OpenMP API ユーザーズガイド

Sun™ Studio 10

Sun Microsystems, Inc.
www.sun.com

Part No. 819-1616-10
2005 年 1 月, Revision A

原典:　*OpenMP API User's Guide : Sun Studio 10*
Part No: 819-0501-10
Revision A

目次

表目次

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。Sun™ 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。さらに、Solaris™ オペレーティング環境または UNIX® について一般的な知識を有することを前提とします。

書体と記号について

書体または記号*	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。	.login ファイルを編集します。 ls -a を実行します。 % You have mail.
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表します。	マシン名% **su** Password:
AaBbCc123 または**ゴシック**	コマンド行の可変部分。実際の名前や値と置き換えてください。	rm *filename* と入力します。 rm **ファイル名** と入力します。
『 』	参照する書名を示します。	『Solaris ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照。 この操作ができるのは「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅をこえる場合に、継続を示します。	% **grep '^#define ** **XV_VERSION_STRING'**

* 使用しているブラウザにより、これら設定と異なって表示される場合があります。

コードの記号	意味	記法	コード例
[ ]	角括弧には、オプションの引数が含まれます。	O[*n*]	-O4,-O
{ }	中括弧には、必須オプションの選択肢が含まれます。	d{y|n}	-dy
|	「パイプ」または「バー」と呼ばれる記号は、その中から 1 つだけを選択可能な複数の引数を区切ります。	B{dynamic|static}	-Bstatic
:	コロンは、コンマ同様に複数の引数を区切るために使用されることがあります。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略記号は、連続するものの一部が省略されていることを示します。	-xinline=*f1*[,…*fn*]	-xinline=alpha,dos

シェルプロンプトについて

シェル	プロンプト
UNIX の C シェル	マシン名%
UNIX の Bourne シェルと Korn シェル	$
スーパーユーザー (シェルの種類を問わない)	#

サポートされるプラットフォーム

この Sun Studio 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をサポートしています。サポートされるシステムの、Solaris オペレーティングシステムのバージョンごとの情報については、`http://www.sun.com/bigadmin/hcl` にあるハードウェアの互換性に関するリストで参照することができます。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" という用語は、AMD64 または Intel Xeon/Pentium 製品ファミリと互換性があるプロセッサを使用して製造された 64 ビットおよび 32 ビットのシステムを指します。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。

Sun Studio ソフトウェアおよびマニュアルページへのアクセス

コンパイラおよびツールは、標準の `/usr/bin/` および `/usr/share/man` の各ディレクトリにはインストールされません。コンパイラおよびツールにアクセスするには、`PATH` 環境変数を正しく設定しておく必要があります (xii ページの「コンパイラとツールへのアクセス方法」を参照)。また、マニュアルページにアクセスするには、`MANPATH` 環境変数を正しく設定しておく必要があります (xiii ページの「マニュアルページへのアクセス方法」を参照)。

PATH 変数についての詳細は、csh(1)、sh(1)、および ksh(1) のマニュアルページを参照してください。MANPATH 変数についての詳細は、man(1) のマニュアルページを参照してください。このリリースにアクセスするために PATH および MANPATH 変数を設定する方法の詳細は、『インストールガイド』を参照するか、システム管理者にお問い合わせください。

注 – この節に記載されている情報は Sun Studio のコンパイラとツールが /opt ディレクトリにインストールされていることを想定しています。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

コンパイラとツールへのアクセス方法

PATH 環境変数を変更して、コンパイラとツールにアクセスできるようにする必要があるかどうか判断するには以下を実行します。

▼ PATH 環境変数を設定する必要があるかどうか判断する

1. **次のように入力して、PATH 変数の現在値を表示します。**

```
% echo $PATH
```

2. **出力内容から /opt/SUNWspro/bin を含むパスの文字列を検索します。**

   パスがある場合は、PATH 変数はコンパイラとツールにアクセスできるように設定されています。このパスがない場合は、次の手順に従って、PATH 環境変数を設定してください。

▼ PATH 環境変数を設定してコンパイラとツールにアクセスする

1. **C シェルを使用している場合は、ホームの .cshrc ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの .profile ファイルを編集します。**

2. **次のパスを PATH 環境変数に追加します。Forte Developer ソフトウェア、Sun ONE Studio ソフトウェア、または Sun Studio の他のリリースをインストールしている場合は、インストール先のパスの前に、次のパスを追加します。**

   /opt/SUNWspro/bin

マニュアルページへのアクセス方法

マニュアルページにアクセスするために MANPATH 環境変数を変更する必要があるかどうかを判断するには以下を実行します。

▼ MANPATH 環境変数を設定する必要があるかどうか判断する

1. 次のように入力して、dbx のマニュアルページを表示します。

```
% man dbx
```

2. 出力された場合、内容を確認します。

   dbx(1) のマニュアルページが見つからないか、表示されたマニュアルページがインストールされたソフトウェアの現バージョンのものと異なる場合は、この節の指示に従って、MANPATH 環境変数を設定してください。

▼ MANPATH 環境変数を設定してマニュアルページにアクセスする

1. C シェルを使用している場合は、ホームの .cshrc ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの .profile ファイルを編集します。
2. 次のパスを MANPATH 環境変数に追加します。

   /opt/SUNWspro/man

統合開発環境へのアクセス方法

Sun Studio 統合開発環境 (IDE) には、C や C++、Fortran アプリケーションを作成、編集、構築、デバッグ、パフォーマンス解析するためのモジュールが用意されています。

IDE を起動するコマンドは、sunstudio です。このコマンドの詳細は、sunstudio(1) のマニュアルページを参照してください。

IDE が正しく動作するかどうかは、IDE がコアプラットフォームを検出できるかどうかに依存します。このため、sunstudio コマンドは、次の 2 つの場所でコアプラットフォームを探します。

- コマンドは、最初にデフォルトのインストールディレクトリ /opt/netbeans/3.5V を調べます。

- このデフォルトのディレクトリでコアプラットフォームが見つからなかった場合は、IDE が含まれているディレクトリとコアプラットフォームが含まれているディレクトリが同じであるか、同じ場所にマウントされているとみなします。たとえば IDE が含まれているディレクトリへのパスが `/foo/SUNWspro` の場合は、`/foo/netbeans/3.5V` ディレクトリにコアプラットフォームがないか調べます。

`sunstudio` が探す場所のどちらにもコアプラットフォームをインストールしていないか、マウントしていない場合、クライアントシステムの各ユーザーは、コアプラットフォームがインストールされているか、マウントされている場所 (*/installation_directory*`/netbeans/3.5V`) を、`SPRO_NETBEANS_HOME` 環境変数に設定する必要があります。

Forte Developer ソフトウェア、Sun ONE Studio ソフトウェア、または 他のバージョンの Sun Studio ソフトウェアがインストールされている場合、IDE の各ユーザーは、`$PATH` のそのパスの前に、*/installation_directory*`/SUNWspro/bin` を追加する必要もあります。

`$PATH` には、*/installation_directory*`/netbeans/3.5V/bin` のパスは追加しないでください。

コンパイラとツールのマニュアルへのアクセス方法

マニュアルには、以下からアクセスできます。

- 製品マニュアルは、ご使用のローカルシステムまたはネットワークの製品にインストールされているマニュアルの索引から入手できます。
  `file:/opt/SUNWspro/docs/ja/index.html`

  製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。
- マニュアルは、`docs.sun.com`℠ の Web サイトで入手できます。以下に示すマニュアルは、インストールされている製品のマニュアルの索引から入手できます (`docs.sun.com` Web サイトでは入手できません)。
  - 『Standard C++ Library Class Reference』
  - 『標準 C++ ライブラリ・ユーザーズガイド』
  - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
  - 『Tools.h++ ユーザーズガイド』
- リリースノートは、`docs.sun.com` で入手できます。

- IDE の全コンポーネントのオンラインヘルプは、IDE 内の「ヘルプ」メニューだけでなく、多くのウィンドウおよびダイアログにある「ヘルプ」ボタンを使ってアクセスできます。

インターネットの Web サイト (`http://docs.sun.com`) から、Sun のマニュアルを参照したり、印刷したり、購入することができます。マニュアルが見つからない場合はローカルシステムまたはネットワークの製品とともにインストールされているマニュアルの索引を参照してください。

注 – Sun では、本マニュアルに掲載した第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関して一切の責任を負いません。Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスのご利用あるいは信頼によって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

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以下の表に示す場所から参照することができます。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

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 (サードパーティ製マニュアルは除く)	形式 : HTML 場所 : http://docs.sun.com
サードパーティ製マニュアル • 『Standard C++ Library Class Reference』 • 『標準 C++ ライブラリ・ユーザーズガイド』 • 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 • 『Tools.h++ ユーザーズガイド』	形式 : HTML 場所 : file:/opt/SUNWspro/docs/ja/index.html のマニュアル索引
Readme およびマニュアルページ	形式 : HTML 場所 : file:/opt/SUNWspro/docs/ja/index.html のマニュアル索引
オンラインヘルプ	形式 : HTML 場所 : IDE 内の「ヘルプ」メニュー
リリースノート	形式 : HTML 場所 : http://docs.sun.com

コンパイラとツールに関する関連マニュアル

以下の表は、file:/opt/SUNWspro/docs/ja/index.html および http://docs.sun.com から参照できるマニュアルの一覧です。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルタイトル	内容の説明
Fortran プログラミングガイド	入出力、ライブラリ、パフォーマンス、デバッグ、並列処理などに関する、Solaris 環境における効果的な Fortran コードの書き方について説明しています。
Fortran ライブラリ・リファレンス	Fortran ライブラリと組み込みルーチンについて詳しく説明しています。
Fortran ユーザーズガイド	`f95` コンパイラのコンパイル時環境とコマンド行オプションについて説明しています。古い `f77` プログラムを `f95` に移行させる際の指針も含まれています。
C ユーザーズガイド	`cc` コンパイラのコンパイル時環境とコマンド行オプションについて説明しています。
C++ ユーザーズガイド	`CC` コンパイラのコンパイル時環境とコマンド行オプションについて説明しています。
数値計算ガイド	浮動小数点演算における数値の正確性に関する問題について説明しています。

関連する Solaris マニュアル

次の表では、`docs.sun.com` の Web サイトで参照できる関連マニュアルについて説明します。

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris Reference Manual Collection	マニュアルページのセクションのタイトルを参照。	Solaris のオペレーティング環境に関する情報を提供しています。
Solaris Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明します。

開発者向けのリソース

http://developers.sun.com/prodtech/cc にアクセスし、以下のようなリソースを利用できます。リソースは頻繁に更新されます。

- プログラミング技術と最適な演習に関する技術文書
- プログラミングに関する簡単なヒントを集めた知識ベース
- コンパイラとツールのコンポーネントのマニュアル、ソフトウェアとともにインストールされるマニュアルの訂正
- サポートレベルに関する情報
- ユーザーフォーラム
- ダウンロード可能なサンプルコード
- 新しい技術の紹介

http://developers.sun.com でも開発者向けのリソースが提供されています。

技術サポートへの問い合わせ

製品についての技術的なご質問がございましたら、以下のサイトからお問い合わせください (このマニュアルで回答されていないものに限ります)。

http://jp.sun.com/service/contacting

第1章

OpenMP API の概要

OpenMP™ Application Program Interface (API) は、共有メモリー型マルチプロセッサアーキテクチャ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。チュートリアルおよびその他の関連ドキュメントなど OpenMP に関する詳細については、次の Web サイトを参照してください。
`http://www.openmp.org/`

OpenMP API は、Solaris™ プラットフォームで動作するすべての Sun Studio コンパイラの推奨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、第 6 章を参照してください。

この章では、OpenMP Version 2.0 API を構成する指令、実行時ライブラリルーチン、環境変数について説明します。この API は、Sun Studio の Fortran 95、C、C++ のコンパイラで実装されています。

1.1 OpenMP 仕様の参照先

この章では、簡潔にするため、OpenMP の概要だけを説明しており、詳細情報の多くを省略しています。詳細については、『OpenMP 仕様書』を参照してください。

Fortran、C、C++ の OpenMP 2.0 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。

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。

[...] (角括弧) 内の要素は省略可能です。

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。

1.3 指令の書式

指令行では「指令名を」1 つだけ指定することができ、指定した指令は後続のプログラム文に適用されます。

Fortran:

Fortran の固定書式では 3 つ、自由書式では 1 つだけ、標識を使用することができます。後述の Fortran の表および例では、自由書式が使用されています。

C/C++:

C および C++ は、**`#pragma omp`** で始まる標準のプリプロセッサ指令を使用します。

OpenMP 2.0 Fortran

固定書式

C$OMP *directive-name optional_clauses...*

!$OMP *directive-name optional_clauses...*

*$OMP *directive-name optional_clauses...*

標識は、カラム 1 から開始する必要があります。継続行については、カラム 6 で空白およびゼロ以外の文字が指定されている必要があります。

コメントは、指令行のカラム 6 以降に、感嘆符 (**!**) に続けて入力することができます。行中の **!** 以降の部分は無視されます。

自由書式

!$OMP *directive-name optional_clauses...*

任意の場所で指定でき、その前に空白があってもかまいません。継続行は、行の最後にあるアンパサンド (**&**) により識別されます。

コメントは、指令行で感嘆符 (!) に続けて入力することができます。以降の部分は無視されます。

OpenMP 2.0 C/C++

#pragma omp *directive-name optional_clauses...*

各プラグマは、改行文字で終了する必要があります。また、標準の C および C++ のコンパイラプラグマの表記に従う必要があります。

プラグマでは、大文字と小文字が区別されます。句の表記順序には意味はありません。**#** の前後および単語の間には空白文字を入力することができます。

指令は、後続の文 (構造ブロックでなければならない) に適用されます。

1.4 条件付きコンパイル

OpenMP API では、条件付きコンパイルに使用するプリプロセッサ記号 **_OPENMP** が定義されています。また、OpenMP Fortran API では、条件付きコンパイル標識を使用できます。

OpenMP 2.0 Fortran

固定書式

```
!$      fortran_95_statement
C$      fortran_95_statement
*$      fortran_95_statement
c$      fortran_95_statement
```

標識は、空白文字を入れずにカラム 1 から開始する必要があります。OpenMP コンパイルが有効な場合は、標識は 2 つの空白文字に置き換えられます。行のそれ以外の部分は、標準の Fortran の固定書式の表記に従って入力する必要があります。
例 :

```
C23456789
!$ 10 iam = OMP_GET_THREAD_NUM() +
!$   1       index
```

自由書式

```
!$ fortran_95_statement
```

この標識は、任意のカラムで指定できます。標識の前には空白文字だけを入力し、1 語で表記する必要があります。行のそれ以外の部分には、Fortran の自由書式の表記を使用します。
例 :

```
C23456789
 !$ iam = OMP_GET_THREAD_NUM() +     &
 !$&         index
```

Fortran プリプロセッサ :

OpenMP を有効にしたコンパイルでは、プリプロセッサ記号 **_OPENMP** が定義されます。

```
#ifdef _OPENMP
     iam = OMP_GET_THREAD_NUM()+index
#endif
```

OpenMP 2.0 C/C++

C/C++ プリプロセッサ :
OpenMP を有効にしたコンパイルでは、マクロ **_OPENMP** が定義されます。

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

1.5 PARALLEL - 並列領域構文

PARALLEL 指令は、並列領域を定義します。並列領域は、複数のスレッドで並列で実行されるプログラムの領域です。

OpenMP 2.0 Fortran

```
!$OMP PARALLEL [clause[[,]clause]...]
   structured-block
!$OMP END PARALLEL
```

OpenMP 2.0 C/C++

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

特別な条件や制限が多数存在します。詳細については、『OpenMP 仕様書』を参照してください。

表 1-1 は、この構文とともに記述できる句を示しています。

1.6 ワークシェアリング構文

ワークシェアリング構文は、構文内のコード領域を、検出したスレッドのチームのメンバー間で分割して実行します。ワークシェアリング構文を並列で実行するには、構文を並列領域内に記述する必要があります。

これらの指令およびそれらが適用されるコードについては特別な条件と制限が多数あります。詳細については、『OpenMP 仕様書』を参照してください。

1.6.1 DO、for 構文

DO または **for** ループの反復が並列に実行されるよう指定します。

OpenMP 2.0 Fortran

```
!$OMP DO [clause[[,] clause]...]
  do_loop
[!$OMP END DO [NOWAIT]]
```

DO 指令は、直後の **DO** ループの反復を並列で実行するように指定します。このループの反復は、ループに結合されている並列領域を実行する、チーム内の既存のスレッド全体に分散されます。この指令は、並列領域内に記述する必要があります。

OpenMP 2.0 C/C++

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

for プラグマは、直後の *for-loop* の反復を並列で実行するように指定します。このループの反復は、ループに結合されている並列領域を実行する、チーム内の既存のスレッド全体に分散されます。このプラグマは、並列領域内に記述する必要があります。**for** プラグマでは、対応する **for** ループの構文を次のような「正規の形式」で記述する必要があります。

```
for (initexpr; var logicop b; increxpr)
```

ここで、各要素の意味は以下のとおりです。

- *initexpr* には、以下のいずれかを指定します。
  - *var = lb*
  - *integer_type var = lb*
- *increxpr* には、以下のいずれかの形式の式を指定します。
  - *++var*
  - *var++*
  - *--var*
  - *var--*
  - *var += incr*
  - *var -= incr*
  - *var = var + incr*
  - *var = incr + var*
  - *var = var - incr*
- *var* は符号付き整数型の変数で、**for** の範囲で暗黙に非公開となります。*var* は、**for** 文の本体内では変更することはできません。値は、**lastprivate** を指定した場合を除き、ループ以降は不定になります。
- *logicop* には、以下のいずれかの論理演算子を指定します。

  `<  <=  >  >=`
- *lb*、*b*、*incr* は、ループの整数の不変式です。

< または <= と > または >= を **for** 文の *logicop* として使用する場合、さらに制限があります。詳細については、『OpenMP C/C++ の仕様』を参照してください。

表 1-1 は、この構文とともに記述できる句を示しています。

1.6.2 SECTIONS 構文

SECTIONS 構文には、チーム内のスレッド間で分割される一連の構造コードブロックが含まれます。各ブロックは、チーム内のスレッドによって 1 回実行されます。

各セクションの前に **SECTION** 指令を記述します。この指令は、最初のセクションについては省略可能です。

OpenMP 2.0 Fortran

```
!$OMP SECTIONS [clause[[,] clause]...]
[!$OMP SECTION]
  structured-block
[!$OMP SECTION]
 structured-block ]
...
!$OMP END SECTIONS [NOWAIT]
```

OpenMP 2.0 C/C++

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

表 1-1 は、この構文とともに記述できる句を示しています。

1.6.3 SINGLE 構文

SINGLE で囲まれた構文ブロックは、チーム内の 1 つのスレッドだけによって実行されます。チーム内で **SINGLE** ブロックを実行していないスレッドは、**NOWAIT** を指定した場合を除き、ブロックの最後で待機します。

OpenMP 2.0 Fortran

```
!$OMP  SINGLE [clause[[,] clause]...]
  structured-block
!$OMP  END  SINGLE [end-modifier]
```

OpenMP 2.0 C/C++

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

表 1-1 は、この構文とともに記述できる句を示しています。

1.6.4 Fortran の **WORKSHARE** 構文

WORKSHARE 構文では、コードブロックを実行する処理が複数の作業単位に分割されます。各単位が 1 回ずつだけ実行されるように、チームのスレッド間で作業が分割されます。

OpenMP 2.0 Fortran

```
!$OMP  WORKSHARE
  structured-block
!$OMP  END  WORKSHARE [NOWAIT]
```

C/C++ で Fortran の WORKSHARE に相当する構文はありません。

1.7 並列/ワークシェアリング複合構文

並列/ワークシェアリング複合構文は、1 つのワークシェアリング構文を含む並列領域を指定するためのショートカットです。

これらの指令およびそれらが適用されるコードについては特別な条件と制限が多数あります。詳細については、適切な OpenMP の仕様書を参照してください。以下の説明は概要であり、完全なものではありません。

表 1-1 は、この構文とともに記述できる句を示しています。

1.7.1 PARALLEL DO、parallel for 構文

DO ループまたは **for** ループを 1 つ含む並列領域を指定するためのショートカットです。**PARALLEL** 指令の直後に **DO** 指令または **for** 指令を続けた場合と同じ意味になります。*clause* には、**PARALLEL** と **DO** または **for** の指令で使用可能な句を指定できますが、**NOWAIT** 修飾子は指定できません。

OpenMP 2.0 Fortran

```
!$OMP PARALLEL DO [clause[[,] clause]...]
  do_loop
[!$OMP END PARALLEL DO ]
```

OpenMP 2.0 C/C++

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

1.7.2 PARALLEL SECTIONS 構文

SECTIONS 指令を 1 つ含む並列領域を指定するためのショートカットです。**PARALLEL** 指令の直後に **SECTIONS** 指令を続けた場合と同じ意味になります。*clause* には、**PARALLEL** および **SECTIONS** 指令で使用可能な句を指定できますが、**NOWAIT** 修飾子は指定できません。

OpenMP 2.0 Fortran

```
!$OMP PARALLEL SECTIONS [clause[[,] clause]...]
[!$OMP SECTION]
  structured-block
[!$OMP SECTION]
  structured-block ]
...
!$OMP END PARALLEL SECTIONS
```

OpenMP 2.0 C/C++

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

1.7.3 PARALLEL WORKSHARE 構文

Fortran の **PARALLEL WORKSHARE** 構文は、**WORKSHARE** 指令を 1 つ含む並列領域を指定するためのショートカットです。*clause* には、**PARALLEL** 指令で使用可能な句を指定できます。

OpenMP 2.0 Fortran

```
!$OMP PARALLEL WORKSHARE [clause[[,] clause]...]
  structured-block
!$OMP END PARALLEL WORKSHARE
```

C/C++ には対応する構文はありません。

1.8 同期構文

次の構文は、スレッド同期化を指定します。これらの構文については特別な条件および制限がありますが、その数が多すぎるため、ここでは説明しません。詳細については、『OpenMP 仕様書』を参照してください。

1.8.1 MASTER 構文

チームのマスタースレッドだけが、この指令で囲まれたブロックを実行します。他のスレッドは、このブロックをスキップして続行します。マスター構文の入り口や出口には、暗黙のバリアはありません。

OpenMP 2.0 Fortran

```
!$OMP MASTER
  structured-block
!$OMP END MASTER
```

OpenMP 2.0 C/C++

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

1.8.2 CRITICAL 構文

構文ブロックへのアクセスを、同時に 1 スレッドだけに制限します。省略可能な *name* 引数には、クリティカル領域の名前を指定します。名前指定のない **CRITICAL** 指令は、すべて同一名にマッピングされます。クリティカル領域の名前はプログラムの大域要素であり、一意の名前を指定する必要があります。Fortran では、**CRITICAL** 指令で *name* に名前を指定した場合は、**END CRITICAL** 指令でもその名前を記述する必要があります。C/C++ の場合は、クリティカル領域を指定する識別子は外部にリンクされていて、ラベル、タグ、メンバー、通常の識別子で使用される名前空間とは別の名前空間に含まれています。

OpenMP 2.0 Fortran

```
!$OMP CRITICAL [(name)]
  structured-block
!$OMP END CRITICAL [(name)]
```

OpenMP 2.0 C/C++

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

1.8.3 **BARRIER** 構文

チーム内のすべてのスレッドの同期をとります。各スレッドは、チーム内の他のすべてのスレッドがこの地点に到達するまで待機します。

OpenMP 2.0 Fortran

```
!$OMP BARRIER
```

OpenMP 2.0 C/C++

```
#pragma omp barrier
```

チーム内のすべてのスレッドがこのポイントに到達すると、チーム内の各スレッドは、**BARRIER** 指令の後ろの文の並列実行を開始します。

barrier プラグマの構文には C/C++ の文が含まれていないため、このプラグマをプログラム内に配置する際には制限があります。詳細については、『OpenMP C/C++ の仕様』を参照してください。

1.8.4 **ATOMIC** 構文

特定のメモリー位置への更新が不可分な形で行われるようにし、複数のスレッドによる同時書き込みが生じないようにします。

OpenMP 2.0 Fortran

```
!$OMP ATOMIC
  expression-statement
```

指令は *expression-statement* だけに適用されます。直後の文は、以下のいずれかの書式で指定します。

x = x operator expression

x = expression operator x

x = intrinsic(x, expr-list)

x = intrinsic(expr-list, x)

ここで、各要素の意味は以下のとおりです。

- *x* は、組み込み型のスカラーです。
- *expression* は、*x* を参照しないスカラー式です。
- *expr-list* は、x を参照しないスカラー式のリストです。空でない、コンマで区切ったリストを指定します (詳細については、OpenMP Fortran 2.0 仕様を参照)。
- *intrinsic* には、**MAX**、**MIN**、**IAND**、**IOR**、**IEOR** のいずれかを指定します。
- *operator* には、**+**、**-**、*****、**/**、**.AND.**、**.OR.**、**.EQV.**、**.NEQV.** のいずれかを指定します。

OpenMP 2.0 C/C++

```
#pragma omp atomic
    expression-statement
```

指令は *expression-statement* だけに適用されます。直後の文は、以下のいずれかの書式で指定します。

x binop = expr

x++

++*x*

x--

--*x*

ここで、各要素の意味は以下のとおりです。

- *x* は、スカラー型の lvalue 式です。
- *expr* は、*x* を参照しないスカラー型の式です。
- *binop* は、多重定義された演算子以外の演算子で、**+**、*****、**-**、**/**、**&**、**^**、**|**、**<<**、**>>** のいずれかです。

実装上は、すべての ATOMIC 指令は *expression-statement* をクリティカル領域内に配置する形に置き換えられます。

1.8.5 FLUSH 構文

スレッド可視の Fortran 変数および C オブジェクトは、この指令の記述されている地点でメモリーに書き戻されます。**FLUSH** 指令は、実行中のスレッドおよび全体のメモリー内の処理間でだけ整合性を保証します。省略可能な *variable-list* は、フラッシュする必要のある変数またはオブジェクトをコンマで区切ったリストです。*variable-list* を指定せずに **FLUSH** 指令を記述した場合は、すべてのスレッド可視の共有変数またはオブジェクトの同期をとります。

OpenMP 2.0 Fortran

```
!$OMP FLUSH [(variable-list)]
```

OpenMP 2.0 C/C++

```
#pragma omp flush [(variable-list)]
```

flush プラグマの構文には C/C++ の文が含まれていないため、このプラグマをプログラム内に配置する際には制限があります。詳細については、『OpenMP C/C++ の仕様』を参照してください。

1.8.6 **ORDERED** 構文

ordered 指令で囲まれた部分のブロックは、ループで逐次実行された場合の順序で実行されます。

OpenMP 2.0 Fortran

```
!$OMP ORDERED
  structured-block
!$OMP END ORDERED
```

ORDERED 指令で囲まれた部分のブロックは、ループで逐次実行された場合の順序で実行されます。**DO** または **PARALLEL DO** 指令の動的な範囲内でだけ指定することができます。**ORDERED** 句は、ブロックを囲むもっとも近い **DO** 指令で指定する必要があります。**DO** 指令が適用されるループでは、同一の **ordered** 指令を 1 回のループで複数回実行することはできません。また、複数の **ordered** 指令を実行することもできません。

OpenMP 2.0 C/C++

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

ordered 指令で囲まれた部分のブロックは、ループで逐次実行された場合の順序で実行されます。このブロックは、**ordered** 句が指定されている **for** または **parallel for** 指令の動的な範囲内でのみ指定できます。
for 構文のあるループでは、同一の **ordered** 指令を 1 回のループで複数回実行することはできません。また、複数の **ordered** 指令を実行することもできません。

1.9 データ環境指令

以下の指令は、並列構文の実行時のデータ環境を設定します。

1.9.1 **THREADPRIVATE** 指令

オブジェクト (Fortran の共通ブロックと名前付き変数、C と C++ の名前付き変数) の *list* を、他のスレッドに対しては非公開に、スレッド内では広域に設定します。

完全な仕様と制限については OpenMP 仕様を参照してください。

OpenMP 2.0 Fortran

```
!$OMP THREADPRIVATE(list)
```

共通ブロック名は、スラッシュで囲む必要があります。共通ブロックを **THREADPRIVATE** に設定するには、この指令をそのブロックのすべての **COMMON** 宣言の後に記述する必要があります。

OpenMP 2.0 C/C++

```
#pragma omp threadprivate (list)
```

list に指定した、ファイル、名前空間、ブロックスコープ内の変数は、字句的にこのプラグマの前に指定されているファイル、名前空間、ブロックスコープ内の変数宣言を参照する必要があります。

1.10 OpenMP 指令の句

ここでは、OpenMP 指令で記述可能な、データスコープおよびスケジュールを指定する句について説明します。

1.10.1 データスコープを指定する句

指令によっては、構文の範囲内で変数のスコープを設定できる句を使用できます。データスコープを指定する句を指令で使用していない場合は、指令が適用される変数のデフォルトのスコープは **SHARED** になります。

Fortran: *list* は、有効範囲内のアクセス可能な名前付き変数または共通ブロックを、コンマで区切ったリストです。共通ブロック名は、スラッシュで囲む必要があります (例 **/ABLOCK/**)。

スコープを指定する句の使用については、重要な制限があります。詳細については、OpenMP 仕様書の適切な個所を参照してください。

表 1-1 は、この構文とともに記述できる句を示しています。

1.10.1.1 PRIVATE 句

private(*list***)**

list に指定した変数 (コンマで区切って複数指定可能) を、チーム内の各スレッドに対して非公開として宣言します。

1.10.1.2 SHARED 句

private(*list***)**

チーム内のすべてのスレッドは、*list* の変数を共有し、同一の記憶領域を使用します。

1.10.1.3 DEFAULT 句

Fortran:

DEFAULT(PRIVATE | SHARED | NONE)

C/C++:

default(shared | none)

並列領域内のすべての変数のスコープを指定します。**THREADPRIVATE** 変数は、この句の影響を受けません。指定しない場合は、**DEFAULT(SHARED)** が使用されます。変数のデフォルトのデータ共有属性は、**private**、**firstprivate**、**lastprivate**、**reduction**、**shared** 句を使用して無効にすることができます。

1.10.1.4 FIRSTPRIVATE 句

firstprivate(*list***)**

list に指定した変数が **PRIVATE** になります。また、変数の非公開コピーは、この構文の前に存在している元のオブジェクトで初期化されます。

1.10.1.5 LASTPRIVATE 句

lastprivate(*list***)**

list に指定した変数が **PRIVATE** になります。また、**LASTPRIVATE** 句を **DO** または **for** 指令で記述している場合は、逐次実行した場合に最後に反復を実行するスレッドが元のオブジェクトを更新します。**SECTIONS** 指令では、最後に記述された **SECTION** を実行するスレッドが元のオブジェクトを更新します。

1.10.1.6 COPYIN 句

Fortran:

COPYIN(*list***)**

COPYIN 句は、**THREADPRIVATE** として宣言された変数、共通ブロック、共通ブロック内の変数だけに適用されます。並列領域で **COPYIN** は、並列領域の最初にチームのマスタースレッドのデータをスレッドの非公開の複製にコピーするように指定します。

C/C++:

copyin(*list***)**

COPYIN 句は、**THREADPRIVATE** として宣言された変数だけに適用されます。並列領域では **COPYIN** は、並列領域の最初にチームのマスタースレッド内のデータをスレッドの非公開の複製にコピーするように指定します。

1.10.1.7 **COPYPRIVATE** 句

Fortran:

COPYPRIVATE(*list***)**

チームの特定メンバーから他のメンバーに値もしくは共有オブジェクトへのポインタをブロードキャストするために、非公開変数を使用します。**COPYPRIVATE** 句は、**END SINGLE** 指令でのみ指定できます。ブロードキャストは、**single** 構文に関連付けられた構造ブロックの実行後、構文の終わりでバリアを離れたチーム内のスレッドの前で発生します。*list* に指定した変数は、**COPYPRIVATE** を指定する **SINGLE** 構文の **PRIVATE** または **FIRSTPRIVATE** 句で使用できません。

C/C++:

copyprivate(*list***)**

チームの特定メンバーから他のメンバーに値をブロードキャストするために、非公開変数を使用します。**copyprivate** 句は、**single** 指令でのみ指定できます。ブロードキャストは、**single** 構文に関連付けられた構造ブロックの実行後、構文の終わりでバリアを離れたチーム内のスレッドの前で発生します。*list* に指定した変数は、この **copyprivate** 句を指定した **single** 指令の **private** または **firstprivate** 句で使用することはできません。

1.10.1.8 **REDUCTION** 句

Fortran:

REDUCTION(*operator* | *intrinsic***:***list***)**

operator には、**+**、*****、**-**、**.AND.**、**.OR.**、**.EQV.**、**.NEQV.** のいずれかを指定します。

intrinsic には、**MAX**、**MIN**、**IAND**、**IOR**、**IEOR** のいずれかを指定します。

list の変数には、組み込み型の名前付き変数を指定する必要があります。

C/C++:

reduction(*operator***:***list***)**

operator には、**+**、*****、**-**、**&**、**^**、**|**、**&&**、**||** のいずれかを指定します。

REDUCTION 句は、縮約変数が縮約文でだけ使用されている領域で使用します。*list* の変数は、構文の中では **SHARED** に設定する必要があります。各変数の非公開の複製が、スレッドごとに **PRIVATE** であるものとして作成されます。縮約の最後に、元の値と非公開の複製の最終的な値とを結合することで、共有変数が更新されます。

REDUCTION 句と構文の詳細および制限については、OpenMP 仕様書の適切な個所を参照してください。

1.10.2 スケジュールを指定する句

SCHEDULE 句は、Fortran の **DO** ループまたは C/C++ の **for** ループでの反復をチーム内のスレッド間で分割する方法を指定します。表 1-1 は、どの指令で **SCHEDULE** 句を記述できるかを示しています。

スケジュールを指定する句を使用する場合は、重要な制限があります。完全な情報については、Fortran 仕様の 2.3.1 節、C/C++ 仕様の 2.4.1 節を参照してください。

schedule(*type* [**,***chunk*]**)**

DO または **for** ループでの反復をチーム内のスレッド間で分割する方法を指定します。*type* には、**STATIC**、**DYNAMIC**、**GUIDED**、**RUNTIME** のいずれかを指定します。**SCHEDULE** 句がない場合は、**STATIC** が使用されます。*chunk* には、整数式を指定する必要があります。

1.10.2.1 STATIC スケジュール指定

schedule(static[**,***chunk*]**)**

反復は、*chunk* で指定したサイズに分割されます。分割された部分は、スレッドの番号順でラウンドロビン形式でチーム内のスレッドに静的に割り当てられます。*chunk* を指定していない場合は、ほぼ同サイズの連続した塊に分割され、スレッドごとに 1 つずつ割り当てられます。

1.10.2.2 DYNAMIC スケジュール指定

schedule(dynamic[**,***chunk*]**)**

反復は、*chunk* で指定したサイズに分割され、待機中のスレッドに割り当てられます。各スレッドは割り当てられた反復の部分を終了すると、反復の次のセットが動的に割り当てられます。*chunk* を指定していない場合は、デフォルトで 1 に設定されます。

1.10.2.3 GUIDED スケジュール指定

schedule(guided[**,***chunk*]**)**

GUIDED を指定した場合は、*chunk* のサイズは、反復のディスパッチごとに指数関数的に減少します。*chunk* は、ディスパッチごとの最小反復回数を指定します (反復の最初のチャンクのサイズは、実装によって異なります。4-2 ページの「GUIDED: チャンクサイズの決定」を参照してください。) *chunk* を指定していない場合は、デフォルトで 2.0 に設定されます。

1.10.2.4 **RUNTIME** スケジュール指定

schedule(runtime)

スケジュール指定は実行時まで遅延されます。スケジュールの *type* および *chunk* は、**OMP_SCHEDULE** 環境変数の設定によって決定されます。デフォルトでは **SCHEDULE(STATIC)** が指定されます。

1.10.3 **NUM_THREADS** 句

NUM_THREADS 句は、**PARALLEL**、**PARALLEL SECTIONS**、**PARALLEL DO**、**PARALLEL for**、**PARALLEL WORKSHARE** 指令で使用できます。

num_threads(*scalar_integer_expression***)**

スレッドが並列領域に入ったときにチーム内で作成されるスレッドの数を指定します。*scalar_integer_expression* には、作成する必要のあるスレッドの数を指定します。指定を行うと、**OMP_SET_NUM_THREADS** ライブラリ関数の呼び出しによって決定されたスレッドの数、または、**OMP_NUM_THREADS** 環境変数の値が無効になります。動的なスレッド管理が有効な場合、指定した数がスレッドの最大数として使用されます。

num_threads は、以降の領域には適用されないことに注意してください。

1.10.4 指令での句の記述

表 1-1 は、以下の指令およびプラグマで記述できる句を示しています。

- **PARALLEL**
- **DO**
- **for**
- **SECTIONS**
- **SINGLE**
- **PARALLEL DO**
- **parallel for**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

表 1-1　句とともに記述できるプラグマ

句/プラグマ	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE[3]
IF	•				•	•	•
PRIVATE	•	•	•	•	•	•	•
SHARED	•				•	•	•
FIRSTPRIVATE	•	•	•	•	•	•	•
LASTPRIVATE		•	•		•	•	
DEFAULT	•				•	•	•
REDUCTION	•	•	•		•	•	•
COPYIN	•				•	•	•
COPYPRIVATE				•[1]			
ORDERED		•			•		
SCHEDULE		•			•		
NOWAIT		•[2]	•[2]	•[2]			
NUM_THREADS	•				•	•	•

1. Fortran のみ : **COPYPRIVATE** を **END SINGLE** 指令で指定できます。
2. Fortran では、**END DO**、**END SECTIONS**、**END SINGLE**、**END WORKSHARE** の各指令でのみ **NOWAIT** 修飾子を使用することができます。
3. **WORKSHARE** および **PARALLEL WORKSHARE** は、Fortran でだけサポートされています。

1.11 OpenMP 実行時ライブラリルーチン

OpenMP は、並列実行環境の設定および照会を実行する呼び出し可能なライブラリルーチン、汎用ロックルーチン、2 つのポータブルタイマールーチンを提供します。詳細については、OpenMP Fortran 仕様、OpenMP C/C++ 仕様を参照してください。

1.11.1 Fortran の OpenMP ルーチン

Fortran の実行時ライブラリルーチンは、外部手続きです。以下の概要では、*int_expr* はスカラー整数式、*logical_expr* はスカラー論理式を示します。

INTEGER(4) および **LOGICAL(4)** を返す **OMP_** 関数は組み込み関数ではないため、正しく宣言する必要があります。宣言しない場合は、コンパイラでは **REAL** を返すものとして処理されます。以下で説明する OpenMP Fortran 実行時ライブラリルーチンのインタフェース宣言は、Fortran の インクルードファイルである **omp_lib.h** および Fortran **MODULE omp_lib** で提供されています。これについては、Fortran OpenMP 仕様で説明されています。

これらのライブラリルーチンを参照するすべてのプログラム単位で、**INCLUDE 'omp_lib.h'** 文、**#include "omp_lib.h"** プリプロセッサ指令、**USE omp_lib** 文のいずれかを記述してください。

-Xlist を指定してコンパイルを実行すると、あらゆる型の不一致が報告されます。

整数パラメータ **omp_lock_kind** は、**OMP_*_LOCK** ルーチンでの単純ロック変数で使用される **KIND** 型のパラメータを定義します。

整数パラメータ **omp_nest_lock_kind** は、**OMP_*_NEST_LOCK** ルーチンでの入れ子可能なロック変数で使用される **KIND** 型のパラメータを定義します。

整数パラメータ **openmp_version** は、**YYYYMM** という書式のプリプロセッサマクロ **_OPENMP** として定義されています。ここで、**YYYY** および **MM** は、OpenMP Fortran API のバージョンを年と月で示したものになります。

1.11.2 C/C++ の OpenMP ルーチン

C/C++ の実行時ライブラリ関数は、外部関数です。

ヘッダー **<omp.h>** では、2 つの型、並列実行環境の設定および照会に使用する複数の関数、データアクセスの同期をとるのに使用するロック関数が定義されています。

omp_lock_t 型は、ロックが使用可能であるか、スレッドがロックを所有しているかのいずれかを示すことができるオブジェクト型です。これらのロックを、単純ロックと呼びます。

omp_nest_lock_t 型は、ロックが使用可能であるか、スレッドがロックを所有しているかのいずれかを示すことができるオブジェクト型です。これらのロックを、入れ子可能なロックと呼びます。

1.11.3 実行時スレッド管理ルーチン

詳細については、それぞれの言語の OpenMP 仕様を参照してください。

1.11.3.1 OMP_SET_NUM_THREADS ルーチン

num_threads() 句が指定していない、それ以降の並列領域で使用するスレッド数を設定します。この呼び出しは、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並列領域に対してだけ有効です。

Fortran:

```
SUBROUTINE OMP_SET_NUM_THREADS(int_expr)
```

C/C++:

```
#include <omp.h>
void omp_set_num_threads(int num_threads);
```

1.11.3.2 OMP_GET_NUM_THREADS ルーチン

チーム内で、呼び出し元の並列領域を実行しているスレッドの個数を返します。

Fortran:

```
INTEGER(4) FUNCTION OMP_GET_NUM_THREADS()
```

C/C++:

```
#include <omp.h>
int omp_get_num_threads(void);
```

1.11.3.3 OMP_GET_MAX_THREADS ルーチン

プログラムのこの部分で検出されるはずの有効な並列領域に **num_threads()** 句が指定されていない場合に、チームを構成するために必要なスレッドの最大数を返します。

Fortran:

```
INTEGER(4) FUNCTION OMP_GET_MAX_THREADS()
```

C/C++:

```
#include <omp.h>
int omp_get_max_threads(void);
```

1.11.3.4 OMP_GET_THREAD_NUM ルーチン

チーム内で、この関数の呼び出しを実行しているスレッドの個数を返します。値の範囲は、0 ～ OMP_GET_NUM_THREADS()-1 になります。0 はマスタースレッドを示します。

Fortran:

```
INTEGER(4) FUNCTION OMP_GET_THREAD_NUM()
```

C/C++:

```
#include <omp.h>
int omp_get_thread_num(void);
```

1.11.3.5 OMP_GET_NUM_PROCS ルーチン

プログラムで使用可能なプロセッサ数を返します。

Fortran:

```
INTEGER(4) FUNCTION OMP_GET_NUM_PROCS()
```

C/C++:

```
#include <omp.h>
int omp_get_num_procs(void);
```

1.11.3.6 OMP_IN_PARALLEL ルーチン

並列領域の動的な範囲内でスレッドを実行するかどうかを決定します。

Fortran:

```
LOGICAL(4) FUNCTION OMP_IN_PARALLEL()
```

並列領域の動的な範囲内から呼び出された場合は .TRUE.、そうでない場合は .FALSE. を返します。

C/C++:

```
#include <omp.h>
int omp_in_parallel(void);
```

有効な並列領域の動的な範囲内から呼び出された場合はゼロ以外の値、そうでない場合はゼロを返します。

有効な並列領域とは、**IF** 句が **TRUE** と評価された並列領域を指します。

1.11.3.7 OMP_SET_DYNAMIC ルーチン

使用可能なスレッドの数の動的調整を有効または無効にします。(デフォルトでは、動的調整が有効に設定されます。) この呼び出しは、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スレッドが検出した以降に続く並列領域に対してだけ有効です。

Fortran:

```
SUBROUTINE OMP_SET_DYNAMIC(logical_expr)
```

logical_expr が .TRUE. の場合は動的調整が有効になり、それ以外の値の場合は無効になります。

C/C++:

```
#include <omp.h>
void omp_set_dynamic(int dynamic);
```

dynamic がゼロ以外の場合は、動的調整が有効になります。それ以外の場合は無効になります。

1.11.3.8 OMP_GET_DYNAMIC ルーチン

プログラムのこの個所で動的なスレッド調整が有効になっているかどうかを判断します。

Fortran:

```
LOGICAL(4) FUNCTION OMP_GET_DYNAMIC()
```

動的調整が有効な場合は .TRUE.、そうでない場合は .FALSE. を返します。

C/C++:

```
#include <omp.h>
int omp_get_dynamic(void);
```

動的調整が有効な場合はゼロ以外の値、そうでない場合はゼロを返します。

1.11.3.9 **OMP_SET_NESTED ルーチン**

入れ子並列処理を有効または無効にします この呼び出しは、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スレッドが検出した以降に続く並列領域に対してだけ有効です。

Fortran:

```
SUBROUTINE OMP_SET_NESTED(logical_expr)
```

logical_expr が **.TRUE.** の場合は入れ子並列処理が有効になり、それ以外の値の場合は無効になります。

C/C++:

```
#include <omp.h>
void omp_set_nested(int nested);
```

nested がゼロ以外の場合は入れ子並列処理が有効になり、ゼロの場合は無効になります。

入れ子並列処理はデフォルトで無効になっています。入れ子並列処理については、第 2 章を参照してください。

1.11.3.10 **OMP_GET_NESTED ルーチン**

プログラムのこの時点で入れ子並列処理が有効になっているかどうかを判断します。

Fortran:

```
LOGICAL(4) FUNCTION OMP_GET_NESTED()
```

入れ子並列処理が有効な場合は **.TRUE.**、そうでない場合は **.FALSE.** を返します。

C/C++:

```
#include <omp.h>
int omp_get_nested(void);
```

入れ子並列処理が有効な場合はゼロ以外の値、そうでない場合はゼロを返します。

入れ子並列処理については、第 2 章を参照してください。

1.11.4 同期ロックを操作するルーチン

単純ロックと入れ子可能なロックの 2 種類のロックがサポートされています。入れ子可能なロックは、ロックを解除する前に同一スレッド内で複数回ロックできます。単純ロックは、すでにロック状態の場合はロックできません。単純ロック変数は、単純ロックルーチンにだけ渡すことができます。入れ子可能なロック変数は、入れ子可能なロックルーチンにだけ渡すことができます。

Fortran:

ロック変数 *var* にアクセスするには、後述のルーチンを使用する必要があります。このとき、**OMP_LOCK_KIND** および **OMP_NEST_LOCK_KIND** というパラメータを使用します (**omp_lib.h INCLUDE** ファイルおよび **omp_lib MODULE** で定義)。たとえば、次のように指定します。

```
INTEGER(KIND=OMP_LOCK_KIND)    :: var
INTEGER(KIND=OMP_NEST_LOCK_KIND) :: nvar
```

C/C++:

単純ロック変数には、**omp_lock_t** 型を使用する必要があります。また、この変数にアクセスするには、後述のロックルーチンを使用する必要があります。すべての単純ロック関数では、**omp_lock_t** 型へのポインタを引数として指定する必要があります。

入れ子可能なロック変数には、**omp_nest_lock_t** 型を使用する必要があります。同様に、すべての入れ子可能なロック関数では、**omp_nest_lock_t** 型へのポインタを引数として指定する必要があります。

1.11.4.1 OMP_INIT_LOCK、OMP_INIT_NEST_LOCK ルーチン

それ以降の呼び出し用にロック変数を初期化します。

Fortran:

```
SUBROUTINE OMP_INIT_LOCK(var)

SUBROUTINE OMP_INIT_NEST_LOCK(nvar)
```

C/C++:

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

void omp_init_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.2 OMP_DESTROY_LOCK、OMP_DESTROY_NEST_LOCK ルーチン

ロック変数を削除します。

Fortran:

```
SUBROUTINE OMP_DESTROY_LOCK(var)
SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar)
```

C/C++:

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.3 OMP_SET_LOCK、OMP_SET_NEST_LOCK ルーチン

実行中のスレッドを、指定したロックが使用可能になるまで待機させます。指定したロックが使用可能になると、スレッドはそのロックの所有者になります。

Fortran:

```
SUBROUTINE OMP_SET_LOCK(var)
SUBROUTINE OMP_SET_NEST_LOCK(nvar)
```

C/C++:

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.4 OMP_UNSET_LOCK、OMP_UNSET_NEST_LOCK ルーチン

実行中のスレッドから、ロックの所有権を解放します。スレッドがそのロックを所有していない場合の動作は未定義です。

Fortran:

```
SUBROUTINE OMP_UNSET_LOCK(var)
SUBROUTINE OMP_UNSET_NEST_LOCK(nvar)
```

C/C++:

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.5 OMP_TEST_LOCK、OMP_TEST_NEST_LOCK ルーチン

OMP_TEST_LOCK ロック変数に関連付けられたロックを設定します。この呼び出しにより、スレッドの実行がブロックされることはありません。

OMP_TEST_NEST_LOCK ロックが正常に設定された場合は、最新の入れ子の数を返します。それ以外の場合は 0 を返します。この呼び出しにより、スレッドの実行がブロックされることはありません。

Fortran:

```
LOGICAL(4) FUNCTION OMP_TEST_LOCK(var)
```

ロックが設定された場合は **.TRUE.**、そうでない場合は **.FALSE.** を返します。

```
INTEGER(4) FUNCTION OMP_TEST_NEST_LOCK(nvar)
```

ロックが正常に設定された場合は、最新の入れ子の数を返します。それ以外の場合は 0 を返します。

C/C++:

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock);
```

ロックが正常に設定された場合は、ゼロ以外の値を返します。それ以外の場合は 0 を返します。

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

ロックが正常に設定された場合は、ロックの最新の入れ子の数を返します。それ以外の場合は 0 を返します。

1.11.5 タイミングルーチン

2 つの関数でポータブル時計時間タイマーをサポートしています。

1.11.5.1 OMP_GET_WTIME ルーチン

過去のある時点から経過した時計時間を秒数で返します。

Fortran:

```
REAL(8) FUNCTION OMP_GET_WTIME()
```

C/C++:

```
#include <omp.h>
double omp_get_wtime(void);
```

1.11.5.2 OMP_GET_WTICK ルーチン

連続するクロック刻みの間隔の秒数を返します。

Fortran:

```
REAL(8) FUNCTION OMP_GET_WTICK()
```

C/C++:

```
#include <omp.h>
double omp_get_wtick(void);
```

第2章

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。

2.1 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、自身を含め他のスレッドとチームを構成します (他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後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。

2.2 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。

2.2.1 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()` 関数を呼び出すことで有効または無効に設定できます (1-26 ページの 1.11.3.9 節「`OMP_SET_NESTED` ルーチン」)。

入れ子並列処理が有効になっている場合に、入れ子になった並列領域を実行する複数のスレッドのチームの例を次に示します。

コード例 2-1　　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、実行すると、次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
```

2.2.2 SUNW_MP_MAX_POOL_THREADS

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。**SUNW_MP_MAX_POOL_THREADS** 環境変数を設定することで、プールに保存しておけるスレッドの最大数を制限できます。この変数のデフォルト値は 1023 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。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。

次の例では、プールに充分な数のスレッドがない場合に並列領域で使用されるスレッドが少なくなるケースを挙げています。コードそのものは前述の例と同じです。同時にすべての並列領域が有効になるために必要なスレッドの数は 8 です。このとき、プールには最小でも 7 つのアイドル状態のスレッドが必要です。ここで **SUNW_MP_MAX_POOL_THREADS** 変数を 5 に設定すると、最も内側の入れ子にある 4 つの並列領域のうち、2 つは必要な数のスレッドを取得できなくなります。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
% setenv SUNW_MP_MAX_POOL_THREADS 5
% 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
```

2.2.3 SUNW_MP_MAX_NESTED_LEVELS

SUNW_MP_MAX_NESTED_LEVELS 環境変数は、複数のスレッドを必要とする入れ子になった並列領域の最大の深さを指定します。

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 句が 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。

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。
SUNW_MP_MAX_NESTED_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 SUNW_MP_MAX_NESTED_LEVELS 4
% a.out |sort +2n
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 SUNW_MP_MAX_NESTED_LEVELS 2
% a.out |sort +2n
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
```

2.3 入れ子並列領域での OpenMP ライブラリ関数の使用

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並列領域に対してだけ有効です。このスレッドが後にこの並列領域より外側の入れ子で検出した並列領域、およびその他のスレッドが検出した並列領域には作用しません。

「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。チームが作成された場合は、スレーブスレッドはマスタースレッドが持つ値を継承します。

コード例 2-2　並列領域中の OpenMP 関数呼び出し

```
#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);       /* A 行 */
        else
            omp_set_num_threads(6);       /* B 行 */

        /* 次の宣言が出力される。
         *
         * 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
            {
                /* 次の宣言が出力される。
                 *
                 * Inner: 4
                 * Inner: 6
                 */
                printf("Inner: %d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* C 行 */
        }
```

```
        /* 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
         * 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.
count me.
```

2.4 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、それぞれ 2 つの並列処理に分割できる処理が、入れ子によって 2 段階になっているとします。さらに、使用できる CPU が 4 つあり、その 4 つをすべて使用してプログラムの実行速度を速めたい場合、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。両方の段階で処理を並列化する必要があります。

- 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。システムに対する過剰な処理要求を防ぐために、**SUNW_MP_MAX_POOL_THREADS** および **SUNW_MP_MAX_NESTED_LEVELS** 環境変数を適切に設定し、使用するスレッドの数を制限します。

- 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 つの並列処理となるプログラムがあったとします。外側にある 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3章

Fortran における自動スコープ宣言

OpenMP の並列領域で変数のスコープ属性を指定することを、**スコープ宣言**といいます。一般に、変数が **SHARED** とスコープ宣言された場合、すべてのスレッドは同じ変数を使用します。変数が **PRIVATE** スコープ宣言された場合は、各スレッドはそれぞれ専用の変数のコピーを使用します。OpenMP には、豊富なデータ環境があります。**SHARED** や **PRIVATE** に加えて、変数のスコープは、**FIRSTPRIVATE**、**LASTPRIVATE**、**REDUCTION**、あるいは **THREADPRIVATE** とも宣言できます。

OpenMP では、並列領域で使用する変数の 1 つ 1 つについて、そのスコープを宣言する必要があります。これは単調でエラーを起こしやすい工程で、多くの人が、OpenMP を使ってプログラムを並列化する作業で最も手間のかかる部分と認識しています。

Sun Studio 9 リリースの Fortran 95 コンパイラ **f95** には、自動的なスコープ宣言機能があります。コンパイラが並列領域の実行および同期パターンを解析して、一群のスコープ宣言規則に基づいて、変数のスコープを決定します。

3.1 自動スコープ宣言用データスコープ句

自動スコープ宣言用データスコープ句は、Fortran OpenMP 仕様に対するサンの拡張です。この後の 2 つある句のいずれかを利用することによって、変数のスコープを自動的に宣言するように指定できます。

3.1.1 __AUTO 句

__AUTO (*list-of-variables***)**

並列領域内のリスト中の指定された変数のスコープをコンパイラが決定します。(**AUTO** の前の下線は 2 つであることに注意してください。)

__AUTO スコープ句は、**PARALLEL**、**PARALLEL DO**、**PARALLEL SECTIONS**、あるいは **PARALLEL WORKSHARE** 指令で使用できます。

__AUTO 句に変数を指定した場合、他のデータスコープ句でその変数を指定することはできません。

3.1.2 DEFAULT(__AUTO) 句

この並列領域におけるデフォルトのスコープ宣言を **__AUTO** に設定します。

DEFAULT(__AUTO) スコープ句は、**PARALLEL**、**PARALLEL DO**、**PARALLEL SECTIONS**、あるいは **PARALLEL WORKSHARE** 指令で使用できます。

3.2 スコープ宣言規則

自動スコープ宣言では、コンパイラは、並列領域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** 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。

3.2.1 スカラー変数に関するスコープ宣言規則

- S1: 並列領域内で使用される変数が、その領域を実行するチーム内でのスレッドに関する「データ競合[1]」状態になることがない場合、その変数のスコープは **SHARED** と宣言されます。
- S2: 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つね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、並列領域の後、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。
- S3: 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。

1. 「データ競合」とは、2 つのスレッドが同じ共有変数にアクセスする可能性があり、同時にそのうちの少なくとも一方がその変数を変更する可能性がある状態です。データ競合状態を排除するには、クリティカル領域内にアクセスを置くか、それらスレッドの同期をとります。

3.2.2 配列に関するスコープ宣言規則

- A1: 並列領域内で使用される配列が、その領域を実行するチーム内でのスレッドに関するデータ競合状態から自由の場合、その配列のスコープは **SHARED** と宣言されます。

3.3 自動スコープ宣言に関する一般的な注意事項

__AUTO(*list-of-variables***)** または **DEFAULT(__AUTO)** を使って次の変数のスコープを自動的に宣言するよう指定された場合、コンパイラは、OpenMP 仕様の暗黙のスコープ宣言規則に従って変数のスコープを宣言します。

- **THREADPRIVATE** 変数
- Cray ポインタの指示先
- 領域または、領域に結合されている並列 DO ループの字句範囲内の逐次ループでだけ使用されるループ反復変数
- 暗黙の **DO** または **FORALL** インデックス
- 領域に結合されているワークシェアリング構文でのみ使用され、かつ、そうした構文のどれについても、データスコープ属性句で指定されている変数

暗黙のスコープを持たない変数を自動スコープ宣言する場合、コンパイラは上記の順序で規則と変数の使われ方を比較検査します。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。規則に一致しない場合、コンパイラは次の規則を試します。一致する規則が見つからなかった場合、その変数に対する自動スコープ宣言は行われません。

変数の自動スコープ宣言に失敗した場合、その変数のスコープは **SHARED** と宣言され、結合する並列領域は、**IF (.FALSE.)** 句が指定されているかのように、直列化されます。

自動スコープ宣言が失敗する理由は 2 つあります。1 つは、変数の使われ方が上記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は配列添え字、メモリー別名、ユーザー実装の同期などです。(3-8 ページの 3.5 節「現在の実装の既知の制限事項」を参照してください。)

3.4 自動スコープ宣言結果の確認

「コンパイラのコメント」を利用して、詳細な自動スコープ宣言結果を調べたり、自動スコープ宣言が失敗したために直列化された並列領域がないか確認したりできます。

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、コード例 3-2 で示しているように **er_src** を使って表示できます。(**er_src** コマンドは、Sun 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『プログラムのパフォーマンス解析』を参照してください。

-vpara 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。自動スコープ宣言の失敗があると、コード例 3-1 で示しているように警告メッセージが出力されます。

コード例 3-1　　-vpara を使ったコンパイル

```
>cat t.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 t.f
"t.f", line 3: Warning: parallel region is serialized
       because the autoscoping of following variables failed
       - x
```

コード例 3-2　　コンパイラのコメントの見方

```
>cat t.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 -g -c t.f
>er_src t.o
Source file: ./t.f
Object file: ./t.o
Load Object: ./t.o

     1.       INTEGER X(100), Y(100), I, T
     2.

Private variables in OpenMP construct below: t,i
Shared variables in OpenMP construct below: y,x
Variables autoscoped as PRIVATE in OpenMP construct below:
       i, t
Variables autoscoped as SHARED in OpenMP construct below:
       y, x
     3. C$OMP PARALLEL DO DEFAULT(__AUTO)

Loop below parallelized by explicit user directive
     4.       DO I=1, 100

Loop below scheduled with steady-state cycle count = 3
Loop below unrolled 2 times
Loop below has 1 loads, 1 stores, 0 prefetches, 0 FPadds, 0 FPmuls,
and 0 FPdivs per iteration
     5.          T = Y(I)
     6.          X(I) = T*T
     7.       END DO
     8. C$OMP END PARALLEL DO
     9.
    10.       END
```

次に、自動スコープ宣言の仕組みを示すもっと複雑な例を紹介します。

コード例 3-3　　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
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。

コンパイラが上記の規則に従って、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。

上記の並列領域では、**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。コンパイラは以下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後に書き込まれます。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一方で、後の 2 つの両方が **CRITICAL** 構文内にあるために、それらが同時に発生しないようにします。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。データ教がないため、**Y()** のスコープは、規則 A1 に従って **SHARED** と宣言されます。

3.5 現在の実装の既知の制限事項

Sun Studio 9 Fortran 95 コンパイラの自動スコープ宣言に関する既知の制限事項は次のとおりです。

- 解析では、OpenMP 指令のみ認識、使用されます。OpenMP API 関数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。
- 解析では、**`BARRIER`** や **`MASTER`** などの OpenMP 同期指令で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。
- コンパイルで **`-xopenmp=noopt`** が使用された場合、自動スコープ宣言はサポートされません。

第4章

実装 - 定義済みの動作

この章では、OpenMP 2.0 Fortran および OpenMP 2.0 C/C++ の各仕様に固有の問題について説明します。最新のコンパイラリリースの最新の情報については、C、C++、Fortran の readme ファイルを参照してください。

- **スケジュール指定**

  OMP_SCHEDULE 環境変数または **SCHEDULE** 句を明示的に設定していない場合は、**static** スケジュール指定がデフォルトで使用されます。

- **スレッド数**

  num_threads() 句、**omp_set_num_threads**() 関数の呼び出し、**OMP_NUM_THREADS** 環境変数の定義が明示的に行われていない場合は、チームのスレッド数はデフォルトで 1 に設定されます。

- **スレッドの動的調整**

  動的調整が有効になっていると、チームに含まれるスレッドの数は次の中で最小の値に調整されます。

  ユーザーが要求したスレッドの数

  プール内で利用できるスレッドの数に 1 を足した数

  使用できるプロセッサの数

  動的調整が無効になっている場合は、チームに含まれるスレッドの数が次の中での最小の値に調整されます。

  ユーザーが要求したスレッドの数

  プール内で利用できるスレッドの数に 1 を足した数

  SUNW_MP_WARN が **TRUE** に設定されているか、sunw_mp_register_warn() を呼び出すことでコールバック関数が登録されている場合で、かつ、提供されたスレッドの数がユーザーが要求した数に満たない場合は、警告メッセージが出力されます。

システム資源の不足など、特別な状況では、提供されるスレッドの数は前述に上げた数より少なくなることがあります。このような状況で、**SUNW_MP_WARN** が **TRUE** に設定されているか、sunw_mp_register_warn() を呼び出すことでコールバック関数が登録されている場合で、かつ、動的調整が有効になっていると、警告メッセージが出力されます。

スレッドのプールと入れ子並列処理の実行モデルの詳細については第 2 章を参照してください。

- **入れ子並列処理**

  入れ子並列処理がサポートされています。入れ子になった並列領域は複数のスレッドで実行できます。入れ子並列処理はデフォルトで無効になっています。有効にするには、OMP_NESTED 環境変数を設定するか、omp_set_nested() 関数を呼び出します。第 2 章を参照してください。

- **ATOMIC 指令**

  実装上は、すべての **ATOMIC** 指令およびプラグマは、クリティカル領域内に文を含める形に置き換えられます。

- **GUIDED: チャンクサイズの決定**

  chunksize が指定されていない場合の **SCHEDULE(GUIDED)** のデフォルトのチャンクサイズは 1 です。OpenMP 実行時ライブラリは、**GUIDED** スケジューリングされたループのチャンクサイズを次の式を使って計算します。

  チャンクサイズ = *unassigned_iterations* / (*weight* × *num_threads*) ここで、各要素の意味は以下のとおりです。

  unassigned_iterations とは、どのスレッドにも割り当てられていないループの反復回数を表します。

  weight (「重み係数」) は、ユーザーが **SUNW_MP_GUIDED_WEIGHT** 環境変数を使って実行時に設定できる浮動小数点定数です (5-5 ページの 5.3 節「OpenMP 環境変数」 を参照してください)。現在のデフォルトでは、ユーザーの指定がない場合には *weight* として 2.0 が、*num_threads* (「スレッド数」) としてループの実行に使用されるスレッド数が指定されます。

  weight に指定された値は、ループ内のスレッドに割り当てられる反復の初期のチャンクとその後のチャンクのサイズに影響し、また、負荷分散に直接影響します。これまでの実験では、デフォルトである 2.0 でほとんどの場合問題なく動作することが確認されています。ただし、アプリケーションによっては異なる値にした方が良いこともあります。

- **明示的にスレッド化されたプログラム**

  POSIX または Solaris のスレッドを使用するプログラムでは、OpenMP 指令を含めたり、Open MP の指令が含まれるルーチンを呼び出したりできます。

- **実行時の警告**

- **SUNW_MP_WARN** 環境変数 (5-5 ページの 5.3 節「OpenMP 環境変数」を参照) を設定すると、OpenMP マルチタスクライブラリによる実行時の有効性の確認機能が有効になります。

  たとえば、次に挙げるコードでは、スレッドが異なるバリアで待ち状態に入り、ループが終了しません。終了させるには、端末で Control キーを押しながら C キーを押します。

```
% cat bad1.c

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

int
main(void)
{
   omp_set_dynamic(0);
   omp_set_num_threads(4);

   #pragma omp parallel
   {
       int i = omp_get_thread_num();

       if (i % 2) {
           printf("At barrier 1.\n");
           #pragma omp barrier
       }
   }
   return 0;
}
% cc -xopenmp -xO3 bad1.c
% ./a.out          run the program
At barrier 1.
At barrier 1.
                   program hung in endless loop
Control-C  to terminate execution
```

しかし、実行前に **SUNW_MP_WARN** を設定しておけば、実行時ライブラリによってこの問題を事前に検出することができます。

```
% setenv SUNW_MP_WARN TRUE
% ./a.out
At barrier 1.
At barrier 1.
WARNING (libmtsk): Threads at barrier from different directives.
    Thread at barrier from bad1.c:11.
    Thread at barrier from bad1.c:17.
    Possible Reasons:
    Worksharing constructs not encountered by all threads in the team in the
       same order.
    Incorrect placement of barrier directives.
```

- C コンパイラでも、エラーが検出されたときのコールバック関数を登録するための関数が提供されています。エラーが起きると、登録されたコールバック関数が呼び出され、エラーメッセージ文字列へのポインタが引数として渡されます。

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

  この関数のプロトタイプを使用する場合は、次の行を追加します。

  ```
  #include <sunw_mp_misc.h>
  ```

次に例を示します。

```
% cat bad2.c
#include <omp.h>
#include <sunw_mp_misc.h>
#include <stdio.h>

void handle_warn(void *msg)
{
    printf("handle_warn: %s\n", (char *)msg);
}

void set(int i)
{
    static int k;
#pragma omp critical
    {
        k++;
    }
#pragma omp barrier
}

int main(void)
{
  int i, rc;
  omp_set_dynamic(0);
  omp_set_num_threads(4);
  if (sunw_mp_register_warn(handle_warn) != 0) {
      printf ("Installing callback failed\n");
  }
#pragma omp parallel for
  for (i = 0; i < 20; i++) {
      set(i);
  }
  return 0;
}

% cc -xopenmp -xO3 bad2.c
% a.out
handle_warn: WARNING (libmtsk): at bad2.c:21 Barrier is not
permitted in dynamic extent of for / DO.
```

handle_warn() は、OpenMP 実行時ライブラリによってエラーが検出された場合に、コールバックハンドラ関数としてインストールされます。この例でのハンドラはライブラリから渡されたエラーメッセージを表示するだけですが、特定のエラーを検出するためにも使用できます。

第5章

OpenMP 用のコンパイル

この章では、OpenMP API を使用するプログラムをコンパイルする方法を説明します。

並列処理プログラムをマルチスレッド環境で実行するには、**OMP_NUM_THREADS** 環境変数をプログラム実行前に設定する必要があります。これにより、プログラムで作成される最大スレッド数を実行時システムに設定します。デフォルトは 1 です。通常は、**OMP_NUM_THREADS** を対象プラットフォームで使用可能なプロセッサ数かそれ以下に設定します。**OMP_NUM_THREADS** で指定したスレッド数を使う場合には、**OMP_DYNAMIC** を **FALSE** に設定します。

コンパイラの readme ファイルで、OpenMP の実装に関する制限および既知の問題を説明しています。readme ファイルは、`-xhelp=readme` フラグを指定してコンパイラを起動するか、HTML ブラウザで以下のマニュアル索引を指定することで表示できます。

```
file:/opt/SUNWspro/docs/ja/index.html
```

5.1 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。このフラグには、キーワードの引数を 1 つ指定することができます (**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、以下のキーワードを指定することができます。

-xopenmp=parallel	OpenMP プラグマが認識されるように設定します。 -xopenmp=parallel 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 -xO3 に上げ、警告を出力します。
-xopenmp=noopt	OpenMP プラグマが認識されるように設定します。最適化のレベルが -xO3 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 -xO3 より下げると、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、並列化されますが、最適化は行われません。 (このオプションは、cc と f95 でのみ使用できます。CC でこのオプションを指定すると、警告が出力され、OpneMP の並列化は行われません。)
-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 ライブラリ関数を呼び出しても 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。たとえば、次のように指定します。 **% cc omp_ignore.c -lompstubs** libompstubs.a と OpenMP 実行時ライブラリ libmtsk.so の両方のリンクはサポートされていません。両方をリンクすると、予期しない動作を引き起こすことがあります。
-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** を指定しないと、**-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** をキーワードなしで指定した場合は、コンパイラでは **-xopenmp=parallel** が指定されます。
- コマンド行で、**-xopenmp** を **-xparallel** または **-xexplicitpar** と共に指定しないでください。
- **-xopenmp=** に **parallel** または **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 **200203L**、Fortran 95 では **200011**) 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。
- 動的 (**.so**) ライブラリの構築でコンパイルに **-xopenmp** を指定する場合は、実行可能ファイルのリンクでも **-xopenmp** を指定する必要があります。実行可能ファイルの作成に使用するコンパイラのバージョンは、**-xopenmp** を付けて動的ライブラリを構築するのに使用したコンパイラのバージョンと同じか新しいものである必要があります。実行可能ファイルとライブラリの作成で **-xopenmp** を使用した場合、コンパイラのバージョンが異なると、予期しない動作になることがあります。

5.2 Fortran 95 OpenMP の妥当性検査

f95 コンパイラのプログラム全体のチェック機能を使用して、Fortran 95 プログラムの OpenMP 指令の手続き間の妥当性検査を静的に実行することができます。OpenMP のチェックは、**-XlistMP** フラグを指定してコンパイルを行うことによって有効になります。(**-XlistMP** を指定した場合の診断メッセージは、ソースファイル名に **.lst** という拡張子の付いた名前の別ファイルの形で出力されます)。コンパイラは、以下の違反と並列化の阻害要因を診断します。

- 並列化指令の仕様の違反 (不正な入れ子を含む)
- 手続き間の依存性解析により検出される、データの使用が原因である並列化の阻害要因
- 手続き間のポインタ解析により検出される並列化の阻害要因

たとえば、ord.f というソースファイルを -XlistMP を指定してコンパイルすると、ord.lst という診断ファイルが生成されます。

```
FILE  "ord.f"
     1  !$OMP PARALLEL
     2  !$OMP DO ORDERED
     3                  do i=1,100
     4                          call work(i)
     5                  end do
     6  !$OMP END DO
     7  !$OMP END PARALLEL
     8
     9  !$OMP PARALLEL
    10  !$OMP DO
    11                  do i=1,100
    12                          call work(i)
    13                  end do
    14  !$OMP END DO
    15  !$OMP END PARALLEL
    16                  end
    17                  subroutine work(k)
    18  !$OMP ORDERED
         ^
**** ERR-OMP:  It is illegal for an ORDERED directive to bind to a
directive (ord.f, line 10, column 2) that does not have the
ORDERED clause specified.
    19                  write(*,*) k
    20  !$OMP END ORDERED
    21                  return
    22                  end
```

この例では、サブルーチン **WORK** 内の **ORDERED** 指令は、**ORDERED** 句がないため、2 番目の **DO** 指令を参照しているという診断が出力されています。

5.3 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 4 つ定義されています。これらの環境変数を下表に示します。

表 5-1　OpenMP 環境変数

環境変数	機能
OMP_SCHEDULE	スケジュール型が **RUNTIME** として指定された **DO**、**PARALLEL DO**、**parallel for**、**for** の指令またはプラグマのスケジュール型を設定します。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 または **PARALLEL**	並列領域の実行中に使うスレッドの数を設定します。この数は **NUM_THREADS** 句または **NUM_THREADS** への呼び出しによって上書きされます。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。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** を設定するのと同じ効果が得られます。ただし、それらが共に異なる値に設定されると、実行時ライブラリはエラーメッセージを発行します。 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	入れ子並列処理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`

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。これらの環境変数を下表に示します。

表 5-2　多重処理に関する環境変数

環境変数	機能
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。**TRUE** に設定した場合は、実行時ライブラリの警告メッセージが stderr に出力されます。**FALSE** に設定した場合は、警告メッセージが無効になります。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。4-2 ページの「実行時の警告」を参照してください。 例： **setenv SUNW_MP_WARN TRUE**
SUNW_MP_THR_IDLE	プログラムの並列部分を実行する各ヘルパースレッドのタスク終了時点の状態を制御します。**SPIN**、**SLEEP** *n***s**、または **SLEEP** *n***ms** のいずれかに設定できます。デフォルトは **SLEEP** です。この場合、スレッドは並列タスクの完了後、新しい並列タスクが到着するまでスリープ状態になります。 **SLEEP** *time* を指定した場合は、並列タスクの完了後にヘルパースレッドがスピンを継続する時間を指定します。スレッドのスピン中にそのスレッド用の新しいタスクが到着した場合は、スレッドは新しいタスクをすぐに実行します。それ以外の場合は、スレッドはスリープし、新しいタスクの到着時に動作を再開します。*time* は、秒数（*n***s**）または（*n*）またはミリ秒（*n***ms**）で指定できます。 引数なしで **SLEEP** を指定すると、スレッドは並列タスクの完了直後にスリープします。**SLEEP**、**SLEEP (0)**、**SLEEP (0s)**、**SLEEP (0ms)** はすべて同義です。 例： **setenv SUNW_MP_THR_IDLE SLEEP(50ms)**
SUNW_MP_PROCBIND	**SUNW_MP_PROCBIND** 環境変数を使用して、OpenMP プログラムの軽量プロセス (LWP) を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が向上することがありますが、同じプロセッサに複数の LWP が結合されると、パフォーマンス低下します。詳細は、5-7 ページの 5.4 節「プロセッサ結合」を参照してください。

表 5-2　多重処理に関する環境変数 (続き)

環境変数	機能
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。指定がない場合のデフォルト値は 1023 です。詳細は、2-2 ページの 2.2 節「入れ子並列処理の制御」を参照してください。
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** 句が 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。指定がない場合のデフォルト値は 4 です。詳細は、2-2 ページの 2.2 節「入れ子並列処理の制御」を参照してください。
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。 例： **setenv STACKSIZE 8192** スレッドのスタックサイズを 8M バイトに設定します。
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。

5.4　プロセッサ結合

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残る、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。

デフォルトでは、軽量プロセス (LWP) はプロセッサに結合されません。プロセッサに LWP を割り当てるスケジュール指定は Solaris に一任されます。OpenMP ライブラリ内のマルチタスクルーチンである `libmtsk` は、必ず 1 対 1 のスレッドモデル、すなわち、各スレッドが 1 つの LWP に対応するスレッドモデルを使用します。

SUNW_MP_PROCBIND 環境変数が示す値は、LWP の結合先の論理的なプロセッサ識別子 (ID) を表します。論理プロセッサ ID は 0 から始まる連続する整数で、実際のプロセッサ ID と同じであっても、そうでなくてもかまいません。*n* 個のプロセッサがオンラインで使用で可能な場合、その仮想プロセッサ ID は、`psrinfo`(1M) に記載されている順に 0、1、...、*n*-1 のようになります。

論理プロセッサ ID と実際のプロセッサ ID の対応は、システムによって異なります。大部分のシステムでは、実際のプロセッサ ID は連続していますが、システムボードを取り外すと、その範囲に穴が生じることがあります。一部システムでは、ID は 4 つで 1 つのグループ単位で、次の単位と間に 32 個の隙間があります。つまり、0、1、2、3、32、33、34、35 というようにプロセッサに番号が振られます。

`libmtsk` が作成するスレッド数は、環境変数かユーザープログラム内の API 呼び出し、またはその両方によって決まります。以下で説明しているように、**SUNW_MP_PROCBIND** では、一群の論理プロセッサを指定します。LWP は、その論理プロセッサに循環的に結合されます。LWP 数がプロセッサ数より少ない場合、一部のプロセッサには LWP は結合されません。LWP 数がプロセッサ数より多い場合、一部のプロセッサには複数の LWP が結合されます。

SUNW_MP_PROCBIND には、次のいずれかの値を指定できます。

- 文字列 **TRUE** か **FALSE** (小文字も可)。次に例を示します。
  ```
  % 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 に指定された値が **FALSE** の場合、プロセッサ結合は行われません。デフォルトは、この動作です。

SUNW_MP_PROCBIND に指定された値が **TRUE** の場合、整数 0 であるかのようにみなされます。

SUNW_MP_PROCBIND に指定された値が非負整数の場合、その整数は、LWP の結合先の開始論理プロセッサ ID を示します。LWP は、その論理プロセッサ ID から始めて、ラウンドロビン式にプロセッサに結合され、論理プロセッサ ID *n*-1 の後は ID 0 に戻ります。

SUNW_MP_PROCBIND に指定された値が、2 つ以上の非負整数のリストの場合、LWP はその論理プロセッサ ID にラウンドロビン式に結合されます。リストにない ID は使用されません。

SUNW_MP_PROCBIND に指定された値が、ハイフン 1 つ (**-**) で区切られた 2 つの非負整数の場合、LWP は、最初の論理プロセッサ ID から 2 つ目の論理プロセッサ ID の範囲のプロセッサにラウンドロビン式で結合されます。範囲にない ID は使用されません。

SUNW_MP_PROCBIND に指定された値が上記 3 つのどの形式にも当てはまらないか、不正な論理プロセッサが指定された場合、**SUNW_MP_PROCBIND** 環境変数は無視され、LWP はプロセッサに結合されません。警告が有効な場合、警告メッセージが表示されます。

5.5 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スレッド用のメインメモリー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` 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。ヘルパースレッドスタックのサイズは、**STACKSIZE** 環境変数で設定されます。

```
demo% setenv STACKSIZE 16384    <- スレッドのスタックサイズを 16 Mb に設定 (C シェル)

demo% STACKSIZE=16384           <- 同上 (Bourne/Korn シェル)
demo% export 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、隣接するスレッドで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** フラグを指定して 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。

第6章

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。

6.1 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。

6.1.1 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句と、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 6-1 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$PAR DOSERIAL`	完全に相当する句はありません。以下で代用することができます。
	`!$omp master`
	loop
	`!$omp end master`

表 6-1　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$PAR DOSERIAL*	完全に相当する句はありません。以下で代用することができます。 !$omp master *loopnest* !$omp end master
C$PAR TASKCOMMON *block[,...]*	!$omp threadprivate (/block/*[,...]*)

DOALL 指令では、以下の修飾句を指定することができます。

表 6-2　DOALL 修飾句とそれに相当する OpenMP の句

Sun の DOALL 句	OpenMP の PARALLEL DO に相当する句
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 6-3 を参照)

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

表 6-3　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*) 句
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。
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) デフォルトの *m* の値は 1 です。

6.1.1.1 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** 句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)** 句を **PARALLEL DO** 指令で使用している場合を除き、デフォルトのデータスコープは **SHARED** です。**DEFAULT(NONE)** 句を使用すると、コンパイラで変数の有効範囲が明示的に設定されません。Fortran での自動スコープに関しては第 3 章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。

6.1.2 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** の修飾句も異なります。

表 6-4 Cray 形式の DOALL 修飾句とそれに相当する Open MP の句

Cray の DOALL 句	OpenMP の PARALLEL DO に相当する句
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** 句か **__AUTO** 句と共に指定します。
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。
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。

6.1.2.1 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。

6.2 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、以下のとおりです。

表 6-5 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。以下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。以下で代用することができます。 `#pragma omp master` *loopnest*

taskloop プラグマでは、以下の句を指定できます。

表 6-6 taskloop の句とそれに相当する OpenMP の句

taskloop の句	OpenMP の parallel for に相当する句
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。

表 6-6　taskloop の句とそれに相当する OpenMP の句 (続き)

taskloop の句	OpenMP の parallel for に相当する句
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 6-7 を参照)

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

表 6-7　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*) 句
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。

6.2.1　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)** 句を使用すると、コンパイラで変数の有効範囲が明示的に設定されません。
- **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。

第7章

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。

さらに詳しい内容は、Rajat Garg および Ilya Sharapov 共著の『Techniques for Optimizing Applications: High Performance Computing』を参照してください。この著作は、`http://www.sun.com/books/catalog/garg.xml` から入手できます。

また、`http://developers.sun.com/prodtech/cc/` にある Sun の開発者向けポータルサイトもご覧ください。OpenMP アプリケーションのパフォーマンス解析と最適化に関する記事および事例研究が掲載されていることがあります。

7.1 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、ロックの仕様を回避してください。
  - 可能な場合は **NOWAIT** 句を使用して、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。

- 並列領域内の **SHARED** 変数がその領域を実行するスレッドによって読み取られるだけで、どのスレッドによっても書き込みが行われない場合は、変数を **SHARED** ではなく、**FIRSTPRIVATE** として宣言してください。そうすることにより、ポインタを参照解除することによって変数へのアクセスが回避され、キャッシュの衝突が回避されます。
- 使用する以上の個数のスレッドを要求することによるリソースの浪費を回避してください。**SUNW_MP_THR_IDLE** を使って、不要になったときにワークスレッドをスリープさせてみてください。5-5 ページの 5.3 節「OpenMP 環境変数」を参照してください。

■ 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。

効率の劣る構文:

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

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

次の方が良い:

```
!$OMP PARALLEL
  ....
  !$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。

効率の劣る構文 *:*

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

次の方が良い *:*

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

- **SUNW_MP_PROCBIND** を使用して、軽量プロセス (LWP) 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。5-7 ページの 5.4 節「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、他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。

7.2 「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、結果的に計算の直接化の原因になることがあります。

7.2.1 「偽りの共有」とは

UltraSPARC III 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して、キャッシュラインをメモリーの内容に戻す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。同じキャッシュライン上の別の要素にアクセスする他のプロセッサは、そのラインに「無効」のマークが付いていることを検出します。このため、そのプロセッサは、アクセスしようとする要素が変更されていなくても、メモリーからそのラインの新しいコピーをフェッチすることになります。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。

この状況を「偽りの共有」といい、OpenMP アプリケーションにおけるパフォーマンスおよびスケーラビリティ低下の大きな原因の 1 つになることがあります。

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。

7.2.2 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、偽りの共有は以下のことを行うことによって減らすことができます。

- 共有データの構造を変更し、非公開データにする。
- 問題のサイズ (反復の長さ) を大きくする。
- プロセッサに対する反復のマッピングを変更して、各プロセッサの反復 1 回あたりの作業量を増やす (チャンクサイズ)
- コンパイラの最適化機能を使用して、メモリーのロードおよびストア命令を取り除く。

7.3 オペレーティングシステム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 6800、SunFire 15K、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 では、プログラムは、仮想メモリーの領域によって異なるページサイズを使用することができます。Solaris 9 のデフォルトのページサイズは 8K バイトです。UltraSPARC™ III Cu および UltraSPARC IV の TLB エントリ数では、わずか数メガバイトのメモリーしかアクセスできないため、大量のメモリーを使用するアプリケーションの場合、このデフォルトの 8K バイトのページサイズでは、大量の TLB ミスが発生する可能性があります。UltraSPARC III Cu および UltraSPARC IV は、8 KB、64 KB、512 KB、および 4MB の 4 つのページサイズをサポートしています。MPSS により、ユーザープロセスはこの 4 つのページサイズのうちの 1 つを要求できます。大量のメモリーを使用するアプリケーションの場合、MPSS によって TLB ミス数を大幅に削減し、パフォーマンス向上につなげることができます。

索引

X

あ

い

お

か

き

く

け

し

す

た

て

と

は

ふ

へ

ま

め

わ