H26年度
スーパーコンピュータの高速化技法入門
並列化による高速化技法

2015年 1月21日

大阪大学サイバーメディアセンター
日本電気株式会社

本資料は，東北大学サイバーサイエンスセンターとＮＥＣの共同により作成され，大阪大学サイバーメディアセンターの環境で実行確認を行い，修正を加えたものです．

無断転載等は，ご遠慮下さい．

目次

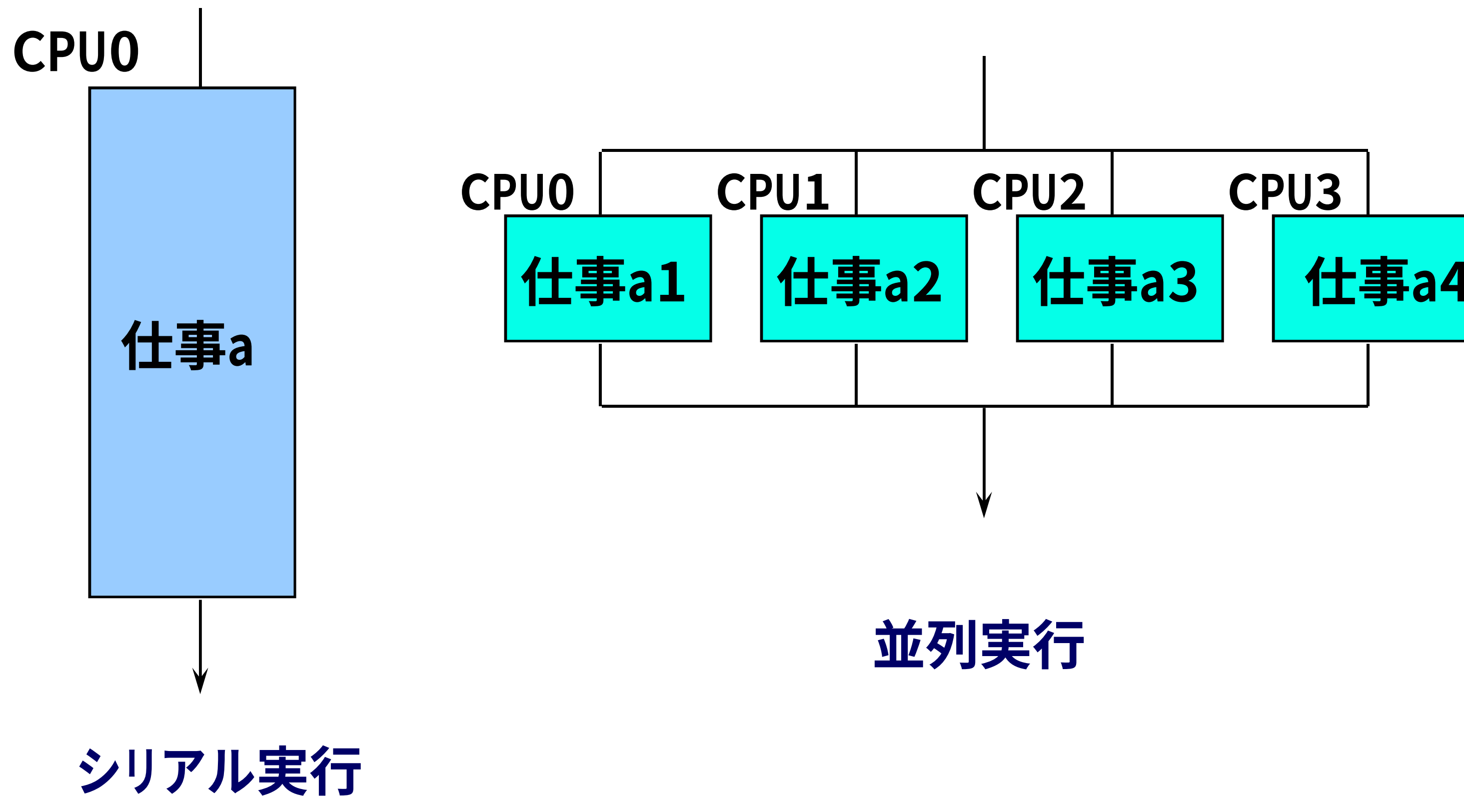
- 並列処理とは
- 並列化における注意事項
- 並列化のチューニング
- OpenMP (参考資料)

目次

- 並列処理とは
- 並列化における注意事項
- 並列化のチューニング
- OpenMP (参考資料)

並列処理とは

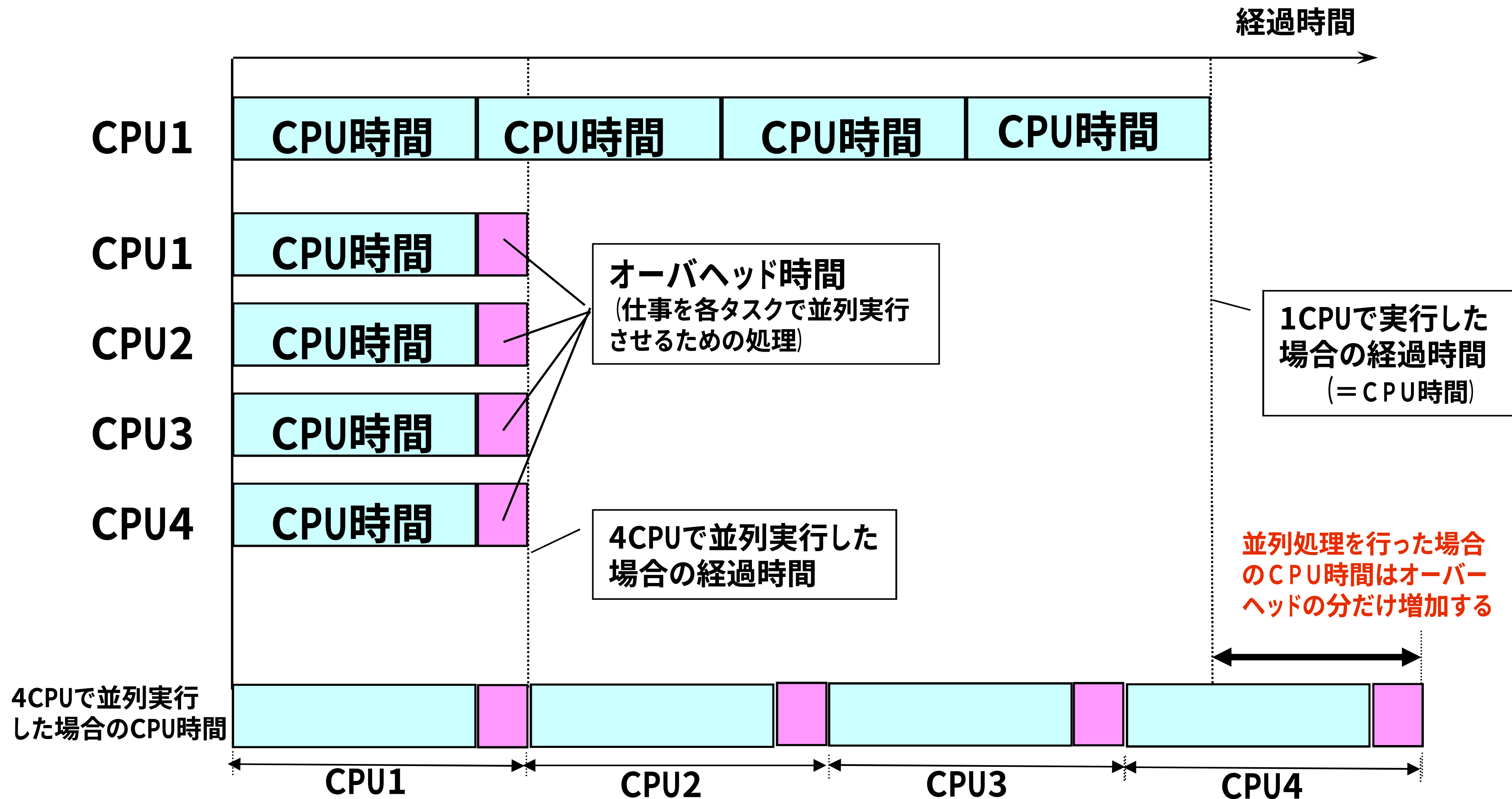
- 1つの仕事を、幾つかの小さな仕事に分割し、複数のタスク (CPU) で実行すること



並列処理による実行時間の短縮

並列処理を行った場合の実行時間の短縮

- CPU時間ではなく、経過時間が短縮される

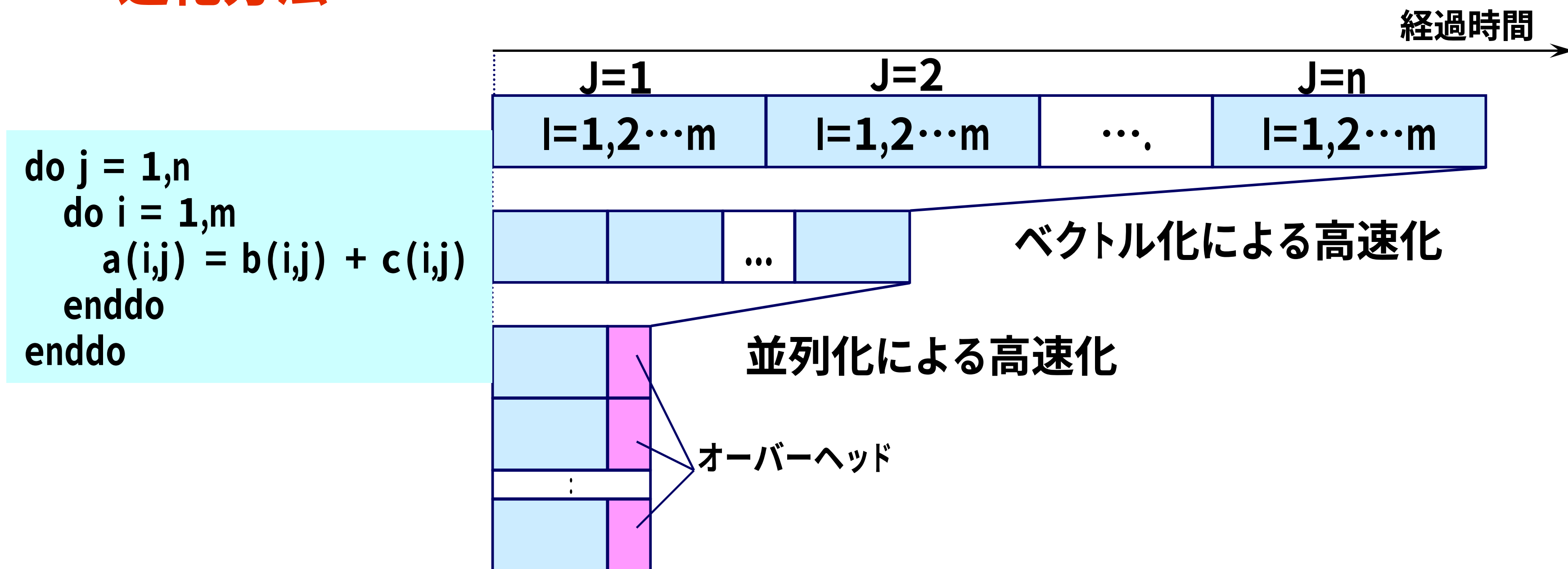


ベクトル化＋並列化による性能向上

- 各タスクで処理する作業量を大きくすれば並列オーバヘッドの割合は小さくなる
- ベクトル化は最内側ループを高速化する

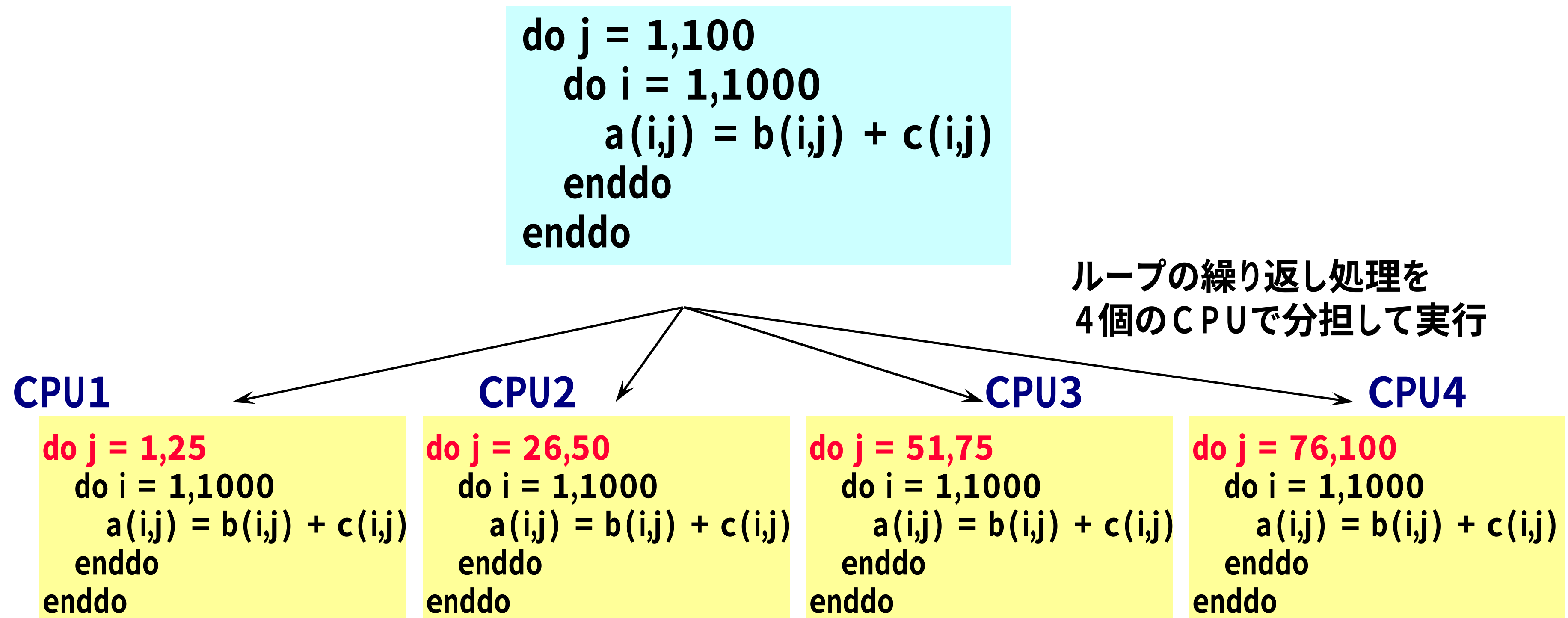
 ➔CPU時間が短縮され、経過時間も同時に短縮される

 ➔内側ループをベクトル化し、外側ループを並列化することが最善の高速化方法



自動並列化とは

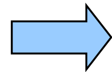
- コンパイラが、並列実行可能なループや文の集まりを抽出し、ループの繰り返しなどを複数のタスクに分割し、複数のCPU上で実行する機能



自動並列化の使用方法

オプション「-P auto」を指定してコンパイル

- プログラムの一部だけを自動並列化する場合
  - 並列実行プログラムは、シリアル実行プログラムとデータの割り付け方や、リンクされる実行時ライブラリなどが異なる



自動並列化を行いたいソースは「-P auto」
それ以外のソースは「-P multi」
を指定してコンパイルする

- 自動並列化用のコンパイラ指示行
  - 並列化指示行
    - CONCUR、NOSYNC、INNER、CNCALL など
  - 強制並列化指示行
    - PARALLEL DO、PARALLEL SECTION など

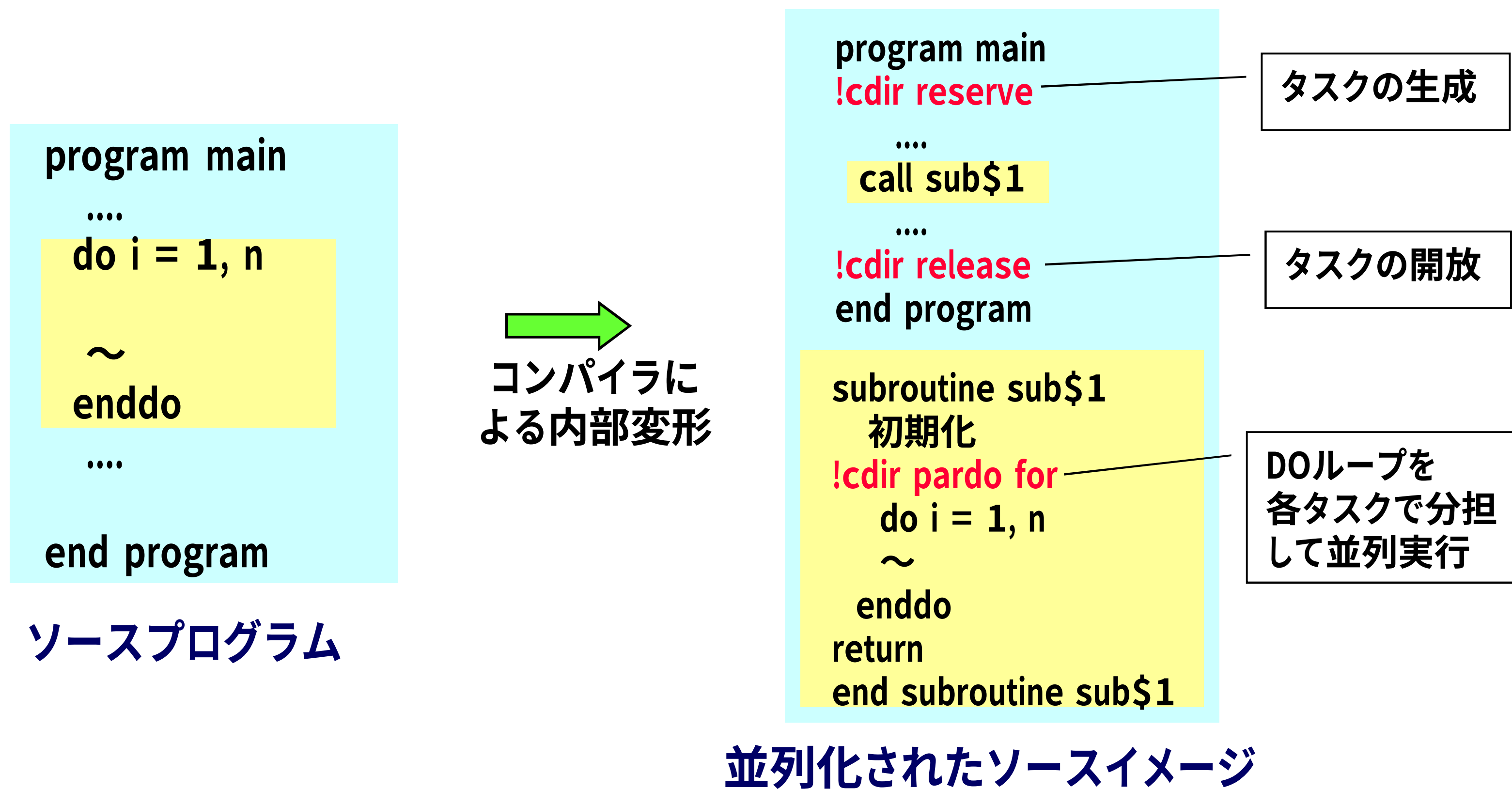
自動並列化方法（基本方針）

- 多重ループの場合、基本的に、内側ループをベクトル化、外側ループを並列化する

```
subroutine sub(a,h)
real a(600,100,100)
integer h(100)
do i = 1,100
    do j = 1,100
        do k=1,599
            a (k,j,i) =(a(k+1,j,i) + a(k,j,i))*0.5
        enddo
    enddo
    h(i)=h(i)+1
enddo
end
```

自動並列化の処理方式

- コンパイラがプログラムを解析し、並列化可能なループと判断した場合に、サブルーチンとして切り出し、並列制御コードを埋め込むことにより並列化を行う



並列処理時のデータの種類

タスク間共有データ

- 各タスクで同一の領域をアクセスするデータ
  - 以下のものは常にタスク間共有データとなる
    - COMMONで宣言されたデータ
    - SAVE文で宣言されたデータ
    - 初期値ありデータ
  - 上記以外のものは、通常はタスク固有データとなる．ただし、自動並列化で必要と判断された場合には、コンパイラがタスク間共有データにする

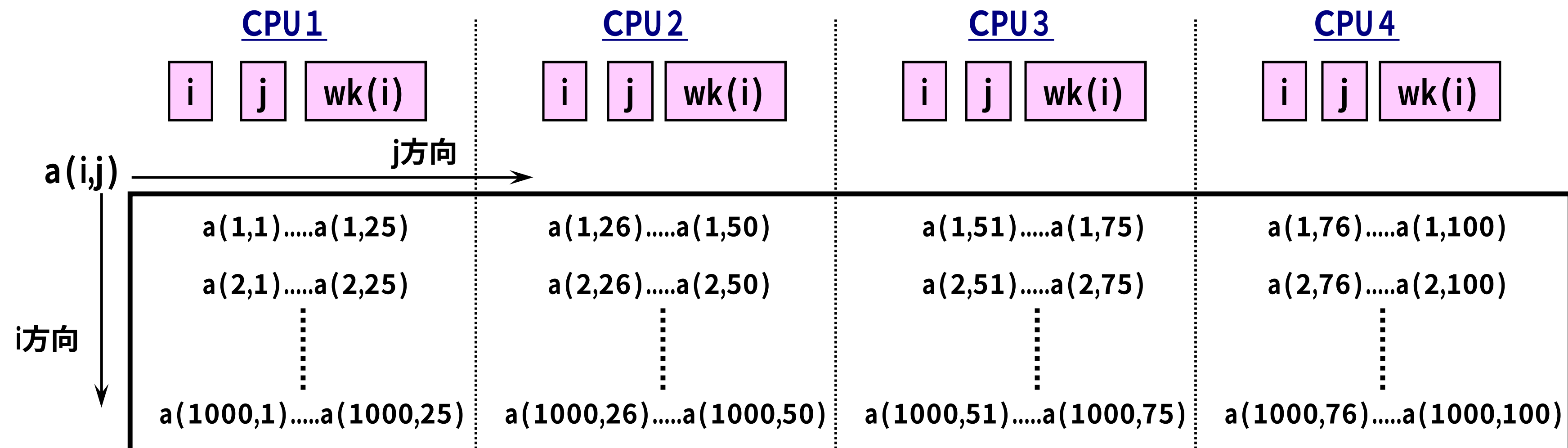
タスク固有データ

- 各タスク毎に別の領域をアクセスするデータ
  - タスク固有データは、各タスク毎に別々の領域がとられる．従って、シリアル時のタスク数倍の領域が必要となる

タスク間共有データとタスク固有データ

```
do j = 1,100
   do i = 1,1000
      wk(i) = b(i,j) +2.0
      a(i,j) = wk(i) *  c(i,j)
   enddo
enddo
```

配列 a、b、c は、タスク間共有データとなる．
各タスクは配列 a の自分の分担部分の要素に値を代入する
DO変数の i や j 、ループ内で中間結果の保存に使用されている配列wk は、コンパイラがタスク固有データにする



自動並列化の阻害要因

ループの各繰り返しの実行順序に制約が付くと並列化はできない

- 並列化を阻害する依存関係
  - ループの異なる繰り返しで定義された変数・配列要素を定義・参照している場合、並列化不可
- 並列化を阻害する制御構造
  - ループ外への条件分岐があると並列化不可
- 並列化を阻害する文

  以下のものはシリアル実行時の実行順序を保たなければ実行結果が変わってしまうため並列化不可
  - 入出力文
  - 乱数生成関数呼び出し

並列化不可の依存関係（1）

ベクトル化可能だが、並列化は不可能なループ

```
do i = 1,n
   a(i) = b(i+1)
   b(i) = c(i)
enddo
```

配列bの要素が異なる繰り返しで
定義・参照されている

ベクトル化の場合

ループの実行順序は、保証される．
bの参照と定義の順番は、保証される

ループの繰り返し	参照	定義
I=1	b(2)	b(1)
I=2	b(3)	b(2)
I=3	b(4)	b(3)
⋮	⋮	⋮

並列化の場合

ループの実行順序は、保証されない．
bの参照と定義の順番は、タイミングによって異なる

ループの繰り返し	参照	定義	
I=1	b(2)	b(1)	CPU1で実行
I=2	b(3)	b(2)	
I=3	b(4)	b(3)	CPU2で実行
I=4	b(5)	b(4)	
⋮	⋮	⋮	

並列化不可の依存関係（2）

変数が定義前に引用されている場合

```
do i = 1,n
    c(i) = t
    t = b(i)
enddo
```

変数tは1回前の繰り返しで定義された値を参照する

※ 総和／内積はこのパターンに該当するが、
コンパイラが認識して特別な方法で並列化する

```
do I=1,10000
   s = s + a(I)*b(I)
enddo
```

特別な方法で並列化

並列化不可の依存関係（3）

IF文下で定義された変数がIF条件外で引用されている場合

```
do j = 1, n
   do i = 1, m
      if (a(i,j) .ge. del ) then
         t = a(i,j) - del
      endif
      c(i,j) = t
   enddo
enddo
```

変数 t はIF条件が成立した繰り返しで定義された値を引用している

ループからの飛出し

- ループから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

```
      do j = 1,n
         do i = 1,n
            if (a(i,j) .lt. 0.0 ) go to 100
          b(i,j) = sqrt(a(i,j) )
         enddo
      enddo
100 continue
```

目次

- 並列処理とは
- 並列化における注意事項
- 並列化のチューニング
- OpenMP (参考資料)

並列化における注意事項（1）

ローカルデータの初期化

- **ローカルデータの初期値は不定**
  - シリアル実行時に初期値ゼロを期待して動作していたプログラムは、並列実行時には、正しく動作しない可能性がある

  ➡ *ローカルデータは、必ず初期化しなければならない*

- **シリアル実行でのデバッグ方法**
  - オプション「-P stack」を指定すると、ローカル変数をスタックに割り当てたシリアル実行プログラムを作成することができる
- **ローカルデータの初期値設定**
  - 詳細オプション「-Wf,-init stack=zero」を指定すると、実行に使用するスタック領域をゼロで初期化することができる。ただし、実行性能が若干低下するため、デバッグのためにだけ使用することが望ましい

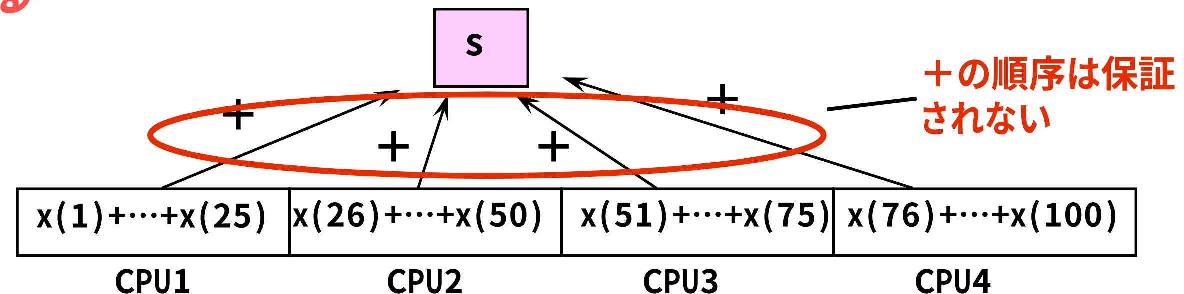
並列化における注意事項（2）

総和演算

- 総和演算は、並列化可能であるが、各タスクの実行順序が一定ではない（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

➡ *シリアル実行時とは、計算結果が異なる（演算誤差が生じる）場合がある．また、同じ並列実行ﾌﾟﾛｸﾞﾗﾑでも、流すたびに結果が変わる可能性がある*

```
do i = 1,100
   s = s + x(i)
enddo
```



乱数組込み関数

- 乱数組込み関数は、並列実行すると、各タスクの実行順序が保証されないため、実行するたびに結果が変わる可能性がある

➡ *乱数組込み関数が使用されている部分の自動並列化は抑止される*

並列化における注意事項(3)

手続のreturn文実行後、ローカルデータの値は保存されない

- retrurn文実行時にローカル変数のある領域は開放される

  ➡ *return文実行後値が保存されていることを期待しているプログラムは正しく動作しない*

  - オプション「-P stack」を指定することによって、シリアル実行によるデバッグが可能

初期値を与えたローカル変数、save文の指定されたローカル変数

- data文などによって初期値を与えたローカル変数やsave文を指定したローカル変数は、並列処理時には、スタックではなく、静的領域に割り当てられるため、各タスクで同一の領域を参照するようになる

  ➡ *各タスクから、非同期に値の更新が行われる可能性があるため自動並列化を阻害する原因となることがある*

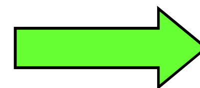
並列化における注意事項（4）

巨大な配列をローカルデータとして宣言すべきではない

- ローカル配列は、タスク固有データであり、各タスク毎に別々に確保されるため、ローカル配列のサイズをタスク数倍した大きさのメモリが必要となる

⇒ *巨大な配列は、できる限り共通データとして宣言するか、単純変数となるように、プログラムの構造を考えることが望ましい*

```
Program  main
real*8  x(10000,100,100)
real*8  y(10000,100,100)
call sub(x,y)
end
```



x, y を common にする

```
Program  main
real*8  x(10000,100,100)
real*8  y(10000,100,100)
common /dummy/ x, y
call sub(x,y)
end
```

並列プログラムのメモリサイズ

sizeコマンド

- 指定したタスク数で実行するのに必要なメモリサイズを表示

形式：

size -fl タスク数 実行ファイル名

例:

% size -fl 4 a.out

4094176(.text) + 668536(.data) + 2157776(.bss) + 292425(.comment) + 5932(.whoami) + *206125400(logical task region)* * 4 = 831720445

タスク数に依存する部分

目次

- 並列処理とは
- 並列化における注意事項
- 並列化のチューニング
- OpenMP (参考資料)

並列化のチューニング

- 並列化における高速化の観点
- 並列化のチューニング手順
- チューニング

並列化における高速化の観点

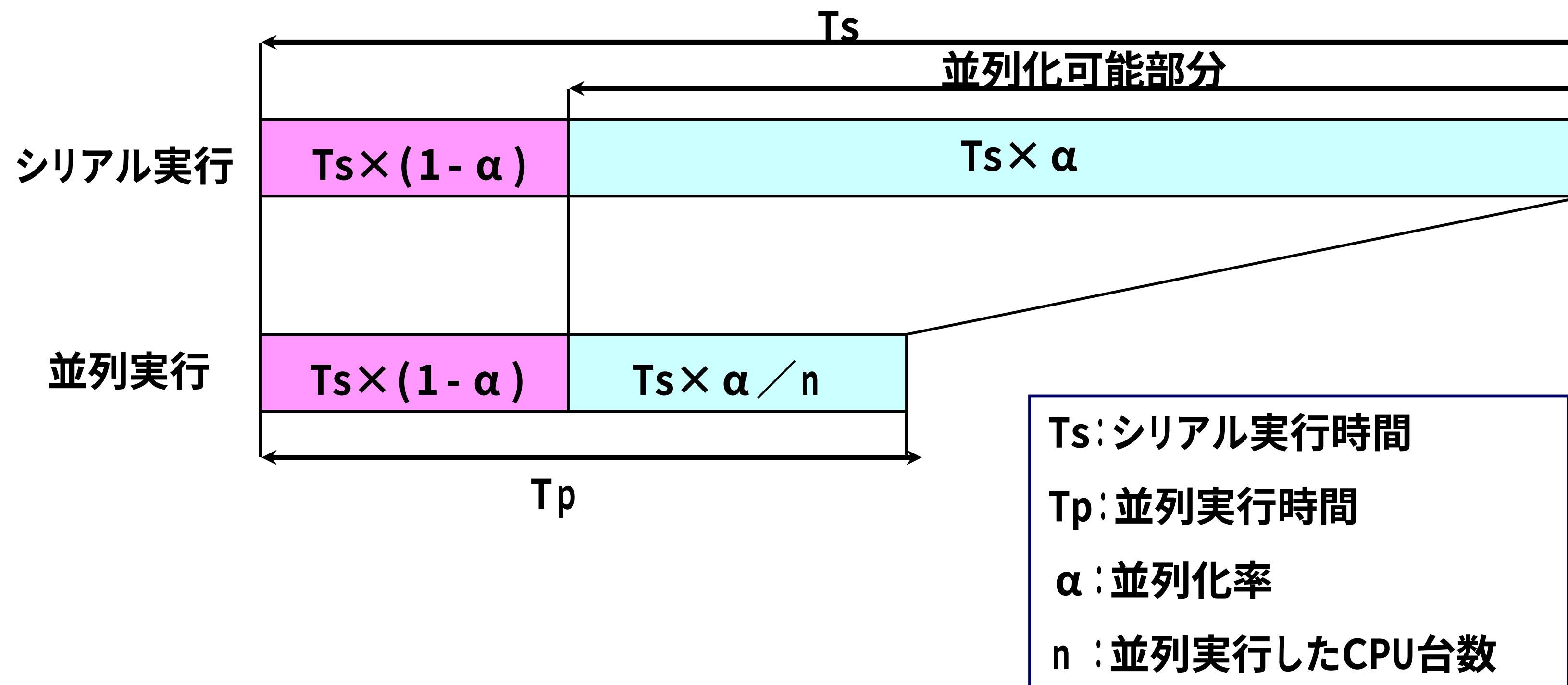
並列化率

並列化効率

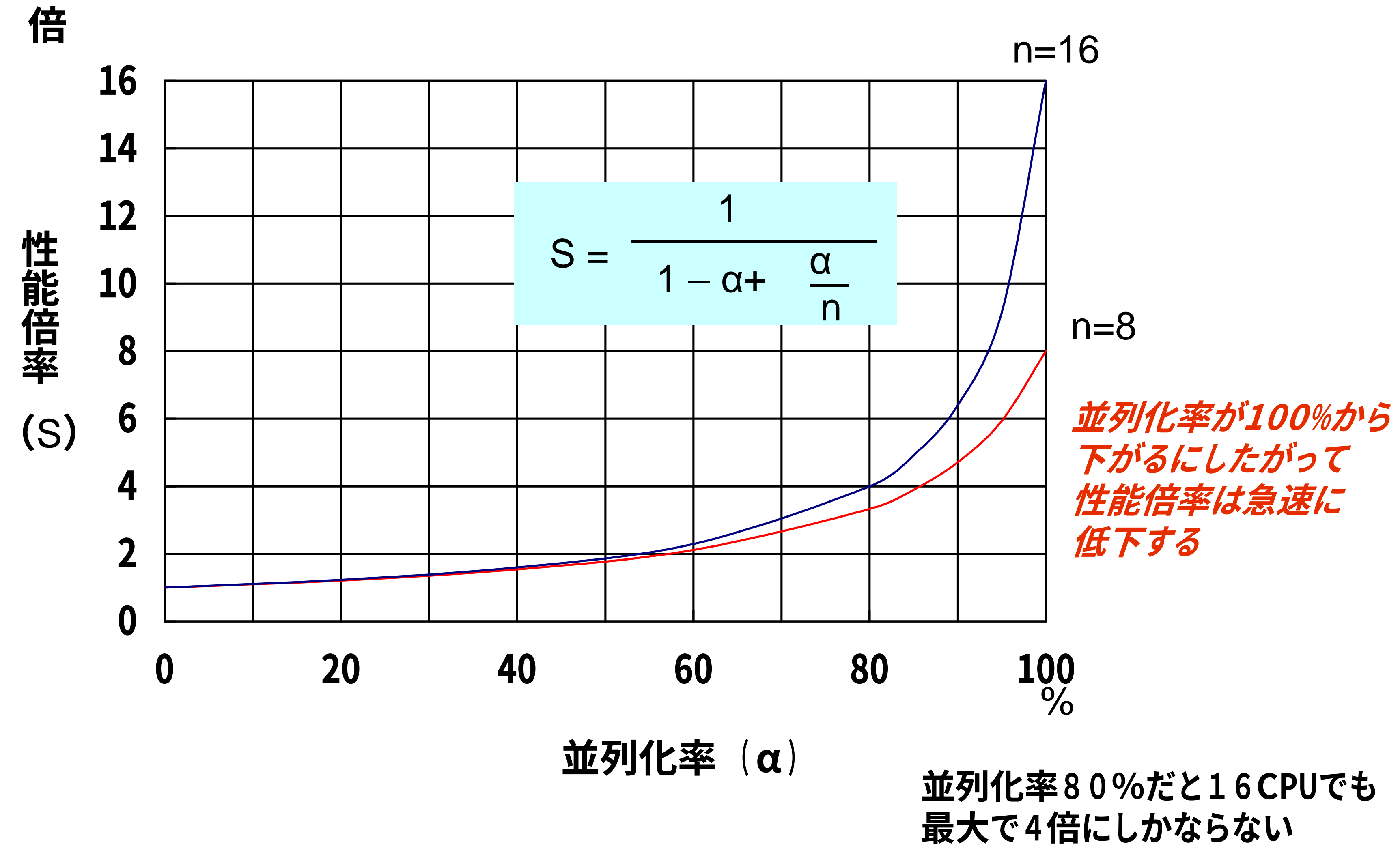
- 並列処理のオーバーヘッド
- 各タスクの負荷バランス

並列化率

- シリアル実行した場合の実行時間に対する、並列実行可能部分の実行時間の割合



アムダールの法則



並列化率８０%だと１６CPUでも
最大で４倍にしかならない

並列化効率

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は十分大きいか
  ➔簡易性能解析機能（ftrace)、プロファイラ

- 並列処理のオーバヘッドが大きくないか
  ➔プロファイラ

- 各タスクの負荷バランスは均一か
  ➔PROGINFの Conc.Time、
  簡易性能解析機能（ftrace）、
  プロファイラ

並列化のチューニング手順

0. ベクトル性能チューニング
   - 並列化の前に、ベクトル化のチューニングを完了させておく
1. 性能分析
   - ftrace情報から、コストの大きいループを含むサブルーチンを見つけ出す
2. 並列化
   - コストの大きなサブルーチンから（自動）並列化
3. チューニング
   - 並列化率の向上
     - 指示行、ソースコードの修正
   - 負荷バランスの改善
     - 各タスクに処理が均等に割り当てられるよう、バランスを調整する

並列化率向上のための技法

並列化阻害要因の除去

- 診断メッセージから並列化阻害要因を知る

ｵﾌﾟｼｮﾝ –Wf,-pvctl fullmsg

依存関係が不明で並列化しない場合のメッセージ

ﾒｯｾｰｼﾞ No.	メッセージ
1033	同一の配列要素に対して定義が複数回行われる可能性がある
1036	異なる繰り返しで定義された値を参照している可能性がある
	(nodep / nosyncを指定すれば最適化を行う)

➔依存関係が並列化可能かどうかコンパイラが判定できない

- 依存関係がない ➔nosync 指示行を指定
- 並列化不可の依存関係がある ➔プログラム修正

指示行による並列化促進（1）

NOSYNC 指示行

ループ中の配列要素に重なりがないことを指定する

例：a(I,k1,j+1)とa(I,k2,j) の依存関係が不明
k1≠k2であることが保証できるならa(I,k1,j+1)とa(I,k2,j) が
同じ要素となることはない

➔nosyncを指定して並列化可能

```
!cdir nosync
      do j = 1, ny
         do i = 1, nx
            a(i,k1,j+1) = a(i,k2,j) + b(i)
         enddo
      enddo   並列化
```

指示行による並列化促進（2）

ユーザ手続呼び出しのため並列化しない場合のメッセージ

ﾒｯｾｰｼﾞ No.	メッセージ
1380	利用者定義の関数参照があるため並列化できない
1382	サブルーチン呼び出しがあるため並列化できない

- その手続がそのループ内の他の繰り返しで定義される配列要素を定義･参照したり、他の繰り返しで参照される配列要素を定義していない場合

  ➔CNCALL指示行を指定

CNCALL 指示行

並列化されてもよい手続であることを指定する

```
!cdir cncall
  do i = 1, n
    call sub(a(i), x)
  enddo
```

並列化

指示行による並列化促進（3）

一重ループの並列化

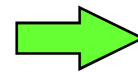
- 既定値では最内側ループは並列化しない
- ループの仕事量が非常に大きい（たとえばループの繰り返し数が十分に大きい）ことが分かっている場合

➔ **INNER指示行で並列化を指定**

INNER指示行

最内側ループあるいは一重ループを並列化の対象とすることを指定する

```
!cdir inner
    do i = 1, n
       a(i) = b(i) ** 2 + c(i) **2
    enddo
```



展開イメージ

条件並列化

```
if (n > 665) then
  ベクトル＋並列コード
else
  ベクトルコード
endif
```

強制並列化指示行（1）

並列化指示行を指定してもコンパイラが自動並列化しない

- 並列実行してもシリアル実行を同じ結果が得られることを保証できる場合
  ➔強制並列化指示行で並列化

強制並列化指示行

- 自動並列化で思うように並列化されない場合でも、ユーザ自身で簡単に並列化を指定できる
- コンパイラはデータの依存関係などのチェックは行わない
  ➔*ユーザが並列化しても大丈夫なことを保証しなければならない*

強制並列化指示行（2）

PARALLEL DO ［PRIVATE(var1［,var2…］)］

- 指定したループを並列実行する
- ループ内で作業用として使われるローカル変数やローカル配列はPRIVATEで指定する

例：

```
!CDIR PARALLEL DO PRIVATE(wk)
          do j= 1, 10
              do i = 1, 100
                 wk(i) = a(i)+b(j)
              enddo
            call sub(x(j),wk)
           enddo
```

強制並列化指示行（3）

- PARALLEL SECTIONS [PRIVATE(var1[,var2…])]
  SECTION
  END PARALLEL SECTIONS
  - PARALLEL SECTIONS/SECTION/END PARALLEL SECTIONSで区切られた各文の集まりを並列実行する

例：

```
!CDIR PARALLEL SECTIONS
          call sub1(x,y,100)
!CDIR SECTION
          call sub2(a,b,n)
!CDIR SECTION
          call sub3(a,b)
!CDIR END PARALLEL SECTIONS
```

強制並列化指示行（4）

ATOMIC

- PARALLEL DOで並列化されたループ中で、総和や内積など、排他的に処理しなければならない代入文の直前に指定する

例：

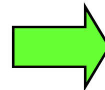
```
!CDIR PARALLEL DO
          do i= 1, n
               call sub(a(i),b(i),x)
!CDIR ATOMIC
            sum=sum+a(i)*b(I)
          enddo
```

プログラム修正による並列化(1)

作業配列を使用した依存関係の除去

```
do j=1,n
  do i=1,m
     処理1
     a(iy(j)) = a(iy(j)) +b(i,j)*c(j)
     処理2
  enddo
enddo
```

並列化不可の依存をもつ配列
a(iy(j))をループの外側に出す



```
do j=1,n
  s=0.0
  do i=1,m
     処理1
    s = s +b(i,j)*c(j)
     処理2
  enddo
  wk(j)=s
enddo
```

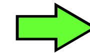
並列化可能

```
do j=1,n
  a(iy(j))=a(iy(j))+wk (j)
enddo
```

プログラム修正による並列化（2）

仮配列の次元数変更により並列化可能とする

```
subroutine sub(a,b,c,nx,ny,nz)
real*8 a(100,100,100),b(0:100,100,100)
real*8 c(0:100)
do k=1,nz
   do j=1,ny
      do i=0,nx
         c(i)=b(i,j,k)/dble(nx)
      enddo
      do i=1,nx
         a(i,j,k)=a(i,j,k)+(c(i-1)+c(i))/2.0
      enddo
   enddo
enddo
return
end
```



```
subroutine sub(a,b,c,nx,ny,nz)
real*8 a(100,100,100),b(0:100,100,100)
real*8 c(0:100, 100)
do k=1,nz
   do j=1,ny
      do i=0,nx
         c(i,k)=b(i,j,k)/dble(nx)
      enddo
      do i=1,nx
         a(i,j,k)=a(i,j,k)+(c(i-1,k)+c(i,k))/2.0
      enddo
   enddo
enddo
return
end
```

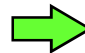
引数として渡ってきたデータはタスク間で共有となるため、配列cはタスク間共有変数となる。最外側ループ(k)で並列化すると、配列 c の領域を各タスクで書き換えるため結果不正となる。

次元の宣言を変更し、外側ループで異なる領域を使用すれば並列化が可能になる。なお、呼出し側のサブルーチンの修正も必要となる。

プログラム修正による並列化（3）

作業領域の受け渡しをしないようにして並列化可能とする

```
subroutine sub(a,b,c,nx,ny,nz)
real*8 a(100,100,100),b(0:100,100,100)
real*8 c(0:100)
do k=1,nz
   do j=1,ny
      do i=0,nx
         c(i)=b(i,j,k)/dble(nx)
      enddo
      do i=1,nx
         a(i,j,k)=a(i,j,k)+(c(i-1)+c(i))/2.0
      enddo
   enddo
enddo
return
end
```



```
subroutine sub(a,b,dummy,nx,ny,nz)
real*8 a(100,100,100),b(0:100,100,100)
real*8 c(0:100),dummy(0:100)
do k=1,nz
   do j=1,ny
      do i=0,nx
         c(i)=b(i,j,k)/dble(nx)
      enddo
      do i=1,nx
         a(i,j,k)=a(i,j,k)+(c(i-1)+c(i))/2.0
      enddo
   enddo
enddo
return
end
```

配列 c の値を呼出し側のサブルーチンが参照せず、
作業配列として使用している場合は、配列 c を引数
ではなくし、サブルーチンsub側で宣言することにより
並列化が可能となる。

負荷バランス

- PROGINFのConc.Timeにばらつき
  - ➔タスクの負荷バランスが悪い
  - ➔最も実行時間の大きなタスクの実行時間で、全体の実行時間が決まってしまう

➔各タスクの仕事量の均一化をはかることにより、実行時間を短縮

PROGINF (プログラム特性情報出力)

```
******  Program Information  ******
Real Time (sec)              :            0.307168
User Time (sec)              :            1.190239
Sys  Time (sec)              :            0.007852
Vector Time (sec)            :            1.167033
Inst. Count                  :           241880273
V. Inst. Count               :           117679849
V. Element Count             :         30126037402
V. Load Element Count        :         10741746602
FLOP Count                   :         17179869334
MOPS                         :        25415.263511
MFLOPS                       :        14433.966064
MOPS   (concurrent)          :       100019.632876
MFLOPS (concurrent)          :        56803.659976
A. V. Length                 :          255.999967
V. Op. Ratio (%)             :           99.589423
Memory Size (MB)             :          512.000000
Max Concurrent Proc.         :                   4
  Conc. Time(>= 1) (sec)     :            0.302443
  Conc. Time(>= 2) (sec)     :            0.301598
  Conc. Time(>= 3) (sec)     :            0.301181
  Conc. Time(>= 4) (sec)     :            0.285894
Event Busy Count             :                   0
Event Wait (sec)             :            0.000000
Lock Busy Count              :                   0
Lock Wait (sec)              :            0.000000
Barrier Busy Count           :                   0
Barrier Wait (sec)           :            0.000000
MIPS                         :          203.219919
MIPS (concurrent)            :          799.754906
I-Cache (sec)                :            0.000145
O-Cache (sec)                :            0.002183
Bank Conflict Time
  CPU Port Conf. (sec)       :            0.000239
  Memory Network Conf. (sec) :            0.714319
ADB Hit Element Ratio (%)    :           20.310514
```

経過時間（秒）
ユーザ時間（秒）
システム時間（秒）
ベクトル命令実行時間（秒）
全命令実行数
ベクトル命令実行数
ベクトル命令実行要素数
ベクトルロード要素数
浮動小数点データ実行要素数
MOPS 値
MFLOPS 値
MOPS 値 （実行時間換算）
MFLOPS 値（実行時間換算）
平均ベクトル長
ベクトル演算率（%）
メモリ使用量（MB）
最大同時実行可能プロセッサ数
1台以上で実行した時間（秒）
2台以上で実行した時間（秒）
3台以上で実行した時間（秒）
4台以上で実行した時間（秒）
イベントビジー回数
イベント待ち時間（秒）
ロックビジー回数
ロック待ち時間（秒）
バリアビジー回数
バリア待ち時間（秒）
MIPS 値
MIPS 値（実行時間換算）
命令キャッシュミス（秒）
オペランドキャッシュミス（秒）
CPUポート競合時間（秒）
メモリネットワーク競合時間（秒）
ADBヒット率（%）

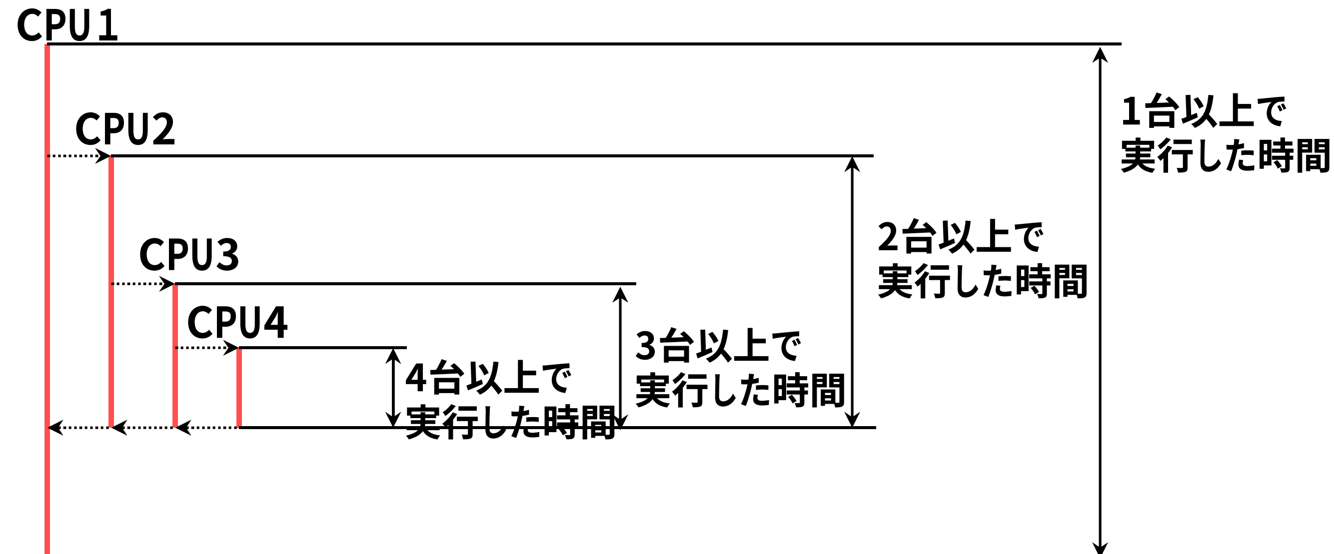
赤：並列処理使用時に表示

PROGINFの出力情報

Conc. Time　(Concurrent Time)

ＣＰＵ n台以上で実行した時間

- プログラムが動作したCPUの時間を知ることができる
- 少なくとも1台のCPUが動いた時間、少なくとも2台のCPUが動いた時間、…を表す



PROGINFを用いた性能分析

Conc. Time(>=1)と比べ、 Conc. Time(>=2)が小さい

```
Conc. Time(>= 1)(sec):        74.154168
Conc. Time(>= 2)(sec):         8.549322
Conc. Time(>= 3)(sec):         8.292376
Conc. Time(>= 4)(sec):         8.071275
```

➔並列化率が低い　➔並列化されていないループを並列化

Conc. Timeの値に偏りがある

```
Conc. Time(>= 1)(sec):        69.503482
Conc. Time(>= 2)(sec):        58.271920
Conc. Time(>= 3)(sec):        33.497481
Conc. Time(>= 4)(sec):        12.927761
```

➔負荷バランスが悪い　➔ループ並列実行方法の変更

簡易性能解析機能 (ftrace)

並列化時の出力情報

- 並列実行されたサブルーチンに対して各タスクの実行情報を表示
- タスクの負荷バランスを確認できる

```
PROG.UNIT   FREQUENCY  EXCLUSIVE         AVER.TIME     MOPS   MFLOPS V.OP   AVER.    VECTOR  …
                       TIME[sec](   % )    [msec]                    RATIO V.LEN     TIME

sub2_$5          2420    177.234( 37.2)     73.237 18711.9   9327.4 99.70 250.3   175.516
 -micro1          605     44.431(  9.3)     73.440 18716.2   9329.6 99.70 250.3    44.011
 -micro2          605     44.415(  9.3)     73.413 18622.1   9282.6 99.70 250.3    43.773
 -micro3          605     43.972(  9.2)     72.680 18795.5   9369.2 99.70 250.2    43.741
 -micro4          605     44.416(  9.3)     73.415 18714.7   9328.8 99.70 250.3    43.992
sub1_$1         24200     23.534(  4.9)      0.972 21650.1   8688.7 99.72 253.0    22.827
 -micro1         6050      6.009(  1.3)      0.993 21245.2   8526.0 99.72 253.0     5.719
 -micro2         6050      5.863(  1.2)      0.969 21697.5   8707.7 99.72 253.0     5.700
 -micro3         6050      5.795(  1.2)      0.958 21919.4   8796.9 99.72 253.0     5.691
 -micro4         6050      5.867(  1.2)      0.970 21751.6   8729.4 99.72 253.0     5.717
    :             :          :
-------------------------------------------------------------------------------------------
total          215401    476.648(100.0)      2.213 19169.7   9032.0 99.51 163.1   466.969  …
```

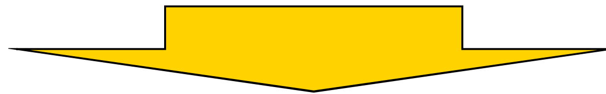
負荷バランスの改善策

ループの分割方法、分割数の変更

- 分割数小
  - メリット：　同期処理の回数が少なくて済むためオーバヘッド小
  - デメリット：各タスクの仕事量がアンバランスになり易い
- 分割数大
  - メリット：　各タスクの仕事量のばらつきは小さくなる
  - デメリット：　同期処理の回数が増えるためオーバヘッド大

ループの並列実行方法

- by=*n*
  - 指定した数*n*の繰り返し数をもつループに分割
- for[=*m*]
  - 繰り返し数を指定した数*m*に分割
  - 数の指定がない場合は実行時に確保されたタスク数に分割
    （自動並列化の既定値）



- ループの並列実行方法を指定するコンパイラオプション

  -Wf,-pvctl {by=*n* ¦ for[=*m*]}

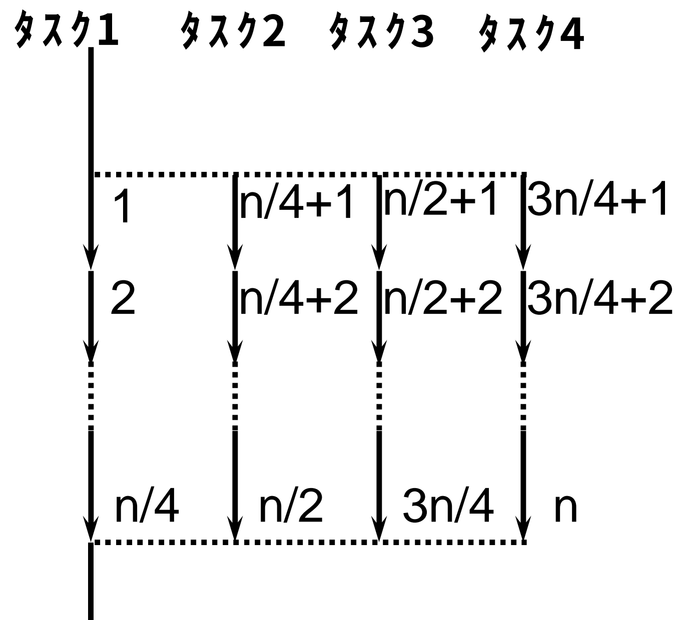
- ループの並列実行方法を指定する指示行

  !CDIR CONCUR(BY=*n* ¦ FOR[=*m*])

forとby

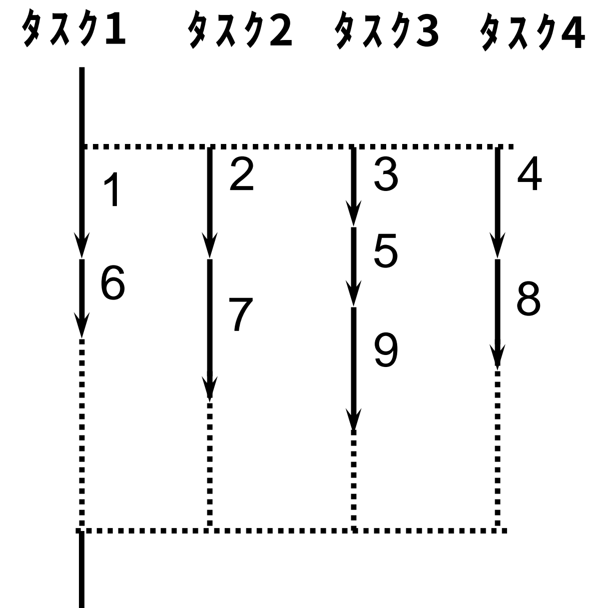
```
do I=1, n
    ～
enddo
```

for=4



繰り返し数nを４分割して４タスクで実行

by=1

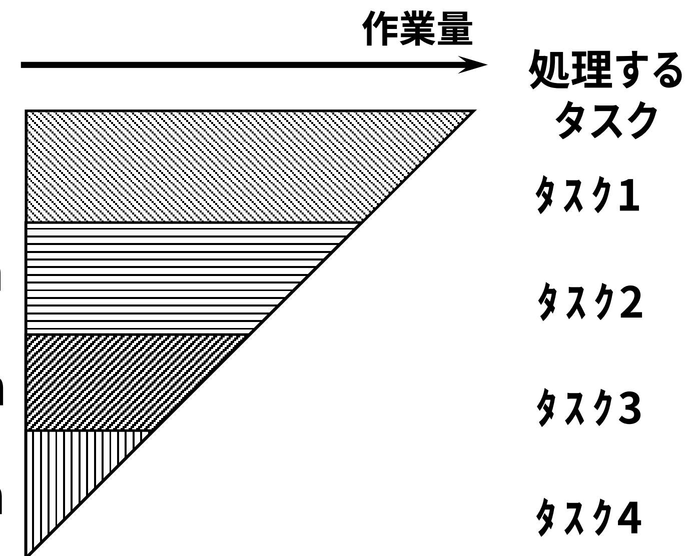


各繰り返しを１個の処理単位として、
４タスクで分担して実行

三角行列の計算（1）

以下のループをタスク数＝4で実行

```
!CDIR CONCUR(FOR=4)
do j=1,8*n
    do i=1,8*n-j+1
        a(i,j)=b(i,j)*c(i,j)
    enddo
enddo
```



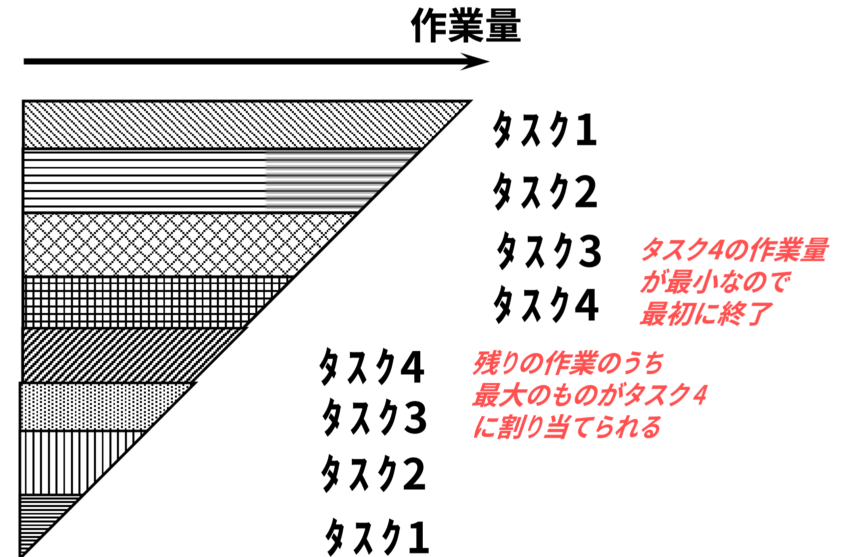
内側ループの繰り返し数＝作業量は、
外側ループの繰り返しが進むにつれて減少

タスク1の作業量はタスク4の7倍
大きなインバランス発生

三角行列の計算（2）

ループの分割数を8に変更し4タスクで実行

```
!CDIR CONCUR(FOR=8)
do j=1,8*n
    do i=1,8*n-j+1
        a(i,j)=b(i,j)*c(i,j)
    enddo
enddo
```



セルフスケジューリングにより
各タスクの作業量が均等になる

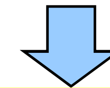
三角行列の計算（3）

以下のループをタスク数＝4で実行

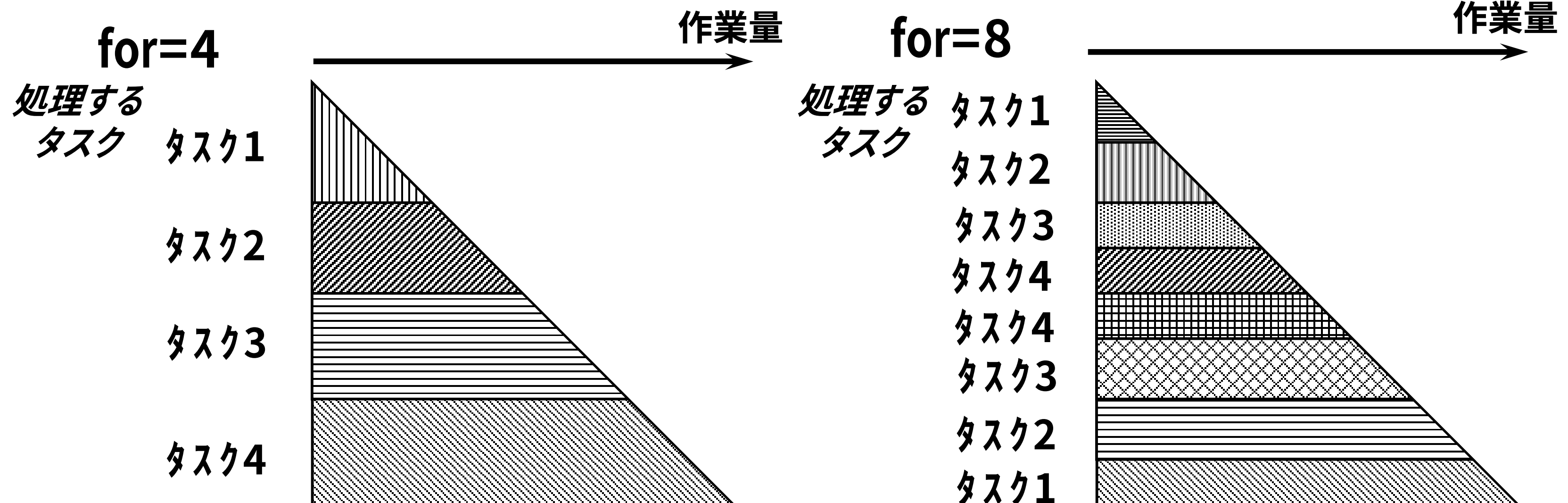
```
do j=1,n
    do i=1, j
        a(i,j)=b(i,j)*c(i,j)
    enddo
enddo
```

内側ループの繰り返し数＝作業量は、
外側ループの繰り返しが進むにつれて***増加***

このパターンでは分割数を8にしても
インバランスは解消されない



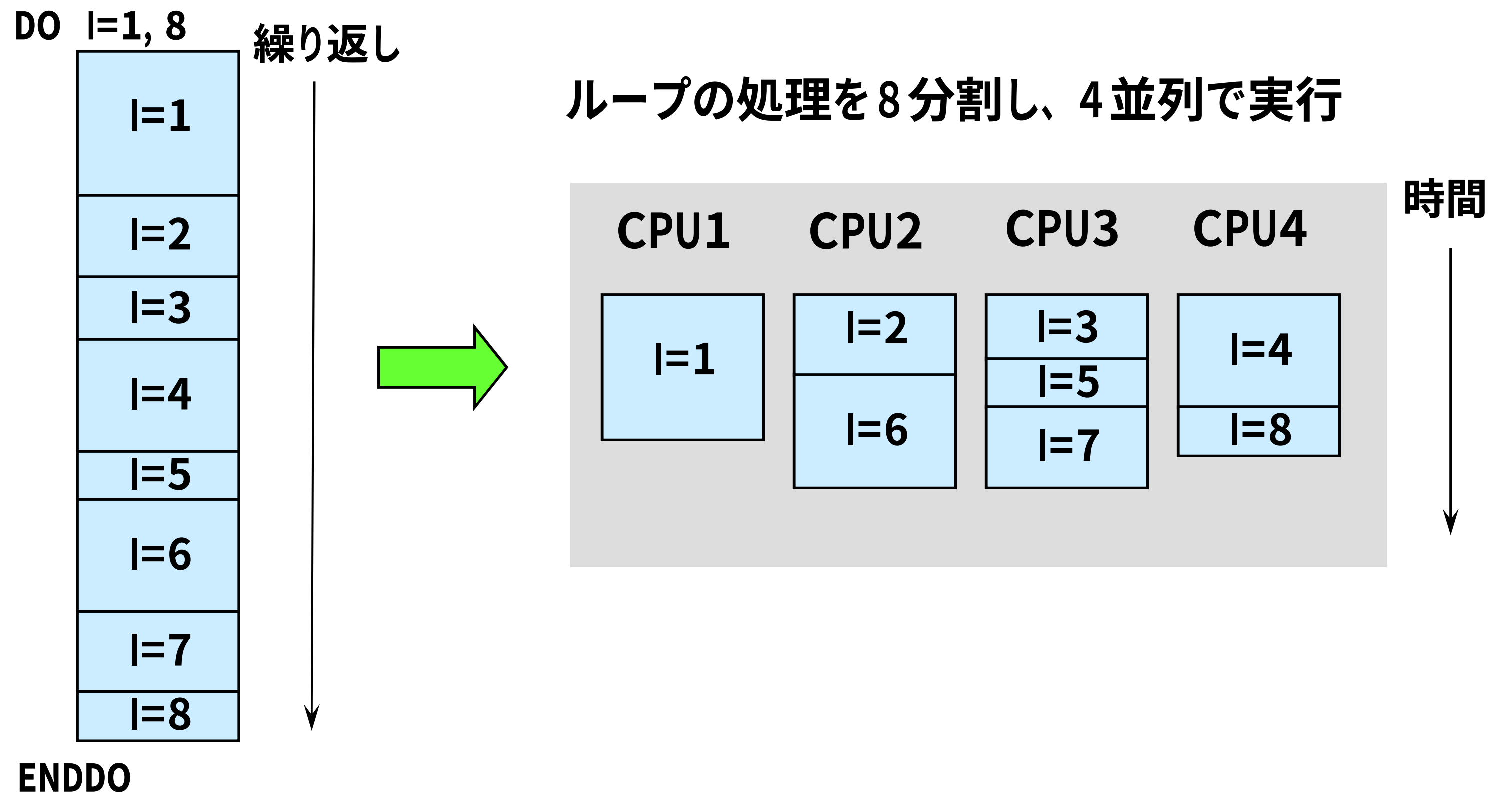
分割数を増やすことにより、
インバランスを小さくすることは可能



並列ループの実行方式

セルフスケジューリング

- 各タスクには、処理が終了した順に次の処理が割り当てられる



目次

- 並列処理とは
- 並列化における注意事項
- 並列化のチューニング
- OpenMP (参考資料)

OpenMP

共有メモリマシン向け並列処理の標準API

- 異なる共有メモリアーキテクチャを持つベンダ間で可搬なプログラミングモデルを提供
- 並列化は利用者がすべて明示的に記述
  - ディレクティブで動作を指示
  - 実行時ライブラリ、環境変数も用意されている

参考：OpenMPに関するWeb情報
http://www.openmp.org/

OpenMPの仕様：形式

ディレクティブ

!$OMP *ディレクティブ名 [clause[[,]clause]…]*

- “!$OMP” は1カラム目から空白なしに記述
- 固定形式の場合は“*$OMP” または “C$OMP” も使用可

条件付きコンパイル

!$ *Fortranの文*

- OpenMPが有効の場合、“!$” が2文字の空白として翻訳される
- 固定形式の場合は “*$” または “C$” も使用可

備考： “!$”, “*$”, “C$” をコメントのままにしたい場合は
オプション −Wf,-ompctl nocondcomp
で条件付きコンパイル機能を無効にする

OpenMPの例

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(field, ispectrum)
               call initialize_field(field, ispectrum)
               call compute_field(field, ispectrum)
               call compute_spectrum(field, ispectrum)
!$OMP END PARALLEL
                  ......
               subroutine initialize_field(field, ispectrum)
                  ......
!$OMP DO
               do i = 1, nzone
                   ispectrum(i) = 0
               enddo
!$OMP enddo NOWAIT
!$OMP DO
               do j = 1, npoints
                   field(j) = 0
               enddo
!$OMP enddo
!$OMP SINGLE
               field(npoints/4) = 1.0
!$OMP END SINGLE
             return
               end
```

並列実行を行う範囲を指定
※共有変数に関する排他制御等は利用者が制御

DOループを並列実行

1スレッドのみ実行

OpenMPの仕様：ディレクティブ

ディレクティブ名

パラレルリージョン構造

PARALLEL / END PARALLEL

Work-Sharing構造

DO / enddo
SECTIONS / SECTION / END SECTIONS
SINGLE / END SINGLE

複合Work-Sharing構造

PARALLEL DO / END PARALLEL DO
PARALLEL SECTIONS / END PARALLEL SECTIONS

同期構造

MASTER / END MASTER
CRITICAL / END CRITICAL
BARRIER
ATMIC
FLUSH
ODERED / END ORDERED

データスコープ

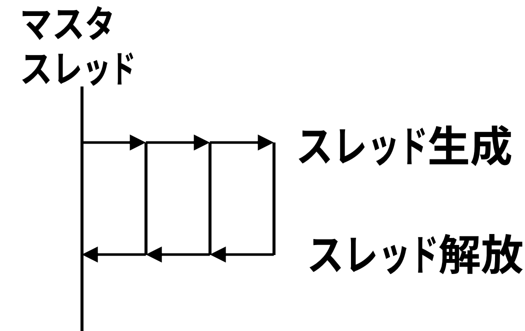
THREADPRIVATE

OpenMPの仕様 (パラレルリージョン)

パラレルリージョン

- 並列に実行されるコードのブロック
- PARALLEL/END PARALLELディレクティブで範囲を指定

```
        :
!$OMP PARALLEL
  ブロック
!$OMP END PARALLEL
        :
```



- パラレルリージョン内からの飛び出し、パラレルリージョン内への飛び込みは許されない。
- パラレルリージョン内からの手続き呼び出しは許される。
  呼び出された手続きは各スレッドで並列実行される。

OpenMPの仕様 : Work - Sharing構造

Work - Sharing構造

- 囲まれたコードの範囲を各スレッドで分割して実行
- パラレルリージョン内になければならない

並列DOループ

```
!$OMP PARALLEL
    …
!$OMP DO
    DO I=1,N
      …
    ENDDO
!$OMP ENDDO
    …
!$OMP END PARALLEL
```

ループの繰り返しを各スレッドで実行

並列セクション

```
!$OMP PARALLEL
    …
!$OMP SECTIONS
        ﾌﾞﾛｯｸA
!$OMP SECTION
        ﾌﾞﾛｯｸB
!$OMP END SECTIONS
    …
!$OMP END PARALLEL
```

ブロックA,Bを別々のスレッドで実行

SINGLEセクション

```
!$OMP PARALLEL
    …
!$OMP SINGLE
        ﾌﾞﾛｯｸA
!$OMP END SINGLE
    …
!$OMP END PARALLEL
```

ブロックAを１個のスレッドだけで実行

OpenMPの例：並列DOループ

例： S に対する総和を並列実行

```
!$OMP PARALLEL PRIVATE(X)
!$OMP DO REDUCTION(+: S)
      DO I = 1, N
         X = W(I) * (I-0.5)
         S = S + FUN(X)
      enddo
!$OMP enddo
!$OMP END PARALLEL
```

総和のようなリダクション演算がある場合、REDUCTION clause を記述

上の例は右のように書くこともできる

```
!$OMP PARALLEL DO PRIVATE(X),REDUCTION(+: S)
      DO I = 1, N
         X = W(I) * (I-0.5)
         S = S + FUN(X)
      enddo
!$OMP END PARALLEL DO
```

OpenMPの仕様 (同期構造)

同期構造

- スレッド間の同期処理を指定
- MASTER, CRITICAL, BARRIER, FLUSH, ORDERED

MASTER

```
!$OMP PARALLEL
     …
!$OMP MASTER
     ﾌﾞﾛｯｸA
!$OMP END MASTER
     …
!$OMP END PARALLEL
```

ﾌﾞﾛｯｸAをﾏｽﾀｽﾚｯﾄﾞだけで実行

CRITICAL

```
!$OMP PARALLEL
     …
!$OMP CRITICAL
     ﾌﾞﾛｯｸA
!$OMP END CRITICAL
     …
!$OMP END PARALLEL
```

ﾌﾞﾛｯｸAは各ｽﾚｯﾄﾞで排他的に実行される

ATOMIC

```
!$OMP PARALLEL DO
     …
!$OMP ATOMIC
     X = X + 式
!$OMP END PARALLEL DO
```

直後の代入文の X のﾛｰﾄﾞからｽﾄｱまでを各ｽﾚｯﾄﾞで排他的に実行する

OpenMPの仕様 : データスコープ属性

データスコープ属性

- パラレルリージョン内の変数が、スレッドで固有(private)となるか共有(shared)となるかを PRIVATE/SHARED clauseで指定
  - データスコープの既定値はshared（DEFAULT clauseで変更可能）

例:

```
!$OMP PARALLEL DO PRIVATE(WK)
   DO I=1,N
     CALL SUB(X,Y,WK)
   ENDDO
!$OMP END PARALLEL DO
```

- WKはprivate, X,Y,Nはshared
- ｻﾌﾞﾙｰﾁﾝSUBのﾛｰｶﾙ変数はprivate
- DO変数 I は private

- THREADPRIVATEディレクティブ
  - 名前付き共通ブロックをスレッド間でprivateとする

例:

```
  COMMON /CM1/A,B
!$OMP THREADPRIVATE(CM1)
```

OpenMPの実行時ライブラリ

実行環境ルーチン

OMP_SET_NUM_THREADS	スレッドの数を設定
OMP_GET_NUM_THREADS	現在のスレッド数を得る
OMP_GET_THREAD_NUM	自分のスレッド番号を得る
OMP_GET_NUM_PROCS	使用可能なプロセッサ数を得る
OMP_SET_DYNAMIC	スレッド数の動的制御のON / OFF
他	

ロックルーチン

OMP_INIT_LOCK	ロックの初期化
OMP_DESTROY_LOCK	ロックの開放
OMP_SET_LOCK	ロックのセット
OMP_UNSET_LOCK	ロックの解除
OMP_TEST_LOCK	ロックのセットを試みる

OpenMPの環境変数

環境変数

OMP_SCHEDULE

並列DOループの分割方式を選択(STATIC / DYNAMIC / GUIDED)

FORTRAN90 / SXの既定値は STATIC

OMP_NUM_THREADS

実行中に使用するスレッド数を設定

OMP_DYNAMIC

実行中に使用するスレッド数の動的変更を有効 / 無効にする

FORTRAN90 / SXの既定値は FALSE (無効)

OMP_NESTED

並列のネストを有効 / 無効にする

FORTRAN90 / SXでは並列のネストは不可(常に無効)

FORTRAN90/SXでの使用方法

オプション -Popenmp を指定してコンパイル・リンク

例：

```
% sxf90 -Popenmp prog.f90
```

注意：

- 自動並列化・マイクロタスク関連のコンパイルオプションは同時に指定できない
- ソース中の自動並列化・マイクロタスク関連のコンパイル指示行は無効となる

OpenMP関連のコンパイラオプション

-Wf,-ompctl condcomp/nocondcomp

condcomp ：条件付き翻訳を有効にする

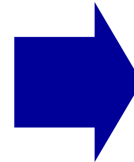
nocondcomp ：条件付き翻訳を無効にする

FORTRAN90 / SXの実装方式

各パラレルリージョンに対し並列サブルーチンが生成される

```
program main
   ･･･…
!$omp parallel
!$omp do
        do i = 1, n
              ･･････.
        enddo
!$omp enddo
!$omp do
        do i = 1, n
            do iter = 1, maxiter
                ･･････
            enddo
        enddo
!$omp enddo
!$omp end parallel
   ･･･…
end
```

ソース



```
program main
   ･･･…
!CDIR OMP_RESERVE
call  main$1
!CDIR OMP_RELEASE
   ･･･…
end

subroutine main$1
!CDIR OMP_PARALLEL
!CDIR OMP_PARDO
        do i = 1, n
           ･･･…
        enddo
!CDIR OMP_PARDO
        do i = 1, n
                do iter = 1, maxiter
                    ･･･…
                enddo
        enddo
end
```

翻訳イメージ