コンパイラーによる並列化機能

ソフトウェア＆ソリューションズ統括部

ソフトウェア製品部

Rev 12/26/2006

コースの内容

並列計算

- なぜ使用するのか?

OpenMP* 入門

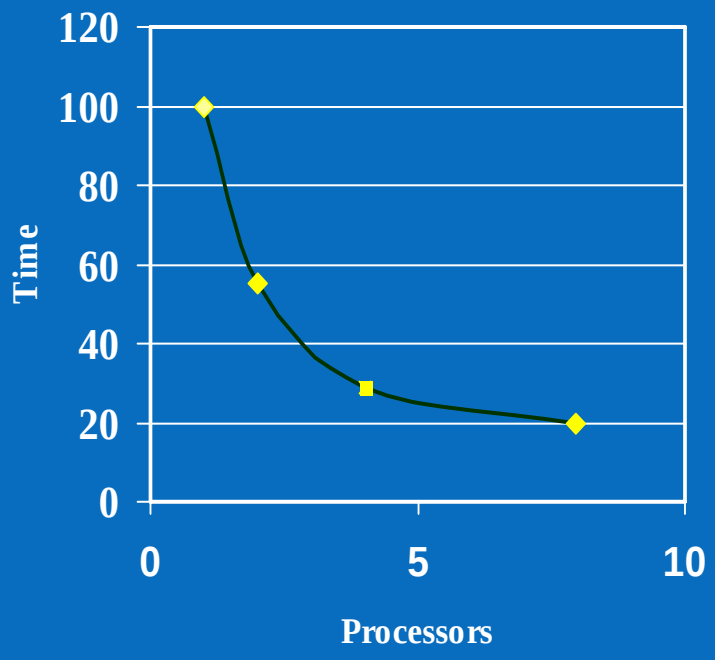
- 宣言子と使用方法
- 演習: Hello world と円周率の計算

並列プログラミング: ヒントとテクニック

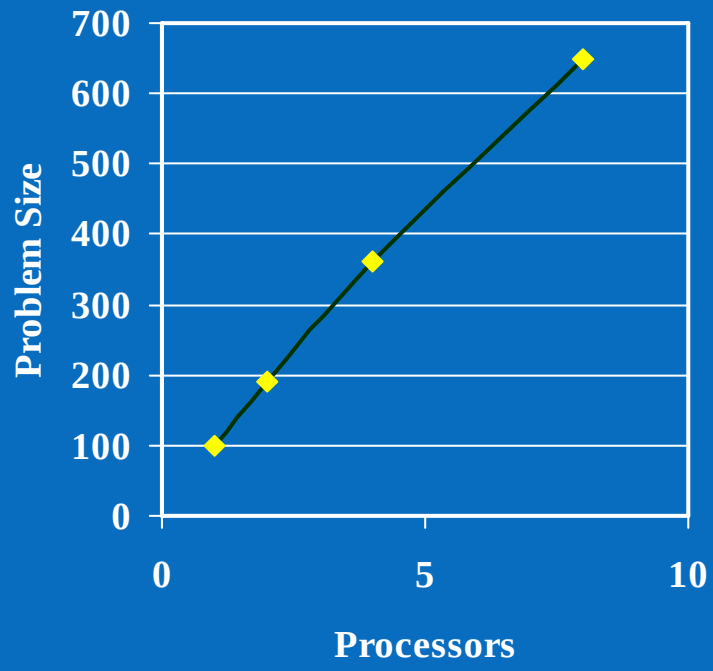
- コード開発で避けるべきこと

なぜ並列処理を使用するのか?

計算をより短い時間で処理



一定の所要時間でより大きな計算を処理



ほとんどのコードには並列処理可能なコードが含まれる

タスク並列処理:

独立したサブプログラム

```
call fluxx(fv,fx)
call fluxy(fv,fy)
call fluxz(fv,fz)
```

データ並列処理:

独立したループ反復

```
for (y=0; y<nLines; y++)
   genLine(model,im[y]);
```

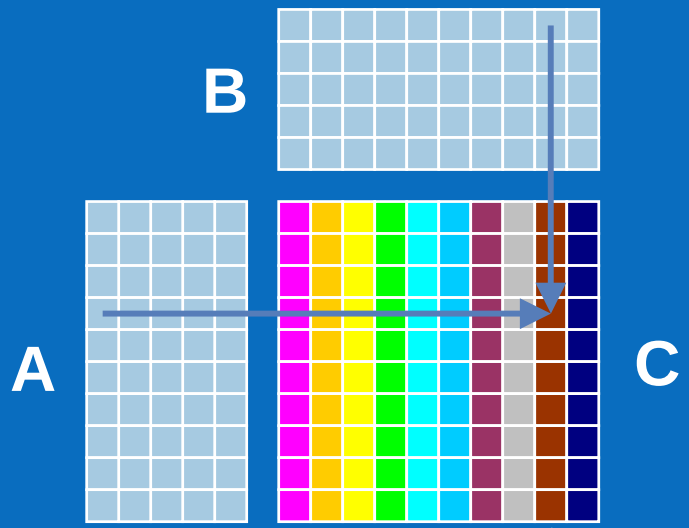
データ並列処理

- 並列処理が最も有効な形式
- 同時に計算できるデータ集合に依存する
- 通常は、大きなループのネストで見つかる

```
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        C[i][j] = 0.0;
```

```
for (i=0; i<M; i++)
    for (k=0; k<L; k++)
        for (j=0; j<N; j++)
            C[i][j] += A[i][k]*B[k][j];
```

例: 行列の乗算




```
for (i=0; i<M; i++)
   for (j=0; j<N; j++)
      C[i,j] = 0.0;

for (i=0; i<M; i++)
   for (k=0; k<L; k++)
      for (j=0; j<N; j++)
         C[i,j] += A[i,k]*B[k,j];
```

列はそれぞれ別々に計算できる

共有メモリー並列処理

マルチスレッド:

- 同時に実行する
- 単一アドレス空間で共有する
- 統一された方法で作業を共有する
- OS によってスケジューリングされる

共有メモリーと複数の CPU が利用可能なシステムが必要

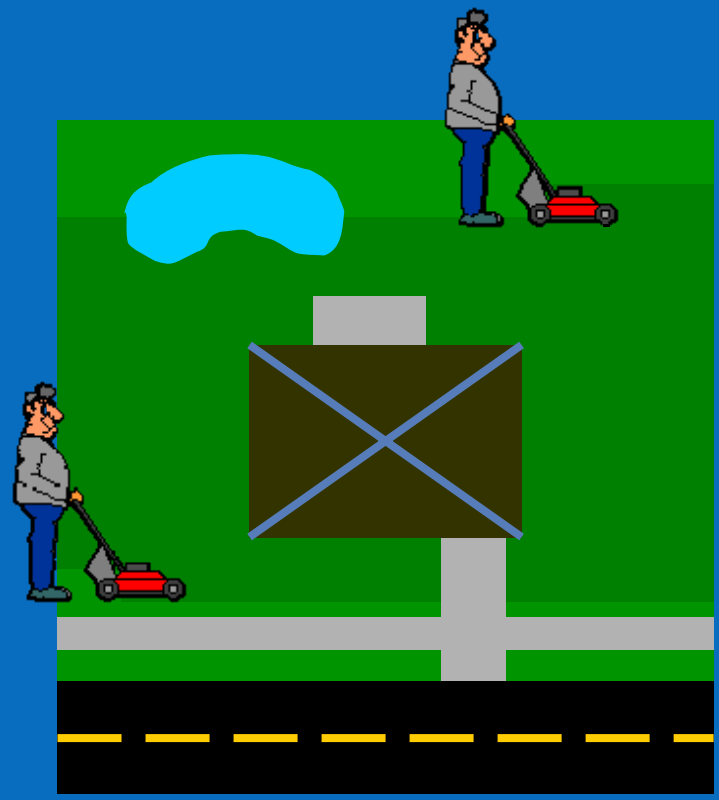
並列処理のポイント

同時処理可能な作業を識別

作業を均等に分割

一般に使用されるリソースのプライベート・コピーを作成する

コストのかかる、または一意の共有リソースへのアクセスを同期させる



並列モデルの比較

	MPI	スレッド	OpenMP*
可搬性	✓		✓
スケーラブル	✓	✓	✓
パフォーマンス指向	✓		✓
並列データのサポート	✓	✓	✓
インクリメンタル並列処理			✓
高レベル			✓
直列コードの保持			✓
正当性の確認			✓
分散メモリー	✓		

コースの内容

並列計算

- なぜ使用するのか?

OpenMP* 入門

- 宣言子と使用方法
- 演習: Hello world と円周率

並列プログラミング: ヒントとテクニック

- コード開発で避けるべきこと

3 つの主要な並列化テクノロジー

スレッド・ライブラリー
- Win32* API
- POSIX スレッド

メッセージ・パッシング・ライブラリー
- メッセージ・パッシング・インターフェイス（MPI）

コンパイラー・ディレクティブ
- OpenMP*: ポータブルな共有メモリー並列処理

OpenMP* とは?

www.openmp.org

ポータブルな、共有メモリー型のマルチプロセッシング
アプリケーション・プログラム・インターフェイス（API）

–Fortran 77、Fortran 90、C、および C++

–Linux* および Windows* 用の複数のベンダーをサポート

ループレベルの並列処理を標準化

粗粒度の並列処理をサポート

シングルソースに直列コードと並列コードを混在

15 年間の対称マルチプロセッシング（SMP）の経験を標準化

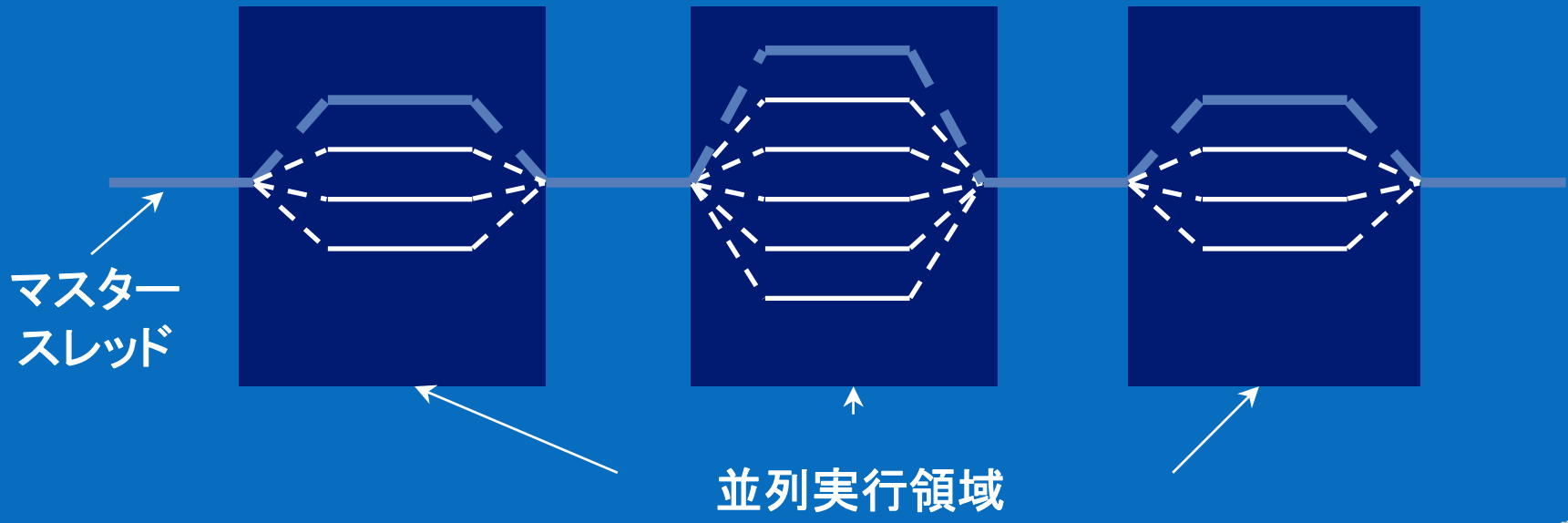
アーキテクチャー

1. Fork-join モデル
2. ワークシェアリング構文
3. 同期構文
4. ディレクティブ/プラグマベースの並列処理
5. より細かい制御が可能な拡張 API

プログラミング・モデル

Fork-join 型の並列処理:

- マスタスレッドは必要に応じて、スレッドのチームを生成する
- 並列処理は動的に追加される。つまり、シリアル処理プログラムは並列処理プログラムへ進化する



ループの並列化

最も時間がかかるループを特定する

そのループをスレッド間で分割する

このループを複数のスレッド間で分割する

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

逐次処理プログラム

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

並列処理プログラム

概要: スレッドはどのように対話するか?

- OpenMP は共有メモリーモデル
  - スレッドは変数を共有して対話する
- 意図しないデータの共有はデータの競合を発生させる
  - データの競合: スレッドが異なってスケジュールされたためにプログラムの結果が異なる場合
- データ競合を制御するには...
  - 同期を使用してデータの矛盾を防ぐ
- 同期処理が大変...
  - 同期で必要な最小限のアクセスになるようにデータのアクセス方法を変更する

構文について説明する前に

OpenMP の構文のほとんどは、コンパイラー宣言子またはプラグマで記述

- C および C++ の場合のプラグマの形式:
  #pragma omp *construct [clause [clause]…]*
- Fortran の場合の宣言子の形式(次のいずれか):
  C$OMP *construct [clause [clause]…]*
  !$OMP *construct [clause [clause]…]*
  *$OMP *construct [clause [clause]…]*

インクルード・ファイルと OpenMP ライブラリーモジュール

```
#include “omp.h”
use omp_lib
```

内容

OpenMP 構文は 5 つのカテゴリーに分けられる

- ランタイム関数/環境変数
- 並列実行領域
- ワークシェアリング
- データ環境
- 同期

OpenMP は C/C++ と Fortran では本質的に同じ

基本的な構文

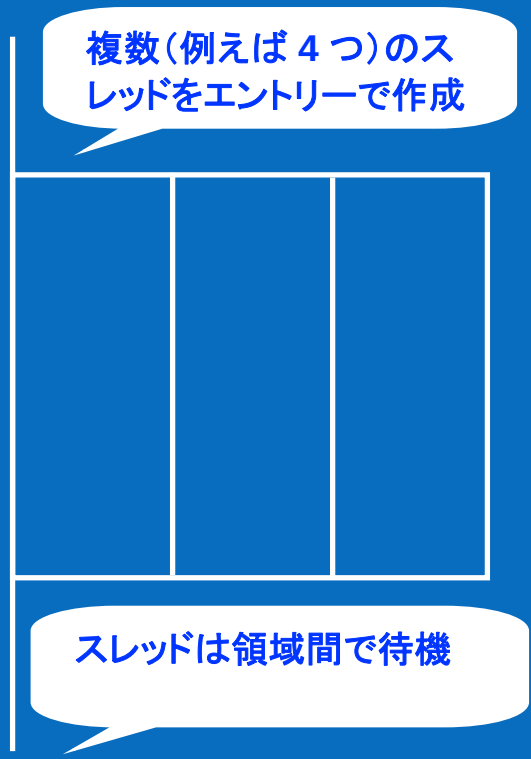
Fork-join モデル

アプリケーションは、逐次セクションと並列セクションで構成される

スレッドは、‘parallel’ プラグマが組み合わされて作成される

データは、スレッド間の共有または各スレッドへのプライベートとして分類される

```
main() {
#pragma omp parallel
    {
        // この範囲のコードを並列処理
        ...
    }
}
```



ライブラリールーチン

ランタイム環境ルーチン:

- スレッドの数を修正/確認する
  - `omp_set_num_threads()`
  - `omp_get_num_threads()`
  - `omp_get_thread_num()`
  - `omp_get_max_threads()`
- 並列実行領域かどうかを確認する
  - `omp_in_parallel()`
- システムにあるプロセッサーの数を確認する
  - `omp_get_num_procs()`

ライブラリールーチン

プログラムで使用するスレッドの数を修正する

- 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);
     }
}
```

プロセッサーの数と同じ数のスレッドを要求する

メモリーストアがアトミックでないため、この操作を保護する

環境変数

使用するスレッドのデフォルト数を設定する

- OMP_NUM_THREADS *int_literal*

“omp for schedule(RUNTIME)” ループがどのようにスケジュールされるかを制御する

- OMP_SCHEDULE “schedule[, chunk_size]”

構造ブロック(C / C++)

OpenMP* 構文のほとんどは構造ブロックに用いる

- 構造ブロック: 1 つの開始点と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()

```
#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");
```

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;
```

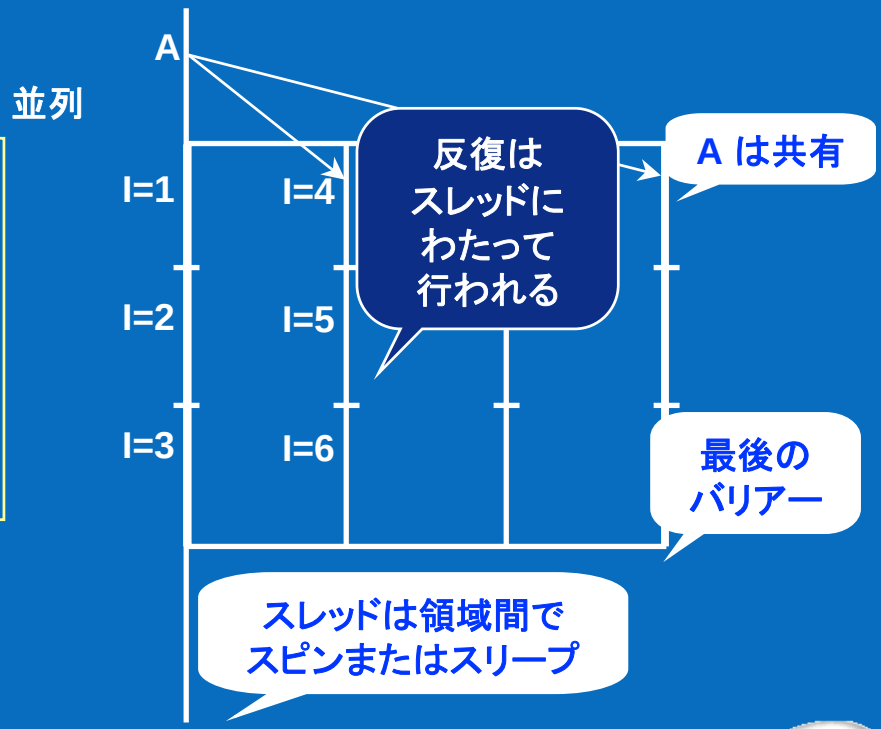
構造ブロックではない

並列ループにおけるデータモデル

スレッドが作成される

データは、共有またはプライベートとして分類される

```
void* work(float* A) {
   omp_set_num_threads(4);
#pragma omp parallel for
   for(i=1; i<=12; i++) {
 /* 各ループはスレッドに分配される */
   }
}
```



内容

OpenMP 構文は 5 つのカテゴリーに分けられる

- ランタイム関数/環境変数
- 並列実行領域
- ワークシェアリング
- データ環境
- 同期

OpenMP は C/C++ と Fortran では本質的に同じ

ワークシェアリングの内容

“for” ワークシェアリング構文は、チームのスレッド間のループ反復を分割する

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

デフォルトでは、“omp for” の最後にバリアーがあるため、“nowait” 句を使用してバリアーをオフにする

ワークシェアリング構文

動機付けの例

逐次コード

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

OpenMP 並列実行領域

```
#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];}
}
```

OpenMP 並列実行領域とワークシェアリング for 構文

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

OpenMP* 入門

for / do 構文:
schedule 句

schedule 句は、ループ反復をどのようにスレッドにマップするか制御する

- schedule(static [,chunk])
  - 各スレッドに、サイズ反復のブロック "チャンク" を加える
- schedule(dynamic[,chunk])
  - 各スレッドは、すべての反復が処理されるまで、キューから "チャンク" を得る
- schedule(guided[,chunk])
  - スレッドは、動的に反復のブロックを得る。ブロックのサイズは最初は大きく、計算が進むとともに、"チャンク" サイズになる
- schedule(runtime)
  - スケジュールおよびチャンクサイズは OMP_SCHEDULE 環境変数から得られる

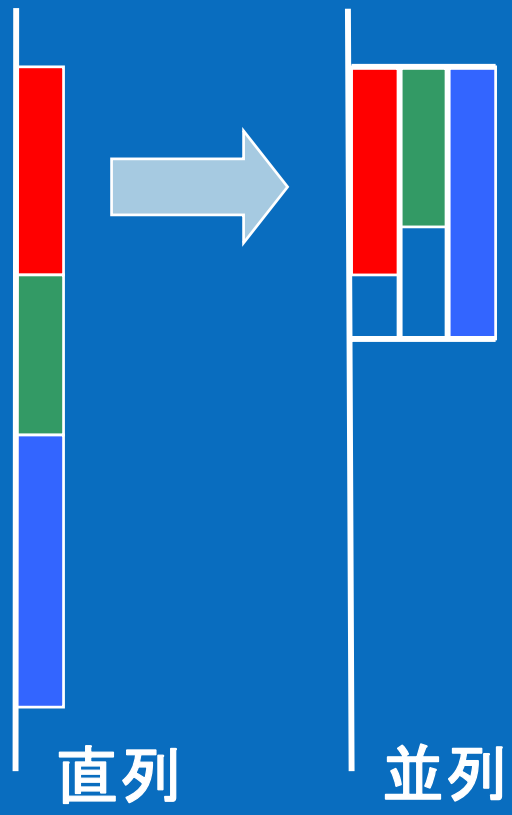
schedule 句

schedule 句	使用対象
STATIC	予測可能、反復あたりの作業量は均等
DYNAMIC	予測不能、反復あたりの作業量は可変
GUIDED	スケジューリング・オーバーヘッドを減らす dynamic の特別なケース

並列セクション（タスク並列処理）

コード内の独立したセクションを平行して実行できる

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



デフォルトでは、“omp section” の最後にバリアーがあるため、“nowait” 句を使用してバリアーをオフにする

並列 / ワークシェアの組み合わせ

OpenMP ショートカット: 同じ行に “parallel” とワークシェアを記述する

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

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

これらのコードは等価

“parallel sections” 構文もある

OpenMP* 例題

円周率プログラム:

逐次プログラム

```
static int num_steps = 1000000000;
double step;
int main ()
{      int i;
       double x, pi, sum = 0.0;
       step = 1.0/(double) num_steps;
       for (i=0; i< num_steps; i++){
              x = (i+0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
       }
       pi = step * sum;
       return 0;
}
```

命題:SPMD プログラムを作成

各スレッドは、任意のスレッド特有の動作を選択するスレッド ID を使用して同じコードを実行する。最大スレッド数を2にセットする。

OpenMP* 例題

```
#include <omp.h>
#define NUM_THREADS 2
static int num_steps = 1000000000;
double step;
int main ()
{       int i;
        double x, pi, sum[NUM_THREADS] ={0};
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
        {       double x;
                        int id, i, nthreads;
                id = omp_get_thread_num();
                nthreads = omp_get_num_threads();
                for (i=id;i< num_steps; i=i+nthreads){
                  x = (i+0.5)*step;
                  sum[id] += 4.0/(1.0+x*x);
                }
        }
        for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
        return 0;
}
```

SPMD プログラム

各スレッドは、任意のスレッド特有の動作を選択するスレッド ID を使用して同じコードを実行する

OpenMP* 例題

```
#include <omp.h>
#define NUM_THREADS 2
static int num_steps = 1000000000;
double step;
int main ()
{       int i;
        double x, pi, sum[NUM_THREADS] ={0.0};
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
        {       double x;
                int i, id;
                id = omp_get_thread_num();
#pragma omp for
                for (i=0;i< num_steps; i++){
                  x = (i+0.5)*step;
                  sum[id] += 4.0/(1.0+x*x);
                }
        }
        for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
        return 0;
}
```

ワークシェアリング・プログラム

各スレッドは、各スレッド用の適切な反復カウントを選択するシステムを使用して同じコードを実行する

OpenMP 構文の有効範囲

OpenMP 構文は複数のソースファイルに分割できる

ファイル: poo.f

```
C$OMP PARALLEL
      call whoami
C$OMP END PARALLEL
```

並列実行領域の字句範囲

並列実行領域の実行範囲は*字句*範囲を含む

ファイル: bar.f

```
      subroutine whoami
      external omp_get_thread_num
      integer iam, omp_get_thread_num
      iam = omp_get_thread_num()
C$OMP CRITICAL
      print*,'Hello from ', iam
C$OMP END CRITICAL
      return
      end
```

親なし(Orphan)ディレクティブが並列実行領域の外に現れる場合がある

内容

OpenMP 構文は 5 つのカテゴリに分けられる

- ランタイム関数/環境変数
- 並列実行領域
- ワークシェアリング
- データ環境
- 同期

OpenMP は C/C++ と Fortran では本質的に同じ

データ環境:
デフォルトの格納属性

共有メモリー・プログラミング・モデル

- ほとんどの変数はデフォルトで共有される

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: ファイルスコープ変数、static

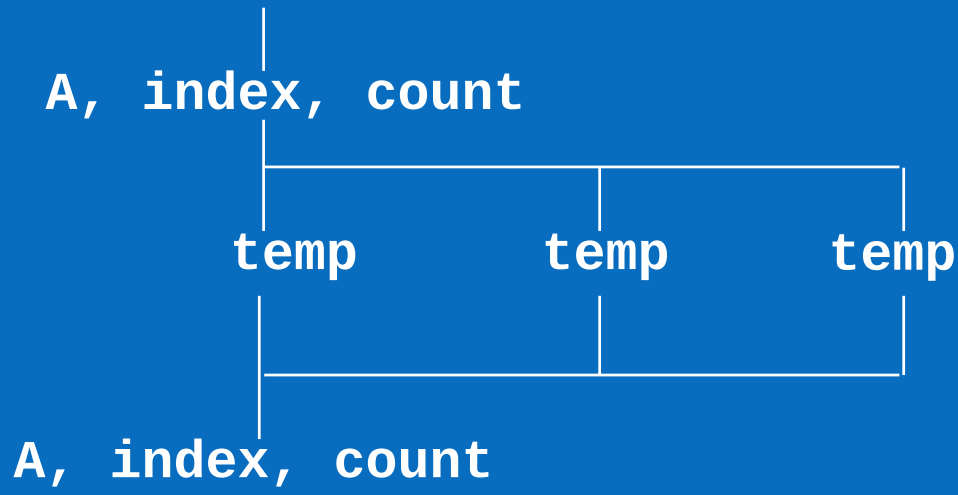
しかし、すべての変数は共有されない

- 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
- ステートメント・ブロック内の自動変数はプライベート

データ共有の例

```
sort(){
   int A[10], count;
   int index[10];
#pragma omp parallel
  {
   work(index);
  }
   printf("Number %d¥n",
   index[1]);
}
```

```
work(int *index){
  float temp[10];
      ……
}
```



***A*、*index*、および *count* はすべてのスレッドで共有されるが、*temp* は各スレッドに対してローカル**

OpenMP* 入門

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変更可能 §

- SHARED
- PRIVATE
- FIRSTPRIVATE
- THREADPRIVATE

このページのすべての句は、OpenMP 構文の字句範囲 にのみ用いられる

並列ループのプライベートの値は、ループ外側のグローバル値に引き渡し可能

- LASTPRIVATE

デフォルトのステータスは、次の句を使用して変更可能

- DEFAULT (PRIVATE | SHARED | NONE)

§ 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

private 句

private(var) は、各スレッド用に var のコピーを作成する
- 値は初期化されない
- プライベートのコピーは、オリジナルと格納先は異なる

```
wrong() {
        int j, IS = 0;
#pragma omp parallel for private(IS)
        for (j=0; j<1000; j++)
             IS = IS + j;
        printf("Number %d¥n",IS);
}
```

初期化に関係なく、IS はここで定義解除される

IS は初期化されていない

firstprivate 句

firstprivate は、private の特別なケース

– マスタースレッドから引継ぐ値で個々のプライベート・コピーを初期化する

```
almost_right() {
      int j, IS = 0;
#pragma omp parallel
firstprivate(IS)
      for(j=0; j<1000; j++)
           IS = IS + j;

      printf("Number %d¥n", IS);
}
```

各スレッドは、初期値 0 の独自の IS を得る

初期化に関係なく、IS はここで定義解除される

lastprivate 句

lastprivate は、最後の反復からのプライベートの値をグローバル変数に渡す

```
Closer() {
      int j, IS = 0;
#pragma omp parallel
firstprivate(IS)
#pragma omp lastprivate(IS)
      for(j=0; j<1000; j++)
           IS = IS + j;

      printf("Number %d¥n", IS);
}
```

各スレッドは、初期値 0 の独自の IS を得る

IS は、最後の反復でその値として定義される（つまり、j=1000）

データ環境のテスト

PRIVATE と FIRSTPRIVATE の例

```
        int A,B,C = 1
#pragma omp parallel private(B)
#pragma omp firstprivate(C)
```

- この並列実行領域の内側では...
  - “A” は、すべてのスレッドで共有される。A = 1
  - “B” と “C” は、各スレッドに対してローカル
    - “B” の初期値は未定義
    - “C” の初期値は 1
- この並列実行領域の外側では...
  - “B” と “C” の値は未定義

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 句での指定と同様に、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) をサポートしている

C/C++ では、default(shared) または default(none)のみ

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: 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

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

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

```
Closer() {
        int j,IS = 0;
        for(j=0;j<1000;j++)
          IS = IS + j;
        printf("Number
%d¥n",IS);
}
```

private、firstprivate および lastprivate の実証に使用されたコード

```
Correct() {
        int j, IS = 0;
#pragma omp parallel for reduction(+:IS)
        for(j=0;j<1000;j++)
          IS = IS + j;
        printf("Number %d¥n",IS);
}
```

このコードを並列化する正しい方法

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

オペランド	初期値
+	0
*	1
-	0
.AND.	すべて 1

オペランド	初期値
.OR.	0
MAX	1
MIN	0
||	すべて 1

演習:

実習 3
マルチスレッドの円周率プログラム

円周率プログラムを、プライベート、リダクションおよびワークシェアリング構文 を使用して並列化する

オリジナルのシリアルプログラムにどの程度似せることができるか確認する

OpenMP* 例題

円周率の計算:
リダクションを使用した並列化

```
#include <omp.h>
static int num_steps = 100000;
double step;
#define NUM_THREADS 2
int main ()
{       int i;  double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
        return 0;
}
```

OpenMP は、2 ～ 4 行のコードを追加する

内容

OpenMP 構文は 5 つのカテゴリに分けられる

- ランタイム関数/環境変数
- 並列実行領域
- ワークシェアリング
- データ環境
- 同期

OpenMP は C/C++ と Fortran では本質的に同じ

同期

OpenMP の以下の構文は同期をサポートする

- critical
- atomic
- barrier
- flush
- ordered
- single
- Master

これについてここで説明するが、実際には
これは同期構文ではない。同期を含むのはワークシェアリング構文である

同様に OpenMP は明示的な Lock メカニズムを提供する

- omp_init_lock, omp_set_lock, omp_unset_lock

同期

critical セクション（C/C++）

一度に 1 スレッドのみ critical セクションを処理できる

他のスレッドは順番がくるまで待機 。一度に 1 スレッドのみ consum() を呼び出す

```
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);
       }
}
```

同期

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

メモリー領域（下の例では X）の更新にのみ用いられる

```
#pragma omp parallel private(B)
{
        B =  DOIT(i);
        tmp = big_ugly();

#pragma omp atomic
        X = X + temp

}
```

同期

barrier: 各スレッドは、すべてのスレッドが到着するまで待機する

```
#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);
}
```

for ワークシェアリング構文の最後にある暗黙的なバリアー

nowait による暗黙的なバリアーはない

並列実行領域の最後にある暗黙的なバリアー

同期

ordered 構文は、ブロックを逐次順にする

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
      for (i=0;i<N;i++){
            tmp = NEAT_STUFF(i);
#pragma ordered
            res += consum(tmp);
      }
```

同期

master 構文は、マスター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();
}
```

暗黙的な同期

以下の OPenMP 構文では、バリアーが暗黙的に指定される

```
end parallel
end do              (nowait が使用されている場合を除く)
end sections        (nowait が使用されている場合を除く)
end single          (nowait が使用されている場合を除く)
```

OpenMP による明示的な Lock

```
#include <omp.h>
<...>
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel for
for (i = 0; i < N; i++)
{
   int type = getType(i);
   double force = computeForce(i);
   omp_set_lock(&lock);
   totForce[type] += force;
   omp_unset_lock(&lock);
}
```

明示的なロックはより細かな同期制御が可能;

- データ構造体の個々の要素に連動するロック
- ネストしたロック

OpenMP の明示的なロックは Windows や PThread の Mutex と同等

コースの内容

並列計算

- なぜ使用するのか?

OpenMP* 入門

- ディレクティブと使用方法
- 演習: Hello world と円周率の計算

並列プログラミング: ヒントとテクニック

- コード開発で避けるべきこと

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ーの最適化も必要

- データの局所性
- キャッシュデータの再利用
- メモリー階層の有効利用

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

まとめ

OpenMP* は...

- 共有メモリーマシン用の並列コードを記述する優れた方法である
- 並列プログラミングへの非常に簡単なアプローチである
- データ競合の可能性を含む

本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。製品に付属の売買契約書『Intel's Terms and conditions of Sales』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的所有権を侵害していないことへの保証を含む) に関しても一切責任を負わないものとします。

インテル製品は、予告なく仕様が変更されることがあります。

* その他の社名、製品名などは、一般に各社の商標または登録商標です。