スケーラブルシステムズ株式会社

# OpenMPプログラミング入門（Part 2）

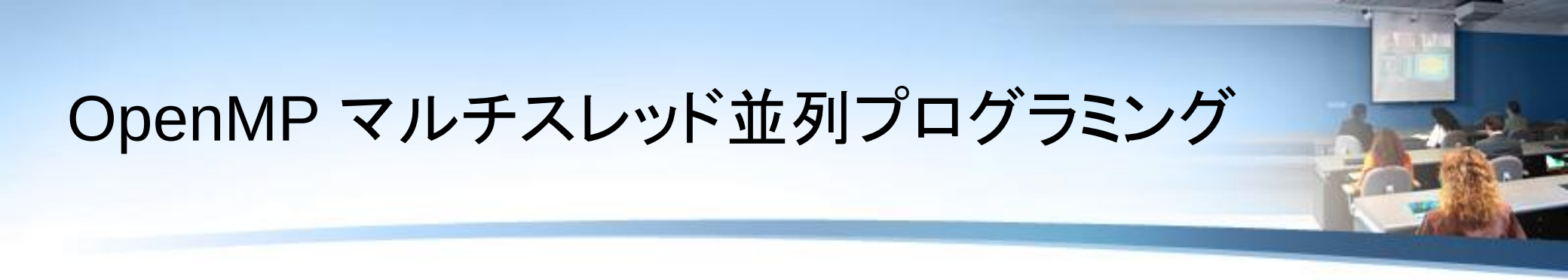
# 講習の内容：Part 2

- OpenMPの概要について
- OpenMP APIのご紹介
  1. 並列実行領域（Parallel Regions）構文
  2. ワークシェアリング（Worksharing）構文
  3. データ環境（Data Environment）構文
  4. 同期（Synchronization）構文
  5. 実行時関数/環境変数（Runtime functions/environment variables）

OpenMPによるマルチスレッドプログラミング

# OpenMPの概要について

# OpenMP マルチスレッド並列プログラミング



- OpenMP は、マルチスレッド並列プログラミングのための API (Application Programming Interface)
  - OpenMP API は、1997年に発表され、その後継続的に、バージョンアップされている業界標準規格
  - 多くののハードウェアおよびソフトウェア・ベンダーが参加する非営利会社（Open MP Architecture Review Board）によって管理されており、Linux、UNIX そして 、Windows システムで利用可能
- OpenMP は、C/C++ や Fortran と言ったコンパイラ言語ではない
  - コンパイラに対する並列処理の機能拡張を規定したもの
  - OpenMP を利用するには、インテルコンパイラ  バージョン 9.0 シリーズのような OpenMP をサポートするコンパイラが必要

# OpenMPの特徴

- コンパイラのサポート
  - コンパイラ・オプションでの適用、非適用の選択が可能（Windows：/Qopenmp スイッチ、Linux:-openmp スイッチ_）
  - スレッドの生成や各スレッドの同期コントロールといった制御を気にする必要がない
  - OpenMP_での並列化を適用していて、計算などが不正になった場合、簡単にその部分だけを逐次実行に切り替えることも可能（プログラムのデバッグが容易）
- 明示的な並列化の指示
  - コンパイラに対して、並列化のためのヒントを与えるのでなく、明示的に並列化を指示→間違った指示行を指定しても、コンパイラはその指示に従って、並列化を行う

# マルチスレッドプログラミングの基本

- 計算負荷の大きなループやプログラムのセクションを複数のスレッドで同時に処理
- 複数のスレッドを複数のプロセッサコア上で、効率良く処理する

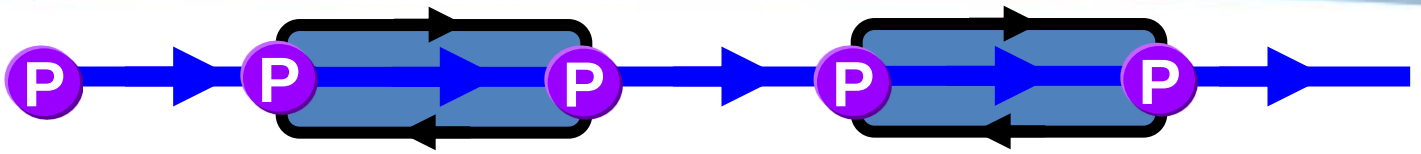
```
void main()
{
    double Res[1000];
// 計算負荷の大きな計算ループに対して、
// マルチスレッドでの並列処理を適用します
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

OpenMP の適用

```
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

# 逐次処理 .vs. マルチスレッド並列処理

逐次処理

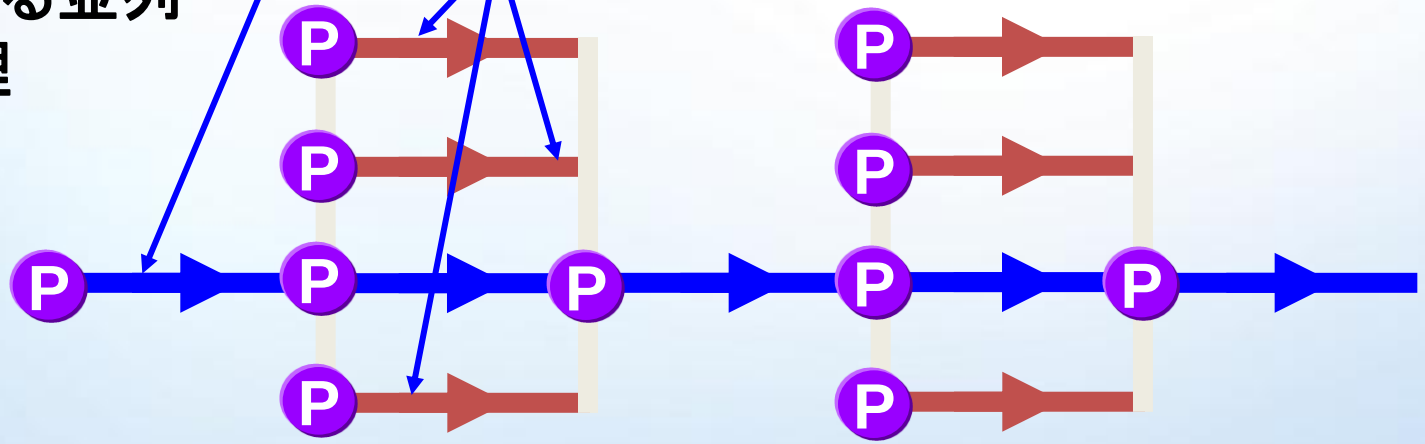



プログラムのループなどの反復計算を複数のスレッドに分割し、並列処理を行う

“マスタースレッド”

“ワーカースレッド”

マルチスレッドによる並列処理



# OpenMP以前

## OpenMP pre-history

- OpenMP based upon SMP direct
  standardization efforts PCF and
  X3H5 – late 80's
- Nobody fully implemented either
- Only a couple of partial solution
- Vendors considered proprietary
  competitive feature:
  - Every vendor had proprietary direc
  - Even KAP, a "portable" multi-platfo
    tool used different directives on ea

PCF – Parallel computing forum KAP – parallelization to

12

## History of OpenMP

SGI
Cray
Merged, needed commonality across products

KAI
ISV - needed larger market

ASCI
was tired of recoding for SMPs. Urged vendors to standardize.

Wrote a rough draft straw man SMP API

Other vendors invited to join

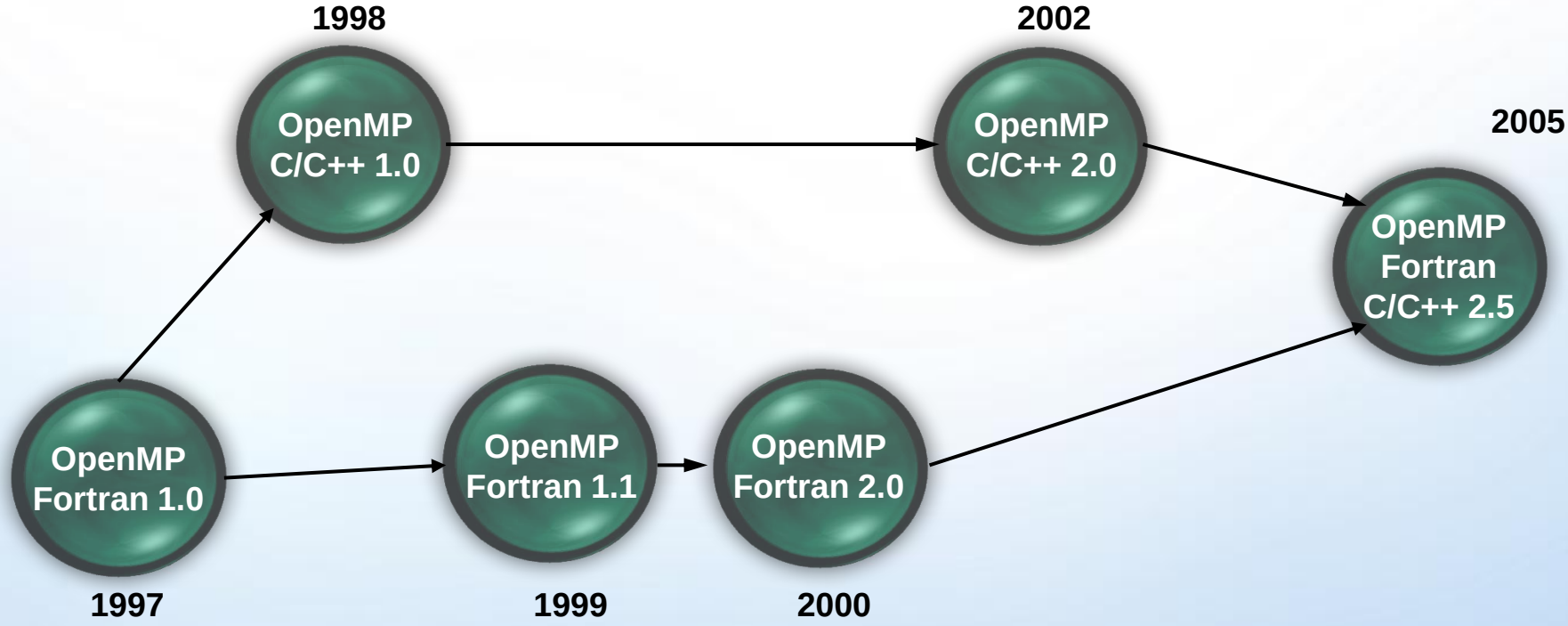
DEC
HP
IBM
Intel

OpenMP
1997

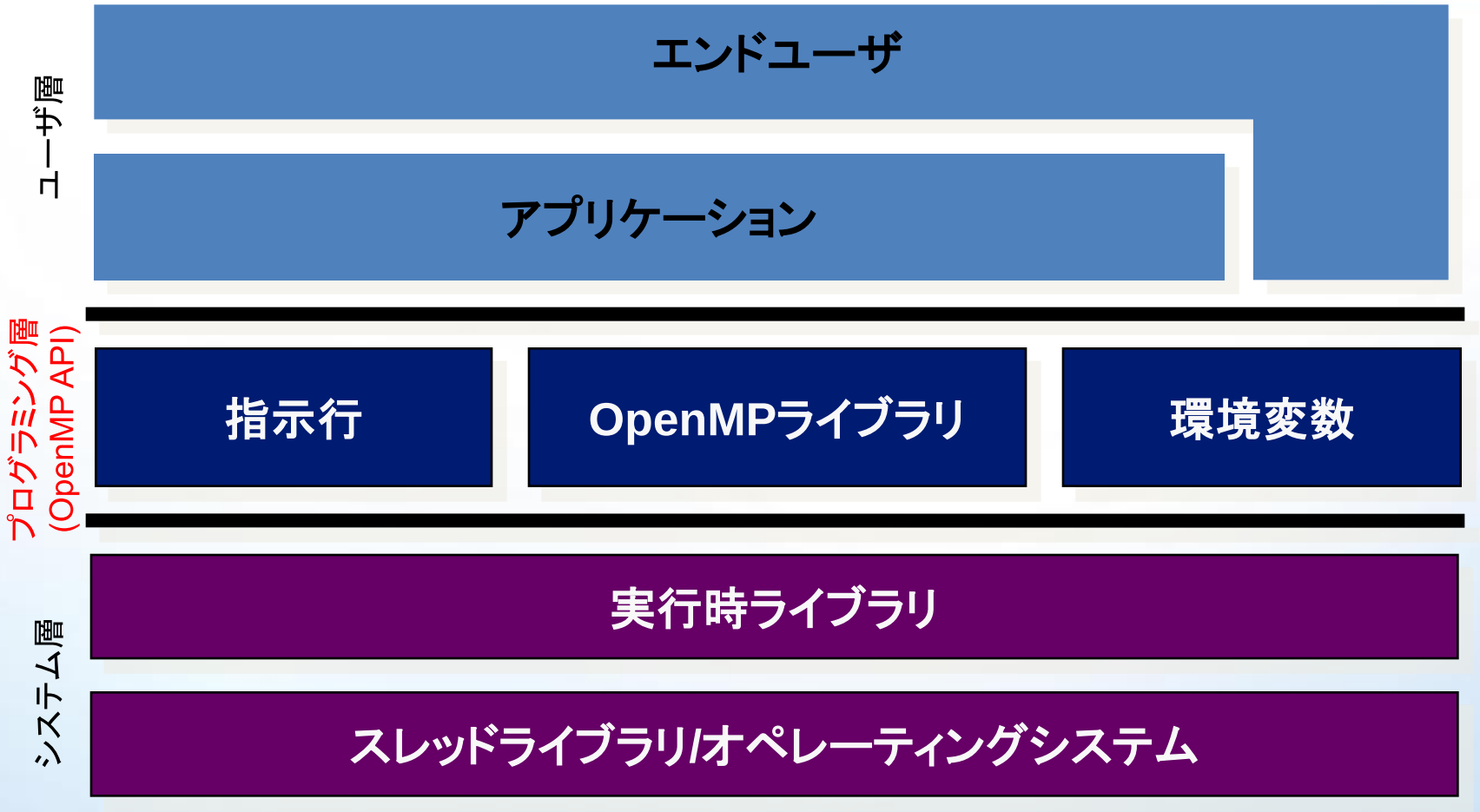
13

# OpenMP APIのリリースの歴史

- OpenMPの詳細な仕様などは、OpenMPのホームページ www.openmp.org で入手することが可能です。最新のOpenMPのリリースは、2005年5月に発表された、OpenMP 2.5であり、この仕様でC/C++とFortranの規格が統合されました。



# OpenMP APIの構造



# OpenMPの特徴

- プログラムの段階的な並列化が可能
  - コードの設計時からOpenMPを利用した並列処理を実装することも可能
  - 既に開発されたプログラムを、OpenMPを利用して、段階的に並列化することも可能
- 自動並列化との併用
  - 自動並列化とOpenMPを併用することも可能であり、プログラムの一部だけをOpenMPで並列化し、他の部分を自動並列化することも可能
- WindowsでもLinuxでも同じAPIが利用可能
  - ソースの互換性
- 疎粒度での並列化の適用
  - 自動では並列化が難しい関数やサブルーチンの呼び出しを含むタスクでの並列化（疎粒度での並列化）も可能
  - 粒度の大きな並列化では、よりオーバーヘッドの小さな並列化処理が可能

# OpenMP マルチスレッド並列プログラミング

- OpenMP の詳細については、OpenMP のホームページ www.openmp.org にその歴史も含めて、詳細な情報があります。最新の OpenMP のリリースは、2005年5月の OpenMP 2.5であり、この仕様では初めて、C/C++ と Fortran の双方の規格の統合がなされました。

# OpenMP プログラムのコンパイルと実行例

```
$ cat -n pi.c
     1  #include <omp.h>                                  // OpenMP実行時関数呼び出し
     2  #include <stdio.h>                                // のためのヘッダファイルの指定
     3  #include <time.h>
     4  static int num_steps = 1000000000;
     5  double step;
     6  int main ()
     7  {
     8          int i, nthreads;
     9          double start_time, stop_time;
    10          double x, pi, sum = 0.0;
    11          step = 1.0/(double) num_steps;            // OpenMPサンプルプログラム:
    12          #pragma omp parallel private(x)           // 並列実行領域の設定
    13          {       nthreads = omp_get_num_threads(); // 実行時関数によるスレッド数の取得
    14                  #pragma omp for reduction(+:sum)  // "for" ワークシェア構文
    15                  for (i=0;i< num_steps; i++){      // privateとreduction指示句
    16                          x = (i+0.5)*step;         // の指定
    17                          sum = sum + 4.0/(1.0+x*x);
    18                  }
    19          }
    20          pi = step * sum;
    21          printf("%5d Threads : The value of PI is %10.7f¥n",nthreads,pi);
    22  }
$ icc -O -openmp pi.c
pi.c(14) : (col. 3) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
pi.c(12) : (col. 2) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
$ setenv OMP_NUM_THREADS 2
$ a.out
    2 Threads : The value of PI is  3.1415927
```

OpenMP指示行

OpenMP実行時関数

コンパイルとメッセージ

環境変数の設定

# 補足説明

- この講習では、コンパイラ・オプションなどのコンパイラの利用方法についての説明は行いません。
- コンパイラの利用方法やコンパイラのメッセージについては、お使いのコンパイラのマニュアルなどをご参照ください。

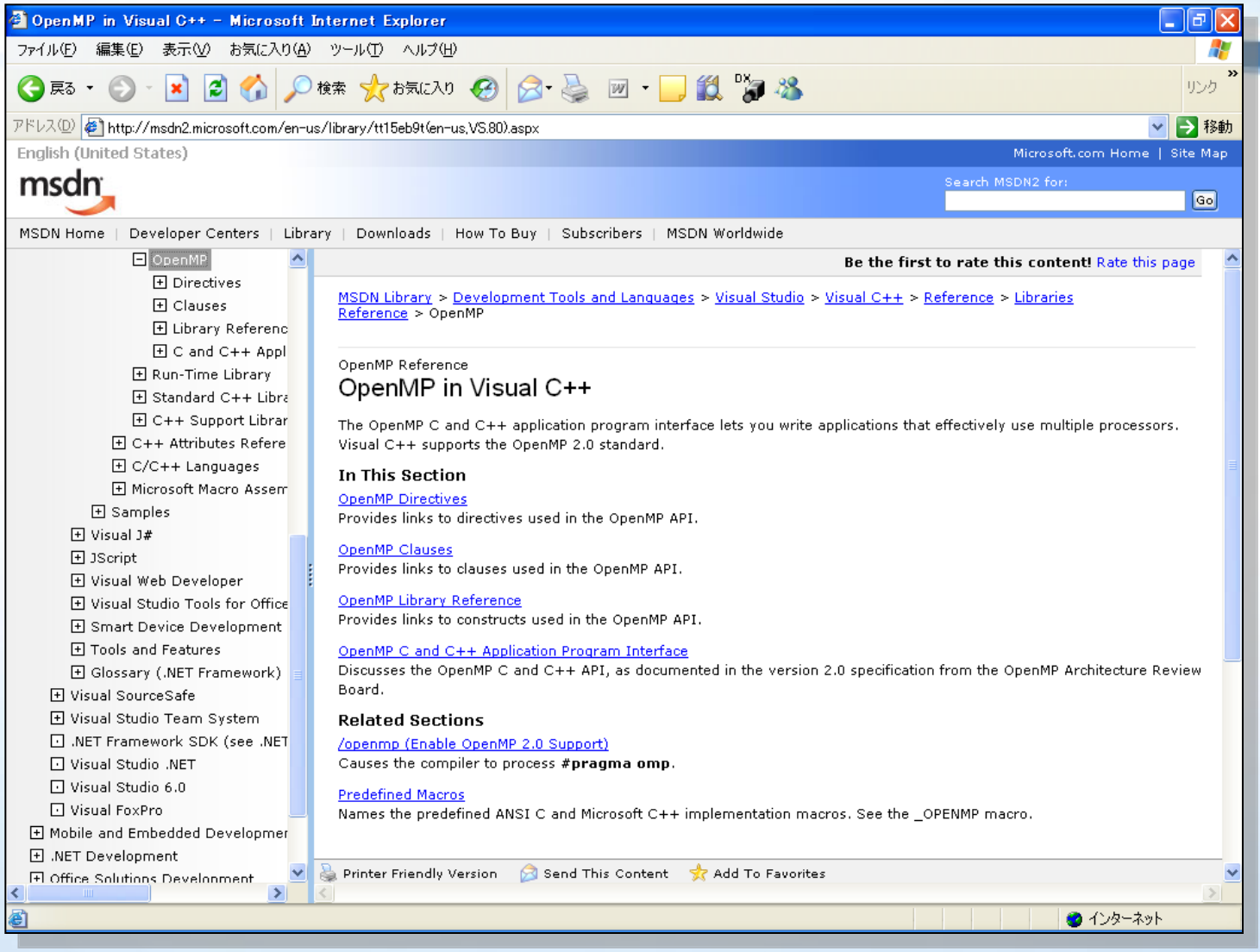
インテルソフトウエア開発製品

- http://www.intel.co.jp/jp/developer/software/products/

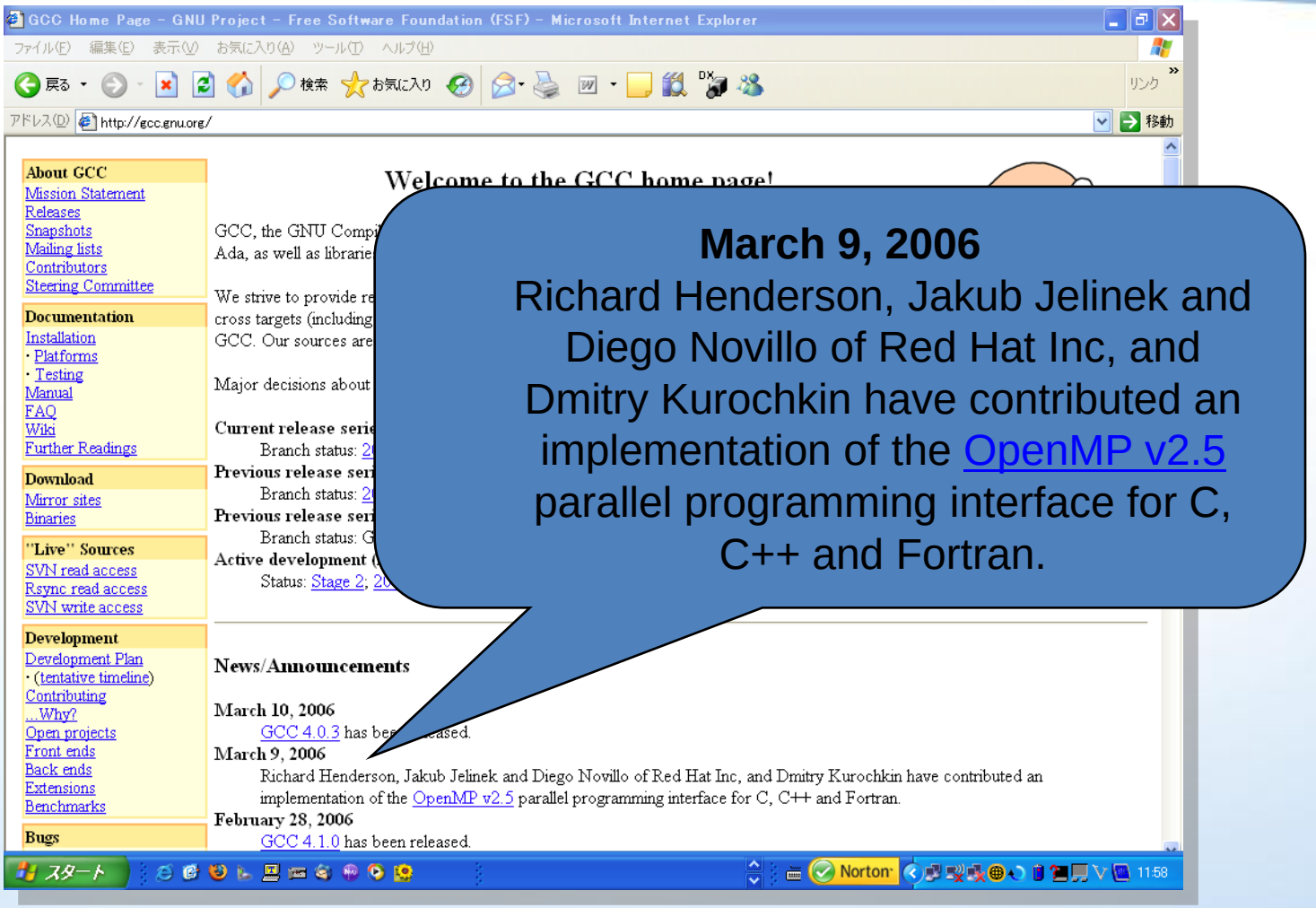
マイクロソフト

- http://msdn2.microsoft.com/en-us/library/tt15eb9t(en-us,VS.80).aspx

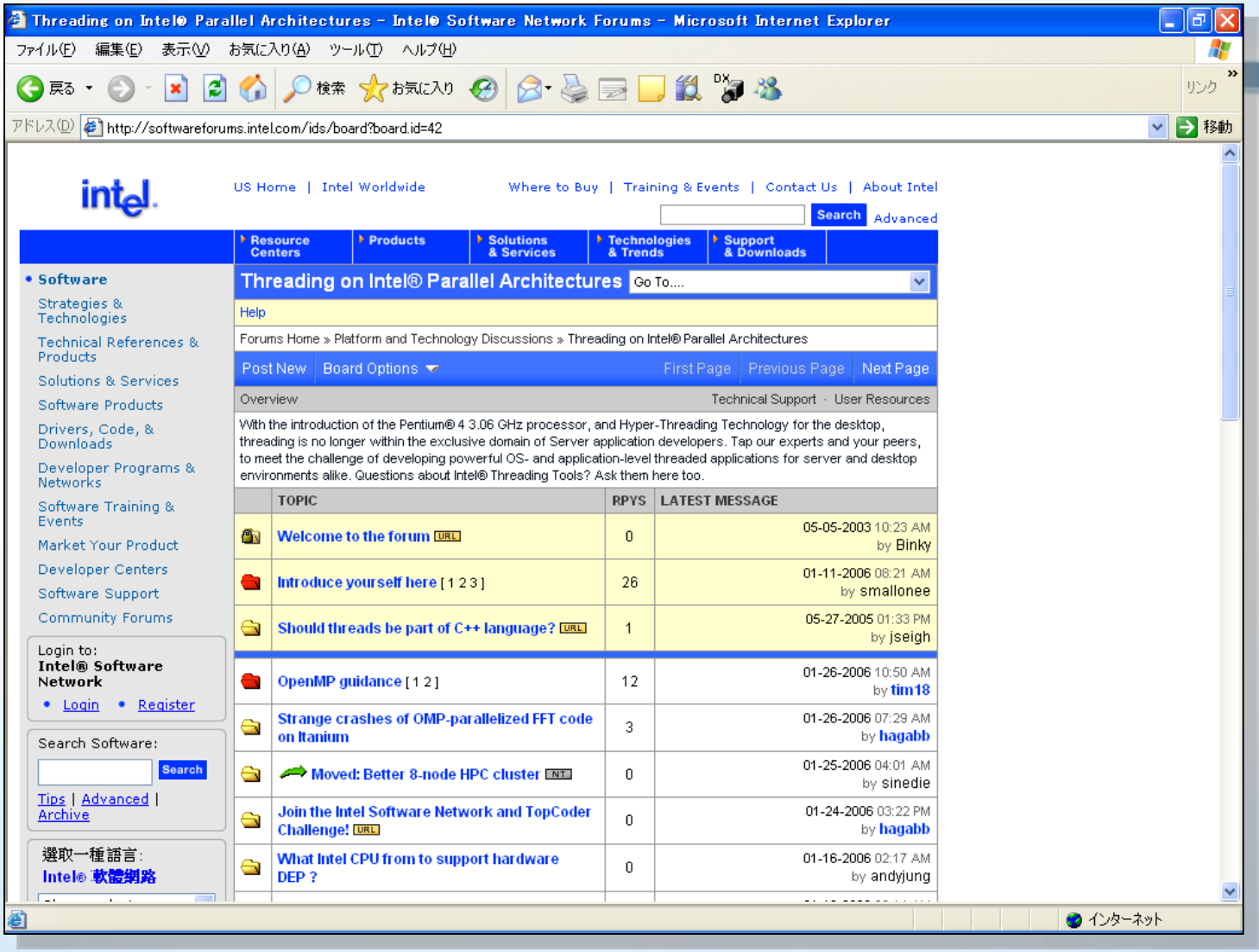
# OpenMP in Visual C++



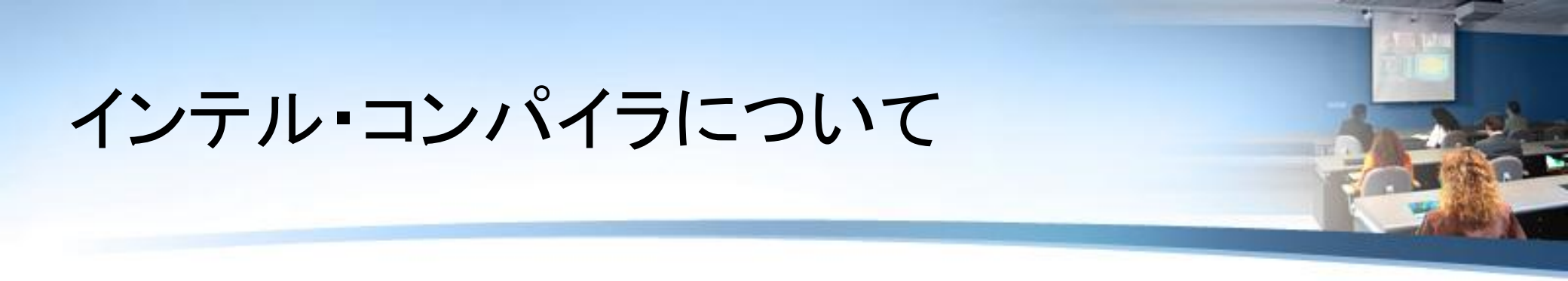
# GNU Compiler Collection



# Intel® Software Network Forums



# インテル・コンパイラについて



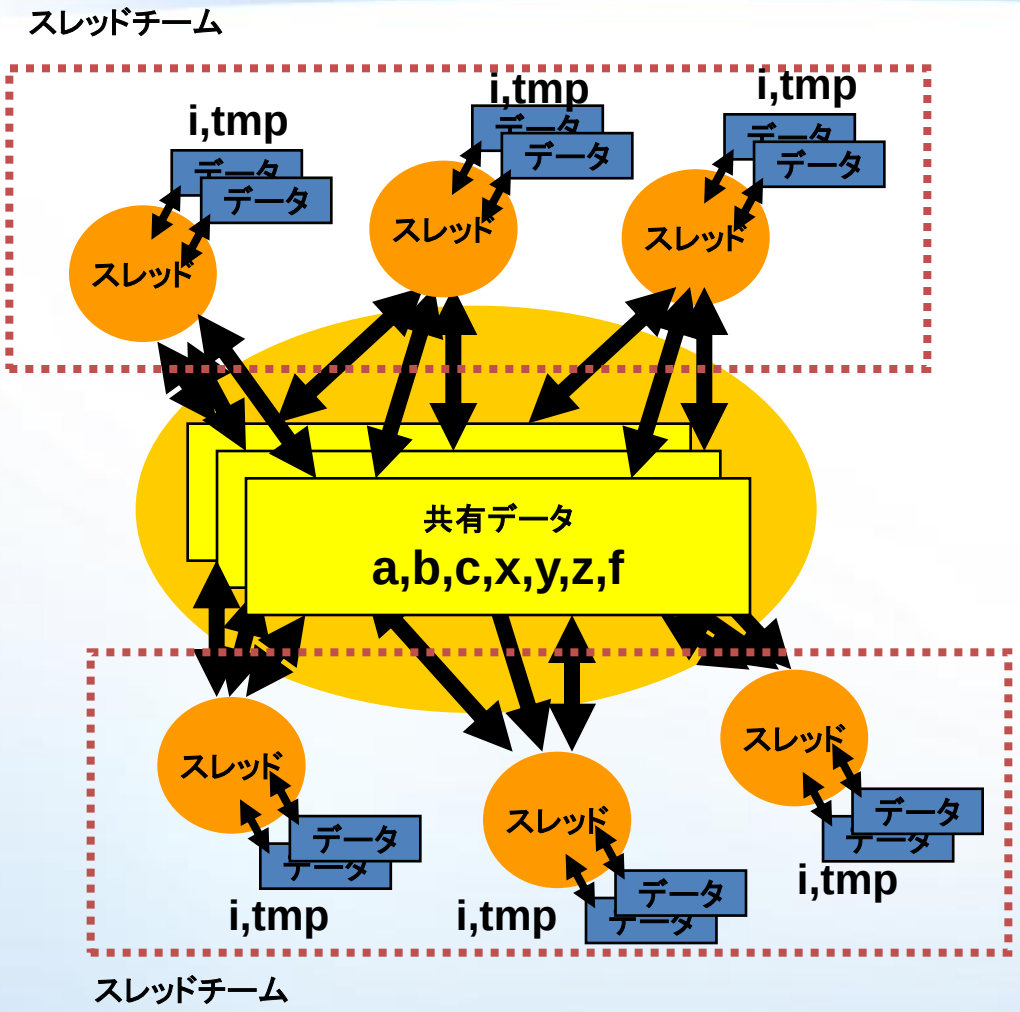
| Windows | Linux | 説明 |
|---|---|---|
| **/Qopenmp** | **-openmp** | OpenMP* ディレクティブに基づいてマルチスレッド・コードを生成する処理をパラレライザに許可します。 |
| **/Qopenmp_report{0\|1\|2}** | **-openmp_report{0\|1\|2}** | OpenMP パラレライザの診断レベルを制御します。デフォルトは、/Qopenmp_report1 です。 |
| **/Qparallel** | **-parallel** | 安全に並列実行可能な並列ループを検出し、そのループに対するマルチスレッド・コードを自動的に生成します。 |
| **/Qpar_report{0\|1\|2\|3}** | **-par_report{0\|1\|2\|3}** | 自動パラレライザの診断レベルを次のように制御します。<br>0 - 診断情報を表示しません。<br>1 - 正常に並列化されたループを示します（デフォルト）。<br>2 - ループの並列化の成功または不成功を表示します。<br>3 - 並列化の妨げになると判断された、または想定される依存関係についての情報が追加されます。 |
| **/Qpar_threshold[n]** | **-par_threshold[n]** | ループの並列化による効果が現れる確率に基づいてループの自動並列化のしきい値を設定します（n=0 から n=100。デフォルト: n=75）。<br>このオプションは、コンパイル時に計算量が確定できないループに使用します。<br>0 - 計算量にかかわらず並列化を行います。<br>100 - ループは並列実行が有効であることが確実な場合にのみ並列化されます。 |

# コンパイラメッセージ

```
 1  PROGRAM ORDERED
 2    IMPLICIT NONE
 3    INTEGER, PARAMETER:: N=1000, M=4000
 4    REAL, DIMENSION(N,M):: X,Y
 5    REAL, DIMENSION(N):: Z
 6    INTEGER I,J
 7    CALL RANDOM_NUMBER(X)
 8    CALL RANDOM_NUMBER(Y)
 9    Z=0.0
10    PRINT *, 'The first 10 values of Z are:'
11  !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(I,J)
12  !$OMP DO SCHEDULE(DYNAMIC,4) ORDERED
13    DO I=1,N
14      DO J=1,M
15        Z(I) = Z(I) + X(I,J)*Y(J,I)
16      END DO
17  !$OMP ORDERED
18      IF(I<11) PRINT *, 'Z(',I,') =',Z(I)
19  !$OMP END ORDERED
20    END DO
21  !$OMP END DO
22  !$OMP END PARALLEL
23  END PROGRAM ORDERED
```

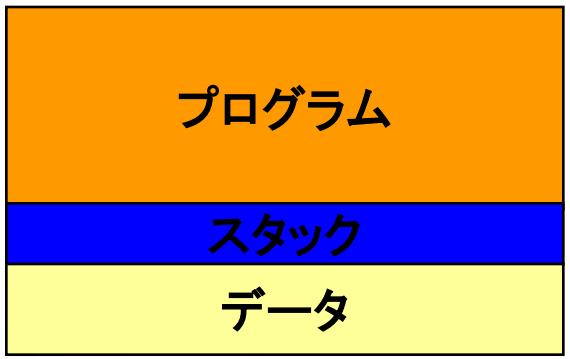
```
% ifort -O3 -openmp -openmp-report2 ordered.f90
ordered.f90(17) : (col. 6) remark: OpenMP multithreaded code generation
    for ORDERED was successful.
ordered.f90(12) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
ordered.f90(11) : (col. 6) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
% ifort -O3 -openmp ordered.f90
ordered.f90(12) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
ordered.f90(11) : (col. 6) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
% ifort -O3 -openmp -openmp-report0 ordered.f90
```

# OpenMPでのスレッド処理



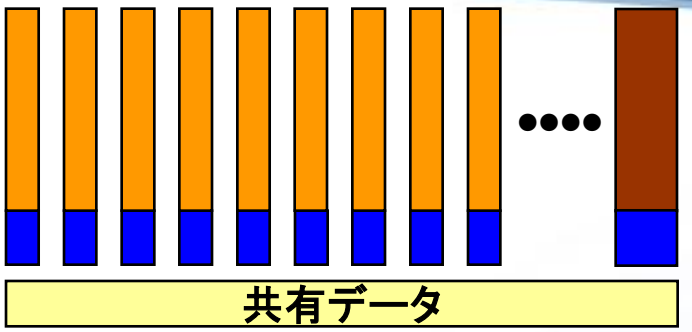

```
f = 3.0;
        …
#pragma omp parallel for
    shared (x,y,z,f) private (i,tmp)
for (i=0; i<n; i++) {
        tmp = x[i] + y[i];
          …..
        z[i] = tmp * f;
}
        …
        …
#pragma omp parallel for
    shared (a,b,c,f) private (i,tmp)
for (i=0; i<n; i++) {
        tmp = b[i] + c[i];
          …..
        a[i] = tmp / f;
}
        …
```

# OpenMPでの並列化



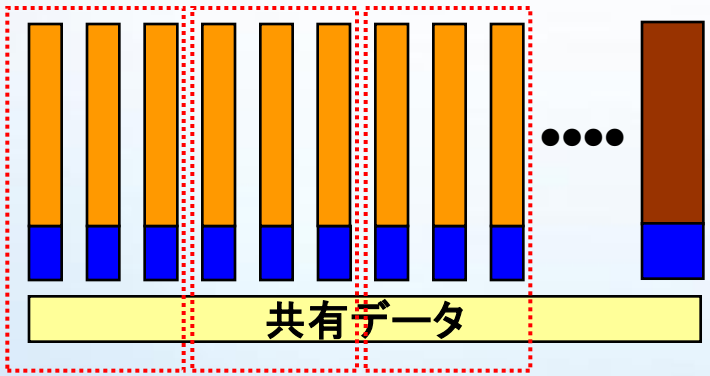

オリジナル

複数のタスクに分割




タスク、ローカルデータと共有データの明確化

分割されたタスクをグループ化




複数のタスクを分割し逐次処理部分の分離

並列処理のためのコード変換

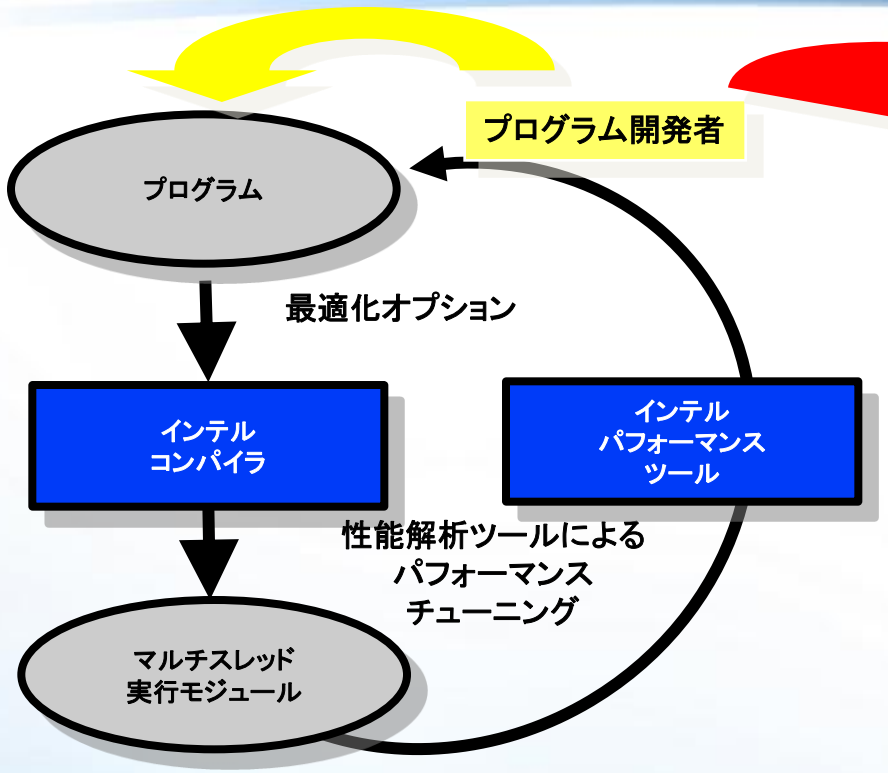
```
Program SPMD_Emb_Par ()
{
  Program SPMD_Emb_Par ()
  {
    Program SPMD_Emb_Par ()
    {
      Program SPMD_Emb_Par ()
      {
        TYPE *tmp, *func();
        global_array Data(TYPE);
        global_array Res(TYPE);
        int Num = get_num_procs();
        int id = get_proc_id();
        if (id==0) setup_problem(N, Data);
        for (int I= ID; I<N;I=I+Num){
            tmp = func(I, Data);
            Res.accumulate( tmp);
        }
      }
    }
  }
}
```

並列化されたプログラム

# OpenMP 適用のためのステップ

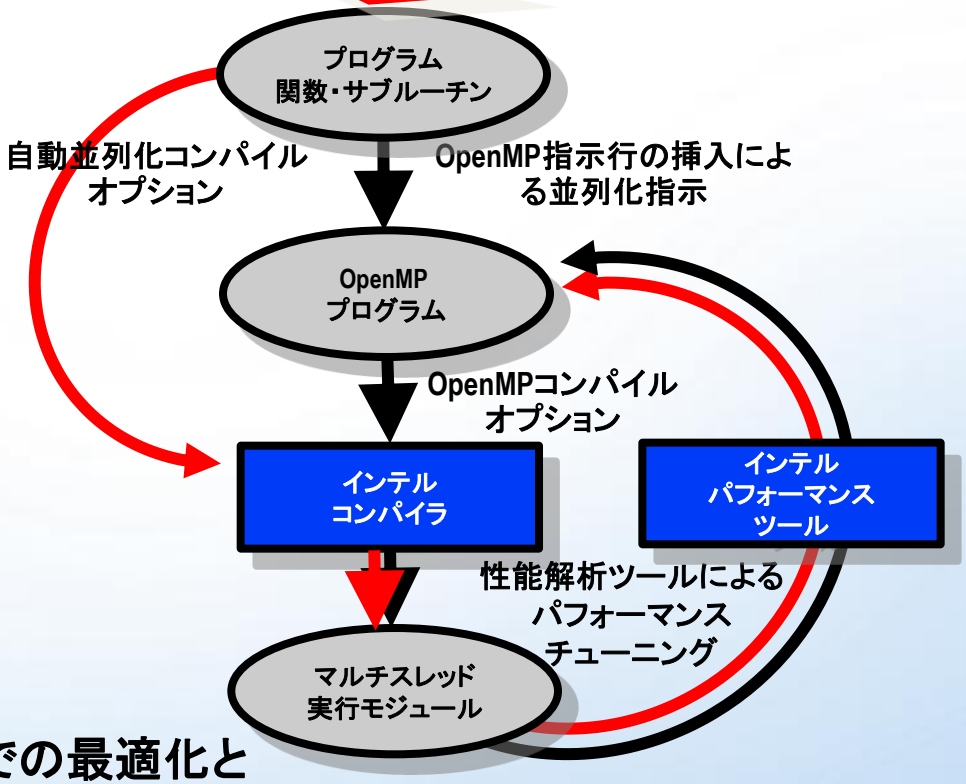
- パフォーマンスツールを使いプログラムの動作の詳細な解析を行う。
  - ホットスポットを見つけることが並列処理では必須
- ホットスポットに対してOpenMP 指示行の適用などを検討
  - データの依存関係などのために並列化出来ない部分などについては、依存関係の解消のために行うプログラムの変更を行う
  - この時、他のハイレベルの最適化手法（ソフトウエアパイプラインやベクトル化）などに影響を与えるときがあるので、この並列化による他のハイレベルの最適化の阻害は避ける必要がある。
  - 並列化の適用時と非適用時の性能を比較検討する必要がある
- 計算コアの部分について、もし可能であれば既にマルチスレッド向けに高度に最適化されているインテル MKL (Math Kernel Library) などを積極的に利用する

# OpenMP プログラムの開発フロー




シングルスレッドでの最適化と
プログラム性能の評価

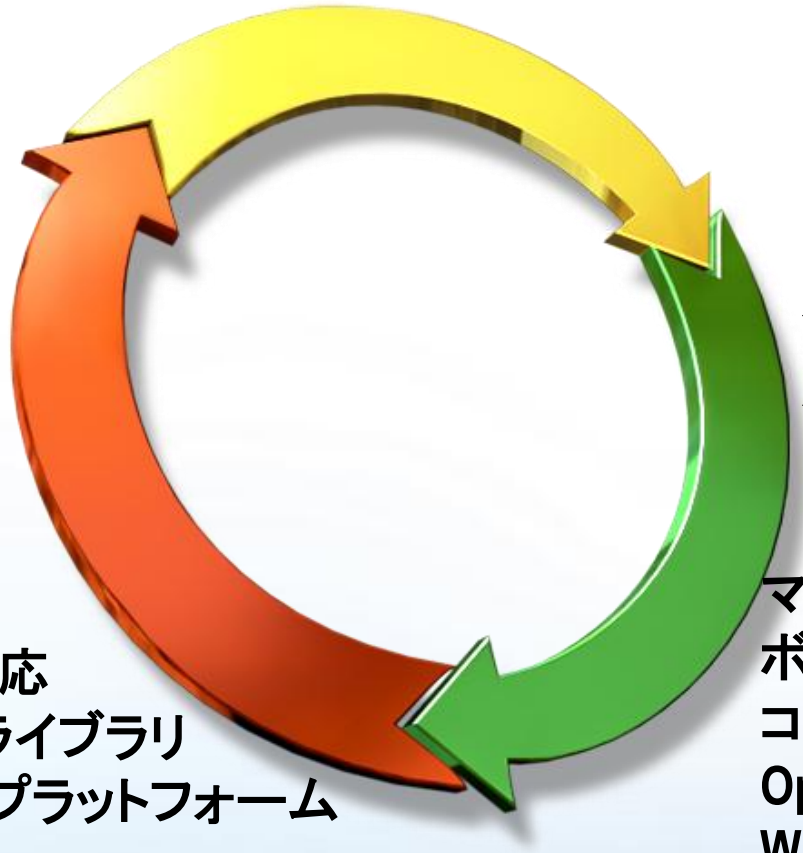
計算負荷の大きな関数やサブルーチンに対する自動並列処理やOpenMPの適用の検討




マルチスレッドでの最適化と
プログラム性能の評価

自動最適化と自動並列化
OpenMPサポート
マルチスレッドプログラミング対応
Windows & Linuxプラットフォーム

## コンパイラ



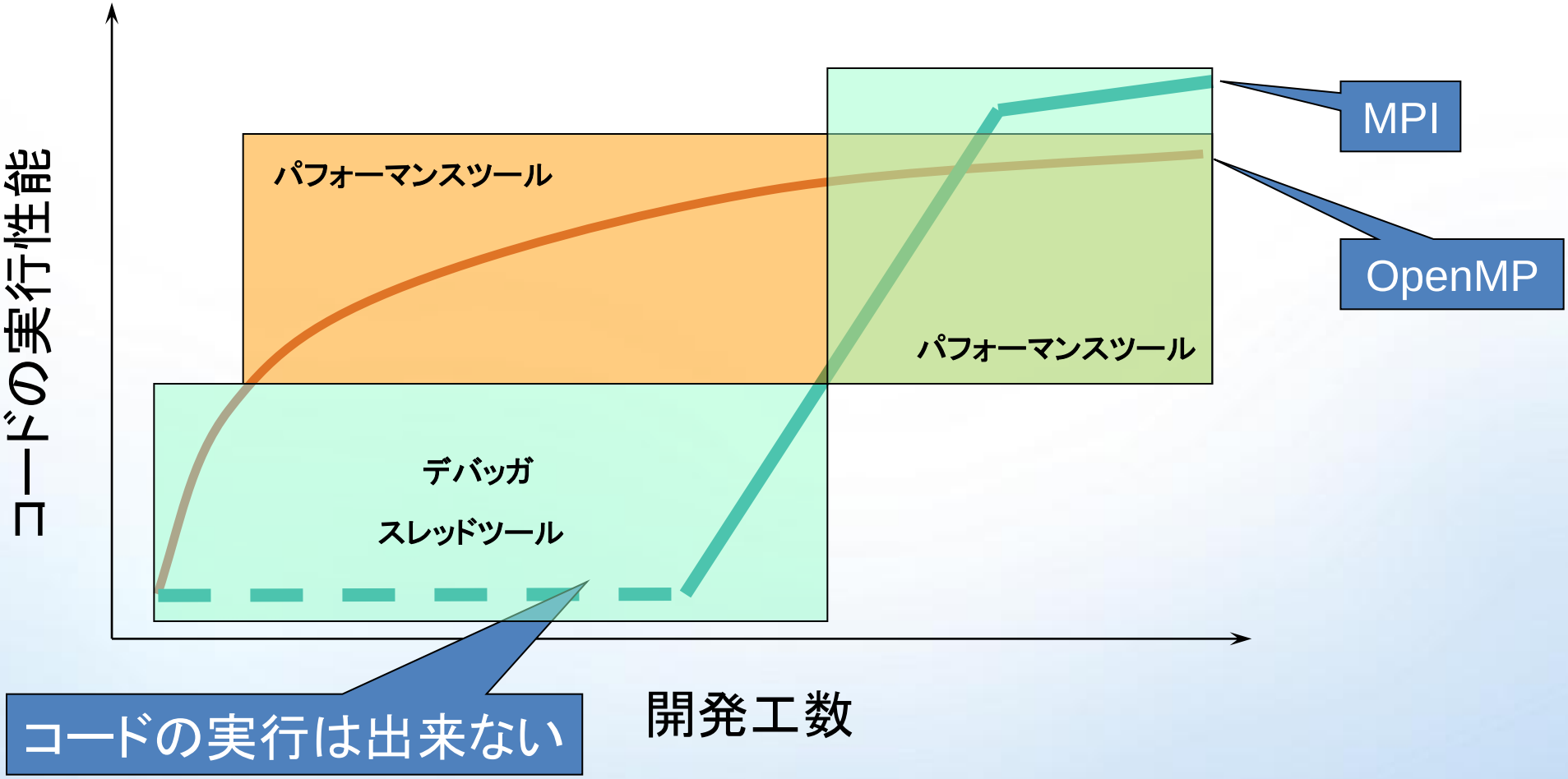
## スレッド性能解析
## スレッド対応デバッガー

マルチスレッド対応
ボトルネック、エラーの容易な検知
コンパイラと一体化した開発環境
OpenMPサポート
Windows & Linuxプラットフォーム

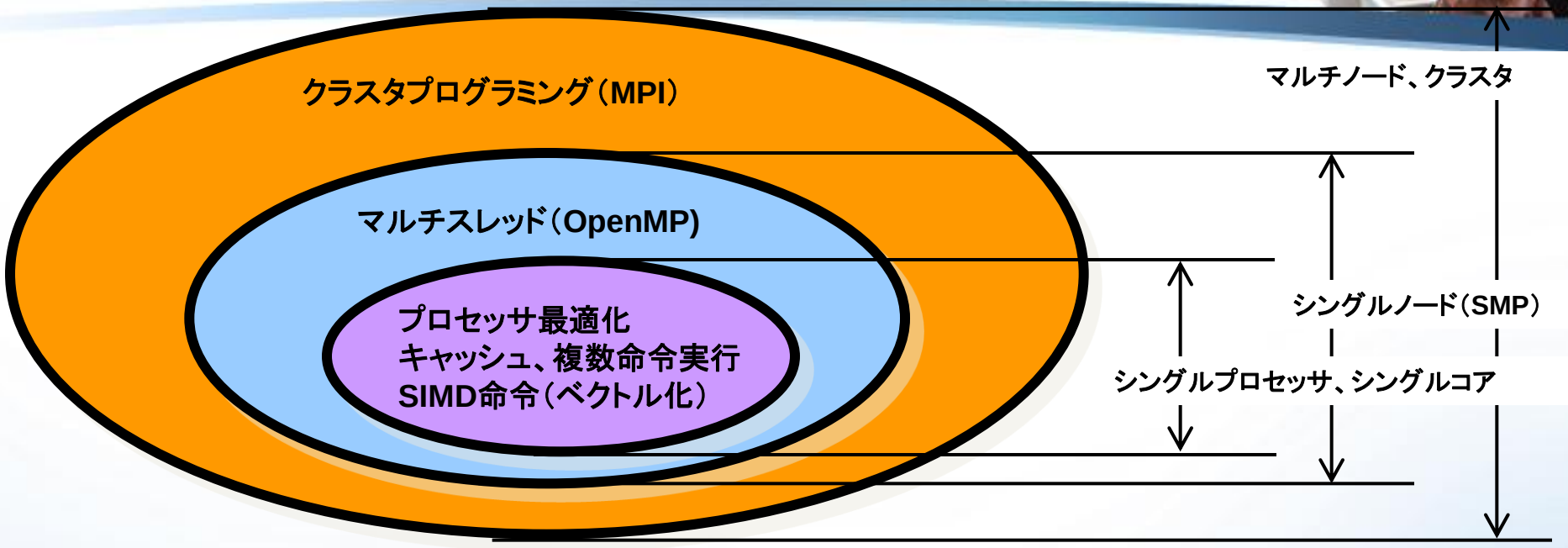
マルチスレッド対応
最適化・並列化ライブラリ
Windows & Linuxプラットフォーム

## ライブラリ

# OpenMP .vs. MPI



# プログラミング階層




```
do izone = 1, nzone
    ..........
    do j = 1 , jmax
        ..........
        do i = 1, imax
          ..........
```

ノード内、ノード間並列化

ノード内でのマルチスレッド並列化

プロセッサリソースの並列利用

OpenMPによるマルチスレッドプログラミング

# OpenMP APIのご紹介

# OpenMPでのプログラミング
## OpenMP APIのご紹介

次の5つのカテゴリに分類されるAPIについて、理解することが必要

1. 並列実行領域（Parallel Regions）構文
2. ワークシェアリング（Worksharing）構文
3. データ環境（Data Environment）構文
4. 同期（Synchronization）構文
5. 実行時関数/環境変数（Runtime functions/environment variables）

# 構文での代表的な指示行

- 並列実行領域（Parallel Regions）構文
  - #pragma omp parallel
- ワークシェアリング（Worksharing）構文
  - #pragma omp for
  - #pragma omp sections
  - #pragma omp single
- データ環境（Data Environment）構文
  - 指示行: threadprivate
  - 指示句: shared、private、lastprivate、reduction、copyin、copyprivate
- 同期（Synchronization）構文
  - 指示句: critical、barrier、atomic、flush、order、master
- 実行時関数/環境変数（Runtime functions/environment variables）

# OpenMPサンプル:Hello Worlds

- ４つのスレッドで実行すると、プログラムの出力は次のようになる。

```
sh-3.00$ cat -n hello.c
     1  #include "omp.h"
     2  main()
     3  {
     4  #pragma omp parallel
     5          {
     6          int ID = omp_get_thread_num();
     7          printf("hello(%d)", ID);
     8          printf("world(%d) ¥n", ID);
     9          }
    10          return 0;
    11  }
sh-3.00$ icc -openmp hello.c
hello.c(4) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
sh-3.00$ export OMP_NUM_THREADS=4
sh-3.00$ ./a.out
hello(0)world(0)
hello(3)world(3)
hello(1)world(1)
hello(2)world(2)
```

スレッドの実行順番の制御は、OSが行うため、ユーザは指定出来ません。

# OpenMP指示構文形式

- C/C++:
  - #pragma omp construct [Clause [clause]]
  - 指示構文名　指示句
- Fortran:
  - !$OMP directive [clause[[,] clause]…]
  - 指示行名　指示句
  - !$OMP directive [clause[[,] clause]…] &
  - !$OMP& [clause[[,] clause]…]　継続行の指定時(自由形式)
  - (Fortran77でのコメント形式である、C$OMPや*$OMPの指定も可能)

# 条件付きコンパイル

- OpenMP では、プログラムの互換性と管理のために条件付きコンパイルが可能

```
C23456789
C$     IAM = OMP_GET_THREAD_NUM()
```

- この場合、OpenMP のコンパイルが有効な場合には、C$ は22つの空白に置き換えられる。（先頭に !$ を指定しても同じ）

```
!$     IAM = OMP_GET_THREAD_NUM() + &
!$     index
```

- #ifdef での条件付きコンパイルも可能

```
#indef _OPENMP
IAM = OMP_GET_THREAD_NUM()
#endif
```

# 構造ブロック(C/C++)

- OpenMPの適用は全て構造ブロック単位で行われる
  - 構造ブロック: 上部に 1 つの開始点と下部に 1 つの終了点を持つブロック
  - 許可される唯一の "分岐" は、Fortran の STOP ステートメントと C/C++ の exit()

**構造ブロック**

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
more: res(id) = do_big_job(id);
  if(conv(res(id)) goto more;
}
printf(" All done ¥n");
```

**誤った構造ブロック**

```
if(go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
  if(conv(res(id)) goto done;
  goto more;
}
done: if(!really_done()) goto more;
```

# 構造ブロックの境界

- C/C++: 単一ステートメントまたはブラケット {} で囲まれたステートメントのグループ

```
#pragma omp parallel
{
  id = omp_thread_num();
  res(id) =    lots_of_work(id);
}
```

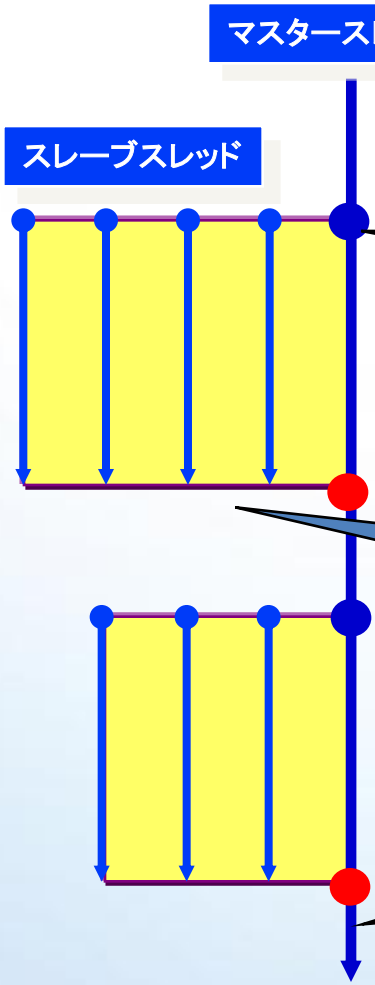
```
#pragma omp for
   for(I=0;I<N;I++){
     res[I] = big_calc(I);
     A[I] = B[I] + res[I];
   }
```

- Fortran: 単一ステートメントまたはディレクティブ/終了ディレクティブのペアで囲まれたステートメントのグループ

```
!$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
!$OMP END PARALLEL
```

```
!$OMP PARALLEL DO
   do I=1,N
   res(I)=bigComp(I)
   end do
!$OMP END PARALLEL DO
```

# ‘Fork-Join’ モデルでの並列処理




最初にマスタースレッドが起動されてプログラムの実行を開始し、プログラムを逐次的に処理

マスタースレッドの実行が並列化指示文の部分に到達すると、スレーブスレッドと呼ばれるスレッドを生成（スレッドを Fork する）し、分割されたプログラムのタスクを並列に処理

このマスタースレッドとスレーブスレッドの処理は、プログラム中での並列実行領域の終了点に到達すると終了

終了時には、全スレッドが各自の処理を終了するまで、先に終了したスレッドは、全てのスレッドの終了を待つ（同期処理）

全スレッドが完了した時点（スレッドを join する）で、プログラムの実行処理は再びマスタースレッドだけが逐次的にプログラムを処理

# OpenMPでのプログラミング
## OpenMP APIのご紹介

次の5つのカテゴリに分類されるAPIについて、理解することが必要

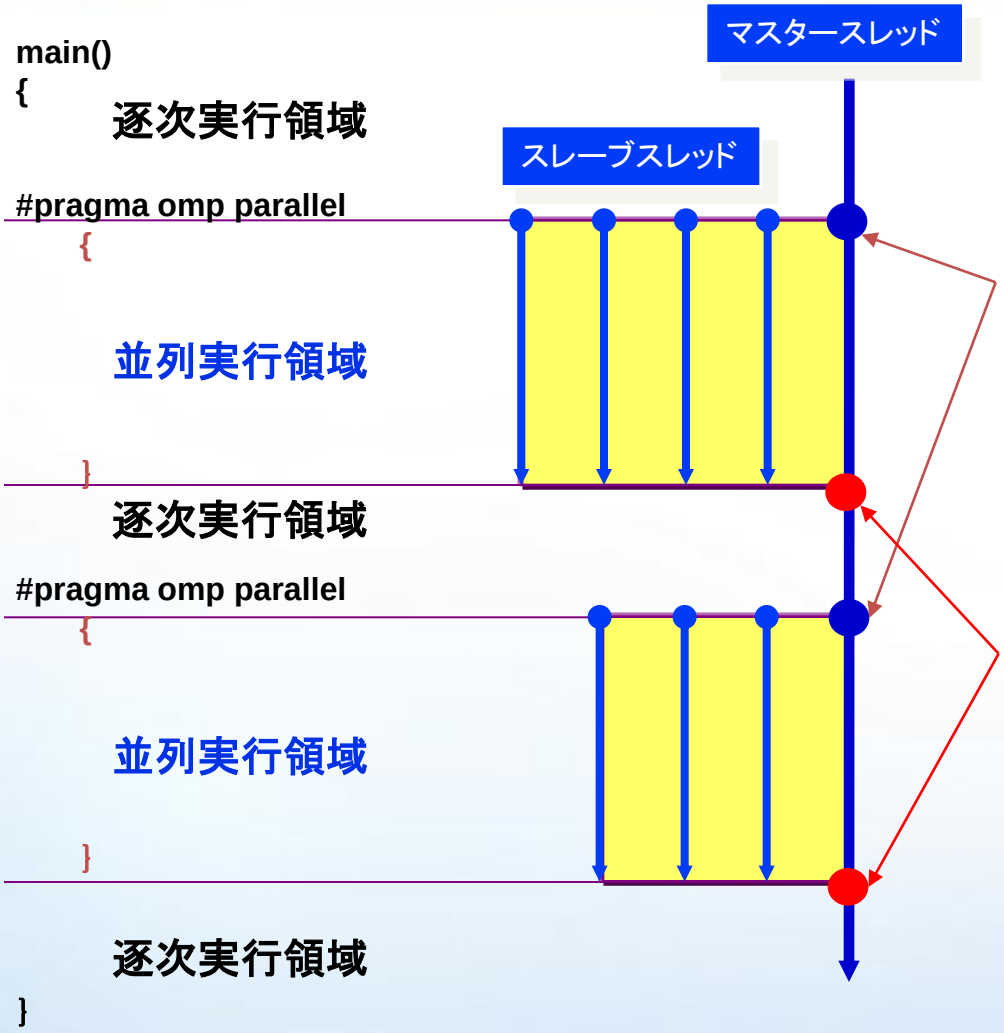
1. **並列実行領域（Parallel Regions）構文**
2. ワークシェアリング（Worksharing）構文
3. データ環境（Data Environment）構文
4. 同期（Synchronization）構文
5. 実行時関数/環境変数（Runtime functions/environment variables）

# 並列実行領域（Parallel Regions）構文

- 並列に実行する領域（以下、並列実行領域）を指示し、その領域内で各スレッドにその領域内の計算を分散することを指示する

```
Fortran:   !$OMP PARALLEL
             block
               !$OMP END PARALLEL
C/C++:     #pragma omp parallel
             {
             block
             }
```

# 並列実行領域の設定




Fork と Join 操作により、逐次実行と並列実行を繰り返す

【 Fork 】複数のスレーブスレッドを生成し、並列実行を開始します。

【 Join 】他のスレッドの処理の完了を待つための同期処理を行います。全てのスレッドの動作が完了するとスレーブスレッドは終了し、マスタースレッドのみが実行を継続します。

# OpenMPでのプログラミング
## OpenMP APIのご紹介

次の5つのカテゴリに分類されるAPIについて、理解することが必要

1. 並列実行領域（Parallel Regions）構文
2. **ワークシェアリング（Worksharing）構文**
3. データ環境（Data Environment）構文
4. 同期（Synchronization）構文
5. 実行時関数/環境変数（Runtime functions/environment variables）

# ワークシェアリング構文

- 並列実行領域は、全スレッドで実行されるため、そのままでは、並列化での速度向上は得られない
- 並列実行領域での‘ワークロード’を各スレッドに分担させることが必要
  - ワークシェアリング構文では、このようなワークロードのスレッドへの分担を指示

Fortran:

```
!$OMP DO [clauses]
    do loop
[ !$OMP END DO ]
```

C/C++:

```
#pragma omp for [clauses]
            for loop
```

# ワークシェアリング構文

1. 逐次コード

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

2. OpenMP 並列実行領域
   - 各スレッドにつけられた番号などを取得し、陽的にワークロードの分散を図る

```
#pragma omp parallel
    {
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;I<iend;i++){
                    a[i] = a[i] + b[i];}
    }
```

3. OpenMP 並列実行領域とワークシェアリング構文

```
#pragma omp parallel
#pragma omp for schedule(static)
        for(i=0;I<N;i++){ a[i] = a[i] + b[i];}
```

```
#pragma omp parallel for schedule(static)
        for(i=0;I<N;i++){ a[i] = a[i] + b[i];}
```

# ワークシェアリング構文

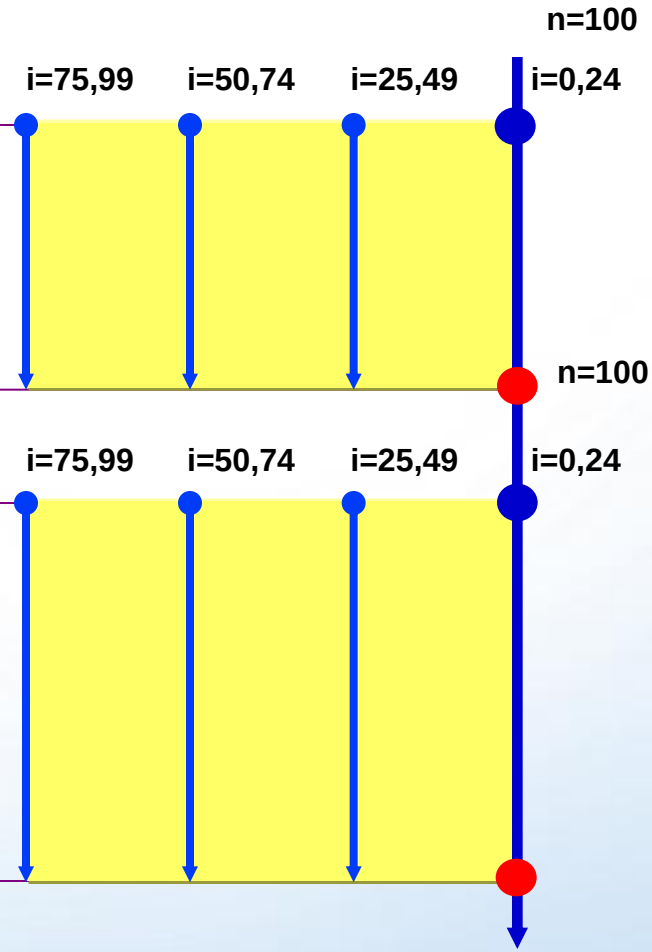
逐次実行するプログラムブロック

```
#pragma omp parallel [clause[[,]clause]…]
{
#pragma omp for
for(int i=0;i<n;i++) {
        ワークシェアリング：forループの繰り返しを各
    }   スレッドに分割し 並列に実行します。
}
```

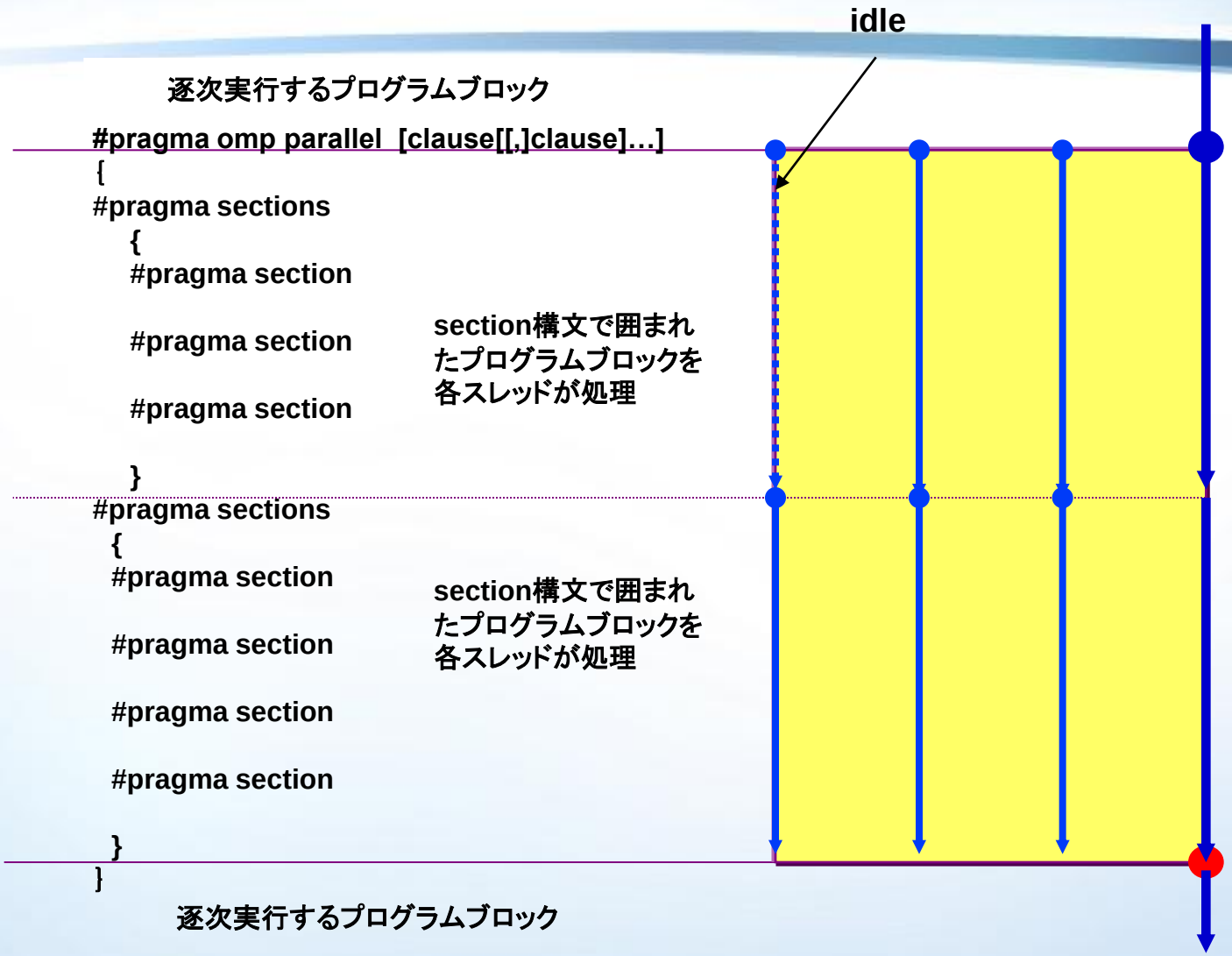
逐次実行するプログラムブロック

```
#pragma omp parallel [clause[[,]clause]…]
{
#pragma omp for
for(int i=0;i<n;i++) {

        ワークシェアリング：forループの繰り返しを各
        スレッドに分割し 並列に実行します。


    }
}
```

逐次実行するプログラムブロック



# ワークシェアリング構文

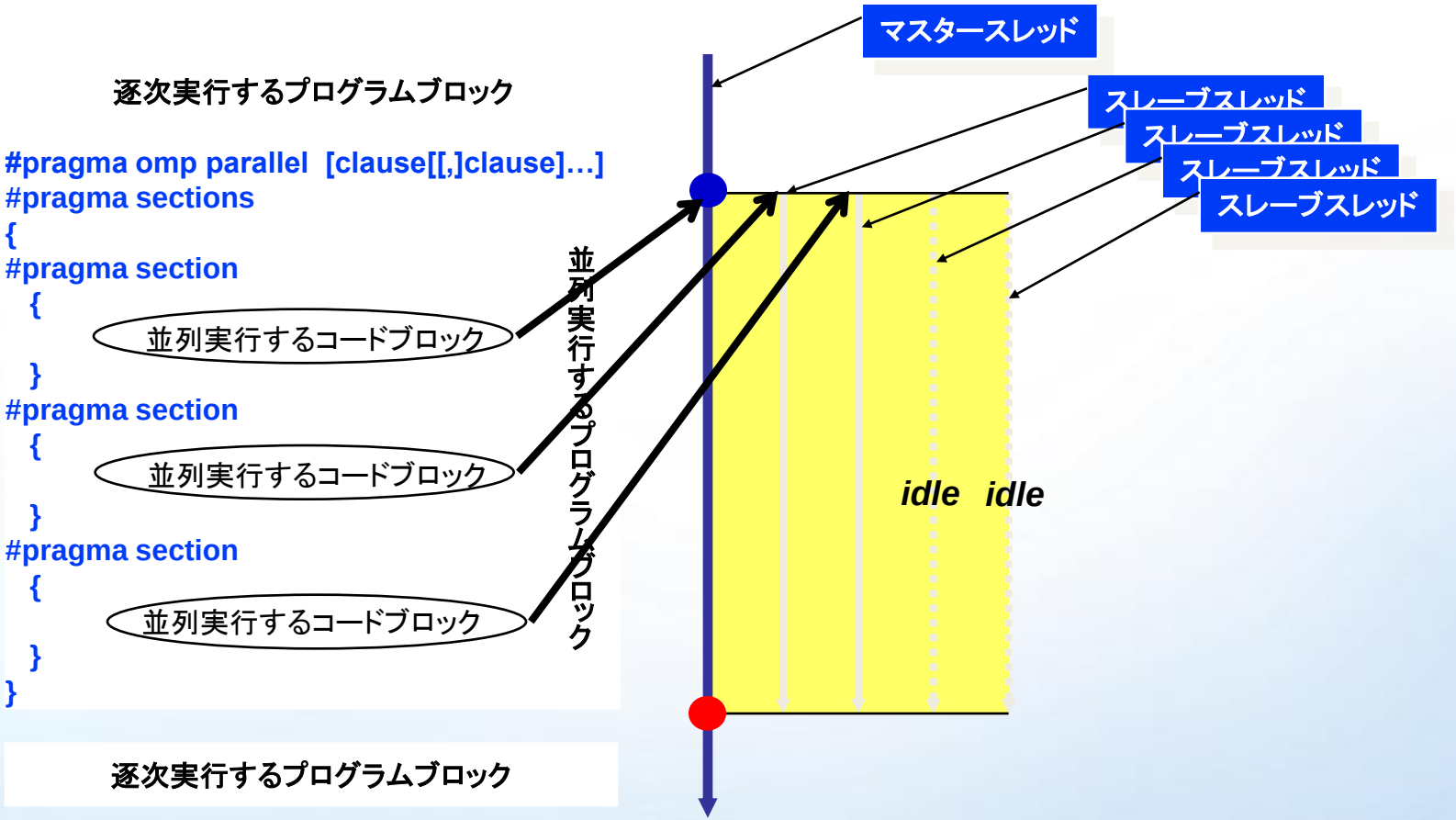


# SECTIONS構文

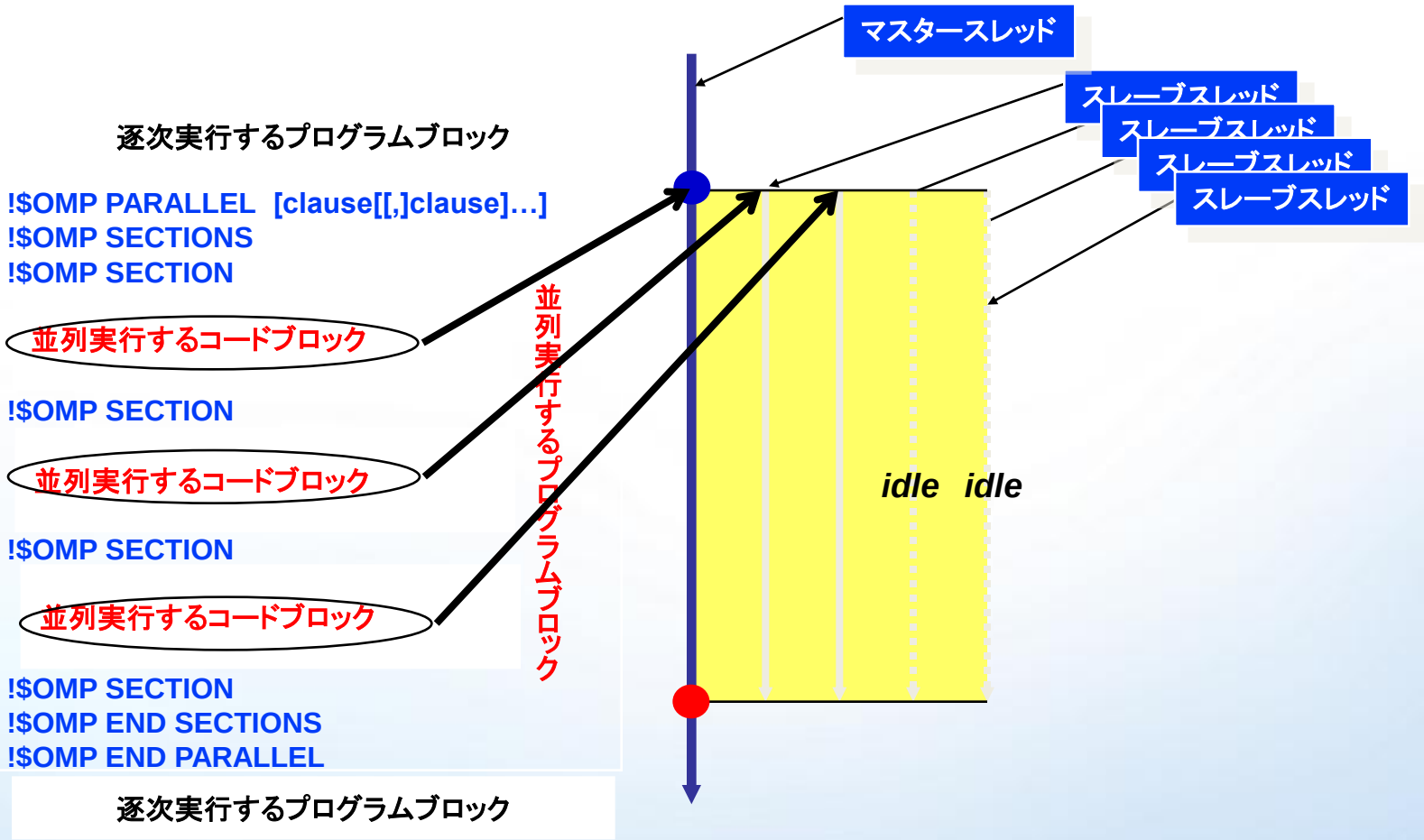
```
 PROGRAM VEC_ADD_SECTIONS
      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N)
!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO
!$OMP PARALLEL SHARED(A,B,C), PRIVATE(I)
!$OMP SECTIONS
!$OMP SECTION
      DO I = 1, N/2
         C(I) = A(I) + B(I)
      ENDDO
!$OMP SECTION
      DO I = 1+N/2, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
      END
```

```
#include <omp.h>
#define N     1000
main ()
{
int i;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N/2; i++)
      c[i] = a[i] + b[i];
    #pragma omp section
    for (i=N/2; i < N; i++)
      c[i] = a[i] + b[i];
    }  /* end of sections */
  }  /* end of parallel section */
}
```

# sections構文の実行（C/C++)



# SECTIONS構文の実行（Fortran)



# ワークシェアリング構文の適用条件

- ループの反復回数がループの実行を開始される時に明らかになっている必要がある。従って、while ループなどの並列化は通常は出来ない。
- ループの並列化に際して、十分な計算負荷がそのループにあることが必要

# ワークシェアリング構文の適用条件

- ループ内の演算は、相互に独立である必要がある。
  - 各反復計算の実行順序が計算の整合性に影響を与えないことが必須
  - ループの各反復が他の反復計算の結果を参照したりする場合には並列化は出来ない

計算式

```
for (i=1;i<= n; i++){
a[インデックス計算式1] = ・・・・・
・・・・・= a[インデックス計算式] ・・・・・・;
 }
```

ここでは、インデックス計算式1の値は、各反復時に異なった値となることが必要

インデックス計算式1とインデックス計算式2の値が異なる場合には、参照関係に依存性がある

# ワークシェアリング構文の適用条件（続き）

- 配列の総和を計算するような場合には、実際には各反復計算の結果は相互依存する
  - ループ構造が明確な場合などは、この見かけの依存性を排除することも可能（REDUCTION構文）
- ループ内に外部関数の呼び出しがあるような場合には、外部関数の呼び出しによる依存関係を明確にして、関数のデータのスコーピングを行う必要がある

# for/do 構文:schedule 句

schedule句は、ループ反復がどのようにスレッドにマップされるかを指定

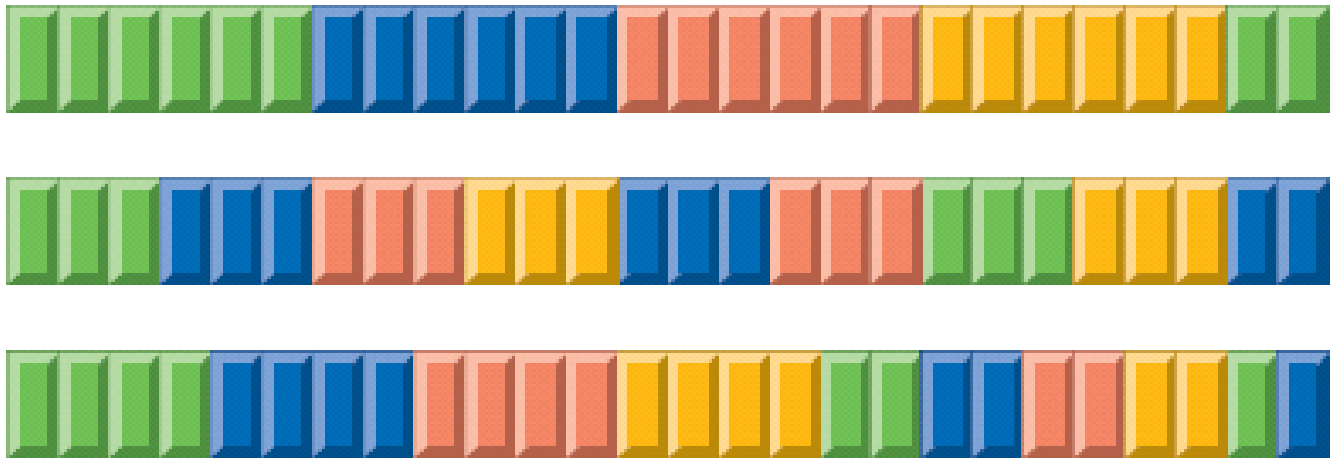
（例） !$OMP DO SCHEDULE(DYNAMIC,4)

| TYPE | 説明 |
|---|---|
| STATIC | 繰り返し数は、chunkで指定したサイズに分割されます。分割された部分は、スレッドの番号順でラウンドロビン形式（全てのスレッドに対して、平等に巡回的に割り当てる）で各スレッドに静的に割り当てられます。 chunkの指定が無い場合には、均等に分割されます。スケジュールの指定が無い場合には、static として、処理がなされます。 |
| DYNAMICS | 繰り返し数は、オプションで指定可能なchunkで指定したサイズに分割され、各スレッドは割り当てられた繰り返し部分を終了すると繰り返し数の次のセットが動的に割り当てられます。オプションのchunkの指定が無い場合には、Defaultでは、1が設定されます。 |
| GUIDED | 繰り返し数は、各反復時に残りの反復数とスレッド数に基いて決定されます。<br>各スレッドに割り当てられる反復数は、chunkに指定された数にまで、コンパイラの実装方法に従って、順次減少します。オプションのchunkの指定が無い場合には、デフォルトでは、1が設定されます。比較的、低いオーバーヘッドでのロードバランスの向上を図ることが可能です。 |
| RUNTIME | スケジュールの指定を実行時に指定します。オプションのtype及びchunkは環境変数OMP_SCHEDULEの設定によって実行時に決定されます。環境変数OMP_SCHEDULEがない場合には、SCHEDULE(STATIC)が設定されます。 |

# ワークロードの分配・分散の指定

```
      ……
      parameter (N=26,NUM_THREADS=4)
      call omp_set_num_threads(NUM_THREADS)
!$omp parallel do schedule( type [ , chunk ] )
      do i =1, N
        ……..
      end do
      ……
```

- この場合に各反復(ループ)をどのスレッドが処理するかを模式的に示しています。並列処理における各スケジュールオプションでのループのスレッドへの割り当てを色別で示します。

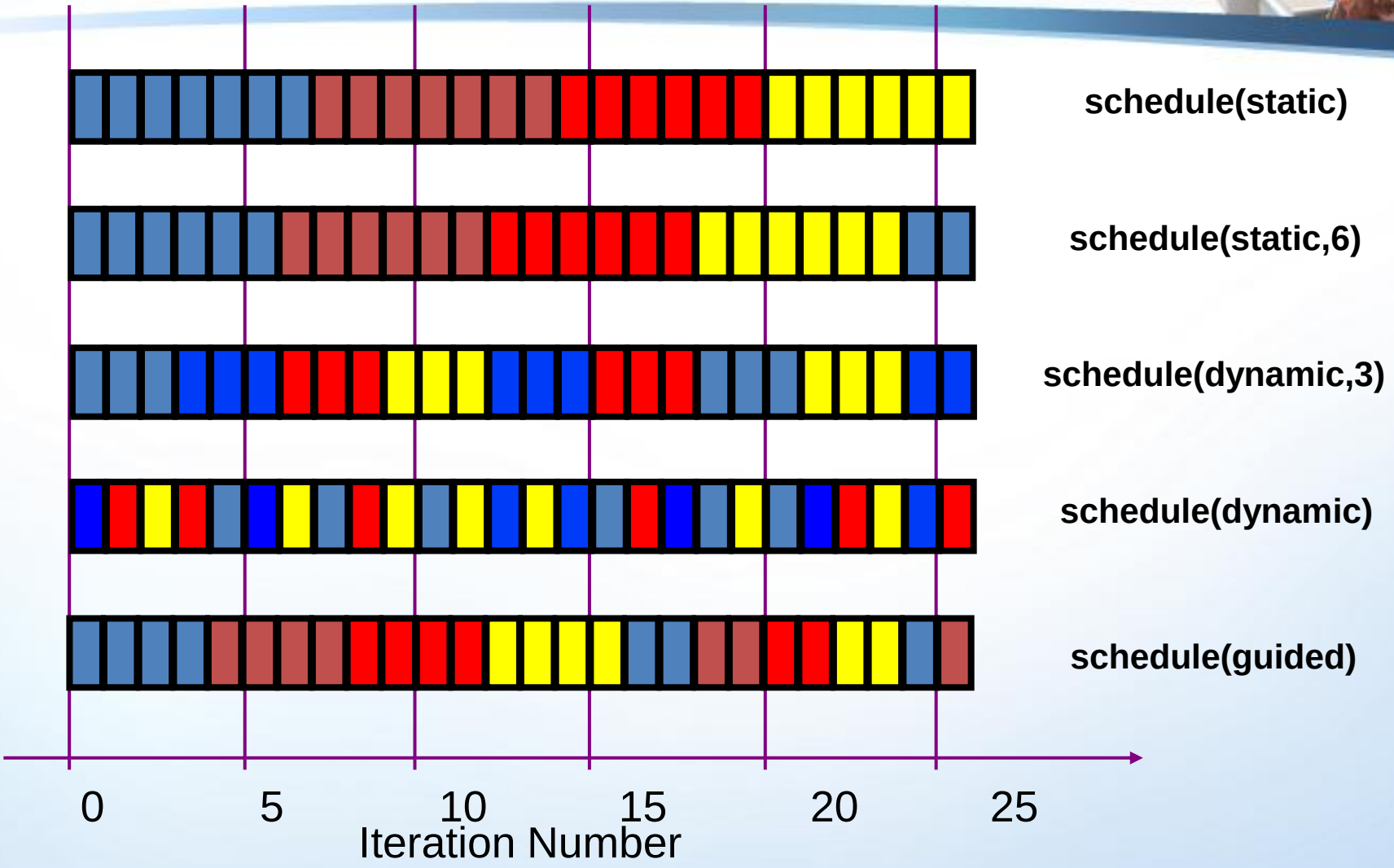


schedule(static,6)
4スレッド

schedule(dynamic,3)
4スレッド

schedule(guided)
4スレッド

# ワークロードの分配・分散の指定(追加)



# 並列/ワークシェアの組み合わせ

- 並列実行領域とワークシェアを結合して記述することも可能
  - parallel for / PARALLEL DO構文
  - parallel sections / PARALLEL SECTIONS構文
  - PARALLEL WORKSHARE 構文(Fortranのみ）

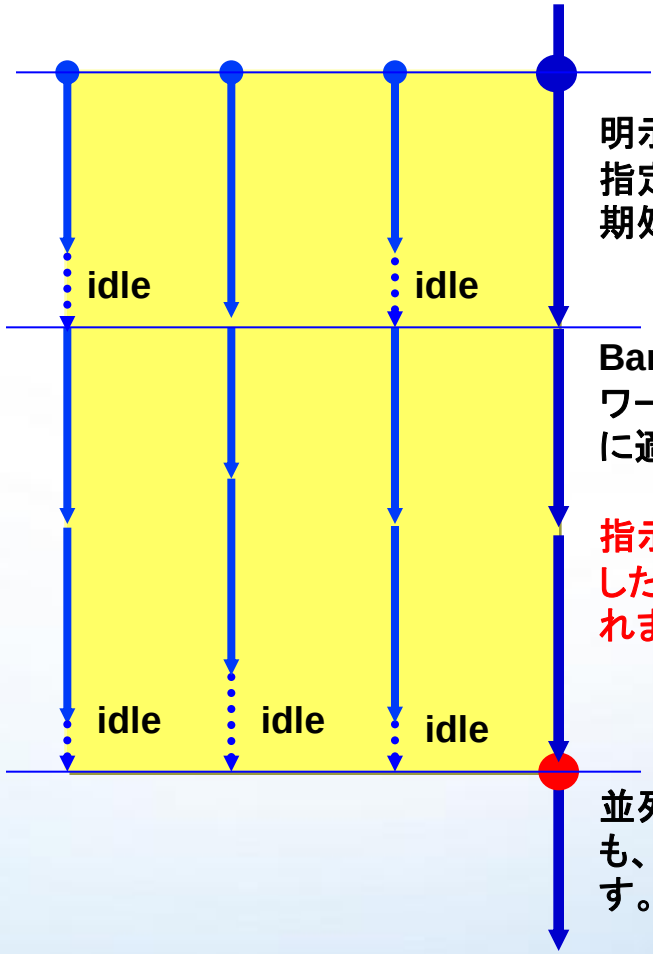
```
double   res[MAX];
int i;
#pragma omp parallel
{
#pragma omp for
    for (i=0;i< MAX; i++)  {
       res[i] = huge();
       }
}
```

等価

```
double   res[MAX];
int i;
#pragma omp parallel for
    for (i=0;i< MAX; i++)  {
         res[i] = huge();
       }
```

# ワークシェアリングでの同期処理

```
        逐次実行するプログラムブロック

#pragma omp parallel
{
#pragma omp barrier
  #pragma omp for
  {
        並列実行するプログラムブロック
  }
  #pragma omp for nowait
  {
        並列実行の終了時の同期処理
        なしのプログラムブロック
  }
  #pragma omp for
  {
        並列実行するプログラムブロック
  }
}
        逐次実行するプログラムブロック
```




明示的なbarrier指示句の指定による各スレッドの同期処理の適用

Barrierによる同期処理がワークシェア構文の終了時に適用されます。

指示句としてnowaitを指定した場合、同期処理は行われません。

並列実行領域の終了時にも、同期処理が適用されます。

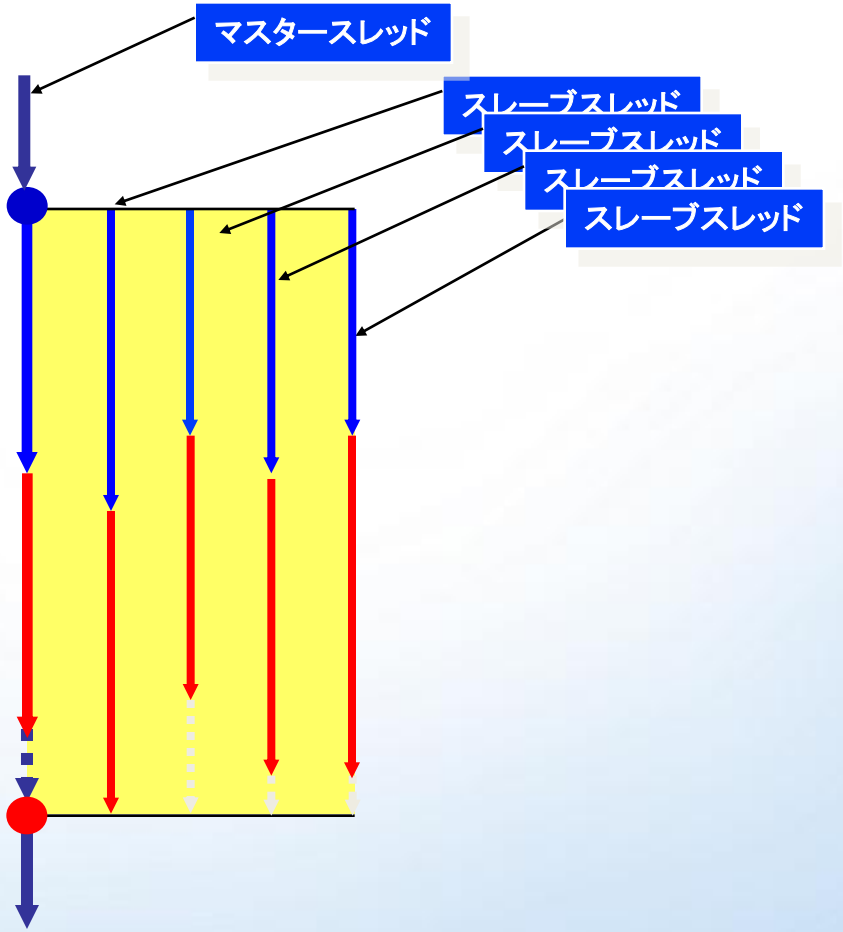
# nowait指定時のスレッド動作(C/C++)

逐次実行するプログラムブロック

```
#pragma omp parallel
{
  #pragma omp for nowait
  {
        並列実行するワークシェアリング
              コードブロック

  }
  #pragma omp for
  {

        並列実行するワークシェアリング
              コードブロック

  }
}
```

逐次実行するプログラムブロック

並列実行するプログラムブロック



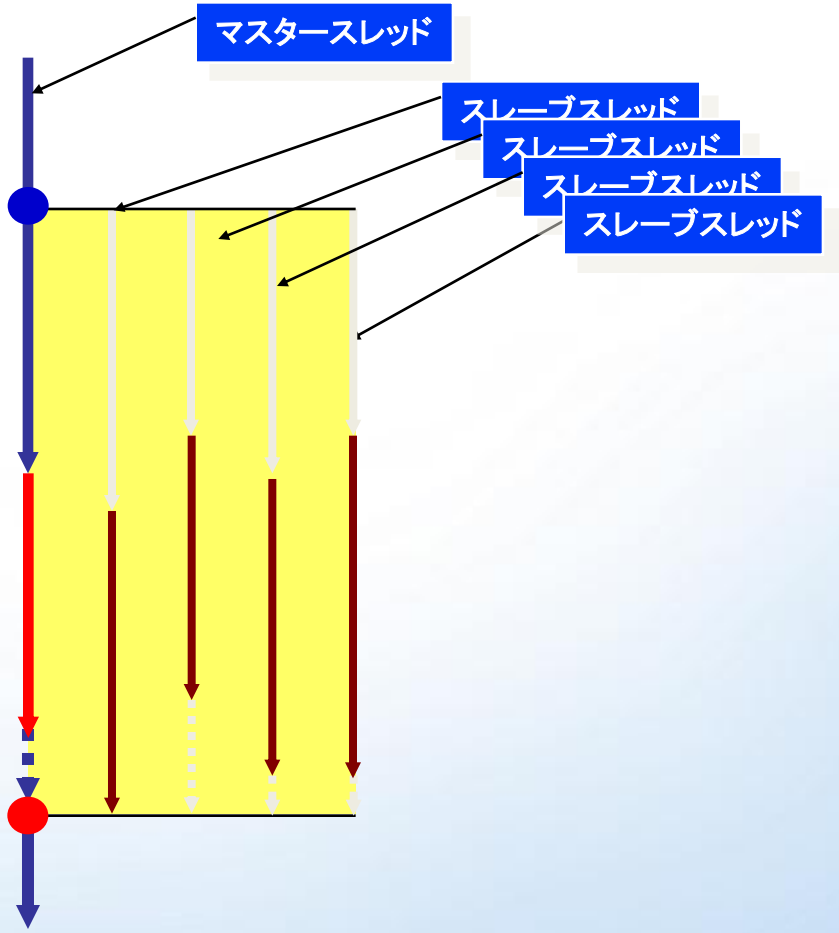
# NOWAIT指定時のスレッド動作(Fortran)

逐次実行するプログラムブロック

```
!$OMP PARALLEL
!$OMP DO
DO I = 1, N
並列実行するコードブロック
END DO

!$OMP END DO NOWAIT
!$OMP DO
DO I = 1, N
並列実行するコードブロック
END DO

!$OMP END DO
!$OMP END PARALLEL
```

逐次実行するプログラムブロック

並列実行するプログラムブロック



# 並列実行領域の動的範囲

- OpenMP 構文は複数のソースファイルに分割することが出来ます。
  - 並列実行領域内での指示行の指定は、プログラムユニット間でも可能です。

**ソースファイル main.c**

```
#include <omp.h>
void main()
{
  int num_threads;
#pragma omp parallel
  num_threads = omp_get_num_threads();
  printf(" num_threads = %d ¥n",num_threads);
  #pragma omp parallel
  {
        whoami ();
  }
  printf("All Done ¥n");
        whoami ();
}
```

構文的範囲

**ソースファイル whoami.c**

```
#include <omp.h>
void whoami ()
{
  int iam;
  iam = omp_get_thread_num();
  #pragma omp critical
  printf("Hello from %d ¥n",iam);
}
```

親無し指示文

この例では、main.c内での2番目のWHOAMIの呼び出しは、逐次実行領域からになりますので、whomai.c 内の並列化指示行は無視され、通常の逐次処理プログラムとして、処理されます。

# 並列実行領域の動的範囲

```
 1  #include <omp.h>
 2  void whoami ()
 3  {
 4    int iam;
 5    iam = omp_get_thread_num();
 6    #pragma omp critical
 7    printf("Hello from %d ¥n",iam);
 8  }
 9
10  int main()
11  {
12    int num_threads;
13    #pragma omp parallel
14    num_threads = omp_get_num_threads();
15    printf(" num_threads = %d ¥n",num_threads);
16    #pragma omp parallel
17    {
18          whoami ();
19    }
20    printf("All Done ¥n");
21
22          whoami ();
23
24    return 0;
25  }
```

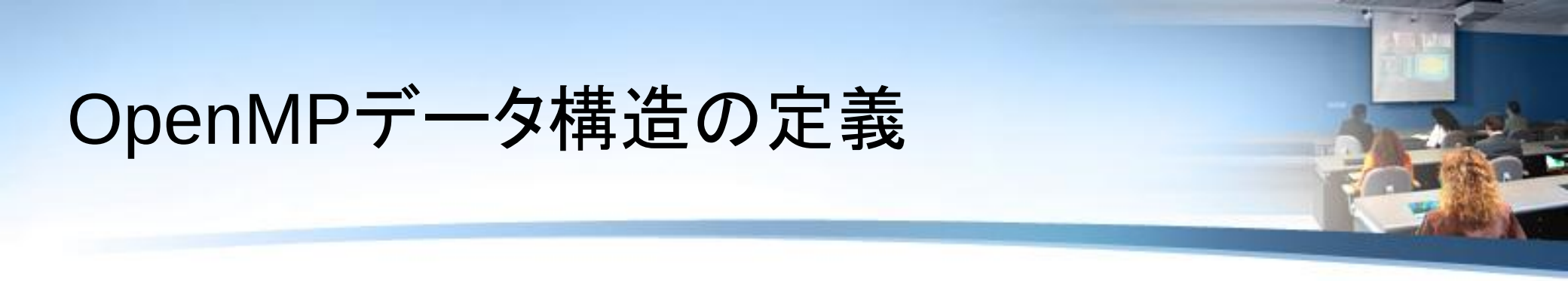
```
% icc -openmp sample.c
sample.c(13) : (col. 3) remark: OpenMP DEFINED
REGION WAS PARALLELIZED.
sample.c(16) : (col. 3) remark: OpenMP DEFINED
REGION WAS PARALLELIZED.
% a.out
 num_threads = 4
Hello from 0
Hello from 3
Hello from 2
Hello from 1
All Done
Hello from 0
```

# OpenMPでのプログラミング
## OpenMP APIのご紹介

次の5つのカテゴリに分類されるAPIについて、理解することが必要

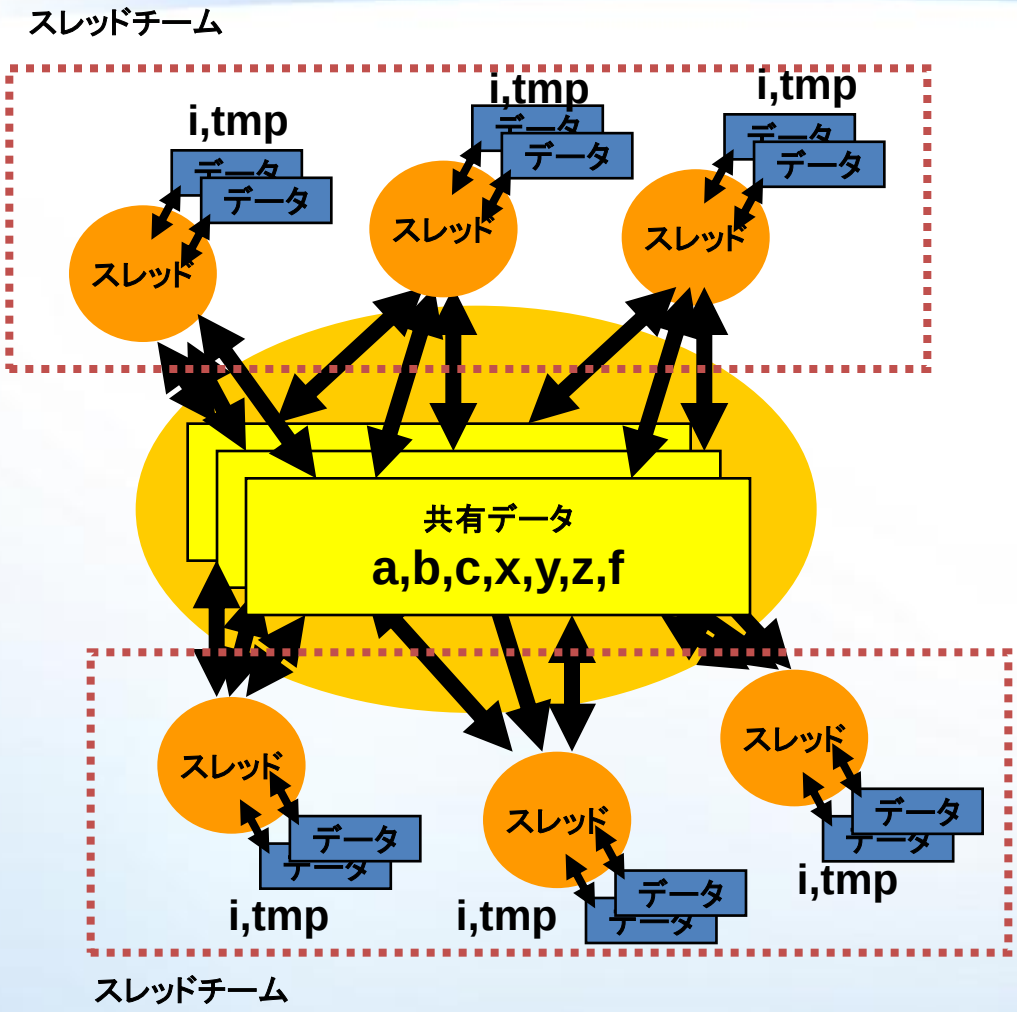
1. 並列実行領域（Parallel Regions）構文
2. ワークシェアリング（Worksharing）構文
3. **データ環境（Data Environment）構文**
4. 同期（Synchronization）構文
5. 実行時関数/環境変数（Runtime functions/environment variables）

# OpenMPデータ構造の定義



- 共有メモリ並列化API
  - OpenMPは共有メモリでの並列化APIであり、全てのデータは、スレッド間で共有されることを前提としている
  - 実際には、スレッドがそれぞれの独自のデータとして保持するもの（PRIVATE：プライベートデータ )とスレッド間で共有するデータ（SHARED：共有データ)を明確に指定する必要がある
- OpenMP指示構文
  - これらのデータの格納属性をOpenMPの指示構文で指定する必要がある

# OpenMPデータ属性の定義



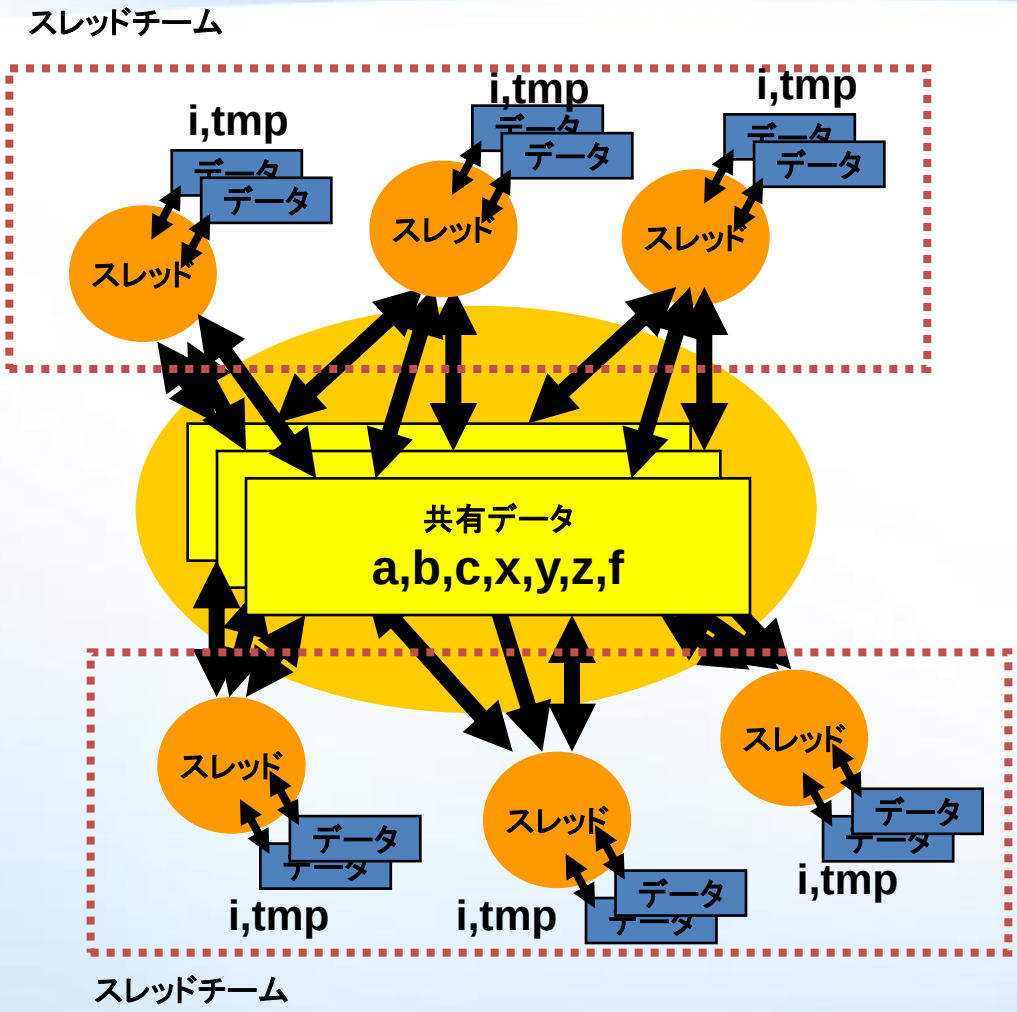

**共有データ**

- 全てのスレッドチーム中のスレッドによって共有されるデータリソース
- したがって、いずれかのスレッドでこのデータを変更した場合、並列実行領域内の全てのスレッドはその変更されたデータを利用することになる

**プライベートデータ**

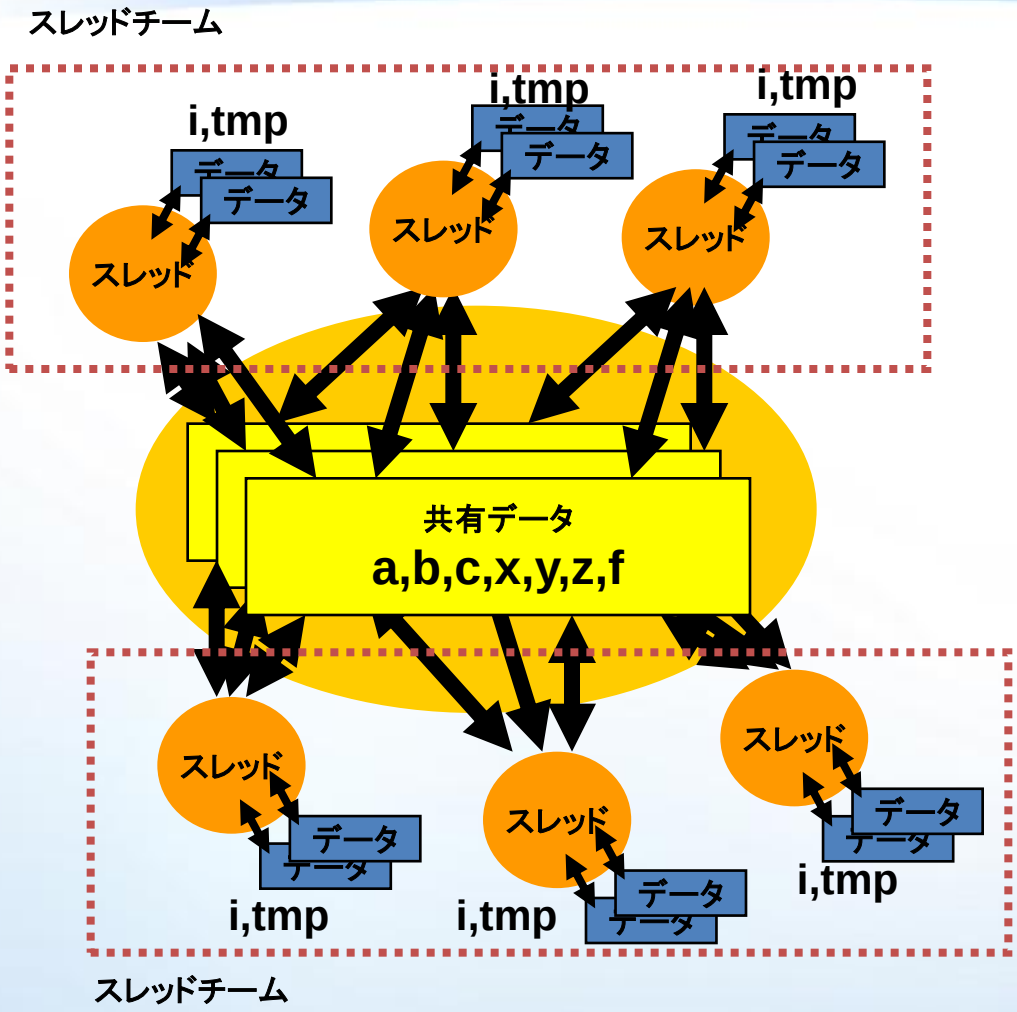
- 各スレッドがそれぞれ個別のリソースとしてその領域を確保
- したがって、スレッド中でそれらのデータが変更されても他のスレッドのデータにはその変更は反映されない

# OpenMPデータ構造の定義(C/C++)




```
f = 3.0;
        …
#pragma omp parallel for
    shared (x,y,z,f) private (i,tmp)
for (i=0; i<n; i++) {
        tmp = x[i] + y[i];
          …..
        z[i] = tmp * f;
}
        …
        …
#pragma omp parallel for
    shared (a,b,c,f) private (i,tmp)
for (i=0; i<n; i++) {
        tmp = b[i] + c[i];
          …..
        a[i] = tmp / f;
}
        …
```

# OpenMPデータ構造の定義(Fortran)




```
  f = 3.0
!$omp parallel do
!$omp& shared (x,y,z,f) private (i,tmp)
      do i = 1, n
        tmp = x(i) + y(i)
              ......
        z(i) = tmp * f
        end do
              ......
!$omp parallel do
!$omp& shared (a,b,c,f) private (i,tmp)
      do i = 1, n
        tmp = b(i) + c(i)
              ......
        a(i) = tmp / f
      end do
              ......
```

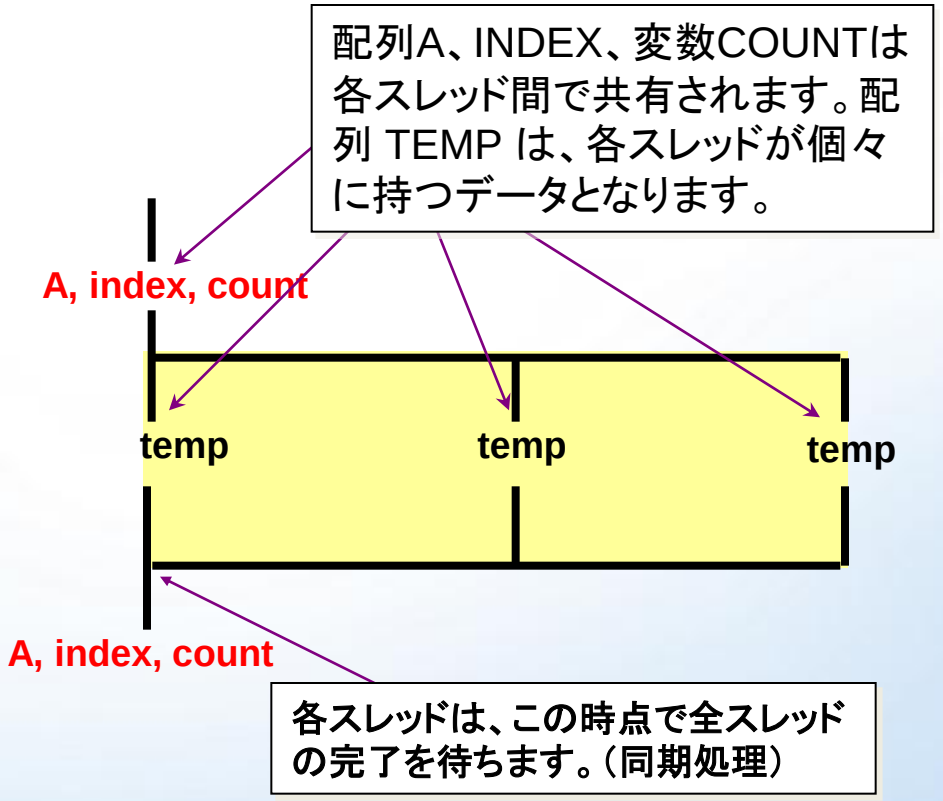
# データ共有

ソースファイル main.f

```
      PROGRAM OMP
      COMMON /WORK/ A(10)
      REAL*8 A
      INTEGER INDEX(10)
!$OMP PARALLEL
      CALL SORT(INDEX)
!$OMP END PARALLEL
      WRITE (6,*) INDEX(1)
      STOP
      END PROGRAM OMP
```

ソースファイル sort.f

```
      SUBROUTINE SORT(INDEX)
      COMMON /WORK/ A(10)
      REAL*8 A
      INTEGER INDEX(*)
      REAL*8 TEMP(10)
      INTEGER COUNT
      SAVE COUNT
       .......
       .......
      RETURN
      END
```



# データ環境:デフォルトの格納属性

- 共有メモリ・プログラミング・モデル
  - ほとんどの変数はデフォルトの設定では共有データとして取り扱う
- グローバル変数はスレッド間で共有される
  - Fortran: COMMON ブロック、SAVE 変数、MODULE 変数
  - C/C++: ファイルスコープ変数、static
- しかし、すべての変数は共有されない
  - 並列実行領域から呼び出されるサブプログラム内のスタック変数はプライベート
  - 並列実行領域でアロケートされる変数はプライベート
  - ステートメント・ブロック内の自動変数はプライベート

# データ環境:格納属性一覧

| | SHARED属性<br>全スレッドがアクセス可能 | PRIVATE属性<br>スレッド毎に独立にデータを割り当て |
|---|---|---|
| **グローバル(大域)変数**<br>**全てのプログラムユニット内でアクセスが可能な変数** | •static 変数<br>•ファイルスコープ変数 | •指示行でのthreadprivate指定 |
| **ローカル(局所)変数**<br>**プログラムユニット内だけでアクセスが可能な変数** | •OpenMPでのデフォルト<br>•指示行でのshared指定 | •指示行でのprivate指定<br>•DO/forループの反復変数<br>•stack 変数:並列実行領域でアロケートされた変数 |

# データ環境:格納属性の変更

- 格納属性は、以下の句を使用して規定可能
- すべてのデータ句は、並列実行領域にのみ用いられる “SHARED” を除いて、並列実行領域とワークシェアリング構文に用いられる
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE
- 並列ループ内側のプライベートの値は、ループ外側のグローバル値に送信できる
  - LASTPRIVATE
- デフォルトのステータスは、次の句を使用して変更できる
  - DEFAULT (PRIVATE | SHARED | NONE)

# private(list)

- スレッドに独自に領域が確保される
- 変数はスレッド内だけでアクセスされる変数となり、各スレッド毎に異なった値となる変数などは、このリストに指定する必要がある
- このリストに指定された変数は、並列実行領域の開始時点では、その値は全て未定義となり、また、並列実行領域の終了時点でそのデータは、破棄される

```
      PROGRAM WRONG
      IS = 0
!$OMP PARALLEL DO PRIVATE(IS)
      DO J=1,1000
            IS = IS + J
      END DO
      PRINT *, IS
      END
```

この並列実行領域の開始時点では、変数ISは未定義であり、並列実行領域の終了時点で変数は、破棄されるため、ISの値をプリントした場合、その数値は、0となる

# private(list)

- スレッドに独自に領域が確保される
- 変数はスレッド内だけでアクセスされる変数となり、各スレッド毎に異なった値となる変数などは、このリストに指定する必要がある
- このリストに指定された変数は、並列実行領域の開始時点では、その値は全て未定義となり、また、並列実行領域の終了時点でそのデータは、破棄される

```
      PROGRAM WRONG
      IS = 0
!$OMP PARALLEL DO PRIVATE(IS)
      DO J=1,1000
            IS = IS + J
      END DO
      PRINT *, IS
      END
```

この並列実行領域の開始時点では、変数ISは未定義であり、並列実行領域の終了時点で変数は、破棄されるため、ISの値をプリントした場合、その数値は、0となる

# firstprivate(list)

- データの取り扱いは、privateと同じ
- 並列実行領域の開始時点に逐次実行部分の変数が各スレッドにコピーされる

```
      PROGRAM WRONG
      IS = 0
!$OMP PARALLEL DO FIRSTPRIVATE(IS)
      DO J=1,1000
            IS = IS + J
      END DO
      PRINT *, IS
      END
```

この並列実行領域の開始時点では、変数ISは逐次実行部分で定義された0が代入される。
並列実行領域の終了時点で変数は、破棄されるため、ISの値をプリントした場合、その数値は、0となる

# lastprivate(list)

- 並列実行領域の終了後は、領域内のプライベートデータの値は保持されない
- lastprivateで指示された変数は、その並列実行領域を最後に終了したスレッドの値がマスタスレッドにコピーされる

```
      IS = 0
!$OMP PARALLEL DO FIRSTPRIVATE(IS) &
!$OMP LASTPRIVATE(IS)
      DO J=1,1000
            IS = IS + J
      END DO
      PRINT *, IS
```

この並列実行領域の開始時点では、変数ISは逐次実行部分で定義された0が代入される。
並列実行領域の終了時点のISの値は、並列実行領域を最後に実行し、終了した値がコピーされるため、0以外の数値となる

# reduction(operator:list)

- DO/for構文におけるリダクション演算で、演算子(operator)と変数のリストを指示
- 変数は共有データの属性である必要があるが、並列実行領域内では、これらの変数は各スレッド毎のプライベートデータとして処理される
- 終了時の同期処理の実行後、リダクション演算の結果が確保される

```
      IS = 0
!$OMP PARALLEL DO REDUCTION(+:IS)
      DO J=1,1000
            IS = IS + J
      END DO
      PRINT *, IS
```

このリダクション演算の指定で、このプログラムは、逐次処理と同じ結果が得られる

# default 句

- デフォルトの格納属性は DEFAULT(SHARED) なので、指定する必要はない
- デフォルトを変更するには: DEFAULT(PRIVATE)
  - 並列実行領域の静的範囲の各 変数は、private 句で指定されているかのように、プライベートになる
  - 主に入力を節約
- DEFAULT(NONE): 静的範囲の変数用のデフォルトはない 静的範囲の各変数のマルチリスト格納属性
- Fortran API のみ、default(private) をサポートしていて、C/C++ では、default(shared) または default(none) のみのサポート

# default 句の例

- 以下のプログラムでの各変数の格納属性は同じ

```
      itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
      np = omp_get_num_threads()
      each = itotal/np
      ……
C$OMP END PARALLEL
```

```
      itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
      np = omp_get_num_threads()
      each = itotal/np
      ……
C$OMP END PARALLEL
```

# threadprivate

- グローバル・データをスレッドに対してプライベートにする
  - Fortran: COMMON ブロック
  - C/C++: ファイルスコープと静的変数
- PRIVATE にすることとは異なる
  - PRIVATE は、グローバル変数をマスクする
  - THREADPRIVATE は、各スレッド内のグローバル・スコープを保存する
- スレッドプライベート変数は、COPYIN または DATA ステートメントを使用して初期化できる

# threadprivate の例

- 並列実行領域内で呼び出される 2 つの異なるルーチンについて考える
- threadprivate 構文により、これらのルーチンを実行する各スレッドは、common ブロック /buf/ の独自のコピーを持つ

```
      subroutine foo
      parameter (N=1000)
      common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
      do i=1, N
        B(i)= const* A(i)
      end do
      return
      end
```

```
      subroutine bar
      parameter (N=1000)
      common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
      do i=1, N
        A(i) = sqrt(B(i))
      end do
      return
      end
```

# THREADPRIVATE利用上の注意

- THREADPRIVATEを使用する場合、スレッドの動的スケジューリングはOFFにする必要がある

```
PROGRAM THREADPRIV
      INTEGER ALPHA(10), BETA(10), I
      COMMON /A/ ALPHA
!$OMP THREADPRIVATE(/A/)
C Explicitly turn off dynamic threads
      CALL OMP_SET_DYNAMIC(.FALSE.)
C First parallel region
!$OMP PARALLEL PRIVATE(BETA, I)
      DO I=1,10
         ALPHA(I) = I
         BETA(I) = I
      END DO
!$OMP END PARALLEL
C Second parallel region
!$OMP PARALLEL
      PRINT *, 'ALPHA(3)=',ALPHA(3),
     &         ' BETA(3)=',BETA(3)
!$OMP END PARALLEL
END
```

```
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
main () {
/* Explicitly turn off dynamic threads */
   omp_set_dynamic(0);
/* First parallel region */
#pragma omp parallel private(i,beta)
   for (i=0; i < 10; i++)
      alpha[i] = beta[i] = i;
/* Second parallel region */
#pragma omp parallel
   printf("alpha[3]= %d and
      beta[3]= %d¥n",alpha[3],beta[3]);
}
```

**このプログラムの実行時の結果**

```
alpha[3]= 3 and beta[3]= 0
alpha[3]= 3 and beta[3]= 0
alpha[3]= 3 and beta[3]= 0
alpha[3]= 3 and beta[3]= 0
```

**並列実行領域間でのデータの保持**

# copyprivate

- copyprivate 句を使用して、スレッドプライベート・データを初期化する

```
      parameter (N=1000)
      common/buf/A(N)
C$OMP THREADPRIVATE(/buf/)
C Initialize the A array
      call init_data(N,A)
C$OMP PARALLEL
C$OMP SINGLE COPYPRIVATE(A)
 … Now each thread sees threadprivate array A
initialized
 … to the global value set in the subroutine
      init_data()
C$OMP END SINGLE
C$OMP END PARALLEL
end
```

# リダクション

- 変数の共有方法に影響するもう 1 つの句
  - reduction (op : list)
- “list” 内の変数は囲まれている並列実行領域内で共有しなければならない
- 並列またはワークシェアリング構文の内側
  - 各 list 変数のローカルコピーは、“op” に依存して作成され、初期化される（例えば、“+” の場合は 0）
  - コンパイラは、“op” を含む標準リダクション式を検索して、ローカルコピーの更新に使用する
  - ローカルコピーは、単一の値にされ、オリジナルのグローバル値と結合される

# リダクションの例

```
      program closer
      IS = 0
      DO J=1,1000
           IS = IS + J
1000  CONTINUE
      print *, IS
```

```
      program correct
      IS = 0
#pragma omp parallel for reduction(+:IS)
      DO J=1,1000
           IS = IS + J
1000  CONTINUE
      print *, IS
```

# リダクション演算例

```
      PROGRAM DOT_PRODUCT
      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=100)
      PARAMETER (CHUNKSIZE=10)
      REAL A(N), B(N), RESULT
! Some initializations
      DO I = 1, N
         A(I) = I * 1.0
         B(I) = I * 2.0
      ENDDO
      RESULT= 0.0
      CHUNK = CHUNKSIZE
!$OMP PARALLEL DO
!$OMP& DEFAULT(SHARED) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)
!$OMP& REDUCTION(+:RESULT)
      DO I = 1, N
         RESULT = RESULT + (A(I) * B(I))
      ENDDO
      PRINT *, 'Final Result= ', RESULT
      END
```

```
#include <omp.h>
main () {
   int i, n, chunk;
   float a[100], b[100], result;
/* Some initializations */
   n = 100;
   chunk = 10;
   result = 0.0;
   for (i=0; i < n; i++)
      { a[i] = i * 1.0; b[i] = i * 2.0; }
#pragma omp parallel for ¥
default(shared) private(i) ¥
schedule(static,chunk) ¥
reduction(+:result)
   for (i=0; i < n; i++)
      result = result + (a[i] * b[i]);
   printf("Final result= %f¥n",result);
}
```

# リダクションのオペランド/初期値

- 一連のアソシエーティブ・オペランドがリダクションで使用できる
- 初期値は、数学的に意味をなすもの

| 演算子 | 初期値 |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| .AND. | 全て1 |

| 演算子 | 初期値 |
|---|---|
| .OR. | 0 |
| MAX | 1 |
| MIN | 0 |
| // | 全て1 |

# OpenMPでのプログラミング
## OpenMP APIのご紹介

次の5つのカテゴリに分類されるAPIについて、理解することが必要

1. 並列実行領域（Parallel Regions）構文
2. ワークシェアリング（Worksharing）構文
3. データ環境（Data Environment）構文
4. **同期（Synchronization）構文**
5. 実行時関数/環境変数（Runtime functions/environment variables）

# 同期構文

- 複数のスレッドが並列に実行されている場合、これらのスレッド間で同期を取ることが必要
- 最も一般的なものは、全スレッドの実行が終了するのを待つbarrier同期
  - OpenMPでは、全ての並列実行領域の終了時にこのbarrier同期を行う
  - barrier同期は、#pragma omp forや#pragma omp sectionsといったワークシェアリング構文ブロックの終了時にも適用
  - ワークシェアリング構文では、nowait指示句を指定することで、barrier同期を行わない
- OpenMPの同期構文としては、以下のようなものが利用可能
  - CRITICAL / ATOMIC / BARRIER / FLUSH / ORDERED / SINGLE / MASTER

# ワークシェアリングでの同期処理

```
#pragma omp parallel shared (A, B, C) private(id)
{
       id=omp_get_thread_num();
       A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
       for(i=0;i<N;i++){
         C[i]=big_calc3(I,A);
       }
#pragma omp for nowait
       for(i=0;i<N;i++){
         B[i]=big_calc2(C,  i);
       }
       A[id] = big_calc3(id);

}
```

ワークシェアリング構文の最後にある暗黙的なバリア

nowait による暗黙的バリア同期の排除

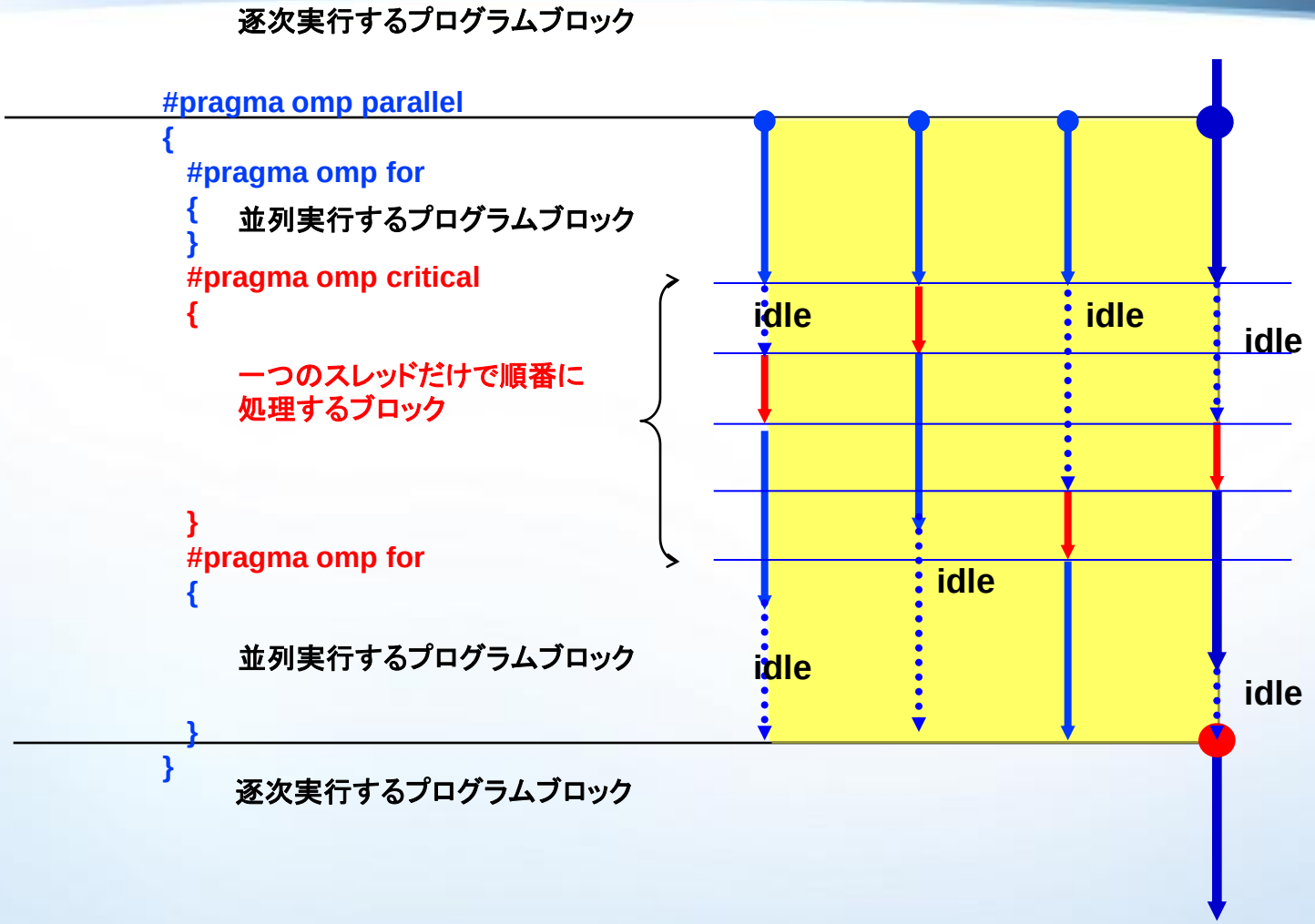
並列実行領域の最後にある暗黙的なバリア

# クリティカルセクション

- クリティカルセクション
  - 一つのスレッドだけで順次実行される並列実行領域内のブロック
  - クリティカルセクションでは特定のスレッドの実行中は、他のスレッドはそのスレッドの実行が終了するのを待つ

```
Fortran: !$OMP CRITICAL [( name )]
                    block
             !$OMP END CRITICAL [( name )]
C/C++:   #pragma omp critical [( name )]
                    structured block
```

# クリティカルセクション



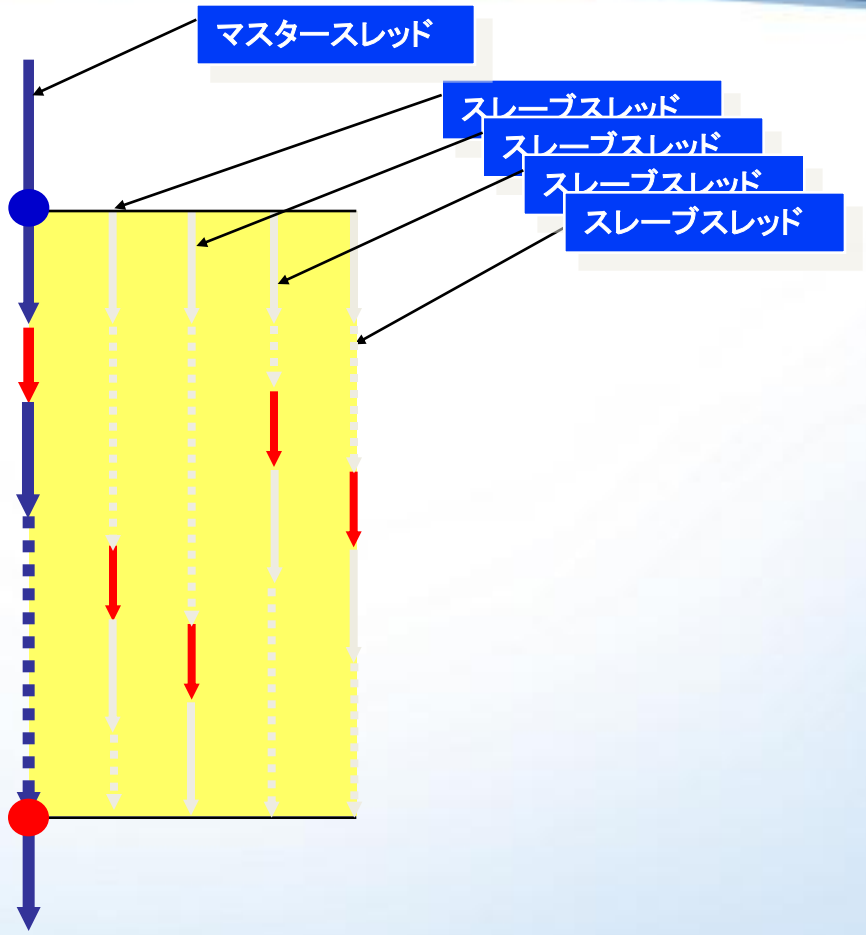
# クリティカルセクションの実行

逐次実行するプログラムブロック

```
!$OMP PARALLEL
!$OMP DO
DO I = 1, N
並列実行するコードブロック
END DO
!$OMP END DO
!$OMP CRITICAL
一つのスレッドだけで順番に処理するコードブロック
!$OMP END CRITICAL
!$OMP DO
DO I = 1, N
並列実行するコードブロック
END DO
!$OMP END DO
!$OMP END PARALLEL
```

逐次実行するプログラムブロック

並列実行するプログラムブロック

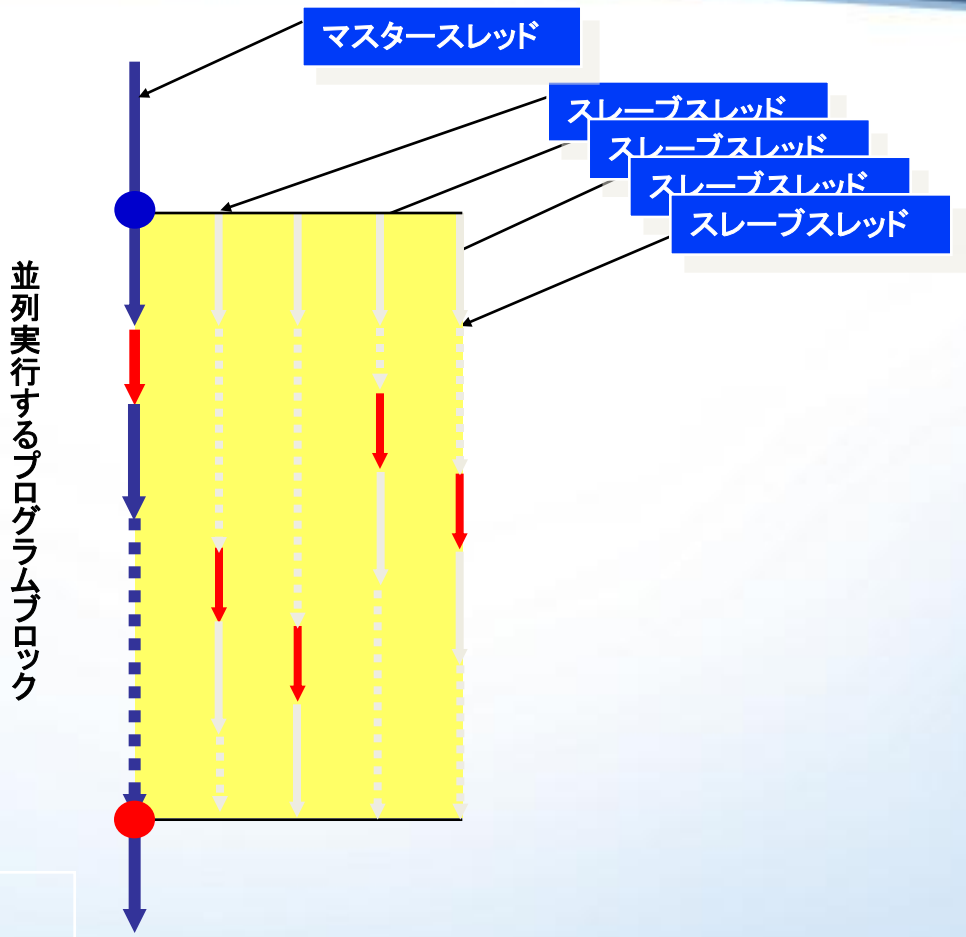


# クリティカルセクションの実行

逐次実行するプログラムブロック

```
#pragma omp parallel
{
  #pragma omp for
  {
        並列実行するワークシェアリング
            コードブロック
  }
  #pragma omp critical
  {
        一つのスレッドだけで順番に処理
            するコードブロック
  }
  #pragma omp for
  {
        並列実行するワークシェアリング
            コードブロック
  }
}
```

逐次実行するプログラムブロック



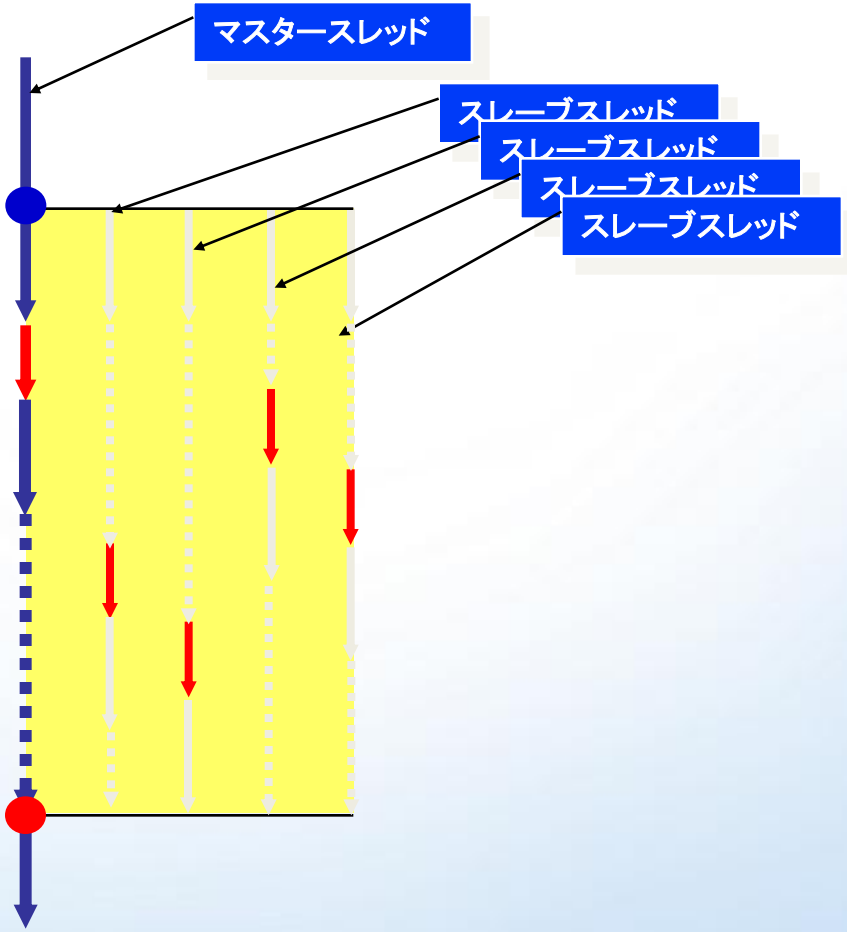
# atomic アップデートの指定

逐次実行するプログラムブロック

```
!$OMP PARALLEL
!$OMP DO
DO I = 1, N
並列実行するコードブロック
END DO

!$OMP END DO
!$OMP ATOMIC
順番に処理するステートメント
!$OMP DO
DO I = 1, N
並列実行するプログラムブロック
END DO
!$OMP END DO
!$OMP END PARALLEL
```

並列実行するプログラムブロック

逐次実行するプログラムブロック



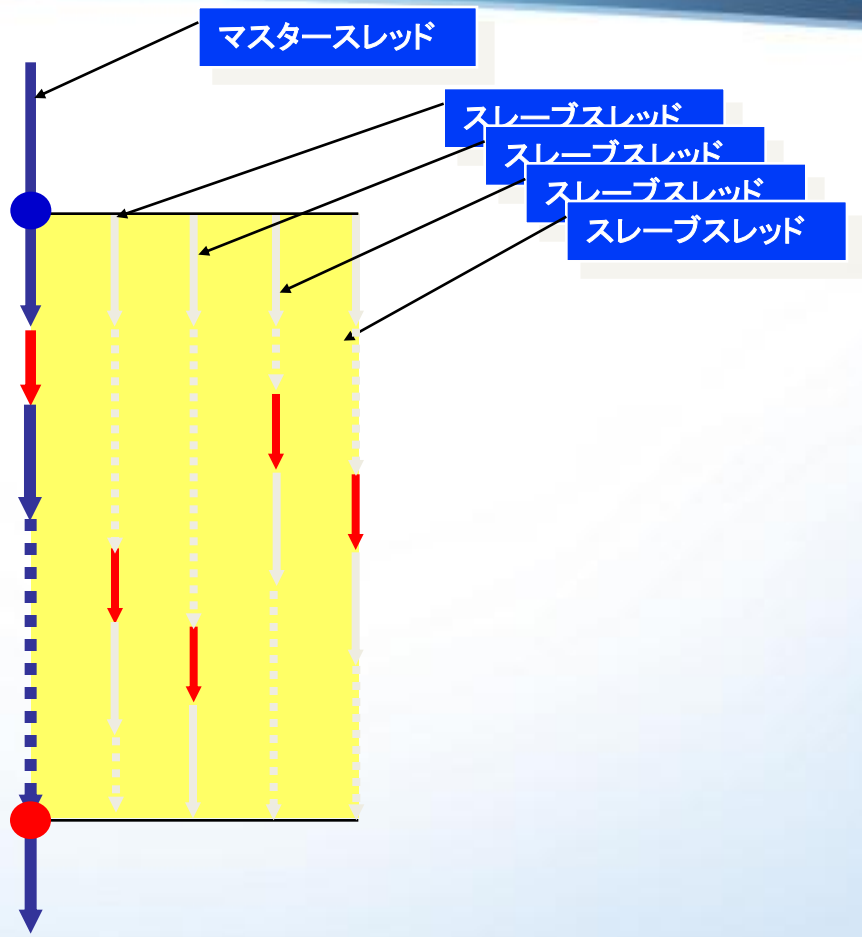
# atomic アップデートの指定

逐次実行するプログラムブロック

```
#pragma omp parallel
{
  #pragma omp for
  {
        並列実行するワークシェアリング
              コードブロック
  }
  #pragma atomic
        順番に処理するステートメント
  #pragma omp for
  {
        並列実行するワークシェアリング
              コードブロック
   }
}
```

逐次実行するプログラムブロック

並列実行するプログラムブロック



# クリティカルセクション

- 一度に 1 スレッドのみ critical セクションを処理できる

```
float res;
#pragma omp parallel
{      float B;    int i;
#pragma omp for
       for(i=0;i<niters;i++){
                B =  big_job(i);
#pragma omp critical
                 consum (B, RES);
       }
}
```

```
C$OMP PARALLEL DO PRIVATE(B)
C$OMP& SHARED(RES)
       DO 100 I=1,NITERS
                B =  DOIT(I)
C$OMP CRITICAL
                CALL CONSUME (B, RES)
C$OMP END CRITICAL
100    CONTINUE
```

# アトミックアップデート

- atomic は、特定の単純なステートメントで使用できる critical セクションの特別なケース
- メモリ領域（下の例では X）の更新にのみ用いられる

```
C$OMP PARALLEL PRIVATE(B)
       B =  DOIT(I)
       tmp = big_ugly();

C$OMP ATOMIC
       X = X + temp

C$OMP END PARALLEL
```

# 実行順序の制御

- ワークシェアリング構文中の指定したブロックを逐次処理で実行した場合と同じ順序で実行することを指定
- ワークシェアリング構文での各反復の実行順序は、実行時に決定されるため、このORDERED構文の指定が無い場合には、その実行順序は不定となる

```
PROGRAM ORDERED
  IMPLICIT NONE
  INTEGER, PARAMETER:: N=1000, M=4000
  REAL, DIMENSION(N,M):: X,Y
  REAL, DIMENSION(N):: Z
  INTEGER I,J
  CALL RANDOM_NUMBER(X)
  CALL RANDOM_NUMBER(Y)
  Z=0.0
  PRINT *, 'The first 10 values of Z are:'
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(I,J)
!$OMP DO SCHEDULE(DYNAMIC,4) ORDERED
  DO I=1,N
    DO J=1,M
      Z(I) = Z(I) + X(I,J)*Y(J,I)
    END DO
!$OMP ORDERED
    IF(I<11) THEN
      PRINT *, 'Z(',I,') =',Z(I)
    END IF
!$OMP END ORDERED
  END DO
!$OMP END DO
!$OMP END PARALLEL
END PROGRAM ORDERED
```

# 実行順序の制御

```
PROGRAM ORDERED
  IMPLICIT NONE
  INTEGER, PARAMETER:: N=1000, M=4000
  REAL, DIMENSION(N,M):: X,Y
  REAL, DIMENSION(N):: Z
  INTEGER I,J
  CALL RANDOM_NUMBER(X)
  CALL RANDOM_NUMBER(Y)
  Z=0.0
  PRINT *, 'The first 10 values of Z are:'
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(I,J)
!$OMP DO SCHEDULE(DYNAMIC,4) ORDERED
  DO I=1,N
    DO J=1,M
      Z(I) = Z(I) + X(I,J)*Y(J,I)
    END DO
!$OMP ORDERED
    IF(I<11) THEN
      PRINT *, 'Z(',I,') =',Z(I)
    END IF
!$OMP END ORDERED
  END DO
!$OMP END DO
!$OMP END PARALLEL
END PROGRAM ORDERED
```

同様な実行順序の制御はプログラムでも可能

```
PROGRAM ORDERED
!$ USE OMP_LIB
        IMPLICIT NONE
        INTEGER, PARAMETER:: N=1000, M=4000
        REAL, DIMENSION(N,M):: X,Y
        REAL, DIMENSION(N):: Z
        INTEGER I,J,WHOIS,ME
        CALL RANDOM_NUMBER(X)
        CALL RANDOM_NUMBER(Y)
        Z=0.0
        PRINT *, 'The first 10 values of Z are:'
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(I,J)
        ME = OMP_GET_THREAD_NUM()
!$OMP DO SCHEDULE(DYNAMIC,4)
        DO I=1,N
           DO J=1,M
              Z(I) = Z(I) + X(I,J)*Y(J,I)
           END DO
           WHOIS = 0
           DO WHILE (WHOIS.EQ.ME)
              IF(I<11) PRINT *, 'Z(',I,') =',Z(I)
!$OMP CRITICAL
              WHOIS = WHOIS +1
!$OMP END CRITICAL
          END DO
        END DO
!$OMP END DO
!$OMP END PARALLEL
END PROGRAM ORDERED
```

# 同期構文：MASTER構文

- MASTER構文は、マスタスレッドによってのみ実行される構造ブロックを示す。他のスレッドはスキップする（暗黙的な同期は行われない）

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp master
        {       exchange_boundaries();    }
#pragma barrier
        do_many_other_things();
}
```

# 同期構文:SINGLE構文

- SINGLE構文は、1 つのスレッドによってのみ実行されるコードのブロックを示す
- 単一ブロックの最後でバリアが暗黙的に指定される
- ワークシェアリング中での同期処理に利用可能

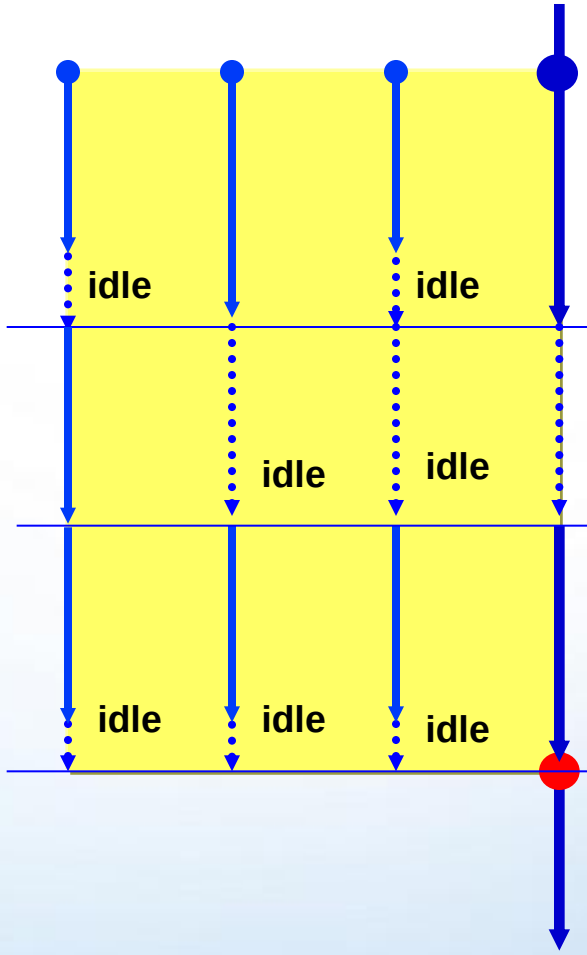
```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp single
        {      exchange_boundaries();    }
        do_many_other_things();
}
```

# 同期構文：SINGLE構文

逐次実行するプログラムブロック

```
#pragma omp parallel
{
#pragma omp for
  {
      並列実行するプログラムブロック
  }
  #pragma omp single
  {
      一つのスレッドだけで処理されるコード
        例えば、データ読み込み、書き込みなど
  }
  #pragma omp for
  {
      並列実行するプログラムブロック
  }
}
```

逐次実行するプログラムブロック




Barrierによる同期処理がワークシェア構文の終了時に適用されます。

指示句としてnowaitを指定した場合、同期処理は行われません。

並列実行領域の終了時にも、同期処理が適用されます。

# SINGLE構文の実行（Fortran)

逐次実行するプログラムブロック

!$OMP PARALLEL

並列実行するコードブロック

!$OMP SINGLE

一つのスレッドだけで処理するコード ブロック

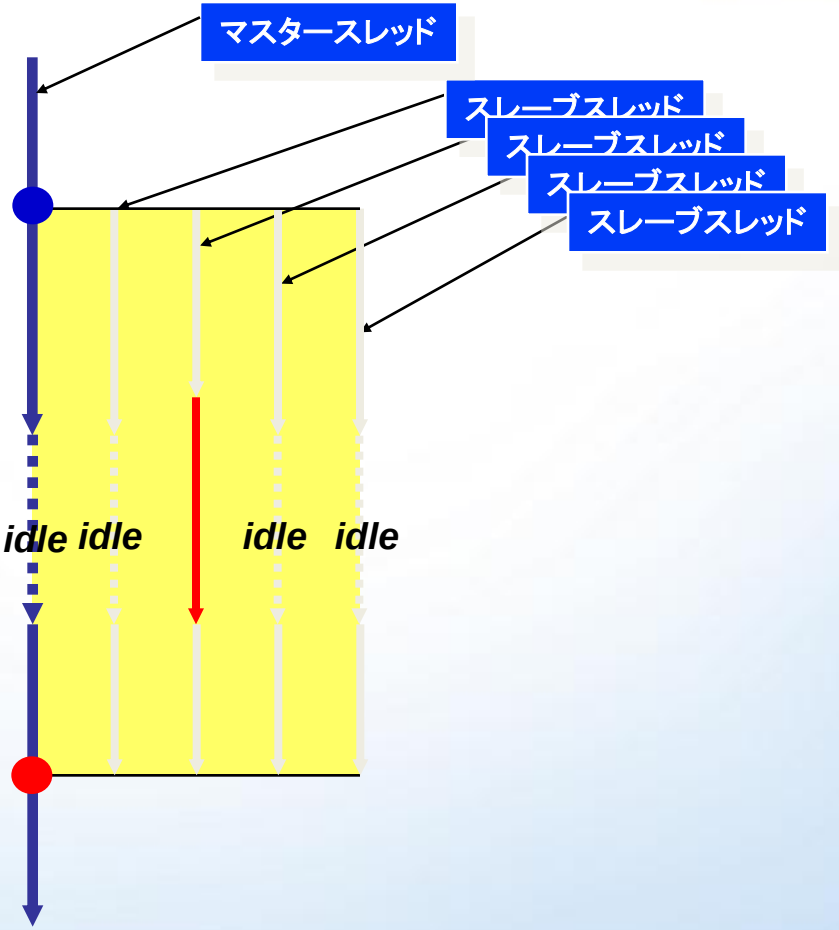
!$OMP END SINGLE

並列実行するコードブロック

!$OMP END PARALLEL

逐次実行するプログラムブロック

並列実行するプログラムブロック



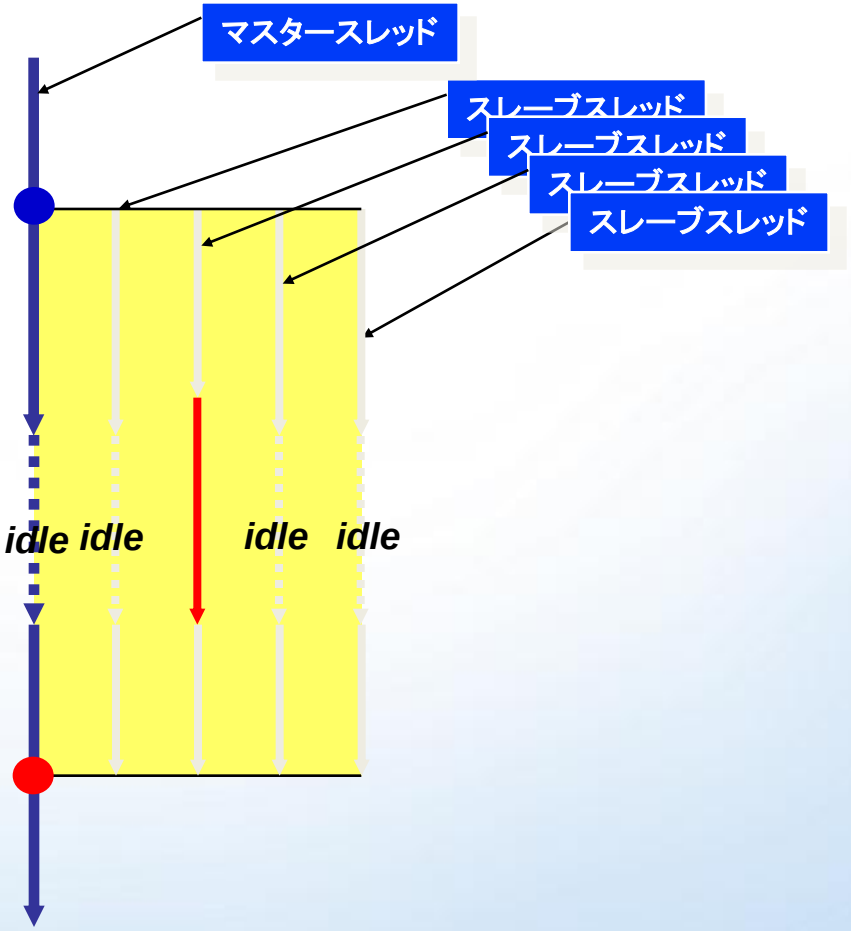
# single構文の実行(C/C++)

逐次実行するプログラムブロック

```
#pragma omp parallel
{
        並列実行するコードブロック

#pragma omp single
  {
        一つのスレッドだけで処理するコードブロック
  }

        並列実行するコードブロック

}
```

逐次実行するプログラムブロック

並列実行するプログラムブロック



# IF指示句

- IF指示句はPARALLEL構文に対して、その構文を実行時に有効とするか無効とするかを指定するためのもの
- IF_指示句の条件式がTRUEの場合のみ、並列実行構文が有効になり、並列実行がなされる

```
#include <omp.h>
void test(int val)
{
    #pragma omp parallel if (val) num_threads(val)
    if (omp_in_parallel())
        #pragma omp single
        printf("val = %d, parallelized with %d
        threads¥n",val, omp_get_num_threads());
    else
        printf("val = %d, serialized¥n", val);
}
int main( )
{
    test(0);
    test(2);
}
```

# NUM_THREADS指示句

- NUM_THREADS指示句によって、PARALLEL_構文を処理するスレッド数を指定することが可能

```
#include <omp.h>
main () {
...
omp_set_dynamic(1);
...
#pragma omp parallel for num_threads(10)
   for (i=0; i<10; i++)
       {
          ...
       }
}
```

# OpenMPでのプログラミング
## OpenMP APIのご紹介

次の5つのカテゴリに分類されるAPIについて、理解することが必要

1. 並列実行領域（Parallel Regions）構文
2. ワークシェアリング（Worksharing）構文
3. データ環境（Data Environment）構文
4. 同期（Synchronization）構文
5. 実行時関数/環境変数（Runtime functions/environment variables）

# OpenMP runtime ライブラリ

| 環境変数 | 説明 | 逐次実行領域での呼び出し | 並列実行領域での呼び出し |
|---|---|---|---|
| call omp_set_num_threads(integer) | 並列実行領域で使用するスレッド数を設定する | スレッド数の設定 | 呼び出し不可 |
| integer omp_get_num_threads() | 並列実行領域のスレッド数を返す | 1 | スレッド数 |
| integer omp_get_max_threads() | 最大のスレッド数を返す | OMP_NUM_THREADSの設定値 | |
| integer omp_get_thread_num() | 並列スレッドの番号を0からの返します | 0 | 各スレッド番号 |
| integer omp_get_num_procs() | プログラムで使用可能なプロセッサ数を返す | システムの物理CPU数 | |
| logical omp_in_parallel() | 並列実行中であれば真を返す | .FALSE. | .TRUE. |
| call omp_set_dynamics (logical) | スレッド数の動的制御有効、無効の設定 | | 呼び出し不可 |
| logocal omp_get_dynamic() | スレッド数の動的制御の判定 | 有効な場合には真を返す | |
| call omp_set_nested(logical) | ネストされた並列実行領域の有効、無効の設定 | | 呼び出し不可 |
| logical omp_get_nested() | ネストされた並列実行領域の判定 | 有効な場合には真を返す | |

# ライブラリ・ルーチン使用時の注意

- Fortran
  - これらの OpenMP 関数は先頭が OMP で始まっているので、型宣言を忘れた場合には、関数の値が実数として返される
  - OpenMP の API では、この問題に対応するために omp_lib というモジュールを定義

```
program omp
!$ use omp_lib
  .
!$omp parallel
Iam = omp_get_thread_num()
!$omp end parallel
  .
end program omp
```

- C/C++
  - OpenMP_関数を利用するためのインクルードファイルは、#include “omp.h”で指定

# ライブラリ・ルーチン使用例

- プログラムで使用するスレッドの数の制御例
  - スレッドの数を設定する
  - 返された数を保存する

```
#include <omp.h>
void main()
{    int num_threads;
     omp_set_num_threads(omp_num_procs());
#pragma omp parallel
     {      int id=omp_get_thread_num();
#pragma omp single
            num_threads = omp_get_num_threads();
            do_lots_of_stuff(id);
      }
}
```

システムに搭載されているプロセッサ数と同じ数のスレッド数を指定する

各スレッドの論理スレッド番号を得る

実行時にForkされたスレッド数を得る。スレッド数は、各スレッドで共有される。

# 環境変数

| 環境変数 | 説明 |
|---|---|
| OMP_NUM_THREADS | スレッド数の動的調整が有効になっている場合、環境変数の値は、使用するスレッドの数の上限として解釈されます。<br>例：setenv OMP_NUM_THREADS 4 |
| OMP_SCHEDULE | 環境変数のデフォルト値は、処理系に依存します。指定では、type[,chunk] の形式で指定し、type には、STATIC/DYNAMIC/GUIDED が指定可能です。chunkの設定は、オプションです。chunkが設定されていない場合には、STATICスケジュールの場合を除き、値 1 が使用されます。STATICスケジュールでは、デフォルトのchunkは、ループカウントを、そのループに適用されるスレッドの数で割った値に設定されます。<br>例：setenv OMP_SCHEDULE “dynamic” |
| OMP_DYNAMIC | 値がTRUE に設定されている場合、並列実行領域の実行に使用されるスレッドの数は、システムのロードなどによって、システムが実行時に調整します。値がFALSE に設定されていると、この動的調整は無効になります。<br>例：setenv OMP_DYNAMIC TRUE |
| OMP_NESTED | 値がTRUE に設定されていると、ネストされた並列実行は有効になり、値がFALSEに設定されていると、ネストされた並列実行は無効になります。デフォルト値はFALSE です。<br>例：setenv OMP_NESTED TRUE |

# 指示構文と指示句の指定

| 構文 | parallel | do/for | sections | workshare | single | parallel do/for | parallel sections | parallel workshape |
|---|---|---|---|---|---|---|---|---|
| if | ○ | | | | | ○ | ○ | ○ |
| schedule | | ○ | | | | ○ | | |
| private | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| share | ○ | | | | | ○ | ○ | ○ |
| default | ○ | | | | | ○ | ○ | ○ |
| firstprivate | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| lastprivate | | ○ | ○ | | | ○ | ○ | |
| copyin | ○ | | | | | ○ | ○ | ○ |
| copyprivate | | | | | ○ | | | |
| reduction | ○ | ○ | ○ | | | ○ | ○ | ○ |
| ordered | | ○ | | | | ○ | | |
| nowait | | ○ | ○ | ○ | | | | |
| num_threads | ○ | | | | | ○ | ○ | ○ |

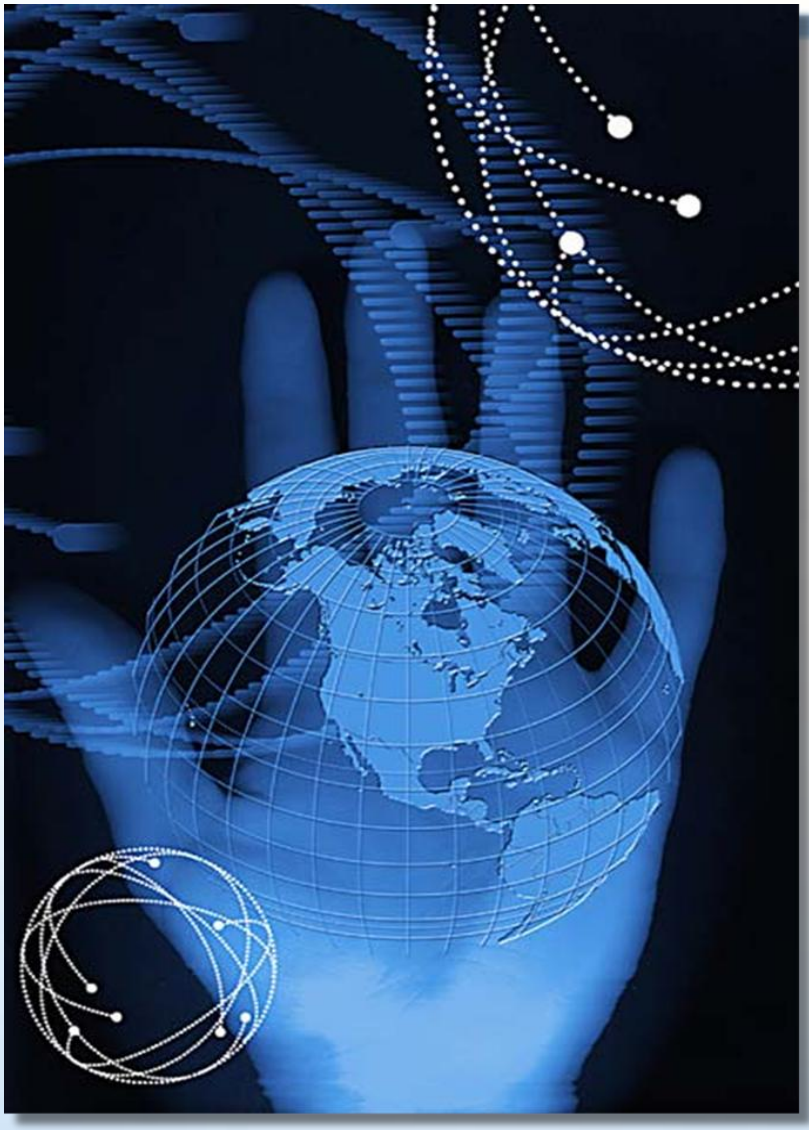
# 環境変数使用例

- OpenMPによりマルチスレッド化されたアプリケーションを実行する場合、その実行環境は、OSに対する環境変数で設定可能となります。
- 実行するスレッド数などは、コンパイル時に指定する必要はなく、同じ実行モジュールを利用して、環境変数で実行するスレッド数を指定します。

# OpenMPでの時間計測

- OMP_GET_WTIME
  - 経過時間を倍精度実数で返す関数
  - スレッド毎の時間の計測に利用可能
  - Fortran:
    - DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
  - C/C++:
    - #include <omp.h>
    - double omp_get_wtime(void)
- OMP_GET_WTICK
  - 経過時間計測のためにclock tick値を倍精度実数で返す関数
  - Fortran:
    - DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
  - C/C++:
    - #include <omp.h> double omp_get_wtick(void)

# この資料について



**お問い合わせ**
**0120-090715**
携帯電話・PHSからは（有料）
**03-5875-4718**
9:00-18:00 (土日・祝日を除く)

WEBでのお問い合わせ
**www.sstc.co.jp/contact**



**1/17/2010**

**スケーラブルシステムズ株式会社**