Oracle® Developer Studio 12.5: OpenMP API ユーザーズガイド

Part No: E71961
2016 年 7 月

ORACLE

Part No: E71961

ドキュメントのアクセシビリティについて

オラクルのアクセシビリティについての詳細情報は、Oracle Accessibility ProgramのWeb サイト(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc)を参照してください。

Oracle Supportへのアクセス

サポートをご契約のお客様には、My Oracle Supportを通して電子支援サービスを提供しています。詳細情報は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info)か、聴覚に障害のあるお客様は (http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs)を参照してください。

目次

例目次

このドキュメントの使用法

- **概要** – Oracle Developer Studio 12.5 C、C++、および Fortran コンパイラでサポートされている OpenMP API について説明します。
- **対象読者** – アプリケーション開発者、システム開発者、アーキテクト、サポートエンジニア
- **必要な知識** – プログラミング経験、ソフトウェア開発テスト、ソフトウェア製品の構築とコンパイルの適性

製品ドキュメントライブラリ

この製品および関連製品のドキュメントとリソースは http://docs.oracle.com/cd/E60778_01 で入手可能です。

フィードバック

このドキュメントに関するフィードバックを http://www.oracle.com/goto/docfeedback からお聞かせください。

♦ ♦ ♦ 第 1 章

1 OpenMP API について

OpenMP Application Program Interface (API) は、多数のコンピュータベンダー、教育機関、および研究者の協力によって開発された、マルチスレッドプログラムを記述するための移植性がある並列プログラミングモデルです。OpenMP の仕様書は OpenMP Architecture Review Board で作成され、発行されています。

OpenMP API は、すべての Oracle Developer Studio コンパイラに対して推奨している並列プログラミングモデルです。

1.1 サポートされている OpenMP の仕様

このマニュアルでは、OpenMP API 仕様バージョン 4.0 (このマニュアルでは OpenMP 4.0 とも呼んでいます) の Oracle Developer Studio 実装に固有の問題について説明します。この仕様は OpenMP の正式な Web サイト (`http://www.openmp.org`) にあります。

注記 - Oracle Solaris プラットフォーム上で最適なパフォーマンスおよび機能を得るために、最新バージョンの OpenMP 実行時ライブラリ `libmtsk.so` が実行中のシステムにインストールされていることを確認してください。

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

注記 - このリリースの Oracle Developer Studio は、OpenMP 4.0 の仕様を完全にサポートしています。ただし、次の点に注意してください。

- SIMD 構文は受け入れられます。ただし、SIMD 構文で SIMD 命令がまったく使用されない場合があります。
- device 構文は受け入れられます。ただし、すべてのコードはホストデバイス上で実行されます。使用可能なデバイスはホストデバイスのみです。

1.2 このドキュメントでの特別な表記

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

大かっこ [...] の中に記述された構造はオプションです。

このマニュアルでは、「Fortran」は Fortran 95 言語およびその Oracle Developer Studio コンパイラである `f95`(1) を示します。

「ディレクティブ」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。OpenMP のディレクティブは、特殊な機能を使用するようコンパイラに指示するためにプログラマによって挿入される、意味のあるコメントです。コメントなので、主体である C、C++、または Fortran 言語の一部ではなく、コンパイラオプションに応じて無視されたり実施されたりします。

♦ ♦ ♦ 第 2 章

2 OpenMP プログラムのコンパイルと実行

この章では、OpenMP API を使用するプログラムに影響するコンパイラオプションおよび実行時設定について説明します。

2.1 コンパイラオプション

OpenMP のディレクティブ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 のコンパイラオプション -xopenmp を指定してプログラムをコンパイルします。f95 コンパイラでは、-xopenmp と -openmp を同義語として使用できます。

-xopenmp フラグは、次の表にあるキーワードサブオプションを受け入れます。

-xopenmp=parallel	OpenMP ディレクティブの認識を有効にします。 -xopenmp=parallel の最小限の最適化レベルは -xO3 です。 最適化レベルが -xO3 より低い場合、コンパイラは最適化レベルを -xO3 に上げ、警告を発行します。
-xopenmp=noopt	OpenMP ディレクティブの認識を有効にします。 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。 -xopenmp=noopt を指定して最適化レベルを -xO2 -xopenmp=noopt のように明示的に -xO3 よりも低く設定すると、エラーが表示されます。 -xopenmp=noopt で最適化レベルを指定しない場合、OpenMP ディレクティブが認識され、それに従ってプログラムが並列化されますが、最適化は行われません。
-xopenmp=stubs	このオプションはサポートされていません。 *C および C++ プログラムの場合のみ:* 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 実行時ルーチンを呼び出しても OpenMP ディレクティブを無視するような 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` **注記 -** `libompstubs.a` と OpenMP 実行時ライブラリの `libmtsk.so` 両方とのリンクはサポートされていません。両方とリンクすると、予期しない動作になることがあります。
`-xopenmp=none`	OpenMP ディレクティブの認識を無効にし、最適化レベルを変更しません。

次の点にも注意してください。

- コマンド行で `-xopenmp` を指定しない場合、コンパイラではデフォルトで `-xopenmp=none` (OpenMP ディレクティブの認識を無効にする) を指定したと見なされます。
- `-xopenmp` をキーワードサブオプションなしで指定した場合、コンパイラでは `-xopenmp=parallel` を指定したと見なされます。
- `-xopenmp=parallel` または `-xopenmp=noopt` を指定すると、C/C++ では `_OPENMP` マクロに 10 進値の `201307L` が定義され、Fortran では `201307` が定義されます。ここで、2013 は年であり、07 は OpenMP 4.0 仕様の月です。
- `dbx` を使用して OpenMP プログラムをデバッグする場合は、`-xopenmp=noopt -g` を指定してコンパイルし、完全なデバッグ機能を有効にします。
- コンパイルで警告メッセージが出力されないようにするには、変更される可能性があるデフォルト値に依存するのではなく、適切な最適化レベルを明示的に指定します。
- Fortran の場合、`-xopenmp`、`-xopenmp=parallel`、または `-xopenmp=noopt` を指定してコンパイルすることは、`-stackvar` を意味します。24 ページの「スタックとスタックサイズ」を参照してください。
- 別々の手順で OpenMP プログラムをコンパイルしてリンクする場合は、コンパイル時およびリンク時にそれぞれ `-xopenmp` を含めます。
- OpenMP プログラミングの潜在的な問題に関するコンパイラの警告を表示するには、`-xvpara` オプションを `-xopenmp` オプションとともに使用します (第7章「スコープチェック」を参照)。

2.2 OpenMP 環境変数

OpenMP 仕様では、OpenMP プログラムの実行を制御する環境変数がいくつか定義されています。詳細は、http://openmp.org にある OpenMP 4.0 の仕様を参照してください。Oracle Developer Studio での OpenMP 環境変数の実装に関する情報については、第9章「OpenMP の実装によって定義される動作」を参照してください。

Oracle Developer Studio は、OpenMP 仕様に含まれていない追加の環境変数をサポートしており、それらは19 ページの「Oracle Developer Studio の環境変数」にまとめられています。

注記 - OpenMP および autopar プログラム用のスレッド数のデフォルト値は、1 ソケットあたりのコア数の倍数 (つまり、1 プロセッサチップあたりのコア数) であり、MIN (合計コア数、32) に等しいかそれよりも小さくなります。

2.2.1 OpenMP の環境変数の動作およびデフォルト値

次の表には、Oracle Developer Studio によってサポートされている OpenMP 環境変数の動作、およびそれらのデフォルト値が記載されています。環境変数に指定する値は、大文字と小文字が区別されないため、大文字でも小文字でもかまいません。

環境変数	動作、デフォルト値、および例
OMP_SCHEDULE	OMP_SCHEDULE に指定されたスケジュール型が有効な型 (static、dynamic、guided、auto、sunw_mp_sched_reserved) ではない場合、環境変数が無視され、デフォルトのスケジュール (チャンクサイズの指定されていない static) が使用されます。SUNW_MP_WARN が TRUE に設定されているか、sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。 OMP_SCHEDULE 環境変数に指定されたスケジュール型が static、dynamic、または guided であるが、指定されたチャンクサイズが負の整数の場合、次のようにチャンクサイズが設定されます。static の場合は、チャンクサイズは設定されません。dynamic または guided の場合は、チャンクサイズは 1 になります。SUNW_MP_WARN が TRUE に設定されているか、sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。 設定しない場合は、デフォルト値の static (チャンクサイズは設定されない) が使用されます。 例: **% setenv OMP_SCHEDULE "GUIDED,4"**
OMP_NUM_THREADS	OMP_NUM_THREADS に指定された値が正の整数ではない場合、この環境変数は無視されます。SUNW_MP_WARN が TRUE に設定されているか、sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。 指定された値が、実装でサポートしているスレッド数よりも大きい場合は、次のアクションが実行されます。 ■ スレッド数の動的調整が有効になっている場合は、スレッド数が減少します。また、SUNW_MP_WARN が TRUE に設定されているか、sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。 ■ スレッド数の動的調整が無効になっている場合は、エラーメッセージが表示され、プログラムの実行が停止します。 設定しない場合、デフォルトは、1 ソケットあたりのコア数の倍数 (つまり、1 プロセッサチップあたりのコア数) であり、MIN (合計コア数、32) 以下になります。 例: **% setenv OMP_NUM_THREADS 16**
OMP_DYNAMIC	OMP_DYNAMIC に指定された値が TRUE でも FALSE でもない場合は、その値は無視され、デフォルト値である TRUE が使用されます。SUNW_MP_WARN が TRUE に設定されている場

環境変数	動作、デフォルト値、および例
	合、または sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。 設定しない場合、デフォルト値は TRUE です。 例: % **setenv OMP_DYNAMIC FALSE**
OMP_PROC_BIND	OMP_PROC_BIND に指定された値が、TRUE、FALSE、あるいは master、close、または spread のコンマ区切りのリストではない場合、プロセスはゼロ以外のステータスで終了します。 初期スレッドを OpenMP の場所リストの最初の場所にバインドできない場合、プロセスはゼロ以外のステータスで終了します。 設定しない場合、デフォルトは FALSE です。 例: % **setenv OMP_PROC_BIND spread**
OMP_PLACES	OMP_PLACES に指定された値が有効ではないか、処理できない場合、プロセスはゼロ以外のステータスで終了します。 設定しない場合、デフォルト値は cores です。 例: % **setenv OMP_PLACES sockets**
OMP_NESTED	OMP_NESTED に指定された値が TRUE でも FALSE でもない場合、その値は無視され、デフォルト値である FALSE が使用されます。SUNW_MP_WARN が TRUE に設定されている場合、または sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。 設定しない場合、デフォルトは FALSE です。 例: % **setenv OMP_NESTED TRUE**
OMP_STACKSIZE	OMP_STACKSIZE に指定された値が指定書式に従っていない場合、その値は無視され、デフォルト値 (32 ビットアプリケーションでは 4M バイト、64 ビットアプリケーションでは 8M バイト) が使用されます。SUNW_MP_WARN が TRUE に設定されている場合、または sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。 ヘルパースレッドのデフォルトのスタックサイズは、32 ビットアプリケーションでは 4M バイト、64 ビットアプリケーションでは 8M バイトです。 例: % **setenv OMP_STACKSIZE 10M**
OMP_WAIT_POLICY	スレッドの ACTIVE の動作は、*スピン*です。スレッドの PASSIVE の動作は、少しの間スピンしたあとでの*スリープ*です。 設定しない場合、デフォルト値は PASSIVE です。 例: % **setenv OMP_WAIT_POLICY ACTIVE**
OMP_MAX_ACTIVE_LEVELS	OMP_MAX_ACTIVE_LEVELS に指定された値が非負整数ではない場合、その値は無視され、デフォルト値 4 が使用されます。SUNW_MP_WARN が TRUE に設定されている場合、または sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。 設定しない場合、デフォルト値は 4 です。 例: % **setenv OMP_MAX_ACTIVE_LEVELS 8**

環境変数	動作、デフォルト値、および例
OMP_THREAD_LIMIT	OMP_THREAD_LIMIT に指定された値が正の整数ではない場合、その値は無視され、デフォルト値 1024 が使用されます。SUNW_MP_WARN が TRUE に設定されている場合、または sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。 設定しない場合、デフォルト値は 1024 です。 例: % **setenv OMP_THREAD_LIMIT 128**
OMP_CANCELLATION	OMP_CANCELLATION に指定された値が TRUE でも FALSE でもない場合、その値は無視され、デフォルト値である FALSE が使用されます。SUNW_MP_WARN が TRUE に設定されている場合、または sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。 設定しない場合、デフォルトは FALSE です。 例: % **setenv OMP_CANCELLATION TRUE**
OMP_DISPLAY_ENV	OMP_DISPLAY_ENV に指定された値が TRUE、FALSE、または VERBOSEではない場合、その値は無視され、デフォルト値である FALSE が使用されます。SUNW_MP_WARN が TRUE に設定されている場合、または sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。 設定しない場合、デフォルトは FALSE です。 例: % **setenv OMP_DISPLAY_ENV VERBOSE**

2.2.2 Oracle Developer Studio の環境変数

次に示す追加の環境変数は、OpenMP プログラムの実行に影響を与えますが、OpenMP 仕様には含まれていません。次の環境変数に指定する値は、大文字と小文字が区別されないため、大文字でも小文字でもかまいません。

2.2.2.1 PARALLEL

従来のプログラムとの互換性のために提供されます。PARALLEL 環境変数を設定すると、OMP_NUM_THREADS を設定したのと同じ効果が得られます。

PARALLEL と OMP_NUM_THREADS の両方を設定する場合は、同じ値に設定する必要があります。

2.2.2.2 SUNW_MP_WARN

OpenMP 実行時ライブラリには、多くの一般的な OpenMP 違反 (領域の不正な入れ子、明示バリアの不正な配置、デッドロック、無効な環境変数の設定など) に関する警告を発行する機能があります。

環境変数 SUNW_MP_WARN は、OpenMP 実行時ライブラリによって発行される警告メッセージを制御します。SUNW_MP_WARN が TRUE に設定されている場合、実行時ライブラリは stderr に警告メッセージを発行します。環境変数が FALSE に設定されている場合、実行時ライブラリは警告メッセージを発行しません。デフォルトは FALSE です。

例:

```
% setenv SUNW_MP_WARN TRUE
```

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

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

コールバック関数のアドレスは、sunw_mp_register_warn() への引数として渡されます。sunw_mp_register_warn() は、コールバック関数が正常に登録された場合は 0、失敗した場合は 1 を返します。

プログラムでコールバック関数が登録されている場合、実行時ライブラリは登録済みの関数を呼び出し、警告メッセージを含むローカライズされた文字列のポインタを渡します。ポインタが指しているメモリーは、コールバック関数から戻ると無効になります。

注記 - 実行時チェックを有効にして、OpenMP 実行時ライブラリからの警告メッセージを表示するには、プログラムのテストまたはデバッグ中に SUNW_MP_WARN に TRUE を設定します。実行時チェックを使用すると、プログラムの実行にオーバーヘッドが加わることに注意してください。

2.2.2.3 SUNW_MP_THR_IDLE

処理の待機中 (アイドル) またはバリアで待機中の OpenMP プログラムのスレッドの動作を制御します。次のいずれかの値を設定できます。SPIN、SLEEP、SLEEP(*time* s)、SLEEP(*time* ms)、SLEEP(*time* mc)。*time* は時間を指定する整数で、s、ms、および mc は時間の単位 (それぞれ、秒、ミリ秒、マイクロ秒) を指定するオプションの接尾辞です。時間単位が指定されていない場合は、秒が使用されます。

SPIN は、処理の待機中 (アイドル) またはバリアで待機中に、スレッドがスピンすることを指定します。time パラメータなしで SLEEP を指定すると、待機中のスレッドはすぐにスリープ状態になります。time パラメータ付きで SLEEP を指定すると、スレッドは指定した時間スピンを継続し、そのあとスリープ状態になります。

デフォルトでは、しばらくスピンして待機したあとにスリープになります。SLEEP、SLEEP(0)、SLEEP(0s)、SLEEP(0ms)、および SLEEP(0mc) はすべて同じです。

SUNW_MP_THR_IDLE と OMP_WAIT_POLICY の両方を設定した場合、OMP_WAIT_POLICY は無視されます。

例:

```
% setenv SUNW_MP_THR_IDLE SPIN
% setenv SUNW_MP_THR_IDLE SLEEP
```

次に示すものはすべて同等です。

```
% setenv SUNW_MP_THR_IDLE SLEEP(5)
% setenv SUNW_MP_THR_IDLE SLEEP(5s)
% setenv SUNW_MP_THR_IDLE SLEEP(5000ms)
% setenv SUNW_MP_THR_IDLE SLEEP(5000000mc)
```

2.2.2.4 SUNW_MP_PROCBIND

SUNW_MP_PROCBIND 環境変数を使用すると、実行しているシステムのハードウェアスレッドに OpenMP スレッドをバインドできます。プロセッサバインディングによりパフォーマンスを向上させることができますが、同じハードウェアスレッドに複数のスレッドがバインドされると、パフォーマンスが低下します。SUNW_MP_PROCBIND と OMP_PROC_BIND の両方を設定することはできません。SUNW_MP_PROCBIND を設定しない場合、デフォルト値は FALSE です。詳細は、第5章「プロセッサバインディング (スレッドアフィニティー)」を参照してください。

2.2.2.5 SUNW_MP_MAX_POOL_THREADS

OpenMP ヘルパースレッドプールの最大サイズを指定します。OpenMP ヘルパースレッドは、並列領域で処理を行うために OpenMP 実行時ライブラリによって作成されるスレッドです。ヘルパースレッドプールには、最初の (メイン) 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、OpenMP ヘルパースレッドプールは空になり、すべての並列領域は最初の (メイン) スレッドによって実行されます。設定しない場合、デフォルト値は 1023 です。詳細は、29 ページの「入れ子並列処理の制御」を参照してください。

SUNW_MP_MAX_POOL_THREADS はプログラムで使用される*非ユーザーの* OpenMP スレッドの最大数を指定し、OMP_THREAD_LIMIT はプログラムで使用される*ユーザーおよび非ユーザーの* OpenMP スレッドの最大数を指定するものである点に留意してください。SUNW_MP_MAX_POOL_THREADS と OMP_THREAD_LIMIT の両方を設定する場合は、同じ値を設定する必要があります。OMP_THREAD_LIMIT の値は、SUNW_MP_MAX_POOL_THREADS の値より 1 大きい必要があります。

2.2.2.6 SUNW_MP_MAX_NESTED_LEVELS

入れ子になったアクティブな並列領域の最大数を設定します。並列領域がアクティブであると見なされるのは、複数のスレッドで構成されているチームによって実行される場合です。SUNW_MP_MAX_NESTED_LEVELS を設定しない場合、デフォルト値は 4 です。詳細は、29 ページの「入れ子並列処理の制御」を参照してください。

2.2.2.7 STACKSIZE

各 OpenMP ヘルパースレッドのスタックサイズを設定します。この環境変数は、オプションの接尾辞 B、K、M、または G の付いた数値も受け付けます。これらの接尾辞はそれぞれ、バイト、キロバイト、メガバイト、ギガバイトを表します。接尾辞を指定しない場合、デフォルトはキロバイトです。

設定しない場合、デフォルトの OpenMP ヘルパースレッドスタックサイズは、32 ビットアプリケーションでは 4M バイト、64 ビットアプリケーションでは 8M バイトです。

例:

```
% setenv STACKSIZE 8192  <- sets the OpenMP helper thread stack size to 8 Megabytes
% setenv STACKSIZE 16M   <- sets the OpenMP helper thread stack size to 16 Megabytes
```

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

2.2.2.8 SUNW_MP_GUIDED_WEIGHT

guided スケジュールが設定されたループのチャンクサイズを決定する重み係数を設定します。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 スケジュールが設定されたループすべてに適用されます。設定しない場合、デフォルトの重み計数は 2.0 です。

schedule(guided, *chunk_size*) 節を for/do ディレクティブとともに指定した場合、スレッドがループの繰り返しを要求したときにループの繰り返しがチャンク内のスレッドに割り当てられ、チャンクサイズは *chunk_size* まで減少します (最後のチャンクはより小さいサイズになることがあります)。スレッドは繰り返しのチャンクを実行し、割り当てるチャンクがなくなるまで別のチャンクを要求します。*chunk_size* が 1 の場合、各チャンクのサイズは未割り当ての繰り返しの数をスレッド数で除算した数に比例し、1 まで減少します。*chunk_size* が *k* (*k* は 1 より大きい値) の場合、各チャンクのサイズは同様に決定されますが、最後のチャンクを除いて、各チャンクに *k* 回未満の繰り返しは含まれていないという制限があります。*chunk_size* が指定されていない場合、デフォルト値は 1 です。

OpenMP 実行時ライブラリ libmtsk.so は、次の計算式を使用して、guided スケジュールが設定されたループのチャンクサイズを計算します。

chunk_size = *num-unassigned-iters* / (*guided-weight* * *num-threads*)

- *num-unassigned-iters* は、スレッドにまだ割り当てられていないループの繰り返し回数です。
- *guided-weight* は、SUNW_MP_THR_GUIDED_WEIGHT 環境変数によって指定された重み計数です (またはこの環境変数が設定されていない場合は 2.0)。
- *num-threads* は、ループを実行するために使用されるスレッドの数です。

具体的に説明するため、guided スケジュールが設定されている、繰り返しが 100 回のループがあるとします。*num-threads* = 4、重み係数 = 1.0 の場合、チャンクサイズは次のようになります。

25, 18, 14, 10, 8, 6, 4, 3, 3, 2, 1,...

また、*num-threads*= 4、重み係数 = 2.0 の場合、チャンクサイズは次のようになります。

12, 11, 9, 8, 7, 6, 5, 5, 4, 4, 3,...

2.2.2.9 SUNW_MP_WAIT_POLICY

プログラム内の処理を待機 (アイドル) するスレッド、バリアで待機するスレッド、およびタスクの完了を待機する OpenMP スレッドの動作を細かく制御できるようにします。これらの待ち状態の動作には、しばらくの間スピンする、しばらくの間プロセッサを明け渡す、呼び起こされるまで休眠状態になるの 3 つがあります。

構文 (csh を使用して示します) は次のとおりです。

```
% setenv SUNW_MP_WAIT_POLICY "IDLE=val:BARRIER=val:TASKWAIT=val"
```

IDLE、BARRIER、および TASKWAIT は、制御する待機の種類を指定するためのオプションのキーワードです。IDLE は処理を待機することを指します。BARRIER は、明示バリアまたは暗黙バリアで待機することを指します。TASKWAIT は、taskwait 領域で待機することを指します。これらの各キーワードのあとには、キーワード SPIN、YIELD、または SLEEP を使用して待機の動作を指定する *val* 設定が続きます。

SPIN(*time*) は、プロセッサを明け渡すまでに、待機中のスレッドがスピンする時間を指定します。*time* は秒、ミリ秒、またはマイクロ秒で指定します (それぞれ s、ms、mc で指定します)。*time* の単位が指定されていない場合は、秒が使用されます。SPIN に *time* パラメータが設定されていない場合、待機中のスレッドは継続的にスピンします。

YIELD(*number*) は、休眠状態になる前にスレッドがプロセッサを明け渡す回数を指定します。プロセッサが明け渡されたあと、オペレーティングシステムの実行予定時間になると、スレッドは再度実行されます。YIELD に *number* パラメータが指定されていない場合、待機中のスレッドは継続的にプロセッサを明け渡します。

SLEEP は、待機中のスレッドをただちに休眠状態にすることを指定します。

特定の種類の待ち状態に対し、SPIN、SLEEP、および YIELD 設定は任意の順序で指定できます。設定はコンマで区切る必要があります。"SPIN(0),YIELD(0)" は、"YIELD(0),SPIN(0)" と同じであり、SLEEP またはすぐにスリープ状態になることと同等です。IDLE、BARRIER、および TASKWAIT の設定を処理する場合、「一番左を最優先」規則が適用されます。「いちばん左を最優先」規則は、同じ種類の待機に異なる値が指定された場合、左端の値が適用されることを意味します。次の例では、IDLE に 2 つの値が指定されます。最初が SPIN で、2 番目は SLEEP です。SPIN が最初に指定されているため (文字列のいちばん左側にあります)、この値が OpenMP 実行時ライブラリによって適用されます。

```
% setenv SUNW_MP_WAIT_POLICY "IDLE=SPIN:IDLE=SLEEP"
```

SUNW_MP_WAIT_POLICY と OMP_WAIT_POLICY の両方を設定した場合、OMP_WAIT_POLICY は無視されます。

例 1:

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

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

例 2:

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

処理待ち状態 (アイドル) のスレッドは 10 ミリ秒スピンし、休眠状態になるまでプロセッサを 5 回明け渡します。

例 3:

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

処理待ち状態 (アイドル) のスレッドは 2 秒スピンし、休眠状態になるまでプロセッサを 2 回明け渡します。バリアで待ち状態のスレッドはただちに休眠状態になります。taskwait で待ち状態のスレッドは、プロセッサを 10 回明け渡してから休眠状態になります。

2.3 スタックとスタックサイズ

スタックは、サブプログラムまたは関数の呼び出し中、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。スレッドのスタックのサイズが小さすぎる場合は、スタックオーバーフローが発生して、通知なしでデータが壊れたり、セグメント例外が発生したりすることがあります。

実行プログラムは、プログラムを実行する最初の (メイン) スレッドのメインスタックを維持します。最初の (メイン) スレッドのスタックサイズを表示または設定するには、C シェルの limit コマンドか、Bourne シェルまたは Korn シェルの ulimit コマンドを使用します。

また、プログラムの各 OpenMP ヘルパースレッドには、独自のスレッドスタックがあります。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 変数は、スレッドスタックに割り当てられます。ヘルパースレッドスタックのデフォルトのサイズは、32 ビットアプリケーションでは 4M バイト、64 ビットアプリケーションでは 8M バイトです。ヘルパースレッドスタックのサイズを設定するには、OMP_STACKSIZE 環境変数を使用します。

-stackvar オプションを指定して 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。-xopenmp、-xopenmp=parallel、または -xopenmp=noopt オプションを指定してコンパイルされたプログラムは、-stackvar を意味しています。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、これによりスタックの

オーバーフローが発生する可能性があります。スタックの大きさが十分であることを確認してください。

C シェルの例:

```
% limit stacksize 32768    <- Sets the main thread stack size to 32 Megabytes
% setenv OMP_STACKSIZE 16384    <- Sets the helper thread stack size to 16 Megabytes
```

Bourne シェルまたは Korn シェルの例:

```
$ ulimit -s 32768    <- Sets the main thread stack size to 32 Megabytes

$ OMP_STACKSIZE=16384    <- Sets the helper thread stack size to 16 Megabytes
$ export OMP_STACKSIZE
```

2.3.1 スタックオーバーフローの検出

スタックオーバーフローを検出するには、`-xcheck=stkovf` コンパイラオプションを指定して C、C++、または Fortran プログラムをコンパイルします。構文は次のとおりです。

```
-xcheck=stkovf[:detect | :diagnose]
```

`-xheck=stkovf:detect` を指定した場合、そのエラーに通常関連付けられているシグナルハンドラを実行することによって、検出されたスタックオーバーフローエラーが処理されます。

`-xcheck=stkovf:diagnose` を指定した場合、関連付けられているシグナルをキャッチし、`stack_violation`(3C) を呼び出してエラーを診断することによって、検出されたスタックオーバーフローエラーが処理されます。スタックオーバーフローが診断されると、エラーメッセージは `stderr` に出力されます。これは `-xcheck=stkovf` のみを指定した場合のデフォルトの動作です。

`-xcheck=stkovf` コンパイラオプションについては、`cc`(1)、`CC`(1)、または `f95`(1) のマニュアルページを参照してください。

2.4 OpenMP 実行時ルーチン

このセクションでは、Oracle Developer Studio コンパイラを使用してプログラムをコンパイルした場合の特定の OpenMP 実行時ルーチンの動作について説明します。

2.4.1 omp_set_num_threads()

`omp_set_num_threads()` の引数が正の整数ではない場合、呼び出しは無視されます。`SUNW_MP_WARN` が `TRUE` に設定されているか、`sunw_mp_register_warn()` の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。

2.4.2 omp_set_schedule()

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

2.4.3 omp_set_max_active_levels()

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

omp_set_max_active_levels() の引数が非負整数ではない場合、呼び出しは無視されます。SUNW_MP_WARN が TRUE に設定されているか、sunw_mp_register_warn() の呼び出しによりコールバック関数が登録されている場合には、警告メッセージが表示されます。

2.4.4 omp_get_max_active_levels()

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

2.5 OpenMP プログラムの確認と分析

Oracle Developer Studio では、OpenMP プログラムのデバッグと分析に役立ついくつかのツールが提供されています。

- dbx は、制御されたプログラムの実行および停止したプログラムの状態の検査を行う機能を提供する対話型のデバッグツールです。dbx は、OpenMP 用に調整された機能 (並列領域へのシングルステップ実行、領域内の shared、private、threadprivate 変数の出力、並列領域とタスク領域に関する情報の出力、同期イベントの追跡など) を提供しています。詳細は、『Oracle Developer Studio 12.5: dbx コマンドによるデバッグ』を参照してください。
- コードアナライザは、静的なソースコード検査、および実行時のメモリーアクセス検査を提供するツールです。検出される静的エラーには、malloc() の戻り値検査の欠落、NULL ポインタ間接参照、関数の復帰なしなどがあります。検出されるメモリーアクセスエラーには、割り当てられていないメモリーの読み取り/書き込み、初期化されていないメモリーの読み取り、解放済みメモリーの読み取り/書き込みなどがあります。詳細は、『Oracle Developer Studio 12.5: コードアナライザユーザーズガイド』を参照してください。
- スレッドアナライザは、マルチスレッドアプリケーションでデータの競合とデッドロックを検出するためのツールです。これは、OpenMP、POSIX スレッド、Oracle Solaris スレッ

ド、またはこれらの組み合わせを使用して記述されたアプリケーションで動作します。詳細は、『Oracle Developer Studio 12.5: スレッドアナライザユーザーズガイド』および `tha`(1) と `libtha`(3) のマニュアルページを参照してください。

- パフォーマンスアナライザは、アプリケーションのパフォーマンスを分析するためのツールです。このツールは、呼び出しスタックの統計的な標本収集に基づいてパフォーマンスデータを収集し、関数、呼び出し元と呼び出し先、ソース行、および命令のパフォーマンスのメトリックを表示します。パフォーマンスアナライザは、OpenMP のパフォーマンス (OMP ワーク、OMP 待ち、および OMP オーバーヘッドのメトリック、アプリケーションのユーザーモードビューとマシンモードビューなど) を理解するために役立つ機能を提供しています。詳細は、『Oracle Developer Studio 12.5: パフォーマンスアナライザ』および `collect`(1) と `analyzer`(1) のマニュアルページを参照してください。

♦ ♦ ♦ 第 3 章

3 OpenMP 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。

3.1 OpenMP 実行モデル

OpenMP は並行実行の fork-join モデルを使用します。スレッドは並列構文を検出すると、それ自体とほかのヘルパースレッドで構成されるチームを構成します (ほかのスレッドがまったくないこともあります)。並列構文を検出したスレッドは、このチームのマスタースレッドとなり、すべてのスレッドは、並列領域内のコードを実行します。各スレッドは並列領域内での処理を終了すると、その並列領域の最後にある暗黙バリアで待ち状態となります。チーム内のすべてのスレッドがバリアで待ち状態に入れば、スレッドは解放されます。マスタースレッドだけは並列構文の処理が終了したあとも続けてプログラム内のユーザーコードを実行しますが、ヘルパー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。入れ子並列処理を無効にすると、入れ子並列領域を実行するチームは 1 つのスレッド (入れ子になった `parallel` 構文を検出したスレッド) のみで構成されます。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ヘルパースレッドがプールされていて、並列領域内での処理に使用されます。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要求する場合は、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、自身のチームの一部にします。プールに十分な数のアイドルスレッドが含まれていない場合は、検出されたスレッドが要求より少ないヘルパースレッドを取得することがあります。チームが並列領域での処理を完了すると、ヘルパースレッドはプールに返されます。

3.2 入れ子並列処理の制御

入れ子並列処理を制御するには、プログラムを実行する前にさまざまな環境変数を設定するか、`omp_set_nested()` 実行時ルーチンを呼び出します。このセクションでは、入れ子並列処理を制御するために使用できるさまざまな環境変数について説明します。

3.2.1 OMP_NESTED

入れ子並列処理は、OMP_NESTED 環境変数を設定することによって、有効または無効にすることができます。デフォルトでは、入れ子並列処理は無効になっています。

3 つのレベルを持つ、入れ子並列構文の例を次に示します。

例 1 入れ子並列処理の例

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

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

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

入れ子並列処理を無効にしてプログラムを実行すると、次の出力が生成されます。

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

3.2.2 OMP_THREAD_LIMIT

OMP_THREAD_LIMIT 環境変数の設定は、プログラム全体で使用される OpenMP スレッドの最大数を制御します。この数値には、最初の (メイン) スレッド、および OpenMP 実行時ライブラリによって作成された OpenMP ヘルパースレッドが含まれます。デフォルトでは、プログラム全体で使用される OpenMP スレッドの最大数は 1024 です (1 つの最初の (メイン) スレッドおよび 1023 個の OpenMP ヘルパースレッド)。

スレッドプールは、OpenMP 実行時ライブラリによって作成された OpenMP ヘルパースレッドのみで構成されます。プールには、最初の (メイン) スレッドやユーザーのプログラムが明示的に作成したスレッドは含まれません。

OMP_THREAD_LIMIT に 1 を設定すると、ヘルパースレッドプールが空になり、すべての並列領域が 1 つのスレッド (最初の (メイン) スレッド) によって実行されます。

次の例の出力では、プール内のヘルパースレッド数が不十分な場合、並列領域で取得されるヘルパースレッド数も少なくなる可能性があることを示しています。このコードは、環境変数 OMP_THREAD_LIMIT に 6 が設定されていることを除いて、例1「入れ子並列処理の例」のコードと同じです。アクティブ化されるすべての並列領域に必要な同時スレッドの数は、8 個です。つまり、プールには少なくとも 7 個のヘルパースレッドが含まれている必要があります。OMP_THREAD_LIMIT に 6 を設定すると、プールには最大 5 個のヘルパースレッドが含まれることになります。このため、もっとも内側にある 4 つの並列領域のうち 2 つが、必要な数のヘルパースレッドを取得できないことがあります。次の例は、可能性のある結果の 1 つを示しています。

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

3.2.3 OMP_MAX_ACTIVE_LEVELS

環境変数 OMP_MAX_ACTIVE_LEVELS は、アクティブな並列領域をいくつまで入れ子にすることができるかを制御します。並列領域がアクティブであると見なされるのは、複数のスレッドで構成されているチームによって実行される場合です。設定しない場合、デフォルト値は 4 です。

この環境変数を設定しても、入れ子になったアクティブな並列領域の最大数が制御されるだけで、入れ子並列処理は有効になりません。入れ子並列を有効にするには、OMP_NESTED を TRUE に設定するか、true() と評価される引数を指定して *omp_set_nested* を呼び出す必要があります。

次のコード例では、4 重の入れ子並列領域を作成しています。

```
#include <omp.h>
#include <stdio.h>
#define DEPTH 4
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);
}
```

次の出力は、DEPTH に 4 を設定してコード例をコンパイルおよび実行した場合の可能性のある結果を示しています。実際の結果は、オペレーティングシステムがどのようにスレッドをスケジューリングしているかによって異なります。

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

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

OMP_MAX_ACTIVE_LEVELS を 2 に設定すると、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。次の例は、可能性のある結果を示しています。

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

3.3 入れ子並列領域での OpenMP 実行時ルーチンの呼び出し

このセクションでは、入れ子並列領域内での次の OpenMP 実行時ルーチンへの呼び出し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()
- omp_set_schedule()
- omp_get_schedule()

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

get 呼び出しは、呼び出し元スレッドに値を返します。スレッドが並列領域の実行時にチームのマスターになる場合は、チームのほかのすべてのメンバーはマスタースレッドが持つ値を継承します。マスタースレッドが入れ子並列領域を終了し、その領域を取り囲む並列領域の実行を続ける場合、そのスレッドの値は、入れ子並列領域を実行する直前に、取り囲んでいる並列領域内での値に戻ります。

例 2 並列領域内での OpenMP 実行時ルーチンの呼び出し

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

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

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

```
            }
            omp_set_num_threads(7);      /* line C */
        }

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

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

次の例は、上記のプログラムを実行した場合の、可能性のある結果を示しています。

```
% 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.
```

3.4 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、2 段階の並列処理があり、OMP_NUM_THREADS に 2 が設定されているプログラムがあるとします。また、システムに 4 個のハードウェアスレッドがあり、4 個のハードウェアスレッドをすべて使用してプログラムの実行を高速化するとします。1 つのレベルを並列化するだけの場合は、2 つのハードウェアスレッドのみが使用されます。4 個のハードウェアスレッドをすべて使用するには、入れ子並列処理を有効にします。

- 入れ子並列領域では、必要以上のスレッドが作成されやすく、システムへの要求が過剰になることがあります。OMP_THREAD_LIMIT および OMP_MAX_ACTIVE_LEVELS を適切に設定して、使用されるスレッド数を制限し、過剰要求を回避します。
- 入れ子並列領域は負荷がかかります。外側の入れ子でも十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効率的です。

  たとえば、2 段階の並列処理があり、負荷が分散されているプログラムがあるとします。システムに 4 個のハードウェアスレッドがあり、4 個のハードウェアスレッドをすべて使用してこのプログラムの実行を高速化するとします。入れ子並列領域では追加のバリアが設定されるため、通常は、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、そのヘルパー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、外側の並列領域で 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 4 章

4 OpenMP のタスク化

この章では、OpenMP のタスク化モデルについて説明します。

4.1 OpenMP のタスク化モデル

タスク化を利用すると、再帰的な構造や *while* ループのように、作業単位が動的に生成されるようなアプリケーションの並列化が容易になります。

4.1.1 OpenMP タスクの実行

OpenMP では、プログラム内の任意の場所に配置できる `task` 構文を使用して*明示的タスク*を指定します。スレッドが `task` 構文を検出すると、新しいタスクが生成されます。

スレッドは、`task` 構文が検出されると、それをただちに実行するか、実行を延期してあとで実行するかを選択できます。タスクの実行が延期される場合、そのタスクは現在の並列領域に関連付けられた概念上のタスクプールに入れられます。現在のチームに属するスレッドは、プールからタスクを取り出し実行するという処理を、プールが空になるまで繰り返します。タスクを実行するスレッドが、最初にタスクを検出してプールに入れたスレッドとは異なる場合があります。

タスクに関連付けられたコードは 1 回だけ実行されます。コードを最初から最後まで同じスレッドで実行する必要がある場合、タスクは*結合*されます。コードを複数のスレッドで実行してもかまわない場合、タスクは結合解除され、タスクコードのさまざまな部分が異なるスレッドで実行されます。デフォルトではタスクは結合されており、タスクが結合解除されるように指定するには、`task` ディレクティブで `untied` 節を使用します。

スレッドは、異なるタスクを実行するために*タスクスケジューリングポイント*でタスク領域の実行を中断できます。中断されたタスクが結合されている場合は、同じスレッドにより中断されたタスクの実行が後に再開されます。中断されたタスクが結合されていない場合は、現在のチームに属するスレッドならどれでもタスクの実行を再開できます。

タスクスケジューリングポイントは、次を含むいくつかの場所で暗黙的に定義されます。

- 明示的タスクの生成直後のポイント
- `task` 領域の完了ポイントのあと

- `taskyield` 領域内
- `taskwait` 領域内
- `taskgroup` 領域の終わり
- 暗黙的および明示的な `barrier` 領域内

OpenMP 仕様では、`task` 構文を使って指定する明示的タスクに加え、*暗黙的*タスクの概念も説明されています。暗黙的タスクとは、暗黙的な並列領域によって生成されるタスク、または実行中に `parallel` 構文が検出されたときに生成されるタスクです。後者の場合、それぞれの暗黙的タスクのコードは `parallel` 構文の内部コードになります。それぞれの暗黙的タスクは、チーム内の異なるスレッドに割り当てられ、*結合*されます。

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

4.1.2 OpenMP タスクのタイプ

OpenMP 仕様には、タスク化のオーバーヘッドを減らすためにプログラマが使用できる各種のタスクが定義されています。

*非延期*タスクとは、その生成元タスクに対して実行が延期されないタスクです。つまり、生成元タスク領域は非延期タスクの実行が完了するまで中断されます。非延期タスクはそれを検出したスレッドによってただちに実行されない場合があります。プールに入れられ、検出したスレッドまたはほかのスレッドによってあとで実行される場合があります。そのタスクの実行が完了したら、生成元タスクが再開できます。非延期タスクの一例は、`false` と評価される *if* 節の式を含むタスクです。この場合、非延期タスクが生成され、検出したスレッドは現在のタスク領域を中断する必要があります。現在のタスク領域の実行は、`if` 節を含むタスクが完了するまで再開できません。

非延期タスクとは異なり、*インクルード*タスクは検出したスレッドによってただちに実行されるため、あとで実行するためにプールに入れられることはありません。このタスクの実行は、生成元タスク領域に順次追加されます。非延期タスクと同様に、生成元タスクはインクルードタスクの実行が完了するまで中断され、完了した時点で再開できます。インクルードタスクの一例は、*最終*タスクの子孫であるタスクです。

*マージ*タスクとは、その生成元タスク領域と同じデータ環境を持つタスクです。`task` ディレクティブに `mergeable` 節が存在し、生成されるタスクが非延期タスクまたはインクルードタスクの場合、実装では代わりにマージタスクの生成を選択できます。マージタスクが生成される場合は、task ディレクティブがまったく存在しないかのように動作します。

*最終*タスクとは、そのすべての子孫タスクが最終タスクおよびインクルードタスクになるように強制するタスクです。`task` ディレクティブに `final` 節が存在し、`final` 節の式が *true* と評価される場合、生成されるタスクは*最終*タスクになります。

4.2 OpenMP のデータ環境

task ディレクティブは、タスクのデータ環境を定義する次のデータ共有属性節を取ります。

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

shared 節に示された変数へのタスク内のすべての参照は、task 構文が検出された時点で存在する同じ名前を持つ変数を参照します。

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

OpenMP 仕様には、parallel、task、および work-sharing の各構文で参照される変数のデータ共有属性の決定方法が説明されています。

構文内で参照される変数のデータ共有属性は、*事前定義*、*明示的定義*、または*暗黙的定義*のいずれでもかまいません。事前定義されたデータ共有属性を持つ変数がいくつかあります。たとえば、parallel for/do 構文内のループ反復変数は private です。明示的に指定されたデータ共有属性を持つ変数は、指定された構文の中で参照される変数で、構文上のデータ共有属性節にリストされています。暗黙的に指定されたデータ共有属性を持つ変数は、指定された構文の中で参照される変数で、事前に決められたデータ共有属性を持たず、構文上のデータ共有属性節にリストがありません。

注記 - 変数のデータ共有属性を暗黙的に決めるための規則は、必ずしも明白ではない場合があります。予期しない事態を避けるため、OpenMP の暗黙的なスコープ宣言規則に依存せずに、データ共有属性節を使用して task 構文で参照されるすべての変数を必ず明示的にスコープ宣言してください。

4.3 タスク化の例

このセクションに記載された C/C++ の例は、OpenMP の task および taskwait ディレクティブをどのように使用するとフィボナッチ数列を再帰的に計算できるかを示したものです。

この例では、並列領域が 4 つのスレッドによって実行されます。single 領域により、スレッドのいずれか 1 つのみが fib(n) を呼び出す print 文を実行するようになります。

fib(n) を呼び出すと、(task ディレクティブで指定された) 2 つのタスクが生成されます。一方のタスクは fib(n-1) を呼び出し、他方のタスクは fib(n-2) を呼び出します。これらの呼び出しの戻り値が加算されて、fib(n) の戻り値が求められます。fib(n-1) および fib(n-2) を呼び

出すと、それぞれが 2 つのタスクを生成し、fib() に渡された引数が 2 より小さくなるまでタスクが再帰的に生成されます。

各 task ディレクティブの final 節に注目してください。final 節の式 (n <= THRESHOLD) が *true* と評価される場合、生成されるタスクは最終タスクになります。最終タスクの実行中に検出された task 構文はすべて、インクルードおよび最終タスクを生成します。fib() が引数 n = 9、8、...、2 で呼び出されたときは、インクルードタスクが生成されます。これらのタスクは、それを検出したスレッドによってただちに実行されるため、タスクをプールに入れるオーバーヘッドが減少します。

taskwait ディレクティブは、同じ fib() の呼び出しで生成された 2 つのタスクが完了 (すなわち、それらのタスクが i と j を計算) してから、fib() の呼び出しが復帰するようにします。

single ディレクティブとそのために fib() の最初の呼び出しを行うスレッドが 1 つだけだったとしても、4 つのすべてのスレッドが、生成されてプールに入れられているタスクの実行にかかわっている点に留意してください。

例 3 タスクを使用したフィボナッチ数列の計算

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

#define THRESHOLD 9

int fib(int n)
{
  int i, j;

  if (n<2)
    return n;

  #pragma omp task shared(i) firstprivate(n) final(n <= THRESHOLD)
  i=fib(n-1);

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

  #pragma omp taskwait
  return i+j;
}


int main()
{
  int n = 30;
  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
```

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

% CC -xopenmp -xO3 task_example.cc

% a.out
fib(30) = 832040
```

4.4 タスクスケジューリングの制約

OpenMP 仕様には、OpenMP タスクスケジューラが従う必要のあるいくつかのタスクスケジューリングの制約が示されています。

1. インクルードタスクは、生成された直後に実行されます。
2. 新しく結合されたタスクのスケジューリングは、現在スレッドに結合されているタスク領域と、バリア領域で中断されていないタスク領域のセットに制約されます。このセットが空の場合は、新しく結合されたタスクをスケジュールできます。それ以外の場合、新しく結合されたタスクをスケジュールできるのは、それがセット内のすべてのタスクの子孫タスクである場合のみです。
3. 従属タスクは、そのタスク依存関係が満たされるまでスケジュールされないものとします。
4. 式が *false* と評価される `if` 節を含む構文によって明示的タスクが生成され、前の制約がすでに満たされている場合、そのタスクは生成された直後に実行されます。

タスクスケジューリングに関するほかの前提条件に依存するプログラムは不適合です。

制約 1 と 4 の 2 つは、OpenMP タスクがただちに実行されるべきケースです。

制約 2 はデッドロックを回避するためのものです。例4「タスクスケジューリングの制約 2 の例」では、Task A、B、および C は結合されたタスクです。Task A を実行しているスレッドはクリティカルな `taskyield` 領域に入ろうとしていますが、そのスレッドはそのクリティカル領域に関連付けられたロックの所有権を保持しています。`taskyield` はタスクスケジューリングポイントであるため、Task A を実行しているスレッドは、Task A を中断して、代わりに別のタスクを実行することを選択できます。Task B と C はタスクプールに入っているものとします。制約 2 によれば、Task B は Task A の子孫ではないため、Task A を実行しているスレッドは Task B を実行できません。この時点でスケジュールできるのは Task C のみです。Task C は Task A の子孫であるからです。

かりに Task A が中断されている間に Task B がスケジュールされるとしたら、Task A が結合されているスレッドは Task B のクリティカル領域に入ることができません。そのスレッドはそのクリティカル領域に関連付けられたロックをすでに保持しているからです。その結果、デッドロックが発生します。制約 2 の目的は、コードが適合しているときにこの種のデッドロックの発生を回避することです。

プログラマが Task C 内でクリティカルなセクションを入れ子にした場合にもデッドロックが発生することがありますが、それはプログラミングエラーであることに注意してください。

例 4 タスクスケジューリングの制約 2 の例

```
#pragma omp task // Task A
{
    #pragma omp critical
    {
        #pragma omp task // Task C
        {
        }
        #pragma omp taskyield
     }
}

#pragma omp task // Task B
{
    #pragma omp critical
     {
     }
}
```

4.5 タスクの依存関係

OpenMP 4.0 仕様では、task ディレクティブの depend 節を紹介しています。これは、タスクのスケジューリングに対する追加の制約を強制するものです。これらの制約では、兄弟タスク間の依存関係のみを確立します。兄弟タスクとは、同じタスク領域の子タスクである OpenMP タスクです。

in 依存関係タイプを depend 節で指定すると、生成されたタスクは、out または inout 依存関係タイプリスト内の少なくともいずれか 1 つのリスト項目を参照する、以前に生成されたすべての兄弟タスクの従属タスクとなります。out または inout 依存関係タイプを depend 節で指定すると、生成されたタスクは、in、out、または inout 依存関係タイプリスト内の少なくともいずれか 1 つのリスト項目を参照する、以前に生成されたすべての兄弟タスクの従属タスクとなります。

次の例はタスクの依存関係を示しています。

例 5 兄弟タスクのみを同期化する depend 節の例

```
% cat -n task_depend_01.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7      int a,b,c;
     8
     9      #pragma omp parallel
```

```
10      {
11          #pragma omp master
12          {
13              #pragma omp task depend(out:a)
14              {
15                  #pragma omp critical
16                  printf ("Task 1\n");
17              }
18
19              #pragma omp task depend(out:b)
20              {
21                  #pragma omp critical
22                  printf ("Task 2\n");
23              }
24
25              #pragma omp task depend(in:a,b) depend(out:c)
26              {
27                  printf ("Task 3\n");
28              }
29
30              #pragma omp task depend(in:c)
31              {
32                  printf ("Task 4\n");
33              }
34          }
35          if (omp_get_thread_num () == 1)
36            sleep(1);
37      }
38      return 0;
39  }
```

```
% cc -xopenmp -O3 task_depend_01.c
% a.out
Task 2
Task 1
Task 3
Task 4

% a.out
Task 1
Task 2
Task 3
Task 4
```

この例では、Task 1、2、3、および 4 はすべて同じ暗黙的タスク領域の子タスクであるため、それらは兄弟タスクです。*a* 引数への依存関係が depend 節に指定されているため、Task 3 は Task 1 と 2 の従属タスクです。したがって、Task 1 と 2 の両方が完了するまで、Task 3 をスケジュールすることはできません。同様に、Task 4 は Task 3 の従属タスクであるため、Task 3 が完了するまで Task 4 をスケジュールすることはできません。

depend 節は兄弟タスクのみを同期化することに注意してください。次の例 (例6「兄弟以外のタスクに影響を及ぼさない depend 節の例」) は、depend 節が兄弟以外のタスクに影響を及ぼさないケースを示しています。

例 6 兄弟以外のタスクに影響を及ぼさない depend 節の例

```
% cat -n task_depend_02.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7      int a,b,c;
     8
     9      #pragma omp parallel
    10      {
    11          #pragma omp master
    12          {
    13              #pragma omp task depend(out:a)
    14              {
    15                  #pragma omp critical
    16                  printf ("Task 1\n");
    17              }
    18
    19              #pragma omp task depend(out:b)
    20              {
    21                  #pragma omp critical
    22                  printf ("Task 2\n");
    23
    24                  #pragma omp task depend(out:a,b,c)
    25                  {
    26                    sleep(1);
    27                    #pragma omp critical
    28                    printf ("Task 5\n");
    29                  }
    30              }
    31
    32              #pragma omp task depend(in:a,b) depend(out:c)
    33              {
    34                  printf ("Task 3\n");
    35              }
    36
    37              #pragma omp task depend(in:c)
    38              {
    39                  printf ("Task 4\n");
    40              }
    41          }
    42          if (omp_get_thread_num () == 1)
    43            sleep(1);
    44      }
    45      return 0;
    46  }

% cc -xopenmp -O3 task_depend_02.c
% a.out
Task 1
Task 2
```

```
Task 3
Task 4
Task 5
```

上記の例では、Task 5 は Task 2 の子タスクであり、Task 1、2、3、または 4 の兄弟ではありません。このため、depend 節が同じ変数 (*a*、*b*、*c*) を参照しているにもかかわらず、Task 5 と Task 1、2、3、または 4 の間に依存関係はありません。

4.5.1 タスクの依存関係に関する注意

タスクの依存関係に関する次のヒントに留意してください。

- depend 節の in、out、および inout 依存関係タイプは読み取りと書き込みの操作に似ています。ただし、in、out、および inout 依存関係タイプはタスクの依存関係を確立するためだけのものです。それらは、タスク領域内でメモリーアクセスパターンを一切示しません。depend(in:*a*)、depend(out:*a*)、または depend(inout:*a*) 節を含むタスクは、その領域内で変数 *a* の読み取りまたは書き込みを行うことはできますが、変数 *a* にアクセスすることは一切できません。
- 同じ task ディレクティブに if 節と depend 節の両方があると、if 節の条件が *false* と評価されたときに負荷が高くなる可能性があります。task に if(*false*) 節が含まれている場合、それを検出したスレッドは生成されたタスク (if(*false*) 節を含むタスク) が完了するまで、現在のタスクを中断させる必要があります。同時に、タスクスケジューラはタスクの依存関係が満たされるまで、生成されたタスクをスケジュールしてはいけません。明示的タスクの生成直後のポイントがタスクスケジューリングポイントとなるため、タスクスケジューラは非延期タスクのタスク依存関係が満たされるようにタスクをスケジュールしようとします。プール内の該当するタスクを検索してスケジュールすると、負荷が高くなる可能性があります。最悪のケースでは、taskwait 領域を使用する場合と同じ負荷がかかる可能性があります。
- 同じタスクまたは兄弟タスクの depend 節で使用されるリスト項目は、同一ストレージまたは別個のストレージを示す必要があります。そのため、配列セクションが depend 節にある場合は、配列セクションが同一または別個のストレージを示していることを確認してください。

4.6 taskwait および taskgroup を使用したタスクの同期化

taskwait または taskgroup ディレクティブを使用してタスクを同期化できます。

スレッドが taskwait 構文を検出すると、現在のタスクは、それが taskwait 領域の前に生成したすべての子タスクの実行が完了するまで中断されます。

スレッドが taskgroup 構文を検出すると、taskgroup 領域の実行が開始されます。taskgroup 領域の終わりで、現在のタスクは、taskgroup 領域でそのタスクが生成したすべての子タスクとそのすべての子孫タスクの実行が完了するまで中断されます。

taskwait と taskgroup の違いに注意してください。taskwait を使用すると、現在のタスクはその子タスクのみを待機します。taskgroup を使用すると、現在のタスクは、taskgroup 領域で生成された子タスクだけでなく、その子タスクのすべての子孫も待機します。次の 2 つの例は、その違いを示しています。

例 7 taskwait の例

```
% cat -n taskwait.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7    #pragma omp parallel
     8    #pragma omp single
     9    {
    10      #pragma omp task
    11      {
    12        #pragma omp critical
    13        printf ("Task 1\n");
    14
    15        #pragma omp task
    16        {
    17          sleep(1);
    18          #pragma omp critical
    19          printf ("Task 2\n");
    20        }
    21      }
    22
    23      #pragma omp taskwait
    24
    25      #pragma omp task
    26      {
    27        #pragma omp critical
    28        printf ("Task 3\n");
    29      }
    30    }
    31
    32    return 0;
    33  }
```

例 8 taskgroup の例

```
% cat -n taskgroup.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
```

```
 7   #pragma omp parallel
 8   #pragma omp single
 9   {
10     #pragma omp taskgroup
11     {
12       #pragma omp task
13       {
14         #pragma omp critical
15         printf ("Task 1\n");
16
17         #pragma omp task
18         {
19           sleep(1);
20           #pragma omp critical
21           printf ("Task 2\n");
22         }
23       }
24     } /* end taskgroup */
25
26     #pragma omp task
27     {
28       #pragma omp critical
29       printf ("Task 3\n");
30     }
31   }
32
33   return 0;
34 }
```

taskwait.c と taskgroup.c のソースコードはほとんど同じですが、taskwait.c の 23 行目に taskwait ディレクティブがあるのに対し、taskgroup.c の 10 行目には Task 1 と Task 2 を含む taskgroup 構文があります。どちらのプログラムでも、taskwait および taskgroup ディレクティブは Task 1 と Task 3 の実行を同期化しています。両者の違いは、それらが Task 2 と Task 3 の実行を同期化しているかどうかにあります。

taskwait.c の場合、Task 2 は、taskwait 領域がバインドされている並列領域によって生成される暗黙的タスクの子タスクではありません。そのため、taskwait 領域の終わりまでに Task 2 を終了させる必要はありません。Task 3 は Task 2 の完了前にスケジュールできます。

taskgroup.c の場合、Task 2 は、taskgroup 領域で生成される Task 1 の子タスクです。そのため、taskgroup 領域の終わりまでに Task 2 を終了させてから、Task 3 の検出およびスケジュールを行う必要があります。

4.7 OpenMP プログラミングの考慮事項

タスク化により、OpenMP プログラムを複雑にする要素が加わります。このセクションでは、タスク関連のプログラミングで考慮する問題について説明します。

4.7.1 threadprivate およびスレッド固有の情報

スレッドがタスクスケジューリングポイントを検出すると、実装時に現在のタスクが中断され、そのスレッドが別のタスクを処理するようにスケジュールされる場合があります。この動作は、タスク内の `threadprivate` 変数またはほかのスレッド固有の情報 (スレッド番号など) がタスクスケジューリングポイントの前後で変わる可能性があることを意味します。

中断されたタスクが結合されている場合、タスクの実行を再開するスレッドは、中断したときのスレッドと同じになります。このため、スレッド番号はタスクの再開後も変更されません。ただし、場合によっては `threadprivate` 変数の値が変わることがあります。スレッドが中断されたタスクを再開する前に、`threadprivate` 変数を変更する別のタスクを処理するようにスケジュールされていた可能性があるからです。

中断されたタスクが結合解除されている場合、タスクの実行を再開するスレッドは、中断したときのスレッドとは異なる場合があります。このため、スレッド番号と `threadprivate` 変数の値のどちらも、タスクスケジューリングポイントの前後で異なる可能性があります。

4.7.2 OpenMP のロック

OpenMP 3.0 以降、ロックの所有者はスレッドではなくタスクです。タスクによってロックが取得されると、そのタスクがロックを所有します。タスクの完了前に、同じタスクがそのロックを解放する必要があります。ただし、`critical` 構文は、スレッドベースの相互排他メカニズムのままです。

ロックはタスクによって所有されるので、ロックを使用する際は十分に注意してください。次の例は OpenMP 2.5 仕様に準拠しています。並列領域でロック *lck* を解放するスレッドが、プログラムの順次処理部分でそのロックを取得したのと同じスレッドであるためです。並列領域のマスタースレッドと初期のスレッドは同一です。ところが、この例はそれ以降の仕様には準拠していません。ロック *lck* を解放するタスク領域が、そのロックを取得したタスク領域と異なるためです。

例 9 OpenMP 3.0 より前のロックの使用

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

int main()
{
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;
```

```
  #pragma omp parallel shared (x)
  {
    #pragma omp master
    {
      x = x + 1;
      omp_unset_lock (&lck);
    }
  }
  omp_destroy_lock (&lck);
}
```

4.7.3 スタックデータへの参照

タスクは、task 構文が使用されているルーチン (ホストルーチン) のスタック上のデータを参照することがあります。タスクの実行は次の暗黙または明示バリアまで延期されることがあるため、ホストルーチンのスタックがポップされ、スタックデータが上書きされたあとでタスクが実行される可能性があり、それによって、タスクが参照したスタックデータが破棄されます。

このセクションの 2 つの例に示すように、必要な同期処理を挿入して、タスクが変数を参照したときに、変数が確実にスタック上にあるようにしておいてください。

例10「スタックデータ: 正しくない参照」では、*i* が `task` 構文で `shared` になるように指定されるため、タスクは、`work()` ルーチンのスタック上に割り当てられている *i* のコピーにアクセスします。

タスクの実行は延期されることがあるため、タスクは `main()` の並列領域の終わりにある暗黙バリアで、`work()` ルーチンの復帰後に実行されます。その時点で、タスクが *i* を参照すると、そのときにたまたまスタック上にあった値にアクセスしてしまいます。

正しい結果が得られるよう、タスクが完了する前に `work()` が復帰しないようにしてください。そのためには、例11「スタックデータ: 修正された参照」に示すように、`taskwait` ディレクティブを `task` 構文のあとに挿入します。あるいは、`task` 構文で、*i* を `shared` ではなく、`firstprivate` になるように指定することもできます。

例 10 スタックデータ: 正しくない参照

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

void work()
 {
   int i;

   i = 10;
   #pragma omp task shared(i)
   {
```

```
    #pragma omp critical
    printf("In Task, i = %d\n",i);
  }
}

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

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

例 11 スタックデータ: 修正された参照

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

void work()
{
  int i;

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

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

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

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

次の例では、`task` 構文中の *j* が `sections` 構文中の *j* を参照しています。このため、タスクは `firstprivate` のコピーである `sections` 構文中の *j* にアクセスします。これは、Oracle Developer Studio では、`sections` 構文のアウトラインルーチンのスタック上のローカル変数です。

タスクの実行は延期されることがあり、タスクが sections 領域の終わりにある暗黙バリアで、sections 構文のアウトラインルーチンの終了後に実行されることがあります。そのため、タスクが *j* を参照すると、スタック上の不確定の値にアクセスしてしまいます。

正しい結果を得るためには、例13「sections データ: 修正された参照」に示すように、taskwait ディレクティブを task 構文のあとに挿入して、sections 領域がその暗黙バリアに到達する前にタスクが実行されるようにしてください。あるいは、task 構文で、*j* を shared ではなく firstprivate になるように指定することもできます。

例 12 sections データ: 正しくない参照

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

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

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

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

例 13 sections データ: 修正された参照

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

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

    #pragma omp parallel shared(j)
    {
```

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

            /* Use TASKWAIT for synchronization. */
            #pragma omp taskwait
         }
      } /* Implicit barrier for sections */
   }/* Implicit barrier for parallel */

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

♦ ♦ ♦ 第 5 章

5 プロセッサバインディング (スレッドアフィニティー)

この章では、プロセッサバインディングについて説明します。

5.1 プロセッサバインディングの概要

プロセッサバインディング (*スレッドアフィニティー*とも呼ばれる) を使用すると、プログラムのすべての実行にわたってプログラム内のスレッドをマシンの同じ場所で実行し、ほかの場所に移動すべきではないことを、プログラムでオペレーティングシステムに指示できます。この場合の*場所*とは、ソケット、コア、またはハードウェアスレッドのグループを意味します。

プロセッサバインディングにより、並列領域や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入れられるというデータ再利用パターンを持つアプリケーションのパフォーマンスを向上させることができます。

コンピュータシステムは、ソケット、コア、およびハードウェアスレッドの階層として表示できます。各ソケットには 1 つ以上のコアが含まれており、各コアには 1 つ以上のハードウェアスレッドが含まれています。

Oracle Solaris プラットフォームでは、`psrinfo`(1M) コマンドを使用すると、利用可能なハードウェアスレッドを一覧表示できます。Linux プラットフォームでは、テキストファイル `/proc/cpuinfo` から、利用可能なハードウェアスレッドに関する情報を取得できます。

オペレーティングシステムがスレッドをプロセッサにバインドすると、そのスレッドは実際には特定のハードウェアスレッドまたはハードウェアスレッドのグループにバインドされます。

プロセッサへの OpenMP スレッドのバインディングを制御するには、OpenMP 4.0 の環境変数 `OMP_PLACES` および `OMP_PROC_BIND` を使用できます。または、Oracle 固有の環境変数 `SUNW_MP_PROCBIND` を使用することもできます。これらの 2 組の環境変数を混在させないようにしてください。これらの環境変数については、54 ページの「`OMP_PLACES` および `OMP_PROC_BIND`」で説明されています。

注記 - この章で説明されている OpenMP 環境変数は、OpenMP スレッドのみ (つまり、OpenMP 実行時ライブラリに記録されているすべてのユーザースレッドと、そのライブラリによって作成されたヘルパースレッド) のバインディングを制御します。それらの環境変数はほかのユーザースレッドのバインディングを制御しません。このライブラリは、ユーザースレッドが OpenMP 構文を検出したか、OpenMP ランタイムルーチンを呼び出した場合にそのユーザースレッドを記録します。

5.2 OMP_PLACES および OMP_PROC_BIND

OpenMP 4.0 では、プログラム内の OpenMP スレッドがプロセッサにバインドされる方法を指定するために OMP_PLACES および OMP_PROC_BIND 環境変数を提供しています。これらの 2 つの環境変数は多くの場合、相互に組み合わせて使用します。OMP_PLACES は、スレッドがバインドされるマシン上の*場所*を指定するために使用します。OMP_PROC_BIND は、スレッドが場所に割り当てられる方法を規定する*バインディングポリシー* (*スレッドアフィニティーポリシー*) を指定するために使用します。OMP_PLACES を単独で設定してもバインディングは有効になりません。OMP_PROC_BIND も設定する必要があります。

OpenMP 仕様では、OMP_PLACES の値は 2 種類の値のどちらかに設定できます。一連の場所 (スレッド、コア、またはソケット) を示す抽象名、または負でない数で示された場所の明示的なリストです。間隔を使用して場所を定義することもできます。その場合、<lowerbound> : <length> : <stride> 表記法を用いると、"<lower-bound>, <lower-bound> + <stride>, …, <lower-bound> + (<length>-1)*<stride>" という数値リストを表現できます。<stride> を省略すると、ユニットの刻み幅が想定されます。OMP_PLACES を設定しない場合、デフォルト値はコア数になります。

例 14 各場所にハードウェアスレッドが 1 つある場合

```
% OMP_PLACES="{0:1}:8:32"

{0:1} defines a place which has one hardware thread only, namely place {0}. The interval {0:
1}:8:32 is therefore equivalent to
{0}:8:32, which defines 8 places starting with place {0}, and the stride is 32. So the list of
 places is as follows:

Place 0: {0}
Place 1: {32}
Place 2: {64}
Place 3: {96}
Place 4: {128}
Place 5: {160}
Place 6: {192}
Place 7: {224}
```

例 15 各場所にハードウェアスレッドが 2 つある場合

```
% OMP_PLACES="{0:2}:32:8"

{0:2} defines a place which has two hardware threads, namely place {0,1}. The interval {0:2}:
24:8 is therefore equivalent to
{0,1}:24:8 which defines 24 places starting with place {0,1}, and the stride is 8. So the list
 of places is as follows:

Place 0: {0,1}
Place 1: {8,9}
Place 2: {16,17}
Place 3: {24,25}
Place 4: {32,33}
Place 5: {40,41}
Place 6: {48,49}
Place 7: {56,57}
Place 8: {64,65}
Place 9: {72,73}
Place 10: {80,81}
Place 11: {88,89}
Place 12: {96,97}
Place 13: {104,105}
Place 14: {112,113}
Place 15: {120,121}
Place 16: {128,129}
Place 17: {136,137}
Place 18: {144,145}
Place 19: {152,153}
Place 20: {160,161}
Place 21: {168,169}
Place 22: {176,177}
Place 23: {184,185}
```

OpenMP 4.0 では、2 つの環境変数 OMP_PLACES および OMP_PROC_BIND に加え、parallel ディレクティブで指定できる proc_bind 節も提供しています。proc_bind 節は、並列領域を実行するスレッドのチームがプロセッサにバインドされる方法を指定するために使用します。

OMP_PLACES および OMP_PROC_BIND 環境変数と proc_bind 節の詳細は、OpenMP 4.0 仕様を参照してください。

5.2.1 OpenMP 4.0 でのスレッドアフィニティーの制御

このセクションでは、OpenMP 4.0 仕様の OpenMP スレッドアフィニティーの制御に関するセクション 2.5.2 について詳しく説明します。

スレッドが proc_bind 節を含む並列構文を検出すると、OMP_PROC_BIND 環境変数を使用して、スレッドを場所にバインディングするためのポリシーが決定されます。並列構文に proc_bind 節が含まれている場合は、その proc_bind 節で指定されたバインディングポリシーによっ

て、OMP_PROC_BIND で指定されたポリシーがオーバーライドされます。チーム内のスレッドが場所に割り当てられると、実装時にそれが別の場所に移動されることはありません。

*マスター*スレッドアフィニティーポリシーは、チーム内のすべてのスレッドをマスタースレッドと同じ場所に割り当てるよう実行環境に指示します。このポリシーで場所パーティションは変更されないため、それぞれの暗黙的タスクは、親の暗黙的タスクの *place-partition-var* 内部制御変数 (ICV) を継承します。

*クローズ*スレッドアフィニティーポリシーは、チーム内のスレッドを親スレッドの場所に近い場所に割り当てるよう実行環境に指示します。このポリシーで場所パーティションは変更されないため、それぞれの暗黙的タスクは、親の暗黙的タスクの *place-partition-var* ICV を継承します。T がチーム内のスレッド数で、P が親の場所パーティション内の場所の数である場合、チーム内のスレッドの場所への割り当ては次のようになります。

- T <= P。マスタースレッドは親スレッド (つまり、並列構文を検出したスレッド) の場所で実行されます。次にもっとも小さいスレッド番号を持つスレッドが場所パーティション内の次の場所で実行され (以下同様に続く)、マスタースレッドの場所パーティションからラップアラウンドします。
- T > P。それぞれの場所 P には連続するスレッド番号を持つ S_p のスレッドが入ります。ここでは、floor(T/P) <= S_p <= ceiling(T/P) です。最初の S_0 スレッド (マスタースレッドを含む) は親スレッドの場所に割り当てられます。次の S_1 スレッドは場所パーティション内の次の場所に割り当てられ (以下同様に続く)、マスタースレッドの場所パーティションからラップアラウンドします。T が P によって均等に分けられない場合、特定の場所に含まれるスレッドの正確な数は実装によって定義されます。

*分散*スレッドアフィニティーポリシーの目的は、T 個のスレッドから成るチームを親の場所パーティションの P 個の場所に配分するための疎配分を作成することです。疎配分を求めるには、最初に親パーティションを T 個のサブパーティション (T <= P の場合) または P 個のサブパーティション (T > P の場合) にさらに分けます。次に、1 つのスレッド (T <= P) または 1 組のスレッド (T > P) が各サブパーティションに割り当てられます。それぞれの暗黙的タスクの *place-partition-var* ICV がそのサブパーティションに設定されます。サブパーティション分割は、疎配分を求めるためのメカニズムであるだけでなく、入れ子並列領域の作成時に使用するスレッドの一部の場所も定義します。スレッドの場所への割り当ては次のとおりです。

- T <= P。親スレッドの場所パーティションは T 個のサブパーティションに分けられ、各サブパーティションには floor(P/T) または ceiling(P/T) の連続する場所が含まれます。各サブパーティションには 1 つのスレッドが割り当てられます。マスタースレッドは、親スレッドの場所で実行され、その場所を含むサブパーティションに割り当てられます。次にもっとも小さいスレッド番号を持つスレッドが次のサブパーティション内の最初の場所に割り当てられ (以下同様に続く)、マスタースレッドの元の場所パーティションからラップアラウンドします。
- T > P。親スレッドの場所パーティションは P 個のサブパーティションに分けられ、それぞれが 1 つの場所から成ります。各サブパーティションには連続するスレッド番号を持つ S_p のスレッドが割り当てられます。ここでは、floor(T/P) <= S_p <= ceiling(T/P) です。最初の S_0 スレッド (マスタースレッドを含む) は親スレッドの場所を含むサブパーティションに割り当てられます。次の S_1 スレッドは次のサブパーティションに割り当てられ (以下同様に続く)、マスタースレッドの元の場所パーティションからラップアラウンドします。T が P によって均等に分けられない場合、特定のサブパーティションに含まれるスレッドの正確な数は実装によって定義されます。

注記 - ラップアラウンドが必要になるのは、すべてのスレッド割り当てが行われる前に、場所パーティションの終わりに達した場合です。たとえば、*クローズ*および T <= P の場合、マスタースレッドが場所パーティション内の最初の場所以外の場所に割り当てられる場合にラップアラウンドが必要になる可能性があります。この場合、スレッド 1 はマスタースレッドの場所のあとの場所に割り当てられ、スレッド 2 はそのあとの場所に割り当てられ、以下同様となります。すべてのスレッドが割り当てられる前に場所パーティションの終わりに達する可能性があります。その場合、スレッドの割り当ては場所パーティション内の最初の場所から再開されます。

5.3 SUNW_MP_PROCBIND

SUNW_MP_PROCBIND は、プロセッサバインディングを指定するための Oracle 固有のレガシー環境変数です。このセクションでは、この変数に設定できる値について説明します。

注記 - SUNW_MP_PROCBIND の値に使用される非負整数は論理ハードウェアスレッド ID を示しており、実際のハードウェアスレッド ID とは異なる場合があります。ハードウェアスレッド ID は連続した値にできますが、隙間が生じる可能性があります。たとえば、16 コアの SPARC システムの場合、ハードウェアスレッド ID は 0、1、2、3、8、9、10、11、512、513、514、515、520、521、522、523 になることがあります。ただし、論理プロセッサ 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ハードウェアスレッドの数が *n* である場合、その論理プロセッサ ID は 0、1、...、*n*-1 となります。

SUNW_MP_PROCBIND に指定できる値は次のとおりです。

- 文字列 FALSE、TRUE、COMPACT、または SCATTER (大文字と小文字のどちらも可)。次に例を示します。

```
% setenv SUNW_MP_PROCBIND "TRUE"
```

  - FALSE – OpenMP スレッドはどのプロセッサにもバインドされません。これはデフォルト設定です。
  - TRUE – OpenMP スレッドは、ラウンドロビン方式でハードウェアスレッドにバインドされます。バインディングに使用される開始ハードウェアスレッドは、最高のパフォーマンスが得られるように実行時ライブラリによって決められます。
  - COMPACT – OpenMP スレッドは、システム上のできるだけ近くにあるハードウェアスレッドにバインドされます。COMPACT では、スレッドがデータキャッシュを共有できるため、データのローカル性が向上します。
  - SCATTER – OpenMP スレッドは、遠く離れたハードウェアスレッドにバインドされます。この設定を使用すると、スレッドごとのメモリー帯域幅を向上させることができます。
- 非負整数 – OpenMP スレッドがバインドされるべきハードウェアスレッドの開始論理 ID を示します。OpenMP スレッドはラウンドロビン方式でハードウェアスレッドにバインドされます。指定された論理 ID を持つハードウェアスレッドから始まり、論理 ID *n*-1 を持つハードウェアスレッドへのバインド後、ラップアラウンドして論理 ID 0 を持つハードウェアスレッドに戻ります。

次に例を示します。

```
% setenv SUNW_MP_PROCBIND "2"
```

- 2 つ以上の非負整数のリスト – OpenMP スレッドは、指定された論理 ID を持つハードウェアスレッドにラウンドロビン方式でバインドされます。指定された以外の論理 ID を持つハードウェアスレッドは使用されません。

  次の例では、4 つのスレッドが使用される場合に、2 つのスレッドをハードウェアスレッド 2 に、1 つのスレッドをハードウェアスレッド 4 に、1 つのスレッドをハードウェアスレッド 6 にそれぞれバインドしています。

  ```
  % setenv SUNW_MP_PROCBIND "2 2 4 6"
  ```

- ハイフン (「-」) で区切られた 2 つの非負整数 – OpenMP スレッドは、最初の論理 ID から 2 番目の論理 ID までの範囲のハードウェアスレッドにラウンドロビン式でバインドされます。最初の整数を 2 番目の整数に等しいか、それよりも小さくする必要があります。この範囲外の論理 ID を持つハードウェアスレッドは使用されません。

  次に例を示します。

  ```
  % setenv SUNW_MP_PROCBIND "0-6"
  ```

SUNW_MP_PROCBIND に指定された値が無効である場合、または無効な論理 ID が指定された場合は、エラーメッセージが表示され、プログラムの実行が終了します。

OpenMP スレッドの数が使用可能なハードウェアスレッドの数よりも多い場合は、一部のハードウェアスレッドに複数の OpenMP スレッドがバインドされます。このような状況では、パフォーマンスが低下する可能性があります。

5.4 プロセッサセットとの相互作用

プロセッサセットとは、指定されたプロセスで排他的に使用するために確保されているシステムの一部のプロセッサです。プロセッサセットを使用すると、単一のプロセッサではなくプロセッサグループにプロセスをバインドできます。プロセッサセットを指定するには、Oracle Solaris プラットフォームでは psrset(1M) ユーティリティーを、Linux プラットフォームでは taskset コマンドを使用します。Linux の taskset コマンドを使って指定されたプロセッサセットは現在、プロセッサバインディングでは考慮されません。

♦ ♦ ♦ 第 6 章

6 変数の自動スコープ宣言

OpenMP 構文で参照される変数のデータ共有属性を決定することを*スコープ宣言*と呼びます。この章では、変数の自動スコープ宣言について説明します。

6.1 変数のスコープ宣言の概要

OpenMP プログラムでは、OpenMP 構文で参照されるすべての変数は、スコープ宣言されます。一般的に、構文で参照される変数は、2 つの方法のうちのどちらかでスコープ宣言されます。プログラマが*データ共有属性節*で変数のスコープを明示的に宣言するか、コンパイラによって暗黙に決まるか事前定義されたスコープに対して規則が自動的に適用されます。これは OpenMP 4.0 仕様のデータ共有属性の規則に関するセクション 2.14.1 に基づいています。データ共有属性については、OpenMP 4.0 仕様のデータ共有属性節に関するセクション 2.14.3 を参照してください。

変数を明示的にスコープ宣言することは、特に大規模で複雑なプログラムの場合、手間がかかりミスもしやすくなります。また、データ共有属性の規則によって、予期しない結果が発生することがあります。`task` ディレクティブを使用すると、スコープ宣言が複雑になり難しくなります。

Oracle Developer Studio コンパイラによってサポートされている自動的にスコープ宣言を行う機能 (*自動スコープ宣言*と呼ばれる) を使用すると、プログラマは変数のスコープを明示的に定義しなくても済みます。自動スコープ宣言では、コンパイラはシンプルなユーザーモデルで、スマートな規則に基づいて変数のスコープを決定します。

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

6.2 自動スコープ宣言用データスコープ節

自動スコープ宣言は、スコープ宣言される変数を `__auto` データスコープ節で指定するか、`default(__auto)` 節を使用することで呼び出されます。どちらも OpenMP 仕様に対する Oracle Developer Studio の拡張機能です。

6.2.1 __auto 節

構文: __auto(*list-of-variables*)

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

__auto 節は、parallel ディレクティブ (parallel for/do、parallel sections、および Fortran の parallel workshare ディレクティブを含む) または task ディレクティブに指定できます。

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

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

6.2.2 default(__auto) 節

構文: default(__auto)

Fortran の場合、DEFAULT(__AUTO) も使用できます。

default(__auto) 節は、parallelディレクティブ (parallel for/do、parallel sections、および Fortran の parallel workshare ディレクティブを含む) または task ディレクティブに指定できます。

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

6.3 parallel 構文のスコープ宣言の規則

自動スコープ宣言を行う場合、コンパイラは、parallel 構文内の変数のスコープを決定する際に、このセクションで説明されている規則を適用します。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 for/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。

6.3.1 parallel 構文内のスカラー変数のスコープ宣言の規則

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

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

6.3.2 parallel 構文内の配列のスコープ宣言の規則

- PA1: 並列構文内で配列を使用しても、その構文を実行するチーム内でスレッドに関するデータ競合状態が発生しない場合、その配列のスコープは shared と宣言されます。

6.4 task 構文内のスカラー変数のスコープ宣言の規則

自動スコープ宣言を行う場合、コンパイラは、task 構文内のスカラー変数のスコープを決定する際に、このセクションで説明されている規則を適用します。

注記 - このリリースの Oracle Developer Studio では、タスクの自動スコープ宣言で配列は処理されません。

---

task 構文で参照され、暗黙に決まるか事前定義されたスコープを持たないスカラー変数を自動スコープ宣言する場合、コンパイラは、変数の使用を規則 TS1 - TS5 に対して順番に確認します。これらの規則は、OpenMP 仕様で暗黙にスコープ宣言される、parallel for/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。

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

- TS3: 変数を使用してもデータ競合が発生せず、task 構文で読み取り専用であり、かつ、タスクの実行中には変数にアクセスできない場合は、その変数は firstprivate と自動スコープ宣言されます。
- TS4: 変数を使用するとデータ競合が発生し、タスク構文を実行する各スレッドで、その変数が同じスレッドによる読み取りの前に常に書き込まれ、タスク内で変数に割り当てられた値がタスク外では使用されない場合、その変数は private と自動スコープ宣言されます。
- TS5: 変数を使用するとデータ競合が発生し、変数が task 構文内で読み取り専用ではなく、タスクで行われる読み取りの中で、タスク外で割り当てられた値が取得されることがあり、タスク内で変数に割り当てられた値がタスク外では使用されない場合、その変数は firstprivate と自動スコープ宣言されます。

6.5 自動スコープ宣言に関する注意事項

_auto(*list-of-variables*) または default(_auto) 節を parallel 構文に対して指定しても、parallel 構文に字句的または動的に包含される task 構文にも同じ節が適用されることを意味するわけではありません。

事前定義された暗黙的スコープを持たない変数を自動スコープ宣言する場合、コンパイラは、変数の使用について、規則に対して順番に確認します。規則に一致する場合、コンパイラはその規則に従って変数のスコープを決定します。一致する規則がない場合、または自動スコープ宣言が変数を処理できない場合、コンパイラは変数を shared とスコープ宣言し、parallel または task 構文を if(0) (Fortran では if(.false.)) 節が指定されているかのように処理します。詳細は、62 ページの「自動スコープ宣言を使用する際の制限事項」を参照してください。

変数の使用がどの規則とも一致しない場合、またはコンパイラが十分な解析を行うにはソースコードが複雑すぎる場合は、通常、変数を自動スコープ宣言できません。こうした原因としてよくあるのは、たとえば、関数呼び出しや複雑な配列添え字、メモリー別名、ユーザー実装の同期などです。

6.6 自動スコープ宣言を使用する際の制限事項

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

- タスクの自動スコープ宣言では、結合解除されたタスクは処理できません。
- タスクの自動スコープ宣言では、ほかのタスクに字句的に包含されているタスクは処理できません。次に例を示します。

```
#pragma omp task /* task 1 */
{
  ...
  #pragma omp task /* task 2 */
  {
    ...
  }
 ...
}
```

この例では、コンパイラは、task 1 に字句的に入れ子になった task 2 の自動スコープ宣言は試行しません。コンパイラは task 2 で参照されているすべての変数を shared とスコープ宣言し、task 2 を if(0) (Fortran では if(.false.)) 節がタスクで指定されているかのように処理します。

- 解析では、OpenMP ディレクティブのみ認識、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 ディレクティブを使用してください。
- データ競合解析では、barrier や master などの OpenMP 同期ディレクティブを使用して指定された同期のみが認識され、使用されます。ビジー待ちなどのユーザー実装の同期は認識されません。

6.7 自動スコープ宣言結果の確認

自動スコープ宣言の詳細な結果は*コンパイラ解説*に表示されます。ソースが -g オプションを指定してコンパイルされている場合、コンパイラはインライン解説を生成します。解説は次の例に示すように er_src コマンドを使用すると表示できます。er_src コマンドは、Oracle Developer 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『Oracle Developer Studio 12.5: パフォーマンスアナライザ』を参照してください。

自動スコープ宣言の結果を簡単に確認するには、-xvpara オプションを指定してコンパイルします。–xvpara を使用してコンパイルすると、特定の構文の自動スコープ宣言が成功したかどうかを把握することができます。

例 16 -xvpara を指定した場合の自動スコープ宣言の結果の確認

```
% cat source1.f
```

```
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END

% f95 -xopenmp -xO3 -xvpara -c -g source1.f
"source1.f", line 2: Autoscoping for OpenMP construct succeeded.
Check er_src for details
```

-xvpara を指定した場合、特定の構文の自動スコープ宣言が失敗すると、次の例に示すような警告メッセージが発行されます。

例 17 -xvpara を指定した場合の自動スコープ宣言の失敗

```
% cat source2.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         CALL FOO(X)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END

% f95 -xopenmp -xO3 -xvpara -c -g source2.f
"source2.f", line 2: Warning: Autoscoping for OpenMP construct failed.
 Check  er_src for details. Parallel region will be executed by
 a single thread.
```

自動スコープ宣言のより詳細な情報は、次の例に示すように、er_src によって表示されるコンパイラ解説に表示されます。

例 18 er_src を使用した自動スコープ宣言の詳細な結果の表示

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

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

   Source OpenMP region below has tag R1
   Variables autoscoped as SHARED in R1: y
   Variables autoscoped as PRIVATE in R1: t, i
   Variables treated as shared because they cannot be autoscoped in R1: x
   R1 will be executed by a single thread because
     autoscoping for some variable s was not successful
```

```
Private variables in R1: i, t
Shared variables in R1: y, x
  2. C$OMP PARALLEL DO DEFAULT(__AUTO)

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

6.8 自動スコープ宣言の例

このセクションでは、自動スコープ宣言規則の使用例をいくつか示します。この規則は、60 ページの「parallel 構文のスコープ宣言の規則」および61 ページの「task 構文内のスカラー変数のスコープ宣言の規則」に記載されています。

例 19 自動スコープ宣言の規則を示す複雑な例

```
 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
```

この例では、関数 FOO() には parallel 構文が 1 つあり、この parallel 構文には、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。

parallel 構文では、I、N、MM、T、W、M、X、および Y という変数が使用されています。コンパイラは次のようにこれらの変数のスコープを決定します。

- スカラー I は、ワークシェリング 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 は並列構文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 PS1 に従って、この変数のスコープは shared と宣言されます。
- 並列構文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 PS1 は適用されません。この書き込みは、同じスレッド内の MM の読み取り前に行われるため、規則 PS2 に従って、MM のスコープは private と宣言されます。
- 同様に、スカラー 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、PS1 および PS2 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 PS3 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 11 行目で書き込まれます。この single 構文の末尾の暗黙バリアは、11 行目の書き込みが 28 行目の読み取りや 29 行目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、28 行目と 29 行目はどちらも 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、parallel 構文内での 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 は構文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 PA1 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。データの競合が発生しないため、Y のスコープは、規則 PA1 に従って shared と宣言されます。

例 20 QuickSort の例

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

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

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

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

er_src shows the following compiler commentary:

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

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

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

例 21 フィボナッチの例

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

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

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

   #pragma omp taskwait
   return x + y;
}

er_src shows the following compiler commentary:

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

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

スカラー *n* は `task` 構文でも `parallel` 構文でも読み取り専用です。そのため、*n* は `TS1` に従って `firstprivate` と自動スコープ宣言されます。

スカラー変数 *x* および *y* は、関数 `fib()` のローカル変数です。両方のタスクが *x* と *y* にアクセスしても、データ競合は起こりません。`taskwait` があるため、2 つのタスクがまず実行を完了してから、`fib()` (タスクを検出して生成) を実行していたスレッドが `fib()` を終了します。これは、2 つのタスクの実行中に *x* と *y* にアクセスできることを意味します。そのため、*x* と *y* は、`TS2` に従い、`shared` と自動スコープ宣言されます。

例 22 `single` 構文および `task` 構文を使用した例

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

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

    #pragma omp single
```

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

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

  return 0;
}

er_src shows the following compiler commentary:

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

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

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

   Source OpenMP region below has tag R4
   Variables autoscoped as PRIVATE in R4: yy
   Variables autoscoped as FIRSTPRIVATE in R4: xx
   Private variables in R4: yy
   Firstprivate variables in R4: xx
    19.     #pragma omp task default(__auto) // task2
    20.     {
    21.        yy = xx;
    22.     }
    23.   }
```

この例では、*xx* は parallel 構文の private 変数です。チームのスレッドの 1 つが *task1* を実行して、xx の初期値を変更します。その後、すべてのスレッドが task2 を検出し、*xx* を使用して何らかの計算を行います。

task1 では、*xx* を使用しても、データ競合は発生しません。single 構文の終わりに暗黙バリアがあり、このバリアを出る前に task1 を完了する必要があるので、*xx* は task1 の実行中もアクセスできます。したがって、TS2 に従い、*xx* は task1 で shared と自動スコープ宣言されます。

task2 では、*xx* は読み取り専用として使用されます。ただし、*xx* の使用は、包含する parallel 構文では読み取り専用ではありません。*xx* は、parallel 構文に対しては private と事前定義されているので、task2 の実行中も *xx* にアクセスできるかどうかはわかりません。したがって、TS3 に従い、*xx* は task2 で firstprivate と自動スコープ宣言されます。

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

例 23 task 構文および taskwait 構文を使用した例

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

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

       #pragma omp atomic
       yy += xx;
    }

    #pragma omp taskwait
  }
  return 0;
}

er_src shows the following compiler commentary:

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

   Source OpenMP region below has tag R2
   Variables autoscoped as SHARED in R2: yy
   Variables autoscoped as FIRSTPRIVATE in R2: xx
   Shared variables in R2: yy
   Firstprivate variables in R2: xx
```

```
 7.     #pragma omp task default(__auto)
 8.     {
 9.        xx += 1;
10.
11.        #pragma omp atomic
12.        yy += xx;
13.     }
14.
15.     #pragma omp taskwait
16.   }
```

task 構文では xx は読み取り専用として使用されないので、データ競合が発生します。ただし、タスクの *x* の読み取りではタスクの外部で定義された *x* の値が取得されます (parallel 構文では *xx* は shared であるため)。そのため、TS5 に従い、*xx* は firstprivate と自動スコープ宣言されます。

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

♦ ♦ ♦ 第 7 章

7 スコープチェック

Oracle Developer Studio C、C++、および Fortran のコンパイラは、スコープチェック機能を備えており、OpenMP プログラムの変数が正しくスコープ宣言されたかどうかがコンパイラによって確認されます。この章では、スコープチェック機能の使用方法について説明します。

7.1 スコープチェックの概要

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

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

7.2 スコープチェック機能の使用

スコープチェックを有効にするには、`-xvpara` および `-xopenmp` オプションを指定して OpenMP プログラムをコンパイルします。最適化レベルは `-xO3` 以上にしてください。スコープチェックは、プログラムが `-xopenmp=noopt` でコンパイルされただけでは動作しません。最適化レベルが `-xO3` 未満の場合、コンパイラは警告メッセージを発行し、スコープチェックを行いません。

スコープチェック時には、コンパイラはすべての OpenMP 構文を確認します。いくつかの変数のスコープ宣言に問題がある場合、コンパイラは警告メッセージを発行し、場合によっては、正しいデータ共有属性節を提案します。たとえば、コンパイラが次の状態を検出すると、警告メッセージが表示されます。

- 異なるループ繰り返し間でデータに依存関係がある場合に、OpenMP ディレクティブを使用して並列化されたループ。

- OpenMP のデータ共有属性節に問題がある可能性がある。たとえば、並列領域内の変数へのアクセスがデータ競合を招く可能性があるときに、並列領域に shared になる変数を指定した場合、または並列領域内の変数に割り当てられた値が並列領域のあとで使用されるときに、並列領域に private になる変数を指定した場合です。

次の例は、スコープチェックを示しています。

例 24 -xvpara を使用したスコープチェック

```
% cat t.c

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

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

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

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

  return 0;
}

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

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

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

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

次の例は、潜在的なスコープエラーがどのように報告されるかを示しています。

例 25 スコープ宣言エラーの例

```
% cat source2.c
```

```
#include <omp.h>

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

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

  a = b;
  return 0;
}
```

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

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

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

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

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

この例は、スコープチェックによって検出される典型的なエラーを示しています。

1. 縮約変数として *r* が指定され、この変数の演算は + となっているが、実際に行われるべき演算は * です。
2. private として *a* が明示的にスコープ宣言されています。private 変数には初期値がないので、行 11 の *a* への参照では、未定義の値が取得されることがあります。コンパイラはこの問題を指摘し、a を firstprivate としてスコープ宣言するように提案します。
3. 変数 *i* はループインデックス変数です。ループインデックスの値が parallel for ループのあとに使用される場合、プログラマはこれを LASTPRIVATE として指定することもあります。ただし、上記の例では、*i* はループ後にまったく参照されません。コンパイラは警告を発行し、*i*

を private としてスコープ宣言するように提案します。private を lastprivate の代わりに使用すると、パフォーマンスが向上します。

4. 変数 *b* にはデータ共有属性が明示的に指定されていません。OpenMP 仕様によると、b は shared として暗黙的にスコープ宣言されます。ただし、*b* を shared としてスコープ宣言すると、データ競合が発生します。*b* の正しいデータ共有属性は reduction です。

7.3 スコープチェックを使用する際の制限事項

- スコープチェックは、最適化レベルが -xO3 以上に設定されている場合にのみ動作します。スコープチェックは、プログラムが -xopenmp=noopt でコンパイルされただけでは動作しません。
- 解析では、OpenMP ディレクティブのみ認識、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 ディレクティブを使用してください。
- データ競合解析では、barrier や master などの OpenMP 同期ディレクティブを使用して指定された同期のみが認識され、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。

注記 - -xvpara コンパイラオプションを使用してスコープチェックを行うと、静的 (コンパイル時) 解析を使用して、プログラムの潜在的な問題が判別されます。一方、スレッドアナライザツールは、動的 (実行時) 解析を使用して、プログラムでのデータ競合およびデッドロックをチェックします。これらの両方のアプローチを使用して、プログラムでできるかぎり多くのエラーを検出してください。

♦ ♦ ♦ 第 8 章

8 パフォーマンス上の検討事項

正しく動作する OpenMP アプリケーションを作成したら、全体的なパフォーマンスについて検討します。この章では、OpenMP アプリケーションの効率性およびスケーラビリティーを改善するためのベストプラクティスについて説明します。

8.1 パフォーマンス上の一般的な推奨事項

このセクションでは、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技法について説明します。

- 同期を最小限に抑える。
  - `barrier`、`critical`、`ordered`、`taskwait`、ロックなどの同期の使用を避けるか、最小限にします。
  - 可能な場合は `nowait` 節を使用して、冗長または不要なバリアを取り除いてください。たとえば、並列領域の最後につねに暗黙のバリアがあります。領域内にある後続のコードがないワークシェアリングループに `nowait` を追加すると、1 つの冗長なバリアが取り除かれます。
  - プログラム内のすべての `critical` セクションによってデフォルトの同じロックが使用されないように、必要に応じて名前付きの `critical` セクションを使用してきめ細かくロックします。
- `OMP_WAIT_POLICY`、`SUNW_MP_THR_IDLE`、または `SUNW_MP_WAIT_POLICY` 環境変数を使用して、待機スレッドの動作を制御します。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。タイムアウト期間の終わりまでに作業が見つからない場合、スレッドはスリープ状態になり、ほかのスレッドを犠牲にしてプロセッササイクルを浪費することを回避します。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適切ではない場合、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。通常、アプリケーションが専用のプロセッサで実行される場合は、待機スレッドをスピンさせるアクティブ待機ポリシーを使用すると、パフォーマンスが向上します。アプリケーションがほかのアプリケーションと同時に実行される場合は、待機スレッドをスリープさせるパッシブ待機ポリシーを使用すると、システムスループットが向上します。
- できるかぎりもっとも高いレベル (いちばん外側のループなど) で並列化してください。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、並列領域をできるかぎり大きくします。たとえば、この構文では効率が低くなります。

```
#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }
}

#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }
}
```

次の構文の方が効率が高くなります。

```
#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }

   #pragma omp for
   {
      ...
   }
}
```

- `parallel` 構文の内部で入れ子になっているワークシェアリング `for/do` 構文の代わりに、`parallel for/do` 構文を使用します。たとえば、この構文では効率が低くなります。

```
#pragma omp parallel
{
   #pragma omp for
   {
      ... statements ...
   }
}
```

この構文の方が効率が高くなります。

```
#pragma omp parallel for
{
```

```
   ... statements ...
}
```

- 可能な場合は、並列ループをマージして、並列処理のオーバーヘッドを減らします。たとえば、2 つの `parallel for` ループをマージします。

```
#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 1 ...
   }

#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 2 ...
   }
```

マージされた単一の `parallel for` ループの方が効率的です。

```
#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 1 ...
     ... statements 2 ...
   }
```

- the `OMP_PROC_BIND` または `SUNW_MP_PROCBIND` 環境変数を使用して、スレッドをプロセッサにバインドします。static スケジュール指定とともにプロセッサバインディングを使用すると、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。第5章「プロセッサバインディング (スレッドアフィニティー)」を参照してください。
- 可能な場所では、できるかぎり `single` ではなく、`master` を使用してください。
  - `master` ディレクティブは、暗黙バリアのない `if` 文として実装されます。`if (omp_get_thread_num() == 0) {...}`
  - `single` 構文は、ほかのワークシェアリング構文に似た実装になります。どのスレッドが最初に `single` に達するかを記録すると、実行時のオーバーヘッドが増加します。さらに、`nowait` が指定されていない場合は暗黙バリアがあり、効率が低くなります。
- 適切なループスケジュールを選択してください。
  - `static` ループスケジュールでは同期が要求されず、データがキャッシュに収まる場合、データの近傍性を維持できます。ただし、`static` スケジュールは、負荷の不均衡をもたらすことがあります。
  - `dynamic` および `guided` ループスケジュールは、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ます。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。

- 効率的でスレッドセーフなメモリー管理を使用します。アプリケーションが `malloc()` 関数および `free()` 関数を明示的に、あるいは暗黙的に動的配列、割り当て可能な配列、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内で使用していることがあります。標準 C ライブラリ `libc.so` 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.so` ライブラリなどのほかのライブラリでは、より高速のバージョンが提供されています。`-lmtmalloc` を指定して、`libmtmalloc.so` とリンクします。
- 小さいデータセットの場合、OpenMP の並列領域が十分に機能しないことがあります。`parallel` 構文では `if` 節を使用して、ある程度のパフォーマンス向上が期待できる場合にのみ領域を並列実行させるように指定します。
- アプリケーションにある程度以上のスケーラビリティーがない場合は、入れ子並列処理を試してください。ただし、すべての入れ子並列領域のスレッドチームはバリアで同期する必要があり、同期のオーバーヘッドが発生するため、入れ子並列処理は注意して使用してください。また、入れ子並列処理によってマシンに過剰な要求が発生し、パフォーマンスが低下することがあります。
- 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。
  - 領域から戻る前に、スレッドのプライベートメモリーから共有メモリーにデータをコピーする必要があります。
  - `lastprivate` のチェックが追加されます。たとえば、`lastprivate` 節を指定したワークシェアリングループのコンパイルされたコードは、順序の最後の繰り返しを実行するスレッドをチェックします。これにより、ループ内の各チャンクの終わりに追加の処理が生じることになり、多数のチャンクがある場合はその負荷が大きくなります。
- 明示的な `flush` の使用には注意してください。flush によってデータがメモリーに格納され、以降のデータアクセスで、メモリーからのリロードが必要になることがあります。このすべてが効率の低下をもたらします。

8.2 偽りの共有の回避

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

8.2.1 「偽りの共有」とは

ほとんどのメモリー共有型マルチプロセッサコンピュータには、各プロセッサに独自のローカルキャッシュがあります。このキャッシュは、低速のメモリーとプロセッサの高速レジスタの間のバッファーとして動作します。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、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、キャッシュから行われます。

*偽りの共有*は、異なるプロセッサのスレッドが同じキャッシュ行にある変数を変更すると発生します。この状況は (真の共有と区別するために) 偽りの共有と呼ばれます。スレッドが同じ変数にアクセスせずに、同じキャッシュ行に偶然あった別の変数にアクセスしているためです。

スレッドがキャッシュ内の変数を変更すると、その変数があるキャッシュ行全体が無効としてマークされます。別のスレッドが同じキャッシュ行の変数にアクセスしようとすると、変更されたキャッシュ行がメモリーに書き戻され、スレッドはメモリーからキャッシュ行を取得します。これが発生するのは、キャッシュ整合性をキャッシュ行のレベルで維持するためであり、個別の変数または要素のためではありません。偽りの共有では、アクセスしようとしている変数が変更されていなくても、スレッドは強制的にキャッシュ行のより新しいコピーをメモリーから取得します。

偽りの共有が頻繁に発生すると、インターコネクトトラフィックが増加し、OpenMP アプリケーションのパフォーマンスとスケーラビリティーが大幅に低下します。偽りの共有によってパフォーマンスが低下するのは、次の条件のすべてが満たされる場合です。

- 複数のスレッドによって共有データが変更される
- 複数のスレッドが同じキャッシュ行内のデータを変更する
- データが頻繁に変更される (密なループなど)

読み取り専用の共有データにアクセスしても偽りの共有にはならないことに注意してください。

8.2.2 偽りの共有の低減

通常、偽りの共有が検出されるのは、特定の変数へのアクセスに著しくコストがかかっていると思われる場合です。アプリケーションの実行で主要な役割を果たす並列ループを綿密に分析することによって、偽りの共有によって引き起こされるパフォーマンスおよびスケーラビリティー上の問題を明らかにすることができます。

一般に、偽りの共有は次の手法を使用して減らすことができます。

- できるだけ多くの private または threadprivate のデータを使用する。
- 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。
- 各スレッドのデータが別個のキャッシュ行に配置されるようにデータ構造をパディングする。パディングのサイズはシステムによって異なり、個々のキャッシュ行にスレッドのデータをプッシュするために必要なサイズです。
- スレッド間のデータの共有が少なくなるようにデータ構造を変更する。

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 (*chunk_size* の値を変更する) ことで偽りの共有が減少することもあります。

8.3 Oracle Solaris OS のチューニング機能

Oracle Solaris オペレーティングシステムは、OpenMP プログラムのパフォーマンスを改善する機能をサポートしています。これらの機能には、メモリー配置の最適化 (MPO) および複数ページサイズサポート (MPSS) があります。

8.3.1 メモリー配置の最適化

メモリー共有型マルチプロセッサコンピュータには、複数のプロセッサが搭載されています。それぞれのプロセッサは、そのコンピュータのすべてのメモリーにアクセスできます。メモリー共有型マルチプロセッサには、プロセッサごとに特定のメモリー領域に対して、より高速なアクセスを可能にするメモリーアーキテクチャーを採用しているものがあります。このため、メモリーにアクセスするプロセッサの近くにメモリーを割り当てると、待機時間が減少し、アプリケーションのパフォーマンスが向上します。

Oracle Solaris オペレーティングシステムには、MPO 機能の一部である近傍性グループ (`lgroup`) の抽象化が導入されています。`lgroup` は、プロセッサやメモリーのようなデバイスのセットであり、セット内の各プロセッサは制限された待機時間間隔内にそのセット内の任意のメモリーにアクセスできます。ライブラリ `liblgrp.so` は、アプリケーションが監視と性能チューニングのために使用できるように、`lgroup` の抽象化をエクスポートします。アプリケーションは `liblgrp.so` API を使用して、次のタスクを実行できます。

- グループ階層の検索
- 指定された `lgroup` の内容と特性の検出
- `lgroup` でのスレッドとメモリー配置の操作

デフォルトでは、Oracle Solaris オペレーティングシステムは、スレッドのホーム `lgroup` からリソースを割り当てようと試みます。たとえば、デフォルトでは、オペレーティングシステムは、スレッドのホーム `lgroup` にあるプロセッサ上でそのスレッドを実行し、スレッドのホーム `lgroup` にそのスレッドのメモリーを割り当てるスケジュールを設定しようと試みます。

次のメカニズムを使用すると、`lgroup` に関係するスレッドおよびメモリーの配置の検出および操作を行うことができます。

- `meminfo()` システムコールを使用すると、メモリーの配置を検出できます。
- `lgrp_home()` 関数を使用すると、スレッドの配置を検出できます。
- `lgrp_affinity_set()` 関数を使用すると、指定した `lgroup` にスレッドのアフィニティーを設定することによって、スレッドおよびメモリーの配置を操作できます。
- 標準 C ライブラリの `madvise()` 関数を使用すると、ユーザーの仮想メモリーの領域が特定のパターンで使用されることが予期されるとオペレーティングシステムに通知できます。`madvise()` に渡される `MADV_ACCESS` フラグを使用すると、`lgroup` 間のメモリー割り当てを操作できます。たとえば、`MADV_ACCESS_LWP` フラグを指定して `madvise()` を呼び出すと、指定したアドレス範囲にアクセスする次のスレッドはそのメモリー領域にもっともアクセスす

るスレッドであることがオペレーティングシステムに通知されます。OS はそれに応じてこの範囲のメモリーおよびスレッドを配置します。

`lgroup` API の詳細は、『Oracle Solaris 11.3 Programming Interfaces Guide』の 第 4 章,「Locality Group APIs」を参照してください。`madvise()` 関数の詳細は、`madvise`(3C) のマニュアルページを参照してください。

8.3.2 複数ページサイズサポート

Oracle Solaris の Multiple Page Size Support (MPSS) 機能を使用すると、アプリケーションで仮想メモリーの領域ごとに異なるページサイズを使用できます。特定のプラットフォームのデフォルトのページサイズは、`pagesize` コマンドを使用すると取得できます。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。

トランスレーションルックアサイドバッファー (TLB) は、仮想メモリーアドレスを物理メモリーアドレスにマップするために使用されるデータ構造です。TLB で使用可能な仮想と物理のマッピング情報がないメモリーアクセスには、性能ペナルティーが関連付けられます。ページサイズが大きいと、TLB が固定数の TLB エントリを使用して、より多くの物理メモリーを割り当てます。このため、ページサイズが大きくなると、仮想と物理のメモリーマッピングのコストが減少し、全体的なシステムパフォーマンスが向上することがあります。

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いくつかあります。

- Oracle Solaris コマンドの `ppgsz`(1) を使用する。
- `-xpagesize`、`-xpagesize_heap`、または `-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、`cc`(1)、`CC`(1)、または `f95`(1) のマニュアルページを参照してください。
- `mpss.so.1` 共有オブジェクトを事前ロードすると、環境変数を使用してページサイズを設定できるようになります。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。

♦ ♦ ♦ 第 9 章

9 OpenMP の実装によって定義される動作

この章では、Oracle Developer Studio コンパイラを使用してプログラムをコンパイルした場合の特定の OpenMP 機能の動作について説明します。実装によって定義されると仕様に記述されている動作のサマリーについては、OpenMP 4.0 仕様の付録 D を参照してください。

9.1 OpenMP メモリーモデル

複数スレッドから同じ変数への非同期のメモリーアクセスが、互いに不可分なものになるとは限りません。アクセスが不可分にものなるかどうかは、実装依存、およびアプリケーション依存の要因による影響を受けます。変数によっては、対象プラットフォームでの最大の不可分なメモリー動作よりも大きい場合があります。変数によっては、境界整列が誤っているか、不明な境界整列である場合があります。より多くのロード (または格納) を使用する、高速なコードシーケンスもあります。このため、コンパイラまたは実行時システムがその変数にアクセスするためには、複数回のロード (または格納) が必要になることがあります。

メモリーの更新がビットフィールド変数に対するものである場合、メモリーの更新によって別の変数 (配列、構造体の要素など) の一部である隣接する変数の読み取りおよび書き込みが行われる可能性がある最小サイズは、基本言語によって要求されるサイズと同じです。メモリーの更新がビットフィールド変数ではない変数に対するものである場合、その更新では別の変数 (配列、構造体の要素など) の一部である隣接する変数の読み取りおよび書き込みは行われません。

9.2 OpenMP 内部制御変数

次の内部制御変数は、実装時に定義されます。

- *bind-var*: 場所へのスレッドのバインドを制御します。*bind-var* の初期値は `FALSE` です。
- *default-device-var*: デフォルトのターゲットデバイスを制御します。*default-device-var* の初期値は 0 (ホストデバイス) です。
- *def-sched-var*: ループ領域のデフォルトスケジューリングとして定義された実装を制御します。*def-sched-var* の初期値は、`static` で、チャンクサイズはありません。
- *dyn-var*: 検出された `parallel` 領域に対してスレッド数の動的な調整を有効にするかどうかを制御します。*dyn-var* の初期値は `TRUE` (動的な調整が有効) です。

- *max-active-levels-var*: アクティブな入れ子の並列領域の最大数を制御します。*max-active-levels-var* の初期値は 4 です。
- *nthreads-var*: 検出された並列領域に対して要求されたスレッド数を制限します。*nthreads-var* の初期値はコアの数と同じです (最大 32)。
- *place-partition-var*: 検出された並列領域の実行環境で使用可能な場所パーティションを制御します。*place-partition-var* の初期値は `cores` です。
- *run-sched-var*: `schedule(runtime)` 節がループ領域に使用するスケジュールを制御します。*run-sched-var* の初期値は、`static` で、チャンクサイズはありません。
- *stacksize-var*: OpenMP の実装が作成するスレッド (ヘルパースレッドとも呼ばれます) のスタックサイズを制御します。*stacksize-var* の初期値は、32 ビットアプリケーションでは 4M バイト、64 ビットアプリケーションでは 8M バイトです。
- *thread-limit-var*: OpenMP プログラムに所属するスレッドの最大数を制御します。*thread-limit-var* の初期値は 1024 です。
- *wait-policy-var*: 待ち状態にあるスレッドの動作を制御します。*wait-policy-var* の初期値は、`PASSIVE` です。

9.3 スレッド数の動的な調整

この実装ではスレッド数を動的に調整する機能が提供されています。動的調整の機能はデフォルトで有効に設定されています。動的調整を無効にするには、`OMP_DYNAMIC` 環境変数を `FALSE`に設定するか、*false* 引数を指定して `omp_set_dynamic()` ルーチンを呼び出します。

スレッドが `parallel` 構文を検出したときにこの実装により提供されるスレッド数は、OpenMP 4.0 仕様のアルゴリズム 2.1 に従って決定されます。システムリソースの不足時などの例外的な状況では、提供されるスレッドの数はアルゴリズム 2.1 で説明されている数よりも少なくなることがあります。

要求された数のスレッドを実装で供給できず、スレッド数の動的調整が有効になっている場合、プログラムの実行は少ない数のスレッドで続行されます。`SUNW_MP_WARN` が `TRUE` に設定されているか、`sunw_mp_register_warn()` の呼び出しによりコールバック関数が登録されている場合は、警告メッセージが表示されます。

要求された数のスレッドを実装で供給できず、スレッド数の動的調整が無効になっている場合は、プログラムでエラーメッセージが発行され、プログラムの実行が停止します。

9.4 OpenMP ループディレクティブ

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

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

9.5 OpenMP 構文

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

`single` — `single` 構文を検出した最初のスレッドが、構文を実行します。

`atomic` — `critical` 構文と名付けられた特別な構文を持つターゲット文または構造体ブロックを挿入することにより、実装によってすべての `atomic` ディレクティブが処理されます。この操作より、プログラム中のすべての atomic 領域間で排他的なアクセスが強制的に行われるようになります。これらの領域が同じメモリーの場所を更新するのか、異なる場所を更新するのかは関係ありません。

9.6 プロセッサバインディング (スレッドアフィニティー)

OpenMP 4.0 仕様では、*プロセッサ*という用語は 1 つ以上の OpenMP スレッドを実行できる実装によって定義されたハードウェアユニットとして定義されています。

この実装では、*プロセッサ*という用語は、Oracle Solaris の `processor_bind`(2) マニュアルページに記述されているように、1 つ以上の OpenMP スレッドをスケジュール、バインド、および実行できる最小のハードウェア実行ユニットとして定義されています。*プロセッサ*の同義語には、CPU、仮想プロセッサ、ハードウェアスレッドがあります。明確にするために、このマニュアルでは、ハードウェアスレッドという用語を一貫して使用しています。

この実装での、`OMP_PLACES` 環境変数で使用される抽象名 (`threads`、`cores`、および`sockets`) の正確な定義は次のとおりです。

- `threads` は、マシン上のハードウェアスレッドを指します。
- `cores` は、マシン上の物理コアを指します。
- `sockets` は、マシン上の物理ソケット (プロセッサチップ) を指します。

詳細は、54 ページの「`OMP_PLACES` および `OMP_PROC_BIND`」を参照してください。OpenMP 4.0 のスレッドのアフィニティーに関連する、Oracle Developer Studio の実装によって定義される動作は次のとおりです。

- *close* スレッドバインドポリシーを使用すると、T > P で P によって T が等分に分割されない場合、場所へのスレッドの割り当ては次のようになります。最初に、P の場所にそれぞれ S = floor(T/P) スレッドが割り当てられます。場所に割り当てられるスレッドの ID は、チーム内のスレッド ID の連続したサブセットになります。次に、最初の T - (P*S) の場所 (親ス

レッドの場所から始まりラップアラウンドする) にそれぞれ追加のスレッドが 1 つ割り当てられます。

- *spread* スレッドバインドポリシーを使用すると、T > P で P によって T が等分に分割されない場合、サブパーティションへのスレッドの割り当ては次のようになります。最初に、P のサブパーティションにそれぞれ S = floor(T/P) スレッドが割り当てられます。サブパーティションに割り当てられるスレッドの ID は、チーム内のスレッド ID の連続したサブセットになります。次に、最初の T - (P*S) のサブパーティション (親スレッドの場所が含まれているサブパーティションから始まりラップアラウンドする) にそれぞれ追加のスレッドが 1 つ割り当てられます。
- アフィニティー要求を処理できない場合、プロセスはゼロ以外のステータスで終了します。
- `OMP_PLACES` 環境変数に指定された数値は、ハードウェアスレッド ID を指しています。
- 抽象名の後ろに数値 *n* を付加することによって *n* 要素の場所のリストを作成する場合、場所のリストは場所のリストが作成されるときにメインスレッドが実行されていたハードウェアスレッドが含まれているリソースで始まる *N* 個の連続したリソースで構成され、使用可能な最後の名前付きリソースに達するとラップアラウンドします。
- マシンで使用可能なリソースよりも多いリソースが要求された場合は、エラーメッセージが発行され、プロセスがゼロ以外のステータスで終了します。リソースに少なくとも 1 つのオンラインのハードウェアスレッドが含まれている場合は、そのリソースを使用できます。
- 実行環境で `OMP_PLACES` リスト内の数値 (明示的に定義されたか、間隔から暗黙的に導出された数値) をターゲットプラットフォーム上のハードウェアスレッドにマップできない場合、または使用できないハードウェアスレッドにマップされる場合は、エラーメッセージが発行され、プロセスがゼロ以外のステータスで終了します。
- `OMP_PLACES` 環境変数が抽象名を使用して定義されている場合、抽象名によって表されたリソースの各ユニットは単一の場所として割り当てられます。割り当てられるユニットの数は、マシン上の使用可能なユニットの合計数よりも小さい値のカウント *n* によって指定できます。Oracle Solaris プラットフォームでは、管理者が `psrset`(1M) を使用してプリエンプティブに予約したハードウェアスレッドは使用可能と見なされません。`OMP_PLACES` によって定義されたセットに使用可能なハードウェアスレッドが残っていない場合は、エラーメッセージが発行され、プロセスがゼロ以外のステータスで終了します。
- parallel 構文のアフィニティー要求を処理できない場合 (OpenMP スレッドをハードウェアスレッドにバインドするシステムコールが失敗した場合など)、結果の動作は未定義です。
- `OMP_PLACES` を使用する場合は、間隔を使用して場所を指定できます。この実装では、間隔によって一連の場所を指定する場合、*length* はそこに含まれる場所の数であり、*stride* はその連続する場所を区切るハードウェアスレッド ID の数であると想定されます。*stride* 値が指定されていない場合は、ユニットの刻み幅が想定されます。

9.7 Fortran の問題

このセクションで説明する問題は Fortran にのみ当てはまります。

9.7.1 THREADPRIVATE ディレクティブ

最初のスレッド以外のスレッドの threadprivate オブジェクト内のデータの値が 2 つの連続した有効な並列領域間で維持されるため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ステータスが「not be currently allocated」になることがあります。

9.7.2 SHARED 節

共有変数を組み込み以外の手続きに渡すと、手続きで参照する前に共有変数の値が一時ストレージにコピーされ、手続きでの参照後に一時ストレージから実際の引数ストレージに戻されることがあります。一時ストレージが使用されることがあるのは、実際の引数に関して次の 3 つの条件を満たしている場合のみです。

1. 実際の引数が次のいずれかの引数である。
   - 共有変数
   - 共有変数のサブオブジェクト
   - 共有変数と関連づけられたオブジェクト
   - 共有変数のサブオブジェクトと関連づけられたオブジェクト
2. 実際の引数が次のいずれかの引数である。
   - 部分配列
   - ベクトル添字のある部分配列
   - assumed-shape 配列
   - ポインタ配列
3. 実際の引数に関連づけられたダミー引数が、形状明示配列または形状引き継ぎ配列である。

9.7.3 実行時ライブラリの定義

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

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

索引

な

は

ま

ら

A

D

E

F

G

I

L

M

O

P

S

T

X