OpenMP API ユーザーズガイド

Sun™ Studio 11

Sun Microsystems, Inc.
www.sun.com

Part No. 819-4819-10
2005 年 11 月, Revision A

原典: *OpenMP API User's Guide : Sun Studio 11*
Part No: 819-3694-10
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

表 P-1　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。

表 P-2　コードについて

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

表 P-2　コードについて (続き)

コードの記号	意味	記法	コード例
|	「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	*machine_name*%
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 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 と互換性のある製品を指します。
- 「x64」は AMD64 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。

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

Sun Studio ソフトウェアおよびマニュアルページは、/usr/bin/ と /usr/share/man ディレクトリにはインストールされません。ソフトウェア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「ソフトウェアへのアクセス方法」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)、ksh(1)、および bash(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 のソフトウェアが Solaris プラットフォームでは /opt ディレクトリ、および Linux プラットフォームでは /opt/sun 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デフォルト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。

ソフトウェアへのアクセス方法

PATH 環境変数を変更してソフトウェア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. Solaris プラットフォームでは、出力内容から /opt/SUNWspro/bin を含むパスの文字列を検索します。Linux プラットフォームでは、出力内容から /opt/sun/sunstudio11/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 変数はソフトウェアの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ソフトウェアにアクセスする

- Solaris プラットフォームでは、次のパスを PATH 環境変数に追加します。以前に Forte Developer ソフトウェア、Sun ONE Studio ソフトウェア、または Sun Studio ソフトウェアのほか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

- Linux プラットフォームでは、次のパスを PATH 環境変数に追加します。

  /opt/sun/sunstudio10u1/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

- Solaris プラットフォームでは、次のパスを MANPATH 環境変数に追加します。
  `/opt/SUNWspro/man`
- Linux プラットフォームでは、次のパスを MANPATH 環境変数に追加します。
  `/opt/sun/sunstudio11/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を調べます。Solaris プラットフォームでは `/opt/netbeans/3.5V11` ディレクトリ、および Linux プラットフォームでは `/opt/sun/netbeans/3.5V11` ディレクトリです。
- 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 Solaris プラットフォームで、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/3.5V11` 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。Linux プラットフォームでは、たとえば IDE が含まれているディレクトリへのパスが `/foo/sunstudio11` の場合は、`/foo/netbeans/3.5V11` 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/3.5V11`) を、`SPRO_NETBEANS_HOME` 環境変数に設定する必要があります。

Solaris プラットフォームでは、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。Linux プラット

フォームでは、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*`/sunstudio11/bin` を追加する必要もあります。

`$PATH` には、*/installation_directory*`/netbeans/3.5V11/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。
  Solaris プラットフォーム: `file:/opt/SUNWspro/docs/ja/index.html`
  Linux プラットフォーム:
  `file:/opt/sun/sunstudio11/docs/ja/index.html`

  製品ソフトウェアが Solaris プラットフォームで `/opt`、Linux プラットフォームで `/opt/sun` 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`sm の Web サイトで入手できます。以下に示すマニュアルは、Solaris プラットフォームの場合のみ、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』
- Solaris プラットフォームおよび Linux プラットフォーム用のリリースノートは、`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、購入したり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: Solaris プラットフォームの `file:/opt/SUNWspro/docs/ja/index.html` のマニュアル索引
Readme	形式: HTML 場所: `http://developers.sun.com/prodtech/cc/documentation/ss11/mr/READMEs` の開発元ポータル

マニュアルの種類	アクセシブルな形式と格納場所
マニュアルページ	形式: HTML 場所: `file:/opt/SUNWspro/docs/ja/index.html` (Solaris プラットフォーム) `file:/opt/sun/sunstudio11/docs/ja/index.html` (Linux プラットフォーム) 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」メニューおよび「ヘルプ」ボタン
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。従来の `f77` のプログラムを `f95` に移行するためのガイドラインも記載されています。
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 OS 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 について

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。

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。

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 Sun 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 2.5 API 仕様の各項目を示してあります。

C、C++、Fortran 95 の OpenMP 2.5 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。

Sun Studio のコンパイラリリース、およびその OpenMP API の実装についての最新情報は、Sun の開発者向けポータルサイト (`http://developers.sun.com/sunstudio`) 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。

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() を呼び出すことで有効または無効に設定できます。

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。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。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。**IF** 句が指定され、それが True と評価された OpenMP 並列領域の場合は、その並列領域は有効であると見なされます。カウントされるのは有効な並列領域のみです。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。
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
```

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。

2.3 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);       /* 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る場合を考えます。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。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3章

変数の自動スコープ宣言

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 C、C++、および Fortran 95 コンパイラには自動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、変数のスコープを自動的に決定します。

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、OpenMP 仕様に対する Sun 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***)** Fortran 95 の指令の場合
__auto(*list-of-variables***)** C および C++ プラグマの場合

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** および **auto** の前の下線は 2 つであることに注意してください。)

__AUTO または **__auto** スコープ句は、**PARALLEL**、**PARALLEL DO**、**PARALLEL SECTIONS**、あるいは Fortran 95 の **PARALLEL WORKSHARE** 指令で使用できます。

句に変数を指定した場合、ほか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

DEFAULT(__AUTO) Fortran 95 の指令の場合
default(__auto) C および C++ プラグマの場合

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 Fortran 95 の **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** 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。

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。

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。

- 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。

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

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、コンパイラは指定された順序で変数の使われ方を規則と比較検査します。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は、その変数のスコープ判定が行われずに **SHARED** とスコープ宣言され、**IF (.FALSE.)** または **if(0)** 句が指定されているかのように、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 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。(3-9 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。)

3.3.1 Fortran 95 の自動スコープ宣言規則

Fortran では、**__AUTO** または **DEFAULT(__AUTO)** 句を使って次の種類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

3.3.2 C/C++ の自動スコープ宣言規則

C/C++ では、**__auto** または **default(__auto)** プラグマ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。

- 並列構文内で変数を宣言
- 変数の属性は **THREADPRIVATE**
- 変数の型は const-qualified
- 変数は **for** または **parallel for** プラグマの直後にある **for** ループのループ制御変数となり、ループ内でその変数を参照

C および C++ の自動スコープ宣言が適用されるのは、基本データ型の整数、浮動小数点、ポインタのみです。ユーザーが構造変数またはクラス変数の自動スコープ宣言を指定した場合、コンパイラは変数を **shared** としてスコープ宣言し、それを包含する並列領域は 1 つのスレッドとして実行されます。

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。

-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。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", 2 行目: 警告: 並列領域は単一スレッドで実行されます
       次の変数の autoscope 化に失敗したため:
       x
```

f95 では -vpara、cc では -xvpara 付きでコンパイルします (CC では、まだこのオプションが実装されていません)。

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 omp_t.o
ソースファイル:       ./omp_t.f
オブジェクトファイル: ./omp_t.o
ロードオブジェクト:   ./omp_t.o

     1.       INTEGER X(100), Y(100), I, T
        <関数: MAIN_>

    以下のソースの OpenMP 領域にタグ R1 があります
    R1 中の PRIVATE と autoscope 化された変数: t, i
    R1 中の SHARD と autoscope 化された変数: x, y
    R1 中の非公開変数: i, t
    R1 中の共有変数: y, x
     2. C$OMP PARALLEL DO DEFAULT(__AUTO)

    次のソースのループにタグ L1 があります
    次のソースのループにタグ L1 があります
    L1 は明示的なユーザー指令で並列化されました
    検出された次のループに L2 があります
    L-unknown は、定常サイクル数 = 3 でスケジュールされました
    L-unknown は 4 回展開されました
    L-unknown には、1 回の繰り返しで 0 個の load、0 個の store、2 個の prefetch、0
個の FPadd、0 個の FPmul、0 個の FPdiv があります
    L-unknown には、1 回の繰り返しで 1 個の int-load、1 個の int-store、4 個の
alu-op、1 個の mul、0 個の int-div、1 個の shift があります
     3.       DO I=1, 100
     4.          T = Y(I)
     5.          X(I) = T*T
     6.       END DO
     7. C$OMP END PARALLEL DO
     8.       END
```

次に、自動スコープ宣言の仕組みを示すより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 S1 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 S1 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

現在の Sun Studio 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 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。
- 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。
- 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、実装によって異なる OpenMP 2.5 仕様に固有の動作について説明します。最新のコンパイラリリースに関する最新情報については、Sun の開発者向けポータルサイト (`http://developers.sun.com/sunstudio`) を参照してください。

- **メモリーモデル**

  複数のスレッドから同じ変数に対する、「同期なし」のメモリーアクセスが互いに対して微小になる保証はありません。

  アクセスが微小になるかどうかは、実装に依存、およびアプリケーションに依存する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微小メモリー動作よりも大きいことがあります。変数によっては、境界割り当てされていなかったり、境界が不明となり、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複数の読み込みと格納の使用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が発生する場合もあります。

- **内部制御変数**

  OpenMP 実行時ライブラリでは、次の内部制御変数が保持されています。

  nthreads-var - 将来の並列領域に対して要求されたスレッド数が格納されます。

  dyn-var - 将来の並列領域で、スレッド数の動的調整を有効にするかどうかを制御します。

  nest-var - 将来の並列領域で、入れ子並列を有効にするかどうかを制御します。

  run-sched-var - **`RUNTIME`** スケジュール句を使用したループ領域で使用する、スケジューリング情報が格納されます。

  def-sched-var - 実装で定義された、ループ領域のデフォルトスケジュール情報が格納されます。

  実行時ライブラリでは、スレッドごとに *nthreads-var*、*dyn-var*、*nest-var* のそれぞれの独立したコピーが保持されます。一方、実行時ライブラリではすべてのスレッドに適用される *run-sched-var* および *def-sched-var* の、それぞれのコピーが 1 つ保持されます。

■ スレッド数

nthreads-var のデフォルト値は 1 です。そのため、明示的な **num_threads()** 句、**omp_set_num_threads()** ルーチンの呼び出し、または **OMP_NUM_THREADS** 環境変数の明示的な定義がない場合、1 つのチーム内のスレッドのデフォルト数は 1 になります。

omp_set_num_threads() の呼び出しでは、呼び出し側のスレッドのみの *nthreads-var* の値が変更され、呼び出し側スレッドで認識された同じレベルまたは内部入れ子レベルの並列領域に適用されます。

要求されたスレッドの数が実装でサポートできるスレッド数よりも多い場合、または値が正の整数でない場合は、**SUNW_MP_WARN** が **TRUE** に設定されているか、**sunw_mp_register_warn()** の呼び出しによってコールバック関数が登録されていると、警告メッセージが出力されます。

■ 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。

nest-var のデフォルト値は *false* です。そのため、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()** ルーチンを呼び出します。

omp_set_nested() の呼び出しでは、呼び出し側のスレッドのみの *nest-var* の値が変更され、呼び出し側スレッドで認識された同じレベルまたは内部入れ子レベルの並列領域に適用されます。

デフォルトでは、サポートされる有効な入れ子レベルの最大数は 4 です。**SUNW_MP_MAX_NESTED_LEVELS** 環境変数を設定することで、この最大数は変更できます。

■ スレッドの動的調整

dyn-var のデフォルト値は *true* です。そのため、デフォルトでは動的調整が有効に設定されます。**OMP_DYNAMIC** 環境変数を設定するか、**omp_set_dynamic()** ルーチンを呼び出すことで、動的調整を無効にできます。

omp_set_dynamic() の呼び出しでは、呼び出し側のスレッドのみの *dyn-var* の値が変更され、呼び出し側スレッドで認識された同じレベルまたは内部入れ子レベルの並列領域に適用されます。

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

一方、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

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に説明し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は、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。

■ ループスケジュール

def-sched-var のデフォルト値は **STATIC** スケジューリングです。ループ領域に別のスケジュールを指定するには、**SCHEDULE** 句を使用します。

run-sched-var のデフォルト値も **STATIC** スケジューリングです。**OMP_SCHEDULE** 環境変数を設定することで、デフォルトを変更できます。

■ **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*)

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 のスレッドを使用して明示的にスレッド化されたプログラムでは、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 実行時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 および 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、コールバック関数としてインストールされます。この例のコールバック関数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。

■ 特定の構文について

sections 構文

sections 構文内の構文ブロックは sections 領域を実行するチームのメンバーごとに分割され、実行されるスレッド数が sections の数とほぼ等しくなります。

single 構文

single 構文の構造ブロックは、先に single 領域を検出したスレッドによって実行されます。

atomic 構文

実装上は、すべての **ATOMIC** 指令およびプラグマは、**CRITICAL** 構文内に文を含める形に置き換えられます。

■ OpenMP ライブラリルーチンの結合スレッドセット

omp_set_num_threads ルーチン

明示的な並列領域の中から呼び出された場合、omp_set_num_threads 領域の結合スレッドセットは呼び出し側スレッドです。

omp_get_max_threads ルーチン

明示的な並列領域の中から呼び出された場合、omp_get_max_threads 領域の結合スレッドセットは呼び出し側スレッドです。

omp_set_dynamic ルーチン

明示的な並列領域の中から呼び出された場合、omp_set_dynamic 領域の結合スレッドセットは呼び出し側スレッドのみです。

omp_get_dynamic ルーチン

明示的な並列領域の中から呼び出された場合、omp_get_dynamic 領域の結合スレッドセットは呼び出し側スレッドのみです。

omp_set_nested ルーチン

明示的な並列領域の中から呼び出された場合、omp_set_nested 領域の結合スレッドセットは呼び出し側スレッドのみです。

omp_get_nested ルーチン

明示的な並列領域の中から呼び出された場合、omp_get_nested 領域の結合スレッドセットは呼び出し側スレッドのみです。

■ Fortran 95 固有の問題

threadprivate 指令

2 つの連続したアクティブ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。

shared 句

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。この一時ストレージへのコピーと読み出しが行われるのは、OpenMP 2.5 仕様のセクション 2.8.3.2 の条件 a、b、および c が保持される場合のみです。

インクルードファイルとモジュールファイル

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。

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。

Sun Studio のコンパイラおよび OpenMP についての最新情報は、Sun の開発者向けポータルサイト (`http://developers.sun.com/sunstudio`) を参照してください。

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、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 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 **200505L**、Fortran 95 では **200505**) 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。
- コンパイラの並列化メッセージを表示するには、C および Fortran 95 のオプション **-xvpara** を使用します。

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  !$OMP ORDERED
          ^
**** ERR-OMP:ORDERED 節が指定されていない DO 指令 (ord.f、10 行目、
2 列目) に ORDERED 指令を結合することは不当です。
ORDERED clause specified.
    19                  write(*,*) k
    20  !$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**、**for**、**parallel 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オーバーライドできます。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。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。4-4 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 プログラムのスレッド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スレッドが結合されると、パフォーマンス低下します。詳細は、5-8 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。 **STACKSIZE** 環境変数には、接尾辞がそれぞれバイト、キロバイト、メガバイト、ギガバイトを表す **B**、**K**、**M**、または **G** の数値も指定できます。デフォルトはキロバイトです。
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

プロセッサ結合では、プログラマはプログラムの実行全体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る必要があることを、オペレーティングシステム (Solaris) 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 (Solaris) 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。たとえば、UltraSPARC IV 物理プロセッサにはそれぞれ 2 つのコアがあります。Solaris OS から見ると、この各コアはスレッドの実行をスケジューリング可能な「仮想」プロセッサです。

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)。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"**

上記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。

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。

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。システムで使用可能な仮想プロセッサの数が *n* 個の場合、その論理 ID は、psrinfo(1M) で表示される順に 0、1、...、*n*-1 のようになります。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
```

SUNW_MP_PROCBIND に指定された値の解釈

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 のプロセッサから始めてラウンドロビン式に仮想プロセッサに結合され、論理 ID *n*-1 のプロセッサのあとは ID 0 のプロセッサに戻ります。**SUNW_MP_PROCBIND** に指定された値が 2 つ以上の非負整数のリストになっている場合、スレッドはラウンドロビン式に指定された ID の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、ハイフン 1 つ (-) 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 に指定された値が上記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。

microtasking 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** 句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。

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` 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。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。

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

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`

`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。
`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。

- 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。

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、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。5-8 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。
- 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 **IF** 句を使用し、ある程度のパフォーマンス向上を期待できる場合のみ、ループを並列に実行することを指定します。
- 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、第 2 章を参照してください。

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、結果的に計算の直列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 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。同じキャッシュライン上の別の要素にアクセスするほかのプロセッサは、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、メモリーなどの場所からそのラインの最新の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。

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。

- 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、メモリーのロード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。

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 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参照)
- 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

指令での句の記述

次の表は、句と指令およびプラグマとの関連を示しています。

表 A-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	•				•	•	•
__AUTO	•				•	•	•

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。

索引