第3回並列ゼミ

指導者 佐野正樹
チーフ 下坂久司
サブチーフ 田村隆一 松山靖彦

1 OpenMP の概要

1.1 OpenMP とは

OpenMP は，共有メモリモデルによる並列化を記述する API(Applicaton Program Interface) である．ベース言語（Fortran/C/C++）を directive（指示文）で並列プログラミングできるように拡張しており，また米国コンパイラ関係の ISV を中心に仕様を決定している．[1]
公開情報としては OpenMP Official Web Site[2]で OpenMP 規定，仕様書，技術白書，アナウンスメントと公開を行なっている．

1.1.1 OpenMP の必要性

共有メモリモデルでは，全てのプロセッサは，システム内の全ての他のプロセッサのメモリに直接アクセス可能である．これは，プロセッサが直接に全ての共有アドレスに対するロード／ストアが可能であることを意味する．プログラマは，当然個々のプロセッサがプライベートで使用する部分の宣言も可能である．これによって，プログラマにとって非常に容易にアプリケーションの並列化を記述し，管理することが可能となる．

共有メモリマルチプロセッサシステムが普及し，また容易にスケーラビリティのあるアプリケーションを開発出来るにも関わらず，多くのアプリケーション開発者が，これらの共有メモリプログラミングでのプログラムの並列化に積極的とならなかった理由は，１つであった．それは移植性の問題（並列化指示文の共通化の問題）であった．共有メモリシステムを提供するベンダーは，各々が独自に Fortran 又は C を拡張することで並列ソフトウェアの開発を行っている．しかしながら，これらの拡張は，各ベンダー独自のものであり，移植性の問題から，MPI(Message Passing Interface) や，PVM(Parallel Virtual Machine) といった，移植性に優れたメッセージ・パッシングモデルを使用してきた．その問題点を解決すべく，共有メモリ並列 API である OpenMP が出現したのである．

そこで，共有メモリ並列 API である OpenMP のプログラミングは，おもに共有メモリアーキテクチャの計算機上で実装されており，共有メモリプログラミングの標準となってきている．OpenMP のプログラミングは，プログラム中に並列実行や同期を指示する指示文を記述することで行う．高レベルデータ並列であり，容易にプログラムの開発を行うことができる．また，コンパイルオプションにより指示文を無視することができるため，デバッグやプログラムの管理もしやすい．

1.2 OpenMP の機能

1. 移植性の高い共有メモリ並列処理 API
2. 細粒度（ループ）での並列化の標準
3. 粗粒度での並列アルゴリズムのサポート
4. より性能面を重視した API
5. 拡張性を含めてシンプルなモデルの提案

1.3 OpenMP の目標

- 複数のプラットホーム上での移植性の向上
- 実行性能におけるスケーラビリティの向上
- ディスクトップからスーパコンピュータまでの並列アプリケーションの開発において，よりシンプルかつ柔軟なインターフェイスの提供を行う．

[1] Oct.1997 Fortran ver.1.0 API
Oct.1998 C/C++ ver.1.0 API
[2] http://www.openmp.org

1.4 OpenMP の設計目標

OpenMP は，異なるプラットフォームで容易にインプリメントできるフレキシブルな規格となるよう設計された．この目標を達成するためにこの規格は４つの異なった部分から構成されている：制御構造，データ環境，同期，実行時ライブラリィである．Fig. 1 は OpenMP のアーキテクチャを表わしている．

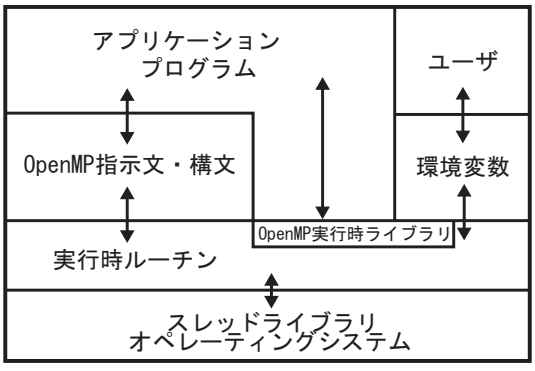

Fig. 1　OpenMp のアーキテクチャ

1.4.1 制御構造

OpenMP は制御構造を極力少なくなるように努力した．多くの並列アプリケーションを書く時に必要となる制御構造は極めて少ないということが分かっている．ユーザが合理的にプログラミングしたものよりも，さらにコンパイラが機能と性能を与えられるという制御構造だけを OpenMP は含んでいる．

1.4.2 データ環境

各プロセスは，実行のためのコンテキストを提供するデータ環境と結びついている．（プログラムスタート時の）初期プロセスは，実行処理中に存在する初期データ環境と結びついている．実行処理中に新しいプロセスが生成されたときのみ，新しいデータ環境を形成しているオブジェクトは３つの属性：SHARED,PRIVATE,REDUCTION のうち，１つを持っている．

1.4.3 同期

同期には陰的同期と陽的同期の 2 種類がある．陰的同期ポイントは，PARALLEL 構文の始めと終わりにあり，さらにその他の全ての制御構文の終わりにある．陽的同期は，処理の順序やデータ依存を管理するためにユーザが指定する．同期はプロセス間のコミュニケーションの形態であり，プログラムの性能に大きな影響を与える．通常，プログラムの同期（陰的，陽的共）要求を最小にすることによって，最良の性能を得ることができる．このため，OpenMP では豊富な同期機能を提供しており，開発者はアプリケーション内の同期を最大にチューニングすることができる．

1.4.4 実行時ライブラリと環境変数

OpenMP は呼び出し可能な実行時ライブラリ（runtime library:RTL）とそれに伴う環境変数を提供する．RTL は問い合わせ関数，実行時間数，ロック関数を含む．アプリケーションは実行時間数を利用して実行モードを指定できる．アプリケーションの開発者は，完了時間よりもシステムのスループット性能を最大にしたいと願っていると仮定する．このような場合，開発者は並列領域で使用するプロセッサ数をシステムに動的に伝える．こうすることによって，プログラムの完了時間への影響を最小限にし，システムのスループット性能を劇的に上げることができる．

1.5 OpenMP の API

OpenMP の API を説明する．

OpenMP の API は新しい言語ではなく，コンパイラ指示文（directive/pragma），ライブラリ，環境変数によりベース言語（Fortran,C,C++）を拡張する．並列実行・同期はプログラマが明示し自動並列化ではなく，また指示文を無視することにより，逐次で実行が可能である．つまり，incremental に並列化することによりプログラム開発，デバッグの面から実用的であり，逐次版と並列版を同じソースで管理できる．Table **??**に MPI と OpenMP の API を示す．

Table 1　MPI・OpenMP・Pthreads の API

	MPI	OpenMP	Pthreads
スケラーブル	Yes	Yes	Sometimes
逐次並列化の適用	No	Yes	No
ポータブル	Yes	Yes	Yes
Fortran サポート	Yes	Yes	No
高レベル	No	Yes	No
データ並列のサポート	No	Yes	No
性能重視	Yes	Yes	No

1.6　OpenMP の実行モデル

OpenMP API は言語の拡張であり，基本的にもとの (逐次の) 言語の機能を並行に実行するように拡張する．

OpenMP Fortran/C API は，Fork-join モデルの並列実行モデルを用いている．OpenMP Fortran で記述されたプログラムは，マスタスレッドと呼ばれる一つのプロセスとして実行を開始する．マスタスレッドははじめて並列構文につき当たるまで，逐次的に実行される．OpenMP C API では，parallel 構文で指定された block statement を並列実行する．並列構文につき当たった時には，マスタスレッドはスレッドの team を生成し，マスタスレッドは team のマスタになる．並列構文に囲まれた文は，その中の文から呼び出されたスレッドルーチンを含めて，チームのそれぞれのスレッドによって並列に実行される．構文内の構文的に囲まれている文は構文の静的な extent を定義する．動的な extent とは，構文の中から呼び出されるルーチンも含む．

並列構文の終りでスレッドは同期し，マスタスレッドのみが実行を続ける．一つのプログラムで複数の並列構文を指定することができる．したがって，プログラムは実行中に何度も fork/join を行うことになる．

OpenMP API では，プログラマは並列構文から呼び出されるルーチン内で directive を使うことができる．並列構文の静的な extent になく，動的な extent にある directive を orphaned directive と呼ぶ．orphaned directive によって，逐次プログラムの最小限の修正により，ユーザはほとんどの部分に実行することができる．この機能を使って，並列構文をプログラムのトップレベルを使い，directive を呼び出されたルーチンで実行を制御することにつかうことができるようになる．

C pre-processor では，_OPENMP を定義する．OpenMP プログラムのみで実行する文については，C preprocessor を用いて指定してもよい．Fig. 2 に実行モデルを示す．

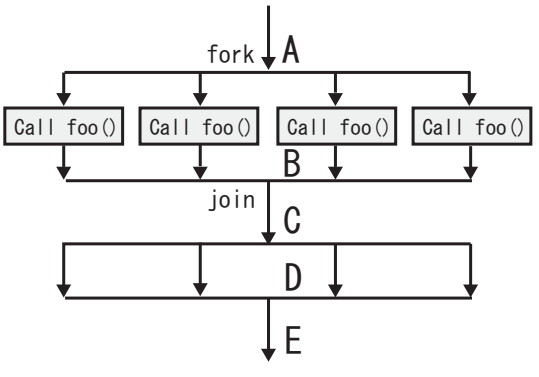

Fig. 2　OpenMP の実行モデル

2　OpenMp の利点・欠点

2.1　利点

- incremental に並列化ができる．
- 逐次実行版と並列実行版を同じソースで管理できる．

- ユーザ指示どおりに並列化できる.
- スレッドプログラミングに比べて，並列性が構造的に記述されている.
  - Work sharing 構文，orphan directive
  - コンパイラで解析が可能

2.2 欠点

- 並列可能性はユーザがチェックする必要がある.
- data mapping が記述できない.
  - Iteration mapping との整合性
  - locality が失われる可能性
- 配列に対して reduction 演算の指定ができない.
- コンパイラが必要
  - pragma による記述

3 OpenMP プログラミング

ここでは OpenMP プログラミングの枠組み,POSIX スレッド (Pthread) と OpenMP との違い, スケジューリング, 同期などについて説明する.

3.1 プログラミングの枠組み

OpenMP では, ベースとなるプログラミング言語 (C や Fortran) 言語で記述されたプログラムの中に, 並列処理する部分 (**Parallel Region**) を指示文 (directive) で指定することによって, 並列プログラミングを行う. この指示文は `#pragma omp` で始まる文 (pragma 文) である.pragma 文のフォーマットを以下に示す.

> `#pragma omp` *directive name*[*clause,clause,.....*]
> *directive name* は指示文の名前 (parallel,atomic など) である.
> *clause* によって, さらに様々な指示を出すことができる.

pragma 文は次に続くブロックを並列化する.pragma 文を無視するように指定することによって, 並列プログラムを逐次プログラムにすることができる. 同じ Parallel region を実行するスレッドを **team** と呼ぶ. つまり Parallel region 内の処理を team で並列実行することになる.

OpenMP プログラミングは大まかに言うと以上のような形で作成できる.OpenMP ではこの他に, 実行時ライブラリ関数と環境変数が提供されている. すなわち,OpenMP はベースとなる言語に以下のものを加えたものである.

1. 並列化指示文
2. 環境変数
3. 実行時ライブラリ

3.1.1 並列化指示文

並列化指示文は先に述べたように (pragma 文のフォーマット),prallel region を明示する. さらに, 条件によって処理を行うスレッド数を変えたり, 並列処理の形態 (平行に同じ処理をするのか, スレッドごとに異なる処理をするのか) などを指定できる. *parallel* **構文**parallel 構文は fork と join の変わり目を指示する.parallel 構文を入れ子構造にすることもできる. これらのことから,parallel 構文がスレッド数の変わり目といえる (Fig. 3). また,if 節によってスレッド数を条件に応じて切り替えることもできる. *work sharing* **構文**work sharing 構文は team 内のスレッドで仕事を分担するように指示する. 仕事の分担の仕方によって,for 構文 (Fig. 4),section 構文,single 構文がある.

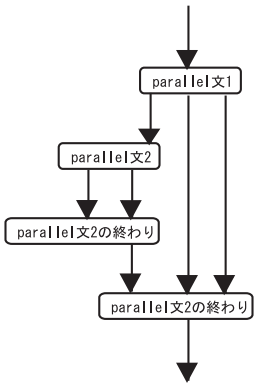

Fig. 3　parallel 文

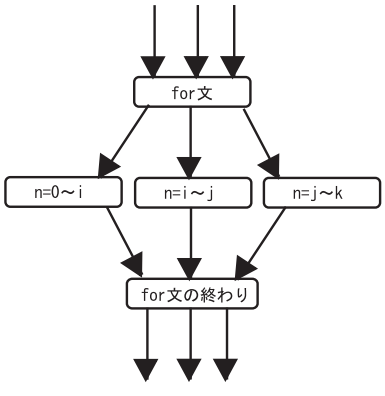

Fig. 4　for 文

3.1.2 master 構文と同期構文

並列処理プログラミングを行うには同期を指示できなければならない. さらに,SMP では, 複数のスレッドが同じメモリ空間を使用するために, ロック処理などに関する指示を行う必要がある. そこで,OpenMP では様々な同期の取り方を指示できる. 同期構文を以下に示す.

1. master 指示文:マスタスレッドのみが実行するブロックを指示する (Fig. 5).
2. critical 指示文:そのブロックを複数のスレッドで同時に実行しないように指示する (Fig. 6).
3. barrier 指示文:すべてのスレッドがそのブロックに到達するまで待ってから実行するように指示する (Fig. 7).
4. atomic 指示文:複数のスレッドによって同時書き込みが行われる可能性のあるブロックにおいてコヒレンシーを維持するように明示する.
5. flush 指示文:メモリ参照においてコヒレンシーを維持するように, 明示する.
6. orderd 指示文:複数のスレッドで処理されるループを, 逐次実行したときと厳密に同じ順序で実行するように指示する.

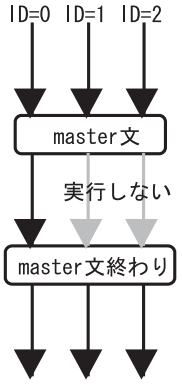

Fig. 5　master 文

3.1.3 データ環境

同じ team のスレッド同士は, 同じメモリ空間を使用できる. しかし, あるスレッド以外にはアクセスさせたくない変数などが存在するときは, その変数をデータ環境によって, 明示する必要がある. 逆にいくつかのスレッドで共有させたい変数も明示できる. これらの明示を行うのがデータ環境指示文である.

3.2 Pthread との違い

POSIX スレッド (Pthread) とは, インタフェースを **POSIX(Portable Operating System Interface for UNIX)**[3]に準拠させた並列プログラムの規格である.Pthread の正式名称は,POSIX 1003.1c-1995(または ISO/IEC 9945-1:1996) である. スレッド規格にはいくつかのものがあるが, 特定の OS やシステムに依存している場合が多く,Pthread が最も汎用性が高い. 表？(一章) に Pthread と OpenMP との違いを示す.

3.3 スケジューリング

ケーキを二つに切り分ける時に, 普通は垂直に切り分ける. 水平に切り分けると下の部分があたった人はスポンジだけになって不公平である.

並列処理でも上の例と同様な場合がありうる. 例えば, ループをネストして, 外側のループのカウンタの回数に応じて,

[3]IEEE によって定められた,UNIX ベースの OS が備えるべき最低限の仕様のセット. 各社の UNIX 互換 OS にはそれぞれ独自の拡張や仕様の変更が施され, 互換性が失われてしまったため, 各 OS 間で最低限の互換性を確保するために定義された. アプリケーションソフトが OS の提供する機能を呼び出すための方法 (システムインターフェース) などを定義している. アメリカ規格協会 (ANSI) や国際標準化機構 (ISO) でも標準として採用され, アメリカ政府機関に納入する UNIX システムが守るべき必須条件となっている.

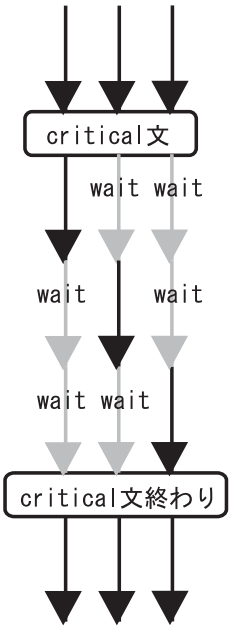

Fig. 6　critical 文

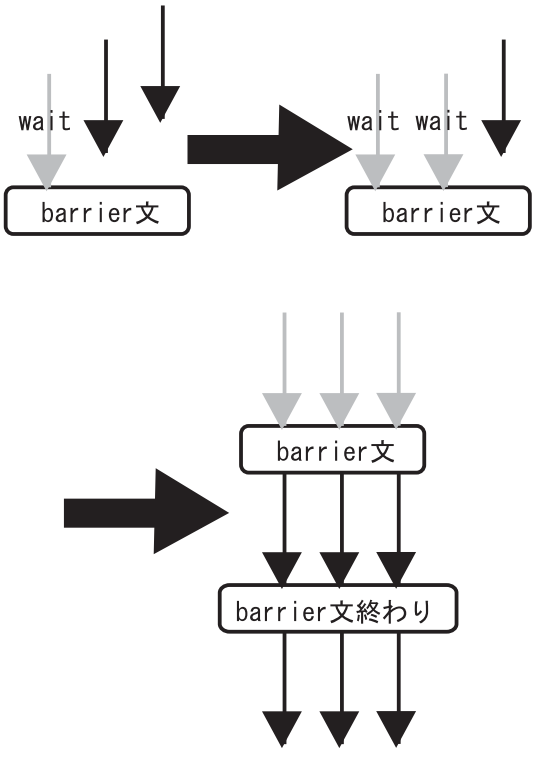

Fig. 7　barrier 文

内側のループの回数を変化させる場合である. 各スレッドが処理すべき仕事の量を同程度にしないと, 並列化の効率化がはかれない.100 回 (i=0;i¡100;i++) の繰り返し実行を 2 つのスレッド 1,2 で並列に実行するときに例えば次の二つの実行の仕方が考えられる.

1. スレッド 1 が i=0〜49, スレッド 2 が i=50〜99 を同時に実行する.
2. スレッド 1 が i=0〜24, スレッド 2 が i=25〜49 を同時に実行し, そのあとに, スレッド 1 が i=49〜74, スレッド 2 が i=75〜99 を同時に実行する.

このように, 同じ並列実行でも, いろいろな仕事の分担の仕方がある. この分担の仕方を, スケジューリングという.

3.4 同期

並列処理をする際の問題の一つとして,「仕事には段取り, 順序が存在する」という問題があった (第 1 回). この問題に対処するために**同期**を取る必要がある. 同期の身近な例を以下に示す.

1. 餅つきで, 一方が餅をこねている間に, もう一人は杵を振り上げる. 杵を振り下ろす作業と, 餅をこねる作業が同時に行われないように, 同期を取っている. この例は critical 文にあたる.
2. 数人で生協に食事に行ったとき, 必ずしも席につくのは同時ではないが, 一緒に行った人が全員席につくまで待ってから食べ始める. この例は,barrier 文にあたる.

プログラミングにおける同期の例を以下に示す.

```
1:      #pragma omp parallel ・・・・・・・・ Parallel Region である事を宣言.
2:      {
3:       int c,b,e,i,ss;
4:       c=1000/omp_get_num_threads();
5:       b=c*omp_get_thread_num();
6:       e=s+c;
7:       ss=0;
8:       for(i=b;i<e;i++) ss+=a[i];
9:       #pragma omp atomic
10:      s+=ss;
11:     }
```

この例で,9,10 行目に注目する.s の値に各スレッドがもつ ss の値を加えている. このときに同期を取っていないと, 例えばスレッド 1,2 の順で s に書き込む場合次のようなことがおこりうる.

1. スレッド 1 がレジスタに s の値を読み込む.
2. スレッド 2 がレジスタに (スレッド 1 がまだ ss の値を加えていない状態にある)s の値を読み込む.
3. スレッド 1 が s の値に ss の値を加える.
4. スレッド 2 が s の値に ss の値を加える.
5. スレッド 1 が s のアドレスに結果を書き込む.
6. スレッド 2 が s のアドレスに結果を書き込む.

このようなタイミングで実行された場合, 最終的な結果は s の値にスレッド 2 の ss の値が加えられただけで, スレッド 1 の ss の値は足しあわされていない. この状況は, ゼミのレジメを同じ資料を用いて分担して作る際に, うまく連絡 (同期) を取り合っていないと, 同じ内容を書いてしまい, 人数×時間分のレジメが作れない状況に似ている. これを同期を取って s にスレッド 1 と 2 の ss の値を加えようとすると, 例えば次のようにすべきである.

1. スレッド 1 がレジスタに s の値を読み込む.

2. スレッド 1 が s の値に ss の値を加える.

3. スレッド 1 が s のアドレスに結果を書き込む.

4. スレッド 2 がレジスタに s の値 (スレッド 1 が ss の値を加えた結果) を読み込む.

5. スレッド 2 が s の値に ss の値を加える.

6. スレッド 2 が s のアドレスに結果を書き込む.

4 OpenMP の利用

4.1 OpenMP の実行環境の例

OpenMP を利用するには専用コンパイラがインストールされたマシンが必要である．今回使用するマシンは「 fraulein： 192.168.30.128 」である．このマシンには/usr/local/bin に RWCP Omni OpenMP Compiler Version 1.2s 「 omcc 」がインストールされている．今回はこのコンパイラを使用する．fraulein はマスタ１台・スレーブ7台のクラスタ環境にあるが，スレッドプログラミングでは共有メモリを利用するので，マスタの一台のみを使用する．このマシンは pentium のデゥアルプロセッサという構成になっている．このため，３つ以上のスレッドを発生させることは，プログラムの実行速度という点では意味がない.

4.2 OpenMP プログラミングの実行方法

1. ログイン ssh を使用し，fraulein にログインする.

2. コンパイル次のような書式でコンパイルを行う.

```
omcc [driver-options] [compiler-options] filename
```

driver-options には次のようなものがある.
-help:ヘルプの表示.
-omniVersion:コンパイラーのバージョンの表示.

compiler-options には次のようなものがある.
-c:オブジェクトファイルの作成
-Dmacro:'macro' で指定した文字で #define 'macro' 1 を作成
-Dmacro=defn:'macro' で指定した文字で #define 'macro' defn を作成
-g:デバッグ
-Idir:Include File のディレクトリ
-Ldir:ライブラリのディレクトリ
-llibrary:ライブラリのファイル
-O:最適化
-o file:出力ファイル
-Umacro:-D の逆
-v:冗長モード
-omp:並列化しない

3. 実行./a.out もしくは-o で指定したファイルを実行する.

4.3 並列構文

4.3.1 parallel 構文

次の構文は並列リージョンを定義する．並列リージョンとは複数のスレッドにより並列に実行されるプログラムの一部である.

#pragma omp parallel [*clause*[*clause*]...] *new-line*
structured-block

clause には次のようなものがあるが，これらに関しては，後のデータ環境で説明する．
if(*scalar-expression*), private(*list*), firstprivate(*list*)
default(shared—none), shared(*list*), copyin(*list*), reduction(*operator:list*)
この時元のスレッドはスレッド番号が 0 であるチームのマスタスレッドになる．マスタスレッドを含む全てのスレッドはリージョンを並列実行する．チーム内のスレッド数は環境変数およびライブラリ呼び出しによって制御できる．if 指示節の式が 0 になる場合リージョンは逐次実行される．並列リージョンの最後では暗黙にバリア同期が実行される．

4.3.2 work sharing 構文

ワークシェアリング構文はこの構文に到着したチームのメンバに，対応するステートメントを分割して実行させる．この時ワークシェアリング指示文は新たにスレッドを生成しない．またワークシェアリング構文の入り口には暗黙のバリア同期は存在しない．

for 構文

for 指示文は対応するループの繰り返し処理を並列で実行すべきリージョンとして，繰り返しのワークシェアリング構文に指定する．for ループの繰り返しは既に存在するスレッド間で分配される．for 指示文の構文は以下の通りである．

#pragma omp for [*clause*[*clause*]...] *clause*
for-loop

clause には次のようなものがある．
private(*list*), firstprivate(*list*), lastprivate(*list*)
reduction(*operator:list*), ordered
schedule(*kind*[,*chunk* __ *size*]):ループをチーム内のスレッドに分割する方法を指定する．
nowait:最後に暗黙のバリア同期をとらない

for ループのスケジューリング

for ループのスケジュールは schdule 指示節で指定する．schedule(*kind*[,*chunk* __ *size*]) で指定する．

1. schedule(static,chunk __ size):ループは *chunk* __ *size* で指定されたサイズのチャンクに分割される．生成された各チャンクはチーム内のスレッドにスレッドの番号順にラウンドロビン方式で静的に割り当てられる．*chunk* __ *size* が指定されていない場合は，イタレーション空間はほぼ同じサイズのチャンクに分割され，各スレッドに１つのチャンクが割り当てられる．

2. schedule(dynamic,chunk __ size):*chunk* __ *size* 回のイタレーション空間のチャンクを各スレッドに割り当てる．スレッドに割り当てられたチャンクの処理が終了すると，残りのチャンクがなくなるまで動的に別のチャンクをスレッドに割り当てる．*chunk* __ *size* が指定されていない時は，既定値は１となる．

3. schedule(guided,chunk __ size):イタレーションのチャンク *chunk* __ *size* に向かって徐々に小さくしながらスレッドに割り当てる．規定値は１である．

4. schedule(runtime,chunk __ size):スケジュール方法は実行時に決定する．スケジューリングの種類やチャンクサイズは環境変数 OMP __ SHEDULE に設定することになる．環境変数が設定されていない場合は実装依存となる．

5. schedule(guided,chunk __ size):イタレーションのチャンク *chunk* __ *size* に向かって徐々に小さくしながらスレッドに割り当てる．規定値は１である．

明示的にスケジューリングが決定されていない場合は，選択されるスケジューリング方法は実装依存である．

sections 構文

section 指示文はチーム内のスレッドで分割して実行する構文の集合を指示する．非繰り返しワークシェアリング構文である．各セクションはチーム内のスレッドにより 1 度だけ実行される．section 指示文の構文は以下の通りである．

#pragma omp sections [*clause*[*clause*]...] *new-line*{
[#pragma omp section *new-line*]
structured-block
[#pragma omp section *new-line*]
structured-block}

clause には次のようなものがある．
private(*list*), firstprivate(*list*), lastprivate(*list*)
reduction(*operator:list*), nowait

single 構文
single 指示文は対応する構造ブロックがチーム内の一つのスレッド (必ずしもマスタスレッドでなくて良い) のみで実行されることを指示する構文である．single 指示文の構文は以下の通りである．

#pragma omp single [*clause*[*clause*]...] *new-line*
structured-block

clause には次のようなものがある．
private(*list*), firstprivate(*list*), nowait

4.3.3 並列 work sharing 組み合わせ構文

並列ワークシェアリング構文は，1 つのワークシェアリング構文を含む並列リージョンを指示するための簡略形である．これらの指示文の意味は，一つのワークシェアリング構文の前に陽に parallel 指示文を指定したのと等しい．
並列ワークシェアリング構文には次のようなものがある．
parallel for 構文 (#pragma omp parallel for)
parallel section 構文 (#pragma omp parallel section)

4.3.4 master 構文・同期構文

次に同期構文について説明する．

master 構文
master 指示文はマスタスレッドのみで実行する構造ブロックを指示する指示文である．

#pragma omp master *new-line*
structured-block

critical 構文
critical 指示文は指定された構造ブロックを同時に複数のスレッドで実行しないように指示する指示文である．

#pragma omp critical *new-line*
structured-block

barrier 構文
barrier 指示文はチーム内の全てのスレッドを同期させる．スレッドがこの指示文に到達すると他の全てのスレッドがこの指示文に到達するまで待つ．

#pragma omp barrier *new-line*

atomic 構文
atomic 指示文は複数の同時書き込みを行う可能性のあるスレッドに対して，指定されたメモリをアトミックに更新す

る事を指示する指示文である．

#pragma omp atomic *new-line*
expression-stmt

flush 構文

flush 指示文はメモリ上にあるオブジェクトに対して，矛盾しないメモリ参照を指示する指示文である．

#pragma omp flush *new-line*

ordered 構文

order 指示文が指定されたブロックは逐次ループで実行された時と同じ順序で実行される．

#pragma omp ordered *new-line*
structured-block

4.3.5 データ環境

並列リージョン実行中にデータ環境を制御する 1 つの指示文と幾つかの指示節について述べる．

threadprivate 指示文

threadprivate 指示文は，(*list*) に指定されたファイルスコープあるいは namespace スコープの変数を，スレッドに固有で且つスレッドからファイルスコープで参照可能な変数にする．

#pragma omp threadprivate (*list*)*new-line*

データスコープ属性指示節幾つかの指示文では，ユーザが指示節を用いて，そのリージョン実行中に変数のスコープ属性を制御することができる．

1. private(*list*):変数がチーム内の各スレッド毎にプライベートであることを宣言
2. firstprivate(*list*):privete 変数と同じ機能を包含し，且つ新しいプライベートなオブジェクトはブロック内に暗黙の宣言があるように初期化され，初期化には変数の元のオブジェクトが使用される
3. lastprivate(*list*):privete 変数と同じ機能を包含し，且つ最後のループもしくは文脈を実行した後の値が元の変数に代入される
4. default(shared — none):変数のデータスコープ属性を制御する
5. shared(*list*):変数をチーム内の全てのスレッドで共有する
6. copyin(*list*):threadprivate 変数に同じ値を代入する
7. reduction(*operator:list*):スカラ変数を演算子でリダクション演算する

4.3.6 実行時ライブラリ関数

1. omp __ get __ num __ threads：並列リージョンを実行しているチームのスレッド数を返す
2. omp __ set __ num __ threads：並列リージョンで使用するスレッド数を指定する

3. omp __ get __ max __ threads:スレッド値の最大値を返す
4. omp __ get __ thread __ num:この関数を実行したスレッドのチーム内の番号を返す
5. omp __ get __ num __ procs:プログラムに割り当て可能なプロセッサ数の最大値を返す
6. omp __ in __ parallel:並列実行中に並列リージョンの動的有効範囲から呼ばれた場合，０以外の値を返す
7. omp __ set __ dynamic:並列リージョン実行時のスレッド数の動的調整機能の有効/無効を指示する
8. omp __ get __ dynamic:スレッド数の動的調整機能が有効の場合，0 以外の値を返す
9. omp __ set __ nested:ネストされた並列実行の有効/無効を指示する
10. omp __ get __ nested:ネストされた並列実行が有効な場合 0 以外の値を返す
11. omp __ init __ lock:ロックの初期化を行う
12. omp __ init __ nest __ lock:ロックの初期化を行う
13. omp __ destroy __ lock:ロック変数を非初期化状態に戻す
14. omp __ destroy __ nest __ lock:ロック変数を非初期化状態に戻す
15. omp __ set __ lock:ロックが可能な状態になるまで，スレッドの停止を行いロックを実行する
16. omp __ set __ nest __ lock:ロックが可能な状態になるまで，スレッドの停止を行いロックを実行する
17. omp __ unset __ lock:ロックの所有権を解放する機能を提供する
18. omp __ unset __ nest __ lock:ロックの所有権を解放する機能を提供する
19. omp __ test __ lock:ロックを試みるがスレッドの実行を阻止しない
20. omp __ test __ nest __ lock:ロックを試みるがスレッドの実行を阻止しない

4.3.7　環境変数

1. OMP __ SCHEDULE:スケジューリングのスケジューリング方法とチャンクサイズを指定する
2. OMP __ NUM __ TEREADS:実行中に使用するスレッド数を指定する
3. OMP __ DYNAMIC:スレッド数の動的調整機能の有効/無効を指示する
4. OMP __ NESTED:ネストされた並列実行の有効/無効を指示する

4.4　OpenMP を用いたプログラムの例

4.4.1　例．1

［単純なループの並列実行］

```
--------------------------------------------------------------------------------

#include <stdio.h>
int main()
{
int i,n,sum=0;
int a[10000],b[10000],c[10000];

scanf("%d",&n);
```

```
#pragma omp parallel for private(i) shared(a,b,c,sum) reduction(+: sum)
{
for(i = 0; i < n; i++){
a[i]=1;b[i]=2;
c[i]=a[i]+b[i];
sum += c[i];
}
}

printf("sum = %d\n",sum);

}
```

実際にはループの繰り返し変数は規定値で private であり，明示的に private 指示節に指示する必要はない.

4.4.2 例．2

```
     [スレッド ID]
--------------------------------------------------------------------------------
#include <stdio.h>
int main()
{
omp_set_dynamic(0);
omp_set_num_threads(omp_get_num_procs());
#pragma omp parallel sections
{
#pragma omp section
printf("1-ThreadID == %d\n",omp_get_thread_num());
#pragma omp section
printf("2-ThreadID == %d\n",omp_get_thread_num());
#pragma omp section
printf("3-ThreadID == %d\n",omp_get_thread_num());
}
}

--------------------------------------------------------------------------------
```

今回の使用する環境では，スレッドはプロセッサ数までしか発生することができないようである．また既定値は 2 のようである．よって，main 関数の最初の 2 行は必要ないが，このようにすると明示的に発生させるスレッド数を変化させることができる.

5 課題

fraulein を使用し，並列プログラムを実行してください．実行する前に, 結果を予測すると, 同期やコヒレンシーについての理解の確認になると思います.

課題 1

```
#include <stdio.h>
void main(){
    int i;
    #pragma omp parallel
    {
        for(i=0;i<10;i++){
            printf("%d\n",i);
        }
    }
}
```

課題 2

```
#include <stdio.h>
void main(){
    int i;
    #pragma omp for
    {
        for(i=0;i<10;i++){
            printf("%d\n",i);
        }
    }
}
```

課題 3

```
#include <stdio.h>
void main(){
    int i;
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0;i<10;i++){
            printf("%d\n",i);
        }
    }
}
```

課題 4

```
#include <stdio.h>
void main(){
    int i;
    #pragma omp parallel private(i)
    {
        for(i=0;i<10;i++){
            printf("%d\n",i);
        }
    }
}
```

課題 5

```
#include <stdio.h>
void main(){
    int i;
    #pragma omp parallel private(i)
    {
        #pragma omp for
        for(i=0;i<10;i++){
            printf("%d\n",i);
        }
    }
}
```

課題 6

```
#include <stdio.h>
void main(){
    int i;
    #pragma omp parallel
    {
        #pragma omp critical
        for(i=0;i<10;i++){
            printf("%d\n",i);
        }
    }
}
```

課題 7

```
#include <stdio.h>
void main(){
    int i;
    #pragma omp critical
    {
        for(i=0;i<10;i++){
            printf("%d\n",i);
        }
    }
}
```

課題 8

```
#include <stdio.h>
void main(){
    int i,j;
    double a;
    #pragma omp parallel
    {
if(omp_get_thread_num()==1){
for(j=0;j<10000;j++)
                a=i^10000000;
        }
        for(i=0;i<10;i++){
```

```
            printf("i=%d::thread id=%d\n",i,omp_get_thread_num());
            for(j=0;j<10000;j++)
                a=i^10000000;
        }
    }
}
```

参考文献

1) 湯銭太一・安村通晃・中田登志之『初めての並列プログラミング』