THE POSSIBILITIES ARE INFINITE FUJITSU

OpenMP Application Program Interface

Version 3.0 May 2008

（日本語版）

原文の注意書

日本語版の注意書

本仕様書は、OpenMP Architecture Review Board により作成された OpenMP Application Program Interface Version 3.0 May 2008 を OpenMP Architecture Review Board の許諾のもと、富士通株式会社が日本語に訳した富士通株式会社の著作物です。
本仕様書は、原文は OpenMP Architecture Review Board の Copyrignt であり、日本語訳にも適用されます。

富士通株式会社　池谷 聡が日本語訳を行いましたが、以下の方々から、翻訳に際し、広範囲に渡り多くのご支援を頂きました。ここに記して謝意を表します（敬称略）。

佐藤 三久	（筑波大学）
草野 和寛、村井 均	（日本電気株式会社）
岩下 英俊、鈴木 敏弘、守屋 勝由、張 遠遠	（富士通株式会社）

2008 年 11 月
富士通株式会社

2009/01/08

本仕様書（日本語版）は、OpenMP Application Program Interface Version 2.5 May 2005 からの変更および新規追加の部分が分かるようにしています。

（１）記述内容の変更部分は、各ページの左部分に「破線 ┆ 」をマークしています。
（２）新規に記述が追加された部分は、各ページの左部分に、「実線 ┃ 」をマークしています。

目次

第１章

はじめに

このドキュメントは、C、C++、および Fortran プログラムにおいて共有メモリ並列性を記述するために使用できるコンパイラ指示文、ライブラリルーチン、および環境変数を規定しています。そして、これらの機能は全体として OpenMP アプリケーションプログラムインタフェース（OpenMP API）の仕様を定義しています。この仕様は、異なるベンダーの共有メモリアーキテクチャ間で移植性のある並列プログラミングの一つのモデルを提供しています。多くのベンダーのコンパイラが OpenMP API をサポートしています。OpenMP についての更なる情報は、以下の Web サイトで参照することができます。

http://www.openmp.org

このドキュメントで定義している指示文、ライブラリルーチン、および環境変数を用いることで、移植性を保ちながらユーザが並列プログラムを作成、管理することができます。指示文は、SPMD（single program multiple data）構文、タスク構文、ワークシェアリング構文、および同期構文によりベース言語である C、C++、および Fortran を拡張しており、さらにデータの共有化とプライベート化をサポートしています。実行環境を制御する機能は、ライブラリルーチンと環境変数によって提供されています。OpenMP API をサポートしているコンパイラの多くは、すべての OpenMP 指示文を有効にし解釈させるためのコマンドラインオプションを持っています。

１．１．範囲

OpenMP API はユーザ指示の並列化だけを対象にしています。すなわち、ユーザはプログラムを並列に実行するため、コンパイラと実行時システムの動作を明示的に指定します。OpenMP に準拠した実装は、データの依存関係、データの不整合、競合状態、またはデッドロック（これらは、準拠したプログラムにおいても起こりえます）をチェックすることは要求されていません。さらに、準拠した実装は非準拠に分類されるプログラムコードをチェックすることも要求されていません。ユーザは、アプリケーションで OpenMP を使って準拠したプログラムを作成する責任があります。OpenMP は、コンパイラによる自動並列化や自動並列化を補助するための指示文は対象としません。

１．２．用語集

１．２．１．スレッドの概念

スレッド
thread
スタックおよび関連付けられた静的なメモリ（スレッドプライベートメモリと呼ばれる）を持つ実行の実体。

OpenMP スレッド
OpenMP thread
OpenMP 実行時システムが管理するスレッド。

スレッドセーフなルーチン
thread-safe routine
２つ以上のスレッドによって同時に実行されても意図した機能を果たすルーチン。

１．２．２．OpenMP 言語の用語

ベース言語
base language
OpenMP 仕様のベースとなるプログラミング言語。
コメント：
・現在の OpenMP のベース言語の一覧はセクション 1.6 を参照してください。

ベースプログラム
base program
ベース言語で記述されたプログラム。

構造化ブロック
structured block
C/C++では、先頭に入口が一つと末尾に出口が一つある実行文か複合文、あるいは OpenMP 構文。
Fortran では、先頭に入口が一つと末尾に出口が一つある実行文のブロック、または OpenMP 構文。
コメント：
［ベース言語共通］
・分岐の結果として、構造化ブロックへアクセスしてはいけません。
・出口点は構造化ブロックからのの分岐であってはいけません。
［C/C++］
・入口点は setjmp（）の呼出しであってはいけません。
・longjmp（）と throw（）は入口と出口の規則を破ってはいけま

せん。
・構造化ブロックの中で exit（）を呼び出すことができます。
・式文、繰り返し文、選択文、または try ブロックは、{ と } で囲まれた対応する複合文が構造化ブロックであるならば、構造化ブロックとみなすことができます。
（try ブロックとは、C++の例外処理のための構文。ブロック内で例外が発生すると catch ブロックで定義する例外処理を実行します。）
[Fortran]
・構造化ブロックの中の STOP 文は許されます。

指示文
directive

OpenMP プログラムの動作を指定する C/C++の#pragma と Fortran のコメント。
コメント：
・OpenMP 指示文の構文の説明はセクション 2.1 を参照してください。

空白
white space

スペースと水平タブのみから成る空でない文字の並び。

OpenMP プログラム
OpenMP program

ベースプログラムと OpenMP の指示文、実行時ライブラリルーチンから成るプログラム。

準拠したプログラム
conforming program

OpenMP の仕様のすべての規則と制限に従った OpenMP プログラム。

宣言指示文
declarative directive

宣言の文脈中にだけ指定することができる OpenMP 指示文。
宣言指示文に関連するユーザの実行コードはありませんが、関連する一つ以上のユーザの宣言文があります。
コメント：
・threadprivate 指示文だけが宣言指示文です。

実行指示文
executable directive

宣言でない OpenMP 指示文。実行の文脈中に指定することができます。
コメント：
・threadprivate 指示文以外のすべての指示文が実行指示文です。

単独指示文
stand-alone directive

関連するユーザの実行コードがない OpenMP 実行指示文。
コメント：
・barrier、flush、および taskwait 指示文だけが単独指示文です。

単純指示文
simple directive

関連するユーザのコードが単純実行文（複文でなく単文）でなければならない OpenMP 実行指示文。
コメント：
・atomic 指示文だけが単純指示文です。

ループ指示文
loop directive

関連するユーザのコードが構造化ブロックのループでなければならない OpenMP 実行指示文。
コメント：
[C/C++]
・for 指示文だけがループ指示文です。
[Fortran]
・do 指示文と省略可能な end do 指示文だけがループ指示文です。

関連付けられたループ
associated loop(s)

ループ指示文によって制御されるループ。
コメント：
・ループ指示文が collapse 指示節を含む場合、２つ以上のループと関連付けることができます。

構文
construct

OpenMP 実行指示文（Fortran の場合、対になる end 指示文があればそれも含む）ともしあれば関連付けられた文、ループ、または構造化ブロック。ただし、呼び出されるルーチンのコードは含みません。言い換えれば、１つの実行指示文の字句範囲です。

リージョン
region

構文または OpenMP ライブラリルーチンのインスタンスの実行中に遭遇したすべてのコード。リージョンは、呼び出されたルーチン内のすべてのコードと、OpenMP の実装によって暗黙的に生成されたコードも含みます。あるスレッドが遭遇した task 指示文でのタスクの生成は、そのスレッドのリージョンの一部です。しかし、task 指示文と関連付けられた明示的な task リージョンは、そのスレッドのリージョンの一部ではありません。
コメント：
・リージョンは、構文または OpenMP ライブラリルーチンの動的な範囲または実行時の範囲として考えることもできます。
・OpenMP プログラムを実行する時、構文は多くのリージョンを

生成する可能性があります。

活動状態の parallel リージョン
active parallel region

２つ以上のスレッドから成るチームによって実行される parallel リージョン。

非活動状態の parallel リージョン
inactive parallel region

１スレッドだけのチームによって実行される parallel リージョン。

逐次部分
sequential part

OpenMP プログラムの実行中に遭遇したコードのうち、parallel 構文に対応する parallel リージョンにも、task 構文に対応する task リージョンにも含まれないすべてのコード。
コメント：
・逐次部分は、非活動状態の parallel リージョンによって囲まれているかのように実行されます。
・呼び出されたルーチンの実行文は、プログラム実行で逐次部分と任意の数の異なる明示的な parallel リージョンの両方に含まれる可能性があります。

マスタースレッド
master thread

parallel 構文に遭遇してチームを作成し、タスクセットを生成し、そしてスレッド番号０としてタスクの１つを実行するスレッド。

親スレッド
parent thread

parallel 構文に遭遇して parallel リージョンを生成したスレッドは、その parallel リージョンのチームの各スレッドの親スレッドとなります。OpenMP のスレッドに関連した資源について、parallel リージョンのマスタースレッドは親スレッドと同じスレッドとなります。

祖先スレッド
ancestor thread

あるスレッドから見て、親スレッドまたは親スレッドの先祖のスレッドの一つ。

チーム
team

parallel リージョンの実行に参加している１つ以上のスレッドのセット。
コメント：
・活動状態の parallel リージョンでのチームは、マスタースレッドと少なくとも１つの追加スレッドから成ります。
・非活動状態の parallel リージョンでのチームは、マスタースレ

ッドだけです。

初期スレッド
initial thread

逐次部分を実行するスレッド。

暗黙の parallel リージョン
implicit parallel region

OpenMP プログラムの逐次部分を囲む非活動状態の parallel リージョン。

ネスト構文
nested construct

別の構文によって（字句的に）囲まれた構文。

ネストリージョン
nested region

別のリージョンによって（動的に）囲まれたリージョン。
言い換えると、別のリージョンを実行中に遭遇したリージョン。
コメント：
・規格に準拠しているネストと、そうでないネストがあります。ネストの制限については、セクション 2.10 を参照してください。

近接ネストリージョン
closely nested region

他のリージョンの内側にネストしたリージョンで、かつその間に parallel リージョンを挟まないもの。

全スレッド
all threads

OpenMP プログラムの実行に関わるすべての OpenMP スレッド。

カレントチーム
current team

最も内側を囲んでいる parallel リージョンを実行しているチームに属するすべてのスレッド。

遭遇したスレッド
encountering thread

あるリージョンに関し、対応した構文に遭遇したスレッド。

全タスク
all tasks

OpenMP プログラムの実行に関わるすべてのタスク。

カレントチームタスク
current team tasks

最も内側の parallel リージョンを実行しているチームのスレッドが遭遇したすべてのタスク。
parallel リージョンを構成している暗黙のタスクと、その暗黙のタスクの実行中に遭遇した子孫のタスクが、結合タスクセットに含まれることに注意してください。

生成しているタスク
generating task

あるリージョンにおいて、スレッドによるタスクの実行がそのリージョンを生成するタスク。

結合スレッドセット
binding thread set

あるリージョンの実行によって影響を受けるスレッド、あるいはリージョン実行のためのコンテキストを提供するスレッドのセット。
あるリージョンに対する結合スレッドセットは、全スレッド、カレントチーム、または遭遇したスレッドのいずれかになります。
コメント：
・特定のリージョンに対する結合スレッドセットは、この仕様の対応したセクションの中で記述されています。

結合タスクセット
binding task set

あるリージョンの実行によって影響を受けるスレッド、あるいはリージョン実行のためのコンテキストを提供するタスクのセット。
あるリージョンに対する結合タスクセットは、全タスク、カレントチームタスク、または生成しているタスクのいずれかになります。
コメント：
・特定のリージョンに対する結合タスクセットがある場合は、この仕様の対応するセクションの中で記述されています。

結合リージョン
binding region

あるリージョンの実行コンテキストを決定するとともに、その有効域を制限する外側のリージョンを結合リージョンと呼びます。
結合リージョンは、対応する結合スレッドセットが全スレッドまたは遭遇したスレッドであるようなリージョンに対しては定義されていません。また、対応する結合タスクセットが全タスクであるようなリージョンに対しても定義されていません。
コメント：
・ordered リージョンの結合リージョンは、それを囲む最も内側のループリージョンです。
・taskwait リージョンの結合リージョンは、それを囲む最も内側のタスクリージョンです。
・結合スレッドセットがカレントチームであるか、または結合タスクセットがカレントチームタスクであるような他の全てのリージョンでは、結合リージョンは、それを囲む最も内側の parallel リージョンです。

・結合タスクセットが生成しているタスクであるようなリージョンでは、結合リージョンは、生成している task リージョンです。
・結合リージョンである parallel リージョンは活動状態である必要も明示的である必要もありません。
・結合リージョンである task リージョンは明示的である必要はありません。
・リージョンは、それを囲む最も内側の parallel リージョンの外側のリージョンには、決して結合しません。

親なし構文
orphaned construct

結合スレッドセットがカレントチームであるようなリージョンを起こすが、結合リージョンを起こす別の構文の内側にはネストしていない構文。

ワークシェアリング構文
worksharing construct

その構文を実行するチーム内のスレッドの１つによって、各々正確に１回だけ実行される一連の処理単位を定義する構文。
C のワークシェアリング構文は、for、sections、および single です。
Fortran のワークシェアリング構文は、do、sections、single、および workshare です。

逐次ループ
sequential loop

OpenMP ループ指示文に関連付けられていないループ。

バリア
barrier

スレッドチームが遭遇したプログラムの実行内のポイント。
チーム内のすべてのスレッドがバリアに到達し、チームによって生成されたすべての明示的なタスクが完了するまで、チーム内のどのスレッドもそのポイントを超えて実行することがありません。

１．２．３．タスクの用語

タスク task	スレッドがtask構文またはparallel構文に遭遇したときに生成する実行コードとそのデータ環境の特定のインスタンス。 コメント： ・スレッドがタスクを実行するとき、スレッドは task リージョンを生成します。
task リージョン task region	タスクの実行中に遭遇したすべてのコードから成るリージョン。 コメント： ・parallel リージョンは１つ以上の暗黙の task リージョンから成ります。
明示的なタスク explicit task	実行中に task 構文に遭遇したときに生成するタスク。
暗黙のタスク implicit task	暗黙の parallel リージョンが生成したタスク、または実行中にparallel 構文に遭遇したときに生成したタスク。
初期タスク initial task	暗黙の parallel リージョンに関連付けられた暗黙のタスク。
カレントタスク current task	あるスレッドが実行している task リージョンに対応したタスク。
子タスク child task	タスクは自分を生成した task リージョンの子タスク。 子 task リージョンは、自分を生成した task リージョンには含まれません。
子孫タスク descendant task	ある task リージョンの子タスク、またはある task リージョンの子孫 task リージョンの一つの子タスク。
タスクの完了 task completion	タスクを生成した構文に関連付けられた構造化ブロックの終わりに到達したときにタスクは完了します。 コメント：初期タスクはプログラムの出口で完了します。

タスクスケジューリングポイント
task scheduling point

後で再開するために実行をサスペンドできるカレント task リージョン実行のポイント、またはタスクが完了するポイント。実行中のスレッドは、そのポイントで異なった task リージョンに実行を切り替えることができます

コメント：

・タイド task リージョンのタスクスケジューリングポイントは以下にあげるものだけです。

- 遭遇した task 構文
- 遭遇した taskwait 構文
- 遭遇した barrier 指示文
- 暗黙の barrier リージョン
- タイド task リージョンの終わり

タスクの切り替え
task switching

あるタスクから別のタスクへ実行を切り替えるスレッドの動作。

タイドタスク
tied task

task リージョンがサスペンドされたとき、その task リージョンをサスペンドしたのと同じスレッドだけが実行を再開できるタスク。つまり、タスクがスレッドに結び付けられています。

アンタイドタスク
untied task

task リージョンがサスペンドされたとき、チーム内のどのスレッドが実行を再開してもよいタスク。つまり、タスクはスレッドに結び付けられていません。

タスク同期構文
task synchronization construct

taskwait 構文または barrier 構文。

１．２．４．データの用語

変数
variable

プログラムの実行中に、値の定義と再定義ができる名前付きのデータ記憶ブロック。
部分配列と部分列は変数ではありません。

プライベート変数
private variable

同じparallelリージョンに結合しているtaskリージョンのセットにおいて、それぞれの task リージョンが同じ名前で記憶域の異なるブロックにアクセスする変数。
別の変数の一部である変数（配列や構造体の要素）は、他の構成要素と独立してプライベートにはできません。

共有変数
shared variable

同じparallelリージョンに結合しているtaskリージョンのセットにおいて、それぞれの task リージョンが同じ名前で記憶域の同じブロックにアクセスする変数。
別の変数の一部である変数（配列や構造体の要素）は、C++のクラスの静的データメンバーを除いて、残りの構成要素と独立して共有にはできません。

スレッドプライベート変数
threadprivate variable

OpenMP の実装によって複製され、スレッドごとに 1 つのインスタンスが存在する変数。その結果、それぞれのスレッドが同じ名前で異なる記憶域のブロックにアクセスします。
別の変数の一部である変数（配列や構造体の要素）は、C++のクラスの静的データメンバーを除いて、残りの構成要素と独立してスレッドプライベートにはできません。

スレッドプライベートメモリ
threadprivate memory

それぞれのスレッドに関連付けられたスレッドプライベート変数のセット。

データ環境
data environment

あるタスクの実行に関連付けられたすべての変数。タスクのデータ環境は、そのタスクの生成時に生成しているタスクのデータ環境から作られます。

定義
defined

変数において有効な値を持っているという属性。

C では、変数の内容が有効な値を持っているという属性。

C++では、POD（plain old data）型の変数の内容が有効な値を持っているという属性。POD クラス型でない変数では、構築された後、消去されてないという属性。

Fortran では、変数の内容が有効な値を持っているという属性。変数の割付け状態または結合状態では有効な状態を持っているという属性。

コメント：

・定義していない変数に依存するプログラムは、規格に準拠していないプログラムです。

クラス型
class type

C++において、class、struct、または union キーワードのいずれかで宣言された変数。

１．２．５．実装の用語

n レベル並列性のサポート
supporting n levels of parallelism
ある活動状態の parallel リージョンが、（n-1）個の活動状態の parallel リージョンに囲まれることを許すこと。

OpenMP のサポート
supporting OpenMP
少なくとも１レベルの並列性をサポートすること。

ネスト並列性のサポート
Supporting nested parallelism
２レベル以上の並列性をサポートすること。

内部制御変数
internal control variable
OpenMP プログラムで、スレッドまたはタスクセットの実行時の動作を規定する概念上の変数。
コメント：
・頭字語の ICV は、この仕様書で内部制御変数（internal control variable）の意味で使用されています。

準拠した実装
compliant implementation
本仕様書の定義に準拠した任意のプログラムをコンパイル、実行できる OpenMP 仕様の実装。
コメント：
・準拠していないプログラムをコンパイルまたは実行した場合、準拠した実装は不定の振舞いをしてもかまいません。

不定の振舞い
unspecified behavior
OpenMP の仕様で定義していない振舞いや結果、または OpenMP プログラムをコンパイル、実行する前に知ることができない振る舞いや結果。
このような不定の振舞いの原因は以下の通りです。:
・OpenMP 仕様が不定の振舞いを持つと明記している問題。
・準拠していないプログラム。
・　実装依存の振舞いを示す準拠したプログラム。

実装依存
implementation defined

実装がドキュメントに記述しなければならない振舞いで、準拠した実装間でその振舞いが異なることを許されているもの。実装は、この振舞いを不定と定義することを許されています。

コメント：

・実装依存の振舞いの一覧は付録 E に記述されています。

１．３．実行モデル

OpenMP API は、並列実行の fork-join モデルを採用しています。複数のスレッドが OpenMP 指示文によって暗黙的または明示的に定義されたタスクを実行します。OpenMP は、（マルチスレッドによる実行と OpenMP ライブラリを完全サポートした）並列プログラムとしても、（指示文の無視と単純な OpenMP スタブライブラリによる）逐次プログラムとしても、正しく実行できるプログラムをサポートすることを目的にしています。しかし、並列プログラムとしては正しく実行できるが、逐次プログラムとしては正しく実行できないプログラムや、逐次プログラムとして実行したときと並列プログラムとして実行したときに異なった結果となるプログラムを開発することは可能であり、認められています。さらに、数値演算の結合の仕方が変わることにより、使用するスレッド数が変わると結果の数値が異なることがあります。例えば、逐次の加算リダクションは、並列リダクションと異なる加算の結合パターンになることがあります。このように結合の仕方が異なることによって、浮動小数点の加算結果が変わってしまうことがあります。

OpenMP プログラムは、初期スレッドと呼ばれる単一スレッドの実行から始まります。この初期スレッドは、暗黙の task リージョン(初期 task リージョンと呼ばれる)に囲まれているかのように逐次的に実行されます。ここで、初期 task リージョンは、プログラム全体を囲む暗黙の非活動状態の parallel リージョンによって定義されます。

parallel 構文に遭遇したスレッドは、そのスレッド自身とゼロ個以上の追加スレッドから成るチームを生成し、新しいチームのマスタースレッドになります。１スレッドに１つの暗黙のタスクセットが作られます。それぞれのタスクのコードは、parallel 構文内のコードで定義されます。それぞれのタスクはチーム内の別々のスレッドに割り当てられ、タイドになります。すなわち、そのタスクは常に最初に割り当てられたスレッドで実行されます。遭遇したスレッドが実行していたタスクの task リージョンはサスペンドされ、新しいチームのそれぞれのメンバーが暗黙のタスクを実行します。
parallel 構文の終わりには暗黙のバリアがあります。parallel 構文の終わりを過ぎると、parallel 構文に遭遇した時にサスペンドした task リージョンを再開することで、マスタースレッドだけが実行を再開します。１つのプログラムで指定できる parallel 構文の数に制限はありません。

parallel リージョンは、任意にネストすることが可能です。ネスト並列が無効であったり、ネスト並列が OpenMP の実装によってサポートされていない場合、parallel リージョンの内側で parallel 構文に遭遇したスレッドが生成する新しいチームは、そのスレッドだけになります。しかし、ネスト並列がサポートされていて、かつそれが有効である場合、新しいチームは２つ以上のスレッドから成ることができます。

チームがワークシェアリング構文に遭遇すると、その構文の内側の処理がチームのメンバーに分配され、それぞれのスレッドが個別に実行するのではなく、協調した実行をします。ワークシェアリング構文の終わりには省略可能なバリアがあります。ワークシェアリング構文が終わった後、チーム内のすべてのスレッドはコードの重複実行を再開します。

スレッドが task 構文に遭遇すると、新しい明示的なタスクを生成します。処理を実行できるスレッドの空き状況に応じて、明示的に生成されたタスクの実行をカレントチームのスレッドの一つに割り当てます。このように、新しいタスクの実行がすぐに始まることもあれば、後になるまで延期されることもあります。他のタスクを実行するために、スレッドはタスクスケジューリングポイントで現在の task リージョンをサスペンドすることが認められています。タイドタスクの task リージョンをサスペンドした場合、最初に割り当てられたスレッドがサスペンドした task リージョンの実行を再開します。アンタイドタスクの task リージョンをサスペンドした場合、任意のスレッドが実行を再開することができます。アンタイド task リージョンの中では、タスクスケジューリングポイントになり得るのは、リージョン内の実装依存の任意のポイントです。タイド task リージョンの中では、タスクスケジューリングポイントになり得るのは、task 構文、taskwait 構文、暗黙/明示的な barrier 構文、およびタスクの完了だけです。ある parallel リージョンに結合しているすべての明示的なタスクは、マスタースレッドがリージョンの終わりの暗黙のバリアを出る前に、完了することが保証されます。ある parallel リージョンに結合しているすべての明示的なタスクのサブセットの完了は、タスク同期構文を使用することで指定できます。暗黙の parallel リージョンに結合しているすべての明示的なタスクは、プログラムの終了までに完了することが保証されます。

parallel リージョンのタスクとデータアクセスを協調して動作させるために、OpenMP の同期構文やライブラリルーチンが利用できます。さらに、ライブラリルーチンと環境変数は、OpenMP プログラムの実行時環境の制御や問い合わせに利用できます。

OpenMP は、並列実行中に同じファイルに対して行う入出力が同期していることは保証していません。この場合、プログラマは提供されている同期構文やライブラリルーチンを使って、入出力を実行する文やルーチンを同期させる責任があります。スレッドがそれぞれ異なるファイルをアクセスする場合は、プログラマが同期を行う必要はありません。

１．４．メモリモデル

１．４．１．OpenMP メモリモデルの構成

OpenMP は、’relaxed-consistency’の共有メモリモデルを提供します。
すべての OpenMP のスレッドは、変数の書き込み・読み出しを行う場所（メモリと呼ばれる）にアクセスします。さらに、各スレッドがそれぞれメモリの一時的なビュー（temporary view）を持つことができます。各スレッドのメモリの一時的なビューは OpenMP メモリモデルの必須の要素ではありませんが、マシンレジスタ、キャッシュ、または他のローカルな記憶域のようにスレッドとメモリとの間に介在する任意の構造に相当します。メモリの一時的なビューはスレッドが変数をキャッシュし、それによって変数を参照するたびにメモリへアクセスすることを避けられます。また、各スレッドは、他のスレッドからアクセスされない別のタイプのメモリ（スレッドプライベートメモリと呼ばれる）にアクセスできます。

訳者注：

relaxed-consistency は、メモリへの read と write や write と write について、プログラムの順序を必ずしも守らないメモリモデルです。Total Store Ordering（read が write を追い越す）、Partial Store Ordering（read および write が write を追い越す）などがあります。対語は sequential consistency になります。

データ共有属性の指示節を指定できる指示文は、指示文に関連付けられた構造化ブロックの中で使用されている変数への２種類のアクセス（共有とプライベート）を決定します。構造化ブロックで参照しているぞれぞれの変数には、プログラムでその構文のすぐ外側に存在する同じ名前のオリジナル変数があります。構造化ブロックでの共有変数への参照は、オリジナル変数への参照になります。構造化ブロックでのプライベート変数の参照では、オリジナル変数の新しいバージョン（オリジナルと同じ型および同じサイズを持つ）を指示文に関連付けられたコードを含むタスクのメモリに生成します。新しいバージョンの生成はオリジナル変数の値を変更しません。しかし、指示文に関連付けられたリージョンの実行中にオリジナル変数へアクセスすることによる影響は不定となります。詳細についてはセクション 2.9.3.3 を参照してください。

構造化ブロック内のプライベート変数の参照は、オリジナル変数に対する現在のタスクのプライベートバージョンへの参照となります。オリジナル変数の値とプライベート変数の初期値や最終値との関係は、それを指定したその指示節（clause）に依存します。この問題についての詳細は、プライベート化に関する他の問題とあわせてセクション 2.9 で述べられています。

複数のスレッドによる、同じ変数、または同じ変数の一部である異なった変数（配列や構造体の要素など）への非同期のメモリアクセスが、互いにアトミックになるようなメモリアクセスの最小サイズは実装依存です。アトミック性に関するその他の制限（アラインメントなど）も実装依存です。

変数への単一のアクセスを複数のロードまたはストア命令で実装している可能性があります。そのため、同じ変数への他のアクセスに対してアトミックであることは保証されていません。実装依存である最小サイズより小さい変数や C/C++のビットフィールドへのアクセスは、それより大きなメモリ単位での参照、変更、上書きによって実装されている可能性があります。そのため、同じメモリユニットに含まれる変数またはフィールドの他の更新と干渉する可能性があります。

上で述べたアトミックに関する事例を含め、同じメモリユニットに複数のスレッドが同期しないで書き込むとデータ競合が発生します。同じように、上で述べたアトミックに関する事例を含め、少なくとも 1 つのスレッドがあるメモリユニットを参照し、少なくとも 1 つのスレッドが同じメモリユニットに同期しないで書き込む場合もデータ競合が発生します。データ競合が発生した場合のプログラムの結果は不定となります。

内側にネストする parallel リージョンを生成した task リージョンのプライベート変数は、内側の parallel リージョンの暗黙のタスクで共有変数にすることができます。ある task リージョン中のプライベート変数は、その task リージョンの実行中に生成された明示的な task リージョンで共有にできます。しかし、プログラマは適切な同期をとって、変数を共有している明示的な task リージョンが完了する前に変数の生存期間が終了しないことを保証する責任があります。あるタスクから別のタスクのプライベート変数へこれ以外でアクセスすると不定の振舞いになります。

１．４．２．フラッシュ操作

あるスレッドが持つメモリに対する一時的なビューが常にメモリの内容と一致していることは要求されていないため、メモリモデルは’relaxed-consistency’（緩い一貫性）となっています。変数に書き込まれた値は、後でメモリへの書き込みが強制されるまでスレッドの一時的なビューにとどめておくことができます。同様に、メモリからの読み込みを強制されていなければ、変数の参照でもスレッドの一時的なビューの値を使用することができます。OpenMP のフラッシュ操作は、一時的なビューとメモリの内容の一貫性をとることを強制します。

フラッシュ操作は、フラッシュセットと呼ばれる変数の集合に対して適用されます。フラッシュ操作は、通常であれば実装によって行われることがあるメモリ操作の順序を入れ替える最適化の適用を制限します。ある変数を対象にしたフラッシュ操作に関して、その変数に対するメモリ操作のコードや、フラッシュ操作のコードの順序を実装が入れ替えることはできません。

もし、あるスレッドが、共有変数を最後にフラッシュした後で、その変数の一時的なビューに書き込みを行っていた場合、次に、その変数のフラッシュを実行したとき、その共有変数のフラッシュはメモリに値の書き込みが終わるまで完了しません。同じ変数に対する２つのフラッシュの間にスレッド

がその変数に複数回の書き込みを行った場合、フラッシュは最後に書き込まれた値がメモリに書き込まれることを保証します。スレッドが実行する変数のフラッシュにより、その変数の一時的なビューが破棄されます。これは、その変数に対する次のメモリ操作が読み込みである場合に、スレッドが一時的なビューに値を再び保存するために、値をメモリから読み込みます。スレッドがフラッシュを実行すると、フラッシュ対象変数に関するフラッシュの後にあるメモリ操作は、同じスレッドのフラッシュが完了するまで開始されません。ある変数の集合に対してスレッドが実行するフラッシュは、フラッシュ前にそのスレッドが行った変数の集合へのすべての書き込みが他のスレッドにメモリ上で参照可能となり、またそのスレッドにおける関連するすべての変数の一時的なビューが破棄された時点で完了すると定義されます。

フラッシュ操作は、スレッドの一時的なビューとメモリの間の一貫性を保証します。そのため、フラッシュ操作は、あるスレッドが変数に書き込んだ値を２番目のスレッドが読み込むことを保証するために使用できます。これを実現するため、２番目のスレッドは前のフラッシュの後にその変数に書き込みをしていないこと、そして以下のイベントが指定した順序で起こることをプログラマが保証しなければなりません。

１．最初のスレッドが値を変数に書き込む。
２．最初のスレッドがその変数をフラッシュする。
３．２番目のスレッドがその変数をフラッシュする。
４．２番目のスレッドがその変数の値を読み込む。

注：セクション 2.8 とセクション 3.3 に記述している OpenMP の同期操作は、この順序を強制する手段として推奨されています。変数を利用した同期も可能ですが、セクション A.2 に示すように、適切にフラッシュ操作を行うタイミングが難しいために推奨されていません。

１．４．３．OpenMP におけるメモリの一貫性

OpenMP が提供している緩いメモリの一貫性（'relaxed memory consistency'）は、S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial", IEEE Computer, 29(12), pp.66-76, December 1996 で記述している weak ordering と類似しています。Weak ordering は、一部のメモリ操作を同期操作として定義し、それらがお互いの順序関係を保つことを要求しています。OpenMP のコンテキストでは、同じ変数に対する２つのフラッシュは同期操作になります。OpenMP では、単一のスレッドによって実行されるメモリ操作の順序入れ替えに対して他のいかなる制限も適用しません。OpenMP のメモリモデルは weak ordering よりもわずかに弱いです。なぜなら、フラッシュセットに重なりのない複数のフラッシュは互いに実行順序付けされていないからです。

セクション 1.4.2 のフラッシュ操作に関する順序入れ替えの制限は、以下のことを保証します。

- ２つの異なるスレッドで実行するフラッシュのフラッシュセットに重複がある場合、２つのフラッシュは、すべてのスレッドから見える一定の逐次順序があるように完了しなければならない。
- １つのスレッド内の２つのフラッシュのフラッシュセットに重複がある場合、その２つのフラッシュはスレッドが実行しているプログラムに記述された順序で完了したように見えなければならない。
- ２つのフラッシュのフラッシュセットに重複がない場合、これらのフラッシュの順序に制限はありません。

フラッシュ操作は flush 指示文を使って指定できます。また、OpenMP プログラム中のさまざまな場所で暗黙のうちに実行されています。詳しくはセクション 2.8.6 を参照してください。メモリモデルを示した例はセクション A.2 を参照してください。

１．５．OpenMP の準拠

１、２、３、４章で記述されているる文法と意味論に従うすべての規格に準拠したプログラムをコンパイル、実行できる場合、かつその場合に限り、OpenMP API の実装は OpenMP に準拠しています。付録A、B、C、D、E、F と Notes（注）と記した部分（セクション 1.7 を参照）は、単に参考情報提供のみを目的としており仕様の一部ではありません。

OpenMP API は、実装がサポートしているベース言語の文脈において機能する構文を定義しています。ベース言語がこのドキュメントに記載されている言語構文をサポートしていない場合は、

OpenMP に準拠した実装でそれをサポートする必要はありません。しかし、例外として Fortran の場合、実装は、指示文と API ルーチンの名前で大文字と小文字を区別しないようにすること、および７文字以上の識別子を使えるようにしなければなりません。

準拠した実装において、ベース言語が提供するライブラリや組込みルーチンは、すべてスレッドセーフでなければなりません。これに加えて、ベース言語の実装もまたスレッドセーフでなければなりません。（例えば、Fortran では、ALLOCATE 文と DEALLOCATE 文はスレッドセーフでなければなりません。）それらのルーチンを異なるスレッドで非同期で並列に使用した場合も、正しい結果を出さなければなりません。（ただし、乱数発生ルーチンの場合などが逐次実行の結果と一致する必要はありません。）

Fortran 90 と Fortran 95 では、明示的に初期化される変数は暗黙的に SAVE 属性を持ちます。これは FORTRAN 77 の場合には当てはまりません。しかし、OpenMP Fortran に準拠した実装では、ベース言語のバージョンに関係なく、そのような変数に SAVE 属性を与えなければなりません。

付録 E に、実装依存である OpenMP API の要素が列挙されています。準拠した実装は、付録 E の各項目に対する振舞いを定義して文書化することが求められています。

１．６．参照規格

・ISO/IEC 9899:1990, *Information Technology - Programming Languages - C.*
この OpenMP API 仕様は、C90 として ISO/IEC 9899:1990 を参照しています。

・ISO/IEC 9899:1999, *Information Technology - Programming Languages - C.*
この OpenMP API 仕様は、C99 として ISO/IEC 9899:1999 を参照しています。

・ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++.*
この OpenMP API 仕様は、C++として ISO/IEC 14882:1998 を参照しています。

・ISO/IEC 1539:1980, *Information Technology - Programming Languages - Fortran.*
この OpenMP API 仕様は、Fortran 77 として ISO/IEC 1539:1980 を参照しています。

・ISO/IEC 1539:1991, *Information Technology - Programming Languages - Fortran.*
この OpenMP API 仕様は、Fortran 90 として ISO/IEC 1539:1991 を参照しています。

・ISO/IEC 1539-1:1997, *Information Technology - Programming Languages - Fortran.*
この OpenMP API 仕様は、Fortran 95 として ISO/IEC 1539-1:1997 を参照しています。

この OpenMP API 仕様が C、C++、または Fortran に言及するとき、その言及は実装がサポートするベース言語に対するものです。

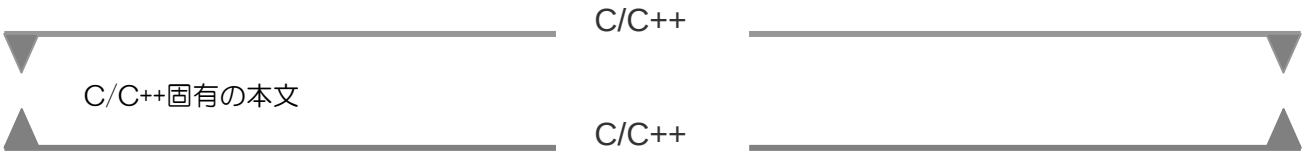
１．７．ドキュメントの構成

このドキュメントの残りの部分は、以下の構成となっています。

- 第２章：指示文
- 第３章：実行時ライブラリルーチン
- 第４章：環境変数
- 付録 A：例
- 付録 B：実行時ライブラリのスタブ
- 付録 C：OpenMP C と C++の文法
- 付録 D：引用仕様宣言
- 付録 E：OpenMP の実装依存の振舞い
- 付録 F：バージョン 2.5 からバージョン 3.0 への変更点

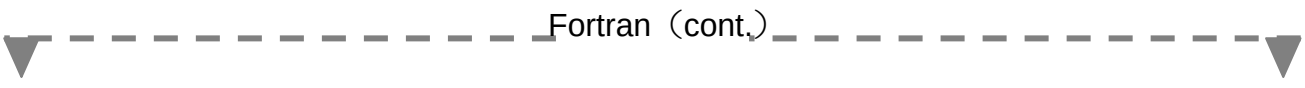
このドキュメントのある部分は、特定のベース言語で書かれたプログラムだけを対象にしています。

ベース言語が C または C++であるプログラムにだけ適用される文は、以下のように示されます。



ベース言語が Fortran であるプログラムにだけ適用される文は、以下のように示されます。

ページ全体が Fortran にだけ適用される文だけで構成されているとき、ページの先頭に以下のような印が付けられます。



一部の文は単に参考情報を提示するためのものであり、仕様の一部ではありません。そのような文は、注記として以下のように示されます。

注：

第２章

指示文

本章では、OpenMP 指示文の文法（syntax）と振舞い（behavior）について、以下のセクションに分けて説明をしています。

- 言語固有の指示文フォーマット（セクション 2.1）
- 条件付きコンパイルを制御する機構（セクション 2.2）
- OpenMP API の内部制御変数の制御（セクション 2.3）
- それぞれの OpenMP 指示文の詳細（セクション 2.4～2.10）

C/C++

C/C++においては、OpenMP 指示文は、C と C++の標準規格によって提供されている #pragma 機構を使用することによって指定されます。

C/C++

Fortran

Fortran においては、OpenMP 指示文は、固有の接頭辞で識別される特別なコメントを使用することによって指定されます。また、条件付きコンパイルのための特別なコメント形式が利用可能です。。

Fortran

したがって、もし、OpenMP をサポートしていない、または、利用できないときは、コンパイラは、OpenMP 指示文と条件付きコンパイルを無視することができます。準拠した実装は、すべての OpenMP 指示文と OpenMP 条件付きコンパイル機構のサポートが有効であることを保証するオプションまたはインタフェースを提供しなければなりません。これ以降のドキュメントでは、OpenMP のコンパイルという言葉は、OpenMP の仕様が利用できるコンパイルという意味で使用します。

Fortran

制限事項

以下の制限は、すべての OpenMP 指示文に適用されます。

- OpenMP 指示文は、PURE または ELEMENTAL 手続きに現れてはいけません。

Fortran

２．１．指示文の形式

C/C++

C/C++の OpenMP 指示文は、pragma プリプロセッサ指示文で指定します。
OpenMP 指示文の文法は、付録 C に示す文法によって正式に規定されています。簡略的には、以下の形式です。

```
#pragma omp directive-name  [clause[ [,] clause] ... ]  new-line
```

それぞれの指示文は、#pragma omp で始まります。指示文の残りの部分は、コンパイラ指示文に関する C および C++標準の慣習に従います。特に、空白は、# の前後で使用することができ、指示文中の単語を区切るために使用しなければならない場合もあります。#pragma omp に続くプリプロセッサのトークンは、マクロ置き換えの対象になります。

指示文は、大文字と小文字が区別されます。

OpenMP の実行指示文は、多くとも一つの後続の文に対して適用され、それは、構造化ブロックでなければなりません。

C/C++

Fortran

Fortran の OpenMP 指示文は、以下のように指定します。

```
sentinel directive-name  [clause[ [,] clause] ... ]  new-line
```

すべての OpenMP コンパイラ指示文は、指示文接頭辞で始まらなければなりません。接頭辞の形式は、セクション 2.1.1 とセクション 2.1.2 で記述されているように、固定形式と自由形式のソースファイルとで異なります。

指示文では、大文字と小文字は区別されません。指示文は、継続した文の中に組み込むことはできません。また、文を指示文の中に組み込むこともできません。

表記を単純化するために、指定のない限り、以降のドキュメントでは、Fortran の OpenMP 指示文の文法を記述するのには、自由形式を使用します。

Fortran

指示文ごとに１つの指示文名（directive-name）しか指定できません。（複合指示文にも当てはまることに注意してください。セクション 2.6 を参照してください。） 指示文中に現れる指示節（clause）の順序は、重要ではありません。それぞれの指示節の説明において列挙されている制限に従って、指示文中の指示節は、必要に応じて繰り返すことができます。

いくつかのデータ共有属性指示節（セクション 2.9.3）、データーコピー指示節（セクション 2.9.4）、threadprivate 指示文（セクション 2.9.2）、および flush 指示文（セクション 2.8.6）には、リストを指定することができます。リストは、１つ以上のリストアイテムのコンマで区切られた集まりから成ります。

C/C++

リストアイテムは、変数名です。リストが現れる指示節や指示文については、記述されているそれぞれのセクションでの制限に従ってください。

C/C++

Fortran

リストアイテムは、変数名または（スラッシュで囲まれた）共通ブロック名です。リストが現れる指示節や指示文については、記述されているそれぞれのセクションでの制限に従ってください。

Fortran

Fortran

２．１．１．固定形式の指示文

固定形式のソースファイルでは、以下の接頭辞が認識されます。

```
!$omp  |  c$omp  |  *$omp
```

接頭辞は、１カラム目から始まり、空白を挟まない１語として記述しなければなりません。Fortran の固定形式の行の長さ、空白、継続、およびカラムの規則は、指示文の行にも適用されます。指示文の開始行は、６カラム目が空白または０でなければならず、指示文の継続行は、６カラム目が空白または０以外の文字でなければなりません。

コメントは、指示文と同じ行に現れてもかまいません。感嘆符記号（！）が６カラム目以降に現れたとき、そこからコメントが始まります。コメントは、ソース行の終端まで続き、無視されます。指示文の開始行または継続行の接頭辞に続く、最初に現れた空白以外の文字が感嘆符の場合、その行は無視されます。

注： 以下の例において、指示文を指定する３つの形式は等価です。（最初の行は、最初の９カラムの位置を表しています。）

```
c23456789
!$omp parallel do shared (a,b,c)

c$omp parallel do
c$omp+shared (a,b,c)

c$omp paralleldoshared (a,b,c)
```

２．１．２．自由形式の指示文

自由形式のソースファイルでは、以下の接頭辞が認識されます。

```
!$omp
```

この接頭辞は、空白（スペースとタブ文字）だけが先行しているのなら、どのカラムに現れてもかまいません。接頭辞は、他の文字を挟まない１語として記述しなければなりません。Fortran の自由形式の行の長さ、空白、継続の規則は、指示文の行にも適用されます。指示文の開始行は、接頭辞の後に空白がなければなりません。継続される指示文の行では、指示文中のコメントより前にある空白以外の最後の文字が＆（アンパサンド）でなければなりません。指示文の継続行には、接頭辞の後に空白（省略可能）を前後に付加した＆（アンパサンド）を記述することができます。

コメントは、指示文と同じ行に記述することができ、感嘆符（！）からコメントが開始します。コメントは、ソース行の終わりまで続き、無視されます。もし、指示文の接頭辞の後の最初の空白でない文字が感嘆符（！）である場合、その行は無視されます。

自由形式では、指示文中の隣接するキーワードを区切るために、一つ以上の空白または水平タブ文字を使用しなければなりません。ただし以下の場合(２つのキーワードの間の空白は省略可能です)を除きます。

```
end critical
end do
end master
end ordered
end parallel
end sections
end single
end task
end workshare
parallel do
parallel sections
parallel workshare
```

注： 以下の例において、指示文を指定する３つの形式は等価です。（最初の行は、最初の９カラムの位置を表しています。）

```
!23456789
      !$omp parallel do &
              !$omp parallel shared (a,b,c)

      !$omp parallel &
     !$omp&do shared (a,b,c)
!$omp paralleldo shared (a,b,c)
```

Fortran

２．２．条件付きコンパイル

プリプロセッサをサポートしている実装では、_OPENMP マクロ名は、実装がサポートしている OpenMP API のバージョンの十進値 yyymm（yyyy と mm は年と月を示す）を持つように定義されています。

もし、このマクロが、#define または#undef のプリプロセッサの指示文の対象である場合は、振舞いは不定となります。

条件付きコンパイルの例は、セクション A.3 を参照してください。

Fortran

以下のセクションで述べるように、OpenMP API に従う実装では、Fortran 行を条件付きコンパイルすることが可能でなくてはなりません。

２．２．１．固定形式の条件付きコンパイルの接頭辞

固定形式のソースファイルでは、以下の条件付きコンパイルの接頭辞が認識されます。

```
!$  |  *$  |  c$
```

条件付きコンパイルの利用を可能にするために、条件付きコンパイルの接頭辞がある行は、次の基準を満たさなければなりません。

- 接頭辞は、１カラム目から始まり、空白を挟まない１語として記述しなければなりません。
- 接頭辞が２つの空白に置き換えられた後、開始行では、６カラム目が空白または０であり、１から５カラム目が空白と数字だけでなければなりません。
- 接頭辞が２つの空白に置き換えられた後、継続行では、６カラム目が空白と０以外の文字で、１から５カラム目は空白だけでなければなりません。

もし、これらの基準が満たされた場合、接頭辞が２つの空白によって置き換えられます。もし、これらの基準が満たされない場合は、その行は変更されません。

注： 以下の例において、固定ソース形式での条件付きコンパイルの指定の２つの形式は等価です。（最初の行は、最初の９カラムの位置を表しています。）

```
c23456789
!$ 10  iam = omp_get_thread_num ( )  +
!$   &     index

#ifdef  _OPENMP
    10  iam = omp_get_thread_num ( )  +
      &  index
#endif
```

２．２．２．自由形式の条件付きコンパイルの接頭辞

自由形式のソースファイルでは、以下の条件付きコンパイルの接頭辞が認識されます。

!$

条件付きコンパイルの利用を可能にするために、条件付きコンパイルの接頭辞がある行は、次の基準を満たさなければなりません。

- 接頭辞は、任意のカラムに現れることができますが、先行していいのは空白だけです。
- 接頭辞は、空白を挟まない１語として記述されなければなりません。
- 開始行は、接頭辞の後に空白がなければなりません。
- 継続行は、条件付きコンパイルの行の中に現れるコメントよりも前にある空白以外の最後の文字が、＆（アンパサンド）でなければなりません。（継続行には、接頭辞の後に空白（省略可能）を前後に付加した＆（アンパサンド）を記述することができます。）

もし、これらの基準が満たされた場合、接頭辞が２つの空白によって置き換えられます。もし、これらの基準が満たされない場合は、行は変更されません。

注： 以下の例において、自由ソース形式での条件付きコンパイルの指定の２つの形式は等価です。（最初の行は、最初の９カラムの位置を表しています。）

```
c23456789
    !$  iam = omp_get_thread_num ( )  +  &
    !$&    index

#ifdef  _OPENMP
     iam = omp_get_thread_num ( )   +  &
          index
#endif
```

Fortran

２．３．内部制御変数

OpenMP の実装は、あたかも OpenMP プログラムの振舞いを制御する内部制御変数（ICV）があるかのように動作しなければなりません。これらの内部制御変数には、将来の parallel リージョンのために使用するスレッドの数、ワークシェアリングループのために使用するスケジュール、およびネスト並列が利用可能かどうかなどの情報が保持されます。内部制御変数には、プログラムの実行中のさまざまな時点（以下に述べます）で値が設定されます。それら内部制御変数は、実装自身によって初期化され、OpenMP 環境変数や OpenMP API ルーチンの呼び出しを通して値を設定することができます。また、プログラムは、OpenMP API ルーチンを通してのみ、これら内部制御変数の値を参照することができます。

詳細な説明の目的のために、このドキュメントでは、特定の名前で内部制御変数に言及します。しかし、実装は、これらの名前を使用することや、内部制御変数へアクセスする方法として、セクション 2.3.2 で示されているもの以外を提供することは要求されていません。

２．３．１．内部制御変数の説明

次の内部制御変数には、parallel リージョンの振舞いに影響する値が保存されます。

- *dyn-var* は、遭遇した parallel リージョンのためにスレッド数の動的調整が可能かどうかを制御します。この内部制御変数は、タスクごとに１つのコピーを持ちます。
- *nest-var* は、遭遇した parallel リージョンのためにネスト並列が可能かどうかを制御します。この内部制御変数は、タスクごとに１つのコピーを持ちます。
- *nthreads-var* は、遭遇した parallel リージョンのために要求されるスレッド数を制御します。この内部制御変数は、タスクごとに１つのコピーを持ちます。
- *thread-limit-var* は、OpenMP プログラムに参加するスレッドの最大数を制御します。この内部制御変数は、プログラム全体で１つのコピーを持ちます。
- *max-active-levels-var* は、ネストされた活動状態の parallel リージョンの最大数を制御します。この内部制御変数は、プログラム全体で１つのコピーを持ちます。

次の内部制御変数には、ループリージョンの操作に影響する値が保存されます。

- *run-sched-var* は、schedule(runtime)指示節がループリージョンに対して使用するスケジュールを制御します。この内部制御変数は、タスクごとに１つのコピーを持ちます。
- *def-sched-var* は、実装で定義されたループリージョンのデフォルトのスケジューリングを制御します。この内部制御変数は、プログラム全体で１つのコピーを持ちます。

次の内部制御変数には、プログラムの実行に影響する値が保存されています。

- *stacksize-var* は、OpenMP の実装が生成するスレッドのためのスタックサイズを制御します。この内部制御変数は、プログラム全体で１つのコピーを持ちます。
- *wait-policy-var* は、ウェイトしているスレッドに期待する振舞いを制御します。この内部制御変数は、プログラム全体で１つのコピーを持ちます。

２．３．２．内部制御変数の値の変更と参照

以下の表は、内部制御変数の値の参照方法ならびに初期値を示しています。

内部制御変数	スコープ	値の変更方法	値の参照方法	初期値
dyn-var	タスク	OMP_DYNAMIC omp_set_dynamic（）	omp_get_dynamic（）	以下のコメントを参照
nest-var	タスク	OMP_NESTED omp_set_nested（）	omp_get_nested（）	*false*
nthreads-var	タスク	OMP_NUM_THREADS omp_set_num_threads（）	omp_get_max_threads（）	実装依存
run-sched-var	タスク	OMP_SCHEDULE omp_set_schedule（）	omp_get_schedule（）	実装依存
def-sched-var	グローバル	（なし）	（なし）	実装依存
stacksize-var	グローバル	OMP_STACKSIZE	（なし）	実装依存
wait-policy-var	グローバル	OMP_WAIT_POLICY	（なし）	実装依存
thread-limit-var	グローバル	OMP_THREAD_LIMIT	omp_get_thread_limit（）	実装依存
max-active-levels-var	グローバル	OMP_MAX_ACTIVE_LEVELS omp_set_max_active_levels（）	omp_get_max_active_levels（）	以下のコメントを参照

コメント：

- 実装がスレッド数の動的調整をサポートしている場合、*dyn-var* の初期値は、実装依存となります。その他の場合は、初期値は、*false* です。
- *max-active-levels-var* の初期値は、実装がサポートしている並列性のレベルの数となります。詳細は、セクション 1.2.5 の「n レベル並列のサポート」の定義を参照してください。

初期値が代入された後（ただし、OpenMP 構文または OpenMP API ルーチンの実行前）、ユーザによって設定された OpenMP 環境変数の値が読み込まれ、それに従って関連する内部制御変数が適切に変更されます。その後は、OpenMP の環境変数を変更しても、内部制御変数に影響しません。

OpenMP 構文の指示節（clause）は、どんな内部制御変数の値も変更しません。

2．3．3．タスクごとの内部制御変数の役割

それぞれの task リージョンは、内部制御変数 *dyn-var*、*nest-var*、*nthreads-var*、および *run-sched-var* のコピーを持ちます。タスクの実行中に、task または parallel 構文に遭遇したとき、生成された子タスクは、これら内部制御変数の値を、生成した（親）タスクから継承します。omp_set_num_threads()、omp_set_dynamic()、omp_set_nested()、および omp_set_schedule() の呼出しでは、その結合している task リージョンに対応した内部制御変数だけが変更されます。

schedule（runtime）が指定されたループワークシェアリングリージョンに遭遇したとき、その結合している parallel リージョンを構成しているすべての暗黙の task リージョンは、同じ値の *run-sched-var* を持たなければなりません。もしそうでない場合は、振舞いが不定となります。

２．３．４．内部制御変数のオーバーライドの関係

さまざまな構文の指示節、OpenMP API ルーチン、環境変数、および内部制御変数の初期値の間のオーバーライドの関係は、以下の表に示されています。

構文の指示節（clause）	API ルーチンの呼出しによるオーバーライド	環境変数の設定によるオーバーライド	初期値のオーバーライド
（なし）	omp_set_dynamic（）	OMP_DYNAMIC	*dyn-var*
（なし）	omp_set_nested（）	OMP_NESTED	*nest-var*
num_threads	omp_set_num_threads（）	OMP_NUM_THREADS	*nthreads-var*
schedule	omp_set_schedule（）	OMP_SCHEDULE	*run-sched-var*
schedule	（なし）	（なし）	*def-sched-var*
（なし）	（なし）	OMP_STACKSIZE	*stacksize-var*
（なし）	（なし）	OMP_WAIT_POLICY	*wait-policy-var*
（なし）	（なし）	OMP_THREAD_LIMIT	*thread-limit-var*
（なし）	omp_set_max_active_levels（）	OMP_MAX_ACTIVE_LEVELS	*max-active-levels-var*

相互参照：

- parallel 構文については、セクション 2.4 を参照してください。
- num_threads 指示節については、セクション 2.4.1 を参照してください。
- schedule 指示節については、セクション 2.5.1.1 を参照してください。
- ループ構文については、セクション 2.5.1 を参照してください。
- omp_set_num_threads ルーチンについては、セクション 3,2,1 を参照してください。
- omp_get_max_threads ルーチンについては、セクション 3.2.3 を参照してください。
- omp_set_dynamic ルーチンについては、セクション 3.2.7 を参照してください。
- omp_get_dynamic ルーチンについては、セクション 3.2.8 を参照してください。
- omp_set_nested ルーチンについては、セクション 3.2.9 を参照してください。

- omp_get_nested ルーチンについては、セクション 3.2.10 を参照してください。
- omp_set_schedule ルーチンについては、セクション 3.2.11 を参照してください。
- omp_get_schedule ルーチンについては、セクション 3.2.12 を参照してください。
- omp_get_thread_limit ルーチンについては、セクション 3.2.13 を参照してください。
- omp_set_max_active_levels ルーチンについては、セクション 3.2.14 を参照してください。
- omp_get_max_active_levels ルーチンについては、セクション 3.2.15 を参照してください。
- OMP_SCHEDULE 環境変数については、セクション 4.1 を参照してください。
- OMP_NUM_THREADS 環境変数については、セクション 4.2 を参照してください。
- OMP_DYNAMIC 環境変数については、セクション 4.3 を参照してください。
- OMP_NESTED 環境変数については、セクション 4.4 を参照してください。
- OMP_STACKSIZE 環境変数については、セクション 4.5 を参照してください。
- OMP_WAIT_POLICY 環境変数については、セクション 4.6 を参照してください。
- OMP_MAX_ACTIVE_LEVELS 環境変数については、セクション 4.7 を参照してください。
- OMP_THREAD_LIMIT 環境変数については、セクション 4.8 を参照してください。

2．4．parallel 構文

概要

この基本的な構文は、並列実行を開始します。OpenMP の実行モデルの一般的な説明については、セクション 1.3 を参照してください。

文法

C/C++

parallel 構文の文法は、以下の通りです。

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

ここで、指示節（*clause*）は、以下のいずれかです。

```
if （scalar-expression）
num_threads （integer-expression）
default（shared | none）
private（list）
firstprivate（list）
```

shared (*list*)
copyin (*list*)
reduction (*operator*: *list*)

C/C++

Fortran

parallel 構文の文法は、以下の通りです。

```
!$omp parallel   [clause[ [,] clause] ...]
   structured-block
!$omp end parallel
```

ここで、指示節（*clause*）は、以下のいずれかです。

if （*scalar-logical-expression*）
num_threads （*scalar-integer-expression*）
default (private | firstprivate | shared | none)
private (*list*)
firstprivate (*list*)
shared (*list*)
copyin (*list*)
reduction ({*operator* | *intrinsic_procedure_name*}: *list*)

end parallel 指示文は、parallel 構文の終わりを示します。

Fortran

結合

parallel リージョンに対する結合スレッドセットは、遭遇したスレッドです。遭遇したスレッドは、新しいチームのマスタースレッドになります。

説明

スレッドが parallel 構文に遭遇すると、スレッドのチームが parallel リージョンを実行するために生成されます。チームに属するスレッドの数を決定する方法、および if 指示節と、num_threads 指示節の評価に関してさらに情報を得るには、セクション 2.4.1 を参照してください。parallel 構文に遭遇したスレッドは、新しいチームのマスタースレッドとなり、新しい parallel リージョンが存続する間スレッド番号0となります。マスタースレッドを含む新しいチーム内のすべてのスレッドは、リー

ジョンを実行します。一度、チームが生成されると、そのチーム内のスレッド数は、parallel リージョンが存続する間一定となります。
parallel リージョンの中では、スレッド番号は、それぞれのスレッドに一意に対応付けられます。スレッド番号は、マスタースレッドの 0 から始まり、チーム内のスレッド数よりも 1 少ない数までの連続した番号となります。スレッドは、omp_get_thread_num ライブラリルーチンを呼び出すことにより、自分自身のスレッド番号を得ることができます。

チーム内のスレッド数と等しい数の暗黙のタスクのセットが、遭遇したスレッドによって生成されます。parallel 構文の構造化ブロックは、それぞれの暗黙のタスクで実行されるコードを決定します。それぞれのタスクは、チーム内の異なったスレッドに割り当てられ、タイドになります。遭遇したスレッドが実行していたタスクの task リージョンはサスペンドされ、チーム内のそれぞれのスレッドは、暗黙のタスクを実行します。それぞれのスレッドは、他のスレッドと異なった文のパスを実行することができます。

実装は、タスクスケジューリングポイントにおいて、暗黙のタスクの実行をサスペンドし、その実行が後に再開されるまで、チーム内のどのスレッドが生成したどの明示的なタスクにでも切り替えて実行させることができます。（詳しくは、セクション 2.7 を参照してください。）

parallel リージョンの終了時には、暗黙のバリアがあります。parallel リージョンの終了後、チームのマスタースレッドだけが、parallel リージョンを囲む task リージョンの実行を再開します。

parallel リージョンを実行しているチーム内のスレッドが、別の parallel 指示文に遭遇した場合、セクション 2.4.1 の規則に従って新しいチームを作成し、新しいチームのマスタースレッドになります。

parallel リージョン内で、あるスレッドの実行が終了した場合、すべてのチーム内のすべてのスレッドの実行が終了します。それらのスレッドの終了順序は不定となります。プログラム中でチームが通過したバリアの前そのにチームが行ったすべての処理が完了していることが保証されます。それぞれのスレッドが、最後のバリアを通過した後かつ終了する前に行う処理の量は不定です。

parallel 構文の例については、セクション A.5 を参照してください。
num_threads 指示節の例については、セクション A.6 を参照してください。

制限事項

parallel 構文の制限について、以下に示します。

- parallel リージョンの中へまたは parallel リージョンから外へ分岐するプログラムは、規格に準拠していません。
- プログラムは、parallel 指示文で指定した指示節の評価の順序、または、指示節の評価の副作用

に依存してはいけません。

- 指示文では、１つだけ if 指示節を指定することができます。
- 指示文では、1 つだけ num_threads 指示節を指定することができます。num_threads の式は、正の整数値として評価されなければなりません。

C/C++

- parallel リージョン内で実行した throw は、同じ parallel リージョン内で実行を再開しなければなりません。また、例外を投げた同じスレッドが捕捉しなければなりません。

C/C++

Fortran

- 同じ装置番号に対して、複数のスレッドによって Fortran 入出力文を非同期に使用した場合は、振舞いは不定となります。

Fortran

相互参照

- default、shared、private、firstprivate、および reduction 指示節については、セクション 2.9.3 を参照してください。
- copyin 指示節については、セクション 2.9.4 を参照してください。
- omp_get_thread_num ルーチンについては、セクション 3.2.4 を参照してください。

２．４．１．parallel リージョンのスレッド数の決定

実行中に parallel 指示文に遭遇した場合、（もしあれば）指示文の if 指示節または num_threads 指示節の値、現在の並列コンテキスト、*nthreads-var*、*dyn-var*、*thread-limit-var*、*max-active-levels-var*、および *nest-var* の内部制御変数の値が、リージョン内で使用するスレッドの数を決定するために使用されます。

parallel 構文の if 指示節または num_threads 指示節の式においてある変数を使用すると、その parallel 構文を囲むすべての構文中に現れるその変数が暗黙的に参照されることに注意してください。if 指示節の式と num_threads 指示節の式は、parallel 構文の外側のコンテキストによって評価されます。そして、それらの評価の順序は、規定されていません。また、num_threads または if 指示節の式の評価の順序や回数、どんな副作用も規定されていません。

スレッドが parallel 構文に遭遇したとき、スレッドの数は、アルゴリズム 2.1 に従って決定されます。

アルゴリズム　２．１

ThreadsBusy ＝現在実行している OpenMP スレッドの数
ActiveParRegions ＝　囲んでいる活動状態の parallel リージョンの数
if　if 指示節が指定されているか？
then　*IfClauseValue*　＝　if 指示節の式の値
else　*IfClauseValue*　＝　*true*
if　num_threads 指示節が指定されているか？
then　*ThreadsRequested*　＝　num_threads 指示節の式の値
else　*ThreadsRequested*　＝　*nthreads-var*
ThreadsAvailable ＝（*thread-limit-var* － *ThreadsBusy* + 1）
if　（*IfClauseValue*　＝　*false*）
then　number of threads　＝　1
else if　（*ActiveParRegions*　＞＝１）　and　（*nest-var*　＝　*false*）
then　number of threads　＝　1
else if　（*ActiveParRegions* ＝ *max-active-levels-var*）
then　number of threads ＝ 1
else if　（*dyn-var* ＝ *true*）　**and**　（*ThreadsRequested* <＝ *ThreadsAvailable*）
then　number of threads ＝［１：ThreadsRequested ］
else if　（*dyn-var* ＝ *true*）　**and**　（*ThreadsRequested* ＞ *ThreadsAvailable*）
then　number of threads ＝［１：ThreadsAvailable ］
else if　（*dyn-var* ＝ *false*）　**and**　（*ThreadsRequested* <＝ *ThreadsAvailable*）
then　number of threads ＝ *ThreadsRequested*
else if　（*dyn-var* ＝ *false*）　**and**　（*ThreadsRequested* ＞ *ThreadsAvailable*）
then　振舞いは実装依存。

注： *dyn-var* 内部制御変数の初期値は実装依存ですので、正しい実行をするために特定のスレッド数に依存するプログラムでは、スレッド数の動的調整を明示的に無効化すべきです。

相互参照

nthreads-var、*dyn-var*、*thread-limit-var*、*max-active-levels-var*、および *nest-var* 内部制御変数については、セクション 2.3 を参照してください。

2．5．ワークシェアリング構文

ワークシェアリング構文は、その構文と関連付けられたリージョンの実行を、構文に遭遇したチームのメンバーに分配します。それぞれのスレッドは、自分が実行している暗黙のタスクのコンテキストにおいてリージョンの一部を実行します。もし、チームがただ一つのスレッドから成る場合は、ワークシェアリングリージョンは、並列には実行されません。

ワークシェアリングリージョンの入口にはバリアはありません。しかし、nowait 指示節を指定しなければ、ワークシェアリングリージョンの終わりには暗黙のバリアが存在します。もし、nowait 指示節がある場合、実装は、ワークシェアリングリージョンの終わりのバリアを省略することができます。この場合、早く実行を終了したスレッドは、チーム内の他のメンバーがワークシェアリングリージョンの実行を完了するのを待たずに、かつ、フラッシュ操作を行わずに、ワークシェアリングリージョンの次の命令を続けて実行することができます。（セクション A.9 の例を参照してください。）

OpenMP は、以下のワークシェアリング構文を定義しています。これらについては、後のセクションで説明をします。

- ループ構文
- sections 構文
- single 構文
- workshare 構文

制限事項

以下の制限が、ワークシェアリング構文に適用されます。

- それぞれのワークシェアリングリージョンは、チーム内のすべてのスレッドで遭遇するか、また

は、どのスレッドによっても遭遇しないかのどちらかでなければなりません。

- 遭遇したワークシェアリングリージョンと barrier リージョンの並び順は、チーム内のすべてのスレッドで同じでなければなりません。

２．５．１．ループ構文

概要

ループ構文は、それと関連付けられた 1 つ以上のループの繰り返しが、チーム内のスレッドによって、各々の暗黙のタスクのコンテキストの中で、並列に実行されることを指定します。繰り返しは、ループリージョンが結合している parallel リージョンを実行しているチーム内に既に存在しているスレッドに分配されます。

文法

C/C++

ループ構文の文法は、以下の通りです。

```
#pragma omp for [clause[[, ] clause] ...]  new-line
    for-loops
```

ここで、指示節（*clause*）は、以下のいずれかです。

private（*list*）
firstprivate（*list*）
lastprivate（*list*）
reduction（*operator*： *list*）
schedule（*kind[, chunk_size]*）
collapse（*n*）
ordered
nowait

for 指示文は、関連付けられたすべての *for-loops* の構造に対する制限があります。具体的には、関連付けられたすべての *for-loops* は、以下の標準形でなければなりません。

---

for (*init-expr; test-expr; incr-expr*) *structured-block*

---

init-expr 以下のいずれかです。

var = lb

integer-type var = lb

random-access-iterator-type var = lb

pointer-type var = lb

test-expr 以下のいずれかです。

var relational-op b

b relational-op var

incr-expr 以下のいずれかです。

++var

var++

--var

var--

var += incr

var -= incr

var = var + incr

var = incr + var

var = var – incr

var 以下のいずれかです。

・符号付き、または符号なしの整数型の変数

・C++の場合、ランダムアクセスイテレータ型の変数

・C の場合、ポインタ型の変数

この変数が、何らかの形で共有されていない場合は、ループ構文内では暗黙的にプライベート化されます。

この変数は、*for-loop* の実行中に *incr-expr* 以外で変更してはいけません。

この変数がループ構文で lastprivate と指定されない限り、ループ後の値は、不定となります。

relational-op 以下のいずれかです。

<

<=

>

>=

lb と *b*　　以下のいずれかです。
　　var の型と一致した型のループ不変式
incr　　ループ不変の整数式

この標準形では、最も外側のループの実行前に、すべての関連付けられたループの繰り返し数を計算することができます。この計算は、それぞれのループに対して整数型で行われます。この型は、次のように var の型から決まります。

- もし、*var* が整数型の場合、型は *var* の型です。
- C++において、*var* がランダムアクセスイテレータ型の場合、型は、*var* の型の変数に適用される *std::distance* が使用する型になります。
- C において、*var* がポインタ型の場合、型は *ptrdiff_t* となります。

繰り返し数の計算に必要な中間結果が、上で決まる型で表現できない場合、振舞いは不定となります。

lb、*b*、または *incr* の式の評価中に暗黙的に同期をとることはありません。*lb*、*b*、または *incr* の式の順序や回数、副作用が起こるかどうかは規定されていません。

注：　ランダムアクセスイテレータは、定数時間内で要素へのランダムアクセスをサポートするために必要です。その他のイテレータが制限により除外されているのは、それらが線形時間を要するか、機能的な制限を持つからです。。したがって、そのような場合を並列化するには、タスクを使用するのが適切です。

C/C++

Fortran

ループ構文の文法は、以下の通りです。

```
!$omp do  [clause[[, ] clause] ...]
 do-loops
[!$omp end do  [nowait] ]
```

ここで、指示節（*clause*）は、以下のいずれかです。

private（*list*）
firstprivate（*list*）
lastprivate（*list*）
reduction（ {*operator* | *intrinsic_procedure_name*}： *list*）
schedule（*kind[, chunk_size]*）

```
collapse (n)
ordered
```

end do 指示文が指定されない場合は、end do 指示文が、*do-loops* の終わりにあると見なされます。

関連付けられたすべての *do-loops* は、Fortran 標準によって定義されている do 構文でなければなりません。end do 指示文が、いくつかのループ文が一つの DO 端末文を共有している do 構文の後に続く場合、その指示文は、これらの DO 文の最外側にだけ指定することができます。セクション A.7 の例を参照してください。

ループの繰り返し変数が何らかの形で共有されてない場合、それらは、ループ構文では、暗黙的にプライベート化されます。セクション A.8 の例を参照してください。ループの繰り返し変数がループ構文で lastprivate が指定されていない限り、ループ後の値は不定となります。

Fortran

結合

ループリージョンに対する結合スレッドセットは、現在のチームです。ループリージョンは、最も内側を囲んでいる parallel リージョンに結合しています。結合している parallel リージョンを実行しているチームのスレッドだけが、ループ繰り返しの実行とループリージョンの暗黙のバリア（省略可能）を行います。

説明

ループ構文は、指示文に続く１重以上のループから構成されるループのネストに関連付けられます。nowait 指示節が指定されない場合、ループ構文の終わりには暗黙のバリアがあります。

collapse 指示節は、ループ構文に何重のループに関連付けられるかを指定するために使われます。collapse 指示節のパラメタは、正の整定数式でなければなりません。もし、collapse 指示節が指定されない場合は、ループ構文に関連付けられたループは、構文の直後に続くループだけになります。

２重以上のループがループ構文に関連付けられている場合、それらすべての関連付けられたループの繰り返しは、１つの大きな繰り返し空間に一重化され、次に schedule 指示節に従って分割されます。すべての関連付けられたループにおける繰り返しの逐次実行によって、一重化された繰り返し空間内の繰り返しの順番が決まります。

それぞれの関連付けられたループに対する繰り返し数は、最も外側のループに入る前に計算されます。もし、いずれかの関連付けられたループの実行において、繰り返し数の計算に使われる変数の値が変更された場合は、振舞いは不定となります。

一重化されたループに対する繰り返し数を計算するために使用される整数型（または、Fortran では種別）は、実装依存です。

ワークシェアリングのループは、0、1、・・・N-1（N はループの繰り返しの番号）に番号付けられた論理的な繰り返しを持ちます。そして、この論理的な番号付けは、関連付けられたループが単一のスレッドによって実行されるとしたときに繰り返しが実行される順序を示します。schedule 指示節は、関連付けられたループの繰り返しを、連続した空でない部分集合（チャンクと呼びます）にどのように分割するか、そして、これらのチャンクをチーム内のスレッドにどのように分配するかを指定します。それぞれのスレッドは、自分の暗黙のタスクのコンテキストの中で、割り当てられたチャンクを実行します。*chunk_size* の式は、ループ構文の中で、プライベート化された変数のオリジナルのリストアイテムを使って評価されます。この式の評価の順序や回数、副作用の発生については規定されていません。ループ構文の schedule 指示節の式で、ある変数を使用すると、そのループ構文を囲むすべての構文内のその変数が参照されたことになります。

同じスケジュールと同じ繰り返し回数の異なったループリージョンは、それらが同じ parallel リージョンにあったとしても、異なったやり方でスレッドに繰り返しを分配されることがあります。唯一の例外は、表２－１で規定している static スケジュールです。スレッドがどのような状況においても特定の繰り返しを実行することに依存するプログラムは、規格に準拠していません。

ワークシェアリングループのスケジュールがどのように決定されるかの詳細については、セクション 2.5.1.1 を参照してください。

スケジュールの種別（*kind*） は、表２－１で規定しているうちの一つです。

表　２－１　schedule 指示節の *kind* の値

static	schedule（static, *chunk_size*）が指定された場合、繰り返しは、*chunk_size* で指定した大きさのチャンクに分割されます。そして、そのチャンクは、round-robin 方式でスレッド番号の順に、チーム内のスレッドに割り当てられます。 *chunk_size* が指定されない場合は、繰り返し空間は、ほぼ等しいサイズのチャンクに分割され、それぞれのスレッドに、多くとも一つのチャンクが分配されます。この場合、チャンクサイズは、不定となります。 以下の条件を満たす場合、static スケジュールに対する準拠した実装は、2 つのループリージョンにおいて、同じやり方で論理繰り返し番号がスレッドに割り当

	てられることを保証しなければなりません。 １）　両方のループリージョンでは、ループ繰り返し数が同じ。 ２）　両方のループリージョンでは、指定された *chunk_size* の値が同じ、または、 両方のループリージョンには、*chunk_size* の指定がない。 ３）　両方のループリージョンが、同じ parallel リージョンに結合している。 ２つのそのようなループの同じ論理的な繰り返しの間のデータ依存関係が、確実に保証されますので、nowait 指示節を安全に使用できます。（例として、セクション A.9 を参照してください。）
dynamic	schedule（dynamic、*chunk_size*）が指定された場合、スレッドから要求されたときに、繰り返しのチャンクが、チーム内のスレッドに分配されます。それぞれのスレッドは、繰り返しのチャンクを実行し、そして、分配すべきチャンクが無くなるまで、別のチャンクを要求します。 最後に分配されるチャンク（少ない繰り返しになるかもしれない）を除いて、それぞれのチャンクは、*chunk_size* の繰り返しを含んでいます。 *chunk_size* を指定しない場合は、１と見なされます。
guided	schedule（guided、*chunk_size*）が指定された場合、スレッドから要求されたときに、繰り返しのチャンクが、チーム内のスレッドに割り当てられます。それぞれのスレッドは、繰り返しのチャンクを実行し、そして、割り当てるべきチャンクがなくなるまで、別のチャンクを要求します。 *chunk_size* が１の場合、それぞれのチャンクサイズは、未割り当ての繰り返し数をチーム内のスレッドの数で割ったものに比例し、最終的には１まで減少します。 *chunk_size* が、k（１より大きい）の場合、それぞれのチャンクサイズは、k より小さくならないという制限の下で上記と同じ方法で決定されます。（ただし、割り当てられる最後のチャンクに含まれる繰り返しの数は、k より小さくなるかもしれません） *chunk_size* を指定しない場合は、１と見なされます。
auto	schedule（auto）が指定された場合、スケジューリングについての決定は、コンパイラ、および/または、実行時システムに委ねられます。プログラマは、実装に対して、繰り返しをチーム内のスレッドにマッピングする、どのような方法をも選択できる自由を与えます。
runtime	schedule（runtime）が指定された場合、スケジュールについての決定は、実行時まで延期され、スケジュールとチャンクサイズは、*run-sched-var* 内部制御変数から取得されます。もし、その内部制御変数に auto が設定されている場合、そのスケジュールは実装依存となります。

注：*p* 個のスレッドのチームと繰り返し数 *n* のループに対して、[*n* / *p*] を整数 q とします。整数 *q* は、*n* = *p* * *q* - *r* （*0* <= *r* < *p*）を満たします。schedule（static）（*chunk_size* の指定なし）のある準拠した実装は、*chunk_size* に値 *q* が指定されたとして振舞います。別の準拠した実装は、最初の（*p*-*r*）個のスレッドに繰り返し数 *q* を割り当て、残りの r 個のスレッドに、繰り返し数（*q* - 1）を割り当てます。これは、なぜ準拠したプログラムが特定の実装に頼ってはいけないかを示しています。

chunk_size の値が k の schedule（guided）の準拠した実装は、最初に利用可能なスレッドに *q* = *n* / *p* の繰り返し数を割り当て、そして、*n* - *q* と *p* * *k* の大きい方を n に割り当てます。次に、この処理を、*q* が残りの繰り返しの数以上になるまで、繰り返します。そのとき、残りの繰り返しは最後のチャンクになります。別の準拠した実装は、*q* = *n* / （*2p*）とすることを除いて、同じ方法を利用することができ、そして、*n* に *n* - *q* と *2* * *p* * *k* の大きい方を設定します。

制限事項

ループ構文の制限について、以下に示します。

- ループ構文に関連付けられたすべてのループは、完全にネストされていなければなりません。すなわち、2 つのループの間に飛び込みのコードや OpenMP の指示文があってはいけません。
- ループ指示文に関連付けられたループのループ制御式の値は、チーム内のすべてのスレッドで同じでなければいけません。
- schedule 指示節は、ループ指示文でただ 1 つだけ指定ができます。
- collapse 指示節は、ループ指示文でただ 1 つだけ指定ができます。
- *chunk_size* は、正の値を持つループ不変の整数式でなければなりません。
- *chunk_size* 式の値は、チーム内のすべてのスレッドで同じでなければなりません。
- *run-sched-var* 内部制御変数の値は、チーム内のすべてのスレッドで同じでなければなりません。
- schedule（runtime）または schedule（auto）が指定されたとき、*chunk_size* は、指定してはいけません。
- ordered 指示節は、ループ指示文でただ 1 つだけ指定ができます。
- もし、ordered リージョンが、ループ構文から生じたループリージョンに結合されている場合、ordered 指示節が、ループ指示文に指定されていなければなりません。
- ループの繰り返しの変数は、threadprivate 指示文に現れてはいけません。

C/C++

- 関連付けられた *for-loops* は、構造化ブロックでなければなりません。
- 最も内側の関連付けられたループの繰り返しだけは、continue 文によって省略することができます。
- どんな文も、関連付けられた for 文に分岐してはいけません。
- nowait 指示節は、for 指示文でただ一つだけ指定ができます。
- *relational-op* が、< または <= の場合、*incr-expr* は、ループのそれぞれの繰り返しで *var* を増加させなければなりません。また、*relational-op* が、> または >= の場合、*incr-expr* は、ループのそれぞれの繰り返しで *var* を減少させなければなりません。
- ループリージョンの内側で実行された throw は、ループリージョンの同じ繰り返しの中で実行を再開しなければなりません。そして、例外を投げたスレッドと同じスレッドがそれを捕捉しなければなりません。

C/C++

Fortran

- 関連付けられた *do-loops* は、構造化ブロックでなければなりません。
- 最も内側の関連付けられたループの繰り返しだけが、CYCLE 文によって省略することができます。
- DO 以外の関連付けられたループ中の文は、ループの外へ分岐することができません。
- *do-loop* の繰り返しの変数は、整数型でなければなりません。
- *do-loop* は、DO WHILE やループ制御の無い DO ループであってはいけません。

Fortran

相互参照

- private、firstprivate、lastprivate、および reduction 指示節については、セクション 2.9.3 を参照してください。
- OMP_SCHEDULE 環境変数については、セクション 4.1 を参照してください。
- ordered 構文については、セクション 2.8.7 を参照してください。

２．５．１．１．ワークシェアリングループのスケジュールの決定

実行が、ループ指示文に遭遇したとき、指示文の schedule 指示節（もし指定があれば）および *run-sched-var* と *def-sched-var* 内部制御変数は、どのようにループの繰り返しをスレッドに割り当てるかを決定するために使用されます。どのように内部制御変数の値が決定されるかは、セクション 2.3 を参照してください。もし、ループ指示文に schedule 指示節が指定されなかった場合、現在の *def-sched-var* 内部制御変数の値が、スケジュールを決定します。もし、ループ指示文に schedule 指示節が指定され、schedule 指示節の *kind* に runtime が指定された場合、現在の *run-sched-var* 内部制御変数の値が、スケジュールを決定します。他は、schedule 指示節の値が、スケジュールを決定します。図２－１は、ワークシェアリングループのスケジュールが、どのように決定されるかを示しています。

相互参照

・　内部制御変数については、セクション 2.3 を参照してください。

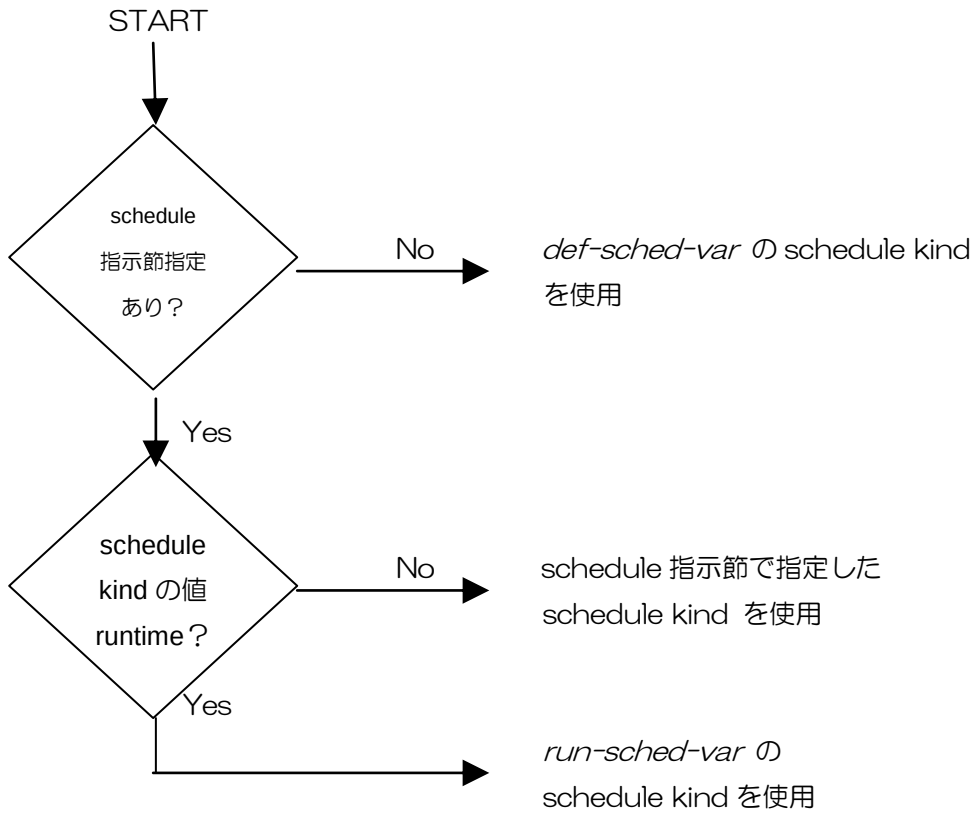


図２－１　ワークシェアリングループのためのスケジュールの決定

２．５．２．sections 構文

概要

sections 構文は、チーム内のスレッドに分配され、スレッドによって実行される構造化ブロックのセットを含む繰り返しのないワークシェアリング構文です。それぞれの構造化ブロックは、暗黙のタスクのコンテキストで、チーム内のスレッドの一つによって１回だけ実行されます。

文法

C/C++

sections 構文の文法は、以下の通りです。

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

指示節（*clause*）は、以下のいずれかです。

private（*list*）
firstprivate（*list*）
lastprivate（*list*）
reduction（*operator*： *list*）
nowait

C/C++

Fortran

sections 構文の文法は、以下の通りです。

```
!$omp sections  [clause[[, ] clause] ...]
   [ !$omp section]
      structured-block
   [ !$omp section
      structured-block ]
   . . .
!$omp end sections  [nowait]
```

指示節（*clause*）は、以下のいずれかです。

private（*list*）
firstprivate（*list*）
lastprivate（*list*）
reduction（{*operator* | *intrinsic_procedure_name*} ： *list*）

Fortran

結合

sections リージョンに対する結合スレッドセットは、現在のチームです。sections リージョンは、最も内側を囲んでいる parallel リージョンに結合しています。結合している parallel リージョンを実行しているチームのスレッドだけが、構造化ブロックの実行と（オプション）sections リージョンの暗黙のバリアに参加できます。

説明

sections 構文内の構造化ブロックは、それぞれ section 指示文で始まります。ただし、最初のブロックの section 指示文は省略可能です。
チーム内のスレッド間への構造化ブロックのスケジューリングの方法は、実装依存です。
nowait 指示節が指定されない限り、sections 構文の終わりに、暗黙のバリアがあります。

制限事項

sections 構文の制限について、以下に示します。

- 親なし section 指示文は、禁止されています。すなわち、section 指示文は、sections 構文内になければならず、sections リージョン以外で遭遇してはいけません。

- sections 構文内で囲まれたコードは、構造化ブロックでなければなりません。
- nowait 指示節は、sections 指示文でただ一つだけ指定できます。

C/C++

- sections リージョンの中で実行される throw は、sections リージョンの同じセクション内で実行を再開しなければなりません。そして、例外を投げたスレッドと同じスレッドで捕捉しなければなりません。

C/C++

相互参照

- private、firstprivate、lastprivate、および reduction 指示節については、セクション 2.9.3 を参照してください。

２．５．３．single 構文

概要

single 構文は、チーム内の一つのスレッドだけ（マスタースレッドである必要はありません）によって、関連付けられた構造化ブロックが暗黙のタスクのコンテキストの中で実行されることを指定します。そのブロックを実行しないチーム内の他のスレッドは、nowait 指示節が指定されない限り、single 構文の終わりにある暗黙のバリアでウェイトします。

文法

C/C++

single 構文の文法は、以下の通りです。

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

指示節（*clause*）は、以下のいずれかです。

private（*list*）
firstprivate（*list*）
copyprivate（*list*）
nowait

C/C++

Fortran

single 構文の文法は、以下の通りです。

```
!$omp single [clause[[, ] clause] ...]
   structured-block
!$omp end single [end_clause[[, ] end_clause] ...]
```

指示節（*clause*）は、以下のいずれかです。

private（*list*）
firstprivate（*list*）

終了指示節（*end_clause*）は、以下のいずれかです。

```
copyprivate（list）
nowait
```

Fortran

結合

single リージョンに対する結合スレッドセットは、現在のチームです。single リージョンは、最も内側を囲んでいる parallel リージョンに結合しています。結合している parallel リージョンを実行しているチームのスレッドだけが、構造化ブロックの実行と（オプション）single リージョンの暗黙のバリアに参加します。

説明

構造化ブロックを実行するスレッドの選択方法は、実装依存です。nowait 指示節が指定されない限り、single 構文の終わりに暗黙のバリアがあります。
single 構文の例については、セクション A.12 を参照してください。

制限事項

single 構文の制限について、以下に示します。

- copyprivate 指示節は、nowait 指示節と一緒に使用することはできません。
- nowait 指示節は、１つだけ single 構文に指定することができます。

C/C++

- single リージョンの中で実行される throw は、同じ single リージョン内で実行を再開しなければなりません。そして、例外を投げたスレッドと同じスレッドで捕捉しなければなりません。

C/C++

相互参照

- private および firstprivate 指示節については、セクション 2.9.3 を参照してください。
- copyprivate 指示節については、セクション 2.9.4.2 を参照してください。

２．５．４．workshare 構文

概要

workshare 構文は、囲まれている構造化ブロックの実行を処理ユニットに分割します。そして、それぞれのユニットが暗黙のタスクのコンテキストの中で、１つのスレッドによって一度だけ実行されるように、チームのスレッドが処理の分配を行います。

文法

workshare 構文の文法は、以下の通りです。

```
!$omp workshare
    structured-block
!$omp end workshare [nowait]
```

構造化ブロックは、以下だけから成ります。

- 配列の代入
- スカラの代入
- FORALL 文
- FORALL 構文
- WHERE 文
- WHERE 構文
- atomic 構文
- critical 構文
- parallel 構文

critical 構文内の文は、critical 構文の制限の制約を受けます。parallel 構文内の文には、制限はありません。

結合

workshare リージョンに対する結合スレッドセットは、現在のチームです。workshare リージョンは、最も内側を囲んでいる parallel リージョンに結合しています。結合している parallel リージョン

を実行しているチームのスレッドだけが、処理のユニットの実行と（オプション）workshareリージョンの暗黙のバリアに参加しています。

説明

nowait指示節の指定がない限り、workshare構文の終わりには暗黙のバリアがあります。
workshare構文の実装は、標準のFortranの文法を維持するために同期を挿入しなければなりません。例えば、構造化ブロック内の一つの文の効果は、後続する文の実行前に現れなければなりません。また、代入の右辺の評価は、左辺への代入が行われるよりも前に完了していなければなりません。

workshare構文内の文は、以下のように、処理のユニットに分割されます。

- それぞれの文中の配列式（配列からスカラ値を計算する変形配列組込み関数を含みます。）
  - 配列式のそれぞれの要素の評価（ELEMENTAL関数への参照を含みます）が、１つの処理のユニットとなります。
  - 変形配列組込み関数の評価は、さらに、幾つかの処理のユニットに分割されてもかまいません。
- 配列代入文については、それぞれの要素の代入が、１つの処理のユニットになります。
- スカラ代入文については、その代入操作が、１つ処理のユニットになります。
- WHERE文または構文については、マスク式の評価とマスクされた代入が、それぞれ１つの処理のユニットになります。
- FORALL文または構文については、マスク式の評価、繰り返し空間の仕様で起こる式の評価、および、マスクされた代入が、それぞれ１つの処理のユニットになります。
- atomic構文については、スカラ変数の更新が、１つの処理のユニットになります。
- critical構文については、構文が、単一の処理のユニットになります。
- parallel構文については、workshare構文に関して１つの処理のユニットになります。parallel構文中の文は、新しいスレッドチームによって実行されます。
- 構造化ブロック内の文の一部が上記のルールが適用されない場合、その文の一部は、１つの処理のユニットになります。

変形配列組込み関数は、MATMUL、DOT_PRODUCT、SUM、PRODUCT、MAXVAL、MINVAL、COUNT、ANY、SPREAD、PACK、UNPACK、RESHAPE、TRANSPOSE、EOSHIFT、CSHIFT、MINLOC、およびMAXLOCです。

処理のユニットがどのように workshare リージョンを実行するスレッドへ割り当てられるかは、不定となります。
もし、ブロック内の配列式が、プライベート変数の値、結合状態または割付け状態を参照する場合、すべてのスレッドによって同じ値が計算されない限り、その配列式の値は、未定義となります。

もし、配列代入、スカラ代入、マスクされた配列代入、または FORALL 代入によって、ブロック内のプライベート変数へ代入をした場合、結果は不定となります。

workshare 指示文は、workshare 構文内だけの処理を分割しますが、それ以外の workshare リージョン内の処理を分割しません。

workshare 構文の例については、セクション A.14 を参照してください。

制限事項

以下の制限が、workshare 指示文にあります。

- 構文は、関数が ELEMENTAL でない限り、ユーザ定義の関数呼出しを含んではいけません。

Fortran

2．6．複合パラレル・ワークシェアリング構文

複合パラレル・ワークシェアリング構文は、parallel 構文のすぐ内側にネストされたワークシェアリング構文を指定するためのショートカットです。これらの指示文の文法は、1 つのワークシェアリング構文だけから成る parallel 構文を明示的に指定するのと同じです。

複合パラレル・ワークシェアリング構文は、parallel 構文とワークシェアリング構文の両方で許されている指示節を許しています。もし、指示節が parallel 構文とワークシェアリング構文のどちらに利用されるかによって異なった振舞いがあれば、プログラムの振舞いは不定となります。

以下のセクションで、複合パラレル・ワークシェアリング構文について説明します。

- パラレルループ構文
- parallel sections 構文
- parallel workshare 構文

２．６．１．パラレルループ構文

概要

パラレルループ構文は、１つのループ構文だけから成る parallel 構文を指定するためのショートカットです。

文法

C/C++

パラレルループ構文の文法は、以下の通りです。

```
#pragma omp parallel for  [clause[[, ] clause] ...]  new-line
    for-loop
```

指示節（*clause*）には、nowait 指示節以外の parallel または for 指示文で許されているどんな指示節も指定することができます。（それらの意味と制限は同じです。）

C/C++

Fortran

パラレルループ構文の文法は、以下の通りです。

```
!$omp parallel do  [clause[[, ] clause] ...]
    do-loop
[!$omp end parallel do]
```

指示節（*clause*）には、parallel または do 指示文で許されているどんな指示節も指定することができます。（それらの意味と制限は同じです。）

もし、end parallel do 指示文が指定されない場合、end parallel do 指示文は、*do-loop* の終わりにあると見なされます。end parallel do 指示文に、nowait 指示節を指定することはできません。

Fortran

説明

C/C++

意味は、明示的に parallel 指示文の直後に for 指示文を指定することと同じです。

C/C++

Fortran

意味は、明示的に parallel 指示文の直後に do 指示文を指定し、かつ、end do 指示文の直後に end parallel 指示文を指定することと同じです。

Fortran

制限事項

parallel 構文とループ構文に対する制限が適用されます。

相互参照

- parallel 構文については、セクション 2.4 を参照してください。
- ループ構文については、セクション 2.5.1 を参照してください。
- データ属性の指示節については、セクション 2.9.3 を参照してください。

２．６．２．parallel sections 構文

概要

parallel sections 構文は、１つの sections 構文だけから成る parallel 構文を指定するためのショートカットです。

文法

C/C++

parallel sections 構文の文法は、以下の通りです。

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

指示節（*clause*）には、nowait 指示節以外の parallel または sections 指示文で許されているどんな指示節も指定することができます。（それらの意味と制限は同じです。）

C/C++

Fortran

parallel sections 構文の文法は、以下の通りです。

```
!$omp parallel sections  [clause[[, ] clause] ...]
   [!$omp section]
       structured-block
   [!$omp section
       structured-block  ]
      . . .
   !$omp end parallel sections
```

指示節（*clause*）には、parallel または sections 指示文で許されているどんな指示節も指定することができます。（それらの意味と制限は同じです。）

最後の section は、end parallel sections 指示文で終わりになります。end parallel sections 指示文では、nowait 指示節を指定することはできません。

Fortran

説明

C/C++

意味は、明示的に parallel 指示文の直後に sections 指示文を指定することと同じです。

C/C++

Fortran

意味は、明示的に parallel 指示文の直後に sections 指示文を指定し、かつ、end sections 指示文の直後に end parallel 指示文をすることと同じです。

Fortran

制限事項

parallel 構文と sections 構文に対する制限が適用されます。

相互参照

- parallel 構文については、セクション 2.4 を参照してください。
- sections 構文については、セクション 2.5.2 を参照してください。
- データ属性の指示節については、セクション 2.9.3 を参照してください。

２．６．３．parallel workshare 構文

概要

parallel workshare 構文は、１つの workshare 構文だけから成る parallel 構文を指定するためのショートカットです。

文法

parallel workshare 構文の文法は、以下の通りです。

```
!$omp parallel workshare   [clause[[, ] clause] ...]
     structured-block
!$omp end parallel workshare
```

指示節（*clause*）には、nowait 指示節以外の parallel で許されているどんな指示節も指定することができます。（これらの意味と制限は同じです。）

end parallel workshare 指示文では、nowait 指示節を指定することはできません。

説明

意味は、明示的に parallel 指示文の直後に workshare 指示文を指定し、かつ、end workshare 指示文の直後に end parallel 指示文を指定することと同じです。

制限事項

parallel 構文と workshare 構文に対する制限が適用されます。

相互参照

- parallel 構文については、セクション 2.4 を参照してください。
- workshare 構文については、セクション 2.5.4 を参照してください。
- データ属性の指示節については、セクション 2.9.3 を参照してください。

２．７．task 構文

概要

task 構文は、明示的なタスクを定義します。

文法

C/C++

task 構文の文法は、以下の通りです。

```
#pragma omp task  [clause[[, ] clause] ...]  new-line
     structured-block
```

指示節（clause）は、以下のいずれかです。

```
if （scalar-expression）
untied
default（shared | none）
private（list）
firstprivate（list）
shared（list）
```

C/C++

Fortran

task 構文の文法は、以下の通りです。

```
!$omp task  [clause[[, ] clause] ...]
   structured-block
!$omp end task
```

指示節（clause）は、以下のいずれかです。

```
if （scalar-logical-expression）
untied
default（private | firstprivate | shared | none）
private（list）
firstprivate（list）
shared（list）
```

Fortran

結合

task リージョンに対する結合スレッドセットは、現在の parallel リージョンです。task リージョンは、最も内側を囲んでいる parallel リージョンに結合しています。

説明

スレッドが、task 構文に遭遇したとき、関連付けられた構造化ブロックのコードから、タスクが生成されます。タスクのデータ環境は、task 構文で指定したデータ共有属性の指示節と適用されるデフォルトに従って作成されます。

遭遇したスレッドは、そのタスクをすぐに実行するか、または、実行を延滞させることができます。後者の場合、チーム内のどんなスレッドにでも、タスクを割り当てられる可能性があります。タスクの完了は、タスク同期構文を使って保証することができます。また、task 構文は、外側のタスクの内側にネストしてもかまいません。しかし、内側のタスクの task リージョンは、外側のタスクの task リージョンの一部分ではありません。

if 指示節が task 構文で指定され、if 指示節の式が false と評価される場合、遭遇したスレッドは、現在の task リージョンをサスペンドし、すぐに生成されたタスクの実行を開始しなければなりません。そして、サスペンドされた task リージョンは、生成されたタスクが完了するまで再開されません。そのタスクは、データ環境、ロック所有権、および同期構文について、全く別の task リージョンとして振舞います。task 構文の if 指示節の式中の変数の使用は、囲んでいるすべての構文の内のその変数に対して暗黙の参照が発生することに注意してください。

task リージョンの中でタスクスケジューリングポイントに遭遇したスレッドは、一時的にその task リージョンをサスペンドするかもしれません。デフォルトでは、タスクはタイドであり、実行を開始したスレッドだけが、そのサスペンドされた task リージョンを再開することができます。もし、task 構文に untied 指示節が指定された場合、チーム内のどんなスレッドでも、サスペンド後にその task リージョンを再開することができます。

task 構文は、その生成しているタスクの task リージョンの中（明示的なタスクの生成直後）に、タスクスケジューリングポイントを含んでいます。それぞれの明示的な task リージョンは、その完了ポイントにタスクスケジューリングポイントを含んでいます。実装は、アンタイドな task リージョンの中のどこにでも、タスクスケジューリングポイントを追加してもかまいません。

注：記憶域が、明示的な task リージョンによって共有されているとき、その明示的な task リージョンの実行が完了する前に、記憶域の寿命の終わりに到達しないことを、適切な同期処理を加えることによって保証することは、プログラマの責任です。

制限事項

task 構文の制限を、以下に示します。

- task リージョンの中へまたは外へ分岐するプログラムは、規格に準拠していません。
- プログラムは、task 指示文の指示節の評価の順序、または、指示節の評価による副作用に依存してはいけません。
- if 指示節は、指示文に１つだけ指定できます。

C/C++

- task リージョンの内側で実行された throw は、同じ task リージョン内で実行を再開しなければなりません。そして、例外を投げたスレッドと同じスレッドで捕捉しなければなりません。

C/C++

Fortran

- 複数のタスクによる同じ装置に関する Fortran 入出力文の非同期の使用は、不定の振舞いとなります。

Fortran

２．７．１．タスクスケジューリング

スレッドがタスクスケジューリングポイントに到達したときはいつも、実装は、そのスレッドのタスクを切り替え、現在のチームに結合している異なったタスクの実行を開始または再開してもかまいません。タスクスケジューリングポイントは、以下の場所にあります。

- 明示的なタスクの生成直後のポイント
- task リージョンの最後の命令の後
- taskwait リージョン
- 暗黙または明示的な barrier リージョン

加えて、実装は、アンタイドタスク内においては、この仕様で特に禁止されていないどんな場所にでもタスクスケジューリングポイントを挿入してかまいません。

スレッドがタスクスケジューリングポイントに遭遇したとき、スレッドは、「タスクスケジューリングの制約（下に記述）」を条件として、以下の一つを行ってかまいません。

- 現在のチームに結合しているタイドタスクの実行の開始
- 現在のチームに結合しているサスペンドしているタイド task リージョンの再開
- 現在のチームに結合しているアンタイドタスクの実行の開始
- 現在のチームに結合しているサスペンドしているアンタイド task リージョンの再開

もし、上の２つ以上の選択が可能な場合は、どれが選択されるかは、規定されていません。

[タスクスケジューリングの制約]

1. *scalar-expression* が *false* と評価された if 指示節を含む構文の明示的なタスクは、タスクの生成後、すぐに実行されます。
2. 新しいタイドタスクの別のスケジュールは、現在のスレッドにタイドされ、かつ、barrier リージョン内でサスペンドさせられていない task リージョンのセットによって制約されます。もし、この task リージョンのセットが空ならば、どんな新しいタイドタスクでも、スケジュールできます。さもなければ、新しいタイドタスクが、このセット内のすべてのタスクの子孫である場合に限り、そのタスクは、スケジュールしてもかまいません。

タスクスケジューリングに関して、これ以外の仮定を期待するプログラムは、規格に準拠していません。

注：タスクスケジューリングポイントは、動的に task リージョンをパーツに分割します。それぞれのパーツは、開始から終了まで切れ目なく実行されます。同じ task リージョンの異なったパーツは、遭遇した順番に実行されます。タスクの同期構文がない場合、スレッドが異なったスケジュール可能なタスクのパーツを実行する順番は、不定となります。

正しいプログラムは、上のルールに合致した考えられるすべてのスケジューリングの順番で、正しく一貫して振舞わなければなりません。

例えば、スレッドプライベートな記憶域が、task リージョンの一つのパーツの中でアクセスされた場合（明示的にソースコードの中で、または、暗黙的にライブラリルーチンの呼出しの中で）、その記憶域の値は、もし、それを変更する別のスケジュール可能なタスクが存在するならば、同じ task リージョンの次のパーツの中に保存されていると仮定することはできません。例 A.13.7c、例 A.13.7f、例 A.13.8c、および例 A.13.8f を参照してください。

別な例として、もし、task リージョンの異なったパーツの中で、ロックの獲得、解放が起こった場合、実行しているスレッドがスケジュールする別のタスクのパーツの中で同じロックを獲得することを試みてはいけません。さもなければ、デッドロックする可能性があります。critical リージョンが、タスクの複数のパーツをスピンして、別のスケジュールできるタスクが、同じ名前を持つ critical リージョンを含むとき、似たような状態が起きます。例 A.13.9f、例 A.13.10c、および例 A.13.10f を参照してください。

if 節がある明示的なタスクの中でスレッドプライベート変数とロックまたは critical セクションの使用は、if 指示節の評価が false のとき、タスクが、[タスクスケジューリングの制約　２．]に関わらず、すぐに実行することを考慮に入れなければなりません。

２．８．マスター・同期構文

以下のセクションで、説明します。

- master 構文
- critical 構文
- barrier 構文
- taskwait 構文
- atomic 構文
- flush 構文
- ordered 構文

２．８．１．master 構文

概要

master 構文は、チームのマスタースレッドによって実行される構造化ブロックを指定します。

文法

C/C++

master 構文の文法は、以下の通りです。

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

C/C++

Fortran

master 構文の文法は、以下の通りです。

```
!$omp master
     structured-block
!$omp end master
```

Fortran

結合

master リージョンに対する結合スレッドセットは、現在のチームです。master リージョンは、最も内側を囲んでいる parallel リージョンに結合しています。結合している parallel リージョンを実行しているチームのマスタースレッドだけが、master リージョンの構造化ブロックの実行に参加します。

説明

チーム内の他のスレッドは、関連付けられた構造化ブロックを実行しません。master 構文への入口または master 構文からの出口のいずれにも、バリアを含みません。
master 構文の例として、セクション A.15 を参照してください。

制限事項

C/C++

- master リージョンの中で実行される throw は、同じ master リージョン内で実行を再開しなければなりません。そして、例外を投げたスレッドと同じスレッドで捕捉しなければなりません。

C/C++

２．８．２．critical 構文

概要

critical 構文は、一度に一つのスレッドだけが関連した構造化ブロックを実行するように制限します。

文法

C/C++

critical 構文の文法は、以下の通りです。

```
#pragma omp critical  [（name）]  new-line
    structured-block
```

C/C++

Fortran

critical 構文の文法は、以下の通りです。

```
!$omp critical  [（name）]
     structured-block
!$omp end critical    [（name）]
```

Fortran

結合

critical リージョンに対する結合スレッドセットは、すべてのスレッドです。リージョンの実行は、ある時点で 1 つのスレッドに制限されます。そのスレッドは、属しているチームに関係なく、プログラム内すべてのスレッドの中の１つです。

説明

オプションの *name*（名前）は、critical 構文を識別するために使用します。*name* 指定のないすべての critical 構文は、同じ未指定の名前（unspecified name）として見なされます。スレッドは、同じ名前の critical リージョンを実行しているスレッドが無くなるまで、critical リージョンの先頭で待ち

ます。critical 構文は、現在のチームのスレッドだけではなく、すべてのスレッドの中ですべての同じ名前の critical 構文について、排他アクセスを強制します。

C/C++

critical 構文を識別するために使う識別子は、外部リンクを持ち、ラベル、タグ、メンバー、および通常の識別子によって使われる名前空間とは別の名前空間の中にあります。

C/C++

Fortran

critical 構文の名前は、プログラムの大域要素です。もし、名前が、他の要素と衝突した場合、プログラムの振舞いは、不定となります。

Fortran

critical 構文の例は、セクション A.16 を参照してください。

制限事項

C/C++

- critical リージョンの中で実行される throw は、同じ critical リージョン内で実行を再開しなければなりません。そして、例外を投げたスレッドと同じスレッドで捕捉しなければなりません。

C/C++

Fortran

以下の制限が、critical 構文に適用されます。

- もし、*name* が、critical 指示文に指定された場合、end critical 指示文にも同じ *name* を指定しなければなりません。
- もし、critical 指示文に *name* が指定されない場合、end critical 指示文に *name* を指定することはできません。

Fortran

２．８．３．barrier 構文

概要

barrier 構文は、この構文が現れたポイントに明示的なバリアを指定します。

文法

C/C++

barrier 構文の文法は、以下の通りです。

```
#pragma omp barrier  new-line
```

barrier 構文の文法は C 言語の文を持たないため、プログラムの中での現れる場所に幾つかの制限があります。barrier 指示文は、ベース言語の文が許されている場所にだけ指定することができます。barrier 指示文は、if、while、do、switch、またはラベル文の次の文であってはいけません。正式な文法については、付録 C を参照してください。これらの制限の説明は、セクション A.23 を参照してください。

C/C++

Fortran

barrier 構文の文法は、以下の通りです。

```
!$omp barrier
```

Fortran

結合

barrier リージョンに対する結合スレッドセットは、現在のチームです。barrier リージョンは、最も内側を囲んでいる parallel リージョンに結合しています。例として、セクション A.18 を参照してください。

説明

結合している parallel リージョンを実行しているチームのすべてのスレッドが、barrier リージョンを実行しなければなりません。そして、全てのスレッドはバリアの後ろの文を実行する前に、結合している parallel リージョン内でこのポイントに至るまでに生成されたすべての明示的なタスクの実行を完了しなければなりません。
barrier リージョンは、現在の task リージョン中の暗黙のタスクスケジューリングポイントを含みます。

制限事項

以下の制限が、barrier 構文に適用されます。

- それぞれの barrier リージョンは、チーム内のすべてのスレッドで遭遇しないか、または、全く遭遇しないかのいずれかでなければなりません。
- 遭遇したワークシェアリングリージョンと barrier リージョンの順序はチーム内のあらゆるスレッドに対して同じでなければなりません。

2．8．4．taskwait 構文

概要

taskwait 構文は、現在のタスクを開始してから生成した子タスクの完了を待つことを指定します。

文法

C/C++

taskwait 構文の文法は、以下の通りです。

```
#pragma omp taskwait  new-line
```

taskwait 構文の文法は、C 言語の文を持たないため、プログラムの中での現れる位置についての幾つかの制限があります。taskwait 指示文は、ベース言語の文が許されている場所にだけ指定できます。taskwait 指示文は、if、while、do、switch、またはラベル文の次の文であってはいけません。正式な文法については、付録 C を参照してください。これらの制限の説明は、セクション A.23 を参照してください。

C/C++

Fortran

taskwait 構文の文法は、以下の通りです。

```
!$omp taskwait
```

Fortran

結合

taskwait リージョンは、現在の task リージョンに結合しています。taskwait リージョンに対する結合スレッドセットは、現在のチームです。

説明

taskwait リージョンは、現在の task リージョンの中の暗黙のタスクスケジューリングポイントを含みます。現在の task リージョンは、taskwait リージョンの前に生成されたすべての子タスクの実行が完了するまで、そのタスクスケジューリングポイントでサスペンドされます。

２．８．５．atomic 構文

概要

atomic 構文は、特定の記憶域が、複数のスレッドによって同時に書き込みされる可能性を排除し、アトミックに更新されることを保証します。

文法

C/C++

atomic 構文の文法は、以下の通りです。

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

expression-stmt は、以下の形式のいずれかの式文です。

```
x binop= expr
x++
++x
x--
--x
```

上記の式において、

- *x* は、スカラ型の左辺値式である。
- *expr* は、スカラ型の式で、*x* で表された変数を参照しません。
- *binop* は、+、*、-、/、&、^、|、<<、または>>のいずれかです。
- *binop*、*binop*=、++、および--は、多重定義演算子ではありません。

C/C++

Fortran

atomic 構文の文法は、以下の通りです。

```
!$omp atomic
     statement
```

statement は、以下の形式の１つです。

```
x = x operator expr
x = expr operator x
x = intrinsic_procedure_name (x, expr_list)
x = intrinsic_procedure_name (expr_list、x)
```

上記の文において、

- *x* は、組込み型のスカラ変数です。
- *expr* は、*x* を参照しないスカラ式です。
- *expr_list* は、コンマで区切られた空でないスカラ式のリストです。そのスカラ式は *x* を参照しません。*intrinsic_procedure_name* が IAND、IOR、または IEOR であるとき、厳密に 1 つの式が *expr_list* に現れなければなりません。
- *intrinsic_procedure_name* は、MAX、MIN、IAND、IOR、または IEOR のうちの１つでなければなりません。
- *operator* は、+、*、-、/、.AND.、.OR.、.EQV.、または.NEQV.のうちの１つでなければなりません。
- *expr* の中の演算子は、*operator* の優先順位と同じかそれより優先順位を高くしなければなりません。x *operator expr* は、*x operator*（*expr*）と数学的に等価でなければならず、また、*expr*

operator x は、*(expr)operator x* に数学的に等価でなければなりません。

- *intrinsic_procedure_name* は、組込み手続き名を参照しなければならず、他のプログラム要素であってはいけません。
- *operator* は、組込み演算子でなければならず、ユーザ定義の演算子であってはいけません。
- 代入は、組込みの代入でなければなりません。
-

Fortran

結合

atomic リージョンに対する結合スレッドセットは、すべてのスレッドです。atomic リージョンは、同じ記憶域 x を更新する他の atomic リージョンに対して、プログラム中のすべてのスレッド（属するチームに構わず）間の排他的なアクセスを強制します。

説明

変数 x のロードとストアだけが、アトミックになります。*expr* の評価は、アトミックではありません。タスクスケジューリングポイントは、変数 x のロードとストアの間にあってはいけません。競合条件を避けるために、その位置に対して並列に行われる可能性あるすべての更新は、atomic 指示文で保護しなければなりません。atomic リージョンは、同じ記憶域 x にアクセスする critical または ordered リージョンに関しては排他的なアクセスは実施しません。しかし、他の OpenMP の同期文では、望み通りの排他的なアクセスを保証することができます。例えば、x への複数のアトミックの更新の後はバリアで後のアクセスはアトミックなアクセスと競争しないように保証できます。
準拠した実装は、異なった記憶域を更新する atomic リージョンの間の排他的なアクセスを実施してもかまいません。これが起こる状況は、実装依存となります。
atomic 構文の例として、セクション A.19 を参照してください。

制限事項

C/C++

以下の制限が、atomic 構文に適用されます。

- プログラムを通じて、記憶域 x へのすべてのアトミックな参照は、互換性のある型であることが要求されます。例として、セクション A.20 を参照してください。

C/C++

Fortran

以下の制限が、atomic 構文に適用されます。

- プログラムを通じて、変数 x の記憶域へのすべてのアトミックな参照は、同じ型と同じ型パラメタを持つことが要求されます。例として、セクション A.20 を参照して下ください。

Fortran

相互参照

- critical 構文については、セクション 2.8.2 を参照してください。

2．8．6．flush 構文

概要

flush 構文は、OpenMP のフラッシュ操作を実行します。この操作は、スレッドのメモリの一時的なビューをメモリと一致させます。そして、明示的または暗黙的に指定された変数のメモリ操作の順序を強制します。詳細は、セクション 1.4 のメモリモデルの説明を参照してください。

文法

C/C++

flush 構文の文法は、以下の通りです。

```
#pragma omp flush  [(list)]   new-line
```

flush 構文の文法は C 言語の文を持たないため、プログラム中の現れる場所について幾つかの制限があります。flush 指示文は、ベース言語の文が許されている場所にだけ指定されます。そして、if、while、do、switch、またはラベルの次の文である場所で使用してはいけません。正式な文法については、付録 C を参照してください。これらの制限の説明は、セクション A.23 を参照してください。

C/C++

Fortran

flush 構文の文法は、以下の通りです。

```
!$omp flush  [(list)]
```

Fortran

結合

flush リージョンに対する結合スレッドセットは、遭遇したスレッドです。flush リージョンの実行は、そのリージョンを実行するスレッドのメモリと一時的なビューだけに作用します。他のスレッドの一時的なビューには影響を及ぼしません。他のスレッドは、遭遇したスレッドのフラッシュ操作の効果を監視することを保証するために、自分自身でフラッシュ操作を実行しなければなりません。

説明

リストの指定のある flush 構文は、リスト中のアイテムに対してフラッシュ操作を行います。そして、指定されたすべてのリストのアイテムに対するフラッシュ操作が完了するまで復帰しません。スレッドで実行されるリストのない flush 構文は、あたかもベース言語によって定義されているようなそのスレッドから見えるプログラムのデータの全体がフラッシュされるかのように操作します。

C/C++

リスト中にポインタが指定された場合、ポインタが参照するメモリブロックではなく、ポインタ自身がフラッシュされます。

C/C++

Fortran

リストアイテムまたはリストアイテムの部分実体が、POINTER 属性を持つ場合、POINTER 項目の割付けや結合状態がフラッシュされます。しかし、ポインタの指示先は、フラッシュされません。リストアイテムが Cray ポインタである場合、そのポインタがフラッシュされます。しかし、それが指す実体はフラッシュされません。リストアイテムが、ALLOCATABLE 属性を持ち、かつ、割付けられている場合、割付けられた配列がフラッシュされます。そうでなければ、割付け状態がフラッシュされます。

Fortran

flush 構文の例として、セクション A.21 とセクション A.22 を参照してください。

注：以下の例は、フラッシュ操作の順序付けを示しています。以下の誤った擬似コードの例において、プログラマは、２つのスレッドによって critical セクションの同時実行を防ぐことを期待しています。しかし、変数 a と b の操作は、適切な順序で実施していませんので、このプログラムは正しく動作しません。

誤った例

```
                    a = b = 0
```

スレッド１	スレッド２
`b = 1`	`a = 1`
`flush``(b)`	*`flush`*`(a)`
`flush``(a)`	*`flush`*`(b)`
`if (a == 0) then`	`if (b == 0) then`
`  `*`critical section`	`  `*`critical section`
`end if`	`end if`

この例の問題は、変数 a と b の操作が、互いに順序付けされていないということです。例えば、コンパイラがスレッド１での b のフラッシュ、または、スレッド２での a のフラッシュを critical セクションの後に移動することは防ぎません。（スレッド１の critical セクションは、b を参照しない、スレッド２の critical セクションでは、a を参照しないと仮定します。）もし、どちらかの順序の移動が起きた場合、両方のスレッドは、critical セクションを同時に実行することができてしまいます。

以下の正しい擬似コードの例は、critical セクションが、どんなときでも、２つのスレッドのうち多くて１つによって実行されることを保証します。この例では、どちらのスレッドによっても critical セクションが実行されないことが正しいと考えられていることに注意してください。それぞれのスレッドが if 文を実行する前に両方のフラッシュが完了している場合、これが起こります。

正しい例

```
                    a = b = 0

スレッド１                          スレッド２
b = 1                               a = 1
flush(a,b)                          flush(a,b)
if (a == 0) then                    if (b == 0)  then
   critical section                    critical section
end if                              end if
```

コンパイラは、if 文が実行される前にそれぞれの代入の完了とデータがフラッシュされることを確実にするために、それぞれのスレッドに対するフラッシュを移動することが禁止されています。

リスト指定がない flush リージョンは、以下の場所で暗黙的に存在します。

- barrier リージョンの間中
- parallel、critical、および ordered リージョンの入口と出口
- nowait の指定がないならば、ワークシェアリングリージョンからの出口
- 複合パラレル・ワークシェアリングリージョンの入口と出口
- omp_set_lock および omp_unset_lock リージョンの間中
- omp_test_lock、omp_set_nest_lock、omp_unset_nest_lock、および omp_test_nest_lock リージョンの間中（もし、リージョンでロックが設定または解除される場合）
- すべてのタスクスケジューリングポイントの直前または直後

リスト指定のある flush リージョンは、以下の場所で暗黙的に存在します。

- atomic リージョンの入口と出口（ここで、リストは atomic 構文で更新される変数だけを含みます）。

注： flush リージョンは、以下の場所では、暗黙的には存在しません。

- ワークシェアリングリージョンの入口
- master リージョンの入口と出口

２．８．７．ordered 構文

概要

ordered 構文は、ループリージョン中の構造化ブロックが、ループ繰り返しの順序で実行されることを指定します。これは、ordered リージョン外のコードが並列に動作することを許す一方で、ordered リージョン内のコードを逐次化し、順序付けます。

文法

C/C++

ordered 構文の文法は、以下の通りです。

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

C/C++

Fortran

ordered 構文の文法は、以下の通りです。

```
!$omp ordered
     structured-block
!$omp end ordered
```

Fortran

結合

ordered リージョンに対する結合スレッドセットは、現在のチームです。ordered リージョンは、最も内側を囲んでいるループリージョンに結合しています。異なったループリージョンに結合している ordered リージョンは、互いに独立して実行します。

説明

ループリージョンを実行しているチーム内のスレッドは、ループの繰り返しの順序で順番に ordered リージョンを実行します。ループの最初の繰り返しを実行しているスレッドが ordered リージョンに

遭遇した場合、待たずに ordered リージョンに入ることができます。その後の繰り返しを実行しているスレッドが ordered リージョンに遭遇した場合、それ以前の全ての繰り返しにおける ordered リージョンの実行が完了するまで、遭遇した ordered リージョンの入口で待ちます。

制限事項

ordered 構文の制限は、以下の通りです。

- ordered リージョンが結合しているループリージョンと対応するループ(またはパラレルループ)構文では ordered 指示節を指定しなければなりません。
- ループリージョン内のループまたはネストループの１つの繰り返しの実行中、スレッドは、同じループリージョンに結合している２つ以上の ordered リージョンを実行してはいけません。

ordered 構文の例として、セクション A.24 を参照してください。

C/C++

- ordered リージョンの中で実行される throw は、同じ ordered リージョン内で実行を再開させなければなりません。そして、例外を投げた同じスレッドによって捕捉されなければなりません。

C/C++

相互参照

- ループ構文については、セクション 2.5.1 を参照してください。
- パラレルループ構文については、セクション 2.6.1 を参照してください。

２．９．データ環境

このセクションでは、parallel、task、およびワークシェアリングリージョンの実行中にデータ環境を制御するための指示文といくつかの指示節について説明します。

- セクション 2.9.1 では、parallel、task、およびワークシェアリングリージョン内で参照される変数のデータ共有属性がどのように決定されるかを説明します。
- スレッドプライベートメモリを生成するために提供される threadprivate 指示文は、セクション 2.9.2 で説明します。
- parallel、task、またはワークシェアリング構文内で参照される変数のデータ共有属性を制御するための指示節は、セクション 2.9.3 で説明されています。
- あるスレッド上のプライベートまたはスレッドプライベートな変数から、チーム内の他のスレッド上の対応する変数へ、データの値をコピーするための指示節は、セクション 2.9.4 で説明されています。

２．９．１．データ共有属性の規則

このセクションでは、parallel、task、およびワークシェアリングリージョン内で参照される変数のデータ共有属性がどのように決定されるかを説明します。以下の２つの場合を、分けて説明します。

- セクション 2.9.1.1 では、構文内で参照される変数のデータ共有属性の規則について説明します。
- セクション 2.9.1.2 では、リージョン内（ただし構文外）で参照される変数のデータ共有属性の規則について説明します。

２．９．１．１．構文内で参照する変数のデータ共有属性の規則

構文内で参照される変数のデータ共有属性は、次の１つとなります。

- 事前に決定されたデータ共有属性（*predetermined*）
- 明示的に決定された共有属性（*explicitly determined*）
- 暗黙的に決定されたデータ共有属性（*implicitly determined*）。

囲まれている構文の firstprivate、lastprivate、または reduction 指示節で指定されている変数は、囲んでいる構文内の対応する変数の暗黙的な参照を引き起こします。

以下の変数は、事前に決定されたデータ共有属性を持ちます：

C/C++

- threadprivate 指示文に現れる変数は、スレッドプライベートとなります。
- 構文内のスコープで宣言されている auto 変数は、プライベートとなります。
- ヒープ領域に割付けられる変数は、共有となります。
- 静的データメンバーは、共有となります。
- for 構文または parallel for 構文の *for-loop* 内のループ繰り返し変数は、その構文内ではプライベートとなります。
- mutable メンバーを持たない const 修飾された変数は、共有となります。
- 構文の内側のスコープ内で宣言された静的変数は、共有となります。

C/C++

Fortran

- threadprivate 指示文に現れる変数と共通ブロックは、スレッドプライベートとなります。
- do または parallel do 構文の *do-loop* 内のループ繰り返し変数は、その構文内ではプライベートとなります。
- parallel または task 構文内の逐次ループのループ繰り返し変数は、最も内側の構文内ではプライベートとなります。
- DO 形反復と forall の指標変数は、プライベートとなります。
- Cray pointee は、Cary ポインタが結合された記憶域のデータ共有属性を継承します。

Fortran

事前に決定されたデータ共有属性を持った変数は、以下にリストされている場合を除いて、データ共有属性の指示節に記述してはいけません。これらの例外だけに対して、データ共有属性の指示節に、事前に決定されたデータ共有属性を持つ変数を記述することが許されます。そして、その変数の事前に決定されたデータ共有属性をオーバーライドします。

C/C++

- for または parallel for 構文の *for-loop* 内のループ繰り返し変数は、private または lastprivate 指示節のリストに記述してもかまいません。

C/C++

Fortran

- do または parallel do 構文の *do-loop* 内のループ繰り返し変数は、private または lastprivate 指示節のリストに記述してもかまいません。
- parallel または task 構文内の逐次ループのループ繰り返し変数として使用されている変数は、他

の制限を満たせば、parallel または task 構文、そして囲まれる構文の private、firstprivate、lastprivate、shared、または reduction 指示節に記述してもかまいません。

- 大きさ引継ぎ配列は、shared 指示節に記述してもかまいません。

Fortran

個々の指示節に出現できる変数の追加制限を、セクション 2.9.3 のそれぞれの指示節で説明します。

明示的に決定されたデータ共有属性を持つ変数は、構文内で参照され、構文のデータ共有属性の指示節に記述されている変数です。

暗黙的に決定されたデータ共有属性を持つ変数は、構文内で参照され、事前に決定されたデータ共有属性を持たなく、そして構文のデータ共有属性の指示節に記述されていない変数です。

暗黙的に決定されたデータ共有属性を持つ変数の規則は、以下となります。

- parallel または task 構文において、もし default 指示節があれば、その変数のデータ共有属性は、default 指示節によって決定されます。（セクション 2.9.3.1 を参照してください。）
- parallel 構文において、default 指示節がない場合、その変数は、共有となります。
- task 構文以外の構文においては、もし、default 指示節がない場合、変数は、囲んでいるコンテキストからデータ共有属性を継承します。
- task 構文において、もし、default 指示節がない場合、最も内側の parallel 構文を含むすべての囲んでいる構文内で共有されることが事前に決定されている変数は、共有となります。
- task 構文において、もし、default 指示節がない場合、以上の規則によってデータ共有属性が決定されない変数は、ファーストプライベートとなります。
- データ共有属性を暗黙的に決定できない task 構文内の変数における追加の制限は、firstprivate 指示節の制限のセクション（セクション 2.9.3.4）で説明されています。

２．９．１．２．リージョン内（構文外）で参照する変数のデータ共有属性の規則

リージョン内（ただし構文外）で参照される変数のデータ共有属性は、以下のように決定されます。

C/C++

- リージョン内から呼び出されたルーチンで宣言された静的変数は、共有となります。
- 呼び出されたルーチンで宣言された mutable メンバーを持たない const 修飾された型の変数は、共有となります。
- リージョン内から呼び出されたルーチンで参照されるファイル・スコープまたはネームスペース・スコープの変数は、threadprivate 指示文に現れない限り、共有となります。
- ヒープ領域に割付けられた変数は、共有となります。
- 静的データメンバーは、threadprivate 指示文に現れない限り、共有となります。
- リージョン内から呼び出されたルーチンの参照によって渡される仮引数は、対応した実引数のデータ共有属性を継承します。
- リージョン内から呼び出されたルーチンで宣言された他の変数は、プライベートとなります。

C/C++

Fortran

- リージョン内から呼び出されたルーチンで宣言され、SAVE 属性を持つか、または、値が初期化されたローカル変数は、threadprivate 指示文に現れない限り、共有となります。
- 共通ブロックに属する、または、モジュール内で宣言されているリージョン内から呼び出されるルーチン内で参照される変数は、threadprivate 指示文に現れない限り、共有となります。
- 参照によって渡されるリージョンから呼び出されたルーチンの仮引数は、対応した実引数のデータ共有属性を継承します。
- Cray pointee は、Cary ポインタが結合された記憶域のデータ共有属性を継承します。
- DO 形反復の指標変数、forall の指標変数、およびリージョン内から呼び出されたルーチンで宣言された他のローカル変数は、プライベートとなります。

Fortran

２．９．２．threadprivate 指示文

概要

threadprivate 指示文は、変数が複製され、それぞれのスレッドがそのコピーを持つことを指定します。

文法

C/C++

threadprivate 指示文の文法は、以下の通りです。

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

list は、不完全型でないファイル・スコープ、名前空間・スコープ、または静的なブロック・スコープのコンマで区切られたリストです。

C/C++

Fortran

threadprivate 指示文の文法は、以下の通りです。

```
!$omp threadprivate（list）
```

list は、名前付き変数と名前付き共通ブロックのコンマで区切られたリストです。共通ブロック名は、スラッシュの間になければなりません。

Fortran

説明

スレッドプライベート変数のそれぞれのコピーは、プログラムによって指定された方法で１回初期化されます。初期化は、そのコピーへの最初の参照より前のプログラム内の明示されていない時点で行います。スレッドプライベート変数のすべてのコピーの記憶域の解放は、ベース言語で静的な変数がどう処理されるかに従って、プログラム内の明示されていない時点で行われます。
あるスレッドが他のスレッドのスレッドプライベート変数のコピーを参照するプログラムは、規格に準拠していません。

もし、実行スレッドが、変数を変更する別のスケジュールできるタスクにスイッチする場合、スレッドプライベート変数の内容は、タスクスケジューリングポイントを通じて変更する可能性があります。タスクスケジューリングの詳細は、セクション 1.3 とセクション 2.7 を参照してください。
parallel リージョンにおいて、マスタースレッドによる参照は、parallel リージョンに遭遇したスレッド内の変数のコピーへの参照となります。
逐次部分中での参照は、初期スレッドの変数のコピーとなります。スレッドプライベート変数の初期スレッドのコピーのデータの値は、プログラム内の変数への任意の２つの連続する参照間で保持されることを保証します。
以下の条件をすべて満たす場合、初期スレッド以外のスレッドのスレッドプライベート変数のデータの値は、２つの連続した活動状態の parallel リージョンの間で保持されることを保証します。

- parallel リージョンが、別の明示的な parallel リージョンの内側にネストされていない。
- 両方の parallel リージョンを実行するために使用するスレッド数が同じ。
- 囲んでいる task リージョン内の内部制御変数 *dyn-var* の値が、両方の parallel リージョンの入口で、false である。

もし、これらの条件が満たされ、かつ、スレッドプライベート変数が両方のリージョンで参照される場合、それぞれのリージョン内の同じスレッド番号のスレッドは、変数の同じコピーを参照します。

C/C++

もし、上の条件が満たされる場合、２番目のリージョンの copyin 指示節に現れないスレッドプライベート変数のコピーの記憶域期間、寿命および値は、保持されます。さもなければ、２番目のリージョンにおける変数のコピーの記憶域期間、寿命および値は、不定となります。
もし、スレッドプライベート変数の明示的な初期化子で参照される変数の値が、スレッドプライベート変数の任意のインスタンスへの最初の参照より前に変更された場合、振舞いは不定となります。

注： クラス型の異なったスレッドプライベート変数に対するコンストラクタの呼び出される順番は、不定となります。クラス型の異なったスレッドプライベート変数に対するデストラクタの呼び出される順番は、不定となります。

C/C++

」

Fortran

もし、変数が copyin 指示節に指定されている、または、copyin 指示節に指定されている共通ブロック中にある場合、その変数は、copyin 指示節によって影響されます。
もし、上の条件を満たす場合、スレッドプライベート変数またはスレッドプライベートの共通ブロック内の変数のスレッドのコピーの定義状態、結合状態、または割付け状態は、２番目のリージョンで指定した copyin 指示節によって影響されなければ、保持されます。さもなければ、２番目のリージョ

ン内で変数のスレッドのコピーの定義状態と結合状態は、未定義となります。また、割付け配列の割付け状態は、実装依存となります。
もし、共通ブロック、または、モジュールのスコープ内で宣言された変数が、threadprivate 指示文に現れる場合、それは暗黙的に SAVE 属性を持ちます。
スレッドプライベート変数またはスレッドプライベート共通ブロック内の変数が、参照される最初の parallel リージョンの copyin 指示節によって影響されなければ、その変数または変数の部分実体が、最初に定義されるのか、または、未定義になるかは、以下の規則に従います。

- ALLOCATABLE 属性を持っている場合、生成されたそれぞれのコピーは、現在割付けられていないという初期割付け状態となります。
- POINTER 属性を持っている場合、
  - 明示的な初期化またはデフォルトの初期化のいずれかによって、空状態という初期の結合状態である場合、生成されたそれぞれのコピーは、空状態の結合状態となります。
  - さもなければ、生成されたそれぞれのコピーは、未定義の結合状態となります。
- POINTER 属性および ALLOCATABLE 属性を持っていない場合、
  - 明示的な初期化またはデフォルトの初期化のいずれかによって最初に定義されている場合、生成されたそれぞれのコピーもまた、定義されます。
  - さもなければ、生成されたそれぞれのコピーは、未定義となります。

Fortran

threadprivate 指示文の例については、セクション A.25 を参照してください。

制限事項

threadprivate 指示文の制限は、以下の通りです。

- スレッドプライベート変数は、copyin、copyprivate、schedule、num_threads、および if 指示節以外の指示節に現れてはいけません。
- アンタイドタスクがスレッドプライベートの記憶域にアクセスするプログラムは、規格に準拠していません。

C/C++

- 別の変数の一部（配列や構造体の要素など）の変数は、C++のクラスの静的データメンバーでない限り、threadprivate 指示節に指定することはできません。
- ファイル・スコープの変数の threadprivate 指示文は、定義または宣言の外側になければなりません。そして、そのリスト中の変数へのすべての参照より字句的に先行していなければなりません。
- 静的クラスメンバーの変数に対する threadprivate 指示文は、そのメンバー変数が宣言されたのと同じスコープ内のクラス定義内になければなりません。そして、リストにある変数へのすべての参照より字句的に先行していなければなりません。
- 名前空間・スコープの変数の threadprivate 指示文は、それ自身の名前空間定義以外の定義や宣

言の外側になければなりません。そして、そのリスト中の変数へのすべての参照より字句的に先行していなければなりません。

- threadprivate 指示文のリスト中の、ファイル、名前空間、またはクラス・スコープのそれぞれの変数は、指示文より字句的に先行している、ファイル、名前空間、またはクラス・スコープの変数宣言を参照しなければなりません。
- 静的なブロック・スコープの変数の threadprivate 指示文は、変数のスコープ内、かつネストされていないスコープになければなりません。そして、そのリスト中の変数へのすべての参照より字句的に先行していなければなりません。
- threadprivate 指示文のリスト中の、ブロック・スコープ内のそれぞれの変数は、指示文より字句的に先行している同じスコープ内の変数宣言を参照しなければなりません。この変数宣言は、**静的**記憶域クラス指定子を使用しなければなりません。
- 変数が、１つのコンパイル単位の中で threadprivate 指示文に指定された場合、その変数が宣言されているすべてのコンパイル単位の中で、その変数を threadprivate 指示文に指定しなければなりません。
- スレッドプライベート変数のアドレスは、アドレス定数ではありません。
- スレッドプライベート変数は、不完全型や参照型であってはいけません。
- クラス型のスレッドプライベート変数は、以下を持たなければなりません。

  ー　初期化子が与えられないデフォルトの初期化の場合、アクセス可能で曖昧ではないデフォルトのコンストラクタ。

  ー　直接的な初期化の場合、与えられた引数を受け付けるアクセス可能で曖昧ではないコンストラクタ。

  ー　明示的な初期化子を伴ったコピー初期化の場合、アクセス可能で曖昧ではないコピーコンストラクタ。

C/C++

Fortran

- 別の変数の一部（配列や構造体の要素など）の変数は、threadprivate 指示節に指定することはできません。
- threadprivate 指示文は、共通ブロックまたは変数が宣言されている有効域の宣言部分に現れなければなりません。共通ブロック内の変数は、参照結合または親子結合によってアクセスされることができますが、共通ブロック名はできません。これは、threadprivate 指示文で指定された共通ブロック名は、threadprivate 指示文と同じ有効域内で共通ブロックであることを宣言しなければならないことを意味します。
- 共通ブロック名を指定した threadprivate 指示文が、１つのプログラム単位に現れた場合、そのような指示文は、同じ名前を指定した COMMON 文を含む他のすべてのプログラム単位にも、現れなければなりません。この指示文は、プログラム単位内で、最後のそのような COMMON 文の後に現れなければなりません。
- 無名共通ブロックは、threadprivate 指示文に現れてはいけません。

- 変数は、それが宣言されるスコープ内でのみ、threadprivate 指示文に現れることができます。その変数は、共通ブロックの要素であったり、または EQUIVALENCE 文に現れてはいけません。
- モジュールのスコープ内で宣言されていない変数が threadprivate 指示文に現れる場合は、SAVE 属性を持たなければなりません。

Fortran

相互参照

- *dyn-var* 内部制御変数については、セクション 2.3 を参照してください。
- parallel リージョンを実行するために使用されるスレッドの数については、セクション 2.4.1 を参照してください。
- copyin 指示節については、セクション 2.9.4.1 を参照してください。

2．9．3． データ共有属性に関する指示節

幾つかの構文は、構文中で参照される変数のデータ共有属性を制御することをユーザに許すための指示節を受け入れます。データ共有属性の指示節は、指示節が現れる構文内で識別できる変数名にだけに適用します。
このセクションでリストされた指示節が、すべての指示文に有効とは限りません。ある指示文で有効な指示節のセットは、その指示文の箇所で説明されます。
殆どの指示節は、コンマで区切られたリストアイテムの並びを受け入れます。（セクション 2.1 を参照してください。）指示節に現れているすべてのリストアイテムは、ベース言語のスコープの規則に従って、識別できなければなりません。default 指示節以外の指示節は、必要なだけ繰り返してもかまいません。変数を指定したリストアイテムは、同じ指示文で 2 つ以上の指示節に現れてはいけません。これの例外として、変数は firstprivate と lastprivate 指示節の両方に指定されることができます。

C/C++

データ共有属性の指示節で参照された変数が、テンプレートから派生した型を持ち、かつ、プログラム内でその変数の参照がどこにもない場合は、その変数に関する振舞いは、不定となります。

C/C++

Fortran

名前付き共通ブロックは、スラッシュで囲まれた名前によって、リスト中に指定することができます。名前付き共通ブロックが、リスト中に現れたとき、その共通ブロックのメンバーすべてが明示的にリスト中に指定されたと同じ意味となります。共通ブロックの明示的なメンバーは、共通ブロック名を

指定しているCOMMON文中の名前の変数であり、また、指示節が現れた同じ有効域で宣言されています。

- 共通ブロック内の変数は、参照結合または親子結合によってアクセスされることができますが、共通ブロック名はできません。これは、データ共有属性の指示節で指定された共通ブロック名は、データ共有属性の指示節が現れる同じ有効域内の共通ブロックであることを宣言されなければならないことを意味します。
- 名前付き共通ブロックが、指示文の private、firstprivate、lastprivate、または shared 指示節に現れたとき、その共通ブロックのメンバーは、その指示文の別のデータ共有属性の指示節で宣言してはいけません。（例として、セクション A.27 を参照してください。）共通ブロックの個々のメンバーが、指示文の private、firstprivate、lastprivate、または reduction 指示節で指定されたとき、指定された変数の記憶域は、共通ブロック自身の記憶域とはもはや結合されません。（例として、セクション A.32 を参照してください。）

Fortran

2．9．3．1．default 指示節

概要

default 指示節は、parallel または task リージョン内で参照される変数のデータ共有属性をユーザが制御をすることを許可します。その変数のデータ共有属性は、暗黙的に決定されます。（セクション 2.9.1.1 を参照してください。）

文法

C/C++

default 指示節の文法は、以下の通りです。

```
default（shared | none）
```

C/C++

Fortran

default 指示節の文法は、以下の通りです。

```
default（private | firstprivate | shared | none）
```

Fortran

説明

default（shared）指示節は、構文内で参照されるすべての変数が、暗黙的に決定されるデータ共有属性を”共有”とすることを指示します。

Fortran

default（firstprivate）指示節は、構文内で参照されるすべての変数が、暗黙的に決定されるデータ共有属性を firstprivate とすることを指示します。
default（private）指示節は、構文内で参照されるすべての変数が、暗黙的に決定されるデータ共有属性を private とすることを指示します。

Fortran

default（none）指示節は、構文内で参照されて、事前に決定されるデータ共有属性を持っていないそれぞれの変数が、データ共有属性の指示節中のリストによってその属性を明示的に決定しなければならないことを要求します。セクション A.28 を参照してください。

制限事項

default 指示節の制限は、以下の通りです。

- default 指示節は、parallel または task 指示文にただ１つだけ指定できます。

２．９．３．２．shared 指示節

概要

shared 指示節は、parallel または task 構文によって生成されたタスクによって共有される１つ以上のリストアイテムを宣言します。

文法

shared 指示節の文法は、以下の通りです。

```
shared (list)
```

説明

タスク内のリストアイテムへのすべての参照は、指示文に遭遇した時点のオリジナル変数の記憶領域を参照します。
明示的な task リージョンによって共有された記憶域に対して、明示的な task リージョンの実行を完了する前に、適切な同期処理の追加によってその記憶域の寿命の終わりに達しないことを確実にすることは、プログラマの責任となります。

Fortran

parallel または task 構文内で、共有ポインタが、private、firstprivate、lastprivate、または reduction 指示節にあるターゲットまたはターゲットの部分実体に結合している場合、その共有ポインタの結合状態は、parallel または task リージョンの入口と出口で未定義となります。
ある条件下で、共有変数を組込みでない手続きに渡す場合、共有変数の値は、手続き参照する前に一時的な記憶域へコピーされ、手続き引用後に実引数の記憶域へ一時的な記憶域が書き出される場合がある可能性があります。この状態が起こったとき、共有変数の値は実装依存となります。この振舞いの例としてはセクション A.29 を参照してください。

注：組込みでない手続きの引用において、以下の３つの条件が実引数に関して満たされるとき、この状態が発生します。

a．実引数が以下のいずれかの場合。
- 共有変数
- 共有変数の部分実体
- 共有変数と結合した実体
- 共有変数のサブオブジェクトと結合した実体

b．さらに、実引数が以下のいずれかの場合。
- 部分配列
- ベクトル添字がある部分配列
- 形状引継ぎ配列
- ポインタ配列

c．この実引数に対応した仮引数は、形状明示配列または大きさ引継ぎ配列である。

これは、手続きの参照中に一時的な記憶域への参照や定義という結果になります。他のタスクによる仮引数と結合された共有記憶領域への参照や定義は、競合状態の可能性を避けるために手続きの引用との同期を取らなければなりません。

Fortran

２．９．３．３．private 指示節

概要

private 指示節は、タスクに対して１つ以上のリストアイテムをプライベートにすることを宣言します。

文法

private 指示節の文法は、以下の通りです。

```
private（list）
```

説明

private 指示節に現れるリストアイテムを、構文内の文の中で参照するそれぞれのタスクは、オリジナルのリストアイテムから得られる言語仕様の属性を持った新しいリストアイテムを受け取ります。構文内において、オリジナルのリストアイテムへのすべての参照は、新しいリストアイテムへの参照へ置き換えられます。

その他のリージョンでは、新しいリストアイテムへの参照なのか、オリジナルのリストアイテムへの参照なのかは、不定となります。従って、オリジナルのアイテムへの参照を試みる場合、そのリージョン後の変数の値も不定となります。

もし、タスクが、private 指示節に現れたリストアイテムを参照しない場合、タスクが新しいリストアイテムを受け取るかどうかは不定となります。

オリジナルのリストアイテムの値および（または）割付け状態は、以下のときだけ変更されます。

- ポインタでアクセスされ、変更された。
- （可能なら）リージョン内であるが構文の外側でアクセスされた。または、
- 指示文または指示節の副作用として。

parallel 構文の private、firstprivate、または reduction 指示節に現れたリストアイテムは、それに囲まれた parallel、task、またはワークシェアリング構文の private 指示節に再び現れてもかまいません。task 構文の private または firstprivate 指示節に現れたリストアイテムは、それに囲まれた parallel または task 構文の private 指示節に再び現れてもかまいません。workshare 構文の private、firstprivate、lastprivate、または reduction 指示節に現れたリストアイテムは、それに囲まれた parallel または task 構文の private 指示節に再び現れてもかまいません。例として、セクション A.31 を参照してください。

C/C++

自動記憶域期間を持つ、同じ型の新しいリストアイテムが、構文に対して割付けられます。このようなリストアイテムの記憶領域と寿命は、それらが生成されたブロックが存在する限り、持続されます。新しいリストアイテムのサイズとアライメントは、その変数の型によって決定されます。もし、タスクが１つの文でリストアイテムを参照する場合、この割付けは、構文によって生成されたそれぞれのタスクに１度だけ行われます。

新しいリストアイテムは、初期値がないローカルな宣言であるように初期化される、または、不定の初期値を持ちます。クラス型の異なったプライベート変数のデフォルトのコンストラクタの呼び出される順序は不定となります。クラス型の異なったプライベート変数のデストラクタの呼び出される順序は不定となります。

C/C++

Fortran

もし、構文が１つの文でリストアイテムを参照する場合、同じ型の新しいリストアイテムは、parallel リージョン内のそれぞれの暗黙のタスクに、または、task 構文によって生成されたそれぞれのタスクに、一度だけ割付けられます。新しいリストアイテムの初期値は未定義です。parallel リージョン、ワークシェアリングまたは task リージョン内で、プライベートポインタの初期状態は未定義です。

ALLOCATABLE 属性を持つリストアイテムに対して、

- リスト**アイテム**が、”現在割付けられていない”場合、新しいリスト**アイテム**は、「現在割付けられていない」という初期状態になります。
- リスト**アイテム**が”割付けられている”場合、新しいリスト**アイテム**は、同じ配列の上下限で「割付けられている」という初期状態になります。

private 指示節の指定があるとき、private 指示節に現れたリストアイテムは、他の変数と記憶域結合されるかもしれません。記憶域結合は、EQUIVALENCE または COMMON のような構文によって存在します。もし、A が private 指示節に現れた変数で、B は A と記憶域結合される変数であるなら、

- B の内容、割付けまたは結合状態は、parallel または task リージョンの入口では未定義です。
- A の定義または A の割付けや結合状態の定義は、B の内容、割付けと結合状態を未定義にします。
- B の定義または B の割付けや結合状態の定義は、A の内容、割付けと結合状態を未定義にします。

例として、セクション A.32 を参照してください。

Fortran

private 指示節の例については、セクション A.30 を参照してください。

制限事項

private 指示節の制限は、以下の通りです。

・ 他の変数の一部である変数（配列または構造体の要素として）は private 指示節に現れることができません。

C/C++

・ private 指示節に現れるクラス型（または、その配列）の変数は、そのクラス型に対するアクセス可能な曖昧性がないデフォルトのコンストラクタを要求します。
・ private 指示節に現れる変数は、mutable メンバーを持つクラス型でない限り、const 修飾された型であってはいけません。
・ private 指示節に現れる変数は、不完全型または参照型であってはいけません。

C/C++

Fortran

・ private 指示節に現れる変数は、定義可能か割付け配列のいずれかでなければなりません。
・ 大きさ引継ぎ配列は、private 指示節に現れてはいけません。
・ namelist 文、可変長書式または文関数定義の式に現れた変数は、private 指示節に現れてはいけません。

Fortran

２．９．３．４．firstprivate 指示節

概要

firstprivate 指示節は、タスクに対して、１つ以上のリストアイテムをプライベートにすることを宣言します。そして、構文に遭遇したとき、対応するオリジナルのアイテムの値でそれらを初期化します。

文法

firstprivate 指示節の文法は、以下の通りです。

```
firstprivate（list）
```

説明

firstprivate 指示節は、private 指示節によって提供される機能のスーパーセットを提供します。
firstprivate 指示節に現れたリストアイテムは、セクション 2.9.3.3 で述べられる private 指示節の意味に従います。加えて、新しいリストアイテムは、構文の前に存在するオリジナルのリストアイテムで初期化されます。新しいリストアイテムの初期化は、構文内の任意の文でリストアイテムを参照するそれぞれのタスクのために、１度だけ行われます。その初期化は、構文実行前に完了します。
parallel または task 構文内の firstprivate 指示節に対して、新しいリストアイテムの初期値は、構文に遭遇する task リージョン内のその構文の直前に存在するオリジナルリストアイテムの値となります。ワークシェアリング構文内の firstprivate 指示節に対しては、ワークシェアリング構文を実行するそれぞれのスレッドの暗黙のタスクの新しいリストアイテムの初期値は、ワークシェアリング構文に遭遇した時点の直前の暗黙のタスクに存在するオリジナルリストアイテムの値となります。
競合状態を避けるために、オリジナルリストアイテムの同時の更新は、firstprivate 指示節の結果として起こるオリジナルリストアイテムの読み込みと同期を取らなければなりません。
もし、リストアイテムが、firstprivate および lastprivate 指示節の両方に現れた場合、lastprivate のために要求された更新は、firstprivate のすべての初期化の後に行われます。

C/C++

非配列型の変数に対して、初期化は、コピー代入によって行われます。非配列型の要素の配列（多次元であってもよい）に対しては、それぞれの要素は、あたかも、オリジナルの配列の要素から対応する新しい配列の要素への代入によるかのように初期化されます。クラス型の変数に対しては、初期化

のためにコピーコンストラクタが呼び出されます。異なったクラス型の変数に対して、コピーコンストラクタが呼び出される順序は不定となります。

C/C++

Fortran

新しいリストアイテムの初期化は、あたかも、代入のように行われます。

Fortran

制限事項

firstprivate 指示節の制限は、以下の通りです。

- 別の変数の一部（配列や構造体の要素など）の変数は、firstprivate 指示節に現れてはいけません。
- ワークシェアリング構文から生成した任意のワークシェアリングリージョンが、parallel 構文から生成した任意の parallel リージョンに結合している場合、parallel リージョン内のプライベートリストアイテムは、ワークシェアリング構文の firstprivate 指示節に現れてはいけません。
- ワークシェアリングまたは task 構文から生成した任意のワークシェアリングまたは task リージョンが、parallel 構文から生成した任意の parallel リージョンに結合している場合、parallel 構文の reduction 指示節に現れるリスト**アイテム**は、ワークシェアリングまたは task 構文の firstprivate 指示節に現れてはいけません。
- ワークシェアリング構文の reduction 指示節に現れるリスト**アイテム**は、ワークシェアリング構文から生成したどんなワークシェアリングリージョンの実行中に遭遇した task 構文の firstprivate 指示節に現れてはいけません。

C/C++

- firstprivate 指示節に現れるクラス型（または、その配列）の変数は、クラス型に対するアクセス可能な曖昧性がないコピーコンストラクタを要求します。
- firstprivate 指示節に現れる変数は、mutable メンバーを持つクラス型でない限り、const-qualified タイプであってはいけません。
- firstprivate 指示節に現れる変数は、不完全型または参照型であってはいけません。

C/C++

Fortran

- firstprivate 指示節に現れる変数は、定義可能でなければなりません。
- Fortran のポインタ、Cray ポインタ、および大きさ引継ぎ配列は、firstprivate 指示節に現れてはいけません。
- namelist 文、可変長書式または文関数定義の式に現れた変数は、firstprivate 指示節に現れてはいけません。

Fortran

２．９．３．５．lastprivate 指示節

概要

lastprivate 指示節は、暗黙のタスクに対して、１つ以上のリストアイテムをプライベートにすることを宣言します。そして、リージョンの終わった後、対応するオリジナルのリストアイテムを更新します。

文法

lastprivate 指示節の文法は、以下の通りです。

```
lastprivate (list)
```

説明

lastprivate 指示節は、private 指示節によって提供される機能のスーパーセットを提供します。
lastprivate 指示節に現れたリストアイテムは、セクション 2.9.3.3 で述べられる private 指示節の意味に従います。加えて、lastprivate 指示節が、ワークシェアリング構文を識別する指示文に現れたとき、それぞれの新しいリストアイテムの値は、関連付けられたループの逐次的に最後の繰り返し、または字句的に最後の section 構文から、オリジナルのリストアイテムに代入されます。

C/C++

非配列型の要素の配列（多次元であってもよい）に対しては、それぞれの要素は、オリジナル配列の対応する要素に代入されます。

C/C++

ループの逐次的に最後の繰り返し、または字句的に最後の section 構文によって値を代入されなかったリストアイテムは、構文後、不定な値となります。代入されなかったサブコンポーネントもまた、構文後、不定な値となります。
もし、構文の終わりに暗黙のバリアが存在する場合、オリジナルリストアイテムは、その時点で定義されるようになります。競合状態を避けるために、オリジナルリストアイテムの同時の読み込みや更新は、lastprivate 指示節の結果として起こるオリジナルリストアイテムの更新と同期を取らなければなりません。
lastprivate 指示文が、nowait が適用されている構文で使用されている場合、オリジナルのリストアイテムへのアクセスは、データ競合を引き起こします。これを避けるために、逐次的に最後の繰り返

し、または字句的に最後の section 構文が、リストアイテムを格納しフラッシュすることを保証するために、同期処理を挿入しなければなりません。
リストアイテムが、firstprivate と lastprivate 指示節の両方に現れた場合、lastprivate のために必要な更新は、firstprivate のためのすべての初期化の後で行われます。
lastprivate 指示節の例については、セクション A.34 を参照してください。

制限事項

lastprivate 指示節の制限は、以下の通りです。

- 別の変数の一部（配列や構造体の要素など）の変数は、lastprivate 指示節に現れてはいけません。
- ワークシェアリング構文と関連付けられたワークシェアリングリージョンが、parallel 構文と関連付けられた parallel リージョンに結合している場合、parallel リージョン内のプライベートである、または、parallel 構文の reduction 指示節に現れるリストアイテムは、ワークシェアリング構文の lastprivate 指示節に現れてはいけません。

C/C++

- lastprivate 指示節に現れるクラス型（または、その配列）の変数は、firstprivate 指示節にも指定されない限り、そのクラス型に対するアクセス可能で曖昧ではないデフォルトのコンストラクタを要求します。
- lastprivate 指示節に現れるクラス型（または、その配列）の変数は、アクセス可能で曖昧ではないコピー代入演算子を要求します。クラス型の異なった変数のためのコピー代入演算子の呼ばれる順序は、不定となります。
- lastprivate 指示節に現れる変数は、mutable メンバーを持つクラス型でない限り、const 修飾された型であってはいけません。
- lastprivate 指示節に現れる変数は、不完全型または参照型であってはいけません。

C/C++

Fortran

- lastprivate 指示節に現れる変数は、定義可能でなければなりません。
- ALLOCATABLE 属性を持つオリジナルリスト**アイテム**は、lastprivate 指示節を含む構文の入口で、割り付けられている状態でなければなりません。逐次的な最後の繰り返し、または、字句的に最後のセクションにおけるリスト**アイテム**は、その繰り返しまたは**関連付けられた**オリジナルリスト**アイテム**と同じ範囲を持つセクションから抜け出る時、割り付けられている状態でなければなりません。
- Fortran ポインタ、Cray ポインタ、および大きさ引継ぎ配列は、lastprivate 指示節に現れてはいけません。
- namelist 文、可変長書式または文関数定義の式に現れた変数は、lastprivate 指示節に現れてはいけません。

Fortran

２．９．３．６．reduction 指示節

概要

reduction 指示節は、一つの演算子と１つ以上のリストアイテムを指定します。それぞれのリストアイテムに対して、プライベートコピーがそれぞれの暗黙のタスクで生成され、その演算子のために適切に初期化されます。リージョンの終了後、オリジナルのリストアイテムは、指定された演算子を使用して、各プライベートコピーの値を用いて更新されます。

文法

C/C++

reduction 指示節の文法は、以下の通りです。

```
reduction（operator：list）
```

以下の表は、有効な演算子とそれらの初期化値を示しています。実際の初期化値は、リダクションのリストアイテムのデータの型に依存します。

Operator（演算子）	初期化値
+	0
*	1
-	0
&	～0
|	0
^	0
&&	1
||	0

C/C++

Fortran

reduction 指示節の文法は、以下の通りです。

```
reduction（ {operator | intrinsic_procedure_name}：list）
```

以下の表は、有効な演算子と組込み手続き名および初期化値を示しています。実際の初期化値は、リダクションのリストアイテムのデータの型に依存します。

Operator/Intrinsic（演算子/組込み）	初期化値
+	0
*	1
-	0
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.
max	リダクションのリストアイテムの型で負の最大値
min	リダクションのリストアイテムの型で最大値
iand	すべてのビットがオン
ior	0
ieor	0

Fortran

説明

reduction 指示節は、漸化計算（数学的に結合の演算子や可換な演算子を含む）のいくつかの形式を並列に実行するために使用することができます。
それぞれのリストアイテムのプライベートコピーは、あたかも private 指示節が使用されたかのように、暗黙のタスクごとに１つ生成されます。プライベートコピーは、上の表で示した通り、演算子に対応する初期化値で初期化されます。reduction 指示節が指定されたリージョンの終わりで、オリジナルリストアイテムは、指定した演算子を使用してそれぞれのプライベートコピーのオリジナル値と最終値の組合せによって更新されます。（減算のリダクションの部分的な結果は、最終値を成形するために加算されます。）

nowait が使用されない場合、リダクションの計算は、構文の終わりで完了します。しかし、nowait のある構文で reduction 指示節が使用された場合、オリジナルのリストアイテムへのアクセスは競合し、不定の効果をもたらします。これを避けるには、すべてのスレッドがすべての繰り返しまたは section 構文の実行を完了し、リダクションの計算の完了とリストアイテムの計算された値が格納さ

れた後に、オリジナルのリストアイテムへのアクセスが行われるように同期化する必要があります。これは、バリア同期を使うことで、最も簡単に保証することができます。

値が組合せられる順序は不定となります。従って、逐次と並列実行を比較した場合やある並列実行と他（使用されるスレッド数が同じでも）を比較した場合、ビット単位で一致する結果が得られたり、副作用（浮動小数点例外のような）が一致するという保証はありません。
競合状態を避けるためには、オリジナルリストアイテムの同時の読み込みや更新は、reduction 指示節の結果として起こるオリジナルリストアイテムの更新と同期を取らなければなりません。

注：reduction 指示節で指定されたリストアイテムは、一般的には囲まれるリージョン内で特定の形式で使用されます。

C/C++

次に示す形式の文は、リダクションの一般的な使用例です。

```
x = x op expr
x binop= expr
x = expr op x   （減算を除く）
x++
++x
x--
--x
```

expr は、*x* を参照しないスカラ型です。
op は、多重定義演算子ではなく、x、*、-、&、^、&& 、または || の一つです。
binop は、多重定義演算子ではなく、+、*、-、& 、または | の一つです。

C/C++

Fortran

次に示す形式の文は、リダクションの一般的な使用例です。

```
x = x op expr
x = expr op x   （減算を除く）
```

op は、x、*、-、.and.、.or.、.eqv. 、または .neqv. です。式は、x を含みません。
リダクション *op* は、右辺で実行される最後の演算です。

次に示す形式の文は、組込み手続きを使用した一般的な使用例です。

```
x = intr (x, expr_list)
x = intr (expr_list, x)
```

intr は、max、min、iand、ior、または ieor です。
expr_list は、x を含まないコンマで区切られた式のリストです。

Fortran

例については、セクション A.35 を参照してください。

制限事項

reduction 指示節の制限は、以下の通りです。

- ワークシェアリング構文の reduction 指示節に現れるリストアイテムは、ワークシェアリング構文から生成したワークシェアリングリージョンに結合している parallel リージョン内では、共有でなければなりません。
- 最も内側を囲んでいるワークシェアリングまたは parallel 構文の reduction 指示節に現れるリストアイテムは、明示的なタスクからアクセスされてはいけません。
- reduction 指示節は、指示文で何度も指定することができます。しかし、一つのリストアイテムは、その指示文の reduction 指示節に 1 度だけしか指定することができません。

C/C++

- reduction 指示節に現れるリストアイテムの型は、リダクション演算子にとって有効でなければなりません。
- 集合型（配列を含む）、ポインタ型、および参照型は、reduction 指示節に指定してはいけません。
- reduction 指示節に現れるリストアイテムは、const 修飾されていてはいけません。
- reduction 指示節に指定される演算子は、その指示節に現れるリストアイテムに対して多重定義することはできません。

C/C++

Fortran

- reduction 指示節に現れるリストアイテムの型は、リダクション演算子や組込み手続きにとって有効でなければいけません。
- reduction 指示節に現れるリストアイテムは、定義可能でなければなりません。
- reduction 指示節に現れるリストアイテムは、組込み型の名前付き変数でなければなりません。

- ALLOCATABLE 属性を持つオリジナルリスト**アイテム**は、reduction 指示節を含む構文の入口で、割付けされた状態でなければなりません。加えて、そのリスト**アイテム**は、リージョン内で割付け、および/または、解放をしてはいけません。
- Fortran ポインタ、Cray ポインタ、および大きさ引継ぎ配列は、reduction 指示節に現れてはいけません。
- 指定した演算子は、組込み演算子でなければなりません。また、*intrinsic_procedure_name* は、許される組込み手続きの一つでなければなりません。リダクションリスト**アイテム**への代入は、組込み代入を通して行わなければなりません。例として、セクション A.35 を参照してください。

Fortran

２．９．４．データコピー指示節

このセクションは、copyin 指示節（parallel 指示文と複合パラレル・ワークシェアリング指示文で有効）と copyprivate 指示節（single 指示文で有効）について、説明をします。
これらの指示節は、１つの暗黙のタスクまたはスレッド上のプライベートまたはスレッドプライベート変数からチーム内の他の暗黙のタスクまたはスレッド上の対応する変数にデータ値をコピーすることをサポートします。
指示節のリストアイテムでは、コンマで区切られたリストを指定します。（セクション 2.1 を参照してください。）指示節に現れているすべてのリストアイテムは、ベース言語のスコープの規則に従って、識別できなければなりません。指示節は、必要な時だけ繰り返して指定してもかまいません。しかし、変数を指定したリストアイテムは、同じ指示文で２つ以上の指示節に現れてはいけません。

２．９．４．１．copyin 指示節

概要

copyin 指示節は、マスタースレッドのスレッドプライベート変数の値を、parallel リージョンを実行するチームの他のメンバーのスレッドプライベート変数へコピーする機能を提供します。

文法

copyin 指示節の文法は、以下の通りです。

```
copyin（list）
```

説明

C/C++

コピーは、チームが形成された後、かつ、関連付けられた構造化ブロックの実行開始前に行われます。非配列型の変数については、コピーは、コピー代入によって行われます。非配列型の要素の配列（多次元であってもよい）においては、それぞれの要素は、あたかもマスタースレッドの配列の要素から対応する他のスレッドの配列の要素への代入によるかのようにコピーされます。クラス型については、コピー代入演算子が呼び出されます。異なったクラス型の変数のコピー代入演算子の呼び出される順序は不定となります。

C/C++

Fortran

コピーは、チームが形成された後、かつ、関連付けられた構造化ブロックの実行開始前に、あたかも代入によるかのように行われます。
parallel リージョンの入口では、parallel リージョンの copyin 指示節によって影響を受けるそれぞれの変数のスレッドのコピーは、以下の規則に従って、マスタースレッドのコピーの割付け状態、結合状態、および、定義状態を取得します。

- POINTER 属性である場合
  - マスタースレッドのコピーが、それぞれのコピーが結合可能なターゲットと結合している場合、それぞれのコピーは、同じターゲットに結合します。
  - マスタースレッドのコピーが、空状態の場合、それぞれのコピーは、空状態となります。
  - そうでなければ、それぞれのコピーは、不定の結合状態となります。

- POINTER 属性を持たない場合は、それぞれのコピーは、組込み代入のように、マスタースレッドのコピーの値で定義されます。

Fortran

copyin 指示節の例については、セクション A.36 を参照してください。

制限事項

copyin 指示節の制限は、以下の通りです。

C/C++

- copyin 指示節に現れるリストアイテムは、スレッドプライベートでなければなりません。
- copyin 指示節に現れるクラス型（あるいは配列）の変数は、そのクラス型に対するアクセス可能で曖昧ではないコピー代入演算子を要求します。

C/C++

Fortran

- copyin 指示節に現れるリスト**アイテム**は、スレッドプライベートでなければなりません。スレッドプライベートの共通ブロックに現れる名前付き変数は、指定してもかまいません。つまり、共通ブロック全体を指定する必要はありません。
- copyin 指示節に現れる共通ブロック名は、copyin 指示節が現れている同じスコープ単位の中で共通ブロックとして宣言されなければなりません。
- ALLOCATABLE 属性を持つ配列は、割付け状態でなければなりません。その配列のぞれぞれのスレッドのコピーは、同じ上下限で割り付けられなければなりません。

Fortran

２．９．４．２．copyprivate 指示節

概要

copyprivate 指示節は、１つの暗黙のタスクのデータ環境から parallel リージョンに属する他の暗黙のタスクのデータ環境へプライベート変数を使って値をブロードキャストする機能を提供します。競合状態を避けるために、リストアイテムの同時の読み込みや更新は、copyprivate 指示節の結果として起こるリストアイテムの更新と同期を取らなければなりません。

文法

copyprivate 指示節の文法は、以下の通りです。

```
copyprivate（list）
```

説明

copyprivate 指示節が指定されたリストアイテムへの効果は、single 構文に関連つけられた構造化ブロックの実行後（セクション 2.5.3 を参照）、および、チーム内の任意のスレッドが構文の終わりのバリアから出る前に生じます。

C/C++

parallel リージョンに属している他のすべての暗黙のタスクにおいては、それぞれの指定されたリストアイテムは、構造化ブロックを実行しているスレッドの暗黙のタスク内の対応するリストアイテムの値によって定義されます。非配列型の変数においては、定義は、コピー代入によって行われます。非配列型の要素の配列（多次元であってもよい）においては、それぞれの要素は、構造化ブロックを実行しているスレッドの暗黙のタスクのデータ環境内の配列の要素から、他の暗黙のタスクのデータ環境の配列の対応する要素へ、コピー代入によってコピーされます。クラス型においては、コピー代入演算子が呼び出されます。異なったクラス型のためのコピー代入演算子の呼び出される順序は不定となります。

C/C++

Fortran

リストアイテムがポインタでない場合は、parallel リージョンに属している他のすべての暗黙のタスクにおいて、そのリストアイテムは、構造化ブロックを実行しているスレッドの暗黙のタスク内の対応するリストアイテムの値で、（あたかも代入によるかのように）定義されます。リストアイテムがポインタの場合、parallel リージョンに属している他のすべての暗黙のタスクにおいて、そのリストアイテムは、構造化ブロックを実行しているスレッドの暗黙のタスク内の対応するリストアイテムで、（あたかもポインタ代入によるかのように）ポインタ結合されます。

Fortran

copyprivate 指示節の例については、セクション A.37 を参照してください。

注：copyprivate 指示節は、共有変数の提供が困難なときに、共有変数を使う代わりの手段となります。（例えば、再帰呼び出しにおいて、それぞれのレベルで異なる変数を要求するとき）

制限事項

copyprivate 指示節の制限は、以下の通りです。

- copyprivate 指示節に現れるすべてのリストアイテムは、囲まれたコンテキストの中で、スレッドプライベートまたはプライベートでなければなりません。
- copyprivate 指示節に現れるリストアイテムは、single 構文の private または firstprivate 指示節に現れてはいけません。

C/C++

- copyprivate 指示節に現れたクラス型（または、その配列）の変数は、そのクラス型に対するアクセス可能で曖昧ではないコピー代入演算子を要求します。

C/C++

Fortran

- copyprivate 指示節に現れた共通ブロックは、スレッドプライベートでなければなりません。
- 大きさ引継ぎ配列は、copyprivate 指示節に現れてはいけません。
- ALLOCATABLE 属性を持つ配列は、copyprivate 指示節によって影響を受けるすべてのスレッドで、同じ上下限で割付けられている状態になければなりません。

Fortran

２．１０．リージョンのネスト

このセクションでは、リージョンのネストに関する制限について説明します。ネストに関する制限は、以下の通りです。

- ワークシェアリングリージョンは、ワークシェアリング、明示的な task、critical、ordered、または master リージョン内に直接ネストされてはいけません。
- barrier リージョンは、ワークシェアリング、明示的な task、critical、ordered、または master リージョン内に直接ネストされてはいけません。
- master リージョンは、ワークシェアリングまたは明示的な task リージョン内に直接ネストされてはいけません。
- ordered リージョンは、critical または明示的な task リージョン内に直接ネストされてはいけません。
- ordered リージョンは、ordered 指示節を持つループリージョン（または、パラレルループリージョン）内に直接ネストされなければなりません。
- critical リージョンは、同じ名前の critical リージョン内に（直接、またはそれ以外でも）ネストされてはいけません。この制限は、デッドロックを防ぐのには十分でないことに注意してさい。

これらの規則を説明した例は、セクション A.17、セクション A.38、セクション A.39、およびセクション A.13 を参照してください。

第３章

実行時ライブラリルーチン

この章では、以下の構成で OpenMP API 実行時ライブラリについて説明します。

- 実行時ライブラリの定義（セクション 3.1）
- 並列実行環境の制御や問合せをするために使用する実行環境ルーチン（セクション 3.2）
- データへのアクセスを同期して行うために使用するロックルーチン（セクション 3.3）
- ポータブルな時間計測ルーチン（セクション 3.4）

本章を通して、ルーチンの説明を簡略化するために *true* と *false* を一般的な用語として使用します。

C/C++

true はゼロでない整数値を、*false* は整数値ゼロを意味します。

C/C++

Fortran

true は TRUE.の論理値を、*false* は.FALSE.の論理値を意味します。

Fortran

Fortran

制限事項

以下の制限が、すべての OpenMP 実行時ライブラリルーチンに適用されます。

- OpenMP 実行時ライブラリルーチンは、PURE または ELEMENTAL 手続きから呼び出してはいけません。

Fortran

３．１．実行時ライブラリの定義

それぞれのベース言語に対して準拠した実装は、OpenMP API 実行時ライブラリルーチンとそのパラメタの特別なデータタイプの定義を提供しなければなりません。その定義は、それぞれの OpenMP API 実行時ライブラリルーチンの宣言とデータ型として simple lock、nestable lock、および schedule の宣言を含んでいなければなりません。さらに、それぞれの定義で他の実装依存の値を定義してもかまいません。

C/C++

ライブラリルーチンは"C"リンケージの外部関数です。

この章で説明する C/C++の実行時ライブラリルーチンのプロトタイプ宣言は、omp.h という名前のヘッダファイルで提供されなければなりません。このファイルは以下を定義します。

- 本章のすべてのルーチンのプロトタイプ
- 型　omp_lock_t
- 型　omp_nest_lock_t
- 型　omp_sched_t

このファイルの例はセクション D.1 を参照してください。

C/C++

Fortran

OpenMP Fortran API 実行時ライブラリルーチンは外部手続きです。これらのルーチンの返却値は、特別な指定がない限りデフォルト種別となります。

この章で説明する OpenMP Fortran 実行時ライブラリルーチンのための引用仕様宣言は、omp_lib.h という名前の Fortran インクルードファイル、または omp_lib という名前の Fortran 90 のモジュールの形式で提供されなければなりません。インクルードファイルとモジュールのどちらが提供されるか、または両方提供されるかは実装依存となります。

このファイルは以下を定義します。

- 本章のすべてのルーチンのインタフェース
- 整数パラメタ omp_lock_kind
- 整数パラメタ omp_nest_lock_kind
- 整数パラメタ omp_sched_kind
- 値 yyyymm を持つ整数パラメタ openmp_version。yyyy と mm は、実装がサポートする OpenMP Fortran API バージョンの年と月です。マクロプリプロセッサをサポートしている場合、この値は C プリプロセッサのマクロ _OPENMP と一致します。（セクション 2.2 を参照してください。）

このファイルの例はセクション D.2 とセクション D.3 を参照してください。

引数を持つ任意の OpenMP 実行時ライブラリルーチンが異なる種別の引数に対応できるように、総称引用仕様で拡張するかどうかは実装依存です。この拡張の例は付録 D.4 を参照してください。

Fortran

3．2．実行環境ルーチン

このセクションで説明するルーチンは、スレッドやプロセッサ、並列環境に影響を与えたり、モニタリングをします。

- omp_set_num_threads ルーチン
- omp_get_num_threads ルーチン
- omp_get_max_threads ルーチン
- omp_get_thread_num ルーチン
- omp_get_num_procs ルーチン
- omp_in_parallel ルーチン
- omp_set_dynamic ルーチン
- omp_get_dynamic ルーチン
- omp_set_nested ルーチン
- omp_get_nested ルーチン
- omp_set_schedule ルーチン
- omp_get_schedule ルーチン
- omp_get_thread_limit ルーチン
- omp_set_max_active_levels ルーチン
- omp_get_max_active_levels ルーチン
- omp_get_level ルーチン
- omp_get_ancestor_thread_num ルーチン
- omp_get_team_size ルーチン
- omp_get_active_level ルーチン

３．２．１．omp_set_num_threads

概要

omp_set_num_threads ルーチンは、*nthreads-var* 内部制御変数にその値を設定することにより、num_threads 指示節の指定がない次の parallel リージョンが使用するスレッド数に影響を与えます。

書式

C/C++

```
void omp_set_num_threads (int num_threads);
```

C/C++

Fortran

```
subroutine omp_set_num_threads (num_threads)
integer num_threads
```

Fortran

引数の制約

このルーチンに渡される引数の値は正の整数でなければなりません。そうでない場合、このルーチンの振舞いは実装依存となります。

結合

omp_set_num_threads リージョンに対する結合タスクセットは、そのリージョンを生成したタスクです。

効果

このルーチンの効果は、*nthreads-var* 内部制御変数の値を引数で指定した値に設定することです。parallel リージョンを実行するスレッド数を決定する規則については、セクション 2.4.1 を参照してください。
omp_set_num_threads ルーチンの例はセクション A.40 を参照してください。

相互参照

- nthreads-var 内部制御変数については、セクション 2.3 を参照してください。
- OMP_NUM_THREADS 環境変数については、セクション 4.2 を参照してください。
- omp_get_max_threads ルーチンについては、セクション 3.2.3 を参照してください。
- parallel 構文については、セクション 2.4 を参照してください。
- num_threads 指示節については、セクション 2.4 を参照ください。

3．2．2．omp_get_num_threads

概要

omp_get_num_threads ルーチンは現在のチームのスレッド数を返します。

書式

C/C++

```
int omp_get_num_threads (void);
```

C/C++

Fortran

```
integer function omp_get_num_threads ()
```

Fortran

結合

omp_get_num_threads リージョンに対する結合リージョンは、囲んでいる parallel リージョンのうち最も内側の parallel リージョンです。

効果

omp_get_num_threads ルーチンは、ルーチンのリージョンに結合している parallel リージョンを実行するチームのスレッド数を返します。プログラムの逐次部分から呼び出された場合、このルーチンは 1 を返します。例として、セクション A.41 を参照してください。
parallel リージョンを実行するスレッド数を決定する規則については、セクション 2.4.1 を参照してください。

相互参照

- parallel 構文については、セクション 2.4 を参照してください。
- omp_set_num_threads ルーチンについては、セクション 3.2.1 を参照してください。
- OMP_NUM_THREADS 環境変数については、セクション 4.2 を参照してください。

3．2．3．omp_get_max_threads

概要

omp_get_max_threads ルーチンは、実行がこのルーチンから戻った後に num_threads 指示節の指定のない parallel リージョンに遭遇した場合に新しいチームが使用できるスレッド数の上限値を返します。

書式

C/C++

```
int omp_get_max_threads (void);
```

C/C++

Fortran

```
integer function omp_get_max_threads ()
```

Fortran

結合

omp_get_max_threads リージョンに対する結合タスクセットは、そのリージョンを生成したタスクです。

効果

omp_get_max_threads が返す値は *nthreads-var* 内部制御変数の値です。この値は、実行がこのルーチンから戻った後に num_threads 指示節の指定がない parallel リージョンに遭遇したた場合に新しいチームが使用できるスレッド数の上限値になります。
parallel リージョンを実行するスレッド数を決定する規則については、セクション 2.4.1 を参照してください。

注：omp_get_max_threads ルーチンの返却値は、後の活動状態の parallel リージョンで作られるチームのすべてのスレッドに十分な記憶域を動的に割り当てるために使うことができます。

相互参照

- nthreads-var 内部制御変数については、セクション 2.3 を参照してください。
- parallel 構文については、セクション 2.4 を参照してください。
- num_threads 指示節については、セクション 2.4 を参照してください。
- omp_set_num_threads ルーチンについては、セクション 3.2.1 を参照してください。
- OMP_NUM_THREADS 環境変数については、セクション 4.2 を参照してください。

3．2．4．omp_get_thread_num

概要

omp_get_thread_num ルーチンは、このルーチンを呼び出した暗黙または明示的な task リージョンを実行しているスレッドの現在のチーム内のスレッド番号を返します。

書式

C/C++

```
int omp_get_thread_num (void);
```

C/C++
Fortran

```
integer function omp_get_thread_num ()
```

Fortran

結合

omp_get_thread_num リージョンに対する結合スレッドセットは現在のチームです。omp_get_thread_num リージョンに対する結合リージョンは、囲んでいる parallel リージョンのうち最も内側の parallel リージョンです。

効果

omp_get_thread_num ルーチンは、このルーチンのリージョンが結合している parallel リージョンを実行するチームにおける現在のスレッドのスレッド番号を返します。スレッド番号は、ゼロから omp_get_num_threads により返される値より１小さい値までの整数です。チームのマスタースレッドのスレッド番号はゼロです。プログラムの逐次部分から呼ばれた場合、このルーチンはゼロを返します。

注：アンタイドタスクの実行中は、スレッド番号がいつ変更されるかわかりません。そのような task リージョン実行中に omp_get_thread_num が返却する値は、一般的にあまり有用ではありません。

相互参照

- omp_get_num_threads ルーチンについては、セクション 3.2.2 を参照してください。

３．２．５．omp_get_num_procs

概要

omp_get_num_procs ルーチンはプログラムで利用可能なプロセッサ数を返します。

書式

C/C++

```
int omp_get_num_procs (void);
```

C/C++

Fortran

```
integer function omp_get_num_procs ()
```

Fortran

結合

omp_get_num_procs リージョンに対する結合スレッドセットはすべてのスレッドです。このルーチンの実行は、他の構文や API ルーチンに対応する特定のリージョンと関係はありません。

効果

omp_get_num_procs ルーチンは、このルーチンが呼び出された時点でプログラムが利用可能なプロセッサ数を返します。omp_get_num_procs ルーチンによって決まる時点と、OpenMP 実装の制御外でシステム動作のために呼び出したコンテキストが読む時点で、この値が変わっている可能性があることに注意してください。

３．２．６．omp_in_parallel

概要

omp_in_parallel ルーチンは、活動状態の parallel リージョン内から呼ばれた場合に *true* を返します。それ以外の場合には *false* を返します。

書式

C/C++

```
int omp_in_parallel (void);
```

C/C++

Fortran

```
logical function omp_in_parallel ()
```

Fortran

結合

omp_in_parallel リージョンに対する結合スレッドセットは、すべてのスレッドです。このルーチンの実行は、特定の parallel リージョンに関係するのではなく、囲んでいるすべての parallel リージョンの状態に依存します。

効果

囲んでいる parallel リージョンのどれかが活動状態の場合に omp_in_parallel ルーチンは *true* を返します。この呼び出しが暗黙の parallel リージョンを含めた非活動状態の parallel リージョンだけで囲まれている場合に *false* を返します。

３．２．７．omp_set_dynamic

概要

omp_set_dynamic ルーチンは、*dyn-var* 内部制御変数にその値を設定することにより、その後の parallel リージョンの実行で利用可能なスレッド数の動的調整を有効または無効にします。

書式

C/C++

```
void omp_set_dynamic (int dynamic_threads);
```

C/C++

Fortran

```
subroutine omp_set_dynamic (dynamic_threads)
logical dynamic_threads
```

Fortran

結合

omp_set_dynamic リージョンに対する結合タスクセットは、そのリージョンを生成したタスクです。

効果

スレッド数の動的調整をサポートする実装において omp_set_dynamic の引数が *true* と評価された場合に動的調整は有効となります。それ以外では動的調整は無効となります。
スレッド数の動的調整をサポートしない実装でこのルーチンを実行しても何の効果もありません。
dyn-var の値は常に false です。
omp_set_dynamic ルーチンの例は、セクション A.40 を参照してください。
parallel リージョンを実行するスレッド数を管理する規則については、セクション 2.4.1 を参照してください。

相互参照

- *dyn-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_get_num_threads ルーチンについては、セクション 3.2.2 を参照してください。
- omp_get_dynamic ルーチンについては、セクション 3.2.8 を参照してください。
- OMP_DYNAMIC 環境変数については、セクション 4.3 を参照してください。

3．2．8．omp_get_dynamic

概要

omp_get_dynamic ルーチンは、スレッド数の動的調整が有効か無効かを決定する *dyn-var* 内部制御変数の値を返します。

書式

C/C++

```
int omp_get_dynamic (void);
```

C/C++

Fortran

```
logical function omp_get_dynamic ( )
```

Fortran

結合

omp_get_dynamic リージョンに対する結合タスクセットは、そのリージョンを生成したタスクです。

効果

スレッド数の動的調整が有効である場合、このルーチンは *true* を返します。それ以外の場合は *false* を返します。実装がスレッド数の動的調整をサポートしていなければ、このルーチンは常に *false* を返します。

parallel リージョンを実行するスレッド数を決定する規則については、セクション 2.4.1 を参照してください。

相互参照

- *dyn-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_set_dynamic ルーチンについては、セクション 3.2.7 を参照してください。
- OMP_DYNAMIC 環境変数については、セクション 4.3 を参照してください。

3．2．9．omp_set_nested

概要

omp_set_nested ルーチンは、*nest-var* 内部制御変数にその値を設定することにより、ネスト並列を有効または無効にします。

書式

C/C++

```
void omp_set_nested (int nested);
```

C/C++

Fortran

```
subroutine omp_set_nested (nested)
logical nested
```

Fortran

結合

omp_set_nested リージョンに対する結合タスクセットは、そのリージョンを生成したタスクです。

効果

ネスト並列をサポートする実装において omp_set_nested の引数が *true* と評価された場合にネスト並列は有効になります。それ以外ではネスト並列は無効になります。

ネスト並列をサポートしない実装では、このルーチンを実行しても何の効果もありません。*nest-var* の値は常に *false* です。
parallel リージョンを実行するスレッド数を決定する規則については、セクション 2.4.1 を参照してください。

相互参照

- *nest-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_set_max_active_levels ルーチンについては、セクション 3.2.14 を参照してください。
- omp_get_max_active_levels ルーチンについては、セクション 3.2.15 を参照してください。
- omp_get_nested ルーチンについては、セクション 3.2.10 を参照してください。
- OMP_NESTED 環境変数については、セクション 4.4 を参照してください。

3．2．10．omp_get_nested

概要

omp_get_nested ルーチンは、ネスト並列が有効か無効かを決定する *nest-var* 内部制御変数の値を返します。

書式

C/C++

```
int omp_get_nested (void);
```

C/C++

Fortran

```
logical function omp_get_nested ( )
```

Fortran

結合

omp_get_nested リージョンに対する結合タスクセットは、そのリージョンを生成したタスクです。

効果

ネスト並列が有効である場合、このルーチンは *true* を返します。それ以外の場合は *false* を返します。実装がネスト並列をサポートしていなければ、このルーチンは常に *false* を返します。
parallel リージョンを実行するために使用するスレッド数を決定する規則については、セクション 2.4.1 を参照してください。

相互参照

- *nest-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_set_nested ルーチンについては、セクション 3.2.9 を参照してください。
- OMP_NESTED 環境変数については、セクション 4.4 を参照してください。

３．２．１１．omp_set_schedule

概要

omp_set_schedule ルーチンは、*run-sched-var* 内部制御変数にその値を設定することにより、スケジュール種別に runtime が指定された場合のスケジュールに影響を与えます。

書式

C/C++

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

C/C++

Fortran

```
subroutine omp_set_schedule (kind, modifier)
integer (kind=omp_sched_kind) kind
integer modifier
```

Fortran

引数の制約

このルーチンの第一引数は、runtime 以外の有効な OpenMP スケジュール種別の一つ、または実装固有のスケジュールです。C/C++ヘッダファイル（omp.h）と Fortran インクルードファイル（omp_lib.h）、Fortran 90 モジュールファイル（omp_lib）で有効な定数を定義します。有効な定数は以下の定数を含まなければなりません。実装固有の定数値を追加することができます。

C/C++

```
typedef enum omp_sched_t {
        omp_sched_static = 1,
        omp_sched_dynamic = 2,
        omp_sched_guided = 3,
        omp_sched_auto = 4
} omp_sched_t ;
```

C/C++

Fortran

```
integer (kind=omp_sched_kind) , parameter :: omp_sched_static = 1
integer (kind=omp_sched_kind) , parameter :: omp_sched_dynamic = 2
integer (kind=omp_sched_kind) , parameter :: omp_sched_guided = 3
integer (kind=omp_sched_kind) , parameter :: omp_sched_auto = 4
```

Fortran

結合

omp_set_schedule リージョンに対する結合タスクセットは、このリージョンを生成したタスクです。

効果

このルーチンの効果は、２つの引数に指定した値を *run-sched-var* 内部制御変数の値に設定することです。スケジュールは第一引数 kind で指定されたスケジュールタイプに設定されます。スケジュールは標準のスケジュールタイプのどれか、または実装固有のスケジュールです。スケジュールタイプ static、dynamic、guided では、*chunk_size* が第二引数の値に設定されます。第二引数の値が１より小さい場合、デフォルトの *chunk_size* が設定されます。スケジュールタイプ auto で第二引数は意味を持ちません。実装固有のスケジュールタイプでの第二引数の値と意味は実装依存となります。

相互参照

- 内部制御変数 *run-sched-var* については、セクション 2.3 を参照してください。
- omp_get_schedule については、セクション 3.2.12 を参照してください。
- OMP_SCHEDULE 環境変数については、セクション 4.1 を参照してください。
- ワークシェアリングループのスケジュールの決定については、セクション 2.5.1.1 を参照してください。

３．２．１２．omp_get_schedule

omp_get_schedule ルーチンは runtime スケジュールが指定されたときに適用されるスケジュールを返します。

書式

C/C++

```
void omp_get_schedule (omp_sched_t * kind, int * modifier);
```

C/C++

Fortran

```
subroutine omp_get_schedule (kind, modifier)
integer (kind=omp_sched_kind) kind
integer modifier
```

Fortran

結合

omp_get_schedule リージョンに対する結合タスクセットは、このリージョンを生成したタスクです。

効果

このルーチンは、ルーチンが結合している parallel リージョンを実行するチームの *run-sched-var* 内部制御変数の値を返します。第一引数 kind は使用されるスケジュールになります。セクション

3.2.11 で定義する標準のスケジュールタイプのどれか、または実装固有のスケジュールタイプです。第二引数の説明はセクション 3.2.11 の omp_set_schedule ルーチンと同じです。

相互参照

- *run-sched-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_set_schedue ルーチンについては、セクション 3.2.11 を参照してください。
- OMP_SCHEDULE 環境変数については、セクション 4.1 を参照してください。
- ワークシェアリングループのスケジュールの決定については、セクション 2.5.1.1 を参照してください。

３．２．１３．omp_get_thread_limit

概要

omp_get_thread_limit ルーチンは、プログラムで利用可能な OpenMP スレッドの最大数を返します。

書式

C/C++

```
int omp_get_thread_limit (void);
```

C/C++

Fortran

```
integer function omp_get_thread_limit ( )
```

Fortran

結合

omp_get_thread_limit リージョンに対する結合スレッドセットは、すべてのスレッドです。このルーチンの実行は、他の構文や API ルーチンに対応する特定のリージョンと何の関係もありません。

効果

omp_get_thread_limit ルーチンは、*thread-limit-var* 内部制御変数に設定されているプログラムで利用可能な OpenMP スレッドの最大数を返します。

相互参照

- *thread-limit-var* 内部制御変数については、セクション 2.3 を参照してください。
- OMP_THREAD_LIMIT 環境変数については、セクション 4.8 を参照してください。

３．２．１４．omp_set_max_active_levels

概要

omp_set_max_active_levels ルーチンは、*max-active-levels-var* 内部制御変数にその値を設定することにより、ネストした活動状態の parallel リージョンの数を制限します。

書式

C/C++

```
void omp_set_max_active_levels (int max_levels);
```

C/C++

Fortran

```
subroutine omp_set_max_active_levels (max_levels)
integer max_levels
```

Fortran

引数の制約

このルーチンに渡される引数の値は負でない整数でなければなりません。負の整数である場合のルーチンの振舞いは実装依存となります。

結合

プログラムの逐次部分から呼び出された場合、omp_set_max_active_levels リージョンに対する結合スレッドセットは、このリージョンに遭遇したスレッドです。任意の明示的な parallel リージョンの中から呼び出された場合、omp_set_max_active_levels リージョンに対する結合スレッドセット（そして必要なら結合しているリージョンも）は実装依存となります。

効果

このルーチンの効果は、引数に指定した値を *max-active-levels-var* 内部制御変数に設定することです。
実装がサポートする数より多い並列レベルを要求した場合、*max-active-levels-var* 内部制御変数の値には実装がサポートする並列レベルが設定されます。
このルーチンは、プログラムの逐次部分から呼び出された場合にだけ上記効果があります。
明示的な parallel リージョン内から呼び出された場合、このルーチンの効果は実装依存となります。

相互参照

- *thread-limit-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_get_max_active_levels ルーチンについては、セクション 3.2.15 を参照してください。
- OMP_MAX_ACTIVE_LEVELS 環境変数については、セクション 4.7 を参照してください。

３．２．１５．omp_get_max_active_levels

概要

omp_get_max_active_levels ルーチンは、ネストした活動状態の parallel リージョンの最大数を決定する *max-active-levels-var* 内部制御変数の値を返します。

書式

C/C++

```
int omp_get_max_active_levels (void);
```

C/C++

Fortran

```
integer function omp_get_max_active_levels ( )
```

Fortran

結合

プログラムの逐次部分から呼び出された場合、omp_get_max_active_levels リージョンに対する結合スレッドセットは、そのリージョンに遭遇したスレッドです。任意の明示的な parallel リージョンの中から呼び出された場合、omp_get_max_active_levels リージョンに対する結合スレッドセット（そして必要なら結合しているリージョンも）は実装依存となります。

効果

omp_get_max_active_levels ルーチンは、ネストした活動状態の parallel リージョンの最大数を決定する *max-active-levels-var* 内部制御変数の値を返します。

相互参照

- *thread-limit-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_set_max_active_levels ルーチンについては、セクション 3.2.14 を参照してください。
- OMP_MAX_ACTIVE_LEVELS 環境変数については、セクション 4.7 を参照してください。

３．２．１６．omp_get_level

概要

omp_get_level ルーチンは、そのルーチンを呼び出したタスクを囲むネストした parallel リージョンの数を返します。

書式

C/C++

```
int omp_get_level (void);
```

C/C++

Fortran

```
integer function omp_get_level ( )
```

Fortran

結合

omp_get_level リージョンに対する結合タスクセットは、このリージョンを生成したタスクです。omp_get_level リージョンに対する結合リージョンは、このリージョンを囲んでいる parallel リージョンで最も内側の parallel リージョンです。

効果

omp_get_level ルーチンは、このルーチンを呼び出すタスクを囲んでいる、暗黙の parallel リージョンを含まないネストした parallel リージョン（活動状態または非活動状態）の数を返します。このルーチンは常に負でない整数を返します。また、プログラムの逐次部分から呼び出された場合はゼロを返します。

相互参照

- omp_get_active_level ルーチンについては、セクション 3.2.19 を参照してください。
- OMP_MAX_ACTIVE_LEVELS 環境変数については、セクション 4.7 を参照してください。

３．２．１７．omp_get_ancestor_thread_num

概要

omp_get_ancestor_thread_num ルーチンは、現在のスレッドにネストレベルを与えることで、先祖のスレッド番号または現在のスレッド番号を返します。

書式

C/C++

```
int omp_get_ancestor_thread_num (int level);
```

C/C++

Fortran

```
integer function omp_get_ancestor_thread_num (level)
integer level
```

Fortran

結合

omp_get_ancestor_thread_num リージョンに対する結合スレッドセットは、このリージョンに遭遇したスレッドです。omp_get_ancestor_thread_num リージョンに対する結合リージョンは、囲んでいる parallel リージョンのうち最も内側の parallel リージョンです。

効果

omp_get_ancestor_thread_num ルーチンは、現在のスレッドで指定したネストレベルの先祖のスレッド番号、または現在のスレッドのスレッド番号を返します。指定したネストレベルが、ゼロから omp_get_level ルーチンが返す現在のスレッドのネストレベルまでの範囲外である場合、ルーチンは－１を返します。

注：omp_get_ancestor_thread_num ルーチンが level=0 で呼び出された場合、このルーチンは常に０を返します。level=omp_get_level( )の場合、このルーチンは omp_get_thread_num ルーチンと同じ結果となります。

相互参照

- omp_get_level ルーチンについては、セクション 3.2.16 を参照してください。
- omp_get_thread_num ルーチンについては、セクション 3.2.4 を参照してください。
- omp_get_team_size ルーチンについては、セクション 3.2.18 を参照してください。

３．２．１８．omp_get_team_size

概要

omp_get_team_size ルーチンは、現在のスレッドの指定したネストレベルに対応した、先祖または現在のスレッドが属するスレッドチームのサイズを返します。

書式

C/C++

```
int omp_get_team_size (int level);
```

C/C++

Fortran

```
integer function omp_get_team_size (level)
integer level
```

Fortran

結合

omp_get_team_size リージョンに対する結合スレッドセットは、そのリージョンに遭遇したスレッドです。omp_get_team_size リージョンに対する結合リージョンは、囲んでいる parallel リージョンのうち最も内側の parallel リージョンです。

効果

omp_get_team_size ルーチンは、先祖または現在のスレッドが属しているスレッドチームのサイズを返します。指定されたネストレベルが、ゼロから omp_get_level ルーチンが返す現在のスレッドのネストレベルまでの範囲外である場合、ルーチンは－１を返します。非活動状態の parallel リージョンは１つのスレッドで実行する活動状態の parallel リージョンと見なされます。

注：omp_get_team_size ルーチンが level=0 で呼び出された場合、このルーチンは常に 1 を返します。level=omp_get_level( )の場合、このルーチンは omp_get_num_threads ルーチンと同じ結果となります。

相互参照

- omp_get_num_threads ルーチンについては、セクション 3.2.2 を参照してください。
- omp_get_level ルーチンについては、セクション 3.2.16 を参照してください。
- omp_get_ancestor_thread_num ルーチンについては、セクション 3.2.17 を参照してください。

３．２．１９．omp_get_active_level

概要

omp_get_active_level ルーチンは、このルーチンを呼び出すタスクを囲んでいる活動状態の parallel リージョンがネストしている数を返します。

書式

C/C++

```
int omp_get_active_level (void);
```

C/C++

Fortran

```
integer function omp_get_active_level ( )
```

Fortran

結合

omp_get_active_level リージョンに対する結合タスクセットは、このリージョンを生成したタスクです。omp_get_active_level リージョンに対する結合リージョンは、囲んでいる parallel リージョンのうち最も内側の parallel リージョンです。

効果

omp_get_active_level ルーチンは、このルーチンを呼び出すタスクを囲んでいる活動状態の parallel リージョンがネストしている数を返します。このルーチンは常に負でない整数を返します。プログラムの逐次部分から呼び出された場合はゼロを返します。

相互参照

・ omp_get_level ルーチンについては、セクション 3.2.16 を参照してください。

３．３．ロックルーチン

OpenMP 実行時ライブラリは、同期のために使用できる汎用的なロックルーチンを含んでいます。この汎用的なロックルーチンは、OpenMP ロック変数で表される OpenMP ロックを操作します。OpenMP ロック変数は、このセクションで記述されるルーチンを通してアクセスしなければなりません。他の方法で OpenMP ロック変数にアクセスするプログラムは規格に準拠していません。

OpenMP ロックの状態は *uninitialized*、*unlocked*、*locked* のうちの一つです。
ロックの状態が *unlocked* である場合、タスクはロックの状態を *locked* に設定することができます。ロックを設定したタスクは「ロックを所有している」と言います。ロックを所有しているタスクは、ロックを解除して *unlocked* の状態に戻すことができます。別のタスクが所有しているロックを解除するタスクを含むプログラムは規格に準拠していません。

単純ロック（simple locks）とネスト可能なロック（nestable locks）の２つのタイプのロックをサポートしています。
ネスト可能なロックは、ロックを解除する前に同じタスクが何度も定することができます。単純ロックは、ロックを設定しようとするタスクがすでに所有しているロックを設定することはできません。
単純ロック変数は単純ロックと関係付けられ、単純ロックルーチンのみに渡すことができます。ネスト可能なロック変数はネスト可能なロックと関係付けられ、ネスト可能なロックルーチンのみに渡すことができます。

それぞれのロックルーチンがアクセスするロックの状態と所有権に関する制約は、それぞれのルーチンとあわせて説明します。この制約が満たされない場合のルーチンの振舞いは不定になります。

OpenMP ロックルーチンがロック変数にアクセスするときは、常に最新のロック変数の値に対して読み込みと更新をします。ロックルーチンはリストの指定がないフラッシュを含んでいます。常に、ロック変数の読込みや更新はフラッシュを伴うアトミックな操作として実装しなければなりません。したがって、ロック変数の値が異なるタスク間で一致することを保証するために、OpenMP プログラムにおいて明示的な flush 指示文を記述する必要はありません。

単純ロックとネスト可能なロックのそれぞれの使用例については、セクション A.44 およびセクション A.45 を参照してください。

結合

すべてのロックルーチンのリージョンに対する結合スレッドセットは、すべてのスレッドです。そのためロックルーチンは、タスクを実行しているスレッドが属しているチームに関係なく、そのOpenMP ロックに対してロックルーチンを呼び出すタスク全てに影響します。

単純ロックルーチン

C/C++

omp_lock_t 型は単純ロックを表現することができるデータ型です。以下のルーチンに対して、単純ロック変数は omp_lock_t 型でなければなりません。すべての単純ロックルーチンは omp_lock_t 型の変数へのポインタが引数に必要です。

C/C++

Fortran

以下のルーチンに対して、単純ロック変数は kind=omp_lock_kind の整数変数でなければなりません。

Fortran

単純ロックルーチンは以下の通りです。

- omp_init_lock ルーチンは単純ロックを初期化します。
- omp_destroy_lock ルーチンは単純ロックを初期化前の状態に戻します。
- omp_set_lock ルーチンは単純ロックが利用可能になるまで待ち、そのロックを設定します。
- omp_unset_lock ルーチンは単純ロックを解除します。
- omp_test_lock ルーチンは単純ロックをテストし、利用可能ならばロックを設定します。

ネスト可能なロックルーチン

C/C++

omp_nest_lock_t 型は、ネスト可能なロックを表現することができるオブジェクトタイプです。以下のルーチンに対して、ネスト可能なロック変数は omp_nest_lock_t 型でなければなりません。すべてのネスト可能なロックルーチンは omp_nest_lock_t 型の変数へのポインタが引数に必要です。

C/C++

Fortran

以下のルーチンに対して、ネスト可能なロック変数は kind=omp_nest_lock_kind の整数変数でなければなりません。

Fortran

ネスト可能なロックルーチンは以下の通りです。

- omp_init_nest_lock ルーチンはネスト可能なロックを初期化します。
- omp_destroy_nest_lock ルーチンはネスト可能なロックを初期化前の状態に戻します。
- omp_set_nest_lock ルーチンはネスト可能なロックが利用可能になるまで待ち、そのロックを設定します。
- omp_unset_nest_lock ルーチンはネスト可能なロックを解除します。
- omp_test_nest_lock ルーチンはネスト可能なロックをテストし、利用可能ならばロックを設定します。

３．３．１．omp_init_lock と omp_init_nest_lock

概要

これらのルーチンは、OpenMP ロックを初期化する唯一の手段を提供します。

書式

C/C++

```
void omp_init_lock (omp_lock_t *lock);
void omp_init_nest_lock (omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_init_lock (svar)
integer (kind=omp_lock_kind) svar

subroutine omp_init_nest_lock (nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

引数の制約

これらのルーチンで状態が *uninitialized* でないロックへアクセスするプログラムは規格に準拠していません。

効果

これらのルーチンの効果はロックを unlocked 状態に初期化することです。(どのタスクもそのロックを所有していない状態です。) さらに、ネスト可能なロックのネスト数はゼロに設定されます。
omp_init_lock ルーチンの例はセクション A.42 を参照してください。

3．3．2．omp_destroy_lock と omp_destroy_nest_lock

概要

これらのルーチンは OpenMP ロックが uninitialized 状態であることを保証します。

書式

C/C++

```
void omp_destroy_lock (omp_lock_t *lock);
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
```

C/C++
Fortran

```
subroutine omp_destroy_lock (svar)
integer (kind=omp_lock_kind) svar

subroutine omp_destroy_nest_lock (nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

引数の制約

これらのルーチンで状態が unlocked でないロックにアクセスするプログラムは規格に準拠していません。

効果

これらのルーチンの効果はロックの状態を uninitialized に変更することです。

３．３．３．omp_set_lock と omp_set_nest_lock

概要

これらのルーチンは OpenMP ロックを設定する手段を提供します。これらのルーチンを呼び出したタスクのリージョンはロックが設定されるまでサスペンドされます。

書式

C/C++

```
void omp_set_lock (omp_lock_t *lock);
void omp_set_nest_lock (omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_set_lock (svar)
integer (kind=omp_lock_kind) svar

subroutine omp_set_nest_lock (nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

引数の制約

これらのルーチンで状態が uninitialized のロックにアクセスするプログラムは規格に準拠していません。omp_set_lock がアクセスする locked 状態の単純ロックは、そのルーチンを呼び出すタスクが所有していてはなりません。そうでないとデッドロックが発生します。

効果

これらのルーチンは、そのルーチンを実行したタスクを指定したロックが利用可能になるまでサスペンドします。その後、ロックを設定します。
単純ロックは unlocked である場合に利用可能です。ロックの所有権はルーチンを実行したタスクに与えられます。
ネスト可能なロックは unlocked であるか、ルーチンを実行しているタスクがすでに所有している場合に利用可能です。ロックの所有権はルーチンを実行したタスクに与えられるか、そのまま持ち続けます。そしてロックのネスト数を加算します。

３．３．４．omp_unset_lock と omp_unset_nest_lock

概要

これらのルーチンは OpenMP ロックを解除する手段を提供します。

書式

C/C++

```
void omp_unset_lock (omp_lock_t *lock);
void omp_unset_nest_lock (omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_unset_lock (svar)
integer (kind=omp_lock_kind) svar

subroutine omp_unset_nest_lock (nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

引数の制約

これらのルーチンで、状態が locked でないか、そのルーチンを呼び出すタスクが所有していないロックにアクセスするプログラムは規格に準拠していません。

効果

omp_unset_lock ルーチンは単純ロックのロックを解除します。
omp_unset_nest_lock ルーチンはネスト可能なロックのネスト数を減算して、その結果がゼロになった場合はロックを解除します。
ロックが解除され、かつロックが利用可能でないために１つ以上の task リージョンがサスペンドされている場合、これらのルーチンの効果は、タスクが１つ選択されてロックの所有権を与えられます。

３．３．５．omp_test_lock と omp_test_nest_lock

概要

これらのルーチンは OpenMP ロックを設定することを試みますが、ルーチンを実行しているタスクの実行はサスペンドしません。

書式

C/C++

```
int omp_test_lock (omp_lock_t *lock);
int omp_test_nest_lock (omp_nest_lock_t *lock);
```

C/C++

Fortran

```
logical function omp_test_lock (svar)
integer (kind=omp_lock_kind) svar

integer function omp_test_nest_lock (nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

引数の制約

これらのルーチンで状態が uninitialized のロックにアクセスするプログラムは規格に準拠していません。omp_test_lock でアクセスする locked 状態の単純ロックがそのルーチンを呼び出すタスクに所有されている場合の振舞いは不定となります。

効果

これらのルーチンは、そのルーチンを実行しているタスクの実行をサスペンドしないことを除いて、omp_set_lock と omp_set_nest_lock と同じ方法でロックの設定を試みます。
単純ロックに対するロックの設定が成功した場合に omp_test_lock ルーチンは *true* を返し、それ以外の場合は *false* を返します。

ネスト可能なロックに対するロックの設定が成功した場合に omp_test_nest_lock ルーチンは新しいネスト数を返し、それ以外の場合はゼロを返します。

３．４．時間ルーチン

このセクションでは可搬的な wall clock timer を説明します。

- omp_get_wtime ルーチン
- omp_get_wtick ルーチン

３．４．１．omp_get_wtime

概要

omp_get_wtime ルーチンは秒単位で wall clock の経過時間を返します。

書式

C/C++

```
double omp_get_wtime (void);
```

C/C++

Fortran

```
double precision function omp_get_wtime ( )
```

Fortran

結合

omp_get_wtime リージョンに対する結合スレッドセットは、このリージョンに遭遇したスレッドです。このルーチンの返却値は任意のスレッド間で一致している保証はありません。

効果

omp_get_wtime ルーチンは、”過去のある時点”から経過した wall clock の時間と等しい秒数を返します。実際の ”過去のある時点”をいつにするかは任意ですが、アプリケーションプログラムの実行中に変わらないことは保証されています。返却された時間は”スレッドごとの時間”です。そのため、これらの時間はアプリケーションを実行しているすべてのスレッドで一致することは要求されていません。

注：このルーチンは、以下に示す例のように経過時間を計るために使用すると予想されます。

C/C++

```
double start ;
double end ;
start = omp_get_wtime( ) ;
 ・・・計測される処理　・・・
end = omp_get_wtime( ) ;
printf(“Work took %f seconds¥n”, end – start) ;
```

C/C++

Fortran

```
DOUBLE PRECISION START, END
START = omp_get_wtime( )
・・・　計測される処理　・・・
END = omp_get_wtime( )
PRINT *, “Work took”, END – START, “seconds”
```

Fortran

３．４．２．omp_get_wtick

概要

omp_get_wtick ルーチンは、omp_get_wtime が使うタイマーの精度を返します。

書式

C/C++

```
double omp_get_wtick (void);
```

C/C++
Fortran

```
double precision function omp_get_wtick ( )
```

Fortran

結合

omp_get_wtick リージョンに対する結合スレッドセットは、そのリージョンに遭遇したスレッドです。このルーチンの返却値が任意のスレッド間で一致する保証はありません。

効果

omp_get_wtick ルーチンは、omp_get_wtime が使用するタイマーの連続する時計の刻みの間の秒数と同じ値を返します。

第４章

環境変数

この章では、OpenMP プログラムの実行に影響する内部制御変数を設定する OpenMP の環境変数について説明します。（セクション 2.3 を参照してください。）
環境変数の名前は大文字でなければなりません。環境変数に指定する値では大文字と小文字の区別はなく、前後に空白があってもかまいません。プログラムが開始した後の環境変数の変更は、プログラム自身による変更であっても、OpenMP の実装によって無視されます。しかし、適切な指示文の指示節や OpenMP API ルーチンを使用することによって、一部の内部制御変数の設定を OpenMP プログラムの実行中に変更することができます。
環境変数には、以下があります。

- OMP_SCHEDULE は、runtime スケジュールタイプとチャンクサイズを制御する *run-sched-var* 内部制御変数を設定します。この変数は、有効な OpenMP スケジュールタイプ（static、dynamic、guided、および auto）のいずれかを設定することができます。
- OMP_NUM_THREADS は、parallel リージョンで使用するスレッドの数を制御する *nthreads-var* 内部制御変数を設定します。
- OMP_DYNAMIC は、parallel リージョンで使用するスレッドの動的調整の機能を制御するための dyn-var 内部制御変数を設定します。
- OMP_NESTED は、ネスト並列機能の有効・無効を制御する *nest-var* 内部制御変数を設定します。
- OMP_STACKSIZE は、OpenMP の実装が生成するスレッドのスタックサイズを指定する *stacksize-var* 内部制御変数を設定します。
- OMP_WAIT_POLICY は、ウェイトしているスレッドに期待する振舞いを *wait-policy-var* 内部制御変数に設定します。
- OMP_MAX_ACTIVE_LEVELS は、ネストした活動状態の parallel リージョンの最大数を制御する *max-active-levels-var* 内部制御変数を設定します。
- OMP_THREAD_LIMIT は、OpenMP プログラムを実行するスレッドの最大数を制御する thread-limit-var 内部制御変数を設定します。

この章の例は、Unix C shell（csh）環境でこれらの変数を設定する方法の説明です。Korn shell（ksh）と DOS 環境においては、以下のようになります。

- csh

```
setenv OMP_SCHEDULE “dynamic”
```

・ ksh

```
export OMP_SCHEDULE=”dynamic”
```

・ DOS

```
set OMP_SCHEDULE=dynamic
```

４．１．OMP_SCHEDULE

OMP_SCHEDULE 環境変数は、*run-sched-var* 内部制御変数にその値を設定することにより、スケジュールタイプが runtime であるすべてのループ指示文のスケジュールタイプとチャンクサイズを制御します。
環境変数の値は以下の形式です。

type[,*chunk*]

・ *type* は、**static**、**dynamic**、**guided**、**auto** のうちの１つです。
・ チャンクサイズを指定する、省略可能な *chunk* は正の整数です。
chunk が指定されている場合は、”,”の前後に空白があってもかまいません。スケジュールタイプの詳細な説明は、セクション 2.5.1 を参照してください。
OMP_SCHEDULE の値が上記の形式に従っていない場合のプログラムの振舞いは、実装依存となります。
実装固有のスケジュールを OMP_SCHEDULE で指定することはできません。このようなスケジュールは、omp_set_schedule の呼び出しによる指定のみが可能です。詳しくは、セクション 3.2.11 を参照してください。

例：

```
setenv OMP_SCHEDULE “guided,4”
setenv OMP_SCHEDULE “dynamic”
```

相互参照

- *run-sched-var* 内部制御変数については、セクション 2.3 を参照してください。
- ループ構文については、セクション 2.5.1 を参照してください。
- パラレルループ構文については、セクション 2.6.1 を参照してください。
- omp_set_schedule ルーチンについては、セクション 3.2.11 を参照してください。
- omp_get_schedule ルーチンについては、セクション 3.2.12 を参照してください。

4．2．OMP_ NUM_THREADS

OMP_NUM_THREADS 環境変数は、*nthreads-var* 内部制御変数の初期値にその値を設定することにより、parallel リージョンで使用するスレッドの数を設定します。OMP_NUM_THREADS 環境変数、num_threads 指示節、omp_set_num_threads ライブラリルーチン、およびスレッドの動的調整の相互作用についての包括的な規則については、セクション 2.3 を参照してください。
この環境変数の値は正の整数でなければなりません。OMP_NUM_THREADS で指定した値が実装がサポートするスレッド数以上であったり、正の整数でない場合のプログラムの振舞いは実装依存になります。

例：

```
setenv OMP_NUM_THREADS 16
```

相互参照

- *nthreads-var* 内部制御変数については、セクション 2.3 を参照してください。
- num_threads 指示節については、セクション 2.4 を参照してください。
- omp_set_num_threads ルーチンについては、セクション 3.2.1 を参照してください。
- omp_get_num_threads ルーチンについては、セクション 3.2.2 を参照してください。
- omp_get_max_threads ルーチンについては、セクション 3.2.3 を参照してください。
- omp_get_team_size ルーチンについては、セクション 3.2.18 を参照してください。

４．３．OMP_ DYNAMIC

OMP_DYNAMIC 環境変数は、*dyn-var* 内部制御変数の初期値にその値を設定することにより、実行している parallel リージョンで使用するスレッド数の動的調整を制御します。この環境変数の値は true または false でなければなりません。この環境変数が true に設定された場合、OpenMP の実装は、システム資源の使用を最適化するために parallel リージョンで使用するスレッド数を調整できます。この環境変数が false に設定された場合、スレッド数の動的調整は無効になります。OMP_DYNAMIC の値が、true でも false でもない場合のプログラムの振舞いは実装依存となります。

例：

```
setenv OMP_DYNAMIC true
```

相互参照

- *dyn-var* 内部制御変数は、セクション 2.3 を参照してください。
- omp_set_dynamic ルーチンについては、セクション 3.2.7 を参照してください。
- omp_get_dynamic ルーチンについては、セクション 3.2.8 を参照してください。

４．４．OMP_ NESTED

OMP_NESTED 環境変数は、*nest-var* 内部制御変数の初期値にその値を設定することにより、ネスト並列を制御します。この環境変数の値は true または false でなければなりません。この環境変数が true に設定された場合にネスト並列は有効になります。この環境変数が false に設定された場合、ネスト並列は使用できません。OMP_NESTED の値が true でも false でもない場合のプログラムの振舞いは実装依存になります。

例：

```
setenv OMP_NESTED false
```

相互参照

- *nest-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_set_nested ルーチンについては、セクション 3.2.9 を参照してください。
- omp_get_nested ルーチンについては、セクション 3.2.10 を参照してください。

４．５．OMP_ STACKSIZE

OMP_STACKSIZE 環境変数は、*stacksize-var* 内部制御変数にその値を設定することにより、OpenMP の実装が生成したスレッドのスタックサイズを制御します。この環境変数は、初期スレッドのスタックサイズは制御できません。
この環境変数の値は以下の形式です。

size | *size***B** | *size***K** |*size***M** | *size***G**

- *size* は、OpenMP の実装が生成するスレッドのスタックサイズを指定する正の整数です。
- B、K、M、および G は、それぞれ、バイト、キロバイト、メガバイト、およびギガバイトを指定する文字です。このうちの１つを指定する場合、*size* との間に空白があってもかまいません。

size だけが指定され、B、K、M、G のどれも指定されなかった場合、*size* はキロバイトと見なされます。
OMP_STACKSIZE が上記の形式に合っていない、または実装が要求サイズのスタックを提供できない場合のプログラムの振舞いは、実装依存になります。

例：

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE “3000 k ”
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE “ 10 M ”
setenv OMP_STACKSIZE “20 m ”
setenv OMP_STACKSIZE “ 1G”
setenv OMP_STACKSIZE 20000
```

相互参照

・*stacksize-var* 内部制御変数については、セクション 2.3 を参照してください。

4．6．OMP_ WAIT_POLICY

OMP_WAITPOLICY 環境変数は、*wait-policy-var* 内部制御変数にその値を設定することにより、ウェイトしているスレッドの期待する振舞いについて OpenMP の実装にヒントを提供します。準拠している OpenMP の実装は、この環境変数の設定に従わなくてもかまいません。
この環境変数の値は以下の形式です。

```
ACTIVE | PASSIVE
```

ACTIVE は、スレッドがウェイトしている時に、主にアクティブで、プロセッササイクルを消費してウェイトしていることを指定します。例えば、OpenMP の実装はウェイトしているスレッドをスピンさせておくことができます。
PASSIVE は、スレッドがウェイトしている時に、主に非活動状態で、プロセッササイクルを消費しないでウェイトしていることを指定します。例えば、OpenMP の実装はウェイトしているスレッドから他のスレッドにプロセッサの実行スレッドを切り替えたり、ウェイトスレッドをスリープさせたりすることができます。
ACTIVE と PASSIVE の振舞いの詳細は実装依存となります。

例：

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

相互参照

・ *wait-policy-var* 内部制御変数については、セクション 2.3 を参照してください。

４．７．OMP_MAX_ACTIVE_LEVELS

OMP_MAX_ACTIVE_LEVELS 環境変数は、*max-active-levels-var* 内部制御変数の初期値にその値を設定することにより、ネストしている活動状態の parallel リージョンの最大数を制御します。
この環境変数の値は、負でない整数でなければなりません。OMP_MAX_ACTIVE_LEVELS の値が実装でサポートしているるネストした活動状態の並列レベルの最大値より大きい場合や、負の整数であった場合のプログラムの振舞いは実装依存となります。

相互参照

- *max-active-levels-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_set_max_active_levels ルーチンについては、セクション 3.2.14 を参照してください。
- omp_get_max_active_levels ルーチンについては、セクション 3.2.15 を参照してください。

４．８．OMP_ THREAD_LIMIT

OMP_THREAD_LIMIT 環境変数は、thread-limit-var 内部制御変数にその値を設定することにより、OpenMP プログラムを実行する OpenMP スレッドの数を設定します。
この環境変数の値は、正の整数でなければなりません。OMP_THREAD_LIMIT で指定した値が実装でサポートしているスレッド数より大きい場合や、正の整数でない場合のプログラムの振舞いは実装依存となります。

相互参照

- *thread-limit-var* 内部制御変数については、セクション 2.3 を参照してください。
- omp_get_thread_limit ルーチンについては、セクション 3.2.13 を参照してください。

付録 A

プログラム例

以下に、このドキュメントで定義されている構文とルーチンの例を示します。

C/C++

指示文に続く文は、必要なときのみ複文にします。単文の場合は、指示文に続く文をインデントします。

C/C++

A．1．単純なパラレルループ

以下の例は、パラレルループ構文で単純ループを並列にする方法（セクション 2.6.1 を参照してください。）について示しています。ループ繰り返し変数は、デフォルトでプライベートとなります。そのため、private 指示節で明示的に指定する必要はありません。

C/C++

Example A.1.1c

```
void a1(int n, float *a, float *b)
{
      int i;

#pragma omp parallel for
      for (i=1; i<n; i++) /* i is private by default */
            b[i] = (a[i] + a[i-1]) / 2.0;
}
```

C/C++

Fortran

Example A.1.1f

```
      SUBROUTINE A1(N, A, B)

      INTEGER I, N
      REAL B(N), A(N)

!$OMP PARALLEL DO !I is private by default
      DO I=2,N
             B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END PARALLEL DO

      END SUBROUTINE A1
```

Fortran

A．2．OpenMP メモリモデル

以下の例では、Print　１において、ｘの値は、スレッドのタイミングとｘへの代入の実装に依存して、２または５になります。Print１においての値が、５にならないかもしれない理由は、２つあります。１つは、ｘへの代入が実行される前に、Print１が実行されるかもしれません。２つ目の理由は、Print１ がｘへの代入の後に実行されても、フラッシュが代入の後にスレッド０によって実行されていないかもしれないので、スレッド１が値５を表示する保証はありません。
Print１の後のバリアは、すべてのスレッドでの暗黙のフラッシュと１つのスレッド同期を含んでいます。そのため、プログラマは、Print２とPrint３の両方で、値５がプリントされることが保証されます。

Example A.2.1c

```
#include <stdio.h>
#include <omp.h>

int main(){
      int x;

      x = 2;
#pragma omp parallel num_threads(2) shared(x)
  {

      if (omp_get_thread_num() == 0) {
          x = 5;
      } else {
           /* Print 1: the following read of x has a race */
          printf("1: Thread# %d: x = %d¥n", omp_get_thread_num(),x );
      }

#pragma omp barrier

      if (omp_get_thread_num() == 0) {
           /* Print 2 */
          printf("2: Thread# %d: x = %d¥n", omp_get_thread_num(),x );
      } else {
           /* Print 3 */
          printf("3: Thread# %d: x = %d¥n", omp_get_thread_num(),x );
      }
  }
     return 0;
}
```

Fortran

Example A.2.1f

```
      PROGRAM A2
       INCLUDE "omp_lib.h" ! or USE OMP_LIB
       INTEGER X

       X = 2
!$OMP PARALLEL NUM_THREADS(2) SHARED(X)

       IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
           X = 5
       ELSE
       ! PRINT 1: The following read of x has a race
           PRINT *,"1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
       ENDIF

!$OMP BARRIER

       IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
       ! PRINT 2
           PRINT *,"2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
       ELSE
       ! PRINT 3
           PRINT *,"3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
       ENDIF

!$OMP END PARALLEL

      END PROGRAM A2
```

Fortran

以下の例は、なぜ、変数を通じて正しい同期を実行するのが難しいかを示しています。flag の値は、スレッド１では両方のプリントで未定義となり、また、data の値は、２番目のプリントでだけ、明確に定義されます。

Example A.2.2c

```
#include <omp.h>
#include <stdio.h>
int main()
{
int data;
int flag=0;
#pragma omp parallel
{
      if(omp_get_thread_num()==0)
      {
            /* Write to the data buffer that will be read by thread */
            data = 42;
            /* Flush data to thread 1 and strictly order the write to data
               relative to the write to the flag */
            #pragma omp flush(flag, data)
            /* Set flag to release thread 1 */
            flag = 1;
            /* Flush flag to ensure that thread 1 sees the change */
             #pragma omp flush(flag)
      }
      else if(omp_get_thread_num()==1)
      {
            /* Loop until we see the update to the flag */
            #pragma omp flush(flag, data)
            while(flag < 1)
            {
                 #pragma omp flush(flag, data)
             }
            /* Values of flag and data are undefined */
            printf("flag=%d data=%d¥n", flag, data);
            #pragma omp flush(flag, data)
            /* Values data will be 42, value of flag still undefined */
            printf("flag=%d data=%d¥n", flag, data);
      }
}
}
```

Fortran

Example A.2.2f

```
      PROGRAM EXAMPLE
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER DATA
      INTEGER FLAG

!$OMP PARALLEL
       IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
            ! Write to the data buffer that will be read by thread 1
            DATA = 42
            ! Flush DATA to thread 1 and strictly order the write to DATA
            ! relative to the write to the FLAG
            !$OMP FLUSH(FLAG, DATA)
            ! Set FLAG to release thread 1
            FLAG = 1;
             ! Flush FLAG to ensure that thread 1 sees the change */
            !$OMP FLUSH(FLAG)
       ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
              ! Loop until we see the update to the FLAG
             !$OMP FLUSH(FLAG, DATA)
             DO WHILE(FLAG .LT. 1)
                  !$OMP FLUSH(FLAG, DATA)
             ENDDO
             ! Values of FLAG and DATA are undefined
             PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
             !$OMP FLUSH(FLAG, DATA)
             !Values DATA will be 42, value of FLAG still undefined */
             PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
        ENDIF
  !$OMP END PARALLEL
      END
```

Fortran

この例は、なぜ、変数を通じて正しい同期を実行するのが難しいかを示しています。スレッド１とスレッド２の文は、いずれの順番でも実行できます。

C/C++

Example A.2.3c

```
#include <omp.h>
#include <stdio.h>
int main()
{
int flag=0;

#pragma omp parallel
{
      if(omp_get_thread_num()==0)
      {
            /* Set flag to release thread 1 */
            #pragma omp atomic
            flag++;
            /* Flush of flag is implied by the atomic directive */
      }
      else if(omp_get_thread_num()==1)
      {
            /* Loop until we see that flag reaches 1*/
            #pragma omp flush(flag)
            while(flag < 1)
            {
                  #pragma omp flush(flag)
            }
            printf("Thread 1 awoken¥n");
            /* Set flag to release thread 2 */
            #pragma omp atomic
            flag++;
            /* Flush of flag is implied by the atomic directive */
      }
      else if(omp_get_thread_num()==2)
      {
            /* Loop until we see that flag reaches 2 */
            #pragma omp flush(flag)
```

```
            while(flag < 2)
            {
                  #pragma omp flush(flag)
            }
            printf("Thread 2 awoken¥n");
        }
    }
    }
```

C/C++

Fortran

Example A.2.3f

```
      PROGRAM EXAMPLE
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER FLAG

!$OMP PARALLEL
      IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
           ! Set flag to release thread 1
           !$OMP ATOMIC
                FLAG = FLAG + 1
           !Flush of FLAG is implied by the atomic directive
      ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
           ! Loop until we see that FLAG reaches 1
           !$OMP FLUSH(FLAG, DATA)
           DO WHILE(FLAG .LT. 1)
                !$OMP FLUSH(FLAG, DATA)
           ENDDO
           PRINT *, 'Thread 1 awoken'
           ! Set FLAG to release thread 2
           !$OMP ATOMIC
           FLAG = FLAG + 1
           !Flush of FLAG is implied by the atomic directive
      ELSE IF(OMP_GET_THREAD_NUM() .EQ. 2) THEN
           ! Loop until we see that FLAG reaches 2
           !$OMP FLUSH(FLAG, DATA)
           DO WHILE(FLAG .LT. 2)
                !$OMP FLUSH(FLAG, DATA)
           ENDDO
           PRINT *, 'Thread 2 awoken'
      ENDIF
!$OMP END PARALLEL
      END
```

Fortran

A．3．条件付きコンパイル

C/C++

以下の例は、OpenMP のマクロ _OPENMP（セクション 2.2 を参照してください。）を使った条件付きコンパイルついて説明をしています。OpenMP でのコンパイルではマクロ _OPENMP が定義されます。

Example A.3.1c

```
#include <stdio.h>
int main()
{

# ifdef _OPENMP
    printf("Compiled by an OpenMP-compliant implementation.¥n");
# endif

    return 0;
}
```

C/C++

Fortran

以下の例は、条件付きコンパイルの接頭辞（セクション 2.2 を参照してください。）の使用について説明をしています。OpenMP でのコンパイルでは、条件付きコンパイルの接頭辞 !$ を認識して、２つのスペースとして取り扱います。固定形式のソースでは、接頭辞によって指定された文は、６カラム以降から始まらなければなりません。

Example A.3.1f

```
      PROGRAM A3

C234567890
!$    PRINT *, "Compiled by an OpenMP-compliant implementation."

      END PROGRAM A3
```

Fortran

A．4．内部制御変数

セクション 2.3 により、OpenMP の実装は、プログラムの振舞いを制御する内部制御変数があるように動作しなければなりません。この例は、２つの内部制御変数（*nthreads-var* と *max-active-levels-var*）について説明します。*nthreads-var* 内部制御変数は、遭遇した parallel リージョンを実行するスレッドの数を制御します。タスクごとにこの内部制御変数の１つのコピーが作られます。*max-active-levels-var* 内部制御変数は、ネストされた活動状態の parallel リージョンの最大数を制御します。この内部制御変数のコピーは、プログラム全体で１つだけ存在します。
以下の例は、*nthreads-var* 内部制御変数の値は、omp_set_num_threads の呼び出しで変更されます。*nthreads-var* の新しい値は、parallel リージョンを実行し、omp_set_num_threads を呼出した暗黙のタスクだけに適用されます。*max-active-levels-var* 内部制御変数はグローバルです。そのため、すべてのタスクで同じ値となります。

C/C++

Example A.4.1c

```
#include <stdio.h>
#include <omp.h>

int main (void)
{
omp_set_nested(1);
omp_set_max_active_levels(8);
omp_set_dynamic(0);
omp_set_num_threads(2);

#pragma omp parallel
    {
       omp_set_num_threads(3);

       #pragma omp parallel
          {
             omp_set_num_threads(4);
             #pragma omp single
               {
                  /*
                    * The following should print:
                    * Inner: max_act_lev=8, num_thds=3, max_thds=4
```

```
                * Inner: max_act_lev=8, num_thds=3, max_thds=4
                */
               printf ("Inner: max_act_lev=%d, num_thds=%d,
                    max_thds=%d¥n",
               omp_get_max_active_levels(), omp_get_num_threads(),
               omp_get_max_threads());

            }
          }

        #pragma omp barrier
        #pragma omp single
          {
            /*
             * The following should print:
             * Outer: max_act_lev=8, num_thds=2, max_thds=3
            */
           printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d¥n",
                omp_get_max_active_levels(), omp_get_num_threads(),
                omp_get_max_threads());
         }
      }
  }
```

C/C++

Example A.4.1f

```
      program icv
      use omp_lib

      call omp_set_nested(.true.)
      call omp_set_max_active_levels(8)
      call omp_set_dynamic(.false.)
      call omp_set_num_threads(2)

!$omp parallel
      call omp_set_num_threads(3)

!$omp parallel
      call omp_set_num_threads(4)
!$omp single!
!     The following should print:
!     Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
!     Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
      print *, ("Inner: max_act_lev=", omp_get_max_active_levels(),
     & ", num_thds=", omp_get_num_threads(),
     & ", max_thds=", omp_get_max_threads())
!$omp end single
!$omp end parallel

!$omp barrier
!$omp single

!     The following should print:
!     Outer: max_act_lev= 8 , num_thds= 2 , max_thds= 3
      print *, ("Outer: max_act_lev=", omp_get_max_active_levels(),
     & ", num_thds=", omp_get_num_threads(),
     & ", max_thds=", omp_get_max_threads())

!$omp end single
!$omp end parallel
      end
```

A．5．parallel 構文

parallel 構文（セクション 2.4 を参照してください。）は、粒度が大きい並列プログラムで使用することができます。以下の例では、parallel リージョン内のそれぞれのスレッドが、スレッド番号によってグローバル配列 x の分配について決定します。

C/C++

Example A.5.1c

```
#include <omp.h>

void subdomain(float *x, int istart, int ipoints)
{
   int i;

   for (i = 0; i < ipoints; i++)
      x[istart+i] = 123.456;
}

void sub(float *x, int npoints)
{
    int iam, nt, ipoints, istart;

#pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt = omp_get_num_threads();
        ipoints = npoints / nt; /* size of partition */
        istart = iam * ipoints; /* starting array index */
        if (iam == nt-1) /* last thread may do more */
           ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
     }
}

int main()
{
     float array[10000];
```

```
    sub(array, 10000);

    return 0;
}
```

Example A.5.1f

```
      SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
        INTEGER ISTART, IPOINTS
        REAL X(*)
        INTEGER I

        DO 100 I=1,IPOINTS
           X(ISTART+I) = 123.456
100    CONTINUE

      END SUBROUTINE SUBDOMAIN

       SUBROUTINE SUB(X, NPOINTS)
           INCLUDE "omp_lib.h" ! or USE OMP_LIB
           REAL X(*)
           INTEGER NPOINTS
           INTEGER IAM, NT, IPOINTS, ISTART

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)

           IAM = OMP_GET_THREAD_NUM()
           NT = OMP_GET_NUM_THREADS()
           IPOINTS = NPOINTS/NT
           ISTART = IAM * IPOINTS
           IF (IAM .EQ. NT-1) THEN
               IPOINTS = NPOINTS - ISTART
           ENDIF
           CALL SUBDOMAIN(X,ISTART,IPOINTS)

!$OMP END PARALLEL
      END SUBROUTINE SUB

      PROGRAM A5
           REAL ARRAY(10000)
           CALL SUB(ARRAY, 10000)
      END PROGRAM A5
```

A．6．num_threads 指示節

以下の例は、num_threads 指示節（セクション 2.4 を参照してください。）について説明しています。parallel リージョンは、最大 10 個のスレッドで実行されます。

C/C++

Example A.6.1c

```
#include <omp.h>
int main()
{
   omp_set_dynamic(1);

   #pragma omp parallel num_threads(10)
   {
      /* do work here */
   }
   return 0;
}
```

C/C++

Fortran

Example A.6.1f

```
      PROGRAM A6
         INCLUDE "omp_lib.h" ! or USE OMP_LIB
         CALL OMP_SET_DYNAMIC(.TRUE.)

!$OMP    PARALLEL NUM_THREADS(10)
            ! do work here
!$OMP    END PARALLEL
      END PROGRAM A6
```

Fortran

A．7．do 構文に関する Fortran の制限

do 構文内で複数の DO 文は同じ DO 端末文を共有する場合、その do 構文の後ろに end do 指示文があれば、do 指示文は最初（つまり、最も外側）の DO 文だけに指定されることができます。詳しくは、セクション 2.5.1 を参照してください。以下の例は、ループ構文の正しい使用を含んでいます。

Example A.7.1f

```
      SUBROUTINE WORK(I, J)
      INTEGER I,J
      END SUBROUTINE WORK

      SUBROUTINE A7_GOOD()
         INTEGER I, J
         REAL A(1000)

         DO 100 I = 1,10
 !$OMP  DO
         DO 100 J = 1,10
            CALL WORK(I,J)
  100  CONTINUE ! !$OMP ENDDO implied here

 !$OMP  DO
         DO 200 J = 1,10
  200    A(I) = I + 1
 !$OMP  ENDDO

 !$OMP  DO
         DO 300 I = 1,10
            DO 300 J = 1,10
               CALL WORK(I,J)
  300  CONTINUE
 !$OMP  ENDDO
         END SUBROUTINE A7_GOOD
```

以下の例は、end do と対応する do 指示文が、最も外側のループの前にないため、非準拠のプログラムとなります。

Example A.7.2f

```
      SUBROUTINE WORK(I, J)
      INTEGER I,J
      END SUBROUTINE WORK

      SUBROUTINE A7_WRONG
          INTEGER I, J

          DO 100 I = 1,10
!$OMP       DO
            DO 100 J = 1,10
                CALL WORK(I,J)
  100     CONTINUE
!$OMP ENDDO
      END SUBROUTINE A7_WRONG
```

A．8．Fortran のプライベートなループ繰り返し変数

一般的に、ループの繰り返し変数は、do と parallel do 構文内の *do-loop* または parallel 構文内の逐次ループで使用されるとき、プライベートとなります。（セクション 2.5.1 とセクション 2.9.1 を参照してください。）
以下の parallel 構文内の逐次ループの例では、ループの繰り返し変数 I は、プライベートです。

Example A.8.1f

```
SUBROUTINE A8_1(A,N)
INCLUDE "omp_lib.h" ! or USE OMP_LIB
REAL A(*)
INTEGER I, MYOFFSET, N

!$OMP PARALLEL PRIVATE(MYOFFSET)
      MYOFFSET = OMP_GET_THREAD_NUM()*N
      DO I = 1, N
        A(MYOFFSET+I) = FLOAT(I)
      ENDDO
!$OMP END PARALLEL
END SUBROUTINE A8_1
```

以下のような例外的なケースでは、ループ繰り返し変数を共有にすることができます。

Example A.8.2f

```
SUBROUTINE A8_2(A,B,N,I1,I2)
REAL A(*), B(*)
INTEGER I1, I2, N

!$OMP PARALLEL SHARED(A,B,I1,I2)
!$OMP SECTIONS
!$OMP SECTION
     DO I1 = I1, N
        IF (A(I1).NE.0.0) EXIT
     ENDDO
!$OMP SECTION
     DO I2 = I2, N
        IF (B(I2).NE.0.0) EXIT
     ENDDO
!$OMP END SECTIONS
!$OMP SINGLE
    IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, ' ARE ALL ZERO.'
    IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, ' ARE ALL ZERO.'
!$OMP END SINGLE
!$OMP END PARALLEL

END SUBROUTINE A8_2
```

しかし、共有されたループの繰り返し変数の使用は、競合状態を引き起こしやすいことに注意をしてください。

A．9．nowait 指示節

parallel リージョン内に複数の独立のループがある場合、以下の例のように、nowait 指示節（セクション 2.5.1 を参照してください。）で、ループ構文の終わりの暗黙のバリアを避けられます。

C/C++

Example A.9.1c

```
#include <math.h>

void a9(int n, int m, float *a, float *b, float *y, float *z)
{
   int i;
   #pragma omp parallel
   {
   #pragma omp for nowait
      for (i=1; i<n; i++)
         b[i] = (a[i] + a[i-1]) / 2.0;

   #pragma omp for nowait
      for (i=0; i<m; i++)
         y[i] = sqrt(z[i]);
    }
}
```

C/C++

Fortran

Example A.9.1f

```
      SUBROUTINE A9(N, M, A, B, Y, Z)

        INTEGER N, M
        REAL A(*), B(*), Y(*), Z(*)

        INTEGER I

!$OMP PARALLEL

!$OMP DO
        DO I=2,N
           B(I) = (A(I) + A(I-1)) / 2.0
        ENDDO
!$OMP END DO NOWAIT

!$OMP DO
        DO I=1,M
          Y(I) = SQRT(Z(I))
        ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

      END SUBROUTINE A9
```

Fortran

以下の例は、static スケジュールの場合の nowait の使用例を示しています。

C/C++

Example A.9.2c

```
#include <math.h>
void a92(int n, float *a, float *b, float *c, float *y, float *z)
{
   int i;
#pragma omp parallel
   {
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0;
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      z[i] = sqrt(c[i]);
#pragma omp for schedule(static) nowait
   for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
   }
}
```

C/C++

Fortran

Example A.9.2f

```
      SUBROUTINE A92(N, A, B, C, Y, Z)
      INTEGER N
      REAL A(*), B(*), C(*), Y(*), Z(*)
      INTEGER I
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC)
      DO I=1,N
         C(I) = (A(I) + B(I)) / 2.0
      ENDDO
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(STATIC)
      DO I=1,N
        Z(I) = SQRT(C(I))
      ENDDO
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(STATIC)
      DO I=2,N+1
        Y(I) = Z(I-1) + A(I)
      ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
      END SUBROUTINE A92
```

Fortran

A．１０．collapse 指示節

以下の例では、k と j のループは一重化され、それらの繰り返し空間は、現在のチームのすべてのスレッドによって実行されます。

Fortran

A.10.1f

```
      subroutine sub()
      !$omp do collapse(2) private(i,j,k)
        do k = kl, ku, ks
          do j = jl, ju, js
            do i = il, iu, is
              call bar(a,i,j,k)
             enddo
          enddo
        enddo
      !$omp end do
      end subroutine
```

Fortran

次の例では、k と j のループが一重化され、それらの繰り返し空間は、現在のチームのすべてのスレッドによって実行されます。この例では、２　３ がプリントされます。

Fortran

Example A.10.2f

```
      program test
!$omp parallel
!$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
      do k = 1,2
         do j = 1,3
            jlast=j
            klast=k
          enddo
       enddo
 !$omp end do
 !$omp single
          print *, klast, jlast
 !$omp end single
 !$omp end parallel
       end program test
```

Fortran

次の例は、collapse 指示節を指定した ordered 構文の使用を説明しています。両方のループが一重化されるため、ordered 構文は、do 構文に関連付けられたすべてのループの内側になければなりません。１つの繰り返しは、１つ以上の ordered リージョンを実行できないため、collapse（2）指示節がないとこのプログラムは正しくはありません。
このプログラムは、以下をプリントします。

```
0 1 1
0 1 2
0 2 1
1 2 2
1 3 1
1 3 2
```

Fortran

Example A.10.3f

```
        program test
        include 'omp_lib.h'
  !$omp parallel num_threads(2)
  !$omp do collapse(2) ordered private(j,k) schedule(static,3)
        do k = 1,3
          do j = 1,2
  !$omp ordered
                print *, omp_get_thread_num(), k, j
  !$omp end ordered
                call work(a,j,k)
          enddo
        enddo
  !$omp end do
  !$omp end parallel
        end program test
```

Fortran

A．１１．parallel sections 構文

以下の例（セクション 2.5.2 を参照してください。）では、ルーチン *xaxis*、*yaxis*、および *zaxis* は、同時に実行させることができます。最初の section 指示文は、省略ができます。
すべての section 指示文は、parallel sections 構文の中にある必要があります。

C/C++

Example A.11.1c

```
void XAXIS();
void YAXIS();
void ZAXIS();
void a11()
{
   #pragma omp parallel sections
   {
      #pragma omp section
        XAXIS();
      #pragma omp section
        YAXIS();
      #pragma omp section
        ZAXIS();
    }
}
```

C/C++

Fortran

Example A.11.1f

```
      SUBROUTINE A11()
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL XAXIS()
!$OMP SECTION
      CALL YAXIS()
!$OMP SECTION
       CALL ZAXIS()
!$OMP END PARALLEL SECTIONS
      END SUBROUTINE A11
```

Fortran

A．１２．single 構文

次の例では、single 構文（セクション 2.5.3）を示します。この例では、１つだけのスレッドが、それぞれのプログレスメッセージをプリントします。他のすべてのスレッドは、single リージョンをスキップし、チーム内のすべてのスレッドが single 構文の終わりにある（暗黙の）バリアに到達するまで、そのバリアでストップしています。他のスレッドが single リージョンを実行するスレッドを待たないで続行することができる場合は、この例の３番目の single 構文のように、nowait 指示節を指定することができます。ユーザは、どのスレッドが single リージョンを実行するかを仮定してはいけません。

C/C++

Example A.12.1c

```
#include <stdio.h>

void work1() {}
void work2() {}

void a12()
{
   #pragma omp parallel
   {
      #pragma omp single
         printf("Beginning work1.¥n");

      work1();

      #pragma omp single
         printf("Finishing work1.¥n");

      #pragma omp single nowait
         printf("Finished work1 and beginning work2.¥n");

      work2();
    }
 }
```

C/C++

Fortran

Example A.12.1f

```
      SUBROUTINE WORK1()
      END SUBROUTINE WORK1

      SUBROUTINE WORK2()
      END SUBROUTINE WORK2

      PROGRAM A12
$OMP PARALLEL

!$OMP SINGLE
        print *, "Beginning work1."
!$OMP END SINGLE

        CALL WORK1()

!$OMP SINGLE
        print *, "Finishing work1."
!$OMP END SINGLE

!$OMP SINGLE
        print *, "Finished work1 and beginning work2."
!$OMP END SINGLE NOWAIT

        CALL WORK2()

!$OMP END PARALLEL

      END PROGRAM A12
```

Fortran

A．１３．タスク構文

以下の例は、明示的なタスクを使用して、ツリー構造を辿る方法を示します。*traverse* 関数は、指定された異なったタスクを並列に実行させるために、parallel リージョン内から呼び出されなければならないことに注意してください。また、同期指示文がないために、タスク間の実行順番が分からないことにも注意してください。従って、traversal が、逐次コードのように、post order で行われると仮定することは誤りです。

C/C++

Example A.13.1c

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
   if (p->left)
#pragma omp task // p is firstprivate by default
      traverse(p->left);
   if (p->right)
#pragma omp task // p is firstprivate by default
      traverse(p->right);
   process(p);
}
```

C/C++

Fortran

Example A.13.1f

```
      RECURSIVE SUBROUTINE traverse ( P )
          TYPE Node
             TYPE(Node), POINTER :: left, right
          END TYPE Node
          TYPE(Node) :: P
          IF (associated(P%left)) THEN
             !$OMP TASK ! P is firstprivate by default
                 CALL traverse(P%left)
             !$OMP END TASK
          ENDIF
          IF (associated(P%right)) THEN
             !$OMP TASK ! P is firstprivate by default
                 CALL traverse(P%right)
             !$OMP END TASK
          ENDIF
          CALL process ( P )
      END SUBROUTINE
```

Fortran

次の例では、taskwait 指示文を加えることによって、ツリーを post order で辿らせる使い方を示します。この例で、現在のノードを処理する前に、左（left）と右（right）の子が実行したということを確実に仮定することができます。

C/C++

Example A.13.2c

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
            postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
            postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

C/C++

Fortran

Example A.13.2f

```
    RECURSIVE SUBROUTINE traverse ( P )
      TYPE Node
         TYPE(Node), POINTER :: left, right
      END TYPE Node
      TYPE(Node) :: P
      IF (associated(P%left)) THEN
        !$OMP TASK ! P is firstprivate by default
            call traverse(P%left)
        !$OMP END TASK
      ENDIF
      IF (associated(P%right)) THEN
        !$OMP TASK ! P is firstprivate by default
            call traverse(P%right)
        !$OMP END TASK
      ENDIF
      !$OMP TASKWAIT
      CALL process ( P )
    END SUBROUTINE
```

Fortran

次の例は、リンクで繋がれたリストの要素を並列に処理するための task 構文の使い方について示します。ポインタ p はデフォルトで、task 構文では firstprivate になります。そのため、firstprivate 指示節で p を指定する必要はありません。

C/C++

Example A.13.3c

```
typedef struct node node;
struct node {
    int data;
    node * next;
};

void process(node * p)
{
    /* do work here */
}
void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
            {
                node * p = head;
                while (p) {
                    #pragma omp task // p is firstprivate by default
                        process(p);
                    p = p->next;
                }
            }
    }
}
```

C/C++

Fortran

Example A.13.3f

```
MODULE LIST
     TYPE NODE
         INTEGER :: PAYLOAD
         TYPE (NODE), POINTER :: NEXT
     END TYPE NODE
CONTAINS
     SUBROUTINE PROCESS(p)
         TYPE (NODE), POINTER :: P
             ! do work here
     END SUBROUTINE
     SUBROUTINE INCREMENT_LIST_ITEMS (HEAD)
         TYPE (NODE), POINTER :: HEAD
         TYPE (NODE), POINTER :: P
     !$OMP PARALLEL PRIVATE(P)
         !$OMP SINGLE
         P => HEAD
         DO
             !$OMP TASK
                 CALL PROCESS(P) ! P is firstprivate by default
             !$OMP END TASK
             P => P%NEXT
             IF ( .NOT. ASSOCIATED (P) ) EXIT
         END DO
         !$OMP END SINGLE
     !$OMP END PARALLEL
     END SUBROUTINE
END MODULE
```

Fortran

この例は、フィボナッチ数を計算します。この関数の呼出しが、parallel リージョン内の１つのスレッドで遭遇した場合、並列に計算を実行するためにネストされた task リージョンが、生成されます。

C/C++

Example A.13.4c

```
int fib(int n) {
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i)
            i=fib(n-1);
        #pragma omp task shared(j)
            j=fib(n-2);
        #pragma omp taskwait
            return i+j;
    }
}
```

C/C++

Fortran

Example A.13.4f

```
      RECURSIVE INTEGER FUNCTION fib(n)
      INTEGER n, i, j
      IF ( n .LT. 2) THEN
        fib = n
      ELSE
!$OMP TASK SHARED(i)
        i = fib( n-1 )
!$OMP END TASK
!$OMP TASK SHARED(j)
        j = fib( n-2 )
!$OMP END TASK
!$OMP END TASKWAIT
        fib = i+j
      END IF
      END FUNCTION
```

Fortran

注：フィボナッチ数の計算は、再帰を示すためのコンピュータ科学の典型的な例です。しかし、それは効率的でない単純なアルゴリズムを示す典型的な例でもあります。もっと効率的な方法は、整数行列に関しての指数を計算します。説明のために、ここでは典型的な再帰アルゴリズムを使用しました。

以下の例では、１つのタスクが大量のタスクを生成し、並列チーム内の全てのスレッドでそれらのタスクを実行する方法を述べます。タスクを生成していると、割り当てられていないタスクの数が実装の上限に達するかもしれません。そのとき、実装は task 指示文内のタスクスケジューリングポイントで、タスクを生成するループを実行しているスレッドのタスクをサスペンドし、割り当てられていないタスクを実行させることができます。割り当てられていないタスクの数が少なくなった時に、そのスレッドはタスク生成ループの実行を再開できます。

Example A.13.5c

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
#pragma omp parallel
  {
    #pragma omp single
     {
        int i;
        for (i=0; i<LARGE_NUMBER; i++)
            #pragma omp task // i is firstprivate, item is shared
              process(item[i]);
     }
   }
}
```

C/C++

Fortran

Example A.13.5f

```
       real*8 item(10000000)
       integer i
!$omp parallel
!$omp single  ! loop iteration variable i is private
       do i=1,10000000
!$omp task
          ! i is firstprivate, item is shared
          call process(item(i))
!$omp end task
       end do
!$omp end single
!$omp end parallel
       end
```

Fortran

以下の例は、タスクがアンタイドタスク内で生成されることを除いて、前の例と同じです。タスクを生成していると、割り当てられていないタスクの数が実装の上限に達するかもしれません。その場合、実装は task 指示文内のタスクスケジューリングポイントで、タスクを生成するループを実行しているスレッドのタスクをサスペンドし、割り当てられていないタスクの実行を開始させることができます。そのスレッドが長い時間が掛かるタスクの実行を開始した場合、他のスレッドは、その実行が完了する前に他のすべてのタスクの実行を完了するかもしれません。

このケースの場合、ループはアンタイドタスク内にありますので、他の任意のスレッドが、タスクを生成するループの実行を再開できます。一方前の例では、タスクを生成するループは、タイドタスク内にありますので、そのループを実行しているスレッドが実行時間の長いタスクが完了するまで、他のスレッドは何もしないで待つことを強いられます。

C/C++

Example A.13.6c

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main()
#pragma omp parallel
 {
    #pragma omp single
     {
          int i;
          #pragma omp task untied // i is firstprivate, item is shared
          {
             for (i=0; i<LARGE_NUMBER; i++)
                 #pragma omp task
                     process(item[i]);
          }
     }
  }
}
```

C/C++

Fortran

Example A.13.6f

```
       real*8 item(10000000)
!$omp parallel
!$omp single
!$omp task untied ! loop iteration variable i is private
       do i=1,10000000
!$omp task ! i is firstprivate, item is shared
          call process(item(i))
!$omp end task
       end do
!$omp end task
!$omp end single
!$omp end parallel
       end
```

Fortran

次の２つの例は、セクション 2.7.1 で説明したスケジュールの規則が、タスク中のスレッドプライベート変数の使用にどう影響するかを示します。実行しているスレッドが、実行待ち状態で同じスレッドプライベート変数を変更する別のタスクの一部分を実行する場合、スレッドプライベート変数の値は、タスクスケジューリングポイントで変わります。タイドタスク内では、ユーザはタスクスケジューリングポイントがコード内に現れる場所を制御できます。

単一スレッドは、*tp* を変更する両方の task リージョンを実行できます。tp を変更する task リージョンは、どんな順番でも実行できます。そのため、var の結果値は、1 または 2 のどちらの値にもなりえます。

C/C++

Example A.13.7c

```
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp task
    {
        /* do work here */
#pragma omp task
        {
            tp = 1;
            /* do work here */
#pragma omp task
            {
                /* no modification of tp */
            }
            var = tp; //value of tp can be 1 or 2
        }
        tp = 2;
    }
}
```

C/C++

Fortran

Example A.13.7f

```
      module example
      integer tp
!$omp threadprivate(tp)
      integer var
      contains
      subroutine work
      use globals
!$omp task
        ! do work here
!$omp task
        tp = 1
        ! do work here
!$omp task
            ! no modification of tp
!$omp end task
        var = tp      ! value of var can be 1 or 2
!$omp end task
        tp = 2
!$omp end task
      end subroutine
      end module
```

Fortran

この例で、スケジュールの規則は、一つのスレッドが *tp* を変更するタスクリージョンをサスペンドしている間に、同じ *tp* を変更する別のタスクをスケジュールすることを禁止しています。従って、書き込まれた値は、タスクスケジューリングポイントを超えても保持されます。

C/C++

Example A.13.8c

```
#include <omp.h>
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp parallel
    {
        /* do work here */
#pragma omp task
        {
            tp++;
            /* do work here */
#pragma omp task
            {
                /* do work here but don't modify tp */
            }
            var = tp; //Value does not change after write above
        }
    }
}
```

C/C++

Fortran

Example A.13.8f

```
      module example1
      integer tp
!$omp threadprivate(tp)
      integer var
      contains
      subroutine work
!$omp parallel
        ! do work here
!$omp task
        tp = tp + 1
        ! do work here
!$omp task
            ! do work here but don't modify tp
!$omp end task
        var = tp ! value does not change after write above
!$omp end task
!$omp end parallel
      end subroutine
      end module
```

Fortran

以下の２つの例は、セクション 2.7.1 で説明したスケジュールの規則が、タスク内のロックと critical セクションの使い方に影響するかを示しています。タスクスケジューリングポイントを越えてロックされたままである場合、そこで実行されるコード内で同じロックを取得することを試みてはいけません。そうでない場合、デッドロックの可能性があります。

下の例では、タスク１を実行しているスレッドがタスク２の実行を保留しているとします。そのスレッドがタスク３の部分でタスクスケジューリングポイントに遭遇したときに、タスク１をサスペンドさせ、タスク２の実行を開始することができます。その結果、スレッドが critical リージョン 1 に入ろうとしたときにデッドロックします。

C/C++

Example A.13.9c

```
void work()
{
     #pragma omp task
     { //Task 1
          #pragma omp task
          { //Task 2
               #pragma omp critical //Critical region 1
               {/*do work here */ }
          }
          #pragma omp critical //Critical Region 2
          {
               //Capture data for the following task
               #pragma omp task
               { /* do work here */ } //Task 3
          }
     }
}
```

C/C++

Fortran

Example A.13.9f

```
      module example
      contains
      subroutine work
!$omp task
      ! Task 1
!$omp task
      ! Task 2
!$omp critical
      ! Critical region 1
      ! do work here
!$omp end critical
!$omp end task
!$omp critical
      ! Critical region 2
      ! Capture data for the following task
!$omp task
      !Task 3
      ! do work here
!$omp end task
!$omp end critical
!$omp end task
      end subroutine
      end module
```

Fortran

以下は、ロックがタスクスケジューリングポイントを超えて所有されたときの例です。しかし、セクション 2.7.1 で述べたスケジュールの制限によれば、スレッドは、ロックを使っている task リージョンの実行が完了する前に、同じロック変数を使ってロックを獲得する、互いに子孫関係がないタスクの実行を開始することはできません。従って、デッドロックは起こりません。

C/C++

Example A.13.10c

```
#include <omp.h>
void work() {
    omp_lock_t lock;
#pragma omp parallel
    {
        int i;
#pragma omp for
        for (i = 0; i < 100; i++) {
#pragma omp task
            {
                omp_set_lock(&lock); // lock is shared by default in the task
                   // Capture data for the following task
#pragma omp task // Task Scheduling Point 1
                { /* do work here */ }
                omp_unset_lock(&lock);
            }
        }
    }
}
```

C/C++

Fortran

Example A.13.10f

```
      module example
      include 'omp_lib.h'
      integer (kind=omp_lock_kind) lock
      integer i
      contains
      subroutine work
!$omp parallel
      !$omp do
      do i=1,100
          !$omp task
              ! Outer task
               call omp_set_lock(lock) ! lock is shared by default in the task
                    ! Capture data for the following task
               !$omp task    ! Task Scheduling Point 1
                    ! do work here
               !$omp end task
               call omp_unset_lock(lock)
           !$omp end task
       end do
!$omp end parallel
      end subroutine
      end module
```

Fortran

A．１４．workshare 構文

以下は、workshare 構文（セクション 2.5.4 を参照してください。）の例です。以下の例では、workshare は、parallel リージョンを実行するスレッドに処理を分配します。そして、最後の文の後にバリアがあります。workshare のブロック内は、Fortran の実行規則に従うように実装しなければなりません。

Example A.14.1f

```
      SUBROUTINE A14_1(AA, BB, CC, DD, EE, FF, N)
      INTEGER N
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), EE(N,N), FF(N,N)
!$OMP PARALLEL
!$OMP   WORKSHARE
          AA = BB
          CC = DD
          EE = FF
!$OMP   END WORKSHARE
!$OMP END PARALLEL
      END SUBROUTINE A14_1
```

以下の例では、最初の workshare リージョンの最後のバリアが、nowait 指示節で削除されます。CC = DD を行うスレッドは、CC = DD の実行を完了した後、すぐに EE = FF の実行を開始します。

Example A.14.2f

```
      SUBROUTINE A14_2(AA, BB, CC, DD, EE, FF, N)
      INTEGER N
      REAL AA(N,N), BB(N,N), CC(N,N)
      REAL DD(N,N), EE(N,N), FF(N,N)
!$OMP  PARALLEL
!$OMP   WORKSHARE
          AA = BB
          CC = DD
!$OMP   END WORKSHARE NOWAIT
!$OMP   WORKSHARE
           EE = FF
!$OMP   END WORKSHARE
!$OMP  END PARALLEL
      END SUBROUTINE A14_2
```

以下の例は、workshare 構文内での atomic 指示文の使い方を示します。SUM(AA)の計算はワークシェアされますが、R の更新は atomic です。

Example A.14.3f

```
      SUBROUTINE A14_3(AA, BB, CC, DD, N)
      INTEGER N
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
      REAL R
      R=0
!$OMP PARALLEL
!$OMP   WORKSHARE
          AA = BB
!$OMP     ATOMIC
            R = R + SUM(AA)
          CC = DD
!$OMP   END WORKSHARE
!$OMP END PARALLEL
      END SUBROUTINE A14_3
```

Fortran の WHERE および FORALL 文は、制御の部分と実行文の部分からなる複文です。ワークシェアが、これらの複文に適用されたとき、制御と実行文の両方の部分が、ワークシェアされます。以下では、workshare 構文内で WHERE 文を使用した例を示します。
それぞれのタスクを、スレッドは順番に実行します。

```
AA = BB   そして、
CC = DD   そして、
EE .ne. 0   そして、
FF = 1 / EE そして、
GG = HH
```

Example A.14.4f

```
      SUBROUTINE A14_4(AA, BB, CC, DD, EE, FF, GG, HH, N)
      INTEGER N
      REAL AA(N,N), BB(N,N), CC(N,N)
      REAL DD(N,N), EE(N,N), FF(N,N)
      REAL GG(N,N), HH(N,N)
!$OMP PARALLEL
!$OMP   WORKSHARE
          AA = BB
          CC = DD
          WHERE (EE .ne. 0) FF = 1 / EE
          GG = HH
!$OMP   END WORKSHARE
!$OMP END PARALLEL
      END SUBROUTINE A14_4
```

次の例では、共有のスカラ変数への代入が、ワークシェアの中で１つのスレッドによって実行されます。チーム内の他のすべてのスレッドはその終了を待っています。

Example A.14.5f

```
       SUBROUTINE A14_5(AA, BB, CC, DD, N)
       INTEGER N
       REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
       INTEGER SHR
!$OMP PARALLEL SHARED(SHR)
!$OMP   WORKSHARE
          AA = BB
          SHR = 1
          CC = DD * SHR
!$OMP   END WORKSHARE
!$OMP END PARALLEL
       END SUBROUTINE A14_5
```

以下の例は、プライベートスカラ変数への代入を含んでいます。この代入はワークシェアの中で１つのスレッドによって実行されます。他のすべてのスレッドはその終了を待っています。プライベートスカラ変数の値は代入文の後で不定となるため、このプログラムは非準拠となります。

Example A.14.6f

```
      SUBROUTINE A14_6_WRONG(AA, BB, CC, DD, N)
      INTEGER N
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
      INTEGER PRI
!$OMP PARALLEL PRIVATE(PRI)
!$OMP   WORKSHARE
          AA = BB
          PRI = 1
          CC = DD * PRI
!$OMP   END WORKSHARE
!$OMP END PARALLEL
      END SUBROUTINE A14_6_WRONG
```

Fortran の実行規則は、workshare 構文内でも守らなければなりません。以下の例では、このコードが逐次に実行されるか複数のスレッドによる OpenMP プログラム内で実行されるにかかわらず、同じ結果が得られます。

Example A.14.7f

```
       SUBROUTINE A14_7(AA, BB, CC, N)
       INTEGER N
       REAL AA(N), BB(N), CC(N)
!$OMP PARALLEL
!$OMP   WORKSHARE
          AA(1:50) = BB(11:60)
          CC(11:20) = AA(1:10)
!$OMP   END WORKSHARE
!$OMP END PARALLEL
       END SUBROUTINE A14_7
```

A．１５．master 構文

以下では、master 構文（セクション 2.8.1 を参照してください。）の例を示します。この例では、マスターは、何回繰り返しが実行されたかを数えており、それを処理レポートとして出力します。他のスレッドは、ウェイトせずに master リージョンをスキップします。

C/C++

Example A.15.1c

```
#include <stdio.h>

extern float average(float,float,float);
void a15( float* x, float* xold, int n, float tol )
{
     int c, i, toobig;
     float error, y;
     c = 0;
     #pragma omp parallel
     {
          do{
            #pragma omp for private(i)
            for( i = 1; i < n-1; ++i ){
               xold[i] = x[i];
            }
            #pragma omp single
            {
               toobig = 0;
            }
            #pragma omp for private(i,y,error) reduction(+:toobig)
            for( i = 1; i < n-1; ++i ){
               y = x[i];
               x[i] = average( xold[i-1], x[i], xold[i+1] );
               error = y - x[i];
               if( error > tol || error < -tol ) ++toobig;
            }
            #pragma omp master
            {
```

```
            ++c;
            printf( "iteration %d, toobig=%d¥n", c, toobig );
          }
        }while( toobig > 0 );
    }
}
```

Example A.15.1f

```
      SUBROUTINE A15( X, XOLD, N, TOL )
      REAL X(*), XOLD(*), TOL
      INTEGER N
      INTEGER C, I, TOOBIG
      REAL ERROR, Y, AVERAGE
      EXTERNAL AVERAGE
      C = 0
      TOOBIG = 1
!$OMP PARALLEL
        DO WHILE( TOOBIG > 0 )
!$OMP    DO PRIVATE(I)
          DO I = 2, N-1
              XOLD(I) = X(I)
          ENDDO
!$OMP    SINGLE
          TOOBIG = 0
!$OMP    END SINGLE
!$OMP    DO PRIVATE(I,Y,ERROR), REDUCTION(+:TOOBIG)
          DO I = 2, N-1
              Y = X(I)
              X(I) = AVERAGE( XOLD(I-1), X(I), XOLD(I+1) )
              ERROR = Y-X(I)
              IF( ERROR > TOL .OR. ERROR < -TOL ) TOOBIG = TOOBIG+1
          ENDDO
!$OMP    MASTER
          C = C + 1
          PRINT *, 'Iteration ', C, ' TOOBIG=', TOOBIG
!$OMP    END MASTER
        ENDDO
!$OMP END PARALLEL
      END SUBROUTINE A15
```

A．１６．critical 構文

次の例は、幾つかの critical 構文（セクション 2.8.2 を参照してください。）を含んでいます。例では、タスクがキューから外され、その処理を行うキューイングモデルを説明します。複数のスレッドが同じタスクをキューから外すことを防ぐために、キューから外す操作は、critical リージョンで行わなければなりません。この例では、２つのキューは独立しているため、異なった名前である *xaxis* と *yaxis* の critical 構文によって守られます。

C/C++

Example A.16.1c

```
int dequeue(float *a);
void work(int i, float *a);
void a16(float *x, float *y)
{
     int ix_next, iy_next;
     #pragma omp parallel shared(x, y) private(ix_next, iy_next)
     {
          #pragma omp critical (xaxis)
               ix_next = dequeue(x);
          work(ix_next, x);
          #pragma omp critical (yaxis)
               iy_next = dequeue(y);
          work(iy_next, y);
     }
}
```

C/C++

Fortran

Example A.16.1f

```
      SUBROUTINE A16(X, Y)

          REAL X(*), Y(*)
          INTEGER IX_NEXT, IY_NEXT

!$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)

!$OMP CRITICAL(XAXIS)
          CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(XAXIS)
          CALL WORK(IX_NEXT, X)

!$OMP CRITICAL(YAXIS)
          CALL DEQUEUE(IY_NEXT,Y)
!$OMP END CRITICAL(YAXIS)
          CALL WORK(IY_NEXT, Y)

!$OMP END PARALLEL

      END SUBROUTINE A16
```

Fortran

A．１７．critical 構文内のワークシェアリング構文

次の例は、critical 構文の内側でのワークシェアリング構文の使い方を示しています（セクション 2.8.2 を参照してください）。この例は、single リージョンと critical リージョンが直接ネストされていないために、規格に準拠するプログラムとなります。

C/C++

Example A.17.1c

```
void a17()
{
     int i = 1;
     #pragma omp parallel sections
     {
          #pragma omp section
          {
               #pragma omp critical (name)
               {
                    #pragma omp parallel
                    {
                         #pragma omp single
                         {
                              i++;
                         }
                    }
               }
          }
     }
}
```

C/C++

Fortran

Example A.17.1f

```
      SUBROUTINE A17()
      INTEGER I
      I = 1
!$OMP PARALLEL SECTIONS
!$OMP     SECTION
!$OMP         CRITICAL (NAME)
!$OMP             PARALLEL
!$OMP                 SINGLE
                          I = I + 1
!$OMP                 END SINGLE
!$OMP             END PARALLEL
!$OMP         END CRITICAL (NAME)
!$OMP END PARALLEL SECTIONS
      END SUBROUTINE A17
```

Fortran

A．１８．barrier リージョンの結合

結合の規則は、barrier リージョンを最も内側で囲んでいる parallel リージョンと結合させます。（セクション 2.8.3 を参照してください。）
以下の例では、*sub3* の中の barrier リージョンが *sub2* の中の parallel リージョンに結合しているため、メインプログラムにある *sub2* の呼出しは、規格に準拠しています。
barrier リージョンが逐次部分を囲んでいる暗黙の非活動状態の parallel リージョンに結合しているため、メインプログラムにある *sub3* の呼出しも規格に準拠しています。また、sub2 から呼ばれた時の sub3 の中の barrier リージョンは sub1 内で生成されたすべてのスレッドではなく、囲んでいる parallel リージョン内のスレッドのチームだけで同期することに注意してください。

C/C++

Example A.18.1c

```
void work(int n) {}

void sub3(int n)
{
    work(n);
    #pragma omp barrier
    work(n);
}

void sub2(int k)
{
    #pragma omp parallel shared(k)
        sub3(k);
}

void sub1(int n)
{
    int i;
    #pragma omp parallel private(i) shared(n)
    {
        #pragma omp for
        for (i=0; i<n; i++)
```

```
            sub2(i);
        }
    }

    int main()
    {
        sub1(2);
        sub2(2);
        sub3(2);
        return 0;
    }
```

C/C++

Fortran

Example A.18.1f

```
      SUBROUTINE WORK(N)
          INTEGER N
      END SUBROUTINE WORK

      SUBROUTINE SUB3(N)
      INTEGER N
          CALL WORK(N)
!$OMP BARRIER
          CALL WORK(N)
      END SUBROUTINE SUB3

      SUBROUTINE SUB2(K)
      INTEGER K
!$OMP PARALLEL SHARED(K)
          CALL SUB3(K)
!$OMP END PARALLEL
      END SUBROUTINE SUB2

      SUBROUTINE SUB1(N)
      INTEGER N
      INTEGER I
```

```
!$OMP PARALLEL PRIVATE(I) SHARED(N)
!$OMP DO
      DO I = 1, N
          CALL SUB2(I)
      END DO
!$OMP END PARALLEL
      END SUBROUTINE SUB1

      PROGRAM A18
          CALL SUB1(2)
          CALL SUB2(2)
          CALL SUB3(2)
      END PROGRAM A18
```

Fortran

A．１９．atomic 構文

以下は、atomic 構文（セクション 2.8.5 を参照してください。）を使用することによって競合状態（複数のスレッドによる *x* の要素の同時更新）を避ける例です。
この例での atomic 構文を使う利点は、*x* の２つの異なった要素の更新を並列に実行できることです。もし、代わりに critical 構文（セクション 2.8.2 を参照してください。）を使用した場合、x の要素への更新は、全て逐次実行されます（実行順序は保証されません）。
atomic 指示文は、直後に続く文に対してだけ有効であることに注意してください。結果として、この例では、*y* の要素はアトミックには更新されません。

C/C++

Example A.19.1c

```
float work1(int i)
{
    return 1.0 * i;
}

float work2(int i)
{
    return 2.0 * i;
}
void a19(float *x, float *y, int *index, int n)
{
    int i;
    #pragma omp parallel for shared(x, y, index, n)
    for (i=0; i<n; i++) {
        #pragma omp atomic
            x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}

int main()
{
```

```
    float x[1000];
    float y[10000];
    int index[10000];
    int i;
    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i]=0.0;
    }
    for (i = 0; i < 1000; i++)
        x[i] = 0.0;
    a19(x, y, index, 10000);
    return 0;
}
```

C/C++

Fortran

Example A.19.1f

```
      REAL FUNCTION WORK1(I)
          INTEGER I
          WORK1 = 1.0 * I
          RETURN
      END FUNCTION WORK1
      REAL FUNCTION WORK2(I)
          INTEGER I
          WORK2 = 2.0 * I
          RETURN
      END FUNCTION WORK2

      SUBROUTINE SUBA19(X, Y, INDEX, N)
          REAL X(*), Y(*)
          INTEGER INDEX(*), N
          INTEGER I
!$OMP PARALLEL DO SHARED(X, Y, INDEX, N)
          DO I=1,N
!$OMP         ATOMIC
               X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
              Y(I) = Y(I) + WORK2(I)
          ENDDO
```

```
END SUBROUTINE SUBA19
PROGRAM A19
    REAL X(1000), Y(10000)
    INTEGER INDEX(10000)
    INTEGER I
    DO I=1,10000
        INDEX(I) = MOD(I, 1000) + 1
        Y(I) = 0.0
    ENDDO
    DO I = 1,1000
        X(I) = 0.0
   ENDDO
    CALL SUBA19(X, Y, INDEX, 10000)
END PROGRAM A19
```

A．２０．atomic 構文の制限

以下の例は、atomic 構文の制限について説明します。詳しい情報については、セクション 2.8.5 を参照してください。

C/C++

プログラム全体を通して、アトミックな代入文の左辺にあるそれぞれの変数の記憶域の位置へのすべてのアトミックな参照は、一致した型を持つことが要求されます。

C/C++

Fortran

プログラム全体を通して、アトミックな代入文の左辺にあるそれぞれの変数の記憶域の位置へのすべてのアトミックな参照は、同じ型と同じ型パラメタを持つことが要求されます。

Fortran

以下は、規格に準拠していない例です。

C/C++

Example A.20.1c

```
void a20_1_wrong ()
{
     union {int n; float x;} u;
#pragma omp parallel
     {
#pragma omp atomic
          u.n++;
#pragma omp atomic
          u.x += 1.0;
/* Incorrect because the atomic constructs reference the same location
    through incompatible types */
     }
}
```

C/C++

Fortran

Example A.20.1f

```
      SUBROUTINE A20_1_WRONG()
          INTEGER:: I
          REAL:: R
          EQUIVALENCE(I,R)
!$OMP PARALLEL
!$OMP ATOMIC
          I = I + 1
!$OMP ATOMIC
          R = R + 1.0
! incorrect because I and R reference the same location
! but have different types
!$OMP END PARALLEL
      END SUBROUTINE A20_1_WRONG
```

Fortran

C/C++

Example A.20.2c

```
void a20_2_wrong ()
{
    int x;
    int *i;
    float *r;
    i = &x;
    r = (float *)&x;
    #pragma omp parallel
    {
#pragma omp atomic
        *i += 1;
#pragma omp atomic
        *r += 1.0;
/* Incorrect because the atomic constructs reference the same location
   through incompatible types */
    }
}
```

C/C++

以下の例は、I と R は同じ位置を参照しているのに違う型を持つため、規格に準拠しません。

Example A.20.2f

```
      SUBROUTINE SUB()
      COMMON /BLK/ R
      REAL R
!$OMP ATOMIC
          R = R + 1.0
      END SUBROUTINE SUB

      SUBROUTINE A20_2_WRONG()
      COMMON /BLK/ I
      INTEGER I
!$OMP PARALLEL
!$OMP ATOMIC
          I = I + 1
      CALL SUB()
!$OMP END PARALLEL
      END SUBROUTINE A20_2_WRONG
```

以下の例は、実装によっては動作するかもしれませんが、これも規格に準拠していないプログラムです。

Example A.20.3f

```
      SUBROUTINE A20_3_WRONG
      INTEGER:: I
      REAL:: R
      EQUIVALENCE(I,R)
!$OMP PARALLEL
!$OMP ATOMIC
          I = I + 1
! incorrect because I and R reference the same location
! but have different types
!$OMP END PARALLEL
!$OMP PARALLEL
!$OMP ATOMIC
       R = R + 1.0
! incorrect because I and R reference the same location
! but have different types
!$OMP END PARALLEL
      END SUBROUTINE A20_3_WRONG
```

Fortran

A．２１．リストを指定する flush 構文

次の例は、二組のスレッド間で指定した変数の点と点（point-to-point）の同期のために flush 構文（セクション 2.8.6 を参照してください。）使用しています。

C/C++

Example A.21.1c

```
#include <omp.h>
#define NUMBER_OF_THREADS 256

int synch[NUMBER_OF_THREADS];
float work[NUMBER_OF_THREADS];
float result[NUMBER_OF_THREADS];

float fn1(int i)
{
    return i*2.0;
}

float fn2(float a, float b)
{
    return a + b;
}

int main()
{
    int iam, neighbor;
#pragma omp parallel private(iam,neighbor) shared(work,synch)
    {
        iam = omp_get_thread_num();
        synch[iam] = 0;
        #pragma omp barrier
        /*Do computation into my portion of work array */
        work[iam] = fn1(iam);

        /* Announce that I am done with my work. The first flush
         * ensures that my work is made visible before synch.
```

```
         * The second flush ensures that synch is made visible.
         */

        #pragma omp flush(work,synch)
        synch[iam] = 1;
        #pragma omp flush(synch)

        /* Wait for neighbor. The first flush ensures that synch is read
         * from memory, rather than from the temporary view of memory.
         * The second flush ensures that work is read from memory, and
         * is done so after the while loop exits.
         */

        neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
        while (synch[neighbor] == 0) {
            #pragma omp flush(synch)
        }

        #pragma omp flush(work,synch)

        /* Read neighbor's values of work array */
        result[iam] = fn2(work[neighbor], work[iam]);
    }

    /* output result here */

    return 0;
}
```

C/C++

Example A.21.1f

```
      REAL FUNCTION FN1(I)
        INTEGER I
        FN1 = I * 2.0
        RETURN
      END FUNCTION FN1

      REAL FUNCTION FN2(A, B)
        REAL A, B
        FN2 = A + B
        RETURN
      END FUNCTION FN2

      PROGRAM A21
        INCLUDE "omp_lib.h" ! or USE OMP_LIB
        INTEGER ISYNC(256)
        REAL WORK(256)
        REAL RESULT(256)
        INTEGER IAM, NEIGHBOR

!$OMP PARALLEL PRIVATE(IAM, NEIGHBOR) SHARED(WORK, ISYNC)
          IAM = OMP_GET_THREAD_NUM() + 1
          ISYNC(IAM) = 0

!$OMP BARRIER

C Do computation into my portion of work array

          WORK(IAM) = FN1(IAM)

C Announce that I am done with my work.
C The first flush ensures that my work is made visible before
C synch. The second flush ensures that synch is made visible.

!$OMP FLUSH(WORK,ISYNC)
          ISYNC(IAM) = 1
```

```
!$OMP FLUSH(ISYNC)

C Wait until neighbor is done. The first flush ensures that
C synch is read from memory, rather than from the temporary
C view of memory. The second flush ensures that work is read
C from memory, and is done so after the while loop exits.

          IF (IAM .EQ. 1) THEN
              NEIGHBOR = OMP_GET_NUM_THREADS()
          ELSE
              NEIGHBOR = IAM - 1
          ENDIF

          DO WHILE (ISYNC(NEIGHBOR) .EQ. 0)
!$OMP FLUSH(ISYNC)
          END DO

!$OMP FLUSH(WORK, ISYNC)
          RESULT(IAM) = FN2(WORK(NEIGHBOR), WORK(IAM))
!$OMP END PARALLEL
      END PROGRAM A21
```

Fortran

A．２２．リストを指定しない flush 構文

以下の例（セクション 2.8.6 を参照してください。）は、リストのない flush 構文に影響される共有変数と影響されない共有オブジェクトを区別します。

C/C++

Example A.22.1c

```
int x, *p = &x;
void f1(int *q)
{
     *q = 1;
     #pragma omp flush
     /* x, p, and *q are flushed */
     /* because they are shared and accessible */
     /* q is not flushed because it is not shared. */
}

void f2(int *q)
{
     #pragma omp barrier
     *q = 2;
     #pragma omp barrier

     /* a barrier implies a flush */
     /* x, p, and *q are flushed */
     /* because they are shared and accessible */
     /* q is not flushed because it is not shared. */
}

int g(int n)
{
     int i = 1, j, sum = 0;
     *p = 1;
     #pragma omp parallel reduction(+: sum) num_threads(10)
     {
          f1(&j);
```

```
        /* i, n and sum were not flushed */
        /* because they were not accessible in f1 */
        /* j was flushed because it was accessible */
        sum += j;

        f2(&j);

        /* i, n, and sum were not flushed */
        /* because they were not accessible in f2 */
        /* j was flushed because it was accessible */
        sum += i + j + *p + n;
    }
    return sum;
}

int main()
{
    int result = g(7);
    return result;
}
```

Example A.22.1f

```
      SUBROUTINE F1(Q)
          COMMON /DATA/ X, P
          INTEGER, TARGET :: X
          INTEGER, POINTER :: P
          INTEGER Q

          Q = 1
!$OMP FLUSH
          ! X, P and Q are flushed
          ! because they are shared and accessible
      END SUBROUTINE F1

      SUBROUTINE F2(Q)
          COMMON /DATA/ X, P
          INTEGER, TARGET :: X
          INTEGER, POINTER :: P
          INTEGER Q

!$OMP BARRIER
              Q = 2
!$OMP BARRIER
              ! a barrier implies a flush
              ! X, P and Q are flushed
              ! because they are shared and accessible
      END SUBROUTINE F2

      INTEGER FUNCTION G(N)
          COMMON /DATA/ X, P
          INTEGER, TARGET :: X
          INTEGER, POINTER :: P
          INTEGER N
          INTEGER I, J, SUM

          I = 1
          SUM = 0
```

```
          P = 1
!$OMP PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
              CALL F1(J)
                   ! I, N and SUM were not flushed
                   ! because they were not accessible in F1
                   ! J was flushed because it was accessible
              SUM = SUM + J

              CALL F2(J)
                   ! I, N, and SUM were not flushed
                   ! because they were not accessible in f2
                   ! J was flushed because it was accessible
              SUM = SUM + I + J + P + N
!$OMP END PARALLEL
          G = SUM
      END FUNCTION G

      PROGRAM A22
          COMMON /DATA/ X, P
          INTEGER, TARGET :: X
          INTEGER, POINTER :: P
          INTEGER RESULT, G

          P => X
          RESULT = G(7)
          PRINT *, RESULT
      END PROGRAM A22
```

Fortran

A．２３．flush、barrier、taskwait 指示文の位置

flush、barrier、および taskwait 指示文は、if 文の直後の副文になることはできないので、以下の例は、規格に準拠していないプログラムです。

Example A.23.1c

```
void a23_wrong()
{
     int a = 1;

#pragma omp parallel
  {
     if (a != 0)
     #pragma omp flush(a)
     /* incorrect as flush cannot be immediate substatement
        of if statement */

     if (a != 0)
     #pragma omp barrier
     /* incorrect as barrier cannot be immediate substatement
        of if statement */

     if (a != 0)
     #pragma omp taskwait
     /* incorrect as taskwait cannot be immediate substatement
        of if statement */
  }
}
```

以下の例は、flush、barrier、および taskwait 指示文が複文の中に囲まれているため、規格に準拠したプログラムです。

Example A.23.2c

```
void a23()
{
     int a = 1;

     #pragma omp parallel
     {
          if (a != 0) {
               #pragma omp flush(a)
          }
          if (a != 0) {
               #pragma omp barrier
          }
          if (a != 0) {
                #pragma omp taskwait
          }
     }
}
```

C/C++

A．２４．ordered 指示節と ordered 構文

ordered 構文（セクション 2.8.7 を参照してください。）は、並列に行われる処理の出力を逐次処理した場合と同じ順序にするために有用です。以下のプログラムは、逐次的な順番でインデックスをプリントアウトします。

C/C++

Example A.24.1c

```
#include <stdio.h>
void work(int k)
{
    #pragma omp ordered
      printf(" %d¥n", k);
}

void a24(int lb, int ub, int stride)
{
    int i;

    #pragma omp parallel for ordered schedule(dynamic)
    for (i=lb; i<ub; i+=stride)
        work(i);
}

int main()
{
    a24(0, 100, 5);
    return 0;
}
```

C/C++

Fortran

Example A.24.1f

```
      SUBROUTINE WORK(K)
          INTEGER k

!$OMP ORDERED
          WRITE(*,*) K
!$OMP END ORDERED

      END SUBROUTINE WORK

      SUBROUTINE SUBA24(LB, UB, STRIDE)
          INTEGER LB, UB, STRIDE
          INTEGER I

!$OMP PARALLEL DO ORDERED SCHEDULE(DYNAMIC)
          DO I=LB,UB,STRIDE
              CALL WORK(I)
          END DO
!$OMP END PARALLEL DO

      END SUBROUTINE SUBA24

      PROGRAM A24
          CALL SUBA24(1,100,5)
      END PROGRAM A24
```

Fortran

ordered 指示節を指定したループリージョン内で複数の ordered 構文を指定することができますす。最初の例は、すべての繰り返しが、２つの ordered リージョンを実行するため、規格に準拠していないプログラムです。ループの 1 つの繰り返しは 2 つ以上の ordered リージョンを実行してはいけません。

C/C++

Example A.24.2c

```
void work(int i) {}
void a24_wrong(int n)
{
     int i;
     #pragma omp for ordered
     for (i=0; i<n; i++) {
     /* incorrect because an iteration may not execute more than one
        ordered region */
     #pragma omp ordered
         work(i);
     #pragma omp ordered
         work(i+1);
     }
}
```

C/C++

Fortran

Example A.24.2f

```
      SUBROUTINE WORK(I)
      INTEGER I
      END SUBROUTINE WORK

      SUBROUTINE A24_WRONG(N)
      INTEGER N
      INTEGER I
!$OMP DO ORDERED
          DO I = 1, N
               ! incorrect because an iteration may not execute more than one
               ! ordered region
!$OMP ORDERED
               CALL WORK(I)
!$OMP END ORDERED
!$OMP ORDERED
               CALL WORK(I+1)
!$OMP END ORDERED
          END DO
      END SUBROUTINE A24_WRONG
```

Fortran

以下の例は、複数の ordered 構文がある規格に準拠したプログラムです。それぞれの繰り返しは、ordered リージョンを一つだけ実行します。

C/C++

```
Example A.24.3c
void work(int i) {}

void a24_good(int n)
{
    int i;

#pragma omp for ordered
    for (i=0; i<n; i++) {
        if (i <= 10) {
            #pragma omp ordered
                work(i);
         }
         if (i > 10) {
             #pragma omp ordered
                 work(i+1);
         }
    }
}
```

C/C++

Fortran

Example A.24.3f

```
      SUBROUTINE A24_GOOD(N)
      INTEGER N

!$OMP DO ORDERED
      DO I = 1,N
          IF (I <= 10) THEN
!$OMP ORDERED
              CALL WORK(I)
!$OMP END ORDERED
          ENDIF

          IF (I > 10) THEN
!$OMP ORDERED
              CALL WORK(I+1)
!$OMP END ORDERED
          ENDIF
      ENDDO
      END SUBROUTINE A24_GOOD
```

Fortran

A．２５．threadprivate 指示文

以下の例は、それぞれのスレッドに異なるカウンターを与える threadprivate 指示文（セクション 2.9.2 を参照してください。）の使い方を示しています。

C/C++

Example A.25.1c

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return(counter);
}
```

C/C++

Fortran

Example A.25.1f

```
      INTEGER FUNCTION INCREMENT_COUNTER()
      COMMON/A25_COMMON/COUNTER
!$OMP THREADPRIVATE(/A25_COMMON/)

      COUNTER = COUNTER +1
      INCREMENT_COUNTER = COUNTER
      RETURN
      END FUNCTION INCREMENT_COUNTER
```

Fortran

C/C++

以下の例は、静的変数にスレッドプライベート指示文を使用しています。

Example A.25.2c

```
int increment_counter_2()
{
    static int counter = 0;
    #pragma omp threadprivate(counter)
    counter++;
    return(counter);
}
```

以下の例は、初期化に現れた変数の変更がどのように不定な振舞いを引き起こすのかを示します。また、補助（auxiliary）オブジェクトとコピーコンストラクタを使用することによってこの問題を避ける方法についても示します。

Example A.25.3c

```
class T {
    public:
        int val;
        T (int);
        T (const T&);
};
```

```
T :: T (int v){
    val = v;
}

T :: T (const T& t) {
    val = t.val;
}

void g(T a, T b){
    a.val += b.val;
}

int x = 1;
T a(x);
const T b_aux(x); /* Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)

void f(int n) {
    x++;
    #pragma omp parallel for
    /* In each thread:
     * a is constructed from x (with value 1 or 2?)
     * b is copy-constructed from b_aux
     */
    for (int i=0; i<n; i++) {
        g(a, b); /* Value of a is unspecified. */
    }
}
```

C/C++

以下の例は、threadprivate 指示文の規格に準拠しない使い方と正しい使い方を示します。詳しい説明は、セクション 2.9.2 とセクション 2.9.4.1 を参照してください。

以下の例は、共通ブロックを参照するサブルーチンでローカルにその宣言がされていないため、規格に準拠していません。

Example A.25.2f

```
      MODULE A25_MODULE
          COMMON /T/ A
      END MODULE A25_MODULE

      SUBROUTINE A25_4_WRONG()
          USE A25_MODULE
!$OMP THREADPRIVATE(/T/)
      !non-conforming because /T/ not declared in A25_4_WRONG
      END SUBROUTINE A25_4_WRONG
```

以下の例もまた、共通ブロックを参照するサブルーチンでローカルにその宣言がされていないため、規格に準拠していません。

Example A.25.3f

```
      SUBROUTINE A25_3_WRONG()
      COMMON /T/ A
!$OMP THREADPRIVATE(/T/)
      CONTAINS
          SUBROUTINE A25_3S_WRONG()
!$OMP PARALLEL COPYIN(/T/)
          !non-conforming because /T/ not declared in A25_3S_WRONG
!$OMP END PARALLEL
          END SUBROUTINE A25_3S_WRONG
      END SUBROUTINE A25_3_WRONG
```

以下の例は、前の例を変更して正しく書き換えたプログラムです。

Example A.25.4f

```
      SUBROUTINE A25_4_GOOD()
      COMMON /T/ A
!$OMP THREADPRIVATE(/T/)
      CONTAINS
          SUBROUTINE A25_4S_GOOD()
          COMMON /T/ A
!$OMP THREADPRIVATE(/T/)
!$OMP PARALLEL COPYIN(/T/)
!$OMP END PARALLEL
          END SUBROUTINE A25_4S_GOOD
      END SUBROUTINE A25_4_GOOD
```

以下は、ローカル変数に対するスレッドプライベート指示文の使い方の例です。

Example A.25.5f

```
      PROGRAM A25_5_GOOD
          INTEGER, ALLOCATABLE, SAVE :: A(:)
          INTEGER, POINTER, SAVE :: PTR
          INTEGER, SAVE :: I
          INTEGER, TARGET :: TARG
          LOGICAL :: FIRSTIN = .TRUE.
!$OMP THREADPRIVATE(A, I, PTR)
          ALLOCATE (A(3))
          A = (/1,2,3/)
          PTR => TARG
          I = 5
!$OMP PARALLEL COPYIN(I, PTR)
!$OMP CRITICAL
          IF (FIRSTIN) THEN
              TARG = 4        ! Update target of ptr
              I = I + 10
              IF (ALLOCATED(A)) A = A + 10
              FIRSTIN = .FALSE.
          END IF
          IF (ALLOCATED(A)) THEN
              PRINT *, 'a = ', A
          ELSE
              PRINT *, 'A is not allocated'
          END IF
          PRINT *, 'ptr = ', PTR
          PRINT *, 'i = ', I
          PRINT *
!$OMP END CRITICAL
!$OMP END PARALLEL
      END PROGRAM A25_5_GOOD
```

上のプログラムを２つのスレッドによって実行した場合、以下の２つの出力セットのどちらかがプリントされます。

```
a = 11 12 13
ptr = 4
i = 15

A is not allocated
ptr = 4
i = 5
```

または、

```
A is not allocated
ptr = 4
i = 15

a = 1 2 3
ptr = 4
i = 5
```

以下は、モジュール変数にスレッドプライベートを使用した例です。

Example A.25.6f

```
      MODULE A25_MODULE6
          REAL, POINTER :: WORK(:)
          SAVE WORK
!$OMP THREADPRIVATE(WORK)
      END MODULE A25_MODULE6

      SUBROUTINE SUB1(N)
      USE A25_MODULE6
!$OMP PARALLEL PRIVATE(THE_SUM)
          ALLOCATE(WORK(N))
          CALL SUB2(THE_SUM)
          WRITE(*,*)THE_SUM
!$OMP END PARALLEL
      END SUBROUTINE SUB1

      SUBROUTINE SUB2(THE_SUM)
      USE A25_MODULE6
          WORK(:) = 10
          THE_SUM=SUM(WORK)
      END SUBROUTINE SUB2

      PROGRAM A25_6_GOOD
          N = 10
          CALL SUB1(N)
      END PROGRAM A25_6_GOOD
```

Fortran

C/C++

次の例は、クラス型 *T* に対するスレッドプライベート変数の初期化を示します。*t1* はデフォルトでコンストラクトされ、*t2* は整数型の引数を１つもつコンストラクタでコンストラクトされ、t3 は引数 *f()*でコピーコンストラクトされます。

Example A.25.4c

```
static T t1;
#pragma omp threadprivate(t1)
static T t2( 23 );
#pragma omp threadprivate(t2)
static T t3 = f();
#pragma omp threadprivate(t3)
```

以下の例は、静的クラスメンバーに対するスレッドプライベートの使い方を示しています。静的クラスメンバーに対する threadprivate 指示文は、クラス定義の内側になければなりません。

Example A.25.5c

```
class T {
  public:
      static int i;
#pragma omp threadprivate(i)
};
```

C/C++

C/C++

A．２６．パラレル・ランダム・アクセス・イテレータループ

以下の例は、パラレル・ランダム・アクセス・イテレータループを示しています。

Example A.26.1c

```
#include <vector>
void iterator_example()
{
    std::vector<int> vec(23);
    std::vector<int>::iterator it;
#pragma omp parallel for default(none) shared(vec)
    for (it = vec.begin(); it < vec.end(); it++)
    {
        // do work with *it //
    }
}
```

C/C++

Fortran

A．２７．共通ブロックの shared と private 指示節の Fortran 制限

名前付き共通ブロックが構文の private、firstprivate、または lastprivate 指示節に指定された場合、その共通ブロックのメンバーは、同じ構文の別のデータ共有属性の指示節で宣言してはいけません。以下の例は、これを示しています。詳しくは、セクション 2.9.3 を参照してください。

この例は、規格に準拠しているプログラムです。

Example A.27.1f

```
      SUBROUTINE A27_1_GOOD()
      COMMON /C/ X,Y
      REAL X, Y

!$OMP PARALLEL PRIVATE (/C/)
          ! do work here
!$OMP END PARALLEL

!$OMP PARALLEL SHARED (X,Y)
          ! do work here
!$OMP END PARALLEL
      END SUBROUTINE A27_1_GOOD
```

以下の例も、規格に準拠しているプログラムです。

Example A.27.2f

```
      SUBROUTINE A27_2_GOOD()
      COMMON /C/ X,Y
      REAL X, Y
      INTEGER I
!$OMP PARALLEL
!$OMP DO PRIVATE(/C/)
      DO I=1,1000
          ! do work here
      ENDDO
!$OMP END DO
!
!$OMP DO PRIVATE(X)
      DO I=1,1000
          ! do work here
      ENDDO
!$OMP END DO
!$OMP END PARALLEL
      END SUBROUTINE A27_2_GOOD
```

以下の例も、規格に準拠しているプログラムです。

Example A.27.3f

```
      SUBROUTINE A27_3_GOOD()
      COMMON /C/ X,Y

!$OMP PARALLEL PRIVATE (/C/)
          ! do work here
!$OMP END PARALLEL

!$OMP PARALLEL SHARED (/C/)
          ! do work here
!$OMP END PARALLEL
      END SUBROUTINE A27_3_GOOD
```

以下の例は、ｘがｃの構成要素のため、規格に準拠していまぜん。

Example A.27.4f

```
      SUBROUTINE A27_4_WRONG()
      COMMON /C/ X,Y
! Incorrect because X is a constituent element of C
!$OMP PARALLEL PRIVATE(/C/), SHARED(X)
          ! do work here
!$OMP END PARALLEL
      END SUBROUTINE A27_4_WRONG
```

以下の例は、共通ブロックを共有とプライベートの両方には宣言できないため、規格に準拠していまぜん。

Example A.27.5f

```
      SUBROUTINE A27_5_WRONG()
      COMMON /C/ X,Y
! Incorrect: common block C cannot be declared both
! shared and private
!$OMP PARALLEL PRIVATE (/C/), SHARED(/C/)
          ! do work here
!$OMP END PARALLEL
      END SUBROUTINE A27_5_WRONG
```

A．２８．default（none）指示節

以下の例は、default（none）指示節によって影響がある変数とそうでない変数とを区別しています。default 指示節の詳細は、セクション 2.9.3.1 を参照してください。

C/C++

Example A.28.1c

```
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void a28(int a) {
    const int c = 1;
    int i = 0;
    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_threads();
            /* O.K. - j is declared within parallel region */
        a = z[j]; /* O.K. - a is listed in private clause */
                  /*      - z is listed in shared clause */
        x = c;    /* O.K. - x is threadprivate */
                  /*      - c has const-qualified type */
        z[i] = y; /* Error - cannot reference i or y here */

        #pragma omp for firstprivate(y)
        for (i=0; i<10 ; i++) {
            z[i] = y; /* O.K. - i is the loop iteration variable */
                      /* - y is listed in firstprivate clause */
        }

        z[i] = y; /* Error - cannot reference i or y here */
    }
}
```

C/C++

Example A.28.1f

```
      SUBROUTINE A28(A)
      INCLUDE "omp_lib.h" ! or USE OMP_LIB

      INTEGER A

      INTEGER X, Y, Z(1000)
      COMMON/BLOCKX/X
      COMMON/BLOCKY/Y
      COMMON/BLOCKZ/Z
!$OMP THREADPRIVATE(/BLOCKX/)

      INTEGER I, J
      i = 1

!$OMP PARALLEL DEFAULT(NONE) PRIVATE(A) SHARED(Z) PRIVATE(J)
      J = OMP_GET_NUM_THREADS();
               ! O.K. - J is listed in PRIVATE clause
      A = Z(J) ! O.K. - A is listed in PRIVATE clause
               !      - Z is listed in SHARED clause
      X = 1    ! O.K. - X is THREADPRIVATE
      Z(I) = Y ! Error - cannot reference I or Y here

!$OMP DO firstprivate(y)
      DO I = 1,10
          Z(I) = Y ! O.K. - I is the loop iteration variable
                   ! Y is listed in FIRSTPRIVATE clause
      END DO

      Z(I) = Y ! Error - cannot reference I or Y here
!$OMP END PARALLEL
      END SUBROUTINE A28
```

A．２９．Fortran の共有変数の暗黙のコピーによって引き起こされた競合状態

以下の例は、大きさ引継ぎ配列の仮引数を持つルーチンへ、部分配列である共有変数を実引数として渡しているため、競合状態を含んでいます。（セクション 2.9.3.2 を参照してください。）
部分配列の引数を渡すサブルーチン呼出しでは、コンパイラはサブルーチンを呼び出す前に一時領域に引数をコピーし、サブルーチンから復帰するときには、一時領域からオリジナルの変数にコピーをします。このコピーが parallel リージョン内で競合を引き起こします。

Example A.29.1f

```
      SUBROUTINE A29

      INCLUDE "omp_lib.h" ! or USE OMP_LIB

      REAL A(20)
      INTEGER MYTHREAD

!$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)

      MYTHREAD = OMP_GET_THREAD_NUM()
      IF (MYTHREAD .EQ. 0) THEN
          CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
      ELSE
          A(6:10) = 12
      ENDIF

!$OMP END PARALLEL

      END SUBROUTINE A29

      SUBROUTINE SUB(X)
      REAL X(*)
      X(1:5) = 4
      END SUBROUTINE SUB
```

A．３０．private 指示節

以下の例において、オリジナルの *i* と *j* の値は parallel リージョンから出たところでも保持され、プライベートで指定された *i* と *j* が parallel 構文内で変更されます。private 指示節の詳細は、セクション 2.9.3.3 を参照してください。

C/C++

Example A.30.1c

```
#include <stdio.h>
#include <assert.h>
int main()
{
     int i, j;
     int *ptr_i, *ptr_j;
     i = 1;
     j = 2;
     ptr_i = &i;
     ptr_j = &j;

     #pragma omp parallel private(i) firstprivate(j)
     {
          i = 3;
          j = j + 2;
          assert (*ptr_i == 1 && *ptr_j == 2);
     }

     assert(i == 1 && j == 2);

     return 0;
}
```

C/C++

Fortran

Example A.30.1f

```
      PROGRAM A30
      INTEGER I, J
      I = 1
      J = 2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
      I = 3
      J = J + 2
!$OMP END PARALLEL
      PRINT *, I, J ! I .eq. 1 .and. J .eq. 2
      END PROGRAM A30
```

Fortran

以下の例において、ルーチン f のループ構文中の変数 a のすべての使用は、プライベートで指定された a を参照していますが、ルーチン g での a への参照はプライベートで指定されたものかオリジナルであるかは不定となります。

C/C++

Example A.30.2c

```
int a;
void g(int k) {
    a = k; /* Accessed in the region but outside of the construct;
            * therefore unspecified whether original or private list
            * item is modified. */
}

void f(int n) {
    int a = 0;
    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++) {
        a = i;
        g(a*2); /* Private copy of "a" */
    }
}
```

C/C++

Example A.30.2f

```
      MODULE A30_2
      REAL A
      CONTAINS

      SUBROUTINE G(K)
      REAL K
      A = K ! Accessed in the region but outside of the
            ! construct; therefore unspecified whether
            ! original or private list item is modified.
      END SUBROUTINE G

      SUBROUTINE F(N)
      INTEGER N
      REAL A
      INTEGER I
!$OMP PARALLEL DO PRIVATE(A)
      DO I = 1,N
          A = I
          CALL G(A*2)
      ENDDO
!$OMP END PARALLEL DO
      END SUBROUTINE F
END MODULE A30_2
```

A．３１．再プライベート化

以下の例は、変数の再プライベート化について示しています。（セクション 2.9.3.3 を参照してください。）
プライベート変数は、ネストされた構文で再度プライベート変数の宣言をすることができます。これらの変数は、囲んでいる parallel リージョン内で共有にされていなくてもかまいません。

C/C++

Example A.31.1c

```
#include <assert.h>
void a31()
{
     int i, a;

     #pragma omp parallel private(a)
     {
          a = 1;
          #pragma omp parallel for private(a)
          for (i=0; i<10; i++)
          {
               a = 2;
          }
          assert(a == 1);
     }
}
```

C/C++

Fortran

Example A.31.1f

```
      SUBROUTINE A31()
      INTEGER I, A
!$OMP PARALLEL PRIVATE(A)
      A = 1
!$OMP PARALLEL DO PRIVATE(A)
      DO I = 1, 10
          A = 2
      END DO
!$OMP END PARALLEL DO
      PRINT *, A ! Outer A still has value 1
!$OMP END PARALLEL
      END SUBROUTINE A31
```

Fortran

Fortran

A．３２．private 指示節を関する記憶域の結合についての Fortran の制限

以下の非準拠の例は、記憶域の結合に関する private 指示節の規則が引き起こす結果について示しています。

Example A.32.1f

```
      SUBROUTINE SUB()
      COMMON /BLOCK/ X
      PRINT *,X       ! X is undefined
      END SUBROUTINE SUB

      PROGRAM A32_1
      COMMON /BLOCK/ X
      X = 1.0
!$OMP PARALLEL PRIVATE (X)
      X = 2.0
      CALL SUB()
!$OMP END PARALLEL
      END PROGRAM A32_1
```

Example A.32.2f

```
      PROGRAM A32_2
      COMMON /BLOCK2/ X
      X = 1.0

!$OMP PARALLEL PRIVATE (X)
      X = 2.0
      CALL SUB()
!$OMP END PARALLEL

      CONTAINS

          SUBROUTINE SUB()
          COMMON /BLOCK2/ Y
          PRINT *,X        ! X is undefined
          PRINT *,Y        ! Y is undefined
          END SUBROUTINE SUB
      END PROGRAM A32_2
```

Example A.32.3f

```
      PROGRAM A32_3
      EQUIVALENCE (X,Y)
      X = 1.0

!$OMP PARALLEL PRIVATE(X)
      PRINT *,Y       ! Y is undefined
      Y = 10
      PRINT *,X       ! X is undefined
!$OMP END PARALLEL

      END PROGRAM A32_3
```

Example A.32.4f

```
      PROGRAM A32_4
      INTEGER I, J
      INTEGER A(100), B(100)
      EQUIVALENCE (A(51), B(1))
!$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
      DO I=1,100
          DO J=1,100
              B(J) = J - 1
          ENDDO
          DO J=1,100
              A(J) = J ! B becomes undefined at this point
          ENDDO
          DO J=1,50
              B(J) = B(J) + 1 ! B is undefined
                              ! A becomes undefined at this point
          ENDDO
      ENDDO
!$OMP END PARALLEL DO    ! The LASTPRIVATE write for A has
                         ! undefined results
      PRINT *, B    ! B is undefined since the LASTPRIVATE
                    ! write of A was not defined
      END PROGRAM A32_4
```

Example A.32.5f

```
      SUBROUTINE SUB1(X)
      DIMENSION X(10)

      ! This use of X does not conform to the
      ! specification. It would be legal Fortran 90,
      ! but the OpenMP private directive allows the
      ! compiler to break the sequence association that
      ! A had with the rest of the common block.
          FORALL (I = 1:10) X(I) = I
      END SUBROUTINE SUB1

      PROGRAM A32_5
      COMMON /BLOCK5/ A
      DIMENSION B(10)
      EQUIVALENCE (A,B(1))
      ! the common block has to be at least 10 words
      A = 0
!$OMP PARALLEL PRIVATE(/BLOCK5/)
      ! Without the private clause,
      ! we would be passing a member of a sequence
      ! that is at least ten elements long.
      ! With the private clause, A may no longer be
      ! sequence-associated.
      CALL SUB1(A)
!$OMP MASTER
          PRINT *, A
!$OMP END MASTER

!$OMP END PARALLEL
      END PROGRAM A32_5
```

A．３３．firstprivate 指示節内の C/C++の配列

以下の例は、firstprivate 指示節（セクション 2.9.3.4）内に指定された配列またはポインタ型のサイズと値について示しています。新しいサイズは、ベース言語によって決定するオリジナルの型に基づいて決定します。

この例では、

- A の型は、int 型の２ 要素×２ 要素の２次元配列です。
- B は関数パラメタであるため、B の型は、int 型の n 要素の配列へのポインタへ合わせます。
- C は関数パラメタであるため、C の型は、int 型へのポインタへ合わせます。
- D の型は、int 型の２ 要素×２ 要素の２次元配列です。
- E の型は、int 型の n 要素×n 要素の配列です。

B と E は、可変長配列の型と関係あることに注意してください。

新しい配列は、オリジナル配列のそれぞれの整数要素の値で新しい配列の対応する要素を初期化します。ポインタ型の場合には、オリジナルから新しいポインタ型変数への代入と同じように初期化されます。

Example A.33.1c

```
#include <assert.h>
int A[2][2] = {1, 2, 3, 4};

void f(int n, int B[n][n], int C[])
{
    int D[2][2] = {1, 2, 3, 4};
    int E[n][n];

    assert(n >= 2);
    E[1][1] = 4;

    #pragma omp parallel firstprivate(B, C, D, E)
    {
        assert(sizeof(B) == sizeof(int (*)[n]));
        assert(sizeof(C) == sizeof(int*));
        assert(sizeof(D) == 4 * sizeof(int));
        assert(sizeof(E) == n * n * sizeof(int));

        /* Private B and C have values of original B and C. */
        assert(&B[1][1] == &A[1][1]);
        assert(&C[3] == &A[1][1]);
        assert(D[1][1] == 4);
        assert(E[1][1] == 4);
    }
}

int main() {
    f(2, A, A[0]);
    return 0;
}
```

A．３４．lastprivate 指示節

正しい実行がループの最後の繰り返しで変数に割り当てる値に依存することがあります。そのようなプログラムは、変数の値が逐次にループを実行した時と同じになるように、lastprivate 指示節（セクション 2.9.3.5 を参照してください。）にそのような変数を全て指定しなければなりません。

C/C++

Example A.34.1c

```
void a34 (int n, float *a, float *b)
{
     int i;
     #pragma omp parallel
     {
          #pragma omp for lastprivate(i)
          for (i=0; i<n-1; i++)
               a[i] = b[i] + b[i+1];
     }
     a[i]=b[i]; /* i == n-1 here */
}
```

C/C++

Fortran

Example A.34.1f

```
      SUBROUTINE A34(N, A, B)
      INTEGER N
      REAL A(*), B(*)
      INTEGER I
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
      DO I=1,N-1
          A(I) = B(I) + B(I+1)
      ENDDO
!$OMP END PARALLEL
      A(I) = B(I) ! I has the value of N here
      END SUBROUTINE A34
```

Fortran

A．３５．reduction 指示節

次の例は、reduction 指示節（セクション 2.9.3.6 を参照してください。）について示しています。

C/C++

Example A.35.1c

```
void a35_1(float *x, int *y, int n)
{
    int i, b;
    float a;
    a = 0.0;
    b = 0;

    #pragma omp parallel for private(i) shared(x, y, n) ¥
                        reduction(+:a) reduction(^:b)
    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
    }
}
```

C/C++

Fortran

Example A.35.1f

```
      SUBROUTINE A35_1(A, B, X, Y, N)
      INTEGER N
      REAL X(*), Y(*), A, B
!$OMP PARALLEL DO PRIVATE(I) SHARED(X, N) REDUCTION(+:A)
!$OMP& REDUCTION(MIN:B)
      DO I=1,N
          A = A + X(I)
          B = MIN(B, Y(I))
! Note that some reductions can be expressed in
! other forms. For example, the MIN could be expressed as
! IF (B > Y(I)) B = Y(I)
      END DO
      END SUBROUTINE A35_1
```

Fortran

前の例の一般的な実装では、以下のように記述されたかのように扱います。

C/C++

Example A.35.2c

```
void a35_2(float *x, int *y, int n)
{
    int i, b, b_p;
    float a, a_p;

    a = 0.0;
    b = 0;
#pragma omp parallel shared(a, b, x, y, n) ¥
                            private(a_p, b_p)
    {
        a_p = 0.0;
        b_p = 0;
#pragma omp for private(i)
        for (i=0; i<n; i++) {
            a_p += x[i];
            b_p ^= y[i];
        }
#pragma omp critical
        {
            a += a_p;
            b ^= b_p;
        }
    }
}
```

C/C++

Example A.35.2f

```
      SUBROUTINE A35_2 (A, B, X, Y, N)
      INTEGER N
      REAL X(*), Y(*), A, B, A_P, B_P

!$OMP PARALLEL SHARED(X, Y, N, A, B) PRIVATE(A_P, B_P)
      A_P = 0.0
      B_P = HUGE(B_P)

!$OMP DO PRIVATE(I)
      DO I=1,N
          A_P = A_P + X(I)
          B_P = MIN(B_P, Y(I))
      ENDDO
!$OMP END DO

!$OMP CRITICAL
      A = A + A_P
      B = MIN(B, B_P)
!$OMP END CRITICAL

!$OMP END PARALLEL
      END SUBROUTINE A35_2
```

以下のプログラムでは、リダクションの種類に組込み手続き名 MAX を指定していますが、その名前は MAX という名前の変数に再定義されています。そのため、このプログラムは規格に準拠していません。

Example A.35.3f

```
      PROGRAM A35_3_WRONG
      MAX = HUGE(0)
      M = 0
!$OMP PARALLEL DO REDUCTION(MAX: M)   ! MAX is no longer the
                                      ! intrinsic so this
                                      ! is non-conforming
      DO I = 1, 100
          CALL SUB(M,I)
      END DO
      END PROGRAM A35_3_WRONG

      SUBROUTINE SUB(M,I)
      M = MAX(M,I)
      END SUBROUTINE SUB
```

以下の規格準拠のプログラムは、組込み関数 MAX の名前が REN に変更されていますが、組込み手続き MAX を使用してリダクションを実行します。

Example A.35.4f

```
      MODULE M
          INTRINSIC MAX
      END MODULE M

      PROGRAM A35_4
      USE M, REN => MAX
      N = 0
!$OMP PARALLEL DO REDUCTION(REN: N)     ! still does MAX
      DO I = 1, 100
          N = MAX(N,I)
      END DO
      END PROGRAM A35_4
```

以下の規格に準拠しているプログラムは、組込み関数 MAX の名前が MIN に変更されでいますが、組込み手続き MAX を使用してリダクションを実行します。

Example A.35.5f

```
      MODULE MOD
          INTRINSIC MAX, MIN
      END MODULE MOD

      PROGRAM A35_5
      USE MOD, MIN=>MAX, MAX=>MIN
      REAL :: R
      R = -HUGE(0.0)
!$OMP PARALLEL DO REDUCTION(MIN: R) ! still does MAX
      DO I = 1, 1000
          R = MIN(R, SIN(REAL(I)))
      END DO
      PRINT *, R
      END PROGRAM A35_5
```

Fortran

次のプログラムは、規格に準拠していません。なぜなら、オリジナルである”a”の初期化（a=0）と、for ループの中でリダクション計算の結果である”a”の更新が同期していないためです。そのため、この例は”a”の誤った値を出力する可能性があります。
この問題を回避するためには、オリジナルである”a”の初期化は、リダクション指示節の結果である”a”を更新する前に完了しなければなりません。これは、代入文 a=0 の後に明示的なバリアを加えることや、（暗黙のバリアを持つ）single 指示文の中で代入文 a=0 を実行すること、または parallel リージョンの開始前に”a”を初期化することによって実現することができます。

C/C++

Example A.35.3c

```
#include <stdio.h>
int main(void)
{
      int a, i;
      #pragma omp parallel shared(a) private(i)
      {
            #pragma omp master
            a = 0;
            // To avoid race conditions, add barrier here.
            #pragma omp for reduction(+:a)
            for (i = 0; I < 10; i++) {
                  a += i;
            }
            #pragma omp single
            printf (“Sum is %d¥n”, a);
      }
}
```

C/C++

Fortran

Example A.35.6f

```
      INTEGER A, I
!$OMP PARALLEL SHARED(A) PRIVATE(I)

!$OMP MASTER
      A = 0
!$OMP END MASTER
      ! To avoid race conditions, add a barrier here.
!$OMP DO REDUCTION(+:A)
      DO I= 0, 9
         A = A + I
      END DO

!$OMP SINGLE
      PRINT *, "Sum is ", A
!$OMP END SINGLE

!$OMP END PARALLEL
      END
```

Fortran

A．３６．copyin 指示節

copyin 指示節（セクション 2.9.4.1 を参照してください。）は、parallel リージョンの入口で、スレッドプライベートデータを初期化するために使用します。マスタースレッド内のスレッドプライベート変数の値が、チームの他のメンバーのスレッドプライベート変数にコピーされます。

C/C++

Example A.36.1c

```
#include <stdlib.h>
float* work;
int size;
float tol;

#pragma omp threadprivate(work,size,tol)

void build()
{
      int i;
      work = (float*)malloc( sizeof(float)*size);
      for ( i = 0, i < size, ++i) work(i) = tol;
}
void a36( float t, int n )
{
    tol = t;
    size = n;
#pragma omp parallel copyin(tol,size)
    {
        build();
    }
}
```

C/C++

Fortran

Example A.36.1f

```
      MODULE M
          REAL, POINTER, SAVE :: WORK(:)
          INTEGER :: SIZE
          REAL :: TOL
!$OMP THREADPRIVATE(WORK,SIZE,TOL)
      END MODULE M

      SUBROUTINE A36( T, N )
      USE M
      REAL :: T
      INTEGER :: N
      TOL = T
      SIZE = N
!$OMP PARALLEL COPYIN(TOL,SIZE)
      CALL BUILD
!$OMP END PARALLEL
      END SUBROUTINE A36

      SUBROUTINE BUILD
      USE M
      ALLOCATE(WORK(SIZE))
      WORK = TOL
      END SUBROUTINE BUILD
```

Fortran

A．３７．copyprivate 指示節

copyprivate 指示節（セクション 2.9.4.2 を参照してください。）は、あるスレッドが取得した値を、他の全てのスレッドのプライベート変数に設定するために使用されます。この例では、ルーチンが逐次部分から呼び出された場合でも、指示文の存在が動作に影響することはありません。parallel リージョンから呼び出された場合には、a と b が関連付けられた実引数はプライベートでなければなりません。入力ルーチンが１つのスレッドによって実行された後、すべてのスレッドのプライベート変数 a、b、x、および y が、読まれた値によって定義されるまで、構文を抜けるスレッドはありません。

C/C++

Example A.37.1c

```
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b ) {
    #pragma omp single copyprivate(a,b,x,y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
```

C/C++

Fortran

Example A.37.1f

```
      SUBROUTINE INIT(A,B)
      REAL A, B
      COMMON /XY/ X,Y
!$OMP THREADPRIVATE (/XY/)

!$OMP SINGLE
         READ (11) A,B,X,Y
!$OMP END SINGLE COPYPRIVATE (A,B,/XY/)

      END SUBROUTINE INIT
```

Fortran

前の例と異なり、入力は特定のスレッド、例えばマスタースレッドによって実行しなければならないとします。この場合、copyprivate 指示節を他のスレッドへの値の書き込みに使用することはできません。しかし、一時的な共有変数にアクセスするために使用することができます。

C/C++

Example A.37.2c

```
#include <stdio.h>
#include <stdlib.h>

float read_next( ) {
    float * tmp;
    float return_val;

    #pragma omp single copyprivate(tmp)
    {
        tmp = (float *) malloc(sizeof(float));
     } /* copies the pointer only */

    #pragma omp master
    {
        scanf("%f", tmp);
    }

    #pragma omp barrier
    return_val = *tmp;
    #pragma omp barrier

    #pragma omp single nowait
    {
        free(tmp);
    }
    return return_val;
}
```

C/C++

Fortran

Example A.37.2f

```
      REAL FUNCTION READ_NEXT()
      REAL, POINTER :: TMP

!$OMP SINGLE
      ALLOCATE (TMP)
!$OMP END SINGLE COPYPRIVATE (TMP)     !copies the pointer only

!$OMP MASTER
      READ (11) TMP
!$OMP END MASTER

!$OMP BARRIER
      READ_NEXT = TMP
!$OMP BARRIER

!$OMP SINGLE
      DEALLOCATE (TMP)
!$OMP END SINGLE NOWAIT
      END FUNCTION READ_NEXT
```

Fortran

parallel リージョン内で必要なロック変数の数は、parallel リージョンに入る前に簡単には決定できないと想定します。copyprivate 指示節は、parallel リージョン内で割り付けられた共有のロック変数へアクセスするために使うことができます。

C/C++

Example A.37.3c

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

omp_lock_t *new_lock()
{
    omp_lock_t *lock_ptr;
    #pragma omp single copyprivate(lock_ptr)
    {
        lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
        omp_init_lock( lock_ptr );
    }
    return lock_ptr;
}
```

C/C++

Fortran

Example A.37.3f

```
      FUNCTION NEW_LOCK()
      USE OMP_LIB ! or INCLUDE "omp_lib.h"
      INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK

!$OMP SINGLE
      ALLOCATE(NEW_LOCK)
      CALL OMP_INIT_LOCK(NEW_LOCK)
!$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
      END FUNCTION NEW_LOCK
```

割付け可能属性を持つ変数への copyprivate 指示節の効果は、ポインタ属性を持つ変数への効果とは異なります。

Example A.37.4f

```
      SUBROUTINE S(N)
      INTEGER N
      REAL, DIMENSION(:), ALLOCATABLE :: A
      REAL, DIMENSION(:), POINTER :: B

      ALLOCATE (A(N))

!$OMP SINGLE
      ALLOCATE (B(N))
      READ (11) A,B
!$OMP END SINGLE COPYPRIVATE(A,B)
      ! Variable A is private and is
      ! assigned the same value in each thread
      ! Variable B is shared

!$OMP BARRIER
!$OMP SINGLE
      DEALLOCATE (B)
!$OMP END SINGLE NOWAIT
      END SUBROUTINE S
```

Fortran

A．38．ネストループ構文

以下のネストループ構文（セクション 2.10 を参照してください。）の例は、内側と外側のループリージョンが異なった parallel リージョンに結合しているため、規格に準拠しているプログラムです。

C/C++

Example A.38.1c

```
void work(int i, int j) {}

void good_nesting(int n)
{
     int i, j;
     #pragma omp parallel default(shared)
     {
          #pragma omp for
          for (i=0; i<n; i++) {
               #pragma omp parallel shared(i, n)
               {
                    #pragma omp for
                    for (j=0; j < n; j++)
                         work(i, j);
               }
          }
      }
  }
```

C/C++

Fortran

Example A.38.1f

```
      SUBROUTINE WORK(I, J)
      INTEGER I, J
      END SUBROUTINE WORK

      SUBROUTINE GOOD_NESTING(N)
      INTEGER N
      INTEGER I
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP PARALLEL SHARED(I,N)
!$OMP DO
          DO J = 1, N
              CALL WORK(I,J)
          END DO
!$OMP END PARALLEL
      END DO
!$OMP END PARALLEL
      END SUBROUTINE GOOD_NESTING
```

Fortran

前の例を変更した以下の例も、規格に準拠したプログラムです。

C/C++

Example A.38.2c

```
void work(int i, int j) {}

void work1(int i, int n)
{
    int j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (j=0; j<n; j++)
            work(i, j);
    }
}

void good_nesting2(int n)
{
    int i;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
    }
}
```

C/C++

Fortran

Example A.38.2f

```
      SUBROUTINE WORK(I, J)
      INTEGER I, J
      END SUBROUTINE WORK

      SUBROUTINE WORK1(I, N)
      INTEGER J
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO J = 1, N
          CALL WORK(I,J)
      END DO
!$OMP END PARALLEL
      END SUBROUTINE WORK1

SUBROUTINE GOOD_NESTING2(N)
      INTEGER N
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
          CALL WORK1(I, N)
      END DO
!$OMP END PARALLEL
      END SUBROUTINE GOOD_NESTING2
```

Fortran

A．３９．リージョンのネストの制限

このセクションでの例は、リージョンのネストの規則について示しています。詳しい情報は、セクション 2.10 を参照してください。
以下の例は、内側と外側のループリージョンが直接ネストしているため、規格に準拠していないプログラムです。

C/C++

Example A.39.1c

```
void work(int i, int j) {}

void wrong1(int n)
{
     #pragma omp parallel default(shared)
     {
          int i, j;
          #pragma omp for
          for (i=0; i<n; i++) {
               /* incorrect nesting of loop regions */
               #pragma omp for
               for (j=0; j<n; j++)
                    work(i, j);
           }
      }
}
```

C/C++

Fortran

Example A.39.1f

```
      SUBROUTINE WORK(I, J)
      INTEGER I, J
      END SUBROUTINE WORK

      SUBROUTINE WRONG1(N)
      INTEGER N
      INTEGER I,J
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP DO ! incorrect nesting of loop regions
          DO J = 1, N
              CALL WORK(I,J)
          END DO
      END DO
!$OMP END PARALLEL
      END SUBROUTINE WRONG1
```

Fortran

次の例は、直前の例の親なしなので、これもまた、規格に準拠していないプログラムです。

C/C++

Example A.39.2c

```
void work(int i, int j) {}
void work1(int i, int n)
{
    int j;
    /* incorrect nesting of loop regions */
#pragma omp for
    for (j=0; j<n; j++)
         work(i, j);
}

void wrong2(int n)
{
#pragma omp parallel default(shared)
    {
        int i;
#pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
    }
}
```

C/C++

Fortran

Example A.39.2f

```
      SUBROUTINE WORK1(I,N)
      INTEGER I, N
      INTEGER J
!$OMP DO ! incorrect nesting of loop regions
      DO J = 1, N
          CALL WORK(I,J)
      END DO
      END SUBROUTINE WORK1

      SUBROUTINE WRONG2(N)
      INTEGER N
      INTEGER I
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
          CALL WORK1(I,N)
      END DO
!$OMP END PARALLEL
      END SUBROUTINE WRONG2
```

Fortran

以下の例は、ループリージョンと single リージョンが直接ネストしているため、規格に準拠していないプログラムです。

C/C++

Example A.39.3c

```
void work(int i, int j) {}
void wrong3(int n)
{
#pragma omp parallel default(shared)
    {
        int i;
#pragma omp for
        for (i=0; i<n; i++) {
        /* incorrect nesting of regions */
#pragma omp single
            work(i, 0);
        }
    }
}
```

C/C++

Fortran

Example A.39.3f

```
      SUBROUTINE WRONG3(N)
      INTEGER N
      INTEGER I
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP SINGLE ! incorrect nesting of regions
          CALL WORK(I, 1)
!$OMP END SINGLE
      END DO
!$OMP END PARALLEL
      END SUBROUTINE WRONG3
```

Fortran

以下の例は、barrier リージョンがループリージョンの内側に直接ネストしているために、規格に準拠していないプログラムです。

C/C++

Example A.39.4c

```
void work(int i, int j) {}
void wrong4(int n)
{
#pragma omp parallel default(shared)
    {
        int i;
#pragma omp for
        for (i=0; i<n; i++) {
            work(i, 0);
            /* incorrect nesting of barrier region in a loop region */
#pragma omp barrier
            work(i, 1);
        }
    }
}
```

C/C++

Fortran

Example A.39.4f

```
      SUBROUTINE WRONG4(N)
      INTEGER N
      INTEGER I
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
          CALL WORK(I, 1)
! incorrect nesting of barrier region in a loop region
!$OMP BARRIER
          CALL WORK(I, 2)
      END DO
!$OMP END PARALLEL
      END SUBROUTINE WRONG4
```

Fortran

以下の例は、barrier リージョンが、critical リージョン内に直接ネストしているため、規格に準拠していないプログラムです。もし、これを許すと、ある時点で１つだけのスレッドが critical リージョンに入りますので、デッドロック状態を引き起こします。

C/C++

Example A.39.5c

```
void work(int i, int j) {}
void wrong5(int n)
{
#pragma omp parallel
    {
#pragma omp critical
        {
            work(n, 0);
/* incorrect nesting of barrier region in a critical region */
#pragma omp barrier
            work(n, 1);
        }
    }
}
```

C/C++

Fortran

Example A.39.5f

```
      SUBROUTINE WRONG5(N)
      INTEGER N
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP CRITICAL
      CALL WORK(N,1)
! incorrect nesting of barrier region in a critical region
!$OMP BARRIER
      CALL WORK(N,2)
!$OMP END CRITICAL
!$OMP END PARALLEL
      END SUBROUTINE WRONG5
```

Fortran

以下の例は、barrier リージョンが single リージョン内に直接ネストしているために、規格に準拠していないプログラムです。もしこれを許すと、1 つのスレッドが single リージョンを実行しますので、デッドロック状態を引き起こします。

C/C++

Example A.39.6c

```
void work(int i, int j) {}
void wrong6(int n)
{
#pragma omp parallel
    {
#pragma omp single
        {
            work(n, 0);
/* incorrect nesting of barrier region in a single region */
#pragma omp barrier
            work(n, 1);
        }
    }
}
```

C/C++

Fortran

Example A.39.6f

```
      SUBROUTINE WRONG6(N)
      INTEGER N
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP SINGLE
      CALL WORK(N,1)
! incorrect nesting of barrier region in a single region
!$OMP BARRIER
      CALL WORK(N,2)
!$OMP END SINGLE
!$OMP END PARALLEL
      END SUBROUTINE WRONG6
```

Fortran

A．４０．omp_set_dynamic と omp_set_num_threads ルーチン

プログラムの正しい実行が、固定または事前に指定したスレッドの数に依存する場合があります。スレッド数の動的調整のデフォルトの設定は実装依存であるため、そのようなプログラムはスレッド数の動的調整の機能を停止して、ポータビリティを保証するために明示的にスレッド数を設定することができます。以下の例は、omp_set_dynamic（セクション 3.2.7 を参照してください。）と omp_set_num_threads（セクション 3.2.1 を参照してください。）を使用してそれを実現する方法を示しています。
この例では、プログラムは 16 スレッドによって実行されるときにだけ、正しく実行できます。もし、実装が 16 スレッドの実行をサポートできない場合、この例の振舞いは実装依存となります。parallel リージョンを実行するスレッドの数は、スレッドの動的調整の設定に係わらず、リージョンを実行している間は一定のままです。スレッドの動的調整メカニズムが、parallel リージョンの開始時点で使用するスレッド数を決め、リージョンを実行している間のスレッド数を一定に保ちます。

C/C++

Example A.40.1c

```
#include <omp.h>
#include <stdlib.h>

void do_by_16(float *x, int iam, int ipoints) {}

void a40(float *x, int npoints)
{
     int iam, ipoints;
     omp_set_dynamic(0);
     omp_set_num_threads(16);
#pragma omp parallel shared(x, npoints) private(iam, ipoints)
     {
          if (omp_get_num_threads() != 16)
               abort();
          iam = omp_get_thread_num();
          ipoints = npoints/16;
          do_by_16(x, iam, ipoints);
     }
}
```

C/C++

Fortran

Example A.40.1f

```
      SUBROUTINE DO_BY_16(X, IAM, IPOINTS)
      REAL X(*)
      INTEGER IAM, IPOINTS
      END SUBROUTINE DO_BY_16

      SUBROUTINE SUBA40(X, NPOINTS)
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER NPOINTS
      REAL X(NPOINTS)
      INTEGER IAM, IPOINTS
      CALL OMP_SET_DYNAMIC(.FALSE.)
      CALL OMP_SET_NUM_THREADS(16)
!$OMP PARALLEL SHARED(X,NPOINTS) PRIVATE(IAM, IPOINTS)
      IF (OMP_GET_NUM_THREADS() .NE. 16) THEN
          STOP
      ENDIF
      IAM = OMP_GET_THREAD_NUM()
      IPOINTS = NPOINTS/16
      CALL DO_BY_16(X,IAM,IPOINTS)
!$OMP END PARALLEL
      END SUBROUTINE SUBA40
```

Fortran

A．４１．omp_get_num_threads ルーチン

以下の例において、omp_get_num_threads（セクション 3.2.2 を参照してください。）の呼出しは、コードの逐次部分では１が返却されます。そのため、*np* は常に１となります。parallel リージョンで使われるスレッドの数を決定するためには、この呼出しは、parallel リージョンの中から行う必要があります。

C/C++

Example A.41.1c

```
#include <omp.h>
void work(int i);

void incorrect()
{
     int np, i;
     np = omp_get_num_threads(); /* misplaced */

#pragma omp parallel for schedule(static)
     for (i=0; i < np; i++)
          work(i);
}
```

C/C++

Fortran

Example A.41.1f

```
      SUBROUTINE WORK(I)
      INTEGER I
      I = I + 1
      END SUBROUTINE WORK

      SUBROUTINE INCORRECT()
      INCLUDE "omp_lib.h"          ! or USE OMP_LIB
      INTEGER I, NP
      NP = OMP_GET_NUM_THREADS()     !misplaced: will return 1
!$OMP PARALLEL DO SCHEDULE(STATIC)
      DO I = 0, NP-1
          CALL WORK(I)
      ENDDO
!$OMP END PARALLEL DO
      END SUBROUTINE INCORRECT
```

Fortran

以下の例は、スレッド数の問合せを含まないのこのプログラムの書き直し方を示しています。

C/C++

Example A.41.2c

```
#include <omp.h>
void work(int i);

void correct()
{
    int i;
#pragma omp parallel private(i)
    {
        i = omp_get_thread_num();
        work(i);
    }
}
```

C/C++

Fortran

Example A.41.2f

```
      SUBROUTINE WORK(I)
      INTEGER I
      I = I + 1
      END SUBROUTINE WORK

      SUBROUTINE CORRECT()
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER I
!$OMP PARALLEL PRIVATE(I)
      I = OMP_GET_THREAD_NUM()
      CALL WORK(I)
!$OMP END PARALLEL
      END SUBROUTINE CORRECT
```

Fortran

A．４２．omp_init_lock ルーチン

以下の例は、omp_init_lock（セクション 3.3.1 を参照してください。）を使用することによって、parallel リージョン内のロックの配列を初期化する方法を示しています。

C/C++

Example A.42.1c

```
#include <omp.h>
omp_lock_t *new_locks()
{
    int i;
    omp_lock_t *lock = new omp_lock_t[1000];
#pragma omp parallel for private(i)
    for (i=0; i<1000; i++)
    {
        omp_init_lock(&lock[i]);
    }
    return lock;
}
```

C/C++

Fortran

Example A.42.1f

```
      FUNCTION NEW_LOCKS()
      USE OMP_LIB ! or INCLUDE "omp_lib.h"
      INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS
      INTEGER I
!$OMP PARALLEL DO PRIVATE(I)
      DO I=1,1000
          CALL OMP_INIT_LOCK(NEW_LOCKS(I))
      END DO
!$OMP END PARALLEL DO
      END FUNCTION NEW_LOCKS
```

Fortran

A．４３．ロックの所有権

ロックの所有権は、OpenMP 2.5 から OpenMP 3.0 で変更されました。OpenMP 2.5 では、ロックはスレッドが所有しています。そのため、omp_unset_lock ルーチンによって解放できるロックは、同じスレッドが所有しているロックに限られます。一方、OpenMP 3.0 では、ロックは task リージョンが所有しています。そのため、task リージョン内の omp_unset_lock ルーチンによって解放するロックは、同じ task リージョンが所有しているロックに限られます。
この所有権の変更のため、ロックを使用する際に特別な注意が必要となります。
以下のプログラムは、OpenMP 2.5 では規格に準拠しているプログラムです。なぜなら、parallel リージョン内でロック *lck* を解放するスレッドは、プログラムの逐次部分でロックを取得したスレッドと同じだからです。（parallel リージョンのマスタースレッドと初期スレッドは同じです。）しかし、このプログラムは、OpenMP 3.0 では規格に準拠していません。なぜなら、ロック lck を解放した task リージョンは、ロックを取得した task リージョンと異なるからです。

C/C++

Example A.43.1c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
int main()
{
     int x;
     omp_lock_t lck;
     omp_init_lock (&lck);
     omp_set_lock (&lck);
     x = 0;
#pragma omp parallel shared (x)
     {
#pragma omp master
          {
               x = x + 1;
               omp_unset_lock (&lck);
          }
          /* Some more stuff. */
     }
     omp_destroy_lock (&lck);
}
```

C/C++

Fortran

Example A.43.1f

```
      program lock
      use omp_lib
      integer :: x
      integer (kind=omp_lock_kind) :: lck
      call omp_init_lock (lck)
      call omp_set_lock(lck)
      x = 0

!$omp parallel shared (x)
!$omp master
      x = x + 1
      call omp_unset_lock(lck)
!$omp end master

! Some more stuff.
!$omp end parallel

      call omp_destroy_lock(lck)
      end
```

Fortran

A．４４．単純ロックルーチン

次の例（セクション 3.3 を参照してください。）において、最初の critical セクションに入るのを待っている間、ロックルーチンのためスレッドはアイドル状態になります。しかし、２番目の critical セクションに入るのを待っている間は、スレッドは他の処理を実行します。omp_set_lock ルーチンは、skip の実行開始をブロックしますが、omp_test_lock ルーチンは skip の実行を妨げません。

C/C++

ロックルーチンの引数は omp_lock_t 型でなければなりません。そして、それをフラッシュする必要はありません。

Example A.44.1c

```
#include <stdio.h>
#include <omp.h>

void skip(int i) {}
void work(int i) {}

int main()
{
     omp_lock_t lck;
     int id;
      omp_init_lock(&lck);
#pragma omp parallel shared(lck) private(id)
     {
          id = omp_get_thread_num();
          omp_set_lock(&lck);
          /* only one thread at a time can execute this printf */
          printf("My thread id is %d.¥n", id);
          omp_unset_lock(&lck);
          while (! omp_test_lock(&lck)) {
               skip(id); /* we do not yet have the lock,
                            so we must do something else */
          }
          work(id); /* we now have the lock,
                       and can do the work */
```

```
            omp_unset_lock(&lck);
        }
        omp_destroy_lock(&lck);
        return 0;
    }
```

ロック変数をフラッシュする必要はありません。

Example A.44.1f

```
      SUBROUTINE SKIP(ID)
      END SUBROUTINE SKIP

      SUBROUTINE WORK(ID)
      END SUBROUTINE WORK

      PROGRAM A44
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER(OMP_LOCK_KIND) LCK
      INTEGER ID

      CALL OMP_INIT_LOCK(LCK)

!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
      ID = OMP_GET_THREAD_NUM()
      CALL OMP_SET_LOCK(LCK)
      PRINT *, 'My thread id is ', ID
      CALL OMP_UNSET_LOCK(LCK)

      DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
           CALL SKIP(ID) ! We do not yet have the lock
                         ! so we must do something else
      END DO

      CALL WORK(ID) ! We now have the lock
                    ! and can do the work

      CALL OMP_UNSET_LOCK( LCK )

!$OMP END PARALLEL

      CALL OMP_DESTROY_LOCK( LCK )

      END PROGRAM A44
```

A．４５．ネスト可能なロックルーチン

以下の例（セクション 3.3 を参照してください。）は、構造体全体およびそのメンバーの一つに対して、更新を同期するためにネスト可能なロックをどう使うことができるかを示します。

C/C++

Example A.45.1c

```
#include <omp.h>

typedef struct {
     int a,b;
     omp_nest_lock_t lck; } pair;

int work1();
int work2();
int work3();

void incr_a(pair *p, int a)
{
     /* Called only from incr_pair, no need to lock. */
     p->a += a;
}

void incr_b(pair *p, int b)
{
     /* Called both from incr_pair and elsewhere, */
     /* so need a nestable lock. */
     omp_set_nest_lock(&p->lck);
     p->b += b;
     omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b)
{
     omp_set_nest_lock(&p->lck);
     incr_a(p, a);
```

```
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void a45(pair *p)
{
    #pragma omp parallel sections
    {
        #pragma omp section
            incr_pair(p, work1(), work2());
        #pragma omp section
            incr_b(p, work3());
    }
}
```

C/C++

Fortran

Example A.45.1f

```
      MODULE DATA
          USE OMP_LIB, ONLY: OMP_NEST_LOCK_KIND
          TYPE LOCKED_PAIR
              INTEGER A
              INTEGER B
              INTEGER (OMP_NEST_LOCK_KIND) LCK
          END TYPE
      END MODULE DATA

      SUBROUTINE INCR_A(P, A)
          ! called only from INCR_PAIR, no need to lock
          USE DATA
          TYPE(LOCKED_PAIR) :: P
          INTEGER A
          P%A = P%A + A
      END SUBROUTINE INCR_A

      SUBROUTINE INCR_B(P, B)
          ! called from both INCR_PAIR and elsewhere,
          ! so we need a nestable lock
```

```
        USE OMP_LIB ! or INCLUDE "omp_lib.h"
        USE DATA
        TYPE(LOCKED_PAIR) :: P
        INTEGER B
        CALL OMP_SET_NEST_LOCK(P%LCK)
        P%B = P%B + B
        CALL OMP_UNSET_NEST_LOCK(P%LCK)
    END SUBROUTINE INCR_B

    SUBROUTINE INCR_PAIR(P, A, B)
        USE OMP_LIB ! or INCLUDE "omp_lib.h"
        USE DATA
        TYPE(LOCKED_PAIR) :: P
        INTEGER A
        INTEGER B

        CALL OMP_SET_NEST_LOCK(P%LCK)
        CALL INCR_A(P, A)
        CALL INCR_B(P, B)
        CALL OMP_UNSET_NEST_LOCK(P%LCK)
    END SUBROUTINE INCR_PAIR

    SUBROUTINE A45(P)
        USE OMP_LIB ! or INCLUDE "omp_lib.h"
        USE DATA
        TYPE(LOCKED_PAIR) :: P
        INTEGER WORK1, WORK2, WORK3
        EXTERNAL WORK1, WORK2, WORK3

!$OMP PARALLEL SECTIONS
!$OMP SECTION
        CALL INCR_PAIR(P, WORK1(), WORK2())
!$OMP SECTION
        CALL INCR_B(P, WORK3())
!$OMP END PARALLEL SECTIONS

    END SUBROUTINE A45
```

付録 B

実行時ライブラリのスタブルーチン

この章では、OpenMP API で定義される実行時ライブラリルーチンのスタブを提供します。スタブは、OpenMP API をサポートしていないプラットフォームでのポータビリティを確保するために提供されています。このようなプラットフォームでは、OpenMP プログラムはスタブルーチンを含むライブラリとリンクしなければなりません。スタブルーチンは OpenMP プログラム内の指示文を無視すると仮定しています。このようにして、逐次の動作をエミュレートします。

ロックルーチンの中に現れるロック変数は、これらのルーチンを通して排他的にアクセスされなければならないことに注意してください。その変数は、ユーザプログラム中で初期化やその他の方法で変更してはいけません。

実際の実装において、ロック変数が割り付けられたメモリブロックのアドレスを保持してもかまいません。しかし、ここでは整数の値を保持するものとしています。ユーザは、スタブ手続きで使用する仕組みをもとに、OpenMP の実装が使用しているロック実装の仕組みに関して仮定をしてはいけません。

Fortran

注：Fortran のスタブファイルをコンパイルするため、インクルードファイル omp_lib.h は 2 つのファイル（omp_lib_kinds.h と omp_lib.h）に分割されました。ファイル omp_lib_kinds.h は必要なところでインクルードされます。実装が分割したファイルを提供することは要求されていません。

Fortran

B．1．C/C++のスタブルーチン

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

void omp_set_num_threads(int num_threads)
{
}

int omp_get_num_threads(void)
{
       return 1;
}

int omp_get_max_threads(void)
{
       return 1;
}

int omp_get_thread_num(void)
{
       return 0;
}

int omp_get_num_procs(void)
{
       return 1;
}
int omp_in_parallel(void)
{
       return 0;
}
```

```
void omp_set_dynamic(int dynamic_threads)
{
}

int omp_get_dynamic(void)
{
       return 0;
}

void omp_set_nested(int nested)
{
}

int omp_get_nested(void)
{
       return 0;
}

void omp_set_schedule(omp_sched_t kind, int modifier)
{
}

void omp_get_schedule(omp_sched_t *kind, int *modifier)
{
       *kind = omp_sched_static;
       *modifier = 0;
}

int omp_get_thread_limit(void)
{
       return 1;
}

void omp_set_max_active_levels(int max_active_levels)
{
}
```

```
int omp_get_max_active_levels(void)
{
	return 0;
}

int omp_get_level(void)
{
	return 0;
}

int omp_get_ancestor_thread_num(int level)
{
	if (level == 0)
	{
		return 0;
	}
	else
	{
		return -1;
	}
}

int omp_get_team_size(int level)
{
	if (level == 0)
	{
		return 1;
	}
	else
	{
		return -1;
	}
}

int omp_get_active_level(void)
{
	return 0;
}
```

```
struct __omp_lock
{
	int lock;
};

enum { UNLOCKED = -1, INIT, LOCKED };

void omp_init_lock(omp_lock_t *arg)
{
	struct __omp_lock *lock = (struct __omp_lock *)arg;
	lock->lock = UNLOCKED;
}

void omp_destroy_lock(omp_lock_t *arg)
{
	struct __omp_lock *lock = (struct __omp_lock *)arg;
	lock->lock = INIT;
}

void omp_set_lock(omp_lock_t *arg)
{
	struct __omp_lock *lock = (struct __omp_lock *)arg;
	if (lock->lock == UNLOCKED)
	{
		lock->lock = LOCKED;
	}
	else if (lock->lock == LOCKED)
	{
		fprintf(stderr,
		"error: deadlock in using lock variable¥n");
		exit(1);
	}
	else
	{
		fprintf(stderr, "error: lock not initialized¥n");
		exit(1);
	}
}
```

```
void omp_unset_lock(omp_lock_t *arg)
{
        struct __omp_lock *lock = (struct __omp_lock *)arg;
        if (lock->lock == LOCKED)
        {
                lock->lock = UNLOCKED;
        }
        else if (lock->lock == UNLOCKED)
        {
                fprintf(stderr, "error: lock not set¥n");
                exit(1);
        }
        else
        {
                fprintf(stderr, "error: lock not initialized¥n");
                exit(1);
        }
}

int omp_test_lock(omp_lock_t *arg)
{
        struct __omp_lock *lock = (struct __omp_lock *)arg;
        if (lock->lock == UNLOCKED)
        {
                lock->lock = LOCKED;
                return 1;
        }
        else if (lock->lock == LOCKED)
        {
                return 0;
        }
        else
        {
                fprintf(stderr, "error: lock not initialized¥n");
                exit(1);
        }
}
```

```
struct __omp_nest_lock
{
        short owner;
        short count;
};

enum { NOOWNER = -1, MASTER = 0 };

void omp_init_nest_lock(omp_nest_lock_t *arg)
{
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
        nlock->owner = NOOWNER;
        nlock->count = 0;
}

void omp_destroy_nest_lock(omp_nest_lock_t *arg)
{
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
        nlock->owner = NOOWNER;
        nlock->count = UNLOCKED;
}

void omp_set_nest_lock(omp_nest_lock_t *arg)
{
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
        if (nlock->owner == MASTER && nlock->count >= 1)
        {
                nlock->count++;
        }
        else if (nlock->owner == NOOWNER && nlock->count == 0)
        {
                nlock->owner = MASTER;
                nlock->count = 1;
        }
        else
        {
                fprintf(stderr,
                "error: lock corrupted or not initialized¥n");
```

```
                exit(1);
        }
}

void omp_unset_nest_lock(omp_nest_lock_t *arg)
{
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
        if (nlock->owner == MASTER && nlock->count >= 1)
        {
                nlock->count--;
                if (nlock->count == 0)
                {
                        nlock->owner = NOOWNER;
                }
        }
        else if (nlock->owner == NOOWNER && nlock->count == 0)
        {
                fprintf(stderr, "error: lock not set¥n");
                exit(1);
        }
        else
        {
                fprintf(stderr,
                "error: lock corrupted or not initialized¥n");
                exit(1);
        }
}

int omp_test_nest_lock(omp_nest_lock_t *arg)
{
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
        omp_set_nest_lock(arg);
        return nlock->count;
}
```

```
double omp_get_wtime(void)
{
/* This function does not provide a working
* wallclock timer. Replace it with a version
* customized for the target machine.
*/
       return 0.0;
}

double omp_get_wtick(void)
{
/* This function does not provide a working
* clock tick function. Replace it with
* a version customized for the target machine.
*/
       return 365. * 86400.;
}
```

B．2．Fortran のスタブルーチン

```
C23456
      subroutine omp_set_num_threads(num_threads)
        integer num_threads
        return
      end subroutine

      integer function omp_get_num_threads()
        omp_get_num_threads = 1
        return
      end function

      integer function omp_get_max_threads()
        omp_get_max_threads = 1
        return
      end function

      integer function omp_get_thread_num()
        omp_get_thread_num = 0
        return
      end function

      integer function omp_get_num_procs()
        omp_get_num_procs = 1
        return
      end function

      logical function omp_in_parallel()
        omp_in_parallel = .false.
        return
      end function
```

```
subroutine omp_set_dynamic(dynamic_threads)
  logical dynamic_threads
  return
end subroutine

logical function omp_get_dynamic()
  omp_get_dynamic = .false.
  return
end function

subroutine omp_set_nested(nested)
  logical nested
  return
end subroutine

logical function omp_get_nested()
  omp_get_nested = .false.
  return
end function

subroutine omp_set_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
  return
end subroutine

subroutine omp_get_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
  kind = omp_sched_static
  modifier = 0
  return
end subroutine
```

```
integer function omp_get_thread_limit()
  omp_get_thread_limit = 1
  return
end function

subroutine omp_set_max_active_levels( level )
  integer level
end subroutine

integer function omp_get_max_active_levels()
  omp_get_max_active_levels = 0
  return
end function

integer function omp_get_level()
  omp_get_level = 0
  return
end function

integer function omp_get_ancestor_thread_num( level )
  integer level
  if ( level .eq. 0 ) then
       omp_get_ancestor_thread_num = 0
  else
       omp_get_ancestor_thread_num = -1
  end if
  return
end function

integer function omp_get_team_size( level )
  integer level
  if ( level .eq. 0 ) then
       omp_get_team_size = 1
  else
       omp_get_team_size = -1
  end if
  return
end function
```

```
integer function omp_get_active_level()
   omp_get_active_level = 0
   return
end function

subroutine omp_init_lock(lock)
   ! lock is 0 if the simple lock is not initialized
   !         -1 if the simple lock is initialized but not set
   !          1 if the simple lock is set
   include 'omp_lib_kinds.h'
   integer(kind=omp_lock_kind) lock
   lock = -1
   return
end subroutine

subroutine omp_destroy_lock(lock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_lock_kind) lock
   lock = 0
   return
end subroutine

subroutine omp_set_lock(lock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_lock_kind) lock
   if (lock .eq. -1) then
        lock = 1
   elseif (lock .eq. 1) then
        print *, 'error: deadlock in using lock variable'
        stop
   else
        print *, 'error: lock not initialized'
        stop
   endif
   return
end subroutine
```

```
subroutine omp_unset_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock
  if (lock .eq. 1) then
       lock = -1
  elseif (lock .eq. -1) then
       print *, 'error: lock not set'
       stop
  else
       print *, 'error: lock not initialized'
       stop
  endif
  return
end subroutine

logical function omp_test_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock
  if (lock .eq. -1) then
       lock = 1
       omp_test_lock = .true.
  elseif (lock .eq. 1) then
       omp_test_lock = .false.
  else
       print *, 'error: lock not initialized'
       stop
  endif
  return
end function

subroutine omp_init_nest_lock(nlock)
  ! nlock is
  !  0 if the nestable lock is not initialized
  ! -1 if the nestable lock is initialized but not set
  !  1 if the nestable lock is set
  ! no use count is maintained
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock
```

```
   nlock = -1
   return
end subroutine

subroutine omp_destroy_nest_lock(nlock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_nest_lock_kind) nlock
   nlock = 0
   return
end subroutine

subroutine omp_set_nest_lock(nlock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_nest_lock_kind) nlock
   if (nlock .eq. -1) then
        nlock = 1
   elseif (nlock .eq. 0) then
        print *, 'error: nested lock not initialized'
        stop
   else
        print *, 'error: deadlock using nested lock variable'
        stop
   endif
   return
end subroutine

subroutine omp_unset_nest_lock(nlock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_nest_lock_kind) nlock
   if (nlock .eq. 1) then
        nlock = -1
   elseif (nlock .eq. 0) then
        print *, 'error: nested lock not initialized'
        stop
   else
        print *, 'error: nested lock not set'
        stop
   endif
```

```
   return
end subroutine

integer function omp_test_nest_lock(nlock)
   include 'omp_lib_kinds.h'
   integer(kind=omp_nest_lock_kind) nlock
   if (nlock .eq. -1) then
        nlock = 1
        omp_test_nest_lock = 1
   elseif (nlock .eq. 1) then
        omp_test_nest_lock = 0
   else
        print *, 'error: nested lock not initialized'
        stop
   endif
   return
end function

double precision function omp_get_wtime()
   ! this function does not provide a working
   ! wall clock timer. replace it with a version
   ! customized for the target machine.
   omp_get_wtime = 0.0d0
   return
end function

double precision function omp_get_wtick()
   ! this function does not provide a working
   ! clock tick function. replace it with
   ! a version customized for the target machine.
   double precision one_year
   parameter (one_year=365.d0*86400.d0)
   omp_get_wtick = one_year
   return
end function
```

付録 C

OpenMP の C と C++の文法

C．1．表記

文法の規則は、non-terminal の名前、次はコロン、次は異なる行にある替わりの選択肢からなります。
シンタックス表現 term opt は、用語が、置き換えの中でオプションであることを示します。
シンタックス表現 term optseq は、以下の規則を追加して、term-seq opt と等価です。

```
term-seq :
   term
   term-seq term
   term-seq , term
```

C．2．規則

表記は、C 標準のセクション 6.1 に記述されます。この文法の付録は、OpenMP C/C++の指示文のためにベース言語の文法への拡張を記述します。

```
/* in C++ (ISO/IEC 14882:1998) */
statement-seq:
       statement
       openmp-directive
       statement-seq statement
       statement-seq openmp-directive


/* in C90 (ISO/IEC 9899:1990) */
statement-list:
       statement
       openmp-directive
       statement-list statement
       statement-list openmp-directive
```

```
/* in C99 (ISO/IEC 9899:1999) */
block-item:
        declaration
        statement
        openmp-directive
statement:
        /* standard statements */
        openmp-construct
openmp-construct:
        parallel-construct
        for-construct
        sections-construct
        single-construct
        parallel-for-construct
        parallel-sections-construct
        task-construct
        master-construct
        critical-construct
        atomic-construct
        ordered-construct
openmp-directive:
        barrier-directive
        taskwait-directive
        flush-directive
structured-block:
        statement
parallel-construct:
        parallel-directive structured-block
parallel-directive:
        # pragma omp parallel parallel-clause_optseq new-line
parallel-clause:
        unique-parallel-clause
        data-default-clause
        data-privatization-clause
        data-privatization-in-clause
        data-sharing-clause
```

```
        data-reduction-clause
unique-parallel-clause:
        if ( expression )
        num_threads ( expression )
        copyin ( variable-list )
for-construct:
        for-directive iteration-statement
for-directive:
        # pragma omp for for-clause_optseq new-line
for-clause:
        unique-for-clause
        data-privatization-clause
        data-privatization-in-clause
        data-privatization-out-clause
        data-reduction-clause
        nowait
unique-for-clause:
        ordered
        schedule ( schedule-kind )
        schedule ( schedule-kind , expression )
        collapse ( expression )
schedule-kind:
        static
        dynamic
        guided
        auto
        runtime
sections-construct:
        sections-directive section-scope
sections-directive:
        # pragma omp sections sections-clause_optseq new-line
sections-clause:
        data-privatization-clause
        data-privatization-in-clause
        data-privatization-out-clause
        data-reduction-clause
        nowait
section-scope:
```

{ section-sequence }

section-sequence:

section-directive$_{opt}$ *structured-block*

section-sequence section-directive structured-block

section-directive:

 # pragma omp section *new-line*

single-construct:

single-directive structured-block

single-directive:

 # pragma omp single *single-clause*$_{optseq}$ *new-line*

single-clause:

unique-single-clause

data-privatization-clause

data-privatization-in-clause

nowait

unique-single-clause:

copyprivate (*variable-list* **)**

task-construct:

task-directive structured-block

task-directive:

 # pragma omp task *task-clause*$_{optseq}$ *new-line*

task-clause:

unique-task-clause

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-sharing-clause

unique-task-clause:

if (*scalar-expression* **)**

untied

parallel-for-construct:

parallel-for-directive iteration-statement

parallel-for-directive:

 # pragma omp parallel for *parallel-for-clause*$_{optseq}$ *new-line*

parallel-for-clause:

unique-parallel-clause

unique-for-clause

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-clause

parallel-sections-construct:

parallel-sections-directive section-scope

parallel-sections-directive:

 # pragma omp parallel sections *parallel-sections-clause*$_{optseq}$ *new-line*

parallel-sections-clause:

unique-parallel-clause

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-clause

master-construct:

master-directive structured-block

master-directive:

 # pragma omp master *new-line*

critical-construct:

critical-directive structured-block

critical-directive:

 # pragma omp critical *region-phrase*$_{opt}$ *new-line*

region-phrase:

(identifier)

barrier-directive:

 # pragma omp barrier *new-line*

taskwait-directive:

 # pragma omp taskwait *new-line*

atomic-construct:

atomic-directive expression-statement

atomic-directive:

 # pragma omp atomic *new-line*

flush-directive:

 # pragma omp flush *flush-vars*$_{opt}$ *new-line*

flush-vars:

```
        ( variable-list )
ordered-construct:
        ordered-directive structured-block
ordered-directive:
        # pragma omp ordered new-line
declaration:
        /* standard declarations */
        threadprivate-directive
threadprivate-directive:
        # pragma omp threadprivate ( variable-list ) new-line
data-default-clause:
        default ( shared )
        default ( none )
data-privatization-clause:
        private ( variable-list )
data-privatization-in-clause:
        firstprivate ( variable-list )
data-privatization-out-clause:
        lastprivate ( variable-list )
data-sharing-clause:
        shared ( variable-list )
data-reduction-clause:
        reduction ( reduction-operator : variable-list )
reduction-operator:
        One of: + * - & ^ | && ||
/* in C */
variable-list:
        identifier
        variable-list , identifier
/* in C++ */
variable-list:
        id-expression
        variable-list , id-expression
```

付録 D

インタフェース宣言

この付録では、第 3 章で規定したように実装によって提供されなければならない C/C++のヘッダファイル、Fortran のインクルードファイル、および Fortran 90 のモジュールの例を示します。
また、ライブラリルーチンのための Fortran 90 の総称引用仕様の例も含んでいます。

D．1．omp.h ヘッダファイルの例

```
#ifndef _OMP_H_DEF
#define _OMP_H_DEF

/*
 * define the lock data types
 */
typedef void *omp_lock_t;

typedef void *omp_nest_lock_t;

/*
 * define the schedule kinds
 */
typedef enum omp_sched_t
{
        omp_sched_static = 1,
        omp_sched_dynamic = 2,
        omp_sched_guided = 3,
        omp_sched_auto = 4
/* , Add vendor specific schedule constants here */
} omp_sched_t;
```

```
/*
 * exported OpenMP functions
 */
#ifdef __cplusplus
extern        "C"
{
#endif

extern void   omp_set_num_threads(int num_threads);
extern int    omp_get_num_threads(void);
extern int    omp_get_max_threads(void);
extern int    omp_get_thread_num(void);
extern int    omp_get_num_procs(void);
extern int    omp_in_parallel(void);
extern void   omp_set_dynamic(int dynamic_threads);
extern int    omp_get_dynamic(void);
extern void   omp_set_nested(int nested);
extern int    omp_get_nested(void);
extern int    omp_get_thread_limit(void);
extern void   omp_set_max_active_levels(int max_active_levels);
extern int    omp_get_max_active_levels(void);
extern int    omp_get_level(void);
extern int    omp_get_ancestor_thread_num(int level);
extern int    omp_get_team_size(int level);
extern int    omp_get_active_level(void);
extern void   omp_set_schedule(omp_sched_t kind, int modifier);
extern void   omp_get_schedule(omp_sched_t *kind, int *modifier);

extern void   omp_init_lock(omp_lock_t *lock);
extern void   omp_destroy_lock(omp_lock_t *lock);
extern void   omp_set_lock(omp_lock_t *lock);
extern void   omp_unset_lock(omp_lock_t *lock);
extern int    omp_test_lock(omp_lock_t *lock);

extern void   omp_init_nest_lock(omp_nest_lock_t *lock);
extern void   omp_destroy_nest_lock(omp_nest_lock_t *lock);
extern void   omp_set_nest_lock(omp_nest_lock_t *lock);
extern void   omp_unset_nest_lock(omp_nest_lock_t *lock);
```

```
extern int    omp_test_nest_lock(omp_nest_lock_t *lock);

extern double omp_get_wtime(void);
extern double omp_get_wtick(void);
#ifdef __cplusplus
}
#endif

#endif
```

D．2．引用仕様宣言のインクルードファイルの例

omp_lib_kinds.h:

```
       integer omp_lock_kind
       parameter ( omp_lock_kind = 8 )
       integer omp_nest_lock_kind
       parameter ( omp_nest_lock_kind = 8 )

       integer omp_sched_kind
       parameter ( omp_sched_kind = 4)

       integer ( omp_sched_kind ) omp_sched_static
       parameter ( omp_sched_static = 1 )
       integer ( omp_sched_kind ) omp_sched_dynamic
       parameter ( omp_sched_dynamic = 2 )
       integer ( omp_sched_kind ) omp_sched_guided
       parameter ( omp_sched_guided = 3 )
       integer ( omp_sched_kind ) omp_sched_auto
       parameter ( omp_sched_auto = 4 )
```

omp_lib.h:

```
C                    default integer type assumed below
C                    default logical type assumed below
C                    OpenMP Fortran API v3.0

       include 'omp_lib_kinds.h'
       integer openmp_version
       parameter ( openmp_version = 200805 )

       external omp_set_num_threads
       external omp_get_num_threads
       integer omp_get_num_threads
       external omp_get_max_threads
       integer omp_get_max_threads
       external omp_get_thread_num
       integer omp_get_thread_num
```

```
external omp_get_num_procs
integer omp_get_num_procs
external omp_in_parallel
logical omp_in_parallel
external omp_set_dynamic
external omp_get_dynamic
logical omp_get_dynamic
external omp_set_nested
external omp_get_nested
logical omp_get_nested
external omp_set_schedule
external omp_get_schedule
external omp_get_thread_limit
integer omp_get_thread_limit
external omp_set_max_active_levels
external omp_get_max_active_levels
integer omp_get_max_active_levels
external omp_get_level
integer omp_get_level
external omp_get_ancestor_thread_num
integer omp_get_ancestor_thread_num
external omp_get_team_size
integer omp_get_team_size
external omp_get_active_level
integer omp_get_active_level

external omp_init_lock
external omp_destroy_lock
external omp_set_lock
external omp_unset_lock
external omp_test_lock
logical omp_test_lock

external omp_init_nest_lock
external omp_destroy_nest_lock
external omp_set_nest_lock
external omp_unset_nest_lock
external omp_test_nest_lock
```

```
integer omp_test_nest_lock

external omp_get_wtick
double precision omp_get_wtick
external omp_get_wtime
double precision omp_get_wtime
```

D．3．Fortran 90 引用仕様宣言のモジュールの例

```
! the "!" of this comment starts in column 1
!23456
       module omp_lib_kinds
       integer, parameter :: omp_integer_kind = 4
       integer, parameter :: omp_logical_kind = 4
       integer, parameter :: omp_lock_kind = 8
       integer, parameter :: omp_nest_lock_kind = 8
       integer, parameter :: omp_sched_kind = 4
       integer(kind=omp_sched_kind), parameter ::
      & omp_sched_static = 1
       integer(kind=omp_sched_kind), parameter ::
      & omp_sched_dynamic = 2
       integer(kind=omp_sched_kind), parameter ::
      & omp_sched_guided = 3
       integer(kind=omp_sched_kind), parameter ::
      & omp_sched_auto = 4
       end module omp_lib_kinds

       module omp_lib

       use omp_lib_kinds
!                    OpenMP Fortran API v3.0
       integer, parameter :: openmp_version = 200805

       interface

       subroutine omp_set_num_threads (number_of_threads_expr)
         use omp_lib_kinds
       integer (kind=omp_integer_kind), intent(in) ::
      &        number_of_threads_expr
       end subroutine omp_set_num_threads
```

```
 function omp_get_num_threads ()
   use omp_lib_kinds
   integer (kind=omp_integer_kind) :: omp_get_num_threads
 end function omp_get_num_threads

 function omp_get_max_threads ()
   use omp_lib_kinds
   integer (kind=omp_integer_kind) :: omp_get_max_threads
 end function omp_get_max_threads

 function omp_get_thread_num ()
   use omp_lib_kinds
   integer (kind=omp_integer_kind) :: omp_get_thread_num
 end function omp_get_thread_num

 function omp_get_num_procs ()
   use omp_lib_kinds
   integer (kind=omp_integer_kind) :: omp_get_num_procs
 end function omp_get_num_procs

 function omp_in_parallel ()
   use omp_lib_kinds
   logical (kind=omp_logical_kind) :: omp_in_parallel
 end function omp_in_parallel

 subroutine omp_set_dynamic (enable_expr)
   use omp_lib_kinds
   logical (kind=omp_logical_kind), intent(in) ::
&       enable_expr
 end subroutine omp_set_dynamic

 function omp_get_dynamic ()
   use omp_lib_kinds
   logical (kind=omp_logical_kind) :: omp_get_dynamic
 end function omp_get_dynamic
```

```
 subroutine omp_set_nested (enable_expr)
    use omp_lib_kinds
    logical (kind=omp_logical_kind), intent(in) ::
&       enable_expr
 end subroutine omp_set_nested

function omp_get_nested ()
   use omp_lib_kinds
   logical (kind=omp_logical_kind) :: omp_get_nested
 end function omp_get_nested

 subroutine omp_set_schedule (kind, modifier)
    use omp_lib_kinds
    integer(kind=omp_sched_kind), intent(in) :: kind
    integer(kind=omp_integer_kind), intent(in) :: modifier
 end subroutine omp_set_schedule

 subroutine omp_get_schedule (kind, modifier)
    use omp_lib_kinds
    integer(kind=omp_sched_kind), intent(out) :: kind
    integer(kind=omp_integer_kind), intent(out)::modifier
 end subroutine omp_get_schedule

 function omp_get_thread_limit()
    use omp_lib_kinds
    integer (kind=omp_integer_kind) :: omp_get_thread_limit
 end function omp_get_thread_limit

 subroutine omp_set_max_active_levels(var)
    use omp_lib_kinds
    integer (kind=omp_integer_kind), intent(in) :: var
 end subroutine omp_set_max_active_levels

 function omp_get_max_active_levels()
    use omp_lib_kinds
    integer (kind=omp_integer_kind) ::
&       omp_get_max_active_levels
 end function omp_get_max_active_levels
```

```
 function omp_get_level()
   use omp_lib_kinds
   integer (kind=omp_integer_kind) :: omp_get_level
 end function omp_get_level

 function omp_get_ancestor_thread_num(level)
   use omp_lib_kinds
   integer (kind=omp_integer_kind), intent(in) ::
&       level
   integer (kind=omp_integer_kind) ::
&       omp_get_ancestor_thread_num
 end function omp_get_ancestor_thread_num

 function omp_get_team_size(level)
   use omp_lib_kinds
   integer (kind=omp_integer_kind), intent(in) ::
&       level
   integer (kind=omp_integer_kind) :: omp_get_team_size
 end function omp_get_team_size

 function omp_get_active_level()
   use omp_lib_kinds
   integer (kind=omp_integer_kind) ::
&       omp_get_active_level
 end function omp_get_active_level

 subroutine omp_init_lock (var)
   use omp_lib_kinds
   integer (kind=omp_lock_kind), intent(out) :: var
 end subroutine omp_init_lock

 subroutine omp_destroy_lock (var)
   use omp_lib_kinds
   integer (kind=omp_lock_kind), intent(inout) :: var
 end subroutine omp_destroy_lock
```

```
subroutine omp_set_lock (var)
  use omp_lib_kinds
  integer (kind=omp_lock_kind), intent(inout) :: var
end subroutine omp_set_lock

subroutine omp_unset_lock (var)
  use omp_lib_kinds
  integer (kind=omp_lock_kind), intent(inout) :: var
end subroutine omp_unset_lock

function omp_test_lock (var)
  use omp_lib_kinds
  logical (kind=omp_logical_kind) :: omp_test_lock
  integer (kind=omp_lock_kind), intent(inout) :: var
end function omp_test_lock

subroutine omp_init_nest_lock (var)
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(out) :: var
end subroutine omp_init_nest_lock

subroutine omp_destroy_nest_lock (var)
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(inout) :: var
end subroutine omp_destroy_nest_lock

subroutine omp_set_nest_lock (var)
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(inout) :: var
end subroutine omp_set_nest_lock

subroutine omp_unset_nest_lock (var)
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(inout) :: var
end subroutine omp_unset_nest_lock
```

```
 function omp_test_nest_lock (var)
   use omp_lib_kinds
   integer (kind=omp_integer_kind) :: omp_test_nest_lock
   integer (kind=omp_nest_lock_kind), intent(inout) ::
&       var
 end function omp_test_nest_lock

 function omp_get_wtick ()
   double precision :: omp_get_wtick
 end function omp_get_wtick

 function omp_get_wtime ()
   double precision :: omp_get_wtime
 end function omp_get_wtime

 end interface
 end module omp_lib
```

D．4．ライブラリルーチンのための総称引用仕様の例

引数を持つ OpenMP のどの実行時ライブラリルーチンも、異なった KIND タイプの引数に適用できるように、総称引用仕様を拡張することができます。KIND=OMP_INTEGER_KIND として、デフォルトの INTEGER と KIND=SHORT_INT の別の INTEGER KIND をサポートする実装を考えてみます。このとき、OMP_SET_NUM_THREADS は、以下のように omp_lib モジュールの中で定義することができます。

```
! the "!" of this comment starts in column 1
      interface omp_set_num_threads

      subroutine omp_set_num_threads_1 ( number_of_threads_expr )
        use omp_lib_kinds
        integer ( kind=omp_integer_kind ), intent(in) :: &
     &        number_of_threads_expr
      end subroutine omp_set_num_threads_1

      subroutine omp_set_num_threads_2 ( number_of_threads_expr )
        use omp_lib_kinds
        integer ( kind=short_int ), intent(in) :: &
     &        number_of_threads_expr
      end subroutine omp_set_num_threads_2

      end interface omp_set_num_threads
```

付録 E

OpenMP における実装依存の振舞い

この付録では、この API で実装依存として記述された振舞いについて要約します。それぞれの振舞いから仕様書本文の解説が相互参照されています。実装は、これらの振舞いを定義し、文書化する必要があります。

- タスクスケジューリングポイント：アンタイド task リージョン内のタスクスケジューリングポイントの位置は実装依存です。（セクション 1.3 を参照してください。）
- メモリモデル：複数のスレッドが同期しないで同じ変数に対してメモリアクセスする場合、互いにアトミックであるかどうか、またそのアクセスにおけるサイズは実装依存です。（セクション 1.4.1 を参照してください。）
- 内部制御変数：*nthreads-var*、*dyn-var*、*run-sched-var*、*def-sched-var*、*stacksize-var*、*wait-policy-var*、*thread-limit-var*、および *max-active-levels-var* の初期値は実装依存です。（セクション 2.3.2 を参照してください。）
- スレッドの動的調整：スレッド数の動的調整の機能が提供されるかどうかは実装依存です。動的調整が使用できない場合でも、実装はアルゴリズム 2-1 で示されるより少ない数のスレッド（最小で 1）を提供することが許されます。（セクション 2.4.1 を参照してください。）
- ループ指示文：一重化したループの繰り返し回数の計算に使用する変数がどの整数型または種類であるかは実装依存です。内部制御変数 *run-sched-var* に auto が設定されたときの schedule(runtime)指示節の影響は実装依存です。（セクション 2.5.1 を参照してください。）
- sections 構文：チーム内のスレッドへ構造化ブロックをスケジュールする方法は実装依存です。（セクション 2.5.2 を参照してください。）
- single 構文：構造化ブロックを実行するスレッドの選択方法は実装依存です。（セクション 2.5.3 を参照してください。）
- atomic 構文：準拠した実装が、記憶域の異なる位置を更新する atomic リージョン間で排他的なアクセスを行うことがあります。これが起こる環境は実装依存です。（セクション 2.8.5 を参照してください。）
- omp_set_num_threads ルーチン：引数が正の整数でない場合の振舞いは実装依存です。（セクション 3.2.1 を参照してください。）
- omp_set_schedule ルーチン：実装依存のスケジュールタイプの振舞いは実装依存です。（セクション 3.2.11 を参照してください。）
- omp_set_max_active_levels ルーチン：明示的な parallel リージョン内から呼ばれたときに omp_set_max_active_levels リージョンに対する結合スレッドセット（および、必要な場合は結

合しているリージョン）は実装依存です。そして、その振舞いも実装依存です。引数がゼロ以上の整数でない場合の振舞いは実装依存です。（セクション 3.2.14 を参照してください。）

- omp_get_max_active_levels ルーチン：明示的な parallel リージョン内から呼ばれたときに omp_get_max_active_levels リージョンに対する結合スレッドセット（および、必要な場合は結合しているリージョン）は実装依存です。（セクション 3.2.15 を参照してください。）
- OMP_SCHEDULE 環境変数：変数の値が定義された形式を満たさないない場合の結果は実装依存です。（セクション 4.1 を参照してください。）
- OMP_NUM_THREADS 環境変数：変数の値が実装がサポートしているスレッド数よりも大きい、または正の整数でない場合の結果は実装依存です。（セクション 4.2 を参照してください。）
- OMP_DYNAMIC 環境変数：値が true または false のいずれでもない場合の振舞いは実装依存です。（セクション 4.3 を参照してください。）
- OMP_NESTED 環境変数：値が true または false のいずれでもない場合の振舞いは実装依存です。（セクション 4.4 を参照してください。）
- OMP_STACKSIZE 環境変数：値が定義された形式を満たしていない、または実装が指定されサイズのスタックを提供することができない場合の振舞いは実装依存です。（セクション 4.5 を参照してください。）
- OMP_WAIT_POLICY 環境変数：ACTIVE と PASSIVE の振舞いの詳細は実装依存です。（セクション 4.6 を参照してください。）
- OMP_MAX_ACTIVE_LEVELS 環境変数：値がゼロ以上の整数でない場合、または実装がサポートできる並列レベルの数よりも大きい場合の振舞いは実装依存です。（セクション 4.7 を参照してください。）
- OMP_THREAD_LIMIT 環境変数：指定した値が実装がサポートできるスレッド数よりも大きい場合、または正の整数でない場合のプログラムの振舞いは実装依存です。（セクション 4.8 を参照してください。）

Fortran

- threadprivate 指示文：2 つの連続した活動状態の parallel リージョン間で、（最初のスレッドではない）スレッドのスレッドプライベートオブジェクト中のデータの値が保存されるための条件をすべて満たしていない場合、2 番目のリージョンでの割付け可能配列の割付け状態は実装依存です。（セクション 2.9.2 を参照してください。）
- shared 指示節：非組込み手続きに共有変数を渡すと、手続の引用前に共有変数の値を一時記憶域にコピーし、手続きの引用後に実引数領域に戻れされることがあります。ここで定義された場合以外にこのような現象が起こる条件は実装依存です。（セクション 2.9.2 を参照してください。）
- 実行時ライブラリ定義：インクルードファイル omp_lib.h またはモジュール omp_lib（または両方）が提供されるかどうかは実装依存です。異なった KIND 型の引数を適用するため、引数を持つ OpenMP 実行時ライブラリルーチンを総称引用仕様で拡張するかどうかは実装依存です。（セクション 3.1 を参照してください。）

Fortran

付録 F

V2.5 から V3.0 への変更

この付録では、OpenMP API 2.5 の仕様と OpenMP 3.0 の仕様の間の主な相違について要約します。

- OpenMP 実行モデルにタスクの概念が追加されました。（セクション 1.2.3 とセクション 1.3 を参照してください。）
- 明示的にタスクを生成するメカニズムである task 構文（セクション 2.7 を参照してください。）が追加されました。
- タスクがすべての子タスクの完了を待つための taskwait 構文（セクション 2.8.4 を参照してください。）が追加されました。
- OpenMP のメモリモデルがアトミックなメモリアクセスをカバーしました。（セクション 1.4.1 を参照してください。）フラッシュに関する volatile の振舞いの記述は削除されました。
- バージョン 2.5 では、内部制御変数 *nest-var*、*dyn-var*、*nthreads-var* と *run-sched-var* はプログラム全体で 1 つでした。バージョン 3.0 では、これらの内部制御変数はタスク単位に 1 つになりました。（セクション 2.3 を参照してください。）結果として、実行時ライブラリルーチン omp_set_num_threads、omp_set_nested、および omp_set_dynamic は、parallel リージョン内から呼び出された場合でも定義された効果を発揮します。（セクション 3.2.1、セクション 3.2.7、およびセクション 3.2.9 を参照してください。）
- 活動状態の parallel リージョンの定義が変わりました。バージョン 3.0 では、parallel リージョンは 2 つ以上のスレッドから成るチームによって実行される場合に活動状態となります。（セクション 1.2.2 を参照してください。）
- parallel リージョンで使用するスレッドの数の決定規則が変更されました。（セクション 2.4.1 を参照してください。）
- バージョン 3.0 において、static スケジュールを指定したループ構文の繰り返しの各スレッドへの割り当て方は決定的になります。（セクション 2.5.1 を参照してください。）
- バージョン 3.0 において、ループ構文は完全にネストした 2 つ以上のループに関連付けることができます。関連付けるループの数は collapse 指示節によって制御できます。（セクション 2.5.1 を参照してください。）
- ランダム・アクセス・イテレータと符号なし整数型の変数をループ構文に関連付けられたループのループイテレータとして使用することができます。（セクション 2.5.1 を参照してください。）
- 実装がループ構文内の繰り返しをチーム内のスレッドへ自由にマッピングすることができるスケジュール種別 auto を追加しました。（セクション 2.5.1 を参照してください。）
- Fortran の大きさ引継ぎ配列は、事前に決定されるデータ共有属性を持ちます。（セクション 2.9.1.1 を参照してください。）

- Fortran では、firstprivate を default 指示節の引数として指定できます。（セクション 2.9.3.1 を参照してください。）
- private 指示節に指定された変数に対して、実装はマスタースレッドの新しいローカル変数の値を保持するためにオリジナルの記憶域を使うことは許されません。もし、parallel リージョン内でオリジナル変数に対する参照がなければ、その値は parallel リージョンの出口でも定義されたままです。（セクション 2.9.3.3 を参照してください。）
- バージョン 3.0 において、Fortran の割付け可能配列は private、firstprivate、lastprivate、reduction、copyin、および copyprivate 指示節に指定できます。（セクション 2.9.2、セクション 2.9.3.3、セクション 2.9.3.4、セクション 2.9.3.5、セクション 2.9.3.6、セクション 2.9.4.1、およびセクション 2.9.4.2 を参照してください。）
- バージョン 3.0 において、静的クラスメンバー変数は threadprivate 指示文に指定できます。（セクション 2.9.2 を参照してください。）
- バージョン 3.0 は、プライベートとスレッドプライベートの変数クラスのコンストラクタ、デストラクタを呼び出す位置とその引数を明確にしました。（セクション 2.9.2、セクション 2.9.3.3、セクション 2.9.3.4、セクション 2.9.4.1、およびセクション 2.9.4.2 を参照してください。）
- 実行時ライブラリルーチン omp_set_schedule と omp_get_schedule が追加されました。これらのルーチンは *run-sched-var* 内部制御変数に対して値の設定と参照をします。（セクション 3.2.11 とセクション 3.2.12 を参照してください。）
- *thread-limit-var* 内部制御変数が追加されました。この変数は OpenMP プログラムに参加するスレッドの最大数を制御します。この内部制御変数の値は、OMP_THREAD_LIMIT 環境変数で設定し、omp_get_thread_limit 実行時ライブラリルーチンで参照します。（セクション 2.3.1、セクション 3.2.13、およびセクション 4.8 を参照してください。）
- *max-active-levels-var* 内部制御変数が追加されました。この変数は、ネスト可能な活動状態の parallel リージョンの数を制御します。この内部制御変数の値は、OMP_MAX_ACTIVE_LEVELS 環境変数と omp_set_max_active_levels 実行時ライブラリルーチンで設定し、omp_get_max_active_levels 実行時ライブラリルーチンで参照します。（セクション 2.3.1、セクション 3.2.14、セクション 3.2.15、およびセクション 4.7 を参照してください。）
- *stack-size-var* 内部制御変数が追加されました。この変数は OpenMP の実装が生成するスレッドのスタックサイズを制御します。この内部制御変数の値は、OMP_STACKSIZE 環境変数で設定します。（セクション 2.3.1 を参照してください。）
- *wait-policy-var* 内部制御変数が追加されました。この変数は、ウェイトしているスレッドの期待する振舞いについて制御します。この内部制御変数の値は、OMP_WAIT_POLICY 環境変数で設定します。（セクション 2.3.1 およびセクション 4.6 を参照してください。）
- omp_get_level 実行時ライブラリルーチンが追加されました。この呼出しを含んでいるタスクを囲む parallel リージョンのネスト数を返します。（セクション 3.2.16 を参照してください。）
- omp_get_ancestor_thread_num 実行時ライブラリルーチンが追加されました。このルーチンは、呼び出したスレッドで指定したネストレベルに対応する先祖にあたるスレッド番号を返します。（セクション 3.2.17 を参照してください。）

- omp_get_team_size 実行時ライブラリルーチンが追加されました。このルーチンは、呼び出したスレッドで指定したネストレベルに対応した先祖が属しているスレッドチームのサイズを返します。（セクション 3.2.18 を参照してください。）
- omp_get_active_level 実行時ライブラリルーチンが追加されました。このルーチンは、このルーチンの呼出しを含んでいるタスクを囲む活動状態の parallel リージョンのネスト数を返します。（セクション 3.2.19 を参照してください。）
- バージョン 3.0 では、ロックはスレッドではなくタスクが所有しています。（セクション 3.3 を参照してください。）