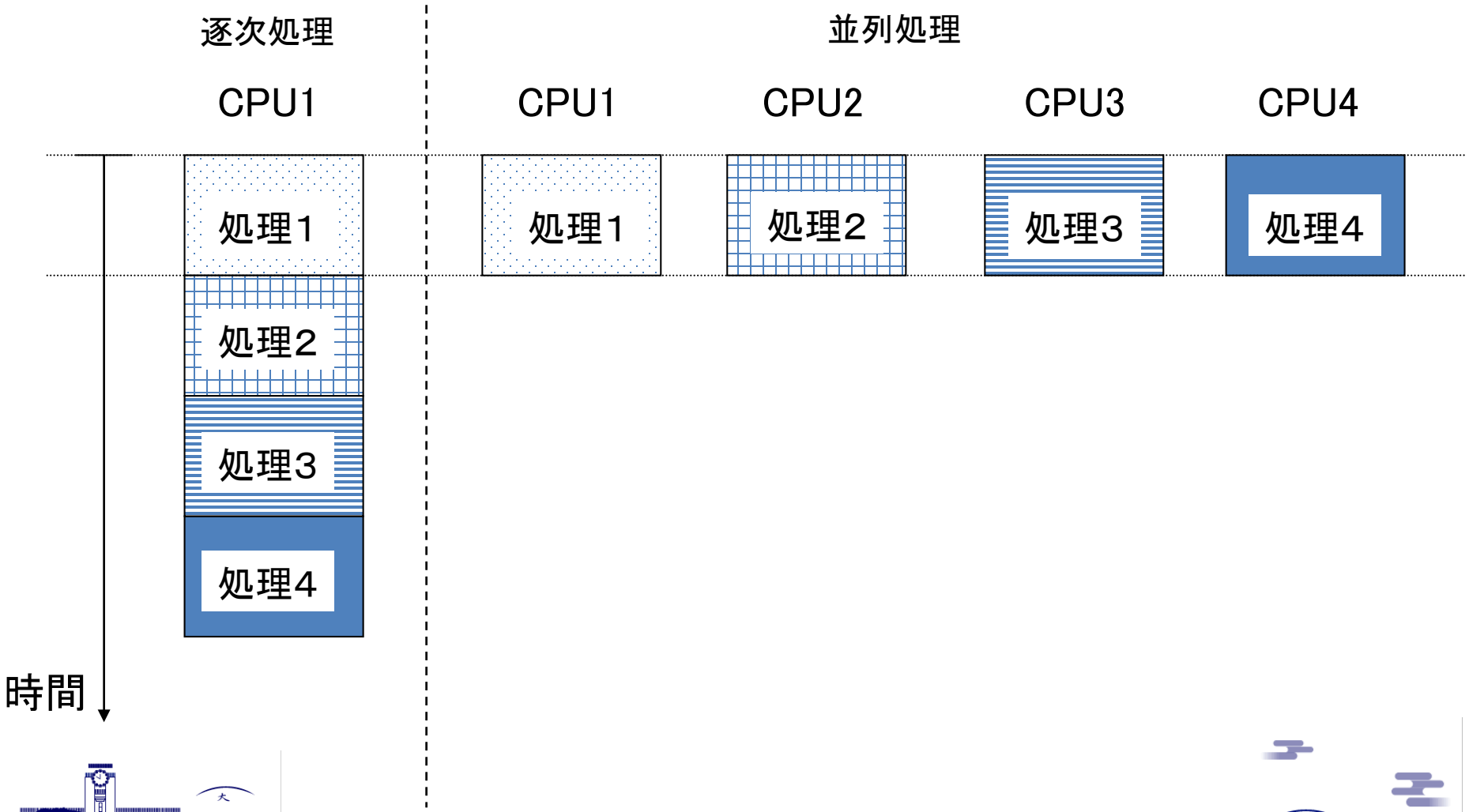
2018/9/19

計算科学演習B
OpenMP基礎

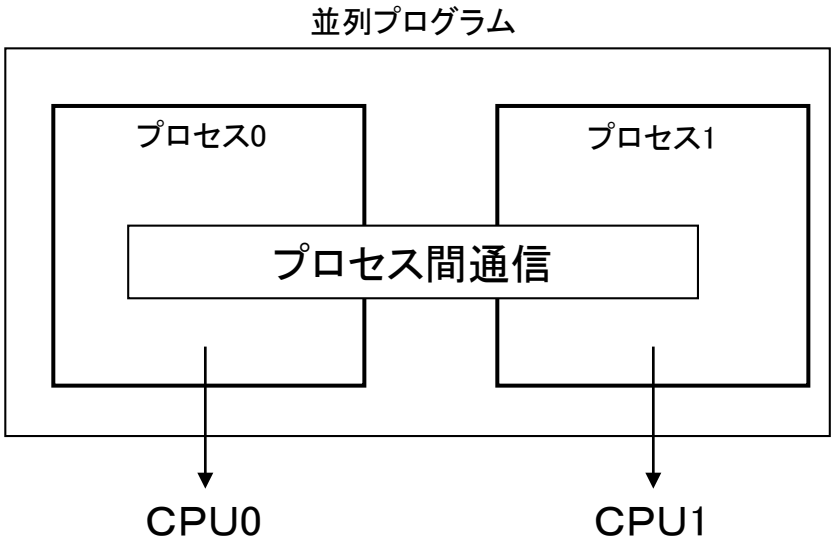
深沢圭一郎
学術情報メディアセンター

並列処理とは


逐次処理
CPU1
処理1
処理2
処理3
処理4
時間
並列処理
CPU1
CPU2
CPU3
CPU4
処理1
処理2
処理3
処理4

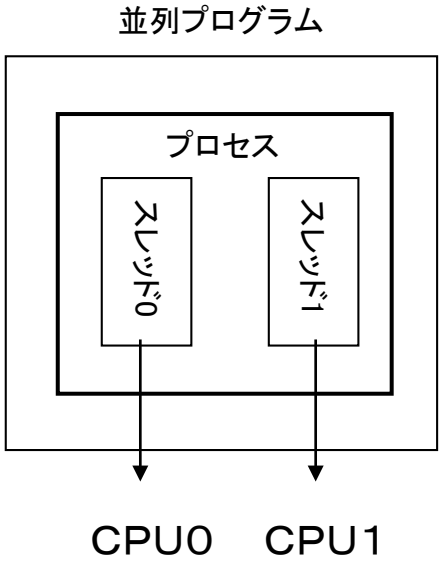
2種類の並列処理方法

プロセス並列



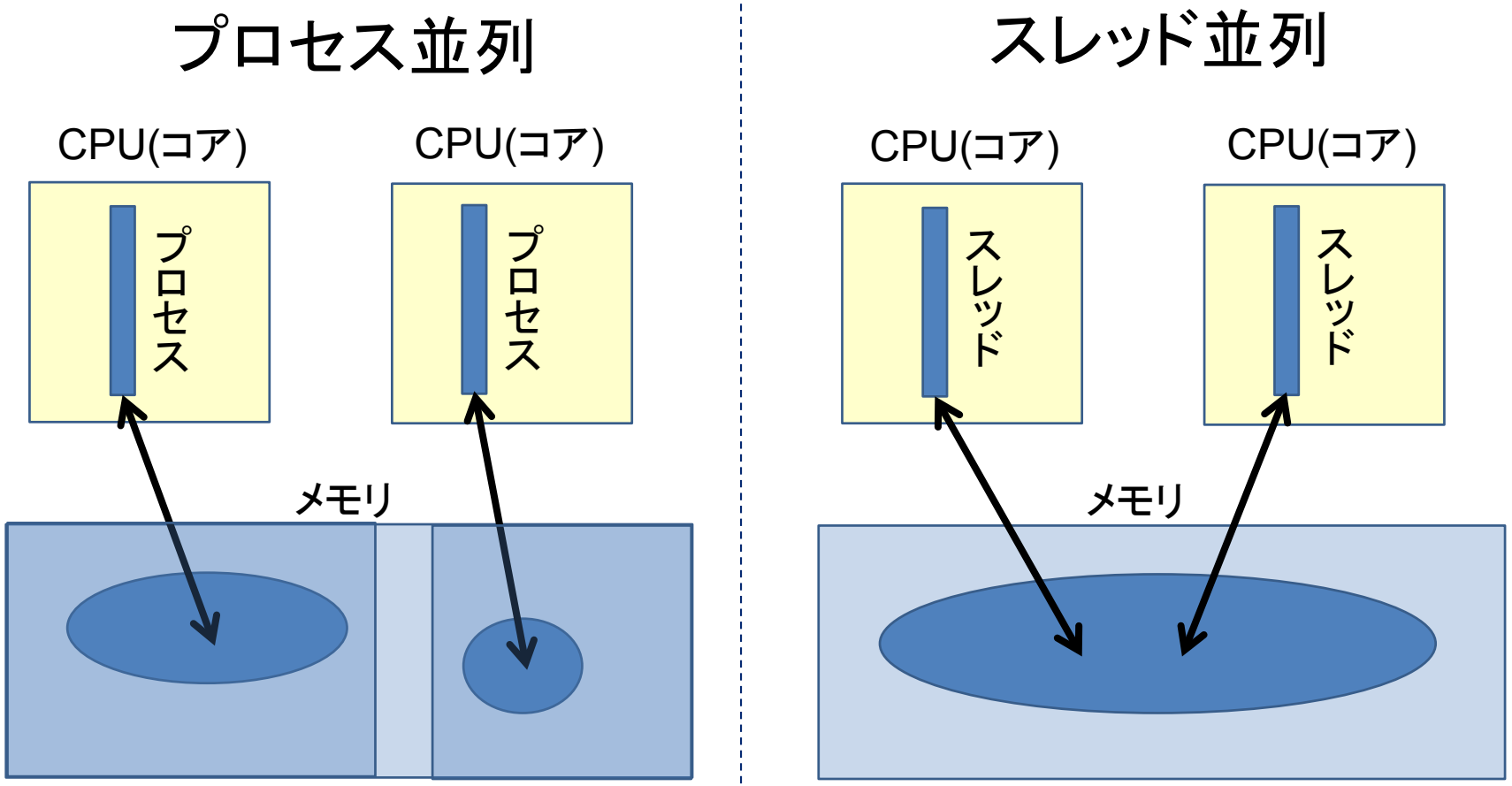
- 分散メモリ型並列計算機向け(例:PCクラスタ)(共有メモリ型でも使用可能)
- メッセージパッシングライブラリ(MPIなど)を用いる

スレッド並列



- 共有メモリ型並列計算機向け(例:計算ノード内)
- OpenMPを用いる
- コンパイラによる自動並列化もある

プロセス並列とスレッド並列



サブシステムA, B, Cにおける並列処理

サブシステムはいずれもクラスタ型の構成
一つor複数のプロセッサによる計算ノードをネットワークにより結合したもの

ノード内での並列処理
→プロセス並列（MPI）、スレッド並列（自動並列/OpenMP）のいずれも可能

複数ノードでの並列処理
→プロセス並列の利用が必須
- プロセス並列のみを利用→Flat-MPI

→プロセス／スレッド併用並列処理
- ハイブリッド並列処理
- MPI＆自動並列，MPI & OpenMP

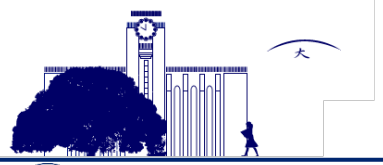
OpenMPとは

共有メモリ型並列計算機における並列プログラミングの統一規格の一つです。

並列実行単位は“スレッド”と呼ばれます。
並列化を指示する指示行をプログラムに挿入し、並列化します。

1997年10月にOpenMP for Fortran Version1.0 が規格
1998年10月にOpenMP for C/C++ Program Version1.0 が規格

その後もバージョンアップに伴い、機能強化されて現在は
2015年11月にOpenMP Version4.5（C, C++, FORTRANの全てを含む）が最新規格
（最新情報はhttp://www.openmp.org を参照）



OpenMPプログラミングの解説

OpenMPを利用する上での基本を学ぶ

1. 指示文（ディレクティブ）の形式
   書式を学ぶ
2. マルチスレッドでの並列実行
   実行イメージを学ぶ
3. Work-Sharing構造
   スレッド並列の書き方
4. 変数の属性
   スレッド並列化に伴う変数の扱い
5. 同期
   並列計算に伴う処理合わせ
6. まとめ

ディレクティブの形式（Fortran）

特別なコメント

- OpenMPコンパイラが解釈。Fortranコンパイラではただのコメント。

形式

- 固定形式：!OMP ,COMP, *$OMP
- 自由形式：!$OMP
  例 ： !$OMP PARALLEL

継続行

- !$OMP PARALLEL DO REDUCTION(+:x)を2行に継続する場合
- 固定形式
```
!$OMP PARALLEL DO
!$OMP+REDUCTION(+:x)
```
- 自由形式
```
!$OMP PARALLEL DO &
!$OMP REDUCTION(+:x)
```

ディレクティブの形式（C言語）

形式

```
#pragma omp 指示子 構造ブロック
 例1 : #pragma omp parallel
        { …
        }
 例2 : #pragma omp for
```

コンパイラによるディレクティブの解釈

コンパイルオプションにより指定

指定ON → OpenMPのディレクティブとして解釈
指定OFF → コメントとして無視する
指定(ON)の例:

- Crayコンパイラ `ftn -h omp ***.f90`
- Intel コンパイラ `ifort -qopenmp ***.f90`

生成された実行バイナリ

環境変数(OMP_NUM_THREADS)により指定されたスレッド数での並列実行が行われる

- プログラム内で指定する方法もある `omp_set_num_threads()`関数を利用

マルチスレッドでの実行イメージ

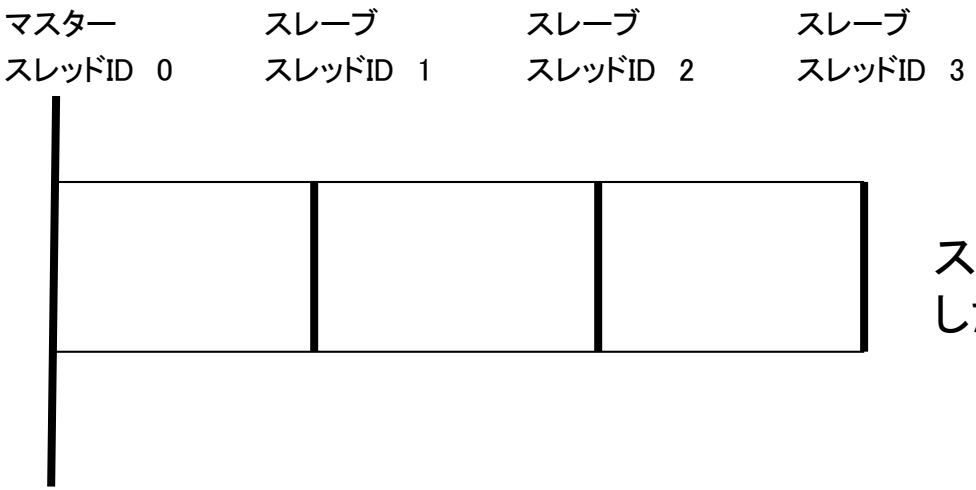
プログラム実行開始時はマスタースレッドのみ

PARALLELディレクティブによりスレーブスレッドを生成

- スレッドID： マスタースレッドは0、スレーブスレッドは1～
- チーム： 並列実行を行うスレッドの集団
- スレッド生成後、全てのスレッドで冗長実行

END PARALLELディレクティブによりスレーブスレッドが消滅

```
program main
        :
!$OMP PARALLEL
        :

!$OMP END PARALLEL
        :
end program main
```

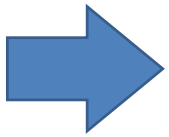


スレッド数4を指定した場合

OpenMPによる並列化プログラムの基本構成例

Fortran90プログラム

```
program main
  integer :: i, j
  real(kind=8) :: a, b
  …
  !$OMP PARALLEL
  …
  …
  …
  !$OMP END PARALLEL
  …
  …
end
```

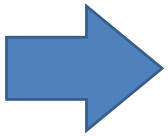


複数のスレッドにより
並列実行される部分

OpenMPによる並列化プログラムの基本構成例

Cプログラム

```
int main(){
   int i, j;
   double a, b;
   …
   #pragma omp parallel
   {
      …
      …
   }
}
```

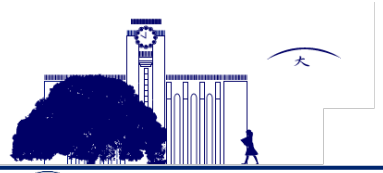


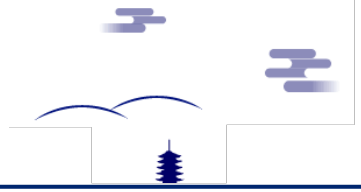
複数のスレッドにより
並列実行される部分

複数スレッドによる冗長実行

```
program main
  integer :: i, j
  real(kind=8) :: a, b
  …
  !$OMP PARALLEL
        a=b
  !$OMP END PARALLEL
  …
  …
end
```

- **OpenMPによる並列化プログラムでは特に何も指示しないと、パラレルリージョン（並列実行される部分）での実行文は冗長実行される。**
- 共有メモリなので、複数のスレッドから見て変数aのメモリ上の物理的な番地（実体）は同じ。
- a=bがスレッドの数だけ行われる





複数スレッドによる冗長実行に関する理解

```
program main
  integer :: i=0
  !$OMP PARALLEL
     i=i+1
  !$OMP END PARALLEL
  write(6,*) i
end
```

このプログラムをスレッド数4を
指定して実行する。

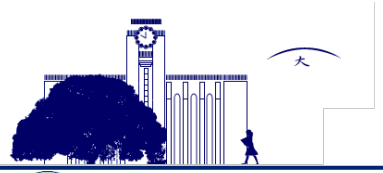
標準出力に出力される値は
なんでしょうか？

a) 1

b) 0

c) 4

d) 不定



複数スレッドによる冗長実行に関するクイズ

```
int main(...) {
  int  i=0;
#pragma omp parallel
  {
    i=i+1;
  }
  printf("%d¥n", i);
}
```

このプログラムをスレッド数4を指定して実行する。

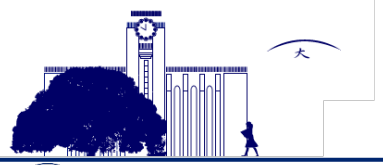
標準出力に出力される値はなんでしょうか？

a) 1

b) 0

c) 4

d) 不定



計算の並列化（Work-Sharing構造）

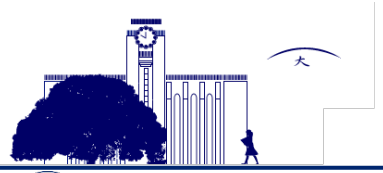
チーム内のスレッドに仕事（Work）を分割(Share)

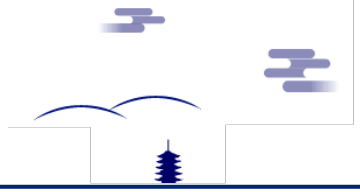
Work-Sharing構造の種類

- DOループを各スレッドで分割 (!$OMP DO, !$OMP END DO)
- 別々の処理を各スレッドが分担(!$OMP SECTIONS, !$OMP END SECTIONS)
- 1スレッドのみ実行(!$OMP SINGLE, !$OMP END SINGLE)

Work-sharing構造ではないが･･･

- マスタスレッドでのみ実行(!$OMP MASTER, !$OMP END MASTER)





計算の並列化（Work-Sharing構造）

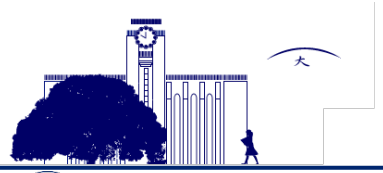
チーム内のスレッドに仕事（Work）を分割(Share)する。

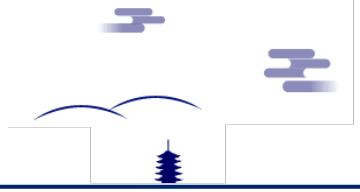
Work-Sharing構造の種類

- for ループを各スレッドで分割 (#pragma omp for)
- 別々の処理を各スレッドが分担(#pragma omp sections)
- 1スレッドのみ実行(#pragma omp single)

Work-sharing構造ではないが・・・

- マスタスレッドでのみ実行(#pragma omp master)





OMP DO(1)

```
integer :: i
real(kind=8) :: a(100), b(100)
!$OMP PARALLEL
!$OMP DO
do i=1,100
    b(i)=a(i)
enddo
!$OMP END DO
!$OMP END PARALLEL
end
```

「直後のdoループを複数のスレッドで分割して実行せよ」という指示

2スレッドの場合：

スレッド0

```
  do i=1,50
    b(i)=a(i)
  enddo
```

スレッド1

```
      do i=51,100
    b(i)=a(i)
  enddo
```

omp for (1)

```
int i;
double a[100], b[100];
#pragma omp parallel
  {
#pragma omp for
  for(i=0; i<100; i++) {
      b[i]=a[i];
    }
  }
```

直後のforループを複数のスレッドで分割して実行せよ　という指示

2スレッドの場合：

スレッド0

```
  for(i=0;i<50;i++) {
        b[i]=a[i];
    }
```

スレッド1

```
      for(i=50;i<100;i++) {
         b[i]=a[i];
      }
```

OMP DO(2)

注意!$OMP DO はdoループの中身が並列実行可能かどうかは関知せず、必ず分割してしまう。

```
integer :: i
real(kind=8) :: a(100), b(0:100)
!$OMP PARALLEL
!$OMP DO
do i=1,100
     b(i)=a(i)+b(i-1)
enddo
!$OMP END DO
!$OMP END PARALLEL
end
```

2スレッドの場合:

スレッド0

```
   do i=1,50
     b(i)=a(i)+b(i-1)
   enddo
```

スレッド1

```
       do i=51,100
     b(i)=a(i)+b(i-1)
   enddo
```

b(50)の結果がないと
本来実行できない

omp for (2)

注意 #pragma omp for はforループの中身が並列実行可能かどうかは関知せず、必ず分割してしまう。

```
int i;
double a[100], btemp[101];
double *b = btemp + 1;
#pragma omp parallel
{
#pragma omp for
  for (i=0:i<100;i++) {
    b[i]=a[i]+b[i-1];
  }
}
```

2スレッドの場合：

スレッド0

```
  for (i=0;i<50;i++) {
        b[i]=a[i]+b[i-1];
    }
```

スレッド1

```
     for (i=50;i<100;i++) {
        b[i]=a[i]+b[i-1];
     }
```

b[49]の結果がないと本来実行できない

OMP DO(3)

分割を規定する

```
integer :: i
real(kind=8) :: a(100), b(100)
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC,4)
do i=1,100
    b(i)=a(i)
enddo
!$OMP END DO
!$OMP END PARALLEL
end
```

1～100を4つのchunkにわけて、それをサイクリックに各スレッドに割り当てる

4スレッド実行時の
マスタスレッド担当行:
→1,2,3,4,17,18,19,20,

omp for (3)

分割を規定する

```
int i;
double a[100], b[100];
#pragma omp parallel
  {
#pragma omp for schedule(static,4)
  for(i=0; i<100; i++) {
      b[i]=a[i];
    }
  }
```

1～100を4つのchunkにわけて、それをサイクリックに各スレッドに割り当てる

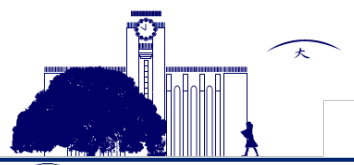
4スレッド実行時の
マスタスレッド担当行：
→1,2,3,4,17,18,19,20,

OMP Sections

```
!$OMP Sections
!$OMP Section
   計算1   (スレッド0)
!$OMP Section
   計算2   (スレッド1)
!$OMP Section
   計算3   (スレッド2)
!$OMP END Sections
```

```
#pragma omp sections
  {
  #pragma omp section
   { 計算1   (スレッド0)}
  #pragma omp section
   { 計算2   (スレッド1)}
  #pragma omp section
   { 計算3   (スレッド2)}
  }
```

Section毎にスレッドに仕事が割り当てられる。
Sectionの数よりもスレッド数が多い場合には
仕事をしないスレッドが発生する。

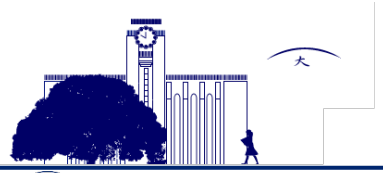


OMP Single

```
!$OMP Parallel
  並列処理
!$OMP Single
  逐次処理
!$OMP END Single
  並列処理
!$OMP END Parallel
```

```
#pragma omp parallel
   {     並列処理
#pragma omp single
   { 逐次処理 }
    並列処理
   }
```

一つのスレッドのみが処理を行う
（冗長実行を防ぐ）



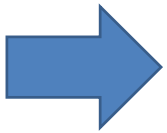
Work Sharing 実行イメージ

```
program  main
         :
!$OMP PARALLEL
!$OMP DO
   do i=1,100
     a(i)=i
   end do
!$OMP END  DO

!$OMP SECTIONS
  !$OMP SECTION
     call sub1
  !$OMP SECTION
     call sub2
!$OMP END  SECTIONS

!$OMP SINGLE
   call sub_s
!$OMP END SINGLE

   b(1)=a(1)
!$OMP END PARALLEL
         :
end  program main
```

複合パラレルWork-Sharing構造

記述方法のひとつ

例
```
!$OMP PARALLEL
!$OMP DO
      do i=1,n
            :
      end do
!$OMP END DO
!$OMP END PARALLEL
```



```
!$OMP PARALLEL DO
      do i=1,n
            :
      end do
!$OMP END PARALLEL DO
```

（!$OMP END PARALLEL DOは省略可）

同様に以下も記述できる

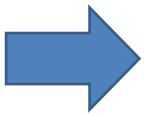
```
!$OMP PARALLEL SECTIONS
```

複合パラレルWork-Sharing構造

記述方法のひとつ

例

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



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

同様に以下も記述できる

```
#pragma omp parallel sections
```

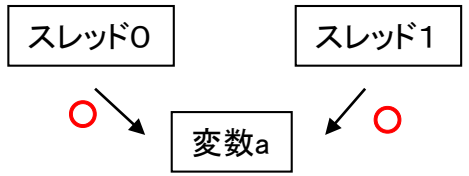
変数の属性

変数属性は大きくわけて次の2つ

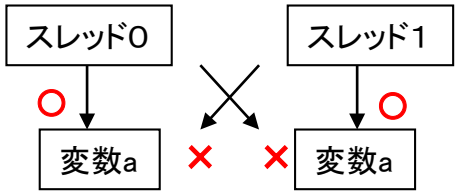
SHARED属性
- プログラムで1つの領域
- どのスレッドからでも参照、更新可能

PRIVATE属性
- スレッド毎に独立した領域
- 各スレッドからだけ参照、更新可能

Shared属性



Private属性



デフォルトは基本的にSHARED属性

OMP PARALLELやOMP DO, OMP SECTIONSでprivate変数を指定可能

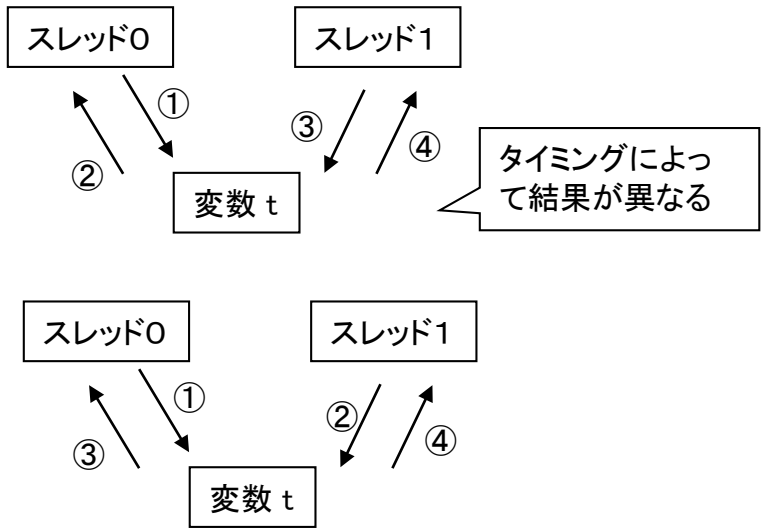
それぞれ並列リージョンやWork-sharing構造内でプライベート化（範囲外では不定の値）

PRIVATE属性であるべき変数

プログラム例

```
!$OMP PARALLEL DO
  do i=1,n
     t = i + 1
     a(i) = t + n
  end do
!$OMP END PARALLEL DO
```

変数tがshared属性だと…



変数tはPRIVATE属性でなくてはならない

属性の宣言と有効範囲

DO ループの制御変数 i はデフォルトでPRIVATE属性

```
!$OMP PARALLEL DO PRIVATE(t)
  do i=1,n
     t = i + 1
     a(i) = t + n
  end do
!$OMP END PARALLEL DO
  write(*,*) t     ! 不定
```

この範囲で t はPRIVATE属性

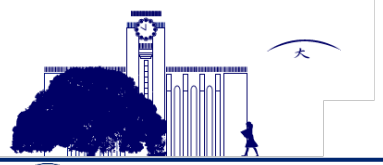
PRIVATE属性の変数は、有効範囲の外では不定

for ループの制御変数 i はデフォルトでPRIVATE属性

```
#pragma omp for private(t)
  for (i=0;i<n;i++) {
     t = i + 1;
     a[i] = t + n;
  }
  printf("%d¥n",t)  //不定
```

この範囲で t はPRIVATE属性

PRIVATE属性の変数は、有効範囲の外では不定



LASTPRIVATE属性（OMP DO, OMP SECTIONS）

最終の繰り返しの値を保存

```
!$OMP PARALLEL DO LASTPRIVATE(t)
  do i=1,n
     t = i + 1
     a(i) = t + n
  end do
!$OMP END PARALLEL DO
   write(*,*) t     ! n+1
```

この範囲で t はPRIVATE属性。
しかし、この範囲から抜けたときに、最後の繰り返しを担当したスレッドが持つ t の値が保持される。

n回目の繰り返しを担当したスレッドが持つtの値

逐次実行時と同じ値を保証

lastprivate属性（omp for, omp sections）

最終の繰り返しの値を保存

```
#pragma omp parallel for lastprivate(t)
  for (i=0;i<n;i++) {
     t = i + 1;
     a[i] = t + n;
  }
```

この範囲で t はPRIVATE属性。しかし、この範囲から抜けたときに、最後の繰り返しを担当したスレッドが持つ t の値が保持される。

```
 printf("%d¥n",t);    // n
```

n回目の繰り返しを担当したスレッドが持つtの値

逐次実行時と同じ値を保証

FIRSTPRIVATE属性（OMP PARALLEL, OMP DO, OMP SECTIONS）

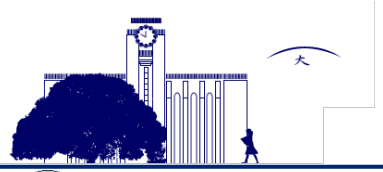
PRIVATE変数を直前の値で初期化する

```
t=1
!$OMP PARALLEL FIRSTPRIVATE(t)
  !$OMP DO
   do i=1,n
     if (a(i)>0) then
       t=t+1
     endif
   enddo
  !$OMP END DO
   write(*,*) t   !not不定
!$OMP END PARALLEL
```

Parallelリージョン内でPRIVATE変数

```
!$OMP PARALLEL
  t=1
  !$OMP DO FIRSTPRIVATE(t)
   do i=1,n
     if (a(i)>0) then
       t=t+1
     endif
   enddo
  !$OMP END DO
   write(*,*) t   !不定
!$OMP END PARALLEL
```

Work-sharing 構造内でPRIVATE変数
構造外で t は不定



firstprivate属性（omp parallel, omp for, omp sections）

PRIVATE変数を直前の値で初期化する

```
t=1;
#pragma omp parallel firstprivate(t)
{
#pragma omp for
   for (i=0;i<n;i++) {
     if (a[i]>0)
       t=t+1;
   }
   printf(“%d¥n”,t); //not不定
}
```

Parallelリージョン内でPRIVATE変数

```
#pragma omp parallel
{
  t=1;
#pragma omp for firstprivate(t)
   for (i=0;i<n;i++) {
     if (a[i]>0)
       t=t+1;
   }
   printf(“%d¥n”,t);   //不定
}
```

Work-sharing 構造内でPRIVATE変数
構造外で t は不定

REDUCTION属性

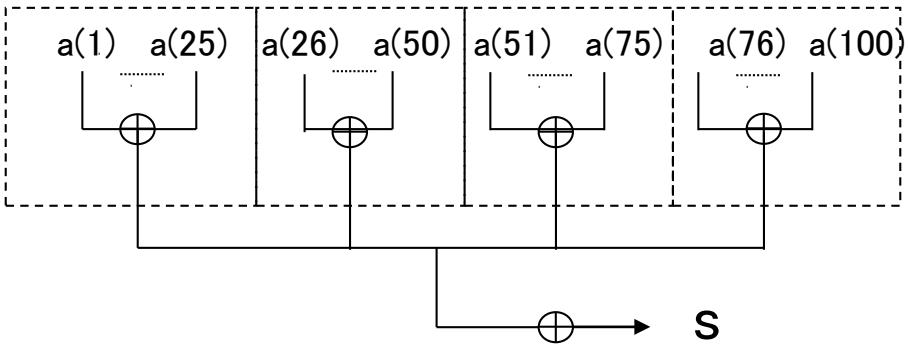
使用可能な演算子(operator)と組み込み関数(intrinsic)

- operator : ＋ , - , * , .and. , .or.
- intrinsic : max , min , iand , ior , ieor

形式

REDUCTION({ operator | intrinsic} : 変数名)

```
    s = 0
!$OMP PARALLEL DO REDUCTION(+:s)
    do i = 1, 100
       s = s + i
    end do
!$OMP END PARALLEL DO
    write(*,*) s      ! 5050
```



各スレッドで部分和を求めて、最後に加算

reduction属性

使用可能な演算子(operator)

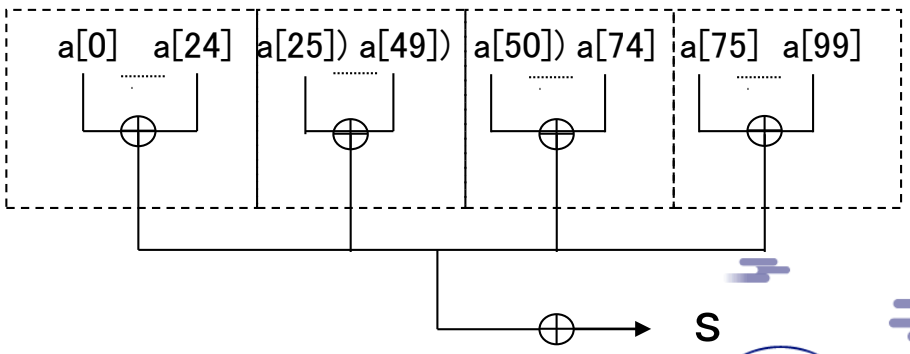
- operator : + , - , * , &, |, ^, &&, ||
- 対象の式 : s = s+式; s = 式+s; s+=式; s++; ++s;

形式

reduction(operator : 変数名)

```
    s = 0;
#pragma omp parallel for reduction(+:s)
    for (i=1;i<=100;i++) {
       s = s + i;
    }
    printf("%d¥n",s)    // 5050
```

各スレッドで部分和を求めて、最後に加算



具体的なプログラム例 (行列・行列積)

```
program matmul
  integer :: i,j,k
  integer,parameter :: n=1000
  real :: a(n,n),b(n,n),c(n,n)

call init(a,b,c)        ! 逐次実行

!$OMP PARALLEL DO PRIVATE(k,i)
do j=1,n                ! jループを分割して並列実行
  do k=1,n
    do i=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    end do
  end do
end do
!$OMP END PARALLEL DO

write(*,*) ”c(1,1)=”,c(1,1) ! 逐次実行

end program matmul
```

*!$OMP DO の直後のループの制御変数は自動的にプライベート化される

具体的なプログラム例 (行列・行列積)

```
int main(int argc, char **argv) {
  int i,j,k;
#define n 1000
  float a[n][n],b[n][n],c[n][n]

  init(a,b,c);       // 逐次実行

#pragma omp parallel for private(k,j)
  for(i=0;i<n;i++) {                    // iループを分割して並列実行
    for (k=0;k<n;k++) {
      for (j=0;j<n;j++) {
        c[i][j]=c[i][j]+a[i][k]*b[k][j];
      }
    }
  }

  printf("c[1][1]=%f¥n", c[1][1]);        // 逐次実行

}
```

*omp for の直後のループの制御変数は自動的にプライベート化される

具体的なプログラム例 (円周率の計算)

```
 program calculate_pi
   integer :: i, n
   real(kind=8) :: w, gsum, pi, v

   n=2000000000                  ! 逐次実行
   w = 1.0d0 / n                 ! 逐次実行
   gsum = 0.0d0                  ! 逐次実行

!$OMP PARALLEL DO PRIVATE(v)  REDUCTION(+:gsum)
   do i = 1, n                                ! iループを分割して並列実行
     v = (real(i,8) - 0.5d0 ) * w             ! 総和演算
     v = 4.0d0 / (1.0d0 + v * v)              !
     gsum = gsum + v                          !
   end do
!$OMP END PARALLEL DO
   pi = gsum * w
   write(*,*) ”PI is ”, pi
 end program calculate_pi
```

具体的なプログラム例 (円周率の計算)

```
int main(int argc, char **argv) {
    int i, n;
    double w, gsum, pi, v;

    n=2000000000;              // 逐次実行
    w = 1.0 / n;               // 逐次実行
    gsum = 0.0;                // 逐次実行

#pragma omp parallel for private(v) reduction(+:gsum)
    for (i=0;i<n;i++) {                    // iループを分割して並列実行
      v = (i - 0.5d0 ) * w;                // 総和演算
      v = 4.0 / (1.0 + v * v);             //
      gsum = gsum + v;                     //
    }
    pi = gsum * w;
    printf("PI is %.16f¥n", pi);
}
```

サブルーチンでの変数の属性

```
program main
  integer, parameter :: n=100
  integer :: i
  real :: a(n), x
  a=1.0
!$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:y)
  do i=1,n
    call sub0(a,i,x)
    y=y+x
  end do
!$OMP END PARALLEL DO
  write(*,*) y
end program main

subroutine sub0(a,i,x)
  integer, parameter :: n=100
  integer :: i
  real :: a(n),x
  real :: tmp
  tmp=a(i)+1
  x=tmp
  return
end subroutine sub0
```

(1)引数の変数の属性は受け継がれる

(2)サブルーチン内で定義された変数はPRIVATE属性

(3)大域変数はSHARED属性

(4)SAVE属性をもつ変数はSHARED属性

なお、(3),(4)はthreadprivate指示文によりPRIVATE属性にすることもできる

関数での変数の属性

```
   int main(int arg, char **argv) {
#define n 100
     int i;
     float a[n],x,y;
     for(i=0;i<n;i++) a[i]=1.0;
#pragma omp parallel for private(x) reduction(+:y)
     for (i=0;i<n;i++) {
        sub0(a,i,&x);
        y=y+x;
     }
     printf("%f¥n", y);
   }

   void sub0(float *a, int i, float *x) {
     float tmp;
     tmp=a[i]+1;
     *x=tmp;
     return:
   }
```

(1)引数の指示先の属性は受け継がれる

(2)関数内で定義された変数は
private属性

(3)大域変数はshared属性

(4)static属性をもつ変数は、shared属性

なお、(3),(4)はthreadprivate指示文により
private属性にすることもできる

同期と制御

バリア同期

チーム内のスレッドの到達を待つ

暗黙のバリア同期

- !$OMP END PARALLEL, Work-Sharing構文の後ろ

陽に指定 !$OMP BARRIER or #pragma omp barrier

バグを作らないためにできる限り活用する

複数スレッド間でshared変数のアクセス制御

!$OMP CRITICAL, !$OMP END CRITICAL

#pragma omp critical

- CRITICALセクションにはひとつのスレッドしか入れない

!$OMP ATOMIC or #pragma omp atomic

- 直後の実行文の左辺の変数に対するアクセスが逐次化

並列性能がでないのでなるべく使わないことが望ましい

OMP Barrier

```
  integer, parameter :: num=10
  integer :: a(num)
!$OMP PARALLEL
  do i=1,num
    a(i)=0.0
  enddo

!$OMP BARRIER

!$OMP DO
  do i=1,num
    a(i)=a(i)+1.0
  end do
!$OMP END DO
!$OMP END PARALLEL
```

!$OMP DOの前に暗黙の同期はとられないため、明示的にバリア同期を指示。

```
#define num 10
  int a[num];
#pragma omp parallel
{
  for(i=0;i<num;i++) {
    a[i]=0.0;
  }

#pragma omp barrier

#pragma omp for
  for(i=0;i<num;i++) {
    a[i]=a[i]+1.0;
  }

}
```

CRITICAL

配列の最大値とその位置および最小値とその位置を求める例

```
  integer, parameter :: n=10
  integer :: i,imax,imax_index,imin,imin_index,ia(n)
                                  :
   imax=ia(1);  imax_index=1
   imin=ia(1);  imin_index=1
!$OMP PARALLEL DO
   do i=2,n
!$OMP CRITICAL (maxlock)
     if ( ia(i) > imax ) then
       imax=ia(i)
       imax_index=I
     end if
!$OMP END CRITICAL (maxlock)
!$OMP CRITICAL (minlock)
     if ( ia(i) < imin ) then
       imin=ia(i)
       imin_index=i
     end if
!$OMP END CRITICAL (minlock)
   end do
!$OMP END PARALLEL DO
   write(*,*) "max index=",imax_index,"max value=",imax ,&
              "min index=",imin_index,"min value=",imin
```

- 2つのクリティカルセクション(maxlock,minlock)を生成
- 各クリティカルセクションには、同時に1スレッドのみが入れる

CRITICAL

配列の最大値とその位置および最小値とその位置を求める例

```
# define n 10
  int i,imax,imax_index,imin,imin_index,ia[n];

  imax=ia[0];  imax_index=0;
  imin=ia[0];  imin_index=0;
#pragma omp parallel for
  for(i=1;i<n;i++) {
#pragma omp critical (maxlock){
   if ( ia[i] > imax ) {
     imax=ia[i];
     imax_index=i;
   }
}
#pragma omp critical (minlock){
   if ( ia[i] < imin ) {
     imin=ia[i];
     imin_index=i;
   }
}

  }
  printf("max index=%d, max value=%d min_index=%d min value=%d¥n",
         imax_index, imax ,imin_index, imin);
```

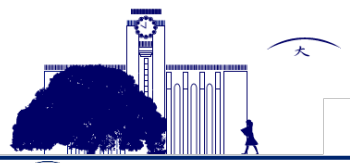
- 2つのクリティカルセクション(maxlock,minlock)を生成
- 各クリティカルセクションには、同時に1スレッドのみが入れる

ATOMIC

ヒストグラムを生成するプログラム例

```
  integer, parameter :: num=10
  integer :: i,a(num),histgram(0:10)
! 生徒数 はnum 人
! 配列 a には各生徒のテストの点数が入っている

! ヒストグラムの初期化
!$OMP PARALLEL DO
  do i=1,10
    histgram(i)=0
  end do
!$OMP END PARALLEL DO

! ヒストグラム作成
!$OMP PARALLEL DO
  do i=1,num
!$OMP ATOMIC
   histgram(a(i))=histgram(a(i))+1
  end do
!$OMP END PARALLEL DO
```

配列histgramの各要素に対し、同時に1スレッドのみが更新可能にする。



ATOMIC

ヒストグラムを生成するプログラム例

```
#define num 10
  int i,a[num],histogram[11];

// 生徒数 はnum 人
// 配列 a には各生徒のテストの点数0～10が入っている

// ヒストグラムの初期化
  for(i=0;i<=10;i++) {
    histgram[i]=0;
  }

// ヒストグラム作成
#pragma omp parallel for
  for(i=0;i<num;i++) {
#pragma omp atomic
    histgram[a[i]]++;
  }
```

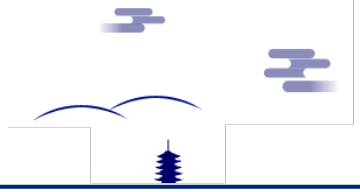
配列histgramの各要素に対し、同時に1スレッドのみが更新可能にする。

並列化を行う際のポイント

1. （当たり前だが）まず、何よりも、どの部分が並列化できるのかを考える!

2. 並列化に伴う変数属性の変更が必要かどうか調べる

3. ループが始まるタイミング、配列のアクセスなど、並列会に伴い同期が必要かどうか調べる



OpenMP利用時の時間計測

OpenMP実行でシリアル実行と同様に時間を計ると正確ではないことがあるため、以下を利用する。

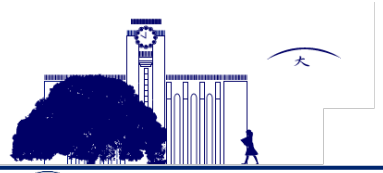
```
 use omp_lib
 real(kind=8) :: t0,t1
 t0 = omp_get_wtime()
!$OMP PARALLEL DO
  do i=1,n
    b(i)=a(i)*2.0
  enddo
  t1 = omp_get_wtime()
  print *, t1-t0
```

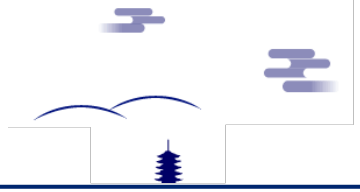
```
#include <omp.h>
double t0, t1;
t0 = omp_get_wtime();
#pragma omp parallel for
  for(i=0;i<n;i++) {
    b[i]=a[i];
  }
  t1 = omp_get_wtime();
printf("%f ¥n",t1-t0);
```

その他

コンパイル・実行方法など

参考資料





コンパイルと実行方法(サブシステム B)

コンパイル

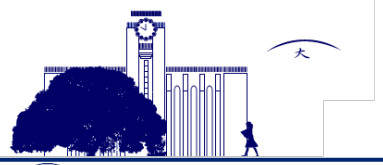
–qopenmp オプションをつける

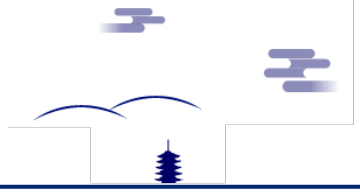
```
% ifort –qopenmp samp-omp.f       (Fortran)
% icc -qopenmp samp-omp.c        (C)
% icpc -qopenmp samp-omp.cpp    (C++)
```

実行

次の環境変数を指定する

- OMP_NUM_THREADS ：スレッド数を指定する。





サブシステムBにおける実行例

会話型（コマンドラインでそのまま実行）

% tssrun -q xxx -A c=4:t=4 ./a.out　　＊4コア・4スレッド実行の場合

バッチ型（実行内容をスクリプトに書いて、ジョブ投入）

```
#サンプルスクリプト（ 4コア・4スレッド実行）
#!/bin/bash
#============ PBS Options ============
#QSUB -q xxx
#QSUB -W 1:00
#QSUB -A p=1:t=4:c=4:m=4G
#============ Shel Script ============
cd $QSUB_WORKDIR
set -x

# automatically
# export OMP_NUM_THREADS=$QSUB_THREADS
./a.out
```

実行例(続き)

バッチ型（続き）

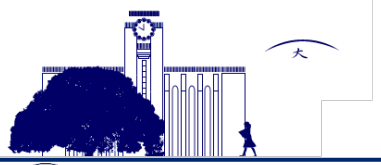
ジョブの投入　　% qsub sample.sh

ジョブの確認　　% qstat

ジョブのキャンセル　　% qdell <job ID>

(job IDはqstatコマンドで確認できる)

結果ファイル　　Bxxxxxx.xxxxx



参考資料など

「Parallel Programming in OpenMP」

- MORGAN KAUFMANN PUBLISHERS
- ISBN 1-55860-671-8

OpenMPホームページ

- http://www.openmp.org/
- 言語仕様書、サンプルプログラムなど。

オンラインマニュアル

- Fortran使用手引書
- C 言語使用手引書
- C++言語使用手引書
  https://web.kudpc.kyoto-u.ac.jp/