の実行イメージ(2命令同時実行時)

```
for (i = 0; i < 100; i++)
{
  a[i] = b[i] + c[i];
  d[i] = e[i] + f[i];
}
```



ベクトル加算命令の実行イメージ

ベクトル命令はループの繰り返し数が十分大きいとき、その最大性能を発揮できる

ベクトル化できるループ

- ベクトル化に適合する演算、型のみを含むループ
  - **char、short、long double**型を含まない
    - 数値計算では、ほとんど使われない型
    - 対応する型のベクトル演算命令がないためベクトル化できない
  - 関数呼び出しを含まない
    - 三角関数、指数関数、対数関数等を除く。これらは、ベクトル処理可能
- 配列や変数の定義・参照関係に、ベクトル化を阻害する依存関係(ベクトル化不可の依存関係)がない
  - 計算順序の変更が可能であること
- ベクトル化によって、性能の向上が期待できる
  - ループ長(ループの繰り返し数)が十分に大きい

ベクトル化不可の依存関係 (1)

以前の繰り返しで定義された配列要素や変数を、後の繰り返しで参照するパターンのとき、計算順序を変更できない

例1

```
for (i=2; i < n; i++)
  a[i+1] = a[i] * b[i] + c[i];
```

ベクトル化すると、更新されたaの値が参照できないので、ベクトル化できない

スカラでの計算順序

```
a[3] = a[2] * b[2] + c[2];
a[4] = a[3] * b[3] + c[3];
a[5] = a[4] * b[4] + c[4];
a[6] = a[5] * b[5] + c[5];
  :
```

ベクトルでの計算順序

```
a[3] = a[2] * b[2] + c[2];
a[4] = a[3] * b[3] + c[3];
a[5] = a[4] * b[4] + c[4];
a[6] = a[5] * b[5] + c[5];
  :
```

更新前の値

a[n] : 更新されたaの値

例2

```
for (i=2; i < n; i++)
  a[i-1] = a[i] * b[i] + c[i];
```

ベクトル化しても、計算順序は変わらないので、ベクトル化できる

スカラでの計算順序

```
a[1] = a[2] * b[2] + c[2];
a[2] = a[3] * b[3] + c[3];
a[3] = a[4] * b[4] + c[4];
a[4] = a[5] * b[5] + c[5];
  :
```

ベクトルでの計算順序

```
a[1] = a[2] * b[2] + c[2];
a[2] = a[3] * b[3] + c[3];
a[3] = a[4] * b[4] + c[4];
a[4] = a[5] * b[5] + c[5];
  :
```

ループの繰り返し間で、右下向きの矢印ができないかに注目する

ベクトル化不可の依存関係 (2)

例3

```
for (i = 0; i < n; i++) {
   a[i] = s;
   s = b[i] + c[i];
}
```

変数の参照が、定義よりも先に現れるループはベクトル化できない

```
a[0] = s
for (i = 1; i < n; i++) {
   s = b[i-1] + c[i-1];
   a[i] = s;
}
s = b[n-1] + c[n-1];
```

プログラムを変更することでベクトル化できる

スカラでの計算順序

```
a[0] = s ;
s = b[0] + c[0] ;
a[1] = s ;
s = b[1] + c[1] ;
      :
```

ベクトルでの計算順序

```
a[0] = s ;
a[1] = s ;
      :
a[n-1] = s ;
s = b[0] + c[0] ;
s = b[1] + c[1] ;
      :
```

スカラでの実行順序

```
a[0] = s ;
s = b[0] + c[0] ;
a[1] = s ;
s = b[1] + c[1] ;
      :
```

ベクトルでの実行順序

```
a[0] = s ;
s = b[0] + c[0] ;
s = b[1] + c[1] ;
      :
a[1] = s ;
a[2] = s ;
      :
```

ベクトル化不可の依存関係 (3)

例4

```
s = 1.0;
for (i=0; i < n; i++) {
   if (a[i] < 0.0)
      s = a[i];
   b[i] = s + c[i];
}
```

変数の参照の前に定義があっても、定義が実行されない可能性があるとベクトル化できない

例5

```
for (i=0; i < n; i++) {
   if (a[i] < 0.0)
      s = a[i];
   else
      s = d[i];
   b[i] = s + c[i];
}
```

sの参照の前に必ずsの定義があるのでベクトル化できる

例6

```
for (i=1; i < n; i++) {
   a[i] = a[i+k] + b[i];
}
```

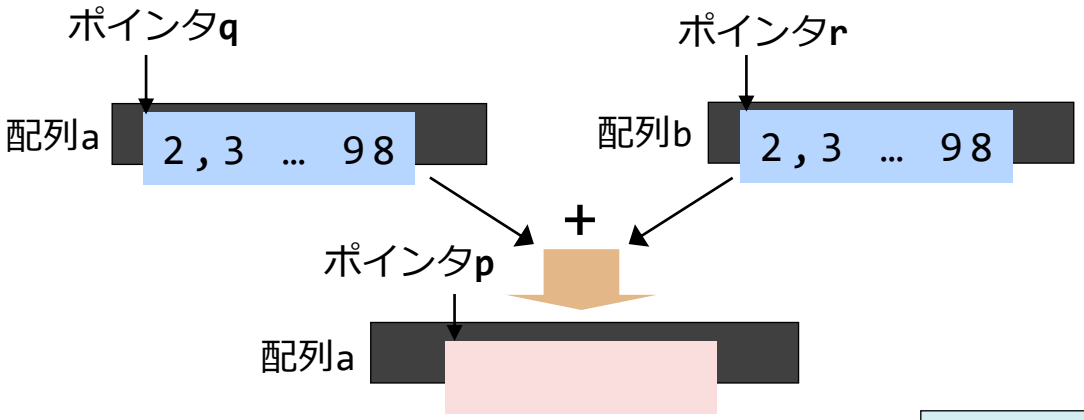
コンパイル時にkの値が不明なため、依存関係の有無が判定できないので、ベクトル化できない

(例1のパターンか例2のパターンか不明)

C/C++のポインタとベクトル化

例1: p=&a[3]、q=&a[2]のときはベクトル化不可

a[i+1]=a[i]+…のパターン



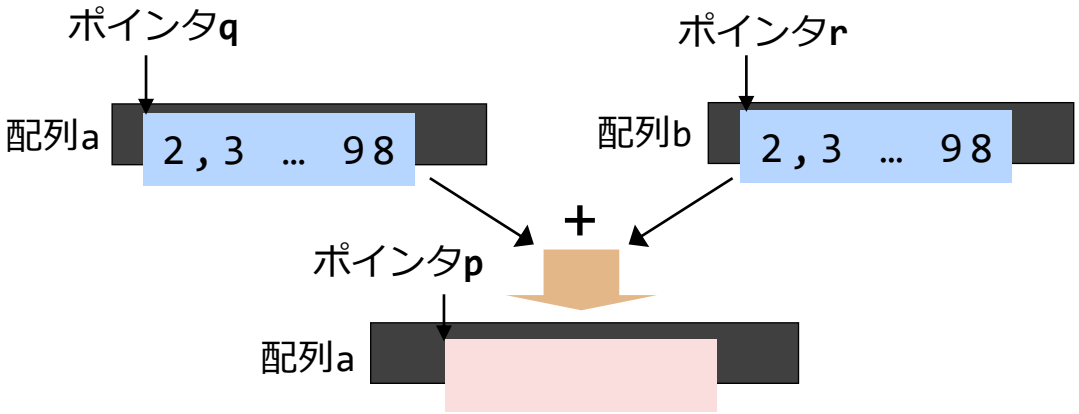

```
for (i = 2 ; i < n; i++) {
  *p = *q + *r;
  p++, q++, r++;
}
```

ポインタ値はプログラム実行時に確定する

結果不正を避けるため、明らかな場合を除いて、ベクトル化不可の依存関係があるものとみなしてベクトル化しない

例2: p=&a[1]、q=&a[2]のときベクトル化できる

a[i-1]=a[i]+…のパターン

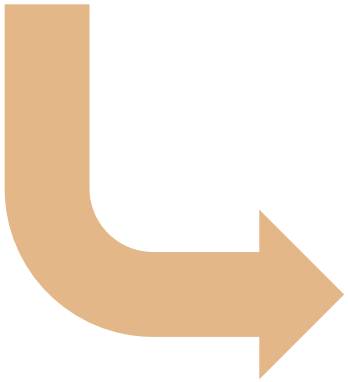


コンパイラオプション、***#pragma***などで、ベクトル化不可の依存関係がないことを伝え、ベクトル化する

if文のベクトル化

条件分岐(if文)もベクトル化される。

```
for (i = 0, i < 100; i++) {
  if (a[i] < b[i]) {
    a[i] = b[i] + c[i];
  }
}
```



ベクトル実行

```
mask[1]   = a[1]  <  b[1]
mask[2]   = a[2]  <  b[2]
  :          :        :
mask[100] = a[100] < b[100]
```

```
if (mask[1] == true)   a[1] = b[1] + c[1]
if (mask[2] == true)   a[2] = b[2] + c[2]
 :            :             :
if (mask[100] == true) a[100] = b[100] + c[100]
```

ベクトル化診断メッセージ

- コンパイラの出力するメッセージ、リストにより、ループのベクトル化状況、ベクトル化不可原因を調べることができる
  - 標準エラー出力　　… **-fdiag-vector=2**（詳細情報出力）
  - リストファイル出力　… **-report-diagnostics**

ベクトル化不可と思われる依存関係がポインタpにあったとみなし、ベクトル化しなかったことを示すメッセージ

```
$ ncc –fdiag-vector=2 abc.c
…
ncc: vec( 103): abc.c, line 1181: Unvectorized loop.
ncc: vec( 113): abc.c, line 1181: Unvectorizable dependency is assumed.: *(p)
ncc: vec( 102): abc.c, line 1234: Partially vectorized loop.
ncc: vec( 101): abc.c, line 1485: Vectorized loop.
…
$ ncc –report-diagnostics abc.c
…
$ less abc.L
FILE NAME: abc.c
…
FUNCTION NAME: func
DIAGNOSTIC LIST

 LINE              DIAGNOSTIC MESSAGE

  1181  vec( 103): Unvectorized loop.
  1181  vec( 113): Unvectorizable dependency is assumed.: *(p)
  1234  vec( 102): Partially vectorized loop.
  1485  vec( 101): Vectorized loop.
…
```

リストファイル名は「ソースファイル名.L」

編集リスト

ソース行とともにループ構造、そのベクトル化状況などを記号で表示

- **-report-format**が指定されたとき出力

```
$ ncc -report-format a.c -c
$ less a.L
:
FUNCTION NAME: func
FORMAT LIST

 LINE    LOOP       STATEMENT

     5:             void func(double *x, double *y, int n )
     6:             {
     7: +------>         for (int j = 0; j < n; j++) {
     8: |V----->             for (int i = 0; i < m; i++)
     9: |V-----                   a[i] += b[i] * c[j];
    10: +------         }
    11:
    12: +------>         for (int j = 0; j < n; j++) {
    13: |+----->             for (int i = 0; i < m; i++)
    14: |+-----                   x[j] = y[j] * a[i];
    15: +------         }
    16:             }
…
```

リストファイル名は「ソースファイル名.L」

ベクトル化されたループ

ベクトル化されなかったループ

拡張ベクトル化機能

拡張ベクトル化機能とは

そのままではベクトル化できない場合や、より効率のよいベクトル化が可能な場合に、コンパイラがプログラムを内部的に変形することで、ベクトル化の効果をさらに高める機能

- 文の入れ換え
- 多重ループの一重化
- 多重ループの入れ換え
- 部分ベクトル化
- 条件ベクトル化
- マクロ演算の認識
- 多重ループのベクトル化
- ループ融合
- インライン展開

文の入れ換え

ソースプログラム

```
for (i = 0; i < 99; i++) {
   a[i] = 2.0;
   b[i] = a[i+1];
}
```

コンパイラによる変形イメージ

```
for (i = 0; i < 99; i++) {
   b[i] = a[i+1];
   a[i] = 2.0;
}
```

そのままベクトル化すると、b[0]～b[98]の値がすべて2.0になってしまうので、このままではベクトル化不可

ループ内の文の順序を入れ換えることにより、ベクトル化できるように変形

多重ループの一重化

ソースプログラム

```
double a[M][N], b[M][N], c[M][N];
for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      a[i][j] = b[i][j] + c[i][j];
```

コンパイラによる変形イメージ

```
double a[M][N], b[M][N], c[M][N];
for (ij = 0; ij < M*N; ij++)
    a[0][ij] = b[0][ij] + c[0][ij];
```

ループ長(ループの繰り返し数)がより長くなるように、多重ループを一重化し、ベクトル命令の効率を高める

多重ループの入れ換え

ソースプログラム

```
for (j = 0; j < M; j++) {
  for (i = 0; i < N; i++) {
    a[i+1][j] = a[i][j] +  b[i][j];
   }
}
```

コンパイラによる変形イメージ

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      a[i+1][j] = a[i][j] + b[i][j];
   }
}
```

```
a[1][0] = a[0][0] + b[0][0];
a[2][0] = a[1][0] + b[1][0];
a[3][0] = a[2][0] + b[2][0];
a[4][0] = a[3][0] + b[3][0];
```

```
a[1][0] = a[0][0] + b[0][0];
a[1][1] = a[0][1] + b[0][1];
a[1][2] = a[0][2] + b[0][2];
a[1][3] = a[0][3] + b[0][3];
```

for (i=0; i<N; i++)でベクトル化しようとすると、配列**a**にベクトル化不可の依存関係がありベクトル化できない

ループを入れ換えると、 for (j=0; j<M; j++)のループに関してはベクトル化不可の依存関係がなくなりベクトル化できる

部分ベクトル化

ソースプログラム

```
for (i = 0; i < N; i++) {
    x = a[i] + b[i];
    y = c[i] + d[i];
    func(x, y);
}
```

コンパイラによる変形イメージ

```
for (i = 0; i < N; i++) {
    wx[i] = a[i] + b[i];
    wy[i] = c[i] + d[i];
}
for (i = 0; i < N; i++) {
   func(wx[i], wy[i]);
}
```

ベクトル化可能

ベクトル化不可

ループ構造に、ベクトル化できる部分とベクトル化できない部分が含まれている場合、ベクトル化可能な部分と不可能な部分に分割し、可能な部分だけをベクトル化する。

このとき必要であれば、作業ベクトル(上の例では配列wx、wy)を使用する。

条件ベクトル化

ソースプログラム

```
for (i = N; i < N+100; i++) {
    a[i] = a[i+k] + b[i];
}
```

コンパイラによる変形イメージ

```
if (k >= 0 || k < -99) {
    // ベクトル化したコード
}
else {
    // ベクトル化しなかったコード
}
```

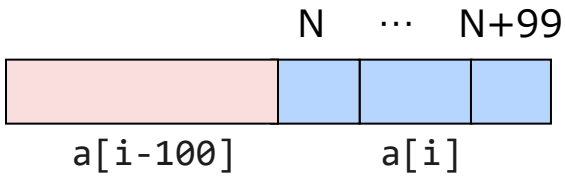
一つのループに対してベクトル化したコードとスカラのコード、特定のパターンのみ高速に実行できるコードなど、数種類のコードを用意し、実行時に条件を調べて、適切なコードを選択して実行するようにループを変形する。

(k=-1のとき)

```
a[i] = a[i-1]+b[i];
```

(k=-100のとき)

```
a[i] = a[i-100]+b[i];
```



マクロ演算の認識

総和型

```
for (i = 0; i < N; i++)
    s = s + a[i];
```

漸化式型

```
for (i = 0; i < N; i++)
    a[i] = a[i-1]*b[i]+c[i];
```

最大/最小型

```
for (i = 0; i < N; i++) {
    if (xmax < x[i])
        xmax = x[i];
}
```

配列や変数の定義・参照関係にベクトル化を阻害する依存関係があり、本来はベクトル化できない場合でも、コンパイラが特別なパターンであることを認識し、特別なベクトル命令を用いることで、ベクトル化する

外側ループのベクトル化

ソースプログラム

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++)
        a[i][j] = 0.0;
    b[i] = 1.0;
}
```

コンパイラによる変形イメージ

```
for (i = 0; i < N; i++) {
    for (j = 0; j <N; j++)
        a[i][j] = 0.0;
}
for (i = 0; i < N; i++)
    b[i] = 1.0;
```

この例ではさらにループの一重化が適用される

ベクトル化は、基本的に最内側ループをベクトル化するが、外側のループを二つに分割することによって、外側のループにのみ含まれる文もベクトル化する

ループ融合

ソースプログラム

```
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
for (j = 0; j < N; j++)
    d[j] = e[j] * f[j];
```

コンパイラによる変形イメージ

```
for (i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
    d[i] = e[i] * f[i];
}
```

コンパイラは同じ繰り返し回数を持つ複数のループを一つにまとめてベクトル化する

インライン展開によるベクトル化

ソースプログラム

```
for (i = 0; i < N; i++) {
    b[i] = func(a[i]);
    c[i] = b[i];
}
…
double func(double x)
{
    return x*x;
}
```

コンパイラによる変形イメージ

```
for (i = 0; i < N; i++) {
    b[i] = a[i] * a[i];
    c[i] = b[i];
}
…
double func(double x)
{
    return x*x;
}
```

-finline-functionsコンパイラオプションを指定すると、
可能であれば関数を呼び出し元にインライン展開する。ループ中に関数の呼び出しがあれば、インライン展開後にベクトル化を試みる

プログラムのチューニング

コンパイラオプションを追加指定したり、プログラムへの***#pragma***行の挿入などにより、
プログラムを高速化する(実行時間を短くする）ことを「チューニング」と呼びます。
チューニングにより、Vector EngineのHW性能を最大限まで引き出すことができます。

チューニングの観点

ベクトル化率を高める

- ベクトル化率とは、プログラム全体のうち、ベクトル命令で実行可能な部分の比率
- ベクトル化不可の要因を取り除き、ベクトル化を促進
  - ベクトル命令で実行可能な部分を増やす

ベクトル命令の効率を高める

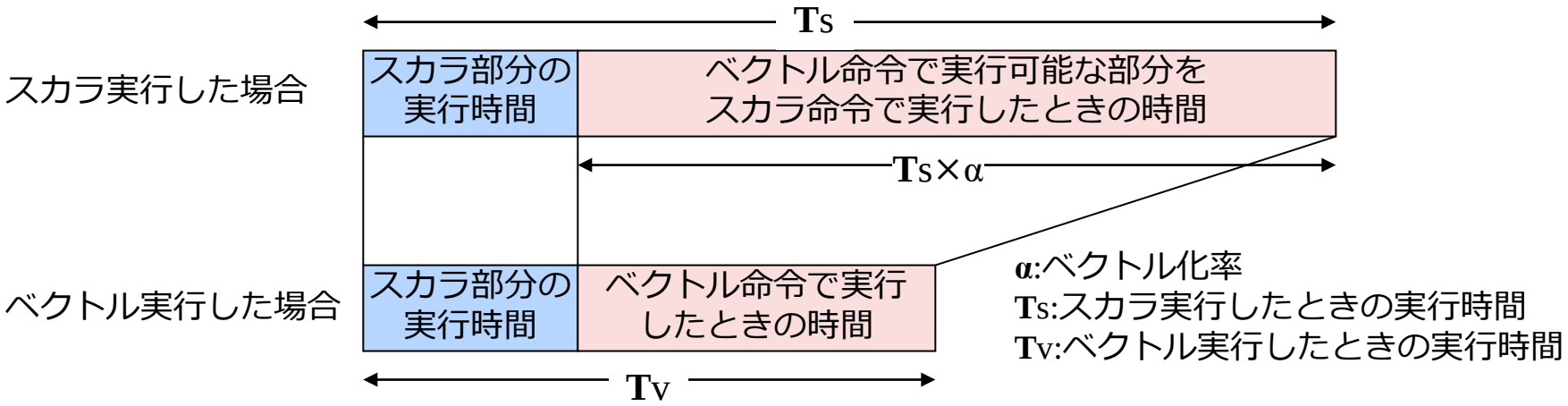
- 一つのベクトル命令で処理されるデータの個数を増やす
  - ループの繰り返し数(ループ長)を大きくする
- ループの繰り返し数が極端に短いループはベクトル化をやめる
  - p.21「命令実行時間の比較」のシートを参照

メモリアクセスの効率を高める

- リストベクトルの使用を避ける

ベクトル化率

- プログラム全体のうち、ベクトル命令で実行可能な部分の比率

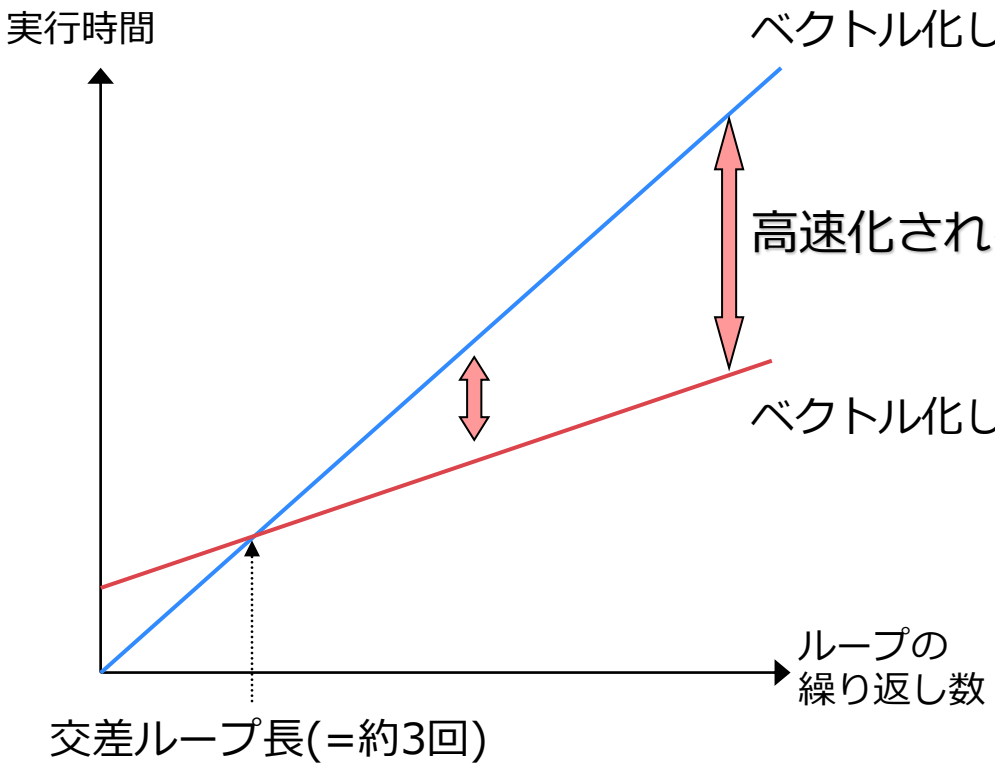


- 一般に、ベクトル化率を正確に求めることは困難であるため、ベクトル演算率で代用

$$\text{ベクトル演算率} = 100 \times \frac{\text{ベクトル命令で処理されたデータの個数}}{\text{全命令実行数} - \text{ベクトル命令実行数} + \text{ベクトル命令で処理されたデータの個数}}$$

ループの繰り返し数と実行時間

- ループの繰り返し数をできるだけ大きくした方が、ベクトル化による高速化の効果が大きい
  - 一つのベクトル命令で処理できるデータの個数が多くなる



繰り返し数をループごとに分析するのは困難

平均ベクトル長で分析

一つのベクトル命令が処理したデータの個数の平均。
最大256個である。

チューニングの手順

- 性能測定機能のプログラムの性能情報より、実行時間の長い関数、ベクトル演算率が低い、平均ベクトル長が短い関数を特定
  - PROGINF
    - プログラム全体の実行時間、ベクトル演算率、平均ベクトル長
  - FTRACE
    - 関数ごとの実行時間、実行回数、ベクトル演算率、平均ベクトル長

- 特定した関数のベクトル化診断メッセージを参照し、ベクトル化されていないループを特定

- コンパイラオプション、**#pragma**等を挿入し、ベクトル化を促進

PROGINF情報の利用

出力例

```
    ********  Program  Information  ********
Real Time (sec)                   :           11.336602
User Time (sec)                   :           11.330778
Vector Time (sec)                 :           11.018179
Inst. Count                       :         6206113403
V. Inst. Count                    :         2653887022
V. Element Count                  :       619700067996
V. Load Element Count             :        53789940198
FLOP count                        :       576929115066
MOPS                              :        73455.206067
MOPS (Real)                       :        73370.001718
MFLOPS                            :        50950.894570
MFLOPS (Real)                     :        50891.794092
A. V. Length                      :          233.506575
V. Op. Ratio (%)                  :           99.572922
L1 Cache Miss (sec)               :            0.010855
CPU Port Conf. (sec)              :            0.000000
V. Arith. Exec. (sec)             :            8.410951
V. Load Exec. (sec)               :            1.386046
VLD LLC Hit Element Ratio (%)     :          100.000000
Power Throttling (sec)            :            0.000000
Thermal Throttling (sec)          :            0.000000
Max Active Threads                :                   1
Available CPU Cores               :                   8
Average CPU Cores Used            :            0.999486
Memory Size Used (MB)             :          204.000000
```

A.V.Length (平均ベクトル長)

- ベクトル命令の効率を表す指標
- 大きいほどよい (最大256)
- 小さいとき、ベクトル化されたループの繰り返し数が小さい。大きくできないか検討する

V.Op.Ratio (ベクトル演算率)

- ベクトル命令で処理されたデータの比率
- 大きいほどよい (最大100)
- 小さいとき、ベクトル化されたループが少ない、あるいは、ループ自体がプログラム中に少ない。ベクトル化できるループが他にないか検討する

FTRACE

関数ごとに性能情報を採取する機能

- PROGINFと同じように、V.OP.RATIO (ベクトル演算率)、AVER.V. LEN (平均ベクトル長) に注目し、関数ごとに分析する

```
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Thu Mar 22 15:47:42 2018 JST
Total CPU Time : 0:00'11"168 (11.168 sec.)


FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS     CONF HIT E.%

    15000     4.767( 42.7)     0.318  77030.2  61964.6 99.45 251.0     4.610   0.002    0.000  100.00 funcA
    15000     3.541( 31.7)     0.236  73505.6  56940.8 99.46 216.0     3.555   0.000    0.000  100.00 funcB
    15000     2.726( 24.4)     0.182  71930.1  27556.5 99.43 230.8     2.725   0.000    0.000  100.00 funcC
        1     0.134(  1.2)   133.700  60368.9  35641.3 98.53 214.9     0.118   0.000    0.000    0.00 main
-------------------------------------------------------------------------------------------------------------
    45001    11.168(100.0)     0.248  74468.3  51657.9  99.44 233.5    11.008   0.002    0.000  100.00 total
```

プログラムの
チューニング・テクニック

コンパイラ指示行

自動ベクトル化、最適化の効果を促進させるために、**#pragma**を書くことで、コンパイル時にわからない情報を与える。これをコンパイラ指示行と呼ぶ

- コンパイラ指示行の形式

  #pragma△**_NEC**△指示オプション　(△:空白)

- 主なベクトル化指示オプション
  - **vector/novector**　　：自動ベクトル化の対象とする/しない
  - **ivdep**　　：ベクトル化不可の依存関係がない

```
#pragma _NEC ivdep
  for (i = 2 ; i < n; i++)
    {
     *p = *q + *r;
     p++, q++, r++;
  }
```

- ベクトル化指示オプションはループの直前に指定
- 空白で区切って指定する
- 指示行の直後のループにのみ効果がある

ベクトル化不可の依存関係の対処 (1)

ベクトル化率を高める

```
ncc: vec( 103): a.c, line 16: Unvectorized loop.
ncc: vec( 113): a.c, line 16: Overhead of loop division is too large.
ncc: vec( 121): a.c, line 18: Unvectorizable dependency.
```

部分ベクトル化を試みるため、このようなメッセージが表示されることがある(以降、省略)

変数tが定義されるかどうかわからないのでベクトル化できない

Unvectorized Loop

```
for (i=0; i<N; i++) {
   if (x[i] < s)
      t = x[i];
   else if (x[i] >= s)
      t = -x[i];
   y[i] = t;
}
```

Vectorized Loop

```
for (i=0; i<N; i++) {
   if (x[i] < s)
      t = x[i];
   else
      t = -x[i];
   y[i] = t;
}
```

変数tが必ず定義されるよう修正

総和型のマクロ演算と認識できない

Unvectorized Loop

```
for (i=0; i<N; i++) {
   if (a[i] < 0.0)
      s = s + b[i];
   else
      s = s + c[i];
}
```

Vectorized Loop

```
for (i=0; i<N; i++) {
   if (a[i] < 0.0)
      t = b[i];
   else
      t = c[i];
   s = s + t;
}
```

総和型のマクロ演算とし、ベクトル化

〈ベクトル化後の診断メッセージ〉

```
ncc: vec( 101): a.c, line 16: Vectorized loop.
ncc: vec( 126): a.c, line 21: Idiom detected.: Sum.
```

総和計算は特別なHW命令を使用してベクトル化

ベクトル化不可の依存関係の対処 (2)

ベクトル化率を高める

```
ncc: vec( 103): vec_dep2.c, line 7: Unvectorized loop.
ncc: vec( 113): vec_dep2.c, line 7: Overhead of loop division is too large.
ncc: vec( 122): vec_dep2.c, line 8: Dependency unknown. Unvectorizable dependency
is assumed.: a
```

- ベクトル化不可の依存関係が仮定されたが、実際にはベクトル化不可の依存関係がないことがわかっているとき、**ivdep**を指定する

Unvectorized Loop

```
#define N 1024
double a[N],b[N],c[N];
void func(int k, int n)
{
    int i;

    for (i=1; i < n; i++)
        a[i+k] = a[i] + b[i];
}
```

a[i-1]=a[i]のパターンか、a[i+1]=a[i]のパターンか不明なのでベクトル化しない

Vectorized Loop

```
#define N 1024
double a[N],b[N],c[N];
void func(int k, int n)
{
    int i;
#pragma _NEC ivdep
    for (i=1; i < n; i++)
        a[i+k] = a[i] + b[i];
}
```

a[i-1]=a[i]のパターンであることが明らかなとき、**ivdep**を指定してベクトル化

〈ベクトル化後の診断メッセージ〉

```
ncc: vec( 101): a.c, line 7: Vectorized loop.
```

ポインタの依存関係の対処

ベクトル化率を高める

```
ncc: vec( 103): a.c, line 12: Unvectorized loop.
ncc: vec( 122): a.c, line 13: Dependency unknown. Unvectorizable dependency
is assumed.: *(p)
```

ベクトル化不可の依存関係が仮定されたが、実際にはベクトル化不可の依存関係がないことがわかっているとき、**ivdep**を指定する

Vectorized Loop

```
main() {
double *p = (double *) malloc(8*N);
double *q = (double *) malloc(8*N);
…
func(p,q);
…
}
void func(double *p, double *q) {
  …
#pragma _NEC ivdep
  for (int i = 0; i < n; i++) {
      p[i] = q[i];
  }
}
```

malloc(3C)で別々に確保された領域であるので、p[i]、q[i]の間にベクトル化不可の依存関係はないが、関数func()内ではそれがわからない

プログラマにはベクトル化不可の依存関係がないことが明らかなので、**ivdep**を指定できる

ivdepが指定されたとしても、あきらかにベクトル化不可の依存関係があるとき、コンパイラはそれを無視せず、ループのベクトル化を行わない

実際にベクトル化不可の依存関係があるとき、*ivdep*を指定してしまうと結果不正となることがあるので注意すること!!

ポインタの依存関係の対処:restrict

ベクトル化率を高める

```
ncc: vec( 103): a.c, line 16: Unvectorized loop.
ncc: vec( 121): a.c, line 18: Unvectorizable dependency is assumed: *(p)
```

restrictキーワード (ポインタ修飾子)

- あるポインタで指された領域が、別のポインタで指されたり、別の変数名で定義、参照されることはないことを示す
- 別のポインタ、変数で定義、参照されないことが保証されており、ポインタ間のベクトル化不可の依存関係がないものとしてベクトル化を適用する

Vectorized Loop

```
void func(double * restrict p, double * restrict q)
{
  …
  for (int i = 0; i < N; i++) {
      p[i] = q[i];
  }
  …
}
```

restrictが指定されたとしても、あきらかに、別のポインタ、変数で定義、参照されているとき、コンパイラはそれを無視せず、ループのベクトル化を行わない

実際に、別のポインタで指されたり、変数で定義、参照されているとき、*restrict*を指定してしまうと結果不正となることがあるので注意すること!!

ループ構造の改善によるベクトル化

ベクトル化率を高める

```
ncc: vec( 103): a.c, line 16: Unvectorized loop.
ncc: vec( 121): a.c, line 16: Unvectorizable loop structure.
```

ループ構造に以下の問題があるとき、ベクトル化できないことがある

- インダクション変数が、型変換されている
- ループ終了条件式に、**!=**、**==**演算子が含まれる
- ループ終了条件式に、**&&**、**||**演算子が含まれる

ループの繰り返し数がループ開始前に算定できない

終了条件式中に分岐が複数存在

```
for (j=0; j < m; j++) {
  for (i=0; i < n; i++) {
    a[i] = b[j] + c[j];
  }
}
```

- i、jがインダクション変数
- j<m、i<nがループ終了条件式

※ インダクション変数 := その値がループの繰り返し数に従って、単調増加、または、単調減少する変数

ループ終了条件式の!=、 ==

ベクトル化率を高める

- !=、 ==がループ終了条件式に現れたとき、条件式が必ずしも成立するとは限らないので、ループの繰り返し数を算定できない
  - 終了条件式では、できるだけ!=、 ==を使わず、 <、 >、 <=、 >=を使う

Unvectorized Loop

```
for (i=0; i != n; i+=2) {
    ……
}
```

nが奇数のとき条件は成立しない

Vectorized Loop

```
for (i=0; i < n; i+=2) {
    ……
}
```

i<nに修正

Unvectorized Loop

```
double *first, *last, *p;
……
for (p=first; p != last; p++)
{
    ……
}
```

C++のiteratorタイプの配列

Vectorized Loop

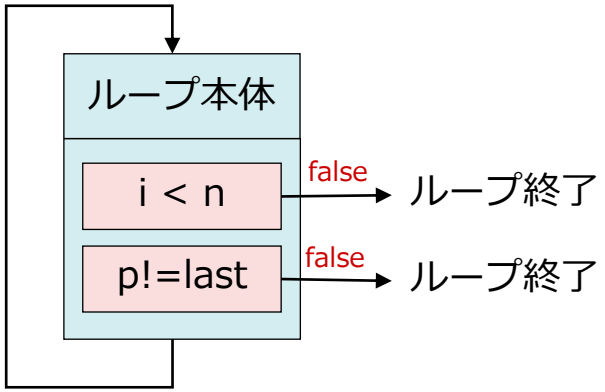
```
double *first, *last, *p;
……
for (p=first; p < last; p++)
{
    ……
}
```

ループ終了条件式の&&、||

ベクトル化率を高める

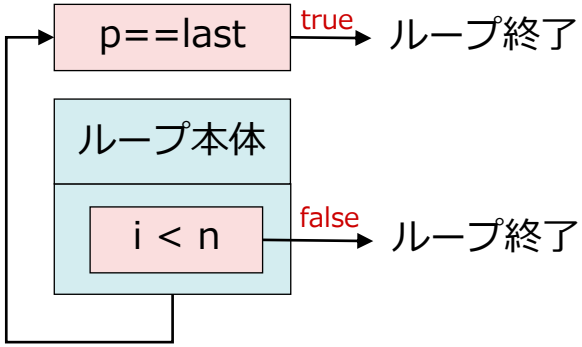
- ループ終了条件式に**&&**、**||**が現れたとき、ループ終了判定からの飛び出しが複数存在することになるので、ベクトル化できない
  - ループ終了条件式中では、**&&**、**||**を使わない
  - 条件式の一部をループ本体に記述し、**break**でループを抜けるようにする

```
double func(double *first, double *last, double *a, int n)
{
    double *p = first;
    double sum = 0.0;
                        /* Unvectorizable loop structure */
    for (int i = 0; i < n && p != last; i++, p++) {
        sum += a[i] * (*p);
    }
    return sum;
}
```



ループ終了条件式の処理

```
double func(double *first, double *last, double *a, int n)
{
    double *p = first;
    double sum = 0.0;
                        /* Vectorizable */
    for (i = 0; i < n; i++, p++) {
        if (p == last) break;
        sum += a[i] * (*p);
    }
    return sum;
}
```



インライン展開によるベクトル化促進

ベクトル化率を高める

```
ncc: vec( 103): a.c, line 9: Unvectorized loop.
ncc: vec( 110): a.c, line 10: Vectorization obstructive procedure reference.: fun
```

- 関数呼び出しがベクトル化を妨げているとき出力される
- 以下のどちらかで関数のインライン展開を試みる
  - **-finline-functions**コンパイラオプション
  - 関数宣言時にinline関数と指定

```
#include <math.h>
double fun(double x, double y)
{
  return sqrt(x)*y;
}
…
for (i=0; i<N; i++) { // Unvectorized
    a[i] = fun(b[i], c[i]) + d[i];
  }
…
```

double sqrt(double)は、ベクトル処理可能な関数でありベクトル化を妨げない

< inline関数を指定する場合>

```
#include <math.h>
inline double fun(double x, double y)
{
  return sqrt(x)*y;
}
…
for (i=0; i<N; i++) {  // Vectorized
    a[i] = fun(b[i], c[i]) + d[i];
  }
…
```

<コンパイラオプションを指定する場合>

```
$ ncc –finline-functions a.c
```

添字式に配列の現れるループ

ベクトル化命令の効率を高める

```
ncc: vec( 103): a.c, line 8: Vectorized loop.
ncc: vec( 126): a.c, line 9: Idiom detected.: List Vector
```

リストベクトルを**ivdep**を指定することでさらに高速化

- 添字式に配列が現れる配列をリストベクトルと呼ぶ
- 両辺に同じリストベクトルが現れたとき、その依存関係が不明であるのでベクトル化できない

Vectorized Loop (**list_vector**指定行)

```
#pragma _NEC list_vector
for (i=0; i < n; i++) {
    a[ix[i]] = a[ix[i]] + b[i];
}
```

Vectorized Loop (**ivdep**指示行)

```
#pragma _NEC ivdep
for (i = 0; i < n; i++) {
    a[ix[i]] = a[ix[i]] + b[i];
}
```

list_vectorを指定するとベクトル化できるが、配列aの要素がループ中で2回以上定義されないとき、つまり、ix[i]の値が同じになるiがなければ、***ivdep***を指定することで、より効率のよいベクトル命令列で計算できる

<**ivdep**によるベクトル化後のメッセージ>

```
ncc: vec( 101): a.c, line 8: Vectorized loop.
```

外側ループのアンローリング

ベクトル化命令の効率を高める

外側ループのアンロールすることでロード、ストア回数を減らす

- ループを展開することをアンローリングという
- 2つ以上のループのネストがあるとき外側ループをアンロールすることで内側ループのインダクション変数のみを使用するロード、ストア数を減らせる

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}
```

outerloop_unroll(4)指示行挿入

括弧内にアンロール段数2^xを指定する

```
#pragma _NEC outerloop_unroll(4)
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}
```

外側ループを4段にアンロール後のプログラム

配列cのベクトルロード1回に付き4回ベクトル演算を行える

```
for (int i = 0; i <(n%3) ; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = b[i][j] + c[j];
    }
}

for (int i =(n%3) ; i < n; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j]   = b[i][j]   +c[j] ;
        a[i+1][j] = b[i+1][j] +c[j] ;
        a[i+2][j] = b[i+2][j] +c[j] ;
        a[i+3][j] = b[i+3][j] +c[j] ;
    }
}
```

outerloop_unroll指示行、または、**-fouterloop-unroll**指示行を指定すると
外側ループ(インダクション変数**i**)のループ長が短くなり、配列**c**のベクトルロード回数が減る

<**outerloop_unroll**指示行による外側ループアンロール後のメッセージ>

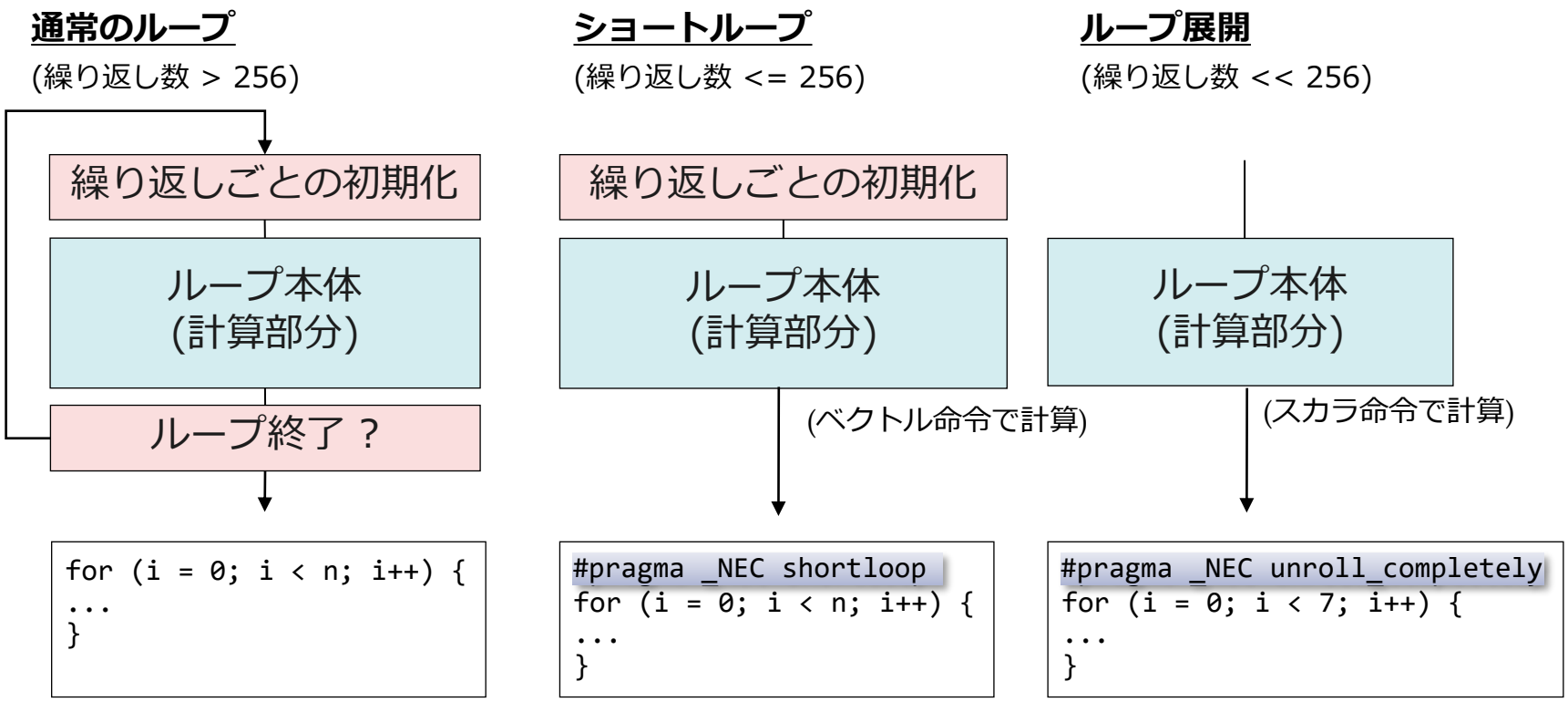
```
ncc: opt(1592): a.c, line 3: Outer loop unrolled inside inner loop.: i ncc:
vec( 101): a.c, line 4: Vectorized loop.
```

繰り返し数の小さいループ

ベクトル化命令の効率を高める

繰り返し数が小さいとき、ループ制御の処理を省いて高速化

- 繰り返し数<=256 … ショートループにし、ループ終了処理を削除
- 繰り返し数<<256 … ループ展開し、計算部分以外を削除



自動ベクトル化における注意事項

自動ベクトル化のレベルと適用される最適化

-04、-03、-02で自動ベクトル化のレベルを変更したとき、次のベクトル化/最適化が自動的に適用される

高 ⟷ 低

適用されるベクトル化・最適化	-04	-03	-02
条件ベクトル化によるベクトル化 (**-m[no-]vector-dependency-test**)	○	○	○
一重化、入れ換え、行列積ライブラリ関数置換によるベクトル化 (**-f[no-]loop-collapse**、**-f[no-]loop-interchange**、**-f[no-]matrix-multiply**)	○	○	–
ループ中の配列を指すポインタ間にはベクトル化不可の依存関係がないと仮定 (**-fnamed-[no]alias**)	○	○	–
外側ループのアンローリングを許可する (**-f[no-]outerloop-unroll**)	○	○	–
ループ終了条件式の!=、==を最適化 (**-f[no-]replace-loop-equation**)	○	–	–

＊主なもののみ記載、()内は個別に指定するときのコンパイラオプション

ベクトル化による演算結果への影響

- ベクトル化した場合としない場合で、演算結果が誤差範囲で異なることがある
  - 最適化・ベクトル化による演算順序の変更や除算の乗算化により、情報落ち・桁落ち・丸め誤差が変わるため
  - ベクトル化された数学関数では、高速にベクトル計算できるようスカラ版の数学関数と異なる計算アルゴリズムを使用しているため
  - 整数型漸化式マクロ演算では、浮動小数点数のベクトル命令を使用するため、52ビットで表現できる整数値のみ計算可能
  - ベクトル融合積和演算(FMA)が使用された場合、途中の積算結果を丸めずに和算が行われるため、使用しない場合に対して異なる演算結果となる可能性がある
- 誤差が気になる場合
  - **novector**指示行でループがベクトル化されないようにする
  - **nofma**指示行でベクトル積和演算が行われないようにする

```
#pragma _NEC novector
    for (i=0; i < n; i++) {
        sum += a[i];
    }
```

ベクトル化による実行時バスエラーの発生

4バイトアラインされた配列を8バイト要素のベクトル命令でロード/ストアしている可能性がある

- 以下の例では、引数で渡された**float**型(4バイトアライン)の配列**a**、**b**が**uint64_t**型にキャストされているため、8バイト要素のベクトル命令でロード/ストアされる。
- 8バイトのベクトルロード/ストア命令は8バイトアラインを要求するため、ロード/ストアする配列が4バイトアラインされている場合、実行時にメモリの不正アクセスによってバスエラーが発生する。

```
void func1(){
  float a[512],b[512];
  func2(a,b);
}

void func2( void* a, void* b ){
  for(int i=0; i<256; ++i){      //!!!<---vectorized loop
    ((uint64_t*)b)[i] = ((uint64_t*)a)[i];
  }
}
```

4バイトデータ型として配列にアクセスするか、**novector**指示行でベクトル化を抑止する

4バイトデータ型として配列にアクセス

```
void func2( void* a, void* b ){
  for(int i=0; i<512; ++i){
    ((uint32_t*)b)[i] = ((uint32_t*)a)[i];
  }
}
```

novector指示行を指定

```
void func2( void* a, void* b ){
#pragma _NEC novector
  for(int i=0; i<256; ++i){
    ((uint64_t*)b)[i] = ((uint64_t*)a)[i];
  }
}
```

自動並列化機能・OpenMP C/C++

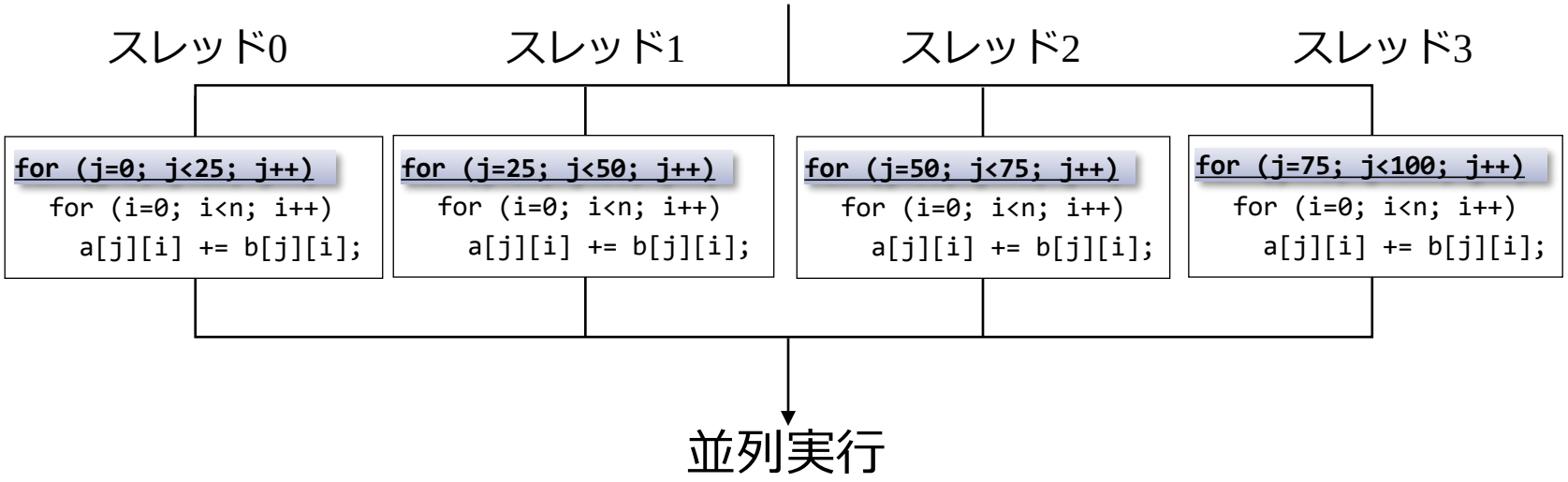
並列処理とは

- 一つの仕事を分割し、複数のスレッドで同時に実行すること
  - ループの繰り返しを分割
  - プログラム内の一連の処理(文の集まり)を分割

```
for (j=0; j<100; j++)
  for (i=0; i<100; i++)
    a[j][i] += b[j][i];
```

シリアル実行

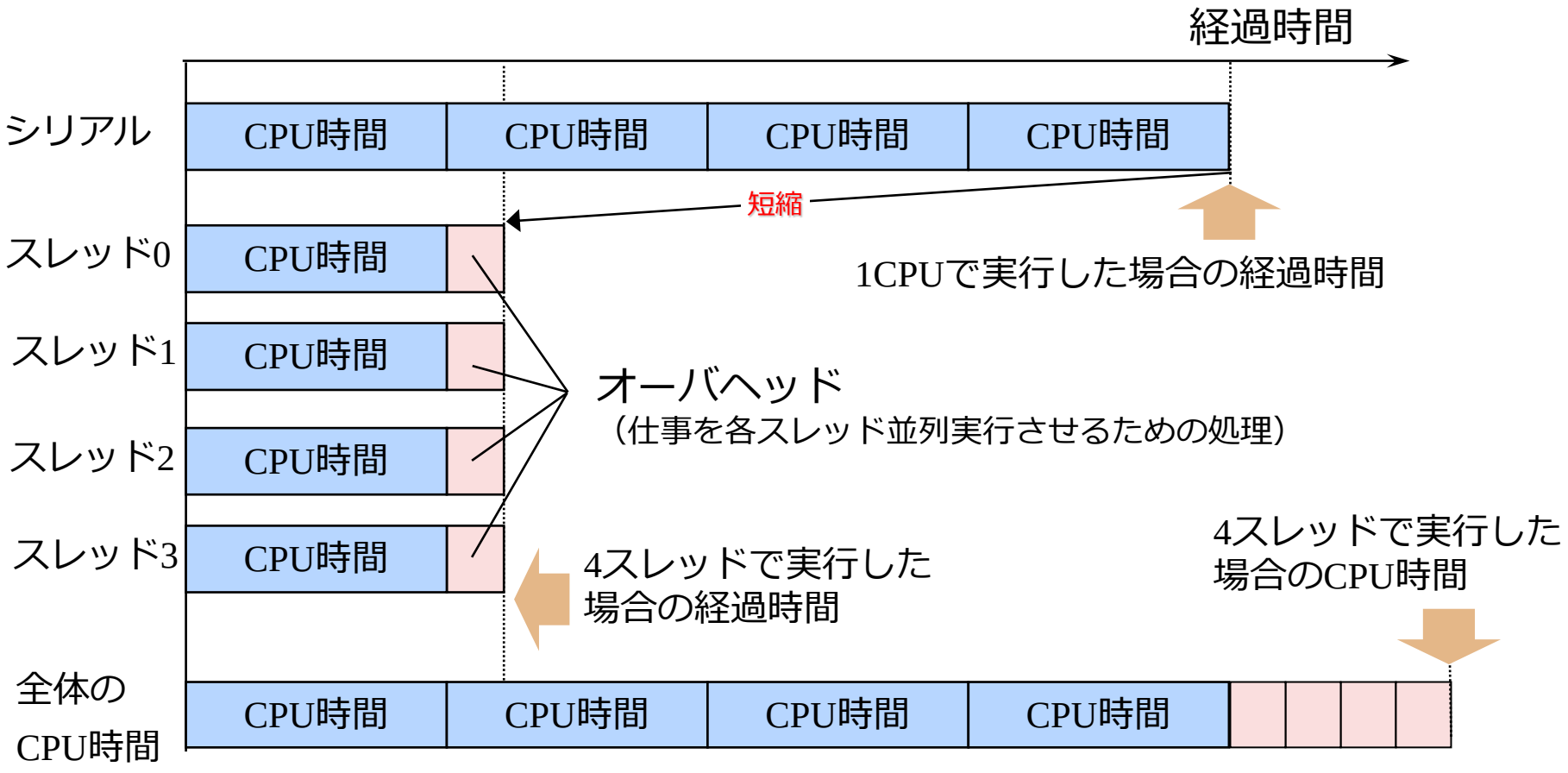
ループの繰り返しを四つに分割したときの例



並列処理による実行時間の短縮

並列処理により経過時間が短縮される

- 総CPU時間は並列処理のためのオーバヘッドなどで増加する



プログラムの並列化

複数のスレッドで並列実行できるようプログラミングすること

- ループや文の集まりを抽出し、並列処理できるようにプログラムを変形
- 自動並列化やOpenMPで並列実行する実行コードを生成

例*1.* 自動並列化による並列化

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;

 for (j=0; j<n; j++)  {
    for (i=0; i<n; i++)
       sum += a[j] + b[i];
  }

  return sum;
}
```

-mparallelを指定して自動並列化を有効にする

```
$ ncc -mparallel a.c
ncc: par(1801): ex1.c, line 6: Parallel routine generated.: sub$1
ncc: par(1803): ex1.c, line 6: Parallelized by "for".
ncc: vec( 101): ex1.c, line 7: Vectorized loop.
```

内側ループをベクトル化

for文を並列化

ループを並列実行するために別関数にして切り出す

並列実行できるループを探す

※ループ以外の部分は並列実行できないものとする

Vector Engineで利用できる並列化プログラミング

OpenMP C/C++

- プログラマが、並列実行できるループや文の集まりを抽出し、それらの並列化方法を示す指示行(OpenMPディレクティブ)を指定
- コンパイラが、その指示を元にプログラムを変形、並列処理制御のための指示行を挿入

自動並列化

- コンパイラが、並列実行できるループや文の集まりを抽出、プログラムを並列処理制御するように変形
- 前ページの「例1」のループの検出、プログラム変形、指示行の挿入のすべての作業をコンパイラが自動的に行う

プログラミング手法	ループ・文の集まりの抽出	指示行の挿入	プログラムの変形	難易度
OpenMP C/C++ (`-fopenmp`)	○	○	–	高
自動並列化 (`-mparallel`)	–	–	–	低

○：プログラマによる作業が必須
–：コンパイラが自動的に実施するので不要

※ チューニング時には「–」の項であってもプログラマによる作業が必要になることがある

OpenMP並列化

OpenMP C/C++

```
$ ncc -fopenmp a.c b.c
```

リンクのときにも**-fopenmp**を指定すること

共有メモリ型並列処理のための指示行・ライブラリなどの国際標準

- NEC C/C++ Compiler for Vector Engineでは、 OpenMP Version 4.5までの一部機能をサポート

プログラミング手法

- プログラマが、並列実行できるループや文の集まりを抽出し、それらの並列化方法を示す指示行(OpenMPディレクティブ)を指定
- コンパイラはその指示を元にプログラムを変形、並列処理制御のための処理を挿入
- **-fopenmp**を指定して、コンパイル、リンク

特徴

- プログラマが並列化部分を選択、指定できるため、自動並列化より高い性能向上が期待
- 並列化部分の切り出し、バリア同期、変数の共有属性にかかわるプログラム変形をコンパイラが行うため、プログラミングが容易

例:OpenMP C/C++による記述

例1の関数subのOpenMP C/C++による並列化

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;
#pragma omp parallel for
 for (j=0; j<n; j++)  {
    for (i=0; i<n; i++)
       sum += a[j] + b[i];
 }
  …
  return sum;
}
```

OpenMP指示行を挿入

並列実行できるループを探す

-fopenmpを指定して**OpenMP**指示行を有効にする

```
$ ncc -fopenmp a.c
ncc: par(1801): a.c, line 5: Parallel routine generated.: sub$1
Ncc: par(1803): a.c, line 6: Parallelized by "for".
ncc: vec( 101): a.c, line 7: Vectorized loop.
```

コンパイラが並列実行できるようにプログラムを変形する

OpenMPの指示行は#pragma ompに続けて並列化方法を指定する

```
#pragma omp parallel for
```

parallel
並列化区間の開始の指定

for
forループの並列化を指定

用語

- OpenMPスレッド (OpenMP thread)
  - 論理的な並列処理の単位。スレッドと略されることもある
- 並列リージョン (Parallel region)
  - 複数のOpenMPスレッドにより並列に実行される文の集まり
- 逐次リージョン (Serial region)
  - 並列リージョンの外側でマスタスレッドでのみ実行される文の集まり
- プライベート (Private)
  - 並列リージョンを実行するOpenMPスレッドのうちの一つのスレッドのみからアクセス可能であること
- 共有 (Shared)
  - 並列リージョンを実行するすべてのOpenMPスレッドからアクセス可能であること

OpenMPディレクティブ

よく使用されるもののみ説明

#pragma omp parallel for [*schedule*句] [nowait]

schedule(static[,*size*]) … schedule(static)が既定値

- *size*回の繰り返しをひとまとまりとし、OpenMPスレッドにラウンドロビン方式で割り当て実行する
- *size*の指定が省略されたとき、*size*をスレッド数で割った値が指定されたものとみなす

schedule(dynamic[,*size*])

- *size*回の繰り返しをひとまとまりとし、OpenMPスレッドに動的に割り当て実行する
- *size*の指定が省略されたとき、1が指定されたものとみなす

schedule(runtime)

- 環境変数**OMP_SCHEDULE**に設定されたスケジュール方法で実行する

nowait

- 並列ループ終了時の暗黙のバリア同期を行わない

#pragma omp single

ひとつのOpenMPスレッドでのみ実行する。マスタスレッドとは限らず、一番最後にディレクティブに到達したタスクで実行する

#pragma omp critical

同時に複数のOpenMPスレッドで実行しないようにする(排他制御)

自動並列化機能

自動並列化機能

自動並列化機能を利用したとき、「プログラムの並列化」で示したようにコンパイラがすべて自動で行う

```
$ ncc -mparallel a.c b.c
```

リンクのときにも**-mparallel**を指定すること

-mparallelを指定してコンパイル、リンク

- 並列実行できるループや文の集まりを抽出し、並列処理できるようにプログラムを変形する
  - 並列化の阻害要因を含まないループの自動選択
  - 多重ループの外側ループを自動選択
    - –最内側ループはベクトル化を使って高速化

コンパイラ指示行による自動並列化の制御

- コンパイラ指示行の形式

  #pragma△**_NEC**△指示オプション　(△:空白)

- 主な指示オプション
  - **concurrent/noconcurrent** … 直後のループの並列化を許可する/しない
  - **cncall** … ループ中に関数の呼出しがあるときでも並列化を許可する

データの共有属性

データ共有属性はコンパイラが自動で判断する

- **共有**
  - 関数スコープ外、**extern**、**static**で宣言された変数
  - 並列ループなどを含む関数の引数等
- **プライベート**
  - 共有変数の条件を満たさない変数

```
double a[N], b[M];
static double x[M];
double func()
{
    double wk[M];
    double sum = 0.0;

    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            wk[i] = a[i] + b[j];
            sum += x[j]*wk[i];
        }
    }
    return sum;
}
```

- 配列**a**、**b**、**x**は、共有として参照される
- ループ内の中間結果を保持するための配列**wk**、ループ制御変数である**i**、**j**は、プライベートとして参照される
- 変数**sum**は、スレッドごとの計算結果を足し合わせる必要があるため、共有として参照される

共有である配列wkはスレッドごとに確保されるため、
そのサイズが大きいとメモリ使用量が著しく増大する。
可能であればスカラ変数に置き換えたほうがよい。

指示行による自動並列化の制御

noconcurrent ... 直後のループの並列化を許可しない

```
(void) func(4);            // 関数呼び出し
...
void func(int m) {
#pragma _NEC noconcurrent
    for (j=0; j < m; j++) { // 実はmの値が小さい
        for (i=0; i < n; i++)
            a[i] = b[j] / c[j];
    }
}
```

繰り返し数の少ないループが並列化されると、並列化のためのオーバヘッドの占める比率が大きく、性能が低下してしまうことがある

noconcurrentで並列化を抑止

cncall ... ループ中に関数呼出しがあるときでも並列化を許可する

```
#pragma _NEC cncall
for (i=0; i < m; i++) {
  a[i] = func(b[i], c[i]);
}
```

関数が並列実行できるかどうかわからないため、関数呼び出しを含むループは自動並列化されない

関数が並列実行できるとき、**cncall**を指定

(関数が並列実行できることは、プログラマが保証しなければならない)

OpenMP・自動並列化機能の同時利用

```
$ ncc -fopenmp -mparallel a.c b.c
```

-fopenmpと**-mparallel**の両方を指定してコンパイル、リンク

- OpenMP並列リージョン外のループが自動並列化の対処となる
- OpenMPディレクティブを含む関数を自動並列化したくないとき、**-mno-parallel-omp-routine**を指定する

```
double sub (double *a, int n)
{
  int i, j;
  double b[n][n];
  double sum = 1.0;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      b[i][j] = i * j;

#pragma omp parallel for
  for (j=0; j<n; j++) {
    for (i=0; i<n; i++)
      sum += a[j] + b[i][j];
  }

  return sum;
}
```

自動並列化される

OpenMP並列化される

```
$ ncc -fopenmp -mparallel t.c
ncc: par(1801): t.c, line 7: Parallel routine generated.: sub$1
ncc: par(1803): t.c, line 7: Parallelized by "for".
ncc: par(1801): t.c, line 11: Parallel routine generated.: sub$2
ncc: vec( 101): t.c, line 8: Vectorized loop.
ncc: par(1803): t.c, line 12: Parallelized by "for".
ncc: vec( 101): t.c, line 13: Vectorized loop.
```

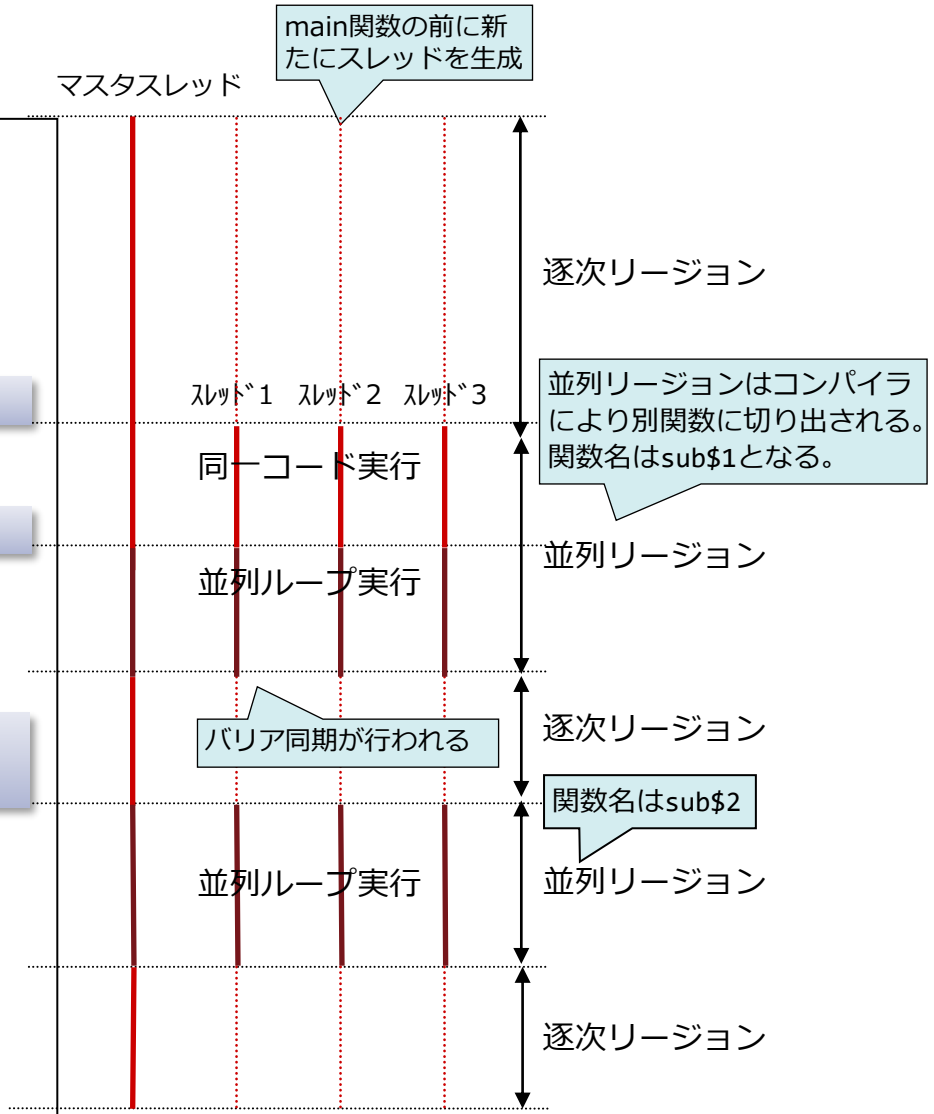
並列処理プログラムの動作

OpenMP並列化されたプログラムの実行イメージ

*OpenMP*を用いて並列化したとき

```
double sub (double *a, int n)
{
  int i, j;
  double b[n];
  double sum = 1.0;
  double derive;
#pragma omp parallel private(derive)
   {
      derive = 12.3;
#pragma omp for
      for (i = 0; i < n; i++)
        b[i] = derive;
   }
   ….
#pragma parallel omp for   ¥
                   reduction(+:sum)
      for (j=0; j<n; j++)  {
         for (i=0; i<n; i++)
            sum += a[j] + b[i];
       }
   …
  return sum;
}
```

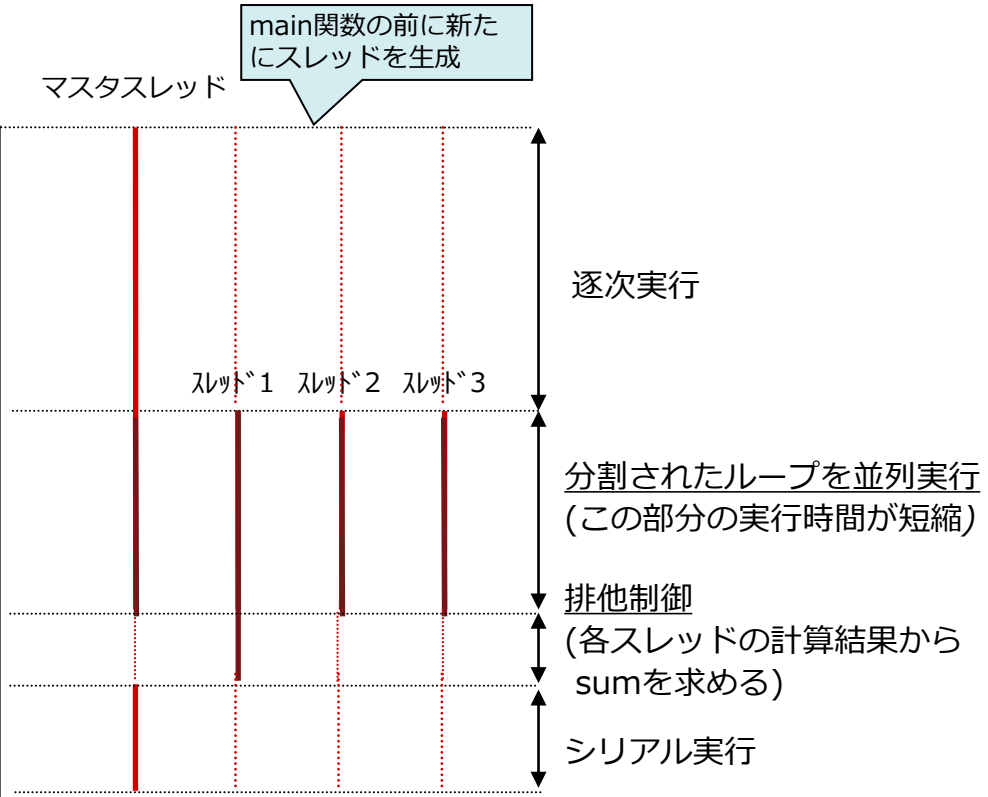
指定がなければ、ループ制御変数を除いて、すべてsharedデータとなる



※ VEでは、ネスト並列 (nested parallelism) はサポートしていない。

自動並列化されたプログラムの実行イメージ

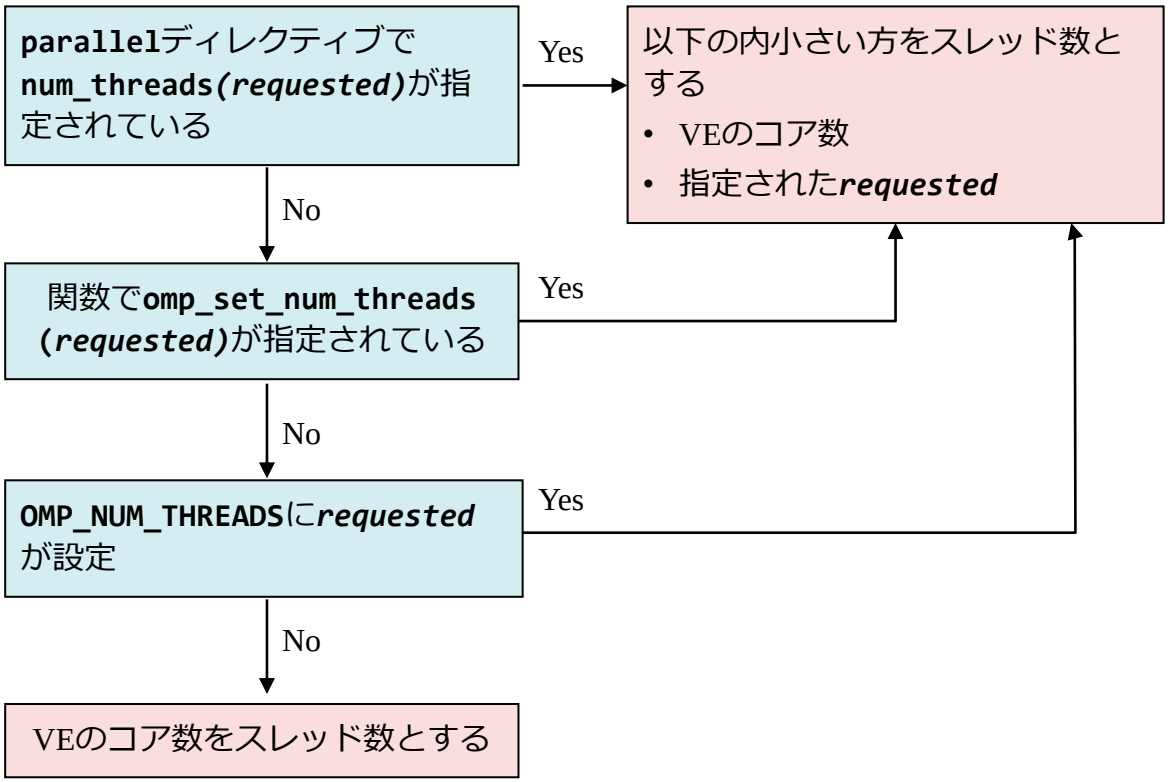
```
double sub (double *a, double *b, int n)
{
  int i, j;
  double sum = 0.0;


  for (j=0; j<n; j++)  {
     for (i=0; i<n; i++)
        sum += a[j] + b[i];
   }

  return sum;

}
```



(実線：プログラムの実行、破線：待ち合わせ処理)

OpenMP・自動並列でのスレッド数の決定

並列処理に使用するスレッド数は以下のルールで決定



※ VEのコア数は8個であるため、８より大きいスレッド数を指定しても
最大８スレッドまでしか生成されない。

並列処理プログラムのチューニング

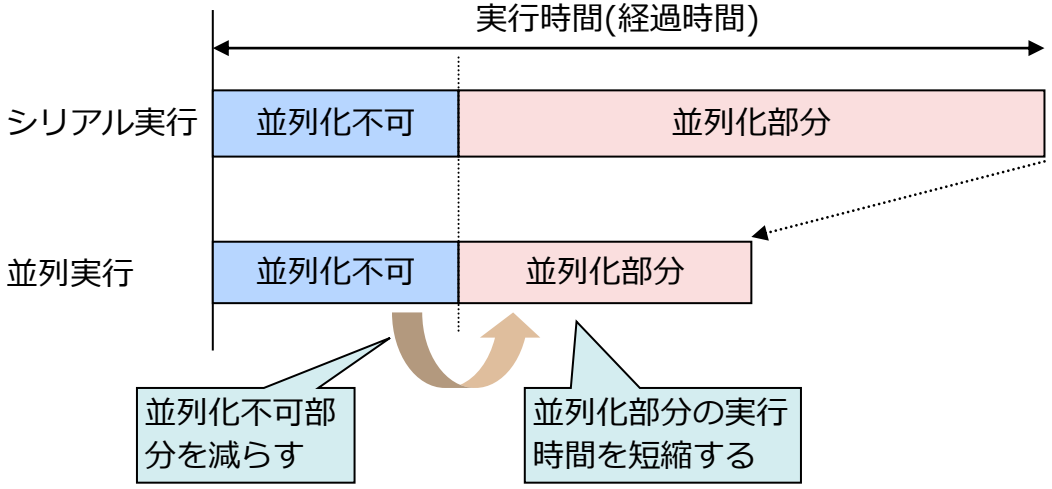
チューニングの観点

並列実行される部分は多いか?

- 並列化しないで実行した場合の経過時間に対する、並列実行可能部分の実行時間の割合が小さくないか? (並列実行部分/並列ループを増やす)

効果的な並列化が行われているか?

- 並列化されているループの実行時間が長いか? (適切なループを並列化する)
- 並列化のためのオーバヘッドが大きくないか? (オーバヘッドを小さくする)
- スレッドごとの作業量が均一か? (ループ内の処理を見直す)



チューニング手順

使用する性能解析機能
- 簡易性能解析(**FTRACE**)

1. 並列対象ループ/関数の抽出
- プロファイラ情報、FTRACEの出力から、実行時間の長い関数を見つけ出す

2. 並列化部分を増やす
- **1.**で見つけた関数中の並列化されていないループが並列化できないか調べ、必要なら指示行の指定、プログラムの変形を行い並列化する

3. ロードバランスの改善
- PROGINF情報、FTRACEの出力から、各タスクに処理が均等に割り当てられるよう、ロードバランスを調整する

※ 並列化の前にベクトル化のチューニングを十分行っておくこと

並列化するループの選択

自動並列化では該当するループが
自動的に探し出され、並列化される

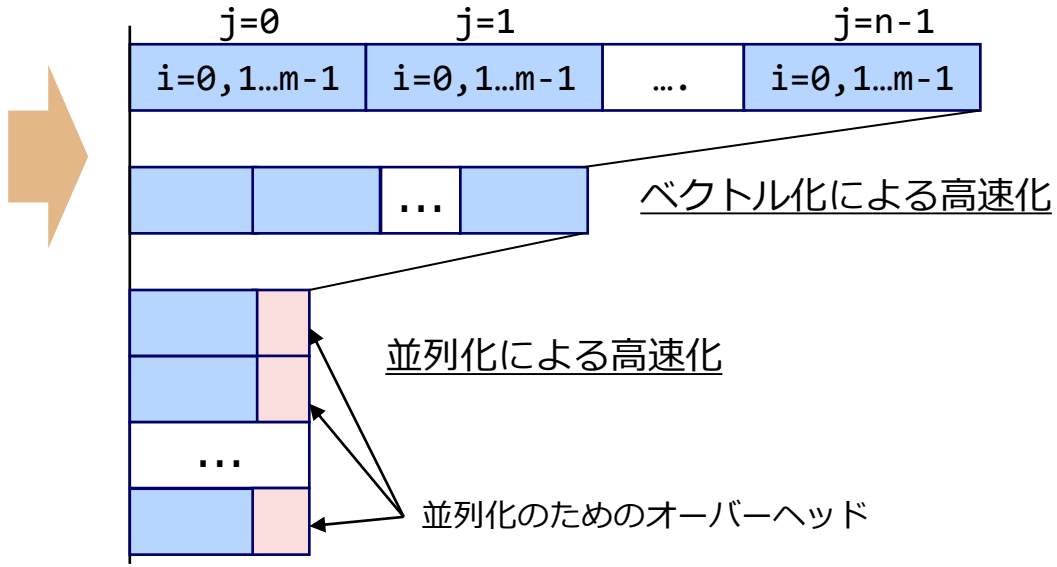
並列化阻害要因のないループ

- 並列化できない依存関係
- 並列化できない制御構造
- I/O関数など実行順序を保証しなければならない関数呼び出し

多重ループの最外側ループ

- 処理時間の長いループ
- 最内側ループはベクトル化での高速化を検討

```
for (j = 0; j < n; j++) {
   for (i = 0; i < m; i++) {
      a[j][i] = b [j][i] + c [j][i];
   }
}
```



並列化できない依存関係

同一配列要素が異なる繰り返しで定義・参照されているループ

同一配列要素の定義・参照

```
for (i=0; i<n; i++) {
   a[i] = b[i+1];
   b[i] = c[i];
}
```

ループの繰り返し	参照	定義	
i=0	b[1]	b[0]	スレッド1で実行
i=1	b[2]	b[1]	
i=2	b[3]	b[2]	スレッド2で実行
i=3	b[4]	b[3]	
⋮	⋮	⋮	

b[2]の参照と定義はどちらが先になるか保証されない

同一スカラ変数が異なる繰り返しで定義・参照されているループ

同一スカラ変数

```
for (i=0; i<n; i++) {
   c[i] = t;
   t = b[i];
}
```

- 定義、参照の順であれば並列化可能
- 総和・累積のパターンは、プログラムの変形、指示行の指定などにより並列化可能。(自動並列利用時は、コンパイラが自動的に認識し並列化する)

if条件下で定義された変数がif条件下以外で参照

```
for (j=0; j<n; j++) {
   for (i=0; i<m; i++) {
      if (a[j][i] >= d ) {
         T = a[j][i] - d;
      }
      c[j][i] = T;
   }
}
```

- 変数Tはif文の条件が成立した繰り返しで定義された値を参照
- この場合は定義・参照の順でも並列化不可

並列化できない制御構造

ループからの飛び出し

- 飛び出す条件が成立した繰り返しより後の繰り返しを実行してはならないため、並列化できない

```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
          if (a[i][j] < 0.0 ) goto label100 ;
       b[i][j] = sqrt(a[j][i] );
    }
}
Label100: ;
```

指示行による並列化促進

```
$ ncc -mparallel -fdiag-parallel=2 a.c -c
ncc: opt(1380): a.c, line 6: User function references not ok without "cncall".: func1
```

- ループ中に関数呼び出しが含まれるとき、その関数が並列実行可能かどうか不明であるため、ループは自動並列化の対象とならない
- 関数が並列実行可能であれば、**cncall**指示行を指定し、ループを自動並列化の対象とする

```
void func()
{
    for (int i = 0; i < M; i++) {
        c[i] = func1(a[i], b[i]);
    }
}
```

```
void func()
{
#pragma _NEC cncall
    for (int i = 0; i < M; i++) {
        c[i] = func1(a[i], b[i]);
    }
}
```

```
$ ncc -mparallel -fdiag-parallel=2 a.c -c
ncc: par(1801): a.c, line 7: Parallel routine generated.: func$1
ncc: par(1803): a.c, line 7: Parallelized by "for".
ncc: vec( 103): a.c, line 7: Unvectorized loop.
```

強制並列化指示行

- 自動並列化機能で並列化されなかった
- プログラマはループが並列化可能なことを知っている

- 強制並列化指示行**parallel for**を指定して並列化
  - ループ、文の集まりを並列化指定できる
  - コンパイラは、データの依存関係を無視して並列化する。

正しい結果が得られることはプログラマが保証しなければならない

```
void func()
{
    double wk[M];
#pragma _NEC parallel for private(wk)
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++)
            wk[i] = a[i] + b[j];
        func1(x[j], wk);
#pragma _NEC atomic
        sum += x[j];
    }
}
```

ループの強制並列化を指定
ループ内で作業用として使用する変数、配列は**private**句で指定する

強制並列化されたループが、総和、累積などのマクロ演算の文を含むとき、**atomic**をその文の直前に指定する

並列化のためのオーバーヘッド

プログラムを並列化したことにより増加する実行時間のこと

- 並列化するためにプログラマによって追加された処理の実行時間
  - プログラム変形による増加時間
  - 並列処理制御のための実行時ライブラリの処理時間
- システムライブラリ内の排他制御による待ち時間
  - システムデータを更新、参照するシステムライブラリ関数内での排他制御による待ち時間
    - –ファイルI/O関数、**malloc()**など
  - C++の**new**演算子では**malloc()**を使用するため、**new**演算子を多用しているC++プログラムでは注意が必要
- 他のスレッドとのバリア同期のための待ち時間

システムライブラリ内の排他制御

- プログラム全体で利用されるデータを参照、更新するとき、別のOpenMPスレッドで同時に更新されないよう排他制御を行う
  - ファイルディスクリプタ、**malloc()**で確保した領域の管理データなど

- システムライブラリ関数の呼び出し回数を削減
  - **malloc()**はできるだけ一つにまとめる
  - 関数内でのみ使用するデータは**new**演算子で確保せず、ローカルデータとして宣言し、スタックに領域を確保する
  - 使用メモリに余裕があるとき、ファイルからの読み込みは一度に行い、内容をメモリに展開し、必要なデータをメモリから読み込むようにする
- 1バイト入出力関数については、並列区間外では**xxx_unlocked()**系関数を使用する
  - `getc(3S)` → `getc_unlocked(3S)`
  - `getchar(3S)` → `getchar_unlocked(3S)`
  - `putc(3S)` → `putc_unlocked(3S)`
  - `putchar(3S)` → `putchar_unlocked(3S)`

バリア同期のための待ち時間の削減 (1)

OpenMPでは、次の場所で自動的にバリア同期が行われる

- **nowait**句のない並列ループの終了時
- **reduction**句の指定された並列ループの終了時(*)
- **copyin**句の指定された並列リージョンの開始時(*)
- 並列リージョンの終了時(*)

自動並列化では、コンパイラが適切に暗黙的バリア同期を行う

(*)は並列処理の仕組み上、バリア同期を省略することができない

スレッドごとの仕事量を均一にする(待ち時間の短縮)

- 繰り返しごとの作業量が変化する並列ループの作業量の均一化には、**schedule(dynamic)**が有効

```
#pragma omp for schedule(static )
for (j=m; j>0; j--) {
  for (i=0; i<j; i++) {
    ...
  }
}
```

```
#pragma omp for schedule(dynamic )
for (j=m; j>0; j--) {
  for (i=0; i<j; i++) {
    ...
  }
}
```

バリア同期のための待ち時間の削減 (2)

- 並列リージョンの連結による暗黙的バリア同期の削除
- **nowait**句指定による不要な暗黙的バリア同期の削除
  - 削除できないバリア同期に**nowait**句が指定された場合、コンパイラは**nowait**句の指定を無視する

```
double a[N], b[N*2], x;
void func()
{
#pragma omp parallel for
    for (i = 0; i < N; i++) {
          a[i] = …
    }
    x = 0.0;
#pragma omp parallel for
    for (i = 0; i < N*2; i++) {
          b[i] = …
    }
    …
}
```

並列リージョンの連結

```
double a[N], b[N*2], x;
void func()
{
   x = 0.0;
#pragma omp parallel
  {
#pragma omp for
    for (i = 0; i < N; i++) {
          a[i] = …
    }
#pragma omp for
    for (i = 0; i < N*2; i++) {
          b[i] = …
    }
  }
    …
}
```

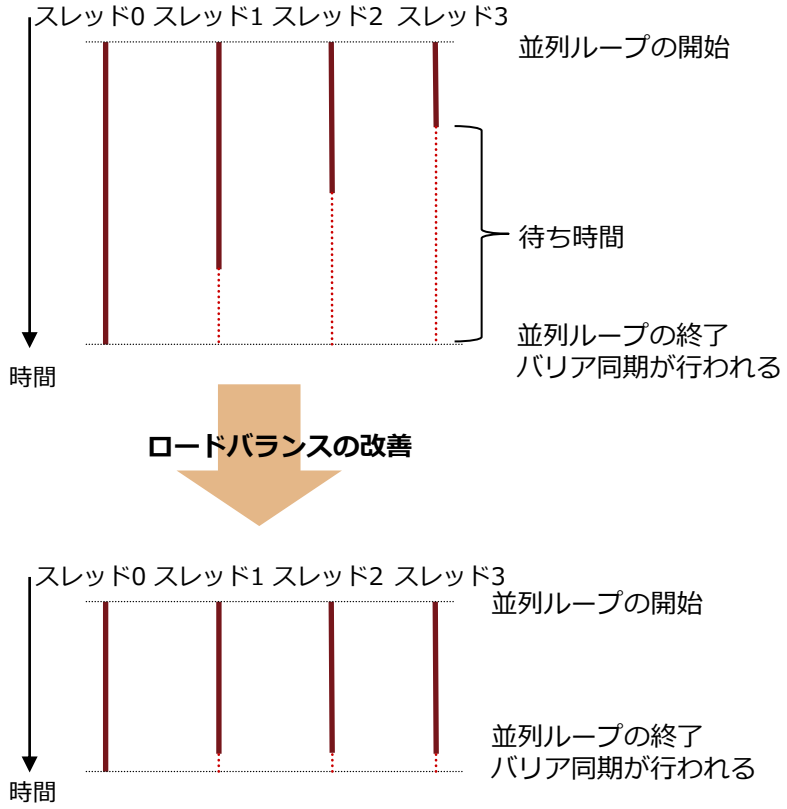
nowait句を指定

```
double a[N], b[N*2], x;
void func()
{
   x = 0.0;
#pragma omp parallel
  {
#pragma omp for nowait
    for (i = 0; i < N; i++) {
          a[i] = …
    }
#pragma omp for nowait
    for (i = 0; i < N*2; i++) {
          b[i] = …
    }
  }
    …
}
```

ロードバランスの改善 (1)

- 次のようなループでは、タスクごとの作業量が均一でなく、ループの終了時点で多くの待ち時間が生じる

並列ループを四つに分割し、四つのスレッドで実行したとき




```
#pragma omp for
    for (j = 1024; j > 0; j--) {
        for (i = 0; i < j; i++) {
            …
        }
    }
```

並列化されたループの繰り返しが進むにつれて内側ループの繰り返し数、すなわち、計算量が減少する

スレッドごとの作業量をできるだけ均一にし、待ち時間を少なくすると、より短時間ですべての計算を終えることができる

ロードバランスの改善 (2)

作業量を均一にするため、並列ループをより細かく分割し、スレッドに割り当てる

OpenMP並列化

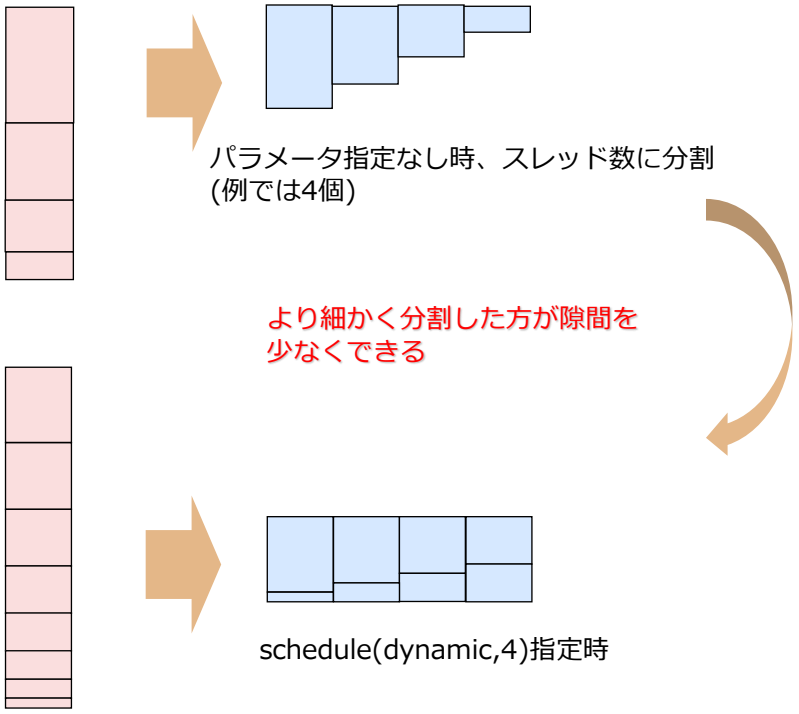
- **schedule**句のパラメータで調整

```
#pragma omp for schedule(dynamic,4)
    for (j = 1024; j > 0; j--) {
        for (i = 0; i < j; i++) {
            …
        }
    }
```

自動並列化

- **concurrent**指示行にOpenMPと同様に **schedule**句を付けてパラメータで調整

```
#pragma _NEC concurrent schedule(dynamic,4)
    for (j = 1024; j > 0; j--) {
        for (i = 0; i < j; i++) {
            …
        }
    }
```



分割数が多いほどスレッド制御のための時間が必要になるので、可能な限り少ない分割数にすること

簡易性能解析機能:FTRACE

スレッドごとの情報より、関数内でのロードバランスを知ることができる

```
REQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
          TIME[sec](  % )    [msec]                     RATIO V.LEN      TIME    MISS     CONF HIT E.%

   60000    62.177( 73.1)      1.036 100641.4  79931.0  99.55 248.5    62.134   0.023    0.000  100.00 funcX$1
   15000     4.467(  5.3)      0.298 107076.2  83033.3  99.47 248.4     4.455   0.005    0.000  100.00  -thread0
   15000    11.552( 13.6)      0.770 104082.7  82404.6  99.54 248.5    11.542   0.006    0.000  100.00  -thread1
   15000    19.000( 22.3)      1.267 101390.4  80683.3  99.55 248.6    18.990   0.006    0.000  100.00  -thread2
   15000    27.157( 31.9)      1.810  97595.1  77842.2  99.56 248.6    27.147   0.006    0.000  100.00  -thread3
   15000    22.711( 26.7)      1.514   1426.9      0.0   0.00   0.0     0.000   0.015    0.000    0.00 funcX
...
-------------------------------------------------------------------------------------------------------------
   79001    85.034(100.0)      1.076  74062.7  58500.4  98.89 248.5    62.249   0.043    0.000  100.00 total
```

ループ直前に#pragma _NEC concurrent schedule(dynamic, 4)を指定

```
REQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT VLD LLC PROC.NAME
          TIME[sec](  % )    [msec]                     RATIO V.LEN      TIME    MISS     CONF HIT E.%

   60000    66.872( 99.6)      1.115  93599.2  74318.7  99.52 248.5    64.077   1.418    0.000  100.00 funcX$1
   15000    16.766( 25.0)      1.118  92992.0  73842.7  99.52 248.5    16.022   0.409    0.000  100.00  -thread0
   15000    16.697( 24.9)      1.113  91671.0  72790.7  99.52 248.5    16.000   0.397    0.000  100.00  -thread1
   15000    16.714( 24.9)      1.114  94854.7  75312.8  99.52 248.5    16.040   0.305    0.000  100.00  -thread2
   15000    16.695( 24.9)      1.113  94880.7  75329.6  99.51 248.5    16.014   0.307    0.000  100.00  -thread3
   15000     0.129(  0.2)      0.009   1284.5      0.1   0.00   0.0     0.000   0.010    0.000    0.00 funcX
...
-------------------------------------------------------------------------------------------------------------
   79001    67.148(100.0)      0.850  93334.5  74082.8  99.51 248.5    64.192   1.430    0.000  100.00 total
```

改善前：**funcX$1**の**-thread0**～**-thread3**のEXCLUSIVE TIMEにばらつきがある(ロードインバランス)
改善後：ばらつきがなくなり、**funcX**のEXCLUSIVE TIMEが短縮(バリア同期時間等が短縮)
　　ただしスレッド制御時間が増すため**funcX$1**は増加

並列化における注意事項

malloc(3C)・new演算子で確保した領域

malloc(3C)・new演算子で確保した領域が共有か、プライベートかは以下で決定

- 領域へのポインタが共有か、プライベートか?
- 領域確保時、並列処理中だったか?

p、q、r : shared
s: private

並列処理区間

```
void func() {
    double *p = malloc(16);
    double *q;
    double *r;
#pragma omp parallel num_threads(4)
    {
        double *s = malloc(16);
#pragma omp critical
        q = malloc(16);
#pragma omp master
        r = malloc(16);
    }
}
```

p = malloc(16)は1回実行。全スレッドで同一の領域を参照

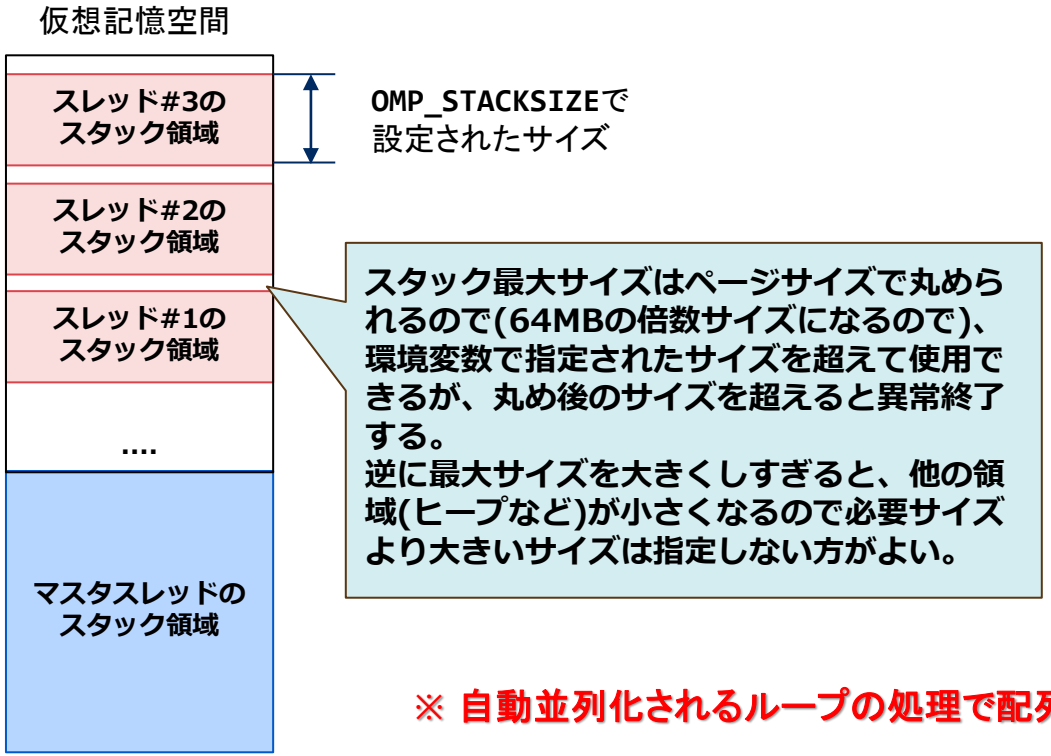
q = malloc(16)は全スレッドで実行され、領域は四つ確保される。ただし、使用される領域はどれかひとつで、全スレッドで同一の領域を参照する。残りの三つの領域は無駄になる。

r = malloc(16)はマスタスレッドでのみ実行され、領域は一つだけ確保される。全スレッドで同一の領域を参照する。

s = malloc(16)は全スレッドで実行され、領域は四つ確保される。各スレッドで別々の領域が使用される。

巨大なローカル配列

- Parallelリージョンなど並列実行される部分において、巨大なローカル配列を使用する場合、環境変数**OMP_STACKSIZE**にその配列より大きいサイズの値を設定する。
  - **OMP_STACKSIZE**は、マスタスレッド以外のスレッドのスタックの最大サイズを設定する環境変数。設定しなかった場合のスタックの最大サイズは4メガバイト。
  - スタックの未使用領域に配列が入り切らなかったとき、プログラムが異常終了する。



スタック最大サイズはページサイズで丸められるので(64MBの倍数サイズになるので)、環境変数で指定されたサイズを超えて使用できるが、丸め後のサイズを超えると異常終了する。
逆に最大サイズを大きくしすぎると、他の領域(ヒープなど)が小さくなるので必要サイズより大きいサイズは指定しない方がよい。

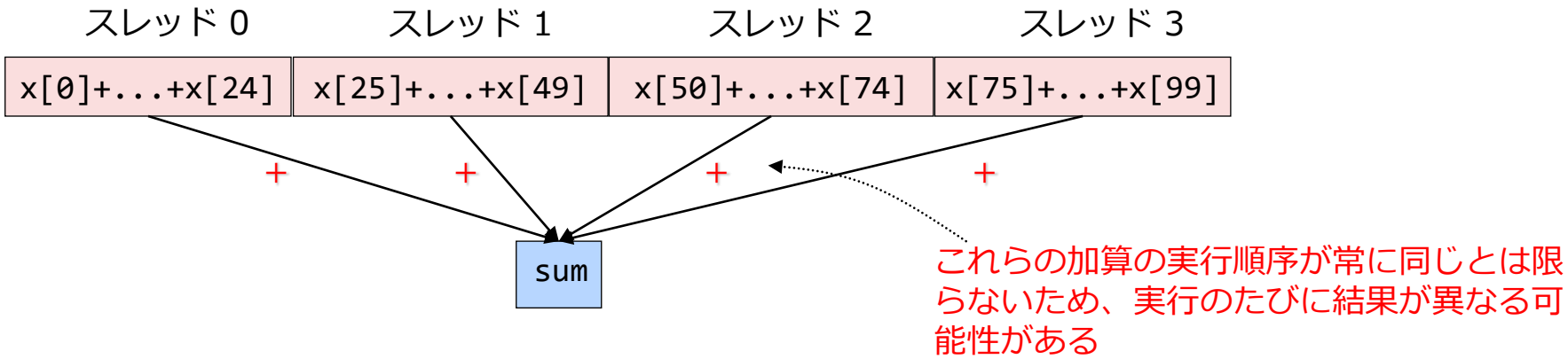
```
$ cat a.c
...
#pragma omp parallel
  {
    double x[16*1024*1024];
    double y[16*1024*1024];
    func(x,y);
  }
...
$ ncc -fopenmp a.c
$ export OMP_STACKSIZE=384M
$ ./a.out
```

※ 自動並列化されるループの処理で配列を宣言している場合も同様の対処が必要。

総和演算

- 総和演算は並列化可能であるが、各スレッドの実行順序が一定でない（実行順序が保証されない）ため、足し込みの順序が実行するたびに変わってしまう可能性がある
  - 演算誤差範囲で、シリアル実行時とは計算結果が異なる、また、並列実行するたびに結果が変わることがある

```
for (i = 0; i < 100; i++) {
    sum = sum + x[i];
}
```




Orchestrating a brighter world
NEC