Linux* 版インテル® Fortran コンパイラ ユーザーズ・ガイド
Vol II: アプリケーションの最適化

免責条項

資料番号: 253260-002J

目次

アプリケーションの最適化: 概要

本書は全 2 巻からなる『インテル® Fortran コンパイラ・ユーザーズ・ガイド』の第 2 巻です。

次のトピックについて説明します:

インテル Fortran コンパイラを使ったハイパフォーマンスを目的としたプログラミング:

- データ型とアライメントの設定
- 配列の効率的な使用
- 入出力性能の向上
- ランタイム効率の向上
- Itanium® ベース・システムにおける組込み関数の使用
- インテル® アーキテクチャのコーディング・ガイドライン

アプリケーションの分析と時間測定:

- インテルが提供するパフォーマンス分析ツールの使用
- アプリケーションの時間測定

インテル Fortran コンパイラの最適化の実装:

- コンパイル処理の最適化
- 効率的なコンパイル
- 自動割り当てと確認のスタックオプション
- アライメント・オプション
- シンボルの可視属性オプション
- 異なるアプリケーション・タイプの最適化オプション
- 浮動小数点演算の最適化
- 特定のプロセッサの最適化
- プロシージャ間の最適化
- プロファイルに基づく最適化
- 高水準言語の最適化 (HLO)

インテル Fortran による並列プログラム

- ベクトル化の自動処理 (IA-32 のみ)
- 自動並列化:
- OpenMP* による並列化
- マルチスレッドのデバッグのプログラム

最適化サポート機能:

- コンパイラ・ディレクティブ
- 最適化とデバッグ
- 最適化機構レポートの作成

本リリースの新機能については、Vol I の「本リリースの新機能」を参照してください。また、製品リリースノートも参照してください。

本書の使い方

本ユーザーズ・ガイドでは、アプリケーションのパフォーマンスを向上するためのインテル® Fortran コンパイラの使い方を説明します。

インテル Fortran コンパイラのさまざまな最適化を使用して、アプリケーションのパフォーマンスを向上することができます。最適化の各機能は、本書の各項で説明するオプションのセットによって実行されます。

コンパイラのコマンドライン・オプションで実行される最適化に加えて、ディレクティブ、組込み関数、ランタイム・ライブラリ・ルーチンおよびユーティリティなどのアプリケーションのパフォーマンスを向上する機能が含まれています。これらの機能については、「最適化サポート機能」セクションを参照してください。

> **注**
> 本書では、対象となるアーキテクチャごとに情報や命令がどのように適用されるかを説明しています。特定のアーキテクチャが指定されていない場合、その説明はサポートされているすべてのアーキテクチャに適用されます。

本書は、Fortran 標準プログラミング言語とインテル® プロセッサ・ アーキテクチャについてよく理解している読者を対象としています。また、ホスト・コンピュータのオペレーティング・システムにも精通している必要があります。

表記規則

本書では、次の表記規則を使用しています。

インテル Fortran	Windows* 版インテル Fortran コンパイラ製品および Linux* 版インテル Fortran コンパイラ製品でサポートされている共通コンパイラ言語の名前を示します。
Fortran 95	これらの用語は、Fortran 言語のバージョンを表します。デフォ

Fortran 90 Fortran 77	ルトは "Fortran" で、すべてのバージョンを対象とします。
THIS TYPE STYLE	文、キーワード、およびディレクティブを、標準フォントを使用してすべて大文字で表記します。 例:「USE 文を追加します」
This type style	メニュー名、メニュー項目、ボタン名、ダイアログ・ウィンドウ名、およびその他のユーザ・インターフェイス項目を、太字の標準テキストで表記します。
[ファイル] > [開く]	(>) 記号で結合したメニュー名およびメニュー項目は、一連の動作を示します。例えば、「**[ファイル] > [開く]** をクリックします」の場合、**[ファイル]** メニュー内の **[開く]** をクリックし、この動作を実行します。
`ifort`	本書の例では、次の規則に従ってコンパイラ・コマンドが使用されます: 各アーキテクチャ間でコンパイラ・コマンドの使用方法に違いがない場合は、1 つのコマンドだけを例に挙げます。使用方法に違いがある場合は、各アーキテクチャ別のコマンドを例に挙げます。
`This type style`	標準のモノスペース・フォントで表記されているテキストは、構文要素、予約語、キーワード、ファイル名、変数、またはコード例のいずれかを表します。 特に大文字が必要でない限り、小文字で表記されます。
`This type style`	太字のモノスペース・フォントで表記されているテキストは、ユーザ入力を示します。例: ユーザが入力するコマンドや値
`This type style`	斜体のモノスペース・フォントで表記されているテキストは、ユーザの入力個所を示します。 また、新しい用語が使われた場合にも、 その用語を斜体で表記します。
`[options]`	大括弧で囲まれている項目は、省略可能です。一部の例では、大括弧は配列を示すためにも使われています。
`{value | value}`	中括弧と縦線は、項目の選択を示します。 すべての項目が大括弧で囲まれていない限りは、項目から 1 つを選択しなければなりません。
`...`	項目の次にくる横方向の反復記号 (3 個のドット) は、反復記号の前の項目が繰り返されていることを示します。コード例では、横方向の反復記号は、一部の文が省略されていることを意味します。
Linux* システム	用語や名前の終わりにあるアスタリスク (*) は、他社の製品登録商標を示します。

ハイパフォーマンスを目的としたプログラミング: 概要

このセクションでは、次の情報を提供します:

- プログラミング・ガイドライン
  インテル® アーキテクチャの機能を活用するための特殊なコーディング手法と、アプリケーションのパフォーマンスを向上するプログラミングのガイドラインについて説明します。
- アプリケーションの分析と時間測定
  インテルが提供するパフォーマンス分析ツールの使用方法と、プログラムの実行時間を測定し、問題箇所について情報を収集する方法について説明します。

プログラミングのガイドライン

データ型とアライメントの設定

データのアライメントを考慮するべき変数には、次のものがあります:

- 動的に割り当てられる変数
- データ構造体の構成要素
- グローバルまたはローカル変数
- スタックに渡されるパラメータ

最高のパフォーマンスを得るには、データを次のようにアライメントします。

- あらゆるアドレスにおける 8 ビットデータをアライメントします。
- アライメントされた 4 バイトワード内に入る 16 ビットデータをアライメントします。
- 32 ビットデータをベースアドレスが 4 の倍数になるようにアライメントします。
- 64 ビットデータをベースアドレスが 8 の倍数になるようにアライメントします。
- 80 ビットデータをベースアドレスが 16 の倍数になるようにアライメントします。
- 128 ビットデータをベースアドレスが 16 の倍数になるようにアライメントします。

アライメントされないデータの原因と自然なアライメントを行う方法

最適な性能を得るため、データが自然にアライメントされるように注意してください。自然境界とは、データ項目のサイズの倍数であるメモリアドレスのことです。例えば、自然境界上にアライメントされた REAL(KIND=8) のデータ項目は、8 の倍数のアドレスを持ちます。配列については、すべての要素がこのようにアライメントされているときに、配列がアライメントされているといいます。

開始アドレスが自然境界上にあるデータ項目は、*「自然にアライメントされている」*といいます。自然境界にアライメントされていないデータは*「アライメントされていないデータ」*と呼びます。

インテル® Fortran コンパイラは、可能ならば各データ項目を自然にアライメントしますが、一部の Fortran 文では、データがアライメントされないことがあります。

コマンドライン・オプション `-align` を使うと、データを自然にアライメントすることができますが、共通ブロック、派生型、レコード構造体中のデータ項目のデータ宣言を検証し、順序を変更することを検討してください。

- データ宣言の順序とサイズを慎重に指定して、データが自然にアライメントされるようにする
- 最初に最も大きな数値項目を指定し、次に小さな数値項目を、最後に非数値（文字）の項目を指定する

次のような文は、アライメントされていないデータの原因となります。

- 共通ブロック（COMMON 文）

  COMMON 文中の変数の順は、それら変数の格納の順序を決定します。共通ブロック中のデータ項目が自然にアライメントされることが確実ではない限り、使用されるデータの最大値に応じて、`-align commons` オプションか `-align dcommons` オプションを指定してください。「アライメント・オプション」を参照してください。

- 派生型（ユーザ定義）データ

  派生型データの構成要素は、TYPE 文の後に宣言されます。

  データに派生型データ構造体が含まれている場合、派生型データ中のデータ項目が自然にアライメントされることが確実ではない限り、`-align records` オプションを指定してください。

SEQUENCE 文を省略すると、-align records オプションはすべてのデータ項目を自然にアライメントします。

SEQUENCE 文を指定すると、-align records オプションは、-align sequence オプションを指定しない限り、アライメントされていないデータの発生を防ぐのに必要なパディングを追加できなくなります (データ項目はパックされます)。SEQUENCE 文を使用する場合、データ宣言の順序を、すべてのデータ項目が自然にアライメントされるように指定する必要があります。

- レコード構造体 (RECORD と STRUCTURE 文)

  インテル Fortran レコード構造体には、通常、複数のデータ項目が含まれます。STRUCTURE 文中の変数の順序は、それら変数の格納の順序を決定します。RECORD 文はレコード構造体の名前を指定します。レコード構造体は、インテル Fortran 言語拡張です。

  データにレコード構造体が含まれている場合、レコード構造体のデータ項目が自然にアライメントされることが確実ではない限り、-align records オプションを指定してください。

- EQUIVALENCE 文

  EQUIVALENCE 文は、アライメントされないデータや、自然境界にまたがるデータを生じさせることがあります。詳細は、『Intel® Fortran Language Reference』(英語) マニュアルを参照してください。

共通ブロック、派生型構造体、またはレコード構造体中にアライメントされないデータが生じるのを防ぐには、次のいずれか、または両方の手段を取ってください。

- 新規のプログラムや、ソースコードの宣言を簡単に変更できるプログラムでは、データ宣言の順序を慎重に計画します。例えば、COMMON 文中では、数値データが大きいものから小さいものに配置され、最後に文字データが配置されるように変数を並べます (以下の「アライメントされないデータの発生を防ぐためのデータ宣言の順序」を参照)。
- ソースコードの変更が簡単に行えない既存のプログラムや、派生型またはレコード構造体を含む配列要素の場合、コマンドライン・オプションを使って、必要に応じて空白をパディングすることで数値データがアライメントされるように、コンパイラに要求することができます。

アライメントされないデータが生じるその他の原因としては、アライメントされていない実引数や、派生型構造体またはレコード構造体を含む配列などがあります。

- プログラム・ユニットの外部からの実引数が自然にアライメントされていないと、アライメントされないデータ参照が発生します。インテル Fortran は、渡されたすべての引数が自然にアライメントされていると仮定します。また、コンパイル時には、プログラムの実行中に実引数を通して渡されるデータに関して何の情報も持っていません。
- 個々の配列要素に派生型構造体やレコード構造体が含まれる配列の場合、配列要素のサイズにより、一部の要素 (ただし先頭の要素以外) がアライメントされていない境界から始まることがあります。
- SEQUENCE 文やレコード構造体がなく、派生型構造体中でデータ項目が自然にアライメントされている場合でも、配列要素のサイズにより一部の配列要素がアライメントされなくなるのを防ぐために、`-align records` オプションを使って必要なパディングを行わなければならないことがあります。
- `-align norecords` を指定するか、`-align records` なしで `-vms` を指定すると、配列要素の間ではパディング・バイトが追加されません。個々の配列要素が SEQUENCE 文付きの派生型構造体を含む場合、どのような Fortran コマンド・オプションが指定されていても、配列要素はパディング・バイトなしにパックされます。この場合、一部の要素がアライメントされなくなります。
- `-align records` オプションが有効な場合、個々の配列要素に対してコンパイラが追加するパディング・バイト数は、構造体中の最も大きなデータ項目のサイズに依存します。コンパイラは、SEQUENCE 文のない派生型構造体、またはレコード構造体中の最も大きなデータ項目の倍数として、配列要素の大きさを計算します。その後、コンパイラは適切な数のパディング・バイトを追加します。例えば、構造体が 8 バイトの浮動小数点数の後に 3 バイトの文字変数を含む場合、各要素は 5 バイトのパディングを含むことになります (16 が 8 の倍数であるため)。しかし、構造体に 4 バイトの浮動小数点数が 1 つ、4 バイト整数が 1 つ、さらに 3 バイトの文字変数が 1 つ含まれていると、各要素は 1 バイトのパディングを含むことになります (12 が 4 の倍数であるため)。

非効率的なアライメントされていないデータの検証

コンパイルの際に、インテル Fortran コンパイラは、できるだけ多くのデータを自然にアライメントします。アライメントされないデータが生じるような例外的な状況については、上記で説明しています。

アライメントされていないデータはランタイム性能を低下させる可能性があるので、次の対処策を取ることをお勧めします。

- 共通ブロック、派生型構造体、またはレコード構造体中のデータ宣言を慎重に検証して、すべてのデータ項目が自然にアライメントされることを確認します (データ宣言については、下記のサブセクションを参照)。データ宣言の格納にモジュールを使うことで、この種のデータのアライメントに一貫性を持たせることができます。

- EQUIVALENCE 文を使わないようにします。使う場合は、アライメントされないデータや、自然境界にまたがるデータを生じさせないようにします。
- プログラム・ユニットの外部から渡される引数が、自然にアライメントされていることを確認します。
- 少なくとも 1 つの派生型構造体かレコード構造体を含む配列要素のサイズを検証して、配列要素がアライメントされた境界から始まるようになっていることを確認します (前のサブセクションを参照)。

アライメントされていないデータをレポートするには、2 つの方法があります:

- コンパイルの際、(すべての警告を抑制する `-warn noalignments` (または `-W0`) オプションを指定していなければ) アライメントされないことがわかっているすべてのデータ項目について、警告メッセージが発行されます。
- プログラムの実行の際、アライメントされていないことが検知されたすべてのデータについて、警告メッセージが発行されます。メッセージには、アライメントされていないアドレスが含まれます。

以下のランタイム・メッセージについて考えてみます:

```
Unaligned access pid=24821 <a.out> va=140000154,
pc=3ff80805d60, ra=1200017bc
```

このメッセージでは次のことを示しています。

- アライメントされていないデータ (プログラム・カウンタ) へアクセスする文は、3ff80805d60 にあります。
- アライメントされていないデータは、140000154 のアドレスにあります。

アライメントされないデータの発生を防ぐためのデータ宣言の順序

新規のプログラムや、ソースコードの宣言を簡単に変更できるプログラムでは、データ宣言の順序を慎重に計画し、共通ブロック、派生型構造体、レコード構造体、または EQUIVALENCE 文によって等価にされたデータ項目が自然にアライメントされるようにします。

アライメントされないデータの発生を防ぐには、次の規則を使用します。

- 必ず最大サイズの数値データ項目を最初に定義するようにします。
- データに文字データと数値データが混在している場合、数値データを先に指定します。

- アライメントされないデータの前に、適切なサイズの小さなデータ項目 (またはパディング) を追加して、それ以降のデータが自然にアライメントされるようにします。

データの宣言時、KIND パラメータを指定するなど、明示的に長さを宣言することを検討してください。例えば、INTEGER より INTEGER(KIND=4) (または INTEGER(4)) を指定します。デフォルトのサイズを使用する場合 (INTEGER、LOGICAL、COMPLEX、REAL など)、`-integer_size{16|32|64}` と `-real_size{32|64|128}` コンパイラ・オプションは、個々のフィールドのデータ宣言サイズを変更できることに注意してください。つまり、慎重に計画されたデータ宣言の順序のデータ・アライメントが変えられてしまう可能性があります。

データ宣言のガイドラインに従うことで、データを自然にアライメントさせるためのパディング・バイトを追加する `-align` *keyword* オプションの必要性を最小限に抑えることができます。`-align` *keyword* オプションが必要な場合でも、データ宣言のガイドラインに従うと、コンパイラが追加するパディング・バイト数を最小限に抑えることができます。

共通ブロック中のデータ項目の配置

`common` 文中のデータ項目の順序は、データ項目が格納される順を決定します。`x` という名前の共通ブロックが次のように宣言されているとします。

```
logical (kind=2) flag
integer          iarry i(3)
character(len=5) name ch
common /x/ flag, iarry_i(3), name_ch
```

図 1-1 に示すように、適切な Fortran コマンド・オプションを省略すると、共通ブロックには `iarry_i` の第 1 配列要素の先頭から、アライメントされていないデータ項目が含まれるようになります。

図 1-1 アライメントされていないデータを含む共通ブロック

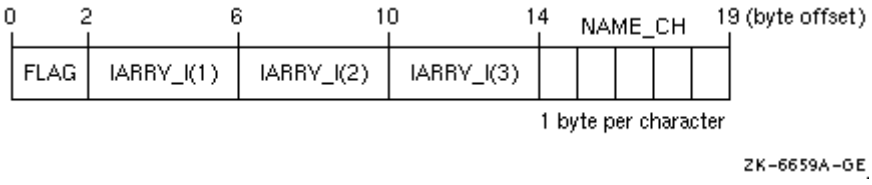
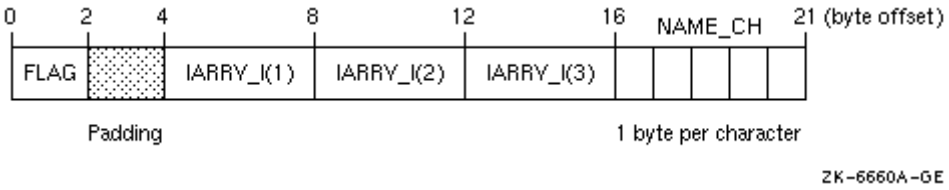


図 1-2 に示すように、共通ブロックを使用しているプログラム・ユニットを `-align commons` オプションを付けてコンパイルすると、データ項目は自然にアライメントされます。

図 1-2 自然にアライメントされたデータを含む共通ブロック



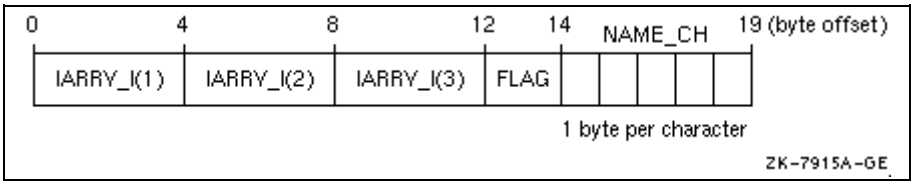
共通ブロック x は、大きさが 32 ビット以下のデータ項目を含むため、-align commons オプションを指定するようにします。共通ブロックが 32 ビットよりも大きくなる可能性のあるデータ項目 (REAL (KIND=8) のデータなど) を含む場合、-align commons オプションを使用します。

共通ブロックデータを使用するソースファイルを簡単に変更できる場合は、COMMON 文中の数値変数をサイズの大きい順に定義して、文字変数を最後に置きます。これは、移植性を高め、パディングなしにデータ項目を自然にアライメントし、コマンドライン・オプション -align commons または -aligndcommons オプションを必要としません。

```
logical (kind=2) flag
integer          iarry_i(3)
character(len=5) name_ch
common /x/ iarry_i(3), flag, name_ch
```

図 1-3 に示すように、変数をサイズの大きい順に並べ、文字データを最後に置けば、データ項目は自然にアライメントされます。

図 1-3 自然にアライメントされるように並べ替えられたデータを含む共通ブロック



共通ブロックデータを使用するソースファイルをすべて変更または作成するときには、宣言に一貫性を持たせるために、共通ブロックデータ宣言を 1 つのモジュールにまとめることを検討してください。互換性の理由から共通ブロックが不要な場合 (ファイルへの格納や、Fortran 77 に使用する場合) には、共通ブロックを使わずに、データ宣言をモジュールに入れることができます。

派生型データのデータ項目のアライメント

共通ブロックと同様に、派生型構造体には複数のデータ項目が含まれることがあります。

派生型構造体中のデータ項目コンポーネントは、SEQUENCE 文と Fortran オプションの使用に関連するいくつかの例外を除いて、最大 64 ビットの境界上に自然にアライメントされます。例外については、「アライメント・オプション」を参照してください。

インテル Visual Fortran は、派生型データを、次のように値の線形な並びとして格納します。

- SEQUENCE 文を指定すると、最初のデータ項目は最初の格納位置に、最後のデータ項目は最後の格納位置に格納されます。個々のデータ項目は宣言された順番に格納されます。Fortran オプションはアライメントされていないデータには影響を与えないので、データを自然にアライメントするためにはデータ宣言を慎重に指定する必要があります。`-align sequence` オプションは、SEQUENCE 派生型のデータ項目を自然境界にアライメントします。
- SEQUENCE 文を省略すると、`-align norecords` オプションが指定されていない限り、インテル Fortran は、データ項目の構成要素を自然にアライメントするために必要なパディング・バイトを追加します。

次に示す例は、派生型 `PART_DT` の配列 `CATALOG_SPRING` の宣言です。

```
module data defs
type part dt
integer            identifier
real               weight
character(len=15)  description
end type part dt
type(part dt) catalog spring(30)
.
.
.
end module data defs
```

配列の効率的な使用

ここで説明する配列アクセス手法の多くは、インテル® Fortran のループ変換最適化により自動的に適用されます。配列の使用法によって、ランタイム・パフォーマンスを向上することができます。

- 配列へのアクセスは、配列全体、または配列の大部分への連続的なアクセスが行われたときに、最も高速になります。分散した配列要素を何回も操作する

よりも、配列全体または配列の大部分にアクセスする 1、2 回の配列操作を実行するようにします。配列アクセスは、明示的なループを使用するよりも、次の行のように配列変数 a のすべての要素をインクリメントするような基本的な配列操作を行うようにします。

```
a = a + 1
```

配列の読み取りや書き出しを行うときには、配列名を使用し、個々の要素番号を指定する DO ループや暗黙的な DO ループは使用しないようにします。
Fortran 95/90 の配列構文では、式中で配列名を使うことで、配列全体を参照することができます。次に例を示します。

```
real :: a(100,100)
a = 0.0
a = a + 1        !Increment all elements

          !of a by 1
.
.
.
write (8) a      !Fast whole array use
```

同様に派生型配列構造体要素は、次のように使用できます。

```
type x
integer a(5)
end type x
.
.
.
type (x) z
write (8)z%a     !Fast array structure
          ! component use
```

- 多次元配列が正しい配列構文で参照されていて、Fortran の *自然昇順の記憶領域順*である*列優先順*でトラバースされていることを確認します。列優先順では、一番左の添字が最も急速に 1 ずつ変化していきます。配列全体のアクセスには列優先順が使用されます。
  C によって行われるような一番右の添字が最も急速に変化する*行優先順*の使用は避けてください。
  次の例は、J ループを最も内側のループとして、2 次元配列を参照するネストされた DO ループを示します。

```
integer x(3,5), y(3,5), i, j
y = 0
do i=1,3                !I outer loop varies slowest
do j=1,5                !J inner loop varies fastest
```

```
x (i,j) = y(i,j) + 1   !Inefficient row-major storage
order
end do                 !(rightmost subscript varies
fastest)
end do
.
.
.
end program
```

最も急速に変化する j が、式 x (i,j) の第 2 配列添字であるため、配列アクセスは行優先順で行われます。
配列アクセスを自然な列優先順で行うためには、配列のアルゴリズムと変更されるデータを確認します。配列 x と y を使っている場合、最も内側のループ変数が一番左の配列次元に対応するように DO ループのネスト順序を変更することで、配列アクセスを自然な列優先順で行うことができます。

```
integer x(3,5), y(3,5), i, j
y = 0
do j=1,5               !J outer loop varies slowest
do i=1,3               !I inner loop varies fastest
x (i,j) = y(i,j) + 1   !Efficient column-major storage
order
end do                 !(leftmost subscript varies
fastest)
end do
.
.
.
end program
```

インテル Fortran の配列全体へのアクセス (x = y + 1) では、効率的な列優先順が使われます。ただし、そのアプリケーションで J を最も急速に変化させる必要がある場合や、結果を変えることなくループの順序を変更することができない場合、アプリケーション・プログラムを変更して、配列次元の順序を変えることを検討してください。プログラムを変更するときには、次の順序を変える必要があります。

- 配列 x(5,3) と y(5,3) の宣言中の次元
- do ループ内の x(j,i) と y(j,i) の代入
- 配列 x と y に対するその他すべての参照

ここでは、J が最も内側のループの場合、元の DO ループのネストが使われます。

```
integer x(3,5), y(3,5), i, j
y = 0
do i=1,3               !I outer loop varies slowest
```

```
do j=1,5                !J inner loop varies fastest
x (j,i) = y(j,i) + 1    !Efficient column-major storage
order
end do                  !(leftmost subscript varies
fastest)
end do
.
.
.
end program
```

多次元配列へのアクセスを行優先順 (C など) またはランダムな順で行うように書かれたコードでは、一般に CPU のメモリ・キャッシュが非効率的に使用されます。レコード I/O 操作時の自然な記憶順についての詳細は、「入出力性能の向上」を参照してください。

- 独自のプロシージャを作成するよりも、利用可能な Fortran 95/90 配列組込みプロシージャを使用します。

  可能な限り同じタスクを実行するために、独自のルーチンを作成するよりも、Fortran 95/90 配列組込みプロシージャを使用するようにします。Fortran 95/90 配列組込みプロシージャは、さまざまなインテル Fortran ランタイム・コンポーネントで効率的に使用できるように設計されています。

  また、標準準拠の配列組込み関数を使用することで、プログラムの移植性を高めることができます。

- 配列要素へのアクセスが非連続に行われる多次元配列では、一番左の配列次元が 2 の累乗 (256、512 など) にならないようにします。

  キャッシュ・サイズは 2 の累乗であるため、配列次元が同様に 2 の累乗になっていると、配列アクセスが非連続な場合にキャッシュの使用効率が低下することがあります。キャッシュ・サイズが一番左の次元の倍数である場合、プログラムによるキャッシュの使用は非効率的になります。ただし、これは連続的でシーケンシャルなアクセスや配列全体へのアクセスには適用されません。

  回避策の 1 つとして、次元を増やして、いくつかの使用されない要素を追加し、一番左の次元を実際に必要な大きさより大きくします。例えば、A の左端の次元を 512 から 520 へ増やすことで、キャッシュをより効率的に使用できます。

```
 real a(512, 100)
do i= 2,511
do j = 2,99
a(i,j)=(a(i+1,j-1) + a(i-1, j+1)) * 0.5
end do
end do
```

このコードでは、配列 a の一番左の次元は 512 で、2 の累乗です。最も内側のループは一番右の次元にアクセスするので (行優先)、非効率なアクセスになります。a の一番左の次元を 520 (`real a (520,100)`) に増やすことで、実際には使われない要素が生じる代わりに、ループのパフォーマンスが向上します。

ループ・インデックス変数 I および J は、計算で使われるため、do ループのネスト順を変更すると結果が変わってしまいます。

配列およびそれらのデータ宣言文については、『Intel ® Fortran Language Reference』(英語) マニュアルを参照してください。

配列引数の効率的な渡し方

Fortran の配列引数には、2 つの一般的な形式があります。

- FORTRAN 77 で使われる明示形状配列

  これらの配列は次元数と範囲が固定されており、コンパイル時に認識されます。形状無指定ではないその他の仮引数 (受け取り側) 配列 (大きさ引継ぎ配列など) は、明示形状配列引数の形式に含めることができます。

- Fortran 95/90 で導入された形状無指定配列

  形状無指定配列の形式には、配列ポインタと割付け配列があります。形状引継ぎ配列引数は通常、形状無指定配列引数の受け渡しについての規則に従います。

配列を引数として渡す場合、配列の開始 (ベース) アドレスか、配列記述子のアドレスを渡します。

- 明示形状配列 (または大きさ引継ぎ配列) を使用して配列を受け取る場合は、配列の開始アドレスが渡されます。
- 形状無指定配列または形状引継ぎ配列を使用して配列を受け取る場合は、配列記述子のアドレスが渡されます (配列記述子はコンパイラが作成します)。

形状引継ぎ配列または配列ポインタを明示形状配列に渡すと、ランタイム・パフォーマンスが低下することがあります。これは、コンパイラが配列全体に対して一時的な配列を作成しなければならないためです。一時的な配列が作成されるのは、渡される配列が連続していない可能性があり、受け取り側の (形状明示) 配列が連続した配列を必要とするからです。一時的な配列が作成される場合、渡される配列のサイズによって、ランタイム・パフォーマンスに与える影響の大きさが決まります。

下記の表に、配列型の組み合わせによって行われる処理についての要約を示します。ランタイム・パフォーマンスの非効率性は、配列のサイズに依存します。

実引数配列の形式 (1 つを選択)	実引数配列の形式 (1 つを選択)	
	明示形状配列	形状無指定と形状引継ぎ配列
明示形状配列	この組み合わせを使用した結果: きわめて効率的。一時的な配列は使用しません。配列記述子は渡しません。インターフェイス・ブロックはオプションです。	この組み合わせを使用した結果: 効率的。形状引継ぎ配列でのみ可能です (形状無指定配列では不可)。一時的な配列は使用しません。配列記述子を渡します。インターフェイス・ブロックは必須です。
形状無指定と形状引継ぎ配列	この組み合わせを使用した結果: 割付け配列の受け渡しでは非常に効率的。一時的な配列は使用しません。配列記述子は渡しません。インターフェイス・ブロックはオプションです。 割付け配列を渡さない場合は、非効率的です。可能な限り、割付け配列を使用するようにしてくださ い。 一時的な配列を使用します。配列記述子は渡しません。インターフェイス・ブロックはオプションです。	この組み合わせを使用した結果: 効率的。仮引数として形状引継ぎまたは配列ポインタが必要です。一時的な配列は使用しません。配列記述子を渡します。インターフェイス・ブロックは必須です。

入出力性能の向上

全体的な入出力性能を向上させることで、デバイス入出力と実際の CPU 時間の両方を最小限に抑えることができます。ここに示すテクニックは、多くのアプリケーションの性能を大幅に向上させることができます。

入出力文は、実行プログラムで最も遅いプロセスに属し、プログラム実行の最大速度を制限します。一部のプログラムでは、入出力が実行時性能の向上を妨げるボトルネックとなっています。入出力の問題を解消するための鍵は、入出力に関連する CPU 時間と入出力デバイス時間を減らすことです。

この問題は、以下の 1 つまたは複数の原因により発生します。

- CPU 時間が大幅に削減されたのにもかかわらず、入出力時間がそれに対応して改善されていない場合
- 次のようなコーディング項目を再考してみてください。
  - データの不要な書式指定、およびその他の CPU 集中型の処理
  - 中間結果の不要な転送
  - 少量データの非効率的な転送
  - アプリケーション要件

コーディングを改善することで、実際のデバイス入出力と実際の CPU 時間を最小限に抑えることができます。

インテルは、デバイス入出力の遅延を最小限に抑えるなど、システム全体の問題に対応するソフトウェア・ソリューションを提供します。

書式付きファイルの代わりに書式なしファイルを使用する

書式なしファイルを可能な限り使用してください。数値データの書式なし入出力は、書式付き入出力よりも効率的で高精度です。ネイティブ書式なしデータは、転送の際に変更する必要がなく、外部ファイルにおける占有空間が小さくなります。

一方、書式付きファイルにデータを書き出すときには、書式付きデータを出力のために文字列に変換しなければならず、1 回の操作で転送できるデータ量が減ります。また、書式付きデータを再び 2 進形式として読み取ると、精度が失われることがあります。

次に示す、配列 `A(25,25)` を書き出す文は、`S1` の方が `S2` よりも効率的です。

```
S1          WRITE (7) A

S2          WRITE (7,100) A
100    FORMAT (25(' ',25F5.21))
```

書式付きデータ・ファイルは、他のシステムへの移植がより簡単ですが、インテル® Fortran は書式なしデータをいくつかの書式に変換することができます (「リトル・エンディアン – ビッグ・エンディアンの変換」を参照)。

配列または文字列全体を書き出す

不要なオーバーヘッドをなくすために、個々の要素を何度も書き出すのではなく、配列または文字列全体を一度に書き出すようにしてください。入出力リスト中の各項目

は、それぞれ独自の呼び出し手順を生成します。この処理のオーバーヘッドは、DO 形ループで最も大きくなります。配列全体を参照するときには、DO 形ループを使う代わりに配列名 (Fortran 配列構文) を使用してください。

自然な記憶順で配列データを書き出す

可能な限り自然な昇順記憶を行ってください。自然な記憶順とは、一番左の添字が最も急速に 1 ずつ変化する列優先順です。(「配列の効率的なアクセス」を参照。) プログラムがこれ以外の順序でデータの読み書きを行っていると、効率的なブロック移動ができなくなります。

全体配列を書き出さない場合、この自然な記憶順が最も効率的な順序です。

不自然な記憶順を使用しなければならないとき、場合によっては、データをメモリに転送し、データの順序を変更してから入出力操作を実行した方が効率的なことがあります。

中間結果にメモリを使用する

中間結果を周辺機器のファイルではなくメモリに格納することで、性能が向上することがあります。逆に中間的な格納領域を使うことがかえって不利になる状況の 1 つは、データがシステムの物理メモリと比べて格段に大きい場合です。このような場合、ページ・フォルトが頻繁に起こって、仮想メモリの性能を大幅に低下させる可能性があります。

ページング・フォルトによるシステムの CPU 性能の低下を特に憂慮する場合、コードにより使用されるファイルの保持には、メモリ・ファイル・システム (mfs) 仮想ディスクを使用することを検討してください。

DO 形ループコラプスを有効にする

DO 形ループコラプスは、入出力処理の大きなオーバーヘッドを軽減します。通常は、入出力リスト中の個々の要素が、インテル Fortran ランタイム・ライブラリ (RTL) への呼び出しを個別に生成します。これらの呼び出し処理が、DO 形ループで最も大きなオーバーヘッドとなります。

インテル Fortran は、最大 7 レベルまでネストされた DO 形ループを、最適化されたランタイム・ライブラリ入出力ルーチンへの 1 回の呼び出しに置き換えることで、DO 形ループ中での呼び出し回数を減らします。このルーチンは多数の入出力要素を一度に転送することができます。

ループコラプスは、書式付き入出力と書式なし入出力のどちらでも可能ですが、いくつかの条件が満たされている必要があります。

- 制御変数は整数でなければなりません。制御変数は、仮引数であったり、EQUIVALENCE または VOLATILE 文に含まれていてはなりません。インテル Fortran は、制御変数が実行時に予期しない形で変更されないことを判断できなければなりません。
- 書式には可変書式の式を使用してはいけません。

VOLATILE 属性と文については、『Intel® Fortran Language Reference』(英語) を参照してください。

ループの最適化については、「ループ変換」、「ループのアンロールのサポート」、「異なるアプリケーション・タイプの最適化の概要」を参照してください。

可変書式の式を使用する

可変書式の式 (インテル Fortran 拡張) は実行時の書式指定とほぼ同じ柔軟性を備えていますが、コンパイラが入出力書式を実行時に解析しなくて済むので、より効率的です。実行時には、少量の処理を行い、実際にデータ転送を行うだけで済みます。

一方、実行時の書式指定は性能を大幅に低下させる可能性があります。例えば、次の文では、書式指定が実行時にではなくコンパイル時に一度だけ行われる S1 の方が S2 よりも効率的です。

```
S1          WRITE (6,400) (A(I), I=1,N)
400   FORMAT (1X, <N> F5.2)
    .
    .
    .
S2          WRITE (CHFMT,500) '(1X,',N,'F5.2)'
500   FORMAT (A,I3,A)
WRITE (6,FMT=CHFMT) (A(I), I=1,N)
```

レコード・バッファとディスク入出力を効率的に使用する

読み取りまたは書き出しが行われるレコードは、ユーザのプログラム・バッファと、インテル Fortran RTL によってファイルが開かれるときに設定される 1 つまたは複数のディスクブロック入出力バッファの間で転送されます。読み取りまたは書き出されるレコードがきわめて大きい場合を除き、ディスク読み書きの際には、ディスクブロック入出力バッファに複数の論理レコードをバッファして、物理ディスク入出力の回数が最小限に抑えられます。

ディスクブロック物理入出力バッファの大きさは、OPEN 文の BLOCKSIZE 指定子を使って指定することができます。デフォルトのサイズは `fstat(2)` により取得できます。OPEN 文で BLOCKSIZE 指定子を省略すると、ファイルが置かれているデバイスタイプで最適な入出力が行われる大きさに設定されます (ネットワーク・アクセスを除く)。

OPEN 文の BUFFERCOUNT 指定子は、入出力バッファの数を指定します。BUFFERCOUNT のデフォルト値は 1 です。1 回のディスク入出力で読み取られるデータの量を増やして 入出力性能を向上させる実験を行うときには、BLOCKSIZE 値ではなく BUFFERCOUNT 値を増やしてください。

OPEN 文に BLOCKSIZE および BUFFERCOUNT 指定子が含まれている場合、内部バッファの大きさ (バイト単位) は、これらの指定子の積となります。`OPEN` 文にこれらの指定子がない場合、デフォルトの内部バッファサイズは 8192 バイトになります。この内部バッファは、最も大きなレコードを収容できるように拡張はされますが、縮小されることはありません。

Fortran ランタイム・システムのデフォルトでは、バッファリングを使用しないディスク書き出しを行います。つまり、デフォルトでは、バッファにためて、後でディスクに書き出す代わりに、各レコードが書かれるとすぐにディスクに書き出されます。

バッファリング付きの書き出しを有効にするには (つまり、バッファがディスクに書き出されるまでディスクデバイスが内部バッファにためられようにするには)、次のいずれかを使用します:

- OPEN 文の BUFFERED 指定子
- `-assume:buffered_io` コマンドライン・オプション
- `FORT_BUFFERED` ランタイム環境変数

OPEN 文の BUFFERED 指定子は、`-assume buffered_io` オプションよりも優先されます。どちらも設定されなかった場合 (デフォルト)、`FORT_BUFFERED` 環境変数が実行時に使用されます。

OPEN 文の BUFFERED 指定子は、特定の論理ユニットに適用されます。これに対して、
`-assume [no]buffered_io` オプションおよび `FORT_BUFFERED` 環境変数は、すべての Fortran ユニットに適用されます。

通常、バッファリング付きの書き込みを使用すると、より大きなデータブロックがディスクに書き込まれ、書き込み回数が減るため、ディスク I/O の効率が向上します。しかし、バッファリングされた書き出しの使用時にシステム障害が起こると、その時点でディスクに書き出されていなかったレコードが失われます。(これらのレコードは、デ

フォルトのバッファリングを使用しない書き出しではディスクに書き出されていたはずです。)

ネットワークを介して入出力アクセスを行う場合、ネットワーク上で送られるネットワーク・データのブロックサイズがアプリケーションの効率に影響することに注意してください。ネットワーク・データを読み取る際、BUFFERCOUNT を増やすという、効率的なディスクの読み取りと同じアドバイスに従ってください。ネットワーク経由でデータを書き込む際、いくつかの点を考慮する必要があります:

- アプリケーションがバッファリングなしによるレコードの書き出しを必要としない限り、上記で説明した方法で、バッファリング付きの書き出しを有効にします。
- 特に大きなファイルの場合、BLOCKSIZE 値を増やすことは、ネットワーク上で送られるブロックサイズを増やし、ネットワーク・データ・ブロックが送られる頻度を増やします。
- 同様の環境で異なる BLOCKSIZE 値を使用し、アプリケーションを測定します。これにより、最適なネットワーク・ブロック・サイズを見つけることができます。

レコードを書き出す際、入出力レコードは UBC (unified buffer cache) システムバッファに書き出されることに注意してください。入出力レコードをプログラム・バッファから UBC システムバッファへ書き出すようにするには、フラッシュ・ライブラリ・ルーチンを使用します (『Intel® Fortran Library Reference』(英語) を参照)。フラッシュを呼び出すと、ユーザバッファの先読みデータを破棄します。

RECL を指定する

レコード長 (OPEN 文の RECL 指定子) とそのオーバーヘッドの和は、ブロックサイズの倍数または除数で、デバイスによって異なります。 例えば、BLOCKSIZE が 8192 だった場合、RECL としては 24576 (3 倍) や 1024 (8 で割った値) が考えられます。

RECL 値は、ブロックの容量にできるだけ近い値が設定されているべきです (ただし容量を超えてはなりません)。このような値であれば、各操作で可能な限り多くのデータを扱え、ブロック内の無駄な空間が最小限に抑えられるので、転送の効率が高くなります。ブロックの容量を超える値を使用すると、超えた分のデータだけで 1 ブロックを消費することになるので、移動の効率が大幅に低下します。これは、バッファのために余分なメモリを割り当て、部分的なブロックを書き出すという非効率的な操作が行われることが原因です。

書式付きファイルの RECL 値の単位は常に 1 バイト単位です。書式なしファイルの場合、RECL の単位は、`-assume byterecl` オプションを指定して 1 バイト単位を指定しない限り、4 バイト単位です (`-assume byterecl` を参照)。

最適なレコード型を使用する

移植性の理由から、特定のレコード型が必要な場合、次に示すように最も効率的なレコード型を選択してください。

- レコードの大きさが一定のシーケンシャル・ファイルの場合、固定長レコード型が最も性能を向上させます。
- レコードの大きさが一定でない書式なしシーケンシャル・ファイルの場合、特に BACKSPACE 操作で、可変長レコード型が最も性能を向上させます。
- レコードの大きさが一定でない書式付きシーケンシャル・ファイルの場合、`Stream_LF` レコード型が最も性能を向上させます。

リダイレクトされた標準入力ファイルから読み取る

Fortran ランタイム・システムは標準入力の整合性を保つ処理を行うため、標準入力がファイルからリダイレクトされると、読み取りが非常に遅くなります。例えば、`myprogram.exe < myinput.data>` のようなコマンドを使用した場合、データは READ(*) または READ(5) 文で読み取られ、パフォーマンスは悪くなります。この問題を回避するためには、次のいずれかを行います:

- OPEN 文を使用して、明示的にファイルを開きます。次に例を示します:

  ```
  open(5, STATUS='OLD', FILE='myinput.dat')
  ```

- 環境変数を使用して入力ファイルを指定します。

このような手法を活用するには、プログラムが標準入力ファイルの共有を使用しないように注意してください。

インテル Fortran データ・ファイルと入出力に関する詳細は、Vol I の「ファイル、デバイス、I/O」を参照してください。OPEN 文の指定子とそのデフォルト値については、『Intel® Fortran Language Reference』(英語) マニュアルの「Open Statement」を参照してください。

ランタイム効率の向上

ソースコード上のガイドラインを実施することで、ランタイム・パフォーマンスを向上させることができます。ランタイム・パフォーマンスをどの程度向上できるかは、文が実行される回数に関連します。例えば、ループ内で多数回実行される算術式を改善すると、ループ外で 1 回実行される同様の式を改善するよりも、パフォーマンス向上の可能性が高くなります。

小さな整数および小さな論理データ項目の回避

32 ビットよりも小さな整数または論理データを使わないようにしてください。16 ビット (または 8 ビット) データ型へアクセスすると、データアクセスの効率性が低下します。特に Itanium® ベース・システムでは、その傾向は顕著となります。

配列データの格納領域とメモリ・キャッシュ・ミスを最小限にするには、64 ビットデータでなく 32 ビットデータを使用します。ただし、8 バイト整数より広い数値範囲や、倍精度浮動小数点数より広い範囲と精度を必要とする場合を除きます。

混合データ型算術式の回避

整数および浮動小数点 (REAL) データを同一の計算に混在させないようにします。浮動小数点算術式 (代入文) のすべての数を浮動小数点値として表現することで、固定形式と浮動小数点形式間のデータ変換が不要となります。また、整数算術式のすべての数を整数値として表現することにも、同様の利点があります。これによってランタイム・パフォーマンスは向上します。

例えば、I と J が共に INTEGER 変数である場合、定数 (2.) を整数値 (2) として表現することで、データを変換する必要がなくなります。

非効率的なコード:

```
INTEGER I, J
I = J / 2.
```

効率的なコード:

```
INTEGER I, J
I = J / 2
```

1 つの式でサイズの異なる同じ汎用データ型を使用しても、ランタイム・パフォーマンスには最小限の影響しかないか、まったく影響がない場合もあります。例えば、同じ浮動少数点算術式で、REAL、DOUBLE PRECISION、COMPLEX 浮動少数点数を使用しても、ランタイム・パフォーマンスへの影響は最小限か、あるいは、皆無です。

効率的なデータ型の使用

変数に複数のデータ型が使える場合、次の順序に従ってデータ型を選択してください。ここに示す順序は効率の高い順にリストされています。

- 整数 (上記の例も参照)

- 単精度実数で、明示的に REAL、REAL (KIND=4)、または REAL*4 と示したもの
- 倍精度実数で、明示的に DOUBLE PRECISION、REAL (KIND=8)、または REAL*8 と示したもの
- 拡張精度実数で、明示的に REAL (KIND=16) または REAL*16 と示したもの

ただし算術式では、整数データと浮動小数点 (REAL) データを混在させないようにしてください (前項の例を参照)。

実行速度の遅い算術演算子の回避

実行速度の遅い算術演算子を避けるためにコードを修正する前に、最適化によって、多数の低速な算術演算子が高速の算術演算子に変換される可能性があることに注意してください。例えば、コンパイラの最適化により、式 `H=J**2` は `H=J*J` に変換されます。

さらに、低速な算術演算子を高速の算術演算子に置き換えることによって、結果の精度が変わったり、またソースコードの保守性 (読み易さ) に影響が生じることがないかどうかを考慮する必要もあります。

低速な算術演算子を高速な算術演算子へ置き換えるのは、重要なコード部分については必要な場合のみ行います。次に、インテル® Fortran 算術演算子を高速のものから順に示します。

- 加算 (+)、減算 (-)、浮動小数点乗算 (*)
- 整数乗算 (*)
- 除算 (/)
- べき乗 (**)

EQUIVALENCE 文の使用の回避

EQUIVALENCE 文は使わないようにします。EQUIVALENCE により、次のような状況が発生します。

- アライメントされないデータや、自然境界にまたがるデータを生じさせます。
- 次に示すようないくつかの最適化を妨げます。
  - 特定の条件下でのグローバル・データ解析 (「-On オプションによる最適化の設定」の `-O2` を参照)
  - `EQUIVALENCE` 文に制御変数が含まれている場合の暗黙的な `DO` ループコラプス

文関数および内部サブプログラムの使用

インテル Fortran コンパイラは、コンパイル時に、使用されているサブプログラムの定義にアクセスできる場合、サブプログラムのインライン化を行うことがあります。文関数および内部サブプログラムを使用することで、インライン化されるサブプログラム参照数が最大になります。特に多数のソースファイルを最適化レベル `-O3` で、まとめてコンパイルする場合に効果があります。

詳細は、「効率的なコンパイル」を参照してください。

DO ループの効率を考慮したコーディング

可能な限り DO ループ内の算術演算や他の演算は最小限に抑えます。不要な演算をループの外に移動することによって、パフォーマンスが向上します (例えば、ループ中の中間不変値が不要な場合など)。

ループの最適化についての詳細は、「Itanium® ベース・アプリケーションのパイプライン化」および「ループのアンロール」を参照してください。インテル Fortran 文の構文については、『Intel® Fortran Language Reference』(英語) マニュアルを参照してください。

Itanium® ベース・システムにおける組込み関数の使用

インテル® Fortran は、すべての標準 Fortran 組込みプロシージャをサポートし、さらに、言語機能を拡張するインテル固有の組込みプロシージャを提供します。インテル Fortran 組込みプロシージャは `libintrins.a` ライブラリに含まれます。組込みプロシージャに関する詳細は、『Intel® Fortran Language Reference』(英語) を参照してください。

ここでは、効果的なアプリケーション開発に役立つインテル拡張組込み関数の例を示します。

CACHESIZE 組込み関数 (Itanium® コンパイラ)

組込み関数 CACHESIZE (n) は Itanium コンパイラでのみ使用されます。
CACHESIZE (n) は n レベルのキャッシュ・サイズをキロバイトで返します。1 は一次キャッシュを表します。キャッシュ・レベルが存在しない場合、ゼロが返されます。

プログラマが、対象となるプロセッサのキャッシュ階層におけるアルゴリズムを調整する際、多くの場合、この組込み関数を使用できます。例えば、アプリケーションはキャッシュ・サイズをクエリし、行列を操作するアルゴリズムでブロックサイズを選択するのにキャッシュ・サイズを使用します。

```
subroutine foo (level)
integer level
if (cachesize(level) > threshold) then
call big bar()
else
call small bar()
end if
end subroutine
```

インテル® アーキテクチャのコーディング・ガイドライン

ここでは、次のコーディング手法における一般的なガイドラインについて説明します。

- MMX® テクノロジおよび、ストリーミング SIMD 拡張命令 (SSE) とストリーミング SIMD 拡張命令 2 (SSE2) をサポートする IA-32 アーキテクチャ
- Itanium® アーキテクチャ

このトピックでは、IA-32 および Itanium プロセッサ・ファミリのパフォーマンスを向上させるアーキテクチャ機能と関連のあるコーディング手法、ツール、規則、および推奨事項について説明します。IA-32 プロセッサ用の最適化に関する詳細は、『インテル® アーキテクチャ最適化リファレンス・マニュアル』を参照してください。Itanium プロセッサ・ファミリ用の最適化に関する詳細は、『インテル® Itanium® 2 プロセッサ・リファレンス・マニュアル: ソフトウェアの開発と最適化』 を参照してください。

注
ガイドラインで特定のアーキテクチャのみに適用される部分には、そのアーキテクチャが明記されています。それ以外の部分はすべて、IA-32 アーキテクチャと Itanium アーキテクチャの両方に該当します。

コンパイラで生成されたコードのパフォーマンスは、使用するコンパイラによって異なります。インテル® Fortran コンパイラは、インテル・アーキテクチャ用に高度に最適化されたコードを生成します。コンパイラのさまざまな最適化オプションを使用して、パフォーマンスを大幅に向上させることができます。ここで説明するガイドラインに従うことで、コンパイラによる Fortran プログラムの最適化の効率性をさらに高めることができます。

Fortran アプリケーションで最大限のプロセッサ・パフォーマンスを得るには、次の手順を実行します:

- メモリアクセスのストールを回避すること
- 浮動小数点のパフォーマンスを高めること
- SIMD 整数処理のパフォーマンスを高めること
- ベクトル化を使用すること

ここで説明するコーディング手法、規則、および推奨事項により、インテル・アーキテクチャ・ベースのプロセッサ上のパフォーマンスを最適化することができます。

メモリアクセス

インテル・コンパイラは Fortran の配列を列優先順に編成します。例えば、2 次元配列の場合、メモリ内では `A(22, 34)` と `A(23, 34)` の各要素が連続して格納されます。最高のパフォーマンスを得るには、内側のループが配列に連続アクセスできるように配列のコーディングを行ってください。次の例について考えてみます。

例 1 のコードは例 2 より優れたパフォーマンスが得られます。

例 1

```
DO J = 1, N
DO I = 1, N
B(I,J) = A(I, J) + 1
END DO
END DO
```

このコードでは、内側のループ `I` で配列 `A` と配列 `B` に連続してアクセスするので、パフォーマンスが向上します。

例 2

```
DO I = 1, N
DO J = 1, N
B(I,J) = A(I, J) + 1
END DO
END DO
```

このコードでは、内側のループ `J` で配列 `A` と配列 `B` へのアクセス方法が非連続的なので、パフォーマンスが低下します。

内側のループがメモリに連続してアクセスできるように、コンパイラによってコードを変換することも可能です。このコード変換を行うには、`-O3` (IA-32 と Itanium アーキテクチャ両方) および `-O3` と `-ax{K|W|N|B|P}` (IA-32 のみ) などの高度な最適化を使用する必要があります。

メモリ・レイアウト

優れたパフォーマンスを得るにはアライメントが重要な役目を果たします。アライメントされたメモリのアクセスは、アライメントされていないメモリのアクセスより高速に処理されます。複数のファイルに対するプロシージャ間の最適化 (`-ipo` オプション) を使用する場合、コンパイラがコードを分析し、アライメントされた境界から配列を開始できるよう配列のパディングを行うべきかどうかを判断します。1 つの共通ブロックで複数の配列を指定すると、コンパイラに余分な負担がかかることがあります。例えば、次のような COMMON 文の場合を検討してみます:

```
COMMON /AREA1/ A(200), X, B(200)
```

ここでは、コンパイラが `A(1)` をアライメントするために、16 バイトにアライメントされたアドレスにパディングを追加した場合、`B(1)` という要素は 16 バイトにアライメントされたアドレスに格納されません。このような場合には、`AREA1` を次のように分割します。

```
COMMON /AREA1/ A(200)
COMMON /AREA2/ X
COMMON /AREA3/ B(200)
```

上記のコードを使用すると、コンパイラが `A` と `B` の両方に必要なパディングを判断するのに最大限の柔軟性をもたせることができます。

浮動小数点アプリケーションの最適化

浮動小数点のパフォーマンスを向上するには、次の一般的な規則に従ってください:

- 計算中に表現可能な範囲を超過しないようにします。表現可能な範囲を超過すると、その処理によりパフォーマンスが低下する可能性があります。 DOUBLE または REAL*8 変数の高精度が必要な場合を除き、REAL 変数を単精度形式で使用してください。また、高精度の形式を持つ変数を使用すると、メモリサイズが増え、より大きな帯域幅が必要となります。
- IA-32 のみ: 複数の変数間で丸めモードを何度も変更するのは避けてください。丸めモードを頻繁に変更すると、非 SSE 命令を使用して計算を行った場合にパフォーマンスが低下する可能性があります。したがって、非 SSE コードを生成する場合に `FLOOR` 命令および `TRUNC` 命令を一緒に使用することも避けてください。これは、`CEIL` と `TRUNC` の使用についても同じです。

  この問題を回避するには、`-x{K|W|N|B|P}` オプションを使って SSE 命令により計算を実行する方法もあります。

- 次の方法で、両方のアーキテクチャについてデノーマル例外の影響を軽減します。

デノーマル例外

アンダーフローのある浮動小数点計算は結果がデノーマル値となり、そのためにパフォーマンスに悪影響を与えることがあります。

IA-32: ストリーミング SIMD 拡張命令 (SSE)、およびストリーミング SIMD 拡張命令 2 (SSE2) の SIMD 機能を利用します。`-x{K|W|N|B|P}` オプションは、SSE 命令および SSE2 命令で FTZ (ゼロ・フラッシュ) モードを有効にします。そのため、アンダーフロー結果が自動的にゼロに変換されるので、アプリケーションのパフォーマンスが改善します。さらに、`-xP` オプションを使用して DAZ (denormals are zeros) モードを有効にする方法もあります。これにより、デノーマルが入力時にゼロに変換され、パフォーマンスをさらに改善できます。IEEE-754 に完全準拠する必要がなく、実行速度を向上させたい場合には、これらのオプションが適しています。FTZ および DAZ の詳細に関する詳細は、「FTZ フラグと DAZ フラグの設定」および、『インテル® アーキテクチャ最適化リファレンス・マニュアル』の「浮動小数点例外」を参照してください。

Itanium アーキテクチャ: -O3 オプションで設定される -ftz オプションを使用して、FTZ (ゼロ・フラッシュ) モードを有効にします。

ベクトル化の自動処理

多くのアプリケーションでは、ベクトル化を実装することでパフォーマンスを大幅に向上することができます。ベクトル化はメインの計算ループにストリーミング SIMD SSE2 命令を使用します。インテル・コンパイラでベクトル化をオンにする (ベクトル化の自動処理) か、コンパイラ・ディレクティブを使用して実装することもできます。

詳細については、「ベクトル化の自動処理 (IA-32 のみ)」を参照してください。

マルチスレッド・アプリケーションの作成

インテル Fortran コンパイラおよびインテルのスレッド化ツールのセットには、マルチスレッド・アプリケーションの開発を支援する機能が用意されています。詳細については、「インテル® Fortran による共用メモリ並列プログラム」を参照してください。 マルチスレッド・アプリケーションは、マルチプロセッサのインテル SMP (対称型マルチプロセッシング) システムや、ハイパースレッディング・テクノロジを使用したインテル® プロセッサ上ではパフォーマンスが大幅に向上します。

アプリケーションの分析と時間測定

インテルが提供するパフォーマンス分析ツールの使用

インテルでは、インテル® アーキテクチャ・ベースの各プロセッサに最適なアプリケーション・パフォーマンス・ツールを各種提供しています。これらツールを利用すると、アセンブリ・コードを記述することなく、最も効率的なプログラムを開発できます。

アプリケーションを分析したり、問題箇所を検出し解決するには、次のパフォーマンス・ツールを使用できます。

- インテル® デバッガ (IDB)

  IDB デバッガは、コマンドラインまたは GUI を通してプログラムのデバッグにおける広範なサポートを提供します。

- インテル® VTune™ パフォーマンス・アナライザ

  VTune アナライザは、インテル・アーキテクチャ固有のソフトウェアのパフォーマンス・データの収集、分析を行い、システム全体を対象にしたデータから、コード内の特定のモジュール、関数、命令を対象とした詳細データまで提供します。詳細は、http://www.intel.co.jp/jp/developer/software/products/vtune/index.htm を参照してください。

- インテル® スレッド化ツール。スレッド化ツールには次が含まれます。
  - インテル® スレッド・チェッカー
  - インテル® スレッド・プロファイラ

詳細は、http://www.intel.com/jp/developer/software/products/index.htm を参照してください。

アプリケーションの時間測定

アプリケーションの時間測定は、パフォーマンス測定の 1 つの目安となります。`time` コマンドを使用して、プログラムのパフォーマンス情報が得られます。次の点を考慮して行ってください。

- ユーザが誰もアクティブではないときに、プログラムの時間測定を行います。時間測定中、1 つまたは複数の CPU 集中型プロセスが起動していると、結果に影響します。
- 最も正確な結果を得るには、毎回、同じ条件でプログラムを実行するようにしてください。特に、同一プログラムの以前のバージョンと比較する際は注意してください。可能ならば、同じ CPU システム（モデル、メモリ、オペレーティング・システムのバージョンなど）で実行してください。
- システムを変更する必要がある場合、両方のシステムでプログラムの同一バージョンの速度を測ります。こうすることによって、システムによる速度の違いを把握できます。
- 数秒以下で実行されるプログラムの場合は、数回測定し、正しい結果であることを確認してください。共有ライブラリをロードするようなオーバヘッドのあるプログラムは、短い時間に大きな影響を与えます。

実行プログラムの名前を指定して、`time` コマンドを使用すると、次の情報が得られます。

- 経過時間、実時間、または ″ウォールクロック″ 時間。これらは、合計 CPU 使用実時間よりも大きくなります。
- CPU 使用実時間。システム、ユーザ実行の両方が表示されます。合計 CPU 使用実時間は、実際のユーザ CPU 時間とシステム CPU 時間の合計です。

例

次の時間測定の例では、サンプル・プログラムは、次のような行を表示します。

```
Average of all the numbers is:    4368488960.000000
```

Bourne* シェルを使用して、次のプログラム時間測定を行うと、合計 CPU 実時間が 1.19 秒（ユーザ・プログラムの CPU 実時間 0.61 秒とシステムの CPU 実時間 0.58 秒の合計）、経過時間が 2.46 秒であることがレポートされます。

```
$ time a.out

Average of all the numbers is:
  4368488960.000000


real    0m2.46s


user    0m0.61s


sys     0m0.58s
```

C シェルを使用すると、合計 CPU 実時間が 1.19 秒 (ユーザ・プログラムの CPU 実時間 0.61 秒とシステムの CPU 実時間 0.58 秒の合計) 、経過時間が 4 秒 (0:04) 、CPU 使用時間が 28% であることがレポートされます。

```
% time a.out

Average of all the numbers is:   4368488960.000000

0.61u 0.58s 0:04 28% 78+424k 9+5io 0pf+0w
```

Bash シェルを使用すると、合計 CPU 実時間が 1.19 秒 (ユーザ・プログラムの CPU 実時間 0.61 秒とシステムの CPU 実時間 0.58 秒の合計) 、経過時間が 2.46 秒であることがレポートされます。

```
[user@system user]$ time ./a.out

Average of all the numbers is:   4368488960.000000

elapsed  0m2.46s

user     0m0.61s

sys      0m0.58s
```

システム時間が大きい場合は、I/O に多くの時間がかかっている可能性があります。この場合、調査する必要があるかもしれません。

プログラムが多くのテキストを表示する場合は、time コマンドラインでプログラムからの出力をリダイレクトできます。プログラムの出力をリダイレクトすると、画面 I/O が減少するため、レポートされる時間が変わります。

詳細は、「time(1)」を参照してください。

time コマンドに加えて、実行時間を測定するためのルーチンを呼び出すようにプログラムを変更してみてください。例えば、SECNDS、DCLOCK、CPU_TIME、SYSTEM_CLOCK、TIME および DATE_AND_TIME などのインテル® Fortran 組込みプロシージャを使用できます。 詳細は、『Intel® Fortran Language Reference』(英語) の「Intrinsic Procedures」を参照してください。

コンパイラの最適化

コンパイラの最適化の概要

インテル® Fortran コンパイラの最適化を使用して、アプリケーションのパフォーマンスを向上することができます。次に示すセクションでは、最適化オプションについて説明します:

- コンパイル処理の最適化 (スタック、アライメント、およびシンボルの可視属性オプションを含む)
- 異なるアプリケーション・タイプの最適化
- 浮動小数点算術演算
- 特定のプロセッサの最適化
- プロシージャ間の最適化 (IPO)
- プロファイルに基づく最適化
- 高水準言語の最適化

コンパイラのコマンドライン・オプションで実行される最適化に加えて、ディレクティブ、組込み関数、ランタイム・ライブラリ・ルーチンおよびユーティリティなどのパフォーマンスを高める機能が提供されています。 これらの機能は、「最適化サポート機能」で説明します。

コンパイル処理の最適化の概要

ここでは、インテル® Fortran コンパイラでコンパイル処理を最適化するオプションについて説明します。デフォルトでは、ソースコードはコンパイラによって実行ファイルに直接変換されます。適切なオプションを指定することで、コンパイル処理を制御したり、コンパイラで生成される出力ファイルを指定できるだけではなく、コンパイル処理自体の効率を向上することができます。

監視用のオプション・グループでは、インテル® コンパイラから生成されたコードの最終的な成果物をプログラムの実行には影響を与えずに監視します。これらのオプションは、スタックメモリの割り当て、変数の設定と変更、レジスタの使用の定義など、いくつかの演算を制御します。

ここで説明するオプションにより、コンパイルを効率化する次の機能が提供されます:

- 変数とスタックの自動割り当て
- データのアライメント

- シンボルの可視属性オプション

効率的なコンパイル

効率的なコンパイル処理を行うことでパフォーマンスが向上します。プログラムのパフォーマンスを向上させるには、プログラムを分析する前に、まずコンパイル自体を効率的に行うことが大切です。アプリケーションの分析結果に基づいて、インテル® Fortran コンパイラの最適化とコマンドライン・オプションのうち、どれを使用するとアプリケーションのランタイム・パフォーマンスが改善するかを判断することができます。

効率的なコンパイル手法

効率的なコンパイル手法は、プログラム開発の初期および後期の段階で使用できます。

プログラム開発の初期の段階では、最適化を最小限に抑えて増分コンパイルを行うことができます。次に例を示します。

ifort -c -g -O0 *sub2.f90* (sub2 のオブジェクト・ファイルを生成)

ifort -c -g -O0 *sub3.f90* (sub3 のオブジェクト・ファイルを生成)

ifort -o *main* -g -O0 *main.f90 sub2.o sub3.o*

これらのコマンドは -O0 を使用して、コンパイラのデフォルトの最適化オプション (-O2 など) をすべてオフにします。-g オプションを使用すると、プログラムに含まれるすべてのルーチンのオブジェクト・コードにシンボリック・デバッグ情報と行番号を生成し、これをソースレベルのデバッグに使用することができます。3 番目のコマンドで生成される main ファイルにも、シンボリック・デバッグ情報が含まれています。

プログラム開発の後期の段階では、一度に複数のソースファイルを指定して、少なくとも -O2 (デフォルト設定) 以上の最適化レベルを使用することをお勧めします。例えば次のコマンドは、デフォルトの最適化レベルである -O2 を使用して 3 つのソースファイルを一緒にコンパイルします:

ifort -o *main main.f90 sub2.f90 sub3.f90*

複数のソースファイルをコンパイルすることで、コンパイラが最適化の対象候補となるコードをより多く検証することができ、その結果次のことが可能になります:

- より多くのプロシージャのインライン化
- より完全なデータフロー解析

- リンク中に解決される外部参照数の削減

非常に大きなプログラムの開発では、すべてのソースファイルを一緒にコンパイルすることが実用的ではない場合があります。その場合、各ソースファイルを別々にコンパイルするのではなく、複数の `ifort` コマンドを使用して関連のあるルーチンを含むソースファイルを一緒にコンパイルすると効率的です。

ランタイムのパフォーマンスを向上させるオプション

次の表に、ランタイムのパフォーマンスを直接改善できるオプションをアルファベット順に示します。これらオプションの大半は結果の精度に影響を与えませんが、一部のオプションではランタイムのパフォーマンスを向上させる一方、数値結果が変わる可能性があります。インテル Fortran コンパイラでは、コマンドライン・オプションを使ってオフに設定しない限り、一部の最適化オプションがデフォルトで実行されます。それ以外の最適化オプションは、コマンド・オプションを使用して有効または無効にできます。

オプション	説明
`-align` *keyword*	変数と配列のメモリ・レイアウトを分析し、再整理します。共通ブロック、派生型データ、およびレコード構造体内に含まれるデータ項目間に、自然なアライメントを行えるようにパディング・バイトを追加するかどうかを制御します。
`-ax{K|W|N|B|P}` IA-32 とインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムのみ	アプリケーションのパフォーマンスを特定のプロセッサ用に最適化します。`-ax` のどれを選択しても、アプリケーションは対象のプロセッサのあらゆる機能を使用するように最適化され、生成されたバイナリファイルはすべての IA-32 プロセッサ上で実行できます。
`-fast`	ランタイムのパフォーマンス用の最適化オプション一式を有効にします。
`-O1`	好ましいコードサイズとコードの局所性に最適化します。「-On オプションによる最適化の設定」を参照してください。
`-O2`	速度について最適化します。パフォーマンス関連のオプションを設定します。-On オプションによる最適化の設定。
`-O3`	ループ変換の最適化をアクティブにします。-On オプションによる最適化の設定。
`-openmp`	パラレライザが OpenMP* ディレクティブに基づいてマルチスレッド・コードを生成できるようにします。
`-parallel`	自動並列化を有効にして、並列で安全に実行できるループのマルチスレッド・コードを生成します。

-qp	プログラムのソースコードの効率性を高めると、ランタイムのパフォーマンスが向上する可能性が高いプログラムの個所を識別するためのプロファイル情報を要求します。適切なソースコードを修正してから、プログラムを再コンパイルして、ランタイムのパフォーマンスをテストしてください。
-tpp{n}	アプリケーションのパフォーマンスを特定のインテル® プロセッサ用に最適化します。「対象とするプロセッサの指定 （-tpp{n}）」を参照してください。
-unroll*n*	最適化レベルが -O3 に指定された場合、ループをアンロールする回数 (*n*) を指定します。-unroll オプションで *n* を省略すると、最適化プログラムがループのアンロール最大回数を判断します。

ランタイムのパフォーマンスを低下させるオプション

次の表に、ランタイムのパフォーマンス低下につながる可能性のあるオプションをアルファベット順に示します。浮動小数点例外処理や丸め処理が必要なアプリケーションでは、-fpe*n* というダイナミック・オプションの使用が必要となることがあります。その他のアプリケーションは、互換性のため -assume dummy_aliases オプションあるいは -vms オプションが必要な場合があります。ランタイムのパフォーマンス低下につながるその他のオプションは、主にトラブルシューティングやデバッグの目的で使用されます。

次の表に、ランタイムのパフォーマンスを低下させる可能性のあるオプションを示します。

オプション	説明
-assume dummy_aliases	コンパイラはプロシージャへの仮引数が他の仮引数と、または参照結合、ホスト結合、または共通ブロックを使用して共有した変数とメモリ位置を共有していることを仮定します これらのプログラム・セマンティクスはパフォーマンスを低下させるので、-assume *dummy_aliases* は、このようなエイリアスに依存する、呼び出される側のサブプログラムにのみ指定してください。 仮エイリアスの使用は Fortran 77 および Fortran 95/90 の各標準に違反しますが、一部の古いプログラムで使用されています。
-check bounds	実行時に配列境界をチェックするための追加コードを生成します。

-check overflow	実行時に算術オーバーフローに対して整数計算をチェックするための追加コードを生成します。プログラムのデバッグが終了したら、このオプションを省略することで、実行ファイルのサイズを小さくして、ランタイムのパフォーマンスを若干向上させることができます。
-fpe 3	このオプションは、リソースを大量に使用する特定の種類の浮動小数点例外処理を有効にします。
-g	オブジェクト・ファイルに追加のシンボルテーブル情報を生成します。このオプションを指定すると、デフォルトの最適化レベルが -O0 または -O0 (最適化なし) に下がります。 **注** -g オプションは、最適化レベルが指定されていない場合にのみパフォーマンスを低下させます。最適化レベルが指定されていないと、-g が -O0 をオンに設定し、その結果コンパイルの速度が低下します。-g、-O2 が指定された場合、コードの実行速度は -g が指定されていない場合とほぼ同じです。
-O0	最適化をオフにします。プログラム開発の初期段階や、デバッガを使用する際に指定します。
-save	ローカル変数が、前回の実行が終了した時点の値を維持するようにします。このオプションでは、レジスタではなくメモリで演算が行われ、結果の丸めが頻繁になるので、プログラムから出力される浮動小数点値の結果が変わる可能性があります。
-vms	特定の VMS に関連した実行時のデフォルト設定 (アライメントを含む) を制御します。-vms オプションを指定する場合、ランタイム・パフォーマンスを最適化するために -align records オプションの併用が必要になることがあります。

リトル・エンディアン – ビッグ・エンディアンの変換

インテル® Fortran コンパイラでは、リトル・エンディアン – ビッグ・エンディアンの変換機能を使用して、ビッグ・エンディアン形式の書式なしシーケンシャル・ファイルの書き込み、およびビッグ・エンディアン形式で作成されたファイルの読み取りが可能です。

インテル Fortran は、IA-32 ベース・プロセッサと Itanium® プロセッサの両方でリトル・エンディアン形式の内部データを処理します。リトル・エンディアン - ビッグ・エンディアンの変換機能は、書式なしシーケンシャル・ファイルにおける Fortran 書式なし入力/出力操作を対象としています。この機能は次のような処理を可能にします:

- ビッグ・エンディアン・データ形式を受け付けるプロセッサ上で開発されたファイルの処理
- リトル・エンディアン・システム上における同様のプロセッサ用ビッグ・エンディアン・ファイルの作成

リトル・エンディアン - ビッグ・エンディアンへの変換は、次の操作によって実現されます:

- WRITE 操作は、リトル・エンディアン形式をビッグ・エンディアン形式に変換します。
- READ 操作は、ビッグ・エンディアン形式をリトル・エンディアン形式に変換します。

この機能は、変数および基本のデータ型の配列 (または配列の添字) の変換を有効にします。派生データ型はサポートされていません。

リトル・エンディアン - ビッグ・エンディアンの変換環境変数

リトル・エンディアン - ビッグ・エンディアンの変換機能を使用するためには、F_UFMTENDIAN 環境変数を設定して、変換に使用されるユニット数を指定します。その後、このユニット数を使用する READ/WRITE 文が対応する変換を実行します。他の READ/WRITE 文は、通常どおり動作します。

一般的に、変数はセミコロンで分割される 2 つの部分で構成されます。F_UFMTENDIAN 値の中では、スペースは使用できません。この変数の構文は、次のとおりです:

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

各アイテムの意味は次のとおりです:

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST,U
U = decimal | decimal -decimal
```

- MODE は、現在のデータ形式を定義し、ファイルで表現されます。 省略できます。
  キーワードの little は、データがリトル・エンディアン形式で、変換されないことを意味します。このキーワードはデフォルトです。
  キーワードの big は、データがビッグ・エンディアン形式で、変換されることを意味します。このキーワードは、コロンで省略できます。
- EXCEPTION は、MODE の例外リストの定義に使用し、省略できます。
  EXCEPTION キーワード (little または big) は EXCEPTION リストからのユニットを結合するファイル内のデータ形式を定義します。この値は、リストされたユニットの MODE 値を無効にします。
- 各リストメンバの U は、シンプルユニット番号またはユニットの数です。リストメンバの上限は、64 です。
  decimal は、 負でない小数で 2^{32} よりも少ない値です。

変換されたデータは、基本のデータ型または基本のデータ型の配列でなければなりません。派生データ型は無効です。

異なるシェルでの変数設定のコマンドライン:

```
Sh: export F_UFMTENDIAN=MODE;EXCEPTION

Csh: setenv F_UFMTENDIAN MODE;EXCEPTION
```

> **注**
> セミコロンがある場合は、環境変数値が引用符で囲まれていなければなりません。

その他の環境変数設定

また、環境変数には次の構文があります:

```
F_UFMTENDIAN=u[,u] ...
```

異なるシェルでの変数設定のコマンドライン:

- Sh: **export F_UFMTENDIAN=u[,u] ...**
- Csh: **setenv F_UFMTENDIAN u[,u] ...**

リトル・エンディアン – ビッグ・エンディアンの変換中に表示される可能性のあるエラー・メッセージを参照してください。すべて致命的なエラーです。そのようなエラーが発生した場合には、インテルのテクニカル・サポートに問い合わせてください。

使用例:

1. `F_UFMTENDIAN=big`

   すべての入力/出力操作で READ 上では、ビッグ・エンディアンからリトル・エンディアンへの変換、WRITE 上ではリトル・エンディアンからビッグ・エンディアンへの変換を実行します。

2. `F_UFMTENDIAN="little;big:10,20"`
   または `F_UFMTENDIAN=big:10,20`
   または `F_UFMTENDIAN=10,20`

   この場合、ユニット番号が 10 および 20 で、入力/出力操作はビッグ・エンディアン – リトル・エンディアンの変換を実行します。

3. `F_UFMTENDIAN="big;little:8"`

   この場合、ユニット番号 8 では変換操作は行われません。その他のすべてのユニットで入力/出力操作はビッグ・エンディアン – リトル・エンディアンの変換を実行します。

4. `F_UFMTENDIAN=10-20`

   変換目的で、10、11、12 …19、20 ユニットを定義します。 これらのユニットでは、入力/出力操作は、ビッグ・エンディアン – リトル・エンディアンの変換を実行します。

5. `F_UFMTENDIAN=10,100` を設定して、次のプログラムを実行したとします。

```
integer*4   cc4
integer*8   cc8
integer*4   c4
integer*8   c8
c4 = 456
c8 = 789

C  prepare a little endian representation of data


open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)

C  prepare a big endian representation of data
```

```
open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)

C  read big endian data and operate with them on
C  little endian machine.


open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4


C  Any operation with data, which have been read

C    ...
close(100)
stop
end
```

lit.tmp ファイルと big.tmp ファイルを od ユーティリティで比較します。

```
> od -t x4 lit.tmp

0000000 00000008 00000315 00000000 00000008
0000020 00000004 000001c8 00000004
0000034

> od -t x4 big.tmp

0000000 08000000 00000000 15030000 08000000
0000020 04000000 c8010000 04000000
0000034
```

これらのファイルでは、バイトの順番が異なることがわかります。

デフォルトのコンパイラの最適化

コンパイラ・オプションを指定せずにインテル® Fortran コンパイラを起動する場合は、各オプションのデフォルトが有効です。次の表は、インテル Fortran コンパイラのデフォルト実行に必要とされるデフォルトがオンのオプションをまとめたものです。表は、機能別にオプションをまとめています。

すべてのオプションのデフォルト状態と値に関する詳細は、『インテル® Fortran コンパイラ・オプション・クイック・リファレンス・ガイド』の「アルファベット順クイック・リファレンス 」の表を参照してください。 表には、オプションの機能を説明したセクションへ

のリンクが含まれます。オプションにデフォルト値がある場合には、その値が示されます。

アプリケーションの要件に応じて、1 つまたは複数のオプションを無効にできます。最適化を無効にする一般的な方法については、Vol I を参照してください。

次の表は、コンパイラがデフォルトの最適化で使用するすべてのオプションをリストしています。

データ設定と Fortran 言語準拠

デフォルトのオプション	説明
-align records	変数と配列のメモリ・レイアウトを分析し、再整理します。
-align rec8byte	アライメント条件として 8 バイト境界を指定します。
-altparam	パラメータ定数宣言の代替形式を識別します。
-ansi_alias	プログラムの ANSI 準拠の前提を有効にします。
-assume cc_omp	OpenMP 条件付きコンパイル・ディレクティブを有効にします。
-ccdefault default	ユニット 6 および * のデフォルトのキャリッジ制御を指定します。
-double size 64	DOUBLE PRECISION 型の宣言、定数、関数、および組込み関数を REAL*8 として定義します。
-dps	DEC* パラメータ文の認識を有効にします。
-error_limit 30	エラーレベルまたは致命的なエラーレベルのコンパイラ・エラーの最大数を指定します。
-fpe3	メイン・プログラム用の実行時の浮動小数点例外処理を指定します。
-integer size 32	デフォルトの INTEGER および LOGICAL 変数の長さは 4 バイトです。INTEGER 宣言および LOGICAL 宣言は、(KIND=4) として扱われます。
-pad	変数と配列のメモリ・レイアウトの変更を有効にします。
-pc80 IA-32 のみ	-pc{32|64|80} は、浮動小数点の仮数部の精度の制御を次のように有効にします: -pc32 では 24 ビットの仮数部、-pc64 では 53 ビットの仮数部、-pc80 では 64 ビットの仮数部。
-real_size 64	REAL および COMPLEX の宣言、定数、関数、および組込み関数のサイズを指定します。

-save	すべての変数を保存します (静的割り当て)。すべての変数を AUTOMATIC に設定する -auto を無効にします。
-Zp8	-Zp*n* は、構造体に、1、2、4、8、または 16 バイトの境界整列条件を指定します。無効にするには、-noalign または -Zp1 を使用します。

最適化

デフォルトのオプション	説明
-assume *cc omp*	OpenMP 条件付きコンパイル・ディレクティブを有効にします。
-fp IA-32 のみ	最適化で ebp レジスタの使用を無効にします。すべての関数について ebp ベースのスタックフレームを使用するように指示します。
-fpe3	メイン・プログラムのランタイム時の浮動小数点例外処理を制御します。オプションを無効にするには -fpe0 を使用します。
-IPF fltacc- Itanium® コンパイラ	コンパイラが浮動小数点の精度に影響を及ぼす最適化を適用します。
-IPF fma Itanium コンパイラ	浮動小数点積和/積差演算を 1 つの演算命令で行うことを有効にします。
-IPF fp speculation *fast* Itanium コンパイラ	コンパイラが浮動小数点演算をスペキュレーションするよう設定します。-IPF_fp_speculationoff は、このスペキュレーションを無効にします。
-O, -O2	最大速度の最適化を行います。
-openmp_report1	並列化されたループ、領域、およびセクションを示す診断を表示します。
-opt_report_level*min*	最適化レポートの最小レベルを指定します。
-par_report1	正常に自動並列化されたループを表示します。
-tpp2 Itanium コンパイラ	インテル® Itanium® 2 プロセッサ向けに Itanium ベース・アプリケーションのコードを最適化します。生成されたコードは、Itanium プロセッサと互換性があります。
-tpp7 IA-32 のみ	インテル® Pentium® 4 プロセッサおよびインテル® Xeon™ プロセッサ向けに IA-32 アプリケーションのコードを最適化します。

-unroll	-unroll[*n*]: *n* を省略すると、アンロールを実行するかどうかをコンパイラが判断します。(デフォルト) *n* を使用して、ループのアンロール回数の上限を設定します。 Itanium ®コンパイラは、現在、互換性に *n* = 0、-unroll0 (無効オプション) のみを使用します。
-vec_report1	正常にベクトル化されたループを示します。

デフォルトのオプションを無効にする

オプションを無効にするには、次のいずれかを使用します。

- 一般的に、1 つまたはグループの最適化オプションを無効にするには -O0 オプションを使用します。次に例を示します。

```
ifort -O2 -O0 input_file(s)
```

注
-O0 オプションは -O0、-O、-O1、-O2、および -O3 を含む相互排他的なオプション・グループの一部です。同じオプション・グループのオプションを 2 つ以上指定した場合、コマンドラインで最後に指定したオプションが、その前に指定したオプションよ りも優先されます。

- [-] として示される "-" を含むオプションを無効にするには、コマンドラインでオプションのそのバージョンを使用します。例: -ftz-
- {n} パラメータを持つオプションを無効にするには、n=0 のバージョンを使用します。例: -unroll0

注
コマンドライン上に、オプションを有効にするバージョンと無効にするバージョンの両方がある場合、最後に指定したバージョンが優先されます。

コンパイル・オプションの使用

スタック: 自動割り当てと確認

このグループのオプションは、コンパイラにより生成されたコードのスタックおよび変数の計算を制御できます。

変数の自動割り当て

-auto

-auto オプションは、ローカルで宣言された変数を、スタティック・ストレージではなくランタイム・スタックに割り当てるよう指定します。プロシージャで定義された変数が SAVE または ALLOCATABLE 属性を持たない場合、それらの変数は、スタックのみに割り当てられます。このオプションは、EQUIVALENCE 文または SAVE 文内の変数、および COMMON 内の変数には作用しません。

-auto の機能は、-automatic および -nosave と同じです。

-auto を指定すると、プログラムのパフォーマンスが向上することがあります。ただし、プログラムが、最後にルーチンが呼び出されたときと同じ値を持つ変数に依存している場合は、プログラムが正しく機能しないことがあります。複数のルーチン呼び出しにわたってその値を保持する必要のある変数は、SAVE 文内になければなりません。

-recursive または -openmp を指定した場合、デフォルトで -auto が使用されます。

-auto_scalar:

-auto_scalar オプションにより、INTEGER、REAL、COMPLEX、または LOGICAL の各組込み型ローカルスカラ変数のスタックへの割り当てが行われます。このオプションは、EQUIVALENCE 文または SAVE 文内の変数、および COMMON 内の変数には影響しません。

-auto_scalar を指定すると、プログラムのパフォーマンスが向上することがあります。ただし、プログラムが、最後にルーチンが呼び出されたときと同じ値を持つ変数に依存している場合は、プログラムが正しく機能しないことがあります。複数のサブルーチン呼び出しにわたってその値を保持する必要のある変数は、SAVE 文内になければなりません。このオプションは、すべてのローカル変数をスタックに割り当てる

-auto によく似ていますが、-auto_scalar は上記の型のスカラ変数のみをスタックに割り当てる点が異なります。

-auto_scalar を指定すると、コンパイラは、プログラムの実行中にどの変数をレジスタに保持すべきかについて、より良い選択が行えます。

-save, -zero

-save オプションは -auto オプションと逆の効果が得られます。-save オプションは、再帰ルーチン内にあるローカル変数を除くすべての変数を静的割り当てに保存します。ルーチンが 2 回以上呼び出される場合は、このオプションを指定すると、最後に終了した呼び出しから得られるローカル変数の値が保持されます。-save オプションにより、ルーチンの終了時に最終結果がメモリに保存され、次回の呼び出しの際に再利用されます。このオプションでは、結果の丸めが頻繁になるので、一部パフォーマンスが低下することがあります。

コンパイラがコードを最適化する際、結果はレジスタに保存されます。-save の機能は -noauto と同じです。

-zero[-] オプションは、INTEGER、REAL、COMPLEX、または LOGICAL 組込み型のローカルスカラ変数で、保存はされているがまだ初期化されていない変数を、すべてゼロに初期化します。-save とともに使用されます。デフォルトは、-zero-です。

まとめ

変数を割り当てるには、 -save、-auto、-auto_scalar の 3 つの方法があります。指定できるのはこれらのうち 1 つだけです。各方法は次のような相互関係にあります:

- -save を指定した場合、-auto が無効になり -noautomatic が設定され、AUTOMATIC として指定されていないすべての変数がスタティック・メモリに割り当てられます。
- -auto を指定した場合、-save が無効になり -automatic が設定され、SAVE として指定されていないすべての変数 (すべての型のスカラおよび配列) がスタックに割り当てられます。
- -auto_scalar:
  - このオプションは、INTEGER、REAL、COMPLEX、および LOGICAL の組込み型ローカルスカラを AUTOMATIC に設定します。
  - これはデフォルトのオプションです。-noauto_scalar というオプションはありません。ただし、-recursive または -openmp が使用された場合は -auto_scalar を無効にして -auto をデフォルトにします。

浮動小数点スタックの状態のチェック (IA-32 のみ) (-fpstkchk)

IA-32 専用の `-fpstkchk` オプションは、プログラムが浮動小数点値を返す関数への呼び出しを正しく行っているかどうかをチェックします。不正な呼び出しが検出されると、このオプションがプログラムに不正な呼び出しをマークするコードを追加します。

アプリケーションが浮動小数点値を返す関数を呼び出すと、返された浮動小数点値は通常、浮動小数点スタックの最上位に保存されます。戻り値が使用されない場合、コンパイラは浮動小数点スタックを正しい状態に保つため、浮動小数点スタックから値をポップします。

アプリケーションが関数のプロトタイプを定義しなかったり、誤ったプロトタイプを定義して関数を呼び出すと、その関数が浮動小数点値を返す必要があるかどうかをコンパイラが判断できず、戻り値が使用されない場合に浮動小数点スタックからその値がポップされません。この場合には、浮動小数点スタックがオーバーフローする可能性があります。

スタックのオーバーフローにより、次のような好ましくない事態が発生します:

- 浮動小数点の計算に NAN 値が使用されることがあります。
- プログラムで予期しない結果が発生することがあります。また、プログラムでエラーが発生した箇所が実際のエラーの個所からかなり離れていることもあります。

`-fpstkchk` オプションは、不正な呼び出しにマークを付けてエラーを検出しやすくします。

注

このオプションは、各関数/サブルーチン呼び出しの後で大量のコードを生成して浮動小数点スタックを正しい状態に維持しますが、コンパイル処理の速度は低下します。デバッグの際に浮動小数点スタックのアンダーフローやオーバーフローの問題を検出するためにのみ使用してください。これらの問題はこのオプションを使用しないと検出するのが困難です。

別名

–common_args

-common_args オプションは、「参照による」サブプログラム引数がお互いの別名を持っているとみなします。

CRAY* ポインタ・エイリアシングを回避する

オプション -safe_cray_ptr は、CRAY* ポインタが他の変数とエイリアスしないように指定します。デフォルトはオフです。

次の例について考えてみます。

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
b(i) = a(i) + 1
enddo
```

-safe_cray_ptr オプションが指定されない場合 (デフォルト)、コンパイラは、b と a がエイリアスされていると仮定します。このような仮定を回避するには、このオプションを指定します。コンパイラは b(i) と a(i) がお互いに独立しているとして処理します。

しかし、変数を CRAY ポインタとエイリアスする場合、-safe_cray_ptr オプションを使用すると、正しい結果が得られません。下記の例のようなコードでは、-safe_cray_ptr オプションを使用しないでください。

```
pb = loc(a(2))
do i=1, n
b(i) = a(i) +1
enddo
```

-ansi_alias

-ansi_alias[-] が有効 (デフォルト) に設定されている場合、コンパイラはプログラムが ANSI Fortran 型のエイリアス規則に準拠していると仮定します。例えば、real 型のオブジェクトは、integer としてアクセスできません。詳しい規則については、ANSI 標準を参照してください。

このオプションは、次の点を前提にしてコンパイラに指示します。

- 配列は、配列の外部からアクセスされません。
- ポインタを非ポインタ型にキャストしません。また、その逆もしません。

- 2 つの異なるスカラ型のオブジェクトへの参照にはエイリアスを設定できません。例えば、整数型のオブジェクトは実数型のオブジェクトと、または実数型のオブジェクトは倍精度型のオブジェクトとエイリアスの設定はできません。

プログラムが上記の条件を満たす場合、-ansi_alias フラグを設定することでプログラムの最適化が向上します。ただし、プログラムが上記の条件を 1 つでも満たさない場合、コンパイラは誤ったコードを生成する可能性があるため、このオプションを無効にしなければなりません。

-ansi_alias のシノニムは、-assume [no]dummy_aliases です。

アライメント・オプション

-align recnbyte または -Zp[n]

-align recnbyte (または -Zp[n]) オプションを使用して、n バイト境界 (-Zp[n] では n = 1、2、4、8、16 のいずれか) へのアライメント条件を指定します。

このオプションを指定すると、構造体の 2 番目以降の各メンバが、メンバの型のサイズまたは n バイト境界 (n = 1、2、4、8、16 のいずれか) のどちらか小さい方で格納されます。

例えば、prog1.f ファイルのすべての構造体と共用体に対するパック境界またはアライメント条件として、2 バイトを指定するには、次のコマンドを使用します。

```
ifort -Zp2 prog1.f
```

IA-32 および Itanium® ベース・システムのデフォルトは、-align *rec8byte* または -Zp8 です。-Zp16 オプションを使用して、共通ブロックなどの Fortran 構造体をアライメントすることができます。Fortran 構造体については、『*Intel® Fortran Language Reference*』(英語) の STRUCTURE 文を参照してください。

n を省略して -Zp を指定すると、構造体は 8 バイト境界にパックされます。

-align と -pad

-align オプションは、共通ブロック内の変数アライメントを変更するフロントエンドのオプションです。

例:

```
common /block1/ch,doub,ch1,int
integer int
character(len=1) ch, ch1
double precision doub
end
```

-align オプションは、doub および int の自然境界上へのアライメントを保証するための、パディングの挿入を有効にします。-noalign オプションは、パディングを無効にします。

-align オプションは主に構造体に適用され、変数と配列のメモリ・レイアウトを分析し、再整理します。基本的には -Zp{n} と同じ動作をします。-noalign でいずれのオプションも無効にできます。

-align *keyword* オプションについては、『インテル® Fortran コンパイラ・ユーザーズ・ガイド Vol I』を参照してください。

-pad オプションは、構造体と派生型に適用されたときには、事実上、-align と同じです。ただし、-pad の範囲はより広く、共通ブロック、派生型、シーケンス形式、VAX* 構造体にも適用できます。

オプションを使用してアライメントを制御する際の推奨事項

次のオプションを使用して、インテル Fortran コンパイラが、共通ブロック、派生型データ、およびインテル Fortran レコード構造体に含まれる複数のデータ項目の自然なアライメントを行うために、必要に応じてパディングを追加するかどうかを制御します。

- デフォルト (-O2) では -align commons オプションは、必要であればパディング・バイトを追加して共通ブロックのデータを最大 4 バイト境界上にアライメントするよう指定します。

  -align nocommons は、共通ブロックデータのバイトを任意にアライメントします。この場合、COMMON 文中のデータ項目が、サイズの大きな数値データ項目から順番に指定され、最後に文字データが指定されているわけではない限り、アライメントの合っていないデータが生じる可能性があります。

- デフォルト (-O2) では -align dcommons オプションは、必要であればパディング・バイトを追加して共通ブロックのデータを最大 8 バイト境界上にアライメントするよう指定します。

-align nodcommons は、共通データに含まれるデータ項目のバイトを任意にアライメントします。

-align *dcommons* オプションは、共通ブロックを使用するアプリケーションに対して指定します。ただし、アプリケーションにアライメントの合っていないデータが全くない場合や、アライメントの合っていないデータがあってもすべてのデータ項目が 4 バイト以下である場合を除きます。すべてのデータ項目が 4 バイト以下であり、共通ブロックを使用するアプリケーションには、-align dcommons ではなく、-align commons を指定できます。

- -align norecords オプションは、派生型データおよびレコード構造体 (インテル Fortran 拡張機能) に含まれる複数のデータ項目に対し、自然なアライメントではなく、バイト境界上への任意のアライメントを行うよう指定します。デフォルトは -align records です。
- -align records オプションは、レコード構造体 (拡張機能) および SEQUENCE 文のない派生型データに含まれる複数のデータ項目に対し、必要であればパディング・バイトを追加して自然なアライメントを行うよう指定します。
- -align recnbyte オプションは、レコードのフィールドおよび派生型のコンポーネントに対し、指定のサイズバイト境界または自然なアライメントを行う境界のどちらか小さい方のアライメントを行うよう指定します。このオプションは、共通ブロックが自然にアライメントされるか、あるいはパックされるかどうかには影響を与えません。
- -align sequence オプションは、SEQUENCE 文で宣言された派生型コンポーネント (シーケンス・コンポーネント) のアライメントを制御します。

  -align nosequence オプションを指定すると、シーケンス・コンポーネントは他のアライメント規則に関係なくパックされます。-align none は -align nosequence を意味します。

  -align *sequence* オプションを指定すると、現在使用しているアライメント規則がシーケンス・コンポーネントに適用されます。したがって、-align record がデフォルト値なので、コマンドラインで -align sequence のみを指定した場合、これらの派生型のフィールドは自然にアライメントされます。

デフォルトの動作では派生型構造体やレコード構造体に含まれる複数のデータ項目は自然にアライメントされますが、共通ブロックのデータ項目はされません (-align nocommons を使用した -align records)。派生型構造体では SEQUENCE 文を使用すると、-align records はデータ項目を自然にアライメントするために必要なパディング・バイトを追加しません。

シンボルの可視属性オプション

シンボル・プリエンプションまたは位置に依存しないコードが不要なアプリケーションは、汎用の ABI 可視属性を利用してパフォーマンスを向上させることができます。

注 可視性オプションは IA-32 コンパイラと Itanium® コンパイラの両方でサポートされますが、現在のところ、最適化によりパフォーマンスを向上できるのは Itanium ベース・システムのみです。

グローバル・シンボルと可視属性

グローバル・シンボルは、宣言されたコンパイル単位 (単一ソースファイルとそのインクルード・ファイル) の外部で見えるシンボルです。コンパイル単位中の各グローバル・シンボル定義や参照には、それらが定義されたコンポーネントの外部からどのように参照されるかを制御する可視属性があります。次の表に、可視属性の値の定義を示します。

EXTERN	コンパイラはシンボルを別のコンポーネントで定義されたものとして扱います。これは別のコンポーネントにある同じ名前の定義によってシンボルが無効にされる (プリエンプトされる) とコンパイラが仮定することを意味します。(「シンボル・プリエンプション」を参照)。関数シンボルの可視属性が extern の場合、コンパイラはそのシンボルが間接的に呼び出され、間接呼び出しスタブをインライン展開できることを認識しています。
DEFAULT	他のコンポーネントはシンボルを参照することができます。シンボル定義は別のコンポーネントにある同じ名前の定義によって無効に (プリエンプト) されます。
PROTECTED	他のコンポーネントはシンボルを参照することができますが、別のコンポーネントにある同じ名前の定義によってプリエンプトすることはできません。
HIDDEN	他のコンポーネントはシンボルを直接参照することはできません。しかし、そのアドレスは間接的に (例えば、別のコンポーネントの関数への呼び出し引数として、または別のコンポーネントの関数でデータ項目参照にそのアドレスを格納して) 他のコンポーネントに渡すことができます。
INTERNAL	シンボルは、シンボルが定義されたコンポーネントの外部から直接的にも間接的にも参照することはできません。

注
可視性は参照と定義の両方に適用されます。シンボル参照の可視属性は、対応する定義にその可視属性が適用されるというアサーションです。

シンボル・プリエンプションと最適化

共有可能なオブジェクトの関数やデータ項目を使用するときに、それらの代わりに独自に定義したものを使用することができます。例えば、アプリケーションが標準のランタイム・ライブラリの共有可能なオブジェクト `libc.so` を使用するときに、独自のヒープ管理ルーチン `malloc` および `free` の定義を使用することができます。この場合、`libc.so` 内での `malloc` および `free` への呼び出しが、`libc.so` にある定義ではなく独自のルーチンの定義を呼び出すことが重要です。独自の定義は、共有可能なオブジェクト内の定義を無効に *(プリエンプト)* します。

このように共有オブジェクトの項目を再定義する機能は、シンボル・プリエンプションと呼ばれます。ランタイム・ローダがコンポーネントをロードするとき、コンポーネント内の default 可視属性のシンボルはすべて、既にロードされているコンポーネント内の同じ名前のシンボルによるプリエンプションに従います。ただし、メイン・プログラムのイメージは常に最初にロードされるので、メイン・プログラムが定義するシンボルがプリエンプト (つまり再定義) されることはありません。

default 可視属性のシンボルは実行時までメモリアドレスにバインドされないため、シンボル・プリエンプションが発生する可能性により多くのコンパイラの最適化は禁止されます。例えば、default 可視属性のルーチンへの呼び出しは、コンパイル単位が共有可能なオブジェクトにリンクされた場合にプリエンプトされるため、インライン展開することができません。プリエンプト可能なデータシンボルは、名前が異なるコンポーネント中のシンボルにバインドされるため、GP 相対アドレス指定を使用してアクセスすることはできません。GP 相対アドレスはコンパイル時にはわかりません。

シンボル・プリエンプションは、コンパイラの最適化と全く逆の効果であるため、ほとんど使用されていない機能です。この理由のため、コンパイラはすべてのグローバル・シンボル定義をデフォルトでプリエンプト可能ではない (つまり、protected 可視属性) として扱います 。別のコンパイル単位で定義されているシンボルへのグローバル参照は、デフォルトでプリエンプト可能である (つまり、default 可視属性) として仮定されます。すべてのグローバル定義や参照をプリエンプト可能にする場合は、`-fpic` オプションを指定してこのデフォルトを無視してください。

シンボルの可視属性の明示的な指定

インテル® Fortran コンパイラには可視属性オプションが備わっており、可視属性のコマンドライン制御、およびこれら属性の完全な範囲を設定するソース構文が提供され

ます。これらオプションにより、ヘッダファイルの変更に依存せずに、機能に直接アクセスできるようになります。可視オプションによって指定された可視属性が、すべてのグローバル・シンボルに適用されます。シンボルの可視属性を明示的に指定するには、次の 2 つのオプションを使用できます:

```
-fvisibility=keyword
-fvisibility-keyword=file
```

最初の形式は、グローバル・シンボルのデフォルトの可視属性を指定します。2 番目の形式は、1 つのファイル内にあるシンボルの可視属性を指定します (この形式は最初の形式を上書きします)。

file は、可視属性を設定するシンボルの一覧が含まれたファイルのパス名を指定します。各シンボルは余白 (ブランク、タブ、または改行) で区切ります。

どちらのオプションとも keyword には extern、default、protected、hidden、または internal を指定します。各キーワードについては上記の定義を参照してください。

注

可視属性を明示的に指定する際はどちらか 1 つの方法のみを使用してください。可視属性は宣言で使用するか、あるいは *file* 内のシンボル名を指定するかのどちらかにします。両方を同時に使用することはできません。

-fvisibility-keyword=*file* オプションでは、keyword に対する 5 つのコマンドライン・オプションのうちいずれかを使用して、複数のシンボルに同じ可視性を指定します:

```
-fvisibility-extern=file
-fvisibility-default=file
-fvisibility-protected=file
-fvisibility-hidden=file
-fvisibility-internal=file
```

file は、可視属性を設定するシンボル名のリストを含むファイルのパス名です。*file* ファイル中のシンボル名は、ブランク、タブ、または改行で区切ります 。例えば、次のコマンドライン・オプションを使用する場合の例を示します:

```
-fvisibility-protected=prot.txt
```

ここでは prot.txt というファイルに a、b、c、d、e というシンボルが含まれており、これら各シンボルに protected の可視属性が設定されます。 これは、各シンボルの宣言で visibility=protected という属性を指定した場合と同じです。

シンボルファイルを使用しない可視属性の設定 (-fvisibility=keyword)

このオプションは、可視属性リストファイルで指定がなく、宣言に visibilty 属性のないシンボルの可視属性を設定します。シンボル・ファイル・オプションが特に指定されない場合、指定した属性がすべてのシンボルに設定されます。コマンドラインの入力例を次に示します:

```
ifort -fvisibility=protected a.f
```

次のコマンドライン・オプションの 1 つを使用してシンボルのデフォルトの可視属性を設定することができます:

```
-fvisibility=extern
-fvisibility=default
-fvisibility=protected
-fvisibility=hidden
-fvisibility=internal
```

上記のオプションは優先度順に記載されています。したがって、属性構文またはコマンドライン・オプションのどちらかを使用して可視属性を明示的に extern に設定した場合、default、protected、hidden、または internal の各設定を上書きします。また、可視属性を明示的に default に設定した場合には、protected、hidden、または internal の各設定を上書きします。

default の可視属性では、コンパイラがデフォルトのシンボル可視属性を変更して、デフォルトの設定を必要とする関数や変数のデフォルト属性を設定することが可能です。internal はプロセッサ固有の属性なので、この属性に一般的なオプションを使用することは好ましくありません。

次に、コマンドライン・オプションを 2 つ併用した例について検討します。

```
-fvisibility=protected -fvisibility-default=prot.txt
```

ここでは prot.txt というファイル (上記を参照) により、a、b、c、d、e を除くすべてのグローバル・シンボルに protected の可視属性が設定されます。しかし、これらの 5 つのシンボルの可視属性は default に設定されプリエンプト可能になります。

可視属性に関連するオプション

-fminshared

コンパイル単位をメイン・プログラムのコンポーネントとして処理し、共有可能オブジェクトの一部としてリンクしないように指示します。

メイン・プログラムで定義されたシンボルはプリエンプトできないため、これによりコンパイラは default 可視属性で宣言されたシンボルを protected 可視属性であるかのように扱います。つまり、-fminshared は -fvisibility=protected を意味します。コンパイラは位置に依存しないコードをメイン・プログラム用に生成する必要はありません。グローバル・オフセット・テーブル (GOT) のサイズを減らしてメモリのトラフィックを減らす、絶対アドレス指定を使用することができます。

-fpic

完全なシンボル・プリエンプションを指定します。グローバル・シンボル定義は、明示的に他の方法で指定されなければ、グローバル・シンボル参照と同様に default 可視属性 (つまり、プリエンプト可能) になります。位置に依存しないコードを生成します。Itanium® ベース・システムでは、共有オブジェクトをビルドする必要があります。

異なるアプリケーション・タイプの最適化

異なるアプリケーション・タイプの最適化の概要

このセクションでは、コマンドライン・オプション、-O0、-O1、-O2 (または -O)、および -O3 について説明します。-O0 オプションは、最適化を無効にします。他の 3 つのオプションを指定した場合、コンパイラによる最適化が有効になります。これらの最適化の 1 つを指定するには、オプションのより詳細な説明で示されているように、アプリケーションの性質と構造を考慮する必要があります。

一般に、-O1、-O2 (または -O)、および -O3 オプションは、次のように最適化を行います:

-O1: コードのサイズと局所性

-O2 (または -O) : コードの速度 (デフォルト)

-O3: -O2 に加えて、さらに強力な最適化を有効にします

fast:-O3 と -ipo を有効にして、プログラム全体の速度を向上させます。

これらのオプションは、この後で説明されているように多少の違いはありますが、IA-32 アーキテクチャと Itanium® アーキテクチャでほぼ同様に動作します。

-On オプションによる最適化の設定

次の表は、-O0、-O1、-O2、-O3、および -fast オプションの効果を説明したものです。表は最初に IA-32 アーキテクチャと Itanium® アーキテクチャの両方で共通の特性を説明し、次に -On オプションおよび -fast オプションの動作の詳細を（ある場合）アーキテクチャ別に説明します。

オプション	効果
-O0	-On で指定した最適化を無効にします。IA-32 システムではこのオプションを指定すると -fp オプションが設定されます。
-O1	好ましいコードサイズとコードの局所性に最適化します。 ループのアンロールを無効にします。 任意の分岐および実行時間がループ内のコードに支配されない、非常に大きなコードサイズのアプリケーションでパフォーマンスを向上させます。 ほとんどの場合は、-O1 よりも -O2 を推奨します。 IA-32 システム: コードサイズを減らす組込み関数のインライン化を無効にします。速度の最適化を有効にします。組込み関数の認識と -fp オプションを無効にします。 Itanium ベース・システム: ソフトウェアのパイプライン化とグローバル・コード・スケジューリングを無効にします。サーバ・アプリケーションの最適化を有効にします（フラットなプロファイルの直列型コードや分岐型コード）。速度の最適化を有効にし、コードサイズも抑えるようにします。例えば、このオプションを使用すると、ソフトウェアのパイプライン化とループのアンロールが無効になります。
-O2, -O	デフォルトの最適化オプションです。ただし、-g が指定されている場合、-O0 がデフォルトになります。 速度について最適化します。 一般的に推奨される最適化レベルです。-g オプションは、コマンドラインで -g とともに -O2（または -O1 または -O3）が明示的に指定されていなければ、デフォルトの -O2 オプションをオフにして -O0 をデフォルトにします。 IA-32 システムではこのオプションは -O1 オプションと同じ動作をします。 Itanium ベース・システム: 速度の最適化を有効にします。これにはグローバル・コード・スケ

	ジューリング、ソフトウェアのパイプライン化、プレディケーション、およびスペキュレーションも含まれます。 これらのシステム上で -O2 オプションを使用すると、組込み関数のインライン展開が有効になります。また、パフォーマンスの向上のために次の機能も有効にします: 定数伝播、コピー伝播、不要コードの排除、グローバル・レジスタ割り当て、グローバル命令スケジューリング、スペキュレーション・コントロール、ループのアンロール、コード選択の最適化、部分冗長の排除、ストレングス・レダクション/誘導変数の簡略化、変数名の変更、例外処理の最適化、末端再帰、ピープホールの最適化、構造体代入の低下および最適化、不要ストアの排除の各機能が使用されます。
-O3	-O2 の最適化を有効にし、それに加えてプリフェッチ、スカラ置換、ループ変換、およびメモリのアクセス変換のような、さらに強力な最適化も有効にします。最大実行速度に最適化を行いますが、ループ変換およびメモリアクセス変換が行われない限り、パフォーマンスが向上しない場合があります。-O3 の最適化はコードによっては -O2 の最適化よりも遅くなります。このオプションは、浮動小数点演算を多く使用するループや大きなデータセットを処理するループを含むアプリケーションに推奨します。 IA-32 システム: -ax{K|W|N| B|P} または -x{K|W|N|B|P} オプションと組み合わせると、コンパイラは -O2 よりも詳細にデータの依存性を分析します。そのためコンパイル時間が長くなる場合があります。 Itanium ベース・システムでは、技術計算を行うアプリケーション (ループを多用するコード) の最適化、つまりループとデータ・プリフェッチの最適化を有効にします。

-fast	このオプションは、ランタイム・パフォーマンスを向上させる最適化機能一式をすべて有効にする、簡単な方法です。このオプションを指定すると、ランタイム・パフォーマンスの向上につながる次の各オプションが設定されます。 -O3: 最高速で高レベルの最適化 (上記参照) -ipo: 複数ファイルにわたるプロシージャ間の最適化を有効にします -static: 共有ライブラリとリンクしないようにします IA-32 およびインテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムでは、-fast でこれらのオプションと -xP が設定されます。 いくつかの重要なコンパイラ最適化機能を一度に設定するよう指示します。-fast によって設定されたオプションを上書きするには、コマンドラインから -fast オプションの後に使用するオプションを指定してください。 -fast で設定されるオプションは、バージョンによって変更になる場合があります。 IA-32 システム: -ax{K|W|N| B|P} または -x{K|W|N|B|P} オプションと合わせてこのオプションを使用すると、最高のランタイム・パフォーマンスが得られます。

最適化の制限

最適化の範囲を絞ったり、最適化を禁止したりする場合は、以下の各オプションを使います。

-O0	最適化を無効にします。-fp オプションを有効にします。
-g	コマンドラインで -g とともに、-O2 (または -O1 または -O3) が明示的に指定されていなければ、デフォルトの -O2 オプションをオフにして -O0 をデフォルトにします。「最適化とデバッグ」を参照してください。
-mp	浮動小数点演算の精度にいくらかの正と負の影響をもたらす

	最適化を制限し、宣言された精度のレベルを保ち浮動小数点演算が ANSI および IEEE* 標準に準拠するようにします。詳細は、「-mp オプション」を参照してください。
-nolib_inline	組込み関数のインライン展開を無効にします。

最適化を制限する詳しい方法については、「-ip と -Qoption 指定子の併用」を参照してください。

浮動小数点演算の最適化

IA-32 アーキテクチャと Itanium® アーキテクチャのオプション

このセクションで説明されているオプションはすべて、IA-32 および Itanium® アーキテクチャの浮動小数点 (FP) 演算のさまざまな精度での最適化を提供します。

-mp1 (IA-32 のみ) および -mp オプションは、浮動小数点精度を向上しますがアプリケーションのパフォーマンスに影響します。 これらのオプションについての詳細は、「浮動小数点演算の精度の向上と制限」を参照してください。

FP オプションは、浮動小数点演算のさまざまな精度の最適化を提供します。これらの最適化を無効にするには、-O0 オプションを指定します。

-mp オプション

-mp オプションで、最適化を制限し、宣言された精度を維持します。例えば、インテル® Fortran コンパイラでは、分母の逆数によって浮動小数点数の除算計算を乗算に変えられます。ただし、これによって、浮動小数点数の除算計算の結果が多少変わることがあります。-mp スイッチを使用すると、実行速度が多少遅くなる場合があります。詳細は、「浮動小数点演算の精度の向上と制限」を参照してください。

-mp1 オプション

-mp1 オプションは、-mp オプションよりもパフォーマンスへの影響が少なく、浮動小数点精度を宣言された精度に近づけるために制限します。このオプションでは、超越関数のオペランドの範囲外チェックを実現し、浮動小数点演算の比較精度を向上させます。

デノーマル値をゼロにフラッシュする (-ftz[-])

`-ftz[-]` オプションは、アプリケーションが漸次アンダーフロー・モードの場合に、デノーマル結果をゼロにフラッシュします。`-ftz` でデノーマル値をフラッシュすると、アプリケーションのパフォーマンスが向上する場合があります。

デフォルトの状態では、`-ftz-` はオフです。デフォルトでは、コンパイラは結果を漸次アンダーフローにします。デフォルトの `-O2` オプションでは、`-ftz[-]` がオフになっています。

Itanium ベース・システム上の -ftz[-]

Itanium ベース・システムの場合のみ、`-O3` オプションを使用すると `-ftz` がオンになります。

`-ftz` オプションにより、プログラムの数値動作で好ましくない結果が出力された場合、次のようにコマンドラインで `-ftz-` を使用して FTZ (ゼロ・フラッシュ) モードをオフに設定できます。これにより、`-O3` 最適化の利点をそのまま活用できます。

`ifort -O3 -ftz-` *`myprog.f`*

使用方法:

- このオプションは、デノーマル値がアプリケーション動作に影響を与えない場合に使用します。
- FTZ モードをオンにするには、`-ftz[-]` オプションを `main` プログラムが含まれているソースに対してのみ使用する必要があります。初期スレッドおよびそのプロセスによってその後に作成されるあらゆるスレッドは、FTZ モードで動作します。

`-ftz[-]` オプションは浮動小数点アンダーフローの結果に対して次のような影響を与えます:

- ゼロへの*漸次*アンダーフロー (gradual underflow) における `-ftz-` の結果: 浮動小数点アンダーフローの結果は正規化されていない数またはゼロになります。
- ゼロへの*突発*アンダーフロー (abrupt underflow) における `-ftz` の結果: 浮動小数点アンダーフローの結果はゼロに設定され、プログラムはそのまま実行されます。`-ftz` は演算でゼロとして扱われる、正規化されていない値も生成するため、無効な浮動小数点例外は発生しません。Itanium ベース・システムの場合、`-O3` オプションを使用すると突発アンダーフロー (abrupt underflow) がゼロに設定されます (`-ftz` がオン)。より低い最適化レベルでは、

ゼロへの漸次アンダーフロー(gradual underflow) が Itanium ベース・システムのデフォルトになります。

IA-32 システムでは、-ftz で突発アンダーフロー (abrupt underflow) を設定すると SSE/SSE2 命令のパフォーマンスが向上することがありますが、x87 命令のパフォーマンスや数値的動作には影響を与えません。したがって -ftz は、比較的新しい IA-32 インテル・プロセッサの命令を有効にする -x または -ax オプションを選択しない限り効果がありません。

Itanium ベース・プロセッサでは、ゼロへの漸次アンダーフロー (gradual underflow) が発生するとパフォーマンスが低下します。高い最適化レベルを使用してデフォルトを突発アンダーフロー (abrupt underflow) にするか、明示的に、-ftz を設定することで、パフォーマンスを改善することができます。
-ftz を使用すると、特に単精度のコードの場合、実際にアンダーフローが発生しなくても、Itanium 2 プロセッサのパフォーマンスが向上することがあります。

浮動小数点例外処理の使用 (-fpen)

-fpen*n* オプションを使用して、例外の処理を制御します。 -fpen*n* オプションは、*n* の値に応じて浮動小数点例外を制御します。

浮動小数点例外には次のような種類があります。

- 浮動小数点オーバーフロー: 演算の結果が浮動小数点データ型よりも大きすぎた場合、例外値 (適切な正または負の無限大) になります。例えば、1E30 * 1E30 は単精度の浮動小数点値をオーバーフローし、結果は正の無限大に、-1E30 * 1E30 の結果は負の無限大になります。
- ゼロによる浮動小数点除算: 演算が 0.0 / 0.0 の場合、結果は、演算が成功しなかったことを意味する例外値 NaN (Not a Number、非数) になります。分子が 0.0 ではない場合、結果は符号付きの無限大になります。
- 浮動小数点アンダーフロー: 演算の結果が浮動小数点データ型よりも小さすぎた場合。各浮動小数点型 (32 ビット、64 ビット、および 128 ビット) には、精度が少し失われることを表す非常に小さな正規化されていない範囲があります。例えば、正規化されている単精度浮動小数点値の下限は約 1E-38 で、正規化されていない単精度浮動小数点値の下限は約 1E-45 です。1E-30 / 1E10 は正規化されている範囲ではアンダーフローになりますが、正規化されていない範囲ではアンダーフローにならないため、結果は正規化されていない例外値 1E-40 になります。1E-30 / 1E30 はすべての範囲でアンダーフローになるため、結果はゼロになります。これは、0 (ゼロ) への漸次アンダーフローと呼ばれます。
- 無効な浮動小数点: 演算の入力として例外値 (符号付き無限大、NaN、正規化されていない数) が使用されると、結果も NaN になります。

-fpe*n* オプションを使用すると、メイン・プログラムのランタイム時に処理される浮動小数点例外の結果をある程度制御することができます。

- -fpe0 は、浮動小数点例外を次のように制限します:
  - 浮動小数点オーバーフロー、ゼロによる浮動小数点除算、および無効な浮動小数点が発生した場合、プログラムはエラー・メッセージを表示して終了します。
  - 浮動小数点アンダーフローが発生した場合、結果はゼロに設定され、プログラムはそのまま実行されます。これは、0 (ゼロ) への突発アンダーフローと呼ばれます。
- -fpe1 は、浮動小数点アンダーフローのみを制限します:
  - 浮動小数点オーバーフロー、ゼロによる浮動小数点除算、および無効な浮動小数点が発生した場合、結果は例外値 (NaN および符号付き無限大) になり、プログラムはそのまま実行されます。
  - 浮動小数点アンダーフローが発生した場合、結果はゼロに設定され、プログラムはそのまま実行されます。
- デフォルトでは、IA-32 アーキテクチャと Itanium アーキテクチャの両方で -fpe3 が使用されます。このオプションは、浮動小数点例外動作を制限しません。
  - 浮動小数点オーバーフロー、ゼロによる浮動小数点除算、および無効な浮動小数点が発生した場合、結果は例外値 (NaN および符号付き無限大) になり、プログラムはそのまま実行されます。
  - 浮動小数点アンダーフローは漸次アンダーフローで、結果はゼロになるまで正規化されていない値になります。

-fpe*n* オプションは、Fortran メイン・プログラムにのみ影響します。Fortran メイン・プログラムによって設定された浮動小数点例外動作は、全プログラムの実行に影響します。メイン・プログラムが Fortran ではない場合、ユーザは Fortran 組込み関数 FOR_SET_FPE を使用して浮動小数点例外動作を設定することができます。

プログラムに含まれる各ルーチンを別々にコンパイルする際は、-fpe*n* オプションで同じ n 値を使用してください。

詳細については、『Linux* 版インテル® Fortran コンパイラ・ユーザーズ・ガイド Vol I』の「浮動小数点例外の制御」を参照してください。

IA-32 システムの浮動小数点演算の精度

-prec_div オプション

インテル® Fortran コンパイラでは、分母の逆数によって浮動小数点数の除算計算を乗算に変えられます。-prec_div を使用して浮動小数点除算から乗算へ変換する最適化処理を無効にして、より正確な除算結果を取得します。速度に少し影響します。

-pc{32|64|80} オプション

-pc{32|64|80} オプションは、浮動小数点数の仮数部の精度を制御するために使用します。特定の IA-32 システムや Itanium® ベースのシステム用に作成した一部の浮動小数点アルゴリズムでは、浮動小数点数値の仮数部または小数部の精度が重要となります。このオプションの各選択肢を使用すると、次に示すように仮数部がそれぞれのビット数に丸められます。

-pc32: 24 ビット (単精度)

-pc64: 53 ビット (倍精度)

-pc80: 64 ビット (拡張精度)

デフォルトは、-pc80 (倍精度) です。

このオプションでは、完全な最適化を実行できます。このオプションで影響されるのは浮動小数点値の小数部だけなので、-mp オプションとは異なり、パフォーマンスに悪影響は与えません。指数の部分は影響を受けません。

注

このオプションは、コンパイルされるモジュールがメイン・プログラムを含むときのみ効果があります。

警告

デフォルトの精度制御方式または丸めモードを変更すると (例えば、-pc32 オプションの使用やユーザの介入によって)、いくつかの算術関数で返された結果に影響する場合があります。

丸め制御 (-rcd、-fp_port)

インテル Fortran コンパイラは -rcd オプションを使用して、浮動小数点から整数への変換において、丸めモードの変更を無効にします。

デフォルトでは、浮動小数点の丸めモードは「最近値への丸め」となっています。つまり、各値は浮動小数点の計算中に丸められます。しかし、Fortran 言語では、浮動小数点数値は整数への変換時に切り捨てられる必要があります。これを行うために、コンパイラは各浮動小数点の変換前に一度丸めモードを切り捨てに変更し、後で元に戻さなければなりません。

-rcd オプションは、浮動小数点数から整数への変換を含むすべての浮動小数点計算において、丸めモードから切り捨てモードへの変更を無効にします。このオプションをオンにするとパフォーマンスは向上できますが、整数への浮動小数点の変換が Fortran のセマンティクスに準拠しなくなります。

-fp_port オプションを使用して、代入とキャストの際に浮動小数点の結果を丸めることもできます。このオプションは処理速度に影響を与えることがありますが、確実に代入時にユーザ定義の精度への丸めが行われます。-mp1 オプションは、-fp_port の指定が含まれます。

Itanium® ベース・システムの浮動小数点演算の精度

次のインテル® Fortran コンパイラ・オプションは、Itanium® ベース・システム上で浮動小数点計算用にコンパイラの最適化を制御できるようにします。

浮動小数点積和/積差演算の縮約

-IPF_fma[-] は、浮動小数点積和/積差演算の 1 つの演算への縮約を有効/無効 にします。-mp が指定されない限り、コンパイラは可能な限りこれらの演算を縮約します。-mp オプションは、縮約を無効にします。

-IPF_fma および -IPF_fma- は、デフォルトのコンパイラ動作を無効にするのに使用されます。例えば、-mp と -IPF_fma をともに使用すると、コンパイラは縮約演算を有効にします。

```
ifort -mp -IPF_fma myprog.f
```

FP スペキュレーション

`-IPF_fp_speculation`*mode* は、次のいずれかのモード (*mode*) で、浮動小数点演算のスペキュレーションを設定します:

fast: 浮動小数点演算のスペキュレーションを有効にします (デフォルト)。

safe: 安全な場合のみ浮動小数点演算のスペキュレーションを有効にします。

strict: すべての状況で浮動小数点ステータスを保持する浮動小数点演算をスペキュレーションします。現在のバージョンでは、このモードは浮動小数点演算のスペキュレーションを無効にします(`off` と同じ)。

off: 浮動小数点演算のスペキュレーションを無効にします。

FP 数値演算関数の最適化

–IPF_fp_relaxed[–] は、処理速度を速くする [無効にする] ことができますが、除算や平方根のような数値演算関数においてコード・シーケンスの精度が少し低くなります。厳密な IEEE* 精度と比較した場合、このオプションを使用すると、そのような関数による浮動小数点演算精度が少し低くなります (通常は、最下位の桁数に制限されます)。デフォルトは `-QIPF_fp_relaxed-` です。

FP 演算の評価

`-IPF_flt_eval_method{0|2}` オプションは、次のように浮動小数点オペランドを含む計算式の評価をコンパイラに指示します:

`-IPF_flt_eval_method0` は、プログラムで宣言された変数型により指定された精度で、浮動小数点式の演算子に関する表現を評価するようにコンパイラに指示します。

`-IPF_flt_eval_method2` は、現在のバージョンではサポートされていません。

FP 結果の精度制御

`-IPF_fltacc` は、浮動小数点の精度に影響を与える最適化を無効にします。デフォルトはそのような最適化を有効にする `-IPF_fltacc-` です。

Itanium コンパイラは、浮動小数点の式を再割り当てしてアプリケーションのパフォーマンスを向上する場合があります。-IPF_fltacc または -mp を使用して、これらの浮動小数点の最適化を無効にしたり、あるいは制限します。

浮動小数点演算の精度の向上と制限

-mp および -mp1 オプションは、それぞれ浮動小数点精度を維持および制限しますがアプリケーションのパフォーマンスに影響します。-mp1 オプションは、-mp オプションよりもパフォーマンスに及ぼす影響は少ないです。-mp1 では、超越関数のオペランドの範囲外チェックを実現し、浮動小数点演算の比較精度を向上させます。IA-32 システムの場合、-mp オプションは -mp1 を含意し、-mp1 オプションは -fp_port を含意します。 -mp を使用するとパフォーマンスが最も低下し、-fp_port を使用するとパフォーマンス低下を最小限に抑えられます。

-mp オプションを指定すると、精度は宣言された水準が保たれます。また、浮動小数点演算の処理は、ANSI および IEEE* 標準にほぼ準拠する結果となります。このオプションを使用すると、メモリへの頻繁な保存や、または一部のデータのレジスタ割り当て失敗が発生します。インテル® アーキテクチャは、 通常、浮動小数点結果をレジスタに維持します。これらのレジスタは、80 ビット長で、倍精度の数よりも大きな精度を保ちます。結果をメモリに保存する必要がある場合は、丸めが起こります。これで、「予測される」精度の結果を得ることができますが、速度が遅くなります。 -pc{32|64|80} オプション (IA-32 のみ) は、さまざまなプロセッサの IEEE フラグに合わせて、浮動小数点の精度および丸めを制御するのに使用されます。

ほとんどのプログラムでは、-mp オプションを指定するとパフォーマンスの低下につながります。目的のアプリケーションにこのオプションが必要かどうかよくわからない場合は、このオプションを指定した場合と指定しない場合で実際にプログラムをコンパイルし、実行して、パフォーマンスと精度に対する効果を評価してみてください。

このオプションを指定すると、プログラムのコンパイルに次の影響が生じます:

- IA-32 システム上で、浮動小数点型として宣言された浮動小数点ユーザ変数は、レジスタに割り当てられません。
- Itanium® ベース・システムでは、浮動小数点ユーザ変数は、レジスタに割り当てられることがあります。式は、ソース・オペランドの精度を使用して評価されます。コンパイラは、積和/積差演算を 1 つの演算命令で実行する浮動小数点積和 (FMA) 命令を使用しません。これは、-IPF_fma オプションを使用して有効にできます。コンパイラは、コンピュータの浮動小数点演算の状態に影響を及ぼすような浮動小数点演算のスペキュレーションを行いません。詳細については、「Itanium® ベース・システムの浮動小数点演算の精度」を参照してください。

- 浮動小数点演算比較は、IEEE 754 に準拠します。
- 演算はコードで指定したとおりに実行されます。例えば、除算が逆数の乗算に変換はされません。
- コンパイラは関連付けを変更せずに、指定された順序で浮動小数点演算を実行します。
- コンパイラは浮動小数点値に関する定数保持による最適化を行いません。定数保持の最適化では、1 による乗算、1 による除算、0 の加算、あるいは 0 の減算も除外されます。例えば、ある数値に 0.0 を加算するコードはそのとおりに実行します。また、コンパイル時の浮動小数点演算も実行しません。これは、浮動小数点例外についてもその状態を変えないようにするためです。

  IA-32 システムでは、式がスピルする場合、64 ビット (DOUBLE PRECISION) ではなく、80 ビット (extended precision) としてスピルされます。浮動小数点演算は、IEEE 754 に準拠します。REAL 型および DOUBLE PRECISION 型への代入を行うと、その精度は、80 ビット (extended) から 32 ビット (REAL) か 64 ビット (DOUBLE PRECISION) に丸められます。`-O0` を指定しなければ、精度が丸められずに変数が再使用される場合もあります。

- `-x{K|W|N|B|P}` オプションによってベクトル化が有効になっている場合でも、コンパイラはリダクション・ループ (ドット積を計算するループ) および精度型が混在しているループはベクトル化しません。同様に、コンパイラは特定のループ変換を有効にしません。例えば、コンパイラは、部分和またはループ交換を実行するためにリダクション・ループを変換しません。

特定のプロセッサの最適化

特定のプロセッサの最適化の概要

ここでは、対象とするプロセッサの指定、およびプロセッサ・ディスパッチと拡張命令のサポート・オプションについて説明します。

`-tpp{5|6|7}` オプションは、IA-32 プロセッサ用に最適化し、オプション `-tpp{1|2}` は、Itanium® プロセッサ・ファミリ用に最適化します。`-x{K|W|N|B|P}` と `-ax{K|W|N|B|P}` の各オプションは、特定のプロセッサの拡張命令に対応する専用コードを生成します。

`-x` オプションおよび `-ax` オプションの `N/W` または `K` を使用して古いプロセッサ用にコードを最適化した場合でも、インテル® Pentium® M プロセッサやストリーミング

SIMD 拡張命令 3 (SSE3) をサポートするインテル Pentium 4 プロセッサのような最新プロセッサ・ベース・システム上でもアプリケーションを実行できます。

対象とするプロセッサの指定 (-tpp{n})

-tpp{n} オプションは、特定のインテル® プロセッサ向けにアプリケーションのパフォーマンスを最適化できます。このオプションを使用すると、指定したバージョンに関連付けられたプロセッサ用にチューニングされたコードが生成されます。例えば、-tpp7 オプションを指定すると、インテル® Pentium® 4 プロセッサ、インテル® Xeon™ プロセッサ、インテル Pentium M プロセッサ、および ストリーミング SIMD 拡張命令 3 (SSE3) をサポートするインテル Pentium 4 プロセッサ用に最適化したコードが生成され、-tpp2 オプションを指定すると、インテル® Itanium® 2 プロセッサ用に最適化したコードが生成されます。

-tpp{n} オプションは常に同じファミリのインテル・プロセッサと下位互換のあるコードを生成します。したがって、-tpp7 を使用して生成したコードは Pentium Pro プロセッサまたは Pentium III プロセッサでも正しく動作しますが、-tpp6 で生成したコードに比べて、これらのプロセッサ上での実行速度が多少遅くなります。同様に、-tpp2 で生成したコードは Itanium プロセッサでも正しく動作しますが、-tpp1 で生成したコードに比べて実行速度が多少遅くなります。

IA-32 システム用のプロセッサ

-tpp5、-tpp6、および -tpp7 の各オプションは、次の表にリストされている特定のインテル IA-32 プロセッサ用にアプリケーションのパフォーマンスを最適化します。最適化を行ったバイナリは、下記の表で記述されているあらゆるプロセッサ上で正しく動作します。

オプション	最適化の対象となるプロセッサ
-tpp5	インテル Pentium プロセッサ、および MMX® テクノロジ Pentium® プロセッサ
-tpp6	インテル Pentium Pro プロセッサ、Pentium II プロセッサ、および Pentium III プロセッサ
tpp7 (デフォルト)	インテル Pentium 4 プロセッサ、インテル Xeon プロセッサ、インテル Pentium M プロセッサ、およびストリーミング SIMD 拡張命令 3 (SSE3) をサポートするインテル Pentium 4 プロセッサ

例

次の各コマンドは、ソース・プログラム `prog.f` から、デフォルトでインテル Pentium 4 プロセッサおよびインテル Xeon プロセッサ向けに最適化されたコンパイル済みバイナリを 1 つ生成します。これらの結果は Pentium、Pentium Pro、Pentium II、および Pentium III プロセッサ上でも実行することができます。

```
ifort prog.f

ifort -tpp7 prog.f
```

ただし、インテル Pentium プロセッサおよび MMX テクノロジ Pentium プロセッサ用にアプリケーションを最適化する場合、次のように `-tpp5` オプションを使用します。

```
ifort -tpp5 prog.f
```

Itanium ベースのシステム用のプロセッサ

`-tpp1` および `-tpp2` の各オプションは、次の表にリストされている特定のインテル IA-32 プロセッサ用にアプリケーションのパフォーマンスを最適化します。最適化を行ったバイナリは、下記の表で記述されている両方のプロセッサ上で正しく動作します。

オプション	最適化の対象となるプロセッサ
`-tpp 1`	インテル Itanium プロセッサ
`-tpp 2` (デフォルト)	インテル Itanium 2 プロセッサ

例

次の各コマンドは、ソース・プログラム `prog.f` から、デフォルトでインテル Itanium 2 プロセッサ向けに最適化されたコンパイル済みバイナリを 1 つ生成します。この結果は Itanium プロセッサ上でも実行することができます。

```
ifort prog.f

ifort -tpp2 prog.f
```

ただし、インテル Itanium プロセッサ向けのアプリケーションを対象にする場合、`-tpp 1` オプションを使用します:

```
ifort -tpp1 prog.f
```

プロセッサ固有の最適化 (IA-32 のみ)

-x{K|W|N|B|P} オプションは、プログラムを特定のインテル® プロセッサ向けに最適化します。生成されるコードには、他のプロセッサでサポートされない機能の無条件の使用が含まれることがあります。

オプション	最適化の対象
-xK	インテル® Pentium® III プロセッサおよび互換性のあるインテル・プロセッサ。
-xW	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。
-xN	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて新しい最適化も有効にします。
-xB	インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ。メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。
-xP	ストリーミング SIMD 拡張命令 3 (SSE3) をサポートするインテル Pentium 4 プロセッサ。メイン・プログラムがこのオプションを使用してコンパイルされると、プログラムは互換性のないプロセッサを検出して、実行時にエラー・メッセージを出力します。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。

インテル以外から提供されている x86 プロセッサ上でプログラムを実行する場合は、-x{K|W|N|B|P} オプションを指定しないでください。

例

次の呼び出しは、インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ向けに myprog.f をコンパイルします。生成されるバイナリは、Pentium プロセッサ、Pentium Pro プロセッサ、Pentium II プロセッサ、Pentium III プロセッサ、MMX® テクノロジ Pentium® プロセッサ、およびインテル以外から提供されている x86 プロセッサ上では正常に実行されない可能性があります。

```
ifort -xN myprog.f
```

注

-x{K|W|N|B|P} を使用してコンパイルされたプログラムを互換性のないプロセッサ上で実行すると、不正な命令例外や他の予期しない動作が発生することがあります。-xN、-xB、または -xP を使用してコンパイルされたプログラムを、サポートされていないプロセッサ (上の表を参照) 上で実行すると、次のランタイム・エラーが表示されます:

```
Fatal error: This program was not built to run on the
processor in your system.
```

プロセッサ固有の自動最適化 (IA-32 のみ)

-ax{K|W| N|B|P} オプションは、指定されたインテル® プロセッサ固有の機能を利用する関数バージョンを生成できるかどうかを確認するようにコンパイラに命令します。コンパイラはこのような箇所を検出すると、プロセッサ固有に生成する関数がパフォーマンスを向上するかどうかを検証します。パフォーマンスが向上すると判明した場合、コンパイラはプロセッサ固有のバージョンと汎用バージョンの関数を生成します。汎用バージョンは、どの IA-32 プロセッサ上でも実行することができます。

プログラムの実行時に、使用しているインテル・プロセッサに応じて、この 2 つのバージョンのどちらを実行するか選択されます。このため、プログラムは以前の IA-32 プロセッサでも正常に動作しながら、新しいインテル・プロセッサにおいてパフォーマンスの大幅な向上を実現できます。

ただし、-ax{K|W|N|B|P} を使用した場合には、次のような欠点があります:

- プロセッサ固有のコード・バージョンと汎用のコード・バージョンの両方が含まれるため、コンパイルされたバイナリのサイズが大きくなります。
- 使用するコードを決定するランタイム・チェックによって、パフォーマンスに影響があります。

注

この方法で特定のプロセッサ向けに最適化したアプリケーションは、あらゆるインテル IA-32 プロセッサ上で実行できます。-x と -ax オプションの両方を指定した場合、-x オプションで指定されたプロセッサの種類と互換性のあるプロセッサでのみ実行できる汎用コードが生成されます。

オプション	最適化の対象となるプロセッサ
-axK	インテル® Pentium® III プロセッサおよび互換性のあるインテル・プロセッサ。
-axW	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。
-axN	インテル Pentium 4 プロセッサおよび互換性のあるインテル・プロセッサ。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。
-axB	インテル Pentium M プロセッサおよび互換性のあるインテル・プロセッサ。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。
-axP	ストリーミング SIMD 拡張命令 3 (SSE3) をサポートするインテル Pentium 4 プロセッサ。このオプションは、インテル・プロセッサ固有の最適化に加えて、新しい最適化も有効にします。

例

下記のコンパイルは、次のような 1 つの実行ファイルを生成します:

- すべての IA-32 プロセッサで使用できる汎用バージョン
- パフォーマンスが向上する場合、Pentium 4 プロセッサ用に最適化されたバージョン
- パフォーマンスが向上する場合、Pentium M プロセッサ用に最適化されたバージョン

```
ifort -axNB prog.f90
```

プロセッサ固有のランタイム・チェック (IA-32 システム)

インテル® Fortran コンパイラの最適化はランタイム時に効果を発揮します。IA-32 システムでは、コンパイラはメインルーチン中に記述されているランタイム・チェックを実行するコード・セグメントを挿入して、プロセッサ固有の最適化を強化します。

-xB、-xB、または -xP による対応プロセッサのチェック

実行エラーを防ぐために、コンパイラはメインルーチン中に正しいプロセッサが使用されていることをチェックするコードを挿入します。-xN、-xB、または -xP オプションを使用してコンパイルされたプログラムは、インテル® Pentium® 4 プロセッサ、インテル

Pentium M プロセッサ、ハイパー・スレデッィング (HT) テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応)、またはこれらと互換性のあるインテル® プロセッサ上で実行されているかどうか、ランタイム時にチェックされます。プログラムがこれらのいずれかのプロセッサ上で実行されていない場合、プログラムはエラーを出力して終了します。

例

プログラム `foo.f90` を HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 用に最適化するには、次のコマンドを使用します。

```
ifort -xP foo.f90 -o foo.exe
```

`foo.exe` が HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) と互換性のないプロセッサ (例えば、インテル Pentium 4 プロセッサ) 上で実行されると、プログラムは終了します。

プログラムを複数の IA-32 プロセッサ上で実行する場合は、プロセッサ固有の最適化を行う `-x` オプションではなく、プロセッサ固有のコードと汎用コードを生成する `-ax` オプションを使用してください。

FTZ と DAZ フラグの設定

これまで、IA-32 プロセッサでは FTZ (ゼロ・フラッシュ) と DAZ (denormals are zeros) フラグはデフォルトでオフになっていました。これらのフラグをオンにすると、IEEE 準拠ではなくなりますが、最新の IA-32 プロセッサ上で実行される漸次アンダーフロー・モードでデノーマルな浮動小数点値が使用され、プログラムのパフォーマンスが大幅に向上します。このため、インテル Pentium III プロセッサ、インテル Pentium 4 プロセッサ、インテル Pentium M プロセッサ、HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応)、および互換性のある IA-32 プロセッサの場合、コンパイラはデフォルトでこれらのフラグをオンにするように変更されました。コンパイラは、プログラムがこれらのインテル・プロセッサのいずれかで実行されていることを確認するため、プロセッサのランタイム・チェックを実行するコードをプログラム中に挿入します。

- インテル Pentium III プロセッサ上でプログラムを実行すると、FTZ フラグは有効になりますが、DAZ フラグは有効になりません。
- インテル Pentium M プロセッサまたは HT テクノロジ インテル Pentium 4 プロセッサ (SSE3 対応) 上でプログラムを実行すると、FTZ と DAZ の両方のフラグが有効になります。

これらのフラグは、サポートが確認されたインテル・プロセッサによってのみオンにされます。

インテル以外のプロセッサの場合、次のインテル Fortran 組込み関数を使用してフラグを手動で設定することができます:

```
RESULT = FOR SET FPE (FOR M ABRUPT UND)
```

プロシージャ間の最適化 (IPO)

プロシージャ間の最適化の概要

プロシージャ間の最適化 (IPO) を有効にするには、-ip と -ipo を使用します。IPO を実行すると、コンパイラがコードを分析し、次の表に一覧表示されている最適化の中から、ユーザにメリットのある最適化を判断します。

IA-32 および Itanium® ベース・アプリケーション

最適化項目	影響を受ける部分
関数のインライン展開	呼び出し、ジャンプ、分岐、ループ
プロシージャ間での定数伝播	引数、グローバル変数、戻り値
モジュール・レベルでの静的変数の監視	現状以上の最適化、ループ不変コード
不要コードの排除	コードのサイズ
関数特性の伝播	呼び出し命令の削除と呼び出し命令の移動
マルチファイルの最適化	-ip と同じですが、複数のファイル全体にわたって最適化を行います。

IA-32 アプリケーションのみ

最適化項目	影響を受ける部分
レジスタ内での引数の受け渡し	呼び出し、レジスタの使用
ループ不変コードの移動	現状以上の最適化、ループ不変コード

関数のインライン展開は、プロシージャ間の最適化機構によって実行する主な最適化機能のうちの 1 つです。コンパイラは、頻繁に実行する関数呼び出しが存在している

と判断した場合、その呼び出し命令を、当該関数自体のコードに置き換えることがあります。

-ip オプションを指定すると、現在のソースファイル内で定義しているプロシージャ内での呼び出し命令について関数のインライン展開を実行します。ただし、-ipo オプションを使用してマルチファイル IPO を指定すると、別々のファイル内で定義しているプロシージャ内での呼び出し命令について関数のインライン展開が実行されます。

IPO の最適化を無効にするには、-O0 オプションを使用します。

注

-ip、-ipo のどちらのオプションを指定した場合も、条件によってはコンパイル時間とコードサイズが著しく増える場合があります。

Itanium ベース・システムの -auto_ilp32 オプション

Itanium ベース・システムで -auto_ilp32 オプションを使用するには、プログラム全体にわたるプロシージャ間の分析が必要です。32 ビットアドレス空間を超えることができない (32 ビットポインタを使用する) アプリケーションを指定します。32 ビットアドレス空間 (2^32) を超えることができるプログラムで -auto_ilp32 オプションを使用すると、プログラム実行中に予期しない問題が発生することがあります。

この最適化にはプログラム全体のプロシージャ間分析が必要なので、-auto_ilp32 オプションと -ipo オプションの両方を使用しなければなりません。

インテル® エクステンデッド・メモリ 64 テクノロジ (インテル® EM64T) システムでは、-xP または -axP を指定しない限り、-auto_ilp32 は何の効果もありません。

IPO コンパイル・モデル

このセクションのトピックでは、IPO という言葉はマルチファイル IPO のことを指します。

-ipo オプションを使用した場合、コンパイラはプログラムの個々のプログラム・モジュールから情報を収集します。コンパイラはこの情報を使用して、複数のモジュール全体にわたって最適化を行います。この最適化を行うには、-ipo オプションをコンパイル・フェーズとリンケージ・フェーズの両方で使用します。

IPO の主な利点の 1 つは、より多くのインライン展開を有効にできることです。インライン展開およびインライン展開の最低基準については、「関数のインライン展開の基準」および「ユーザ関数のインライン展開の制御」を参照してください。インライン展開

およびその他の最適化は、プロファイル情報によって向上します。プロファイル情報を用いたマルチファイル IPO を実行する方法は、「プロファイルに基づく最適化の例」を参照してください。

コンパイル・フェーズ

各ソースファイルがコンパイルされるたびに、コンパイラはソースコードの中間表現 (IR) を擬似オブジェクト・ファイルに格納します。このオブジェクト・ファイルには、最適化に使うサマリ情報が含まれています。

特に指定しない限り、コンパイラは、マルチファイル IPO のコンパイル・フェーズの最中に、擬似オブジェクト・ファイルをいくつか生成します。実際のオブジェクト・ファイルの代わりに擬似オブジェクト・ファイルを生成すると、マルチファイル IPO のコンパイル・フェーズに要する時間が短くなります。各擬似オブジェクト・ファイルは、対応するソースファイルの IR を含みますが、実コードもデータも含みません。この擬似オブジェクトは、`ifort` で `-ipo` オプションを使用するか `xild` ツールを使用してコンパイラにリンクする必要があります。(「xild を使用したマルチファイル IPO 実行ファイルの生成」を参照してください。)

> **注**
> `ifort` および `-ipo` または `xild` を使用して擬似オブジェクト・ファイルをリンクしないとリンケージ・エラーが発生します。場合によっては、擬似オブジェクト・ファイルを使用できない場合があります。詳細については、「実際のオブジェクト・ファイルの生成」を参照してください。

リンケージ・フェーズ

リンカを起動する際に、コマンドラインに `-ipo` を追加すると、コンパイラはリンカの直前に起動されます。IPO は、IR を含むオブジェクト・ファイルすべてを対象にして実行されます。最初に、コンパイラはすべてのサマリ情報を解析し、IR を含むアプリケーションの部分をコンパイルします。各アプリケーションの部分をコンパイル中に、アプリケーションに関するグローバル情報を取得することで、最適化の質を向上します。

注
スタティック・ライブラリ (`.a` ファイル) についてはマルチファイル IPO は利用できません。詳細については、「実際のオブジェクト・ファイルの生成」を参照してください。

`-ipo` を指定すると、ドライバとコンパイラで自動的にプログラム全体を検出できるようになります。プログラム全体を検出すれば、プロシージャ間の定数伝播、スタックフレームのアライメント、データ・レイアウト、共通ブロックのパディングといった最適化を

もっと効率よく実行しますが、削除される不要な関数の数も増えます。このオプションは安全です。

コマンドラインによる IPO 実行ファイルの作成

IA-32 アーキテクチャを対象とするコンパイルと Itanium® アーキテクチャを対象とするコンパイルでは、IPO を有効にするコマンドライン・オプションは同じです。

中間表現 (IR) を含む疑似オブジェクト・ファイルを作成するには、`-ipo` を使用して次のようにソースファイルをコンパイルします。

```
ifort -ipo -c a.f b.f c.f
```

上記のコマンドは、`a.o`、`b.o`、および `c.o` オブジェクト・ファイルを生成します。 これらのファイルには、コンパイルされたソースファイル `a.f`、`b.f`、および `c.f` に対応するインテル® コンパイラの中間表現 (IR) が含まれます。`.o` ファイルが生成された後で、コンパイルを停止するには、`-c` が必要です。

これで、リンク・コマンドラインに `-ipo` を追加して、プロシージャ間の最適化を行うことができます。次の例では、実行ファイル `app` を作成します。

```
ifort -oapp -ipo a.o b.o c.o
```

このコマンドは IR を含むオブジェクトに対してコンパイラを起動して、リンクされるオブジェクトの新しい一覧を生成します。その後 、GCC `ld` を呼び出して指定されたオブジェクト・ファイルにリンクし、`-o` オプションで指定された名前のアプリケーション (`app`) を生成します。IPO は IR を含むソースファイルにだけ適用され、IR を含まないオブジェクト・ファイルはリンク段階に渡されます。

注
上記の手順で `ifort` の代わりに `xild` ツールを使用できます。

また、上記の 2 つのコマンドを次のように 1 つに組み合わせることもできます。

```
ifort -ipo -oapp a.f b.f c.f
```

複数の IPO オブジェクト・ファイルの生成

多くの場合、IPO はリンク時のコンパイルで 1 つのオブジェクト・ファイルを生成します。これは、非常に大きいアプリケーションでは効率が悪く、最悪の場合、アプリケー

ションで -ipo を使用できないこともあります。この問題を回避するには、次の 2 つの方法のいずれかを行います。1 つめの方法は、サイズベースのヒューリスティックです。このヒューリスティックでは、大きいアプリケーションのリンク時のコンパイルにおいて複数のオブジェクト・ファイルを自動生成します。 2 つ目の方法は、明示的なコマンドライン・コントロールを 1 つまたは 2 つ使用して、マルチオブジェクト IPO を処理するようコンパイラに指示します。

- -ipo*N*: *N* は生成するオブジェクト・ファイルの数です。
- -ipo_separate: 各ソースファイルに対して個別の IPO オブジェクト・ファイルを作成するようコンパイラに指示します。

これらのオプションは、-ipo オプションの代わりとして使用されます。マルチオブジェクト IPO コンパイルを明示的に指示すると、サイズベースのヒューリスティックはオフになります。

リンク時のコンパイルで生成されるファイルの数は、-ipo_c または -ipo_S オプションが使用されていない限り表示されません。 この場合、コンパイラは番号をファイル名に追加します。 例えば、次のコマンドラインを考えてみます。

```
ifort a.o b.o c.o -ipo_separate -ipo_c
```

このコマンドラインで、a.o、b.o、および c.o にはすべて IR が含まれているため、コンパイラは ipo_out.o、ipo_out1.o、ipo_out2.o、および ipo_out3.o を生成します。

最初のオブジェクト・ファイルには、グローバル・シンボルが含まれています。 その他のオブジェクト・ファイルは、ソースファイルに対応しています。

また、命名規則にはユーザ指定の名前が適用されます。 例:

```
ifort a.o b.o c.o -ipo_separate -ipo_c -o appl.o
```

コンパイラは、appl.o、appl1.o、appl2.o、および appl3.o を生成します。

IPO の中間出力の取得

-ipo_c と -ipo_S オプションは、IPO の効果を分析する場合や、プログラム完成前のモジュールで、IPO をテストする場合に便利です。

複数のファイル全体にわたって最適化を行い、オブジェクト・ファイルを生成するには、-ipo_c オプションを使用します。このオプションを使用すると、-ipo の解説で述べた最適化処理を行いますが、最後のリンク段階に進む前にその処理は停止し、最適

化されたオブジェクト・ファイルはそのまま残ります。このファイルのデフォルト名は ipo_out.o です。-o オプションを使用して、別のファイル名を指定できます。次に例を示します:

```
ifort -tpp6 -ipo_c -ofilename a.f b.f c.f
```

複数のファイル全体にわたって最適化を行い、アセンブリ・ファイルを生成するには、-ipo_S オプションを使用します。このオプションを使用すると、-ipo の解説で述べた最適化処理を行いますが、最後のリンク段階に進む前にその処理は停止し、最適化されたアセンブリ・ファイルはそのまま残ります。このファイルのデフォルト名は ipo_out.s です。-o オプションを使用して、別のファイル名を指定できます。次に例を示します:

```
ifort -tpp6 -ipo_S -ofilename a.f b.f c.f
```

マルチオブジェクト IPO を使用した場合、-ipo_c および -ipo_S オプションは、複数の出力を生成します。 最初のファイルの名前は、-o オプションで指定されるファイル名から取得します。 それ以降のファイルの名前には、取得したファイル名に番号が追加されます。例えば、最初のオブジェクト・ファイルの名前が foo.o の場合、次のオブジェクト・ファイルの名前は foo1.o となります。

コンパイラは、生成される各オブジェクトまたはアセンブリ・ファイルの名前を示すメッセージを表示します。これらのファイルは、実際のリンク段階で追加することで、最終的なアプリケーションをビルドすることができます。

xild を使用したマルチファイル IPO 実行ファイルの作成

「コマンドラインでのマルチファイル IPO の作成」の手順 2 の代わりに、インテル® リンカ、xild を使用します。リンカ xild は次のように動作します:

1. IR を含むオブジェクトが見つかった場合、コンパイラを起動して IPO を実行します。
2. GCC ld を起動して、アプリケーションをリンクします。

xild のコマンドラインの構文は、GCC リンカと同じです。

```
xild [<options>] <LINK_commandline>
```

各アイテムの意味は次のとおりです:

- [<options>] (オプション) には、あらゆる GCC リンカ・オプション、または xild でのみサポートされるオプションを含めることができます。
- <*LINK_commandline*> は、ld への有効な引数のセットを含むリンカ・コマンドラインです。

IPO を使用して app を作成する場合、オプション -o*filename* を次のように使用します:

```
xild -oapp a.o b.o c.o
```

xild は、IR を含むオブジェクトの IPO を実行するためにコンパイラを呼び出し、リンクされるオブジェクトの新しい一覧を作成します。そして、xild は ld を呼び出して新しいリストで指定されたオブジェクト・ファイルにリンクし、app を生成します。

注

-ipo オプションを使用した場合、コマンドラインにオブジェクト・ファイルとリンカ引数を複数入力すると、その並び順が変わってしまうことがあります。したがって、コマンドラインに引数を複数入力する場合は、その順番を崩してはならないプログラムに対して -ipo を使用すると、プログラムが誤動作することがあります。

xild コマンドは、3 種類の IPO スイッチを認識します (-ipo、-ipo*N*、および -ipo_separate)。

使用規則

次のような場合、インテル・リンカ xild を使用して、アプリケーションをリンクする必要があります:

- ソースファイルが IPO を有効にしてコンパイルされた場合。IPO が -ipo コマンドライン・オプションを指定して有効にされた場合。
- 通常、アプリケーションをリンクするのに GCC リンカ (ld) を使用する場合。

xild のオプション

xild がサポートしている追加オプションは、IPO の結果を検証するために使用することができます。次の表で、これらのオプションを説明します。

-qipo_fa[*file.s*]	IPO コンパイルのアセンブリ・リストを生成します。リストファイルの名前またはファイルを配置するディレクトリ (バックスラッシュ付き) を指定し

	ます。デフォルトのリスト名は、ipo_out.s です。
-qipo fo[*file.o*]	IPO コンパイルのオブジェクト・ファイルを生成します。オブジェクト・ファイルの名前またはファイルを配置するディレクトリ（バックスラッシュ付き）を指定します。デフォルトのオブジェクト・ファイル名は、ipo_out.o です。
-ipo fcode-asm	アセンブリ・リストにコードバイトを追加します。
-ipo fsource-asm	アセンブリ・リストに高水準言語のソースコードを追加します。
-ipo fverbose-asm, -ipo fnoverbose-asm	バージョンと xild のアセンブリ・リストで使用されるオプションを含むコメントの挿入を有効または無効にします。

xild の呼び出しによって IPO マルチオブジェクトのコンパイルが行われた場合（アプリケーションが大きい、またはユーザによって明示的に複数のオブジェクトを要求されたため）、最初の .s ファイルは -qipo_fa オプションから名前を取得します。それ以降の .s ファイルの名前には、番号が追加されます（例えば、-qipo_fafoo.s では foo.s と foo1.s）。-qipo_fo オプションの場合も同様です。

コード・レイアウトおよびマルチオブジェクト IPO

IPO コンパイル中に実行される最適化には、コード・レイアウトがあります。IPO 解析は、IR が含まれたすべてのルーチンのレイアウト順序を確認します。1 つのオブジェクトが生成される場合、コンパイラはルーチンを任意の順序でコンパイルすることでレイアウトを生成します。

マルチオブジェクト IPO コンパイルでは、コンパイラはリンカにリンクを行う順序を渡す必要があります。最初に、コンパイラは各ルーチンを、名前が付けられたテキストのセクションに格納します（例えば、1 番目のルーチンを .text00001、2 番目のルーチンを .text00002）。次に、コンパイラはリンカ・スクリプトを生成します。このスクリプトは、リンカが .text00001、.text00002 という順序でリンクを行うように指示します。リンク時のコンパイルと最後のリンクで同じ ifort（または xlid）呼び出しが行われる場合、リンクは透過的に行われます。

ただし、-ipo_c または -ipo_S が使用される場合、リンカ・スクリプトを考慮する必要があります。これらのスイッチを使用した場合、IPO コンパイルと実際のリンクでは異なる ifort 呼び出しが行われます。この場合、ifort は明示的なリンカ・スクリプト ipo_layout.script を生成していることを示すメッセージを表示します。

通常、生成されたスクリプト ipo_layout.script を使用するにはリンクコマンドを変更します:

```
--script=ipo_layout.script
```

アプリケーションで既にカスタム・リンカ・スクリプトを使用している場合は、ipo_layout.script の内容をそのスクリプトに追加することができます。レイアウト順序は、ipo_layout.script の .text セクションの最初に記述されています。例えば、12 個のルーチンのレイアウト順序は次のとおりです。

```
.text      :
{
*(.text00001) *(.text00002) *(.text00003) *(.text00004)
*(.text00005)
*(.text00006) *(.text00007) *(.text00008) *(.text00009)
*(.text00010)
*(.text00011) *(.text00012)
...
```

他のリンカ・スクリプトが必要なアプリケーションには、.text セクションの記述をカスタム・リンカ・スクリプトに追加することができます。これらの記述をリンカ・スクリプトに追加する場合は、今後の開発を考慮してエントリを余分に追加することが望ましいでしょう。"*(...)" 構文は、余分に追加したエントリをオプションとして処理するため、問題ありません。

アプリケーションでリンカ・スクリプトを使用しない場合、アプリケーションはビルドされますが、レイアウト順序はランダムになります。リンカ・スクリプトを使用しない場合は、特に大きなアプリケーションのパフォーマンスに悪影響を与える可能性があります。

実際のオブジェクト・ファイルの生成

状況によっては、-ipo を使用して実際のオブジェクト・ファイルを生成する必要があります。IPO を行う場合、擬似オブジェクト・ファイルではなく実際のオブジェクト・ファイルを強制的に生成するために、ipo_obj オプションと -ipo オプションを組み合わせて使用します。

次の場合は、-ipo_obj を使用する必要があります:

- -ipo のコンパイル・フェーズで生成したオブジェクトが、xiar を使用せずにスタティック・ライブラリに置かれる場合。スタティック・ライブラリについてはマルチファイル IPO を利用できないため、スタティック・ライブラリはすべてリンカに渡されます。擬似オブジェクト・ファイルを含むスタティック・ライブラリにリンクするとリンケージ・エラーが発生します。-ipo_obj を指定すると、スタティック・ライブラリの中で使用できるオブジェクト・ファイルが生成されます。

- 一方、xiar を使用して作成したスタティック・ライブラリは、普通のライブラリとして機能します。
- -ipo を指定したコンパイル・フェーズで生成したオブジェクトを、-ipo オプションと xiar を使用しないでリンクする場合。
- -ipo を指定してコンパイルを行っている最中に、-S を用いてソースファイルごとにアセンブリ・リストを生成する場合。-ipo_obj ではなく -S と一緒に -ipo を使用すると、警告メッセージが出て、コンパイルしたソースファイルごとに空のアセンブリ・ファイルが生成されます。

バージョン番号による .il ファイルの管理

IPO コンパイルは、コンパイル・フェーズとリンクフェーズの 2 つに分けて行われます。コンパイル・フェーズでは、コンパイラは中間言語 (IL) バージョンのコードを生成します。リンクフェーズでは、コンパイラは IL を読み込んでコンパイルを完了し、実際のオブジェクト・ファイルまたは実行ファイルを生成します。

一般に、コンパイラのバージョンが異なると、異なる定義に基づいて IL が生成されるため、IL は互換がない場合があります。インテル® Fortran コンパイラは各コンパイラの IL 定義に独自のバージョン番号を割り当てます。コンパイラがバージョン番号が異なるファイルの IL を読み込もうとすると、コンパイルは続行されますが、IL は破棄され、コンパイルでは使用されません。コンパイラはその後、互換性のない IL が検出され破棄されたという警告メッセージを出力します。

インテル® コンパイラが生成した IL は、.il という拡張子の付いたファイル名で ライブラリに保存されます。 このライブラリがライブラリに配置された IL を生成した同じコンパイラで起動されて IPO コンパイルに使用された場合、コンパイラはライブラリから .il を抽出してプログラムの最適化に使用することができます。例えば、ライブラリで定義されている関数を、ユーザのソースコードにインライン展開することができます。

IPO オブジェクトからのライブラリの作成

通常、ライブラリは ar などのライブラリ・マネージャを使用して作成されます。ライブラリ・マネージャは、オブジェクト・リストを読み取り、そのオブジェクトを、次のリンク段階で使用するライブラリに挿入します。

```
xiar cru user.a a.o b.o
```

上記のコマンドは、a.o と b.o オブジェクトを含む user.a というライブラリを作成します。

ただし、-ipo -c を使用してオブジェクトが作成された場合、オブジェクトは有効なオブジェクトではなく、そのオブジェクト・ファイルの中間表現 (IR) だけを含みます。次に例を示します。

```
ifort -ipo -c a.f b.f
```

リンク時のコンパイルに使用される IR だけを含んだ a.o と b.o が作成されます。ライブラリ・マネージャでは、このオブジェクトをライブラリに挿入できません。

この場合は、ライブラリ・ドライバ xild -ar を使用しなければなりません。このドライバは、オブジェクト・ファイルに保存している IR 上にコンパイラを呼び出し、ライブラリに挿入できる有効なオブジェクトを生成します。

```
xild -lib cru user.a a.o b.o
```

「xild を使用したマルチファイル IPO 実行ファイルの作成」を参照してください。

-ip と -Qoption 指定子の併用

メモリとプロシージャ間の最適化を試してみることにより、インテル® Fortran コンパイラの最適化機能をアプリケーションに合わせて調整できます。

特定のインライン展開およびループ最適化を選択するには、適切なキーワードを用いて -Qoption オプションを入力します。このオプションを入力する場合は、次の例に示すように、-ip または -ipo を指定しなければなりません。

```
-ip[-Qoption,tool,opts]
```

tool が Fortran (f) で、opts は -Qoption 指定子です (下記参照)。これらの指定子がコンパイラのインライン手法に与える影響については、「関数のインライン展開の基準」を参照してください。

他のツールへのオプションの受け渡しに関する詳細は、「/Qoption,*tool*,*opts*」を参照してください。

-Qoption 指定子

-Qoption なしで -ip または -ipo を指定すると、コンパイラは次のことを行います:

- 関数のインライン展開
- 定数の引数伝播

- レジスタ内の引数の受け渡し
- モジュール・レベルでの静的変数の監視

次の -Qoption 指定子を使用して、プロシージャ間の最適化を設定できます。これを有効にするには、例に示すように、-Qoption オプションとともに -ip または -ipo を入力しなければなりません:

-ip -Qoption,*f*,*ip_specifier*

ここで、ip_specifier は、次の表にある -Qoption 指定子の 1 つです。

-Qoption 指定子	
-ip args in regs=0	レジスタ内での引数の受け渡しを無効にするオプションです。デフォルトでは、ローカルに呼び出された外部関数はレジスタ内で引数の受け渡しができます。通常、スタティック関数のみはレジスタ内での引数の受け渡しができますが、それには条件が 2 つあり、1 つはその関数のアドレスが取得されないこと、もう 1 つは個数の定まらない引数をその関数が使用しないことです。
-ip_ninl_max_stats=n	インライン展開される関数内の中間言語文の有効な数を設定します。n は正の整数です。通常は、中間言語の文の個数は、ソース言語の文の実際の個数を超えます。n のデフォルト値は 230 です。
-ip ninl min stats=n	インライン展開する関数 1 個について、中間言語の文を最小何個まで許容させるかを設定するオプションです。n は正の整数です。ip_ninl_min_stats のデフォルト値は、次のとおりです。 IA-32 コンパイラ: ip_ninl_min_stats = 7 Itanium® コンパイラ: ip_ninl_min_stats = 15
- ip_ninl_max_total_stats=n	インライン化のために、中間言語文で、関数のサイズに展開できる最大数を設定します。n は正の整数です。n のデフォルト値は 2000 です。

次のコマンドは source.f でプロシージャおよびプロシージャ間の最適化を有効にし、各関数に対して中間言語の文の数の最大増加数を 5 に設定します。

```
ifort -ip -Qoption,f,-ip_ninl_max_stats=5 source.f
```

関数のインライン展開

関数のインライン展開の基準

インライン化の対象とする呼び出しは、一定の最低基準を満たしていなければなりません。呼び出しには、3 つの主要な要素があります。

*呼び出しサイト*とは、インライン化する関数の呼び出しのサイトです。

*呼び出し元*は、呼び出しサイトを含む関数です。

*呼び出し先*は、呼び出されてインライン化する関数です。

呼び出しサイトの最低基準:

- 実引数の数は、呼び出し先の仮引数の数と一致していなければなりません。
- 戻り値の数は、呼び出し先の戻り値の数と一致していなければなりません。
- 実引数と仮引数のデータ型の間に互換性がなければなりません。
- 多言語のインライン化は実行できません。呼び出し元と呼び出し先は同じソース言語で記述されていなければなりません。

呼び出し元の最低基準:

- 呼び出し元にインライン化するすべての呼び出しサイトから、呼び出し元にインライン化される中間文は最大 2000 です。この値は次のオプションを使用して変更できます。

  -Qoption,*f*,-*ip_ninl_max_total_stats*=new value

- 関数がスタティックとして宣言されている場合は、呼び出す必要があります。そうでない場合は削除されます。

呼び出し先の最低基準:

- 可変長引数リストを持ちません。
- 名前から実行頻度が低いと考えられません。名前に次の部分文字列を含むルーチンはインライン展開されません。`abort`、`alloca`、`denied`、`err`、`exit`、`fail`、`fatal`、`fault`、`halt`、`init`、`inerrupt`、`invalid`、`quit`、

rare、stop、timeout、trace、trap、および warn といった部分文字列がこれに当たります。
- その他の理由ではインラインできないと考えられません。

インライン展開のルーチンの選択 (PGO 使用/PGO 非使用の両方)

以上の諸条件が満たされると、コンパイラはどのルーチンをインライン展開すればプログラムのパフォーマンスに最も寄与するかを調べて選び出します。これは、デフォルトのヒューリスティックを使用して行われます。プロファイルに基づく最適化 (-prof_use) を使用するかどうかによって、コンパイラが使用するヒューリスティックは異なります。

-ip または -ipo でプロファイルに基づく最適化を使用する場合、コンパイラは次のヒューリスティックを使用します:

- デフォルト・ヒューリスティックでは、プログラムに対して収集されたプロファイル情報を基に、最も頻繁に実行される呼び出しサイトに重点がおかれます。
- デフォルトでは、コンパイラは 230 以上の中間文では、関数のインライン化は行いません。この値は、オプション -Qoption,*f*,-*ip_ninl_max_stats*=new value を使用して変更できます。
- 通常のデフォルト・ヒューリスティックは、直接再帰 (direct recursion) を検出すると停止します。
- デフォルト・ヒューリスティックを使用すると、インライン化の最低条件を満たしている非常に小さな関数は必ずインライン化します。
  - Itanium® ベース・アプリケーションのデフォルト値: ip_ninl_min_stats = 15
  - IA-32 アプリケーションのデフォルト値: ip_ninl_min_stats = 7
- これらの制限は、-Qoption,*f*,-*ip_ninl_min_stats*=new value オプションで変更できます。

  「-Qoption 指定子」および「プロファイルに基づく最適化 (PGO)」を参照してください。

-ip または -ipo でプロファイルに基づく最適化を使用しない場合、コンパイラは次のヒューリスティックを使用します: インライン展開が最終的なプログラムのサイズを増加しない場合、関数をインライン展開します。

インライン展開とプリエンプション

関数のプリエンプションとは、ランタイム時にその関数を実装するコードを、他のコードで置換することです。関数をプリエンプトすると、関数の元のバージョンではなく置換後の新しいバージョンが実行されます。プリエンプションを使用して、関数のコードにエラーのあるバージョンや効率の悪いバージョンを、正しいバージョンまたは改善されたバージョンに置き換えることができます。

コンパイラは、-ip がオンの場合には外部からアクセスできる関数はすべてプリエンプトされる可能性があり、したがってインライン展開の対象にならないものとみなします。これは、現時点では、-ip がオンの場合には内部プロシージャを除くすべての Fortran サブプログラムがインライン化できないことを意味します。

ただし、-ipo と -ipo_obj をファイルごとに使用した場合には、関数のインライン化を行うことができます。「実際のオブジェクト・ファイルの生成」を参照してください。

ユーザ関数のインライン展開の制御

次の表に示すオプションを使用して、関数のインライン展開を制御することができます。

オプション	効果
-ip_no_inlining	このオプションは、-ip または -ipo オプションが指定されている場合に使用できます。この場合、-ip_no_inlining オプションは、-ip によるプロシージャ間の最適化の結果から生じるインライン展開を無効にします。しかし、他のプロシージャ間の最適化には何の影響も与えません。
-inline_debug_info	呼び出しサイトのソース位置をインライン展開されたコードに割り当てる代わりに、インライン展開されたコードのソース位置を保持します。
-ip no pinlining	部分的なインライン展開を無効にします。このオプションは、-ip または -ipo オプションも指定している場合に使用できます。

ライブラリ関数のインライン展開

デフォルトでは、標準ライブラリ関数と数値演算ライブラリ関数のいくつかは、関数の呼び出し時点で、コンパイラによって自動的に展開します。通常、この処理によって計算速度が速くなります。

しかし、インライン化されたライブラリ関数では、インライン展開時に errno 変数は設定されません。したがって、errno 変数を設定するかしないかによって動作が異なるコードに対しては、-nolib_inline オプションを使用しなければなりません。また、コンパイラから提供されるライブラリ関数と同じ名前を持つ関数はライブラリ関数と見なすため、元々の呼び出し命令は、インライン化したものと置き換わります。

そのため、プログラムに標準のいずれかのライブラリ・ルーチンと同じ名前の関数が定義されている場合は、-nolib_inline オプションを使用して、ユーザが定義した関数が必ず使用されるように指定しなければなりません。
-nolib_inline オプションは、すべての組込み関数のインライン化を無効にします。

> **注**
> ライブラリ関数の自動インライン展開は、プロシージャ間の最適化処理中にコンパイラが行うインライン展開とは関連がありません。例えば、次のコマンドは、数値演算ライブラリ関数を展開せずに、プログラム sum.f をコンパイルします。

```
ifort -ip -nolib_inline sum.f
```

プロファイルに基づく最適化

プロファイルに基づく最適化の概要

プロファイルに基づく最適化 (PGO) を行うと、アプリケーションのどの領域が最も頻繁に実行するかがコンパイラに伝えられます。コンパイラはこの領域を知ると、より慎重かつ明確にアプリケーションの最適化を行います。例えば、PGO を使用するとコンパイラは関数のインライン化について的確な判断が下せる場合が多くなり、その結果、プロシージャ間の最適化の効率が向上します。

インストルメント済みプログラム

プロファイルに基づく最適化では、ソースコードおよびコンパイラからの特別なコードからインストルメント済みプログラムを作成します。インストルメント済みコードを実行するたびにインストルメント済みプログラムは動的情報ファイルを生成します。もう一度コンパイルすると、生成した動的情報がマージされて 1 個のサマリファイルができます。このプロファイル情報を使用して、コンパイラはプログラム内で最も頻繁に使用するパスの実行の最適化を図ります。

サイズや速度に注目する他の最適化と異なり、IPO および PGO の結果はばらつきます。その理由は、プログラムが違えばプロファイルも異なり、最適化の見込みがあるかどうかもさまざまだからです。このガイドラインは、IPO および PGO を使用することにより利点が得られるどうかを判断するのに役立ちます。最適化の原理を理解し、ソースコードの独自な特性を知っておく必要があります。

PGO で向上するパフォーマンス

このバージョンの インテル® Fortran コンパイラでは、PGO は、次のように性能向上を行っています。

- レジスタの割り当てには、プロファイル情報を使用して、スピルコードの場所を最適化します。
- 間接関数呼び出しでは、最も可能性の高い対象を識別することにより、分岐予測が向上されます。Pentium® 4 プロセッサおよびインテル® Xeon™ プロセッサの長いパイプラインでは、分岐の予測の向上により、高いパフォーマンス・ゲインを得られます。
- コンパイラは、実行の繰返し回数が少ないループを検出しベクトル化せず、ベクトル化により追加される可能性のあるランタイム・オーバーヘッドを減少させます。

プロファイルに基づく最適化の方法とその使用モデル

PGO は、コンパイルの時点での予測が困難な、繰返し実行される分岐を含むコードの最適化として最も有効に働きます。この例として、ほとんどの場合にエラー条件が偽になるようなエラーチェック処理を多数含むコードが挙げられます。このような、いわゆる「コールド」(cold) なエラー処理コードを、分岐予測をほとんど誤ることのないように、配置できます。"ホット"コードに対する"コールド"コードの挿入を最小限に抑えると、命令キャッシュの動作が改善されます。

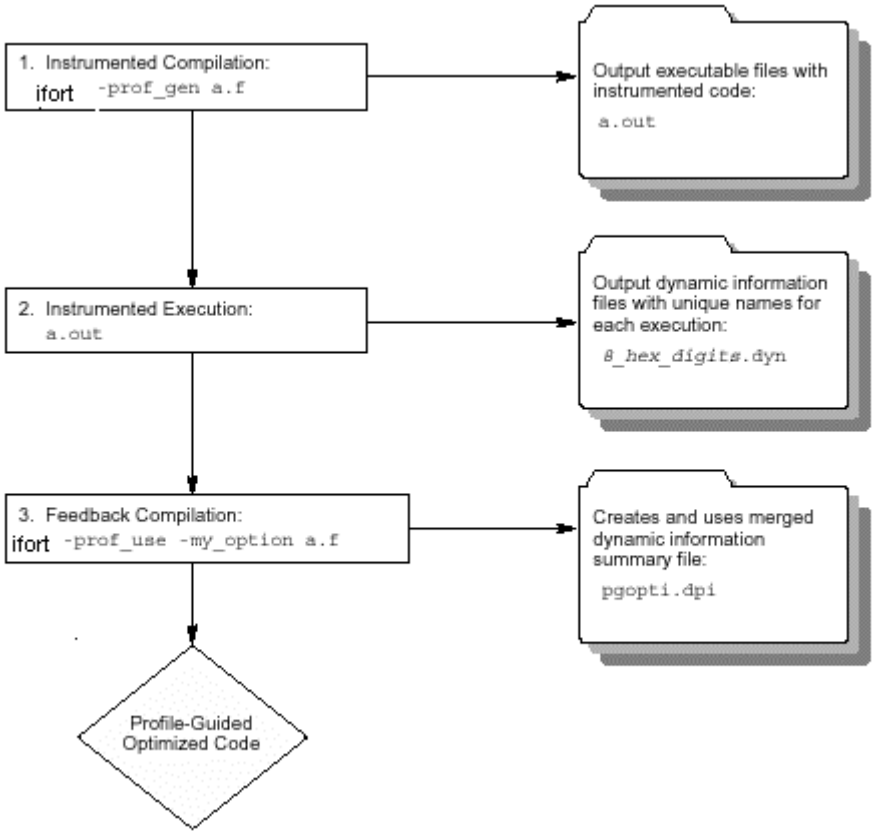
PGO フェーズ

PGO 手法には、次の 3 つのフェーズおよびオプションが必要です。

1. `-prof_gen` による**インストルメンテーション・コンパイル**とインストルメンテーション・リンク
2. 実行ファイルの実行による**インストルメント済みのプログラムの実行**。結果として、動的情報ファイル (`.dyn`) が作成されます。
3. `-prof_use` オプションによる**フィードバック・コンパイル**

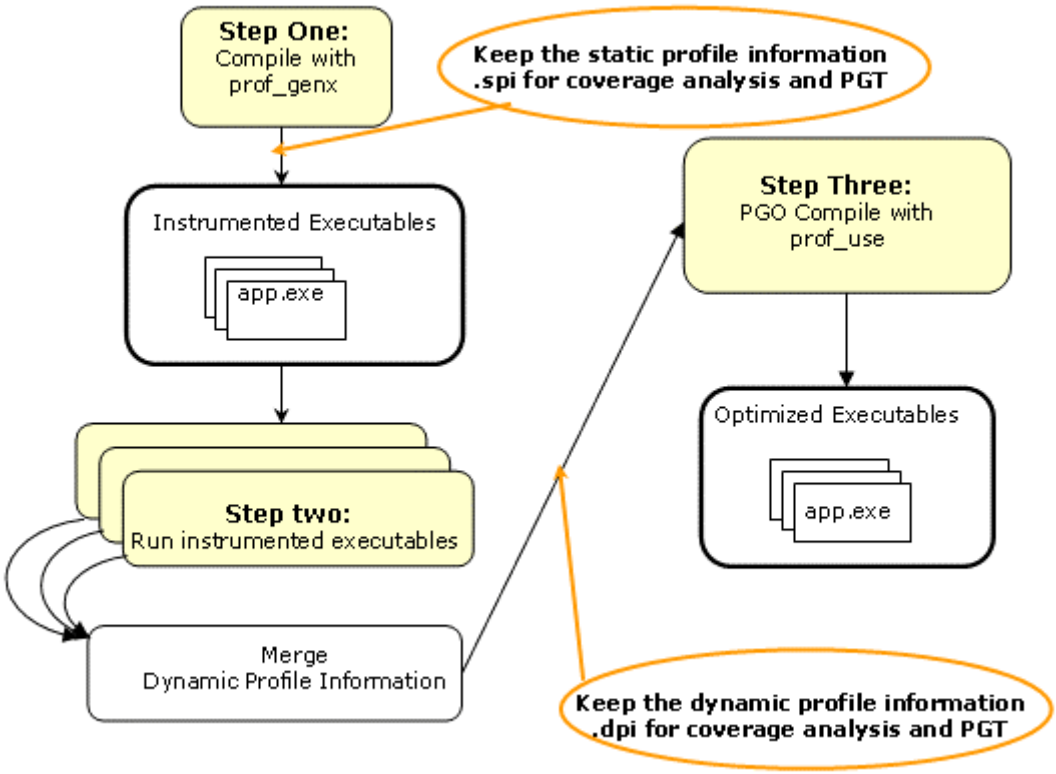
下記のフローチャートは、IA-32 アーキテクチャのコンパイルと Itanium® アーキテクチャのコンパイルの場合のこのプロセスを示したものです。PGO の適否を決める手掛かりの 1 つは、コードのどの部分に処理が集中しているかを知ることです。例えば、プログラムに提供するデータセットが大変類似していて、実行のたびに同じような処理が繰り返されるような場合には、PGO によってプログラムの実行を最適化できます。しかし、さまざまなデータセットを使用している場合は、呼び出されるアルゴリズムも多様化する可能性があります。このような場合、プログラムは実行のたびに違った動作になるときがあります。

基本的なプロファイルに基づく最適化のフェーズ



PGO の使用モデル

下記の図は PGO の使用モデルを示しています。



IA-32 システム用の簡単な例 (`myApp.f90`) について順に説明します。

1. 設定

   `PROF_DIR=c:/myApp/prof_dir`

2. コマンドの発行

   `ifort -prof_genx myApp.f90`

   このコマンドは、プログラムをコンパイルして、インストルメント済みバイナリ `myApp.exe` と静的プロファイル情報ファイル `pgopti.spi` を生成します。

3. `myApp` の実行

   `myApp` が呼び出される度に、インストルメント済みアプリケーションが実行され、`PROF_DIR` で指定されたディレクトリ内に拡張子 `.dyn` の動的プロファイル情報ファイルが新規に生成されます。

4. コマンドの発行

   ifort -prof_use myApp.f90

   このステップでは、コンパイラがすべての .dyn ファイルを、アプリケーションの全プロファイル情報を含む 1 つの .dpi ファイルにマージし、最適化されたバイナリを生成します。.dpi ファイルのデフォルト名は、pgopti.dpi です。

基本的な PGO オプション

基本的な PGO 最適化に使用されるオプションは、次のとおりです。

- -prof_gen、インストルメント済みコードの作成
- -prof_use、プロファイルによって最適化された実行ファイルの作成
- -prof_format_32 .dyn および .dpi ファイル用の 32 ビットのカウンタの作成

コードの動作が実行ごとに大きく異なる場合は、プロファイル情報によって得られるメリットが、最新のプロファイルの維持に必要な労力に見合うものであるかどうか検討する必要があります。基本となるプロファイルに基づく最適化には、次のオプションが PGO のフェーズで使用されます。

インストルメント済みコードの作成、-prof_gen

-prof_gen オプションは、各基本ブロックの実行カウントを取得するために、プロファイル用プログラムをインストルメントします。これは、PGO の第 1 フェーズで使用され、インストルメント済み実行の準備として、インストルメント済みコードをオブジェクト・ファイル内に生成するようにコンパイラに指示します。-prof_gen のコンパイル処理には、自動的に並列 make がサポートされます。

プロファイルによって最適化された実行ファイルの生成、-prof_use

-prof_use オプションは、PGO のフェーズ 3 で使用され、プロファイルによって最適化された実行ファイルを生成し、使用できる動的情報 (.dyn) ファイルを pgopti.dpi ファイルにマージするようコンパイラに指示します。

注

動的情報ファイルは、フェーズ 2 でインストルメント済みの実行ファイルを実行した時点で生成します。

インストルメント済みプログラムを何回か実行する場合は、-prof_use を指定すると、実行するたびに同じ動的情報ファイル同士がマージされ、その前の pgopti.dpi ファイルは上書きされます。

32 ビットのカウンタの使用、-prof_format_32

デフォルトでは、インテル® Fortran コンパイラは、.dyn および .dpi ファイルにおける多数のイベントを処理するために 64 ビットのカウンタを作成します。旧バージョンのコンパイラとの互換性を維持するには、-prof_format_32 オプションで、32 ビットのカウンタを生成します。.dyn および .dpi ファイルの形式が現在のコンパイルで使用されている形式と非互換の場合、次のメッセージが生成されます。

```
Error:  xxx.dyn has old or incompatible file format -
delete file and redo instrumentation compilation/execution.
```

.dyn および .dpi ファイル用の 64 ビットのカウンタやポインタは、各種プラットフォームの異なるポインタサイズのために生じる不一致を回避します。

関数分割の無効、-fnsplit-

-fnsplit- Itanium® ベース・システムで関数分割を無効にします。フェーズ 3 で -prof_use により関数分割は有効になります。これは、ルーチンを異なるセクションに分割することによって、コードの局所性を向上させるためです。異なるセクションとは、コールドまたは、あまり実行されないコードを含むセクションと、残りのコード (ホットコード) を含むセクションです。

次のような理由により、-fnsplit- を使用して、関数分割を無効にできます。

- 最も重要なことは、デバッグの機能を向上させることです。デバッグのシンボルテーブルでは、分割ルーチン、すなわちホット・コード・セクションとコールド・コード・セクションを持つルーチンを表示するのは困難です。-fnsplit- オプションは、ルーチン内の分割を無効にしますが、関数のグループ化を有効にします。これは、コールド・コード・セクションまたはホット・コード・セクションのいずれかにルーチン全体を配置する最適化機能です。 関数のグループ化は、デバッグ機能を低下させません。

- 別の理由としては、プロファイル・データが実際のプログラム動作をしなかった場合、つまり、ルーチンが実際は稀ではなく頻繁に使用される場合です。

  注

  Itanium ベース・アプリケーションでは、`-prof_use` オプションと -O3 レベルの最適化を組み合わせて使用する場合は、`-O3` オプションがオンになっていなければなりません。`-prof_use` オプションと `-O2` またはそれ以下のレベルの最適化を組み合わせて使用する場合は、デフォルトのオプションを使用したプロファイル・データを生成できます。

「PGO の使用例」を参照してください。

高度な PGO オプション

高度な PGO 最適化を制御するオプションは、次のとおりです。

- `-prof_dir`*dirname*
- `-prof_file`*filename*.

`-prof_dir`*dirname* オプションを使用して、作成される動的情報ファイル (`.dyn`) を配置するディレクトリを指定します。デフォルトは、プログラムのコンパイルが行われるディレクトリです。指定するディレクトリは既に存在していなければなりません。

インストルメンテーション・コンパイルとフィードバック・コンパイルには、どちらも同じ `-prof_dir`*dirname* オプションを指定しなければなりません。また、`.dyn` ファイルを移動する場合は、新しいパスを指定する必要があります。

`-prof_file`*filename* オプションは、プロファイルのサマリファイルにファイル名を指定します。

高度な PGO の使用ガイドライン

高度な拡張 PGO を使用する際は、次のガイドラインに従ってください。

- インストルメント済みコードを実行してからフィードバック・コンパイルを行うまでの間は、プログラムに加える変更を最小限に抑えます。フィードバック・コンパイルでは、情報が生成した後に変更された関数の動的情報は無視します。

注
コンパイラは、動的情報の生成後に関数が変更された場合、動的情報が関数に対応していないことを示す警告を発行します。

- インストルメント済みコードを実行してからフィードバック・コンパイルを行うまでの間にソースファイルに多数の変更を加える場合は、インストルメンテーション・コンパイルを繰返します。
- プロファイルのサマリファイル名は `-prof_file`*`filename`* オプションを使用して指定します。

詳細は、「PGO の環境変数」を参照してください。

PGO の環境変数

環境変数は、動的情報ファイルを保存するディレクトリを指定したり、`pgopti.dpi` ファイルを上書きするかどうか指定するために使用します。下の表に、PGO の環境変数の説明を示します。

変数	説明
`PROF_DIR`	動的情報ファイルの作成先ディレクトリを指定する変数です。この変数は、プロファイル処理の 3 つのフェーズすべてに適用されます。
`PROF_DUMP_INTERVAL`	インストルメント済みのユーザ・アプリケーション内で、インターバル・プロファイルのダンプを開始します。
`PROF_NO_CLOBBER`	フィードバック・コンパイル・フェーズでの処理を少し変更する変数です。デフォルトでは、フィードバック・コンパイル・フェーズで、コンパイラがすべての動的情報ファイルのデータをマージし、`pgopti.dpi` ファイルが既に存在する場合にも新しい pgopti.dpi ファイルを作成します。この変数を設定すると、コンパイラは既存の `pgopti.dpi` ファイルを上書きせずに、警告を発行します。新しい動的情報ファイルを使用するには、既存の `pgopti.dpi` ファイルを削除する必要があります。

環境変数の設定方法については、ご使用のオペレーティング・システムのマニュアルも参照してください。

プロファイルに基づく最適化の例

次に、基本 PGO フェーズでの具体例を示します。

1. **インストルメンテーション・コンパイルとリンク**
   -prof_gen を使用して、インストルメント済み情報付きの実行ファイルを生成します。特に複数のディレクトリにソースファイルを含むアプリケーションの場合には、ほとんどの場合に -prof_dir オプションの使用が推奨されます。-prof_dir は、プロファイル情報が 一定の場所に作成されるようにします。次に例を示します。

   ```
   ifort -prof_gen -prof_dir/usr/profdata -c a1.f a2.f
   a3.f
   ifort -oa1 a1.o a2.o a3.o
   ```

   2 番目のコマンドの代わりに、リンカ (ld) を使用して、インストルメント済みのプログラムを直接生成もできます。このような場合、必ず libirc.a ライブラリとリンクしなければなりません。

2. **インストルメント済みのプログラムの実行**
   対応するデータセットを使用してインストルメント済みのプログラムを実行し、動的情報ファイルを作成します。

   ```
   prompt> a1
   ```

   作成される動的情報ファイルには、a1 を実行するたびに別個の名前と .dyn 拡張子が付けられます。また、このインストルメント済みファイルは、特定のデータセットでこのプログラムを実行したときの振舞いを予測するのに役立ちます。このプログラムは、入力データを変えて何度でも実行できます。

3. **フィードバック・コンパイル**
   -prof_use を指定してソースファイルのコンパイルとリンクを行い、動的情報を使用して、そのプロファイルに従ってプログラムを最適化します。

   ```
   ifort -prof_use -prof_dir/usr/profdata -ipo a1.f a2.f
   a3.f
   ```

   最適化のほかに、コンパイラは pgopti.dpi ファイルを生成します。一般に、フェーズ 1 ではデフォルトの最適化オプション (-O2) を指定し、フェーズ 3 ではさらに高度の最適化オプション (-ip または -ipo) を指定します。この例では、フェーズ 1 で -O2 オプションを指定し、フェーズ 3 で -ipo オプションを指定しています。


注

-ip または -ipo オプションは、-prof_gen オプションとともに使用すると、コンパイラによって無視されます。

詳細は、「基本的な PGO オプション」を参照してください。

.dyn ファイルのマージ

.dyn ファイルをマージするには、profmerge ユーティリティを使用します。-prof_use を指定すると、フィードバック・コンパイル・フェーズで自動的に profmerge が実行されます。

profmerge をコマンドラインで使用するには、次のように指定します。

profmerge [-nologo] [-prof_dir*dirname*]

ここで **-prof_dir*dirname*** は、profmerge ユーティリティ・オプションです。

これにより、カレント・ディレクトリ、または -prof_dir で指定されているディレクトリ内のすべての .dyn ファイルがマージされ、サマリファイル pgopti.dpi が作成されます。

-prof_file*filename* オプションで、.dpi ファイルの名前を指定することができます。

profmerge を -prof_file*filename* とともにコマンドラインで使用するには、次のように指定します。

profmerge [-nologo] [-prof_file*filename*]

ここで **/prof_file*filename*** は、profmerge ユーティリティ・オプションです。


注

profmerge ツールは指定されたディレクトリに存在するすべての .dyn ファイルをマージします。テストの過去の実行時などからの関連のない .dyn ファイルがそのディレクトリ内に存在していないことを確認する必要があります。 この確認を怠ると、プロファイル情報は不適切なプロファイル・データに基づくことになります。その結果、不適切なカバレッジ情報が生成され、最適化されたコードのパフォーマンスに悪影響を与えることがあります。


注

profmerge ツールを使用すれば、アプリケーションをコンパイルせずに、.dyn ファイルを .dpi ファイルにマージできます。

プロファイル・データのダンプ

ここでは、Fortran から C PGO API ルーチンを呼び出す方法の一例を紹介します。PGO API サポート ルーチンの詳細については、「PGO API: プロファイル情報生成サポート」を参照してください。

プロファイルに基づく最適化のインストルメント済みプログラムの実行フェーズの一環として、インストルメント済みのプログラムはプロファイル・データを動的情報ファイル (.dyn ファイル) に書き込みます。このファイルは、インストルメント済みのプログラムが main() から正常に戻るか、または標準的な exit 関数を呼び出した後に書き込まれます。プログラムが正常に終了しない場合に備えて、_PGOPTI_Prof_Dump 関数を用意しています。インストルメンテーション・コンパイル (-prof_gen) 時に、プログラムにこの関数の呼び出しを追加できます。次に例を示します。

```
INTERFACE
SUBROUTINE PGOPTI PROF DUMP()
!DEC$ ATTRIBUTES C,
ALIAS:'PGOPTI Prof Dump'::PGOPTI PROF DUMP
END SUBROUTINE
END INTERFACE
CALL PGOPTI_PROF_DUMP()
```


注

-prof_use を使用してフィードバック・コンパイルを実行する前に、この呼び出しを削除するか、コメントアウトしなければなりません。

profmerge によるソースファイルの再配置

コンパイラは、各ルーチンのソースファイルのフルパスを使用して、そのルーチンに関連するプロファイル・サマリ情報を検索します。デフォルトでは次のことができません。

- アプリケーションのソースを移動する場合、プロファイル・サマリ・ファイル (.dpi) を使用する。
- 異なるディレクトリにある同一アプリケーションのソースをビルドする別のユーザとプロファイル・サマリ・ファイルを共有する。

プロファイル・サマリ・ファイルの共有とアプリケーション・ソースの移動を可能にするには、-src_old オプションと -src_new オプションとともに profmerge を使用します。次に例を示します。

```
prompt>profmerge -prof_dir c:/work -src_old c:/work/sources
-src_new d:/project/src
```

上記のコマンドで、c:/work/pgopti.dpi ファイルが読み込まれます。ソースパスが c:/work/sources プリフィックスで始まる pgopti.dpi ファイルにある各ルーチンは、profmerge により、d:/project/src のプリフィックスに置換されます。c:/work/pgopti.dpi ファイルは、新規のソースパス情報に更新されます。

次の規則を使用します。

- profmerge は、pgopti.dpi ファイルに対して何度でも実行できます。これは、複数のディレクトリにソースファイルがある場合に有効です。 次に例を示します。

```
profmerge -src_old "c:/program files" -src_new
"e:/program files"

profmerge -src_old c:/proj/application -src_new d:/app
```

- -src_old と -src_new に指定される値では、大文字と小文字の区別はありません。 同様に、フォワード・スラッシュ (/) とバック・スラッシュ (\) は、同じ文字とみなされます。
- profmerge によるソースの再配置機能は、pgopti.dpi ファイルを変更するため、ソースの再配置を行う前に、必要に応じて、ファイルのバックアップをとってください。

コード・カバレッジ・ツール

インテル® コンパイラのコード・カバレッジ・ツールは、IA-32 アーキテクチャと Itanium® アーキテクチャの両方で使用できるツールで、さまざまな方法により、開発効率を改善し、問題を少なくして、アプリケーションのパフォーマンスを向上させます。 コード・カバレッジ・ツールの主な特徴は次のとおりです:

- 色分けしたアプリケーションのコード・カバレッジ情報のビジュアル・プレゼンテーション
- アプリケーションの各基本ブロックの動的実行カウントの表示
- アプリケーションを 2 回実行した場合の差分カバレッジまたはプロファイルの比較

このツールの構文は次のとおりです。

codecov [-codecov_option]

-codecov_option はコード・カバレッジの実行に使用するツール・オプションです。オプションを使用しない場合、ツールはプログラム全体についてトップレベルのコード・カバレッジを出力します。

次の表は、ツールが使用するオプションの一覧です。

オプション	説明	デフォルト
-help	コード・カバレッジ・ツールのすべてのオプションを出力します。	
-spi *file*	静的プロファイル情報ファイル (.spi) のパス名を設定します。	pgopti.spi
-dpi *file*	動的プロファイル情報ファイル (.dpi) のパス名を設定します。	pgopti.dpi
-prj	プロジェクト名を設定します。	
-counts	動的実行カウントを生成します。	
-nopartial	一部カバーされたコードを完全にカバーされたコードとして扱います。	
-comp	処理するファイルのリストを含むファイル名を設定します。	
-ref	ref_dpi_file に関する差分カバレッジを検索します。	
-demang	関数名とその引数の両方を復号します。	
-mname	Web ページの所有者の名前を設定します。	
-maddr	Web ページの所有者のメールアドレスを設定します。	
-bcolor	カバーされなかったブロックの html カラー名またはコードを設定します。	#ffff99
-fcolor	カバーされなかった関数の html カラー名またはコードを設定します。	#ffcccc
-pcolor	一部カバーされたコードの html カラー名またはコードを設定します。	#fafad2
-ccolor	カバーされたコードの html カラー名またはコードを設定します。	#ffffff
-ucolor	不明なコードの html カラー名またはコードを設定します。	#ffffff

アプリケーションのコード・カバレッジのビジュアル・プレゼンテーション

アプリケーションをテストするときにインストルメント済みバイナリを実行して集められたプロファイル情報を基に、インテル・コンパイラはコード・カバレッジ・ツールを使用して HTML ファイルを作成します。これらの HTML ファイルはテストによって実行された(実行されなかった) ソースコードの部分を示します。パフォーマンス・ワークロードのプロファイルが適用されると、コード・カバレッジ情報は実行されたワークロードがアプリケーションの重要なコードをどの程度カバーするかを示します。プロファイルに基づく最適化の利点を完全に受けるためには、パフォーマンス・クリティカルなモジュールの高いカバレッジが必要です。

コード・カバレッジ・ツールは、2 つのレベルのカバレッジを作成することができます:

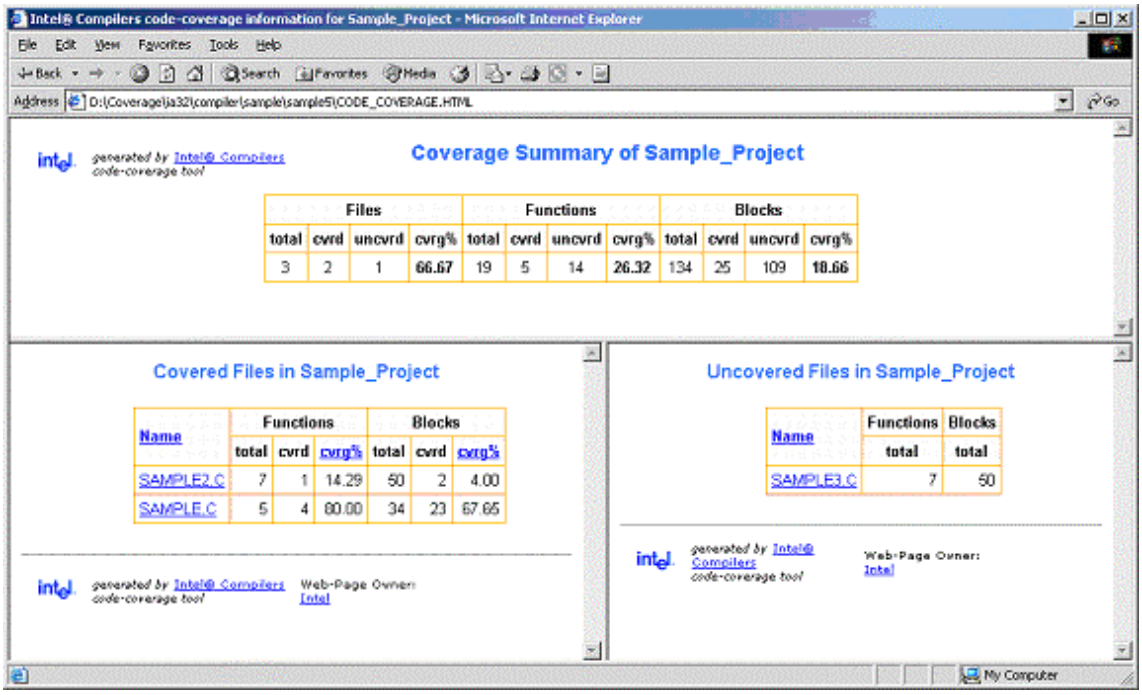
- トップレベル -- 選択されたモジュールのグループ
- 個々のモジュール・ソース・ビュー

トップレベルのカバレッジ

トップレベルのカバレッジは選択されたモジュールのコード・カバレッジ全体をレポートします。次のオプションがあります:

- 処理するモジュールを選択することができます。
- 選択されたモジュールについて、ツールはモジュールとそのカバレッジ情報のリストを生成します。情報には、モジュールにある関数とブロックの総数、カバーされた部分が含まれます。
- レポートされた表でカラムのタイトルをクリックすると、リストが次の項目に基づいて昇順または降順でソートされます:
  - 基本ブロック・カバレッジ
  - 関数カバレッジ
  - 関数名

次の例は、プロジェクトのトップレベル・カバレッジの要約を示しています。モジュール名 (例えば、SAMPLE.C) をクリックすると、ブラウザはそのモジュールのカバレッジ・ソース・ビューを表示します。



フレームのブラウジング

コード・カバレッジ・ツールは、カバーされなかったコードが容易に識別できるようにフレームを作成します。上のフレームにはカバーされなかった関数のリストが、下のフレームにはカバーされた関数のリストがそれぞれ表示されます。カバーされなかった関数については、各関数の基本ブロックの総数も表示されます。カバーされた関数については、ブロックの総数とカバーされたブロックの数、その比率 (つまり、カバレッジ率) が表示されます。

例えば、66.67(4/6) は、対応する関数の 6 つのブロックのうち 4 つのブロックがカバーされたことを示します。関数のブロック・カバレッジ率は 66.67% になります。これらのリストは、カバレッジ率、ブロック数、関数名でソートすることができます。 関数名はソースビューで関数が開始する場所とリンクされています。つまり、クリック 1 つで、ユーザはリストで最も少なくカバーされた関数を見ることができます。もう一回クリックすると、ブラウザは関数のソースを表示します。その後、ユーザはソースビューでスクロールして関数のソースを参照することができます。

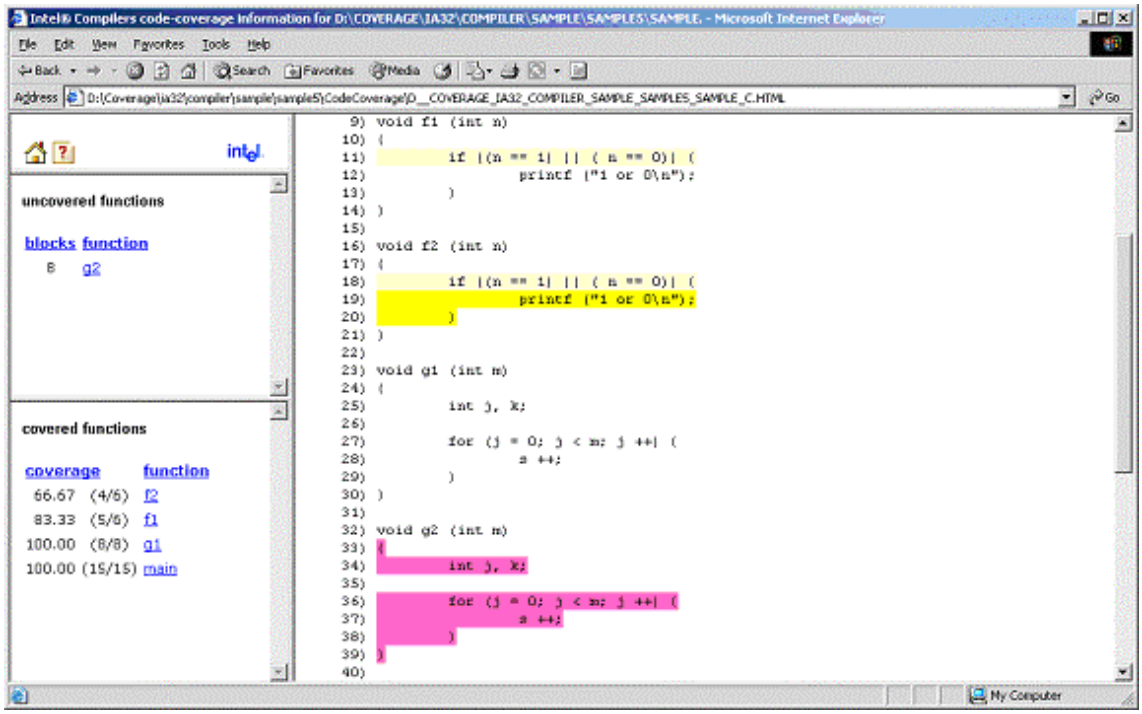
個々のモジュール・ソース・ビュー

個々のモジュール・ソース・ビューで、ツールはカバーされなかった関数のリストとカバーされた関数のリストを表示します。リストは、ソースコードを容易に参照できるように、2 つの別々のフレームに表示されます。リストは、次の項目でソートすることができます:

- カバーされなかった関数内のブロックの数

- カバーされた関数のブロック・カバレッジ
- 関数名

次の図は、SAMPLE.C のカバレッジ・ソース・ビューの例です。



コード・カバレッジ用の配色の設定

ツールは、次のようにカバレッジ・カテゴリを色分けします:

- カバーされたコード
- カバーされなかった基本ブロック
- カバーされなかった関数
- 一部カバーされたコード
- 不明

カバレッジ情報を示すためにツールが使用するデフォルト色を、次の表に示します。

色の種類	意味
カバーされたコード	テストが実行されたコードの部分を示します。デフォルト色は -ccolor オプションで無効にすることができます。
カバーされなかった基本ブロック	テスト中に実行された関数内にある、 テストが実行されなかった基本ブロックを示します。 デフォルト色は -bcolor オプションで無効にすることができます。

カバーされなかった関数	テスト中に呼び出されなかった関数を示します。デフォルト色は -fcolor オプションで無効にすることができます。
一部カバーされたコード	この場所でコード用に複数の基本ブロックが生成されたことを示します。カバーされたブロックとカバーされなかったブロックがあります。デフォルト色は -pcolor オプションで無効にすることができます。
不明	このソース行に対してコードが生成されなかったことを示します。ほとんどの場合、この場所のソースはコメント、ヘッダファイルのインクルード、または変数宣言です。デフォルト色は -ucolor オプションで無効にすることができます。

デフォルト色は、上記の表の各カバレッジ・カテゴリで説明されているオプションを使用して、任意の HTML カラーにカスタマイズすることができます。

コード・カバレッジ・プレゼンテーションでは、コード・カバレッジ・ツールは次のヒューリスティックを使用します。ソース文字は、プロファイル情報によって基本ブロックの先頭として示されたソースの場所に達するまでスキャンされます。その基本ブロックのプロファイル情報でカバレッジのカテゴリが変わったことが示された場合、ツールはコードのその部分のカバレッジ条件に対応するように色を変更し、HTML ファイルに適切なタグを挿入します。

注

コードのコンテキスト中の色を解釈する必要があります。例えば、実行されなかった基本ブロックに続くコメント行はカバーされなかったブロックと同じ色になります。C/C++ アプリケーションの閉じ括弧も同様です。

モジュール・サブセットのカバレッジ解析

インテル・コンパイラのコード・カバレッジ・ツールの機能の 1 つに、アプリケーションのモジュール・サブセットの効率的なカバレッジ解析があります。この解析を行うには、-comp オプションを選択してツールを実行します。

アプリケーション全体、またはサブセットのプロファイル情報を生成し、カバーされたモジュールを異なるコンポーネントに分割してから、コード・カバレッジ・ツールを使用して各コンポーネントのカバレッジ情報を取得します。アプリケーション・モジュールのサブセットのみが -prof_genx オプション付きでコンパイルされた場合、カバレッジ情報はこのコンパイル・オプションに関係するモジュール用にだけ生成されるため、他のモジュール用のプロファイルを生成するオーバーヘッドを回避することができます。

処理するモジュールを指定するには、-comp オプションを使用します。このオプションは引数としてファイル名を指定します。指定するファイルは、解析するモジュール名またはディレクトリ名を含むテキストファイルでなければなりません: 次に例を示します。

```
codecov -prj Project_Name -comp component1
```


注

コンポーネント・ファイルの各行には 1 つのモジュール名のみを記述するようにしてください。

コンポーネント・ファイルに記述されているアプリケーションのすべてのモジュールについてカバレッジ解析が行われます。例えば、上記の例で、ファイル `component1` のある行に `mod1.f90` と記述されていた場合、同じ名前のモジュールがあるアプリケーションのすべてのモジュールが選択されます。ユーザは、特定のパス情報を記述することで特定のモジュールを指定することができます。例えば、ある行に `/cmp1/mod1.f90` と記述されていた場合、ディレクトリ `cmp1` にある mod1.c という名前のモジュールのみが選択されます。コンポーネント・ファイルが指定されていない場合、`-prof_genx` オプション付きでコンパイルされたすべてのファイルがカバレッジ解析用に選択されます。

動的カウンタ

この機能は、アプリケーションの各基本ブロックの動的実行カウントを表示するもので、カバレッジおよびパフォーマンスの調整に役立ちます。

コード・カバレッジ・ツールは、動的実行カウントに関する情報を生成するように設定することができます。この設定を行うには、`-counts` オプションを使用します。カウント情報は、対応する基本ブロックが開始するソースの場所で、^ 記号の後のコードで正確に表示されます。そのソースの場所でコードに複数の基本ブロックが生成される場合 (例えば、マクロ)、そのようなブロックの総数と実行されたブロックの総数が実行カウントの前に表示されます。

例えば、コードの行 11 は `IF` 文です。

```
11    IF ((N .EQ.1).OR.(N .EQ.0))
^ 10 (1/2)
12       PRINT N
^ 7
```

行 11 と 12 のカバレッジ行には次の情報が含まれています。

- 行 11 の `IF` 文は 10 回実行された。
- 行 11 の `IF` 文に対して 2 つの基本ブロックが生成された。
- これら 2 ブロックのうち 1 つだけが実行されているため、部分的なカバレッジの色で表示される。
- 10 回のうち 7 回のみ、変数 n が 0 または 1 の値となった。

特定の状況では、1 つのソース用に生成されたすべてのブロックを 1 つのエンティティとして考慮する必要があります。この場合、少なくとも 1 つのブロックがカバーされたのであれば、1 つのソースの場所用に生成されたすべてのブロックがカバーされたと仮定する必要があります。仮定するには、-nopartial オプションを使用します。このオプションが指定されれば、カバレッジの決定は無効になり、関連する統計がそれに従って調節されます。コードの 11 行目および 12 行目は、12 行目の PRINT 文がカバーされたことを示しています。しかし、11 行目の条件の 1 つのみが常に真であったとします。-nopartial オプションを使用すると、ツールは (11 行目のコードのように) 一部がカバーされたコードを完全にカバーされたコードとして扱います。

差分カバレッジ

コード・カバレッジ・ツールを使用すると、アプリケーションの 2 つの実行 (リファレンス実行と新規実行) を比較して、新規実行ではカバーされるがリファレンス実行ではカバーされないコードを識別することができます。この機能は、アプリケーションがカスタマによって実行される場合に、アプリケーションのテストでカバーされないアプリケーションのコードの部分を検索するために使用することができます。また、アプリケーションのテストスペースに新しく追加されたテストのインクリメンタル・カバレッジの影響を検索するためにも使用されます。

差分カバレッジ用のリファレンス実行の動的プロファイル情報は、次のコマンドのように、-ref オプションで指定されます。

```
codecov -prj Project_Name -dpi customer.dpi -ref
appTests.dpi
```

差分カバレッジのカバレッジ統計は、新規実行で実行され、リファレンス実行で実行されなかったコードの比率をパーセントで表示します。この場合、コード・カバレッジ・ツールはカバーされなかったコードを含むモジュールのみを表示します。

ソースビューの配色も同様に解釈すべきです。コードが両方の実行で同じカバレッジ・プロパティ (カバーされたまたはカバーされなかった) の場合、コードはカバーされたコードとして扱われます。コードが新規実行でカバーされ、リファレンス実行でカバーされなかった場合、コードはカバーされなかったコードとして扱われます。逆に、コードがリファレンス実行でカバーされ、新規実行でカバーされなかった場合、差分カバレッジのソースビューはコードをカバーされたコードとして表示します。

差分カバレッジ用の実行

インテル・コンパイラのコード・カバレッジ・ツールを差分カバレッジ用に実行するには、次のファイルが必要です:

- アプリケーションのソース
- インテル・コンパイラでインストルメント済みバイナリ用に -prof_genx オプション付きでアプリケーションをコンパイルしたときに生成される .spi ファイル。
- インテル・コンパイラの profmerge ツールで各アプリケーション・テストの動的プロファイル情報ファイル (.dyn) をマージして生成される .dpi ファイル、またはインテル・コンパイラで -prof_use オプション付きでアプリケーションをコンパイルしたときに暗黙的に生成される .dpi ファイル。

「PGO の使用モデル」を参照してください。

一旦必要なファイルが利用可能になると、コード・カバレッジ・ツールを次のコマンドラインから起動できます:

```
codecov -prj Project_Name -spi pgopti.spi -dpi pgopti.dpi
```

-spi および -dpi オプションは、対応するファイルへのパスを指定します。

コード・カバレッジ・ツールには、各 HTML ページの最後に -mname および -maddr オプションを使用して指定されたアドレスに電子メールを送信するリンクを生成する、次の追加オプションもあります。

```
codecov -prj Project_Name -mname John_Smith -maddr
js@company.com
```

テスト・プライオリタイゼーション・ツール

インテル® コンパイラのテスト・プライオリタイゼーション・ツールを使用すると、以前のアプリケーション実行プロファイルを基にアプリケーション・テストの選択と重要度付けを行う、プロファイルに基づく最適化を行うことができます。ツールは、テストがボトルネックとなる大規模なアプリケーションのテストと開発にかかる時間を大幅に節約します。ツールは、IA-32 アーキテクチャと Itanium® アーキテクチャの両方で使用することができます。

ツールは、アプリケーションのコードのサブセットに対して最も適切なテストを選択し、重要度付けを行います。アプリケーションの特定のモジュールが変更された場合、テスト・プライオリタイゼーション・ツールはその変更によって最も影響を受けるテストを知らせます。ツールは、以前に実行されたアプリケーションのプロファイル・データを分析して、アプリケーションのコンポーネントとテストの依存性を確認し、この情報を基にテストのプロセスをガイドします。

特徴と利点

ツールは、アプリケーションのコード・カバレッジを基に、効率的なテスト階層を示します。ツールを使用する利点は、次のように要約することができます:

- アプリケーションの任意のサブセットで全体をカバーするために必要なテストの数を最小限にする。ツールはテストの全セットと全く同じコード・カバレッジを達成するアプリケーション・テストの最小サブセットを定義します。
- テストのターンアラウンド時間を短くする。長い時間を費して多くの失敗を見つける代わりに、ユーザは、セットの変更によって問題が発生する少数のテストを素早く見つけることができます。
- テストの実行時間のデータを基に、最短時間で特定のレベルのコード・カバレッジを達成するテストを選択して重要度付けを行います。

コマンドライン構文

このツールの構文は次のとおりです。

`tselect -dpi_list` *`file`*

`-dpi_list` は必須のツール・オプションで、重要度付けが必要なテストの `.dpi` ファイルのリストを含む DPI リストファイル (*`file`*) へのパスを設定します。

ツールのオプション

次の表は、ツールが使用するオプションの一覧です。

オプション	説明
`-help`	テスト・プライオリタイゼーション・ツールのすべてのオプションを出力します。
`-spi` *`file`*	静的プロファイル情報ファイル (`.spi`) のパス名を設定します。デフォルトは `pgopti.spi` です。
`-dpi_list` *`file`*	動的プロファイル情報ファイル (`.dpi`) の名前を含むファイルのパス名を設定します。 ファイルの各行には 1 つの `.dpi` ファイル名のみを記述し、オプションでファイル名の後にその実行時間を記述します。名前はテストを一意に識別しなければなりません。
`-prof_dpi` *`file`*	出力レポートファイルのパス名を設定します。
`-comp`	処理するファイルのリストを含むファイル名を設定します。
`-cutoff` *`value`*	累積ブロック・カバレッジがあらかじめ計算された合計カバレッジの *`value`* % に達した場合に終了します。*`value`* は 0.0 より大きくなければなりません (例えば、99.00)。100 まで

	設定できます。
-nototal	あらかじめ合計カバレッジを計算しません。
-mintime	テストの実行時間を最小限にします。各テストの実行時間は、dpi_list ファイルのテスト名が含まれる行に、テスト名に続いて dd:hh:mm:ss 形式で記述されていなければなりません。
-verbose	プログラム進行に関してより多くのロギング情報を生成します。

使用要件

アプリケーションのテストでテスト・プライオリタイゼーション・ツールを実行するには、次のファイルが必要です:

- インテル・コンパイラでインストルメント済みバイナリ用に -prof_genx オプション付きでアプリケーションをコンパイルしたときに生成される .spi ファイル。
- インテル・コンパイラの profmerge ツールで各アプリケーション・テストの動的プロファイル情報ファイル (.dyn) をマージして生成される .dpi ファイル。ユーザは、個々のテスト用に生成されたすべての .dyn ファイルに profmerge ツールを適用して、テストを一意に識別できるように .dpi ファイルに名前を付ける必要があります。profmerge ツールは指定されたディレクトリに存在するすべての .dyn ファイルをマージします。

  注
  以前に実行した、または他のテストで使用した、無関係な .dyn ファイルがディレクトリ中に含まれていないことを確認することは非常に重要です。この確認を怠ると、プロファイル情報は不適切なプロファイル・データに基づくことになります。その結果、不適切なカバレッジ情報が生成され、最適化されたコードのパフォーマンスに悪影響を与えることがあります。

- 重要度付けするテストのリストを含む、ユーザが作成したファイル。

  ツールを正しく実行するには、次の操作を行ってください:

  - 各テストが一意に識別できるように、各テスト .dpi ファイルに名前を付けます。
  - DPI リストファイル (すべての .dpi テストファイルの名前を記述したテキストファイル) を作成します。このファイルの名前がテスト・プライオリタイゼーション・ツールの実行コマンドの入力になります。DPI リストファイルの各行には、1 つの .dpi ファイル名のみを記述するようにしてく

ださい。オプションで、ファイル名に続けてテストの実行時間を dd:hh:mm:ss 形式で記述することができます。

例: Test1.dpi 00:00:60:35 は、Test1 が (0 日 0 時間) 60 分 35 秒実行されたことを示します。

実行時間の記述はオプションです。しかし、実行時間が記述されていない場合、ツールは実行時間を最小限にするテストの重要度付けを行いません。テストの数を最小限にする重要度付けのみを行います。

使用モデル

次の図は、テスト・プライオリタイゼーション・ツールの使用モデルを示しています。

IA-32 システム用の簡単な例 (myApp.f90) について順に説明します。

1. 次のように設定します:

```
PROF_DIR=c:/myApp/prof_dir
```

2. 次のコマンドを使用します:

   ```
   ifort -prof_genx myApp.f90
   ```

   このコマンドは、プログラムをコンパイルして、インストルメント済みバイナリ myApp と静的プロファイル情報ファイル pgopti.spi を生成します。

3. 次のコマンドを使用します:

   ```
   rm PROF_DIR /*.dyn
   ```

   関係しない .dyn ファイルを削除します。

4. 次のコマンドを使用します:

   ```
   myApp < data1
   ```

   このコマンドは、インストルメント済みアプリケーションを実行して、PROF_DIR で指定されたディレクトリに拡張子 .dyn で 1 つ以上の新規動的プロファイル情報ファイルを生成します。

5. 次のコマンドを使用します:

   ```
   profmerge -prof_dpi Test1.dpi
   ```

   このステップで、profmerge ツールは、すべての .dyn ファイルを、Test1 におけるアプリケーションの合計のプロファイル情報を表わす 1 つのファイル (Test1.dpi) にマージします。

6. 次のコマンドを使用します:

   ```
   rm PROF_DIR /*.dyn
   ```

   関係しない .dyn ファイルを削除します。

7. 次のコマンドを使用します:

   ```
   myApp < data2
   ```

   このコマンドは、インストルメント済みアプリケーションを実行して、PROF_DIR で指定されたディレクトリに拡張子 .dyn で 1 つ以上の新規動的プロファイル情報ファイルを生成します。

8. 次のコマンドを使用します:

   profmerge -prof_dpi Test2.dpi

   このステップで、profmerge ツールは、すべての .dyn ファイルを、Test2 におけるアプリケーションの合計のプロファイル情報を表わす 1 つのファイル (Test2.dpi) にマージします。

9. 次のコマンドを使用します:

   rm PROF_DIR /*.dyn

   関係しない .dyn ファイルを削除します。

10. 次のコマンドを使用します:

    myApp < data3

    このコマンドは、インストルメント済みアプリケーションを実行して、PROF_DIR で指定されたディレクトリに拡張子 .dyn で 1 つ以上の新規動的プロファイル情報ファイルを生成します。

11. 次のコマンドを使用します:

    profmerge -prof_dpi Test3.dpi

    このステップで、profmerge ツールは、すべての .dyn ファイルを Test3 におけるアプリケーションの合計のプロファイル情報を表わす 1 つのファイル (Test3.dpi) にマージします。

12. tests_list という名前で 3 行のファイルを作成します。1 行目に Test1.dpi、2 行目に Test2.dpi、3 行目に Test3.dpi と記述します。

    これらの項目が利用可能になると、次の例で記述されているように PROF_DIR ディレクトリでコマンドラインからテスト・プライオリタイゼーション・ツールを起動できます。

すべての例で、同じセットのデータを参照しています。

例 1 テストの数を最小限にする

```
tselect -dpi_list tests_list -spi pgopti.spi
```

/spi オプションは .spi ファイルへのパスを指定します。

テスト・プライオリタイゼーション・ツールを実行した場合の出力例を次に示します:

```
Total number of tests   =  3

Total block coverage    ~  52.17

Total function coverage ~  50.00
```

Num	%RatCvrg	%BlkCvrg	%FncCvrg	Test Name @ Options
1	87.50	45.65	37.50	Test3.dpi
2	100.00	52.17	50.00	Test2.dpi

この例で、テスト・プライオリタイゼーション・ツールは次の情報を出力しています:

- 3 つのテストすべてを実行して、52.17% のブロック・カバレッジと 50.00% の関数カバレッジを達成した。
- Test3 はアプリケーションの基本ブロックの 45.65% をカバーしている。これは 3 つのテストすべてで達成できる合計ブロック・カバレッジの 87.50% である。
- Test2 を追加することで、52.17% の累積ブロック・カバレッジまたは Test1、Test2、Test3 の合計ブロック・カバレッジの 100% を達成した。
- Test1 を除去しても合計ブロック・カバレッジに悪影響はない。

例 2 実行時間を最小限にする

tests_list ファイルで、各テストの実行時間が次のように記述されているとします:

```
Test1.dpi 00:00:60:35

Test2.dpi 00:00:10:15

Test3.dpi 00:00:30:45
```

次のコマンドは、実行時間を最小限にする -mintime オプション付きでテスト・プライオリタイゼーション・ツールを実行します:

```
tselect -dpi_list tests_list -spi pgopti.spi -mintime
```

出力例を次に示します:

```
Total number of tests   =  3

Total block coverage    ~  52.17
```

```
Total function coverage ~  50.00

Total execution time    =  1:41:35
```

num	elapsedTime	%RatCvrg	%BlkCvrg	%FncCvrg	Test Name @ Options
1	10:15	75.00	39.13	25.00	Test2.dpi
2	41:00	100.00	52.17	50.00	Test3.dpi

この例では、すべてのテストを続けて実行すると 1 時間 45 分 35 秒必要ですが、選択したテストを実行すると 41 分で同じ合計ブロック・カバレッジを達成できることが示されています。

注

重要度付けが実行時間を最小限にすることを基に行われる場合のテストの順序 (最初に Test2、次に Test3) は、重要度付けがテストの数を最小限にすることを基に行われる場合と異なることがあります。例 1 では、最初に Test3、次に Test2 になります。例 2 では、Test2 は実行時間あたりのカバレッジが最も高くなるテストです。このため、最初のテストとして選択されます。

他のオプションの使用

-cutoff オプションは、テスト・プライオリタイゼーション・ツールが指定されたレベルの基本ブロック・カバレッジに達した場合、ツールを終了します。

```
tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00
```

ツールが上記の例で 85.00 の cutoff 値で実行された場合、45.65% のブロック・カバレッジを達成する Test3 のみが選択されます。これは 3 つのテストすべてで達する合計ブロック・カバレッジの 87.50% に相当します。

テスト・プライオリタイゼーション・ツールは、すべてのテストを実行して得られる合計カバレッジを決定するために、すべてのプロファイル情報を最初にマージします。-nototal オプション このステップはスキップされます。この場合、全体的なカバレッジは不明なまま、確実なカバレッジ情報のみがレポートされます。

PGO API: プロファイル情報生成サポート

PGO API サポートの概要

プロファイル情報生成サポート (Profile Information Generation Support、Profile IGS) で、プロファイルに基づいた最適化のインストルメント済みの実行フェーズ中にプロファイル情報の生成を制御できます。

通常、プロファイル情報は、インストルメント済みアプリケーションが、標準的な `exit()` 関数を呼び出してアプリケーションを終了したときにインストルメント済みアプリケーションによって生成されます。

プロファイル情報の生成を確実にするには、このセクションで説明される関数が必要となる場合があり、また次の状況で役立ちます。

- 非標準の終了ルーチンでインストルメント済みアプリケーションが終了する場合。
- インストルメント済みアプリケーションが 非終了 アプリケーションの場合 (`exit()` は呼び出されません)。
- プロファイル情報の生成時にインストルメント済みアプリケーションが制御する必要がある場合。

関数のセットおよび環境変数が Profile IGS を構成します。

Profile IGS (Profile Information Generation Support:プロファイル情報生成サポート) 関数

Profile IGS 関数は、関数が使用されるソースファイルの上にヘッダファイルを挿入することによりアプリケーションで使用することができます。

```
#include "pgouser.h"
```

注

Profile IGS 関数は、C 言語で記述されています。Fortran アプリケーションは、C 関数を呼び出す必要があります。

このセクションの残りのトピックでは Profile IGS 関数について説明します。

注
インストルメンテーションがされていないと、Profile IGS 関数は PGO API サポートすることができません。

Profile IGS 環境変数

Profile IGS の環境変数は、PROF_DUMP_INTERVAL です。この環境変数は、インストルメント済みのユーザ・アプリケーションでインターバル・プロファイル・ダンプに使用されることがあります。詳細は、_PGOPTI_Set_Interval_Prof_Dump() の推奨する使用方法を参照してください。

プロファイル情報のダンプ

_PGOPTI_Prof_Dump() 関数は、インストルメント済みのアプリケーションにより収集されたプロファイル情報をダンプします。この関数は次のプロトタイプを持ちます。

```
void _PGOPTI_Prof_Dump(void);
```

プロファイル情報は、.dyn ファイル (PGO のフェーズ 2 で生成) に作成されます。

推奨する使用方法

ユーザ・アプリケーションを終了する関数本体にこの関数への呼び出しを挿入します。通常、_PGOPTI_Prof_Dump() の呼び出しは、一度だけでなければなりません。

この関数を _PGOPTI_Prof_Reset() 関数とともに使用して、複数の .dyn ファイル (入力データの複数のセットから) を作成することができます。

例:

```
!selectively collect profile information
! for the portion of the application
! involved in processing input data


input data = get input data()
do while (input data)
call  PGOPTI Prof Reset()
call process data(input data)
call  PGOPTI Prof Dump();
input data = get input data();
end do
```

動的プロファイル・カウンタのリセット

_PGOPTI_Prof_Reset() 関数は、動的なプロファイルのカウンタをリセットし、次のプロトタイプを持ちます。

```
void _PGOPTI_Prof_Reset(void);
```

推奨する使用方法

この関数を使用し、インストルメント済みのアプリケーションのセクションでプロファイル情報を収集する前にプロファイル・カウンタをクリアにします。
_PGOPTI_Prof_Dump() の例を参照してください。

プロファイル情報のダンプとリセット

_PGOPTI_Prof_Dump_And_Reset() 関数は、プロファイル情報を新規の .dyn ファイルにダンプしてから、動的プロファイル・カウンタをリセットします。その後、インストルメント済みのアプリケーションの実行が続行します。この関数のプロトタイプは次のとおりです。

```
void _PGOPTI_Prof_Dump_And_Reset(void);
```

この関数は、非終了アプリケーションで使用され 1 回以上呼び出される場合があります。

推奨する使用方法

この関数の定期的な呼び出しにより、非終了アプリケーションは、1 つまたは複数のプロファイル情報ファイル（.dyn ファイル）を作成することができます。これらのファイルは、プロファイルに基づく最適化のフィードバック・フェーズ（フェーズ 3）でマージされます。この関数の直接的な使用により、プロファイル情報の生成時にアプリケーションが正確に制御することができます。

インターバル・プロファイル・ダンプ

_PGOPTI_Set_Interval_Prof_Dump() 関数は、インターバル・プロファイル・ダンプをアクティブにし、ダンプ発生時のおおよその頻度を設定します。この関数呼び出しのプロトタイプは次のとおりです。

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

この関数は、非終了アプリケーションで使用します。

インターバル・パラメータは、プロファイルのダンプが発生する時間の間隔 (ミリ秒単位) で指定します。例えば、インターバルが 5000 として設定された場合、プロファイル・ダンプとリセットは約 5 秒ごとに行われます。ダンプとリセットの時間測定は、アプリケーションのインストルメントされた関数の実行時に行われるため、インターバルはおおよそとなります。

 **注**

1. インターバルを 0 または負の数に設定すると、インターバル・プロファイルのダンプを無効にします。
2. インターバルが非常に少ない値で設定されると、インストルメント済みのアプリケーションがプロファイル情報のダンプにほとんどの時間を費やしてしまう場合があります。インターバルには、できるだけ大きな値を設定し、アプリケーションが本来の作業を行え、十分なプロファイル情報が収集されるようにします。

推奨する使用方法

この関数は、インターバル・プロファイル・ダンプを開始するために、非終了ユーザ・アプリケーションの開始時に呼び出すことができます。環境変数の PROF_DUMP_INTERVAL をアプリケーションが開始する前に必要な interval の値に設定して、インターバル・プロファイル・ダンプを開始することもできます。

インターバル・プロファイル・ダンプの目的は、ソースコードへの変更を最小限に抑えて、非終了アプリケーションをプロファイルできるようにすることです。

高水準言語の最適化 (HLO)

高レベルな最適化の概要

高レベルの最適化は、Fortran および C++ などの高級プログラミング言語で開発されるアプリケーション内のソースコード構造 (例えば、ループと配列) の特性を利用します。高レベルの最適化には、ループ交換、ループ融合、ループのアンロール、ループ分配、アンロール・アンド・ジャム、ブロッキング、データ・プリフェッチ、スカラ置換、データ・レイアウトの最適化およびループのアンロール手法があります。

高レベルの最適化をオンにするには、-O3 オプションを指定します。-O3 によりオンになる最適化の範囲は IA-32 および Itanium® ベースのアプリケーションで異なります。「最適化レベルの設定」を参照してください。

IA-32 および Itanium® ベース・アプリケーション

-O3 オプションは、-O2 オプションを有効にし、またさらに強力な最適化 (例えば、ループ変換やプリフェッチ) を追加します。-O3 は、最大速度について最適化を行いますが、パフォーマンスが向上しないプログラムもあります。

IA-32 アプリケーション

ベクトル化オプション -ax{K|W|N|B|P} および -x{K|W|N|B|P} と -O3 を組み合わせて指定すると、-O2 よりも詳細にデータの依存性を分析します。このため、コンパイル時間が長くなる可能性があります。

Itanium ベース・アプリケーション

-ivdep_parallel オプションは、IVDEP ディレクティブの指定したループにループキャリー依存がないことを断定します。この手法は、スパース・マトリックス・アプリケーションに役立ちます。

Itanium ベース・アプリケーションの主要なチューニング手法

Itanium ベース・システムでアプリケーションをチューニングするには、以下の手順に従います。

1. -O3 と -ipo を使用してプログラムをコンパイルします。 可能な限り、プロファイルに基づく最適化 (PGO) を使用します。
2. コード内の Hotspot を識別します。
3. 最適化レポートを有効にします。
4. ループに対するソフトウェアのパイプライン化が行われていない理由を確認します。
    - CDEC$ ivdep を使用して、依存がないことをコンパイラに通知します。 ループキャリー依存がないことを示す -ivdep_parallel オプションが必要な場合があります。
    - CDEC$ swp を使用して、ソフトウェアのパイプライン化を有効にします (不適当なコントロールや不明なループカウントに役立ちます)。

- 必要に応じて、CDEC$ loop count(n) を使用します。
- クレイポインタが使用されている場合、safe_cray_ptr を使用してエイリアシングがないことを示します。
- CDEC$ distribute point を使用して、大きなループを分割します (通常、これは自動的に行われます)。

5. プリフェッチが正しく設定されていることを確認します。CDEC$ prefetch を使用して、必要に応じて設定を上書きします。

ループ変換

ループ変換の手法には次のものがあります。

- ループ正規化
- ループ逆転
- ループ交換/並び替え
- ループ分配
- ループ融合
- スカラ置換
- IVDEP ディレクティブによるループ・キャリー・メモリ依存の不在
- ランタイム・データ依存チェック (Itanium® ベース・システムのみ)

上記のループ変換は、データの依存関係によりサポートされています。ループ変換手法には、次のものがあります:

- 誘導変数の排除
- 定数の伝播
- コピーの伝播
- 先行代入
- および不要コード排除

以上は IA-32 アーキテクチャと Itanium アーキテクチャの両方で使用できるループ変換ですが、これ以外にも Itanium アーキテクチャではコラプス手法が利用できます。

スカラ置換 (IA-32 のみ)

スカラ置換の目的は、メモリ参照を減らすことです。これは、主に、配列参照をレジスタ参照に置き換えると行えます。

-O1 または -O2 が指定されている場合、コンパイラはいくつかの配列参照をレジスタ参照に置き換えますが、-O3 (-scalar_rep) が指定されている場合は、さらに強

力な置換を行います。例えば、-O3 を指定すると、ループキャリー依存があるかまたはメモリの一義化のためにデータ依存解析が必要な場合にコンパイラが置換を試みます。

-scalar_rep[-]	ループ変換中に実行されるスカラ置換を有効 (デフォルト) または無効にします (-O3 が必要)。

ループのアンロール (-unroll[n])

-unroll[*n*] オプションは、次のように使用します。

- *-unrolln* は、ループのアンロール回数の上限を指定します。次の例では、最大 4 回ループをアンロールします。

  ifort -unroll4 *a.f*

  ループのアンロールを無効にするには、*n* を 0 に指定します。IA-32 システムでは、0 を指定することにより、キャッシュ・ライン分割問題を解決するために、必要なアンロールを除いてベクトライザのアンローラを無効にします。次の例では、ループのアンロールが無効になります。

  ifort -unroll0 *a.f*

- -unroll (*n* を省略) は、アンロールを実行するかどうかをコンパイラが判断します。これはデフォルト設定です。コンパイラは、デフォルトのヒューリスティックを使用するか、または *n* を定義します。
- -unroll0 (*n* = 0) は、アンローラが無効になります。

Itanium® コンパイラは、現在、(*n* = 0) のみを使用します。 他の値は NOP です。

ループのアンロールの利点と制限

ループのアンロールの利点は、次のとおりです:

- アンロールにより、分岐およびいくらかのコードが減ります。
- 変数を有効に保持するのに十分な空きレジスタがある場合は、アンロールにより、積極的にループをスケジューリング (またはパイプライン化) して、遅延を隠せます。
- 反復回数が予測可能で、ループ内に条件付き分岐がない場合、Pentium® 4 プロセッサおよびインテル® Xeon™ プロセッサは、16 以下の反復を持つ内部ループの終了分岐を正しく予測できます。したがって、ループ本体のサイズが

大きすぎず、反復の予測回数がわかる場合、
– Pentium 4 プロセッサまたはインテル Xeon プロセッサの内部ループを、最大 16 の反復回数までアンロールします。
– Pentium III プロセッサまたは Pentium II プロセッサの内部ループを、最大 4 の反復回数までアンロールします。

考えられる欠点は、過度なアンロール、または非常に大きなループのアンロールにより、コードサイズが大きくなる場合あることです。

`-unroll[n]` での最適化方法の詳細は、『インテル® Pentium® 4 プロセッサおよびインテル® Xeon™ プロセッサ最適化リファレンス・マニュアル』を参照してください。

IVDEP ディレクティブのメモリの依存性

Itanium® ベース・アプリケーションの場合、`-ivdep_parallel` オプションは `IVDEP` ディレクティブが指定されたループにループ・キャリー・メモリ依存が確実にないことを示します。この手法は、スパース・マトリックス・アプリケーションに役立ちます。

例えば、次のループは、`a()` への格納でループ・キャリー依存がないことを示す IVDEP ディレクティブの他に `-ivdep_parallel` を必要とします。

```
!DIR$ IVDEP
do j=1,n
a(b(j)) = a(b(j))+1
enddo
```

「ベクトル化のサポート」も参照してください。

プリフェッチ

プリフェッチ を挿入する目的は、データをキャッシュにロードするタイミングのヒントをプロセッサに知らせてキャッシュ・ミスを減らすことです。プリフェッチによる最適化では、次のオプションを実装しています。

`-prefetch[-]`	プリフェッチ挿入を有効または無効 (`-prefetch-`) にします。このオプションでは、`-O3` が指定されている必要があります。`-O3` でのデフォルト設定は `-prefetch` です。

コンパイラの最適化を容易にするには、次のことを考慮します。

- グローバル変数およびグローバル・ポインタの使用を最小限に抑えます。

- 複雑な制御フローの使用を最小限に抑えます。
- データ型を慎重に選択し、型キャストを行わないようにします。

-prefetch[-] を使用した最適化方法の詳細は、『インテル® Pentium® 4 プロセッサおよび インテル® Xeon™ プロセッサ最適化リファレンス・マニュアル』を参照してください。

-prefetch オプションの他に、組込みサブルーチン MM_PREFETCH およびコンパイラ・ディレクティブ PREFETCH も使用できます。サブルーチン MM_PREFETCH は、1つのメモリ・キャッシュ・ライン上の指定のアドレスからデータをプリフェッチします。コンパイラ・ディレクティブ PREFETCH は、メモリからのデータ・プリフェッチを有効にします。

次の例は Itanium® ベース・システムでのみ有効です。

```
do j=1,lastrow-firstrow+1
  i = rowstr(j)
 iresidue = mod( rowstr(j+1)-i, 8 )
 sum = 0.d0
CDEC$ NOPREFETCH a,p,colidx
do k=i,i+iresidue-1
  sum = sum +  a(k)*p(colidx(k))
enddo
CDEC$ NOPREFETCH colidx
CDEC$ PREFETCH a:1:40
CDEC$ PREFETCH p:1:20
 do k=i+iresidue, rowstr(j+1)-8, 8
  sum = sum + a(k  )*p(colidx(k  ))
&       + a(k+1)*p(colidx(k+1)) + a(k+2)*p(colidx(k+2))
&       + a(k+3)*p(colidx(k+3)) + a(k+4)*p(colidx(k+4))
&       + a(k+5)*p(colidx(k+5)) + a(k+6)*p(colidx(k+6))
&       + a(k+7)*p(colidx(k+7))
  enddo
 q(j) = sum
enddo
```

詳細については、『Intel ® Fortran Language Reference』(英語) マニュアルを参照してください。

インテル® Fortran による共用メモリ並列プログラム

並列処理: 概要

ここでは、インテル® Fortran コンパイラのサポートする並列プログラミングの主要機能である OpenMP*、自動並列化、ベクトル化の自動処理について説明します。これらの機能はそれぞれ、プロセッサ数、ターゲット・アーキテクチャ (IA-32 または Itanium® アーキテクチャ)、アプリケーションの性質に応じて、アプリケーションのパフォーマンスの向上を実現します。OpenMP、自動並列化、ベクトル化の自動処理の 3 つの機能を任意に組み合わせることによって、アプリケーション・パフォーマンスを高めることもできます。

並列プログラミングを*明示的に*行う場合は、プログラマが OpenMP ディレクティブを使用して定義します。また、*暗黙的に*行う場合は、コンパイラにより自動検出されます。暗黙的な並列処理は、最も外側のループの自動並列化、または内側のループのベクトル化自動処理のいずれか (または両方) により行われます。

OpenMP と自動並列化のディレクティブで定義される並列処理は、スレッドレベルの並列処理 (TLP) に基づくものです。ベクトル化の自動処理手法により定義される並列処理は、命令レベルの並列処理 (ILP) に基づきます。

インテル Fortran コンパイラは、マルチプロセッサ・システム向けの IA-32 と Itanium の両アーキテクチャ上、ならびに、ハイパー・スレッディング・テクノロジ搭載の単一 IA-32 プロセッサ上で、OpenMP と自動並列化をサポートします (ハイパー・スレッディング・テクノロジについては、『IA-32 インテル® アーキテクチャ最適化リファレンス・マニュアル』(英語) を参照してください)。ベクトル化の自動処理は、Pentium® ファミリ、MMX® テクノロジ Pentium® プロセッサ、Pentium II プロセッサ、Pentium III プロセッサ、および Pentium 4 プロセッサでサポートされています。ベクトル化の自動処理を使ってコードのコンパイル処理を強化するために、ベクトライザ・ディレクティブをプログラムに追加することもできます。Itanium ベース・システムではこれに関連したソフトウェアのパイプライン化 (SWP) という手法が提供されています。

下記の表は、インテル Fortran コンパイラで利用可能な並列処理の各種実装方法の概要を示します。

<table>
<tr><th colspan="3">並列処理</th></tr>
<tr><th>明示的</th><th colspan="2">暗黙的</th></tr>
<tr><td>ユーザによってプログラミングされる並列処理</td><td colspan="2">コンパイラおよびユーザ指定のヒントによって生成される並列処理</td></tr>
<tr><td>OpenMP (TLP)

IA-32 および Itanium アーキテクチャ</td><td>最も外側のループの自動並列化 (TLP)

IA-32 および Itanium アーキテクチャ</td><td>最も内側のループのベクトル化自動処理 (ILP)

IA-32 のみ

Itanium アーキテクチャ向けソフトウェアのパイプライン化</td></tr>
<tr><td colspan="2">サポートするシステム</td><td>サポートするシステム</td></tr>
<tr><td colspan="2">IA-32 または Itanium ベースによるマルチプロセッサ・システム

ハイパー・スレッディング・テクノロジ対応の IA-32 システム</td><td>Pentium プロセッサ、MMX テクノロジ Pentium プロセッサ、Pentium II プロセッサ、Pentium III プロセッサ、Pentium 4 プロセッサ</td></tr>
</table>

並列プログラムの開発

インテル Fortran コンパイラは、www.openmp.org で提供されている OpenMP Fortran バージョン 2.0 API 仕様をサポートしています。OpenMP ディレクティブは、反復のパーティショニング、データの共用、スレッドのスケジューリング、および同期化に関する下位の詳細レベルを処理して、ユーザの負担を軽減します。

インテル Fortran コンパイラの自動並列化機能は、入力プログラムのシリアルな部分を同等のマルチスレッド・コードに自動的に変換します。自動並列化機能は、ワークシェアリング可能なループを特定し、正しい並列実行を確認するためにデータフロー分析を行い、OpenMP ディレクティブを使用したプログラミングに必要な、スレッドコード生成のデータをパーティショニングします。OpenMP と自動並列化アプリケーションでは、マルチプロセッサ・システムおよびハイパー・スレッディング・テクノロジ対応の IA-32 プロセッサ上の共用メモリによるパフォーマンス・ゲインも実現します。

ベクトル化の自動処理機能は、並列で実行できるプログラム内の演算を検出し、シーケンシャル・プログラムをデータ型に応じて、2、4、8、または 16 までの要素を 1 つの演算で処理するように変換します。場合によっては、自動並列化とベクトル化を組み合わせて最良のパフォーマンスを得ることができます。下記のコードでは、TLP は最も外側のループで、ILP は最も内側のループで使用できます。

```
DO I = 1, 100              !execute groups of iterations in
different
!threads (TLP)
DO J = 1, 32           !execute in SIMD style with
multimedia
!extension (ILP)
A(J,I) = A(J,I) + 1
ENDDO
ENDDO
```

ベクトル化の自動処理は、Pentium プロセッサ、MMX テクノロジ Pentium II プロセッサ、Pentium III プロセッサ、Pentium 4 プロセッサを基にしたシステム上で動作するアプリケーションのパフォーマンスを向上します。

次の表に、ベクトル化の自動処理、自動並列化、OpenMP サポートを有効にするオプションをリストします。

ベクトル化の自動処理 (IA-32 のみ)	
-x{K|W|N|B|P}	{K|W|N|B|P} で指定された拡張命令をサポートするプロセッサでのみ動作する専用コードを生成します。
-ax{K|W|N|B|P}	{K|W|N|B|P} で指定される拡張命令専用のコードを 1 つのバイナリ上に生成し、汎用の IA-32 コードも生成します。通常は、汎用コードの方が実行速度が遅くなります。
-vec_report{0|1|2|3|4|5}	ベクトライザの診断メッセージを制御します。詳細は、表の次のサブセクションを参照してください。
自動並列化、IA-32 および Itanium アーキテクチャ	
-parallel	自動パラレライザを有効にして、並列で安全に実行できるループのマルチスレッド・コードを生成します。デフォルト: オフ
-par_threshold{n}	ループの並列実行が有効な確率と比較する自動並列化のしきい値を設定します (n=0 から 100)。n=0 は "常に" を意味します。デフォルト: n = 100
-par_report{0|1|2|3}	自動並列化の診断レベルを制御します。デフォルト: -par_report1
OpenMP、IA-32 および Itanium アーキテクチャ	
-openmp	OpenMP ディレクティブに基づいてマルチスレッド・コードを生成する処理を、パラレライザに許可します。デフォルト: オフ
-openmp_report{0|1|2}	OpenMP パラレライザの診断レベルを制御します。デフォルト: /Qopenmp_report1

-openmp_stubs	シーケンシャル・モードで OpenMP プログラムのコンパイルを有効にします。OpenMP ディレクティブが無視され、OpenMP スタブ・ライブラリがリンクされます。デフォルト: オフ

> **注** -openmp と -parallel の両方がコマンドラインで指定されると、-parallel オプションは、OpenMP ディレクティブを含まないルーチンでのみ有効となります。OpenMP ディレクティブを含むルーチンでは、-openmp オプションのみが有効です。

適切なオプションを選択することには、以下の利点があります。

- 最小限の操作で、アプリケーション・パフォーマンスの向上を実現できます。
- コンパイラ機能を使用して、マルチスレッド・プログラムをより短時間で開発できます。

OpenMP ディレクティブを各自のコードに追加するだけの簡単な処理で、プログラマはシーケンシャル・ プログラムを並列プログラムに変換することができます。次に、コード内の OpenMP ディレクティブの例を示します。

```
!OMP$ PARALLEL PRIVATE(NUM), SHARED (X,A,B,C)
!Defines a parallel region
!OMP$ PARALLEL DO !Specifies a parallel region that
!implicitly contains a single DO directive
DO I = 1, 1000
NUM = FOO(B(i), C(I))
X(I) = BAR(A(I), NUM)
!Assume FOO and BAR have no side effects
ENDDO
```

自動並列化およびベクトル化の自動処理ディレクティブの例を参照してください。

ベクトル化の自動処理 (IA-32 のみ)

ベクトル化の概要

このベクトライザはインテル® Fortran コンパイラのコンポーネントで、MMX® テクノロジ、SSE、SSE2、および SSE3 命令セットで SIMD 命令を自動的に使用します。ベクトライザは、並列に実行できるプログラム内の演算を検出し、データ型により、2、4、8、または 16 までの要素を並列で処理する 1 つの SIMD 命令のようなシーケンシャル演算を変換します。

このセクションでは、インテル Fortran コンパイラのベクトル化に関するオプションとガイドラインについて、例を挙げて説明します (ベクトル化は、IA-32 コンパイラだけがサポートする機能です)。詳細については、「コンパイラの最適化に関する文献」を参照してください。

このセクションの内容は次のとおりです。

- ベクトル化を制御するためのオプションの説明
- ベクトル化のプログラミングにおける主要ガイドライン
- ベクトル化の各段階についての解説と一般的なガイドライン
  - 自動ベクトル化
  - ユーザの介入によるベクトル化
- ベクトル化を行うときの問題点とその解決方法の具体例

インテル Fortran コンパイラは、効率的なベクトル命令を生成する多様なディレクティブをサポートしています。ベクトル化をサポートするコンパイラ・ディレクティブを参照してください。

ベクトライザのオプション

ベクトル化は、IA-32 固有の機能で、次の表は、ベクトル化に関するコマンドライン・オプションについてまとめたものです。ベクトル化は、メモリ参照を一義化するコンパイラの機能に依存します。ある種のオプションを使用すると、コンパイラによるベクトル化を改善できます。これらのオプションによって、ベクトル化だけでなく、それ以外の最適化も有効になります。`-x{K|W|N|B|P}` または `-ax{K|W|N|B|P}` オプションを使用する場合、`-O2` (デフォルトではオン) も有効であると、ベクトライザは有効になります。`-x{K|W|N|B|P}` または `-ax{K|W|N|B|P}` オプションの場合は、`-O1` および `-O3` オプションでも、ベクトライザが有効になります。

`-x{K|W|N|B|P}`	`{K|W|N|B|P}` で指定された拡張命令をサポートするプロセッサでのみ動作する専用コードを生成します。詳細は、「プロセッサ固有の自動最適化 (IA-32 のみ)」を参照してください。
`-ax{K|W|N|B|P}`	`{K|W|N|B|P}` で指定される拡張命令専用のコードを 1 つのバイナリ上に生成し、汎用の IA-32 コードも生成します。通常は、汎用コードの方が実行速度が遅くなります。詳細は、「プロセッサ固有の自動最適化 (IA-32 のみ)」を参照してください。

`-vec_report {0|1|2|3|4|5}`
デフォルト:
`-vec_report1`
	ベクトライザの診断メッセージを制御します。詳細は、表の次のサブセクションを参照してください。

ベクトル化レポート

-vec_report{0|1|2|3|4|5} オプションは、異なる情報のレベルでベクトル化レポートを生成するようにコンパイラに指示します:

-vec_report0: 診断情報を表示しません。

-vec_report1: ループが正常にベクトル化されたことを示す診断情報を表示します(デフォルト)。

-vec_report2: -vec_report1 のメッセージに加えて、ループのベクトル化が失敗したことを示す診断情報も表示します。

-vec_report3: -vec_report2 のメッセージに加えて、判明した依存関係または想定される依存関係についての補足情報も表示します。
-vec_report4: 非ベクトル化ループを示します。
-vec_report5: 非ベクトル化ループを示し、ベクトル化されない理由を示します。

数字の指定なしで -vec_report を指定すると、-vec_report1 のデフォルトが使用されます。

その他のオプションの使用方法

ベクトル化レポートは、実行ファイルの作成時の最終コンパイル・フェーズで生成されます。そのため、コマンドラインで -c オプションおよび -vec_report{n} オプションを使用すると、レポートは作成されません。

-c、-ipo と、-x{ K|W|N| B|P} または -ax{ K|W|N| B|P} と -vec_report{n} を使用すると、コンパイラは警告メッセージを表示し、レポートを生成しません。

前述のオプションを使用してレポートを生成するには、-ipo_obj オプションを追加する必要があります。-c および -ipo_obj の組み合わせで 1 つのファイルをコンパイルし、それが、オブジェクト・コードを生成し、最終的にレポートが生成されます。

次のコマンドで、ベクトル化のレポートが生成されます。

ifort -x{K|W| N| B|P } -vec_report3 *file.f*

ifort -x{K|W| N| B|P } -ipo -ipo_obj -vec_report3 *file.f*

ifort -c -x{K|W| N| B|P } -ipo -ipo_obj -vec_report3 *file.f*

ループの並列化とベクトル化

`-parallel` オプションと `-x{K|W|N|B|P}` オプションを組み合わせると、同一のコンパイルで、自動ループ並列化と自動ループベクトル化の両方を試みるようにコンパイラに指示します。多くの場合、コンパイラは、並列化には最も外側のループ、ベクトル化には最も内側のループを認識します。しかし、有効であると判断された場合、コンパイラは、同じループに並列化とベクトル化を適用します。「効率的な自動並列化の使用方法のガイドライン」および「ベクトル化のプログラミングにおける主要ガイドライン」を参照してください。

ループ並列化（自動または OpenMP*ディレクティブのいずれか）の成功は、コンパイラにレポートされる非ベクトル化ループにおけるメッセージに影響する場合があるので注意してください。

ベクトル化のプログラミングにおける主要ガイドライン

ベクトル化を行うコンパイラの目的は、SIMD（single-instruction multiple data）処理を自動的に活用することですが、コンパイラに追加情報（ディレクティブなど）を提供することで、コンパイラのベクトル化を助けます。次に示すガイドラインと制約条件をよく確認し、先の項に示すコードの具体例を見て、それらに照らしてコードの良否を判断し、最適なベクトル化を阻害する曖昧な部分が見つかったら修正してください。

ループに変更が必要な場合がよくあります。ループ本体のガイドラインを次に示します。

好ましいもの:

- わかりやすいコード（単一の基本ブロック）
- ベクトルデータのみ。すなわち、代入文の右辺には、配列と不変式を使用します。代入式の左辺には配列参照を使用してもかまいません。
- 代入文のみ

避けるべきもの:

- 関数呼び出し
- ベクトル化できない演算（算術以外）
- ベクトル化できる複数の型を同じループの中に混在させること
- データ依存性を持つループ出口条件
- ループのアンロール（これはコンパイラが行います）

- 本体の中で複数の文で構成している 1 つのループを複数の単文ループに解体すること

注意が必要な制約条件があります。ベクトル化は、2 つの主要な要因により制約されます。

- ハードウェア: コンパイラは、それを動かすハードウェアからいくつか制約を受けます。ストリーミング SIMD 拡張命令を実行する場合、ベクトルメモリ演算は、16 バイトでアライメントしたメモリ参照が優先するため、アクセスがストライド 1 に制限されます。つまり、コンパイラは、たとえループをベクトル化可能と抽象的に認めたとしても、別のターゲット・アーキテクチャに対してはベクトル化をしないかもしれません。
- コードの書きかた: ソースコードの書きかたによっては最適化の妨げになりえます。例えば、グローバル・ポインタを使用する場合に、2 つのメモリ参照が別々の場所で行われるかどうかをコンパイラが判定できないという問題がよく起こります。そうなると、一部の変換処理は順序の並べ替えができなくなります。

コンパイラによるベクトル化の自動処理を阻害する多くの要因は、ループ構造の書きかたにあります。ループ本体にはキーワード、演算子、データ参照、およびメモリ演算が含まれており、それらが互いに作用し合うため、ループの動作がよく見えなくなるのです。

しかし、これらの制約を理解し、診断メッセージの読みかたを知れば、そうした既知の制約条件を克服して効率よくベクトル化できるようプログラムを修正できます。以降の各セクションでは、ループ構造についてベクトライザの機能と制約条件を簡単に述べます。

データ依存性

データ依存の関係とは、連続したいくつかのループに含まれている各演算の実行順序を制限する関係のことです。ベクトル化によって演算の実行順序が並び替えられるため、自動ベクトライザでは任意のデータ依存性の解析を自由に使用できなければなりません。

データの依存関係によりベクトル化が妨げられる例を次に示します。この例に示す配列の各要素の値は、前の繰返しで計算された前後の要素の値により決まります。

データ依存性を持つループの例:

```
REAL DATA(0:N)
INTEGER I
```

```
DO I=1, N-1
DATA(I) =DATA(I-1)*0.25+DATA(I)*0.5+DATA(I+1)*0.25
END DO
```

上記の例に示すループは、ベクトル化できません。これは、現在の要素 DATA(I) への WRITE が直前の要素 DATA(I-1) の使用に依存しており、この要素が直前の反復時にすでに書き込まれ変更されているためです。このことは、次の例に示すように、最初の 2 回の反復における配列のアクセスパターンを見ればわかります。

データ依存性を持つループをベクトル化したものの例:

```
I=1: READ DATA (0)
READ DATA (1)
READ DATA (2)
WRITE DATA (1)
I=2: READ DATA(1)
READ DATA (2)
READ DATA (3)
WRITE DATA (2)
```

このループが示す通常のシーケンスでは、2 回目の反復時に読み込まれる DATA(1) の値は、最初の反復時に書き込まれます。ベクトル化を行うためには、元のループのセマンティクスを変えることなく、対象となるすべての反復を並列に実行しなければなりません。

データ依存性の解析

データ依存性の解析とは、2 つのメモリアクセスの重なり合う条件を見つけることです。その条件は、1 つのプログラムの中で参照を 2 回行うと仮定した場合は、次の 2 つの事項によって規定されます。

- 参照した変数がメモリ内の同一 (オーバラッピング) 領域および配列参照の別名であるかどうか
- 添字間の関係

IA-32 の場合、配列参照のためのデータ依存アナライザは一連のテストとして構成され、時間とスペースコストに加えて性能においても段階的に強化していきます。どれか 1 つの次元の中にでも独立性が認められれば、それによって依存関係が排除できるため、最初は 1 次元ずつ単純な検定をいくつか実行します。宣言されている次元境界を超える恐れのある多次元配列参照は、テストを実施する前に、線形形式に変換できます。

簡単なテストとして、高速最大公約数 (GCD) テストや拡張限界テストなどを使用できます。GCD テストでは、ループ・インデックスの係数の GCD で定数項を均等に等分できない場合、データの独立性が証明されます。拡張限界テストでは、添字式におい

て極値がオーバラップする可能性があるかどうかをチェックします。どのような簡単なテストでもデータの独立性を証明できなかった場合は、Fourier-Motzkin 消去法を使用する強力な階層化された依存関係ソルバを最終的に用いて、すべての次元におけるデータの依存関係に関する問題を解決します。

ループ構造

ループには、一般的な DO-END DO や DO WHILE、あるいは IF/GOTO 文やラベルを使用できます。ただし、ループは、入口が 1 つだけでかつ出口が 1 つだけでなければ、ベクトル化できません。ループ構造の正しい使用例と誤った使用例を次に示します。

正しい使用例:

```
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100), C(100)
INTEGER I
I = 1
DO WHILE (I .LE.100)
A(I) = B(I) * C(I)
IF (A(I) .LT.0.0) A(I) = 0.0
I = I + 1
ENDDO
RETURN
END
```

誤った使用例:

```
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100), C(100)
INTEGER I
I = 1
DO WHILE (I .LE.100)
A(I) = B(I) * C(I)
C The next statement allows early
C exit from the loop and prevents
C vectorization of the loop.
IF (A(I) .LT.0.0) GOTO 10
I = I + 1
ENDDO
10 CONTINUE
RETURN
END
```

ループ出口条件

ループ出口条件とは、1 つのループの反復回数を決める条件のことです。例えば、ループが固定インデックスを持つ場合、ループの反復回数はその固定インデックスによって決められます。ループの反復はカウントできるものでなければなりません。すなわち、反復の回数は、次のいずれかで表す必要があります。

- 定数
- ループ不変項
- 最も外側のループ添字の線形関数

ループの出口が計算結果によって左右されるようなループは、その反復回数を数えられません。次の例は、可算/不可算ループの構成要素を表しています。

可算ループによる正しい使用: 例 1

```
SUBROUTINE FOO (A, B, C, N, LB)
DIMENSION A(N),B(N),C(N)
INTEGER N, LB, I, COUNT
!Number of iterations is "N - LB + 1"
COUNT = N
DO WHILE (COUNT .GE.LB)
A(I) = B(I) * C(I)
COUNT = COUNT - 1
I = I + 1
ENDDO  !LB is not defined within loop
RETURN
END
```

可算ループによる正しい使用: 例 2

```
!Number of iterations is (N-M+2) /2
SUBROUTINE FOO (A, B, C, M, N, LB)
DIMENSION A(N),B(N),C(N)
INTEGER I, L, M, N
I = 1;
DO L = M,N,2
A(I) = B(I) * C(I)
I = I + 1
ENDDO
RETURN
END
```

不可算ループによる誤った使用の例

```
!Number of iterations is dependent on A(I)
SUBROUTINE FOO (A, B, C)
```

```
DIMENSION A(100),B(100),C(100)
INTEGER I
I = 1
DO WHILE (A(I) .GT.0.0)
A(I) = B(I) * C(I)
I = I + 1
ENDDO
RETURN
END
```

ベクトル化されるループの種類

整数ループの場合、64 ビットの MMX® テクノロジおよび 128 ビットのストリーミング SIMD 拡張命令 2 (SSE2) は 32 ビット、16 ビット、および 8 ビットの整数データ型を使用するほとんどの算術演算子と論理演算子に対して SIMD 命令を提供します。整数丸め演算の最終的な精度を保持するのであればベクトル化できるときがあります。例えば、最後に格納された値が 16 ビット整数である場合には、32 ビットの右シフト演算子は 16 ビットモードではベクトル化されません。また、MMX テクノロジ、および SSE2 命令セットは完全に直交型ではない (例えば、バイト・オペランド上のシフトはサポートされていない) ため、実際にはすべての整数演算をベクトル化ができないので注意してください。

32 ビット単精度および 64 ビット倍精度の浮動小数点数を操作するループの場合、SSE/SSE2 は算術演算子 '+'、'−'、'*'、'/' に対して SIMD 命令を提供します。また、SSE/SSE2 は 2 項演算子 `MIN` と `MAX` および単項演算子 `SQRT` に対して SIMD 命令を提供します。これ以外の複数の数値演算子の SIMD バージョン (三角関数 SIN、COS、TAN など) は、インテル® Fortran コンパイラに添付されているベクトル数値ランタイム・ライブラリ内のソフトウェアでサポートしています。

ストリップ・マイニングとクリーンアップ

ループのセクション化として知られるストリップ・マイニングは、メモリのパフォーマンスを改善する手段を提供し、ループの SIMD エンコーディングを可能にするループ変換テクニックです。大きなループをより小さなセグメントやストリップに断片化することで、このテクニックは次の 2 つの方法でループ構造を変更します:

- データがアルゴリズムの異なるパスで再使用可能な場合、データ・キャッシュ中の一時的な空間が増加します。
- 各 "ベクトル" の長さ、または SIMD 演算ごとに実行される演算の数により、ループの反復数は減ります。ストリーミング SIMD 拡張命令の場合、このベクトルまたはストリップの長さは 4 回 (1 つのストリーミング SIMD 拡張単精度浮動小数点 SIMD 演算が処理されるごとに 4 つの浮動小数点データ項目が) 減ります。

ベクトライザに最初に導入される場合、このテクニックは指定されたベクトルマシンの最大ベクトル長以下のサイズで各ベクトル演算が完了したときに生成されるコードからなります。

コンパイラは、自動的にループをストリップ・マイニングし、クリーンアップ・ループを生成します。

ストリップ・マイニングとクリーンアップ・ループの例:

!ベクトル化前

```
i = 1
do while (i<=n)
a(i) = b(i) + c(i) !Original loop code
i = i + 1
end do
```

!ベクトル化後

```
!The vectorizer generates the following two loops
i = 1
do while (i < (n - mod(n,4)))
!Vector strip-mined loop.
a(i:i+3) = b(i:i+3) + c(i:i+3)
i = i + 4
end do
do while (i <= n)
a(i) = b(i) + c(i)      !Scalar clean-up loop
i = i + 1
end do
```

ループ・ブロッキング

ループ・ブロッキングを、2 次元またはそれ以上の次元おけるストリップ・マイニングとして処理することができます。ループ・ブロッキングは、メモリ・パフォーマンスの最適化に有用な手法です。その主な目的は、できるだけ多くのキャッシュ・ミスを排除することです。メモリ領域全域をシーケンシャルにトラバースするのではなく、小さなチャンクに変換します。特定の計算用の全データがキャッシュに格納できるように、各チャンクのサイズは小さいことが条件になります。これにより、最大限のデータ再利用が可能になります。

次の例について考えてみます。2 次元配列 A は、まず j (列) 方向に、その後、i (行) 方向に参照されます (列優先順)。配列 B は逆の順序で参照されます (行優先

順)。メモリ配置が列優先順であると仮定し、そのため、このコード の配列 A と B のアクセスのストライドは、それぞれ、1 と MAX になります。

B. の例では、BS = block_size となり、MAX は BS で割り切れる必要があります。

配列のループ・ブロッキングの例:

A. 元のループ

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX
 DO J = 1, MAX

   A(I,J) = A(I,J) + B(J,I)

 ENDDO
ENDDO
```

B. ブロッキングの後の変換されたループ

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX, BS
 DO J = 1, MAX, BS

   DO II = I, I+MAX, BS-1

     DO J = J, J+MAX, BS-1

       A(II,JJ) = A(II,JJ) + B(JJ,II)

     ENDDO

   ENDDO

 ENDDO
ENDDO
```

ループ本体の文

ベクトル化可能な演算は、浮動小数点データと整数データとで異なります。

浮動小数点配列の演算

ループ本体の文には、REAL 型の演算 (通常は配列) を使用できます。数値演算は、加算、減算、乗算、除算、否定、平方根、最大値、最小値および SIN 、COS などの数値関数をサポートしています。浮動小数点からの変換、または浮動小数点への変換はできないものがあります。DOUBLE PRECISION 型の演算は、-xW または -axW

コンパイラ・オプションを使用したインテル® Pentium® 4 プロセッサ、およびインテル® Xeon™ プロセッサ・システム、インテル Pentium M プロセッサの最適化以外では無効です。

整数配列の演算

ループ本体の文には、算術演算または論理演算 (これも、通常は配列) を使用できます。算術演算は、加算、減算、ABS、MIN、および MAX に制限されます。論理演算には、ビット単位の AND、OR、および XOR の演算子を含んでいます。変換を実行しても精度が失われない場合にのみ、異なるデータ型を混在できます。異なるデータ型を混在できる演算子の例には、乗算演算子、シフト演算子、単項演算子などがあります。

その他の演算

前述の浮動小数点演算と整数演算以外の文は使用できません。上記で説明された以外、ループの本体には、関数呼び出しは使用できません。

ベクトル化の例

このセクションでは、ベクトル・プログラミングの一般的な問題を簡単な例を用いていくつか紹介します。

引数のエイリアシング : ベクトルコピー

この例はベクトルのコピー操作を行うループですが、コンパイラが DEST(A(I)) と DEST(B(I)) の相違を証明できないため、ベクトル化は行われません。

相違を証明できないためにベクトル化不能なコピーの例:

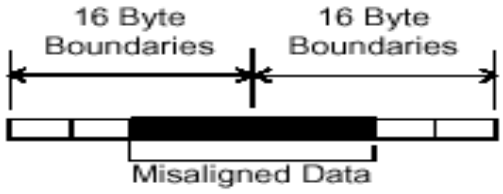
```
SUBROUTINE VEC COPY(DEST,A,B,LEN)
DIMENSION DEST(*)
INTEGER A(*), B(*)
INTEGER LEN, I
DO I=1,LEN
DEST(A(I)) = DEST(B(I))
END DO
RETURN
END
```

データのアライメント

16 バイト以上のデータ構造体または配列は、ベースアドレスが 16 の倍数になるように、各構造体要素または配列要素の先頭をアライメントしなければなりません。

「16 バイト境界をまたがる間違ってアライメントされたデータ」(下図) は、間違ってアライメントされたデータが原因でデータ・キャッシュ・ユニット (DCU) が分割した場合の影響を示したものです。アライメントの合っていないデータをロードすると 16 バイト境界にまたがるため、メモリアクセスが 1 回余分に発生し、その結果、6～12 サイクルのストールが発生します。データがアライメントされていることがわかっており、このアライメントを使用するよう指定すれば、このストールを回避できます。

16 バイト境界にまたがる間違ってアライメントされたデータ



ベクトル化の実行後、下の図に示すように、ループが実行されます。

ベクトルおよびスカラのクリーンアップ反復処理

要素 `A(1)` と `B(1)` が両方とも 16 バイトでアライメントされている場合、ベクトル反復 `A(1:4) = B(1:4);` と `A(5:8) = B(5:8);` はどちらも、アライメントされた移動によって実現できます。

> **注**不正なアライメント・オプションでベクトライザを指定すると、コンパイラの動作は予測がつかなくなります。
> 特に、アライメントの合っていないデータに対してアライメントされた移動を使用すると、不正命令例外が発生します。

アライメント手法

コンパイラは、コンパイル時にデータ構造のアライメントがわからない場合に備えて、いくつかのアライメント手法を持っています。簡単な例を下に示します (これ以外にもいくつかの手法がサポートされています)。次の例のループにおいて、A のアライメントがわかっていない場合、コンパイラはプレリュード・ループを生成し、ほとんどの場合に発生する配列参照がアライメントされたアドレスにヒットするまでループを繰返します。これにより、A のアライメント・プロパティが判明し、そのプロパティに応じてベクトルループが最適化されます。この場合、ベクトライザはインテル® Fortran 特有の機能であるダイナミックなループ・ピーリングを実行します。

データのアライメントの例:

元のループ

```
SUBROUTINE DOIT(A)
REAL A(100)      !alignment of argument A is unknown
DO I = 1, 100
A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

データのアライメント:

```
!The vectorizer will apply dynamic loop peeling as follows:
SUBROUTINE DOIT(A)
REAL A(100)
!let P be (A%16)where A is address of A(1)
IF (P .NE.0) THEN
P = (16 - P) /4    !determine run-time peeling

           !factor
DO I = 1, P
A(I) = A(I) + 1.0
ENDDO
ENDIF
!Now this loop starts at a 16-byte boundary,
!and will be vectorized accordingly
DO I = P + 1, 100
A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

ループ交換と添字: マトリックス乗算

マトリックス積は、一般に次の例のように記述されます。

```
DO I=1, N
 DO J=1, N
   DO K=1, N
     C(I,J) = C(I,J) + A(I,K)*B(K,J)

   END DO

 END DO
END DO
```

B(K,J) を使用するのは、ストライド-1 での参照ではないため、通常はベクトル化できません。しかし、ループを交換すると、次の「ストライド-1 でのマトリックス積の例」に示すように、すべての参照がストライド-1 となります。

> **注**
> 依存関係があるため、交換は常に可能であるとは限りません。依存関係によって異なる結果になる可能性があります。

ストライド-1 でのマトリックス積の例:

```
DO J=1,N
 DO K=1,N

   DO I=1,N
     C(I,J) = C(I,J) + A(I,K)*B(K,J)

   ENDDO

 ENDDO
ENDDO
```

詳細については、「コンパイラの最適化に関する文献」を参照してください。

自動並列化

自動並列化の概要

インテル® Fortran コンパイラの自動並列化機能は、入力プログラムの連続部分を同等のマルチスレッド・コードに自動的に変換します。自動パラレライザは、プログラム

のループのデータフローを分析して、安全かつ効率的に並列実行可能なループに対するマルチスレッド・コードを生成します。これにより、SMP (対称型マルチプロセッサ) システムの並列アーキテクチャを活用できます。

自動並列化は次のようなユーザの負担を軽減します。

- ワークシェアリング候補であるループを見つけなければならない。
- 正しい並列実行を確認するためにデータフロー分析を行う。
- OpenMP* ディレクティブのプログラミングに必要な場合、スレッドコード生成のデータをパーティショニングする。

並列ランタイム・サポートは、ループの反復修正、スレッド・スケジューリング、および同期化の詳細を処理するような、OpenMP と同じランタイム機能を提供します。

OpenMP ディレクティブはシリアル・アプリケーションを素早く並列アプリケーションに変換できる一方、並列処理を含み、適切なコンパイラ・ディレクティブを追加するアプリケーション・コードの特定部分を、プログラマは明示的に識別する必要があります。-parallel オプションで起動された自動並列化は、並列処理を含むループ構造を自動的に識別します。コンパイル中、コンパイラは、並列処理のためにコード・シーケンスを別々のスレッドに自動的に分解しようと試みます。他にプログラマにかかる負荷はありません。

次の例は、2 つのスレッドで同時に実行できるようにループの反復を分割する方法を示します。

オリジナルの連続コード

```
do i=1,100
a(i) = a(i) + b(i) * c(i)
enddo
```

変換された並列コード

```
Thread 1
do i=1,50
a(i) = a(i) + b(i) * c(i)
enddo
Thread 2
do i=51,100
a(i) = a(i) + b(i) * c(i)
enddo
```

自動並列化のプログラミング

自動並列化機能は、ワークシェアリング構造 (PARALLEL DO ディレクティブ) などの OpenMP* のいくつかのコンセプトを取り入れています。ワークシェアリング構造につ

いては、「OpenMP によるプログラミング」を参照してください。ここでは、自動並列化について説明します。

効率的な自動並列化の使用方法のガイドライン

次の場合、ループは並列化が可能です。

- ループがコンパイル時にカウント可能な場合。つまり、ループの実行回数（ループ反復回数）を表す式が、ループに入る直前に生成されることを意味しています。
- FLOW（WRITE の後の READ）、OUTPUT（WRITE の後の WRITE）、または ANTI（READ の後の WRITE）ループ・キャリー・データ依存性がない場合。同じメモリの場所が、ループの異なる反復で参照されるときに、ループ・キャリー・データ依存が起こります。コンパイラの判断で、推測されたループキャリー依存性がランタイム依存性のテストで解決されると、ループは並列化されることがあります。

コンパイラは、コンパイル時に定数ではないループ・パラメータを持つ parallel for ループで、実行するメリットを検証するためにランタイム・テストを生成します。

コーディング・ガイドライン

次のコーディング・ガイドラインにより自動パラレライザの威力と効率を強化することができます。

- 可能な限りループの最大繰返し回数を明記します。特に最大繰返し回数が既知でローカル変数にループ引数を保存する場合には定数を使用します。
- コンパイラがキャリー依存データと判断する可能性のある構造（プロシージャ呼び出し、不明瞭な間接参照、グローバル参照など）をループ本体内に配置しないでください。
- 想定されたデータ依存を一義化するには、!DEC$ PARALLEL ディレクティブを挿入します。
- スレッド間の共用で生じるオーバーヘッドを正当化するには、適切ではない動作を含むループの前に !DEC$ NOPARALLEL ディレクティブを挿入します。

自動並列化のデータフロー

自動並列化の処理では、コンパイラは次のステップを実行します。

データフローの解析 ----> ループの分類----> 依存性の解析 ----> 高度な並列化 ----> データのパーティショニング ----> マルチスレッド・コードの生成

これらのステップには次のものが含まれます。

- データフローの解析: プログラムを通してデータのフローを計算します。
- ループの分類: しきい値解析で示されるように 正確さと効率に基づいて並列化のループの候補を決定します。
- 依存性の解析: 各ループのネストで参照における依存性の解析を計算します。
- 高度な並列化:
  - 依存性のグラフを解析し、並列で実行できるループを決定します。
  - ランタイムの依存性を計算します。
- データのパーティショニング: SHARED、PRIVATE、および FIRSTPRIVATE のアクセスのタイプに基づいて、データ参照とパーティションを検査します。
- マルチスレッド・コードの生成:
  - ループのパラメータを修正します。
  - スレッドタスクごとに入口/出口を生成します。
  - スレッド生成と同期化の並列ランタイム・ルーチンへの呼び出しを生成します。

自動並列化: 有効、オプション、ディレクティブ、および環境変数

自動パラレライザを有効にするには、-parallel オプションを使用します。-parallel オプションは、並列で安全に実行できる並列ループを検出し、自動的にこれらのループのマルチ・スレッドコードを生成します。自動並列化を使用するコマンド例は、次のとおりです。

```
ifort -c -parallel myprog.f
```

自動並列化オプション

-O2 (または -O3) 最適化オプションがオン (デフォルトは -O2) の場合、 -parallel オプションは自動並列化を有効にします。-parallel オプションは、並列で安全に実行できる並列ループを検出し、自動的にこれらのループのマルチ・スレッドコードを生成します。

-parallel	自動パラレライザを有効にします。

-par_threshold{0-100}	自動並列化に必要な作業しきい値を制御します。
-par_report{1|2|3}	自動並列化の診断メッセージを制御します。詳細は、後のサブセクションを参照してください。

自動並列化ディレクティブ

自動並列化では、2 つの特定のディレクティブ !DEC$ PARALLEL と !DEC$ NOPARALLEL が使用されます。

インテル® Fortran 自動並列化コンパイラ・ディレクティブの形式は次のとおりです:

```
<prefix> <directive>
```

上記のカッコの意味は次のとおりです:

- <xxx>: プリフィックスとディレクティブが必要

固定形式のソース出力の場合、プリフィックスは !DEC$ または CDEC$ です。

自由形式のソース出力の場合、プリフィックスは !DEC$ のみです。

プリフィックスの後には、ディレクティブがきます。次に例を示します:

!DEC$ PARALLEL

自動並列化ディレクティブは感嘆符で始まるため、-parallel オプションを省略した場合は、ディレクティブはコメントの形式をとります。

例

!DEC$PARALLEL ディレクティブは、すぐ後のループで存在していると想定され、正常な自動並列化を妨げる依存を無視するようにコンパイラに指示します。しかし、依存が証明されると無視されません。

!DEC$ NOPARALLEL ディレクティブは、すぐ後のループの自動並列化を無効にします。

```
program main
parameter (n=100)
integer x(n),a(n)
```

```
!DEC$ NOPARALLEL
do i=1,n
x(i) = i
enddo

!DEC$ PARALLEL
do i=1,n
a( x(i) ) = i
enddo
end
```

自動並列化の環境変数

オプション	説明	デフォルト
OMP_NUM_THREADS	使用されるスレッド数を制御します。	実行ファイルを生成する際にシステムに現在搭載されているプロセッサ数
OMP_SCHEDULE	ランタイム・スケジューリングのタイプを指定します。	スタティック

自動並列化のしきい値制御と診断

しきい値制御

-par_threshold{*n*} オプションは、並列ループ実行の有効性に基づいて、ループの自動並列化のしきい値を設定します。*n* の値は 0 から 100 までを設定できます。デフォルト値は 100 です。-par_threshold{*n*} オプションは、コンパイル時に計算量が確定できないループに使用します。

n に使用する値には、次の意味があります。

- *n* = 100。コンパイラ分析データに基づいてパフォーマンス・ゲインが予測される場合にのみ並列化が実行されます。これはデフォルトです。-par_threshold{*n*} がコマンドラインで指定されていないか、n の値が指定されないまま使用されている場合に、この値が使用されます。
- *n* = 0、-par_threshold0 は指定されます。ループは、計算量に関わらず自動並列化を行います。つまり、常に並列化します。
- 1 から 99 の値は、有効な速度の向上の可能性の比率を表します。例えば、*n*=50 の場合、並列実行された場合にコードの速度が向上する可能性が 50 パーセントの場合に並列化を行います。

コンパイラは、作成された複数のスレッドのオーバーヘッドとスレッド間を共有できるワーク量のバランスをとろうとするヒューリスティックを適用します。

診断

-par_report{0|1|2|3}オプションは、自動パラレライザの診断レベル 0、1、2、3 を次のように制御します。

-par_report0 = 診断情報を表示しません。

-par_report1 = 正常に自動並列化されたループを表示します。(デフォルト) 並列ループに対して "LOOP AUTO-PARALLELIZED" メッセージを出力します。

-par_report2 = 正常に自動並列化されたループおよび自動並列化が失敗したループを表示します。

-par_report3 = 2 と同じです。また、自動並列化を妨げる、実証された依存および推測された依存についての追加情報を示します (並列化されない理由)。

次の例は、コマンドからの結果として -par_report3 により生成された出力を示しています。

ifort -c -parallel -par_report3 *myprog.f90*

ここで、myprog.f90 は、次のとおりです:

```
 program myprog
      integer a(10000), q
C Assumed side effects
      do i=1,10000
         a(i) = foo(i)
      enddo
C Actual dependence
      do i=1,10000
         a(i) = a(i-1) + i
      enddo
      end
```

-par_report 出力の例

```
program myprog
procedure: myprog
serial loop: line 5: not a parallel candidate
due to statement at line 6
serial loop: line 9
  flow data dependence from line 10 to line
```

```
10, due to "a"
12 Lines Compiled
```

トラブルシューティングのヒント

- -par_threshold0 は、コンパイラが計算量が十分でないと想定しているかどうかを確認します。
- -par_report3 で診断を表示します。
- !DIR$PARALLEL ディレクティブで想定されたデータ依存を排除します。
- -ipo を使用して、推定される関数の呼び出しへの副作用を回避します。

OpenMP* による並列化

OpenMP*による並列化の概要

インテル® Fortran コンパイラは、OpenMP* Fortran バージョン 2.0 API 仕様のうち、WORKSHARE ディレクティブを除くすべてをサポートしています。OpenMP は、次の主な機能を持った対称型マルチプロセッシング (SMP) を提供します:

- 反復のパーティショニング、データの共有、スレッドのスケジューリング、および同期化に関する下位の詳細レベルを処理して、ユーザの負担を軽減します。
- 共有メモリ、マルチプロセッサ・システムの場合に得られるパフォーマンスを提供します。また、IA-32 システムではハイパー・スレッディング・テクノロジが有効な場合に得られるパフォーマンスを提供します。ハイパー・スレッディング・テクノロジについては、『IA-32 インテル® アーキテクチャ最適化リファレンス・マニュアル』(英語) を参照してください。

インテル Fortran コンパイラは、ソース・プログラムのユーザの OpenMP ディレクティブの指定に従って、コード変換を実行し、マルチスレッド・コードを生成して、既存のソフトウェアへスレッドを追加しやすくします。インテル® コンパイラは、現在の業界標準の OpenMP ディレクティブのすべてに対応しています。ただし、WORKSHARE および OpenMP ディレクティブの注釈のある並列実行プログラムのコンパイラは除きます。

さらに、インテル Fortran コンパイラは、ランタイム・ライブラリ・ルーチンおよび環境変数を含む OpenMP Fortran バージョン 2.0 仕様にインテル独自の拡張機能を提供します。

インテル Fortran コンパイラの OpenMP 機能の全オプションについては、「概要: 並列プログラミング」を参照してください。OpenMP 標準の詳細については、Web サイト

http://www.openmp.org をご覧ください。Fortran 言語の詳細な仕様については、「OpenMP Fortran version 2.0 仕様」を参照してください。

OpenMP による並列処理

OpenMP でコンパイルするには、Fortran プログラム・コメント形式の OpenMP ディレクティブでコードを注釈するプログラムを準備する必要があります。インテル Fortran コンパイラは、はじめにアプリケーションを処理して、コードのマルチスレッド・バージョンを生成してからコードをコンパイルします。その出力は、並列領域または構造を実行するスレッドによって実装される並列処理の Fortran 実行ファイルです。「OpenMP によるプログラミング」を参照してください。

パフォーマンス分析

プログラムのパフォーマンス分析には、パフォーマンス情報を表示するインテル® VTune™ アナライザや、インテルのスレッド化ツールを使用できます。コードのどの部分が最も多く実行時間を必要としているか、また並列パフォーマンスの問題個所の特定に関する詳細情報が得られます。

OpenMP* によるプログラミング

インテル® Fortran コンパイラは、OpenMP* ディレクティブを含む Fortran プログラムを入力ファイルとして処理し、マルチスレッド・バージョンのコードを生成します。並列プログラムが実行を開始する際に存在する単一スレッドは、マスタスレッドと呼ばれます。並列領域に到達するまで、マスタスレッドはシーケンシャルに処理を続行します。

並列領域

並列領域とは、1 つのスレッドチームによって並列に実行されなければならない 1 ブロックのコードです。OpenMP Fortran API では、並列構造は OpenMP ディレクティブ PARALLEL をコード・セグメントの最初に、そして END PARALLEL を最後に配置することによって定義されます。このように境界のあるコード・セグメントは、並列実行が可能です。

コードの構造ブロックとは、最初と最後に単一の入口/出口ポイントを持つ、1 つまたは複数の実行文の集まりです。

インテル Fortran コンパイラはワークシェアリング構造および同期化構造をサポートします。これらの各構造は、特定の 1 つまたは 2 つの OpenMP ディレクティブと、囲ま

れた、または後続するコードの構造ブロックから構成されます。構造の詳細な定義については、「OpenMP Fortran version 2.0 仕様」を参照してください。

並列領域の最後で、スレッドはすべてのチームメンバが到達するまで待機します。そして、チームは論理的になくなり（次の並列領域で再利用されることもあります）、マスタスレッドは次の並列領域が検出されるまでシーケンシャルに処理を続行します。

ワークシェアリング構造

ワークシェアリング構造は、それを囲む並列領域に入る時点で作成されたチームメンバ間で、囲まれたコード領域の実行を分割します。マスタスレッドが並列領域に入ると、スレッドチームが形成されます。並列領域の最初から開始して、ワークシェアリング構造が検出されるまで、コードは複製されます（すべてのチームメンバによって実行されます）。ワークシェアリング構造は、チームのメンバ間で、囲まれたコード領域の実行を分割します。

OpenMP の SECTIONS または DO 構造は、その囲まれた作業を現在のチームのスレッド間に分配するため、ワークシェアリング構造として定義されます。ワークシェアリング構造は、並列領域の動的実行中である場合にのみ分配されます。ワークシェアリング構造が並列領域の記述範囲内にあれば、チームのメンバ間で処理を分配することにより、そのワークシェアリング構造は常に実行されます。ワークシェアリング構造が並列領域に字句的（明示的）に囲まれていない場合（つまり、構造が孤立（orphan）している場合）、そのワークシェアリング構造は、最も近い動的範囲の並列領域のチームメンバ間に分配されます（これが存在する場合）。動的範囲の並列領域が存在しない場合、構造はシーケンシャルに実行されます。

1 つのスレッドがワークシェアリング構造の最後に到達すると、構造内のすべてのチームメンバが処理を終了するまで待機する場合があります。ワークシェアリング構造によって定義されたすべての処理が終了すると、チームは ワークシェアリング構造を出て、後に続くコードの実行を続行します。

並列/ワークシェアリング複合構造とは、ワークシェアリング構造を 1 つだけ含む並列領域を示します。

並列処理ディレクティブ・グループ

並列処理ディレクティブには次のグループが含まれます:

並列領域

- PARALLEL および END PARALLEL

ワークシェアリング構造

- DO および END DO ディレクティブは、ループの繰返しを並列実行するように指定します。
- SECTIONS および END SECTIONS ディレクティブは、任意のシーケンシャル・コードを並列実行するように指定します。各 SECTION はチーム内の 1 つのスレッドにより 1 回実行されます。

- SINGLE および END SINGLE ディレクティブは、1 つのスレッドによってのみ実行されるコードのセクションを定義します。このセクションを実行するように指定されていないスレッドはコードを無視します。

並列/ワークシェアリング複合構造

並列/ワークシェアリング複合構造は、単一のワークシェアリング構造が含まれる並列領域を指定するための短縮形を提供します。並列/ワークシェアリング複合構造は次のとおりです:

- PARALLEL DO および END PARALLEL DO
- PARALLEL SECTIONS および END PARALLEL SECTIONS

同期化構造および MASTER

同期化とは、共有データの一貫性を保証し、スレッド間の並列実行を調整するスレッド間通信です。データがアクセスされる際にすべてのスレッドが同じ値を取得する場合、スレッドのチームの共有データは一貫しています。同期化構造は、 共有データの一貫性を保証するために使用されます。

- OpenMP の同期化ディレクティブには、CRITICAL、ORDERED、ATOMIC、FLUSH および BARRIER があります。
    - 並列領域またはワークシェアリング構造内では、一度に 1 つのスレッドのみが CRITICAL 構造のコードを実行することができます。
    - ORDERED ディレクティブは、コードのセクションを順次実行するために、DO または SECTIONS 構造とともに使用されます。
    - ATOMIC ディレクティブは、他のスレッドに割り込まれずにメモリ位置を更新するために使用されます。
    - FLUSH ディレクティブは、チーム内のすべてのスレッドが一貫性のあるメモリのビューを持つために使用されます。
    - BARRIER ディレクティブは、コード内の特定の位置ですべてのチームのメンバが集合するために使用します。BARRIER を実行する各チームのメンバは、他のメンバが到達するまで BARRIER で待機します。ワークシェアリングまたは他の同期化構造内では、デッドロックの可能性があるため BARRIER を使用することはできません。

- MASTER ディレクティブは、マスタスレッドで実行するために使用します。

詳細は、「OpenMP ディレクティブと節」のリストを参照してください。

データの共有

データの共有は、SHARED および PRIVATE 節を使用して、並列領域または ワークシェアリング構造の最初で指定されます。SHARED 節に含まれているすべての変数は、チームメンバ間で共有されます。以下の項目は、アプリケーション側で行う必要があります:

- これらの変数へのアクセスを同期化します。PRIVATE 節に含まれているすべての変数は、各チームのメンバに対してプライベートです。並列領域全体では、例えば、`t` チームメンバでは、`PRIVATE` 節に含まれているすべての変数の `t+1` のコピーができます (アクティブなグローバル・コピーが並列領域外に 1 つ、各チームのメンバ用に PRIVATE のコピーが 1 つあります)。
- FIRSTPRIVATE 節が指定されていない限り、並列領域の開始時点で PRIVATE 変数を初期化します。この場合、FIRSTPRIVATE 節が指定されている構造の開始時点で、PRIVATE コピーはグローバル・コピーによって初期化されます。
- 並列領域の最後で PRIVATE 変数のグローバル・コピーを更新します。しかし、DO ディレクティブの LASTPRIVATE 節を使用することで、ループの最後の繰返しを順次実行したチームメンバからのグローバル・コピーを更新することができます。

また、`SHARED` および PRIVATE 変数に加えて、THREADPRIVATE ディレクティブを使用することで、個々の変数および 共通ブロック全体をプライベート化することができます。

孤立ディレクティブ

OpenMP には、並列ディレクティブの表現力を大幅に向上する孤立化と呼ばれる機能が含まれています。孤立化では、並列領域に付けられているディレクティブが 1 つのプログラム・ユニットの記述範囲内にある必要がありません。CRITICAL、BARRIER、SECTIONS、SINGLE、MASTER および DO などのディレクティブは、ランタイム時に、囲まれた並列領域に動的に “バインド” し、プログラム・ユニット内に出現することができます。

孤立ディレクティブは、コードに最小限の変更を行うだけで、既存のコードを並列処理することができます。また、孤立化は 1 つの並列領域を、呼び出されたサブルーチン内にある複数の `DO` ディレクティブとバインドすることでパフォーマンスの向上を可能にします。次のコード・セグメントを参照してください。

```
...
!$omp parallel
call phase1
call phase2
!$omp end parallel
...

subroutine phase1
!$omp do private(i) shared(n)
do i = 1, n
call some work(i)
end do
!$omp end do
end


subroutine phase2
!$omp do private(j) shared(n)
do j = 1, n
call more work(j)
end do
!$omp end do
end
```

次に示す孤立ディレクティブの使用規則が適用されます。

- 1 つの孤立した ワークシェアリング構造 (SECTIONS、SINGLE、DO) は、単一のスレッドからなる 1 つのチームによって実行されます。つまり、順次実行されます。
- ワークシェアリング構造内で実行された集合操作 (ワークシェアリング構造または BARRIER) は無効です。
- 集合操作 (ワークシェアリング構造または BARRIER) を同期化領域内 (CRITICAL/ORDERED) から実行することはできません。
- ディレクティブ・ペアである開始および終了ディレクティブ (例えば DO と END DO など) は、プログラムの 1 つのブロック内になければなりません。
- 変数のプライベート・スコーピングは、ワークシェアリング構造で指定することができます。共有されるスコーピングは、並列領域で指定する必要があります。詳細は、「OpenMP Fortran version 2.0 仕様」を参照してください。

OpenMP 処理を行うコードの準備

OpenMP を使用するためにコードを準備するには、次の段階と手順に従ってください。一般的に、最初の 2 つの段階は単一プロセッサ・システムまたはマルチプロセッサ・システムのいずれでも行うことができますが、その後の段階は通常、マルチプロセッサ・システムで行います。

OpenMP ディレクティブを挿入する前に

OpenMP 並列ディレクティブを挿入する前に、次の方法でコードが安全に並列実行されることを確認してください:

- ローカル変数をスタックに配置します。これは、-openmp が使用されると、インテル Fortran コンパイラはデフォルトで行います。
- -automatic または -auto_scalar を使用してローカル変数を自動 (automatic) にします。これは、-openmp が使用されると、インテル Fortran コンパイラはデフォルトで行います。ローカル変数のスタックへの割り当てを妨げる -save オプションを使用しないでください。デフォルト (-auto_scalar) では 、ローカル・スカラ変数はスレッド間で共有されるため、同期化コードを追加して、スレッドが正常にアクセスできるようにする必要があります。

解析

解析の主な手順は次のとおりです:

- プログラムのプロファイルを作成して、最も多くの時間が費やされている個所を見つけます。この個所は、並列処理の恩恵が最も得られるプログラムの部分です。この作業を行うには、インテル® VTune™ アナライザまたは基本的な PGO オプションを使用できます。
- プログラムがネストされたループを含む場合は、常に繰返し間の依存関係が少ない、最も外側のループを選択します。

再編成

OpenMP を正しく実装するためにプログラムを再編成するには、次のいくつか、またはすべてを実行します。

1. 選択したループが繰返しを並列に実行できる場合、このループに PARALLEL DO 構造を取り入れます。
2. アルゴリズムを書き直して、繰返し間の依存関係を取り除くようにします。
3. 依存関係にかかわる変数の使用と割り当ての周囲に CRITICAL 構造を配置し、残りの繰返し間の依存関係を同期化します。
4. ループに存在する変数を適切な SHARED、PRIVATE、LASTPRIVATE、FIRSTPRIVATE、または REDUCTION 節内にリストします。
5. 並列ループの DO インデックスを PRIVATE としてリストします。この手順はオプションです。
6. グローバル・スコープが保持される場合は、COMMON ブロックの要素を PRIVATE リストに配置してはいけません。THREADPRIVATE ディレクティブを使用して、それらの変数をグローバル・スコープに持つ COMMON ブ

ロックを、各スレッドに対してプライベート化します。THREADPRIVATE は、チーム内の各スレッド用に COMMON ブロックのコピーを作成します。

7. 並列領域のあらゆる I/O を同期化します。
8. より多くの並列ループを特定し、それらを再編成します。
9. 可能な場合は、隣接する PARALLEL DO 構造を、複数の DO ディレクティブが含まれる 1 つの並列領域にマージし、実行のオーバーヘッドを減らします。

チューニング

チューニング・プロセスでは、SCHEDULE 節または `omp_schedule` 環境変数を使用して、クリティカル・セクションのシーケンシャル・コードを最小化し、負荷のバランスをはかります。

注
通常、この手順はマルチプロセッサ・システムで実行されます。

並列処理スレッドモデル

ここでは、並列化されたプログラム処理の説明と、並列プログラミングで使用される用語の定義を追加説明します。

実行フロー

OpenMP の Fortran API コンパイラ・ディレクティブを持つプログラムは、単一のプロセスとして実行を開始します。これは、*マスタスレッド*の実行と呼ばれます。最初の*並列構造*が検出されるまで、マスタスレッドは、シーケンシャルに実行します。

OpenMP Fortran API では、PARALLEL および END PARALLEL ディレクティブは、並列構造を定義します。マスタスレッドが並列構造を検出すると、そのマスタスレッドがチームのマスタとなるように、スレッドの *チーム*を作成します。並列構造に囲まれたプログラム文は、チーム内の各スレッドごとに並列して実行されます。これらの文は、囲まれた文の中から呼び出されるルーチンを含みます。

構造内で字句的に囲まれた文は、構造の*静的範囲*を定義します。*動的範囲*は、構造内から呼び出されるルーチンと同様に静的範囲を含みます。END PARALLEL ディレクティブが検出されると、チーム内のスレッドはその時点で同期し、そのチームは消滅して、マスタスレッドのみが実行を続けます。チーム内の他のスレッドは、待機状態になります。

単一プログラム内で並列構造は何回でも指定できます。結果として、プログラム実行中に、スレッドのチームは何度も生成され消滅します。

孤立ディレクティブの使用

並列構造内で呼び出されたルーチンで、ディレクティブを使用することができます。並列構造の字句範囲ではなく、動的範囲のディレクティブは、孤立ディレクティブと呼ばれます。孤立ディレクティブは、プログラムのシーケンシャル・バージョンに最小限の変更を行うだけで、プログラムの主要部分を並列に実行できます。この機能を使用すると、プログラム・コール・ツリーの最上位レベルで並列構造をコーディングでき、ディレクティブを使用して呼び出されるすべてのルーチンの実行を制御することができます。次に例を示します。

```
subroutine F
...
!$OMP parallel...
...
call G
...
subroutine G
...
!$OMP DO...
...
```

!$OMP DO が実行される並列領域が G の記述範囲内にないため、!$OMP DO は孤立ディレクティブとなります。

データ環境ディレクティブ

データ環境ディレクティブは、並列構造の実行中にデータ環境を制御します。

並列およびワークシェアリング構造内でデータ環境を制御できます。ディレクティブとディレクティブのデータ環境節を使用して次のことが可能です:

- THREADPRIVATE ディレクティブを使用して、名前付き共通ブロックをプライベート化します。
- THREADPRIVATE ディレクティブの節を使用して、データスコープ属性を制御します。

  データスコープ属性節:

  - COPYIN
  - DEFAULT
  - PRIVATE
  - FIRSTPRIVATE

- LASTPRIVATE
- REDUCTION
- SHARED

複数のディレクティブ節を使用して、変数のデータスコープ属性を指定した構造の継続期間中にその属性を制御することができます。ディレクティブでデータスコープ属性節を指定しない場合、ディレクティブに影響を受ける変数のデフォルトは SHARED になります。

節の詳細は、「OpenMP Fortran バージョン 2.0 仕様」を参照してください。

並列処理モデルの疑似コード

一般的な OpenMP ディレクティブを使用したサンプル・プログラムを次に示します。この例ではまた、シリアル領域と並列領域の相違も示しています。

```
PROGRAM MAIN Begin Serial Execution
...                       !Only the master thread executes
!$OMP PARALLEL Begin a Parallel Construct, form a team
...                       !This is Replicated Code where each
team !
                          !member executes the same code
!$OMP SECTIONS!Begin a Worksharing Construct
!$OMP SECTION!One unit of work
...                       !
!$OMP SECTION!Another unit of work
...                       !
!$OMP END SECTIONS!Wait until both units of work complete
...                       !More Replicated Code
  !$OMP DO!Begin a Worksharing Construct,
      DO!each iteration is a unit of work
      ...                 !Work is distributed among the team
      END DO                    !
!$OMP END DO NOWAIT!End of Worksharing Construct, NOWAIT
                          ! is specified (threads need not
wait
                          ! until all work is completed before
                          ! proceeding)
...                       !More Replicated Code
!$OMP END PARALLEL!End of PARALLEL Construct, disband
team !
                          !and continue with serial execution
...                       !Possibly more PARALLEL Constructs
END PROGRAM MAIN!End serial execution
```

OpenMP*、ディレクティブ形式、および診断でのコンパイル

OpenMP* モードでインテル® Fortran コンパイラを起動するには、-openmp オプションでインテル・コンパイラを起動する必要があります。

ifort -openmp *input_file(s)*

マルチスレッド・コードを起動する前に、OpenMP 環境変数 OMP_NUM_THREADS で使用するスレッドの数を設定できます。詳細は、「OpenMP の環境変数」のセクションを参照してください。「インテル拡張ルーチン」では、インテル Fortran コンパイラの仕様に追加された OpenMP 拡張機能について説明します。

-openmp オプション

-openmp オプションは、パラレライザが OpenMP ディレクティブに基づいてマルチスレッド・コードを生成できるようにします。このコードは、単一プロセッサ・システムとマルチプロセッサ・システムのいずれでも並列実行が可能です。

-openmp オプションは、-O0 (最適化なし) と -O1、-O2 (デフォルト) および -O3 の最適化レベルで動作します。-openmp と -O0 を指定すると、OpenMP アプリケーションのデバッグに役立ちます。

-openmp オプションを使用すると、コマンドラインで他のオプションを指定しない限り、-auto オプションが設定されます (すべての変数を、ローカル静的記憶域ではなく、スタックに割り当てます)。

OpenMP ディレクティブ形式と構文

OpenMP ディレクティブの形式は次のとおりです:

```
<prefix> <directive> [<clause> [[,] <clause> ...]]
```

上記のカッコの意味は次のとおりです:

- <xxx>: プリフィックスとディレクティブが必要です。
- [<xxx>]: ディレクティブ が 1 つ以上の節を使用する場合、節が必要です。
- [,]: <clause> 間のカンマはオプションです。

固定形式のソース入力の場合、プリフィックスは !$omp または c$omp です。

自由形式のソース入力の場合、プリフィックスは !$omp のみです。

プリフィックスの後には、ディレクティブがきます。次に例を示します:

```
!$omp parallel
```

OpenMP ディレクティブは感嘆符で始まるため、-openmp オプションを省くと、ディレクティブはコメント形式になります。

ソースコードの並列領域の構文

並列領域を定義している OpenMP 構造の構文形式は、次のいずれかです。

```
!$omp <directive>
<structured block of code>
!$omp end <directive>
```

または

```
!$omp <directive>
<structured block of code>
```

または

```
!$omp <directive>
```

<directive> は、特定の OpenMP ディレクティブの名前です。

OpenMP 診断レポート

-openmp_report{0|1|2} オプションは、OpenMP パラレライザの診断レベル 0、1、2 を次のように制御します:

-openmp_report0 = 診断情報を表示しません。

-openmp_report1 = 正常に並列化されたループ、領域、およびセクションを示す診断を表示します。

-openmp_report2 = -openmp_report1 の診断に加えて、正常に処理された MASTER 構造、SINGLE 構造、CRITICAL 構造、ORDERED 構造、ATOMIC ディレクティブなどを示す診断を表示します。

デフォルトは -openmp_report1 です。

OpenMP* ディレクティブと節の概要

このトピックでは、OpenMP* ディレクティブと節の概要を説明します。詳細は、「OpenMP Fortran バージョン 2.0 仕様」を参照してください。

OpenMP ディレクティブ

ディレクティブ	説明
PARALLEL END PARALLEL	並列実行領域を定義します。
DO END DO	関連付けられたループの反復が並列実行される領域を指定する、反復的なワークシェアリングの構造を識別します。
SECTIONS END SECTIONS	チーム内のスレッド間で分割される一連の構造ブロックを指定する、非反復的なワークシェアリングの構造を識別します。
SECTION	囲まれた SECTION 構造の部分として、関連する構造ブロックを並列実行するように指定します。
SINGLE END SINGLE	対応する構造化ブロックがチーム内の 1 つのスレッドだけで実行されるように指定する構造を識別します。
PARALLEL DO END PARALLEL DO	1 つの DO ディレクティブを含む並列領域のショートカット。 **注** OpenMP ディレクティブ PARALLEL DO または DO の直後には、DO 文 (ANSI Fortran 規格の R818 に定義された `do-stmt`) を続けなければなりません。PARALLEL DO または DO ディレクティブと DO 文の間に、他の文または OpenMP ディレクティブがあると、インテル® Fortran コンパイラは構文エラーを生成します。
PARALLEL SECTIONS END PARALLEL SECTIONS	1 つの SECTIONS 構造を含む並列領域を指定するショートカット形式。
MASTER END MASTER	チームの MASTER スレッドのみで実行する構造ブロックを指定する構成体を示します。
CRITICAL[*lock*] END CRITICAL[*lock*]	関連する構造ブロックの実行を一度に 1 スレッドだけに制限する構成体を示します。関連する構造ブロックの実行を一度に 1 スレッドだけに制限する構造を示します。各スレッドは、他のスレッドが同じ *lock* 引数でクリティカル構

	造を実行しなくなるまで、クリティカル構造の最初で待機します。
BARRIER	チーム内のすべてのスレッドを同期化します。各スレッドは、チーム内の他のすべてのスレッドがこのポイントに到達するまで待機します。
ATOMIC	特定のメモリ・ロケーションをアトミックに更新し、同時に複数のスレッドによる書き込みの危険性を回避するようにします。
FLUSH [(*list*)]	"クロススレッド" シーケンス・ポイントを指定します。このポイントでは、チーム内のすべてのスレッドから見たメモリ内の特定のオブジェクトの状態の整合性が保たれるように、プログラム上で保証する必要があります。オプションの *list* 引数は、フラッシュする変数をカンマ区切りでリストします。
ORDERED END ORDERED	ORDERED ディレクティブに続く構造ブロックを、シーケンシャル・ループ内で反復が実行される順序で実行します。
THREADPRIVATE (*list*)	名前付きの COMMON ブロックまたは変数をスレッドに対してプライベートにします。*list* 引数は、COMMON ブロックまたは変数をカンマ区切りでリストします。

OpenMP の節

節	説明
PRIVATE (*list*)	*list* の変数がチーム内の各スレッドに対して PRIVATE になるように宣言します。
FIRSTPRIVATE (*list*)	PRIVATE と同じですが、*list* の各変数のコピーは、並列処理構造の前に存在するオリジナルの変数の値を使用して初期化されます。
LASTPRIVATE (*list*)	PRIVATE と同じですが、*list* のオリジナルの変数は、DO 構造ルー プまたは最後の SECTION 構造における最後の繰返しで、対応する PRIVATE 変数に割り当てられた値に更新されます。
COPYPRIVATE (*list*)	単一構造の最後で、チーム内メンバから他のメンバに値をブロードキャストするために、*list* の PRIVATE 変数を使用するか、または共有オブジェク

	ト へのポインタを使用します。
NOWAIT	実行終了までワークシェアリング構造の最後でスレッドを待機させる必要がないことを指示します。スレッドは、実行する作業がなくなるとすぐに、ワークシェアリング構造の最後から先に進むことができます。
SHARED (*list*)	チーム内のすべてのスレッド間で *list* 内の変数を共有します。
DEFAULT (*mode*)	他の節により明示的に指定されていない変数のデフォルト・データスコープ属性を決定します。*mode* は、PRIVATE、SHARED、または NONE のいずれかです。
REDUCTION ({*operator*|*intrinsic*}:*list*)	演算子 operator または組込み関数プロシージャ名 intrinsic を使用して、*list* にある変数のリダクションを実行します。operator は、+、*、.and.、.or.、.eqv.、.neqv. のいずれかです。intrinsic は、MAX、MIN、IAND、IOR または IEOR のいずれかです。
ORDERED END ORDERED	DO または SECTIONS 構造と共に使用され、コード部分を順次実行します。ORDERED 構造が DO 構造の動的範囲に含まれる場合、ORDERED 節は DO ディレクティブになければなりません。
IF (*scalar_logical_expression*)	囲まれた並列領域は、*scalar_logical_expression* が .TRUE. と評価された場合にのみ並列で実行されます。そうでない場合、並列領域は順次実行されます。
NUM_THREADS (*scalar_integer_expression*)	並列領域の *scalar_integer_expression* により指定されるスレッド数を要求します。
SCHEDULE (*type*[,*chunk*])	DO 構造の繰返しをチームのスレッド間でどのように分割するかを指定します。*type* 引数の値は、STATIC、

	DYNAMIC、GUIDED、または RUNTIME です。オプションの *chunk* 引数は、正のスカラ整数式でなければなりません。
COPYIN (*list*)	並列領域の最初で、マスタスレッドのデータ値を、THREADPRIVATE 共通ブロックのコピー、または *list* で指定された変数に複写するように指定します。

ディレクティブと節の対応表

ディレクティブ	使用する節
PARALLEL END PARALLEL	COPYIN、DEFAULT、PRIVATE、FIRSTPRIVATE、REDUCTION、SHARED
DO END DO	PRIVATE、FIRSTPRIVATE、LASTPRIVATE、REDUCTION、SCHEDULE
SECTIONS END SECTIONS	PRIVATE、FIRSTPRIVATE、LASTPRIVATE、REDUCTION
SECTION	PRIVATE、FIRSTPRIVATE、LASTPRIVATE、REDUCTION
SINGLE END SINGLE	PRIVATE、FIRSTPRIVATE
PARALLEL DO END PARALLEL DO	COPYIN、DEFAULT、 PRIVATE、FIRSTPRIVATE、LASTPRIVATE、REDUCTION、SHARED、SCHEDULE
PARALLEL SECTIONS END PARALLEL SECTIONS	COPYIN、DEFAULT、 PRIVATE、FIRSTPRIVATE、LASTPRIVATE、REDUCTION、SHARED
MASTER END MASTER	なし
CRITICAL[*lock*] END CRITICAL[*lock*]	なし
BARRIER	なし
ATOMIC	なし
FLUSH [(*list*)]	なし
ORDERED END ORDERED	なし
THREADPRIVATE (*list*)	なし

OpenMP ディレクティブの説明

並列領域ディレクティブ

PARALLEL ディレクティブおよび END PARALLEL ディレクティブは、次のように並列領域を定義します:

```
!$OMP  PARALLEL
!parallel region
!$OMP  END  PARALLEL
```

スレッドが並列領域を検出すると、スレッドのチームを形成して、そのスレッドがチームのマスタになります。チーム内のスレッド数は環境変数またはランタイム・ライブラリ・コール、あるいはその両方により制御することができます。

PARALLEL ディレクティブには、オプションで次のように指定した節をカンマ区切りのリストで指定できます。節には次のものが含まれます。

- IF: 並列領域の文をスレッドのチームによって並列に実行するか、単一のスレッドによって順次に実行するかどうか
- PRIVATE、FIRSTPRIVATE、SHARED、REDUCTION: 変数型
- DEFAULT: 変数データスコープ属性
- COPYIN: マスタスレッドの共通ブロックの値は、THREADPRIVATE 共通ブロックのコピーに格納されます。

スレッド数の変更

一旦チームを形成すると、チーム内のスレッド数は並列領域では一定です。次の並列領域で使用されるスレッド数を明示的に変更するには、OMP_SET_NUM_THREADS ランタイム・ライブラリ・ルーチンをプログラムのシリアルな部分から呼び出します。このルーチンは、OMP_NUM_THREADS 環境変数で設定した値より優先されます。

例えば、OMP_NUM_THREADS 環境変数を使用してスレッド数を 6 に設定した場合、並列領域間のスレッド数は次のように変更できます。

```
    CALL  OMP_SET_NUM_THREADS(3)
!$OMP  PARALLEL
.
.
.
!$OMP  END  PARALLEL
CALL  OMP_SET_NUM_THREADS(4)
```

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

作業単位の設定

DO、SECTIONS、および SINGLE のようなワークシェアリング・ディレクティブを使用して、並列領域の文を作業単位に分割し、各単位が 1 つのスレッドにより実行されるように分配できます。

次の例では、!$OMP DO および !$OMP END DO ディレクティブと、これらのディレクティブに囲まれた文はすべて、並列領域の静的範囲を構成します。

```
!$OMP PARALLEL
!$OMP DO
DO I=1,N
B(I) = (A(I) + A(I-1))/ 2.0
END DO
!$OMP END DO
!$OMP END PARALLEL
```

次の例では、!$OMP DO および !$OMP END DO ディレクティブと、これらのディレクティブで囲まれた文（WORK サブルーチンに含まれるすべての文を含む）はすべて、並列領域の動的範囲を構成します。

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
DO I=1,N
CALL WORK(I,N)
END DO
!$OMP END DO
!$OMP END PARALLEL
```

条件付き並列領域実行の設定

IF 節が PARALLEL ディレクティブに指定されていると、囲まれたコード領域は、スカラ論理式が .TRUE. に評価される場合にのみ並列で実行されます。それ以外の場合は、並列領域は順次実行されます。IF 節がない場合は、その領域はデフォルトで並列に実行されます。

次の例では、!$OMP DO および !$OMP END DO ディレクティブ内で囲まれた文は、4 つ以上のプロセッサが利用できる場合にのみ、並列で実行されます。それ以外の場合は、文は順次実行されます。

```
!$OMP PARALLEL IF (OMP_GET_NUM_PROCS() .GT.3)
!$OMP DO
DO I=1,N
Y(I) = SQRT(Z(I))
END DO
!$OMP END DO
!$OMP END PARALLEL
```

並列領域を実行中のスレッドが別の並列領域を検出すると、新しいチームを形成してそのチームのマスタになります。デフォルトでは、ネストされた並列領域は常に 1 つのスレッドで構成されるチームによって実行されます。


注

順次実行よりもパフォーマンスを向上させるには、並列領域に 1 つまたは複数のワークシェアリング構造を含めて、スレッドチームが並列で作業を実行できるようにする必要があります。ワークシェアリング構造を並列領域に含めることで、並列処理によるパフォーマンスの向上を得ることができます。

ワークシェアリング構造ディレクティブ

ワークシェアリング・ディレクティブを並列で実行する場合、並列領域内でワークシェアリング構造を動的に囲む必要があります。新規のスレッドは起動されません。また、ワークシェアリング構造に入る時点で暗黙的なバリアはありません。

ワークシェアリング構造は、次のとおりです。

- DO および END DO ディレクティブ
- SECTIONS、SECTION、および END SECTIONS ディレクティブ
- SINGLE および END SINGLE ディレクティブ

DO および END DO

DO ディレクティブは、DO ループ直後の繰返しがスレッドのチームに振り分けられるようにし、それぞれの繰返しが 1 つのスレッドにより実行されるようにします。DO ディレクティブに続くループには、ループ制御が実行されない DO WHILE または DO ループは使用できません。DO ループの繰返しは、既存のスレッドチーム間で振り分けられます。

DO ディレクティブは、オプションで次のことが可能です:

- データスコープ属性の制御 (「データスコープの制御」を参照してください。)
- SCHEDULE 節を使用して、スケジュールの型とチャンクサイズを指定 (「スケジュールの型とチャンクサイズの設定」を参照してください)。

使用される節

DO ディレクティブの節では、次を指定します:

- 変数が PRIVATE、FIRSTPRIVATE、LASTPRIVATE、または REDUCTION かどうか。
- ループの繰返しがスレッドにスケジュールされる方法。
- また、ORDERED ディレクティブが DO ディレクティブの動的範囲にある場合は ORDERED 節を指定する必要があります。
- オプションの NOWAIT 節を END DO ディレクティブで指定しない場合は、スレッドは END DO ディレクティブで同期します。NOWAIT を指定する場合、スレッドは同期化せず、先に完了したスレッドは、END DO ディレクティブの次の命令に直接進みます。

使用規則

- GOTO 文または他の文を使用して、DO 構造の内部または外部に制御を移すことはできません。
- オプションの END DO ディレクティブを指定する場合、DO ループの最後のすぐ後に指定しなければなりません。END DO ディレクティブを指定しない場合は、END DO ディレクティブは DO ループの最後にあると仮定され、スレッドはその時点で同期します。
- ループの繰返し変数は、デフォルトではプライベートであるため、明示的に宣言する必要はありません。

SECTIONS、SECTION および END SECTIONS

非反復的なワークシェアリング SECTIONS ディレクティブを使用して、囲まれたコードのセクションをチーム内で分割します。各セクションは、1 つのスレッドによって 1 回だけ実行されます。

SECTION ディレクティブがオプションである最初のセクションを除き、各セクションは、SECTION ディレクティブから開始されます。 SECTION ディレクティブは、SECTIONS ディレクティブと END SECTIONS ディレクティブの記述範囲内になければなりません。

最後のセクションは、END SECTIONS ディレクティブで終わります。NOWAIT を指定しない限り、スレッドがセクションを終了し、振り分けられていないセクションがなければ、スレッドは END SECTION ディレクティブで待機します。

SECTIONS ディレクティブには、変数に PRIVATE、FIRSTPRIVATE、LASTPRIVATE、または REDUCTION を指定した節をカンマ区切りのリストで指定できます。

次の例では、SECTIONS ディレクティブと SECTION ディレクティブを使用して、X_AXIS、Y_AXIS、Z_AXIS サブルーチンを並列に実行する方法を示します。最初の SECTION ディレクティブはオプションです。

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
CALL X_AXIS
!$OMP SECTION
CALL Y_AXIS
!$OMP SECTION
CALL Z_AXIS
!$OMP END SECTIONS
!$OMP END PARALLEL
```

SINGLE および END SINGLE

囲まれたコードのブロックを、チームの 1 つのスレッドにのみ実行させる場合は、SINGLE ディレクティブを使用します。

SINGLE ディレクティブを実行していないスレッドは、NOWAIT を指定しない限り、END SINGLE ディレクティブで待機します。

SINGLE ディレクティブには、変数に PRIVATE または FIRSTPRIVATE を指定した節をカンマ区切りのリストで指定できます。

END SINGLE ディレクティブが検出されると、暗黙的なバリアができ、スレッドはすべてのスレッドが完了するまで待機します。これは、NOWAIT オプションを使用して無効にできます。

次の例では、SINGLE ディレクティブを最初に検出したスレッドが、OUTPUT と INPUT サブルーチンを実行します。

```
!$OMP PARALLEL DEFAULT(SHARED)
CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
CALL OUTPUT(X)
CALL INPUT(Y)
!$OMP END SINGLE
CALL WORK(Y)
!$OMP END PARALLEL
```

並列/ワークシェアリング複合構造

並列/ワークシェアリング複合構造は、単一のワークシェアリング構造が含まれる並列領域を指定するための短縮形を提供します。並列/ワークシェアリング複合構造は次のとおりです:

- PARALLEL DO
- PARALLEL SECTIONS

PARALLEL DO および END PARALLEL DO

PARALLEL DO ディレクティブを使用して、暗黙的に単一の DO ディレクティブを含む並列領域を指定します。

PARALLEL ディレクティブと DO ディレクティブに 1 つ以上の節を指定できます。

次の例では、単純なループを並列化する方法を示します。ループの繰返し変数は、デフォルトではプライベートであるため、明示的に宣言する必要はありません。END PARALLEL DO ディレクティブはオプションです。

```
!$OMP PARALLEL DO
 DO I=1,N
   B(I) = (A(I) + A(I-1)) / 2.0
 END DO
!$OMP END PARALLEL DO
```

PARALLEL SECTIONS および END PARALLEL SECTIONS

PARALLEL SECTIONS ディレクティブを使用して、暗黙的に単一の SECTIONS ディレクティブを含む並列領域を指定します。

PARALLEL ディレクティブと SELECTION ディレクティブに 1 つ以上の節を指定できます。

最後のセクションは、END PARALLEL SECTIONS ディレクティブで終わります。

以下の例では、サブルーチン `X_AXIS`、`Y_AXIS`、および `Z_AXIS` が同時に実行できます。最初の SECTION ディレクティブはオプションです。すべての SECTION ディレクティブは、PARALLEL SECTIONS/END PARALLEL SECTIONS 構造の記述範囲内になければなりません:

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
 CALL X AXIS
!$OMP SECTION
 CALL Y AXIS
!$OMP SECTION
 CALL Z AXIS
!$OMP END PARALLEL SECTIONS
```

同期化構造

同期化構造とは、共有データの一貫性を保証し、スレッド間の並列実行を調整するのに使用されます。

同期化構造のディレクティブは次のとおりです:

- ATOMIC ディレクティブ
- BARRIER ディレクティブ
- CRITICAL ディレクティブ
- FLUSH ディレクティブ
- MASTER ディレクティブ
- ORDERED ディレクティブ

ATOMIC ディレクティブ

ATOMIC ディレクティブを使用して、特定のメモリ・ロケーションをアトミックに更新し、同時に複数のスレッドによる書き込みの危険性を回避するようにします。

このディレクティブは、直後に続く文にのみ適用され、次の形式のいずれかでなければなりません:

```
x = x operator expr

x = expr operator x

x = intrinsic (x, expr)

x = intrinsic (expr, x)
```

上記の文について、次に説明します。

- *x* は、組込み型のスカラ変数です。
- *expr* は、x を参照しないスカラ式です。
- *intrinsic* は、MAX、MIN、IAND、IOR、または IEOR のいずれかです。

- *operator* は、+、*、-、/、.AND.、.OR.、.EQV. または .NEQV. のいずれかです。

このディレクティブは、代入に関して、クリティカル・セクションで行われる最適化以上の最適化を可能にします。クリティカル・セクションで文を囲むことによって、すべてのATOMIC ディレクティブを置換することができます。これらすべてのクリティカル・セクションは、同一で一意な名前を使用する必要があります。

x のロードとストアのみがアトミックです。*expr* の評価は、アトミックではありません。競合状態を避けるには、競合状態ではないことがわかっているロケーションを除いて、そのロケーションのすべての並列の更新において、ATOMIC ディレクティブを使用して保護する必要があります。*intrinsic* 関数、*operator* 演算子、および代入は、組込みの関数、演算子および代入でなくてはなりません。

次の制限は、ATOMIC ディレクティブに適用されます: ストレージ・ロケーション *x* へのすべての参照は、同じ型のパラメータでなければなりません。

次の例では、Y のロケーションの集合は、アトミックに更新されます。

```
!$OMP ATOMIC
Y = Y + B(I)
```

BARRIER ディレクティブ

並列領域内ですべてのスレッドを同期化するには、BARRIER ディレクティブを使用します。このディレクティブは、PARALLEL ディレクティブにより定義される並列領域内でのみ使用できます。BARRIER ディレクティブは、DO、PARALLEL DO、SECTIONS、PARALLEL SECTIONS、SINGLE ディレクティブ内では使用できません。

各スレッドは、すべてのスレッドがディレクティブに到達するまで BARRIER ディレクティブで待機します。

次の例では、BARRIER ディレクティブは、すべてのスレッドが最初のループを実行し、2 番目のループの実行が安全に行われることを保証します。

```
c$OMP PARALLEL
c$OMP DO PRIVATE(i)
DO i = 1, 100
b(i) = i
END DO
c$OMP BARRIER
c$OMP DO PRIVATE(i)
DO i = 1, 100
a(i) = b(101-i)
```

```
END DO
c$OMP END PARALLEL
```

CRITICAL および END CRITICAL

CRITICAL および END CRITICAL ディレクティブを使用して、クリティカル・セクションとして参照されるコードブロックへのアクセスを、一度に 1 つのスレッドに制限します。

スレッドは、同じ名前を持つクリティカル・セクションを実行するチーム内の他のスレッドがなくなるまで、クリティカル・セクションの最初で待機します。

スレッドが、クリティカル・セクションに入ると、ラッチ変数がクローズに設定され、他のスレッドは、ロックアウトされます。そのスレッドが、END CRITICAL ディレクティブでクリティカル・セクションから出ると、ラッチ変数は、オープンに設定され、別のスレッドがクリティカル・セクションにアクセスできるようになります。

クリティカル・セクションの名前を CRITICAL ディレクティブで指定した場合、END CRITICAL ディレクティブでも同じ名前を指定する必要があります。CRITICAL ディレクティブの名前を指定しない場合、END CRITICAL ディレクティブの名前を指定することはできません。

名前を指定されていないすべての CRITICAL ディレクティブは、同じ名前にマップします。クリティカル・セクションの名前は、プログラムに対してグローバルです。

次の例では、複数の CRITICAL ディレクティブが含まれ、1 つのタスクがキューから取り出されて処理されるキューイングのモデルを示しています。複数のスレッドが同じタスクをキューから取り出さないために、デキューの操作はクリティカル・セクションになければなりません。この例では、2 つの独立したキューがあるため、各キューは、それぞれ X_AXIS と Y_AXIS という異なる名前を持つ CRITICAL ディレクティブによって保護されます。

```
!$OMP PARALLEL DEFAULT(PRIVATE,SHARED(X,Y)
!$OMP CRITICAL(X_AXIS)
CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(X_AXIS)
CALL WORK(IX_NEXT, X)
!$OMP CRITICAL(Y_AXIS)
CALL DEQUEUE(IY_NEXT,Y)
!$OMP END CRITICAL(Y_AXIS)
CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

名前を指定されていないクリティカル・セクションは、Pthread パッケージからグローバル・ロックを使用します。これにより、同じロックを使用して他のコードと同期することが

できます。名前の付いたロックは、コンパイラにより作成され保守されて、さらに効率的です。

FLUSH ディレクティブ

FLUSH ディレクティブを使用して、メモリの一貫性のあるビューが提供される同期ポイントを識別します。スレッドから見える変数は、このポイントでメモリに書き戻されます。

このポイントでスレッドから見えるすべての変数のフラッシュを避けるには、フラッシュする名前付きの変数をカンマ区切りのリストに含めます。

次の例では、FLUSH ディレクティブを使用して、変数 ISYNC のスレッド 0 とスレッド 1 の Point To Point (ポイント間) の同期化を示します。

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(ISYNC)
IAM = OMP GET THREAD NUM()
ISYNC(IAM) = 0
!$OMP BARRIER
CALL WORK()
!I Am Done With My Work, Synchronize With My Neighbor
ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
!Wait Till Neighbor Is Done
DO WHILE (ISYNC(NEIGH) .EQ.0)
!$OMP FLUSH(ISYNC)
END DO
!$OMP END PARALLEL
```

MASTER および END MASTER

MASTER および END MASTER ディレクティブを使用して、マスタスレッドのみが実行するコードブロックを識別します。

チームの他のスレッドは、コードをスキップして実行を続けます。END MASTER ディレクティブには、暗黙的なバリアはありません。

次の例では、マスタスレッドのみが OUTPUT と INPUT ルーチンを実行します。

```
!$OMP PARALLEL DEFAULT(SHARED)
CALL WORK(X)
!$OMP MASTER
CALL OUTPUT(X)
CALL INPUT(Y)
!$OMP END MASTER
CALL WORK(Y)
!$OMP END PARALLEL
```

ORDERED および END ORDERED

ORDERED および END ORDERED ディレクティブを DO 構造内で使用すると、オーダード（ordered）セクション内では処理をシーケンシャルに実行し、セクション外では、処理を並列に実行することができます。

ORDERED ディレクティブを使用する場合は、DO ディレクティブでも ORDERED 節を指定する必要があります。

オーダード・セクションには、一度に 1 つのスレッドのみがループの繰返し順序で入ることができます。

次の例では、インデックスをシーケンシャルにプリントアウトします。

```
!$OMP DO ORDERED,SCHEDULE(DYNAMIC)
DO I=LB,UB,ST
CALL WORK(I)
END DO
SUBROUTINE WORK(K)
!$OMP ORDERED
WRITE(*,*) K
!$OMP END ORDERED
```

THREADPRIVATE ディレクティブ

THREADPRIVATE ディレクティブを使用して、名前付きの共通ブロックをスレッドに対してプライベートに、スレッド内ではグローバルにすることができます。

各スレッドは、共通ブロックのコピーを取得し、1 つのスレッドにより書き込まれるデータが、他のスレッドから直接見えないようにします。プログラムのシリアルな部分と MASTER セクションでは、アクセスは共通ブロックのマスタスレッドのコピーに対して行われます。

スレッド・プライベートの共通ブロックやその構成変数は、COPYIN 節以外の節では使用できません。

次の例では、共通ブロックの `BLK1` および `FIELDS` がスレッド・プライベートとして指定されています。

```
COMMON /BLK1/ SCRATCH
 COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/,/FIELDS/)
```

OpenMP 節の説明

データスコープ属性節の概要

複数のディレクティブ節を使用して、変数のデータスコープ属性を指定した構造の継続期間中にその属性を制御することができます。ディレクティブでデータスコープ属性節を指定しない場合、ディレクティブに影響を受ける変数のデフォルトは SHARED になります。

各データスコープ属性節には、スコーピング・ユニットでアクセスできる名前付き変数、または名前付き共通ブロックのリストをカンマ区切りで指定できます。名前付き共通ブロックを指定する際は、間にスラッシュを指定する必要があります (*/name/*)。

すべての節が全ディレクティブで使用できるわけではありません。各節に適用されるディレクティブは、その節の説明に記述されます。

データスコープ属性節:

- COPYIN
- DEFAULT
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- SHARED

COPYIN 節

PARALLEL、PARALLEL DO および PARALLEL SECTIONS ディレクティブで COPYIN 節を使用して、マスタスレッドの共通ブロックにあるデータを共通ブロックのスレッド専用コピーに複写します。これは、並列領域の最初に実行されます。COPYIN 節は、THREADPRIVATE が宣言された共通ブロックにのみ適用されます。

コピーする共通ブロック全体を指定する必要はなく、THREADPRIVATE 共通ブロックにある名前付き変数を指定することができます。下記の例では、共通ブロック `BLK1` と `FIELDS` をスレッド専用として指定しますが、コピーに指定するのは、共通ブロック `FIELDS` の変数の 1 つだけです。

```
  COMMON /BLK1/ SCRATCH
 COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
```

```
!$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(/BLK1/,ZFIELD)
```

DEFAULT 節

PARALLEL、PARALLEL DO および PARALLEL SECTIONS ディレクティブで DEFAULT 節を使用して、並列領域の記述範囲内の全変数にデフォルトのデータスコープ属性を指定します。THREADPRIVATE 共通ブロックの変数は、この節には影響を受けません。1 つのディレクティブでは、DEFAULT 節は 1 つのみ指定できます。デフォルトのデータスコープ属性は、次のいずれかです:

- PRIVATE

  並列領域の記述範囲内におけるすべての名前付きオブジェクトを、1 つのスレッドに対してプライベートにします。オブジェクトには、共通ブロック変数が含まれますが、THREADPRIVATE 変数は含まれません。

- SHARED

  並列領域の記述範囲内におけるすべての名前付きオブジェクトを、チームの全スレッド間で共有にします。

- NONE

  変数が PRIVATE または SHARED であるかという暗黙的なデフォルトがないことを宣言します。並列領域の記述範囲内における各変数に対してスコープ属性を明示的に指定しなければなりません。

DEFAULT 節を指定しない場合、デフォルトでは DEFAULT(SHARED) になります。ただし、ループ制御変数は、デフォルトでは常に PRIVATE です。

次の例のように、並列領域上で他のスコープ属性を使用することにより、デフォルトのデータスコープ属性の変数を省略することができます:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE),
FIRSTPRIVATE(I),SHARED(X),
!$OMP& SHARED® LASTPRIVATE(I)
```

PRIVATE、FIRSTPRIVATE、および LASTPRIVATE 節

PRIVATE

PRIVATE 節は、PARALLEL、DO、SECTIONS、SINGLE、PARALLEL DO、および PARALLEL SECTIONS ディレクティブで使用し、変数をチーム内の各スレッドに対してプライベートとして宣言します。

変数が PRIVATE として宣言されると、次の処理が行われます:

- 同じ種類とサイズの新しいオブジェクトがチーム内の各スレッドに対して一度宣言されます。そのオブジェクトはオリジナルのオブジェクトと関連するストレージではありません。
- ディレクティブ構造の記述範囲内にあるオリジナルのオブジェクトへの参照はすべて、プライベート・オブジェクトへの参照に置き換えられます。
- PRIVATE として定義された変数は、構造の入口では各スレッドに対して未定義であり、対応する共有変数は、並列構造の出口で未定義です。
- PRIVATE として定義された変数の内容、割り当て状態、関連付け状態は、呼び出し先のルーチンに実引数として渡されている場合を除き、構造の記述範囲外 (ただし、動的範囲内) での参照時には未定義です。

次の例では、I および J の値は並列領域の出口では未定義です:

```
INTEGER I,J

 I =1

 J =2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)

 I =3

 J =J+ 2
!$OMP END PARALLEL

 PRINT *, I, J
```

FIRSTPRIVATE

FIRSTPRIVATE 節は、PARALLEL、DO、SECTIONS、SINGLE、PARALLEL DO、PARALLEL SECTIONS ディレクティブで使用し、PRIVATE 節機能のスーパーセットを提供します。

PRIVATE 節機能に加えて、変数のプライベート・コピーは、並列構造の前にあるオリジナルのオブジェクトにより初期化されます。

LASTPRIVATE

LASTPRIVATE 節は、DO、SECTIONS、PARALLEL DO、PARALLEL SECTIONS ディレクティブで使用し、PRIVATE 節機能のスーパーセットを提供します。

LASTPRIVATE 節が、DO または PARALLEL DO ディレクティブに置かれている場合、最後の繰返しをシーケンシャルに実行するスレッドは、その構造に入る前に持っていたオブジェクトのバージョンを更新します。

LASTPRIVATE 節が、SECTIONS または PARALLEL SECTIONS ディレクティブに置かれている場合、記述上において最後のセクションを実行するスレッドは、その構造に入る前に持っていたオブジェクトのバージョンを更新します。

DO ループの最後の繰返し、または記述上において最後の SECTION ディレクティブにより値を割り当てられていないサブオブジェクトは、その構造の後で未定義となります。

正常に実行されるかどうかは、ループの最後の繰返しにより割り当てられる変数の値に依存する場合があります。そのような変数は、すべて引数として LASTPRIVATE 節にリストし、変数の値が、ループがシーケンシャルに実行された場合と同じになるようにします。次の例では、並列領域の最後で、I の値は、N+1 と等しく、シーケンシャルに実行された場合と同じになります。

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
DO I=1,N
  A(I) = B(I) + C(I)
END DO
!$OMP END PARALLEL
CALL REVERSE(I)
```

REDUCTION 節

REDUCTION 節は、PARALLEL、DO、SECTIONS、PARALLEL DO、PARALLEL SECTIONS ディレクティブで使用し、次に示すように演算子または組込み関数により指定された変数に対してリダクションを実行します。

```
REDUCTION (
```

演算子

または

組込み関数

:list)

*演算子*は、次のいずれかを使用できます: +、*、-、.AND.、.OR.、.EQV.、または .NEQV.。

*組込み関数*は、次のいずれかを使用できます: MAX、MIN、IAND、IOR、または IEOR。

指定される変数は、組込み関数タイプの名前付きスカラ変数で、囲まれたコンテキスト内では SHARED でなければなりません。指定された各変数のプライベート・コピーは、PRIVATE 節を使用したかのように各スレッドに対して作成されます。プライベート・コピーは、次の表で示すように、演算子または組込み関数によって値が初期化されます。実際の初期値は、リダクション変数のデータ型と合致します。

演算子/組込み関数およびリダクション変数の初期値

演算子/組込み関数	初期値
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	表現可能な最大値
MIN	表現可能な最小値
IAND	全ビットがオン
IOR	0
IEOR	0

リダクションが適用される構造の最後で共有変数が更新され、指定された演算子を使用して SHARED リダクション変数のオリジナルの値と各プライベート・コピーの最後の値を結合した結果が反映されます。

減算を除くすべてのリダクション演算子は結合可能で、コンパイラは自由に最終値の計算を再結合できます。減算リダクションの部分的な結果は、最終値を作成するために付加されます。

共有変数の値は、最初のスレッドがリダクションを含む節に到達したときに未定義となり、そのリダクション計算が完了するまで未定義の状態になります。通常、この計算は

REDUCTION 構造の最後で完了します。ただし、NOWAIT も適用されている構造で REDUCTION 節を使用すると、共有変数はバリア同期化が行われるまで未定義のままになります。これにより、すべてのスレッドが REDUCTION 節を完了したことが保証されます。

REDUCTION 節は、1 つの領域またはワークシェアリング構造で使用されるように意図されており、リダクション変数は次のいずれかの形式を持つリダクション文でのみ使用されます。

```
x = x operator expr
x = expr operator x (except for subtraction)
x = intrinsic (x,expr)
x = intrinsic (expr, x)
```

一部のリダクションは、他の形式で使用できます。例えば、MAX リダクションは、次のように表現できます。

```
IF (x .LT.expr) x = expr
```

また、リダクションはサブルーチン・コール内に隠すこともできます。REDUCTION 節で指定された演算子はリダクション演算子と一致することに注意してください。

ディレクティブに指定できるリダクション節の数は任意ですが、次の例のように、ディレクティブの REDUCTION 節では、1 つの変数は一度しか使用できません。

```
!$OMP DO REDUCTION(+: A, Y),REDUCTION(.OR.: AM)
```

次の例は、REDUCTION 節の使用方法を示します。

```
!$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(A,B,REDUCTION(+:
A,B)
DO I=1,N
CALL WORK(ALOCAL,BLOCAL)
A = A + ALOCAL
B = B + BLOCAL
END DO
!$OMP END PARALLEL DO
```

SHARED 節

SHARED 節は、PARALLEL、PARALLEL DO、および PARALLEL SECTIONS ディレクティブ上で使用し、チーム内のすべてのスレッド間で共有できるようにします。

次の例では、変数 X および NPOINTS が、チーム内のすべてのスレッドで共有されます:

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(X,NPOINTS)
IAM = OMP GET THREAD NUM()
NP = OMP GET NUM THREADS()
IPOINTS = NPOINTS/NP
CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

スケジュールの型とチャンクサイズの設定

DO または PARALLEL DO ディレクティブの SCHEDULE 節は、DO ループの繰返しをチーム内のスレッドに分割し、振り分ける方法を決定するスケジューリング・アルゴリズムを指定します。SCHEDULE 節は、現在の DO または PARALLEL DO ディレクティブにのみ適用されます。

SCHEDULE 節内では、*スケジュールの型* と、オプションとして *チャンクサイズ* を指定します。*チャンク* とは、スレッドに振り分けられた連続する繰返しのグループです。チャンクサイズは、スカラ整数式でなければなりません。

次のリストでは、スケジュールの型とチャンクサイズがスケジューリングに与える影響を説明します:

- STATIC

  繰返しは、チャンクにより指定されたサイズに分割されます。分割された繰返しは、ラウンドロビン方式 (総当り) でスレッド番号の順番にチームのスレッドへ静的に振り分けられます。

  チャンクが指定されていない場合、最初に繰返しの回数をチーム内のスレッド数で割って、連続する繰返しの断片に分割します。それぞれの断片は、ループの実行が開始される前に、スレッドに振り分けられます。

- DYNAMIC

  繰返しは、チャンクにより指定されたサイズに分割されます。各スレッドが、現在振り分けられている繰返しの断片を終了すると、そのスレッドに対し、次の断片が動的に振り分けられます。

  チャンクが指定されていない場合、デフォルトは 1 です。

- GUIDED

チャンクサイズは、振り分けが行われるたびに指数関数的に減少します。チャンクは、毎回振り分けられる最小の繰返し回数を指定します。残りの繰返し回数がチャンクサイズよりも少なくなると、残りが振り分けられます。

チャンクが指定されていない場合、デフォルトは 1 です。

- RUNTIME

  スケジューリングに関する決定は、実行時まで延期されます。スケジュールの型とチャンクサイズは、OMP_SCHEDULE 環境変数を使用して、実行時に指定できます。

  RUNTIME の指定時には、チャンクサイズを指定することはできません。

次のリストは、使用されるスケジュールの型の優先度を示します:

1. 現在の DO または PARALLEL DO ディレクティブの SCHEDULE 節で指定されたスケジュールの型
2. 現在の DO または PARALLEL DO ディレクティブの型が RUNTIME の場合、OMP_SCHEDULE 環境変数に指定されたデフォルト値
3. コンパイラのデフォルトのスケジュールの型である STATIC

次のリストは、使用されるチャンクサイズの優先度を示します:

1. 現在の DO または PARALLEL DO ディレクティブの SCHEDULE 節で指定されたチャンクサイズ
2. スケジュールの型が RUNTIME の場合、OMP_SCHEDULE 環境変数で指定された値
3. スケジュールの型が DYNAMIC および GUIDED の場合、デフォルト値 1
4. 現在の DO または PARALLEL DO ディレクティブのスケジュールの型が STATIC の場合、チーム内のスレッド数により分割されたループの繰返し空間

OpenMP* のサポート・ライブラリ

OpenMP* をサポートするインテル® Fortran コンパイラは、製品サポート・ライブラリ libguide.a を提供します。このライブラリを使用して、アプリケーションを異なる実行モードで実行することができます。これは、既にチューニングされているアプリケーションによる通常実行またはパフォーマンスが重要な実行において使用します。

実行モード

OpenMP をサポートするコンパイラは、ランタイム時に指定した実行モードでアプリケーションを実行することができます。このライブラリは、シリアル (serial)、ターンアラウンド (turnaround) およびスループット (throughput) モードをサポートします。これらのモードは、実行時に `kmp_library` 環境変数を使用することによって選択されます。

ターンアラウンド

並列マシン上のロードが一定ではない、またはジョブ・ストリームが予測できないマルチユーザ環境下では、スループット用にデザインおよびチューニングする方が良い場合もあります。これにより、複数のジョブを同時に実行した際の合計時間を最小限に抑えることができます。このモードでは、作業スレッドは追加の並行作業の待機中、他のスレッドへ作業を渡します。

スループット・モードは、プログラムにその実行環境を認知させ (つまりシステムの読み込み)、リソースの使用を調整することで、動的環境における効率の良い実行を行えるよう設計されています。このモードはデフォルトです。

1 つの並列領域の実行が完了すると、スレッドは新しい並列作業が使用可能になるまで待機します。その後一定期間が経過すると、スレッドが待機状態からスリープ状態に移行します。スリープ状態のスレッドは、追加の並列作業が使用可能になるまでの間、並列領域間で実行される非 OpenMP のスレッドコードか、他のアプリケーションによってのみ使用されることができます。スリープ状態に移行するまでの待機時間を設定するには、`KMP_BLOCKTIME` 環境変数または `kmp_set_blocktime()` 関数を使用します。`KMP_BLOCKTIME` の値を小さくすると、並列領域間で実行される非 OpenMP スレッドコードを含むアプリケーションの場合には、全体的なパフォーマンスが向上します。`KMP_BLOCKTIME` 値を大きくすると、スレッドを OpenMP 実行専用に予約する場合に適していますが、他の同時実行 OpenMP やスレッド・アプリケーションに悪影響を与える可能性があります。

スループット

すべてのプロセッサが、プログラムの全実行に対し排他的に割り当てられる専用 (バッチまたはシングルユーザ) 並列環境では、常にすべてのプロセッサを効果的に利用することが最も重要です。ターンアラウンド・モードは、並列計算を行うすべてのプロセッサをアクティブな状態で維持して、単一ジョブの実行時間を最小限に抑えるよう設計されています。作業スレッドは、追加の並列作業を他のスレッドにわたすことなく、アクティブな状態で待機します。

注
過剰なシステムリソースの割り当てを避けてください。過剰なシステムリソースの割り当ては、スレッドが多すぎるか、または実行時に利用可能なプロセッサが少なすぎる場合に発生します。システムリソースが過剰に割り当てられると、このモードはパフォーマンスの低下を起こします。この問題が発生した場合、スループット・モードを使用してください。

OpenMP* の環境変数

このトピックでは、標準 OpenMP* 環境変数 (OMP_ プリフィックス付き) およびインテル独自の環境変数 (KMP_ プリフィックス付き) について説明します。インテル独自の環境変数とは、標準 Fortran コンパイラへ追加されたインテル拡張機能です。

標準環境変数

変数	説明	デフォルト
OMP_SCHEDULE	ランタイム・スケジュールの型とチャンクサイズを設定します。	STATIC、チャンクサイズの指定なし
OMP_NUM_THREADS	実行時に使用するスレッド数を設定します。	プロセッサの数
OMP_DYNAMIC	スレッド数の動的な調整を有効 (true) または無効 (false) にします。	false
OMP_NESTED	ネストされた並列処理を有効 (true) または無効 (false) にします。	false

インテル拡張環境変数

環境変数	説明	デフォルト
KMP_ALL_THREADS	並列領域で使用可能な最大スレッド数を設定します。	32、4 * OMP_NUM_THREADS、4 * プロセッサ数のいずれかの最大値

KMP_BLOCKTIME	並列領域の実行が終了した後、スレッドがスリープ状態になるまでスレッドが待機する時間（ミリ秒単位）を設定します。 スループット実行モードおよび KMP_LIBRARY 環境変数も参照してください。オプションの s、m、h、d サフィックスを使用して、秒、分、時間、日数を指定します。	200 ミリ秒
KMP_LIBRARY	OpenMP ランタイム・ライブラリ・スループットを選択します。この変数の値には、実行モードを示す serial、turnaround、または throughput があります。この変数が指定されなかった場合、デフォルト値の throughput が使用されます。	throughput（実行モード）
KMP_MONITOR_STACKSIZE	プログラム実行中のブックキーピングに使用される監視スレッドに割り当てるバイト数を設定します。オプションの b、k、m、g または t サフィックスを使用して、確保するバイト数をバイト、キロバイト、メガバイト、ギガバイトまたはテラバイトで指定してください。	32k、システム最小のスレッドのスタックサイズのいずれかの最大値
KMP_STACKSIZE	各並行スレッドがプライベート・スタックとして使用するバイト数を設定します。オプションの b、k、m、g または t サフィックスを使用して、確保するバイト数をバイト、キロバイト、メガバイト、ギガバイトまたは	IA-32: 2m Itanium® コンパイラ: 4m

	テラバイトで指定してください。	
KMP_VERSION	プログラム実行中、OpenMP ランタイム・ライブラリのバージョン情報の出力を有効 (set) または無効 (unset) にします。	無効

OpenMP* ランタイム・ライブラリ・ルーチン

OpenMP* には、並列モードでプログラムを管理しやすくするために、いくつかのランタイム・ライブラリ・ルーチンを用意しています。これらのランタイム・ライブラリ・ルーチンの多くには、デフォルトとして設定可能な環境変数が対応付けられています。ランタイム・ライブラリ・ルーチンを使用すれば、これらの変数を動的に変更でき、プログラムを簡単に制御できます。いずれの場合も、ランタイム・ライブラリ・ルーチンを呼び出すと、それに対応する環境変数は無効になります。

次の表は、これらランタイム・ライブラリ・ルーチンとのインターフェイスを明記したものです。ルーチン名はユーザ名前空間に保存しています。omp_lib.f、omp_lib.h および omp_lib.mod ヘッダファイルは、コンパイラがインストールされたディレクトリ以下の INCLUDE ディレクトリにあります。omp_lib.h ヘッダファイルは、コンパイラがインストールされたディレクトリ以下の INCLUDE ディレクトリにあり、Fortran の INCLUDE 文で使用します。omp_lib.mod ファイルは、INCLUDE ディレクトリにあり、Fortran の USE 文で使用します。

このファイルには、下の表の関数で使用される 2 種類のロック omp_lock_t および omp_nest_lock_t の定義が格納されています。

このトピックでは、OpenMP ランタイム・ライブラリ・ルーチンの概要を説明します。詳細は、「OpenMP Fortran バージョン 2.0 仕様」を参照してください。

関数	説明
実行環境ルーチン	
subroutine omp_set_num_threads(*num_threads*)	後続の並列領域に使用するスレッド数を設定します。

integer *num_threads*	
integer function omp_get_num_threads()	現在の並列領域に使用されているスレッドの数を返します。
integer function omp_get_max_threads()	並列実行に利用可能なスレッドの最大数を返します。
integer function omp_get_thread_num()	このコード部分を現在実行しているスレッドを表す一意のスレッド番号を取得します。
integer function omp_get_num_procs()	プログラムで利用できるプロセッサ数を決定します。
logical function omp_in_parallel()	並列で実行されている並列領域の動的な範囲内で呼ばれた場合、.TRUE. を返します。そうでない場合は、.FALSE. を返します。
subroutine omp_set_dynamic(*dynamic_threads*) logical *dynamic_threads*	並列領域の実行に使用するスレッド数の動的な調整を有効または無効にします。*dynamic_threads* が .TRUE. の場合は、動的スレッドは有効です。*dynamic_threads* が .FALSE. の場合は、動的スレッドは無効です。動的スレッドはデフォルトでは無効です。
logical function omp_get_dynamic()	動的スレッドの調整が有効の場合は、.TRUE. を返します。そうでない場合は、.FALSE. を返します。
subroutine omp_set_nested(*nested*) integer *nested*	ネストされた並列処理を有効または無効にします。*nested* が .TRUE. の場合は、ネストされた並列処理は有効です。*nested* が .FALSE. の場合は、ネストされた並列処理は無効です。 ネストされた並列処理はデフォルトでは無効です。
logical function omp_get_nested()	ネストされた並列処理が有効な場合 .TRUE. を返します。そうでない場合、.FALSE. を返します。
ロックルーチン	
subroutine omp_init_lock(*lock*) integer (kind=omp_lock_kind)::*lock*	後続の呼び出しに使用する *lock* に関連付けられているロックを初期化します。

subroutine omp_destroy_lock(*lock*) integer (kind=omp_lock_kind)::*lock*	*lock* に関連付けられているロックを未定義にします。
subroutine omp_set_lock(*lock*) integer (kind=omp_lock_kind)::*lock*	*lock* に関連付けられているロックが使用可能な状態になるまで実行中のスレッドを強制的に待機させます。lock が使用可能になると、スレッドに lock の所有権が与えられます。
subroutine omp_unset_lock(*lock*) integer (kind=omp_lock_kind)::*lock*	*lock* に関連付けられているロックの所有権から実行スレッドを解放します。*lock* に関連付けられているロックを実行中のスレッドが所有していない場合の動作は不定です。
logical omp_test_lock(*lock*) integer (kind=omp_lock_kind)::*lock*	*lock* に関連付けられているロックを設定しようと試みます。成功した場合、.TRUE. を返します。そうでない場合、.FALSE. を返します。
subroutine omp_init_nest_lock(lock) integer(kind=omp_nest_lock_kind)::*lock*	後続の呼び出しに使用する *lock* に関連付けられているネストされたロックを初期化します。
subroutine omp_destroy_nest_lock(*lock*) integer(kind=omp_nest_lock_kind)::*lock*	*lock* に関連付けられているネストされたロックを未定義にします。
subroutine omp_set_nest_lock(lock) integer(kind=omp_nest_lock_kind)::*lock*	*lock* に関連付けられているネストされたロックが使用可能な状態になるまで実行中のスレッドを強制的に待機させます。ロックが使用可能になると、スレッドにはそのネストされたロックの所有権が与えられます。
subroutine omp_unset_nest_lock(lock) integer(kind=omp_nest_lock_kind)::*lock*	ネストしているカウント数がゼロの場合は、*lock* に関連付けられているネストされたロックの所有権から実行中のスレッドを解放します。*lock* に関連付けられているネストされたロックを実行中のスレッドが所有していない場合の動作は不定です。

integer omp_test_nest_lock(*lock*) integer(kind=omp_nest_lock_kind)::*lock*	*lock* に関連付けられているネストされたロックを設定しようと試みます。成功した場合はネスト数を返し、失敗した場合は 0 を返します。
タイミング・ルーチン	
double-precision function omp_get_wtime()	任意の参照時間から経過したウォールクロック時間(秒)に等しい倍精度値を返します。参照時間は、プログラム実行中には変更されません。
double-precision function omp_get_wtick()	連続するクロック刻みの間隔の秒数に等しい倍精度値を返します。

インテル拡張ルーチン

インテル® Fortran® コンパイラでは、OpenMP* ランタイム・ライブラリへの拡張機能として、並列スレッドのスタックサイズの取得と設定およびメモリの割り当てルーチングループをサポートしています。

ここで説明するインテル拡張ルーチンは、ライブラリ・コードとアプリケーションが目的どおりに機能することを確認する低レベルのデバッグに使用できます。これらのルーチンを使用するには、プログラムをシーケンシャルに実行する -openmp_stubs コマンドライン・オプションを使用しなければならないため、充分注意して使用してください。これらのルーチンはまた、一般的に他のベンダの OpenMP 互換コンパイラには認識されません。これらのコンパイラでは、リンクの段階で失敗します。

スタックサイズ

多くの場合、環境変数は拡張ライブラリ・ルーチンの代わりに使用されます。例えば、並列スレッドのスタックサイズは、kmp_set_stacksize() ライブラリ・ルーチンではなく、KMP_STACKSIZE 環境変数を使用して設定します。

注
インテル拡張ルーチンへのランタイムの呼び出しは、対応する環境変数の設定よりも優先します。

kmp_set_stacksize() と kmp_get_stacksize() の各ルーチンは 32 ビットの引数のみを受け付けます。kmp_set_stacksize_s() と

kmp_get_stacksize_s() の各ルーチンは、64 ビット整数を使用できる size_t 引数を受け付けます。

Itanium® ベース・システムでは、常に kmp_set_stacksize() および kmp_get_stacksize() を使用することをお勧めします。スタックサイズを ≥ 2**32 バイト (4 GB) に設定するには、_s() の付くバージョンのルーチンを使用する必要があります。

下の表でスタックサイズのルーチンの定義を参照してください。

メモリの割り当て

インテル Fortran コンパイラでは、OpenMP ランタイム・ライブラリに対する拡張機能として、メモリ割り当てルーチンを実装しています。そのため、スレッドは各スレッドにローカルなヒープからメモリを割り当てることが可能です。これらのルーチンは、kmp_malloc、kmp_calloc および kmp_realloc です。

これらのルーチンによって割り当てられたメモリは、kmp_free ルーチンによって解放する必要があります。あるスレッドによってメモリを割り当て、別のスレッドでメモリを kmp_free'd を呼び出しても不正な処理ではありませんが、このような処理によってパフォーマンスが多少低下します。

次の表でこれらのルーチンの定義を参照してください。

関数/ルーチン	説明
スタックサイズ	
function kmp_get_stacksize_s() integer(kind=kmp_size_t_kind)kmp_get _stacksize_s	各並列スレッドがプライベート・スタックとして使用するバイト数を返します。この値は、最初の並列領域の前に kmp_get_stacksize_s で変更するか、または KMP_STACKSIZE 環境変数で変更できます。
function kmp_get_stacksize() integer kmp_get_stacksize	このルーチンは、下位互換のみ提供します。異なるインテル® プロセッサとの互換性には kmp_get_stacksize_s ルーチンを使用します。
subroutine kmp_set_stacksize_s(*size*) integer (kind=kmp_size_t_kind)	各並列スレッドがプライベート・スタックとして使用するバイト数を

size	*size* に設定します。この値は、KMP_STACKSIZE 環境変数で設定することもできます。kmp_set_stacksize_s を有効にするには、プログラムの最初の（動的に実行された）並列領域の先頭の前に呼び出す必要があります。
subroutine kmp_set_stacksize(*size*) integer *size*	このルーチンは、下位互換のみ提供します。異なるインテル・プロセッサとの互換性には kmp_set_stacksize_s(*size*) を使用します。
メモリの割り当て	
function kmp_malloc(*size*) integer(kind=kmp_pointer_kind)kmp_malloc integer(kind=kmp_size_t_kind)*size*	スレッド・ローカル・ヒープから *size* バイトのメモリブロックを割り当てます。
function kmp_calloc(*nelem*,*elsize*) integer(kind=kmp_pointer_kind)kmp_calloc integer(kind=kmp_size_t_kind)*nelem* integer(kind=kmp_size_t_kind)*elsize*	スレッド・ローカル・ヒープからサイズ *elsize* の *nelem* 要素の配列を割り当てます。
function kmp_realloc(*ptr*, *size*) integer(kind=kmp_pointer_kind)kmp_realloc integer(kind=kmp_pointer_kind)*ptr* integer(kind=kmp_size_t_kind)*size*	スレッド・ローカル・ヒープからアドレス *ptr* および *size* バイトにメモリブロックを再割り当てします。
subroutine kmp_free(*ptr*) integer (kind=kmp_pointer_kind) *ptr*	スレッド・ローカル・ヒープからアドレス *ptr* のメモリブロックを解放します。メモリは、以前に kmp_malloc、kmp_calloc、または kmp_realloc に割り当てられている必要があります。

OpenMP* の使用例

次の例では、OpenMP* 機能の使用方法を示します。「OpenMP Fortran バージョン 2.0 仕様」の例も参照してください。

DO: 単純な差分演算子

この例は、各繰返しごとに命令数が異なる単純な並列ループを示したものです。負荷のバランスをとるために、動的スケジューリングを使用しています。並列領域の最後に暗黙的な BARRIER があるため、END DO に NOWAIT が含まれています。

```
       subroutine do 1 (a,b,n)
real a(n,n), b(n,n)
c$omp parallel
c$omp&   shared(a,b,n)
c$omp&   private(i,j)
c$omp do schedule(dynamic,1)
do i = 2, n
do j = 1, i
b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp end parallel
end
```

DO: 2 つの差分演算子

この例では、fork/join のオーバーヘッドを減らすために融合される 2 つの並列領域を示します。2 番目のループで使用するすべてのデータは最初のループで使用されるすべてのデータと異なるため、最初の END DO には NOWAIT を含んでいます。

```
       subroutine do 2 (a,b,c,d,m,n)
real a(n,n), b(n,n), c(m,m), d(m,m)
c$omp parallel
c$omp&   shared(a,b,c,d,m,n)
c$omp&   private(i,j)
c$omp do schedule(dynamic,1)
do i = 2, n
do j = 1, i
b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp do schedule(dynamic,1)
do i = 2, m
do j = 1, i
d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp end parallel
end
```

SECTIONS: 2 つの差分演算子

この例では、SECTIONS ディレクティブの使用方法を示します。ロジックは、前述の DO の例と同じですが、DO の代わりに SECTIONS を使用します。ここでは、2 つの作業単位しかないため、速度の向上は、2 が限度です。上記の「do: 2 つの差分演算子」では、作業単位は `n-1 + m-1` です。

```
  subroutine sections 1 (a,b,c,d,m,n)
real a(n,n), b(n,n), c(m,m), d(m,m)
!$omp parallel
!$omp& shared(a,b,c,d,m,n)
!$omp& private(i,j)
!$omp sections
!$omp section
do i = 2, n
do j = 1, i
b(j,i)=( a(j,i) + a(j,i-1) ) / 2
enddo
enddo


!$omp section
do i = 2, m
do j = 1, i
d(j,i)=( c(j,i) + c(j,i-1) ) / 2
enddo
enddo
!$omp end sections nowait
!$omp end parallel
end
```

SINGLE: 共有スカラの更新

この例では、共有配列 a の要素を更新する `SINGLE` 構造の使用方法を示します。最初のループの後にくるオプションの NOWAIT は取り除かれています。これは、SINGLE 構造に進む前にループの最後で待機する必要があるためです。

```
  subroutine sp 1a (a,b,n)
real a(n), b(n)
!$omp parallel
!$omp& shared(a,b,n)
!$omp& private(i)
!$omp do
do i = 1, n
a(i) = 1.0 / a(i)
enddo
!$omp single
a(1) = min( a(1), 1.0 )
!$omp end single
```

```
!$omp do
do i = 1, n
b(i) = b(i) / a(i)
enddo
!$omp end do nowait
!$omp end parallel
end
```

マルチスレッド・プログラムのデバッグ

マルチスレッド・プログラムのデバッグの概要

ここで説明するマルチスレッド・プログラムのデバッグは、OpenMP* Fortran API とインテル® Fortran 並列コンパイラ・ディレクティブの両方に適用されます。プログラムが並列分解ディレクティブを使用するとき、誤った文または不正な並列分解ディレクティブのいずれかによって、バグが発生する可能性があることを考慮する必要があります。どちらの場合でも、デバッグするプログラムは、同時にマルチスレッドで実行できます。

次のアプリケーションなどを使用して、マルチスレッド・プログラムをデバッグすることが可能です。

- IA-32 アプリケーション用のインテル® デバッガと Itanium® ベース・アプリケーション用のインテル・デバッガ (idb)
- インテル Fortran コンパイラのデバッグ・オプションとメソッド。
- 低レベルデバッグ用のインテル並列化拡張ルーチン
- 問題のある個所を明確にするインテル® VTune™ パフォーマンス・アナライザ

その他のよく知られているデバッグ方法とヒントは次のとおりです。

- 単一プロセッサ環境のシングルスレッドでプログラムを修正する
- 静的にロックを解析する
- トレース文 (PRINT 文など) を使用する
- 前提を少なく、並列思考する。
- コード全体をステップ実行する。
- スレッドとコールスタック情報を理解する
- プライマリ・スレッドを識別する
- デバッグしているスレッドを知る

- 1 つのスレッドで 1 つずつステップ実行することは、他の複数のスレッドで 1 つずつステップ実行することとは異なります。
- コンテキストの切り替えを監視する

マルチスレッド・プログラムに対するデバッガの制限

IA-32 アプリケーション用のインテル・デバッガ (IDB) や Itanium ベース・アプリケーション用のインテル・デバッガ (IDB) などのデバッガは、マルチスレッドで実行されるプログラムのデバッグをサポートしています。しかし、これらのデバッガの現在のバージョンは、並列分解ディレクティブのデバッグを直接的にはサポートしていません。つまり、デバッグの機能には制限があります。

OpenMP で使用される新しい機能のいくつかは、まだ完全にデバッガにはサポートされていません。よって、デバッグ方法を知るために、これらの機能がどのように動作するかを理解することは重要です。問題となる個所は次の 2 つです。

- 複数のエントリポイント
- 共有変数

インテル・デバッガ (IDB) は、マルチスレッドに関連した独自の OpenMP 機能を処理しません。

並列領域のデバッグ

コンパイラは、領域のコードを有効にして、コンパイラが生成した、別のエントリポイントにコードを置くことによって、並列領域をサポートします。これは、アウトライン化 (他のコンパイラに使用される手法、つまりサブルーチンの作成) とは異なりますが、同じデバッグ手法を適用できます。

エントリポイント名の構成

コンパイラが生成する並列領域のエントリポイント名は、次の連続した文字列で構成されます。

- "__" 文字
- オリジナルのルーチンのエントリポイント名 (例: `_parallel`)
- "_" 文字
- 並列領域の行番号
- OpenMP* 並列領域の `__par_region` (`!$OMP PARALLEL`)
  OpenMP 並列ループの `__par_loop` (MP PARALLEL DO)、

OpenMP 並列セクションの __par_section(!$OMP PARALLEL SECTIONS)

- 並列領域の連続番号 (各ソースファイルで、連続番号は 0 から始まります。)

ルーチン名 (例: padd) やエントリ名 (例 : _PADD, ___PADD_6__par_loop0) が使用されると、Fortran コンパイラは次のように動作します:
デフォルトでは、コンパイラは最初に小文字および大文字/小文字の混在したルーチン名を大文字に変換します。例えば、pAdD() は PADD() に変換され、これに "_" を付けることによりエントリ名になります。その後に、第 2 のエントリ名の変換が行われます。そのため、エントリ名の "__par_loop" 部分は小文字のままになります。これは、デバッガがブレークポイントを設定する際、大文字のルーチン名 "PADD" を認識せず、代わりに、小文字の "padd" を認識していたためです。

例 1 では、並列領域によるコードのデバッグを示します。例 1 は次のコマンドで生成されます。

ifort -openmp -g -O0 -S *file.f90*

例 1 のサブルーチン parallel のコードを検証します。

サブルーチン PARALLEL() のソースリスト

```
1     subroutine parallel
2     integer id,OMP_GET_THREAD_NUM
3  !$OMP PARALLEL PRIVATE(id)
4     id = OMP_GET_THREAD_NUM()
5  !$OMP END PARALLEL
6     end
```

3 行目が並列領域です。コンパイラは 2 つのエントリポイント、parallel_ と ___parallel_3__par_region0 を作成しました。最初のエントリポイントはサブルーチン parallel() に対応し、2 番目のエントリポイントは、3 行目の OpenMP 並列領域に対応しています。

例 1: 並列領域によるコードのデバッグ

サブルーチン parallel() のマシン・コード・リスト

```
        .globl parallel_
parallel_:
..B1.1:                         # Preds ..B1.0
..LN1:
pushl     %ebp                                  #1.0
```

```
movl      %esp, %ebp                                    #1.0
subl      $44, %esp                                     #1.0
pushl     %edi                                          #1.0
... ... ... ... ... ... ... ... ... ... ... ... ... ...
..B1.13:                      # Preds ..B1.9
addl      $-12, %esp                                    #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.2, (%esp)        #6.0
movl      $0, 4(%esp)                                   #6.0
movl      $_parallel__6__par_region1, 8(%esp)           #6.0
call      __kmpc_fork_call                              #6.0

       # LOE
..B1.31:                      # Preds ..B1.13
addl      $12, %esp                                     #6.0

       # LOE
..B1.14:                      # Preds ..B1.31 ..B1.30
..LN4:
leave                                                   #9.0
ret                                                     #9.0

       # LOE

.type parallel_,@function
.size parallel_,.-parallel_
.globl _parallel__3__par_region0
_parallel__3__par_region0:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
..B1.15:                      # Preds ..B1.0
pushl     %ebp                                          #9.0
movl      %esp, %ebp                                    #9.0
subl      $44, %esp                                     #9.0
..LN5:
call      omp_get_thread_num_                           #4.0

       # LOE eax
..B1.32:                      # Preds ..B1.15
movl      %eax, -32(%ebp)                               #4.0

       # LOE
..B1.16:                      # Preds ..B1.32
movl      -32(%ebp), %eax                               #4.0
movl      %eax, -20(%ebp)                               #4.0
..LN6:
leave                                                   #9.0
ret                                                     #9.0

       # LOE
.type _parallel__3__par_region0,@function
.size _parallel__3__par_region0,._parallel__3__par_region0
.globl _parallel__6__par_region1
_parallel__6__par_region1:
# parameter 1: 8 + %ebp
```

```
# parameter 2: 12 + %ebp
..B1.17:                        # Preds ..B1.0

pushl     %ebp                                          #9.0
movl      %esp, %ebp                                    #9.0
subl      $44, %esp                                     #9.0
..LN7:
call      omp_get_thread_num_                           #7.0

        # LOE eax
..B1.33:                        # Preds ..B1.17
movl      %eax, -28(%ebp)                               #7.0

        # LOE
..B1.18:                        # Preds ..B1.33
movl      -28(%ebp), %eax                               #7.0
movl      %eax, -16(%ebp)                               #7.0
..LN8:
leave                                                   #9.0
ret                                                     #9.0
.align    4,0x90
# mark_end;
```

このレベルでプログラムをデバッグすることは、POSIX スレッド を直接使用するプログラムをデバッグするようなものです。他のルーチンと同じように、スレッドコード内で、ブレークポイントを設定できます。GNU デバッガの場合、ソースレベル・ルーチン名 (parallel など) にブレークポイントを設定できます。また、エントリポイント名 (parallel_、_parallel_3_par_region0 など) にも設定できます。Linux* 版インテル® Fortran コンパイラが、大文字の Fortran サブルーチン名を小文字に変換したことに注意してください。

マルチスレッドのデバッグ

デバッガでは、1 つのスレッドから別のスレッドに切り替えることができます。各スレッドは独自のプログラム・カウンタを持つため、各スレッドをコード中の異なる場所に配置できます。例 2 は、Fortran サブルーチンの `PADD()` を示します。OpenMP* の並列領域のエントリポイントにブレークポイントを設定することが可能です。

サブルーチン `PADD()` のソースリスト

```
12.       SUBROUTINE PADD(A, B, C, N)
13.       INTEGER N
14.       INTEGER A(N), B(N), C(N)
15.       INTEGER I, ID, OMP GET THREAD NUM
16. !$OMP PARALLEL DO SHARED (A, B, C, N) PRIVATE(ID)
17.       DO I = 1, N
18.         ID = OMP GET THREAD NUM()
19.         C(I) = A(I) + B(I) + ID
20.       ENDDO
```

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

コールスタック・ダンプ

下記の最初のコールスタックは、GNU デバッガを使用して、サブルーチン PADD へのエントリ時点でブレークしています。この時点では、プログラムはどの OpenMP 領域にも実行されていません。よって、スレッドは 1 つだけです。コールスタックは、システム・ランタイム __libc_start_main 関数が Fortran メイン・プログラムの parallel() 関数を呼び出し、そして parallel() 関数がサブルーチン padd() を呼び出すことを示しています。プログラムが複数のスレッドにより実行される際、1 つのスレッドから別のスレッドにプログラムを切り替えることができます。2 番目、3 番目のコールスタックは、並列領域へのエントリ時点でブレークしています。マスタのコールスタックには、一連の呼び出しが含まれています。コールスタックの上位は、_padd__6__par_loop0() 関数です。スレッドのエントリポイントは、インテル OpenMP ライブラリ関数呼び出し (__kmp プリフィックスによる) のレイヤーを起動します。ワーカースレッドのコールスタックには、インテル OpenMP ライブラリ関数呼び出しのレイヤーで始まる、部分的な一連の呼び出しが含まれています。

ERRATA: GNU デバッガは、インテル OpenMP ライブラリ関数 __kmpc_fork_call() の即時の呼び出し元のコールスタックを正常にアンワインドできない場合があります。

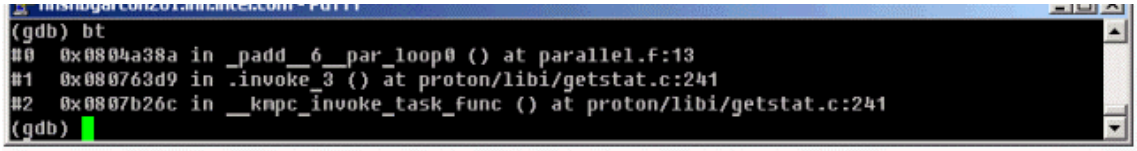
PADD サブルーチンへのエントリ時のマスタスレッドのコールスタック・ダンプ

```
(gdb) bt
#0  0x0804a031 in padd (a=(), b=(), c=(), n=10) at parallel.f:1
#1  0x0804a595 in parallel () at parallel.f:27
#2  0x400a6507 in __libc_start_main (main=0x804a3b6 <parallel>, argc=1, ubp_av=0xbffff8f4,
    init=0x8049854 <_init>, fini=0x8080dc4 <_fini>, rtld_fini=0x8000dc14 <_dl_fini>,
    stack_end=0xbffff8ec) at ../sysdeps/generic/libc-start.c:129
(gdb)
```

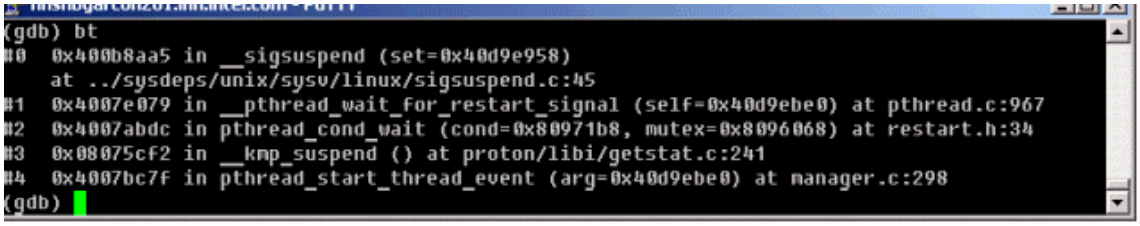
1 つのスレッドから別のスレッドへの切り替え

```
(gdb) info threads
* 4 Thread 2051 (LWP 17512)  0x0804a38a in _padd__6__par_loop0 () at parallel.f:13
  3 Thread 1026 (LWP 17511)  0x40144a31 in __libc_nanosleep () from /lib/i686/libc.so.6
  2 Thread 2049 (LWP 17510)  0x4016f9f7 in __poll (fds=0x80abd5c, nfds=1, timeout=2000)
    at ../sysdeps/unix/sysv/linux/poll.c:63
  1 Thread 1024 (LWP 17493)  0x0804a38a in _padd__6__par_loop0 () at parallel.f:13
(gdb)
```

並列領域へのエントリ時におけるマスタスレッドのコールスタック・ダンプ



並列領域へのエントリ時におけるワーカースレッドのコールスタック・ダンプ



例 2: 共有変数を持つマルチスレッドを使用するコードのデバッグ

サブルーチン PADD() のマシン・コード・リスト

```
      .globl padd
padd :
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4(n): 20 + %ebp
..B1.1:                         # Preds ..B1.0
..LN1:
pushl     %ebp                                          #1.0
... ... ... ... ... ... ... ... ... ... ... ... ... ...

..B1.19:                        # Preds ..B1.15
addl      $-28, %esp                                    #6.0
movl      $.2.1 2 kmpc loc struct pack.1, (%esp)        #6.0
movl      $4, 4(%esp)                                   #6.0
movl      $ padd  6  par loop0, 8(%esp)                 #6.0
movl      -196(%ebp), %eax                              #6.0
movl      %eax, 12(%esp)                                #6.0
movl      -152(%ebp), %eax                              #6.0
movl      %eax, 16(%esp)                                #6.0
movl      -112(%ebp), %eax                              #6.0
movl      %eax, 20(%esp)                                #6.0
lea       20(%ebp), %eax                                #6.0
movl      %eax, 24(%esp)                                #6.0
call        kmpc fork call                              #6.0

                # LOE
..B1.39:                        # Preds ..B1.19
addl      $28, %esp                                     #6.0
jmp       ..B1.31         # Prob 100%                   #6.0

                # LOE
```

```
..B1.20:                        # Preds ..B1.30
... ... ... ... ... ... ... ... ... ... ... ... ...

call        kmpc for static init 4               #6.0

                # LOE
..B1.40:                        # Preds ..B1.20
addl      $36, %esp                              #6.0

                # LOE
... ... ... ... ... ... ... ... ... ... ... ... ...

..B1.26:                        # Preds ..B1.28 ..B1.21
addl      $-8, %esp                              #6.0
movl      $.2.1 2 kmpc loc struct pack.1, (%esp) #6.0
movl      -8(%ebp), %eax                         #6.0
movl      %eax, 4(%esp)                          #6.0
call        kmpc for static fini                 #6.0

                # LOE
..B1.41:                        # Preds ..B1.26
addl      $8, %esp                               #6.0
jmp       ..B1.31         # Prob 100%            #6.0

                # LOE
..B1.27:                        # Preds ..B1.28 ..B1.25
..LN7:
call      omp get thread num                     #8.0

                 # LOE eax
..B1.42:                         # Preds ..B1.27
... ... ... ... ... ... ... ... ... ... ... ... ...

cmpl      %edx, %eax                             #10.0
jle       ..B1.27       # Prob 50%               #10.0
jmp       ..B1.26       # Prob 100%              #10.0
             # LOE
.type padd ,@function
.size padd ,.-padd
.globl  padd  6  par loop0
 padd  6  par loop0:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4: 20 + %ebp
# parameter 5: 24 + %ebp
# parameter 6: 28 + %ebp
..B1.30:                         # Preds ..B1.0
..LN16:
pushl     %ebp                                   #13.0
movl      %esp, %ebp                             #13.0
subl      $208, %esp                             #13.0
movl      %ebx, -4(%ebp)                         #13.0
..LN17:
```

```
movl      8(%ebp), %eax                                 #6.0
movl      (%eax), %eax                                  #6.0
movl      %eax, -8(%ebp)                                #6.0
movl      28(%ebp), %eax                                #6.0
..LN18:
movl      (%eax), %eax                                  #7.0
movl      (%eax), %eax                                  #7.0
movl      %eax, -80(%ebp)                               #7.0
movl      $1, -76(%ebp)                                 #7.0
movl      -80(%ebp), %eax                               #7.0
testl     %eax, %eax                                    #7.0
jg        ..B1.20       # Prob 50%                      #7.0

              # LOE
..B1.31:                      #
Preds ..B1.41 ..B1.39 ..B1.38 ..B1.30
..LN19:
movl      -4(%ebp), %ebx                                #13.0
leave                                                   #13.0
ret                                                     #13.0
.align    4,0x90
# mark end;
```

共有変数のデバッグ

あるブロック上の PRIVATE、FIRSTPRIVATE、LASTPRIVATE、または REDUCTION の節で変数を使用するとき、その変数は、ブロック内で再宣言することで、並列領域にプライベートになります。しかし、スレッドコード内では、SHARED データは宣言されません。代わりに、ルーチンレベルで宣言します。マシン・コード・レベルでは、これらの共有変数は、スレッドのエントリポイント（`___PADD_6__par_loop0` など）への受信関数呼び出しの引数になります。

例 2 では、エントリポイント `___PADD_6_par_loop0` には、6 つの受信パラメータがあります。対応する OpenMP* の並列領域には、4 つの共有変数があります。最初の 2 つのパラメータ（parameter 1 と 2）は、コンパイラの使用に予約されています。そして、残りの 4 つの各パラメータは、1 つの共有変数に対応しています。これら 4 つのパラメータは、`PADD` 関数のマシンコードにある `__kmpc_fork_call()` への最後の 4 つのパラメータと一致します。

注

FIRSTPRIVATE、LASTPRIVATE、および REDUCTION 変数もまた、並列領域への値を受け渡しするために共有変数を必要とします。

デバッガが未対応のために、共有変数（オリジナルの名前）とその内容間の対応は、スレッドのエントリ・ポイント・レベルでは、表示することができません。しかし、関数の

コールスタックへ移動し、そのレベルで変数の内容を検証することは可能です。この手法は、共有変数の内容を検証するのに使用できます。例 2 では、PARALLEL() 関数のコールスタックへ移動したら、共有変数 A、B、C、および N の内容を検証することができます。

最適化サポート機能

最適化サポート機能の概要

サポートされているコンパイラの最適化として、アプリケーションのパフォーマンスを向上するディレクティブ、組込み関数、ランタイム・ライブラリ・ルーチンおよびユーティリティなどのインテル® Fortran 機能について、ここでは説明します。これらは、ソースコードを直接最適化するインテル Fortran 言語拡張機能です。また、パフォーマンスの分析に役立つインテル拡張ディレクティブ、組込み関数、ライブラリ・ルーチンによる最適化の例を示します。

インテル Fortran コンパイラ・ディレクティブの詳細および使用例については、『Intel® Fortran Language Reference』(英語) の第 14 章、「Directive Enhanced Compilation」を参照してください。組込み関数については、『Intel® Fortran Language Reference』(英語) の第 9 章、「Intrinsic Procedures」を参照してください。

主なコンパイラ・フェーズ、および最適化レポートを生成するオプションについては、最終トピックで説明します。最適化レポート機能は、Itanium® ベース・アプリケーションにのみ使用されます。

コンパイラ・ディレクティブ

コンパイラ・ディレクティブの概要

ここでは、ソフトウェア・パイプライン処理、ループのアンロール、プリフェッチ、およびベクトル化のようなアプリケーション・コードの最適化を向上する、インテル® Fortran の言語拡張ディレクティブについて説明します。インテル Fortran コンパイラ・ディレクティブのリスト、説明、コード例については、『Intel ® Fortran Language Reference』(英語) マニュアルの「Directive Enhanced Compilation」にある「General Directives」を参照してください 。

Itanium® ベース・アプリケーションのパイプライン処理

SWP と NOSWP ディレクティブは、ループに対してソフトウェアのパイプライン化を行うかどうかを指示します。SWP ディレクティブは、データ依存性を使用しません。しかし、プロファイル・カウントまたは lop-sided コントロール・フローに基づいたヒューリスティックを上書きします。

SWP ディレクティブで起動されるソフトウェアのパイプライン化の最適化は、最も内側のループをスケジューリングする命令を適用します。また、ループ内で命令が異なるステージに分割され、増大した命令レベルの並列処理を許可します。これで、長い待ち時間の演算による影響を減らし、結果として、より速いループを実行します。ソフトウェアのパイプライン化に選択されるループは、インライン化されないプロシージャ呼び出しを含まない、常に最も内側のループです。最適化は、完全にアンロールされたループを最も内側のループとしてもはや認識しないため、完全にアンロールするループは、追加のループを最も内側のループにすることができます (`-unroll[n]` を参照してください)。最適化のレポートをリクエスト、および表示して、ソフトウェアのパイプライン化が適用されたかどうかを確認できます (「最適化機構レポートの作成」を参照してください)。

```
!DEC$ SWP
do i = 1, m
if (a(i) .eq.0)  then
b(i) = a(i) + 1
else
b(i) = a(i)/c(i)
endif
enddo
```

これらのディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

ループカウントとループ分配

LOOP COUNT (N) ディレクティブ

LOOP COUNT (n) ディレクティブはループカウントが n になるように指定します。n は整数定数です。

LOOP COUNT の値はソフトウェアのパイプライン化、ベクトル化およびループ変換に使用されるヒューリスティックに影響を与えます。

```
!DEC$ LOOP COUNT (10000)
do i =1,m
b(i) = a(i) +1 !This is likely to enable
!the loop to get software-
!pipelined
enddo
```

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

ループ分配ディレクティブ

DISTRIBUTE POINT ディレクティブはループ分配の実行優先度を示します。

ループ分配は、大きなループを小さなループに分配することがあります。これは、より多くのループにおいてソフトウェアのパイプライン化を有効にします。ディレクティブをループの内側に置く場合、分配はディレクティブの後で行われ、あらゆるループキャリーの依存性が無視されます。ディレクティブをループの前に置く場合、コンパイラは分配する場所を決定し、データ依存性を監視します。現在、ループの内側に置かれた場合、1 つの distribute ディレクティブのみがサポートされます。

```
!DEC$ DISTRIBUTE POINT
do i =1, m
b(i) = a(i) +1
....
c(i) = a(i) + b(i) !Compiler will decide where

 !to distribute.

 !Data dependency is observed
....
d(i) = c(i) + 1
enddo


do i =1, m
b(i) = a(i) +1
....
!DEC$  DISTRIBUTE POINT
call  sub(a, n)     !Distribution will start here,

 !ignoring all loop-carried

 !dependency
c(i) = a(i) + b(i)
....
d(i) = c(i) + 1
enddo
```

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

ループのアンロールのサポート

UNROLL[n] ディレクティブは、カウントされたループをアンロールする回数をコンパイラに伝えます。

n は 0 から 255 までの整数定数です。

UNROLL ディレクティブは、各 DO ループが動作する DO 文の前になければなりません。

n が指定されると、最適化機構はループを *n* 回アンロールします。*n* が省略されるか、n が有効範囲外の場合、最適化機構はループをアンロールする回数を割り当てます。

UNROLL ディレクティブは、コマンドラインから行われるループ・アンロールの設定を変更します。

現在、ディレクティブは最も内側のループのネストにのみ適用されます。外側のループのネストに適用された場合、無視されます。コンパイラは、*n* とループカウンタを比較することによって、正しいコードを生成します。

```
DEC$ UNROLL(4)
do i = 1, m
b(i) = a(i) + 1
d(i) = c(i) + 1
enddo
```

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

プリフェッチのサポート

PREFETCH と NOPREFETCH ディレクティブは、データ・プリフェッチがメモリ参照に生成されるか、されないかを指定します。これは、コンパイラが使用するヒューリスティックに影響を与えます。

ループの前に `PREFETCH a` を置いて、ループ内で式 `a(j)` を使用する場合、コンパイラはループ内の `a(j+d)` のプリフェッチを挿入します。`d` はコンパイラによって決定されます。このディレクティブは `-O3` オプションがオンの場合にサポートされます。

```
CDEC$ NOPREFETCH c
CDEC$ PREFETCH a
do i = 1, m
b(i) = a(c(i)) + 1
enddo
```

これらのディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

ベクトル化のサポート

ここで説明するディレクティブはベクトル化をサポートします。

IVDEP ディレクティブ

IVDEP ディレクティブは、ベクトル依存性が存在していると推定されてもそれを無視するようコンパイラに命令します。正しいコードにするため、コンパイラは、想定される依存性を証明された依存性として扱います。これは、ベクトル化を行わないようにします。このディレクティブは、その決定を無視します。推定されたループの依存性が安全で、無視できる場合にのみ IVDEP を使用してください。

例えば、下記のコード・フラグメントで式 `j >= 0` が常に true の場合、IVDEP ディレクティブはコンパイラにこの情報を通信することができます。このディレクティブは、値 `j < 0` の想定されたループ・キャリー・フロー依存性が安全に無視されることをコンパイラに知らせます。

```
!DEC$ IVDEP
do i = 1, 100
a(i) = a(i+j)
enddo
```

> **注**
> ベクトル化を防ぐ証明された依存性は無視されず、想定された依存性のみが無視されます。

ディレクティブの使用方法は、ループのフォームにより異なります。

ループ 1

```
Do i
= a(*) + 1
a(*) =
enddo
```

ループ 2

```
Do i
a(*) =
= a(*) + 1
enddo
```

フォーム 1 のループの場合、a の古い値を使用しますが、DEF から USE には、ループ・キャリー・フロー依存がないと仮定します。

フォーム 2 のループの場合、a の新規の値を使用しますが、USE から DEF には、ループ・キャリー・アンチ依存がないと仮定します。

いずれの場合も、ループを分割することが有効で、ループキャリー出力依存がなくなります。

例 1

```
CDEC$ IVDEP
do  j=1,n
a(j) = a(j+m) + 1
enddo
```

例 2

```
CDEC$ IVDEP
do  j=1,n
a(j) = b(j) +1
b(j) = a(j+m) + 1
enddo
```

例 1 では、下位依存の可能性を無視して、ループのソフトウェアのパイプライン化を有効にします。

例 2 は、このループの配列 a と関わる上位依存および下位依存の可能性を示し、依存サイクルを作成しています。IVDEP では、下位依存は無視されます。

IVDEP には、IVDEP:LOOP および IVDEP:BACK の 2 つのオプションがあります。
IVDEP:LOOP オプションは、ループキャリー依存がないことを意味します。
IVDEP:BACK オプションは、下位依存がないことを意味します。

IVDEP ディレクティブは、Itanium® ベース・アプリケーションに対しても `-ivdep_parallel` オプションとともに使用されます。

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」セクションにある「Directive Enhanced Compilation」を参照してください。

ベクトライザの効率性ヒューリスティックの変更

IVDEP ディレクティブのほかにも、ベクトライザの効率性ヒューリスティックを変更するのに使用されるディレクティブがあります。

VECTOR ALWAYS
NOVECTOR
VECTOR ALIGNED
VECTOR UNALIGNED
VECTOR NONTEMPORAL

VECTOR ディレクティブは、プログラム中の後に続くループのベクトル化を制御しますが、コンパイラはネストされたループには適用しません。ネストされたそれぞれのループは、その前に、固有のディレクティブが必要です。VECTOR ディレクティブは、ループ制御文の前に配置してください。

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

VECTOR ALWAYS および NOVECTOR ディレクティブ

VECTOR ALWAYS ディレクティブは、ベクトライザの効率性ヒューリスティックを変更しますが、実際にループのベクトル化が可能な場合にのみこれは動作します。従って、想定された依存性を無視するには IVDEP を使用します。

VECTOR ALWAYS ディレクティブは、次の状況でデフォルトのコンパイラの動作を変更するのに使用されます。参照が 1 以外のストライドのベクトル化では、通常、速度の向上はみられないので、コンパイラでは参照が 1 以外のストライドが (1 のストライドの場合より) 多い場合にはベクトル化をデフォルトで行いません。次のループは、`stride 2` で 2 つの参照があります。ベクトル化はデフォルトにより無効にされますが、ディレクティブがこの動作を変更します。

```
!DEC$ VECTOR ALWAYS
do i = 1, 100, 2
a(i) = b(i)
enddo
```

一方、ループのベクトル化の回避が望ましい場合 (ベクトル化によりパフォーマンスが向上せずに劣化する場合)、NOVECTOR ディレクティブをソーステキストで使用し、ループのベクトル化を無効にします。例えば、インテル® コンパイラは、デフォルトで次のループをベクトル化します。この動作が不適切な場合は、次のように NOVECTOR ディレクティブを使用します。

```
!DEC$ NOVECTOR
do i = 1, 100
```

```
a(i) = b(i) + c(i)
enddo
```

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

VECTOR ALIGNED ディレクティブと UNALIGNED ディレクティブ

VECTOR ALWAYS と同様に、これらのディレクティブも、効率性ヒューリスティックを変更します。異なる点は、UNALIGNED 指示子および ALIGNED 指示子が、すべての配列参照に対して、コンパイラがそれぞれアライメントの合っていないデータ用の移動命令、およびアライメントされたデータ用の移動命令を使用するように指示することです。これにより、プログラム・コンテキストからアライメント状況を検出したり、参照のアライメントを合わせるダイナミックなループ・ピーリングの使用など、コンパイラの高度なアライメントの最適化はすべて無効になります。

> **注**
> ディレクティブ VECTOR [ALWAYS, UNALIGNED, ALIGNED] は、注意して使用してください。コンパイラの効率性ヒューリスティックの変更は、ベクトル化によりパフォーマンスが向上することが確実である場合にのみ行います。また、アライメントの合ったデータ移動命令ですべての配列参照をコンパイラに実装すると、アライメントの合っていないアクセスパターンがある場合にランタイム例外が発生します。

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

VECTOR NONTEMPORAL ディレクティブ

VECTOR NONTEMPORAL ディレクティブは、Pentium® 4 プロセッサ・ベース・システムのストリーミング・ストアをもたらします。生成されたアセンブリとのループ (浮動小数点型) の例は、次のとおりです。n が大きくなると、Pentium 4 プロセッサ・システムでの非ストリーミング組込み関数のパフォーマンスが大幅に向上します。

次に、VECTOR NONTEMPORAL ディレクティブの使用例を示します:

```
       subroutine set(a,n)
 integer i,n
 real a(n)
!DEC$ VECTOR NONTEMPORAL
!DEC$ VECTOR ALIGNED
 do i = 1, n
   a(i) = 1
 enddo
 end
```

```
program setit
parameter(n=1024*1204)
real a(n)
integer i
do i = 1, n
  a(i) = 0
enddo
call set(a,n)
do i = 1, n
   if (a(i).ne.1) then
     print *, 'failed nontemp.f', a(i), i
     stop
   endif
enddo
print *, 'passed nontemp.f'
end
```

ディレクティブの詳細は、『Intel® Fortran Language Reference』(英語) の「General Directives」にある「Directive Enhanced Compilation」を参照してください。

最適化とデバッグ

ここでは、コンパイルのデバッグやコンパイル・エラーの表示およびチェックに使用できるコマンドライン・オプションについて説明します。

最適化のときにデバッグ情報を得ることができるオプションは次のとおりです:

-O0	最適化を無効にします。-fp オプションを有効にします。
-g	ソースレベルでのデバッガで使用するため、オブジェクト・コードの中にシンボリック・デバッグ情報と行番号を生成します。コマンドラインで -g とともに -O2 (または -O1、または -O3) が明示的に指定されていなければ、デフォルトの -O2 オプションをオフにして -O0 をデフォルトにします。
-debug *キーワード*	デバッグを向上させる設定を指定します。このオプションを使用するには、-g も指定する必要があります。*キーワード* には、ローカルスカラ変数を検出するのに役立つデバッグ情報を生成する variable_locations のみが選択できます。
-fp IA-32 のみ	最適化で ebp レジスタの使用を無効にします。すべての関数について ebp ベースのスタックフレームを使用するように指示します。

シンボリック・デバッグのサポート (-g)

-g オプションを使用して、ソースレベルでのデバッガが使用するオブジェクト・コード内に、シンボリック・デバッグ情報と行番号を提供するコードを生成するよう、コンパイラに指示します。次に例を示します。

```
ifort -g prog1.f
```

コマンドラインで -g とともに -O2 (または -O1、または -O3) が明示的に指定されていなければ、デフォルトの -O2 オプションをオフにして -O0 をデフォルトにします。

ebp レジスタの使用

-fp (IA-32 のみ)

ほとんどのデバッガは、スタック・バックトレースを生成するためのスタック・フレーム・ポインタとして ebp レジスタを使用します。-fp オプションは、最適化に ebp レジスタを使用できないようにして、すべての関数のスタック・フレーム・ポインタとして ebp レジスタを使用するコードを生成するようにコンパイラに指示します。これにより、デバッガは -O1、-O2、または -O3 の最適化をオフにすることなく、スタック・バックフレームをそのまま生成することができます。

このオプションを使用すると、利用可能な汎用レジスタが 1 つ減るため、生成するコードの効率が少し低下する場合があります。

-fp のまとめ

デフォルト	オフ
-O1、-O2、または -O3	-fp を無効にします。
-O0	-fp を有効にします。

-traceback オプション

-traceback オプションもまた、コンパイラにスタック・フレーム・ポインタとして ebp を強制的に使用するよう指示します。さらに、-traceback オプションを指定すると、コンパイラがオブジェクト・ファイルに追加情報を生成するので、ランタイム・エラーが発生した場合にシンボリック・スタック・トレースバックを生成することができます。

最適化とデバッグの組み合わせ

-O0 オプションはすべての最適化をオフにするため、最適化が行われる前にプログラムをデバッグすることができます。デバッグ情報を得るには、-g オプションを使用します。

コマンドラインで、オブジェクト・ファイルにシンボリック・デバッグ情報を生成する -g オプションとともに、最適化オプション -O1、-O2、または -O3 が指定されると、コンパイラはシンボリック・デバッグ対応のコードを生成します。

-g オプションとともに -O1、-O2、または -O3 オプションを指定すると、返されるデバッグ情報の一部が最適化の副作用として不正確になることがあります。

最適化とデバッグ・オプションは明示的に選択するようにしてください:

- 最適化の影響を受けないようにプログラムをデバッグする必要がある場合は、すべての最適化をオフにする -O0 オプションを使用してください。
- 最適化を有効にしてプログラムをデバッグする必要がある場合は、コマンドラインで -g オプションとともに -O1、-O2、または -O3 オプションを指定することができます。

注

最適化レベル (-O*n*) を指定しないで -g オプションを指定すると、プログラムの速度は遅くなります。この場合、-g オプションはプログラムの速度を遅くする -O0 オプションをオンにするためです。しかし、例えば、-O2 と -g の両方が指定された場合、コードは -g が指定されていない場合とほぼ同じ速度で実行されます。

次の表は、-g オプションと最適化オプションを組み合わせて使用した場合の効果を説明したものです。

オプション	結果
-g	デバッグ情報が生成され、-O0 (最適化無効) が有効になります。IA-32 を対象にコンパイルする場合は、-fp が有効になります。
-g -O1	デバッグ情報が生成され、-O1 による最適化が有効になります。
-g -O2	デバッグ情報が生成され、-O2 による最適化が有効になります。

-g -O3 -fp	デバッグ情報が生成され、-O3 による最適化が有効になります。IA-32 を対象にコンパイルする場合は、-fp が有効になります。

デバッグとアセンブル

アセンブリ・リスト・ファイルはデバッグ情報なしで生成されますが、オブジェクト・ファイルを生成すると、そのオブジェクト・ファイルにはデバッグ情報が含まれます。このため、オブジェクト・ファイルをリンクしてから GDB デバッガを使用すると、完全なシンボリック表現が得られます。

最適化機構レポートの作成

インテル® Fortran コンパイラは、最適化レポートを生成、管理するオプションを提供します。

- -opt_report は、最適化レポートを作成し、-opt_report_file*filename* で指定されたファイルに配置します。-opt_report_file が指定されていない場合、-opt_report はレポートを stderr に送ります。デフォルト状態はオフです。レポートは生成されません。
- -opt_report_file*filename* は、最適化レポートを作成し *filename* で指定されたファイルに送ります。
- -opt_report_level{*min*|*med*|*max*} は、最適化レポートの詳細レベルを指定します。*min* 引数は概要を、*max* 引数は、完全なレポートを作成します。デフォルトは -opt_report_level*min* です。
- -opt_report_routine [*substring*] は、名前の一部に *substring* を含むすべてのルーチンからレポートを作成します。[*substring*] が指定されていない場合、すべてのルーチンからのレポートが作成されます。デフォルトでは、コンパイル中のすべてのルーチンのレポートを作成します。

レポートを作成する最適化の指定

コンパイラは、-opt_report_phase*phase* オプションの *phase* 引数で指定される最適化機構のレポートを作成できます。

複数の最適化機構のレポートを作成するために同じコマンドライン上に複数回、オプションを使用できます。

現在、次の最適化機構レポートがサポートされています:

最適化機構の論理名	最適化機構のフルネーム
ipo	Interprocedural Optimizer (プロシージャ間の最適化機構)
hlo	High-level Language Optimizer (高水準言語の最適化機構)
ilo	Intermediate Language Scalar Optimizer (中間言語スカラ最適化機構)
ecg	Itanium®-based Compiler Code Generator (Itanium コンパイラのコード・ジェネレータ)
all	すべての最適化機構

上記の最適化機構の論理名の 1 つが指定されるとその最適化機構からのすべてのレポートが作成されます。例えば、-opt_report_phase*ipo* および -opt_report_phase*ecg* は、プロシージャ間の最適化機構およびコード・ジェネレータからのレポートを作成します。

各最適化機構は、特定の最適化を行うことができます。これらの各最適化は、論理名の 1 つがプリフィックスになります。次に例を示します。

Optimizer_optimization	フルネーム
ipo_inl	Interprocedural Optimizer (プロシージャ間の最適化機構)、inline expansion of functions (関数のインライン展開)
ipo_cp	Interprocedural Optimizer (プロシージャ間の最適化機構)、copy propagation (コピー伝播)
hlo_unroll	High-level Language Optimizer (高水準言語の最適化機構)、loop unrolling (ループのアンロール)
hlo_prefetch	High-level Language Optimizer (高水準言語の最適化機構)、Prefetching (プリフェッチ)
ilo_copy_propagation	Intermediate Language Scalar Optimizer (中間言語スカラ最適化機構)、Copy Propagation (コピー伝播)
ecg_swp	Itanium®-based Compiler Code Generator (Itanium コンパイラのコード・ジェネレータ)、Software Pipelining (ソフトウェア・パイプライン処理)

次のコマンドでは、Itanium コンパイラ のコード・ジェネレータ (ecg) のレポートを生成します。

```
ifort -c -opt_report -opt_report_phase ecg myfile.f
```

各アイテムの意味は次のとおりです。

- -c は、オブジェクト・コードの生成時にコンパイラに停止するよう命令します。リンク時ではありません。
- -opt_report は、レポート・ジェネレータを起動します。
- -opt_report_phase*ecg* は、レポートを生成するフェーズ (ecg) を指定します。オプションとフェーズ間のスペースは任意です。

最適化機構内の特定の最適化のフルネームを指定する必要はありません。数個の文字だけで十分です。 指定された最適化機構のプリフィックスと一致するすべての最適化レポートが作成されます。例えば、-opt_report_phase *ilo_co* が指定された場合、定数伝播およびコピー伝播の両方からのレポートが作成されます。

利用可能なレポート生成

-opt_report_help オプションは、レポートの作成に現在利用可能な最適化機構と最適化の論理名をリストします。

IA-32 システムの場合、次のレポートを生成できます。

- ilo
- -O3 がオンの場合、hlo
- -ip または -ipo でプロシージャ間の最適化を実行した場合、ipo
- -O3 と -ip または -ipo がオンの場合、all (上記の最適化機構)

Itanium ベース・システムの場合、次のレポートを生成できます。

- ilo
- ecg
- -O3 がオンの場合、hlo
- -ip または -ipo でプロシージャ間の最適化を実行した場合、ipo
- -O3 と -ip または -ipo がオンの場合、all (上記の最適化機構)

注

hlo または ipo のレポートを指定しても、制御オプション (-O3 と -ip または -ipo) がそれぞれオンではない場合、コンパイラは空のレポートを生成します。

用語集

アライメント	スタックの適切な境界でデータを格納すること。
代替ループ変換	ループのコピーを生成し、境界のサイズに応じて新しいループを実行する最適化。
分岐カウント・プロファイラ	プログラムが各分岐文を実行する回数をカウントするツール。このツールは、プログラムがどのように実行されたかを示すデータベースも生成します。
分岐確率データベース	分岐カウント・プロファイラによって生成されるデータベース。データベースには各分岐が実行された回数も含まれます。
キャッシュ・ヒット	プロセッサが必要とする情報がキャッシュ内にある状況。
呼び出しサイト	呼び出しサイトは、呼び出し命令の直前の命令と呼び出し命令から成る。
共通部分式の排除	冗長な計算を検出して結合する最適化。
条件文	特定の条件が真 (true) であるかどうかによって行う操作。
定数の伝播	ルーチンの引数を実際の定数値に置換する最適化。その後、コンパイラは実際の引数として使用される定数値を伝播します。
不変分岐	常に同じ分岐が行われる条件文。
定数の畳込み	プログラムを実行する場合に、演算のために数と演算子を格納する代わりに、定数式を評価して結果を使用する最適化。
コピーの伝播	変数を使用する代わりに、割り当てられた値を使用して不必要な割り当てを排除する最適化。
データフロー	システム内のデータの流れ。
不要コードの排除	未使用値を生成するコードやプログラム中で実行されないコードを排除する最適化。
ダイナミック・リンク	ランタイム時に、共有オブジェクトがプログラムの仮想アドレス空間にマップされるプロセス。
空の宣言	セミコロンの前に何も記述されていない宣言。
フレームポインタ	現在のスタックのベースアドレスを保持し、スタックフレー

	ムのアクセスに使用されるポインタ。
インライン関数の展開	各関数呼び出しを、拡張された関数に置換する最適化。
誘導変数の排除	追加分のみを使用して、複雑な配列インデックス計算処理を減らすする最適化。
命令スケジューリング	複数の命令を並列に実行できるように、生成された機械語命令を並べ替える最適化。
命令のシーケンス化	効率的でない命令を、特定のプロセッサの機能を利用する命令シーケンスに置換する最適化。
プロシージャ間の最適化	ライブラリ・ルーチンを除くプログラム全体に適用される最適化。
ループ・ブロッキング	内部ループの反復をすべて完了する前に、外部ループの反復を実行できるように命令の実行順を並べ替える最適化。
ループのアンロール	ループの反復数を減らすために、ループ内で実行された文を複製する最適化。
ループ不変コードの移動	ループ内で変更されない演算の複数のインスタンスを検出する最適化。
パディング	サイズおよびアライメント条件に一致するように、各データ型の最後にバイトまたはワードを追加すること。
プリロード	ベクトルを一度に 1 つのキャッシュでロードする最適化。ループの演算中に外部バスのターンアラウンドの数が減ります。
プロファイリング	プログラムの実行に関する詳細情報を生成するプロセス。
レジスタ変数の検出	メモリに格納する必要がない変数を検出して、レジスタ変数に配置する最適化。
副作用	コードサイズや処理時間を増加させる可能性がある最適化プロセスの結果。
スタティック・リンク	プログラムで使用している関数を含むオブジェクト・ファイルのコピーをリンク時に実行ファイルに組込むプロセス。
ストレングス・レダクション	追加分のみを使用して、複雑な配列インデックス計算処理を簡素化する最適化。
ストリップ・マイニング	キャッシの中に保持できるベクトルで内部ループ演算を可能にするためにネストのレベルを追加する最適化。この最適化により内部ループのサイズが減るため、内部

	ループで必要なすべてのデータがキャッシュに格納できるようになります。
トークンペースト	コメントで分離された 2 つのトークンを 1 つとして扱うプロセス (例えば、a/**/b は ab になる)。
変換	コードの再配置。対照的に、最適化とはランタイム・パフォーマンスの向上が保証されているコードの再配置です。
未到達コード	コンパイラで決して実行されない命令。
未使用コード	プログラム中で使用されない結果を生成する命令。
変数名の変更	別の要素を参照する変数のインスタンス名を変更する最適化。

キーワード

E

F

と

は

ひ

ふ

へ