Fortran プログラミングガイド

Forte Developer 7

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-4922-10
2002 年 6 月 Revision A

原典： *Fortran Programming Guide*
Part No: 816-2456-10
Revision A

目次

表目次

はじめに

このマニュアルでは、Forte™ Developer Fortran 95 コンパイラを使用して、効率的なアプリケーションを開発する必要があるプログラマに重要な情報を提供します。入出力、プログラム開発、ソフトウェアライブラリの使用方法と作成、プログラムの解析とデバッグ、数値精度、移植、性能、最適化、並列化、C と Fortran 間のインタフェースについて説明します。

このマニュアルは、Fortran に関する実用的な知識を持ち、Sun Fortran コンパイラの効率的な使用法を学ぼうとしている、科学者、技術者、プログラマを対象に書かれています。また、Solaris オペレーティング環境や UNIX の一般的な知識を持つ読者を対象としています。

Fortran 95 コンパイラ `f95` のコンパイル時環境とコマンド行オプションについては、関連マニュアル『Fortran ユーザーズガイド』を参照してください。

書体と記号について

次の表と記述は、このマニュアルで使用している書体と記号について説明しています。

書体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コーディング例。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 machine_name% You have mail.
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表わします。	machine_name% **su** Password:
AaBbCc123 または ゴシック	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	rm *filename* と入力します。 rm ファイル名 と入力します。
『 』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合、バックスラッシュは、継続を示します。	machinename% grep '^#define \ XV_VERSION_STRING'

- 記号 Δ は、空白が意味を持つ場合に 1 つの空白を示します。

```
ΔΔ36.001
```

- FORTRAN 77 規格では、「FORTRAN」とすべて大文字で表記する旧表記規則を使用しています。Forte Developer Fortran コンパイラの製品マニュアルでは、「FORTRAN」と「Fortran」の両方を使用しています。現在の表記規則では、「Fortran 95」と小文字を使用しています。
- オンラインマニュアル (man) ページへの参照は、トピック名とセクション番号とともに表示されます。たとえば、ライブラリルーチン `GETENV` への参照は `getenv(3F)` と表記されます。したがって、このマニュアルページにアクセスするためのコマンドは `man -s 3F getenv` になります。

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

シェルプロンプトについて

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

Forte Developer の開発ツールとマニュアルページへのアクセス

Forte Developer の製品コンポーネントとマニュアルページは、標準の `/usr/bin/` と `/usr/share/man` の各ディレクトリにインストールされません。 Forte Developer のコンパイラとツールにアクセスするには、 `PATH` 環境変数に Forte Developer コンポーネントディレクトリを必要とします。Forte Developer マニュアルページにアクセスするには、 `MANPATH` 環境変数に Forte Developer マニュアルページディレクトリが必要です。

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

注 - この節に記載されている情報は Forte Developer 製品が `/opt` ディレクトリにインストールされていることを想定しています。Forte Developer 製品が `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

Forte Developer コンパイラとツールへのアクセス方法

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

▼ PATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、 PATH 変数の現在値を表示します。

```
% echo $PATH
```

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

   パスがある場合は、PATH 変数は Forte Developer 開発ツールにアクセスできるように設定されています。パスがない場合は、次の指示に従って、 PATH 環境変数を設定してください。

▼ PATH 環境変数を設定して Forte Developer のコンパイラとツールにアクセスする

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

   /opt/SUNWspro/bin

Forte Developer マニュアルページへのアクセス方法

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

▼ MANPATH 環境変数を設定する必要があるかどうか判断するには

1. 次のように入力して、 dbx マニュアルページを表示します。

```
% man dbx
```

2. 出力された場合、内容を確認します。

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

▼ MANPATH 変数を設定して Forte Developer マニュアルページにアクセスする

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

   /opt/SUNWspro/man

Forte Developer マニュアルへのアクセス

Forte Developer の製品マニュアルには、以下からアクセスできます。

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

製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

- マニュアルは、docs.sun.com の Web サイトで入手できます。 次に示すマニュアルは、インストールされている製品のマニュアルの索引から入手できます (docs.sun.com Web サイトでは入手できません)。
  - 『Standard C++ Library Class Reference』
  - 『標準 C++ ライブラリ・ユーザーズガイド』
  - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
  - 『Tools.h++ ユーザーズガイド』

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

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

アクセスできる製品マニュアル

Forte Developer 7 製品マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセス可能なマニュアルは以下の表に示す場所から参照することができます。製品のソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルの種類	アクセス可能な形式と格納場所
マニュアル (サードパーティ製マニュアルは除く)	形式：HTML 場所： `http://docs.sun.com`
サードパーティ製マニュアル: 『Standard C++ Library Class Reference』 『標準 C++ ライブラリ・ユーザーズガイド』 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 『Tools.h++ ユーザーズガイド』	形式：HTML インストール製品について 場所：`file:/opt/SUNWspro/docs/ja/index.html` のマニュアル索引
Readme およびマニュアルページ	形式：HTML インストール製品について 場所： `file:/opt/SUNWspro/docs/ja/index.html` のマニュアル索引
リリースノート	製品 CD 内の HTML ファイル

関連する Forte Developer マニュアル

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

マニュアルタイトル	内容の説明
Fortran ユーザーズガイド	`f95` コンパイラのコンパイル時環境とコマンド行オプションについて説明しています。従来の `f77` のプログラムを `f95` に移行するためのガイドラインも記載されています。
Fortran ライブラリ・リファレンス	Fortran ライブラリと組み込みルーチンについて詳しく説明しています。
OpenMP API ユーザーズガイド	OpenMP 多重処理 API の概要とその Forte Developer 実装の詳細について説明します。
数値計算ガイド	浮動小数点演算における数値の精度に関する問題について説明しています。

関連する Solaris マニュアル

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

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris 8 Reference Manual Collection	マニュアルページの節を参照。	Solaris のオペレーティング環境に関する情報を提供しています。
Solaris 8 Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris 8 Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明します。

ご意見の送付先

米国 Sun Microsystems, Inc. では、マニュアルの向上に力を注いでおり、ユーザーのご意見やご提案をお待ちしております。ご意見などがありましたら、次のアドレスまで電子メールをお送りください。

docfeedback@sun.com

第1章

ご使用になる前に

このマニュアルおよび関連マニュアル『Fortran ユーザーズガイド』に説明されている Forte™ Developer Fortran 95 コンパイラ、`f95` は、SPARC および UltraSPARC™ フォーム上の Solaris オペレーティング環境で利用できます。このコンパイラは、公開されている Fortran 言語規格に準拠しています。また、マルチプロセッサ並列化、最適化されたコードコンパイル、C と Fortran 言語の混在のサポートなど、さまざまな拡張機能を提供します。

また、`f95` コンパイラは FORTRAN 77 と互換性があり、古い FORTRAN 77 のソースコードのほとんどを利用できます。Forte Developer には、独立した FORTRAN 77 コンパイラは含まれていません。FORTRAN 77 との互換性および FORTRAN 77 からの移行に関する問題については、『Fortran ユーザーズガイド』の第 5 章を参照してください。

規格への準拠

- `f95` は、ANSI X3. 198-1992、ISO/IEC 1539:1991、ISO/IEC 1539:1997 規格に準拠するように設計されました。
- 浮動小数点演算は、IEEE 754-1985 規格および 国際規格の IEC 60559:1989 に準拠しています。
- `f95` は、UltraSPARC の実装を含む、SPARC V8 および SPARC V9 の機能を利用した最適化をサポートします。これらの機能は『SPARC アーキテクチャマニュアル バージョン 8』(トッパン刊) およびバージョン 9 (ISBN 0-13-099227-5) で定義されています。

- このマニュアルで、「規格」は、上記の規格のバージョンに準拠していることを意味します。これらの規格の範囲外の機能を「規格外」または「拡張機能」と呼んでいます。

上記の規格は、標準化団体の責任によって改訂される場合があります。これらのコンパイラが準拠する適用可能な規格のバージョンは改訂されたり他の規格バージョンで置き換えられることがあります。その結果、Sun Fortran コンパイラの将来のバージョンが、それ以前のバージョンと機能的に互換性を持たなくなる場合があります。

Fortran 95 コンパイラの機能

Forte Developer の Fortran 95 コンパイラは、次の機能と拡張を提供します。

- 引数、共通ブロック、パラメータなどの整合性をルーチン間で調べる大域的なプログラム検査機能
- マルチプロセッサシステム用に最適化された、自動選択による明示的なループの並列化機能
- 次に挙げる機能などの、VAX/VMS Fortran 拡張機能。
  - 構造体、記録、共用体、マップ
  - 再帰
- OpenMP 並列化指令。
- Cray 形式の並列化指令。 `TASKCOMMON` など。
- 大域的、局所的、および並列化が可能な最適化は、高性能のアプリケーションを生成します。ベンチマークによると、最適化されたアプリケーションは、最適化していないコードに比べると、はるかに高速に実行できます。
- Solaris システム上の共通呼び出し規約によって、C、C++ で書かれたルーチンを Fortran プログラムと結合できます。
- UltraSPARC プロセッサにおける 64 ビット Solaris 7 環境のサポート。
- `%VAL` を使用した、値による呼び出し。
- FORTRAN 77 と Fortran 95 のプログラムおよびオブジェクトバイナリの間の互換性。

- 区間演算プログラミング。
- ストリーム入出力など、「Fortran 2000」の一部の機能。

各ソフトウェアリリースのコンパイラに追加された新規および拡張機能についての詳細は、『Fortran ユーザーズガイド』の付録 B を参照してください。

その他の Fortran ユーティリティ

次のユーティリティは、Fortran でソフトウェアプログラムを開発するときに役立ちます。

- Forte Developer Performance Analyzer — シングルスレッドアプリケーションやマルチスレッドアプリケーションの強力なパフォーマンス解析ツール。`analyzer`(1) を参照してください。
- **`asa`** — この Solaris ユーティリティは、1 桁目に Fortran のキャリッジ制御文字が入っているファイルを印刷するための Fortran 出力フィルタです。Fortran のキャリッジ制御規約によりフォーマットされたファイルを、UNIX のラインプリンタ規約によりフォーマットされたファイルに変換するとき、`asa`(1) を使用します。`asa`(1) を参照してください。
- **`fdumpmod`** — ファイルまたはアーカイブに含まれているモジュールの名前を表示するユーティリティ。`fdumpmod`(1) を参照してください。
- **`fpp`** — Fortran ソースコードプリプロセッサ。`fpp`(1) を参照してください。
- **`fsplit`** — 複数のルーチンが含まれる Fortran ファイルを複数のファイルに分割して、1 ファイル 1 ルーチンとします。`fsplit` は、FORTRAN 77 または Fortran 95 のソースファイルで使用します。`fsplit`(1) を参照してください。

デバッグユーティリティ

次のデバッグユーティリティを使用できます。

- **`-Xlist`** — 引数、COMMON ブロックなどの整合性を複数のルーチンについてチェックするコンパイラオプション。

- Forte Developer **dbx** — 安定した機能豊富な実行時および静的デバッガ。パフォーマンスデータコレクタを含んでいます。

Sun Performance Library

Sun Performance Library™ は、線形代数やフーリエ変換の数値計算用に最適化されたサブルーチンや関数のライブラリです。これは、LAPACK、BLAS1、BLAS2、BLAS3、FFTPACK、VFFTPACK、一般に Netlib (`www.netlib.org`) から使用できる LINPACK といった標準ライブラリが基になっています。

Sun Performance Library の各サブルーチンは、標準ライブラリと同じ処理を実行し、同じインタフェースを持っていますが、一般に処理速度が格段に速く、より正確で、多重処理環境で使用できます。

詳細については、`performance_library` の README ファイルと『Sun Performance Library User's Guide』を参照してください。Performance Library ルーチンのマニュアルページは、セクション 3P にあります。

区間演算

Fortran 95 コンパイラでは、`-xia` と `-xinterval` の 2 つのコンパイルフラグが提供されます。これにより、コンパイラは新しい言語拡張機能を使用し、適切なコードを生成して区間演算を実装することが可能です。

詳細については、『Fortran 95 区画演算プログラミングリファレンス』を参照してください。

マニュアルページ

オンラインマニュアル (`man`) ページは、コマンド、関数、サブルーチン、またはそれらに関する即時文書を提供します。Forte Developer の マニュアルページにアクセスするために、`MANPATH` 環境変数を設定する方法については、『はじめに』を参照してください。

次のコマンドを実行すれば、マニュアルページを表示できます。

```
demo% man topic
```

Fortran 文書では、マニュアルページのリファレンスはトピック名と man セクション番号で表されています。たとえば、f95(1) にアクセスするには、man f95 というコマンドを使用します。また、たとえば ieee_flags(3M) のように表記されているセクションにアクセスするには、次のように man コマンドで -s オプションを指定します。

```
demo% man -s 3M ieee_flags
```

Fortran ライブラリルーチンについては、マニュアルページのセクション 3F に記述されています。

次の表に、Fortran ユーザーに関係のあるマニュアルページを示します。

f95(1)	Fortran 95 のコマンド行オプション
analyzer(1)	Forte Developer Performance Analyzer
asa(1)	Fortran 復帰制御の印刷出力ポストプロセッサ
dbx(1)	コマンド行対話型デバッガ
fpp(1)	Fortran ソースコードプリプロセッサ
cpp(1)	C ソースコードプリプロセッサ
fdumpmod(1)	MODULE (.mod) ファイルの内容を表示します
fsplit(1)	プリプセロッサは Fortran のソースルーチンを分割し、1 ファイル 1 ルーチンとします
ieee_flags(3M)	浮動小数点の例外ビットを調査、設定、クリアします
ieee_handler(3M)	浮動小数点の例外を処理します
matherr(3M)	数学ライブラリのエラー処理ルーチン
ild(1)	オブジェクトファイルのインクリメンタルリンクエディタ
ld(1)	オブジェクトファイルのリンクエディタ

README ファイル

README ディレクトリには、新しい機能や、マニュアルの印刷後に発見されたソフトウェアの非互換性、バグ、および情報について説明したファイルが格納されています。このディレクトリの場所は、ソフトウェアのインストール先により異なります。パスは /opt/SUNWspro/READMEs/ja です。

表 1-1　目的の README

README ファイル	内容...
fortran_95	Fortran 95 コンパイラの f95 の新機能と変更された機能、既知の制限事項、正誤表。
fpp_readme	fpp 機能と特性の概要。
interval_arithmetic	f95 の区間演算機能の概要。
math_libraries	使用可能な最適化、特化された数学ライブラリ。
profiling_tools	パフォーマンスプロファイリングツール、prof、gprof、tcov の使用法。
runtime_libraries	エンドユーザーライセンスの観点から再配布可能なライブラリと実行可能プログラム。
performance_library	Sun Performance Library の概要。

各コンパイラの README は、-xhelp=readme コマンド行オプションを使用して簡単に表示できます。たとえば、次のようなコマンドを使用します。

```
% f95 -xhelp=readme
```

このコマンドによって、fortran_95 の README ファイルが直接表示されます。

コマンド行ヘルプ

次に示すように、コンパイラの -help オプションを起動すると、f95 のコマンド行オプションの要約を表示できます。

```
% f95 -help=flags
[ ]内の項目は省略可能。< > 内の項目は変数パラメータ。
縦線「|」はリテラル値の選択を意味します。
-someoption[=yes|no] の場合、-someoption は -someoption=yes と同等です。

_____________________________________________________________________
-a                      tcov 基本ブロックごとのプロファイル処理用データ (旧形式)
-aligncommon[=<a>]      共通のブロックエレメントを指定された境界に
                        整列させる。<a>={1|2|4|8|16}
-ansi                   ANSI 規格以外の拡張機能を報告
-autopar                自動選択によるループの並列化
-Bdynamic               動的なリンクも許容
-Bstatic                静的なリンクのみ許容
-C                      実行時の添字の範囲検査を行う
-c                      コンパイルのみ。.o ファイルを生成し、リンクは行わない
```

第2章

Fortran 入出力

Forte Developer Fortran 95 が提供する入出力機能を説明します。

Fortran プログラムからファイルに探査する

プログラムとデバイスまたはファイルとの間でデータの転送は、Fortran 論理ユニットを通じて行います。論理ユニットは、入出力文において、論理ユニット番号で識別されます。論理ユニット番号は、0 から 4 バイト整数の最大値まで (2,147,483,647) です。

文字 * が論理ユニット識別子として現れることがあります。アスタリスクは、READ 文に現れたときは標準入力ファイル、WRITE 文または PRINT 文に現れたときは標準出力ファイルを表します。

Fortran 論理ユニットは、OPEN 文を通じて、特定の名前付きファイルに関連付けることができます。また、割り当て済みユニットは、プログラムの実行開始時に自動的に特定のファイルに関連付けられます。

名前付きファイルに探査する

OPEN 文の FILE= 指定子は、実行時に、名前付き物理ファイルへの論理ユニットの関連付けを行います。ファイルはあらかじめ存在しているものでもかまいません。また、プログラムの実行時に作成することもできます。

OPEN 文の FILE= 指定子は、簡単なファイル名 (FILE='myfile.out') を指定することも、絶対ディレクトリパスか相対ディレクトリパスを前に付けたファイル名 (FILE='../Amber/Qproj/myfile.out') を指定することもできます。また、指定子は、文字定数、変数、文字式のどれでもかまいません。

ライブラリルーチンを使用して、コマンド行引数と環境変数を文字変数としてプログラムに渡し、OPEN 文でファイル名として使用できます。

次の例 (GetFilNam.f) は、入力された名前から絶対パスファイル名を作成する 1 つの方法を示しています。このプログラムは、ライブラリルーチンの GETENV、LNBLNK、GETCWD を使用して、$HOME 環境変数の値を取り出し、文字列中の最後の非空白を見つけ、現在の作業用ディレクトリを決定するものです。

```
      CHARACTER F*128, FN*128, FULLNAME*128
      PRINT*, 'ENTER FILE NAME:'
      READ *, F
      FN = FULLNAME( F )
      PRINT *, 'PATH IS:',FN
      END

      CHARACTER*128 FUNCTION FULLNAME( NAME )
      CHARACTER NAME*(*), PREFIX*128
C       これは、C シェルを仮定しています。
C       絶対パス名は変更しません。
C       '~/' で名前を始めると、チルド(~)はホームディレクトリに
C       置換されます。
C       それ以外の場合、現在のディレクトリのパスを
C       相対パス名の前に置きます。
      IF ( NAME(1:1) .EQ. '/' ) THEN
            FULLNAME = NAME
      ELSE IF ( NAME(1:2) .EQ. '~/' ) THEN
            CALL GETENV( 'HOME', PREFIX )
            FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
     1                     NAME(2:LNBLNK(NAME))
      ELSE
            CALL GETCWD( PREFIX )
            FULLNAME = PREFIX(:LNBLNK(PREFIX)) //
     1                     '/' // NAME(:LNBLNK(NAME))
      ENDIF
      RETURN
      END
```

GetFilNam.f のコンパイルと実行の結果は、次のようになります。

```
demo% pwd
/home/users/auser/subdir
demo% f95 -o getfil GetFilNam.f
demo% getfil
 ENTER FILE NAME:
getfil
 PATH IS: /home/users/auser/subdir/atest.f

demo%
```

これらのルーチンについての詳細は、13 ページの「ファイル名をプログラムに渡す」を参照してください。また、getarg(3F)、getcwd(3F)、および getenv(3F) のマニュアルページも合わせて参照してください。『Fortran ライブラリ・リファレンス』には、その他の実用的なライブラリルーチンについても記述されています。

名前を指定しないでファイルを開く

OPEN 文には名前を指定する必要はありません。 実行時システムがいくつかの規約に従い、ファイル名を補います。

一時ファイルとして開く場合

OPEN 文で STATUS='SCRATCH' を指定すると、システムは tmp.F*AAAxnnnnn* という形式の名前でファイルを開きます。*nnnnn* は現在のプロセス ID で置き換えられます。*AAA* は 3 文字の文字列を示し、*x* は英字を示します。 *AAA* と *x* によってファイル名が一意になります。プログラムを終了するか、CLOSE 文を実行すると、このファイルは直ちに削除されます。FORTRAN 77 互換モード (-f77) でコンパイルするときに、この一時ファイルが削除されないようにするには、CLOSE 文に STATUS='KEEP' を指定します。(これは規格外の拡張機能です。)

すでに開いている場合

すでにプログラムによってファイルが開かれている場合は、その後の OPEN 文を使用してファイルの特性 (たとえば、BLANK と FORM) を変更できます。この場合は、変更するファイルの論理ユニット番号とパラメータだけを指定します。

あらかじめ接続されているか暗黙の名前付きユニット

プログラムの実行開始時、3 つのユニット番号が自動的に特定の標準入出力ファイルに関連付けられます。あらかじめ接続されるユニットは、標準入力、標準出力、標準エラーです。

- 標準入力は論理ユニット 5　(Fortran 95 ユニット100 も接続)
- 標準出力は論理ユニット 6　(Fortran 95 ユニット101 も接続)
- 標準エラーは論理ユニット 0　(Fortran 95 ユニット102 も接続)

通常、標準入力は、ワークステーションのキーボードから入力を受け取ります。標準出力と標準エラーは、ワークステーションの画面に出力を表示します。

その他の場合、つまり、OPEN 文に論理ユニット番号を指定し、「FILE = 名前」を指定しない場合、ファイルは fort.*n* という形式の名前で開かれます。 *n* は論理ユニット番号です。

OPEN 文を使用せずにファイルを開く

デフォルトの規約が想定できる場合、OPEN 文は使用しなくてもかまいません (任意です)。論理ユニットへの最初の操作が OPEN または INQUIRE 以外の入出力文である場合は、ファイル fort.*n* が参照されます。*n* は論理ユニット番号です (特別な意味を持つ、0、5、6 を除きます)。

これらのファイルは、プログラムの実行の前に存在する必要はありません。ファイルへの最初の操作が OPEN 文または INQUIRE 文でない場合、ファイルは作成されます。

例 :次のコード中の WRITE 文がユニット 25 に発行される最初の入出力文である場合、ファイル fort.25 が作成されます。

```
demo% cat TestUnit.f
      IU=25
      WRITE( IU, '(I4)' ) IU
      END
demo%
```

このプログラムは、ファイル fort.25 を開いて、そのファイルに書式付き記録を 1 つ書き込みます。

```
demo% f95 -o testunit TestUnit.f
demo% testunit
demo% cat fort.25
  25
demo%
```

ファイル名をプログラムに渡す

ファイルシステムは、Fortran プログラム中の論理ユニット番号を自動的に物理ファイルに関連付けるための機能をもっていません。

しかし、Fortran プログラムにファイル名を渡す方法はいくつかあります。

実行時引数と GETARG を経由する

ライブラリルーチン getarg(3F) を使用して、実行時にコマンド行引数を文字変数に読み込むことができます。引数はファイル名として解釈され、OPEN 文の FILE= 指定子で使用されます。

```
demo% cat testarg.f
        CHARACTER outfile*40
C  ユニット 51 の出力ファイル名として最初の引数を取得する。
        CALL getarg(1,outfile)
        OPEN(51,FILE=outfile)
        WRITE(51,*) 'Writing to file:', outfile
        END
demo% f95 -o tstarg testarg.f
demo% tstarg AnyFileName
demo% cat AnyFileName
 Writing to file:AnyFileName
demo%
```

環境変数と GETENV を経由する

同様に、ライブラリルーチン getenv(3F) を使用して、実行時に環境変数の値を文字変数に読み込むことができます。 この値はファイル名として解釈されます。

```
demo% cat testenv.f
          CHARACTER outfile*40
C  ユニット 51 の出力ファイル名として $OUTFILE を取得する
          CALL getenv('OUTFILE',outfile)
          OPEN(51,FILE=outfile)
          WRITE(51,*) 'Writing to file:', outfile
          END
demo% f95 -o tstenv testenv.f
demo% setenv OUTFILE EnvFileName
demo% tstenv
demo% cat EnvFileName
 Writing to file:EnvFileName
demo%
```

getarg または getenv を使用するときには、前後の空白に気をつけなければなりません。Fortran 95 プログラムは組み込み関数 TRIM を使用でき、古い FORTRAN 77 はライブラリルーチン LNBLNK() を使用できます。この章のはじめにある例の FULLNAME 関数行を用いれば、相対パス名を利用できるようにもプログラムできます。

コマンド行における入出力のリダイレクトとパイプ

物理ファイルをプログラムの論理ユニット番号と関連付けるもう 1 つの方法は、あらかじめ接続された標準入出力ファイルをリダイレクトまたはパイプする方法です。リダイレクトやパイプは、実行時の実行コマンド上で行われます。

この方法において、標準入力 (ユニット 5) を読み取り、標準出力 (ユニット 6) か標準エラー (ユニット 0) に書き込むプログラムは、リダイレクトによって (コマンド行上で <, >, >>, >&, |, |&, 2>, 2>&1 を使用することによって)、他の名前付きファイルを読み取ったり、書き込んだりできます。

これを次の表に示します。

表 2-1　csh/sh/ksh のコマンド行におけるリダイレクトとパイプ

動作	c シェルを使用する場合	Bourne または Korn シェル を使用する場合
標準入力 - mydata から読み取る	`myprog < mydata`	`myprog < mydata`
標準出力 - myoutput に書き込む (上書き)	`myprog > myoutput`	`myprog > myoutput`
標準出力 - myoutput に書き込む (追加)	`myprog >> myoutput`	`myprog >> myoutput`
標準エラーをファイルにリダイレクトする	`myprog >& errorfile`	`myprog 2> errorfile`
標準出力を他のプログラムの入力としてパイプする	`myprog1 | myprog2`	`myprog1 | myprog2`
標準エラーと標準出力を他のプログラムにパイプする	`myprog1 |& myprog2`	`myprog1 2>&1 | myprog2`

コマンド行におけるリダイレクトとパイプについての詳細は、csh、ksh、および sh のマニュアルページを参照してください。

直接探査入出力

直接探査入出力、つまりランダム入出力を使用すると、記録番号によってファイルに直接探査できます。記録番号は、記録が書き込まれたときに割り当てられます。順番入出力とは異なり、直接探査出力記録は、どのような順番でも読み取り、あるいは書き込みできます。しかし、直接探査ファイルでは、すべての記録が同じ固定長でなければなりません。直接探査ファイルは、そのファイルの OPEN 文の ACCESS='DIRECT'指定子によって宣言されます。

直接探査ファイル中の論理記録は、OPEN 文の RECL= 指定子によって指定されたバイト長をもつ文字列です。READ 文と WRITE 文で、定義された記録サイズより大きな論理記録を指定してはなりません。(記録サイズはバイト数で指定します。)論理記録の

ほうが短い場合は差し支えありません。書式なし直接探査書き込みでは、記録中の書き込まれていない部分は不定となります。書式付き直接探査書き込みでは、書き込まれていない記録は空白で埋められます。

直接探査の READ 文と WRITE 文には、REC=*n* 引数が追加されており、読み取りや書き込みを行う記録番号を指定するようになっています。

例 :直接探査、書式なし

```
      OPEN( 2, FILE='data.db', ACCESS='DIRECT', RECL=200,
&              FORM='UNFORMATTED', ERR=90 )
      READ( 2, REC=13, ERR=30 ) X, Y
```

このプログラムでは、ファイルを直接探査、書式なし入出力、200 バイトの固定記録長で開いた後で、13 番目の記録を X と Y に読み込みます。

例 :直接探査、書式付き

```
      OPEN( 2, FILE='inven.db', ACCESS='DIRECT', RECL=200,
&              FORM='FORMATTED', ERR=90 )
      READ( 2, FMT='(I10,F10.3)', REC=13, ERR=30 ) X, Y
```

このプログラムでは、ファイルを直接探査、書式なし入出力、200 バイトの固定記録長で開きます。 次に 13 番目の記録を読み取り、(I10,F10.3) の書式を使用して変換します。

書式付きファイルの場合、書き込まれる記録のサイズは、FORMAT 文によって決定されます。上記の例では、FORMAT 文は記録を 20 文字 (またはバイト) に定義しています。並び上のデータの量が FORMAT 文で指定された記録長より大きい場合、1 つの書式付き書き込みで複数の記録を書き込むことができます。このような場合、各後続の記録には、連続する記録番号が割り当てられます。

例 :直接探査、書式付き、複数記録書き込み:

```
      OPEN( 21, ACCESS='DIRECT', RECL=200, FORM='FORMATTED')
      WRITE(21,'(10F10.3)',REC=11) (X(J),J=1,100)
```

直接探査装置 21 への書き込みによって、10 の要素からなる 10 の記録が作成されます。 なぜなら、記録ごとに 10 要素であると書式で指定しているからです。 これらの記録には、11 から 20 までの番号が割り当てられます。

バイナリ 入出力

Forte Developer Fortran 95 では OPEN 文の機能が拡張されて「バイナリ」の入出力ファイルを宣言できるようになりました。

ファイルを開くときに FORM='BINARY' と指定すると、レコード長がファイルに組み込まれないことを除いて、FORM='UNFORMATTED' とだいたい同じ結果になります。このデータがなければ、1 レコードの開始点と終了点を知らせる方法がありません。そのため、後退する場所を知らせることができないので、FORM='BINARY' ファイルに対して BACKSPACE を実行できません。'BINARY' ファイルに対して READ を実行すると、入力リストの変数を設定するために必要な量のデータが読み込まれます。

- WRITE 文。データはバイナリでファイルに書き込まれ、出力リストで指定された量のバイトが転送されます。
- READ 文。入力リストの変数にデータが読み込まれ、リストで必要なだけのバイトが転送されます。ファイルにはレコードマークがないので、「レコードの終端」エラーは検出されません。検出されるエラーは、「ファイルの終端」または異常システムエラーだけです。
- INQUIRE 文。ファイルに INQUIRE を実行するときに FORM="BINARY" と指定すると、次の結果が返されます。
  FORM="BINARY"
  ACCESS="SEQUENTIAL"
  DIRECT="NO"
  UNFORMATTED="YES"
  RECL= と NEXTREC= は未定義です。
- BACKSPACE 文。許可されていません。エラーが返されます。
- ENDFILE 文。通常とおり、現在の場所でファイルを切り捨てます。
- REWIND 文。通常とおり、ファイルの位置をデータの先頭に変更します。

ストリーム入出力

新しいストリーム入出力スキーマが Fortran 2000 規格の草案として提案され、`f95` で実装されています。ストリーム入出力アクセスは、データファイルを、1 から始まる正の整数でアドレス指定できる連続バイト列として扱います。`OPEN` 文で、ストリーム入出力を、`ACCESS='STREAM'` 指定子をつけて宣言してください。バイトアドレスへファイルを位置付けると、`READ` または `WRITE` 文に `POS=`*scalar_integer_expression* 指定子が必要になります。`INQUIRE` 文では、`ACCESS='STREAM'`、指定子 `STREAM=`*scalar_character_variable*、および `POS=`*scalar_integer_variable* が使用できます。

ストリーム入出力は、C プログラムで作成するファイルまたは読み取るファイルと相互に運用する場合に便利です。次にその例を示します。

```
C の fwrite() で作成したファイルを Fortran 95 プログラムで読み取る

program reader
 integer:: a(1024), i, result
 open(file="test", unit=8, access="stream",form="unformatted")
! a のすべてを読み取る
 read(8) a
 do i = 1,1024
   if (a(i) .ne. i-1) print *,&srq;error at &srq;, i
 enddo
! ファイルを逆方向に読み取る
 do i = 1024,1,-1
   read(8, pos=(i-1)*4+1) result
   if (result .ne. i-1) print *,&srq;error at &srq;, i
 enddo
 close(8)
end

C プログラムでファイルに書き込む

#include <stdio.h>
int binary_data[1024];

/* 32 ビットの整数を 1024 個含むファイルを作成する*/
int
main(void)
{
    int i;
    FILE *fp;

    for (i = 0; i < 1024; ++i)
        binary_data[i] = i;
    fp = fopen("test", "w");
    fwrite(binary_data, sizeof(binary_data), 1, fp);
    fclose(fp);
}
```

C プログラムでは、`fwrite()` を使用して、ファイルに 1024 個の 32 ビット整数を書き込んでいます。Fortran 95 のプログラム `reader` は、これらの整数をまず配列として読み込んでから、ファイルの最後から先頭まで逆方向に個々の整数を読み込んでいます。2 番目の `read` 文に含まれる `pos=` 指定子を見ると、位置がバイト 1 から始まるバイト数で指定されていることがわかります (C では、バイト 0 から始まります)。

内部ファイル

内部ファイルは、変数、部分列、配列、配列要素、構造化記録の欄のような、CHARACTER 型のオブジェクトです。内部ファイルからの READ の場合は、文字列の定数であってもかまいません。内部ファイルにおける入出力は、データをある文字実体から他のデータ実体に転送し、変換することによって、書式付き READ と WRITE 文をシミュレートします。ファイルの入出力は実行されません。

内部ファイルを使用するときには

- WRITE 文では、装置番号の代わりに、データを受け取る文字実体の名前が現れます。READ 文では、装置番号の代わりに、文字実体のソースの名前が現れます。
- ファイル中の 1 つの記録は、定数、変数、部分列のいずれかの実体から構成されます。
- 配列実体の場合、個々の配列要素が 1 つの記録に対応します。
- 内部ファイルへの直接探査入出力。(Fortran 95 の規格では、内部ファイルに許可されているのは順番書式付き入出力だけです。)これは、外部ファイルに対する直接探査入出力と似ていますが、ファイルにある記録数を変更できない点が異なります。この場合、記録は、文字列の配列の 1 つの要素です。この規格外の拡張機能を利用できるのは、FORTRAN 77 互換モードで -f77 を使用してコンパイルした場合だけです。
- 順番探査の READ または WRITE 文は、内部ファイルの先頭から処理を始めます。

例 :内部ファイルから書式付きで順番に読み取ります (1 記録のみ)。

```
demo% cat intern1.f
      CHARACTER X*80
      READ( *, '(A)' ) X
      READ( X, '(I3,I4)' ) N1, N2 !内部ファイル X を読み取る
      WRITE( *, * )  N1, N2
      END
demo% f95 -o tstintern intern1.f
demo% tstintern
 12 99
 12 99
demo%
```

例 :内部ファイルから書式付きで順番に読み取ります (3 記録)。

```
demo% cat intern2.f
      CHARACTER LINE(4)*16
      DATA  LINE(1) / ' 81  81 ' /
      DATA  LINE(2) / ' 82  82 ' /
      DATA  LINE(3) / ' 83  83 ' /
      DATA  LINE(4) / ' 84  84 ' /
      READ( LINE,'(2I4)') I,J,K,L,M,N
      PRINT *, I, J, K, L, M, N
      END
demo% f95 intern2.f
demo% a.out
   81  81  82  82  83  83
demo%
```

例 : 内部ファイルから `f77` 互換モードで直接探査により読み取ります (1 記録)。

```
demo% cat intern3.f
      CHARACTER LINE(4)*16
      DATA  LINE(1) / ' 81  81 ' /
      DATA  LINE(2) / ' 82  82 ' /
      DATA  LINE(3) / ' 83  83 ' /
      DATA  LINE(4) / ' 84  84 ' /
      READ ( LINE, FMT=20, REC=3 ) M, N
20     FORMAT( I4, I4 )
      PRINT *, M, N
      END
demo% f95 -f77 intern3.f
demo% a.out
   83  83
demo%
```

その他の入出力について

Forte Developer 6 Fortran 95 のプログラムと古い FORTRAN 77 のプログラムには、入出力の互換性があります。`f77` のコンパイルと `f95` のコンパイルが混在する実行ファイルでは、プログラムの `f77` 部分と `f95` 部分のどちらからでも同じ装置に対して入出力を行えます。

ただし、Fortran 95 には次のような新しい機能が追加されています。

- 次のように ADVANCED='NO' を指定すると、停留入出力が可能になります。

```
      write(*,'(a)',ADVANCE='NO')  'Enter size= '
      read(*,*) n
```

- NAMELIST 入力機能
  - f95 では、入力するときにグループ名の前に $ や & を付けることができます。Fortran 95 の標準規格で認められているのは & だけで、NAMELIST 書き込みではこれが出力されます。
  - f95 では、グループの最終データ項目が CHARACTER である ($ は入力データとして扱われる) 場合を除き、$ は入力グループの終了を示します。
  - f95 では、NAMELIST 入力を記録の最初の桁から開始することができます。
- ENCODE と DECODE は、f77 と同様に f95 でも認識および実装されます。

Fortran 95 の入出力拡張機能、およびf95 と f77 の違いについての詳細は、『Fortran ユーザーズガイド』を参照してください。

第3章

プログラム開発

この章では、Fortran プログラミングプロジェクトに使用すると大変便利な 2 つの強力なプログラム開発ツール、make と SCCS を簡単に説明します。

現在では、make および SCCS の使用方法について、優れた本が何冊も市販されています。その中に、Andrew Oram および Steve Talbott 著の『Managing Projects with make』と Don Bolinger および Tan Bronson 著の『Applying RCS and SCCS』があります。これらはともに O'Reilly & Associates から出版されています。

make ユーティリティを使用してプログラムの構築を簡単にする

make ユーティリティは、プログラムのコンパイルとリンク作業の効率を上げます。通常、大きなアプリケーションはいくつかのソースファイルと INCLUDE ファイルから構成され、さらに、いくつかのライブラリとリンクする必要があります。1 つまたは複数のソースファイルを変更すると、プログラムのその部分をコンパイルし、リンクし直さなければなりません。アプリケーションを構成するファイル間の相互依存性を指定し、各部分をコンパイルし、リンクし直すのに必要なコマンドを指定することによって、このプロセスを自動化できます。指令ファイル中にあるこれらの指定を使用して、make は、コンパイルし直す必要のあるファイルだけをコンパイルし、ユーザーが実行可能ファイルの構築に必要な、オプションとライブラリを使用してリンクします。以降の節では、簡単な例を使用して make の使用法を説明します。 要約については、make(1) のマニュアルページを参照してください。

メークファイル

「メークファイル」と呼ばれるファイルは、ソースファイルとオブジェクトファイルがお互いにどのように依存するかを構造化された方法で make に伝えるものです。さらに、これらのファイルをコンパイルし、リンクするのに必要なコマンドを定義します。

たとえば、4 つのソースファイルから成るプログラムとメークファイル (ファイル名 makefile) があるとします。

```
demo% ls
makefile
commonblock
computepts.f
pattern.f
startupcore.f
demo%
```

この例では、pattern.f と computepts.f が commonblock をインクルードするものと仮定します。そして、各 .f ファイルをコンパイルして、3 つの再配置可能なファイル (および一連のライブラリ) を pattern というプログラムにリンクします。

この場合の makefile は次のようになります。

```
demo% cat makefile
pattern:pattern.o computepts.o startupcore.o
      f95 pattern.o computepts.o startupcore.o -lcore95 \
      -lcore -lsunwindow -lpixrect -o pattern
pattern.o:pattern.f commonblock
      f95 -c -u pattern.f
computepts.o:computepts.f commonblock
      f95 -c -u computepts.f
startupcore.o:startupcore.f
      f95 -c -u startupcore.f
demo%
```

makefile の最初の行では、pattern の作成が pattern.o、computepts.o、startupcore.o に依存することを表しています。次の行以降は、再配置可能な .o ファイルとライブラリから pattern を作成するコマンドです。

makefile の各行は、ターゲットオブジェクトの依存性を表す規則と、そのオブジェクトを作成するのに必要なコマンドです。規則の構造は次のようになります。

ターゲット :依存性リスト
<TAB>構築コマンド

- 依存性 - 個々の項目は、ターゲットファイルの名前とそのターゲットが依存するすべてのファイル名を列挙した行で始まります。
- コマンド - 個々の項目には、引き続く行が 1 行以上あり、当該項目がターゲットとするファイルを構築する Bourne シェルコマンドを指定します。これらのコマンド行は、タブでインデントさせておきます。

make コマンド

make コマンドは、引数なしで、単純に次のように指定して実行できます。

```
demo% make
```

make ユーティリティは、現作業ディレクトリから makefile または Makefile という名前のファイルを検索し、その中から指示を取り出します。

make ユーティリティの一般的な動作は次のとおりです。

- 処理しなければならないターゲットファイル、それらが依存するファイル、ターゲットファイルを構築するためのコマンドをメークファイルから読み取る。
- 各ファイルが最後に変更された日付と時刻の情報を取り出す。
- ターゲットファイルの変更の日付と時刻が、依存するファイルよりも古ければ、メークファイルにあるそのターゲットに関するコマンドを使用してターゲットファイルを再度構築する。

マクロ

make ユーティリティのマクロ機能を使用すると、簡単なパラメータなしの文字列置換を行うことができます。たとえば、pattern という名のターゲットプログラムを考えてみると、それを構成する再配置可能なファイルのリストを 1 つのマクロ文字列として表現できるので、変更しやすくなります。

マクロ文字列を定義するときは、次のような形式を使用します。

名前 = 文字列

マクロ文字列を使用するときは、次のように指定します。

$（名前）

これは、make によって、マクロ文字列の実際の値に置換されます。

次の例は、すべてのオブジェクトファイルを指定するマクロ定義をメークファイルの最初に追加します。

```
OBJ = pattern.o computepts.o startupcore.o
```

これによって、メークファイルの中で、このマクロを依存性リストに使用したり、ターゲット pattern の f95 リンクコマンド上で使用したりできます。

```
pattern:$(OBJ)
      f95 $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
```

マクロ文字列の名前が 1 文字の場合、括弧は省略できます。

マクロ値を置換する

make マクロの初期値は、make のコマンド行オプションで置換できます。たとえば、次のようにします。

```
FFLAGS=-u
OBJ = pattern.o computepts.o startupcore.o
pattern:$(OBJ)
      f95 $(FFLAGS) $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
pattern.o:pattern.f commonblock
      f95 $(FFLAGS) -c pattern.f
computepts.o:
      f95 $(FFLAGS) -c computepts.f
```

この状態で、引数なしの make コマンドを実行すると、上記 FFLAGS の値が使用されます。しかし、次のようなコマンド行を使用すると、この値を置換できます。

```
demo% make "FFLAGS=-u -O"
```

make コマンド行上の FFLAGS マクロの定義は、メークファイルの初期値を無効にし、-O フラグと -u フラグを f95 に渡します。また、"FFLAGS=" をコマンド行上で使用すると、マクロに NULL 文字列を指定したことになり、マクロの影響を無効にできます。

make の接尾辞規則

メークファイルを簡単に書けるようにするため、make はターゲットファイルの接尾辞に従って、独自のデフォルト規則を使用します。

デフォルトの規則は /usr/share/lib/make/make.rules というファイルに定義されています。デフォルトの接尾辞規則を認識すると、make は、FFLAGS マクロで指定されたすべてのフラグと、-c フラグ、コンパイルすべきソースファイルの名前を引数として渡します。また、make.rulesでは、FC によって割り当てられた名前を、使用すべき Fortran コンパイラの名前として使用します。

次の例では、この規則を 2 回利用しています。

```
FC = f95
OBJ = pattern.o computepts.o startupcore.o
FFLAGS=-u
pattern:$(OBJ)
      f95 $(OBJ) -lcore95 -lcore -lsunwindow \
      -lpixrect -o pattern
pattern.o:pattern.f commonblock
      f95 $(FFLAGS) -c pattern.f
computepts.o:computepts.f commonblock
startupcore.o:startupcore.f
```

make はデフォルトの規則を使用して、computepts.f と startupcore.f をコンパイルします。

.f90 ファイルには、f95 コンパイラを起動するデフォルトの接尾辞規則があります。

しかし、FC マクロを f95 として定義しない限り、.f ファイルと .F ファイルのデフォルトの接尾辞規則は、f95 ではなく f77 を呼び出します。

また、.f95 ファイルおよび .F95 ファイルには現在は接尾辞規則が存在しません。.mod Fortran 95 モジュールファイルはモジュールコンパイラを起動します。これに対処するには、make が呼び出されるディレクトリに make.rules ファイルをコ

ピーします。このコピーを変更して、.f95 と .F95 の接尾辞規則を追加し、.mod の接尾辞規則を削除します。 詳細は、make(1S) のマニュアルページを参照してください。

SCCS によるバージョンの追跡と管理

SCCS とは、ソースコード管理システム (Source Code Control System) のことです。SCCS には次のような機能があります。

- ソースファイルの変更の記録 (変更履歴) を管理します。
- 複数のプログラマが、同時にソースファイルを変更することを防ぎます。
- バージョンスタンプによってバージョン番号を記録します。

SCCS の基本操作は次の 3 つです。

- ファイルを SCCS 管理下に置きます。
- 編集のためにファイルをチェックアウトします。
- ファイルをチェックインします。

この節では、SCCS を使用してこれらの操作を行う方法を説明し、前のプログラムを使用した簡単な例を示します。ここでは、基本的な SCCS についてのみ説明し、SCCS コマンドのうち、create、edit、delget の 3 つだけを紹介します。

SCCS を使用してファイルを管理する

ファイルをSCCS の管理下に置くには、次の処理を行う必要があります。

- SCCS ディレクトリを作成します。
- SCCS ID キーワードをファイルに挿入します (任意)。
- SCCS ファイルを作成します。

SCCS ディレクトリを作成する

まず最初に、プログラム開発を行っているディレクトリに SCCS サブディレクトリを作成しなければなりません。次のコマンドを使用します。

```
demo% mkdir SCCS
```

なお、SCCS は必ず大文字にします。

SCCS ID キーワードを挿入する

ファイルごとにいくつかの SCCS ID キーワードを挿入する開発者もいますが、これは必須ではありません。後で、SCCS の get または delget コマンドによってファイルがチェックインされるたびに、キーワードはバージョン番号によって識別されます。キーワードの文字列は次の 3 か所によく置かれます。

- コメント行
- PARAMETER 文
- 初期化データ

キーワードを使用する利点は、ソースリストの中にも、コンパイルされたオブジェクトプログラムの中にも、バージョン情報が現れることです。文字列 @(#) を前に付けておけば、what コマンドを使用して、オブジェクトファイル中のキーワードを出力できます。

パラメータとデータの定義文だけを含むヘッダーファイルをインクルードした場合は、初期化データの生成は行われず、ファイルに対するキーワードは、通常コメントの中か PARAMETER 文に付けられます。ASCII データファイルやメークファイルのようなファイルの場合には、SCCS 情報はコメントに現れます。

SCCS キーワードは %キーワード% の形式で現れ、SCCS の get コマンドによって本来の値に展開されます。頻繁に使用されるキーワードは、次のとおりです。

%Z% は、what コマンドで認識される識別子文字列 @(#) に展開されます。
%M% は、ソースファイルの名前に展開されます。%I% は、当該 SCCS が管理するファイルのバージョン番号に展開されます。%E% は、現在の日付に展開されます。

たとえば、メークファイルのコメント中で次のようなキーワードを使用すると、メークファイルを指定することができます。

```
#       %Z%%M%        %I%        %E%
```

ソースファイルの startupcore.f、computepts.f、pattern.f は、次の形式の初期化データによって指定できます。

```
      CHARACTER*50 SCCSID
      DATA SCCSID/"%Z%%M%       %I%       %E%\n"/
```

このファイルを SCCS で処理し、コンパイルし、SCCS の what コマンドでオブジェクトファイルを処理すると、次のように表示されます。

```
demo% f95 -c pattern.f
...
demo% what pattern
pattern:
      pattern.f 1.2 96/06/10
```

また、get でファイルに探査するたびに自動的に更新される、CTIME という名前の PARAMETER も作成できます。

```
      CHARACTER*(*) CTIME
      PARAMETER ( CTIME="%E%")
```

INCLUDE ファイルは、SCCS スタンプが入っている Fortran のコメントで注釈できます。

```
C   %Z%%M%       %I%       %E%
```

注 – Fortran 95 ソースコードファイルから取得した 1 文字の型成分名を使用すると、SCCS キーワード認識と競合する可能性があります。たとえば、Fortran 95 構造体成分参照 X%Y%Z は、SCCS から渡された場合、SCCS の get を実行した後に XZ となります。ここで、Fortran 95 プログラムで SCCS を使用するとき、構造体成分を定義するのに 1 文字の英字を使用しないように注意します。たとえば、Fortran 95 プログラムの構造体参照が X%YY%Z の場合、%YY% は SCCS によりキーワード参照として解釈されません。その他の方法としては、SCCS で get -k オプションを指定すると、SCCS キーワード ID を拡張しなくてもファイルが取得されます。

SCCS ファイルを作成する

これで、SCCS の `create` コマンドによって、これらのファイルを SCCS の管理下に置くことができます。

```
demo% sccs create makefile commonblock startupcore.f \
  computepts.f pattern.f
demo%
```

ファイルのチェックアウトとチェックイン

ソースコードを SCCS 管理下に置いた後は、ユーザーは SCCS を 2 つの主な作業に使用します。編集を可能にするためにファイルをチェックアウトすることと、編集の完了したファイルをチェックインすることです。

ファイルのチェックアウトには、`sccs edit` コマンドを使用します。たとえば、次のようにします。

```
demo% sccs edit computepts.f
```

この例では、SCCS は `computepts.f` の書き込み可能なコピーを現在のディレクトリに作成し、ユーザーのログイン名を記録します。あるユーザーがファイルをチェックアウトしている間、他のユーザーはそのファイルをチェックアウトできません。 しかし、他のユーザーは、誰がそのファイルをチェックアウトしているかを知ることはできます。

編集が完了したら、`sccs delget` コマンドを使用して、修正したファイルをチェックインします。たとえば、次のようにします。

```
demo% sccs delget computepts.f
```

このコマンドを実行すると、SCCS システムは次の作業を行います。

- ログイン名を比較して、ユーザーがそのファイルをチェックアウトしたユーザーかどうかを確認する。
- 変更に関するコメントを入力するようにユーザーに求める。
- この編集セッションで何が変更されたかを記録する。

- 現在のディレクトリから `computepts.f` の書き込み可能なコピーを削除する。
- 書き込み可能なコピーを、SCCS キーワードが展開された読み取り専用のコピーで置き換える。

`sccs delget` コマンドは、より簡単な SCCS の 2 つのコマンド、`delta` と `get` を組み合わせたものです。`delta` コマンドが上記の項目のうちの最初の 3 つを実行し、`get` コマンドが最後の 2 つの作業を実行します。

第4章

ライブラリ

この章では、副プログラムのライブラリを使用する方法と作成する方法を説明します。静的ライブラリと動的ライブラリの両方を説明します。

ライブラリについて

ソフトウェアライブラリとは、通常、すでにコンパイルされ、1 つのバイナリライブラリファイルにまとめられた副プログラムの集合のことです。この集合の個々のメンバーは、ライブラリの要素またはモジュールと呼ばれています。リンカーはライブラリファイルを検索し、ユーザーのプログラムによって参照されるオブジェクトモジュールを読み込み、実行可能バイナリプログラムを構築します。詳細は、`ld`(1) のマニュアルページと Solaris の『リンカーとライブラリ』を参照してください。

基本的にソフトウェアライブラリには、次の 2 種類があります。

- 静的ライブラリ - 実行前にモジュールが実行可能ファイルに結合されるライブラリ。静的ライブラリには、一般的に `lib`*name*`.a` という名前が付けられます。`.a` 接尾辞はアーカイブを意味します。
- 動的ライブラリ - 実行時にモジュールが実行可能ファイルに結合されるライブラリ。動的ライブラリには、一般的に `lib`*name*`.so` という名前が付けられます。`.so` 接尾辞は共有オブジェクトを意味します。

静的 (`.a`) バージョンと動的 (`.so`) バージョンの両方をもつ一般的なシステムライブラリを次に示します。

- Fortran 95 ライブラリ: `libfsu`、`libfui`、`libfai`、`libfai2`、`libfsumai`、`libfprodai`、`libfminlai`、`libfmaxlai`、`libminvai`、`libmaxvai`、`libifai`、`libf77compat`
- C ライブラリ: `libc`

ライブラリを使用すると、次の 2 つの利点があります。

- プログラムが呼び出すライブラリルーチンのソースコードが必要ありません。
- 必要なモジュールだけが読み込まれます。

プログラムでライブラリファイルを使用すると、一般的に使用されるサブルーチンをより簡単に共有できるようになります。プログラムのリンク時にライブラリに名前を指定するだけで、プログラム内のリファレンスを解釈処理するこれらのライブラリモジュールがリンクされて、実行可能ファイルにマージされます。

リンカーのデバッグオプションの指定

ライブラリの使用と読み込みに関する要約情報を得るには、`LD_OPTIONS` 環境変数を介してリンカーに追加オプションを渡します。コンパイラは、オブジェクトのバイナリファイルを生成するときに、これらのオプション (およびその他の必要なオプション) を使用してリンカーを呼び出します。

直接リンカーを呼び出すより、コンパイラを使用することをお勧めします。多くのコンパイラオプションが特定のリンカーオプションまたはライブラリリファレンスを必要としており、これらのオプションやリファレンスなしでリンクすると予期せぬ結果を招くおそれがあるからです。

例 : `LD_OPTIONS` 環境変数を使用してロードマップを作成する場合

```
demo% setenv LD_OPTIONS '-m -Dfiles'
demo% f95 -o myprog myprog.f
```

リンカーのオプションには、コンパイラのコマンド行と同じものがあり、`f95` コマンド上に直接指定できます。 これらのオプションは、`-B`*x*、`-d`*x*、`-G`、`-h`*name*、`-R`*path*、および `-ztext` です。詳細は、`f95`(1) のマニュアルページか『Fortran ユーザーズガイド』を参照してください。

リンカーのオプションと環境変数の例と説明は、Solaris の『リンカーとライブラリ』を参照してください。

ロードマップを作成する

リンカーの -m オプションは、ライブラリのリンク情報を表示するロードマップを生成します。実行可能バイナリプログラムの構築中にリンクされるルーチンが、そのルーチンが取り出されたライブラリと共にリストされます。

例 : -m を使用してロードマップを作成する場合

```
demo% setenv LD_OPTIONS '-m'
demo% f95 any.f
any.f:
 MAIN:
            リンクエディターメモリーマップ

出力        入力        仮想
セクション  セクション   アドレス          サイズ

.interp             100d4          11
            .interp 100d4          11 (なし)
.hash               100e8          2e8
            .hash    100e8         2e8 (なし)
.dynsym             103d0          650
            .dynsym 103d0          650 (なし)
.dynstr             10a20          366
            .dynstr 10a20          366 (なし)
.text               10c90          1e70
.text               10c90     00 /opt/SUNWspro/lib/crti.o
.text               10c90     f4 /opt/SUNWspro/lib/crt1.o
.text               10d84     00 /opt/SUNWspro/lib/values-xi.o
.text               10d88     d20 sparse.o
...
```

他の情報をリストする

他にもリンカーのデバッグ機能があり、リンカーの -D*keyword* オプションで利用できます。完全なリストを表示するには、-Dhelp を使用します。

例 : -Dhelp オプションを使用して、リンカーのデバッグ支援オプションをリストします。

```
demo% ld -Dhelp

    …
デバッグ:args      入力引数の処理を表示します (ld のみ)。
デバッグbindings シンボルバインディングを表示します;
デバッグdetail    詳しい情報を提供します。
デバッグentry     エントランス条件の記述子を表示します。

    …
demo
```

たとえば、-Dfiles リンカーオプションは、リンクの処理中に参照されるすべてのファイルとライブラリをリストします。

```
demo% setenv LD_OPTIONS '-Dfiles'
demo% f95 direct.f
direct.f:
 MAIN direct:
デバッグファイル=/opt/SUNWspro/lib/crti.o  [ ET_REL ]
デバッグファイル=/opt/SUNWspro/lib/crt1.o  [ ET_REL ]
デバッグファイル=/opt/SUNWspro/lib/values-xi.o  [ ET_REL ]
デバッグファイル=direct.o  [ ET_REL ]
デバッグファイル=/opt/SUNWspro/lib/libM77.a  [ アーカイブ ]
デバッグファイル=/opt/SUNWspro/lib/libF77.so  [ ET_DYN ]
デバッグファイル=/opt/SUNWspro/lib/libsunmath.a  [ アーカイブ ]
    …
```

他のリンカーオプションについての詳細は、『リンカーとライブラリ』を参照してください。

整合性のあるコンパイルとリンク

コンパイルとリンクを別のステップで行う場合は、整合性のあるコンパイルとリンクのオプションを選択することが重要です。オプションによっては、プログラムの一部をコンパイルするときに使用したら、リンクするときにも同じオプションを使用する必要があります。また、いくつかのオプションでは、リンクステップを含め、すべてのソースファイルをそのオプションでコンパイルする必要があります。

そのようなオプションは、『Fortran ユーザーズガイド』のオプションに関する説明の中で示されています。

例 : `-fast` を使用して `sbr.f` をコンパイルし、C ルーチンをコンパイルしてから、別のステップでリンクします。

```
demo% f95 -c -fast sbr.f
demo% cc -c -fast simm.c
demo% f95 -fast sbr.o simm.o   リンクステップ ; -fast をリンカーに渡す
```

ライブラリ検索のパスと順序の設定

リンカーは、いくつかの場所で、指定された順序でライブラリを検索します。検索の対象となるのは、標準のパスと、コンパイラオプション `-R`*path*、`-l`*library*、`-L`*dir* で指定された場所、環境変数 `LD_LIBRARY_PATH` で設定されている場所です。

標準ライブラリパスの検索順序

リンカーによって使用される標準のライブラリ検索パスはインストールパスによって決定されます。これらのパスは、静的な読み込みか動的な読み込みかによって異なります。ソフトウェアの標準インストールでのパスは `/opt/SUNWspro/` であり、ここに Forte Developer ソフトウェアおよび Fortran コンパイラがインストールされています。

静的リンク

静的リンカーは、実行可能ファイルの構築中に、次のパス (他にもあります) で、指定された順序で、ライブラリを検索します。

`/opt/SUNWspro/bin`	Forte Developer の共有ライブラリ
`/usr/ccs/lib/`	SVr4 ソフトウェアの標準の場所
`/usr/lib`	UNIX ソフトウェアの標準の場所

上記パスは、リンカーによって使用されるデフォルトのパスです。

動的リンク

動的リンカーは、実行時に、指定された順序で、共有ライブラリを検索します。

- ユーザーが -R*path* で指定したパス
- /opt/SUNWspro/lib/
- /usr/lib 標準 UNIX デフォルト

検索パスは、実行可能ファイルに組み込まれます。

LD_LIBRARY_PATH 環境変数

LD_LIBRARY_PATH 環境変数を使用して、-l*library* オプションで指定したライブラリをリンカーが検索すべきディレクトリパスを指定します。

複数のディレクトリはコロンで区切って指定できます。通常、ALD_LIBRARY_PATH 変数は、コロンで区切ったディレクトリのリストを、次のようにセミコロンで区切って 2 つ持ちます。

dirlist1 ; dirlist2

最初に、*dirlist1* のディレクトリが検索され、次に、コマンド行上で明示的に指定された -L*dir* ディレクトリが検索され、最後に、*dirlist2* と標準ディレクトリが検索されます。

つまり、次のように、複数の -L でコンパイラが呼び出された場合

f95 ... -L*path1* ... -L*pathn* ...

検索順序は次のようになります。

dirlist1 path1 ... pathn dirlist2 standard_paths

LD_LIBRARY_PATH 変数に、コロンで区切ったディレクトリリストが 1 つだけ含まれる場合、そのリストは *dirlist2* として解釈されます。

Solaris オペレーティング環境では、64 ビットの依存関係を検索するときに、類似の環境変数 LD_LIBRARY_PATH_64 を使用して LD_LIBRARY_PATH を無効にできます。詳細は、Solaris の『リンカーとライブラリ』および ld(1) マニュアルページを参照してください。

- 32 ビット SPARC プロセッサでは、LD_LIBRARY_PATH_64 は無視されます。

- LD_LIBRARY_PATH だけを定義している場合は、32 ビットと 64 ビットの両方のリンクに使用されます。
- LD_LIBRARY_PATH と LD_LIBRARY_PATH_64 を定義している場合は、32 ビットのリンクには LD_LIBRARY_PATH が使用され、64 ビットのリンクには LD_LIBRARY_PATH_64 が使用されます。

注 – 実際に運用するソフトウェアでは、可能な限り LD_LIBRARY_PATH 環境変数を使用しないでください。実行時リンカーの検索パスに影響を与える一時的なメカニズムとしては便利ですが、この環境変数を参照できる動的な実行可能ファイルはすべてその検索パスを変更します。そのため、予想できない結果になるか、パフォーマンスが低下する可能性があります。

ライブラリ検索のパスと順序 - 静的リンク

-l*library* コンパイラオプションを使用して、リンカーが外部参照を解決するときに検索する追加のライブラリを指定します。たとえば、オプション -lmylib は、ライブラリ libmylib.so か libmylib.a を検索リストに追加します。

リンカーは標準ディレクトリパスを探して、追加の libmylib ライブラリを見つけます。-L オプション (および、LD_LIBRARY_PATH 環境変数) は、標準パス以外でライブラリを探す場所をリンカーに伝えるパスのリストを作成します。

libmylib.a がディレクトリ /home/proj/libs にある場合、オプション -L/home/proj/libs は、実行可能ファイルを構築するときに探すべき場所をリンカーに伝えます。

```
demo% f95 -o pgram part1.o part2.o -L/home/proj/libs -lmylib
```

-l*library* オプションのコマンド行順序

特定の参照が解決されていない場合、ライブラリは 1 度だけ検索され、さらに、検索中のその時点で未定義のシンボルだけが検索されます。コマンド行上に複数のライブラリをリストする場合、これらのライブラリは、コマンド行に指定された順序で検索されます。-l*library* オプションは、次のように配置します。

- -l*library* オプションは .f、.for、.F、.f95、または .o ファイルの後に配置します。

- lib*x* 中の関数を呼び出し、これらの関数が lib*y* 中の関数を参照する場合、-l*x* は -l*y* より前に配置します。

-L*dir* オプションのコマンド行順序

-L*dir* オプションは、*dir* ディレクトリパスをライブラリ検索リストに追加します。リンカーは、まず、-L オプションで指定されたディレクトリでライブラリを検索し、次に、標準ディレクトリで検索します。このオプションは、適用する -l*library* オプションより前に配置された場合だけ有効です。

ライブラリ検索のパスと順序 - 動的リンク

動的ライブラリで、ライブラリ検索のパスと読み込みの順序の変更は、静的リンクのときとは異なります。実際のリンクは、構築時ではなく、実行時に行われます。

構築時に動的ライブラリを指定する

実行ファイルを構築するとき、リンカーは共有ライブラリへのパスを実行可能ファイル自身に記録します。これらの検索パスは、-R*path* オプションで指定できます。対照的に、-L*dir* オプションは、構築時に -l*library* オプションで指定されたライブラリを見つける場所を示しますが、このパスをバイナリ実行可能ファイルに記録しません。

実行可能ファイルが作成されたときに構築されるディレクトリパスは、dump コマンドを使用して表示できます。

例 : a.out に構築されたディレクトリパスをリストします。

```
demo% f95 program.f -R/home/proj/libs -L/home/proj/libs -lmylib
demo% dump -Lv a.out | grep RPATH
[5]      RPATH    /home/proj/libs:/opt/SUNWspro/lib
```

実行時に動的ライブラリを指定する

実行時、リンカーは、実行可能ファイルに必要な動的リンクを探す場所を次から決定します。

- 実行時の LD_LIBRARY_PATH の値
- 実行可能ファイルが構築されたときに、-R で指定されたパス

すでに説明したように、LD_LIBRARY_PATH の使用は予想できない副作用があるので、お勧めできません。

動的リンク中のエラーの修正

必要なライブラリを見つけることができなかったとき、動的リンカーは次のようなエラーメッセージを発行します。

```
ld.so:prog:fatal:libmylib.so:can't open file:
```

メッセージは、そこにあるべきライブラリが存在しなかったことを示しています。実行可能ファイルを構築したときには共有ライブラリのパスを指定したが、その後でライブラリが移動された可能性があります。たとえば、*/my/libs/* 中のユーザー独自の動的ライブラリを使用して a.out を構築し、その後でライブラリを他のディレクトリに移動した場合などです。

ldd を使用して、実行可能ファイルがライブラリを検索する場所を検出します。

```
demo% ldd a.out
libfui.so.1 =>   /opt/SUNWspro/lib/libfui.so.1
    libfai.so.1 =>   /opt/SUNWspro/lib/libfai.so.1
    libfai2.so.1 =>  /opt/SUNWspro/lib/libfai2.so.1
    libfsumai.so.1 =>    /opt/SUNWspro/lib/libfsumai.so.1
    libfprodai.so.1 =>   /opt/SUNWspro/lib/libfprodai.so.1
    libfminlai.so.1 =>   /opt/SUNWspro/lib/libfminlai.so.1
    libfmaxlai.so.1 =>   /opt/SUNWspro/lib/libfmaxlai.so.1
    libfminvai.so.1 =>   /opt/SUNWspro/lib/libfminvai.so.1
    libfmaxvai.so.1 =>   /opt/SUNWspro/lib/libfmaxvai.so.1
    libfsu.so.1 =>   /opt/SUNWspro/lib/libfsu.so.1
    libsunmath.so.1 =>   /opt/SUNWspro/lib/libsunmath.so.1
    libm.so.1 =>     /usr/lib/libm.so.1
    libc.so.1 =>     /usr/lib/libc.so.1
    libdl.so.1 =>    /usr/lib/libdl.so.1
    /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1
```

可能であれば、適切なディレクトリにライブラリを移動またはコピーするか、リンカーが検索するディレクトリ中にそのディレクトリへのソフトリンクを作成します (ln -s を使用します)。または、LD_LIBRARY_PATH が正しく設定されていない可能性があります。LD_LIBRARY_PATH が実行時に必要なライブラリへのパスを含んでいるかどうかを検査します。

静的ライブラリを作成する

静的ライブラリファイルは、ar(1) ユーティリティを使用して、すでにコンパイルされたオブジェクトファイル (.o ファイル) から構築します。

リンカーは、リンクするプログラム中で参照される入口を持つ要素をライブラリから抽出します。たとえば、副プログラム、入口名、BLOCKDATA 副プログラム中で初期化される COMMON ブロックなどです。これらの抽出された要素 (ルーチン) は、リンカーによって生成される a.out 実行可能ファイルに恒久的にリンクされます。

静的ライブラリの長所と短所

静的ライブラリとリンクには、動的なライブラリとリンクと比較した場合、主に 3 つの問題に注意しなければなりません。

- 静的ライブラリは自己依存性 (独立性) に優れていますが、適用性に劣ります。

a.out 実行可能ファイルを静的にリンクすると、必要なライブラリルーチンは実行可能バイナリファイルの一部となります。しかし、a.out 実行可能ファイルにリンクされた静的ライブラリルーチンを更新する必要が出てきた場合、a.out ファイル全体をリンクし、生成し直さなければ、更新されたライブラリを利用することができません。動的ライブラリを使用すれば、ライブラリは a.out ファイルの一部とはならず、リンクは実行時に行われます。更新された動的ライブラリを利用するために必要なことは、新しいライブラリをシステムにインストールするだけです。

- 静的ライブラリの「要素」は個々のコンパイル単位 .o ファイルです。

1 つのコンパイル単位 (ソースファイル) には複数の副プログラムが含まれている場合があるので、いっしょにコンパイルすると、これらのルーチンは静的ライブラリ中の 1 つのモジュールとなります。つまり、コンパイル単位中のすべてのルーチンがいっしょに a.out 実行可能ファイルに読み込まれるが、実際に呼び出されるのはこれら副プログラムの 1 つだけであるということを意味します。この状況は、複数のライブラリルーチンを複数のコンパイル可能ソースファイルに分散するという最適化によって改良できます。(ただし、プログラムによって実際に参照されるライブラリモジュールだけが実行可能ファイルに読み込まれます。)

- 静的ライブラリのリンクでは、リンクの順序が重要です。

リンカーは、コマンド行に現れる順番、すなわち左から右に入力ファイルを処理します。リンカーがライブラリの要素を読み込むべきかどうかは、すでに処理されたライブラリの要素によって決定されます。この順番は、要素がライブラリファイル中で現れる順番に依存するだけでなく、コンパイルコマンド行上で指定されたライブラリの順番にも依存します。

例 : Fortran プログラムが `main.f` と `crunch.f` の 2 つのファイルに記述され、`crunch.f` だけがライブラリにアクセスする場合、`crunch.f` または `crunch.o` より前に Sun Performance Library のライブラリを参照するとエラーになります。

```
demo% f95 main.f -lmylibrary crunch.f -o myprog          (誤)
demo% f95 main.f crunch.f -lmylibrary -o myprog          (正)
```

簡単な静的ライブラリを作成する

1 つのプログラムのルーチンすべてがいくつかのソースファイルのグループに分散されており、また、これらのソースファイルすべてがサブディレクトリ `test_lib/` にあるものと仮定します。

さらに、それぞれのファイルがユーザーのプログラムによって呼び出される 1 つの副プログラムと、その副プログラムからは呼び出されるがライブラリ中の他のルーチンからは呼び出されない「ヘルパー」ルーチンをもつように、ファイルを編成すると仮定します。また、複数のライブラリルーチンから呼び出されるヘルパールーチンはすべて 1 つのソースファイルにまとめられているとします。これによって、合理的に上手に編成されたソースファイルとオブジェクトファイルのセットができます。

各ソースファイルの名前は、そのファイルの中の最初のルーチンの名前から決定すると仮定します。 ほとんどの場合、それはライブラリ中の主要なファイルです。

```
demo% cd test_lib
demo% ls
total 14           2 dropx.f      2 evalx.f      2 markx.f
   2 delte.f       2 etc.f        2 linkz.f      2 point.f
```

低レベルの「ヘルパー」ルーチンはすべてファイル `etc.f` にまとめられます。他のファイルには、1 つまたは複数の副プログラムが入ります。

まず、-c オプションを使用して、各ライブラリソースファイルをコンパイルし、対応する再配置可能な .o ファイルを生成します。

```
demo% f95 -c *.f
demo% ls
total 42
 2 dropx.f     4 etc.o      2 linkz.f    4 markx.o
 2 delte.f     4 dropx.o    2 evalx.f    4 linkz.o     2 point.f
 4 delte.o     2 etc.f      4 evalx.o    2 markx.f     4 point.o
demo%
```

次に、ar を使用して、静的ライブラリ testlib.a を作成します。

```
demo% ar cr testlib.a *.o
```

このライブラリを使用するためには、コンパイルコマンド上にライブラリファイルを指定するか、-l と -L コンパイルオプションを使用します。次の例では .a ファイルを直接使用します。

```
demo% cat trylib.f
C    testlib ルーチン群をテストするためのプログラム
            x=21.998
            call evalx(x)
            call point(x)
            print*, 'value ',x
            end
demo% f95 -o trylib trylib.f test_lib/testlib.a
demo%
```

主プログラムがライブラリ中の 2 つのルーチンだけを呼び出しているところに注目してください。ライブラリ中の呼び出されないルーチンが実行可能ファイルに読み込まれていないことを確認するには、nm によって表示される実行可能ファイル中の名前のリストで調べます。

```
demo% nm trylib | grep FUNC | grep point
[146]       |     70016|     152|FUNC |GLOB |0    |8      |point_
demo% nm trylib | grep FUNC | grep evalx
[165]       |     69848|     152|FUNC |GLOB |0    |8      |evalx_
demo% nm trylib | grep FUNC | grep delte
demo% nm trylib | grep FUNC | grep markx
demo% ...etc
```

上記の例では、grep は名前のリストから、実際に呼び出されたライブラリルーチンの項目だけを見つけます。

ライブラリを参照するもう 1 つの方法は、-l*library* と -L*path* オプションを使用する方法です。ここでは、lib*name*.a の規則に従うため、ライブラリの名前を変更しなければなりません。

```
demo% mv test_lib/testlib.a test_lib/libtestlib.a
demo% f95 -o trylib trylib.f -Ltest_lib -ltestlib
```

-l*library* と -L*path* オプションは、他のユーザーが参照できるように、/usr/local/lib のようなシステム上の一般的にアクセス可能なディレクトリにインストールされたライブラリに使用できます。たとえば、libtestlib.a を /usr/local/lib に置いた場合、次のコマンドを使用してコンパイルするよう、他のユーザーに知らせてください。

```
demo% f95 -o myprog myprog.f -L/usr/local/lib -ltestlib
```

静的ライブラリ中の置換

2、3 の要素だけをコンパイルし直す場合、ライブラリ全体をコンパイルし直す必要はありません。ar の -r オプションを使用すると、静的ライブラリ中の個々の要素を置換できます。

例 :静的ライブラリ中の 1 つのルーチンをコンパイルし直し、置換します。

```
demo% f95 -c point.f
demo% ar -r testlib.a point.o
```

静的ライブラリ中のルーチンを整列する

ar を使用して構築しているときに静的ライブラリ中の要素を整列するには、コマンド lorder(1) と tsort(1) を使用します。

```
demo% ar -cr mylib.a 'lorder exg.o fofx.o diffz.o | tsort'
```

動的ライブラリを作成する

動的ライブラリファイルは、リンカー ld によって、実行開始後に実行可能ファイルにリンクできるコンパイル済みオブジェクトモジュールから構築されます。

動的ライブラリのもう 1 つの特長は、各プログラムのメモリーにモジュールを複製することなく、システムで実行中の他のプログラムからモジュールを使用できることです。この理由のため、動的ライブラリは共有ライブラリとも呼ばれます。

動的ライブラリには次のような特長があります。

- オブジェクトモジュールは、コンパイルとリンクの処理中に、リンカーによって実行可能ファイルにリンクされるのではありません。リンクは実行時まで延期されます。
- 共有ライブラリのモジュールは、実行プログラムがそのモジュールを初めて参照したときに、システムメモリーにリンクされます。以降の実行プログラムがそのモジュールを参照した場合、その参照は最初のコピーにマップされます。
- 動的ライブラリを使用すると、プログラムの管理が簡単になります。更新された動的ライブラリをシステムにインストールすると、すぐに、そのライブラリを使用するすべてのアプリケーションに影響を与えます。 実行可能ファイルにリンクし直す必要はありません。

動的ライブラリの長所と短所

動的ライブラリは、いくつかの長所と短所を考慮しなければなりません。

- より小さな a.out ファイル

実行時までライブラリルーチンのリンクを延期するということは、実行可能ファイルのサイズが、ライブラリの静的バージョンを呼び出す同等な実行可能ファイルより小さいということを意味します。 つまり、実行可能ファイルは、ライブラリルーチンのバイナリを含みません。

- プロセスメモリーの利用率が減少する可能性

ライブラリを使用するいくつかのプロセスが同時にアクティブになったとき、ライブラリの 1 つのコピーだけがメモリーに常駐し、そのコピーがすべてのプロセスによって共有されます。

- オーバーヘッドが増加する可能性

実行時、ライブラリルーチンを読み込み、リンク編集するための余分なプロセッサ時間が必要になります。また、ライブラリの位置独立コーディングのため、静的ライブラリにおける再配置可能なコーディングよりも実行速度が遅くなる可能性があります。

- システム全体のパフォーマンスが向上する可能性

ライブラリの共有によるメモリー利用率の減少が、システム全体のパフォーマンスにとってはよい結果となるはずです (メモリースワップの入出力アクセス時間が減少します)。

プログラムのパフォーマンス状況は、各プログラムによって大きく異なります。動的ライブラリと静的ライブラリの間でパフォーマンスの向上 (または低下) を予想することは必ずしもできません。しかし、必要なライブラリの両方の形式が利用できる場合、それぞれのライブラリを使用してユーザーのプログラムのパフォーマンスを評価する価値はあります。

位置独立コードと -xcode

位置独立コード (PIC) は、リンクエディタによる再配置を必要とせず、プログラムの任意のアドレスにリンクできます。このようなコードは、本質的に同時プロセス間で共有できます。したがって、動的共有ライブラリを構築する場合、-xcode コンパイラオプションを使用して、位置に依存しないように構成要素ルーチンをコンパイルしなければなりません。

位置独立コードの中では、大域的なデータへの個々の参照は、大域的なオフセットテーブルへのポインタを通じての参照としてコンパイルされます。関数呼び出しはそれぞれ、手続きリンケージテーブルを通して、相対アドレッシングモードでコンパイルされます。この大域的なオフセットテーブルのサイズは、SPARC プロセッサでは 8K バイトに制限されています。

コンパイラフラグ -xcode=*v* を使用すると、バイナリオブジェクトのコードアドレス空間を指定できます。このコンパイラフラグを使用して、32 ビット、44 ビット、または 64 ビットの絶対アドレス、さらに小型モデルと大型モデルの位置に依存しないコードを生成できます。(-xcode=pic13 は古い -pic と等価で、-xcode=pic32 は -PIC と等価です。)

-xcode=pic32 コンパイラオプションは -code=pic13 と似ていますが、さらに、大域的なオフセットテーブルが 32 ビットのアドレス空間に渡ることを許可します。詳細は、f95(1) のマニュアルページか『Fortran ユーザーズガイド』を参照してください。

リンクオプション

コンパイル時、ライブラリのリンクが動的であるか静的であるかを指定できます。このようなオプションは実際にはリンカーオプションですが、コンパイラによって認識されリンカーに渡されます。

-Bdynamic | -Bstatic

-Bdynamic は、可能であれば必ず共有動的リンクの優先を設定します。-Bstatic は、リンクを静的ライブラリだけに制限します。

静的バージョンと動的バージョンの両方とも利用できるとき、このオプションを使用して、コマンド行上から設定を切り替えます。

```
f95 prog.f -Bdynamic -lwells -Bstatic -lsurface
```

-dy | -dn

実行可能ファイル全体に対して動的リンクを許可または禁止します。(このオプションをコマンド行で使用できるのは 1 回だけです。)

-dy は、動的共有ライブラリへのリンクを許可します。-dn は、動的ライブラリへのリンクを禁止します。

64 ビット環境でのリンク

libm.a や libc.a などの静的システムライブラリによっては、Solaris の 64 ビットオペレーティング環境では使用できないものもあります。このようなライブラリは動的ライブラリ専用として提供されます。64 ビット環境で -dn を使用すると、いくつかの静的システムライブラリが見つからないことを示すエラーが出力されます。また、コンパイラのコマンド行を -Bstatic で終了しても同じ結果になります。

特定のライブラリの静的バージョンとリンクするには、次のようなコマンド行を使用します。

```
f95 -o prog prog.f -Bstatic -labc -lxyz -Bdynamic
```

この場合、ユーザーの libabc.a と libxyz.a ファイルがリンクされて (libabc.so や libxyz.so ではない)、最後の -Bdynamic によってシステムライブラリを含めて残りのライブラリが動的にリンクされます。

さらに複雑な状況では、適切な -Bstatic や -Bdynamic を必要に応じて使用して、リンク手順で各システムライブラリとユーザーライブラリを明示的に参照する必要があります。まず、LD_OPTIONS に '-Dfiles' を設定して、必要なライブラリをすべてリストします。次に、-nolib (システムライブラリの自動リンクを抑制する) を指定してリンク手順を実行し、必要なライブラリを明示的に参照します。たとえば、次のようにします。

```
f95 -xarch=v9 -o cdf -nolib cdf.o -Bstatic -lsunmath \
-Bdynamic -lm -lc
```

命名規則

リンクローダーとコンパイラによって想定された動的ライブラリの命名規則に従うために、ユーザーが作成した動的ライブラリの名前には接頭辞 lib と接頭辞 .so を付けます。たとえば、libmyfavs.so は、コンパイラオプション -lmyfavs によって参照できます。

また、リンカーは任意のバージョン番号接頭辞も受け付けます。たとえば、libmyfavs.so.1 はこのライブラリのバージョン 1 です。

コンパイラの -h*name* オプションは、構築される動的ライブラリの名前として *name* を記録します。

簡単な動的ライブラリ

動的ライブラリを構築するには、-xcode オプションとリンカーオプション -G、-ztext、-h*name* を使用して、ソースファイルをコンパイルしなければなりません。これらのリンカーオプションは、コンパイラのコマンド行で利用できます。

静的ライブラリの例と同じファイルを使用して、動的ライブラリを作成できます。

例 : -pic と他のリンカーオプションを使用してコンパイルします。

```
demo% f95 -o libtestlib.so.1 -G -xcode=pic13 -ztext  \
-hlibtestlib.so.1 *.f
```

-G は、動的ライブラリを構築することをリンカーに伝えます。

-ztext は、位置独立コード以外のもの (たとえば、再配置可能なテキストなど) があった場合に警告を発します。

例 : 動的ライブラリを使用して実行可能ファイル a.out を作成します。

```
demo% f95 -o trylib -R&slq;pwd&slq; trylib.f libtestlib.so.1
demo% file trylib
trylib:ELF 32-bit MSB 実行可能 SPARC バージョン 1 [動的にリンクされて
います][取り除かれていません]
demo% ldd trylib
     libtestlib.so.1 => /export/home/U/Tests/libtestlib.so.1
     libfui.so.1 => /opt/SUNWspro/lib/libFfui.so.1
     libfai.so.1 => /opt/SUNWspro/lib/libFfai.so.1
     libc.so.1 => /usr/lib/libc.so.1
```

例では、-R オプションを使用して、動的ライブラリへのパス (現在のディレクトリ) を実行可能ファイルにリンクしていることに注目してください。

file コマンドは、実行可能ファイルが動的にリンクされていることを表示します。

共通ブロックを初期化する

動的ライブラリを構築する際、初期化された共通ブロックを同じライブラリに集め、その他のライブラリの前に参照することにより、共通ブロックを正しく初期化する (DATAまたは BLOCK DATA を使用) ことを保証します。

たとえば、次のようにします。

```
demo% f95 -G -xcode=pic32 -o init.so blkdat1.f blkdat2.f blkdat3.f
demo% f95 -o prog main.f init.so otherlib1.so otherlib2.so
```

最初のコンパイルにより、共通ブロックを定義し、BLOCK DATA 単位でそれらを初期化するファイルの動的ライブラリが作成されます。2 番目のコンパイルにより、実行可能バイナリが作成され、コンパイル済み主プログラムをアプリケーションが必要とする動的ライブラリにリンクします。すべての共通ブロックを初期化する動的ライブラリは、その他すべてのライブラリより前に最初に現れます。これにより、ブロックが正しく初期化されたことが保証されます。

Sun Fortran コンパイラが提供するライブラリ

次の表に、コンパイラと共にインストールされるライブラリを示します。

表 4-1　コンパイラと共に提供される主なライブラリ

ライブラリ	名前	必要なオプション
f95 サポート組み込み手続き	libfsu	なし
f95 インタフェース	libfui	なし
f95 配列組み込み手続き	libf*ai	なし
f95 区間演算組み込みライブラリ	libifai	-xinterval
サンの数学関数ライブラリ	libsunmath	なし

出荷可能なライブラリ

ユーザーの実行可能ファイルが、`runtime.libraries` README ファイルにリストされている Sun 動的ライブラリを使用している場合、ユーザーのライセンスには、そのライブラリをユーザーの顧客に再配布する権利が含まれます。

この README ファイルは次の READMEs ディレクトリにあります。

```
/opt/SUNWspro/READMEs/ja
```

ヘッダーファイル、ソースコード、オブジェクトモジュール、オブジェクトモジュールの静的ライブラリは、いかなる形式でも再配布または公開しないでください。

詳細は、ソフトウェアライセンスを参照してください。

第5章

プログラムの解析とデバッグ

この章では、プログラムの解析とデバッグを容易にするコンパイラの機能について説明します。

大域的なプログラムの検査 (-Xlist)

`-Xlist` オプションは、ソースプログラムに不整合がないか、実行時に発生しそうな問題がないかを解析します。コンパイラが行う解析は、大域的に、つまり副プログラム間で行われます。

`-Xlists` は、境界整列のエラー、副プログラムの引数、共通ブロック、パラメータの数や型の対応のエラー、およびその他のさまざまな種別のエラーを報告します。

`-Xlist` はまた、詳細なソースコードのリストとクロスリファレンステーブルも作成します。

`-Xlist` オプションでコンパイルされたプログラムには、自動的にその解析ファイルがバイナリファイル中に構築されます。それによって、ライブラリのプログラム間で大域的なプログラム検査を行うことができます。

GPC の概要

大域的なプログラムの検査 (GPC) は、`-Xlist`*x* オプションで呼び出され、次のことを行います。

- 特に、別々にコンパイルされるルーチン間で、通常より厳重な Fortran の型検査の規則を適用する。

- 別のマシンや異なるオペレーティングシステムの間で、プログラムを移動するときに必要な移植上のいくつかの制約を適用する。
- 正当ではあっても無駄が多かったりエラーにつながりそうな構造を検出する。
- その他のバグやあいまいな箇所を指摘する。

さらに具体的には、大域的なチェック機能によって次のような問題が報告されます。

- インタフェースの問題
  - 仮引数と実引数の数と型の衝突
  - 関数値の間違った型
  - 別々の副プログラムの共通ブロックで、データ型の不適合のために生じる衝突の可能性
- 使用上の問題
  - サブルーチンとして使用された関数、または関数として使用されたサブルーチン
  - 宣言されたが使用されていない関数、サブルーチン、変数、文番号
  - 参照されたが宣言されていない関数、サブルーチン、変数、文番号
  - 不定の変数の使用
  - 実行されることのない文
  - 暗黙の型の変数
  - 名前付き共通ブロックの長さ、名前、配置の不整合

大域的なプログラム検査の起動方法

`-Xlist` オプションをコマンド行に指定すると、コンパイラの大域的なプログラムアナライザが起動されます。以降の節では、`-Xlist` のサブオプションについて説明します。

例 :基本的な大域的なプログラム検査用に 3 つのファイルをコンパイルします。

```
demo% f95 -Xlist  any1.f  any2.f  any3.f
```

上記の例では、コンパイラは次のことを行います。

- `any1.lst` ファイルに出力リストを生成する。
- エラーがなければプログラムのコンパイルとリンクを行う。

画面への出力

通常、-Xlist*x* によって生成される出力リストはファイルに書き込まれます。直接画面に表示するには、-Xlisto を使用して、出力ファイルを /dev/tty に書き込みます。

例 :端末に表示します。

```
demo% f95 -Xlisto /dev/tty  any1.f
```

デフォルトの出力機能

-Xlist オプションは、出力で利用できる機能を組み合わせたものです。他の -Xlist オプションを指定していない場合は、デフォルトで次のことを行います。

- 出力リストのファイル名は、最初に現れた入力ソースまたはオブジェクトのファイルの名前で、接尾辞は .lst に置換される。
- 行番号付きソースリスト
- ルーチン間の不整合に関するエラーメッセージ (リストの当該箇所に埋め込まれる)
- 識別子のクロスリファレンステーブル
- 1 ページ 66 行、1 行 79 カラムのページ割り
- コールグラフは生成しない。
- include ファイルは展開しない。

ファイル形式

検査プロセスは、コンパイラコマンド行に指定されたすべてのファイル (接尾辞 .f、.f90、.f95、.for、.F、.F95、.o の付くファイル) を認識します。.o ファイルは、サブルーチンと関数の名前など、大域的な名前に関する情報だけをプロセスに提供します。

-Xlist と大域的なプログラム検査の例

次の例で使用される Repeat.f ソースコードを示します。

```
demo% cat Repeat.f
        PROGRAM repeat
          pn1 = 27.005
          CALL subr1 ( pn1 )
          CALL newf ( pn1 )
          PRINT *, pn1
        END

        SUBROUTINE subr1 ( x )
          IF ( x .GT. 1.0 ) THEN
           CALL subr2 ( x * 0.5 )
          END IF
        END

        SUBROUTINE newf( ix )
          INTEGER PRNOK
          IF (ix .eq. 0) THEN
                 ix = -1
          ENDIF
          PRINT *, prnok ( ix )
        END

        INTEGER FUNCTION prnok ( x )
          prnok = INT ( x ) + .05
        END

        SUBROUTINE unreach_sub()
          CALL sleep(1)
        END

        SUBROUTINE subr2 (x)
          CALL subr1(x+x)
        END
```

例 : -XlistX を使用して、エラー、警告、および相互参照を表示します。

```
demo% f95 -XlistX Repeat.f
demo% cat Repeat.lst
Repeat.f                    (水)  5月  8 11:35:19 2002                    ページ 1

FILE  "Repeat.f"
program  repeat
     4             CALL newf ( pn1 )
                                 ^
**** エラー  #418:実引数 "pn1" は real ですが、仮引数は integer です。
                "Repeat f" の 14 行目を参照してください。
     5          PRINT *, pn1
                         ^
**** エラー  #570:  変数 "pn1" は real として参照されていますが、4 行目では integer として設
定されています。

subroutine  newf
    19            PRINT *, prnok ( ix )
                                   ^
**** エラー  #418:引数 "ix" は integer ですが、仮引数は real です。
                "Repeat f" の 22 行目を参照してください。
function  prnok
    23            prnok = INT ( x ) + .05
                  ^
**** 警告  #1024: 型 "integer*4" の値への型 "real*4" の疑わしい代入値

subroutine  unreach_sub
    26          SUBROUTINE unreach_sub()
                           ^
**** 警告  #338:  サブルーチン "unreach_sub" はプログラムから呼び出されません。
subroutine  subr2
    31            CALL subr1(x+x)
                  ^
**** 警告  #348:"subr1" の再帰的呼び出し。動的呼び出しを参照してください:
                  "Repeat f" の 10 行目
                  "Repeat f" の 3 行目

相互参照                    (水)  5月  8 11:35:19 2002                  ページ 2




          相 互 参 照 表

  ソースファイル:Repeat.f


凡例:
D         定義 / 宣言
U         単純な使用
M         変更箇所
A         実引数
C         サブルーチン / 関数呼び出し
I         初期設定:DATA または拡張宣言
E         EQUIVALENCE での出現
N         NAMELIST での出現
L         モジュールを使用します
```

```
相互参照                  (水)  5月  8 11:35:19 2002                ページ 3

プ ロ グ ラ ム 形 式

 プログラム
 -------

repeat          <repeat>        D      1:D

相互参照                  (水)  5月  8 11:35:19 2002                ページ 4


 関数とサブルーチン
 ------------------------

INT      intrinsic
                <prnok>         C     23:C

newf            <repeat>        C      4:C
                <newf>          D     14:D

prnok    int*4  <nwfrk>        DC     15:D     19:C
                <prnok>        DM     22:D     23:M

sleep           <unreach_sub>            C     27:C

subr1           <repeat>        C      3:C
                <subr1>         D      8:D
                <subr2>         C     31:C

subr2           <subr1>         C     10:C
                <subr2>         D     30:D

unreach_sub     <unreach_sub>            D     26:D

相互参照                  (水)  5月  8 11:35:19 2002                ページ 5


 変数と配列
 --------------------

ix       int*4   仮
                <newf>      DUMA    14:D   16:U    17:M    19:A

pn1       real*4 <repeat>   UMA      2:M     3:A      4:A      5:U

x        real*4 仮
                <subr1>       DU      8:D      9:U     10:U
                <subr2>       DU     30:D     31:U     31:U
                <prnok>       DA     22:D     23:A

------------------------------------------------------------------------------


統計情報                    (水)  5月  8 11:35:19 2002                  ページ 6


日付:           (水)  5月  8 11:35:19 2002
オプション:     -XlistX
ファイル:       2 個 (ソース: 1 個、ライブラリ: 1 個)
行:             32 行 (ソース: 32 個、ライブラリ副プログラム： 1 個)
ルーチン:       6 個 (MAIN: 1 個、サブルーチン:  4 個、関数: 1 個)
メッセージ:     6 個 (エラー:  3 個、警告: 3 個)
```

ルーチン間の大域的な検査を行うサブオプション

大域的にクロスチェックする標準的なオプションは (サブオプションなしの) -Xlist です。このオプションは、それぞれが個別に指定できるサブオプションの組み合わせです。

以降に、リスト、エラー、クロスリファレンステーブルを生成するオプションを説明します。複数のサブオプションをコマンド行に指定することもできます。

サブオプションの構文

サブオプションは次の規則に従って追加します。

- -Xlist にサブオプションを追加します。
- -Xlist とサブオプションの間には空白を置きません。
- 1 つの -Xlist に指定できるサブオプションは 1 つだけです。

-Xlist とサブオプション

サブオプションの組み合せは次の規則に従います。

- 最も一般的なオプションは、-Xlist です (リスト、エラー、クロスリファレンステーブル)。
- 特定の機能は、-Xlistc、-XlistE、-XlistL、-XlistX を組み合わせて使用することによって指定できます。
- これら以外のオプションは、細部指定オプションです。

例 :次の 2 つのコマンド行は同じ結果を生成します。

```
demo% f95  -Xlistc  -Xlist  any.f
```

```
demo% f95  -Xlistc  any.f
```

次の表に、これらの基本的な -Xlist サブオプションだけで生成したレポートを示します。

表 5-1　Xlist の基本的なサブオプション

生成されるレポート	オプション
エラー、リスト、クロスリファレンス	-Xlist
エラーのみ	-Xlist**E**
エラーとソースリストのみ	-Xlist**L**
エラーとクロスリファレンステーブルのみ	-Xlist**X**
エラーとコールグラフのみ	-Xlist**c**

次に、-Xlist のすべてのサブオプションを示します。

表 5-2　-Xlist サブオプションの全リスト

オプション	動作
-Xlist (サブオプションなし)	エラー、リスト、クロステーブルを表示する
-Xlist**c**	コールグラフとエラーを表示する -Xlistc は単独ではリストまたはクロスリファレンスを表示しません。コールグラフは印字可能な文字を使用したツリー形式で生成されます。主プログラムから呼び出されないサブルーチンがあれば、複数のコールグラフが表示されます。各初期値設定プログラムは主プログラムとは切り離して別個に出力されます。 デフォルトではコールグラフは出力されません。
-Xlist**E**	エラーを表示する -XlistE は単独ではクロスルーチンエラーだけを表示し、リストまたはクロスリファレンスを表示しません。

表 5-2　-Xlist サブオプションの全リスト (続き)

オプション	動作
-Xlist**err**[*nnn*]	検証レポートから *nnn* 番のエラーを削除する リストやクロスリファレンスから番号付きのエラーメッセージを抑制するときに使用します。 たとえば、次のようにします。-Xlisterr338 はエラーメッセージ 338 を抑制します。他の特定のエラーを抑制する場合は、このオプションを繰り返し使用してください。*nnn* が指定されていない場合は、すべてのエラーメッセージが抑制されます。
-Xlist**f**	出力を高速化する 完全なコンパイルを行わずに、ソースファイルのリストとクロスチェックレポートを生成してソースを検査するには、-Xlistf を使用します。
-Xlist**h**	クロスチェックのエラーの場合、コンパイルを停止する -Xlisthを使用すると、プログラムのクロスチェック中にエラーが検出された場合に、コンパイルが停止します。この場合の記録は、*.lst ファイルではなく標準出力 stdout にリダイレクトされます。
-Xlist**I**	include ファイルのリストとクロスチェック -XlistI サブオプションだけを使用した場合、標準の -Xlist 出力 (行番号付きリスト、エラーメッセージ、クロスリファレンステーブル) とともに、include ファイルも表示または走査されます。 リスト - リストが抑制されていない場合は、include ファイルは所定の場所でリストされます。このため、インクルードされるたびに何回でもファイルがリストされることになります。リストされるファイルは、ソースファイル、#include ファイル、INCLUDE ファイルです。 クロスリファレンステーブル - クロスリファレンステーブルが抑制されていない場合は、クロスリファレンステーブルの生成中に次のファイルが走査されます。ソースファイル、#include ファイル、INCLUDE ファイルです。 デフォルトでは、include ファイルは表示されません。

表 5-2　-Xlist サブオプションの全リスト (続き)

オプション	動作
-Xlist**L**	リストとエラーを表示する リストとクロスルーチンエラーのみを生成するときに -XlistL を使用します。このサブオプションは単独ではクロスリファレンステーブルを表示しません。デフォルトでは、リストとクロスリファレンステーブルの両方が表示されます。
-Xlist**l***n*	改ページを設定する ページの長さをデフォルトのページサイズ以外の長さに設定するときに -Xlistl を使用します。たとえば、-Xlistl45 とすると、1 ページの長さは 45 行になります。デフォルトは 66 行です。 *n*=0 (-Xlistl0) の場合、このオプションは、改ページをせずにリストとクロスリファレンステーブルを表示します。これは、画面上で表示するときに便利です。
-Xlist**MP**	OpenMP 指令の整合性を検査する ソースコードファイル内に指定された OpenMP 指令間の非整合性を報告するときに -XlistMP を使用します。-XlistMP によって発行される診断についての詳細は、『OpenMP API ユーザーズガイド』を参照してください。
-Xlisto *name*	-Xlist 出力報告ファイルのリネーム -Xlisto を使用して、生成されたレポート出力ファイルの名前を変更します。o と *name* の間には空白文字が必要です。 -Xlisto name と指定すると、出力は name.list ファイルに書き込まれます。 画面に直接表示するときは -Xlisto /dev/tty コマンドを使用します。
-Xlist**s**	クロスリファレンステーブルから参照されない識別子を削除する include ファイルで定義されているが、ソースファイルで参照されていない識別子を、クロスリファレンステーブルから抑制します。 このサブオプションは、-XlistI が指定されている場合には効力がありません。 デフォルトでは、#include または INCLUDE ファイルでの出現は表示されません。

表 5-2　-Xlist サブオプションの全リスト (続き)

オプション	動作
-Xlist**v***n*	検査の「厳密度」を設定する *n* には 1、2、3、4 のいずれかを設定します。デフォルトは 2 です (-Xlistv2)。 • -Xlistv1 すべての名前についてクロスチェックした情報を行番号のない、簡潔な形式でのみ表示します。検査の厳密度としてはもっとも低いレベルで、構文エラーだけを検査します。 • -Xlistv2 クロスチェックした情報に、注釈と行番号を付けて表示します。検査の厳密度としてはデフォルトのレベルで、構文エラーに加えて、引数の不整合なエラー、変数の使用上のエラーも検査します。 • -Xlistv3 クロスチェックした情報に注釈と行番号を付けて表示し、共通ブロックのマップを表示します。検査の厳密度としては高いレベルで、別の副プログラムにある共通ブロックでデータ型を不正に使用したことによるエラーも検査します。 • -Xlistv4 クロスチェックした情報に、注釈、行番号、共通ブロックのマップ、EQUIVALENCE ブロックのマップを付けて表示します。最高の検査の厳密度で、最大限のエラーを検出します。
-Xlist**w**[*nnn*]	出力行の幅を設定する 出力行の幅を設定するときに -Xlistw を使用します。たとえば、-Xlistw132 とすると、ページ幅は 132 カラムになります。デフォルトは 79 カラムです。
-Xlist**war**[*nnn*]	レポートから nnn 番の警告を削除する 出力レポートから特定の警告メッセージを抑制するときに -Xlistwar を使用します。*nnn* が指定されていない場合は、すべての警告メッセージが出力から抑制されます。たとえば、-Xlistwar338 とすると、338 番の警告メッセージが抑制されます。すべてではない複数の警告を対象にするときは、このオプションを繰り返して指定します。
-Xlist**X**	クロスリファレンステーブルとエラーだけを表示する -XlistX は、クロスリファレンステーブルとクロスルーチンエラーリストは生成しますが、ソースリストは生成しません。

特別なコンパイラオプション

デバッグに便利なコンパイルオプションもあります。これらのオプションによって、添字の検査、未宣言変数の印付け、コンパイルとリンク処理の経過の表示、ソフトウェアのバージョンの表示などを行います。

Solaris リンカーには、新しいリンカーデバッグ支援オプションがあります。ld(1) のマニュアルページを参照するか、シェルプロンプトでコマンド `ld -Dhelp` を実行してオンラインマニュアルを表示してください。

添字の境界 (-C)

`-C` を付けてコンパイルする場合は、コンパイラは、実行時に境界を超えている各配列の添字への参照を検査します。このオプションは、セグメンテーションフォルトの原因を見つけるときに役立ちます。

例 :範囲外の索引

```
demo% cat range.f
      REAL a(10,10)
      k = 11
      a(k,2) = 1.0
      END
demo% f95 -o range range.f
demo% range

  ****** Fortran RUN-TIME SYSTEM******
添字が上下限を超えています。位置: 'range.f' 3 行め  9 桁め
添字番号 1 の値は 11 （配列 'A' 中）です。
異常終了
demo%
```

未宣言の変数型 (-u)

`-u` オプションは、未宣言の変数を検査します。

-u オプションは、最初、すべての変数を未宣言として扱います。したがって、型宣言文や IMPLICIT 文で明示的に宣言されていない変数はすべてエラーとなります。-u オプションは、名前の入力を間違えた変数の発見に役立ちます。-u を設定すると、すべての変数は、明示的に宣言されるまで未宣言として扱われます。未宣言変数を使用している箇所については、エラーメッセージが表示されます。

コンパイラのバージョンのチェック (-V)

-V オプションは、コンパイラのさまざまなフェーズの名前とバージョン ID を表示できます。このオプションは、不明確なエラーメッセージの原因を追跡したり、コンパイラの失敗をレポートするのに役立ちます。また、インストールされたコンパイラパッチのレベルを検証するためにも使用できます。

```
demo% f95 -V wh.f
f95: Forte Developer 7 Fortran 95 7.0 DEV 2002/01/30
f90comp: Forte Developer 7 Fortran 95 7.0 DEV 2002/01/30
f90comp: 9 ソース行
f90comp: 0 個のエラー、0 個の警告、0 個の他のメッセージ、0 個の ANSI
ld: Software Generation Utilities - Solaris-ELF (4.0)
```

dbx によるデバッグ

Forte Developer は、Fortran、C、C++ で書かれたアプリケーションをデバッグするための密接に統合された環境を提供します。

dbx プログラムは、イベント管理、プロセス制御、データ検査を提供します。プログラムの実行中になにが起こっているかを表示でき、次の作業を行うことができます。

- ルーチンを修正した後、他のルーチンをコンパイルし直さずに実行を継続できます。
- ウォッチポイントを設定して、指定した項目が変更された場合に停止または追跡できます。
- パフォーマンス調整のためのデータを収集できます。
- 変数、構造体、配列を監視できます。
- 行単位、または関数単位でブレークポイント (プログラム中で停止する場所) を設定できます。

- 値を表示できます。つまり、停止して、変数、配列、構造体を表示または変更できます。
- ソース行またはアセンブリ行ごとにプログラムをステップ実行できます。
- プログラムの流れを追跡できます。つまり、行われた一連の呼び出しを表示できます。
- デバッグすべきプログラム中の手続きを呼び出すことができます。
- 関数呼び出しを通り過ぎたり、関数呼び出しに入り込み、そこで 1 行ずつ進んだり、関数呼び出しから抜けたりできます。
- 次の行、または他の行で、実行、停止、継続ができます。
- デバッグの実行のすべてまたは一部を保存したり再生したりできます。
- 呼び出しスタックを検査したり、コールスタックを上下に移動したりできます。
- 埋め込み Korn シェル中でスクリプトをプログラムできます。
- プログラムが `fork`(2) と `exec`(2) を実行すると、それを追跡します。

最適化されたプログラムをデバッグするには、`dbx fix` コマンドを使用して、デバッグしたいルーチンをコンパイルし直します。

1. 適切な `-O`*n* 最適化レベルでプログラムをコンパイルします。
2. `dbx` の制御下で実行します。
3. `fix -g` *any*`.f` を使用します。デバッグしたいルーチンには最適化を行いません。
4. コンパイルしたルーチンで `continue` を使用します。

   コマンド行に `-g` オプションがある場合、一部の最適化機能が制限されます。詳細は、『dbx コマンドによるデバッグ』を参照してください。

   詳細は、Forte Developer の『dbx コマンドによるデバッグ』、および `dbx`(1) のマニュアルページを参照してください。

第6章

浮動小数点演算

この章では、浮動小数点演算について説明し、数値計算の誤差を回避および検出するための方針を示します。

SPARC プロセッサ上の浮動小数点計算についての詳細は、Forte Developer の『数値計算ガイド』を参照してください。

はじめに

SPARC プロセッサ上での Forte Developer Fortran 95 の浮動小数点環境は、『IEEE Standard 754 for Binary Floating Point Arithmetic』で指定された演算モデルを実装しています。この環境では、安定した、高性能の、移植可能な数値計算アプリケーションを開発できます。また、数値計算プログラムによる異常な動作を調査するためのツールも提供します。

数値計算プログラムでは、計算誤差を引き起こす次のような潜在的な原因がたくさんあります。

- 計算モデルが間違っている
- 使用されているアルゴリズムが数値的に不安定である
- データが正確でない
- ハードウェアが予測できない結果を生み出す

間違った数値計算の誤差の原因を見つけることはたいへん困難です。市販の検査済みライブラリパッケージを使用することで、コーディングの誤りの可能性を減らすことはできます。アルゴリズムを選択することも重要な問題です。また、コンピュータの特性を考慮した優れた算術演算法を使用することも重要です。

この章では、数値誤差の解析については説明していません。ここでは、Forte Developer Fortran 95 によって実装された IEEE 浮動小数点モデルを紹介します。

IEEE 浮動小数点演算

IEEE 演算は、無効演算、ゼロ除算、オーバーフロー、アンダーフロー、結果不正確などの問題を発生させる算術演算を取り扱う、比較的新しい方式です。丸め、ゼロに近い数の扱い方、その機種で取り扱える最大数に近い数の扱い方に違いがあります。

IEEE 規格は、例外、丸め、精度のユーザー処理をサポートします。その結果、IEEE 規格は、区間演算と変則性の診断をサポートします。IEEE 規格 754 は、*exp* や *cos* などの基本関数の標準化、高精度の演算、数値計算と記号代数計算の結合を実現します。

IEEE 演算では、他の浮動小数点演算と比べると、ユーザーが計算を大きく制御できます。IEEE 規格は、数値計算的に洗練された移植性のあるプログラムを作成する作業を簡単にします。浮動小数点演算についての多くの質問は、基本的な数の演算に関連します。たとえば、次のとおりです。

- 無限に正確な結果をコンピュータのハードウェア内で表現できない場合、演算結果はどうなるか
- 乗算や加算のような基本演算は可換か

もう 1 つのクラスの質問は、浮動小数点例外や例外処理に関連するものです。 たとえば、次のとおりです。

- 同符号の 2 つの巨大な数を乗算するとどうなるか
- ゼロ以外の値をゼロで除算するとどうなるか
- ゼロをゼロで除算するとどうなるか

旧式の演算モデルでは、最初のクラスの質問は期待された答えにならず、2 番目のクラスの例外ケースはすべて同じ結果になります。つまり、プログラムは当該箇所で異常終了するか、演算は続行されるが結果は意味のないものとなります。

IEEE 規格は、演算が数学的に期待される結果を、期待される特性で出すことを保証します。また、ユーザーが特に他を選択しない限り、例外ケースが指定された結果を出すことも保証します。

たとえば、例外値 +Inf、-Inf、NaN は次のように直感的に理解できます。

```
big*big = +Inf       正の無限大
big*(-big) = -Inf    負の無限大
num/0.0 = +Inf       num > 0.0  のとき
num/0.0 = -Inf       num < 0.0  のとき
0.0/0.0 = NaN        非数
```

また、次のような 5 つの種類の浮動小数点例外も発生します。

- 無効演算 －数学的に定義できない演算。0.0/0.0、sqrt(-1.0)、log(-37.8) など。
- ゼロ除算 －除数がゼロで、被除数が有限かつゼロでない数。9.9/0.0 など。
- オーバーフロー －指数の範囲を超える結果を出す演算。MAXDOUBLE+0.0000000000001e308 など。
- アンダーフロー －通常の数として表現できないほど小さな結果を出す演算。MINDOUBLE * MINDOUBLE など。
- 結果不正確 － 無限に正確に表現できない結果を出す演算。2.0/3.0、log(1.1)、0.1 の入力など。

IEEE 規格の実装については、Forte Developer の『数値計算ガイド』を参照してください。

`-ftrap=`*mode* コンパイラオプション

`-ftrap=`*mode* オプションは、浮動小数点例外用のトラップを有効にします。`ieee_handler()` 呼び出しによってシグナルハンドラが設定されていない場合、例外はプログラムを終了し、メモリーダンプコアファイルを作成します。このコンパイラオプションに関する詳細は、『*Fortran* ユーザーズガイド』を参照してください。たとえば、オーバーフロー、ゼロ除算、無効演算のトラップを有効にするには、`-ftrap=common-` を付けてコンパイルします。これが `f95` のデフォルトです。

注 – トラップを有効にするには、アプリケーションの主プログラムを `-ftrap=` を付けてコンパイルする必要があります。

浮動小数点演算の例外

f95 プログラムは例外を自動的に報告しません。プログラムの終了時に、発生した浮動小数点演算の例外リストを表示するには、ieee_retrospective(3M) を明示的に呼び出す必要があります。一般的には、無効演算、ゼロ除算、オーバーフローのいずれかの例外が発生すると、メッセージが表示されます。実際のプログラムでは結果不正確の例外は多く発生するため、結果不正確の例外のメッセージは表示されません。

発生した例外の通知

ieee_retrospective 関数は、浮動小数点のステータスレジスタを調べて、どの例外が発生したのかを突き止め、標準エラーにメッセージを表示し、どの例外が発生してクリアーされていないのかをプログラマに知らせます。メッセージは通常、次のようになります (リリースによって若干異なります)。

```
Note:IEEE floating-point exception flags raised:
    Division by Zero;
IEEE floating-point exception traps enabled:
    inexact;  underflow;  overflow;  invalid operation;
See the Numerical Computation Guide, ieee_flags(3M),
    ieee_handler(3M)
```

Fortran 95 プログラムでは、ieee_retrospective を明示的に呼び出し、-xlang=f77 を使用してコンパイルして、f77 互換性ライブラリを使用してリンクする必要があります。

-f77 互換性フラグを使用してコンパイルすると、プログラムの終了時に自動的に ieee_retrospective を呼び出す FORTRAN 77 の規則が有効になります。

ieee_retrospective を呼び出す前に例外ステータスフラグをクリアーすると、ieee_flags() を使用して、メッセージの一部また全部を表示させないようにすることができます。

例外処理

IEEE 規格準拠の例外処理は、SPARC および x86 プロセッサ上ではデフォルトで行われます。ただし、浮動小数点例外の検出と、浮動小数点例外に対するシグナル (SIGFPE) の生成には、違いがあります。

浮動小数点演算中にトラップしていない例外が発生すると、IEEE 規格に従って、次の 2 つの処理が行われます。

- システムはデフォルトの結果を返します。たとえば、0/0 (演算不可能) の場合は NaN を結果として返します。
- 例外が発生したことを示すフラグが設定されます。たとえば、0/0 (演算不可能) の場合は「無効演算」のフラグが設定されます。

浮動小数点演算の例外をトラップする

浮動小数点演算の例外を処理する方法は、`f95` と以前の `f77` では大きく異なります。

`f95` のデフォルトでは、ゼロ除算、オーバーフロー、無効演算の場合に自動的にトラップします。`f77` のデフォルトでは、浮動小数点演算例外において実行プログラムを中断するためのシグナルを自動的に生成しません。これは、トラップはパフォーマンスを低下させるということと、期待された値が戻ってくれば、ほとんどの例外は重要でないという前提に基づいていました。

`f95` のコマンド行オプション `-ftrap` を使用すると、このデフォルトを変更できます。`f95` のデフォルトは -ftrap=common です。以前の `f77` のデフォルトに従うには、`-ftrap=%none` を使用して主プログラムをコンパイルします。

非標準の算術演算

段階的なアンダーフローと呼ばれる、標準の IEEE 算術演算の 1 つは手作業で無効にできます。無効にしたとき、プログラムは非標準の算術演算で実行していると考えられます。

算術演算に関する IEEE 規格では、アンダーフローとなった結果は、有効桁の小数部の基点を動的に調整することにより、段階的に扱うように指定しています。IEEE の浮動小数点の形式では、有効桁の前に小数点が現れ、暗黙的な 1 の先行ビットがあります。 浮動小数点の演算結果がアンダーフローとなったときに、段階的なアンダーフローでは、暗黙的な先行ビットをゼロにクリアーし、小数点を有効桁方向にシフトさせるようになっています。これは、SPARC プロセッサではハードウェアではなく、ソフトウェアで行われます。このため、プログラムにアンダーフローが多数発生すると (アルゴリズムに問題があることを示す可能性があります)、パフォーマンスが低下することになります。

段階的なアンダーフローを無効にするには、-fns オプションを付けてコンパイルします。 または、プログラムの中からライブラリルーチン nonstandard_arithmetic() を呼び出して、段階的なアンダーフローを無効にします。standard_arithmetic() を呼び出して、段階的なアンダーフローを有効に戻します。

注 – 有効にするためには、アプリケーションの主プログラムを -fns を使用してコンパイルする必要があります。『Fortran ユーザーズガイド』を参照してください。

古いアプリケーションの場合、次のことに注意してください。

- standard_arithmetic() サブルーチンは、以前の gradual_underflow() という名前のルーチンに対応しています。
- nonstandard_arithmetic() サブルーチンは、以前の abrupt_underflow() という名前のルーチンに対応しています。

注 – -fns オプションと nonstandard_arithmetic() ライブラリルーチンは、一部の SPARC システム上だけで有効です。

IEEE ルーチン

次のインタフェースは、IEEE 演算を使用するユーザーの役に立ちます。これらのほとんどは、数学ライブラリ libsunmath と、いくつかの .h ファイルに置かれています。

- ieee_flags(3m) - 丸めの方向と丸めの精度を制御し、例外ステータスの問い合わせと例外ステータスのクリアーを行います。
- ieee_handler(3m) - 例外ハンドラルーチンを設定します。
- ieee_functions(3m) - 個々の IEEE 関数の名前と目的をリストします。
- ieee_values(3m) - 特別な値を返す関数をリストします。
- その他の libm 関数 (本節で説明)

- ieee_retrospective
- nonstandard_arithmetic
- standard_arithmetic

最新の SPARC プロセッサには浮動小数点ユニットが含まれており、整数の乗算と除算の命令とハードウェアによる平方根演算機能を備えています。

コンパイルしたコードが実行時の浮動小数点ハードウェアに適切に一致したとき、最高のパフォーマンスが得られます。コンパイラの -xtarget= オプションは、実行時ハードウェアの指定を許可します。たとえば、-xtarget=ultra は、UltraSPARC プロセッサ上で最高のパフォーマンスを得られるオブジェクトコードを生成することをコンパイラに伝えます。

fpversion ユーティリティは、浮動小数点のどのハードウェアがインストールされているかを表示し、指定すべき適切な -xtarget 値を示します。このユーティリティは、すべての Sun SPARC アーキテクチャ上で動作します。詳細は、fpversion(1) のマニュアルページ、『Fortran ユーザーズガイド』、『数値計算ガイド』を参照してください。

フラグと ieee_flags()

ieee_flags 関数は、例外ステータスフラグの問い合わせやクリアーに使用します。この関数は、Sun コンパイラとともに出荷される libsunmath ライブラリに組み込まれていて、次の作業を行うことができます。

- 丸めの方向と丸めの精度の制御
- 例外ステータスフラグの検査
- 例外ステータスフラグのクリアー

ieee_flags の一般的な呼び出し方法は次のとおりです。

```
flags = ieee_flags( action, mode, in, out )
```

4 つの引数はすべて文字列です。入力は、*action*、*mode*、および *in* です。出力は、*out* と *flags* です。ieee_flags は、整数値の関数です。*flags* は 1 ビットフラグの集合で、ここには有用な情報が返されます。詳細は、ieee_flags(3m) のマニュアルページを参照してください。

パラメータの取り得る値は次のとおりです。

表 6-1　ieee_flags (*action, mode, in, out*) の引数の値

引数	使用可能な値
action	get, set, clear, clearall
mode	direction, exception
In, out	nearest, tozero, negative, positive, extended, double single, inexact, division, underflow, overflow, invalid all, common

これらはリテラル文字列であること、出力パラメータ *out* は最低でも CHARACTER*9 でなければならないことに注目してください。*in* と *out* の取り得る値の意味は、使用時の動作とモードによって異なります。これらの関係を次の表に要約します。

表 6-2　ieee_flags の 引数 *in* と *out* の意味

in と *out* の値	意味
nearest, tozero, negative, positive	丸めの方向
extended, double, single	丸めの精度
inexact, division, underflow, overflow, invalid	例外
all	5 種類すべての例外
common	3 種類の一般的な例外。 invalid (無効演算)、 division (ゼロ除算)、 overflow (オーバーフロー)

たとえば、フラグが立てられている例外のうちで何が最も優先順位が高いかを判別するときは、引数の *in* にヌル文字列を渡します。

```
      CHARACTER *9, out
      ieeer = ieee_flags( 'get', 'exception', '', out )
      PRINT *, out, ' flag raised'
```

また、overflow 例外フラグが立てられているかどうかを判別するときは、引数の *in* に overflow を設定します。復帰時に、*out* が overflow になっていれば、overflow 例外のフラグが立てられているということです。そうでない場合は、その例外は発生していません。

```
      ieeer = ieee_flags( 'get', 'exception', 'overflow', out )
      IF ( out.eq. 'overflow') PRINT *,'overflow flag raised'
```

例 :invalid 例外をクリアーします。

```
      ieeer = ieee_flags( 'clear', 'exception', 'invalid', out )
```

例 :すべての例外をクリアーします。

```
      ieeer = ieee_flags( 'clear', 'exception', 'all', out )
```

例 :ゼロに近づけるように丸めます。

```
      ieeer = ieee_flags( 'set', 'direction', 'tozero', out )
```

例 :丸めの精度を double に設定します。

```
      ieeer = ieee_flags( 'set', 'precision', 'double', out )
```

ieee_flags を使用して警告メッセージを抑制する

次の例のように、動作 clear で ieee_flags を呼び出すと、クリアーされてない例外すべてがリセットされます。プログラムが終了する前にこの呼び出しを置くと、プログラム終了時の浮動小数点例外に関するシステム警告メッセージを抑制します。

例 :ieee_flags() を使用して、発生していたすべての例外をクリアーします。

```
      i = ieee_flags('clear', 'exception', 'all', out )
```

ieee_flags を使用して例外を検出する

次の例は、前の計算によってどの浮動小数点例外フラグが立ったかを決定する方法を示しています。システム include ファイル floatingpoint.h に定義されたビットマスクは、ieee_flags により返された値に適用されます。

この例 DetExcFlg.F では、インクルードファイルは #include 前処理部指令を使用して導入されます。この指令には、.F 接尾辞のソースファイルを指定しなければなりません。アンダーフローの原因は、最小の倍精度数を 2 で除算しているためです。

例 : ieee_flags を使用して例外を検出し、復号化します。

```
#include "floatingpoint.h"
         CHARACTER*16 out
         DOUBLE PRECISION d_max_subnormal, x
         INTEGER div, flgs, inv, inx, over, under

      x = d_max_subnormal() / 2.0              !アンダーフローを発生

       flgs=ieee_flags('get','exception','',out) !どの例外が発生したのか ?

       inx   = and(rshift(flgs, fp_inexact)   , 1)   ! ieee_flags
       div   = and(rshift(flgs, fp_division) , 1)    ! によって
       under = and(rshift(flgs, fp_underflow), 1)     ! 返された
       over  = and(rshift(flgs, fp_overflow) , 1)      ! 値を
      inv   = and(rshift(flgs, fp_invalid)  , 1)       ! 復号化する

       PRINT *, "Highest priority exception is:", out
       PRINT *, ' invalid  divide  overflo underflo inexact'
       PRINT '(5i8)', inv, div, over, under, inx
       PRINT *, '(1 = exception is raised; 0 = it is not)'
      i = ieee_flags('clear', 'exception', 'all', out) !すべてクリアー
       END
```

例 :前出の例 (DetExcFlg.F) をコンパイルして実行します。

```
demo% f95 DetExcFlg.F
demo% a.out
 Highest priority exception is:underflow
  invalid  divide  overflo underflo inexact
       0       0       0       1       1
 (1 = exception is raised; 0 = it is not)
demo%
```

IEEE 極値関数

コンパイラは、特別な IEEE 極値を返すために呼び出すことができる関数のセットを提供します。無限大や最小の正規数などの値は、アプリケーションプログラムの中で直接使用できます。

例 :収束のテストはハードウェアによってサポートされる最小数に基づき、次のようになります。

```
      IF ( delta .LE. r_min_normal() ) RETURN
```

利用できる値を次の表にリストします。

表 6-3　IEEE の値を使用する関数

IEEE の値	倍精度	単精度
infinity （無限大）	d_infinity()	r_infinity()
quiet NaN （シグナルを発しない非数）	d_quiet_nan()	r_quiet_nan()
signaling NaN （シグナルを発する非数）	d_signaling_nan()	r_signaling_nan()
min normal （最小の正規数）	d_min_normal()	r_min_normal()

表 6-3　IEEE の値を使用する関数

IEEE の値	倍精度	単精度
min subnormal (最小の非正規数)	d_min_subnormal()	r_min_subnormal()
max subnormal (最大の非正規数)	d_max_subnormal()	r_max_subnormal()
max normal (最大の正規数)	d_max_normal()	r_max_normal()

2 つの NaN 値 (quiet と signaling) は順序がないものなので、IF(X.ne.r_quiet_nan())THEN... のように比較の中で使用してはなりません。値が NaN であるかどうかを判別するときは、ir_isnan(r) か id_isnan(d) 関数を使用します。

これらの関数の Fortran 名は、次のマニュアルページにリストされています。

- libm_double(3f)
- libm_single(3f)
- ieee_functions(3m)

また、次も参照してください。

- ieee_values(3m)
- The floatingpoint.h ヘッダーファイルと floatingpoint(3f)

例外ハンドラと ieee_handler()

通常、IEEE 例外について、次のことを知っておく必要があります。

- 例外が発生するときのシステムの動作。
- ieee_handler() を使用して、ユーザー関数を例外ハンドラとして設定する方法。
- 例外ハンドラとして使用できる関数を作成する方法。
- 例外の発生場所を調べる方法。

ユーザールーチンへの例外トラップは、システムの浮動小数点例外に対するシグナルの生成から始まります。「浮動小数点例外のシグナル」を表す UNIX の公式名は SIGFPE です。SPARC プラットフォームは、デフォルトでは、例外が発生しても

SIGFPE を生成しません。システムが SIGFPE を生成するためには、まず、例外トラップを有効にしなければなりません (これは通常は、ieee_handler() への呼び出しによって行います)。

例外ハンドラ関数を設定する

例外ハンドラとして関数を設定するときは、監視したい例外や対応動作といっしょに、関数の名前を ieee_handler() に渡します。ハンドラを一度設定すると、特定の浮動小数点の例外が発生するたびに、SIGFPE シグナルが生成され、指定した関数が呼び出されます。

ieee_handler() を起動する形式を次の表に示します。

表 6-4　ieee_handler(*action*, *exception*, *handler*) の引数

引数	種類	可能な値
action	文字列	get、set、clear のいずれか
exception	文字列	invalid、division、overflow、underflow、inexact のいずれか
handler	関数名	ユーザーハンドラ関数の名前、または、SIGFPE_DEFAULT、SIGFPE_IGNORE、SIGFPE_ABORT のいずれか
戻り値	integer	0 =*OK*

f95 でコンパイルする Fortran 95 ルーチンで ieee_handler() を呼び出す場合は、次の宣言も必要です。

```
#include 'floatingpoint.h'
```

特別な引数 SIGFPE_DEFAULT、SIGFPE_IGNORE、および SIGFPE_ABORT は、これらのインクルードファイルで定義され、特定の例外に対するプログラムの動作を変更するのに使用できます。

SIGFPE_DEFAULT または SIGFPE_IGNORE	指定した例外が発生しても何も動作しない。
SIGFPE_ABORT	例外発生時にはプログラムが異常終了し、恐らくダンプファイルを生成する。

ユーザー例外ハンドラ関数を作成する

ユーザーの例外ハンドラが行う動作はユーザーが自由に設定できます。しかし、ルーチンは、次に示す 3 つの引数を取る、整数型関数でなければなりません。

handler_name(sig, sip, uap)

- *handler_name* は整数関数の名前です。
- sig は整数です。
- sip は構造体 siginfo を持つ記録です。
- uap は使用されません。

例 :例外ハンドラ関数

```
      INTEGER FUNCTION hand( sig, sip, uap )
      INTEGER sig, location
      STRUCTURE /fault/
           INTEGER address
           INTEGER trapno
      END STRUCTURE
      STRUCTURE /siginfo/
           INTEGER si_signo
           INTEGER si_code
           INTEGER si_errno
           RECORD /fault/ fault
      END STRUCTURE
      RECORD /siginfo/ sip
      location = sip.fault.address
      ... ... ユーザーの行う処理 ...
      END
```

STRUCTURE 内のすべての INTEGER 宣言を INTEGER*8 で置き換えて、この例を、SPARC V9 アーキテクチャ (-xarch=v9 または v9a) で実行できるよう変更する必要があります。

ieee_handler() によって有効にされるハンドラルーチンが例で示すように Fortran で書かれている場合、ハンドラルーチンは 1 番目の引数 (sig) に対しては、一切の参照を行ってはなりません。1 番目の引数は値でルーチンに渡され、loc(sig) としてのみ参照できます。この値はシグナル番号です。

ハンドラを使用して例外を検出する

次の例では、浮動小数点の例外を検出するためのハンドラルーチンを作成する方法を示します。

例 : 例外を検出して、異常終了します。

```
demo% cat DetExcHan.f
      EXTERNAL myhandler
      REAL ::r = 14.2 , s = 0.0
      i = ieee_handler ('set', 'division', myhandler )
      t = r/s
      END

      INTEGER FUNCTION myhandler(sig,code,context)
      INTEGER sig, code, context(5)
      CALL abort()
      END
demo% f95 DetExcHan.f
demo% a.out
異常終了
demo%
```

SIGFPE は、浮動小数点演算の例外が発生すると必ず生成されます。SIGFPE が検出されると、制御が myhandler 関数に渡され、即座に異常終了します。-g を付けてコンパイルし、dbx を使用すると、例外箇所を突き止めることができます。

ハンドラを使用して例外箇所を突き止める

例 : 例外箇所を突き止め (アドレスを出力して)、異常終了します。

```
demo% cat LocExcHan.F
#include "floatingpoint.h"
      EXTERNAL Exhandler
      INTEGER Exhandler, i, ieee_handler
      REAL:: r = 14.2 ,  s = 0.0 , t
C  ゼロ除算の検出
      i = ieee_handler( 'set', 'division', Exhandler )
      t = r/s
      END

      INTEGER FUNCTION Exhandler( sig, sip, uap)
      INTEGER sig
      STRUCTURE /fault/
                  INTEGER address
      END STRUCTURE
      STRUCTURE /siginfo/
                  INTEGER si_signo
                  INTEGER si_code
                  INTEGER si_errno
                  RECORD /fault/ fault
      END STRUCTURE
      RECORD /siginfo/ sip
      WRITE (*,10)  sip.si_signo, sip.si_code, sip.fault.address
10    FORMAT('Signal ',i4,' code ',i4,' at hex address ', z8 )
      Exhandler=1
      CALL abort()
      END
demo%  f95 -g LocExcHan.F
demo% a.out
Signal    8 code    3 at hex address    11230
異常終了
demo%
```

SPARC V9 環境では、各 STRUCTURE 内の INTEGER 宣言を INTEGER*8 で置き換えて、書式中の i4 を i8 で置き換えます。この例では、f95 コンパイラへの拡張により、VAX Fortran の STRUCTURE 文を受け付けられるようにしています。

ほとんどの場合、例外の実際のアドレスを知るということは、dbx だけに意味があります。

```
demo% dbx a.out
(dbx) stopi at 0x11230                ブレークポイントをアドレスに設定する
(2) stopi at &MAIN+0x68
(dbx) run                                  プログラムを実行する
実行中:a.out
(プロセス id 18803)
MAIN で 0x10d00 で停止しました
MAIN+0x68:fdivs   %f3, %f2, %f2
(dbx) where                                例外が発生した行番号を表示する
=>[1] MAIN()、"LocExcHan.F" の 7 行目
(dbx) list 7                               ソースコード行を表示する
    7   t = r/s
(dbx) cont              ブレークポイント後、継続してハンドラルーチンに入る
Signal    8 code    3 at hex address    11230
異常終了:abort が呼び出されました
シグナル ABRT (異常終了) 関数 _kill 0xef6e18a4 で
0xef6e18a4:_kill+0x0008:bgeu    _kill+0x30
現関数 :exhandler
    CALL abort()
(dbx) quit
demo%
```

もちろん、エラーの原因となるソース行を決定するためのより簡単な方法があります。しかし、この例は、例外処理の基本を示すのが目的です。

IEEE の例外のデバッグ

どこで例外が発生したかを突き止めるためには、トラップを有効にした例外が必要です。そのためには、-ftrap=common オプション (f95 でコンパイルする場合のデフォルト) を付けてコンパイルするか、ieee_handler() により例外ハンドラルーチンを呼び出します。例外トラップを有効にして、dbx からプログラムを実行し、dbx catch FPE コマンドを使用すれば、どこでエラーが発生するかを見つけることができます。

-ftrap=common を付けてコンパイルすることの利点は、例外をトラップするようにソースコードを変更する必要がないことです。しかし、ieee_handler() を呼び出すことによって、探すべき例外をより限定できます。

例 : コンパイルし直して、dbx を使用します。

```
demo% f95 -g myprogram.f
demo% dbx a.out
a.out の読み込み中
...
(dbx) catch FPE
(dbx) run
実行中:a.out
(プロセス id 19739)
シグナル FPE (ゼロによる浮動小数点除算) 関数 MAIN 行番号 212
ファイル "myprogram.f"
  212               Z = X/Y
(dbx)  print Y
y = 0.0
(dbx)
```

プログラムが終了したときオーバーフローと他の例外が発生していた場合、ieee_handler() を呼び出してオーバーフローだけをトラップすることによって、最初のオーバーフローを突き止めることができます。このためには、次の例に示すように、主プログラムのソースコードを必要最小限だけ修正する必要があります。

例 :他の例外も発生しているときにオーバーフローを突き止めます。

```
demo% cat myprog.F
#include "floatingpoint.h"
        program myprogram
...
      ier = ieee_handler('set','overflow',SIGFPE_ABORT)
...
demo% f95 -g myprog.F
demo% dbx a.out
a.out の読み込み中
...
(dbx) catch FPE
(dbx) run
実行中:a.out
(プロセス id 19793)
シグナル FPE (浮動小数点オーバーフロー) 関数 MAIN 行番号 55  ファイル
"myprog.F"
  55            w = rmax * 200.                  !オーバーフローの原因
(dbx) cont                                      !実行を継続して、終了する
実行完了。終了コードは、0 です
(dbx)
```

例外を選択するため、この例では、#include を導入しています。このため、ソースファイルの名前を .F 接尾辞に変更し、ieee_handler() を呼び出す必要があります。さらに一歩進んで、オーバーフロー例外時に呼び出されるユーザー独自のハンドラ関数を作成し、アプリケーション特有な解析を行い、異常終了する前に中間結果またはデバッグ結果を出力することもできます。

数値に関連したその他の問題

この節では、無効演算、ゼロ除算、オーバーフロー、アンダーフロー、結果不正確の例外を予想外に生成する算術演算に関連する、より実際的な問題について説明します。

たとえば、IEEE 規格より前には、2 つの極めて小さな数をコンピュータで乗算すると、結果はゼロになっていました。メインフレームやミニコンピュータのほとんどが、この方式でした。これに対し、IEEE の算術演算では、段階的なアンダーフローが計算の範囲を動的に拡張します。

たとえば、表現可能な最小値が 1.0E-38 である 32 ビットプロセッサを想定し、2 つの小さな数値を乗算してみます。

```
a = 1.0E-30
b = 1.0E-15
x = a * b
```

旧式の算術演算では結果が 0.0 になりますが、IEEE の算術演算では (同じワード長で) 結果は 1.40130E-45 となります。アンダーフローは、マシンが本来表せる値よりも小さい結果になったということを示します。この結果は、いくつかのビットが仮数部から「盗まれ」、指数部へシフトされたことによってできました。結果 (非正規化数) は、ある場合には精度が低くなりますが、逆に高くなる場合もあります。これに関する詳細な説明はこのマニュアルの範囲外です。 詳細に興味のある方は、1980 年 1 月 に発行された『Computer』誌 (第 13 巻 No.1)、特に J. Coonen 氏の論文「Underflow and the Denormalized Numbers」を参照してください。

科学技術プログラムのほとんどは、方程式を解いたり、行列の因数分解など、丸めに敏感に反応するコードセクションがあります。段階的なアンダーフローがなければ、プログラマは不正確なしきい値への接近を検出する独自の方法を実装します。実装できなければ、独自のアルゴリズムの安定した実装はあきらめるかしかありません。

これらの話題に関する詳細は、Forte Developer の『数値計算ガイド』を参照してください。

単純なアンダーフローを防ぐ

アプリケーションの中には、実際に、非常にゼロに近いところで多くの処理をしているものがあります。これは、誤差や差分補正のアルゴリズムでよく発生します。数値的に安全で最大のパフォーマンスを得るには、重要な演算は拡張精度で行うべきです。アプリケーションが単精度の場合は、重要な演算を倍精度にすればかまいません。

例 :単精度の、簡単なドット積の演算

```
      sum = 0
      DO i = 1, n
         sum = sum + a(i) * b(i)
      END DO
```

`a(i)` と `b(i)` が極めて小さい数であれば、多数のアンダーフローが発生することになります。演算を倍精度に移行すると、ドット積をより正確に計算でき、アンダーフローもなくなります。

```
      DOUBLE PRECISION sum
      DO i = 1, n
         sum = sum + dble(a(i)) * dble(b(i))
      END DO
      result = sum
```

アンダーフローに関して、SPARC プロセッサを旧式のシステムのように動作 (Store Zero) させることができます。 このためには、ライブラリルーチン `nonstandard_arithmetic()` への呼び出しを追加するか、アプリケーションの主プログラムを `-fns` オプションを付けてコンパイルします。

間違った答えのまま継続する

結果が明らかに間違っている場合でも、処理が継続されることに疑問を思われるかもしれません。IEEE の算術演算では、`NaN` や `Inf` など、どの種別の間違った答えを無視できるかをユーザーが指定できます。そして、そのような区別に基づいて決定が行われます。

たとえば、回路シミュレーションを想定してみましょう。特有の 50 行の演算の中で、引数の目的となる重要な変数は電圧量だけであり、この変数のとり得る値は +5v、0、-5v だけであると仮定します。

途中の演算結果が正当な範囲にくるように、演算の各部分を詳細に調整できます。

- 計算された値が *4.0* よりも大きい場合は、*5.0* を返します。
- 計算された値が *-4.0* 以上 *+4.0* 以下の場合は、*0* を返します。
- 計算された値が *-4.0* よりも小さい場合は、*-5.0* を返します。

さらに、`Inf` は許可されない値であるので、大きな数を乗算しないような特別なロジックが必要です。

IEEE 算術演算ではロジックはかなり単純にできます。演算を明確な形式で記述し、最終結果を正しい値に導くだけでかまいません。なぜなら、`Inf` の発生する可能性があり、それは簡単にテストできるからです。

さらに、0/0 の特殊なケースも検出でき、ユーザーの望む形で処理できます。不必要な比較を行わなくてもすむため、結果は読みやすく、実行も高速です。

SPARC:アンダーフローの頻発

2 つの極めて小さな数を乗算すると、結果はアンダーフローとなります。

あらかじめ、乗算 (または減算) の被演算子が小さくなり、アンダーフローしそうであることがわかっていれば、計算を倍精度で行い、その後で結果を単精度に変換します。

たとえば、ドット積ループは次のようになります。

```
real sum, a(maxn), b(maxn)
...
do i =1, n
    sum = sum + a(i)*b(i)
enddo
```

a(*) と b(*) は、小さな要素が入ることがわかっているので、倍精度で実行し、数値の正確性を保持します。

```
real a(maxn), b(maxn)
double sum
...
do i =1, n
    sum = sum + a(i)*dble(b(i))
enddo
```

このようにすることによって、オリジナルのループが原因であるアンダーフロー頻発をソフトウェア的に解決し、パフォーマンスを上げることができます。しかし、これに関しては、確実で手間のかからない方法はありません。 計算が多いコードを使用するユーザー自身の経験のほうが、より適切な解決策を決定することでしょう。

区間演算

Forte Developer Fortran 95 コンパイラの f95 では、組み込みデータタイプとして *intervals* をサポートしています。区間は、 $[a,b] = \{z \mid a \leqq z \leqq b\}$ で表され、1 組の数字は $a \leqq b$ となります。

- 非線形問題を解決します
- 厳密なエラー分析を実行します
- 数値の不安定性のソースを検出します

区間を組み込みデータタイプとして Fortran 95 に導入したので、開発者は Fortran 95 の適用可能な構文や記号をすべてすぐに使用できるようになります。INTERVAL データタイプのほかに、f95 により次の区間拡張機能が Fortran 95 に取り込まれます。

- 3 クラスの INTERVAL 関係演算子
  - Certainly (断定的な関係)
  - possibly (可能性のある関係)
  - Set (集合の関係)
- INF、SUP、WID、HULL などの組み込み INTERVAL 固有の演算子
- 1 数字入出力などの INTERVAL 入出力編集記述子
- 算術関数、三角関数、およびそのほかの数学関数に対する区間拡張機能

- 式コンテキストに依存した INTERVAL 定数
- 混合モードの区間式処理

f95 コマンド行オプション -xinterval により、コンパイラの区間演算機能が使用できるようになります。『Fortran ユーザーズガイド』を参照してください。

Fortran 95 の区間演算の詳細については、『Fortran 95 区間演算プログラミングリファレンス』を参照してください。

第7章

移植

この章では、古い Fortran プログラムを Forte Developer Fortran 95 に移植するときに起きる可能性がある問題について説明します。

Fortran 95 の拡張機能、および FORTRAN 77 と互換性を保つための機能については、『Fortran ユーザーズガイド』で説明しています。

キャリッジ制御

Fortran キャリッジ制御は、Fortran が最初に開発されていたときに使用されていた、機能の限られた装置から発達したものです。同じ歴史上の理由のため、UNIX オペレーティングシステムから派生したオペレーティングシステムには、Fortran のキャリッジ制御がありません。しかし、次の 2 つの方法で Fortran 95 でこの機能をシミュレートできます。

- `asa` フィルタを使用して、Fortran のキャリッジ制御規則を UNIX のキャリッジ制御書式に変換してから (`asa`(1) のマニュアルページを参照してください) `lpr` を使用してファイルを出力してください。
- FORTRAN 77 コンパイラ `f77` では、`OPEN(N, FORM='PRINT')` を使用して、1 行送りや 2 行送り、用紙送り、および 1 カラム目の除去を行っていました。`f95 -f77` とともに `FORM='PRINT'` を使用してプログラムをコンパイルすると、現在でもこの機能を利用できます。`-f77` を使用してコンパイルすると、装置 6 を開き直して `FORM` パラメータを `PRINT` に変更できます。たとえば、次のようにします。

```
      OPEN( 6, FORM='PRINT')
```

このようにして開いたファイルを、lp(1) コマンドを使用して出力できます。

ファイルを扱う

Fortran の初期のシステムは名前付きファイルを使用せず、実際のファイル名と内部装置番号を対応させるコマンド行機構を提供していました。この機能は、標準の UNIX のリダイレクトなど、いくつかの方法でエミュレートできます。

例 :stdin を redir.data からリダイレクトします。 csh(1) を使用した例です。

```
demo% cat redir.data              データファイル
 9 9.9

demo% cat redir.f                 ソースファイル
      read(*,*) i, z              プログラムは標準入力を読み取る
      print *, i, z
      stop
      end

demo% f95 -o redir redir.f    コンパイル
demo% redir < redir.data              リダイレクトでデータファイルを読み
取り
  9 9.90000
demo%
```

科学技術計算用メインフレームから移植する

アプリケーションコードが本来、CRAY や CDC などの 64 ビット (または 60 ビット) メインフレーム用に開発されていた場合、UltraSPARC-II プラットフォームへポーティングしているときに、これらのコードを次のオプションを付けてコンパイルしたい場合があります。

```
-fast -xarch=v9a -xchip=ultra2  \
-xtypemap=real:64,double:64,integer:64
```

このオプションは、自動的にすべてのデフォルトの REAL 変数および定数を REAL*8 に、デフォルトの COMPLEX を COMPLEX*16 に昇格させます。宣言されていない変数または単に REAL や COMPLEX であると宣言されている変数だけを昇格させ、明示的に宣言された変数 (REAL*4 など) は昇格させません。単精度の REAL 定数もすべて REAL*8 に昇格されます。(ターゲットプラットフォームに対して適切な -xarch や -xchip を設定します。)デフォルトの DOUBLE PRECISION データも REAL*16 に昇格させるには、-xtypemap の例で double:64 を double:128 に変更します。

詳細は、『Fortran ユーザーズガイド』または f95(1) のマニュアルページを参照してください。

データ表現

Fortran におけるデータオブジェクトのハードウェア表現に関する詳細は、『Fortran ユーザーズガイド』および Forte Developer の『数値計算ガイド』を参照してください。通常、システムやハードウェアプラットフォーム間でデータ表現が異なると、移植性に関して重大な問題が生じます。

次のことに注意してください。

- サンでは、浮動小数点演算に関しては IEEE 754 規格に準拠しています。このため、REAL*8 の最初の 4 バイトは REAL*4 と同じではありません。
- 実数型、整数型、論理型のデフォルトサイズは、-xtypemap= オプションを使用して変更する場合を除き、Fortran 95 規格に記述されています。
- 文字変数は、自由に他の変数と EQUIVALENCE 文で結合できます。しかし、境界合わせの問題が生じる可能性があるので注意が必要です。
- f95 の IEEE 浮動小数点演算では、オーバーフローまたはゼロ除算に関する例外が発生しますが、デフォルト (f95 では -ftrap=common がデフォルトです) では SIGFPE を発行したり、トラップしたりしません。例外のシグナルが発行される場合は、結果は IEEE の不定形式になります。これについては、第 6 章 を参照してください。

- 正規化された無限値が決定されることがあります。libm_single(3F) と libm_double(3F) のマニュアルページを参照してください。書式付き、および並びによる入出力文を使用すると、不定書式の書き込みや読み取りを行うことができます。

ホレリスデータ

古い Fortran アプリケーションの多くは、ホレリス ASCII データを数値データオブジェクトに格納します。1977 Fortran 規格 (および Fortran 95) において、CHARACTER データ型はこの目的のために提供され、その使用が推奨されています。現在でも古い Fortran のホレリス (*n*H) 機能を使用して変数を初期化できますが、標準的な使い方ではありません。次の表に、データ型に適合する文字の最大数を示します。この表では、太字のデータ型は、-xtypemap= コマンド行フラグによって昇格させられるデフォルトの型を示します。

表 7-1　データ型の最大文字数

	標準 ASCII 文字の最大文字数			
データ型	デフォルト	**INTEGER:64**	**REAL:64**	**DOUBLE:128**
BYTE	1	1	1	1
COMPLEX	8	8	16	16
COMPLEX*16	16	16	16	16
COMPLEX*32	32	32	32	32
DOUBLE COMPLEX	16	16	32	32
DOUBLE PRECISION	8	8	16	16
INTEGER	4	8	4	8
INTEGER*2	2	2	2	2
INTEGER*4	4	4	4	4
INTEGER*8	8	8	8	8
LOGICAL	4	8	4	8
LOGICAL*1	1	1	1	1

表 7-1　データ型の最大文字数 (続き)

	標準 ASCII 文字の最大文字数			
データ型	デフォルト	**INTEGER:64**	**REAL:64**	**DOUBLE:128**
LOGICAL*2	2	2	2	2
LOGICAL*4	4	4	4	4
LOGICAL*8	8	8	8	8
REAL	4	4	8	8
REAL*4	4	4	4	4
REAL*8	8	8	8	8
REAL*16	16	16	16	16

例 :ホレリスを使用して変数を初期化します。

```
demo% cat FourA8.f
      double complex x(2)
      data x /16Habcdefghijklmnop, 16Hqrstuvwxyz012345/
      write( 6, '(4A8, "!")') x
      end

demo% f95 -o FourA8 FourA8.f
demo% FourA8
abcdefghijklmnopqrstuvwxyz012345!
demo%
```

そのデータの型で使用しなければならない場合は、ホレリス定数によってデータ項目を初期化し、それを他のルーチンへ引き渡してください。

ホレリス定数を引数として渡したり、式や比較の中で使用しようとすると、ホレリス定数は文字型の式として解釈されます。コンパイラオプション -xhasc=no を使用して、コンパイラが副プログラム呼び出しにおいて、ホレリス定数を引数の型なしデータとして扱うようにしてください。古い Fortran プログラムを移植するときにこの処理が必要な場合があります。

非標準コーディングの手順

一般的に、アプリケーションプログラムをあるシステムのコンパイラから別のシステムのコンパイラに移植するとき、非標準のコーティングを削除すれば、移植は簡単になります。あるシステムで成功した最適化や回避策が、他のシステムでは曖昧であり、コンパイラを混乱させることもあります。特に、特定のアーキテクチャ用に最適化された手作業によるチューニングは、他の場所ではパフォーマンスを低下させる原因となる可能性もあります。パフォーマンスとチューニングに関しては、性能と調整に関する章で説明します。しかし、次の話題は、移植に際して、一般的に考慮すべきことです。

初期化されない変数

局所変数や COMMON 変数を自動的にゼロに初期化するシステムもあれば、「非数値」(NaN) に初期化するシステムもあります。しかし、標準的な取り決めはありません。したがって、プログラムは変数の初期値に関して仮定を行うべきではありません。移植性を最大限保証するために、プログラムはすべての変数を初期化すべきです。

別名参照と -xalias オプション

別名参照は、同じ記憶領域アドレスが複数の名前で参照されるときに発生します。通常、これは、ポインタを使用している場合や、副プログラムへの実引数が、それら実引数間で、あるいは副プログラム内の COMMON 変数間でオーバーラップしている場合に起こります。たとえば、引数 X と Z は同じ記憶領域の位置を参照します。 B と H も同様です。

```
COMMON /INS/B(100)
REAL S(100), T(100)
...
CALL SUB(S,T,S,B,100)
...
SUBROUTINE SUB(X,Y,Z,H,N)
REAL X(N),Y(N),Z(N),H(N)
COMMON /INS/B(100)
...
```

古い Fortran プログラムの多くは、このような別名での参照を、当時の Fortan 言語では利用できなかった、ある種の動的なメモリー管理の手段として利用していました。

別名での参照は、すべての移植可能なコードの中で避けるべきです。一部のプラットフォーム上では、`-O2` よりも高い最適化レベルを使用してコンパイルすると、予測できない結果になることがあります。

`f95` コンパイラでは、規格に準拠したプログラムのコンパイルを前提としています。Fortran の規格に厳密に準拠していないプログラムをコンパイルすると、コンパイラによる解析や最適化に支障を来す状況が生じることがあります。その状況によっては、誤った結果が得られる可能性があります。

たとえば、配列の境界を越えて添字を付ける、ポインタを使用する、または、大域的変数を直接使用しているときに副プログラムの引数としても渡すといったことが行われると、コンパイラの機能が制限されて、すべての状況で正しい最適なコードを生成できなくなる可能性があります。

プログラム内に別名参照が存在することが明らかな場合は、`-xalias` オプションを使用して、コンパイラが考慮すべきレベルを指定してください。場合によっては、適切な `-xalias` を指定しないで、`-O2` よりも高い最適化レベルでコンパイルすると、プログラムが正しく実行されなくなることがあります。

このオプションのフラグには、別名で参照する状況の種類を示すキーワードをコンマで区切って並べたリストを指定します。各キーワードには、別名参照が存在しないことを示す no% という接頭辞を付けることができます。

表 7-2　-xalias のキーワードとその意味

-xalias のキーワード	別名で参照する状況
dummy	副プログラムの仮引数が互いに別名で参照し合ったり、大域的な変数を別名で参照することができます。
no%dummy	Fortran 規格に準拠しています。実際の呼び出しで、仮引数が互いを別名で参照し合ったり、大域的な変数を別名で参照することはありません。これがデフォルトです。
craypointer	プログラムは、任意の場所を指すことのできる Cray ポインタを使用します。これがデフォルトです。
no%craypointer	Cray ポインタは、常に特定のメモリー領域を指しているか、使用されていません。
ftnpointer	どの Fortran 95 ポインタも、型、種類、ランクに関係なく任意のターゲット変数を指すことができます。
no%ftnpointer	Fortran 95 ポインタは規格の規則に従っています。これがデフォルトです。

表 7-2　-xalias のキーワードとその意味 (続き)

-xalias のキーワード	別名で参照する状況
overindex	配列の参照時に添字の境界を越えることによって発生する、配列の境界を越えた添字付けには、4 つの状況があります。プログラムの中でこれらが発生することを許可します。 • COMMON ブロック内の配列の要素を参照したときに、COMMON ブロックまたは等価なグループ内の要素が参照されることがあります。 • COMMON ブロックまたは等価なグループの要素を実引数として副プログラムに渡すと、その COMMON ブロックまたは等価なグループの任意の要素にアクセスできようになります。 • 連続構造型の変数が COMMON ブロックとして扱われます。そのような変数の要素は、同じ変数の他の要素を別名で参照することができます。 • 配列の参照がその配列内にあっても、配列の添字の各境界が越えられることがあります。 overindex は、配列構文、WHERE 文、FORALL 文には適用されません。これらの構文で配列の境界を越えた添字付けが発生する場合は、DO ループとして構文を書き直す必要があります。
no%overindex	配列の境界を越えることはありません。配列の参照が他の変数を参照することもありません。これがデフォルトです。
actual	コンパイラは、副プログラムの実引数を大域的な変数として扱います。副プログラムに引数を渡すと、Cray ポインタを使用した別名参照が行われる可能性があります。
no%actual	副プログラムに引数を渡しても、そこから別名参照が行われることはありません。これがデフォルトです。

次に、別名参照が行われる一般的な状況の例を示します。f95 コンパイラで高い最適化レベル (-O3 以上) でコンパイルする場合は、以下に示す別名参照の状況がプログラムに含まれないようにし、-xalias=no%*keyword* を使用してコンパイルした方が、より良いコードを生成できます。

場合によっては、生成したコードによって正しい結果が得られるようにするために、-xalias=*keyword* を使用してコンパイルしなければならないことがあります。

仮引数や大域的な変数による別名参照

次の例の場合は、-xalias=dummy を使用してコンパイルする必要があります。

```
parameter (n=100)
integer a(n)
common /qq/z(n)
call sub(a,a,z,n)
...
subroutine sub(a,b,c,n)
integer a(n), b(n)
common /qq/z(n)
a(2:n) = b(1:n-1)
c(2:n) = z(1:n-1)
```

コンパイラは、仮変数や共通の変数がオーバーラップする可能性があることを前提とする必要があります。

Cray ポインタによる別名参照

この例が有効なのは、-xalias=craypointer (これはデフォルトです) を使用してコンパイルした場合だけです。

```
parameter (n=20)
integer a(n)
integer v1(*), v2(*)
pointer (p1,v1)
pointer (p2,v2)
p1 = loc(a)
p2 = loc(a)
a = (/ (i,i=1,n) /)
...
v1(2:n) = v2(1:n-1)
```

コンパイラは、これらの場所がオーバーラップしている可能性があることを前提とする必要があります。

次に、オーバーラップしない Cray ポインタの例を示します。この場合は、-xalias=no%craypointer を使用してコンパイルします。この方が、より良い性能を期待できます。

```
parameter (n=10)
integer a(n+n)
integer v1(n), v2(n)
pointer (p1,v1)
pointer (p2,v2)
p1 = loc(a(1))
p2 = loc(a(n+1))
...
v1(:) = v2(:)
```

Cray ポインタは、オーバーラップしたメモリー領域を指していません。

Fortran 95 ポインタによる別名参照

次の例を、-xalias=ftnpointer を使用してコンパイルします。

```
parameter (n=20)
integer, pointer :: a(:)
integer, target :: t(n)
interface
  subroutine sub(a,b,n)
    integer, pointer :: a(:)
    integer, pointer :: b(:)
  end subroutine
end interface

a => t
a = (/ (i, i=1,n) /)
call sub(a,a,n)
....
end
subroutine sub(a,b,n)
 integer, pointer :: a(:)
 real, pointer :: b(:)
 integer i, mold

 forall (i=2:n)
   a(i) = transfer(b(i-1), mold)
```

コンパイラは、*a* と *b* がオーバーラップする可能性があることを前提とする必要があります。

この例では、コンパイラは、a と b が別のデータ型のデータを指していてもオーバーラップする可能性があることを前提とする必要があります。これは、Fortran 規格に反しています。コンパイラは、このような状況を検出すると警告を出力します。

配列の境界を越えた添字付けによる別名参照

次の例を、-xalias=overindex を使用してコンパイルします。

```
integer a,z
common // a(100),z
z = 1
call sub(a)
print*, z
subroutine sub(x)
  integer x(10)
  x(101) = 2
```

コンパイラは、副プログラムの呼び出しが z への書き込みを引き起こす可能性があることを前提とするでしょう。
-xalias=overindex を使用してコンパイルすると、プログラムは *1* ではなく *2* を出力します。

配列の境界を越えた添字付けは古い FORTRAN 77 のプログラムの多くに含まれていますが、これらは避けるべきものです。多くの場合は、結果を予測することができません。正しい結果が得られることを確かめるには、プログラムをコンパイルし、-C (境界を越えている添字の検査) オプションを使用してテストします。これによって、配列の添字に問題があるかどうかを調べることができます。

一般に、overindex は古い FORTRAN 77 のプログラムにだけ使用することをお勧めします。-xalias=overindex は、配列、構文式、部分配列、WHERE 文、FORALL 文には適用されません。

正しいコードが生成されるように、Fortran 95 のプログラムを常に Fortran 規格の添字の規則に準拠させるようにしてください。たとえば、次の例では、配列構文式の中で不明確な添字付けが行われているため、常に、配列の境界を越えた添え字付けによる誤った結果が得られます。

配列の境界を越えて添字が付けられるこの配列構文の例では、正しい結果が得られません。

```
   parameter (n=10)
   integer a(n),b(n)
   common /qq/a,b
   integer c(n)
   integer m, k
   a = (/ (i,i=1,n) /)
   b = a
   c(1) = 1
   c(2:n) = (/ (i,i=1,n-1) /)

   m = n
   k = n + n
C
C   a への参照は、実際には b を参照しています。
C   これは本来は b(2:n) = b(1:n-1) であるべきです。
C
   a(m+2:k) = b(1:n-1)

C  またはこれを逆に行います。
   a(k:m+2:-1) = b(n-1:1:-1)
```

ユーザーは直感的に、配列 *b* が今度は配列 *c* のようになることを期待しますが、実際の結果は予測できません。

overindex フラグは配列構文式には適用されないため、xalias=overindex フラグを使用してもこの状況は修正されません。この例をコンパイルすることはできますが、生成されたコードからは正しい結果を得られません。この例を書き換えて、配列構文を等価な DO ループに置き換えると、-xalias=overindex を使用してコンパイルしたときにこのフラグが働くようになります。しかし、このようなプログラミングはそもそも避けるべきです。

実引数による別名参照

コンパイラは、局所変数がどのように使用されるかを予測して、副プログラムの呼び出しによって変更されない変数について仮説を立てます。次の例では、副プログラムで使用されているポインタが原因で、コンパイラによる最適化処理が正しく行われず、結果が予測できないものになっています。正しい結果が得られるようにするには、`-xalias=actual` フラグを使用してコンパイルする必要があります。

```
program foo
     integer i
     call take_loc(i)
     i = 1
     print * , i
     call use_loc()
     print * , i
  end

  subroutine take_loc(i)
     integer i
     common /loc_comm/ loc_i
     loc_i = loc(i)
  end subroutine take_loc

  subroutine use_loc()
     integer vi1
     pointer (pi,vi)
     common /loc_comm/ loc_i
     pi = loc_i
     vi1 = 3
  end subroutine use_loc
```

`take_loc` が `i` のアドレスを取得して保存し、`use_loc` がそれを使用しています。これは、Fortran 規格に反しています。

`-xalias=actual` を使用してコンパイルすると、副プログラムへのすべての引数がそのコンパイル単位内の大域的な変数として見なされるべきであるとコンパイラに伝えられます。このため、コンパイラは、実引数のように見える変数について仮説を立てるときに、より慎重になります。

Fortran 規格に反するこのようなプログラミングは避けるべきです。

-xalias のデフォルト

リストを付けないで -xalias を指定した場合は、そのプログラムが Fortran の別名参照の規則に反していないものと仮定されます。これは、別名参照の全キーワードに no% が付いていると仮定するのと同じことです。

-xalias を指定しないでコンパイルする場合のコンパイラのデフォルトは次のとおりです。

```
-xalias=no%dummy,craypointer,no%actual,no%overindex,no%ftnpointer
```

Cray ポインタを使用していても、Fortran の別名参照の規則に準拠している場合、つまり、不明確な状況でもポインタの参照によって別名参照が行われない場合は、-xalias を使用してコンパイルすることによって、より良い性能のコードが生成される可能性があります。

あいまいな最適化

古いコードには、古いベクトル化コンパイラに特定のアーキテクチャに最適なコードを生成させるための、通常の計算の DO ループを再構成しているソースコードが含まれていることがあります。ほとんどの場合、この再構成は必要がないもので、しかもプログラムの移植性を下げます。よく使用される再構成は、Strip-mining (ストリップマイニング) とループの展開の 2 つです。

ループセクショニング (strip-mining)

いくつかのアーキテクチャ上では、固定長のベクトルレジスタのために、プログラマは手作業でループ内の配列計算について、セグメントの中にループセクショニングをしなければなりませんでした。

```
REAL TX(0:63)
...
DO IOUTER = 1,NX,64
   DO IINNER = 0,63
      TX(IINNER) = AX(IOUTER+IINNER) * BX(IOUTER+IINNER)/2.
      QX(IOUTER+IINNER) = TX(IINNER)**2
   END DO
END DO
```

ループセクショニングは最近のコンパイラには適切でありません。このループは、次のようにより明瞭に書くことができます。

```
DO IX = 1,N
  TX = AX(I)*BX(I)/2.
  QX(I) = TX**2
END DO
```

ループの展開

以前、手作業によるループの展開はソースコード最適化のための典型的なテクニックでした。しかし、現在はコンパイラがこの再構成を自動的に行います。次にループの例を示します。

```
 DO       K = 1, N-5, 6
    DO    J = 1,N
       DO I = 1,N
          A(I,J) = A(I,J) + B(I,K  ) * C(K  ,J)
*                         + B(I,K+1) * C(K+1,J)
*                         + B(I,K+2) * C(K+2,J)
*                         + B(I,K+3) * C(K+3,J)
*                         + B(I,K+4) * C(K+4,J)
*                         + B(I,K+5) * C(K+5,J)
       END DO
    END DO
 END DO
 DO       KK = K,N
    DO    J = 1,N
       DO I = 1,N
          A(I,J) = A(I,J) + B(I,KK) * C(KK,J)
       END DO
    END DO
 END DO
```

上記ループは、本来意図していたとおり、次のように書き換えるべきです。

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

時間と日付関数

時刻や CPU の経過時間を戻すライブラリ関数は、システムによって異なります。

次の表に、Fortran ライブラリでサポートされる時間関数を示します。

表 7-3　Sun Fortran 時間関数

名前	機能	マニュアルページ
time	1970 年 1 月 1 日からの経過秒数を返す	time(3F)
date	日付を文字列で返す	date(3F)
fdate	現在の日付と時刻を文字列で返す	fdate(3F)
idate	現在の月、日、年を整数配列で返す	idate(3F)
itime	現在の時、分、秒を整数配列で返す	itime(3F)
ctime	time 関数の返した時間を文字列に変換する	ctime(3F)
ltime	time 関数の返した時間を現地時刻に変換する	ltime(3F)
gmtime	time 関数の返した時間をグリニッジ標準時に変換する	gmtime(3F)

表 7-3　Sun Fortran 時間関数 (続き)

名前	機能	マニュアルページ
etime	シングルプロセッサ: プログラムの実行で経過したユーザー時間とシステム時間を返す。複数のプロセッサ: 実測時間を返す。	etime(3F)
dtime	最後に dtime を呼び出した時点から経過したユーザー時間とシステム時間を返す	dtime(3F)
date_and_time	日付と時刻を文字と数値で返す	date_and_time(3F)

詳細は、『Fortran ライブラリ・リファレンス』、またはそれぞれの関数のマニュアルページを参照してください。次に、これら時間関数を使用した簡単な例を示します (TestTim.f)。

```
      subroutine startclock
      common / myclock / mytime
      integer mytime, time
      mytime = time()
      return
      end
      function wallclock()
      integer wallclock
      common / myclock / mytime
      integer mytime, time, newtime
      newtime = time()
      wallclock = newtime - mytime
      mytime = newtime
      return
      end
      integer wallclock, elapsed
      character*24 greeting
      real dtime, timediff, timearray(2)
c   見出しを出力
      call fdate( greeting )
      print*,  "Hello, Time Now Is:",  greeting
      print*,"See how long 'sleep 4' takes, in seconds"
      call startclock
      call system( 'sleep 4' )
      elapsed = wallclock()
     print*, "Elapsed time for sleep 4 was:", elapsed," seconds"
c   ここで簡単な計算に必要な CPU 時間をテスト
      timediff = dtime( timearray )
      q = 0.01
      do 30 i = 1, 1000
            q = atan( q )
30    continue
      timediff = dtime( timearray )
      print*, "atan(q) 1000 times took:", timediff ," seconds"
      end
```

このプログラムを実行すると、次のような結果になります。

```
demo% TimeTest
       Hello, Time Now Is:Thu Feb 8 15:33:36 2001
 See how long 'sleep 4' takes, in seconds
 Elapsed time for sleep 4 was:4  seconds
 atan(q) 100000 times took:0.01  seconds
demo%
```

次の表に示すルーチンは、VMS Fortran のシステムルーチン `idate` と `time` との互換機能を提供します。これらのルーチンを使用するときは、`f95` のコマンド行で `-lV77` オプションを指定しなければなりません。この場合、標準の `f95` バージョンの代わりに VMS バージョンの方が使用されることになります。

表 7-4 非標準 VMS Fortran システムルーチンの要約

名前	定義	呼び出し手順	引数の型
`idate`	日、月、年 (d,m,y) 形式の日付	`call idate(d, m, y)`	`integer`
`time`	時分秒 (*hhmmss*) 形式の現在時刻	`call time(t)`	`character*8`

注 – `date`(3F) ルーチンおよび `idate`(3F) ルーチンの VMS バージョンは年を示す場合に 2 桁の値しか返さないので、2000 年問題に対応していません。これらのルーチンから返される日付を差し引いて継続時間を計算するプログラムは、1999 年 12 月 31 日以降は正しく機能しなくなります。代わりに、Fortran 95 のルーチン `date_and_time`(3F) を使用してください。詳細は、『Fortran ライブラリ・リファレンス』を参照してください。

問題の解決方法

ここでは、Developer Fortran 95 に移植したプログラムが予想どおりに動かないときに何をすればいいのかを提案します。

結果が近いけれども十分ではない場合

次の内容を試みてください。

- サイズと工学上の単位に注意してください。ゼロに非常に近い数が異なる場合がありますが、この差異はあまり問題ではありません。特にこの数が 2 つの巨大数の差である場合などは問題ではありません。たとえば、1.9999999e-30 と -9.9992112e-33 は異なりますが、差異はほとんどありません。

VAX の数学演算は IEEE の数学演算ほど正確ではありません。IEEE プロセッサ間でも結果が異なる場合があります。特に、これは三角関数を多く含んでいる場合に顕著です。複雑な要因がからんでおり、また、標準仕様が厳密に定義するのは基本的な算術関数だけです。このため、IEEE マシンの間にさえ微妙に差異があります。このマニュアルの第 6 章を確認してください。

- `call nonstandard_arithmetic` を使用して実行してみてください。これもパフォーマンスをかなり向上させ、サンのワークステーションをより VAX システムに似せて動作させます。VAX または他のシステムが手近にある場合、その上でも実行してみてください。多くの数値アプリケーションが、浮動小数点の実装により多少異なる結果を生成するのは、ごく一般的なことです。

- `NaN`、`+Inf` やその他の考えられるエラーがないか検査してください。さまざまな例外をトラップする命令については、このマニュアルの第 6 章か、`ieee_handler`(3m) のマニュアルページを参照してください。ほとんどのマシンでは、これらの例外は単に実行を中止させるだけです。

- 2 つの数が 6 x 10^{29} だけ異なっていても、浮動小数点の表現は同じになることもあります。次に、違う数であるのに同じ表現の例を示します。

```
      real*4 x,y
      x=99999990e+29
      y=99999996e+29
      write (*,10), x, x
10    format('99,999,990 x 10^29 = ', e14.8, ' = ', z8)
      write(*,20) y, y
20    format('99,999,996 x 10^29 = ', e14.8, ' = ', z8)
      end
```

出力結果は次のようになります。

```
99,999,990 x 10^29 = 0.99999993E+37 = 7cf0bdc1
99,999,996 x 10^29 = 0.99999993E+37 = 7cf0bdc1
```

この例では、差は 6 x 10^{29} です。このような大きな差異が生じる理由は、IEEE の単精度で保証されているのは 10 進 - 2 進変換に対して 10 進の 6 桁だけだからです。7 桁や 8 桁を正しく変換できる場合もありますが、これは値によって異なります。

警告なしにプログラムが異常終了する

警告なしにプログラムが異常終了する場合で、実行のたびに異常が発生するまでの時間が異なる場合は、次のように対処してください。

- 最低の最適化 (`-O1`) でコンパイルしてください。プログラムが動作するようであれば、いくつかのルーチンを選んで、最適化レベルを上げてコンパイルしてください。
- オプティマイザは、プログラムに関して前提条件を付けなければならないことを理解しておいてください。ユーザーが標準以外の処理を行った場合は、問題を引き起こす可能性があります。すべてのオプティマイザが、プログラムに対してあらゆるレベルの最適化を行うわけではありません。96 ページの「別名参照と -xalias オプション」を参照してください。

第8章

パフォーマンスプロファイリング

この章では、プログラムのパフォーマンスの測定と表示方法を説明します。プログラムがどこでその計算サイクルを最も費やしているか、またどのような効率でシステム資源を使用しているかを知ることが、パフォーマンスのチューニングの前提条件となります。

Forte Developer パフォーマンスアナライザ

ハイパフォーマンスのアプリケーションを開発するには、コンパイラ機能を組み合わせたり、最適化されたルーチンのライブラリ、パフォーマンス解析のためのツールが必要です。

Forte Developer ソフトウェアでは、プログラムのパフォーマンスデータを収集し、分析するための高度なツールが提供されています。

- 標本コレクタは、プロファイリングとよばれる統計ベースでパフォーマンスデータを収集します。そのデータには、呼び出しスタックの統計プロファイル、スレッド同期遅延イベント、ハードウェアカウンタのオーバーフロープロファイル、アドレス空間データ、およびオペレーティングシステムの要約情報を含めることができます。
- パフォーマンスアナライザは、ユーザーが情報を調査できるように、標本コレクタにより記録されたデータを表示します。アナライザはデータを処理し、関数、呼び出し元-呼び出し先、ソース行、分解指示、プログラムのレベルでさまざまなパフォーマンスメトリックを表示します。これらのメトリックは 3 つのグループに分類されます。時間ベースの測定基準、同期遅延測定基準、およびハードウェアカウンタ測定基準です。

また、標本アナライザを使用すれば、アプリケーションのアドレス空間での関数のロード順序を改善するためのマップファイルを作成することで、アプリケーションのパフォーマンスを細かく調整できます。

これら 2 つのツールは、以下のような質問に答えるのに役立ちます。

- プログラムが使用するリソースはどれくらいですか？
- リソースの大部分を使用するのはどの関数またはロードオブジェクトですか？
- リソースの大部分を使用するのはどのソース行や逆アセンブリ命令ですか？
- プログラムはどのようにして実行時の現在のポイントに達しますか？
- 関数やロードオブジェクトによってどのリソースが使用されていますか？

パフォーマンスアナライザのメインウィンドウは、各関数の排他的および組み込みメトリック (計測データ) をもつプログラムの関数リストを表示します。リストは、ロードオブジェクト、スレッド、軽量プロセス (LWP) およびタイムスライスによりフィルタ処理できます。選択された関数に対し、subsidiary ウィンドウは関数の呼び出し先と呼び出し元を表示します。このウィンドウは呼び出しツリーを操作するのに使用できます。たとえば、高いメトリック値の検索などです。さらに 2 つのウィンドウが、行ごとにパフォーマンスメトリックの注釈付きのソースコードや、コンパイラコメントでインタリーブされたソースコード、各命令にメトリックの注釈付きの逆アセンブリコードを表示します。ソースコードやコンパイラコメントは、可能な場合に、命令によりインタリーブされます。

ソフトウェア開発者にとってパフォーマンスのチューニングが主な仕事でないとしても、コレクタとアナライザはソフトウェア開発者向けに設計されています。これらは、一般的に使用されているプロファイルツール `prof` と `gprof` より柔軟性のある、詳細で正確な解析を提供し、`gprof` の属性エラーに依存しません。

使用可能なコレクタとアナライザのコマンド行等価ユーティリティは、次のとおりです。

- `collect`(1) コマンドを使用して、データ収集が行えます。
- `collector` サブコマンドを使用して、Collector を `dbx` から実行できます。
- コマンド行ユーティリティの `er_print`(1) は、さまざまなアナライザ表示の ASCII バージョンを出力します。
- コマンド行ユーティリティ `er_src`(1) はソースおよび逆アセンブリコードリストをコンパイラのコメント付きで、パフォーマンスデータなしで表示します。

詳細については、Forte Developer のマニュアル『プログラムのパフォーマンス解析』を参照してください。

time コマンド

プログラムのパフォーマンスと資源の利用状況に関する基本的なデータを収集するには、time(1) コマンドを使用するか、または、csh で set time コマンドを発行するのが最も簡単な方法です。

time コマンドでプログラムを実行すると、プログラム終了時に時間情報行が出力されます。

```
demo% time myprog
   The Answer is: 543.01
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
demo%
```

各欄の意味は次のとおりです。

ユーザー — システム — 時計時間 — 資源 — メモリー — 入出力 — ページング

- ユーザー — ユーザーコード中で約 6.5 秒

```
6.5u 17.1s 1:16 31% 11+21k 354+210io 135pf+0w
```

- システム — このタスクのシステムコード中で約 17.1 秒
- 時計時間 — 実行完了までに 1 分 16 秒
- 資源 — このプログラムのために使用されたシステム資源は 31 %
- メモリー — 共有プログラムメモリーは 11K バイト、プライベートデータメモリーは 21K バイト
- 入出力 — 読み取りは 354 回、書き込みは 210 回
- ページング — ページフォルトは 135 回、スワップアウトは 0 回

time 出力のマルチプロセッサ解釈

プログラムがマルチプロセッサ環境で並列に実行されたとき、結果の時間の解釈方法は異なります。/bin/time はユーザー時間を異なるスレッドで累積するので、実測時間だけが使用されます。

表示されるユーザー時間にはすべてのプロセッサ上で費やされた時間が含まれるので、かなり大きくなり、パフォーマンスの測定方法としては適していません。より適している測定は実時間、つまり、実測時間です。これは、並列化されたプログラムの正確な時間を得るには、ユーザーのプログラムだけに専念するシステム上で実行しなければならないということも意味します。

tcov プロファイリングコマンド

tcov(1) コマンドは、-xprofile=tcov オプションを付けてコンパイルしたプログラムとともに使用すると、どの文がどれくらい実行されたかを示す、ソースコードの文ごとのプロファイルを生成します。また、プログラムの基本ブロック構造に関する情報の要約も提供します。

拡張された文レベルのカバレージは、-xprofile=tcov コンパイラオプションと tcov -x オプションによって呼び出されます。出力はソースファイルのコピーであり、各文のマージンに実行回数が注釈されています。

注 – tcov により生成されたコード適用範囲レポートは、コンパイラがルーチン呼び出しをインライン化した場合は信頼性が低くなります。コンパイラは、最適化レベルが -0.3 以上で -inline オプションが指定されている場合は呼び出しをインライン化します。それにより、コンパイラはルーチンへの呼び出しを呼び出し先ルーチンの実コードに置き換えます。このとき、呼び出しがないので、これらのインライン化されたルーチンへの参照は tcov により報告されません。そのため正しい適用範囲レポートを取得するには、コンパイラのインライン化機能を有効にしてはなりません。

拡張 tcov 解析

tcov を使用するには、-xprofile=tcov を付けてコンパイルします。プログラムを実行するとき、カバレージデータは *program*.profile/tcovd に格納されます。*program* は実行可能ファイルの名前です。実行可能ファイルが a.out の場合、a.out.profile/tcovd が作成されます。

tcov -x *dirname source_files* を実行して、ソースファイルごとにマージされたカバレージ解析を作成します。レポートは、現在のディレクトリにある *file*.tcov に書き込まれます。

簡単な例を実行します。

```
demo% f95 -o onetwo -xprofile=tcov one.f two.f
demo% onetwo
     ... プログラムからの出力が表示される
demo% tcov -x onetwo.profile one.f two.f
demo% cat one.f.tcov two.f.tcov
                        program one
      1 ->              do i=1,10
     10 ->                    call two(i)
                        end do
      1 ->              end
       .....etc
demo%
```

環境変数 $SUN_PROFDATA と $SUN_PROFDATA_DIR を使用すると、中間データ収集ファイルが格納される場所を指定できます。中間データ収集ファイルは *.d と tcovd ファイルで、それぞれ古いスタイルの tcov と新しいスタイルの tcov によって作成されます。

これらの環境変数を使用して、異なる実行から収集されたデータを分けることができます。これらの環境変数を設定すると、実行プログラムは実行データを $SUN_PROFDATA_DIR/$SUN_PROFDATA/ 中のファイルに書き込みます。

同様に、tcov が読み出すディレクトリは、tcov -x $SUN_PROFDATA で指定されます。$SUN_PROFDATA_DIR が設定された場合、tcov はそれを前に付けて、作業中のディレクトリではなく、$SUN_PROFDATA_DIR/$SUN_PROFDATA/ 中でファイルを探します。

それぞれ、この後の実行のたびに tcovd ファイルに、さらにデータが追加されます。各オブジェクトファイルのデータは、ソースファイルが再コンパイルされた後にプログラムがはじめて実行されるときにクリアーされます。プログラム全体のデータは、tcovd ファイルを削除したときにクリアーされます。

詳細は、tcov(1) のマニュアルページを参照してください。

第9章

パフォーマンスと最適化

この章では、数値処理が多い Fortran プログラムのパフォーマンスを上げる可能性のある最適化のテクニックについて考えます。アルゴリズム、コンパイラオプション、ライブラリルーチン、コーディング技術を適切に使用することで、パフォーマンスを大幅に上げることができます。この章では、キャッシュ、入出力、システム環境のチューニングについては述べません。並列化の話題は、次の章で扱います。

この章では、次の話題を取り上げます。

- パフォーマンスを上げる可能性のあるコンパイラオプション
- 実行時パフォーマンスプロファイルからのフィードバックを使用したコンパイル
- 共通手続きの最適化されたライブラリルーチンの使用
- 重要なループのパフォーマンスを上げるためのコーディング技術

最適化とパフォーマンスチューニングという問題は複雑すぎて、ここでそのすべてを扱うことはできません。しかし、この章を読んで、読者が少しでも上記の話題を知ってもらえればかまいません。この章の終わりに、この問題をより深く掘り下げて説明している書籍のリストを掲載しています。

最適化とパフォーマンスチューニングは、何を最適化するか、あるいは何をチューニングするかを決定できるかどうかに大きく依存する技法です。

コンパイラオプションの選択

適切なコンパイラオプションを選択することは、パフォーマンスを上げるための第一歩です。Sun コンパイラは、オブジェクトコードに影響する幅広いオプションを提供します。デフォルトの (コンパイルコマンド行にオプションを何も明示的に指定しない) 場合、ほとんどのオプションはオフです。パフォーマンスを上げるには、これらのオプションを明示的に選択しなければなりません。

パフォーマンスオプションは通常デフォルトではオフです。なぜなら、ほとんどの最適化によって、コンパイラは、ユーザーのソースコードについて仮定を行うからです。標準のコーディング技術に準拠し、潜在的な副作用を発生させないプログラムは、正しく最適化できるはずです。しかし、標準の技術を恣意的に扱うプログラムは、コンパイラの仮定のいくつかと衝突する可能性があります。この結果作成されるコードは高速に実行するかもしれませんが、計算の結果は間違っている可能性があります。

推奨できる方法は、まず、すべてのオプションをオフにしてコンパイルし、計算の結果が正確であることを検証し、これらの最初の結果とパフォーマンスプロファイルをベースラインとして使用する方法です。それから、実行結果とパフォーマンスをベースラインと比較しながら、段階的に、オプションを追加してコンパイルし直します。数値結果が変わるようであれば、そのプログラムには疑わしいコードがあるといえます。その問題がどこにあるのかを注意深く解析して、プログラムし直す必要があります。

最適化オプションを追加した結果、パフォーマンスがあまり上がらない (あるいは下がってしまった) 場合、そのコーディングにはコンパイラがパフォーマンスを上げる余地がないのかもしれません。次の段階は、プログラムをソースコードレベルで解析し、構造を変更することによって、パフォーマンスを上げることです。

パフォーマンスオプション

次の表にリストしたコンパイラオプションによって、デフォルトのコンパイルで作成されるプログラムのパフォーマンスを上げるための方法のレパートリーは広がります。このリストには、コンパイラの中でもよりパフォーマンスに影響を与えるオプションだけを紹介しました。完全なリストについては、『Fortran ユーザーズガイド』を参照してください。

表 9-1　パフォーマンスに影響を与えるオプション

動作	オプション
さまざまな最適化オプションをいっしょに使用する	`-fast`
コンパイラの最適化レベルを *n* に設定する	`-O`*n* (`-O = -O3`)
ターゲットハードウェアを指定する	`-xtarget=`*sys*
特定の命令セットアーキテクチャを指定する	`-xarch=`*isa*
パフォーマンスプロファイルデータを使用して最適化する (`-O5` で)	`-xprofile=use`
ループを *n* まで展開する	`-unroll=`*n*
浮動小数点の簡約化と最適化を許可する	`-fsimple=1|2`
依存関係の解析を行い、ループを最適化する	`-depend`
内部手続きの最適化を実行する	`-xipo`

このようなオプションはコンパイル時間を増やすものもあります。なぜなら、プログラムをより深く解析するからです。オプションの中には、呼び出すルーチンと呼び出されるルーチンを同じファイルに集めておくと (それぞれを別々なファイルに入れておくよりも) うまく動作するものもあります。これによって、解析が大域的に行われるからです。

`-fast`

このオプション 1 つで、いくつものパフォーマンスオプションを選択したことになります。

注 – このオプションは、リリースやコンパイラにより異なるその他のオプションから特別に選択されるものです。-fast により選択されるいくつかのオプションはすべてのプラットフォームで使用できない可能性があります。-fast を拡張するには、-v (verbose) フラグを指定してコンパイルしてください。

-fast オプションを指定すると、特定のベンチマークアプリケーションのパフォーマンスが向上します。しかし、オプションによっては、アプリケーションで使用できない場合があります。-fast を使用して、最大のパフォーマンスを得るためにアプリケーションをコンパイルしてください。しかし、さらに調整が必要な場合があります。-fast を指定してコンパイルしたプログラムが正しく動作しない場合、-fast を形成している個々のオプションを調査して、プログラムを正しく動作させるオプションだけを呼び出してください。

-fast でコンパイルされたプログラムのパフォーマンスは最適で、結果が正確なデータセットもあれば、そうでないものもあります。浮動小数点演算の特定のプロパティに依存しているプログラムは、-fast を指定してコンパイルしないでください。

-fast により選択されたオプションの中には、リンクを含んでいるものがあるので、別々のステップでコンパイルとリンクを行う場合は、-fast とリンクすることに注意してください。

-fast では以下のオプションが選択されます。

- -dalign
- -depend
- -fns
- -fsimple=2
- -ftrap=common
- -libmil
- -xtarget=native
- -O5
- -xlibmopt
- -pad=local
- -xvector=yes
- -xprefetch=yes
- -xprefetch_level=2

-fast は、コンパイラの最適化能力のほとんどを簡単に引き出すための方法です。複合オプションは個別にも指定できます。また、それぞれに注意すべき副作用があります (『Fortran ユーザーズガイド』を参照)。-fast の後に別のオプションを追加して、さらに最適化を指定できます。たとえば、次のようにします。

```
f95 -fast -xarch=v9a ...
```

64 ビット可能な UltraSPARC Solaris プラットフォーム向けにコンパイルします。

-fast には、-dalign、-fns、-fsimple=2 が含まれます。このため、-fast を指定してプログラムをコンパイルすると、結果として、非標準の浮動小数点演算、非標準のデータ配列、式評価の非標準の順序になる可能性があります。ほとんどのプログラムでは、この選択は適切ではありません。

-O*n*

-O オプションを明示的に (あるいは、-fast などのマクロオプションで暗黙的に) 指定しない限り、コンパイラは最適化を行いません。ほとんどすべての場合、コンパイルの最適化レベルを指定すると、プログラムの実行パフォーマンスは上がります。一方、最適化レベルを上げるほど、コンパイル時間が増え、コードのサイズも大きくなる可能性があります。

ほとんどの場合、パフォーマンス、コードのサイズ、コンパイル時間を最もバランスよくコンパイルするのはレベル -O3 です。レベル -O4 は、呼び出し側と同じソースファイルに入っているルーチンの呼び出しの自動インライン化を追加します。副プログラム呼び出しのインライン化の詳細については、『Fortran ユーザーズガイド』を参照してください。

レベル -O5 は、低いレベルには適用できない、さらに積極的な最適化テクニックを追加します。一般的に、-O3 より上のレベルは、プログラム中で最も計算が多い、つまりパフォーマンスが上がる見込みが大きい部分のルーチンだけに指定するものです。ちなみに、異なる最適化レベルでコンパイルしたプログラムをいっしょにリンクしても何の問題もありません。

PRAGMA OPT=*n*

C$ PRAGMA SUN OPT=*n* 指令を使用して、ソースファイルのルーチンごとに異なる最適化レベルを設定します。この指令はコンパイラのコマンド行の -O*n* フラグに優先しますが、-xmaxopt=*n* フラグで最大最適化レベルを設定して使用しなければなりません。詳細は、f95(1) のマニュアルページを参照してください。

実行時プロファイルのフィードバックを使用した最適化

-xprofile=use と組み合わせた場合、コンパイラはレベル -O3 以上の最適化をより効率的に適用します。このオプションを使用すると、オプティマイザは、-xprofile=collect でコンパイルしたプログラムが典型的な入力データを使用して生成した実行時実行プロファイルから指示を受けます。フィードバックプロファイルは、どこで最適化が最大の効果を発揮するかをコンパイラに示します。これは特に -O5 で重要になります。次に示す例は、より高い最適化レベルでプロファイルを収集する典型的な例です。

```
demo% f95 -o prg -fast -xprofile=collect prg.f ...
demo% prg
demo% f95 -o prgx -fast -O5 -xprofile=use:prg.profile prg.f ...
demo% prgx
```

上記の例の最初のコンパイルで、実行時に文カバレージ統計を生成する実行可能ファイルが生成されます。2 回目のコンパイルで、このパフォーマンスデータを使用して、プログラムを最適化しています。

-xprofile オプションに関する詳細は、『Fortran ユーザーズガイド』を参照してください。

-dalign

-dalign を使用すると、コンパイラはダブルワードのロード命令またはストア命令を (可能であれば) 生成できます。データの移動量が多いプログラムは、このオプションを付けてコンパイルすれば、その恩恵を十分に受けることができます。-dalign は、-fast によって選択されるオプションの 1 つです。ダブルワード命令の速度は、同等のシングルワード命令と比べると、ほとんど倍になります。

しかし、-dalign を使用するときは (したがって、-fast を使用するときも) 十分注意しなければなりません。なぜなら、COMMON ブロック中のデータの特定の境界合わせを予想してコーディングされたプログラムのうち、問題を起こすものがあるからです。-dalign を使用すると、コンパイラはパディングを追加して、倍精度と 4 倍精度のデータをすべて (REAL も COMPLEX も) ダブルワード境界にそろえようとします。その結果、次のようなことが起こります。

- パディングを追加したために、COMMON ブロックが予想よりも大きくなることがあります。

- COMMON を共有するプログラム単位のいずれか 1 つでも -dalign を付けてコンパイルした場合、すべての単位を -dalign を付けてコンパイルしなければなりません。

たとえば、複数のデータ型が混在する COMMON ブロック全体を 1 つの配列として別名付けを行うことによって、データを書き込むプログラムは -dalign を付けるとうまく動作しません。なぜなら、倍精度変数や 4 倍精度変数のパディングのために、プログラムが予想するよりもブロックが大きくなるからです。

-depend

(SPARC プラットフォーム上で) 最適化レベル -O3 以上に -depend を追加すると、DO ループとループの入れ子に関するコンパイラの最適化能力が拡張されます。このオプションを使用すると、オプティマイザは反復間のデータの依存関係を解析し、そのループ構造を変形できるかどうか決定します。データの依存関係のないループだけがその構造を変形できます。しかし、この解析を追加すると、コンパイル時間が増えます。

-fsimple=2

指示しない限り、コンパイラは浮動小数点計算を簡易化しようとしません (デフォルトは -fsimple=0)。-fsimple=2 を追加すると、オプティマイザはさらに簡易化を行うことができます。しかし、簡易化を行うと、丸めの影響によって、結果がわずかに違うという問題が発生する可能性があります。-fsimple レベル 1 か 2 を使用する場合は、すべてのプログラム単位を同じようにコンパイルし、数値精度の整合性が失われないようにしなければなりません。このオプションについての重要な情報は、『Fortran ユーザーズガイド』を参照してください。

-unroll=*n*

長い繰り返しを持つ短いループを展開すると、いくつかのルーチンはその恩恵を受けることがあります。しかし、展開はプログラムのサイズを増やすことにもなり、他のループのパフォーマンスを下げることにもなります。*n*=1 を使用すると (デフォルト)、オプティマイザは自動的にループを展開しません。*n* が 1 より大きいときは、オプティマイザは、深さが *n* までループを展開しようとします。

コンパイラのコードジェネレータはループの展開をさまざまな要因に応じて決定します。コンパイラは、オプションが *n*>1 で指定されている場合でもループを展開しないことがあります。

繰り返しが可変の DO ループを展開する場合、展開したループとオリジナルのループの両方がコンパイルされます。繰り返し数を実行時にテストして、展開したループを実行するのが適切かどうかを決定します。ループを展開すると、特に文が 1 つか 2 つしかないループの場合は、反復ごとに行われる計算量が増えるので、オプティマイザがレジスタをスケジュールし演算を単純化する機会が増えます。繰り返しの数、ループの複雑さ、展開の深さの選択のかね合いは簡単に決定できず、ある程度の経験が必要となるでしょう。

次に示す例は、-unroll=4 を指定して、簡単なループを深さが 4 まで展開する様子を示しています (このオプションを使用しても、ソースコードは変更されません)。

元のループ:

```
DO I=1,20000
   X(I) = X(I) + Y(I)*A(I)
END DO
```

深さ 4 まで展開すると以下のコーディングと同じようになります:

```
DO I=1, 19997,4
   TEMP1 = X(I) + Y(I)*A(I)
   TEMP2 = X(I+1) + Y(I+1)*A(I+1)
   TEMP3 = X(I+2) + Y(I+2)*A(I+2)
   X(I+3) = X(I+3) + Y(I+3)*A(I+3)
   X(I) = TEMP1
   X(I+1) = TEMP2
   X(I+2) = TEMP3
END DO
```

この例は、固定した繰り返しの簡単なループを示しています。可変の繰り返し数を持つループに対しては、構造の変更はもっと複雑になります。

-xtarget=*platform*

コンパイラにターゲットのコンピュータハードウェアの正確な情報を伝えると、パフォーマンスが上がるプログラムもあります。プログラムパフォーマンスが重要なとき、ターゲットハードウェアを適切に指定することは非常に重要な問題となります。特に、新しい SPARC プロセッサ上で実行する場合です。しかし、ほとんどのプログラムと古い SPARC プロセッサの場合、パフォーマンスはそれほど上がらず、汎用指定だけで十分です。

『Fortran ユーザーズガイド』には、-xtarget= が認識するすべてのシステム名がリストされています。特定のシステム名に対して (たとえば、UltraSPARC-II なら ultra2)、-xtarget は、システムに適切に一致するように、-xarch、-xcache、-xchip の組み合わせに展開されます。オプティマイザはこれらの指定を使用して、従うべき方法と生成する命令を決定します。

-xtarget=native は特別な設定で、これを指定すると、オプティマイザはホストシステム (コンパイルを行うシステム) をターゲットとしてコードをコンパイルします。コンパイルと実行を同じシステム上で行うときは、このオプションが断然便利です。実行システムが不明であるときは、汎用のアーキテクチャ用にコンパイルするのが望ましい方法です。そのため、最適のパフォーマンスを得ることはできませんが、-xtarget=generic がデフォルトになります。

UltraSPARC-III サポート

-xtarget フラグおよび -xchip フラグは、ultra3 を受け入れ、UltraSPARC-III プロセッサ用に最適化されたコードを生成します。UltraSPARC-III プラットフォームでアプリケーションをコンパイルおよび実行する際は、-fast フラグを指定して、プラットフォームに適したコンパイラ最適化オプションを自動的に選択します。

クロスコンパイル (UltraSPARC-III 以外のプラットフォームでコンパイルしますが、UltraSPARC-III で実行できるようにバイナリを生成します) には、以下のフラグを使用してください。

```
-fast -xtarget=ultra3 -xarch=v8plusb (または -xarch=v9b)
```

64 ビットコード生成用にコンパイルするには -xarch=v9b を使用してください。

-xarch=v8plusb または v9b を使用して、UltraSPARC-III プラットフォーム専用にコンパイルされたプログラムは、UltraSPARC-III 以外のプラットフォームでは実行できません。UltraSPARC-I、UltraSPARC-II、および UltraSPARC-III の互換で実行できるようにプログラムをコンパイルするには、-xarch=v8plusa (または 64 ビットコードを生成する場合は v9a) を使用してください。

-xprofile=collect: および -xprofile=use: を使用したパフォーマンスプロファイルは、特に UltraSPARC-III プラットフォームで有効です。これは、コンパイラが最も頻繁に実行されるプログラムのセクションを特定し、局所的な最適化を実行して、最高の性能を引き出すことができるからです。

-xipo を使用した内部手続きの最適化

この新しい f95 コンパイラフラグは、Forte Developer 6 update 2 リリースで導入されたもので、内部手続き解析パスを呼び出して、プログラム全体の最適化を実行します。-xcrossfile と異なり、-xipo はリンクステップですべてのオブジェクトファイルを最適化し、コンパイルコマンドのソースファイルだけに限定されません。

-xipo は大規模な複数ファイルにわたるアプリケーションをコンパイルおよびリンクする際に特に有効です。-xipo でコンパイルされたオブジェクトファイルは、その中に保存された解析情報を持っています。これにより、ソースおよびコンパイル済みプログラムファイルの内部手続き解析ができるようになります。

内部手続き解析を効果的に使用する方法について詳細は、『Fortran ユーザーズガイド』を参照してください。

パフォーマンスに関するその他の方針

さまざまな最適化オプションを使用し、プログラムをコンパイルし、実際の実行時パフォーマンスを測定したと仮定します。次の段階は、Fortran ソースプログラムを調べて、さらにチューニングできるかどうかを決定します。

計算時間のほとんどを消費するプログラムの部分だけに注目し、次の方針を考えます。

- 手作業で作成した手続きを、最適化された同等のライブラリへの呼び出しに置き換える。
- 重要なループから入出力、呼び出し、不必要な条件操作を削除する。
- 最適化を抑制する可能性がある別名を削除する。
- ブロック IF を使用して、複雑なコードを整理する。

上記は、パフォーマンスを上げる可能性を持つプログラミング技術の一例です。さらに、特定のハードウェア構成にあわせて手作業でソースコードを調整することもできます。しかし、このような作業はコードをわかりにくくするだけでなく、コンパイラのオプティマイザもパフォーマンスを上げにくくなります。手作業でソースコードをチューニングしすぎると、その手続きの本来の意図が隠され、異なるアーキテクチャ上ではパフォーマンスに重大な悪影響を与えかねません。

最適化されたライブラリの使用

ほとんどの場合、商業用 (あるいはシェアウェア) の最適化されたライブラリは、ユーザーが手作業でコーディングしたものよりも、はるかに効率的に標準の計算手続きを実行します。

たとえば、Sun Performance Library™ は、標準の LAPACK、BLAS、FFTPACK、VFFTPACK、LINPACK ライブラリをベースとした数学サブルーチンで、高度に最適化されています。このライブラリのルーチンを使用すると、パフォーマンスは手作業でコーディングしたときよりも大幅に上がります。詳細は、『Sun Performance Library User's Guide』を参照してください。

パフォーマンスの抑制要因を除去する

Forte Developer パフォーマンス解析を使用して、プログラムの重要な計算部分を調べます。そして、注意深くループまたはループの入れ子を解析し、オプティマイザが最適なコードを生成するのを抑制している、つまりパフォーマンスを下げているコーディングを除去します。標準以外のコーディングが多いと、移植が困難になり、さらにはコンパイラによる最適化を抑制する可能性があります。

パフォーマンスを上げるためのプログラムの書き直しテクニックに関しては、この章の最後に紹介する、さまざまな参考文献で取り上げられています。ここでは 3 つの代表的なアプローチを説明します。

キーとなるループから入出力をなくす

プログラムの重要な計算作業を囲んでいるループ、あるいはループの入れ子内の入出力は、パフォーマンスを大幅に下げる原因となります。入出力ライブラリで消費される CPU 時間は、そのループで消費される時間のほとんどを占めます。また、入出力はプロセス割り込みの原因ともなるので、プログラムスループットを下げます。可能な限り、入出力を計算ループの外に出すことで、入出力ライブラリへの呼び出し回数が大幅に減ります。

副プログラムの呼び出しを削減する

副プログラムがループの深い入れ子から呼び出されると、何千回と呼び出される可能性もあります。呼び出しごとの各ルーチン内で消費される時間は少なくても、その合計の影響はかなりのものです。また、副プログラムの呼び出しは、その呼び出しを含むループの最適化を抑制します。なぜなら、コンパイラは、その呼び出しのレジスタの状態に関して仮定を行うことができないからです。

副プログラム呼び出しの自動インライン化 (`-inline=`*x,y,..z*、または `-O4` を使用する) は、コンパイラが実際の呼び出しを副プログラム自身で置き換える (副プログラムをループの中に入れる) ための 1 つの方法です。インライン化されるべきルーチンの副プログラムのソースコードは、呼び出し側のルーチンと同じファイルに存在しなければなりません。

副プログラム呼び出しを削減する方法は他にもあります。

- 文関数を使用する。呼び出される外部関数が単純な数学関数である場合、その関数を文関数 (あるいは文関数の集合) として書き直すことができます。文関数はコンパイル時にインライン化され、最適化できます。
- ループを副プログラムに入れる。つまり、副プログラムを書き換えて、(ループの外で) 呼び出される回数を減らし、呼び出しごとに値のベクトルあるいは配列を操作するようにします。

複雑なコードを整理する

計算が多いループ内の操作が複雑であると、コンパイラの最適化は抑制される可能性があります。一般的に、算術的な IF と論理的な IF をすべてブロック IF に置き換えるのがよい方法であるとされています。

元のコード:

```
    IF(A(I)-DELTA) 10,10,11
10  XA(I) = XB(I)*B(I,I)
    XY(I) = XA(I) - A(I)
    GOTO 13
11  XA(I) = Z(I)
    XY(I) = Z(I)
    IF(QZDATA.LT.0.)GOTO 12
    ICNT = ICNT + 1
    ROX(ICNT) = XA(I)-DELTA/2.
12  SUM = SUM + X(I)
13  SUM = SUM + XA(I)
```

整理されたコード:

```
    IF(A(I).LE.DELTA) THEN
      XA(I) = XB(I)*B(I,I)
      XY(I) = XA(I) - A(I)
    ELSE
      XA(I) = Z(I)
      XY(I) = Z(I)
      IF(QZDATA.GE.0.) THEN
        ICNT = ICNT + 1
        ROX(ICNT) = XA(I)-DELTA/2.
      ENDIF
      SUM = SUM + X(I)
    ENDIF
    SUM = SUM + XA(I)
```

ブロック IF を使用すると、コンパイラが最適なコードを生成する機会が多くなるだけでなく、読みやすくなるので、移植性も確保されます。

コンパイラのコメントを表示する

-g デバッグオプションを使用してコンパイルする場合、Forte Developer パフォーマンス解析ツールの一部である er_src(1) ユーティリティを使用して、コンパイラにより生成されたソースコードの注釈を表示することができます。生成されたアセンブリ言語の注釈付きソースコードを表示することもできます。次に、er_src によって生成された、単純な do ループに関するコメントの例を示します。

```
demo% f95 -c -g -O4 do.f
demo% er_src do.o
ソースファイル: /home/hatake/prog/f/do.f
オブジェクトファイル: do.o
ロードオブジェクト: do.o

     1.         program do
     2.         common aa(100),bb(100)

   Function x inlined from source file do.f into the code for the following line
   Loop below pipelined with steady-state cycle count = 3 before unrolling
   Loop below unrolled 5 times
   Loop below has 2 loads, 1 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
     3.         call x(aa,bb,100)
     4.         end
     5.         subroutine x(a,b,n)
     6.         real a(n), b(n)
     7.         v = 5.
     8.         w = 10.

   Loop below pipelined with steady-state cycle count = 3 before unrolling
   Loop below unrolled 5 times
   Loop below has 2 loads, 1 stores, 0 prefetches, 1 FPadds, 1 FPmuls, and 0 FPdivs per iteration
     9.       do 1 i=1,n
    10. 1       a(i) = a(i)+v*b(i)
    11.       return
    12.       end
```

コメントのメッセージにより、コンパイラにより実行された最適化処理の詳細が分かります。この例では、サブルーチンの呼び出しをインライン化し、ループを 5 回展開しています。この情報を検証することで、将来の最適化戦略に役立てることができるでしょう。

コンパイラのコメントおよび逆アセンブルコードの詳細については、Forte Developer のマニュアル『プログラムのパフォーマンス解析』を参照してください。

参考文献

次の参考文献には、さらに詳細な説明があります。

- 『High Performance Computing』、Kevin Dowd および Charles Severance 著、O'Reilly & Associates、第 2 版、1998
- 『Techniques for Optimizing Applications:High Performance Computing』、Rajat Garg および Ilya Sharapov 著、サン・マイクロシステムズ Press Blueprint、2001

第10章

並列化

この章では、マルチプロセッサの並列化の概要を示し、SPARC プロセッサ上の Forte Developer Fortran 95 コンパイラの機能について説明します。

Rajat Garg およびIlya Sharapov 著、サン・マイクロシステムズ Press Blueprint 社 (`http://www.sun.com/blueprints/pubs.html`) 発行の『Techniques for Optimizing Applications: High Performance Computing』も参照してください。

基本概念

アプリケーションの並列化 (またはマルチスレッド化) とは、マルチプロセッサシステム上で実行できるよう、またはマルチスレッド環境、コンパイルされたプログラムを分散することです。並列化によって、1 つのタスク (`DO` ループなど) を複数のプロセッサ (またはスレッド) を使って実行できるので、実行速度が上がる可能性があります。

Ultra™ 60、Enterprise™ Server 6500、または Sun Enterprise Server 10000 のようなマルチプロセッサシステム上でアプリケーションプログラムを効率的に実行できるようにするためには、そのアプリケーションプログラムをマルチスレッド化する必要があります。つまり、並列実行できるタスクを識別し、複数のプロセッサまたはスレッドを横にしてその計算を分配するようにプログラムを変更する必要があります。

アプリケーションのマルチスレッド化は、`libthread` プリミティブを適切に呼び出すことによって、手作業で行うことができます。しかし、膨大な量の解析とプログラムの変更が必要となります。詳細は、Solaris の『マルチスレッドのプログラミング』を参照してください。

Sun コンパイラは、マルチプロセッサシステム上で動作できるようにマルチスレッド化されたオブジェクトコードを自動的に生成できます。Fortran コンパイラは、並列性をサポートする主要な言語要素としての DO ループに焦点をあわせます。並列化は、Fortran ソースプログラムに一切手を加えることなく、ループの計算作業を複数のプロセッサに分配します。

どのループを並列化するか、またそのループをどのように分配するかは、完全にコンパイラに任せることも (-autopar)、ソースコード指令を使用してプログラマが明示的に決定することも (-explicitpar)、その両方を組み合わせることも (-parallel) できます。

注 – 独自の (明示的な) スレッド管理を行うプログラムをコンパイルするときは、コンパイラのどのような並列化オプションも付けてはなりません。明示的なマルチスレッド化 (libthread プリミティブへの呼び出し) は、並列化オプションを付けてコンパイルしたルーチンと組み合わせることはできません。

プログラム中のすべてのループが有効に並列化されるわけではありません。計算作業量の少ないループを並列化すると、(並列タスクの起動と同期に費やされるオーバーヘッドと比べると) 実際には実行が遅くなることもあります。また、安全に並列化できないループもあります。このようなループは、文間あるいは反復間の依存関係のため、並列化すると異なる結果を生成します。

明示的な DO ループとともに暗示的なループ (IF ループと Fortran 95 配列構文など) が、Fortran コンパイラでの自動並列化の対象となります。

f95 は、安全にそして有効に並列化できる可能性のあるループを自動的に検出できます。しかし、ほとんどの場合、隠れた副作用の恐れがあるので、この解析はどうしても控え目になります (どのループが並列化され、どのループが並列化されていないかは、-loopinfo オプションで表示できます)。ループの前にソースコード指令を挿入することによって、特定のループを並列化するかどうかを明示的に制御できます。しかし、このように明示的に並列化を指定したループによって結果が間違ったとしても、それはユーザーの責任になります。

Forte Developer Fortran 95 コンパイラは、OpenMP 2.0 Fortran API 指令を実装することによって明示的に並列化を行います。古いプログラムに対応するために、f95 は古い Sun 形式および Cray 形式の指令もサポートしています。OpenMP は、Fortran 95、C、C++ での明示的な並列化の非公式の標準となっています。古い指令形式には OpenMP をお勧めします。

Open MP については、Forte Developer の『OpenMP API ユーザーズガイド』か、OpenMP の Web サイト (`http://www.openmp.org/`) を参照してください。

古い並列化指令については、157 ページの「Sun 形式の並列化指令」および169 ページの「Cray 形式の並列化指令」を参照してください。

速度向上 — 何を期待するか

4 つのプロセッサ上で動作するようにプログラムを並列化した場合、そのプログラムは、1 つのプロセッサ上で動作させるときの約 1/4 の時間で処理できる (4 倍の速度向上になる) と期待できるでしょうか。

おそらく、答えは「ノー」です。プログラムの全体的な速度向上は、並列実行しているコード中で消費される実行時間の割り合いによって厳密に制限されると証明できます (アムダールの法則)。適用されるプロセッサがいくつになろうとも、これは常に真です。事実、並列実行した実行プログラムの合計時間のパーセンテージを p とすると、理論的な速度向上の制限は 100/(100-c) となります。したがって、プログラムの 60% だけが並列実行した場合、プロセッサの数にかかわらず、速度向上は最大 2.5 倍です。そして、プロセッサが 4 つの場合、このプログラムの理論的な速度向上は、最大限の効率が発揮されたと仮定しても、1.8 倍です。4 倍にはなりません。

最適化のことを考えると、ループの選択は重要です。プログラムの合計実行時間のほんの一部としか関わらないループを並列化しても、最小の効果しか得られません。効果を得るためには、実行時間の大部分を消費するループを並列化しなければなりません。したがって、どのループが重要であるかを決定し、そこから始めるのが第一歩です。

問題のサイズも、並列実行するプログラムの割合を決定するのに重要な役割を果たし、その結果、速度向上にもつながります。問題のサイズを増やすと、ループの中で行われる作業量も増えます。3 重に入れ子にされたループは、作業量が 3 乗になる可能性があります。入れ子の外側のループを並列化する場合、問題のサイズを少し増やすと、(並列化していないときのパフォーマンスと比べて) パフォーマンスが大幅に向上します。

プログラムの並列化のための手順

次に、アプリケーションの並列化に必要な手順について、極めて一般的な概要を示します。

1. 最適化。適切なコンパイラオプションのセットを使用して、1 つのプロセッサ上で最高のパフォーマンスを得ます。
2. プロファイル。典型的なテストデータを使用して、プログラムのパフォーマンスプロファイルを決定します。最も重要なループを見つけます。
3. ベンチマーク。逐次処理でのテストの結果が正確かどうかを決定します。これらの結果とパフォーマンスプロファイルをベンチマークとして使用します。
4. 並列化。オプションと指令の組み合わせを使用して、並列化した実行可能ファイルをコンパイルし、構築します。
5. 検証。並列化したプログラムを 1つのプロセッサや 1 つのスレッド上で実行し、結果を検査して、その中の不安定さやプログラミングエラーを見つけます (`$PARALLEL` または `$OMB_NUM_THREADS` に 1 を設定します。141 ページを参照してください) 。
6. テスト。複数のプロセッサ上でさまざまな実行を試し、結果を検査します。
7. ベンチマーク。専用のシステムで、プロセッサの数を変えながらパフォーマンスを測定します。問題のサイズを変化させて、性能の変化を測定します (スケーラビリティ)。
8. 手順 4 から 手順 7 を繰り返す。パフォーマンスに基づいて、並列化スキームを改良します。

データ依存性の問題

すべてのループが並列化できるわけではありません。複数のプロセッサ上でループを並列実行すると、実行している反復の順序が変わる可能性があります。さらに、ループを並列実行する複数のプロセッサがお互いに干渉する可能性もあります。このような状況が発生するのは、ループ中にデータ依存性がある場合です。

データ依存性の問題が発生する場合は、再帰、縮約、間接アドレス指定、データに依存するループが繰り返されています。

データに依存したループ

ループを書き直して、並列化することで、データへの依存をなくすことができます。しかし、拡張再構成が必要な場合があります。

以下は、いくつかの一般的な規則です。

- すべての繰り返しが個々のメモリー位置に書き込む場合のみ、ループはデータから独立しています。
- いずれの繰り返しも同じ位置に書き込まない限り、繰り返しはその位置から読み取る場合があります。

これらは並列化の一般的な条件です。ループを並列化するかどうか決める際に、コンパイラの自動並列化解析により、追加の条件が検討されます。しかし、抑制により誤った結果を出すループも含め、ループを明示的に並列化する指令を使用することができます。

再帰

ループのある反復で設定され、後続の反復で使用される変数は、反復間依存性、つまり再帰の原因となります。ループ中で再帰を行う場合は、反復が適切な順序で実行されなければなりません。

```
DO I=2,N
   A(I) = A(I-1)*B(I)+C(I)
END DO
```

たとえば、上記コードでは、以前の反復中で `A(I)` 用に計算された値が、現在の反復中で (`A(I-1)` として) 使用されなければなりません。各反復を並列実行して、1 つのプロセッサで実行したときと同じ結果を生成するためには、反復 `I` は、反復 `I+1` が実行できる前に完了していなければなりません。

縮約

縮約操作は、配列の要素を 1 つの値に縮約します。たとえば、配列の要素の合計を 1 つの変数にまとめる場合、その変数は反復ごとに更新されます。

```
DO K = 1,N
  SUM = SUM + A(I)*B(I)
END DO
```

このループを並列実行する各プロセッサが反復のサブセットを取る場合、`SUM` の値を上書きしようとして、各プロセッサはお互いに干渉します。うまく処理するためには、各プロセッサが 1 度に 1 回ずつ合計を実行しなければなりません。しかし、順序は問題になりません。

ある共通の縮約操作は、コンパイラによって、特別なケースであると認識され、処理されます。

間接アドレス指定

ループ依存性は、値が未知である添字によってループの中の添字付けられた配列への格納から発生する可能性があります。たとえば、添字付の配列中に繰り返される値がある場合、間接アドレス指定は順序に依存することがあります。

```
DO L = 1,NW
  A(ID(L)) = A(L) + B(L)
END DO
```

上記例中、ID 中で繰り返される値は、 A の要素を上書きする原因となります。逐次処理の場合、最後の格納が最終値です。並列処理の場合、順序は決定されていません。使用される A(L) の値 (古い値か更新された値) は、順序に依存します。

並列オプションと指令についての要約

次の表に、f95 の並列化に関するコンパイルオプションを示します。

表 0-1　並列化オプション

オプション	フラグ
自動 (のみ)	`-autopar`
自動、縮約	`-autopar -reduction`
明示（のみ)	`-explicitpar`
自動、明示	`-parallel`
自動、縮約、明示	`-parallel -reduction`
並列化されるループを表示	`-loopinfo`
明示に関連する警告を表示	`-vpara`
局所変数をスタックに割り当て	`-stackvar`
Sun 形式の MP 指令を使用	`-mp=sun`

表 0-1　並列化オプション

オプション	フラグ
Cray 形式の MP 指令を使用	-mp=cray
OpenMP 指令を使用	-mp=openmp
OpenMP 並列化用にコンパイル	-openmp

オプションについての注意

- -reduction を指定するときは -autopar も必要です。
- -autopar には -depend とループ構造の最適化が含まれます。
- -parallel は -autopar -explicitpar と同義です。
- 打ち消しのオプションには、-noautopar、-noexplicitpar、-noreduction があります。
- 並列化オプションはどのような順序で指定してもかまいません。しかし、必ずすべてを小文字にしなければなりません。
- 明示的に並列化されたループに対して、縮約操作は解析されません。
- いずれの並列化オプションを使用する場合にも、WorkShop のライセンスが必要です。
- -openmp はオプション組み合わせのマクロです。

  ```
  -mp=openmp -stackvar -explicitpar
  ```

- オプション -loopinfo、-vpara、-mp は、並列化オプション -autopar、-explicitpar、-parallel のいずれかとともに使用しなければなりません。

スレッドの数

PARALLEL (または OMP_NUM_THREADS) 環境変数は、プログラムで使用可能なスレッドの最大数を制御します。環境変数を設定することにより、実行時システムに、プログラムで使用可能なスレッドの最大数が知らされます。デフォルトは 1 です。一般に、PARALLEL 変数または OMP_NUM_THREADS 変数に、ターゲットプラットフォームで使用可能なプロセッサ数を設定します。

次の例で、その設定方法を示します。

```
demo% setenv PARALLEL 4          C シェル
                                 または
demo% PARALLEL=4                 Bourne/Korn シェル
demo% export PARALLEL 4
```

上記例では、**PARALLEL** を **4** に設定することで、プログラムの実行は最大 4 つのスレッドを使用できます。ターゲットマシンが 4 つのプロセッサを利用できる場合、各スレッドはプロセッサ 1 つずつにマップされます。利用可能なプロセッサが 4 つより少ない場合、スレッドのいくつかは他のスレッドと同じプロセッサ上で実行されるので、パフォーマンスは下がります。

SunOS コマンド psrinfo(1M) は、システムで利用可能なプロセッサのリストを表示します。

```
demo% psrinfo
0   on-line   since 03/18/96 15:51:03
1   on-line   since 03/18/96 15:51:03
2   on-line   since 03/18/96 15:51:03
3   on-line   since 03/18/96 15:51:03
```

スタック、スタックサイズ、並列化

プログラムの実行は、プログラムを最初に実行したスレッドのためにメインメモリーのスタックを保持し、各ヘルパースレッドのために個々のスタックを保持します。スタックとは、副プログラムの呼び出し時に引数と AUTOMATIC 変数を保持するために使用される一時的なメモリーアドレス空間です。

メインスタックのデフォルトのサイズは、約 8M バイトです。Fortran コンパイラは、通常、局所変数と配列を (スタックにではなく) STATIC として割り当てます。しかし、
-stackvar オプションを使用すると、すべての局所変数と配列をスタックに割り当てます (あたかもそれが AUTOMATIC 変数であるかのように)。-stackvar は並列化とともに使用することを推奨します。なぜなら、ループ中の CALL を並列化するオプティマイザの能力を向上させるからです。-stackvar は、副プログラム呼び出しを持つ明示的に並列化されたループには必須です。-stackvar については、『Fortran ユーザーズガイド』を参照してください。

C シェル (csh) を使用し、limit コマンドにより現在のメインスタックのサイズを表示し、設定します。

```
demo% limit                          C シェルの例
cputime 制限無し
filesize 制限無し
datasize 2097148 kbytes
stacksize 8192 kbytes                <- 現在のメインスタックのサイズ
coredumpsize 1 kbytes
descriptors 64
memorysize 制限無し
demo% limit stacksize 65536          <- メインスタックを 64M バイトに設定
demo% limit stacksize
stacksize 65536 kbytes
```

Bourne シェルまたは Korn シェルの場合、対応するコマンドは ulimit です。

```
demo% >limit -a              Korn シェルの例
cputime(seconds)             制限無し
filesize(blocks)             制限無し
datasize(kbytes)             2097148
stacksize(kbytes)            8192
coredumpsize(blockes)        0
descriptors(descriptors)     64
memorysize(kbytes)           制限無し
demo% ulimit -s 65536
demo% ulimit -s
65536
```

マルチスレッド化されたプログラムの各スレッドは、独自のスレッドスタックを持っています。このスタックは、初期スレッドのスタックと似ています。しかし、スレッド固有のものです。スレッドの PRIVATE 配列と変数 (スレッドに局所的な) は、スレッドスタックに割り当てられます。SPARC V9 (UltraSPARC) プラットフォームでのデフォルトのサイズは 8 メガバイトです。その他のプラットフォームでは 4 メガバイトです。このサイズは、STACKSIZE 環境変数で設定されます。

```
demo% setenv STACKSIZE 8192      <- スレッドスタックサイズを
                                        8M バイトに設定 C シェル
          または
demo% STACKSIZE=8192                  Bourne/Korn シェル
demo% export STACKSIZE 8192
```

いくつかの並列化された Fortran コードに対しては、スレッドスタックのサイズをデフォルトより大きく設定することが必要になります。しかし、どれくらいの大きさに設定すればいいのかを知る方法はなく、試行錯誤してみるしかありません。特に、専用配列または局所配列が関連する場合はわかりません。スタックのサイズが小さすぎてスレッドが実行できない場合、プログラムはセグメンテーションフォルトで異常終了します。

自動並列化

`-autopar` オプション と `-parallel` オプションを使用すると、`f77` および `f95` コンパイラは、効率的に並列化できる `DO` ループを自動的に見つけます。このようなループは変形され、利用可能なプロセッサに対してその反復が均等に分配されます。コンパイラは、このために必要なスレッド呼び出しを生成します。

ループの並列化

コンパイラによる依存性の解析は、`DO` ループを並列化可能なタスクに変形します。コンパイラは、ループの構造を変形して、逐次実行する、並列化できないセクションを切り離します。次に、利用可能なプロセッサに対して作業を均等に分配します。各プロセッサが反復の異なったブロックを実行します。

たとえば、4 つの CPU と 1,000 回の反復を持つ並列化ループの例で、各スレッドは 250 回の反復をまとめて実行します。

プロセッサ 1 が実行する反復	1	から	250
プロセッサ 2 が実行する反復	251	から	500
プロセッサ 3 が実行する反復	501	から	750
プロセッサ 4 が実行する反復	751	から	1000

並列化できるのは、計算の実行順序に依存しないループだけです。コンパイラによる依存性の解析は、本質的にデータ依存性をもつループを拒否します。ループ中のデータフローを完全に決定できない場合、コンパイラは保守的に動作し、並列化を行いません。また、パフォーマンスの向上よりもオーバーヘッドが勝る場合、ループを並列化しないことを選択する可能性もあります。

コンパイラは常に、静的ループスケジューリング (つまり、ループ中の作業を単純に均等な反復ブロックに分割する方法) を使用して、ループを並列化することを選択することに注意してください。明示的な並列化指令を使用すれば、他の分配スキームも指定できます。この指令については、この章の後半で説明します。

配列、スカラー、純スカラー

自動並列化という観点から、2、3 の定義が必要です。

配列とは、最低でも 1 次元で宣言された変数のことです。

スカラーとは、配列でない変数のことです。

純スカラーとは、別名付けされていない (EQUIVALENCE 文や POINTER 文で参照されていない) スカラー変数のことです。

例 : 配列とスカラー

```
    dimension a(10)
    real m(100,10), s, u, x, z
    equivalence ( u, z )
    pointer ( px, x )
    s = 0.0
    ...
```

m とa は両方とも配列変数です。s は純スカラーです。変数 u、x、z、px はスカラー変数ですが、純スカラーではありません。

自動並列化の基準

反復間データ依存性をもたない DO ループは、-autopar か -parallel によって自動的に並列化されます。自動並列化のための一般的な基準は次のとおりです。

- 明示的な DO ループと、IF ループや Fortran 95 配列構文などの暗黙的なループのみが、並列化されます。
- ループの各反復に対する配列変数の値は、そのループの他の反復に対する配列変数の値に依存してはなりません。
- ループ内の計算は、ループの終了後に参照される純スカラー変数を条件によって変更してはなりません。

- ループ内の計算は、反復にまたがるスカラー変数を変更してはなりません。これは「ループ伝達の依存性」と呼ばれます。
- ループの本文内の処理量は、並列化のオーバーヘッドよりも多くなければなりません。

見かけの依存性

コンパイラは、コンパイルされたコードを変形するときに、ループ中のデータ依存の原因になりそうな (見かけの) 参照を自動的に取り除きます。このような多数の変換の 1 つは、一部の配列の専用バージョンを使用します。コンパイラがこの処理を行うことができるのは、一般的には、そのような配列が本来のループで一時領域としてのみ使用されていることが判断できる場合です。

例 : `-autopar` を使用しています。専用配列によって依存が取り除かれます。

```
parameter (n=1000)
real a(n), b(n), c(n,n)
do i = 1, 1000            <-- 並列化される
  do k = 1, n
    a(k) = b(k) + 2.0
  end do
  do j = 1, n-1
    c(i,j) = a(j) + 2.3
  end do
end do
end
```

上記の例では、外側のループが並列化され、別々のプロセッサ上で実行されます。配列 `a` を参照する内側のループはデータ依存性の原因になるように見えますが、コンパイラはその配列の一時的な専用コピーを作成して、外側のループの反復を依存しないようにしています。

自動並列化の抑制要因

自動並列化では、次のいずれかが発生すると、コンパイラはループを並列化しません。

- `DO` ループが、並列化される別のループ内の入れ子になっているとき
- フロー制御で、`DO` ループの外に飛び出す可能性があるとき
- ループ内で、ユーザーレベルの副プログラムが起動されているとき

- ループ内に入出力文があるとき
- ループ内の計算が別名付きスカラー変数を変更するとき

入れ子にされたループ

マルチプロセッサシステムでは、最も内側のループではなく、ループの入れ子の最も外側のループを並列化するのが最も効果的です。並列処理は一般にループのオーバーヘッドがかなり大きいため、最も外側のループを並列化することでループのオーバーヘッドが最小になり、各プロセッサの処理量が最大になります。自動並列化では、コンパイラは入れ子の最も外側のループからループの解析を始め、並列化可能なループが見つかるまで、内側に進んでいきます。入れ子の中でループが 1 つでも並列化されたら、並列ループの中に含まれるループは無視されます。

縮約操作を使用した自動並列化

配列をスカラーに変形する計算のことを「縮約操作」と呼びます。典型的な縮約操作は、ベクトルの要素の合計や積です。縮約操作は、ループ内の計算が反復にまたがって累積的に変数を変更しないという基準には反するものです。

例 : ベクトルの要素の合計を縮訳する

```
s = 0.0
do i = 1, 1000
    s = s + v(i)
end do
t(k) = s
```

しかし、一部の操作では、並列化を妨げるのが縮約だけの場合は、この基準にかかわらず並列化できます。共通の縮約操作が頻繁に発生するので、コンパイラはこれらの操作を特別なケースであると認識し、並列化します。

`-reduction` コンパイラオプションが `-autopar` か `-parallel` とともに指定されていなければ、縮約操作の認識は、自動並列化解析の中には含まれません。

並列化可能なループが表 0-2 にリストされた縮約操作のいずれか 1 つを持つ場合、`-reduction` が指定されていれば、コンパイラはそのループを並列化します。

認識される縮約操作

次の表に、f77 および f95 が認識する縮約操作をリストします。

表 0-2　認識される縮約操作

数学的な操作	Fortran 文のテンプレート
合計	`s = s + v(i)`
積	`s = s * v(i)`
ドット積	`s = s + v(i) * u(i)`
最小	s = amin(s, v(i))
最大	`s = amax( s, v(i))`

OR	`do i = 1, n`
	`  b = b .or. v(i)`
	`end do`

AND	`b = .true.`
	`do i = 1, n`
	`  if (v(i) .le. 0) b=b .and. v(i)`
	`end do`

ゼロでない要素の計数	`k = 0`
	`do i = 1, n`
	`  if ( v(i) .ne. 0 ) k = k + 1`
	`end do`

MIN 関数と MAX 関数はすべての形式で認識されます。

数値的な正確性と縮約操作

次の条件のため、浮動小数点の合計や積の縮約操作が不正確になることがあります。

- 計算が並列実行されるときの順序が、1 つのプロセッサ上で逐次実行されるときの順序と違う場合
- 計算の順序が、浮動小数点数の合計や積に影響を与えた場合。ハードウェア浮動小数点の加算や乗算は結合則を満たしません。どのように演算対象が関連付られているかによって、丸め、オーバーフロー、アンダーフローが発生する可能性があります。たとえば、`(X*Y)*Z` と `X*(Y*Z)` は、数値的には意味が違う可能性があります。

状況によって、エラーが受けつけられない場合があります。

例 : 丸めの例です。-1 と +1 の間の 100,000 個の乱数を合計します。

```
demo% cat t4.f
    parameter ( n = 100000 )
    double precision d_lcrans, lb / -1.0 /, s, ub / +1.0 /, v(n)
   s = d_lcrans ( v, n, lb, ub ) ! n 個の -1 と +1 の間の乱数を求める。
    s = 0.0
    do i = 1, n
    s = s + v(i)
    end do
    write(*, '(" s = ", e21.15)') s
     end
demo% f95 -autopar -reduction t4.f
```

結果は、プロセッサの数によって異なります。次の表に、-1 と +1 の間の 100,000 個の乱数の合計を示します。

プロセッサの数	出力
1	s = 0.568582080884714E+02
2	s = 0.568582080884722E+02
3	s = 0.568582080884721E+02
4	s = 0.568582080884724E+02

この状況では、丸めの誤差はおよそ 10-14 なので、この乱数のデータは容認できます。詳細は、『数値計算ガイド』を参照してください。

明示的な並列化

この節では、どのループを並列化するか、どの方針を使用するかを明示的に指示するための、`f95` によって認識されるソースコード指令について説明します。

Fortran 95 コンパイラは、OpenMP の Fortran 並列化指令を受け付けます。詳細は、『OpenMP API ユーザーズガイド』を参照してください。。

`f95` コンパイラは、従来の Sun 形式と Cray 形式の並列化指令も受け付けるため、明示的に並列化されたプログラムを他のプラットフォームから移植しやすくなっています。

プログラムを明示的に並列化するためには、アプリケーションコードの事前解析と深い理解、そして、共有メモリー並列化の概念が必要です。

DO ループに並列化のためのマークを付けるには、ループの直前に指令を置きます。OpenMP Fortran 95 指令が認識されて DO ループが並列化されるようにするには、-openmp を使用してコンパイルします。古い Sun 形式または Cray 形式の指令の場合は、-parallel または -explicitpar を使用してコンパイルします。並列化指令は、その指令後の DO ループを並列化する (または並列化しない) ようにコンパイラに伝えるコメント行です。指令は、プラグマともいいます。

どのループに並列化のマークを付けるかを選択するときは注意してください。並列を実行するときに間違った結果を計算してしまうデータ依存性がループにある場合でも、コンパイラは、DOALL 指令でマークを付けられたすべてのループに対して、スレッド化された並列コードを生成します。

libthread プリミティブを使用して独自のマルチスレッド化コーディングを行う場合は、コンパイラのいかなる並列化オプションも付けてはなりません。コンパイラは、スレッドライブラリへのユーザーの呼び出しを使用してすでに並列化されているコードを並列化することはできません。

並列可能なループ

次のような場合、ループは明示的な並列化に適しています。

- DO ループであって、DO WHILE または Fortran 95 の配列構文ではない場合
- ループの各反復に対する配列変数の値が、そのループの他の反復に対する配列変数の値に依存しない場合
- ループがスカラーを変更する場合、そのスカラーがループ終了後に参照されない場合。このようなスカラー変数は、ループ終了後定義された値をもつとは保証されません。なぜなら、コンパイラはこのような変数に対しては適切な書き戻しを自動的に行わないからです。
- 各反復において、ループの内側から呼び出される副プログラムが、他の反復に対する配列変数の値を参照しない、または変更しない場合。
- DO ループの添字が必ず整数である場合。

スコープ規則: 非公開と共有

非公開変数または専用配列は、ループの 1 回の反復だけで使用されます。ある反復で非公開変数または非公開配列に代入された値は、そのループの別の反復には伝達されません。

共有変数または共有配列は、他のすべての反復で共有されます。ある反復で共有変数または共有配列に代入された値は、そのループの別の反復からも参照されます。

明示的に並列化されたループで共有の値を参照する場合、共有によって正確性の問題が発生しないように注意してください。共有変数が更新またはアクセスされたとき、コンパイラは同期処理を行いません。

あるループの中で変数が非公開であると指定した場合、さらに、その変数の唯一の初期化が他のループの中にある場合、その変数の値はループの中で未定義のままとなる可能性があります。

ループでの副プログラム呼び出し

ループで (または呼び出し元ルーチン内から呼び出された副プログラムで) 副プログラムを呼び出すと、データ依存性が生じる可能性があり、これは呼び出しのチェーンをたどってデータや制御フローを深く分析しなければ気づかないでしょう。作業量の多い一番外側のループを並列化すればよいのですが、これらは副プログラムをいくつも呼び出してループがとても深くなっている傾向があります。

このような手続き間の分析は難しく、またコンパイル時間がかなり長くなってしまうので、自動並列化モードでは行われません。明示的な並列化では、コンパイラは、`PARALLEL DO` または `DOALL` 指令によりマークしたループ内にサブプログラムへの呼び出しが含まれていても、そのループの並列化コードを生成します。この場合も、ループ内に、また呼び出し先サブプログラムを含めてループ内のすべてにおいてデータ依存が存在しないようにすることはプログラマの仕事です。

さまざまなスレッドから 1 つのルーチンを何度も起動すると、局所静的変数への参照でお互いに干渉し合うような問題が発生することがあります。ルーチン内のすべての局所変数を静的変数ではなく自動変数にすることで、この問題は防ぐことができます。このようにしてサブプログラムを起動すると、そのたびに局所変数が固有の領域に保存され、それらがスタック上で保守されるので、何度起動してもお互いに干渉することはなくなります。

局所サブプログラム変数は、自動変数にすることが可能で、AUTOMATIC 文で指定するか、または -stackvar オプションを指定してサブプログラムをコンパイルすることでスタック上に常駐させることができます。ただし、DATA 文で初期化された局所変数については、実際の割り当てで初期化されるように書きかえる必要があります。

注 – 局所変数をスタックに割り当てると、スタックがオーバーフローしてしまう可能性があります。スタックのサイズを大きくする方法については、142 ページの「スタック、スタックサイズ、並列化」を参照してください。

明示的並列化の抑制

一般に、ユーザーがコンパイラにループを並列化するように明示的に指示している場合、コンパイラはそのようにします。ただし、例外もあり、ループによってはコンパイラが並列化を行わないものがあります。

次に、DO ループの明示的な並列化を妨げる抑制の中で、検出可能なものを示します。

- DO ループが、並列化された別の DO ループ内にネストされている場合。

  この例外は、間接ネストについても当てはまります。ユーザーがサブルーチンを呼び出しているループを明示的に並列化すると、コンパイラにそのサブルーチン内のループを並列化するように要求しても、これらのループは実行時に並列で実行されません。
- フロー制御文により、DO ループから外部へのジャンプが許可されている場合。
- ループの添字変数が、等価になるなどの影響を受ける場合。

-vpara および -loopinfo を指定してコンパイルすると、コンパイラが明示的にループを並列化している最中に問題を検出すると診断メッセージが発せられます。

次に、一般にコンパイラにより検出される並列化の問題を示します。

表 0-3　明示的な並列化時の問題

問題	並列化	警告メッセージ
ループは、並列化されている別のループ内にネストされています。	いいえ	いいえ
ループは、並列化されたループの本文内で呼び出されているサブルーチン内にあります。	いいえ	いいえ
フロー制御文で、ループから外部へのジャンプが許可されています。	いいえ	はい
ループの添字変数が、悪影響を受けています。	はい	いいえ
ループ内の変数に、ループ繰越の依存があります。	はい	はい
I ループ内の入出力文―通常、出力順序は予想できないので賢明な処理ではありません。	はい	いいえ

例: ネストされたループ

```
      ...
!$OMP PARALLEL DO
      do 900 i = 1, 1000        !  並列化されます ( 外側のループ )
        do 200 j = 1, 1000      !  並列化されません。警告も発しません
           ...
200   continue
900      continue
      ...
```

例: サブルーチン内で並列化されたループ

```
  program main
        ...
! $OMP PARALLEL DO
        do 100 i = 1, 200       <- 並列化されます
        ...
        call calc (a, x)
        ...
100      continue
      ...
subroutine calc ( b, Y )
      ...
! $OMP PARALLEL DO
        do 1 m = 1, 1000        <- 並列化されません
          ...
1       continue
         return
        end
```

この例では、サブルーチン自体が並列で実行されているので、その中のループは並列化されません。

例: ループから外部へのジャンプ

```
!$omp parallel do
      do i = 1, 1000     ! ← 並列化されず、エラーとなります
        ...
        if (a(i) .gt. min_threshold ) go to 20
        ...
      end do
20      continue
      ...
```

並列化のマークが付いたループの外にジャンプがあると、コンパイラはエラーと診断します。

例: ループ依存性を持つループの変数

```
demo% cat vpfn.f
      real function fn (n,x,y,z)
      real y(*),x(*),z(*)
      s = 0.0
!$omp parallel do private(i,s) shared(x,y,z)
      do  i = 1, n
         x(i) = s
      s = y(i)*z(i)
      enddo
      fn=x(10)
      return
      end
demo% f95 -c -vpara -loopinfo -openmp -O4 vpfn.f
"vpfn.f", 4 行目: 警告: ループには参照を無効にする並列化が含まれているかもしれません
"vpfn.f", 5 行目: 並列化されます、ユーザープラグマの使用
```

ループは並列化されますが、可能なループ依存性は警告中で診断されます。しかし、ループ依存性のすべてがコンパイラによって診断できないことに注意してください。

明示的並列化での入出力

並列に実行するループで入出力を実行できます。ただし、次の条件があります。

- さまざまなスレッドからの出力がインタリーブされても問題とならないこと (プログラム出力は確定的ではありません)。
- ループの並列実行の安全性が確実であること。

例: ループ内の入出力文

```
!$OMP PARALLEL DO PRIVATE(k)
      do i = 1, 10      ! 並列化されます
        k = i
        call show ( k )
      end do
      end
      subroutine show( j )
      write(6,1) j
1      format('Line number ', i3, '.')
      end
demo% f95  -opemp t13.f
demo% setenv PARALLEL 4
demo% a.out
Line number      9.
Line number     10.
Line number      4.
Line number      5.
Line number      6.
Line number      1.
Line number      2.
Line number      3.
Line number      7.
Line number      8.
```

ただし、入出力が再帰的な場合、つまり、入出力文に、入出力を行う関数への呼び出しが含まれている場合は、実行時エラーが発生します。

OpenMP 並列化指令

OpenMP は、マルチプロセッサプラットフォーム用の並列プログラミングモデルで、Fortran 95、C、C++ のアプリケーションの標準的なプログラミング方法となってきているものです。Forte Developer コンパイラでは、この並列プログラミングモデルを推奨しています。

OpenMP 指令を有効にするには、-openmp オプションフラグを使用してコンパイルします。Fortran 95 OpenMP 指令は、指令名と従属句の前に付く、コメントのような !$OMP という符号によって識別されます。

!$OMP PARALLEL は、プログラム内の並列領域を識別します。!$OMP DO は、並列領域内で並列化すべき DO ループを識別します。この 2 つの指令を組み合せて 1 つの !$OMP PARALLEL DO 指令とすることができます。この指令は、DO ループの直前に配置します。

OpenMP の仕様には、プログラムの 1 つの並列領域内で作業を共有および同期化するための多数の指令と、データのスコープ指定および制御のための従属句が含まれています。

OpenMP 指令と古い Sun 形式の指令の最も大きな違いは、OpenMP では、非公開または共有のいずれかとして明示的にデータのスコープを指定する必要があることです。

Sun や Cray の並列化指令を使用して古いプログラムを変換するためのガイドラインも含めて、詳細は、Forte Developer の『OpenMP API ユーザーズガイド』を参照してください。

Sun 形式の並列化指令

Sun 形式の指令は、`-explicitpar` オプションや `-parallel` オプションを指定してコンパイルした場合に、デフォルトで (または `-mp=sun` オプションを指定して) 使用できます。

Sun 並列化指令の構文

並列化指令は、1 つまたは複数の指令行で構成されます。Sun 形式の指令行は次のように定義されます。

```
C$PAR  Directive   [ Qualifiers ]        <- 最初の指令行
C$PAR& [More_Qualifiers]                <- オプションの継続行
```

- 指令行は、大文字と小文字の区別がありません。
- 指令行の最初の 5 文字は、`C$PAR`、`*$PAR`、`!$PAR` のいずれかです。
- ソースが固定形式の場合
  - 最初の指令行の 6 桁目は空白です。
  - 継続指令行の 6 桁目は空白以外の文字です。
  - `-e` オプションが指定されていない限り、72 桁目以降は無視されます。
- ソースが Fortran 95 の自由形式の場合
  - 先行する空白は、後に符合があれば使用できます。
  - 認識される符号は、`!$PAR` だけです。
- 修飾子がある場合、指令と同じ行または継続行の指令の後に指定します。
- 1 行に複数の修飾子を指定する場合、コンマで区切ります。
- 指令や修飾子の前後、またはその間にある空白は無視されます。

Sun 形式の並列化指令は、次のとおりです。

指令	動作
TASKCOMMON	COMMON ブロックの変数をスレッド非公開として宣言する。
DOALL	次のループを並列化する。
DOSERIAL	次のループを並列化しない。
DOSERIAL*	次のループの入れ子を並列化しない。

Sun 形式の並列化指令の例

```
C$PAR TASKCOMMON ALPHA                          ブロックを非公開として宣言
      COMMON /ALPHA/BZ,BY(100)

C$PAR DOALL                                     修飾子なし

C$PAR DOSERIAL

C$PAR DOALL SHARED(I,K,X,V), PRIVATE(A)
     この 1 行の指令は、次の 3 行の指令と同じ意味です。
C$PAR DOALL
C$PAR& SHARED(I,K,X,V)
C$PAR& PRIVATE(A)
```

TASKCOMMON 指令

TASKCOMMON 指令は、グローバルな COMMON ブロックの変数をスレッド非公開として宣言します。共通ブロックで宣言した変数はすべてスレッドに対して非公開変数になりますが、スレッド内ではグローバルなままです。指定した COMMON ブロックだけが TASKCOMMON として宣言できます。

指令の構文は次のとおりです。

C$PAR TASKCOMMON *comon_block_name*

指令は、その指定されたブロックの COMMON 宣言の直後に指定しなければなりません。

この指令が有効になるのは、-explicitpar または -parallel オプションを付けてコンパイルしたときだけです。それ以外の場合では、この指令は無視され、ブロックは通常の共通ブロックとして扱われます。

TASKCOMMON ブロックで宣言した変数は、すべての DOALL ループや、DOALL ループ内から呼び出されているルーチンでスレッド非公開変数として処理されます。各スレッドはそれぞれ COMMON ブロックのコピーを取得するので、あるスレッドにより書き込まれたデータはその他のスレッドから直接参照することはできません。プログラムの連続部分では、最初のスレッドの COMMON ブロックコピーがアクセスされます。

TASKCOMMON ブロックの変数は、PRIVATE、SHARED、READONLY などの DOALL 修飾子では使用されません。

同じ共通ブロックが定義されているコンパイル単位のうち、すべてではなく一部だけでそのブロックをタスク共通として宣言するとエラーになります。-commonchk=yes フラグを付けてプログラムをコンパイルすることで、タスク共通の整合性の実行時検査を行うことができます。実行時検査は、パフォーマンスを下げることのできるプログラム開発の段階だけで行ってください。

DOALL 指令

DOALL 指令は、コンパイラにその直後に続く DO ループを並列化するコードを生成するように要求します (-parallel オプションまたは -explicitpar オプションを指定してコンパイルした場合)。

注 – ループが明示的に並列化されている場合、そのループ内の縮約操作の解析と変形は行われません。

例 : ループの明示的な並列化

```
demo% cat t4.f
    ...
C$PAR DOALL
    do i = 1, n
       a(i) = b(i) * c(i)
    end do
    do k = 1, m
       x(k) = x(k) * z(k,k)
    end do
    ...
demo% f77 -explicitpar t4.f
```

DOALL の修飾子

Sun 形式のDOALL 指令のすべての修飾子はオプションです。次の表にそれらをまとめます。

表 0-4　DOALL の修飾子

修飾子	動作	構文
PRIVATE	変数 *u1*、*u2*、... を反復間で共有しない。	DOALL PRIVATE(*u1*,*u2*,...)
SHARED	変数 *v1*、*v2*、... を反復間で共有する。	DOALL SHARED(*v1*,*v2*,...)
MAXCPUS	*n* 個を超える CPU (スレッド) を使用しない。	DOALL MAXCPUS(*n*)
READONLY	指定の変数を DOALL ループで変更しない。	DOALL READONLY(*v1*,*v2*,...)
SAVELAST	DO ループの最後の反復におけるすべての専用変数の値を保存する。	DOALL SAVELAST
STOREBACK	DO ループの最後の反復における変数 *v1*、*v2*、... の値を保存する。	DOALL STOREBACK(*v1*,*v2*,...)
REDUCTION	変数 *v1*、*v2*、... を縮約変数として扱う。	DOALL REDUCTION(*v1*,*v2*,...)
SCHEDTYPE	スケジューリング型を *t* に設定する。	DOALL SCHEDTYPE(*t*)

PRIVATE(*varlist*)

PRIVATE(*varlist*) 修飾子は、変数リスト *varlist* 中のすべてのスカラーと配列が DOALL ループの非公開であることを指定します。配列とスカラーは両方とも非公開として指定できます。配列の場合、DOALL ループのスレッドごとに配列全体のコピーが作成されます。DOALL ループで参照されるスカラーや配列のうち、変数リストに含まれないものはすべて、デフォルトのスコープ規則に従います (151 ページの「スコープ規則: 非公開と共有」参照)。

例: ループ i で配列 a を非公開として指定します。

```
C$PAR DOALL PRIVATE(a)
    do i = 1, n
      a(1) = b(i)
      do j = 2, n
      a(j) = a(j-1) + b(j) * c(j)
    end do
    x(i) = f(a)
    end do
```

SHARED(*varlist*)

SHARED(*varlist*) 修飾子は、変数リスト *varlist* 中のすべてのスカラーと配列が DOALL ループにおいて共有されることを指定します。配列とスカラーは両方とも共有として指定できます。共有スカラーと共有配列は、DOALL ループのすべての反復で共通です。DOALL ループで参照されるスカラーや配列のうち、変数リストに含まれないものはすべて、デフォルトのスコープ規則に従います。

例 : 共有変数を指定します。

```
C$PAR DOALL SHARED(y)
    do i = 1,n
      a(i) = y
    end do
```

上記の例では、変数 y は、その値が i ループの反復間で共有される変数であると指定されています。

READONLY(*varlist*)

READONLY (*varlist*) 修飾子は、変数リスト *varlist* 中のすべてのスカラーと配列が DOALL ループにおいて読み取り専用であることを指定します。読み取り専用のスカラーと配列は、DOALL ループ中のどの反復においても変更されないという、共有スカラーと共有配列の特別なクラスです。スカラーや配列を READONLY として指定すると、コンパイラは、DOALL ループの各スレッドごとに、その変数または配列の別々のコピーを使用する必要がないと判断します。

例 : 読み取り専用変数を指定します。

```
    x = 3
C$PAR DOALL SHARED(x),READONLY(x)
    do i = 1, n
      b(i) = x + 1
    end do
```

上記の例では、x は共有変数です。しかし、READONLY が指定されているので、コンパイラは、x の値が I ループの反復においても変更されないことを信頼できます。

STOREBACK (*varlist*)

STOREBACK 変数または STOREBACK 配列とは、その値が DOALL ループで計算される変数または配列のことです。計算された値は、そのループの終了後に使用できます。言い換えると、ループの最後の反復における STOREBACK スカラーと STOREBACK 配列の値は、DOALL ループの外から参照できます。

例 : ループインデックス変数を STOREBACK として指定します。

```
C$PAR DOALL PRIVATE(x), STOREBACK(x,i)
    do i = 1, n
      x = ...
    end do
    ... = i
    ... = x
```

上記の例では、変数 x と i は両方とも i ループの非公開変数であり、STOREBACK 変数でもあります。x 値が最後の反復が終わった時点の値であるのに対し、ループの後の i 値は n+1 となります。

STOREBACK には、留意すべきいくつかの潜在的な問題があります。

最後の反復が、STOREBACK 変数または STOREBACK 配列の値を最後に更新する反復と同じ場合でも、STOREBACK 操作は明示的に並列化されたループの最後の反復時に発生します。

例 : STOREBACK 変数は、逐次バージョンとは異なる可能性があります。

```
C$PAR DOALL PRIVATE(x), STOREBACK(x)
    do i = 1, n
      if (...) then
        x = ...
      end if
    end do
    print *,x
```

上記の例では、出力される STOREBACK 変数 x の値は、i ループの逐次バージョンで出力された結果と異なる可能性があります。明示的に並列化された場合、i ループの最後の反復(i = n) を処理し、x の STOREBACK 操作を行うプロセッサは、現在 x の最後に更新された値をもっているプロセッサとは異なる可能性があります。コンパイラはこのような潜在的な問題に関する警告メッセージを出します。

SAVELAST

SAVELAST 修飾子は、非公開スカラーと非公開配列のすべてが DOALL ループにおいて STOREBACK であることを指定します

例 : SAVELAST を指定します。

```
C$PAR DOALL PRIVATE(x,y), SAVELAST
    do i = 1, n
      x = ...
      y = ...
    end do
    ... = i
    ... = x
    ... = y
```

この例では、変数 x、y、i が STOREBACK 変数です。

REDUCTION(*varlist*)

REDUCTION (*varlist*) 修飾子は、変数リスト *varlist* 中のすべての変数が DOALL ループにおいて縮約変数であることを指定します。縮約変数 (または配列) とは、その部分的な値を別々のプロセッサ上で個々に計算し、その部分的な値をもとにして最後の値を計算できる変数のことです。

縮約変数のリストを指定すると、コンパイラが、DOALL ループが縮約ループであるかどうかを識別し、そのループの並列縮約コードを生成するのを助けます。

例 : 縮約変数を指定します。

```
C$PAR DOALL REDUCTION(x)
    do i = 1, n
      x = x + a(i)
    end do
```

上記の例では、変数 x は (合計の) 縮約変数です。i ループは (合計の) 縮約ループです。

SCHEDTYPE(*t*)

SCHEDTYPE(*t*) 修飾子は、特定のスケジューリング型を指定して DOALL ループをスケジュールすることを指定します。

表 0-5　DOALL SCHEDTYPE の修飾子

スケジューリング型	動作
STATIC	当該 DO ループに対して、静的スケジューリングを使用する。(これは、Sun 形式の DOALL のデフォルトスケジューリング型である。) すべての反復を均一に利用可能なプロセッサに分配する。 例: 反復が 1,000 回で、プロセッサが 4 個の場合、各スレッドは 250 回の連続反復を 1 かたまりとして取得する。
SELF[(*chunksize*)]	当該 DO ループに対して、自己スケジューリングを使用する。各スレッドは、一度に *chunksize* 回の反復を 1 かたまりとして取得する。それは、すべての反復が処理されるまで不確定順序で分配される。反復のかたまりは、使用可能なすべてのスレッドに一様に配布されることはない。 • *chunksize* が指定されない場合は、コンパイラは値を選択する。 例: 反復が 1,000 回で、*chunksize* が 4 の場合、各スレッドはすべての反復が処理されるまで一度に 4 回分の反復を取得する。

表 0-5　DOALL SCHEDTYPE の修飾子 (続き)

スケジューリング型	動作
FACTORING[(*m*)]	この DO ループに対して、印紙化スケジューリングを使用する。初期の反復が n 回で、スレッド数が k 個の場合、すべての反復はいくつかの反復かたまりのグループに分けられる。最初のグループには、それぞれが n/(2k) 回の反復が k かたまりだけある。そして、2 番目のグループには n/(4k) 回の反復が k かたまりだけある。以降同様である。各グループのかたまりのサイズは、2k で除算した残りの反復となる。FACTORING は動的なので、各スレッドが各グループから正確に 1 つずつかたまりを取得するとは限らない。 • 各スレッドに、m 回以上の反復を割り当てる必要がある。 • 最後の 1 回は、余った小さな値でもかまわない。 • m を指定しない場合、コンパイラにより値が選択される。 例: 反復が 1,000 回で、FACTORING(3) を指定し、スレッドが 4 個の場合、最初のグループに 125 回の反復を、2 番目のグループに 4 チャンクの 62 回の反復を、そして 3 番目のグループに 4 チャンクの 31 回の反復を、というように割り当てる。
GSS[(*m*)]	この DO ループに対して、ガイド付き自己スケジューリングを使用する。 初期の反復が *n* 回で、CPU が *k* 個の場合、次のようになる。 • 1 番目のプロセッサに m/k 回の反復を割り当てる。 • すべての反復が処理されるまで、*k* で除算した残りの反復を 2 番目のスレッドに、というように割り当てる。 GSS は動的なので、反復かたまりが使用可能なすべてのスレッドに一様に配布されることはない。 • 各スレッドに m 回以上の反復を割り当てなければならない。 • 最後の 1 回は、余った小さな値でもかまわない。 • m を指定しない場合、コンパイラにより値が選択される。 例: 反復が 1,000 回で、GSS(10) と指定され、スレッドが 4 個の場合、最初のスレッドに 250 回の反復が、2 番目のスレッドに 187 回の反復が、そして 3 番目のスレッドに 140 回の反復が、というように割り当てられる。

複数の修飾子

修飾子は複数回指定でき、この場合は効果が累積されます。修飾子が衝突する場合は、警告メッセージが出力され、最後に出現する修飾子が優先されます。

例 : 3 行の Sun 形式の指令です (MAXCPUS 、SHARED、および PRIVATE 修飾子の衝突に注意)。

```
C$PAR DOALL MAXCPUS(4) READONLY(S) PRIVATE(A,B,X) MAXCPUS(2)
C$PAR DOALL SHARED(B,X,Y) PRIVATE(Y,Z)
C$PAR DOALL READONLY(T)
```

例 : 上記 3 行と同じ内容を 1 行で指定します。

```
C$PAR DOALL MAXCPUS(2), PRIVATE(A,Y,Z), SHARED(B,X), READONLY(S,T)
```

DOSERIAL 指令

DOSERIAL 指令は、指定したループの並列化を無効にします。この指令は、指令の直後にあるループ 1 つだけに適用されます。

例 : 1 つのループを並列化から除外します。

```
    do i = 1, n
C$PAR DOSERIAL
      do j = 1, n
        do k = 1, n
          ...
        end do
      end do
    end do
```

この例では、-parallel を指定してコンパイルすると、j ループは並列化されませんが、i または k ループは並列化されます。

DOSERIAL* 指令

DOSERIAL* 指令は、ループの指定した入れ子の並列化を無効にします。この指令は、指令の直後にあるループの入れ子全体に適用されます。

例 : ループの入れ子全体を並列化から除外します。

```
    do i = 1, n
C$PAR DOSERIAL*
      do j = 1, n
        do k = 1, n
           ...
        end do
      end do
    end do
```

上記のループで parallel を用いてコンパイルすると、j のループは並列化されず、i または k ループが並列化されます。

DOSERIAL* と DOALL の相互作用

DOSERIAL* と DOALL の両方が同じループに指定されている場合、最後の指令が使用されます。

例 : DOSERIAL と DOALL を両方とも指定します。

```
C$PAR DOSERIAL*
    do i = 1, 1000
C$PAR DOALL
      do j = 1, 1000
        ...
      end do
    end do
```

上記の例では、i ループは並列化されず、j ループは並列化されます。

また、DOSERIAL* 指令のスコープは、テキスト上で DOSERIAL* 指令の直後にあるループの入れ子を超えることはありません。DOSERIAL* 指令は、DOSERIAL* 指令がある関数またはサブルーチンに限定されます。

例 : DOSERIAL* は、呼び出されたサブルーチンのループまで拡張されません。

```
    program caller
    common /block/ a(10,10)
C$PAR DOSERIAL*
    do i = 1, 10
      call callee(i)
    end do
    end

    subroutine callee(k)
    common /block/a(10,10)
    do j = 1, 10
      a(j,k) = j + k
    end do
    return
    end
```

上記の例では、サブルーチン callee への呼び出しがインライン化されているかどうかにかかわらず、DOSERIAL* は i ループにしか適用されず、j ループには適用されません。

Sun 形式のデフォルトのスコープ規則

Sun 形式 (C$PAR) の明示的な指令では、コンパイラはデフォルトの規則を適用して、スカラーや配列が共有か非公開かを判別します。デフォルトの規則を変更するには、ループの中で参照されるスカラーや配列の属性を指定します。Cray 形式の !MIC$ 指令では、ループ内に現れるすべての変数は、DOALL 指令を使用して、共有か非公開用かを明示的に宣言しなければなりません。

コンパイラは、次のデフォルトの規則を適用します。

- スカラーはすべて非公開として扱われます。スレッドがループを実行するたびに局所的なスカラーのコピーが作成され、その局所的なコピーはスレッドでのみ使用されます。
- 配列参照はすべて共有参照として扱われます。あるスレッドが配列要素へ書き込んだ内容は、どのスレッドからも参照できます。ただし、共有変数へのアクセス時に、同期処理は行われません。

ループに反復間依存性が存在する場合、実行すると間違った結果になる可能性があります。ユーザーは、このような事態が発生しないように注意しなければなりません。コンパイラは、このような状況をコンパイル時に検出し、警告を発することもあります。しかし、コンパイラは、このようなループの並列化を無効にするわけではありません。

例: 問題が発生する可能性がある equivalence 文

```
      equivalence (a(1),y)
C$PAR DOALL
      do i = 1,n
        y = i
        a(i) = y
      end do
```

この例では、スカラー変数 y は a(1) と等価であるため、デフォルトでこのスカラー変数 y を非公開変数として、また a(:) を共有変数として扱ってしまいます。つまり、並列化された I ループを実行したときに、間違った結果を引き起こす可能性があります。この場合、診断は発行されません。

この例を修正するには、C$PAR、DOALL、PRIVATE (y) を使用します。

Cray 形式の並列化指令

Cray 形式の指令を使用する場合は、-mp=cray を付けてコンパイルする必要があります。

Sun 形式の指令を付けてコンパイルしたプログラム単位と Cray 形式の指令を付けてコンパイルしたプログラム単位を混在させると、異なる結果を生成する可能性があります。

Sun 形式の指令と Cray 形式の指令の主な違いは、AUTOSCOPE が指定されていない限り、Cray 形式では、ループ中のすべてのスカラーと配列に対して、SHARED か PRIVATE のどちらかによる明示的なスコープの指定が必要であることです。

次の表に、Cray 形式の指令の構文を示します。

```
!MIC$ DOALL
!MIC$&  SHARED( v1, v2,  … )
!MIC$&  PRlIVATE( u1, u2,  … )
    ...任意の修飾子
```

Cray 形式の指令の構文

並列化指令は、1 つまたは複数の指令行から構成されます。指令行は、次の点を除いて、Sun 形式 (157 ページの「Sun 形式の並列化指令」) と同じ構文で定義されます。

- 符号は CMIC$、*MIC$、!MIC$ のいずれかでありますが、f95 の自由形式で認識されるのは !MIC$ だけです。
- ループ内で参照されているすべての変数や配列は、SHARED 修飾子または PRIVATE 修飾子に記述します。

Cray 指令は、Sun 形式と似ています。

Cray 指令	Sun 形式との比較
DOALL	修飾子セットとスケジューリングが異なります。
TASKCOMMON	Sun 形式と同じです。
DOSERIAL	Sun 形式と同じです。
DOSERIAL*	Sun 形式と同じです。

DOALL 修飾子

Cray 形式の DOALL では、PRIVATE 修飾子が必要です。DO ループ内の各変数は、非公開または共有として修飾されなければならず、DO ループの添字は常に非公開でなければなりません。次の表に、利用可能な Cray 形式の修飾子を要約します。

表 0-6　DOALL 修飾子 (Cray 形式)

修飾子	動作
SHARED(*v1*, *v2*, …)	変数 *v1*、*v2*、... を反復間で共有する。つまり、これらの変数はすべてのタスクからアクセス可能である。
PRIVATE(*x1*, *x2*, …)	変数 *x1*、*x2*、... を反復間で共有しない。つまり、各タスクがこれらの変数の独自のコピーをもつ。
AUTOSCOPE	PRIVATE 修飾子または SHARED 修飾子によって明示的にスコープを指定されていない変数や配列は、この後に示すスコープ規則に従ってスコープが指定されます。
SAVELAST	DO ループの最後の反復における非公開変数の値を保存する。
MAXCPUS(*n*)	*n* 個を超える CPU を使用しない。

AUTOSCOPE 自動スコープ規則

AUTOSCOPE を指定することにより、コンパイラが以下の規則を使用して、PRIVATE または SHARED として明示的にスコープされていない変数または配列のスコープを決定することができます。

SHARED で指定されている変数や配列については、以下のいずれかが真となる必要があります。

- 変数または配列が読み取り専用
- ループインデックスによる配列の添字付け
- 変数または配列の読み取り後、書き込み

PRIVATE への変数または配列については、以下が真となる必要があります。

- 変数または配列の書き込み後、読み取り

コンパイル時に、AUTOSCOPE が常に変数や配列のスコープを決定するとは限りません。中でもループへの条件付きパスが、コンパイラが決定できない方法のスコープを変更することができます。明示的な PRIVATE および SHARED 修飾子で変数をスコープするとより安全です。

Cray 形式のスケジューリング修飾子

Cray 形式の指令では、DOALL 指令は、1 つのスケジューリング修飾子 (たとえば、!MIC$& CHUNKSIZE(100)) を指定できます。表 0-7 に、Cray 形式の DOALL 指令を示します。

表 0-7　DOALL Cray スケジューリング

修飾子	動作
GUIDED	ガイド付き自己スケジューリングを使用して、反復を分配する。この分配は、動的な負荷バランスを行うことで同期のオーバーヘッドを最小にする。デフォルトのチャンクサイズは 64 です。GUIDED は、Sun 形式の GSS(64) と同じです。
SINGLE	使用可能なスレッドごとに 1 回の反復を配布する。SINGLE は動的であり、Sun 形式の SELF(1) と等価である。
CHUNKSIZE(*n*)	使用可能なスレッドごとに *n* 回の反復を分配する。 *n* は、整数の式でなければならない。最高のパフォーマンスを得るためには、*n* は整数の定数でなければならない。 CHUNKSIZE(n) は Sun 形式の SELF(n) と等価である。 例: 反復が 100 回で、CHUNKSIZE(4) の場合、各スレッドに一度に 4 回の反復を分配する。
NUMCHUNKS(*m*)	*n* 回の反復がある場合、利用可能なプロセッサごとに *n/m* 回の反復を分配する。最後の 1 回は、余った小さい値でもかまわない。*m* は式である。NUMCHUNK(m) は、Sun 形式の SELF(n/m) と等価である。ただし、*n* は反復の合計回数である。 例: 反復が 100 回で、NUMCHUNK(4) の場合、各スレッドは一度に 25 回分の反復を取得する。

デフォルトのスケジューリング型は Sun 形式の STATIC です(Cray 形式の DOALL 指令でスケジューリング型が指定されていない場合)。これと等価の Cray 形式のスケジューリング型はありません。

環境変数

並列化で使用される環境変数には、次の 3 つがあります。

- PARALLEL と OMP_NUM_THREADS
- SUNW_MP_WARN
- SUNW_MP_THR_IDLE

(142 ページの「スタック、スタックサイズ、並列化」 の説明も参照してください)

PARALLEL と OMP_NUM_THREADS

並列化されたプログラムをマルチスレッド環境で実行するには、実行前に、PARALLEL または OMP_NUM_THREADS 環境変数を設定しなければなりません。これにより、実行時システムに、プログラムで作成可能なスレッドの最大数が知らされます。デフォルトは 1 です。一般に、PARALLEL または OMP_NUM_THREADS 環境変数には、ターゲットプラットフォームで使用可能なプロセッサ数を設定します。
例: SETENV PARALLEL 4

SUNW_MP_WARN

実行時マルチタスクライブラリが出力する警告メッセージを制御します。TRUE に設定されると、ライブライは警告メッセージを stderr に出力します。FALSE に設定されると、警告メッセージが無効になります。FALSE はデフォルトです。
例: SETENV SUNW_MP_WARN TRUE

SUNW_MP_THR_IDLE

プログラムの並列処理を実行する、マスタースレッド以外の各スレッドのタスク終端ステータスを制御するには、SUNW_MP_THR_IDLE 環境変数を使用します。この変数は、以下の値のいずれかを設定できます。

値	意味
SPIN	並列タスクの分担の処理が終わったとき、新しい並列タスクが届くまでスレッドはスピン (またはビジーウェイト) します。(デフォルト)
SLEEP （*time*）	並列タスクの分担の処理が終わったとき、スレッドがスピン待ちする時間を指定します。スレッドがスピンしている間に、新しいタスクが届くと、スレッドは新しいタスクをすぐに実行します。それ以外の場合は、スレッドは新しいタスクが届くまではスリープ状態になります。 *time* は *n* 秒 (*n*s) または *n* ミリ秒 (*n*ms) で指定できます。 引数なしの SLEEP が指定されると、並列タスクの処理が終わると、スレッドは直ちにスリープ状態になります。SLEEP、SLEEP (0)、SLEEP (0s) および SLEEP (0ms) はすべて同じ意味です。

SUNW_MP_THR_IDLE が明示的に指定されない場合のデフォルトは、SPIN です。

例

```
% setenv SUNW_MP_THR_IDLE 50ms
% setenv PARALLEL 4
% myprog
```

この例では、プログラムで多くても 4 個のスレッドが作成されます。並列タスクが終了した後、スレッドは 50 ミリ秒間スピン待機します。その時間内にそのスレッドに新しいタスクが到着すると、スレッドはそのタスクを実行します。それ以外の場合、スレッドは新しいタスクが届くまでスリープ状態に入ります。

並列化されたプログラムをデバッグする

```
    real x / 1.0 /, y / 0.0 /
    print *, x/y
    end
```

```
    character  string*5, out*20
    double precision value
    external exception_handler
    i = ieee_handler('set', 'all', exception_handler)
    string = '1e310'
    print *, 'Input string ', string, ' becomes: ', value
    print *, 'Value of 1e300 * 1e10 is:', 1e300 * 1e10
    i = ieee_flags('clear', 'exception', 'all', out)
    end

    integer function exception_handler(sig, code, sigcontext)
    integer sig, code, sigcontext(5)
    print *, '*** IEEE exception raised!'
    return
    end
```

```
 *** IEEE exception raised!
 Input string 1e310 becomes:  Infinity
 Value of 1e300 * 1e10 is: Inf
 Note: Following IEEE floating-point traps enabled; see
ieee_handler(3M):
 Inexact;  Underflow;  Overflow;  Division by Zero;  Invalid
Operand;
Sun's implementation of IEEE arithmetic is discussed in
 the Numerical Computation Guide.
```

並列化されたプログラムをデバッグするには、新たな作業が必要になります。次に、その方法をいくつか示します。

デバッグの最初の手順

エラーの原因を特定するためにすぐに試してみることができる手順がいくつかあります。

- 並列化をオフにする。

  次の 2 つの方法があります。

  - 並列化オプションをオフにする。並列化を行わず、-O3 か -O4 を付けてコンパイルし、プログラムが正しく動作するかを確認します。
  - スレッド数を 1 に設定し、並列化オプションをオンにしてコンパイルします。つまり、環境変数 PARALLEL に 1 を設定してプログラムを実行します。

  問題が解決された場合は、問題の原因が複数のスレッドを使用していることにあることがわかります。

  - また、-C を付けてコンパイルし、添字の上下限を超えた配列の参照がないかを調べます。
  - -autopar を使用している場合、コンパイラが並列化すべきでないものを並列化していることもあります。

- -reduction をオフにする。

  -reduction オプションを使用している場合、合計縮約が発生し、わずかに異なる答えを出している可能性があります。このオプションをはずして実行してみてください。

- 個々のループの自動並列化オプションを選択しながらオフにするには、DOSERIAL 指令を使用します。

- fsplit または f90split を使用する。

  ユーザーのプログラムに多数のサブルーチンがある場合は、fsplit(1) を使用してサブルーチンを別個のファイルに分割します。次に、一部のサブルーチンに -parallel を付けて、一部には付けずにコンパイルし、f90 を使用して .o ファイルをリンクします。このリンクステップでは -parallel を指定する必要があります

  バイナリを実行し、結果を検証します。

  この手順を繰り返して、問題を 1 つのサブルーチンにしぼり込みます。

- -loopinfo を使用する。

  並列化されているループと、並列化されていないループを調べます。

- ダミーのサブルーチンを使用する。

何もしないダミーのサブルーチンや関数を作成します。並列化されているいくつかのループにこのサブルーチンへの呼び出しを挿入します。そして、コンパイルし直して、実行します。-loopinfo を使用して、どのループが並列化されているかを調べます。

正しい結果が得られるようになるまで、この処理を続けます。

- 明示的な並列化を使用する。

  並列化されている 2 つのループに C$PAR DOALL 指令を追加します。-explicitpar を付けてコンパイルして実行し、その結果を検証します。-loopinfo を使用して、どのループが並列化されているかを調べます。この方法で、並列化されたループに入出力文も追加できます。

  この手順を繰り返して、間違った結果の原因となるループを突き止めます。

注 – -explicitpar だけが必要な場合 (-autopar は必要でない場合) は、-explicitpar と -depend を付けてコンパイルしないでください。この方法は、-parallel を付けてコンパイルするのと同じことであり、この結果、-autopar が含まれてしまいます。

- ループを逆方向に逐次実行する。

  DO I=1,N を DO I=N,1,-1 で置き換えます。結果が異なるときは、データ依存性があることを示しています。

- ループインデックスを使用しない。

次のコードは:

```
    DO I=1,N
      ...
      CALL SNUBBER(I)
      ...
    ENDDO
```

次のように書き換える:

```
    DO I1=1,N
       I=I1
      ...
      CALL SNUBBER(I)
      ...
    ENDDO
```

dbx による並列コードのデバッグ

並列化されたループに dbx を使用するためには、一時的にプログラムを次のように書き直します。

- ループ本体を入れるファイルと、サブルーチンを入れるファイルを別々に分けます。
- オリジナルのルーチンでは、ループ本体を新しいサブルーチンの呼び出しで置き換えます。
- 新しいサブルーチンは、-g を付けて、並列化オプションを付けずにコンパイルします。
- 変更されたオリジナルのルーチンは、並列化オプションを付けて、-g を付けずにコンパイルします。

例 : 並列化されたプログラムで dbx が使用できるように、ループを手作業で変形します。

```
オリジナルコード
demo% cat loop.f
C$PAR DOALL
    DO i = 1,10
          WRITE(0,*) 'Iteration ', i
    END DO
    END
呼び出し元ループとサブルーチンとしてのループ本体の 2 つの部分に分割。
demo% cat loop1.f
C$PAR DOALL
    DO i = 1,10
           k = i
           CALL loop_body ( k )
    END DO
    END

demo% cat loop2.f
    SUBROUTINE loop_body ( k )
    WRITE(0,*) 'Iteration ', k
    RETURN
    END
呼び出し元ループを並列化オプションをつけてコンパイル。デバッグ用オプションはつけない。
demo% f95 -O3 -c -explicitpar loop1.f
サブルーチンをデバッグ用オプションをつけ、並列化せずコンパイル。
demo% f95 -c -g loop2.f
両方を a.out にリンク。
demo% f95 loop1.o loop2.o -explicitpar
dbx 制御下で a.out を実行し、ブレークポイントをループ本体のサブルーチンに設定。
demo% dbx a.out   ← さまざまな dbx のメッセージはここでは省略。
(dbx) stop in loop_body
(2) stop in loop_body
(dbx) run
Running: a.out
(process id 28163)
dbx はブレークポイントで停止。
t@1 (l@1) stopped in loop_body at line 2 in file
    "loop2.f"
    2       WRITE(0,*) 'Iteration ', k
k を表示。
(dbx) print k
k = 1               ← 1 以外のさまざまな値が考えられる。
(dbx)
```

関連文書

並列化に関する詳細な情報は以下を参照してください。

- 『Techniques for Optimizing Applications: High Performance Computing』、Rajat Garg、lya Sharapov著、サン・マイクロシステムズ Press Blueprint、2001
- 『High Performance Computing』、Kevin Dowd、Charles Severance 著、O'Reilly & Associates、第 2 版、1998
- 『Parallel Programming in OpenMP』、by Rohit Chandra 他著、Morgan Kaufmann Publishers、2001
- 『Parallel Programming』、by Barry Wilkinson著、Prentice Hall、1999
- Forte Developer の『OpenMP API ユーザーズガイド』

第11章

C と Fortran のインタフェース

この章では、Fortran と C の相互運用性に関する問題を取り上げます。ここで説明する内容は、Forte Developer Fortran 95 と C コンパイラの仕様のみに該当します。

互換性について

ほとんどの C と Fortran のインタフェースでは、次に示すことを正しく理解しておく必要があります。

- 関数とサブルーチンの定義と呼び出し
- データ型の互換性
- 引数の参照渡しと値渡し
- 引数の順番
- 手続き名 - 大文字、小文字、または末尾に下線 (_) 付き
- 正しいライブラリ参照をリンカーに渡す

また、一部の C と Fortran のインタフェースでは、次に示すことを正しく理解しておく必要があります。

- 配列の添字付けと順序
- ファイル記述子と `stdio`
- ファイルのアクセス権

関数とサブルーチン

「関数」という言葉の意味は、C と Fortran では異なります。状況によって、どちらの意味で解釈するかが重要です。

- C では、すべての副プログラムが関数です。それらの中には、NULL (void) 値を返すものも含めます。
- Fortran では、関数とは値を返すものであり、値を返さないものはサブルーチンといいます。

Fortran ルーチンから C 関数を呼び出す場合

- 値を返す C の関数は、Fortran から関数として呼び出します。
- 値を返さない C の関数は、Fortran からサブルーチンとして呼び出します。

C 関数から Fortran 副プログラムを呼び出す場合

- Fortran 副プログラムが関数の場合は、C から、対応するデータ型を返す関数として呼び出します。
- Fortran 副プログラムがサブルーチンの場合は、C から `int` (これは Fortran の `INTEGER*4` に対応します) または `void` を返す関数として呼び出します。Fortran のサブルーチンが選択戻りをする場合は 1 つの値が戻されます。この場合、`RETURN` 文にある式の値です。`RETURN` 文に式がない場合、または `SUBROUTINE` 文で選択戻りが 宣言されている場合、ゼロが戻されます。

データ型の互換性

この後に示す表では、Fortran 95 のデータ型と C のデータ型のデータサイズとデフォルトの境界整列を比較しています。いずれの表でも、以下の点に注意してください。

- C のデータ型 `int`、`long int`、`long` は、32 ビット環境では等価です (4 バイト)。しかし、64 ビット環境で -xarch=v9 または `v9a` を使用してコンパイルすると、`long` とポインタは 8 バイトになります。これは LP64 データモデルと呼ばれます。
- 64 ビット環境で -xarch=v9 または `v9a` を使用してコンパイルすると、`REAL*16` と `COMPLEX*32` は 16 バイト境界に揃えられます。
- 4/8 と示されている境界整列は、デフォルトでは 8 バイト境界を意味しますが、共通ブロックでは 4 バイト境界を意味しています。共通ブロックでの最大境界整列は 4 バイトです。
- 配列と構造体の要素および欄はそれぞれ互換性がなければいけません。
- 配列、文字列、構造体を値で渡すことはできません。

- 呼び出し側で %VAL(*arg*) を使用すると、Fortran 95 ルーチンから C ルーチンに値で引数を渡すことができます。C から Fortran 95 に値で引数を渡すことはできますが、Fortran ルーチンに、VALUE 属性とともに仮引数を宣言している明示的なインタフェースブロックがあることが条件となります。

Fortran 95 と C のデータ型

次の表では、Fortran 95 と C のデータ型を比較しています。ここでは、境界に影響したり、適用されるデフォルトのデータサイズを昇格させたりするコンパイルオプションを指定しないものとします。

表 11-1　データサイズと境界 - (バイト数での) 参照渡し (f95 と cc)

Fortran 90 のデータ型	C のデータ型	サイズ	境界
BYTE X	char x	1	1
CHARACTER x	unsigned char x ;	1	1
CHARACTER (LEN=n) x	unsigned char x[*n*] ;	*n*	1
COMPLEX x	struct {float r,i;} x;	8	4
COMPLEX (KIND=4) x	struct {float r,i;} x;	8	4
COMPLEX (KIND=8) x	struct {double dr,di;} x;	16	4/8
COMPLEX (KIND=16) x	struct {long double, dr,di;} x;	32	4/8/16
DOUBLE COMPLEX x	struct {double dr, di;} x;	16	4/8
DOUBLE PRECISION x	double x ;	8	4
REAL x	float x ;	4	4
REAL (KIND=4) x	float x ;	4	4
REAL (KIND=8) x	double x ;	8	4/8
REAL (KIND=16) x	long double x;	16	4/8/16
INTEGER x	int x ;	4	4

表 11-1　データサイズと境界 - (バイト数での) 参照渡し (f95 と cc) (続き)

Fortran 90 のデータ型	C のデータ型	サイズ	境界
INTEGER (KIND=1) x	signed char x ;	1	4
INTEGER (KIND=2) x	short x ;	2	4
INTEGER (KIND=4) x	int x ;	4	4
INTEGER (KIND=8) x	long long int x;	8	4
LOGICAL x	int x ;	4	4
LOGICAL (KIND=1) x	signed char x ;	1	4
LOGICAL (KIND=2) x	short x ;	2	4
LOGICAL (KIND=4) x	int x ;	4	4
LOGICAL (KIND=8) x	long long int x;	8	4

大文字と小文字

C と Fortran では、字種 (大文字/小文字) に関する扱いが異なります。

- C では字種に意味があり、大文字と小文字を別のものとして扱います。
- Fortran では、デフォルトでは字種に意味がありません。

f95 のデフォルトでは、副プログラム名を小文字に変換して、字種を無視します。つまり、文字列定数の中を除き、すべての大文字を小文字に変換します。

大文字と小文字に関する問題には、一般に次のような 2 つの解決策があります。

- C の副プログラムで、C の関数名をすべて小文字にします。
- -U オプションを付けて Fortran プログラムをコンパイルします。これは、コンパイラに、関数名と副プログラム名における既存の字種の区別をそのまま保持させるオプションです。

上記 2 つの解決策のどちらか 1 つを使用してください。両方を使用してはなりません。

この章の例のほとんどは、C の関数名に小文字だけを使用しています。 f95 の -U コンパイラオプションは使用していません。

ルーチン名の下線

Fortran コンパイラは、通常、入口定義と呼び出しの両方に現れる副プログラムの名前に下線 (_) を追加します。これによって、ユーザー割り当て名が同じである C の手続きや外部変数と区別します。名前がちょうど 32 文字である場合は、下線は追加されません。Fortran ライブラリのほとんどすべての手続き名には、ユーザーが割り当てるサブルーチンとの競合を減らすため、先頭に 2 つの下線が付けられています。

下線に関する問題には、一般に次の 3 つの解決策があります。

- C の関数で、下線を追加して関数名を変更します。
- `C()` プラグマを使用して、FORTRAN コンパイラに末尾の下線を省かせます。
- `f95` の `-ext_names` オプションを使用すると、下線を使用しないで外部名への参照をコンパイルできます。

これらの中のいずれか 1 つを使用してください。

この章の例は、`C()` コンパイラプラグマを使用して、下線をなくしています。`C()` プラグマ指令は、外部関数の名前を引数として取ります。これは、このような関数が C 言語で書かれていることを示します。このため、Fortran コンパイラは、通常は外部名に対して追加する下線を、当該の関数名には追加しません。特定の関数に対する `C()` 指令は、その関数への最初の参照より前に指定しなければなりません。また、そのような参照を含む副プログラムごとに指定しなければなりません。使用規則は次のとおりです。

```
      EXTERNAL ABC, XYZ      !$PRAGMA C( ABC, XYZ )
```

このプラグマを使用する場合は、C の関数のほうでは名前に下線を追加してはなりません。プラグマ指令については、『Fortran ユーザーズガイド』で説明しています。

引数の参照渡しと値渡し

一般的には、Fortran ルーチンは引数を参照で渡します。呼び出し時に、非標準関数の `%VAL()` に引数を入れると、呼び出し元のルーチンはその引数を値で渡します。

Fortran 95 で引数を値で渡す場合の標準的な方法は、VALUE 属性と INTERFACE ブロックを使用する方法です。199 ページの「データ引数の値渡し」を参照してください。

一般的には、C は引数を値で渡します。引数の前にアンパサンド記号 (&) を付けた場合は、C はその引数をポインタを使用して参照で渡します。配列と文字列に関しては、C でも常に参照で渡します。

引数の順序

文字列の引数の場合を除くと、Fortran と C は引数を同じ順序で渡します。ただし、各文字列引数については、Fortran ではさらに文字列の長さを示す引数も渡します。文字列長は、値で渡される C の `long int` の量と同じです。

引数の順序は次のとおりです。

- 各引数 (データであっても関数であっても) のアドレス
- 各文字引数に対する `long int` (文字列長の並び全体は、他の引数の並び全体の後にきます)

例 :

Fortran コードの一部	対応する C のコード
`CHARACTER*7 S`	`char s[7];`
`INTEGER B(3)`	`int b[3];`
…	…
`CALL SAM( S, B(2) )`	`sam_( s, &b[1], 7L ) ;`

配列の添字付けと順番

配列の添字付けと順番については Fortran と C とでは異なります。

配列の添字付け

C の配列は常にゼロから始まりますが、Fortran の配列はデフォルトでは 1 から始まります。この問題には、次のような 2 つの解決策があります。

- 前述の例のように、Fortran のデフォルトを使用します。このときは、Fortran の `B(2)` 要素は C の `b[1]` 要素と同義になります。

■ Fortran の配列 B を B(0) で始まるように指定します。

```
      INTEGER B(0:2)
```

このときは、Fortran の要素 B(1) が C の b[1] 要素と同義になります。

配列の順番

Fortran の配列は列主導の順番で、次のように格納されます。 A(3,2)

```
A(1,1)  A(2,1)  A(3,1)  A(1,2)  A(2,2)  A(3,2)
```

C の配列は行主導の順番で、次のように格納されます。 A[3][2]

```
A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]
```

1 次元の配列では、これによって問題は生じません。しかし、多次元の配列では、すべての参照と宣言における添字の順番と使用法に気をつけてください。なんらかの調整が必要になります。

たとえば、行列操作の一部を C で行い、残りを Fortran で行うのは混乱が生じる可能性があります。一方の言語で全体の配列をルーチンに渡し、そのルーチン内ですべての行列操作を実行すれば、混乱を避けることができます。

ファイル記述子と stdio

Fortran の入出力チャネルは、装置番号で表されます。プラットフォームとして使用している SunOS オペレーティングシステムは、装置番号ではなく、ファイル記述子を扱います。Fortran の実行時のシステムが装置番号からファイル記述子に変換するので、ほとんどの Fortran プログラムはファイル記述子を認識する必要はありません。

C プログラムの多くは、標準入出力 (stdio) と呼ばれるサブルーチンセットを使用しています。Fortran の入出力関数の多くは標準入出力を使用しており、これはオペレーティングシステムの入出力呼び出しを使用しています。このような入出力システムの特性の一部を次の表に示します。

表 11-1 Fortran と C の入出力の比較

	Fortran 装置	標準入出力のファイルポインタ	ファイル記述子
ファイルを開く	読み書き用に開く	読み取り用、書き込み用、読み書き両用、または追加用に開く。 OPEN(2) 参照	読み取り用、書き込み用、または読み書き両用に開く。
属性	書式付き、書式なし	常に書式なし、ただし、書式解釈ルーチンによる読み書きは可能	常に書式なし
探査	直接、順番	物理ファイルの表現が直接探査の場合は直接探査、ただし、常に順番に読み取り可。	物理ファイルの表現が直接探査の場合は直接探査、ただし、常に順番に読み取り可。
構造	記録	バイトストリーム	バイトストリーム
形式	任意の負でない 0 から 2147483647 までの整数	ユーザーのアドレス空間における構造体へのポインタ	0 から 1023 までの整数

ライブラリと f95 コマンドでのリンク

適切な Fortran および C ライブラリをリンクするためには、f95 コマンドを使用して、リンカーを起動します。

例 1 : コンパイラを使用してリンクします。

```
demo% cc -c someCroutine.c
demo% f95 theF95routine.f someCroutine.o  <- このコマンド行でリンクを実行
demo% a.out
 4.0 4.5
 8.0 9.0
demo%
```

Fortran 初期化ルーチン

f95 によりコンパイルされた主プログラムでは、プログラム起動時にライブラリのダミー初期化ルーチン f90_init を呼び出します。ライブラリのルーチンは、ダミーであり、何も処理しません。コンパイラにより生成される呼び出しでは、プログラムの引数と環境へのポインタが渡されます。これらの呼び出しにより、各自の C 言語ルーチンでプログラムの起動前に独自の方法でプログラムを初期化するときに使用できるソフトウェアのフックが提供されます。

このような初期化ルーチンの使用法の一つに、国際化された Fortran プログラムに対して setlocale を呼び出す方法があります。setlocale は、libc が静的にリンクされている場合機能しないので、libc に動的にリンクされる Fortran プログラムだけが国際化できます。

ライブラリの init ルーチンのソースコードは、次のとおりです。

```
void f90_init(int *argc_ptr, char ***argv_ptr, Char ***envp_ptr) {}
```

f90_init は、f95 主プログラムにより呼び出されます。各引数には、argc のアドレス、argv のアドレス、envp のアドレスが設定されます。

データ引数の参照渡し

Fortran ルーチンと C 手続きとの間でデータを渡す標準的な方法は、参照渡しです。C の手続きから見ると、Fortran の副プログラムまたは関数呼び出しは、すべての引数をポインタで表す手続き呼び出しのようになります。唯一異なる点は、Fortran が文字列と関数を引数として扱う方法と、CHARACTER*n 関数の戻り値として扱う方法です。

単純なデータ型

単純なデータ型の場合 (COMPLEX または CHARACTER 文字列以外)、次に示すように、C ルーチンにおいてそれぞれ関連する引数をポインタにより定義するか、または渡します。

表 11-2 単純型データを渡す

Fortran が C を呼び出す	C が Fortran を呼び出す

integer i
real r
external CSim
i = 100
call CSim(i,r)
...
--- 	
void csim_(int *i, float *r)
{
*r = *i;
}
	int i=100;
	float r;
	extern void fsim_(int *i, float *r);
	fsim_(&i, &r);
	...
	--- 	
	subroutine FSim(i,r)
	integer i
	real r
	r = i
	return
	end

複素数データ

Fortran の複素数データ項については、2 つの `float` または 2 つの `double` からなる 1 つの C の構造体へのポインタとして渡します。

表 11-3 複素数データを渡す

Fortran が C を呼び出す	C が Fortran を呼び出す

`complex w`
`double complex z`
`external CCmplx`
`call CCmplx(w,z)`
`...`
`----------`
`struct cpx {float r, i;};`
`struct dpx {double r,i;};`
`void ccmplx_(`
`struct cpx *w,`
`struct dpx *z)`
`{`
`w -> r = 32.;`
`w -> i = .007;`
`z -> r = 66.67;`
`z -> i = 94.1;`
`}`
	`struct cpx {float r, i;};`
	`struct cpx d1;`
	`struct cpx *w = &d1;`
	`struct dpx {double r, i;};`
	`struct dpx d2;`
	`struct dpx *z = &d2;`
	`fcmplx_( w, z );`
	`...`
	`---------------`
	`subroutine FCmplx( w, z )`
	`complex w`
	`double complex z`
	`w = (32., .007)`
	`z = (66.67, 94.1)`
	`return`
	`end`

64 ビット環境で、`-xarch=v9` でコンパイルすると、`COMPLEX` 値がレジスタに戻されます。

文字列

C と Fortran ルーチンとの間で文字列を渡すことは推奨できません。これは、標準的なインタフェースがないからです。ただし、次を考慮してください。

- すべての C 文字列は参照で渡される。

- Fortran の呼び出しは、引数リストにある character 型のすべての引数についてそれぞれもう 1 つの引数を渡します。この追加引数は、文字列の長さを渡すもので、値で渡される C の long int と同じです。ただし、これは実装方式に依存します。この文字列の長さを渡す追加引数は、呼び出しの明示的に指定した引数の後に現れます。

次の例で、文字列を引数とする Fortran 呼び出しを、対応する C のコードとともに示します。

表 11-4 CHARACTER 文字列を渡す

Fortran が C を呼び出す	C に対応する Fortran のコード

`CHARACTER*7 S`
`INTEGER B(3)`
`...`
`CALL CSTRNG( S, B(2) )`
`...`
	`char s[7];`
	`int b[3];`
	`...`
	`cstrng_( s, &b[1], 7L );`
	`...`

文字列の長さが呼び出されたルーチンで必要なければ、追加の引数は無視されます。ただし、Fortran では C のように明示的なヌル文字で文字列を自動的に終了させません。これは、呼び出し側のプログラムで追加する必要があります。

文字配列への呼び出しは、単一文字変数への呼び出しと似ています。配列の開始アドレスが渡され、それが使用する長さが配列の単一要素の長さになります。

1 次元配列

C では配列の添え字が 0 で始まります。

表 11-5 1 次元配列を渡す

Fortran が C を呼び出す	C が Fortran を呼び出す

integer i, Sum
integer a(9)
external FixVec
...
call FixVec (a, Sum)
...
--- 	
void fixvec_ (
int v[9], int *sum
)
{
int i;
*sum = 0;
for (i = 0; i <= 8; i++)
*sum = *sum +
v[i];
}
	extern void vecref_
	(int[], int
	*);
	...
	int i, sum;
	int v[9] = ...
	vecref_(v, &sum);
	...
	--- 	
	subroutine VecRef(v, total)
	integer i, total, v(9)
	total = 0
	do i = 1,9
	total = total + v(i)
	end do
	...

2 次元配列

C と Fortran とでは行と列が入れ替わります。

表 11-6 2 次元配列を渡す

Fortran が C を呼び出す	C が Fortran を呼び出す

REAL Q(10,20)
...
Q(3,5) = 1.0
CALL FIXQ(Q)
...
------------------------------ 	
void fixq_(float a[20][10])
{
...
a[5][3] = a[5][3] + 1.;
...
}
	extern void
	qref_(int[][10], int
	*);
	...
	int m[20][10] = ... ;
	int sum;
	...
	qref_(m, &sum);
	...
	------------------------------ 	
	SUBROUTINE QREF(A,TOTAL)
	INTEGER A(10,20), TOTAL
	DO I = 1,10
	DO J = 1,20
	TOTAL = TOTAL + A(I,J)
	END DO
	END DO
	...

構造体

C と Fortran 95 の構造型は、対応する成分間に互換性があれば、それぞれのルーチン間で相互に受け渡しすることができます (Forte Developer `f95` は古い `STRUCTURE` 文を受け付けます)。

表 11-7 古い FORTRAN 77 の `STRUCTURE` 記録を渡す

Fortran が C を呼び出す	C が Fortran を呼び出す

`  STRUCTURE /POINT/`
`       REAL X, Y, Z`
`  END STRUCTURE`
`  RECORD /POINT/ BASE`
`  EXTERNAL FLIP`
`  ...`
`  CALL FLIP( BASE )`
`  ...`
`------------------------------`
`struct point {`
`    float x,y,z;`
`};`
`void flip_( struct point *v )`
`{`
`       float t;`
`       t = v -> x;`
`       v -> x = v -> y;`
`       v -> y = t;`
`       v -> z = -2.*(v -> z);`
`}`
	`struct point {`
	`  float x,y,z;`
	`};`
	`void fflip_ ( struct point *)`
	`;`
	`  ...`
	`  struct point d;`
	`  struct point *ptx = &d;`
	`  ...`
	`  fflip_ (ptx);`
	`  ...`
	`------------------------------`
	`  SUBROUTINE FFLIP( P )`
	`  STRUCTURE /POINT/`
	`     REAL X,Y,Z`
	`  END STRUCTURE`
	`  RECORD /POINT/ P`
	`  REAL T`
	`  T = P.X`
	`  P.X = P.Y`
	`  P.Y = T`
	`  P.Z = -2.*P.Z`
	`  ...`

表 11-8 Fortran 95 構造体を渡す

Fortran 95 が C を呼び出す	C が Fortran 95 を呼び出す

TYPE point
SEQUENCE
REAL ::x, y, z
END TYPE point
TYPE (point) base
EXTERNAL flip
...
CALL flip(base)
...
---------- 	
struct point {
float x,y,z;
};
void flip_(struct point *v)
{
float t;
t = v -> x;
v -> x = v -> y;
v -> y = t;
v -> z = -2.*(v -> z);
}
	struct point {
	float x,y,z;
	};
	extern void fflip_ (
	struct point
	*) ;
	...
	struct point d;
	struct point *ptx = &d;
	...
	fflip_ (ptx);
	...
	---------- 	
	SUBROUTINE FFLIP(P)
	TYPE POINT
	SEQUENCE
	REAL ::X, Y, Z
	END TYPE POINT
	TYPE (POINT) P
	REAL ::T
	T = P%X
	P%X = P%Y
	P%Y = T
	P%Z = -2.*P%Z
	...

Fortran 95 規格では、記憶領域の並び順がコンパイルによって保持されるように、構造型の定義に SEQUENCE 文を含めなければならないことになっています。

ポインタ

FORTRAN 77 (Cray) のポインタは、ポインタへのポインタとして C ルーチンに渡すことができます。これは、Fortran ルーチンが引数を参照で渡すからです。

表 11-9 FORTRAN 77 (Cray) の POINTER を渡す

Fortran が C を呼び出す	C が Fortran を呼び出す
REAL X POINTER (P2X, X) EXTERNAL PASS P2X = MALLOC(4) X = 0. CALL PASS(P2X) ...	extern void fpass_(float**); ... float *p2x; ... fpass_(&p2x) ; ...
----------	----------

void pass_(p)
float **p;
{
**p = 100.1;
}
	SUBROUTINE FPASS (P2X)
	REAL X
	POINTER (P2X, X)
	X = 0.
	...

C ポインタは Fortran 95 のスカラーポインタとは互換性がありますが、配列ポインタとは互換性がありません。

Fortran 95 がスカラーポインタで C を呼び出す

Fortran 95 ルーチン:

```
 INTERFACE
     SUBROUTINE PASS(P)
      REAL, POINTER :: P
     END SUBROUTINE
 END INTERFACE

 REAL, POINTER :: P2X
 ALLOCATE (P2X)
 P2X = 0
 CALL PASS(P2X)
 PRINT*, P2X
 END
```

C ルーチン:

```
 void pass_(p);
  float **p;
  {
    **p = 100.1;
  }
```

Cray ポインタと Fortran 95 ポインタの主な違いは、Cray ポインタには常にターゲットが指定されることです。多くの場合、Fortran 95 ポインタを宣言すると、ターゲットは自動的に特定されます。また、呼び出された側の C ルーチンにも明示的な `INTERFACE` ブロックが必要です。

配列または部分配列への Fortran 95 ポインタを渡すには、次の例のように特定の INTERFACE ブロックが必要です。

Fortran 95 ルーチン:

```
INTERFACE
  SUBROUTINE S(P)
    integer P(*)
  END SUBROUTINE S
END INTERFACE
integer, target :: A(0:9)
integer, pointer :: P(:)
P => A(0:9:2) !! ポインタは A の要素を 1 つおきに選択
call S(P)
...
```

C ルーチン

```
void s_(int p[])
{
   /* 中央の要素を変更する */
   p[2] = 444;
}
```

C ルーチン S は Fortran 95 ルーチンでありません。このため、INTERFACE ブロックでは、想定された形 (integer P(:)) でルーチン S を定義することはできません。C ルーチンが配列の実際のサイズを知る必要がある場合は、配列を引数として C ルーチンに渡す必要があります。

C と Fortran では添字付けの方法が異なり、C 配列の添字は 0 から始まることに注意してください。

データ引数の値渡し

Fortran 95 のプログラムでは、C から呼び出されるときに仮引数の VALUE 属性を使用し、Fortran 95 から呼び出す C ルーチンのために INTERFACE ブロックを追加する必要があります。

表 11-10 C と Fortran 95 の間で単純なデータ要素を渡す

Fortran 95 が C を呼び出す	C が Fortran 95 を呼び出す

PROGRAM callc
INTERFACE
  INTEGER FUNCTION crtn(I)
!$pragma C(crtn)
  INTEGER, VALUE, INTENT(IN) ::
I
  END FUNCTION crtn
END INTERFACE
 	
  M = 20
  MM = crtn(M)
  WRITE (*,*) M, MM
END PROGRAM
---------- 	
--
int crtn(int x)
{
  int y;
  printf("%d input \n", x);
  y = x + 1;
  printf("%d returning \n",y);
  return(y);
}
---------- 	
--
Results:
20 input
21 returning
 20 21
	#include <stdlib.h>
	int main(int ac, char *av[])
	{
	  to_fortran_(12);
	}
	
	---------- 	
	-
	
	SUBROUTINE to_fortran(i)
	 INTEGER, VALUE :: i
	 PRINT *, i
	 END
	
	---------- 	
	-

C ルーチンを実引数として別のデータ型で呼び出す場合は、INTERFACE ブロックに !$PRAGMA IGNORE_TKR I を含めることによって、実引数と仮引数の型、種類、ランクの一致をコンパイラから要求されないようにします。

古い FORTRAN 77 での値による呼び出しは、単純型のデータでだけ、FORTRAN 77 ルーチンが C のルーチンを呼び出す場合に限り使用できました。C のルーチンから FORTRAN 77 ルーチンを呼び出して、値で引数を渡す方法はありません。配列、文字列、構造体は、参照で渡すようにしてください。

FORTRAN 77 ルーチンから C ルーチンに値を渡すには、非標準の Fortran 関数 `%VAL(`*arg*`)` を呼び出しの中で引数として使用してください。

次の例では、FORTRAN 77 ルーチンが値により `x` を渡し、参照により `y` を渡しています。C のルーチンは `x` と `y` を増分しましたが、`y` だけが変更されます。

Fortran が C を呼び出す

Fortran ルーチン*:*

```
  REAL x, y
  x = 1.
  y = 0.
  PRINT *, x,y
  CALL value( %VAL(x), y)
  PRINT *, x,y
  END
```

C ルーチン*:*

```
void value_( float x, float *y)
{
  printf("%f, %f\n",x,*y);
  x = x + 1.;
  *y = *y + 1.;
  printf("%f, %f\n",x,*y);
}
```

コンパイルと実行による出力*:*

```
     1.00000  0.    Fortran からの x と y
1.000000, 0.000000  C からの x と y
2.000000, 1.000000  C からの新しい x と y
     1.00000  1.00000  Fortran からの新しい x と y
```

値を戻す関数

BYTE、INTEGER、REAL、LOGICAL、DOUBLE PRECISION、または REAL*16 型の値を戻す Fortran 関数は、互換性のある型を戻す C の関数と等価です (表 11-1 を参照)。文字関数の戻り値には引数が 2 つ追加され、複素数関数の戻り値には引数が 1 つ追加されます。

単純型データを戻す

次の例は REAL または float 値を戻します。BYTE、INTEGER、LOGICAL、DOUBLE PRECISION、および REAL*16 も同じような方法で扱われます。

表 11-11 REAL または float の値を戻す関数

Fortran が C を呼び出す	C が Fortran を呼び出す

`  real ADD1, R, S`
`  external ADD1`
`  R = 8.0`
`  S = ADD1( R )`
`  ...`
`------------------------------`
`float add1_( pf )`
`float *pf;`
`{`
`    float f ;`
`    f = *pf;`
`    f++;`
`    return ( f );`
`}`
	`float r, s;`
	`extern float fadd1_() ;`
	`r = 8.0;`
	`s = fadd1_( &r );`
	`  ...`
	`------------------------------`
	`  real function fadd1 (p)`
	`  real p`
	`  fadd1 = p + 1.0`
	`  return`
	`  end`

複素数データを戻す

COMPLEX または DOUBLE COMPLEX を戻す Fortran 関数は、メモリーにある戻り値を指す追加の引数を最初の引数として持つ C の関数と同じです。Fortran 関数と対応する C 関数の一般的な形式は次のとおりです。

Fortran 関数	C 関数
`COMPLEX FUNCTION CF(`*a1, a2, ..., an*`)`	`cf_ (`*return, a1, a2, ..., an*`)` `struct { float r, i; } *`*return*

表 11-12 COMPLEX を戻す関数

Fortran が C を呼び出す	C が Fortran を呼び出す

`  COMPLEX U, V, RETCPX`
`  EXTERNAL RETCPX`
`  U = ( 7.0, -8.0)`
`  V = RETCPX(U)`
`  ...`
`----------`
`struct complex { float r, i;`
`};`
`void retcpx_( temp, w )`
`struct complex *temp, *w;`
`{`
`    temp->r = w->r + 1.0;`
`    temp->i = w->i + 1.0;`
`    return;`
`}`
	`struct complex { float r, i;`
	`};`
	`struct complex c1, c2;`
	`struct complex *u=&c1, *v=&c2;`
	`extern retfpx_();`
	`u -> r = 7.0;`
	`u -> i = -8.0;`
	`retfpx_( v, u );`
	`  ...`
	`----------`
	`  COMPLEX FUNCTION RETFPX(Z)`
	`  COMPLEX Z`
	`  RETFPX = Z + (1.0, 1.0)`
	`  RETURN`
	`  END`

64 ビット環境で、-xarch=v9 でコンパイルすると、COMPLEX 値が浮動小数点レジスタに戻されます。COMPLEX と DOUBLE COMPLEX はそれぞれ %f0 と %f1 に、COMPLEX*32 は %f0、%f1、%f2、%f3 に戻されます。これらのレジスタは C プログラムでは直接アクセスできないので、この場合の SPARC V9 プラットフォームでは Fortran と C 言語の間で相互操作性は実現されません。

CHARACTER 文字列を戻す

C と Fortran ルーチンの間で文字列を渡すことは推奨できません。ただし、Fortran の文字列の値を持つ関数は、データアドレスとデータ長の 2 つの引数がはじめに追加された C の関数と同じです。Fortran 関数と対応する C 関数の一般的な形式は次のとおりです。

Fortran 関数	C 関数

`CHARACTER*`*n* `FUNCTION C(`*a1, ..., an*`)`	`void c_ (`*result, length, a1, ..., an*`)`
	`char` *result*`[ ];`
	`long length;`

次に例を示します。

表 11-13 CHARACTER 文字列を戻す関数

Fortran が C を呼び出す

```
  CHARACTER STRING*16, CSTR*9
  STRING = ' '
  STRING = '123' // CSTR('*',9)
  ...
------------------------------
 void cstr_( char *p2rslt,
               long rslt_len,
               char *p2arg,
               int *p2n,
               long arg_len )
{ /* 引数の n 個のコピーを戻す */
     int count, i;
     char *cp;
     count = *p2n;
     cp = p2rslt;
     for (i=0; i<count; i++) {
         *cp++ = *p2arg ;
     }
}
```

C が Fortran を呼び出す

```
void fstr_( char *, long,
        char *, int *, long );
char sbf[9] = "123456789";
char *p2rslt = sbf;
int rslt_len = sizeof(sbf);
char ch = '*';
int n = 4;
int ch_len = sizeof(ch); …
/* sbf に ch の n 個のコピーを作る
 */
  fstr_( p2rslt, rslt_len,
           &ch, &n, ch_len );
 ...
------------------------------
 FUNCTION FSTR( C, N)
 CHARACTER FSTR*(*), C
 FSTR = ''
 DO I = 1,N
    FSTR(I:I) = C
 END DO
 FSTR(N+1:N+1) = CHAR(0)
 END
```

この例では、C 関数と呼び出し側の C ルーチンは、リストの最初に 2 つの引数 (結果として戻される文字列へのポインタと文字列長) が、そして、リストの最後に 1 つの追加引数 (文字列引数の長さ) が追加されています。C から呼び出された Fortran ルーチンでは最後のヌル文字を明示的に追加する必要があることに注意してください。Fortran の文字列は、デフォルトでは NULL 終端されません。

名前付き COMMON

Fortran の名前付き COMMON は、大域的構造体を使用して C の中で代替できます。

表 11-14 名前付き COMMON

Fortran のCOMMON 定義	C の COMMON 定義

COMMON /BLOCK/ ALPHA,NUM
...
 	
	extern struct block {
	float alpha;
	int num;
	};
	extern struct block block_ ;
	
	main ()
	{
	...
	block_.alpha = 32.;
	block_.num += 1;
	...
	}

C ルーチンにより確立された外部名は、Fortran プログラムにより作成されたブロックとリンクさせるために下線で終了しなければなりません。C 指令 #pragma pack は Fortran のときと同じ埋め込みが必要な場合があります。

f95 は、デフォルトで、共通ブロックのデータを最大で 4 バイトの境界に配列します。共通ブロック内のすべてのデータ要素の自然配列取得し、デフォルトの構造配列と一致させるには、Fortran ルーチンをコンパイルする際にオプション -aligncommon=16 を使用します。

Fortran と C との入出力の共有

Fortran の入出力と C の入出力を混合すること (C ルーチンと Fortran ルーチンの両方から入出力呼び出しを発行すること) は推奨できません。すべて Fortran の入出力で行うか、すべて C の入出力で行うかのどちらかに統一するのが安全です。

Fortran の入出力ライブラリは、大部分が C の標準入出力ライブラリに追加する形で実装されています。Fortran プログラムで開いた装置はすべて、標準入出力のファイル構造と対応付けられています。stdin、stdout、stderr のストリームに関しては、ファイル構造を明示的に参照する必要がなく、共有できます。

Fortran の主プログラムが入出力を行うために C を呼び出す場合は、プログラム起動時に Fortran の入出力ライブラリを初期化して、装置 0、5、6 をそれぞれ stderr、stdin、stdout に接続します。ファイル記述子を開いて入出力を実行する場合、C の関数は Fortran の入出力環境を考慮しなければなりません。

しかし、C の主プログラムが Fortran の副プログラムを呼び出して入出力を行う場合は、装置 0、5、6 を stderr、stdin、stdout に接続するための Fortran 入出力ライブラリの自動初期化は行われません。この接続は通常 Fortran の主プログラムにより行われます。Fortran 主プログラムによって通常の入出力の初期化が行われないと、Fortran 関数が stderr ストリーム (装置 0) を参照しようとしたときに、出力は stderr ストリームではなく、fort.0 に書き込まれます。

C の主プログラムは、プログラムの先頭でライブラリルーチン f_init() を呼び出すことにより Fortran 入出力を初期化し、装置 0、5、6 を事前に接続することができます。また、必要に応じて終了時に f_exit() を呼び出すこともできます。

たとえ主プログラムが C で書かれていても、f95 でリンクしなければならないことに注意してください。

選択戻り (あまり使用されません)

FORTRAN 77 の選択戻り機能はすでに廃止されているので、移植上の問題がなければ使用しないでください。選択戻りに対応する機能は C にはありません。したがって、問題は C のルーチンが選択戻りを持つ Fortran のルーチンを呼び出す場合だけです。Forte Developer Fortran 95 は FORTRAN 77 の選択戻りを受け付けますが、これを利用することはお勧めしません。

選択戻りは、RETURN 文の式の int 値を戻します。これは実装方式にかなり依存するため、使用しないでください。

表 11-15 選択戻り (あまり使用されません)

C が Fortran を呼び出す	例の実行

`int altret_ ( int * );`
`main ()`
`{`
`  int k, m ;`
`  k =0;`
`  m = altret_( &k ) ;`
`  printf( "%d %d\n", k, m);`
`}`
`------------------------------`
`SUBROUTINE ALTRET( I, *, *)`
`  INTEGER I`
`  I = I + 1`
`  IF(I .EQ. 0) RETURN 1`
`  IF(I .GT. 0) RETURN 2`
`  RETURN`
`END`
	`demo% cc -c tst.c`
	`demo% f95 -o alt alt.f tst.o`
	`alt.f:`
	`  altret:`
	`demo% alt`
	`1 2`
	
	C ルーチンは、*Fortran* ルーチンが RETURN 2 を実行するため、戻り値 *2* を受け取ります。

索引

G

I

L

M

N

O

P

R

S

T

U

V

X

Y

Z

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

へ

ほ

ま

る

れ

ろ