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 3）

講習の内容：Part 3

プログラミングでの注意事項の説明

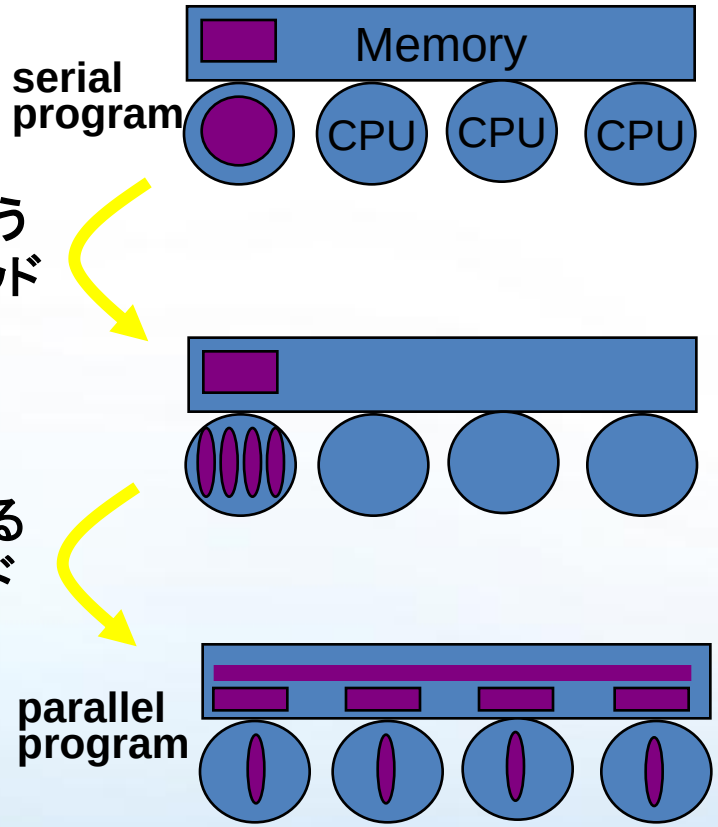
- 並列化における性能劣化の原因
- OpenMP利用上の注意点
- 自動並列化

OpenMPによる並列化について

プログラミングでの注意事項の説明

並列化における性能劣化の原因

- 並列化によって、逆に性能が劣化した場合の原因について



オーバヘッドの原因:

- 並列実行のスタートアップ
- 短いループ長
- 並列化のためにコンパイラが追加するコード
- 最内側ループの並列化
- 並列ループに対する最適化の阻害

- 不均等な負荷
- 同期処理
- メモリアクセス（ストライド）
- 共有アドレスへの同時アクセス
- 偽キャッシュ共有など

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

並列ループの選択

- 並列化の適用は、可能なかぎり粒度を大きくすること
  - ループでの並列化の場合には、最外側のループ
  - より大きなループカウントのループ
  - 関数やサブルーチンの呼び出しを含むループでも、その呼び出しを含んで並列化できるかどうかの検討が必要
- データの局所性の維持
  - 可能な限り、全ての共有データに対する各スレッドの処理を固定する
  - 複数のループを並列化し、それらのループで同一の共有データにアクセスするのであれば、並列化の適用を同じループインデックスに対して行う

並列化の阻害要因とその対策

- OpenMPによるマルチスレッド化をfor/DOループに適用して並列化する場合、ループ構造によって並列処理の適用ができない場合がある
  - ループの反復回数がループの実行を開始する時点で明らかになっている必要がある
  - 現在のOpenMPの規格では、whileループなどの並列化はできない
  - 並列処理の適用には十分な計算負荷が必要
  - 並列処理では、for/DOループの実行は相互に独立である必要がある

1.

```
do i=2,n
   a(i)=2*a(i-1)
end do
```

2.

```
ix = base
do i=1,n
  a(ix) = a(ix)*b(i)
  ix  = ix + stride
end do
```

3.

```
do i=1,n
   b(i)= (a(i)-a(i-1))*0.5
end do
```

プライベート変数

- OpenMP では、データはデフォルトでは共有される
  - プライベートとして、宣言する必要のあるデータの適切な指定（OpenMPには、幾つかのデータのプライベート宣言があるので、それらを正しく利用する）

```
common /blk/ l,m,n
!$omp THREADPRIVATE(/blk/)
!$omp parallel
!$omp& PRIVATE(x,y,z)
!$omp& FIRSTPRIVATE (q)
!$omp& LASTPRIVATE(i)
```

```
integer tmpArea[1000];
#pragma omp threadprivate(tmpArea)
#pragma omp parallel ¥
   private(x,y,z)    ¥
   firstprivate(q)   ¥
   lastprivate(i)
```

プライベート変数

- 呼び出したルーチンのローカル変数は自動的にプライベート
- 並列実行領域内で宣言された変数はプライベート

- 共通のアクセスパターン

PRIVATE：書き込みの後に読み取りを行う ワーク変数
FIRSTPRIVATE:最初の反復で読み取った後に書き込む変数
LASTPRIVATE:最後の反復の後に読み取る変数

プライベートなワーク配列の利用

- FORTRAN:既知のサイズの配列はプライベートとすることが可能
- C/C: ポインタをプライベートにした後、スタックへの配列の割り当て

```
    subroutine diffx(n,m)
    real work1(n,m)
!$omp parallel private(work1)
    [...]
!$omp end parallel
```

```
#include <alloca.h>
void diffx(int n, int m) {
    float* work1;
#pragma omp parallel ¥
            private(work1)
{
 work1=alloca(m*n*sizeof(float));
 [...]
}
```

共有変数

- OpenMPでは、指定がない変数は全て共有変数となる（一部の例外を除いて）

- インデックスが並列ループのインデックスと一致する配列データ
- 並列実行領域の読み取り専用変数
- 読み取りと書き込みの実行に同期が必要な変数（リダクション演算など）

配列データのリダクション処理

```
!$omp parallel private(hist1,i,j,ibin)
      do i=1,nbins
         hist1(i) = 0
      enddo
!$omp do
      do i=1,m
        do j=1,m
          ibin = 1 + data(j,i)*rscale*nbins
          hist1(ibin) = hist1(ibin) + 1
        enddo
      enddo
!$omp critical
      do i=1,nbins
        hist(i) = hist(i) + hist1(i)
      enddo
!$omp end critical
!$omp end parallel
```

グローバル変数

- プログラム全体で利用されるグローバル変数の取り扱い
  - ファイルスコープまたは名前空間スコープ（C/C++）あるいはCOMMON（Fortran）の変数は threadprivate を使用してプライベートにできる
- COMMONブロックまたは構造体で共有とプライベートが混在した変数の取り扱い
  - 必要な場合には、COMMONブロック中のデータや構造体を分割して、共有されるデータとプライベートなデータに分離

並列実行領域の融合

- 複数の並列実行領域を統合し、Fork-Joinのオーバーヘッドの低減を図る
- SINGLE構文などを利用

```
#pragma omp parallel
{
   work1();
}
   printf(...);
#pragma omp parallel
{
   work2();
}
```

```
#pragma omp parallel
{
   work1();
#pragma omp single
   printf(...);
   work2();
}
```

OpenMP利用上の注意点

- OpenMPは共有メモリでの並列プログラミングであり、共有メモリでのプログラミングでは、リソースの共有に伴うプログラムエラーの可能性がある
- シングルスレッドでのプログラミングでは、見られなかったエラーが引き起こされ、予期しなかった問題への対応が必要になる場合がある
  - RACE CONDITION
  - 偽共有 （False Sharing)

データ属性の指定ミス

- スレッドが個別に持つ必要のあるデータをプライベートデータとして、設定する必要のあるデータの設定ミスは、致命的なエラーとなる

```
void main ()
{       int i,id; double x, sum = 0.0;
#pragma omp parallel
        {       id = omp_get_thread_num();
#pragma for reduction(+:sum)
for (i=0;i<= N; i++){
x = func (i);
sum += x;
}
        res(id)= calc (sum);
        }
}
```

この例では、xをprivateとして定義していないため、各スレッドが共有変数のxを同時に更新するため、その値は不定となる。

このような不注意でのデータ環境の設定ミスは、OpenMPでの並列処理では、最も起こりやすい問題

同期処理のオーバーヘッド

- nowaitは、同期処理のオーバーヘッドの削減には、非常に効果的ですが、その利用には、注意が必要

```
void main ()
{       int i,id; double x, sum = 0.0;
#pragma omp parallel private(x)
        {       id = omp_get_thread_num();
#pragma omp for reduction(+:sum) nowait
for (i=0;i<= N; i++){
        x = func (i);
        sum += x;
}
        res(id)= calc (sum);
        }
}
```

reduction指示句を含むワークシェア構文に nowait を加えた場合、計算結果は不正になる

Race Condition

- 各スレッドが共有リソース（共有メモリ）の更新を行う場合には、その更新の順番で計算が変わるような場合、‘Race Condition’と呼び、並列処理では注意が必要

```
#pragma ompparallel sections
{
#pragma omp section
        a = b + c ;
#pragma omp section
        b = a + c ;
#pragma omp section
         c = a + b ;
}
```

このようなパラレルセクション構文の場合、各セクションの実行は並列に行われる。

その実行順序は、任意ですので、このプログラムをシングルスレッドで実行した場合とは、計算の順序が変わる可能性がある

ループ実行での依存性

- DO/forループは反復計算となりますが、その反復計算の順序が計算の整合性に影響を与えないことが必要

```
for(i=1; i<n; ++i)          // loop1
    a[i]=a[i-1]+b[i];

for(i=0; i<n; ++i)          // loop2
    x[i]=x[i+1]+b[i];
```

loop1はI番目の計算にi-1番目の計算結果が必要

loop2はloop1と違ってI番目の計算にはi-1番目の計算結果を参照することはありませんが、i+1番目の計算結果を参照することになり、整合性を取る必要がある

ループ実行での依存性

（計算式）

```
for(i=1; i<n; ++i){

  a[インデックスの計算式1]=
        =a[インデックスの計算式2] + …….

}
```

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生じる

ループ実行での依存性

- ループを並列化する場合には、各ループにこのような相互依存がなく、どのような順番で実行しても計算結果に整合性があることが必要
  - コンパイラは、間違って依存関係のあるループをOpenMPで並列化しても、その整合性のチェックは行わない
- 配列の総和計算のようなリダクション演算では、実際には各反復計算の結果が相互参照される

（例）

```
 RESULT= 0.0
!$OMP PARALLEL DO
!$OMP& REDUCTION(+:RESULT)
      DO I = 1, N
        RESULT = RESULT + (A(I) * B(I))
      ENDDO
```

OpenMPの指示句でリダクション演算の指定をすることでこの見かけの依存性を排除することが可能となり、正しく並列化することが可能

ループ実行での依存性

- 並列処理では、forループの実行は相互に独立であることが必要

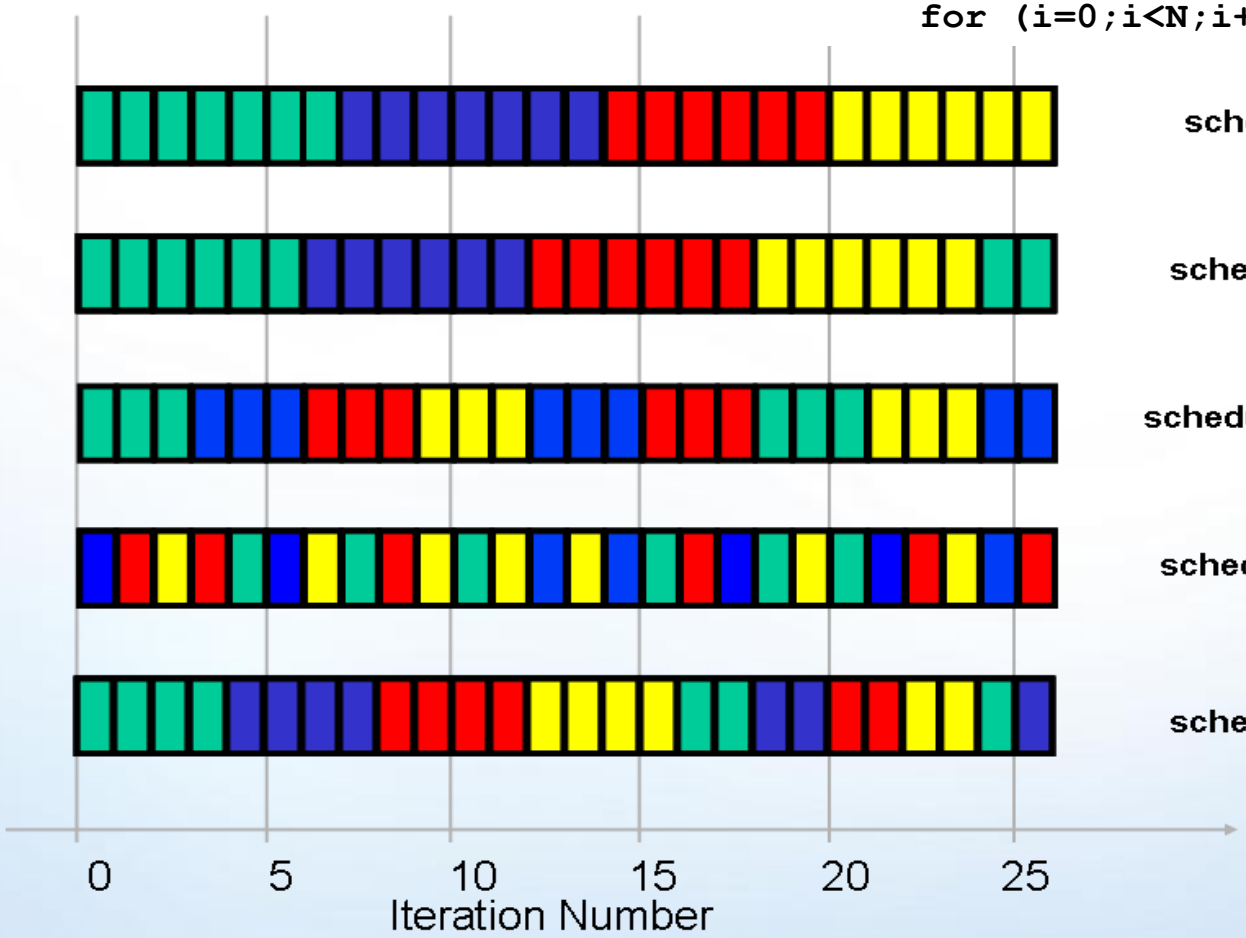
（例）

```
for(i=1; i<n; ++i)       // loop1
    a[i]=a[i-1]+b[i];
for(i=0; i<n; ++i)       // loop2
    x[i]=x[i+1]+b[i];
```

- 例えば、loop1はi番目の計算にi-1番目の計算結果が必要であり、ループの計算順序を変えることは出来ません。
- loop2はloop1と違ってi番目の計算にはi-1番目の計算結果を参照することはありませんが、i+1番目の計算結果を参照することになり、各ループの反復は相互に依存します。

スケジュールオプション

```
#define N 26
#define NUM_THREADS 4
void main ()
{ omp_set_num_threads(NUM_THREADS);
#pragma omp for schedule(type[,chunk ])
     for (i=0;i<N;i++) {      ….       }
```



計算粒度

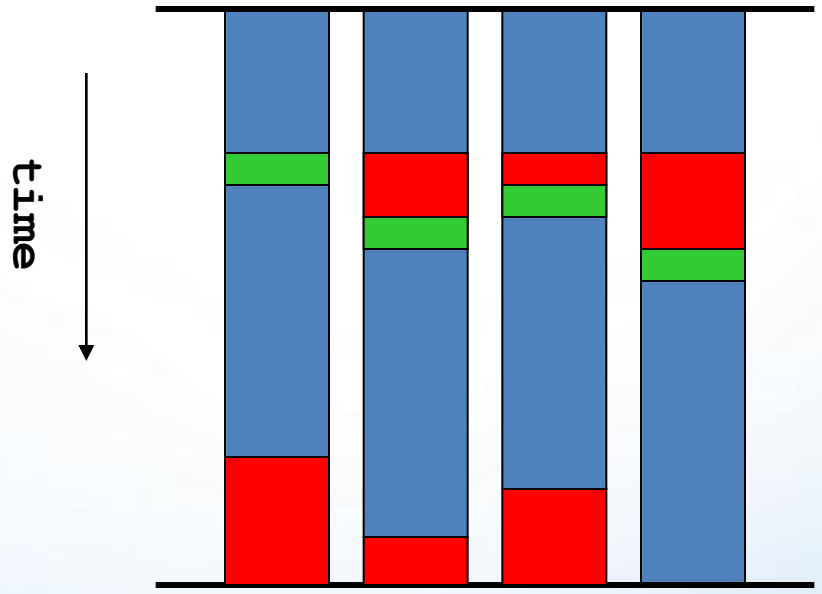
- 計算粒度：並列計算における処理レベルや並列タスクの仕事量
  - OpenMPでの並列処理では、細粒度から疎粒度まで、広範囲な粒度での並列処理が可能
  - マルチスレッドでの並列処理においては、その並列化の粒度について、理解することが必要
- 細粒度
  - プログラム中の計算ループで並列化するような場合
  - 粒度が小さ過ぎると実際の計算よりもスレッドの切替や通信のオーバヘッド、スレッド間での同期処理の負荷が大きく効率がよくない
- 疎粒度
  - 関数やサブルーチン呼び出しなどを含むプロシージャ間での並列処理
  - 粒度が粗すぎると、負荷の不均衡のためにパフォーマンスが低下する可能性がある

同期処理

- 並列処理においては、複数のスレッドが同時に処理を行うため、スレッド間での同期処理が必要
- OpenMPでは、全てのワークシェアリングの終了時に同期処理を行う
  - プログラムによっては、不要な同期処理がプログラム中に加えられる可能性がある
- クリティカルセクションのような並列実行領域内での排他制御も、スレッド数が多くなると性能に大きな影響を与えることになる

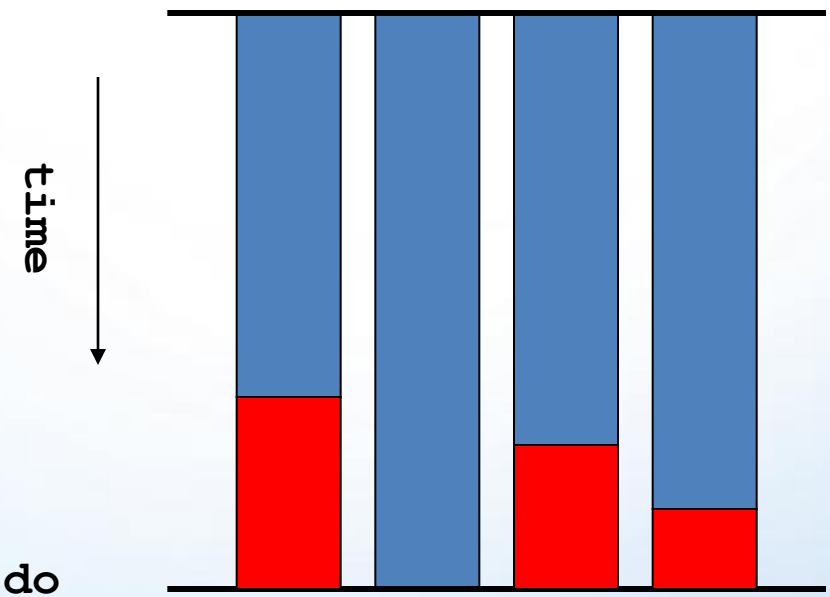
同期処理

```
!$omp parallel

!$omp critical

!$omp end critical



!$omp end parallel
```



ロードバランス

```
!$omp parallel do
```



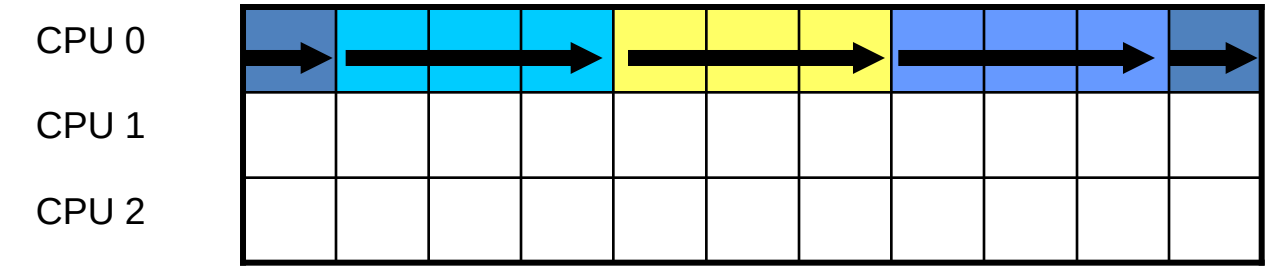

```
!$omp end parallel do
```

ロードバランス

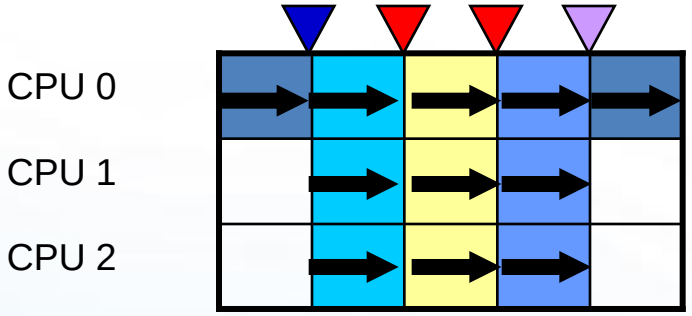
- 並列タスクの処理量は、可能なかぎり各スレッドに当分に分散することが必要
- 並列計算の速度向上は、もっとも時間のかかるタスクの処理時間に依存する
  - 各タスクのワークロードのバランスが悪い場合には、スレッドの待ち時間が大きくなるという問題が発生
  - マルチスレッドでの並列処理では、各スレッドが等分なワークロードを処理することが理想
- 対策
  - ワークロードの陽的な分散、スケジューリングオプションなど

計算粒度：細粒度と疎粒度での並列処理

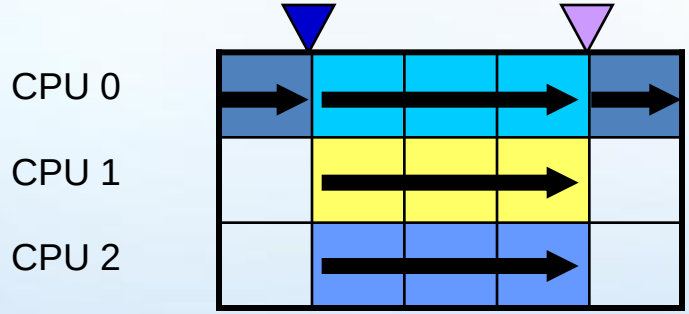
逐次処理



並列処理：細粒度での並列処理

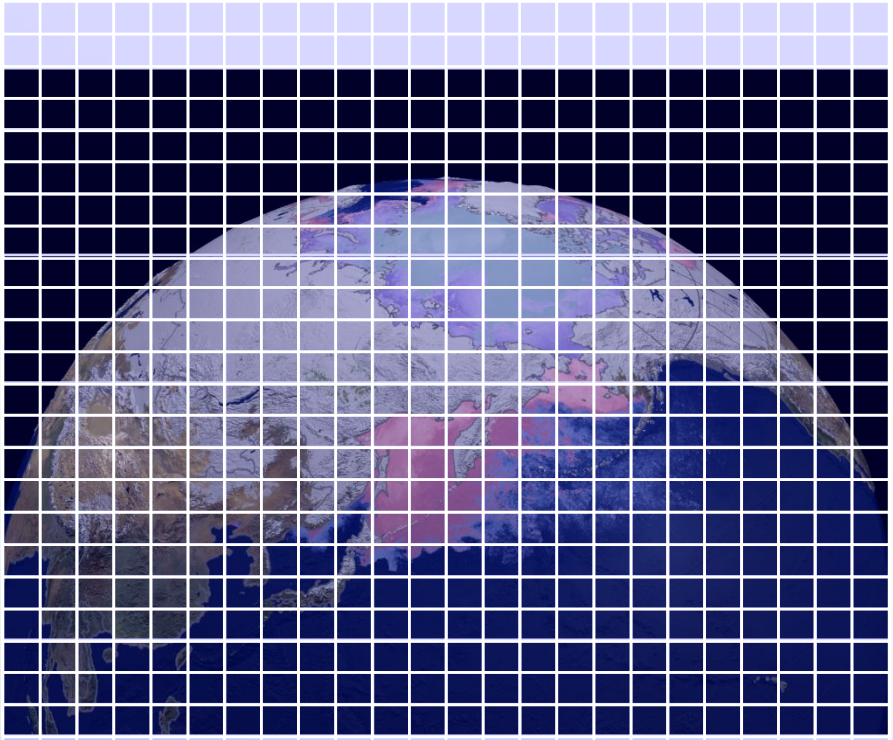


並列処理：疎粒度での並列処理

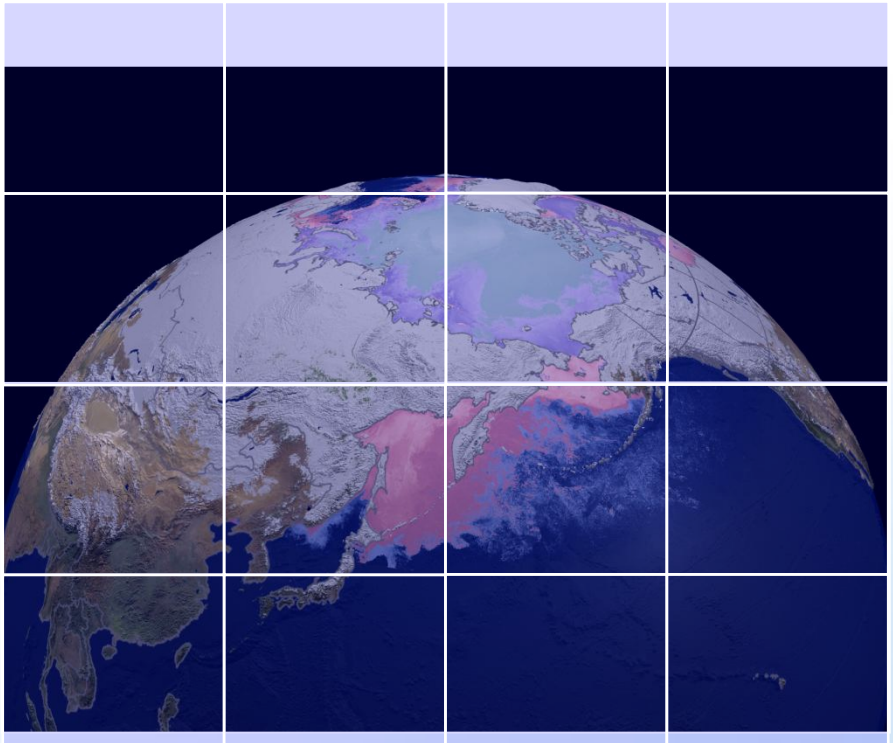


計算粒度 – 領域分割

細粒度での領域分割



疎粒度での領域分割

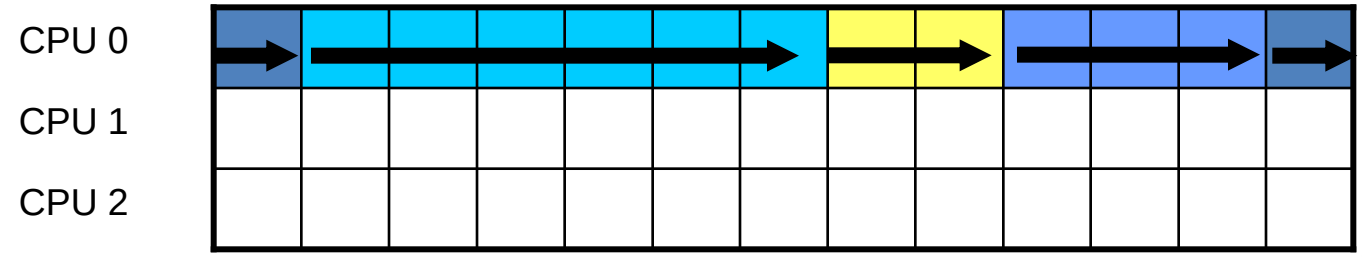


Graphics courtesy of NASA/Goddard Space Flight Center Scientific Visualization Studio

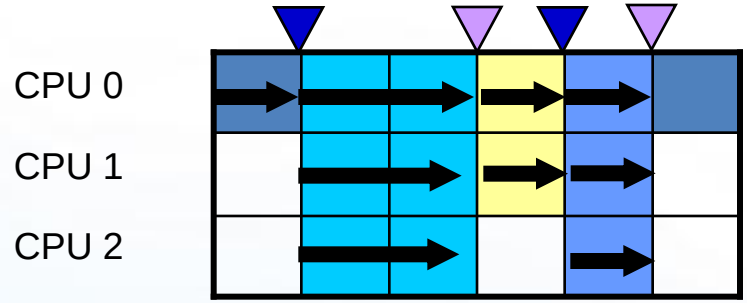
計算粒度を模式的に示した図：領域を例えばスレッド数に分割し、それぞれの領域を個々のスレッドが計算するような場合は疎粒度での領域分割となります。一方、領域を細かく分割し、各スレッドが複数の領域を順次処理するような場合は細粒度での領域分割となります。

計算粒度とワークロードの分散

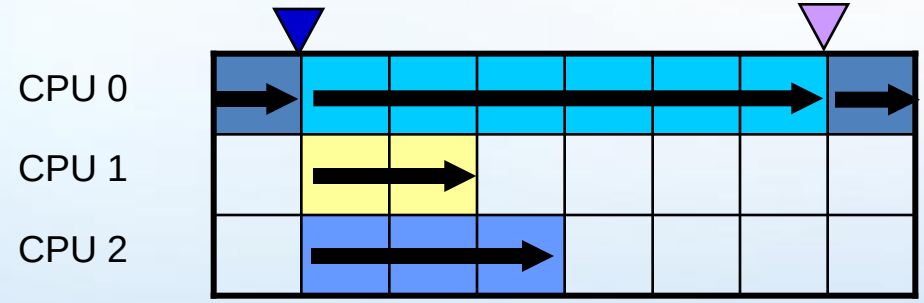
逐次処理



並列処理:細粒度での並列処理



並列処理:疎粒度での並列処理



ネストしたループ

- 外部ループが小さな場合は内部ループを使用する
- ネストしたループの外側に並列実行領域を定義する
- nowait を使用して不要なバリアを削除する

```
void copy(int imx, int jmx, int kmx,
          double**** w,double**** ws)
{
int k, j, i, nv;
#pragma omp parallel private(k,j,i,nv)
  for (nv = 0; nv < 2; nv++)
  #pragma omp for nowait
    for (k = 0; k < kmx; k++)
      for (j = 0; j < jmx; j++)
        for (i = 0; i < imx; i++)
          ws[nv][k][j][i] = w[nv][k][j][i];
}
```

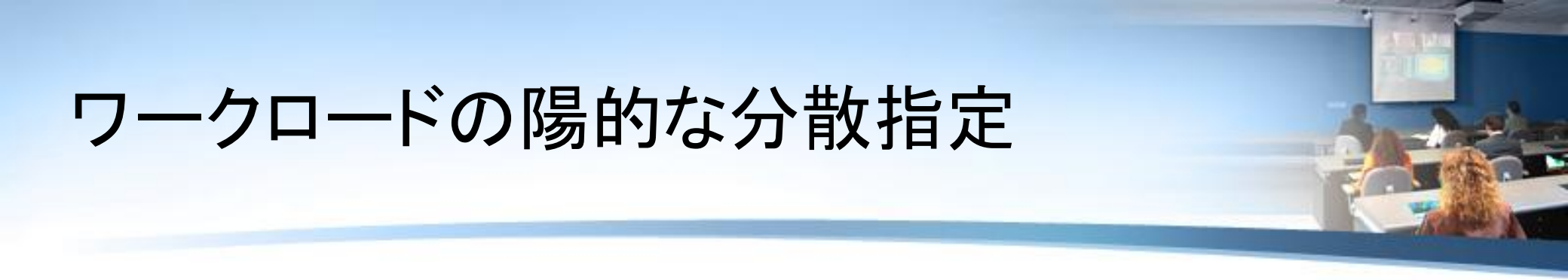
ネストした並列ループ

- ループ長の短い複数のループを統合し、そのループを並列化
  - 並列度の向上
  - ロードバランスの改善

```
      subroutine copy(jmx,kmx,w,ws)
      dimension ws(jmx,kmx)
!$omp parallel do private(k,j)
      do k = 1,kmx
         do j = 1,jmx
            ws(j,k) = w(j,k)
         end do
      end do
```

```
      subroutine copy(jmx,kmx,w,ws)
      dimension ws(jmx,kmx)
!$omp parallel do
      do kj = 1,kmx*jmx
         k = (kj-1)/jmx + 1
         j = kj - (k-1)*jmx
         ws(j,k) = w(j,k)
      end do
```

ワークロードの陽的な分散指定

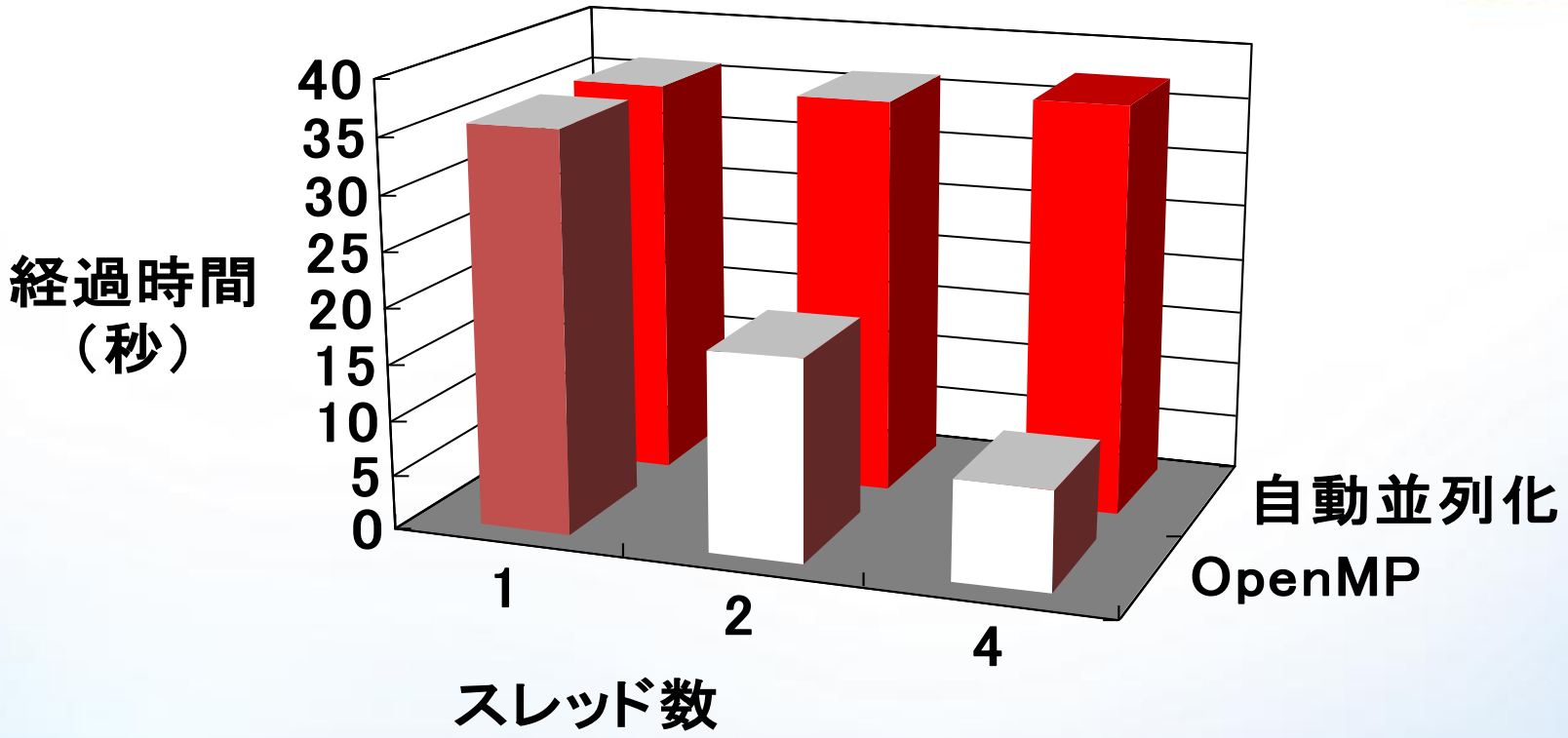


- DO/for構文を利用した場合、DO/forループの反復は指定された分配方法で各スレッドが実行
- OpenMPライブラリを利用してスレッド番号とスレッド数を利用して陽的にforループを各スレッドに割り振ることも有効
  - ユーザが陽的に各スレッドのワークロードを制御
  - 動的なワークロードの制御

スケジューリングオプション

- プログラムの並列実行をユーザが制御
- ワークシェアリング構文でサポートされているスケジューリングオプションの利用
  - ワークシェアリングでは、ループの各反復を各スレッドに分配するが、その分配方法を指定

自動並列化とOpenMP



- OpenMP_ホームページ（http://www.openmp.org）にある_OpenMP_のサンプルプログラム md.f をOpenMP_で並列化したものと自動で並列化したものを比較

自動並列化での性能向上

- プログラム内の多くのループは自動で並列化されている

```
$  ifort -O3 -parallel -par_threshold0  md.f
md.f(35) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(98) : (col. 8) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(161) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(181) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
md.f(212) : (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.
```

- なぜ、自動並列化では性能が向上しないのか？

OpenMPでの並列実行領域の指定

```
 96        do i=1,np
 97      ! compute potential energy and forces
 98          f(1:nd,i) = 0.0
 99          do j=1,np
100               if (i .ne. j) then
101                 call dist(nd,box,pos(1,i),pos(1,j),rij,d)
102      ! attribute half of the potential energy to particle 'j'
103                 pot = pot + 0.5*v(d)
104                 do k=1,nd
105                   f(k,i) = f(k,i) - rij(k)*dv(d)/d
106                 enddo
107               endif
108          enddo
109      ! compute kinetic energy
110          kin = kin + dotr8(nd,vel(1,i),vel(1,i))
111       enddo


148        subroutine dist(nd,box,r1,r2,dr,d)
.............
160        d = 0.0
161        do i=1,nd
162          dr(i) = r1(i) - r2(i)
163          d = d + dr(i)**2.
164        enddo
165        d = sqrt(d)
166
167        return
168        end
```

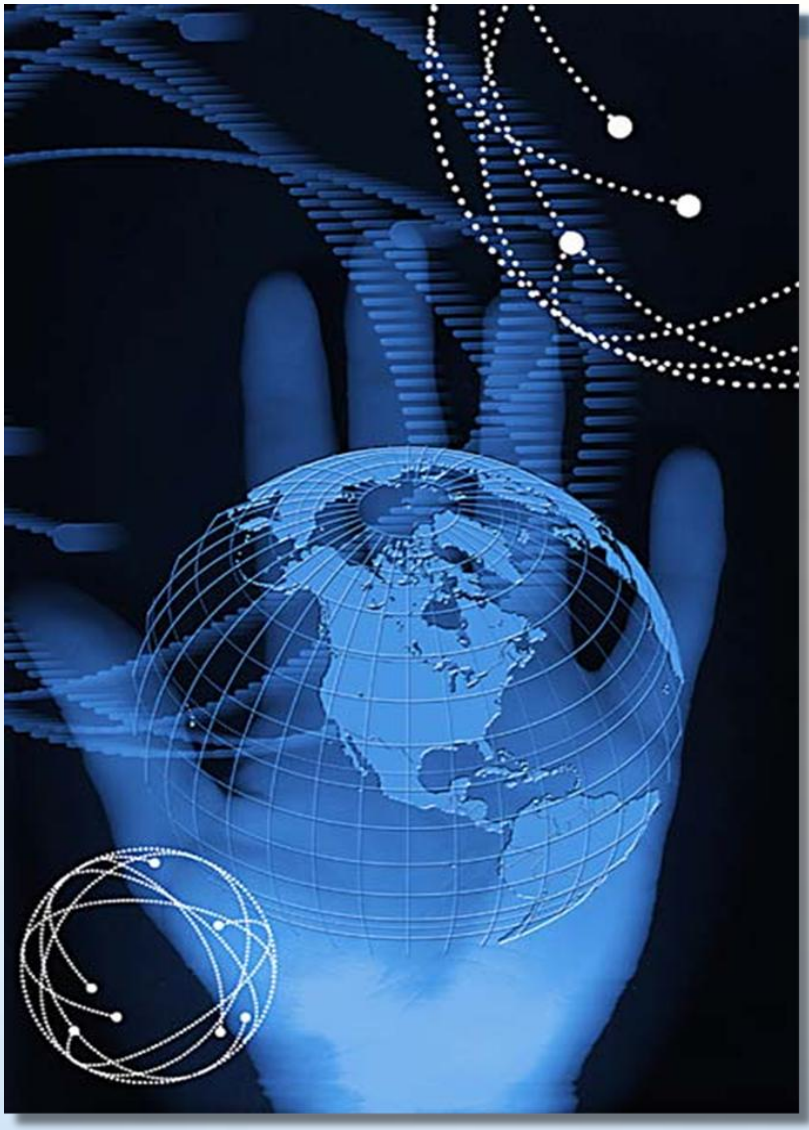
赤字の部分は自動並列化
OpenMPでの並列化では、以下のような領域の並列実行を指定可能

```
!$omp  parallel do
!$omp& default(shared)
!$omp& private(i,j,k,rij,d)
!$omp& reduction(+ : pot, kin)
      do i=1,np
! compute potential energy and forces
        f(1:nd,i) = 0.0
        do j=1,np
             if (i .ne. j) then
               call dist(nd,box,pos(1,i),pos(1,j),rij,d)
! attribute half of the potential energy to particle 'j'
               pot = pot + 0.5*v(d)
               do k=1,nd
                 f(k,i) = f(k,i) - rij(k)*dv(d)/d
               enddo
             endif
        enddo
! compute kinetic energy
        kin = kin + dotr8(nd,vel(1,i),vel(1,i))
      enddo
!$omp  end parallel do
      kin = kin*0.5*mass
```

OpenMPでの最適化について

- プログラムの並列化は、もちろん、最優先課題
- シングルプロセッサの最適化も必要
  - データの局所性
  - キャッシュデータの再利用
  - メモリ階層の有効利用
- 同期処理を出来るだけ少なくする
  - OpenMPは、ワークシェア構造に同期処理を自動で挿入（不要な場合には、NOWAITの追加）
  - クリティカルセクションやアトミックアップデートは、負荷の大きなオペレーション
  - SPMDプログラムやデータのプライベート化の検討

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