Oracle® Solaris Studio 12.2: OpenMP API ユーザーガイド

Part No: 821–2493
2010 年 9 月

ORACLE®

110412@25097

目次

はじめに

『OpenMP API ユーザーズガイド』では、マルチプロセッサ対応のアプリケーションを構築するための OpenMP Fortran 95、C、C++ アプリケーションプログラムインタフェース (API) について解説します。Oracle Solaris Studio のコンパイラは、OpenMP API をサポートしています。このマニュアルは、Fortran、C、C++ 言語、および OpenMP 並列プログラミングモデルの知識を有する科学者、エンジニア、プログラマを対象としています。また、Oracle Solaris オペレーティングシステムや UNIX の一般的な知識を持つ読者を対象としています。

サポートされるプラットフォーム

この Oracle Solaris Studio のリリースは、SPARC および x86 ファミリ (UltraSPARC、SPARC64、AMD64、Pentium、Xeon EM64T) プロセッサアーキテクチャーを使用するシステムをサポートしています。使用の Solaris オペレーティングシステムのバージョンに対するシステムのサポート状況は、ハードウェア互換性リスト (http://www.sun.com/bigadmin/hcl) をご参照ください。ここには、すべてのプラットフォームごとの実装の違いについて説明されています。

このドキュメントでは、x86 関連の用語は次のものを指します。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換製品を指します。
- 「x64」は、AMD 64 または EM64T システムで、特定の 64 ビット情報を指します。
- 「32 ビット x86」は、x86 ベースシステムで特定の 32 ビット情報を指します。

サポートされるシステムについては、ハードウェアの互換性に関するリストを参照してください。

Solaris Studio マニュアルへのアクセス方法

マニュアルには、次の場所からアクセスできます。

- マニュアルは、次に示すマニュアル索引のページからアクセスできます。http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation。

- IDE、パフォーマンスアナライザ、dbxtool、および DLight の全コンポーネントのオンラインヘルプは、これらのツール内の「ヘルプ」メニューだけでなく、F1 キー、および多くのウィンドウやダイアログにある「ヘルプ」ボタンを使用してアクセスできます。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは次の表に示す場所から参照することができます。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアル	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
『Oracle Solaris Studio 12.2 リリースの新機能』(以前はコンポーネントの README ファイル)	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
マニュアルページ	`man` コマンドを使用して Oracle Solaris 端末に表示されます。
オンラインヘルプ	HTML 形式。IDE、dbxtool、DLight、およびパフォーマンスアナライザの「ヘルプ」メニュー、「ヘルプ」ボタン、および F1 キーを使用して表示
リリースノート	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択

関連するサードパーティの **Web** サイトリファレンス

このマニュアルには、詳細な関連情報を提供するサードパーティの URL が記載されています。

注 – このマニュアルで紹介するサードパーティ Web サイトが使用可能かどうかについては、Oracle は責任を負いません。このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、広告、製品、またはその他の資料についても、Oracle は保証しておらず、法的責任を負いません。また、このようなサイトやリソースから直接あるいは経由することで利用できるコンテンツ、商品、サービスの使用または依存が直接のあるいは関連する要因となり実際に発生した、あるいは発生するとされる損害や損失についても、Oracle は一切の法的責任を負いません。

開発者向けのリソース

http://www.oracle.com/technetwork/server-storage/solarisstudio を参照して、次の頻繁に更新されるリソースを確認してください。

- リソースは頻繁に更新されます。
- ソフトウェアのマニュアル、およびソフトウェアとともにインストールされる一連のマニュアル
- Oracle Solaris Studio ツールを使用して行う開発タスク全体を順を追って説明するチュートリアル
- サポートレベルに関する情報
- http://forums.sun.com/category.jspa?categoryID=113 にあるユーザーフォーラム

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1　表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。

表 P–1 表記上の規則 (続き)

字体または記号	意味	例
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	`sun% grep '^#define \` `XV_VERSION_STRING'`

コード例は次のように表示されます。

- C シェル

  ```
  machine_name% command y|n [filename]
  ```

- C シェルのスーパーユーザー

  ```
  machine_name# command y|n [filename]
  ```

- Bourne シェルおよび Korn シェル

  ```
  $ command y|n [filename]
  ```

- Bourne シェルおよび Korn シェルのスーパーユーザー

  ```
  # command y|n [filename]
  ```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

マニュアル、サポート、およびトレーニング

追加リソースについては、次の Web サイトを参照してください。

- マニュアル (http://docs.sun.com)
- サポート (http://www.oracle.com/us/support/systems/index.html)
- トレーニング (http://education.oracle.com) – 左側のナビゲーションバーで Sun へのリンクをクリックしてください。

ご意見の送付先

マニュアルの品質や使いやすさに関するご意見やご提案をお待ちしています。間違いやその他の改善すべき箇所がありましたら、http://docs.sun.com で「Feedback」をクリックしてお知らせください。ドキュメント名とドキュメントの Part No.、および、可能な場合は章、節、ページ番号を記載してください。返答が必要な場合はお知らせください。

Oracle 技術ネットワーク (http://www.oracle.com/technetwork/index.html) では、Oracle ソフトウェアに関するさまざまなリソースを提供しています。

- 技術上の問題やソリューションについては、ディスカッションフォーラム (http://forums.oracle.com) を参照してください。
- 実践的なステップ・バイ・ステップのチュートリアルについては、Oracle By Example (http://www.oracle.com/technology/obe/start/index.html) を参照してください。
- サンプルコードのダウンロードについては、サンプルコード (http://www.oracle.com/technology/sample_code/index.html) を参照してください。

◆ ◆ ◆ 第 1 章

1 OpenMP API について

OpenMP Application Program Interface (API) は、共有メモリー型マルチスレッドアーキテクチャー用の移植性のある並列プログラミングモデルで、多数のコンピュータベンダーと共同で開発されました。仕様書は OpenMP Architecture Review Board で作成され、発行されています。

OpenMP API は、Solaris プラットフォームで動作するすべての Solaris Studio コンパイラに対して推奨している並列プログラミングモデルです。従来の Fortran および C の並列化指令を OpenMP 指令に変換する方法については、付録を参照してください。

1.1 OpenMP 仕様の参照先

このマニュアルに示す資料は、OpenMP API の Solaris Studio 実装に固有の内容を説明したものです。詳細については、必ず OpenMP 仕様書を参照してください。
このマニュアルの中でも、参照先として OpenMP 3.0 API 仕様の各項目を示してあります。

C、C++、Fortran 95 の OpenMP 3.0 仕様については、OpenMP の公式 Web サイト (http://www.openmp.org) を参照してください。

チュートリアルおよびその他の開発者向け関連ドキュメントなど OpenMP に関する詳細については、COMPunity の Web サイト (http://www.compunity.org/) を参照してください。

Solaris Studio コンパイラの全リリースと、各々での OpenMP API の実装に関する最新情報については、Oracle Solaris Studio ポータル (http://www.oracle.com/technetwork/server-storage/solarisstudio) を参照してください。

1.2 このマニュアルで使用している特別な表記

後述の表および例では、Fortran の指令およびソースコードは大文字で表記されていますが、実際には大文字と小文字は区別されません。

structured-block は、ブロックの内外への転送を行わない Fortran 文または C/C++ 文のブロックを指します。

大かっこ [...] の中に記述された構造はオプションです。

このマニュアルでは、「Fortran」は Fortran 95 言語およびそのコンパイラである **f95** を示します。

「指令」および「プラグマ」は、このマニュアルでは同義で使用されています。

◆ ◆ ◆ 第 2 章

2 OpenMP プログラムのコンパイルと実行

この章では、OpenMP API を使用するプログラムに影響するコンパイラおよび実行時オプションを説明します。

注 – 並列化されたプログラムをマルチスレッド環境で実行するには、プログラムのスレッド数に 1 より大きい数値を設定する必要があります。これを行うには、プログラムを実行する前に **OMP_NUM_THREADS** 環境変数の値を 1 より大きい数値に設定するか、**omp_set_num_threads()** 関数の呼び出しで実行されるプログラムから行うか、**PARALLEL** 指令の **num_threads** 節を使用します。

2.1 使用するコンパイラオプション

OpenMP の指令を使用して明示的に並列化を有効にするには、**cc**、**CC**、または **f95** のオプションフラグ **-xopenmp** を指定してプログラムをコンパイルします。**f95** コンパイラでは、**-xopenmp** と **-openmp** を同義語として使用することができます。

-xopenmp フラグには、次のキーワードサブオプションを指定することができます。

-xopenmp=parallel	OpenMP プラグマの認識を有効にします。 **-xopenmp=parallel** の最小限の最適化レベルは **-xO3** です。 コンパイラは、必要に応じて最適化のレベルを低いレベルから **-xO3** に上げ、警告を出力します。

-xopenmp=noopt	OpenMP プラグマの認識を有効にします。 最適化のレベルが **-x03** より低い場合でも、コンパイラは最適化のレベルを上げません。 **-xO2 -openmp=noopt** のように、最適化レベルを明示的に **-x03** よりも下げると、コンパイラはエラーを出力します。 **-openmp=noopt** を使用して最適化レベルを指定しない場合、OpenMP プラグマが認識され、プログラムが並列化されますが、最適化は行われません。
-xopenmp=stubs	このオプションはサポートされていません。 OpenMP スタブライブラリは、ユーザーの便宜上の理由で提供されています。 OpenMP ライブラリルーチンを呼び出しても OpenMP プラグマを無視するような OpenMP プログラムをコンパイルするには、**-xopenmp** オプションを指定しないでコンパイルします。その後、**libompstubs.a** ライブラリを使ってオブジェクトファイルをリンクします。 次に例を示します。**% cc omp_ignore.c -lompstubs** **libompstubs.a** と OpenMP 実行時ライブラリの **libmtsk.so** 両方とのリンクはサポートされていません。両方とリンクすると、予期しない動作になることがあります。
-xopenmp=none	OpenMP プラグマの認識を無効にし、最適化レベルを変更しません

その他の注

- コマンド行で **-xopenmp** を指定しない場合、コンパイラでは **-xopenmp=none** (OpenMP プラグマ の認識を無効にする) を指定したと見なされます。
- **-xopenmp** をキーワードサブオプションなしで指定した場合、コンパイラでは **-xopenmp=parallel** を指定したと見なされます。
- **-xopenmp=parallel** または **-xopenmp=noopt** を指定すると、**_OPENMP** プリプロセッサトークンが YYYYMM (C/C++ では **200805L**、Fortran 95 では **200805**) として定義されます。
- **dbx** を使用して OpenMP プログラムをデバッグする場合、**-xopenmp=noopt -g** を使用してコンパイルします。
- **-xopenmp** のデフォルトの最適化レベルは、将来のリリースで変更される可能性があります。適切な最適化レベルを明示的に指定することによって、コンパイル警告メッセージの表示を防止することができます。
- Fortran 95 では、**-xopenmp**、**-xopenmp=parallel**、**-xopenmp=noopt** を指定すると、**-stackvar** が自動的に追加されます。

- 別々の手順で OpenMP プログラムをコンパイルしてリンクする場合は、コンパイル時およびリンク時にそれぞれ **-xopenmp** を含めます。
- コンパイラの並列化メッセージを表示するには、**-xvpara** C/C++ オプションまたは **-vpara** Fortran 95 オプションを使用します。
- Solaris プラットフォーム上で最適なパフォーマンスおよび機能を得るために、最新の OpenMP 実行時ライブラリ **libmtsk.so** が実行中のシステムにインストールされていることを確認してください。

2.2 OpenMP 環境変数

OpenMP 仕様では、OpenMP プログラムの実行を制御する環境変数がいくつか定義されています。これらは、17 ページの「2.2.1 一般的な OpenMP 環境変数」にまとめてあります。詳細は、openmp.org にある OpenMP API Version 3.0 の仕様を参照してください。OpenMP 仕様に含まれないその他の環境変数は、Solaris Studio コンパイラのこのリリースで定義されています。それらについては、18 ページの「2.2.2 Solaris Studio 固有の環境変数」を参照してください。

2.2.1 一般的な OpenMP 環境変数

OMP_SCHEDULE

スケジュール型が **RUNTIME** として指定された **DO**、**PARALLEL DO**、**for**、**parallel for** の指令またはプラグマのスケジュール型を指定します。

定義しない場合は、デフォルト値の **STATIC** が使用されます。*value* は *"type[,chunk]"* という書式で指定します。

例: **setenv OMP_SCHEDULE 'GUIDED,4'**

OMP_NUM_THREADS

並列領域の実行中に使用するスレッド数を設定します。

この数は **num_threads** 節または **omp_set_num_threads** への呼び出しによって上書きできます。

設定しない場合は、デフォルト値の 1 が使用されます。*value* には必ず正の整数を指定します。

例: **setenv OMP_NUM_THREADS 16**

OMP_DYNAMIC

並列領域の実行で使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。

設定しない場合は、デフォルト値の **TRUE** が使用されます。*value* には、**TRUE** または **FALSE** を指定します。

例: **setenv OMP_DYNAMIC FALSE**

OMP_NESTED

入れ子並列性を有効または無効にします。

value には、**TRUE** または **FALSE** を指定します。

デフォルトは **FALSE** です。

例: **setenv OMP_NESTED FALSE**

OMP_STACKSIZE

OpenMP により作成されたスレッドのスタックサイズを設定します。

サイズは 通常 K バイト単位の正の整数で指定されます。また、接尾辞 **B**、**K**、**M**、もしくは **G** を付けて、それぞれバイト、K バイト、M バイト、または G バイト単位で指定することもあります。

例: **setenv OMP_STACKSIZE 10M**

OMP_WAIT_POLICY

待機中のスレッドに対して望まれるポリシー、**ACTIVE** または **PASSIVE** を設定します。

ACTIVE を指定すると、スレッドは待機中にも CPU 時間を消費します。 **PASSIVE** を指定すると、スレッドは CPU 時間を消費しないか、スリープ状態に入ります。

OMP_MAX_ACTIVE_LEVELS

入れ子になったアクティブな並列領域のレベルの最大数に非負の整数値を設定します。

OMP_THREAD_LIMIT

OpenMP プログラム全体で使用するスレッド数に正の整数を設定します。

2.2.2 Solaris Studio 固有の環境変数

これ以外にも、OpenMP プログラムの実行に影響を与える多重処理に関する環境変数がありますが、OpenMP 仕様には含まれていません。

PARALLEL

従来のプログラムとの互換性のため、**PARALLEL** 環境変数を設定すると、**OMP_NUM_THREADS** を設定したのと同じ効果が得られます。ただし、**PARALLEL** と **OMP_NUM_THREADS** の両方に設定する場合は、同じ値を設定する必要があります。

SUNW_MP_WARN

OpenMP の実行時ライブラリで出力される警告メッセージを制御します。**SUNW_MP_WARN** が **TRUE** に設定されている場合、実行時ライブラリは **stderr** に警

告メッセージを発行します。さらに、実行時ライブラリは、情報を提供するために、すべての環境変数の設定を出力します。環境変数が **FALSE** に設定されている場合、実行時ライブラリは警告メッセージを発行せず、設定も出力しません。デフォルトは **FALSE** です。

OpenMP 実行時ライブラリは、不正な入れ子やデッドロックなど、共通の OpenMP 違反を調べることができます。ただし、実行時チェックを使用するとプログラムの実行時にオーバーヘッドが加わります。第 3 章「実装によって定義される動作」を参照してください。**SUNW_MP_WARN** を **TRUE** に設定している場合、実行時ライブラリは **stderr** に警告メッセージを出力します。

次に例を示します。

```
setenv SUNW_MP_WARN TRUE
```

また、警告メッセージを認証するためにプログラムでコールバック関数が登録されている場合も、実行時ライブラリは警告メッセージを出力します。次の関数を呼び出すことにより、プログラムでユーザーコールバック関数を登録できます。

```
int sunw_mp_register_warn (void (*func)(void *));
```

コールバック関数のアドレスは、**sunw_mp_register_warn()** に引数として渡されます。この関数は、コールバック関数の登録に成功した場合 0 を、失敗した場合 1 を返します。

プログラムでコールバック関数が登録されている場合、**libmtsk** はエラーメッセージを含むローカライズされた文字列にポインタを渡し、登録済みの関数を呼び出します。メモリーの割り当て先は、コールバック関数から戻ると無効になります。

注 – プログラムのテストやデバッグを行うときは、**SUNW_MP_WARN** を **TRUE** に設定してください。これによって、OpenMP 実行時ライブラリからの警告メッセージが表示されるようになります。

SUNW_MP_THR_IDLE

OpenMP プログラムのバリアで待ち状態または新しい並列領域の実行待ち状態のアイドルスレッドの状態を制御します。次のいずれかの値を設定できます。**SPIN**、**SLEEP**、**SLEEP(***time***s)**、**SLEEP(***time***ms)**、または **SLEEP(***time***mc)**。*time* には時間を整数で指定し、**s**、**ms**、および **mc** には時間の単位 (それぞれ、秒、ミリ秒、マクロ秒) を指定します。

SPIN を指定すると、アイドルスレッドは、バリアで待機中または新しい並列領域の実行待ちの間スピンをします。時間引数なしで **SLEEP** を指定すると、アイドルスレッドはすぐにスリープ状態になります。時間引数付きで **SLEEP** を指定すると、スレッドは指定した時間スピンを継続し、そのあとスリープ状態になります。

デフォルトのアイドルスレッド状態は、スリープ状態ですが、ある程度の時間スピンして待機したあとスリープ状態になることがあります。**SLEEP**、**SLEEP(0)**、**SLEEP(0s)**、**SLEEP(0ms)**、および **SLEEP(0mc)** はすべて同じです。

次に例を示します。

```
setenv SUNW_MP_THR_IDLE SPIN
setenv SUNW_MP_THR_IDLE SLEEP
setenv SUNW_MP_THR_IDLE SLEEP(2s)
setenv SUNW_MP_THR_IDLE SLEEP(20ms)
setenv SUNW_MP_THR_IDLE SLEEP(150mc)
```

SUNW_MP_PROCBIND

この環境変数は、Solaris および Linux システムの両方で使用されます。**SUNW_MP_PROCBIND** 環境変数を使用して OpenMP プログラムのスレッドを実行中のシステムの仮想プロセッサに結合できます。プロセッサに結合することでパフォーマンスを向上することができますが、同じ仮想プロセッサに複数のスレッドが結合されると、パフォーマンスが低下します。詳細は、24 ページの「2.3 プロセッサ結合」を参照してください。

SUNW_MP_MAX_POOL_THREADS

スレッドプールの最大数を指定します。プールにあるのは、OpenMP ライブラリが作成した非ユーザースレッドだけです。マスタースレッドやユーザーのプログラムが明示的に作成したスレッドは含まれません。この環境変数をゼロに設定すると、スレッドのプールは空になり、すべての並列領域は 1 つのスレッドによって実行されます。指定しない場合のデ

フォルトは、1023 です。詳細は、38 ページの「4.2 入れ子並列処理の制御」を参照してください。

SUNW_MP_MAX_POOL_THREADS はプログラム全体で使用される非ユーザーの OpenMP スレッドの最大数を指定し、**OMP_THREAD_LIMIT** はプログラム全体で使用されるユーザーおよび非ユーザーの OpenMP スレッドの最大数を指定するものである点に留意してください。**SUNW_MP_MAX_POOL_THREADS** と **OMP_THREAD_LIMIT** の両方が設定されている場合は、両社の値の間に一貫性が保たれている必要があります。たとえば、**OMP_THREAD_LIMIT** の値は **SUNW_MP_MAX_POOL_THREADS** の値よりも 1 だけ大きい数値に設定します。

SUNW_MP_MAX_NESTED_LEVELS

有効な入れ子になった並列領域の深さの最大数を指定します。この環境変数で指定した数を超える有効な入れ子を持つ並列領域は、1 つのスレッドによって実行されます。**if** 節が False になっている OpenMP 並列領域の場合は、その並列領域は無効であると見なされます。指定しない場合のデフォルトは、4 です。詳細は、38 ページの「4.2 入れ子並列処理の制御」を参照してください。

SUNW_MP_MAX_NESTED_LEVELS と **OMP_MAX_ACTIVE_LEVELS** の両方を設定する場合は、両者は同じ値に設定される必要がある点に留意してください。

STACKSIZE

各スレッドのスタックサイズを設定します。値はキロバイト単位で指定します。デフォルトのスレッドスタックサイズは、32 ビット SPARC V8 および x86 プラットフォームで 4M バイト、64 ビット SPARC V9 および x86 プラットフォームで 8M バイトです。

次に例を示します。

setenv STACKSIZE 8192 スタックサイズを *8 MB* に設定します。

STACKSIZE 環境変数は、接尾辞 **B**、**K**、**M**、または **G** の付いた数値も受け付けます。これらの接尾辞はそれぞれ、バイト、キロバイト、メガバイト、ギガバイトを表します。デフォルトはキロバイトです。

STACKSIZE と **OMP_STACKSIZE** の両方を設定する場合は、両者は同じ値に設定される必要がある点に留意してください。

SUNW_MP_GUIDED_WEIGHT

チャンクのサイズを決定する重み係数を設定します。このチャンクサイズは、**GUIDED** スケジューリングによってループ中のスレッドに割り当てられます。値には、正の浮動小数点数を指定します。この値は、同一プログラム中で **GUIDED** スケジューリングが設定されたループすべてに適用されます。指定がない場合のデフォルト値は 2.0 です。

SUNW_MP_WAIT_POLICY

作業待ち状態 (アイドル)、バリアーで待ち状態、またはタスク待ち状態のスレッドの動作を制御します。これらの待ち状態の動作には、しばらくの間スピンする、しばらくの間 CPU 資源を明け渡す、または呼び起こされるまで休眠状態になる、の 3 つがあります。

構文は次のとおりです (csh で表示)。

setenv SUNW_MP_WAIT_POLICY IDLE=*val*
:BARRIER=*val***:TASKWAIT=** *val*

IDLE=*val*、**BARRIER=** *val*、および **TASKWAIT=** *val* は、制御される待機の種類を指定する、オプションのキーワードです。

これらのキーワードに対し、それぞれ *val* 設定があり、これによって待機中の動作、つまり **SPIN**、**YIELD**、または **SLEEP** が指定されます。

SPIN(*time***)** は、CPU 資源を明け渡すまでに、スレッドがスピンする時間を指定します。*time* は秒、ミリ秒、またはマイクロ秒で指定します (それぞれ **s**、**ms**、および **mc** で指定します)。時間単位が指定されていなければ、秒が使用されます。**SPIN** に時間パラメータが設定されていなければ、待機中のスレッドは継続的にスピンします。

YIELD(*number***)** は、休眠状態になる前にスレッドが CPU 資源を明け渡す回数を指定します。CPU 資源が明け渡されたあと、オペレーティングシステムの実行予定時間になると、スレッドは再度実行されま

す。YIELD に *number* パラメータが指定されていなければ、待機中のスレッドは継続的に CPU 資源を明け渡します。

SLEEP は、スレッドをただちに休眠状態にすることを指定します。

特定の種類の待ち状態に対し、**SPIN**、**SLEEP**、および **YIELD** 設定は任意の順序で指定できます。設定はコンマで区切ります。**"SPIN(0),YIELD(0)"** は、**SLEEP**、つまり、即座に休眠状態にするのと同じです。**IDLE**、**BARRIER**、および **TASKWAIT** の設定を処理する場合、「一番左を最優先」規則が適用されます。

次に例を示します。

```
% setenv SUNW_MP_WAIT_POLICY “BARRIER=SPIN”
```

バリアーで待ち状態のスレッドは、チーム内のすべてのスレッドがバリアーに到達するまでスピンします。

```
% setenv SUNW_MP_WAIT_POLICY
“IDLE=SPIN(10ms),YIELD(5)”
```

作業待ち状態 (アイドル) のスレッドは 10 ミリ秒スピンし、休眠状態になるまで CPU 資源を 5 回明け渡します。

```
% setenv SUNW_MP_WAIT_POLICY
“IDLE=SPIN(10ms),YIELD(2):BARRIER=SLEEP:TASKWAIT=YIELD(1
```

作業待ち状態 (アイドル) のスレッドは 10 ミリ秒スピンし、休眠状態になるまで CPU 資源を 2 回明け渡します。バリアーで待ち状態のスレッドはただちに休眠状態になります。taskwait で待ち状態のスレッドは、CPU 資源を 10 回明け渡してから休眠状態になります。

2.3 プロセッサ結合

プロセッサ結合では、プログラマはプログラムの実行を通じてプログラム内のスレッドを同じプロセッサで実行すべきであることを、オペレーティングシステムに指示します。

static スケジュール指定とともにプロセッサ結合を使用すると、並列領域またはワークシェアリング領域の前回呼び出し以降、その領域内のスレッドがアクセスするデータがローカルキャッシュに存在する、特定のデータ再利用パターンを持つアプリケーションにメリットがあります。

ハードウェアから見ると、コンピュータシステムは 1 つまたは複数の「物理」プロセッサから構成されています。オペレーティングシステムから見ると、これらの「物理」プロセッサはそれぞれ、プログラム内のスレッドを実行可能な 1 つまたは複数の「仮想」プロセッサにマッピングされます。*n* 個の仮想プロセッサを使用する場合、*n* 個のスレッドを同時に実行するようスケジューリングできます。システムによっては、仮想プロセッサは実際のプロセッサやコアなどの場合もあります。

たとえば、UltraSPARC T2 物理プロセッサには 8 つのコアがあり、各コアはスレッドを 8 つ同時に処理できます。Solaris OS から見ると、スレッドの実行をスケジューリング可能な仮想プロセッサは 64 個あります。Solaris プラットフォームでは、**psrinfo**(1M) コマンドを使用して仮想プロセッサの数を特定できます。Linux システムでは、ファイル `/proc/cpuinfo` に利用可能なプロセッサの情報が記述されています。

オペレーティングシステムがスレッドをプロセッサに結合すると、スレッドは実質的に「物理」プロセッサではなく、特定の「仮想」プロセッサに結合されます。

OpenMP プログラム内のスレッドを特定の仮想プロセッサに結合するには、**SUNW_MP_PROCBIND** 環境変数を設定します。**SUNW_MP_PROCBIND** には、次のいずれかの値を指定できます。

- 文字列「**TRUE**」または「**FALSE**」(小文字の「**true**」または「**false**」も可)。
  たとえば、次を見てください。
  % setenv SUNW_MP_PROCBIND "false"
- 非負整数。
  次に例を示します。**% setenv SUNW_MP_PROCBIND "2"**
- 1 つ以上の空白で区切った 2 つ以上の非負整数のリスト。
  次に例を示します。**% setenv SUNW_MP_PROCBIND "0 2 4 6"**
- ハイフン 1 つ ("-") で区切った 2 つの非負整数 *n1* と *n2*。*n1* は *n2* 以下である必要があります。
  次に例を示します。**% setenv SUNW_MP_PROCBIND "0-6"**

SUNW_MP_PROCBIND によって許可される値の解釈については、26 ページの「2.3.3 **SUNW_MP_PROCBIND** に指定された値の解釈」を参照してください。

前述の非負整数は論理識別子 (ID) を表しています。論理 ID は「仮想」プロセッサ ID とは異なります。その違いを次に示します。

2.3.1 仮想プロセッサ ID

システム内の各仮想プロセッサは一意のプロセッサ ID を持ちます。Solaris OS の **psrinfo**(1M) コマンドを使用すると、システム内のプロセッサに関するプロセッサ ID などの情報を表示できます。さらに、**prtdiag**(1M) コマンドを使用すると、システム構成および診断情報を表示できます。

Solaris の最近のリリースでは **psrinfo -pv** を使用すると、システム内のすべての物理プロセッサ、および各物理プロセッサに関連付けられた仮想プロセッサを一覧表示できます。

仮想プロセッサ ID は、連番になることも、ID 番号が飛ぶこともあります。たとえば、8 個の UltraSPARC IV プロセッサ (16 コア) を持つ Sun Fire 4810 では、仮想プロセッサ ID が 0、1、2、3、8、9、10、11、512、513、514、515、520、521、522、523 のようになります。

2.3.2 論理 ID

前述のとおり、**SUNW_MP_PROCBIND** に指定された非負整数は論理 ID です。論理 ID は、0 から始まる連続した整数です。システムで利用可能な仮想プロセッサの数が *n* の場合、論理 ID は、0、1、…、*n*-1 のように **psrinfo**(1M) に示された順番になります。次の Korn シェルスクリプトを使用すると、仮想プロセッサ ID から論理 ID へのマッピングを表示できます。

```
#!/bin/ksh

NUMV=` psrinfo | fgrep "on-line" | wc -l`
set -A VID ` psrinfo | cut -f1`

echo "Total number of on-line virtual processors = $NUMV"
echo

let "I=0"
let "J=0"
while [[ $I -lt $NUMV ]]
do
  echo "Virtual processor ID ${VID[I]} maps to logical ID ${J}"
  let "I=I+1"
  let "J=J+1"
done
```

1 つの物理プロセッサが複数の仮想プロセッサにマッピングされているシステムでは、同じ物理プロセッサに属す仮想プロセッサにどの論理 ID が対応しているかを知っておくと便利です。次の Korn シェルスクリプトを最近のリリースの Solaris で使用すると、この情報が表示されます。

```
#!/bin/ksh

NUMV=` psrinfo | grep "on-line" | wc -l`
set -A VLIST ` psrinfo | cut -f1`
set -A CHECKLIST ` psrinfo | cut -f1`

let "I=0"

while [ $I -lt $NUMV ]
do
  let "COUNT=0"
  SAMELIST="$I"

  let "J=I+1"

  while [ $J -lt $NUMV ]
  do
    if [ ${CHECKLIST[J]} -ne -1 ]
    then
      if [ ` psrinfo -p ${VLIST[I]} ${VLIST[J]}`  = 1 ]
      then
    SAMELIST="$SAMELIST $J"
    let "CHECKLIST[J]=-1"
    let "COUNT=COUNT+1"
      fi
    fi
    let "J=J+1"
  done

  if [ $COUNT -gt 0 ]
  then
    echo "The following logical IDs belong to the same physical processor:"
    echo "$SAMELIST"
    echo " "
  fi

  let "I=I+1"
done
```

2.3.3 SUNW_MP_PROCBIND に指定された値の解釈

SUNW_MP_PROCBIND に指定された値が **TRUE** の場合、スレッドは、ラウンドロビン方式で仮想プロセッサに結合されます。結合を行う際の最初のプロセッサは、最適なパフォーマンスを実現するために、実行時ライブラリによって決定されます。

SUNW_MP_PROCBIND に指定された値が **FALSE** の場合、スレッドはどのプロセッサにも結合されません。これはデフォルト設定です。

SUNW_MP_PROCBIND に指定された値が非負整数の場合、その整数はスレッドの結合先の仮想プロセッサの開始論理 ID を表します。スレッドは、指定された論理 ID を持つプロセッサから順にラウンドロビン方式で仮想プロセッサに結合されます。論理 ID が *n-1* のプロセッサに結合した後は、論理 ID が 0 のプロセッサに続きます。

SUNW_MP_PROCBIND に指定された値が非負の整数の 2 項目以上のリストの場合、スレッドはラウンドロビン方式で指定された論理 ID を持つ仮想プロセッサに結合されます。指定された以外の論理 ID を持つプロセッサは使用されません。

SUNW_MP_PROCBIND に指定された値が、マイナス記号 (-) で区切られた 2 つの非負整数の場合、スレッドは最初の論理 ID から 2 番目の論理 ID の範囲の仮想プロセッサに、ラウンドロビン式で結合されます。この範囲に含まれない論理 ID を持つプロセッサは使用されません。

SUNW_MP_PROCBIND に指定された値が前述のどの形式にも当てはまらないか、不正な論理 ID が指定された場合は、エラーメッセージが出力され、プログラムの実行が終了します。

OpenMP 実行時ライブラリ **libmtsk** で作成されるスレッドの数は、環境変数、ユーザーのプログラム内の API 呼び出し、および **num_threads** 節によって異なります。**SUNW_MP_PROCBIND** は、スレッドの結合先となる仮想プロセッサの論理 ID を指定します。スレッドは、その一連のプロセッサにラウンドロビン式で結合されます。プログラム内で使用しているスレッドの数が、**SUNW_MP_PROCBIND** で指定された論理 ID の数よりも少ない場合、一部の仮想プロセッサはそのプログラム内で使用されません。**SUNW_MP_PROCBIND** で指定された論理 ID の数よりもスレッドの数が多い場合、一部の仮想プロセッサには複数のスレッドが結合されます。

2.3.4 OS プロセッサセットとの相互作用

Solaris プラットフォーム上では **psrset** ユーティリティーを、Linux プラットフォーム上では **taskset** コマンドを使うと、プロセッサセットを指定できます。**SUNW_MP_PROCBIND** ではプロセッサセットは考慮されません。プログラマがプロセッサセットを使用する場合、**SUNW_MP_PROCBIND** の設定と、使用しているプロセッサセットとの整合性の確認は、プログラマの責任で行ってください。この確認を怠ると、Linux システム上では **SUNW_MP_PROCBIND** の設定がプロセッサセットの設定に上書きされ、Solaris システム上ではエラーメッセージが表示されます。

2.4 スタックとスタックサイズ

実行プログラムは、各スレーブスレッド用の個別スタックのほか、プログラムを実行する初期 (またはメイン) スレッド用のメインスタックを保持します。スタックは、サブプログラムまたは関数参照の呼び出し中、引数および自動変数を保持するために使用される一時的なメモリーアドレス空間です。

デフォルトのメインスタックのサイズは 8M バイトです。**f95** にオプション **-stackvar** を指定して Fortran プログラムをコンパイルすると、自動変数であるかのようにスタック上にローカル変数と配列が割り当てられます。OpenMP プログラムでの **-stackvar** 指定は、明示的に並列化されたプログラムで必要になります。これは、オプティマイザのループでの呼び出しの並列化機能を向上させるためです (**-stackvar** フラグについては、『Fortran ユーザーズガイド』を参照)。ただし、スタックに十分なメモリーが割り当てられていない場合は、スタックのオーバーフローが発生する可能性があります。

メインスタックのサイズを表示または設定するには、C シェルの **limit** コマンド、または ksh、sh の **ulimit** コマンドを使用します。

OpenMP プログラムの各スレーブスレッドは、それぞれスレッドスタックを持ちます。このスタックは最初の (メイン) スレッドスタックに似ていますが、そのスレッドに固有のものです。スレッドの **PRIVATE** 配列および変数 (スレッドにローカル) は、スレッドスタックに割り当てられます。デフォルトのサイズは、32 ビット SPARC V8 および x86 プラットフォームで 4M バイト、64 ビット SPARC V9 および x86 プラットフォームで 8M バイトです。スレーブスレッドスタックのサイズは、**OMP_STACKSIZE** 環境変数で設定されます。

```
demo% setenv OMP_STACKSIZE 16384     <-Set thread stack size to 16 Mb (C shell)

demo$ OMP_STACKSIZE=16384            <-Same, using Bourne/Korn shell
demo$ export OMP_STACKSIZE
```

最適なスタックサイズを判定するには、試行とエラーを経る必要があるかもしれません。スタックサイズがスレッドに対して小さすぎて実行できない場合、エラーメッセージが出力されないまま、データ破壊やセグメントエラーが発生する可能性があります。スタックオーバーフローが発生するかどうか不確かな場合、**-xcheck=stkovf** コンパイラオプションを指定して Fortran や C、C++ プログラムをコンパイルすると、スタックオーバーフローのセグメント例外を発生させることができます。この場合、データ破壊が発生する前にプログラムの実行が停止します

2.5 OpenMP プログラムの確認と分析

Solaris Studio スレッドアナライザツールを使用して、OpenMP プログラムのデータ競合やデッドロックをチェックできます。詳細は、スレッドアナライザのマニュアルおよび **tha**(1) のマニュアルページを参照してください。

Solaris Studio パフォーマンスアナライザを使用し、OpenMP プログラムのパフォーマンスを分析できます。詳細は、パフォーマンスアナライザのマニュアルか、**collect**(1) および **analyzer**(1) マニュアルページを参照してください。

◆ ◆ ◆ 第 3 章

3 実装によって定義される動作

この章では、OpenMP 3.0 仕様の、実装に依存する特定の動作について説明します。

3.1 タスクスケジューリングポイント

拘束を受けていないタスク領域にあるタスクスケジューリングポイントは、拘束されたタスク領域と同じポイントで発生します。このため、拘束を受けていないタスク領域内では、タスクスケジューリングポイントは次の場所にのみ現れます。

- 検出されたタスク構造
- 検出された taskwait 構文
- 検出されたバリアー指令
- 暗黙のバリアー領域
- 拘束を受けていないタスク領域の最後

3.2 メモリーモデル

複数スレッドから同じ変数への非同期のメモリーアクセスが、互いのそれぞれのアクセスに対して不可分なものになる保証はありません。アクセスが不可分にものなるかどうかは、実装依存、およびアプリケーション依存の要因による影響を受けます。変数によっては、対象プラットフォームでの最大の不可分なメモリー動作よりも大きい場合があります。変数によっては、不正な境界整列が行われていたり境界が不明である場合があります。その場合、コンパイラまたは実行時システムがその変数にアクセスするためには、複数回の読み込みおよび格納が必要になることがあります。より多くの読み込みと格納を使用する、高速なコードシーケンスもあります。

3.3 内部制御変数

次の内部制御変数は、実装時に定義されます。

- *nthreads-var*: 検出された並列領域に対して要求されたスレッド数を制限します。*nthreads-var* の初期値は 1 です。
- *dyn-var*: 検出された並列領域に対してスレッド数の動的な調整を有効にするかどうかを制御します。*dyn-var* の初期値は `TRUE` (動的な調整が有効) です。
- *run-sched-var*: 実行時スケジュール節がループ領域に使用するスケジュールを制御します。*run-sched-var* の初期値は、`static` で、チャンクサイズはありません。
- *def-sched-var*: ループ領域のデフォルトスケジューリングとして定義された実装を制御します。*def-sched-var* の初期値は、`static` で、チャンクサイズはありません。
- *stacksize-var*: OpenMP の実装が作成するスレッドのスタックサイズを制御します。*stacksize-var* の初期値は、32 ビットアプリケーションでは 4M バイト、64 ビットアプリケーションでは 8M バイトです。
- *wait-policy-var*: 待ち状態にあるスレッドの動作を制御します。*wait-policy-var* の初期値は、`PASSIVE` です。
- *thread-limit-var*: OpenMP プログラムに所属するスレッドの最大数を制御します。thread-limit-var の初期値は 1024 です。
- *max-active-levels-var*: アクティブな入れ子の並列領域の最大数を制御します。*max-active-levels-var* の初期値は 4 です。

3.4 スレッドの動的調整

実装後には、スレッド数を動的に調整する機能があります。動的調整の機能はデフォルトで有効に設定されています。**`OMP_DYNAMIC`** 環境変数を **`FALSE`** に設定するか、引数を適切に指定して **`omp_set_dynamic()`** ルーチンを呼び出して、動的調整を無効にします。

スレッドが並列構文を検出したときにこの実装により提供されるスレッド数は、OpenMP 3.0 仕様の 35 - 36 ページに記載されたアルゴリズム 2.1 に従って決定されます。システムリソースの不足時などの例外的な状況では、与えられるスレッドの数はアルゴリズム 2.1 で求めた数よりも少なくなることがあります。このような状況では、 **`SUNW_MP_WARN`** が **`TRUE`** に設定されているか、コールバック関数が **`sunw_mp_register_warn()`** の呼び出しにより登録されている場合でも、警告メッセージが表示されます。

3.5 ループ指令

収縮されたループの繰り返し回数の計算に使われる整数型は **long** です。

run-sched-var 内部制御変数が *auto* に設定されているときの **schedule(runtime)** 節の効果は、チャンクサイズを指定しないときの static と同じです。

3.6 コンストラクト

3.6.1 SECTIONS

コンストラクトの節にある構造化ブロックは、チャンクサイズが指定されていない形式の static 状態のチームに含まれるスレッドに割り当てられます。そのため、各スレッドはほぼ同数の連続する構造化ブロックを取得します。

3.6.2 SINGLE

singleコンストラクトを検出した最初のスレッドが、コンストラクトを実行します。

3.6.3 ATOMIC

critical コンストラクトと名付けられた特別なコンストラクトを持つターゲット文を挿入することにより、実装によってすべての **atomic** 指令が置き換えられます。これにより、プログラム中のすべての atomic 領域間で排他的なアクセスが行われるようになります。これらの領域が同じストレージロケーションを更新するのか、異なるロケーションを更新するのかは関係ありません。

3.7 ルーチン

3.7.1 omp_set_schedule()

Solaris Studio 特有の **sunw_mp_sched_reserved** スケジュールの動作も、チャンクサイズが指定されていない static の場合と同様です。

3.7.2 omp_set_max_active_levels()

omp_set_max_active_levels() がアクティブな並列領域の内部から呼び出された場合、その呼び出しは無視されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。

omp_set_max_active_levels() の引数が負の整数の場合、呼び出しが無視されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。

3.7.3 omp_get_max_active_levels()

omp_get_max_active_levels() プログラムのどこからでも呼び出すことができます。この呼び出しによって、内部制御変数 *max-active-levels-var* の値が返されます。

3.8 環境変数

変数名	実装
OMP_SCHEDULE	**OMP_SCHEDULE** に指定されたスケジュール型が有効な型 (**static**、**dynamic**、**guided**、または **auto**) ではない場合、環境変数が無視され、デフォルトのスケジュール (チャンクサイズの指定されていない **static** 型) が使用されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。 **OMP_SCHEDULE** 環境変数に指定されたスケジュール型が **static**、**dynamic**、または **guided** であるが、チャンクに指定されたサイズが負の整数の場合、次のようにチャンクサイズが設定されます。**static** の場合には、チャンクサイズは設定されません。**dynamic** または **guided** の場合は、チャンクサイズは 1 になります。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。

変数名	実装
OMP_NUM_THREADS	変数の値が正の整数ではない場合、環境変数は無視されます。また、**SUNW_MP_WARN** が TRUE に設定されているか、**sunw_mp_register_warn()** の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。 変数の値が、実装でサポートしているスレッド数よりも大きい場合は、次のアクションが実行されます。 - スレッド数の動的調整が有効になっている場合は、スレッド数が減少します。また、**SUNW_MP_WARN** が TRUE に設定されているか、**sunw_mp_register_warn()** の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。 - 一方、スレッド数の動的調整が無効になっている場合は、エラーメッセージが表示され、プログラムが停止します。
OMP_DYNAMIC	**OMP_DYNAMIC** に指定された値が TRUE でも FALSE でもない場合は、その値は無視され、デフォルト値である TRUE が使用されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。
OMP_NESTED	**OMP_NESTED** に指定された値が TRUE でも FALSE でもない場合は、その値は無視され、デフォルト値である FALSE が使用されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。
OMP_STACKSIZE	**OMP_STACKSIZE** に指定された値が指定書式に従っていない場合は、その値は無視され、デフォルト値 (32 ビットアプリケーションでは 4M バイト、64 ビットアプリケーションでは 8M バイト) が使用されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。
OMP_WAIT_POLICY	スレッドの ACTIVE の動作は、スピンです。スレッドの PASSIVE の動作は、少しの間スピンした後でのスリープです。
OMP_MAX_ACTIVE_LEVELS	**OMP_MAX_ACTIVE_LEVELS** に指定された値が負の整数の場合、その値は無視され、デフォルト値 (4) が使用されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。
OMP_THREAD_LIMIT	**OMP_THREAD_LIMIT** に指定された値が正の整数ではない場合、その値は無視され、デフォルト値 (1024) が使用されます。**SUNW_MP_WARN** が TRUE に設定されている場合、またはコールバック関数が **sunw_mp_register_warn()** の呼び出しにより登録されている場合は、警告メッセージが表示されます。

3.9 Fortran の問題

次の事項は、Fortran の場合にのみ適用されます。

3.9.1 THREADPRIVATE 指令

2 つの連続した有効な並列領域間で維持される、スレッド (最初のスレッド以外) の threadprivate オブジェクト内のデータの値の条件がすべては保持されない場合、2 番目の領域の割り当て可能な配列の割り当て状態が「not currently allocated」になることがあります。

3.9.2 SHARED 節

共有変数を組み込み以外の手続きに渡すと、手続きで参照する前に共有変数の値が一時ストレージにコピーされ、手続きでの参照後に一時ストレージが実際の引数ストレージに戻されるることがあります。一時ストレージとの間のコピーは OpenMP 3.0 仕様の節 2.9.3.2 (88 ページ) の条件 a、b、および c が適用される場合に起こります。すなわち、

- 実際の引数は、次のいずれかです。
  - 共有変数
  - 共有変数のサブオブジェクト
  - 共有変数と関連づけられたオブジェクト
  - 共有変数のサブオブジェクトと関連づけられたオブジェクト
- 実際の引数は、次のいずれかの場合もあります。
  - 部分配列
  - ベクトル添字のある部分配列
  - 形状引き継ぎ配列
  - ポインタ配列
- この実際の引数に関連づけられたダミー引数は、形状明示配列または形状引き継ぎ配列です。

3.9.3 実行時ライブラリの定義

この実装では、インクルードファイル **omp_lib.h** とモジュールファイル **omp_lib** の両方が提供されます。

Solaris プラットフォームでは、引数をとる OpenMP 実行時ライブラリルーチンが generic インタフェースで拡張されたため、異なる Fortran の **KIND** 型の引数に対応できます。

◆ ◆ ◆ 第 4 章

4 入れ子並列処理

この章では、OpenMP の入れ子並列処理について説明します。

4.1 実行モデル

OpenMP は並列実行の fork-join モデルを使用しています。スレッドは並列構文を検出すると、自身を含めほかのスレッドとチームを構成します (ほかのスレッドがまったくないこともあります)。並列構文を検出したスレッドは、このチームのマスタースレッドとなり、チーム内のその他のスレッドは、スレーブスレッドとなります。すべてのスレッドは、並列構文内のコードを実行します。各スレッドは並列構文内での処理を終了すると、その並列構文の最後にある暗黙バリアで待ち状態となります。チーム内のすべてのスレッドがバリアで待ち状態に入れば、スレッドは解放されます。マスタースレッドだけは並列構文の処理が終了したあとも続けてユーザーコードを実行しますが、スレーブスレッドは今度は別のチームを構成するための呼び出しの待ち状態に入ります。

OpenMP での並列領域は、互いに入れ子にすることができます。スレッドが並列領域内で並列構文を検出してチームを作成する際に、入れ子並列処理が無効になっていると、チームに含まれるスレッドは並列構文を検出したスレッドだけとなります。入れ子並列処理が有効になっていれば、複数のスレッドでチームが作成されます。

OpenMP 実行時ライブラリにはスレッドがプールされていて、並列領域内でのスレーブスレッドとして使用されます。あるスレッドが並列構文の検出時に複数のスレッドで構成されるチームを作成する必要がある場合は、そのスレッドは、最初にプールを調べてアイドル状態のスレッドを選択し、自身のチームのスレーブスレッドにします。このとき、充分な数のアイドル状態のスレッドがプールにないと、マスタースレッドが取得できるスレーブスレッドの数は必要な数を満たさないこともあります。チームが並列領域での処理を完了すると、スレーブスレッドはプールに返されます。

4.2 入れ子並列処理の制御

入れ子並列処理は、プログラムの実行前にさまざまな環境変数を設定することでその実行を制御できます。

4.2.1 OMP_NESTED

入れ子並列処理は、**OMP_NESTED** 環境変数を設定するか **omp_set_nested()** を呼び出すことで有効または無効に設定できます。

3 つのレベルを持つ、入れ子並列構文の例を次に示します。

例 **4–1** 入れ子並列処理の例

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team - %d\n",
                  level, omp_get_num_threads());
    }
 }
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

入れ子並列処理を有効にして、このプログラムをコンパイルおよび実行すると、次のようなソート済みの結果が出力されます。

```
% setenv OMP_NESTED TRUE
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

```
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

入れ子並列処理を無効にして同じプログラムを実行した場合と比べてみましょう。

```
% setenv OMP_NESTED FALSE
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

4.2.2 OMP_THREAD_LIMIT

OpenMP 実行時ライブラリにはスレッドがプールされていて、並列領域内でのスレーブスレッドとして使用されます。**OMP_THREAD_LIMIT** 環境変数の設定は、プール内のスレッド数を制御します。デフォルトでは、プール内のスレッド数は最大で 1023 個です。

プールにあるのは、実行時ライブラリが作成した非ユーザースレッドだけです。最初のスレッドやユーザーのプログラムが明示的に作成したスレッドは含まれません。

OMP_THREAD_LIMIT が 1 に設定されている場合 (または **SUNW_MP_MAX_POOL_THREADS** がゼロに設定されている場合)、スレッドプールは空になり、すべての並列領域は 1 つのスレッドによって実行されます。

次の例では、プール内のスレッド数が不十分な場合、並列領域で取得されるスレッド数も少なくなることを示しています。ここでも、前の例と同じコードを使用します。アクティブ化されるすべての並列領域に必要な同時スレッドの数は、8 個です。つまり、プールには少なくとも 7 個のスレッドが含まれている必要があります。**OMP_THREAD_LIMIT** を 6 に設定した場合 (または **SUNW_MP_MAX_POOL_THREADS** を 5 に設定した場合)、プールには最大で 5 個のスレーブスレッドが含まれます。これは、内側から数えて 4 つの並列領域のうち 2 つが、必要な数のスレーブスレッドを取得できないことを示します。実行結果はさまざまですが、1 つの例を見てみましょう。

```
% setenv OMP_NESTED TRUE
% OMP_THREAD_LIMIT 6
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

4.2.3 OMP_MAX_ACTIVE_LEVELS

環境変数 **OMP_MAX_ACTIVE_LEVELS** は、スレッドを複数必要とする有効な並列領域を何層まで入れ子にすることができるかを制御します。

この環境変数で指定した数を超える有効な入れ子を持つ有効な並列領域は、1 つのスレッドによって実行されます。並列領域が有効であると見なされるのは、**if** 節がない場合と、**if** 節の評価が *true* となるときです。有効な入れ子レベルのデフォルトの最大数は 4 です。

次に、4 重の入れ子になった並列領域のコードの例を示します。**OMP_MAX_ACTIVE_LEVELS** が 2 に設定されると、3 番目と 4 番目の深さにある入れ子並列領域は 1 つのスレッドによって実行されます。

```
#include <omp.h>
#include <stdio.h>
#define DEPTH 5
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team - %d\n",
               level, omp_get_num_threads());
    }
}
void nested(int depth)
{
    if (depth == DEPTH)
        return;

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(depth);
        nested(depth+1);
    }
}
int main()
{
    omp_set_dynamic(0);
    omp_set_nested(1);
    nested(1);
    return(0);
}
```

入れ子の深さの最大数を 4 に設定してこのプログラムをコンパイル、実行すると、次のような結果が出力されます。実際の結果は、OS がどのようにスレッドをスケジューリングしているかによって異なります。

```
% setenv OMP_MAX_ACTIVE_LEVELS 4
% a.out |sort
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
```

```
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
```

入れ子の深さを 2 に設定して実行した場合の結果は次のとおりです。

```
% setenv OMP_MAX_ACTIVE_LEVELS 2
% a.out |sort
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
```

この例は、可能性のある結果の一部のみを示しています。実際の結果は、OS がどのようにスレッドをスケジューリングしているかによって異なります。

4.3 入れ子並列領域での OpenMP ライブラリルーチンの使用

ここでは、入れ子並列領域内で次の OpenMP ルーチンを呼び出す実行について説明します。

```
- omp_set_num_threads()
- omp_get_max_threads()
- omp_set_dynamic()
- omp_get_dynamic()
- omp_set_nested()
- omp_get_nested()
```

「set」呼び出しは、呼び出しスレッドが検出した並列領域と同じレベルまたはその内側で入れ子になっている、呼び出し以降の並列領域に対してのみ有効です。ほかのスレッドが検出した並列領域には無効です。

「get」呼び出しは、呼び出しスレッドが設定した値を返します。スレッドが並列領域の実行時にチームのマスターになる場合は、チームのほかのすべてのメンバーは

マスタースレッドが持つ値を継承します。マスタースレッドが入れ子並列領域を終了し、その領域を取り囲む並列領域の実行を続ける場合、そのスレッドの値は、入れ子並列領域を実行する直前に、取り囲んでいる並列領域内での値に戻ります。

例 4–2 並列領域内での OpenMP ルーチンの呼び出し

```
#include <stdio.h>
#include <omp.h>
int main()
{
    omp_set_nested(1);
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num() == 0)
            omp_set_num_threads(4);       /* line A */
        else
            omp_set_num_threads(6);       /* line B */

        /* The following statement will print out
         *
         * 0: 2 4
         * 1: 2 6
         *
         * omp_get_num_threads() returns the number
         * of the threads in the team, so it is
         * the same for the two threads in the team.
         */
        printf("%d: %d %d\n", omp_get_thread_num(),
               omp_get_num_threads(),
               omp_get_max_threads());

        /* Two inner parallel regions will be created
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         */
        #pragma omp parallel
        {
            #pragma omp master
            {
                /* The following statement will print out
                 *
                 * Inner: 4
                 * Inner: 6
                 */
                printf("Inner: %d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* line C */
        }

        /* Again two inner parallel regions will be created,
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         *
         * The omp_set_num_threads(7) call at line C
         * has no effect here, since it affects only
         * parallel regions at the same or inner nesting
```

例 4–2 並列領域内での OpenMP ルーチンの呼び出し (続き)

```
         * level as line C.
         */

        #pragma omp parallel
        {
            printf("count me.\n");
        }
    }
    return(0);
}
```

このプログラムをコンパイル、実行すると次のような結果が出力されます。

```
% a.out
0: 2 4
Inner: 4
1: 2 6
Inner: 6
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
```

4.4 入れ子並列処理を使う際のヒント

- 並列領域を入れ子にすると、計算で使用できるスレッドの数を簡単に増やすことができます。

  たとえば、並列性に 2 つのレベルがあり、各レベルでの並列性の度合いが 2 であるプログラムがあるとしましょう。また、システムには CPU が 4 個搭載され、このプログラムの実行速度を上げるために、これら 4 つの CPU をすべて使用するものとします。どの段階であったとしても単に並列処理にしただけでは、使用する CPU は 2 つに留まります。入れ子によって両方の段階で処理が並列化されます。
- 並列領域を入れ子にするだけでは、スレッドばかりが増えてシステムへの要求が過剰になります。**OMP_THREAD_LIMIT** および **OMP_MAX_ACTIVE_LEVELS** を適切に設定して、使用されるスレッド数を制限し、過剰要求を回避します。
- 入れ子になった並列領域を作成すると負荷がかかります。外側の入れ子でも充分な並列処理が実行されていて、負荷が平均に分散されていれば、現在の処理より内側に入れ子の並列領域を作成するよりは、外側の入れ子で全スレッドを使用する方が一般的には効率的です。

たとえば、2 段階の並列処理となるプログラムがあったとします。外側の処理は 4 つの並列処理となっていて、負荷は平均に分散されています。システムには 4 つの CPU があり、すべての CPU を使用してプログラムの実行を高速化したいとします。この場合は、外側の並列処理で 4 つのスレッドのうち 2 つだけを使い、かつ、そのスレーブスレッドとして内側の並列処理で 2 つのスレッドを使うよりは、外側の並列処理で 4 つのスレッドすべてを使用した方が優れたパフォーマンスを得ることができます。

◆ ◆ ◆ 第 5 章

5 タスク化

この章では、OpenMP 3.0 のタスク化モデルについて説明します。

5.1 タスク化モデル

OpenMP 仕様のバージョン 3.0 では、タスク化と呼ばれる新しい機能が追加されました。タスク化の機能を利用すると、再帰的な構造や *while* ループのように、作業単位が動的に生成されるようなアプリケーションの並列化が容易になります。

OpenMP では、明示的タスクを **task** 指令を使って指定します。**task** 指令は、タスクとそのデータ環境に関連づけられたコードを定義します。タスクのコンストラクトは、プログラムのどこに置いても構いません。スレッドがタスクコンストラクトを検出すると、新しいタスクが生成されます。

スレッドがタスクコンストラクトを検出すると、スレッドはそれをすぐに実行するか、実行を延期して後で実行するかを選択することがあります。タスクの実行が保留されると、そのタスクは現在の並列領域に関連づけられた概念上のタスクプールに置かれます。現在のチームに属するスレッドは、プールからタスクを取り出し実行するという処理を、プールが空になるまで繰り返します。タスクを実行するスレッドは、タスクを検出した元のスレッドとは異なる場合があります。

タスクコンストラクトに関連づけられたコードは、1 回だけ実行されます。コードが最初から最後まで同じスレッドにより実行されると、タスクは結合されます。コードが複数のスレッドにより実行可能な場合は、タスクは結合解除されます。この場合、コードの異なる部分が別のスレッドにより実行されます。デフォルトでは、タスクは結合されていますが、結合解除節を **task** 指令で使用することにより、結合解除に指定することができます。

スレッドは、タスクのスケジューリングポイントでタスク領域の実行を中断して、異なるタスクを実行することができます。中断されたタスクが結合されている

場合は、同じスレッドにより中断されたタスクの実行が後に再開されます。中断されたタスクが結合されていない場合は、現在のチームに属するスレッドならどれでもタスクの実行を再開できます。

OpenMP 仕様には、結合されたタスクに対して次のタスクスケジューリングポイントが定義されています。

- タスクコンストラクトを検出したポイント
- タスク待ちコンストラクトを検出したポイント
- 暗黙的または明示的なバリアーを検出したポイント
- タスクの終了ポイント

Solaris Studio コンパイラによる実装に従い、上記のスケジューリングポイントは結合解除されたタスクのタスクスケジューリングポイントでもあります。

タスク指令を使って指定された明示的タスクに加え、OpenMP 仕様のバージョン 3.0 では暗黙的タスクの概念も取り入れられています。暗黙的タスクは、暗黙的な並列領域により生成されるタスク、または実行中に並列構文を検出したときに生成されるタスクです。それぞれの暗黙的タスクのコードは、**parallel** コンストラクトの内部コードです。暗黙的タスクはそれぞれチーム内の異なるスレッドに割り当てられ、結合されます。すなわち、暗黙的タスクは最初から最後まで常に最初に割り当てられたスレッドにより実行されます。

parallel コンストラクトが検出されたときに生成されたすべての暗黙的タスクは、マスタースレッドが並列領域の最後で暗黙的バリアーを終了するときに完了することが保証されます。一方、並列領域内に生成されるすべての明示的タスクは、並列領域内の次の暗黙的または明示的バリアーの終了時に完了することが保証されます。

if 節が **task** コンストラクトにあり、スカラー式の評価値が **false** の場合には、タスクを検出したスレッドはただちにそのタスクを実行する必要があります。**if** 節は、細かく組まれた多数のタスクを生成し、それらを概念上のプールに配置するというオーバーヘッドを避けるために使用することができます。

5.2 データ環境

task 指令は、タスクのデータ環境を定義する次のデータ属性節を取ります。

- **default** (**private** | **firstprivate** | **shared** | **none**)
- **private** (*list*)
- **firstprivate** (*list*)
- **shared** (*list*)

shared 節にリスト指定された変数へのタスク中のすべての参照は、 **task** 指令の直前に存在する同じ名前の変数を参照します。

private および **firstprivate** 変数のそれぞれに対し、新しいストレージが作成され、**task** コンストラクトの字句エクステントにある元の変数へのすべての参照は、新しいストレージへの参照に置き換えられます。**firstprivate** 変数は、タスクが検出された時点の元の変数値で初期化されます。

OpenMP 3.0 仕様のバージョン 3.0 (節 2.9.1) には、parallel、task、および worksharing 領域で参照される変数のデータ共有属性の決定方法が説明されています。

コンストラクト中で参照される変数のデータ共有属性は、事前定義、明示的定義、または暗黙的定義のいずれかです。明示的に指定されたデータ共有属性を持つ変数は、指定されたコンストラクトの中で参照される変数で、コンストラクト上のデータ共有属性節にリストされています。暗黙的に指定されたデータ共有属性を持つ変数は、指定されたコンストラクトの中で参照される変数で、事前に決められたデータ共有属性を持たず、コンストラクト上のデータ共有属性節にリストがありません。

変数のデータ共有属性を暗黙的に決めるための規則は、必ずしも明白ではない場合があります。予期せぬ事態を避けるため、プログラミングの際には、OpenMP の暗黙的なスコープ宣言規則に依存せずに、データ共有属性節を使用してタスク構文で参照されるすべての変数を明示的にスコープ宣言することをお勧めします。

5.3 TASKWAIT 指令

指定された並列領域に結合されたすべての明示的タスクのサブセットの完了は、**taskwait** 指令の使用により指定される場合があります。**taskwait** 指令は、現在の (暗黙的または明示的) タスクの開始以降に生成された子タスクの完了を待つことを指定します。**taskwait** 指令は、直接の子タスクの完了を待つことを指定するもので、より下位のタスクすべてに対するものではない点に留意してください。

5.4 タスク化の例

次の C/C++ プログラムは、OpenMP タスクと **taskwait** 指令をどのように使うとフィボナッチ数列を再帰的に計算できるかを示したものです。

この例では、**parallel** 指令は 4 つのスレッドにより実行される並列領域を表しています。並列構文中で **single** 指令が使用され、1 つのスレッドだけが `fib(n)` を呼び出す **print** 文を実行することが示されています。

`fib(n)` を呼び出すと、**task** 指令に指定された 2 つのタスクが生成されます。一方のタスクは `fib(n-1)` を計算し、他方のタスクは `fib(n-2)` を計算します。2 つの返り値が加算され、`fib(n)` の返り値が求められます。`fib(n-1)` および `fib(n-2)` を呼び出すと、それぞれが 2 つのタスクを生成します。タスクは、`fib()` に渡された引数が 2 より小さくなるまで、再帰的に生成されます。

taskwait 指令は、fib() の呼び出しにより生成された 2 つのタスクの完了 (すなわち、タスクが i と j を計算) が、fib() の呼び出しが戻る前に行われるようにします。

single 指令と fib(n) の呼び出しを行うスレッドが 1 つだけだったとしても、4 つのすべてのスレッドが生成されるタスクの実行に係わっている点に留意してください。

この例は、Solaris Studio 12.2 の C++ コンパイラでコンパイルしました。

例 5–1 タスク化の例: フィボナッチ数列の計算

```
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
       #pragma omp task shared(i) firstprivate(n)
       i=fib(n-1);

       #pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
    }
}

int main()
{
  int n = 10;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}


% CC -xopenmp -xO3 task_example.cc
% a.out
fib(10) = 55
```

5.5 プログラミング上の留意点

タスク化により、OpenMP プログラムを複雑にする要素が加わります。プログラマは、タスクを使用するプログラムがどのように動作するかについて特別な注意を払う必要があります。プログラミング上、留意する必要のある項目について以下に説明します。

5.5.1 THREADPRIVATE およびスレッド特有の情報

スレッドがタスクスケジューリングポイントを検出したときは、実装時に現在のタスクが中断され、そのスレッドが他のタスクを処理するようスケジュールされる設定となる場合があります。これは、**threadprivate** 変数の値、またはスレッド番号など他のスレッド固有の情報がタスクスケジューリングポイントの前後で変更されたことを暗黙的に示しています。

中断されているタスクが結合されている場合は、タスクの実行を再開するスレッドは、中断したときのスレッドと同じになります。このため、スレッド番号はタスクの再開後も変更されません。ただし、**threadprivate** 変数の値は変更されることがあります。それは、スレッドは他のタスクの処理を行うようスケジューリングされることがあり、中断されたタスクを再開する前に **threadprivate** 変数が変更される場合があるからです。

中断されているタスクが結合解除場合は、タスクの実行を再開するスレッドは、中断したときのスレッドと異なる場合があります。このため、スレッド番号と **threadprivate** 変数の値の両方とも、タスクスケジューリングポイントの前後で異なることがあります。

5.5.2 ロック

OpenMP 3.0 では、ロックはスレッドではなく、タスクに所有されていると規定されています。ロックが取得されると、現在のタスクがそれを所有します。タスクの終了時には、同じタスクがそのロックを解放する必要があります。

一方、**critical** コンストラクトは、スレッドベースの相互排他機構として残されています。

ロックの所有者の変更により、ロックを使用する際にはより慎重な処理が必要となります。次のプログラム (OpenMP 仕様のバージョン 3.0 の例 A.43.1c として紹介されています) は OpenMP 2.5 に適合しています。これは、並列領域にあるロック `lck` を解放するスレッドは、プログラムの順次処理部分で使用されるそのロックを取得したのと同じスレッドであるためです (並列領域のマスタースレッドと初期のスレッドが同一)。ところが、このプログラムは OpenMP 3.0 では適合しません。これは、ロック `lck` を解放するタスク領域が、ロックを取得したタスク領域と異なるためです。

例 5–2　ロックの使用例: OpenMP 3.0 での不適合

```
#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);
    }
  }
  omp_destroy_lock (&lck);
}
```

5.5.3　スタックデータへの参照

タスクには、タスクコンストラクトが現れるルーチンのスタック上のデータへの参照が付加されていることがよくあります。タスクの実行は次の暗黙的または明示的バリアーまで保留されることがあるため、指定されたタスクが現れるルーチンのスタックがポップされ、スタックデータが上書きされた後に、そのタスクが実行されることがあります。そのため、タスクに共有されたということでリストアップされたスタックデータが破棄されます。

必要な同期処理を挿入し、タスクが変数を参照したときに、変数が確実にスタック上にあるようにしておくことは、プログラマの責任です。2 つの例を次に示します。

最初の例では、`i` は **`task`** コンストラクトの中で **`shared`** に指定されています。タスクは、`work()` のスタック上に割り当てられている `i` のコピーにアクセスします。

タスクの実行は保留されることがありますので、タスクは `main()` の並列領域の最後の暗黙的バリアーで、`work()` ルーチンの処理の終了後に実行されます。そのため、タスクが `i` を参照すると、その時にたまたまスタック上にあった値にアクセスしてしまうことになります。

正しい結果を得るためには、プログラマはタスクが完了する前に `work()` を終了しないようにしておく必要があります。**`taskwait`** 指令を **`task`** コンストラクトの後に挿入することにより、この処理を追加することができます。あるいは、**`task`** コンストラクトで、`i` に対して **`shared`** ではなく、**`firstprivate`** を指定することもできます。

例 5–3 スタックデータ: 例 1 - 正しくないバージョン

```
#include <stdio.h>
#include <omp.h>
void work()
 {
   int i;

   i = 10;
   #pragma omp task shared(i)
   {
     #pragma omp critical
     printf("In Task, i = %d\n",i);
   }
 }

int main(int argc, char** argv)
 {
    omp_set_num_threads(8);
    omp_set_dynamic(0);

    #pragma omp parallel
    {
      work();
    }
 }
```

例 5–4 スタックデータ: 例 1 - 修正されたバージョン

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

void work()
 {
   int i;

   i = 10;
   #pragma omp task shared(i)
   {
     #pragma omp critical
     printf("In Task, i = %d\n",i);
   }

   /* Use TASKWAIT for synchronization. */
   #pragma omp taskwait
 }

int main(int argc, char** argv)
 {
    omp_set_num_threads(8);
    omp_set_dynamic(0);

    #pragma omp parallel
    {
      work();
    }
 }
```

この次に示す例では、task 構文中の **j** が sections 構文中の **j** を参照しています。このため、タスクは **firstprivate** 型のコピーである sections 構文中の **j** にアクセスします。これは、**sections** 構文のアウトラインルーチンのスタック上のローカル変数です (Solaris Studio コンパイラを含む一部の実装の場合)。

タスクを **sections** 領域の最後の暗黙的バリアーで **sections** コンストラクトのアウトラインルーチン終了後に実行するため、タスクの実行は保留されることがあります。このため、タスクから j が参照されるときには、スタック上の不確定の値がアクセスされてしまいます。

正しい結果を得るためには、プログラマは **sections** 領域が暗黙的バリアーに達する前にタスクが実行されるようにしておく必要があります。**taskwait** 指令を **task** コンストラクトの後に挿入することにより、この処理を追加することができます。あるいは、**task** コンストラクトで、j に対して **shared** ではなく **firstprivate** を指定することもできます。

例 **5–5** 例 2 - 正しくないバージョン

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

int main(int argc, char** argv)
 {
    omp_set_num_threads(2);
    omp_set_dynamic(0);
    int j=100;

    #pragma omp parallel shared(j)
    {
       #pragma omp sections firstprivate(j)
       {
          #pragma omp section
          {
             #pragma omp task shared(j)
             {
               #pragma omp critical
               printf("In Task, j = %d\n",j);
             }
          }
       }
    }

    printf("After parallel, j = %d\n",j);
 }
```

例 **5–6** 例 2 - 修正されたバージョン

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

int main(int argc, char** argv)
 {
    omp_set_num_threads(2);
    omp_set_dynamic(0);
```

例 5–6 例 2 - 修正されたバージョン (続き)

```
    int j=100;

    #pragma omp parallel shared(j)
    {
       #pragma omp sections firstprivate(j)
       {
          #pragma omp section
          {
             #pragma omp task shared(j)
             {
               #pragma omp critical
               printf("In Task, j = %d\n",j);
             }

             /* Use TASKWAIT for synchronization. */
             #pragma omp taskwait
          }
       }
    }

    printf("After parallel, j = %d\n",j);
 }
```

◆ ◆ ◆ 第 6 章

6 変数の自動スコープ宣言

OpenMP 構文で参照される変数のデータ共有属性の宣言は、スコープ宣言と呼ばれます。各データ共有属性の説明は、OpenMP 3.0 仕様の 2.9.3 節にあります。

OpenMP プログラムでは、OpenMP 構文で参照されるすべての変数は、スコープ宣言されます。一般的に、構文で参照される変数は、2 つの方法のうちのどちらかでスコープ宣言されます。プログラマがデータ共有属性節で変数のスコープを明示的に宣言するか、あるいは OpenMP API がコンパイラに実装されていることで、暗黙に決まるか事前定義されたスコープに対して規則が自動的に適用されます。これは OpenMP 3.0 仕様の 2.9.1 節に基づいています。

ほとんどのユーザーにとって、OpenMP パラダイムの中でもっとも困難なのは、スコープ宣言です。変数を明示的にスコープ宣言することは、特に大規模で複雑なプログラムの場合、手間がかかりミスもしやすくなります。さらに、OpenMP 3.0 の仕様では、暗黙に決まるか事前定義された変数スコープに対する規則が指定されているため、これにより予期せぬ結果を招くことがあります。OpenMP 仕様 3.0 で **task** 指令が追加されたことにより、スコープ宣言がさらに複雑で難しくなりました。

自動的にスコープ宣言を行う、自動スコープ宣言と呼ばれる機能が Solaris Studio コンパイラでサポートされています。これは非常に便利なツールで、プログラマは変数のスコープを明示的に定義しなくても済みます。自動スコープ宣言では、コンパイラは非常にシンプルなユーザーモデルで、スマートな規則に基づいて変数のスコープを決定します。

コンパイラの過去のリリースでは、変数の自動スコープ宣言は **parallel** 構文でしか行えませんでした。最新の Solaris Studio コンパイラでは、自動スコープ宣言機能が **task** 構文で参照される変数にも拡張されました。

6.1 自動スコープ宣言用データスコープ節

自動スコープ宣言は、自動的にスコープ宣言される変数を **__auto** データスコープで指定するか、**default(__auto)** 節を使用することで呼び出されます。どちらも、Solaris Studio コンパイラで提供される OpenMP で拡張された仕様です。

6.1.1 __auto 節

構文: **__auto(***list-of-variables***)**

Fortran の場合、**__AUTO(***list-of-variables***)** も使用できます。

並列構文またはタスク構文上の **__auto** 節は、コンパイラが構文中で指定された変数のスコープを自動的に特定するように指示します。**auto** の前の下線は 2 つであることに注意してください。

__auto 節は、**PARALLEL**、**PARALLEL DO/for**、**PARALLEL SECTIONS**、Fortran 95 **PARALLEL WORKSHARE**、または **TASK** 指令で使用できます。

__auto 節で変数を指定した場合、ほかのデータ共有属性節でその変数を指定できません。

6.1.2 default(__auto) 節

構文: **default(__auto)**

Fortran の場合、**DEFAULT(__AUTO)** も使用できます。

並列構文またはタスク構文上の **default(__auto)** 節は、どのデータスコープ節でも明示的にスコープ宣言されていない、構文内で参照される変数すべてのスコープを、コンパイラが自動的に決定するように指示します。

default(__auto) 節は、**PARALLEL**、**PARALLEL DO/for**、**PARALLEL SECTIONS**、Fortran 95 **PARALLEL WORKSHARE**、または **TASK** 指令で使用できます。

6.2 並列構文のスコープ宣言の規則

自動スコープ宣言では、コンパイラは、並列構文内の変数のスコープを決定する際に次の規則を適用します。

これらの規則は、OpenMP 仕様で暗黙にスコープ宣言される、ワークシェアリング **DO** または **FOR** ループのループインデックス変数などの変数には適用されません。

6.2.1 スカラー変数に関するスコープ宣言規則

並列構文内で参照され、暗黙に決まるか事前定義されたスコープを持たないスカラー変数を自動宣言する場合、コンパイラは、変数の使用を次の規則 **PS1** - **PS3** に対して順番に確認します。

- **PS1**: 並列領域内で変数を使用しても、その領域を実行するチーム内でスレッドに関するデータ競合状態が発生しない場合、その変数のスコープは **SHARED** と宣言されます。
- **PS2**: 並列領域を実行するすべてのスレッドで、変数が同じスレッドによる読み取りの前に常に書き込まれる場合、その変数のスコープは **PRIVATE** と宣言されます。変数が **PRIVATE** とスコープ宣言することが可能で、並列領域のあと、書き込みの前に読み取られ、構文が **PARALLEL DO** か **PARALLEL SECTIONS** のいずれかである場合、その変数のスコープは、LASTPRIVATE と宣言されます。
- **PS3**: 変数がコンパイラの認識可能な縮約処理で使用されている場合、その変数のスコープは、その特定の型を持つ **REDUCTION** と宣言されます。

6.2.2 配列に関するスコープ宣言規則

- **PA1**: 並列領域内で変数を使用しても、その領域を実行するチーム内でスレッドに関するデータ競合状態が発生しない場合、その配列のスコープは **SHARED** と宣言されます。

6.3 task 構文のスコープ宣言規則

自動スコープ宣言では、コンパイラは、**task** 構文内の変数のスコープを決定する際に次の規則を適用します。

これらの規則は、OpenMP 仕様で暗黙にスコープ宣言される、**PARALLEL DO** や **FOR** ループのループインデックス変数などの変数には適用されません。

6.3.1 スカラー変数に関するスコープ宣言規則

タスク構文で参照され、暗黙に決まるか事前定義されたスコープを持たないスカラー変数を自動スコープ宣言する場合、コンパイラは、変数の使用を次の規則 **TS1** - **TS5** に対して順番に確認します。

- **TS1: task** 構文内で変数が読み取り専用として使用され、その task 構文を包含する並列構文内でも読み取り専用である場合、その変数は **FIRSTPRIVATE** として自動スコープ宣言されます。
- **TS2:** 変数を使用してもデータ競合が発生せず、タスクの実行中にその変数にアクセスできる場合、その変数は **SHARED** と自動スコープ宣言されます。

- **TS3:** 変数を使用してもデータ競合が発生せず、タスク構文で読み取り専用であり、かつ、タスクの実行中には変数にアクセスできない場合は、その変数は **FIRSTPRIVATE** と自動スコープ宣言されます。
- **TS4:** 変数を使用するとデータ競合が発生し、タスク領域を実行する各スレッドで、その変数が同じスレッドによる読み取りの前に常に書き込まれる場合、その変数は **PRIVATE** と自動スコープ宣言されます。
- **TS5**: 変数を使用するとデータ競合が発生し、タスク領域内で読み取り専用ではなく、かつ、タスク領域で行われる読み取りの中で、タスク外で定義された値が取得されることがある場合は、その変数は **FIRSTPRIVATE** と自動スコープ宣言されます。

6.3.2 配列に関するスコープ宣言規則

タスクの自動スコープ宣言では、配列は処理しません。

6.4 自動スコープ宣言に関する一般的な注意事項

タスクの自動スコープ宣言規則と、自動スコープ宣言の結果は、今後のリリースで変更される可能性があります。また、暗黙的に決定されたスコープ宣言規則と自動スコープ宣言規則が適用される順序も、今後のリリースで変更される可能性があります。

プログラマは **_auto(***list-of-variables***)** 節または **default(_auto)** 節を使用し、明示的に自動スコープ宣言を要求します。**default(_auto)** または **_auto(***list-of-variables***)** 節を **parallel** 構文に対して指定しても、**parallel** 構文に字句的または動的に包含される **task** 構文にも同じ節が適用されることを意味するわけではありません。

事前定義された暗黙的スコープを持たない変数を自動スコープ宣言する場合、コンパイラは、変数の使用について、前述の規則に対して順番に確認します。規則に一致する場合、コンパイラはその規則に従って変数のスコープを決定します。一致する規則がない場合、または自動スコープ宣言が変数を処理できない場合 (ただし、後述のように、いくつかの制限事項はあります)、コンパイラは変数を **SHARED** とスコープ宣言し、**parallel** または **task** 構文を **IF (.FALSE.)** または **if(0)** 節が指定されているかのように結合並列領域が直列化されます。

自動スコープ宣言が失敗する理由は、一般的に 2 つあります。1 つは、変数の使われ方が前述のどれにも一致しないため、もう 1 つは、ソースコードが複雑すぎて、コンパイラが十分な解析を行えないためです。こうした原因としてよくあるのは、たとえば、関数呼び出しや複雑な配列添え字、メモリー別名、ユーザー実装の同期などです。

6.5 制限事項

- 自動スコープ宣言を有効にするには、最適化レベルを **-xO3** かそれ以上に設定してから **-xopenmp** でプログラムをコンパイルする必要があります。自動スコープ宣言は、プログラムが **-xopenmp=noopt** だけでコンパイルされている場合は有効になりません。
- C および C++ の並列およびタスク構文の自動スコープ宣言では、基本的なデータ型、つまり整数型、浮動小数点型、およびポインタ型しか処理できません。
- タスクの自動スコープ宣言では、配列は処理できません。
- C および C++ でのタスクの自動スコープ宣言では、グローバル変数は処理できません。
- タスクの自動スコープ宣言では、結合解除されたタスクは処理できません。
- タスクの自動スコープ宣言では、ほかのタスクに字句的に包含されているタスクは処理できません。次に例を示します。

```
#pragma omp task /* task1 */
{
  ...
  #pragma omp task /* task 2 */
  {
    ...
  }
 ...
}
```

  前述の例では、コンパイラは、`task1` に字句的に入れ子になった `task2` の自動スコープは試行しません。コンパイラは `task2` で参照されているすべての変数を **SHARED** とスコープ宣言し、`task2` を **IF(.FALSE.)** または **if(0)** 節がタスクで指定されているかのように処理します。

- 解析では、OpenMP 指令のみ認識、使用されます。OpenMP 実行時ルーチンの呼び出しは認識されません。たとえばプログラムが **omp_set_lock()** および **omp_unset_lock()** を使用してクリティカル領域を実装している場合、コンパイラはそのクリティカル領域の存在を検出できません。可能な場合は、**CRITICAL** および **END CRITICAL** 指令を使用してください。
- データ競合解析では、**BARRIER** や **MASTER** などの OpenMP 同期指令を使用して指定された同期のみ認識、使用されます。ビジー待ちなどのユーザー実装の同期は認識されません。

6.6 自動スコープ宣言結果の確認

自動スコープ宣言の結果を確認したり、自動スコープ宣言が失敗したために直列化した並列領域の有無を確認したりするには、コンパイラのコメントを使用します。

コンパイルで **-g** が付けられていると、コンパイラはインラインコメントを生成します。このコメントは、次に示すように **er_src** を使って表示できます(**er_src** コマンドは、Solaris Studio ソフトウェアの一部として提供されています。詳細は、**er_src**(1) のマニュアルページまたは『Solaris Studio パフォーマンスアナライザ』を参照してください)。

-xvpara コンパイルオプションを使用することからスタートすることを推奨します。—xvpara を使用してコンパイルすると、特定の構文の自動スコープ宣言が成功したかどうかを把握することができます。次に例を示します。

例 6–1 **-vpara** による自動スコープ宣言

```
%cat source1.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END
%f95 -xopenmp -xO3 -vpara -c -g source1.f
"source1.f", line 2: Autoscoping for OpenMP construct succeeded.
Check er_src for details
```

特定の構文の自動スコープ宣言が失敗すると、この例に示すような警告メッセージが (**-xvpara** によって) 発行されます。

例 6–2 **-vpara** による自動スコープ宣言の失敗

```
%cat source2.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         CALL FOO(X)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END
%f95 -xopenmp -xO3 -vpara -c -g source2.f
"source2.f", line 2: Warning: Autoscoping for OpenMP construct failed.
 Check er-src for details. Parallel region will be executed by
 a single thread.
```

詳細は、**er_src** で表示される、コンパイラコメントに示されます。

```
% er_src source2.o
Source file: source2.f
Object file: source2.o
Load Object: source2.o


     1.         INTEGER X(100), Y(100), I, T

   Source OpenMP region below has tag R1
   Variables autoscoped as SHARED in R1: y
   Variables autoscoped as PRIVATE in R1: t, i
   Variables treated as shared because they cannot be autoscoped in R1: x
   R1 will be executed by a single thread because autoscoping for some variable s was not successful
   Private variables in R1: i, t
   Shared variables in R1: y, x
     2. C$OMP PARALLEL DO DEFAULT(__AUTO)

   Source loop below has tag L1
   L1 parallelized by explicit user directive
   L1 autoparallelized
   L1 parallel loop-body code placed in function _$d1A2.MAIN_ along with 0 inne r loops
   L1 could not be pipelined because it contains calls
     3.         DO I=1, 100
     4.                 T = Y(I)
     5.                 CALL FOO(X)
     6.                 X(I) = T*T
     7.         END DO
     8. C$OMP END PARALLEL DO
     9.         END
    10.
```

6.7 自動スコープ宣言の例

自動スコープ宣言規則がどのように使用されるかを、次の例に示します。

例 6–3 より複雑な例

```
 1.      REAL FUNCTION FOO (N, X, Y)
 2.      INTEGER       N, I
 3.      REAL          X(*), Y(*)
 4.      REAL          W, MM, M
 5.
 6.      W = 0.0
 7.
 8. C$OMP PARALLEL DEFAULT(__AUTO)
 9.
10. C$OMP SINGLE
11.       M = 0.0
12. C$OMP END SINGLE
13.
14.       MM = 0.0
15.
16. C$OMP DO
17.       DO I = 1, N
```

例 6–3 より複雑な例 *(続き)*

```
18.          T = X(I)
19.          Y(I) = T
20.          IF (MM .GT. T) THEN
21.             W = W + T
22.             MM = T
23.          END IF
24.       END DO
25. C$OMP END DO
26.
27. C$OMP CRITICAL
28.       IF ( MM .GT. M ) THEN
29.          M = MM
30.       END IF
31. C$OMP END CRITICAL
32.
33. C$OMP END PARALLEL
34.
35.      FOO = W - M
36.
37.      RETURN
38.      END
```

関数 **FOO()** には並列領域が 1 つあり、この並列領域には、**SINGLE** 構文とワークシェアリングの **DO** 構文、**CRITICAL** 構文がそれぞれ 1 つあります。こうした OpenMP 並列構文をすべて無視した場合、並列領域内のコードが行うのは、次のことです。

1. 配列 **X** 内の値を配列 **Y** にコピーします。
2. **X** 内の正の最大値を検出し、その値を **M** に格納します。
3. **X** の一部要素の値を変数 **W** に蓄積します。

コンパイラが前述の規則に従って、この並列領域内の変数に適切なスコープを発見する仕組みをみてみましょう。

前述の並列領域では、**I**、**N**、**MM**、**T**、**W**、**M**、**X**、および **Y** という変数が使用されています。コンパイラは次のことを決定します。

- スカラー **I** は、ワークシェアリング **DO** ループのループインデックスです。OpenMP 仕様では、**I** のスコープは **PRIVATE** 宣言することが必須です。
- スカラー **N** は並列領域内で読み取られるだけで、データ競合を起こしません。このため、規則 **S1** に従って、この変数のスコープは **SHARED** と宣言されます。
- 並列領域を実行するスレッドはすべて、スカラー **MM** の値を 0.0 に設定する 14 行目を実行します。この書き込みはデータ競合の原因になるため、規則 **S1** は適用されません。この書き込みは、同じスレッド内のあらゆる **MM** の読み取りの前に起きるため、規則 **S2** に従って、**MM** のスコープは **PRIVATE** と宣言されます。
- 同様に、**T** も **PRIVATE** とスコープ宣言されます。
- スカラー **W** は 21 行目でいったん読み取られたあとに書き込まれます。このため、**S1** および S2 は適用されません。加算は連想および伝達の両方の要素が含まれるため、規則 **S3** に従って **W** のスコープは **REDUCTION(+)** と宣言されます。

- スカラー **M** は、**SINGLE** 構文にある文 11 で書き込まれます。この **SINGLE** 構文の末尾のバリアは、文 11 の書き込みが文 28 の読み取りや文 29 の書き込みと同時に発生しないようにするためのものです。また、文 28 と 29 はどちらも **CRITICAL** 構文内にあるため、同時に発生しないようになっています。2 つのスレッドが同時に **M** にアクセスすることはできません。このため、並列領域内での **M** 読み取りと書き込みがデータ競合を起こすことはなく、規則 **S1** に従って、**M** のスコープは **SHARED** と宣言されます。
- 配列 **X** は領域内では読み取りだけで、書き込みは行われません。このため、この配列のスコープは、規則 **A1** に従って **SHARED** と宣言されます。
- 配列 **Y** への書き込みはスレッド間で分散され、2 つのスレッドが **Y** の同じ要素に書き込むことはありません。データの競合がないため、**Y** のスコープは、規則 **A1** に従って **SHARED** と宣言されます。

例 **6–4** QuickSort の例

```
static void par_quick_sort (int p, int r, float *data)
{
   if (p < r)
   {
      int q = partition (p, r, data);

      #pragma omp task default(__auto) if ((r-p)>=low_limit)
      par_quick_sort (p, q-1, data);

      #pragma omp task default(__auto) if ((r-p)>=low_limit)
      par_quick_sort (q+1, r, data);
   }
}

int main ()
{
  ...
  #pragma omp parallel
  {
     #pragma omp single nowait
     par_quick_sort (0, N-1, &Data[0]);
  }
  ...
}

er_src result:

      Source OpenMP region below has tag R1
      Variables autoscoped as FIRSTPRIVATE in R1: p, q, data
      Firstprivate variables in R1: data, p, q
        47. #pragma omp task default(__auto) if ((r-p)>=low_limit)
        48. par_quick_sort (p, q-1, data);

      Source OpenMP region below has tag R2
      Variables autoscoped as FIRSTPRIVATE in R2: q, r, data
      Firstprivate variables in R2: data, q, r
        49. #pragma omp task default(__auto) if ((r-p)>=low_limit)
        50. par_quick_sort (q+1, r, data);
```

例 6–4 QuickSort の例 *(続き)*

スカラー変数 p および q、およびポインタ変数データは、タスク構文でも並列領域でも読み取り専用です。 そのため、これらは **TS1** に従って **FIRSTPRIVATE** と自動スコープ宣言されます。

例 6–5 もう 1 つの例

```
int fib (int n)
{
   int x, y;
   if (n < 2) return n;

   #pragma omp task default(__auto)
   x = fib(n - 1);

   #pragma omp task default(__auto)
   y = fib(n - 2);

   #pragma omp taskwait
   return x + y;
}

er_src result:

   Source OpenMP region below has tag R1
   Variables autoscoped as SHARED in R1: x
   Variables autoscoped as FIRSTPRIVATE in R1: n
   Shared variables in R1: x
   Firstprivate variables in R1: n
    24.         #pragma omp task default(__auto) /* shared(x) firstprivate(n) */
    25.         x = fib(n - 1);

   Source OpenMP region below has tag R2
   Variables autoscoped as SHARED in R2: y
   Variables autoscoped as FIRSTPRIVATE in R2: n
   Shared variables in R2: y
   Firstprivate variables in R2: n
    26.         #pragma omp task default(__auto) /* shared(y) firstprivate(n) */
    27.         y = fib(n - 2);
    28.
    29.         #pragma omp taskwait
    30.         return x + y;
    31. }
```

スカラー n はタスク構文でも並列構文でも読み取り専用です。そのため、n は TS1 に従って **FIRSTPRIVATE** と自動スコープ宣言されます。

スカラー変数 x および y は、ローカル関数 `fib()` のローカル変数です。両方のタスクが x と y にアクセスしても、データ競合は起こりません。**taskwait** があるため、2 つのタスクがまず実行を完了してから、タスクを検出後 `fib()` を実行していたスレッドが `fib()` を終了します。つまり、x と y は、2 つのタスクが実行している間でも利用可能なのです。そのため、x と y は、TS2 に従い、**SHARED** と自動スコープ宣言されます。

例 6–6 もう 1 つの例

```
int main(void)
{
  int yy = 0;

  #pragma omp parallel default(__auto) shared(yy)
  {
    int xx = 0;

    #pragma omp single
    {
       #pragma omp task default(__auto) // task1
       {
          xx = 20;
       }
    }

    #pragma omp task default(__auto) // task2
    {
       yy = xx;
    }
  }

  return 0;
}

er_src result:

   Source OpenMP region below has tag R1
   Variables autoscoped as PRIVATE in R1: xx
   Private variables in R1: xx
   Shared variables in R1: yy
     7.   #pragma omp parallel default(__auto) shared(yy)
     8.   {
     9.     int xx = 0;
    10.

   Source OpenMP region below has tag R2
    11.     #pragma omp single
    12.     {

   Source OpenMP region below has tag R3
   Variables autoscoped as SHARED in R3: xx
   Shared variables in R3: xx
    13.        #pragma omp task default(__auto) // task1
    14.        {
    15.           xx = 20;
    16.        }
    17.     }
    18.

   Source OpenMP region below has tag R4
   Variables autoscoped as PRIVATE in R4: yy
   Variables autoscoped as FIRSTPRIVATE in R4: xx
   Private variables in R4: yy
   Firstprivate variables in R4: xx
    19.     #pragma omp task default(__auto) // task2
    20.     {
```

例 6–6 もう 1 つの例 *(続き)*

```
    21.        yy = xx;
    22.     }
    23.   }
```

この例では、xx は並列領域の private 変数です。チームのスレッドの 1 つが xx の初期値を変更します (task1 を実行します)。その後、すべてのスレッドが task2 を検出し、xx が同じ計算を行います。

task1 では、xx を使用しても、データ競合は発生しません。1 つの構文の終わりに暗黙的なバリアーがあり、このバリアーを出る前に task1 を完了する必要があるので、xx は task1 が実行している間でも利用可能なのです。したがって、TS2 により、xx は task1 で **SHARED** と自動スコープ宣言されます。

task2 では、xx は読み取り専用として使用されます。ただし、xx の使用は、包含する並列構文では読み取り専用ではありません。xx は、並列構文に対しては **PRIVATE** と事前定義されているので、task2 が実行している間でも xx が利用可能かどうかはわかりません。そのため、TS3 に従い、task2 では **xx** は FIRSTPRIVATE と自動スコープ宣言されます。

task2 では、yy を使用することでデータ競合が発生し、task2 を実行する各スレッドでは、変数 yy は同じスレッドによる読み取りの前に常に書き込まれます。そのため、TS4 に従い、yy は task2 では **PRIVATE** と自動スコープ宣言されます。

例 6–7 もう 1 つの例

```
int foo(void)
{
  int xx = 1, yy = 0;

  #pragma omp parallel shared(xx,yy)
  {
    #pragma omp task default(__auto)
    {
       xx += 1;

       #pragma omp atomic
       yy += xx;
    }

    #pragma omp taskwait
  }
  return 0;
}

er_src result:

   Source OpenMP region below has tag R1
   Shared variables in R1: yy, xx
     5.   #pragma omp parallel shared(xx,yy)
     6.   {
```

例 6–7 もう 1 つの例 *(続き)*

```
Source OpenMP region below has tag R2
Variables autoscoped as SHARED in R2: yy
Variables autoscoped as FIRSTPRIVATE in R2: xx
Shared variables in R2: yy
Firstprivate variables in R2: xx
  7.     #pragma omp task default(__auto)
  8.     {
  9.        xx += 1;
 10.
 11.        #pragma omp atomic
 12.        yy += xx;
 13.     }
 14.
 15.     #pragma omp taskwait
 16.   }
```

task 構文では xx は読み取り専用として使用されないので、データ競合が発生します。しかし、タスク領域で x を読み取ると、タスク外で x の値が定義されます(この例では、xx は並列領域に対して **SHARED** なので、x の定義は並列領域外で行われます)。そのため、TS5 に従い、xx は **FIRSTPRIVATE** と自動スコープ宣言されます。

task 構文での yy の使用は読み取り専用ではありませんが、データ競合は発生しません。**taskwait** が発生しているので、yy はタスクの実行中でもアクセスできます。そのため、TS2 に従い、yy は **SHARED** と自動スコープ宣言されます。

◆ ◆ ◆ 第 7 章

7 スコープチェック

自動スコープ宣言を使用すると、変数をどのようにスコープ宣言するかをプログラマが決定できます。ただし、複雑なプログラムの場合、自動スコープ宣言が実行されなかったり、自動スコープ宣言の結果が予期しないものになったりすることがあります。不正なスコープ宣言により、目立たないが深刻な問題が多数発生することがあります。たとえば、一部の変数を **SHARED** として不正にスコープ宣言するとデータ競合が起きるという可能性や、変数のスレッド固有化を正しく行わないと構文外でその変数が未定義の値になるという可能性があります。

Solaris Studio C、C++、および Fortran のコンパイラは、コンパイル時のスコープチェック機能を備えており、OpenMP プログラムの変数が正しくスコープ宣言されたかどうかがコンパイラによって確認されます。

スコープチェックを行えば、コンパイラの機能に応じて、データ競合、不適切な変数のスレッド固有化や縮約、およびその他のスコープ宣言上の不具合など、潜在的な問題を検出することができます。スコープチェック時には、プログラマによって指定されたデータ共有属性、コンパイラによって決定された暗黙的なデータ共有属性、および自動スコープ宣言の結果が、コンパイラによって確認されます。

7.1 スコープチェック機能の使用

スコープチェックを有効にするには、**-xvpara** および **-xopenmp** オプションを使用するとともに最適化レベルを **-x03** かそれ以上に設定してから OpenMP プログラムをコンパイルする必要があります。スコープチェックは、プログラムが **-xopenmp=noopt** でコンパイルされただけでは動作しません。最適化レベルが **-x03** 未満の場合、コンパイラは警告メッセージを発行し、スコープチェックを行いません。

スコープチェック時には、コンパイラはすべての OpenMP 構文を確認します。いくつかの変数のスコープ宣言に問題がある場合は、コンパイラは警告メッセージを発行します。場合によっては、正しいデータ共有属性節が提案されます。

次に例を示します。

例 7–1 スコープチェック

```
% cat t.c

#include <omp.h>
#include <string.h>

int main()
{
  int g[100], b, i;

  memset(g, 0, sizeof(int)*100);

  #pragma omp parallel for shared(b)
  for (i = 0; i < 100; i++)
  {
    b += g[i];
  }

  return 0;
}

%  cc -xopenmp -xO3 -xvpara source1.c
"source1.c", line 10: Warning: inappropriate scoping
         variable 'b' may be scoped inappropriately as 'shared'
         . write at line 13 and write at line 13 may cause data race

"source1.c", line 10: Warning: inappropriate scoping
         variable 'b' may be scoped inappropriately as 'shared'
         . write at line 13 and read at line 13 may cause data race
```

コンパイラは、最適化レベルが **-xO3** 未満の場合はスコープチェックを行いません。

```
%  cc -xopenmp=noopt  -xvpara source1.c
 "source1.c", line 10: Warning: Scope checking under vpara compiler
option is supported with optimization level -xO3 or higher.
 Compile with a higher optimization level to enable this feature
```

より複雑な例:

例 7–2 source2

```
% cat source2.c

#include <omp.h>

int main()
{
  int g[100];
  int r=0, a=1, b, i;

  #pragma omp parallel for private(a) lastprivate(i) reduction(+:r)
  for (i = 0; i < 100; i++)
  {
    g[i] = a;
    b = b + g[i];
    r = r * g[i];
```

例 7–2 source2 *(続き)*

```
  }

  a = b;
  return 0;
}

% cc -xopenmp -xO3 -xvpara source2.c
"source2.c", line 8: Warning: inappropriate scoping
        variable 'r' may be scoped inappropriately as 'reduction'
        . reference at line 13 may not be a reduction of the specified type

"source2.c", line 8: Warning: inappropriate scoping
        variable 'a' may be scoped inappropriately as 'private'
        . read at line 11 may be undefined
        . consider 'firstprivate'

"source2.c", line 8: Warning: inappropriate scoping
        variable 'i' may be scoped inappropriately as 'lastprivate'
        . value defined inside the parallel construct is not used outside
        . consider 'private'

"source2.c", line 8: Warning: inappropriate scoping
        variable 'b' may be scoped inappropriately as 'shared'
        . write at line 12 and write at line 12 may cause data race

"source2.c", line 8: Warning: inappropriate scoping
        variable 'b' may be scoped inappropriately as 'shared'
        . write at line 12 and read at line 12 may cause data race
```

上記の疑似的な例は、スコープチェックによって検出される典型的なスコープ宣言エラーを示します。

1. 縮約変数として `r` が指定され、この変数の演算は **+** となっているが、実際に行われるべき演算は `*` です。
2. **PRIVATE** として `a` が明示的にスコープ宣言されています。**PRIVATE** 変数には初期値がないので、行 11 の `a` への参照では、何らかのガベージ値が取得されることがあります。コンパイラはこの問題を指摘し、**FIRSTPRIVATE** としてスコープ宣言することをプログラマに提案します。
3. 変数 `i` はループインデックス変数です。ループインデックスがループ後に使用される場合、プログラマはこれを **LASTPRIVATE** として指定することもあります。しかし、これは上記の例には当てはまりません。`i` はループ後にはまったく参照されていません。コンパイラは警告を発行し、`i` を **PRIVATE** としてスコープ宣言するようプログラマに提案します。**PRIVATE** を **LASTPRIVATE** の代わりに使用すると、パフォーマンスが向上します。
4. プログラマは、変数 `b` に対してはデータ共有属性を明示的に指定しません。OpenMP 仕様 3.0 の 79 ページ、27 - 28 行によると、`b` は **SHARED** として暗黙的にスコープ宣言されます。ただし、`b` を **SHARED** としてスコープ宣言すると、データ競合が発生します。`b` の正しいデータ共有属性は **REDUCTION** です。

7.2 制限事項

- 上述のように、スコープチェックは、最適化レベルが **-xO3** かそれ以上に設定されている場合のみ動作します。スコープチェックは、プログラムが **-xopenmp=noopt** でコンパイルされただけでは動作しません。
- データ競合解析では、**BARRIER** や **MASTER** などの OpenMP 同期指令を使用して指定された同期のみ認識、使用されます。ビジー待ちなどのユーザー実装の同期は認識されません。

◆ ◆ ◆ 第 8 章

8 パフォーマンス上の検討事項

正しく機能する OpenMP プログラムを作成したら、その全体のパフォーマンスを検討してみてください。OpenMP アプリケーションの効率性とスケーラビリティーを向上させる際に利用できる一般的なテクニック、および Sun プラットフォームに固有のテクニックがあります。ここでは、そうしたテクニックを簡単に説明します。

追加情報については、Darryl Gove が作成した『Solaris Application Programming』(http://www.sun.com/books/catalog/solaris_app_programming.xml) を参照してください。

また、Oracle Solaris Studio ポータル (http://www.oracle.com/technetwork/server-storage/solarisstudio) では、OpenMP アプリケーションのパフォーマンス分析や最適化に関する記事やケーススタディを随時掲載していますので、ご参照ください。

8.1 一般的な推奨事項

OpenMP アプリケーションのパフォーマンスを向上させる一般的なテクニックとして、次のようなものがあります。

- 同期を回避する。
  - できる限り、**BARRIER**、**CRITICAL** 領域、**ORDERED** 領域、ロックの使用を回避してください。
  - 可能な場合は **NOWAIT** 節を使用して、冗長または不要なバリアを取り除いてください。たとえば、並列領域の最後につねに暗黙のバリアがあります。領域の最後の **DO** に **NOWAIT** を追加することによって、1 つの冗長なバリアが取り除かれます。
  - 名前付きの **CRITICAL** 領域を使用して、きめの細かいロックを行なってください。

- 明示的な **FLUSH** の使用には注意してください。フラッシュは、データキャッシュの内容をメモリーに退避させ、以降のデータアクセスで、メモリーからの再読み込みが必要になることがあります。このすべてが効率の低下になります。

デフォルトでは、アイドル状態のスレッドがある時間経過後にスリープします。デフォルトのタイムアウト期間がアプリケーションに対して不十分な場合、スレッドがスリープするのが早すぎたり、遅すぎたりすることがあります。**SUNW_MP_THR_IDLE** 環境変数を使用するとデフォルトのタイムアウト期間を上書きでき、アイドル状態のスレッドがスリープすることなく、常にアクティブなままにすることもできます。

- 外側の **DO/FOR** などをできる限り並列化させてください。1 つの並列領域で複数のループを囲みます。一般に、並列化のオーバーヘッドを抑制するには、並列領域をできる限り大きくします。次に例を示します。

This construct is less efficient:

```
!$OMP PARALLEL
  ....
  !$OMP DO
    ....
  !$OMP END DO
  ....
!$OMP END PARALLEL

!$OMP PARALLEL
  ....
   !$OMP DO
     ....
   !$OMP END DO
  ....
!$OMP END PARALLEL
```

than this one:

```
!$OMP PARALLEL
  ....
  !$OMP DO
    ....
  !$OMP END DO
  .....

  !$OMP DO
    ....
  !$OMP END DO

!$OMP END PARALLEL
```

- 並列領域では、ワークシェアリング **DO/FOR** 指令ではなく、**PARALLEL DO/FOR** を使用してください。複数のループが含まれることがある一般的な並列領域よりも、**PARALLEL DO/FOR** を実装した方が効率的です。次に例を示します。

This construct is less efficient:

```
!$OMP PARALLEL
  !$OMP DO
    .....
  !$OMP END DO
!$OMP END PARALLEL
```

than this one:

```
!$OMP PARALLEL DO
   ....
!$OMP END PARALLEL
```

- Solaris システムでは、**SUNW_MP_PROCBIND** を使用してスレッドをプロセッサに結合してください。static スケジュール指定とともにプロセッサ結合を使用すると、並列領域の前回呼び出し以降、その領域内のスレッドがアクセスするデータがローカルキャッシュに存在する、特定のデータ再利用パターンを持つアプリケーションにメリットがあります。詳細は、24 ページの「2.3 プロセッサ結合」を参照してください。
- 可能な場所では、できる限り **SINGLE** ではなく、**MASTER** を使用してください。
  - **MASTER** 指令は、暗黙の **BARRIER** のない **IF** として実装されます。
    IF(omp_get_thread_num() == 0) {…}
  - **SINGLE** 指令は、ほかのワークシェアリング構文に似た実装になります。どのスレッドが最初に **SINGLE** に達するかを記録するのは、実行時のオーバーヘッドの増加になります。**NOWAIT** が指定されていない場合、暗黙の **BARRIER** があります。これは効率の低下です。

  適切なループスケジュール指定を選択してください。

  - **STATIC** は同期オーバーヘッドの原因にならず、データがキャッシュに収まったとき、データのローカル性を維持できます。ただし、**STATIC** は、負荷の不均衡をもたらすことがあります。
  - **DYNAMIC,GUIDED** は、どのチャンクが割り当てられたかを記録するため、同期オーバーヘッドを招き、そのスケジュールによってデータのローカル性の低下をもたらすことがあります。ただし、負荷均衡が改善することがあります。チャンクのサイズを変えて試してください。

  オーバーヘッドが大きくなる可能性があるため、**LASTPRIVATE** の使用には注意してください。

  - 並列構文からの復帰時、データを占有領域から共有領域にコピーする必要があります。
  - コンパイル済みのコードは、どのスレッドが論理的に最後の反復を実行したか確認します。つまり、並列 **DO**/**FOR** 内の個々の分割単位の終わりで余分な仕事が生じることになります。分割数が多いと、オーバーヘッドが増加します。

  効率的なスレッドセーフのメモリー管理を使用してください。

- アプリケーションが明示的に、あるいは動的/割り当て可能な配列やベクトル化された組み込み関数などのコンパイラ生成のコードで **malloc()** および **free()** が使用されていることがあります。
- **libc** にあるスレッドセーフな **malloc()** および **free()** には、内部ブロックを原因とする大きな同期オーバーヘッドがあります。**libmtmalloc** ライブラリでは、より高速のバージョンが提供されています。**libmtmalloc** ライブラリを使用するには、リンクに **-lmtmalloc** を使用してください。

データが小さい場合、OpenMP の並列ループが十分に機能しないことがあります。**PARALLEL** 構文では **IF** 節を使用して、ある程度のパフォーマンス向上が期待できる場合にのみループを並列実行させるように指定します。

- 可能であれば、ループをマージしてください。次に例を示します。

2 つのループをマージ

```
!$omp parallel do
  do i = ...
```

statements_1

```
  end do
!$omp parallel do
  do i = ...
```

statements_2

```
  end do
```

1 つのループにする

```
!$omp parallel do
  do i = ...
```

statements_1

statements_2

```
  end do
```

- アプリケーションにある程度以上のスケーラビリティーがない場合は、入れ子並列処理を試してください。OpenMP での入れ子並列処理についての詳細は、14 ページの「1.2 このマニュアルで使用している特別な表記」を参照してください。

8.2 「偽りの共有」とその回避方法

OpenMP アプリケーションで不注意に共有メモリー構造体を使用すると、パフォーマンスおよびスケーラビリティーが低下することがあります。メモリー上の連続する共有データを複数のプロセッサが更新すると、マルチプロセッサインターコネクタに過度のトラフィックが生じ、結果的に計算の直列化の原因になることがあります。

8.2.1 「偽りの共有」とは

UltraSPARC プロセッサなどの大部分の高性能プロセッサでは、低速のメモリーと CPU の高速レジスタの間にキャッシュバッファーが 1 つ挿入されています。メモリー上の場所にアクセスすると、その要求された場所を含む実際のメモリーのスライス (キャッシュライン) がキャッシュにコピーされます。同じメモリー上の場所またはその周囲の場所への以降の参照は、多くの場合、キャッシュとメモリー間の整合性を維持する必要があるとシステムが判断するまで、キャッシュから満たすことができます。

ただし、同じキャッシュライン内の個々の要素に対する、異なるプロセッサからの同時更新があると、それらの更新が互いに論理的に独立していても、キャッシュライン全体の妥当性が失われます。このため、キャッシュラインの個別要素の更新があると、その都度、そのラインには「無効」のマークが付けられます。同じ行の別の要素にアクセスしている他のプロセッサは、*invalid* とマークされた行を参照しています。プロセッサは、アクセスされた要素に対して変更が加えられていない場合でも、より新しい行のコピーをメモリーなどから取得するようになっています。これは、キャッシュ整合性をキャッシュラインのレベルで維持するためであり、個別の要素のためではありません。この結果、インターコネクトのトラフィックとオーバーヘッドが増加することになります。また、キャッシュラインが更新中、そのライン上の要素へのアクセスは禁止されます。

この状態は「偽りの共有」と呼ばれます。頻繁にこの状態になる場合は、OpenMP アプリケーションのパフォーマンスとスケーラビリティーが大幅に低下します。

偽りの共有によってパフォーマンスが低下するのは、次の条件のすべてが満たされる場合です。

- 複数のプロセッサによって共有データが変更される。
- 複数のプロセッサが同じキャッシュライン内のデータを更新する。
- この更新が頻繁に発生する (たとえば、密なループなど)。

ループ内で読み取り専用の共有データは偽りの共有にはならないことに注意してください。

8.2.2 偽りの共有の低減

アプリケーションの実行で主要な役割を果たす並列ループを綿密に分析することによって、偽りの共有によって引き起こされるパフォーマンスおよびスケーラビリティー上の問題を明らかにすることができます。一般に、偽りの共有は次のことを行うことによって減らすことができます。

- できるだけ多くの非公開データを使用する。
- コンパイラの最適化機能を使用して、メモリーの読み込みおよびストア命令を取り除く。

場合によっては、大きなサイズの問題を処理しているときは共有が少ないために、偽りの共有の影響がわかりにくいことがあります。

偽りの共有を追跡するための技法は、アプリケーションによって大きく異なります。データの割り当て方法を変更すると、偽りの共有が減少する場合があります。スレッドの反復のマッピングを変更し、チャンクごとの各スレッドの作業量を増やす (*chunksize* 変数を変更する) ことでも、偽りの共有が減少することもあります。

8.3 Solaris OS のチューニング機能

Solaris 9 以降のオペレーティングシステムでは SunFire システム向けにスケーラビリティーとパフォーマンス向上が導入されています。中でも、MPO (Memory Placement Optimizations、メモリー配置の最適化) および MPSS (Multiple Page Size Support、複数ページサイズのサポート) が、ハードウェアのアップグレードなしに OpenMP プログラムのパフォーマンスを向上させる Solaris 9 の新機能として組み込まれました。

MPO によって、OS は、アクセスするプロセッサの近くにあるページをプロセッサに割り当てることができます。SunFire E20K および SunFire E25K システムは、同じ UniBoard 内と異なる UniBoard 間でメモリー待ち時間が異なります。「first-touch」というデフォルトの MPO ポリシーでは、メモリーに最初に接触するプロセッサが装着されている UniBoard 上のメモリーが割り当てられます。first-touch ポリシーは、first-touch 配置で、たいていのデータアクセスが各プロセッサにローカルのメモリーに行われるアプリケーションのパフォーマンスを大幅に改善することができます。メモリーがシステム全体に均等に分散されるランダムメモリー配置ポリシーと比較して、アプリケーションのメモリー待ち時間を短縮して帯域幅を増加することができ、その結果、パフォーマンスの向上につながります。

MPSS 機能は Solaris 9 OS リリース以降でサポートされ、プログラムが仮想メモリーの異なる領域で異なるページサイズを使用できます。Solaris のデフォルトのページサイズは比較的小さくなっています (UltraSPARC プロセッサで 8 K バイト、AMD64 Opteron プロセッサで 4 K バイト)。TLB ミスが多いと影響を受けるアプリケーションでは、大きいページサイズを使用するとパフォーマンスが向上することがあります。

TLB ミスは、Sun Performance Analyzer を使用して測定できます。

特定のプラットフォームでのデフォルトのページサイズは、Solaris OS コマンドの **/usr/bin/pagesize** を使用して取得できます。このコマンドで **-a** オプションを指定すると、サポートされるすべてのページサイズが表示されます。詳細は、**pagesize(1)** のマニュアルページを参照してください。

アプリケーションのデフォルトのページサイズを変更する方法は 3 つあります。

- Solaris OS コマンドの **ppgsz**(1) を使用する
- **-xpagesize**、**-xpagesize_heap**、および **-xpagesize_stack** の各オプション付きでアプリケーションをコンパイルする。詳細は、コンパイラのマニュアルページを参照してください。
- MPSS 固有の環境変数を使用する。詳細は、**mpss.so.1**(1) のマニュアルページを参照してください。

◆ ◆ ◆ 付 録 A

指令での節の記述

次の表は、節と指令およびプラグマとの関連を示しています。

表 A–1 節とともに記述できるプラグマ

節/プラグマ	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE
IF	はい				はい	はい	はい
PRIVATE	はい	はい	はい	はい	はい	はい	はい
SHARED	はい				はい	はい	はい
FIRSTPRIVATE	はい	はい	はい	はい	はい	はい	はい
LASTPRIVATE		はい	はい		はい	はい	
DEFAULT	はい				はい	はい	はい
REDUCTION	はい	はい	はい		はい	はい	はい
COPYIN	はい				はい	はい	はい
COPYPRIVATE				はい (1)			
ORDERED		はい			はい		
SCHEDULE		はい			はい		
NOWAIT		はい (2)	はい (2)	はい (2)			
NUM_THREADS	はい				はい	はい	はい
__AUTO	はい				はい	はい	はい

1. Fortran のみ: **COPYPRIVATE** を **END SINGLE** 指令で指定できます。
2. Fortran では、**NOWAIT** 修飾子を **END DO**、**END SECTIONS**、**END SINGLE**、または **END WORKSHARE** 指令でのみ使用できます。

3. **WORKSHARE** および **PARALLEL WORKSHARE** は、Fortran でだけサポートされています。

◆ ◆ ◆ 付録 B

OpenMP への変換

この章では、Sun または Cray の指令およびプラグマを使用する従来のプログラムを OpenMP に変換するための指針を説明します。

注 – 従来の Sun および Cray の並列化指令は廃止予定で、Solaris Studio コンパイラでサポートされなくなりました。

B.1 従来の Fortran 指令の変換

従来の Fortran プログラムでは、Sun または Cray 形式の並列化指令が使用されています。これらの指令の詳細については、『Fortran プログラミングガイド』の「並列化」に関する章を参照してください。

B.1.1 Sun 形式の Fortran の指令の変換

次の表は、Sun の並列化指令およびその従属節と、それに相当する OpenMP の指令の概要です。これらは、変換の一例です。

表 B–1 Sun の並列化指令を OpenMP の指令に変換する

Sun の指令	OpenMP の指令
C$PAR DOALL *[qualifiers]*	**!$omp parallel do** *[qualifiers]*
C$PAR DOSERIAL	完全に相当する句はありません。次で代用することができます。 **!$omp master** *loop* **!$omp end master**

表 B–1 Sun の並列化指令を OpenMP の指令に変換する (続き)

Sun の指令	OpenMP の指令
C$PAR DOSERIAL*	完全に相当する句はありません。次で代用することができます。 **!$omp master** *loopnest* **!$omp end master**
C$PAR TASKCOMMON *block[,...]*	**!$omp threadprivate (/block/***[,...]* **)**

DOALL 指令では、次の修飾節を指定することができます。

表 B–2 **DOALL** 修飾節とそれに相当する OpenMP の節

Sun の DOALL 節	OpenMP の PARALLEL DO に相当する節
PRIVATE(*v1,v2,...***)**	**private(***v1,v2,...***)**
SHARED(*v1,v2,...***)**	**shared(***v1,v2,...***)**
MAXCPUS(*n***)**	**num_threads(***n***)**完全に相当する句はありません。
READONLY(*v1,v2,...***)**	完全に相当する句はありません。**firstprivate(***v1,v2,...***)** を使用して同じ効果を得ることができます。
STOREBACK(*v1,v2,...***)**	**lastprivate(***v1,v2,...***)**.
SAVELAST	完全に相当する句はありません。**lastprivate(***v1,v2,...***)** を使用して同じ効果を得ることができます。
REDUCTION(*v1,v2,...***)**	**reduction(operator:***v1,v2,...***)** 縮約演算子および変数リストを指定する必要があります。
SCHEDTYPE(*spec***)**	**schedule(***spec***)** (表 B–3 を参照)

SCHEDTYPE(*spec***)** 節では、次のスケジュール指定を使用することができます。

表 B–3 **SCHEDTYPE** のスケジュール指定とそれに相当する OpenMP の **schedule**

SCHEDTYPE(spec)	OpenMP の schedule(*spec*) 節
SCHEDTYPE(STATIC)	**schedule(static)**
SCHEDTYPE(SELF(*chunksize***))**	**schedule(dynamic,***chunksize***)** デフォルトの *chunksize* の値は 1 です。
SCHEDTYPE(FACTORING(*m***))**	完全に相当する句はありません。

表 B–3 **SCHEDTYPE** のスケジュール指定とそれに相当する OpenMP の **schedule** *(続き)*

SCHEDTYPE(spec)	OpenMP の schedule(*spec*) 節
SCHEDTYPE(GSS(*m***))**	**schedule(guided,** *m***)** デフォルトの *m* の値は 1 です。

B.1.1.1 Sun 形式の Fortran の指令と OpenMP の変換の問題

- OpenMP では、非公開変数のスコープを明示的に宣言する必要があります。Sun の指令では、**PRIVATE** または **SHARED** 節で明示的にスコープが指定されていない変数の場合は、コンパイラは専用のデフォルトのスコープ規則を使用します。つまり、すべてのスカラーは **PRIVATE**、すべての配列参照は **SHARED** として処理されます。OpenMP では、**DEFAULT(PRIVATE)** 節を **PARALLEL DO** 指令で使用されている場合を除き、デフォルトのデータスコープは **SHARED** です。**DEFAULT(NONE)** 節を使用すると、コンパイラで変数のスコープが明示的に指定されません。Fortran での自動スコープに関しては 43 ページの「4.4 入れ子並列処理を使う際のヒント」を参照してください。
- **DOSERIAL** 指令がないため、自動と明示的な OpenMP の並列化を混在させると異なる結果になることがあります。Sun の指令では並列化されていなかったループが、自動的に並列化されることがあります。
- OpenMP では並列領域と並列セクションを用意しているため、並列化モデルが豊富です。したがって、Sun の指令を使用するプログラムの並列化戦略を再設計し、OpenMP の機能を利用するようにすることでパフォーマンスの向上を実現することができます。

B.1.2 Cray 形式の Fortran の指令の変換

Cray 形式の Fortran 並列化指令は、指令を示す標識が **!MIC$** である点を除き、Sun 形式のものと同一です。また、**!MIC$ DOALL** の修飾節も異なります。

表 B–4 Cray 形式の **DOALL** 修飾節とそれに相当する Open MP の節

Cray の DOALL 節	OpenMP の PARALLEL DO に相当する節
SHARED(*v1,v2,...***)**	**SHARED(***v1,v2,...***)**
PRIVATE(*v1,v2,...***)**	**PRIVATE(***v1,v2,...***)**
AUTOSCOPE	相当する句はありません。スコープは必ず、明示的に指定するか、**DEFAULT** 節か **__AUTO** 節と共に指定します。
SAVELAST	完全に相当する句はありません。**lastprivate** を使用して同じ効果を得ることができます。
MAXCPUS(n)	**num_threads(***n***)**完全に相当する句はありません。

表 B–4 Cray 形式の **DOALL** 修飾節とそれに相当する Open MP の節 *(続き)*

Cray の DOALL 節	OpenMP の PARALLEL DO に相当する節
GUIDED	**schedule(guided,** *m***)** デフォルトの *m* の値は 1 です。
SINGLE	**schedule(dynamic,1)**
CHUNKSIZE(n)	**schedule(dynamic,***n***)**
NUMCHUNKS(m)	**schedule(dynamic,***n/m***)** ここで、*n* には反復数を指定します。

B.1.2.1 Cray 形式の Fortran の指令と OpenMP の指令の変換の問題

両者の違いは、Cray の **AUTOSCOPE** に相当するものがない点を除き、Sun 形式の指令の場合と同様です。

B.2 従来の C プラグマの変換

C コンパイラでは、明示的な並列化用の従来のプラグマを使用することができます。これらのプラグマについては、『C ユーザーズガイド』を参照してください。Fortran の指令の場合と同様に、これらは一例です。

従来の並列化プラグマは、次のとおりです。

表 B–5 C の並列化プラグマを OpenMP に変換する

従来の C プラグマ	相当する OpenMP プラグマ
#pragma MP taskloop *[clauses]*	**#pragma omp parallel for** *[clauses]*
#pragma MP serial_loop	完全に相当する句はありません。次で代用することができます。 **#pragma omp master** *loop*
#pragma MP serial_loop_nested	完全に相当する句はありません。次で代用することができます。 **#pragma omp master** *loopnest*

taskloop プラグマでは、次の節を指定できます。

表 B–6 **taskloop** の節とそれに相当する OpenMP の節

taskloop の節	OpenMP の **parallel for** に相当する節
maxcpus(*n***)**	完全に相当する句はありません。**num_threads(***n***)** を使用します。
private(*v1,v2,...***)**	**private(***v1,v2,...***)**
shared(*v1,v2,...***)**	**shared(***v1,v2,...***)**
readonly(*v1,v2,...***)**	完全に相当する句はありません。**firstprivate(***v1,v2,...***)** を使用して同じ効果を得ることができます。
storeback(*v1,v2,...***)**	**lastprivate(***v1,v2,...***)** を使用して同じ効果を得ることができます。
savelast	完全に相当する句はありません。**lastprivate(***v1,v2,...***)** を使用して同じ効果を得ることができます。
reduction(*v1,v2,...***)**	**reduction(operator:***v1,v2,...***)** 縮約演算子および変数リストを指定する必要があります。
schedtype(*spec***)**	**schedule(***spec***)** (表 B–7 を参照)

schedtype(*spec***)** 節では、次のスケジュール指定を使用することができます。

表 B–7 **SCHEDTYPE** のスケジュール指定とそれに相当する OpenMP の **schedule**

schedtype(spec)	OpenMP の schedule(*spec*) 節
SCHEDTYPE(STATIC)	**schedule(static)**
SCHEDTYPE(SELF(*chunksize***))**	**schedule(dynamic,***chunksize***)** 注: デフォルトの *chunksize* の値は 1 です。
SCHEDTYPE(FACTORING(*m***))**	完全に相当する句はありません。
SCHEDTYPE(GSS(*m***))**	**schedule(guided,** *m***)** デフォルトの *m* の値は 1 です。

B.2.1 従来の C のプラグマと OpenMP の変換の問題

- OpenMP では、並列構文内で宣言された変数のスコープは **private** になります。**#pragma omp parallel for** 指令で **default(none)** 節を使用すると、コンパイラで変数のスコープが明示的に指定されません。

- **serial_loop** 指令がないため、自動と明示的な OpenMP の並列化を混在させると異なる結果になることがあります。従来の C の指令では並列化されていなかったループが、自動的に並列化されることがあります。
- OpenMP の方が並列化モデルが豊富なため、従来の C の指令を使用するプログラムの並列化戦略を再設計し、OpenMP の機能を利用することで、多くの場合はパフォーマンスを向上できます。

索引