PGI® コンパイラ使用ガイド

Parallel Fortran, C and C++ for Scientists and Engineers

2017 年 6 月版 (Rev. 9.0)

株式会社 ソフテック HPC ソリューション部

(http://www.softek.co.jp/SPG/Pgi/)

SofTek

この使用ガイドは、PGI コンパイラを初めて利用する際のコンパイラの使用方法を簡単に説明した資料です。様々な活用シーンにおける一般的に使用するコンパイル・オプションと PGI ツールの使用法について説明しています。この内容は、今後、随時更新する予定です。なお、ここで記述したコンパイラ・オプションは特に断らない限り、PGI 6.1 ～ PGI 2017 バージョンのものを使用しています。これ以前のバージョンでは、一部、オプションとして存在しないものもありますのでご注意ください。

目 次

株式会社ソフテック
〒 154-0004 東京都世田谷区太子堂 1-12-39
http://www.softek.co.jp

1 はじめに

この章では、PGI コンパイラのコマンドライン上での使用方法と、その際に必要とする入力ファイル、生成される出力ファイルに関して簡単に説明します。また、PGI コンパイラが提供する並列化機能の概略を説明します。

1.1 コマンドレベル PGI コンパイラの使用

PGI コンパイラにおける Fortran(F77/F2003)、HPF、C、C++プログラムをコンパイルするためのコマンドは、それぞれ、pgf77、pgfortran (pgf95,pgf90 も同じコマンドとして認識する)、pgcc、pgc++(Windows 版の pgcpp/pgCC コマンドは廃止)となります。

表 1 – 1 PGI コンパイラとツールのコマンド

コンパイラ名	コンパイラ言語	コマンド名
PGF77	FORTRAN77（F77）専用	pgf77
PGFORTRAN	FORTRAN 77 、Fortran 90/95/2003 全てをカバー（一部 F2008 取込）	pgfortran pgf95,pgf90
PGCC	ANSI C11and K&R C （C11 準拠）	pgcc
PGC++	(Windows 版の C++は、2016 年 1 月に終息)	
(Linux/OS X only)	ANSI C++14、GNU g++と ABI 互換性がある C++	pgc++
PGDBG	OpenMP 対応ソースコードデバッガ (DBG)	pgdbg
PGPROF	OpenMP 対応性能解析プロファイラ (PROF)	pgprof

※pgc++ for Linux は、PGI 13.1 以降、pgc++ for OS X は PGI 15.1 以降
以前の pgcpp(pgCC)コンパイラは、PGI 2015 版で終了。PGI 2016 以降は終息。

一般的な実行モジュールを生成するまでの流れは、以下のようになります。

- 必要な場合、ソーステキストファイルのプリプロセス処理を実行します
- ソーステキストの構文をチェックします
- アセンブリ言語ファイルを作成します
- 後続のアセンブラ、リンクステージへ制御を渡します

例えば、次のような内容の簡単な Fortran プログラム hello.f があります。

```
Print *, "hello"
End
```

このプログラムファイルを pgfortran コンパイラで次のようにコンパイルします。

```
PGI$ pgfortran hello.f
PGI$
```

生成される実行モジュール名のデフォルトは、Linux 上では a.out と言う名前となります（Windows システムでは、コマンドラインに指定した最初のファイル名の接頭の名前が使用され、hello.exe と言う実行モジュール名になります)。また、明示的に -o オプションで指定するファイル名が、実行形式モジュールファイルとなります。プログラムの実行は、以下の例の場合、hello (Windows 上では hello.exe)と言う実行形式モジュールファイルをコマンドラインに指定して Return あるいは Enter キーを押します。

```
PGI$ pgfortran –o hello hello.f
```

```
PGI$ ./hello
  hello
PGI$
```

1.2 コンパイラのコマンドラインの指定方法

PGI の F77, F2003, C, C++ のコンパイラを使用する際のオプションを以下に示しました。以下の例は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。なお、各オプションの詳細は、付録 PGI Compiler Option 一覧 をご覧ください。

```
pgfortran -[options] [path] filename
pgfortran –fastsse  -Minfo=all –Mvect   -L /opt/lib   user.a test.f  （例）
          [options]                     [path]    [filename]
```

必要とするオプションを -[option] 形式でブランクを空けて指定します。また、-M オプションは、最適化オプションを詳細に指定するものであり、-M に引き続きブランクを空けずにフラグを指定します。なお、-M にさらに、サブフラグがある場合は、-M[flag]={subflag} の形式で指定します。サブフラグを指定しない場合は、コンパイラの default 設定のサブフラグが使用されます。

[options]	各コンパイル・オプションを指定する。指定順序は基本的に制約はない 但し、ライブラリ・パス等の順序は重要であり、その順位で反映される
[path]	リンカへのライブラリ等のパスを指定する。指定しない場合は、コンパイル処理しているカレントディレクトリが使用される。
[filename]	ソースファイル、オブジェクトファイル、アセンブリ言語ファイル等を指定する

なお、主要なコンパイル・オプションの使用方法については 2 章をご覧ください。

1.3 ファイル名に関するコンベンション（慣例的に使用される名前）

PGI コンパイラのコマンドライン上に指定されるファイル名について説明します。

■ 入力ファイル

PGI コンパイラで使用される入力ファイル名形式の意味とコンパイラが行う処理形態について、以下の表に纏めました。ファイル名の接尾辞（拡張子）の違いにより、プリプロセス処理を行うファイルであるかの判断を行います。

表 1 - 2　入力ファイル名

ファイル名の形式	ファイルの意味と処理形態
filename.f filename.for	Fortran ソースファイル（72 カラム固定形式と認識）
filename.F	マクロあるいはプリプロセッサ・ディレクティブを含む Fortran ソースファイル（プリプロセッシングを行います）
Filename.FOR	マクロあるいはプリプロセッサ・ディレクティブを含む Fortran ソースファイル（プリプロセッシングを行います）
filename.F90 filename.F95	マクロあるいはプリプロセッサ・ディレクティブを含む Fortran 77/90/95/03 ソースファイル（プリプロセッシングを行います）

filename.f90	自由書式形式の Fortran 77/90/95/03 ソースファイル
filename.f95	自由書式形式の Fortran 77/90/95/03 ソースファイル
filename.cuf	CUDA Fortran 自由書式形式ソースファイル
filename.CUF	マクロあるいはプリプロセッサ・ディレクティブを含む CUDA Fortran 自由書式形式ソースファイル
filename.c	マクロあるいはプリプロセッサ・ディレクティブを含む C ソースファイル（プリプロセッシングを行います）
filename.i	プリプロセス後の C あるいは C++ ソースファイル
filename.C	マクロあるいはプリプロセッサ・ディレクティブを含む C++ソースファイル（プリプロセッシングを行います）
filename.cc filename.cpp	マクロあるいはプリプロセッサ・ディレクティブを含む C++ソースファイル（プリプロセッシングを行います）
filename.cu	NVIDIA CUDA C/C++ソースファイル
filename.S	プリプロセッサ・ディレクティブを含むアセンブリ言語ファイル (PGI 7.0 以降)
filename.s	アセンブリ言語ファイル
filename.o	オブジェクトファイル（Linux/OS X）
filename.obj	オブジェクトファイル（Windows）
filename.a	オブジェクトファイルのライブラリ（Linux/OS X）
filename.lib	オブジェクトファイルの Static-linked ライブラリ（Windows）
filename.so	シェアード（共有）オブジェクトファイル(Linux systems only)
filename.dll	オブジェクトファイルのダイナミック・リンク・ライブラリ (Windows systems only)
filename.dylib	オブジェクトファイルのダイナミック・リンク・ライブラリ (OS X only)

■ 出力ファイル

PGI コンパイラによって生成される実行形式モジュール名は、デフォルトでは a.out (Linux の場合)、Windows 版の場合は、コマンドラインに指定した最初に指定したファイル名の接頭名が使用され、*.exe の形式となります。また、明示的に –o オプションで実行形式モジュールのファイル名を指定することもできます。

PGI コンパイラでは、以下の表に示したコンパイル・オプションの設定により、それに応じたファイルを出力してコンパイル処理を停止します。途中結果ファイルの出力を行いたい時に使用します。以下の表では、各オプションの停止ステージと有効な入力ファイルと出力されるファイルを説明しています。

表 1−3　コンパイル・オプションによる入出力ファイル

オプション	停止ステージ	入力ファイル名	出力ファイル名
-E	プリプロセス後	ソースファイル (Fortran の場合、*.F 名であること)	プリプロセス処理結果は標準出力へ
-F	プリプロセス後	ソースファイル、*.F 名であること （このオプションは pgf77/ pgfortran/ pghpf のみ有効）	プリプロセス処理後、-- .f に書き込まれる
-P	プリプロセス後	ソースファイル (このオプションは pgcc,	プリプロセス処理後、--.i に書き込まれる

		pgc++のみ有効)	
-S	コンパイル処理後	ソースファイルあるいは、プリプロセス後のファイル	アセンブリ言語ファイル--.s
-c	アセンブル処理後	ソースファイル、プリプロセス後のファイル、あるいは、アセンブリ言語ファイル	リンケージされていないオブジェクトファイル --.o or .obj
none	リンケージ終了後	ソースファイル、プリプロセス後のファイル、アセンブリ言語ファイル、オブジェクトファイル、ライブラリ・ファイル	実行モジュール形式ファイル -- a.out or .exe

もし、複数の入力ファイルを指定するか、あるいはオブジェクトファイル名を指定しない場合、コンパイラは、入力ファイル名に対応して次に示す形式のデフォルトの出力ファイル名を使用します。ここで、*filename* は拡張子のない入力ファイル名を意味します。なお、これらの生成されるファイルは、コンパイルを行っている「カレントディレクトリ」に出力されます。既存の同じファイル名が存在する場合は、上書きされます。

filename.**f**　プリプロセス処理済みファイル（-F オプションを指定したコンパイル）
filename.**lst**　–Mlist オプションを指定して作成されたリスティング・ファイル
filename.**o** or *filename*.**obj**　–c オプションを指定して作成されたオブジェクトファイル
filename.**s**　–S オプションを指定して作成されたアセンブリ言語ファイル

次の例で、出力ファイルの拡張子の説明をしてみましょう。

pgfortran –c proto.f proto1.F

この例では、出力ファイルとして、proto.o 並びに proto1.o と言うバイナリ形式のオブジェクトファイルが生成されます。なお、コンパイルの前に、proto1.F ファイルは .F 拡張子であるためプリプロセス処理がなされます。

1.4 Fortran プログラムのファイル名

ここでは、ファイル名のサフィックスによって、コンパイラは、どのような Fortran 構文形式として認識するかと言うことについて説明します。前項 1.3 の「入力ファイル」の項で説明したように、Fortran プログラムファイル名のサフィックスによって、コンパイラの構文認識と処理形式が異なります。ファイル名のサフィックスの違いにより、コンパイラは、そのファイルをどのように認識するかと言う点について以下の表に纏めました。

表 1 - 4　Fortran のファイル名形式

サフィックス	コンパイラが認識するファイルの意味	プログラム記述形式
*.f	一般的な Fortran ファイル(F77,F90/F95/03 問わず)と認識	72 カラム固定書式形式 継続行は 6 カラム目に指示
*.F	一般的な Fortran ファイル、かつプリプロセス処理を行うファイルと認識	72 カラム固定書式形式 継続行は 6 カラム目に指示
*.f90 or *.f95	Fortran77/90/95/2003 構文形式ファイルと認識	自由書式形式として認識する
*.F90 or F95	Fortran77/90/95/2003 構文形式、かつプリプ	自由書式形式として認識する

	ロセス処理を行うファイルと認識	

(注意：ファイル名が*.f90,*.f95 形式で、その中身が FORTRAN77 構文であっても、問題なくその Fortran 構文は解釈されます)

例えば、test.f は、*.f 形式ですので、プログラム記述上の構文エリアは従来の FORTRAN77 互換のために 72 カラムの固定書式形式に制限されます。一方、test.f95 の場合は、*.f95 形式ですので、デフォルトは自由フォーマット形式と認識され、構文記述エリアの 72 カラム制限はなくなります。よく生じるエラーとして、固定形式の 72 カラム以上にプログラムが記述されたファイルがあり、そのファイル名が *.f (*.F) 形式の場合、以下のような構文エラーを引き起こす場合があります。この場合、73 カラム以上は無視されますので、Fortran 構文上の括弧が閉じられていないと言うエラーとなります。

```
$ pgfortran –Minfo test.f
PGF90-S-0023-Syntax error - unbalanced parentheses (test.f: 50)
  0 inform,   0 warnings,   1 severes, 0 fatal for MAIN
```

このような 72 カラムの制限に関して、プログラムを修正しないで回避するには、いくつかの方法があります。

(1) *.f ファイル形式のまま、コンパイル・オプションで回避する

① pgfortran コマンドを使用して、132 カラムまで認識（-Mextend）するようにコンパイラに指示するオプション -Mextend を付加する

```
$ pgfortran -Mextend test.f
```

② pgfortran コマンドで、明示的に F90 自由フォーマット形式のファイルであることをコンパイラに指示する。これは、-Mfree を付加すると可能です。但し、もし、プログラムが 72 カラム固定形式でこの-Mfree を指定した場合、プログラム行の継続は、F90 書式の「&」に変更しなければなりません。（6 カラム目の継続カラムは使用出来ない）

```
$ pgfortran -Mfree test.f
```

(2) ファイル名のサフィックスを *.f から *.f90（*.f95） に変更する。f90,f95,F95,F90 のサフィックスファイル名は、72 カラムの制約がない自由記述形式を許しています。但し、プログラム行の継続は F90 書式の「&」に変更しなければなりません。（6 カラム目の継続カラムは使用出来ない）

```
$ pgfortran test.f90
```

一般的には、72 カラム固定形式のプログラムの場合、簡単にコンパイラが 132 カラムまで認識できるようにするためには、-Mextend を利用することをお勧めします。なお、pgf77 は、純然たる FORTRAN77 構文のみを受け付けるコンパイラですので、一般には、pgfortran を使用することをお勧めします。なお、古い、レガシーな F77 プログラムでは、pgf77 でコンパイルする方が、FORTRAN77 規約外の言語方言をより解釈できる場合があります。

（強制的なプリプロセス処理の方法）

プリプロセス処理を行わないサフィックス (***.f ,**.f90 等)のファイルを明示的に cpp 形式のプリプロセス処理を行う方法。以下のように、コンパイル・オプション –Mpreprocess を指定する。

```
$ pgfortran  -Mpreprocess  test.f
```

1.5 PGI コンパイラが提供する並列化機能

PGI の F77, F2003, C, C++ の全ての言語コンパイラには、マルチ（コア）・プロセッサに対応する自動並列化機能および、OpenMP を用いた並列化コンパイル機能が実装されております。また、2009 年には、GPU をアクセラレータとして構成するハードウェアに対する PGI アクセラレータ機能（ディレクティブでコンパイラに GPU 上の並列化を行う場所を指示する方法）も、コンパイラメーカとして業界で初めて提供しました。この機能は、2011 年策定された OpenACC 規約のベースとなりました。さらに、マルチプロセス用の並列化である MPI ライブラリを利用した実行モジュールも簡単に作成できます。以下、各並列化方法を簡単に記述致します。なお、並列化のためのコンパイラ・オプション設定等の詳細は 2 章においても説明します。

以下に述べる三つの並列化方法のうち、簡単に実行でき、ある程度の性能が得られるのは、"-Mconcur" オプションによる自動並列化です（プログラム特性によっては性能が得られない場合もあります）。"-Minfo" オプションを付けると、プログラム中のどの部分が並列化されたか分かりますので、更に性能を向上させたい場合は、さらに OpenMP 指示行を入れてみるなどの指示行ベースの並列化を行ってみることができます。なお、自動並列化機能と OpenMP 指示行による並列化の混在も可能です。

[用語解説]

マルチコアプロセッサ

一つの CPU ダイ（あるいは CPU ソケット）の中に複数の独立したプロセッサユニット（コア）を有するプロセッサ。

■ マルチコア上での自動並列化

コンパイル時にコンパイル・オプション **"-Mconcur"** を付けます。また、作成された実行形式モジュールを実行時に、環境変数 OMP_NUM_THREADS（あるいは NCPUS）に使用する並列スレッド数を設定致します。この並列化は、コンパイラが、プログラム中で並列化可能な部分（DO / for ループ部分等）の依存性解析を行い、可能であれば並列化を行います。そのため、並列化による効果が少ない場合、並列化による性能向上が図れない場合があります（逆に、プログラム特性によっては性能が落ちる場合もありますので、性能の検証を行うことが必要です）。この並列化で指定できるスレッド数（CPU コア）の上限は 256 までです。

（例）

```
$ pgfortran –fastsse -Minfo -o test -Mconcur test.f
$ export OMP_NUM_THREADS =2 （bash 使用時で、CPU/スレッド数 2 を指定）
$ ./test
```

■ OpenMP 指示行による並列化

ユーザプログラム中に OpenMP 指示行 （http://www.openmp.org ご参照下さい）を挿入することで、ユーザがディレクティブ（C の場合はプラグマ）で指定した部分の並列化を行います。 コンパイル時のコンパイル・オプションとして、**"-mp"** を付けます。 作成された実行形式モジュールを実行時に、環境変数 OMP_NUM_THREADS(あるいは NCPUS) に並列スレッド数を設定します。特徴としては、自動並列化に較べ、並列化の場所をユーザが明示的に指定し最適化できるので、性能がより向上することが挙げられます。ただし OpenMP 指示行の宣言子（宣言句）を理解していないと、正常に動作しないプログラムになる場合があります（性能低下、計算結果が異なる等）。OpenMP 宣言子（宣言句）は、コンパイラに対して補足情報（あるいはヒント）を与えるためのものではなく、

明示的に宣言子を指定し並列化を指示するためのものです。従って、誤った指示を与えた場合でも、忠実に並列化を行いますので、並列計算に依存性が内在する場合、計算結果が不正となります。また、OpenMP 指示行を入れたにも拘らず、「性能向上しない」と言う問題は OpenMP のデータ環境宣言句（SHARED、PRIVATE 等）の設定の誤り等が原因の場合が多いです。この並列化で指定できるスレッド数（CPU コア）の上限は 256 までです。

```
(例）ユーザプログラム例 (test.f)
    --------------------------------
    C       USER PROGRAM
            INTEGER A(1056,1024)
    C$OMP PARALLEL SHARED ...        ← OpenMP 指示行（通常のコンパイラで
    C$OMP DO                           はコメントと見なされます）
            DO J = 1, 1024
            ...
            END DO
    C$OMP END DO NOWAIT
    --------------------------------

    $ pgfortran –fastsse –Minfo -o test -mp test.f
    $ export NCPUS=2  （bash 使用時で、CPU/スレッド数 2 を指定）
    $ ./test
```

（注意） Linux 上での自動並列実行時あるいは OpenMP 並列実行時の STACKSIZE

Linux システム上で自動並列あるいは OpenMP 並列実行時に、セグメンテーション・フォールト等で異常終了する場合があります。これが起こるのは、Linux 上のデフォルトの per-thread stack size (例えば 2MB) が並列プログラムで使用するサイズより小さい場合に生じる問題のケースが多いです。以下の環境変数 **MPSTKZ** をより大きなサイズ（例えば 8MB 等）で設定してから、再度実行してみてください。環境変数の設定は、以下のようなコマンドで変更が可能です。なお、この目的には **OMP_STACKSIZE** (Proposed OpenMP 3.0 Feature) 環境変数も使用できます。OpenMP の詳細に関しては、2.3 項をご覧下さい。

```
$ setenv MPSTKZ 8M
  in csh, or with
$ MPSTKZ=8M; export MPSTKZ
  in bash, sh, or ksh.
```

■ MPI ライブラリによる並列化

PGI コンパイラ製品には、直ぐに MPI プログラムをコンパイルできるように、MPICH1 ライブラリがバンドルされております。コンパイル・オプションを付加することにより、MPI ライブラリを使用した実行モジュールが生成できます。また、PGI CDK 製品には、MPICH1、MPICH2、MVAPICH1 等のライブラリもバンドルされております。MPI ライブラリを使用する方法関しては、http://www.softek.co.jp/SPG/Pgi/TIPS/opt_mpi.html ページをご覧下さい。

■ GPU アクセラレータ用並列化（OpenACC、CUDA Fortran）

汎用アクセラレータとしてグラフィックス･プロセッシング･ユニット(GPU) 上で、その GPU コアを使用して並列計算を行うための並列化機能を提供します。これは、PGI Accelerator™ 機能と称し

ます。PGI Accelerator™ 機能は、NVIDIA 社の GPU/GPGPU とその CUDA 開発環境を実装したシステム上で、GPU を活用するためのコンパイラを含めたプログラム開発環境を提供します。具体的には、OpenMP 形式のようなディレクティブ挿入による x64+GPU 用実行バイナリの自動生成機能（OpenACC 準拠）、PGI CUDA Fortran 機能、そして、CUDA C/C++ プログラムを GPU を有しないマルチコア x64 プロセッサ上で動作させる機能を有します。この詳細に関しては、「GPU 対応 PGI アクセラレータ™ コンパイラ」に関するホームページをご覧下さい。

2　コンパイル・オプションの使用例

この章では、利用者がコンパイルを行う際に使用するコンパイル・オプションの使用事例を以下のように分類して紹介します。ここで説明する事例は、以下の通りです。

- 最適な性能を得るために、一般的に使用するオプション
- 64 ビットシステム(AMD64/Intel 64)上で、2GB 以上のメモリ領域を必要とするオブジェクトがある場合に使用するオプション
- 自動並列化、OpenMP 並列化を行うためのオプション
- コンパイル時の様々な実行情報を表示するためのオプション
- オブジェクトのリンク時に使用するオプション
- プログラムを開発・検証する際に便利なオプション
- 異なる CPU ターゲット用にクロスコンパイルする際のオプション

2.1　最適な性能を得るために一般的に使用するコンパイル・オプション

PGI の F77, F2003, C, C++ のコンパイラを使用する際に、性能を最大限最適化するためのオプション（複数のオプションを組み合わせた「複合オプション」があります。以下は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。

■ STEP 1 ： 一般的に使用する性能を最大化するベース・オプション

pgfortran **-fast** –Minfo test.f 90
pgfortran **–O2** –Minfo test.f90
(PGI 13.1 から -O2 でも自動ベクトル化(-Mvect=sse) 機能が加わりましたので、-fastsse 機能に近いオプションとなります）
あるいは、
pgfortran **-fast -Mipa=fast** –Minfo test.f

さらに、**-Mipa=inline** オプションを追加すると手続き間を含めた自動インライン化が有効となり、性能向上が望めます。特に、**C/C++**プログラムでは、インライン化は大きな性能向上が期待できます。PGI 7.1 以降では、64 ビット用の C/C++コンパイラの -fast 複合オプションの中に、**-Mautoinline** オプションが加わりました。C/C++ コンパイルで、-fast (-fastsse も含む）を指定すると、自動インラインモードでコンパイルされます。

もう一つ、C/C++ の場合は、-**Msafeptr** と言うコマンド・オプションは、ポインタ間でその記録されているストレージ上の重なりが存在しない場合（no pointer aliasing）、C/C++プログラムの性能を劇的に向上させます。但し、このオプションは、デフォルトでは有効ではありませんので、こうしたポインタ間の alias 依存性がない場合は、**-Msafeptr** オプションを指定するとさらなる最適化が行われます。

PGI 13.1(2013 年）から -O オプションの最適化レベルの一部変更

最適化レベルを指定する -O、-O2 の内容が変更されました。これは、PGI 13.1 から適用されています。以下にその概要を記します。下線部分が従来のものとの変更点です。

- -O0 は、レベル 0 であり、最適化を行わない。個々の言語文に対して基本ブロックが生成される。
- -O1 は、レベル 1 であり、局所最適化を行う。基本ブロックのスケジューリングが行われる。またレジスタ割り当て最適化も実施される。

- -O は、レベルが指定されない場合、以下のレベル 2 グローバル最適化を行う。これには、従来のスカラ最適化、導入変数の削除や問題のないループの移動等の最適化を含む。ただし、SIMD ベクトル化は行わない。
- **-O2** は、レベル 2 であり、グローバル最適化を行う。このレベルは、全てのレベル 1 局所最適化、上記-O オプションで説明したレベル 2 グローバル最適化が実施される。これに加え、SIMD コード生成やキャッシュ整列、部分的な冗長性排除等の高度な最適化も実施する。（**従来の -fast 機能を加えた意味合いとなる**）
- -O3 は、レベル 3 であり、アグレッシブなグローバル最適化を行う。全てのレベル 1, 2 の最適化だけでなく、効果のあるなしに関わらず、スカラの置き換え、より積極的な最適化を行う。
- -O4 レベル 4 であり、全てのレベル 1, 2, 3 の最適化だけでなく、浮動小数点演算式の中で不変変数に対する巻上げ最適化を行う。

pgcc/pgc++ -fast **-Mipa=fast,inline** -Minfo test.f90
(PGI 16.1 以降の Windows 版では C++コンパイラの提供はありません)

C++プログラムにおいては、特にインライン展開は重要であるため、-Mipa=inline オプションを使用しない場合は、明示的に自動インライン化を行うための **-Minline=levels:10** オプションを**常に使用することを強く推奨します**。また、**--no_exceptions** も性能向上する場合があります。（なお、この --no_exceptions を伴ってコンパイルしたプログラム自体が例外処理を行っている場合は、実行時にエラーとなりますのでご注意ください）PGI 11.0 以降、C++コンパイラは、低コスト例外処理のハンドリング（--zc_eh）をデフォルトとしました。

pgc++ –fast **-Minline=levels:10 --no_exceptions** -Minfo test.f90

- **-fast（= -fastsse）** オプションは、以下のオプションを組み合わせた複合オプションです。

  -fast : -O2 -Munroll=c:1 -Mnoframe –Mlre –Mpre
  -fast -Mvect=sse -Mscalarsse -Mcache_align –Mflushz

  -fast オプションは、最適化レベル 2 でかつ、プロセッサ内の SSE インストラクションを利用した、ループ内のベクトル化並びにスカラ計算部の SSE 化を行うものです。一般的にコンパイラを使用する際は、この複合オプションを付けることにより、使用するプロセッサに最適なコードが生成されます。

  なお、上記複合オプションの中で、-Mlre オプションは、loop-carried redundant removal と言う最適化手法を有効にします。loop-carried とは、ループ iteration 処理内と言う意味で、この中で共通数式(冗長数式）あるいは冗長な配列の参照を削除する最適化となります。この高度な最適化によって、数値結果に差異が生じる場合があります。この場合は以下のようにして、-Mlre の最適化のみを抑止することが可能です。-fastsse の後に -Mlre=noassoc を指定することにより、-Mlre 最適化が抑止されます。同様に、-Mpre オプションも冗長演算部の削除を行うものですが、-Mnopre によって抑止されます。

  pgfortran -fast -Mlre=noassoc -Mnopre

  PGI 7.0 以降の 64 ビット CPU ターゲットに対するコンパイル環境

  64 ビットコンパイル環境においては、PGI 7.0 以降より、-fast オプションが-fastsse オプションと同じ機能を有するものに変更されました。従来の-fast と等価な機能として、-nfast と言うオプションが新設されました。

PGI 7.1 以降の 64 ビット CPU ターゲットに対するコンパイル環境

C/C++ コンパイラの -fast 複合オプションの中に、-Mautoinline オプションが加わりました。

- **-Mipa=fast** オプションは、内部手続き（関数）間の各種グローバルな最適化を 1 パスで行うためのオプションです。なお、このオプションを指定して作成されるオブジェクトファイルのサイズは大きくなります。fast フラグは、-Mipa=align,arg,const,f90ptr,shape,globals,localarg,ptr を組み合わせたフラグ・セットです。このオプションでコンパイル時、あるいは実行時に問題が生じた場合は、各フラグの検証を行う必要があります。問題が生じた場合は、-Mipa を**外すことを推奨します**。
- **-Mipa=inline** オプションは、インライン対象となるサブルーチン、あるいは関数を探し、これらを呼び出す（call）ブロック内に自動インライン化するためのオプションです。特に、IPA 最適化の中でこのオプション・フラグを有効化した場合、全ての手続き間（C/C++の場合は、複数のソースファイル間）でインライン対象となる関数等を検出し、インライン化を行います。後述の -Minline オプションは、さらに、明示的かつ厳格にインライン対象を指示するために使用するオプションとなります。
- **-Minline** オプションは、明示的、かつ細かな指示を行うためのインライン展開のオプションです。現在は、-Mipa=inline オプションによって手続き間のインラインが可能であるため、こちらで代用することが多いですが、-Mipa オプションを使用しない場合のインライン展開は、-Minline オプションによって行います。


上記の -fastsse あるいは、-Mipa オプションに関してはコンパイル時だけでなく、**リンク時においても指定する必要があります**。特に Makefile 等で、コンパイルフェーズとリンク・フェーズを分けて行う場合は、リンケージのオプションにも同じように指定してください。リンク時において、上記のオプションを付けない場合、リンク時にエラーが発生します。


使用するコンパイル・オプションの意味を知りたい場合は、以下の **–flags** あるいは **–help** を指定すると、そのオプションの意味が表示されます。

```
$ pgfortran -O2 -flags
  Reading rcfile /opt/pgi/linux86-64/15.1/bin/.pgfortranrc
-O2                 Set opt level. All -O optimizations plus SIMD code generation, cache
alignment, and partial redundancy elimination performed
                    == -Mvect=sse -Mcache_align -Mpre
-M[no]vect[=[no]altcode|[no]assoc|cachesize:<c>|[no]fuse|[no]gather|[no]idiom|levels:<n
>|nocond|[no]partial|prefetch|[no]short|[no]simd|[no]sizelimit[:n]|[no]sse|[no]tile|[no
]uniform]
                    Control automatic vector pipelining
    [no]altcode     Generate appropriate alternative code for vectorized loops
    [no]assoc       Allow [disallow] reassociation
    cachesize:<c>   Optimize for cache size c
    [no]fuse        Enable [disable] loop fusion
    [no]gather      Enable [disable] vectorization of indirect array references
    [no]idiom       Enable [disable] idiom recognition
    levels:<n>      Maximum nest level of loops to optimize
    [no]partial     Enable [disable] partial loop vectorization via inner loop distribution
    prefetch        Generate prefetch instructions
```

```
    [no]short       Enable [disable] short vector operations
    [no]simd        Generate [don't generate] SIMD instructions
     128            Use 128-bit SIMD instructions
     256            Use 256-bit SIMD instructions
    [no]sizelimit[:n]
                   Limit size of vectorized loops
    [no]sse         Generate [don't generate] SSE instructions
    [no]tile        Enable [disable] loop tiling
    [no]uniform     Perform consistent optimizations in both vectorized and residual loops;
this may affect the performance of the vectorized loop
-Mcache_align       Align large objects on cache-line boundaries
-M[no]pre           Enable partial redundancy elimination
```

■ STEP 1 ： Advanced ： ベクトル化最適化オプションを使用したファイン・チューニング

ループ内処理の様々なベクトル化最適化は、**-Mvect=**[flags]オプションを指定することにより行われますが、さらにフラグを用いて適用する最適化方法を指定できます。最適化複合オプションである -fastsse は、-Mvect=sse （SSE インストラクションを用いて SIMD 形式のベクトル処理を行う）がデフォルトで適用されますが、その他によく用いられるチューンニング・フラグのいくつかを以下に示します。これらの追加するオプションは、コマンドライン上で -fastsse の後に指定してください。なお、これらのフラグを指定したとしても、全てについて性能が向上するという訳ではありませんので、必ず各フラグによる性能を確認してから、最適なオプションをご使用ください。
その他のオプション/フラグの内容については、PGI User's Guide をご覧ください。

① pgfortran –fastsse –Mvect=**cachesize**:1048576 –Minfo test.f
② pgfortran –fastsse –Mvect=**[no]assoc** –Minfo test.f
③ pgfortran –fastsse –Mvect=**nosizelimit** –Minfo test.f
④ pgfortran –fastsse –Mvect=cachesize:1048576,nosizelimit,assoc –Minfo test.f

- ①で示した–Mvect=cachesize:**n** では、プロセッサの L2 キャッシュサイズ **n** を指定します。このキャッシュサイズをコンパイラは認識し、ループ内処理のキャッシュ最適化（ストリップマイニング処理等）を施します。デフォルト値は、プロセッサ依存となっております。例えば、L2 キャッシュサイズが、512KB の場合は 524288、1MB の場合は 1048576 となります。デフォルトでは、**各 CPU ターゲットに応じた L2 キャッシュサイズがセット**されております。

- ②で示した –Mvect=[no]assoc は、コンパイラにコードの演算の association（結合）変換による最適化を指示するものです。浮動小数点の演算結果に影響を与える生成コード上の演算式等の変換を許します。具体的にはループの再構成や、乗算から加算／減算への分割などを行いますので累積的な丸め誤差等が発生する可能性があります。従って、これを適用した場合は、**必ず数値結果の誤差評価**を行ってください。–Mvect=noassoc を指定すると、このような最適化は一切行いません。

- ③で示した–Mvect=nosizelimit は、ループ内のステートメント数に拘らず全てのループをベクトル最適化の対象とすることを指示するものです。一般的に最内側のループにおいてベクトル化処理を行いますが、ループ内のステートメント数が多い場合、コンパイラの最適化対象とするデフォルトのステートメント数を超えることにより、ベクトル化が阻害される場合があります。これは、-Mneginfo オプションで最適化阻害部分の情報を表示することで発見できます。なお、PGI 6.2 以降では、–Mvect=nosizelimit が**デフォルト**となっております。

- ④は、コマンドライン上で複数のベクトル化フラグを指定する場合を示したものです。各フラグ間で、カンマ（,）で区切ります。

■ STEP 2 ： STEP 1 の次にさらに付加するオプション

【SSE/SSE2/SSE3 機構を備えたマシン】
pgfortran –Mipa=fast -fast **-O3 -Mvect=prefetch=d:8** –Minfo test.f

- **-O3** オプションは、-O2 よりもアグレッシブな最適化を行いますが、コード特性により常に高速化が図られると言うわけではありません。-fastsse あるいは -fast より**後に記述**してください。-fast 系の複合オプションに内在する -O2 を上書きします。
- **-O4** オプションは、PGI 7.0 から新設されました。-O4 レベルの最適化は、浮動小数点演算式の中で不変変数に対する巻上げ最適化を行うようになります。PGI 7.1 では、algebraic transformation とレジスタ・アロケーション最適化が追加されました。
- -Mvect=prefetch=d:<p>,n:<q> オプションは、ループのベクトル化による最適化において、メモリ階層でのキャッシュへのプリフェッチを極力行うためのオプションです。性能が向上する場合があります。この prefetch フラグには、さらに細かい制御を行うためのサブフラグ（例：-Mprefetch=distance:8,nta）があります。この詳細については、4.4 項で説明しておりますので参考にしてください。

性能最適化のオプションを使い分ける際は、必ず、それぞれのオプションを指定して、性能を評価してください。プログラム自体の特性によって、個々に性能は異なります。PGI に限らず、コンパイラ共通の特性により、場合によっては遅くなる場合もあるため、最速なオプションを評価し使用して下さい。

■ Advanced Tuning：メモリ・アロケーション・Huge ページの使用・TLB エントリ数の最適化

【プロセッサの TLB エントリ数のチューニング　オプション】
pgcc/pgc++/pgfortran -fastsse **-Msmartalloc=huge:448** {source_file}

【オプション指定方法】
-M[no]smartalloc[=huge | huge:n | hugebss]

- -Msmartalloc オプション自体（サブオプションなしの指定方法）は、メインルーチン中に最適化された <B>mallopt</B> ルーチンのコールを加えます。これは、PGI 6.2 以降に新設されたものですが、Core2/Barcelona 以降の新しいプロセッサ以外の従来のプロセッサに対しても有効です。これを有効にするためには、Fortran、C、C++の「メインプログラム」を含むファイルをコンパイルする際に、このオプションを付する必要がある。デフォルトは、-Mnosmartalloc。
- PGI 7.1 以降、-Msmartalloc オプションは、Linux 並びに Windows 上での large TLBs をサポートするために強化されました。このオプションは、最適な malloc ルーチンを有効にするために、メインプログラムをコンパイルする際に使用することが必要です。
- サブオプション huge は、シングルプログラムで使用される大きな 2MB ページを有効にするために指定します。これは、実行するために必要な TLB エントリ数を削減する効果があります。このオプションは、AMD の Barcelona やインテル(R)の Core2 以降のシステムで特に有効です。古いプロセッサ・アーキテクチャでは、TLB エントリの数が少ないため、大きな効果は期待できない可能性があります。サポートするサブオプションは、以下のとおりです。

  huge　　: huge page のランタイムライブラリをリンクします
  huge: n　: huge page のランタイムライブラリをリンクし、使用されるページの数の限度を n に設定します
  hugebss　: huge page の中に BSS セクションを置きます

  huge サブオプションは、それ自身、必要とされる huge page の数をアロケートしようとします。Huge page の数は、:n サブオプションで制限を設けることができ、あるいは、環境変数 PGI_HUGE_SIZE でも設定できます。hugebss は、プログラムの初期化されていないデータセクションを huge page の中に置きます。


上記の -Msmartalloc オプションは、コンパイル時とリンク時の両方で指定する必要があります。特に Makefile 等で、コンパイルフェーズとリンク・フェーズを分けて行う場合は、どちらにも本オプション指定してください。

■ Advanced Tuning：明示的に行う関数・サブプログラムの自動インライン展開

【インライン展開される側のソースライン数 約 1000 行まで許す】
pgcc/pgc++/pgfortran -fastsse **-Minline=size:1000** {source_file}

【インライン展開される多重レベル数】
pgcc/pgc++/pgfortran -fastsse **-Minline=levels:3** {source_file}

【複数のサブオプションを組み合わせる場合】
pgcc/pg++/pgfortran -fastsse **-Minline=size:1000,levels:3** {source_file}

- 関数・サブプログラムをコールする親プログラムの中に、これらをインライン展開することは、コール・オーバーヘッドを削減する上で、性能が向上する場合があります。これは、特に C 言語系のプログラムにおいて有効となります。PGI のインライン展開機能を指示するサブオプションは、上記に示した以外にもありますが、ここでは、よく使用する二つのサブオプションを紹介します。
- -Minline=size:number オプションは、インライン展開の対象となる関数・サブプログラムの最大ソース行数 （number） を指示するものです。
- -Minline=levels:number オプションは、インライン展開される関数のレベル数を指示するものです。デフォルトは、1 です。これは、インライン展開された関数・サブプログラム自体がさらにその上位で展開される段数を指定するものです。

その他のインライン展開を行うための機能フラグは、以下のとおりです。インライン対象となる関数名を指定することができます。

-Minline[=[lib:]<inlib> | [name:]<func> | except:<func> | size:<n> | levels:<n>]

[lib:]<inlib>	"inlib"ライブラリの中の関数を抽出してインライン展開する
[name:]<func>	関数"func"をインライン展開する
except:<func>	関数"func"をインライン展開しない
size:<n>	インライン対象のソース行数 n 以下のものをインライン展開
levels:<n>	インライン展開される関数のレベル数

■ Advanced Tuning：関数・サブプログラムのインライン・ライブラリ化

上記示した自動インライン展開の方法は、**コマンドライン上に指定した**（複数の）入力ソースファイル内に存在するインライン可能な候補ルーチンのみを抽出するものです。例えば、一つのソースファイルのみをコンパイルしようとした場合、そのソースファイル内だけに閉じて抽出された関数・サブプログラムのみがインライン展開されることになります。しかし、Fortran/C プログラム等では、複数のソースファイル中に存在する、インライン可能な候補ルーチンが別の入力ファイルのルーチン内に展開されるような場合は、工夫が必要となります。簡単に行う場合は、複数の入力ソースファイルをコマンドライン上に羅列すればよいのですが、例えば、Makefile を使用する場合は、このようなことは不可能な場合があります（Makefile を使用すると基本的に、1 ソースファイル毎にコンパイルされます）。このような場合は、予め、インライン対象候補を抽出して「インライン・ライブラリ」を作成し、次に、このライブラリをコンパイル時に指定することにより、インライン展開を実現します。これによって、インライン展開される範囲が広がります。

最初に、インライン対象候補を抽出してインライン・ライブラリを作成してみましょう。pgf77、pgfortran、pgcc、pgc++（pgcpp）コマンド共に共通の方法となりますが、抽出には、**-Mextract** オプションを使用して行います。

【ソースライン数 100 行までのインライン展開の対象ルーチンの情報ライブラリ作成】

```
$ pgfortran –Mextract=size:100 -o src.il src1.f src2.f src3.f
```

【インライン src.il ライブラリを参照して適用可能な部分にインライン展開】

```
$ pgfortran –Minfo=inline –Minline=size:100,lib:src.il,levels:3 src1.f src2.f src3.f
```

- **-Mextract** オプションは、インライン展開候補となるルーチンの情報を **–o** オプションで指

定されたファイル名に記録する（これが、**src.il** と言うインライン・ライブラリです）

- 次に、-Minline オプションのフラグの中に、lib:{name}の形態でインライン・ライブラリを指定し、実際のコンパイル・リンク処理を行います。–Minfo=inline オプションは、インライン展開中のエラー等の情報メッセージを出力します。

Makefile を使用してインライン・ライブラリを更新する方法

```
SRC = mydir
FC = pgfortran
FFLAGS = –O2
main.o: $(SRC)/main.f $(SRC)/global.h
        $(FC) $(FFLAGS) -c $(SRC)/main.f
utils.o: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
        $(FC) $(FFLAGS) -c $(SRC)/utils.f
utils.il: $(SRC)/utils.f $(SRC)global.h $(SRC)/utils.h
        $(FC) $(FFLAGS) -Mextract=15 -o utils.il
parser.o: $(SRC)/parser.f $(SRC)/global.h utils.il
        $(FC) $(FFLAGS) -Minline=utils.il -c
        $(SRC)/parser.f
alloc.o: $(SRC)/alloc.f $(SRC)/global.h utils.il
        $(FC) $(FFLAGS) -Minline=utils.il -c
        $(SRC)/alloc.f
myprog: main.o utils.o parser.o alloc.o
        $(FC) -o myprog main.o utils.o parser.o alloc.o
```

■ C/C++ コンパイラにおける–Mautoinline 自動インライン・オプション追加（PGI 2010 以降）

```
-Mautoinline[=levels:n,mazsize:n,totalsize:n]
```

C/C++ルーチンをインライン化するように、コンパイラに指示するオプションです。C/C++コンパイラのオプションです。コンパイル・オプション –O2 (-fast) を指定した際 –Mautoinline は有効化されます。

- levels:n は、インラインを行うレベル階層を最大 n まで行うことを指示。デフォルトは 10。
- maxsize:n は、 n サイズを超える関数のインラインを行わない。デフォルトは 100。
- totalsize:n は、インライン h 対象が、n サイズ時にインラインを止めることを指示。 デフォルトは 800。

■ Advanced Tuning：高度な最適化（コンパイラ・ディレクティブを使用する）

コンパイラのオプションだけではなく、ソースプログラムに直接コンパイラへの最適化指示を行うためのディレクティブも用意されています。詳細については、PGI User's Guide を参考にしてください。以下の例は、メモリ階層でのデータ・プリフェッチを明示的に指示するための例です。

```
      real*8 a(m,n), b(n,p), c(m,p), arow(n)
...
      do j = 1, p
cmem$ prefetch arow(1),b(1,j)
cmem$ prefetch arow(5),b(5,j)
cmem$ prefetch arow(9),b(9,j)
        do k = 1, n, 4
cmem$ prefetch arow(k+12),b(k+12,j)
          c(i,j) = c(i,j) + arow(k) * b(k,j)
          c(i,j) = c(i,j) + arow(k+1) * b(k+1,j)
          c(i,j) = c(i,j) + arow(k+2) * b(k+2,j)
          c(i,j) = c(i,j) + arow(k+3) * b(k+3,j)
        enddo
      enddo
```

■ AMD Barcelona プロセッサ向け最適化オプション（PGI 7.1 以降）

```
-M[no]fpmisalign
```

-M[no]fpmisalign ― AMD barcelona プロセッサに対して、16-byte 境界に整列されていないアドレスを持つメモリ・オペランドのベクトル演算命令の使用を許可します。デフォルト設定は、Barcelona を含めて、全てのプロセッサにおいて -Mnofpmisalign です。本オプションは、-tp barcelona-64 あるいは、-tp barcelona の設定時、あるいは、barceona 上でコンパイルされたときにのみ効果があります。また、このオプションでコンパイルされたコードは、barcelona プロセッサ上だけで実行できるものとなりますのでご注意下さい。

```
-M[no]loop32
```

-M[no]loop32 ― barcelona 上での 32-byte 境界上にある最内側ループを整列します。barcelona 上で 32-byte 境界で整列されている場合、小さなループは性能が向上する可能性があります。しかし、実際には、ほとんどのアセンブラが、まだ効果的なパッディング(padding) 最適化を実装していません。その結果、このオプションで遅くなる可能性もあります。Barcelona に対して最適化されたアセンブラを有するシステム上でこのオプションを使用してください。デフォルトは、-Mnoloop32 です。

2.2 2GB 以上のメモリ領域を必要とするオブジェクトがある場合に使用するオプション

PGI 64 ビット対応 Linux 版の製品における F77, F2003, C, C++ のコンパイラを使用して、2GB 以上のプログラム・オブジェクトがある場合のコンパイル・オプションの例を示します。なお、PGI 6.0 以降のバージョンでは、C++ を含む全ての言語コンパイラは 2GB 以上の単一オブジェクトに対応するコンパイルができます。

Windows x64 / Apple macOS での扱い

Windows x64 上、並びに Apple OS X 64-bit 上では、PGI コンパイラでは、2GB 以上の単一オブジェクトを扱うことができる -mcmodel=medium オプションと等価な機能を提供しておりません。これは、Microsoft(R) Win64 プログラミングモデルが、2GB を超える静的な単一データオブジェクトのハンドリングをサポートしていないためです。従って、単一あるいは、メモリ空間総量が 2GB を超える静的配列を扱うようなプログラムでは制約があります。例え Windows 64 ビット版であっても、2GB を超える単一配列オブジェクトがプログラム上に存在する場合、静的な配列宣言はできません。これを回避する方法として、静的配列宣言を動的な配列宣言（Allocatable 配列宣言）に変更する方法がありますが、プログラムの改修が伴います。こうした制約は、PGI コンパイラの制約ではなく、Win64 プログラミングモデル上の制約です。Apple Mac OS 上においても、同様な理由で、2GB を超える静的な単一データオブジェクトを使用できません。従って、2GB 以上のデータオブジェクトを扱うプログラムをコンパイルする場合は、現在のところ、このような制約がない Linux 版の方が適しています。

以上の制約は、あくまでも 2GB 以上の静的配列を扱う場合のものですが、**動的な配列宣言の配列**を使用することにより、Windows(64bit) / Apple macOS でも 2GB 以上のメモリ空間を使用することができます。特に **Win64** 上では、**2GB 以上のアドレッシング（インデックスの計算）の処理を有効にするためのオプション**があります。Win64 環境での PGI コンパイラでは、以下のオプション(-Mlargeaddressaware) を付加することにより、アプリケーションが 2GB を超えるアドレスを処理することをリンカに知らせることができます。これについては、マイクロソフト社のページを参照して下さい。なお、Apple macOS には、こう言った類のオプションは存在しません。

pgfortran -Minfo -fastsse **–Mlargeaddressaware** test.f90
コンパイル時、リンク時共に指定して下さい。

- **-Mlargeaddressaware** (PGI 7.2 新設 : Windows のみ) Windows x64 用に 2GB 以上のアドレス・インデックシングを Windows のリンカへ指示します。（RIP-relative addressing を使用する)。デフォルトは、no で、direct addressing 形式となります。
- **-Mlarge_arrays** オプションは、配列添え字（インデックス）を 64 ビット整数で扱えるように変更します。この意味は、必要に応じて、64 ビット整数変数あるいは定数が、インデックスの計算において使用されることを意味します。但し、コンパイラが暗黙に 32 ビット整数から 64 ビット整数に変更することによって、思わぬところで副作用が現れるかもしれないことに注意してください。一般に、64 ビットアドレッシングが必要なインデックス変数は、明示的に 64 ビット整数宣言をすることが最も安全な方法で、これ行ってかつ、このオプションを指定することを推奨します。

■ 2GB を超えるプログラムの実行モジュールの生成オプション（PGI 6.0 以降の場合）

```
pgf77/pgfortran –fastsse -mcmodel=medium -i8   -Minfo test.f
pgcc/pgc++   -fastsse -mcmodel=medium     -Minfo test.c
```

- **-mcmodel=medium** オプションは、64 ビットメモリ空間モデルの medium タイプを指定するものです。
- **-Mlarge_arrays** オプションは、PGI 5.2 バージョンの場合は必要でしたが、PGI 6.0 以降では、-mcmodel=medium オプションの中に内包されましたので指定する必要はありません。このオプションを付加すると、単一オブジェクト（配列等）あるいは .bss のサイズが 2GB 以上をサポートすると共に、配列のインデックス計算において 64 ビット整数（定数、変数）空間のサイズで計算されます。このオプションは、コンパイラが暗黙的に使用している 32 ビット整数を 64 ビット整数にプロモートします（この点は留意が必要です）。この機能を無効にする場合は、**-Mnolarge_arrays** を−mcmodel=medium の後に指定します。
- **-i8** オプションは、INTEGER で宣言された、あるいは暗黙に宣言された整数を 8 バイト整数にコンパイラが置き換えてコードを生成します。4 バイト INTEGER は、2GB 以上のアドレス値を処理できないため、強制的にこの措置をプログラムに適用する場合に使用します。なお、プログラムコーディング上、明示的に INTEGER*4 として宣言されているものに対しては、このオプションは機能しません。この場合は、プログラムの修正が必要となります。-mcmodel=medium を使用する場合は、このオプションを使用しておくことが無難ですが、暗黙的に 8 バイト整数に変更されることに留意してください。
- 上記のオプションは、64 ビットサポートと-fastsse オプションのみ例示しています。必要であれば、その他の最適化オプションも添えて、コンパイルしてください。

■ プログラム内部に INTEGER 宣言があるが、4 Byte 整数以上の計算を行う場合

```
pgfortran –fastsse –Minfo -mcmodel=medium -Mlarge_arrays -i8 test.f
```

- **-i8** オプションは、INTEGER で宣言された、あるいは暗黙の宣言された整数を 8byte 整数（INTEGER*8）に自動的に変換します。これにより、配列のインデックス計算で 4byte 整数の範囲を超えたとしても、エラーが生じません。-i8 あるいは INTEGER*8 宣言された整数は、8Byte 長のストレージとなり、さらに 64bit の処理が行われます。なお、明示的に INTEGER*4 として記述宣言されているものに対しては、このオプションは機能しません。

■ Linux 64 ビットシステムでのプログラミングモデルに関して

PGF77、PGFORTRAN、PGCC、PGC++ は、x86-64 Application Binary Interface で定義されている、-mcmodel=small 並びに -mcmodel=medium アドレッシング・モデルの両方をサポートします。次に示す表は、これらのプログラミングモデルでの制約事項を示します。また、下表は様々なコンパイル/リンクオプションの組み合わせで 32 ビット、64 ビット実行モジュールを作成した際の 32 ビット、64 ビットの処理形態を示しております。

● 64 ビットシステムでのプログラミングモデル
（メモリ空間、アドレッシング）

Programming Models on 64-bit Linux86-64 Systems						
Combined Options	Address Arithmetic		Maximum Data Size in Gbytes			コメント
	A	I	AS	DS	TS	
-tp {32bit target}	32	32	2	2	2	32bit linux86 互換
-tp {64bit target}	64	32	2	2	2	64bit Addressing だが、メモリモデルは -mcmodel=small であり、データ領域は 2GB に制約される
-tp {64bit target} -fpic	64	32	2	2	2	-mcmodel=medium を伴うコンパイルでは、-fpic は指定できない
-tp {64bit target} -mcmodel=medium	64	64	>2	>2	>2	64ビットデータ・アドレッシングをフル・サポートする (-Mlarge_arraysオプションを包括している)

（凡例）

Address Type **(A)** アドレス計算に使用されるデータのビット数のサイズ （32-bit or 64-bit）

Index Arithmetic **(I)** 配列並びに他のデータ構造のインデックス計算で使用されるデータのビット数のサイズ。I が 32bit の場合、単一データオブジェクトのサイズは、2GB に制約される。

Maximum Array Size **(AS)** 任意の単一データオブジェクトの最大サイズ

Maximum Data Size **(DS)** 実行モジュールの中の .bss セクションにおける全てのデータオブジェクトの総和最大サイズ

Maximum Total Size **(TS)** 実行プログラムにおける全てのデータオブジェクトと実行テキスト・コードの総和した際の最大サイズ

-tp オプションは、明示的な CPU ターゲットを指定してクロスコンパイルを行うためのオプションです。デフォルトでは、コンパイルを行う「システム」のプロセッサ・ターゲットがその最適化のために使用され、かつ、当該システムの OS のビット数(64bit or 32bit)に応じた実行モジュールが生成されます。従って、64bit Linux のシステムの場合は、デフォルトで 64bit 実行バイナリが生成されますので、そのもとで、2GB 以上のメモリ空間を必要とするプログラムの場合は、上記の -mcmodel=medium を指定して下さい。

プログラムエリアとは、Linux オペレーティングシステムとユーザプログラムによって使用されるそれらが総和された領域です。ほとんどの 32 ビット Linux システムでは、32 ビットアドレッシングですので理論的には 2GB までアクセス可能ですが、通常 1GB のみがデータ領域として使用することができます。

一方、64 ビットシステム上でのデフォルトである small memory model の環境においても、ユーザデータあるいは実行モジュールの総和量は、1GB までに制約されます。これは、32 ビットシステムと同様にシステム・ルーチンと共有ライブラリ、スタック等のアドレスが、2GB 空間の内、その他の 1GB 領域を使用するため、この部分が Linux のカーネル管理領域として占有されます。但し、共有ライブラリを使用せず、静的なリンクにより固定アドレスで開始されるプログラムは、ほとんどの領域のメモリアクセス(2GB 以内)が可能となります。

64 ビットシステム上での medium memory model の環境を構築するためには、-mcmodel=medium オプションをコンパイル／リンク時に指定する必要があります。このメモリモデルでは、2GB 以上のデータオブジェクト並びに .bss セクションのサイズであっても、問題なく動作します。

リンク時に、-mcmodel=medium オプションが必要となる生成実行モジュールにリンクされるオブジェクトファイルは、-mcmodel=medium オプションあるは、-fpic のどちらかでコンパイルされたものに限ります。-fpic オプションと -mcmodel=medium オプションを同時に使用してコンパイルすることはできません。すなわち、位置独立な実行モジュール (-fpic) の作成を行う場合は、-mcmodel=medium を使用することはできません。

● -mcmodel=medium 時において実際に生じる可能性のある制約事項と対処

64 ビット環境における 64 ビットアドレッシング機能は、データサイズが非常に大きな場合、予期せぬ問題が生じる場合があります。

初期化における問題（Initializing）

データ文での非常に大きな配列の初期化では、非常に大きなアセンブラあるいはオブジェクトファイルのサイズになることがあります。これは、アセンブラ・ソースのライン数が初期化する配列の個々の要素ごとに必要なためです。また、コンパイルとリンクの時間は極めて長い時間を費やします。これを避けるには、非常に大きな配列の初期化は、宣言文で行うよりもむしろループ処理において定義するように変更することです。

スタックスペース（stack space）

スタックスペースは、スタック・ベースのデータを使用する場合に生じる問題で、具体的にはスタックサイズが小さいと言う問題で異常終了する場合があります。Linux シェル環境での limit stacksize unlimited コマンドは、できるだけ多くのスタックサイズの確保を指示するものですが、その明確なサイズが指定されていないことと、物理メモリ量に依存するという問題があります。そのために、まず、limit stacksize 512M と指定した時、unlimited で指定されたものより大きなサイズかどうかを確認してください。もしそうならば、OS で指定されたスタックサイズのハード・リミットが設定されている可能性があり、必要であれば、ハード・リミット等の変更が必要となります。

Windows x64 上では、以下のコンパイル・オプションで**スタックサイズを指定**します。N は、必要とする任意のスタックサイズ値を指定します。

> -Wl,-stack:N
>
> PGI 7.0 以降では、Windows のみに有効な以下の -stack オプションが新設されました。
>
> -stack={ (reserved bytes)[,(committed bytes)] }{, [no]check }
>
> -stack を指定しない場合のデフォルトは、以下のとおりです。
>
> Win32 : -stack:2097152,2097152、2MB がデフォルトで設定されます。
> Win64 : デフォルトの設定はありません。
>
> 例： pgfortran -stack=524288,262144,nocheck myprog.f

プログラムの実行において、全体でトータル 524,288 stack bytes (512KB)の領域を予約し、各ルーチンのために OS が割り当てるスタックエリアとして 262,144 stack bytes (256KB)を指定します。また、ルーチンに入る際のスタックの初期化を行わないように nocheck 引数をコンパイラに指示すると言う指定の方法です。

reserved bytes	: プログラムで使用するトータル・スタックサイズ値を指定
committed bytes	: 各ルーチンのために OS が割り当てるスタックエリアのサイズを指定 デフォルトは、4096 バイト。
[no]check	: ルーチンに入る際にスタックの初期化を行うコードを生成するか、 行わないコードの生成かの指示を行う。"check"がデフォルトです。

スタック初期化コードは、「ルーチンのスタックサイズ」が committed bytes を超えたときに必要とされます。指定した committed bytes が reserved bytes に等しいか、あるいは、各ルーチンで必要とされるスタックバイト数に等しいとき、stack=nocheck オプションを指定して、スタック初期化コードの生成を抑止することができます。こうすることによって、コンパイラは十分なコミットスタックスペースを指定しているものと理解し、プログラムはそれ自身のスタックサイズの管理を必

要とされません。
（注意）
-stack=(reserved bytes),(committed bytes) はリンク時のオプションです。
-stack=[no]check は、コンパイル時のオプションです。

ページ・スワッピング（page swapping）
もし、実行モジュールが実装されている物理メモリ量よりも大きな場合は、ページ・スワッピングが頻繁に起こり、プログラムの実行が非常に遅くなるか、もしくは、途中で異常終了します。これは、コンパイラの問題ではありません。問題がページ・スワッピングによる問題かどうかを判断するためには、データサイズを小さくして検証してください。小さくして正常に動作する場合は、このページ・スワッピングの問題が考えられます。

コンフィギュアド・スペース(configured space)
アプリケーションが必要とする十分なシステムのスワップ領域を確保しているかどうかを確認してください。もし、メモリ＋スワップ領域が十分大きくない場合、実行時にセグメンテーション・フォールトが生じる場合があります。

● -mcmodel=medium and Large Array in C プログラム(Linux)

静的配列のサイズの総量が 2GB を越えた C プログラムの例で、セグメンテーション・フォールトを起こしてみます。

■ C プログラム bigadd.c

```
$ cat bigadd.c
#include <stdio.h>
#define SIZE 600000000 /* > 2GB/4 */

static float a[SIZE], b[SIZE];
int main()
{
long long i, n, m;
float c[SIZE]; /* goes on stack */

n = SIZE;
m = 0;
  for (i = 0; i < n; i += 10000) {
  a[i] = i + 1;
  b[i] = 2.0 * (i + 1);
  c[i] = a[i] + b[i];
  m = i;
  }
printf("a[0]=%g b[0]=%g c[0]=%g¥n", a[0], b[0], c[0]);
printf("m=%lld a[%lld]=%g b[%lld]=%gc[%lld]=%g¥n",m,m,a[m],m,b[m],m,c[m]);
return 0;
```

■ Linux 上で実行してみる

【コンパイル】 -mcmodel=medium オプション

```
$ pgcc -mcmodel=medium -fastsse -Minfo bigadd.c
main:
     12, Loop not vectorized: mixed data types
         Loop not vectorized: may not be beneficial
         Unrolled inner loop 4 times
```

【実行】

```
$ ./a.out
Segmentation fault
```
（セグメンテーション・フォールト発生）

【コンパイル】 -Mchkstk（実行時のスタックサイズの出力）を追加し、ランタイムで stack size を把握する

```
$ pgcc -mcmodel=medium -fastsse -Minfo -Mchkstk bigadd.c
main:
     12, Loop not vectorized: mixed data types
         Loop not vectorized: may not be beneficial
         Unrolled inner loop 4 times

$ ./a.out
Error: in routine main there is a
stack overflow: thread 0, max 10228KB, used 0KB, request -1894967208B
```
スタックサイズが足りないことが分かる。ここで表示されるサイズの量は、あまり厳密に考えない。

【現在のスタックサイズの把握（bash)】

```
$ ulimit -s
10240
```

【スタックサイズの変更（bash) 3GB に変更後、実行】

```
$ ulimit -s 3000000000

$ ./a.out
a[0]=1 b[0]=2 c[0]=3
m=599990000   a[599990000]=5.9999e+08
b[599990000]=1.19998e+09c[599990000]=1.79997e+09
```

実行モジュール a.out の bss セクションのサイズ表示（2GB を越えていることが分かる）

```
$ size --format=sysv a.out | grep bss
.bss                 4800000064     6295360
$ size --format=sysv a.out | grep Total
Total                4800003980
}
```

● -mcmodel=medium and Large Array in Fortrran プログラム(Linux)

2GB を越える Fortran プログラムで -mcmodel=medium を指定して実行する

■ Fortran プログラム mat.f90 （6GB 以上）

```
$ cat mat.f90
program mat
integer i, j, k, size, l, m, n
parameter (size=16000) ! >2GB
parameter (m=size,n=size)
real*8 a(m,n),b(m,n),c(m,n),d

do i = 1, m
do j = 1, n
  a(i,j)=10000.0D0*dble(i)+dble(j)
  b(i,j)=20000.0D0*dble(i)+dble(j)
enddo
enddo

!$omp parallel
!$omp do
do i = 1, m
do j = 1, n
  c(i,j) = a(i,j) + b(i,j)
enddo
enddo

!$omp do
do i=1,m
do j = 1, n
  d = 30000.0D0*dble(i)+dble(j)+dble(j)
  if(d .ne. c(i,j)) then
  print *,"err i=",i,"j=",j
  print *,"c(i,j)=",c(i,j)
  print *,"d=",d
  stop
  endif
enddo
enddo
!$omp end parallel

print *, "M =",M,", N =",N
print *, "c(M,N) = ", c(m,n)
end
```

■ Linux 上で実行してみる

【コンパイル】 -mcmodel=medium, -i8 オプション

```
$ pgfortran -mcmodel=medium -i8 -mp -fast -Minfo mat.f90
mat:
      7, Loop interchange produces reordered loop nest: 8,7
         Unrolled inner loop 4 times
```

```
      Used combined stores for 2 stores
   14, Parallel region activated
   16, Parallel loop activated with static block schedule
      Loop interchange produces reordered loop nest: 17,16
      Generated an alternate version of the loop
      Generated vector sse code for the loop
      Generated 2 prefetch instructions for the loop
   22, Barrier
   23, Parallel loop activated with static block schedule
   24, Loop not vectorized/parallelized: contains call
   34, Barrier
      Parallel region terminated
```

【実行】

```
$ ulimit -s  (stacksize を調べる)
10240

$ time ./a.out
 M =                    16000 , N =                    16000
 c(M,N) =       480032000.0000000

real    0m2.832s
user    0m7.645s
sys     0m1.899s
```

(a.out は 6GB 以上使用するプログラム)

```
$ size --format=sysv a.out | grep bss
.bss                6144000416    6300672
$ size --format=sysv a.out | grep Total
Total               6144008305
```

2.3 自動並列化、OpenMP 並列化を行うためのコンパイル・オプション

PGI の F77, F2003, C, C++ のコンパイラを使用して、並列化を行う際のコンパイル・オプションの例を示します。以下は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。**PGFORTRAN、PGCC、PGC++コンパイラ**において、OpenMP 3.1 に準拠しております。

【注意】インテルのプロセッサでは、「**Hyperthreading**」 と言う機能を有しています。物理的なプロセッサ・コア数の 2 倍の並列スレッド実行を行えるようにした機能ですが、HPC(High Performance Computing) 並列実行環境では、BIOS レベルで、この機能を disable にして下さい。この「Hyperthreading」によって、物理コアの数を越えた並列度の加速性が得られる訳ではありません。「Hyperthreading」機構は、各コアに、そのレジスタ群を 2 セット用意しているものであり、演算器セットを 2 セット用意しているものではありません。従って、各コア当たり、必ず演算性能が 2 倍になる構造とはなっていません。以下の機能で述べる、並列度数の指定では、物理コア数による指定を行うことをお勧めします。例えば、Intel Sandybridge プロセッサで、4 コア（ハイパースレッドで 8 論理コアのプロセッサ）の場合の並列度の指定では、OMP_NUM_THREADS=4 がその最大値の指定となります。

■ 自動並列化を行うためのコンパイル・オプション

```
pgfortran –fastsse –Minfo -Mconcur test.f
```

- **-Mconcur** オプションは、プログラム内のループレベルの並列性を抽出し、自動並列化を行うためのオプションです。concur=フラグ なしで使用しても構いません。
- -Mconcur=**dist:{block|cyclic}** のサブフラグを使用して、並列分割実行のブロック分け制御が可能です。block がデフォルトです。block は並列計算範囲を等分に分ける方法であり、cyclic はラウンドロビン形式に、例えば、cpu 0 が 0, 3, 6, .. を cpu 1 が 1, 4, 7,...を cpu 2 が 2, 5, 8,..を受け持つような計算分割を行います。
- -Mconcur=**cncall** は、ループ内に関数あるいはサブルーチンが含まれている場合の並列化を行うことは安全であることをコンパイラに指示します。このフラグが指定しないデフォルトは、ループ内に関数あるいはサブルーチンが含まれている場合の並列化を行いません。
- -Mconcur=**[no]innermost** は、最内側ループの自動並列化を許可する（しない）オプションです。これは、粒度の小さなものを並列化の対象にするため、この適用は性能の効果を確かめてからお使いください。デフォルトは、最内側ループの自動並列化を許可しません。このオプションは、PGI 6.1 から適用されました。
- -Mconcur=**altcode** は、並列化ループに対して、並列コードだけではなく、並列ではないスカラコードも同時に生成するように指示します。altcode のみの指定の場合、パラレライザは適切なカットオフ値（ループ長がその値より大きい場合に並列ルールを実行し、それよりも小さい場合にスカラコードを実行する、その閾値を言う）を設定します。また、altcode:n と指定した場合、ループ長が **n** 以上のとき並列コードを実行するようなプログラムが生成されます。-Mconcur=noaltcode は代替スカラコードを生成しません。
- -Mconcur=**altreduction[:n]** は、リダクション演算を含む並列ループに対して、並列コードだけではなく、スカラコードも同時に生成するように指示します。並列化ループにリダクションが含まれていた場合、ループ長が **n** 以下の場合のみ、代替スカラコードも生成します。

- -Mconcur=[**no]assoc** は、リダクションを伴うループの並列化を有効化/無効化します。リダクションとは、S = S + {operations} 形式の演算を言います。
- -Mconcur=[**no]numa** は、システムの NUMA ライブラリ（libnuma） を明示的にリンクする/させないためのオプションです（PGI 6.1 以降）。
- -Mconcur=**allcores** は、実行時の環境変数 OMP_NUM_THREADS あるいは NCPUS がセットされていない場合に、すべての有効なコアを使用するようにする。このフラグは、NUMA 構成のシステムでは、付けておいた方が無難でしょう。（リンク時に指定すること）。PGI 8.0 以降で有効です。
- -Mconcur=**bind** は、スレッドをコアあるいはプロセッサにバインドするようにします（リンク時に指定すること）。PGI 8.0 以降の Linux 上で有効です。
- -Mconcur=**level:n** は、多重ループ構成の場合の並列化対象の深さ(レベル)を指定するものです。デフォルトは n=3 です。
- -Mconcur オプションは、コンパイル時だけではなく、**リンク時のオプションとしても必要**です。特に、Makefile を使用している場合はご注意ください。指定しなければ、リンク時のエラーとなります。
- 上記のオプションは、自動並列化のオプションのみ例示しています。実際には、その他の最適化オプションも添えて、コンパイルしてください。

並列化プログラムを実行する際には、以下の **NCPUS** もしくは、**OMP_NUM_THREADS** 環境変数（並列実行で使用するスレッド数）を予めセットしてから実行してください（Windows 上でも同様に、export コマンドで環境変数を設定してください）。

```
setenv NCPUS 4    (csh 系)
    export NCPUS=4   (bash 系)
OpenMP の場合は
    setenv OMP_NUM_THREADS 4    (csh 系)
    export OMP_NUM_THREADS=4 (bash 系)
```

■ OpenMP 並列プログラムに対するコンパイル・オプション

```
pgfortran –fastsse –Minfo -mp[=align,numa,allcores] test.f
```

- **-mp** オプションは、共有メモリ型プログラミングモデルである OpenMP ディレクティブを用いた 並列プログラムを並列コンパイルする場合に使用します。このオプションは、コンパイル時だけではなく、**リンク時のオプションとしても必要**です。
- **-mp=[no]align** フラグは、並列化と SSE によるベクトル化の両方が適用されるループにおいて、ベクトル化のためのアライメント（整列）を最大化するようなアルゴリズムを使用して、OpenMP スレッドにループ回数を割り当てるようにするものです。この機能は、このような特性を帯びた多くのループがプログラムに存在する時に性能が向上します。しかし、一方、各ループの中で、非常に大きなタスク・ワークを含むループで、そのループ長が相対的に短いプログラムにおいては、結果としてロードバランスの問題が生じて大きく性能を落とす場合がありますので注意が必要です。このオプションを適用し性能を確認してから使用してください。（この align

サブオプションは、PGI 6.1 以降のオプションです)

- **-mp=[no]numa** フラグは、Linux/Windows が提供している NUMA ライブラリ（libnuma.so） を一般的なスレッドライブラリの代わりにリンクする機能を有する。また、それに伴い、コンパイラの最適化機能を改善しています。NUMA アーキテクチャの Linux/Windows システムでは、メモリアクセスと OpenMP 計算がより効率的に行えます。PGI 6.1 以降においては、-mp=numa の numa フラグは使用しなくても自動的にセットされます。PGI コンパイラをインストールした時点で、libnuma ライブラリを備えたシステムであれば、-Mconcur あるいは-mp の使用時に自動的にリンクできるようにコンパイラ・システム内部で設定されます。libnuma をリンクしたくない場合は、-Mconcur=nonuma あるいは -mp=nonuma を設定してください。PGI 6.2 以降では、システムに libnuma ライブラリが存在しない場合、PGI コンパイラは、そのための stub（仲介）ライブラリを提供します。

- **-mp=allcores** は、実行時の環境変数 OMP_NUM_THREADS あるいは NCPUS がセットされていない場合に、すべての有効なコアを使用するようにする。このフラグは、NUMA 構成のシステムでは、付けておいた方が無難でしょう。（リンク時に指定すること）。PGI 8.0 以降で有効です。

- **-mp=bind** は、スレッドをコアあるいはプロセッサにバインドするようにします（リンク時に指定すること）。PGI 8.0 以降の Linux 上で有効です。

■ OpenMP/自動並列化 実行時に設定する環境変数

- **OMP_NUM_THREADS** 環境変数は、並列実行で使用するスレッド数を指定するものです。

- **OMP_SCHEDULE** 環境変数は、DO あるいは PARALLEL DO ワークシェアリング並列実行時の分割方法のスケジューリング方法を定義します。デフォルトは、PGI の STATIC かつチャンクサイズは 1 です。以下は、設定方法の一例です。なお、この環境変数は OpenMP 並列性能を大きく変化させるものです。デフォルトは STATIC スケジューリングですが、これで性能が思わしくない場合、DYNAMIC 等をお試しください。この場合のプログラム上の OMP directive は、!$OMP DO schedule(runtime)であることが望ましいです。(CPU とメモリとの間のチャネル数にも依存して、並列性能は大きく変わります。使用スレッド数を物理的なコア数よりも小さく設定したほうが速い場合も往々にしてあります)

  ```
  $ setenv OMP_SCHEDULE “STATIC, 5”
  $ setenv OMP_SCHEDULE “GUIDED, 8”
  $ setenv OMP_SCHEDULE “DYNAMIC”
  ```

- **OMP_NESTED**（ネスト）は、以下の二つの条件の場合にのみ有効となります(PGI 2012 以降)。デフォルトでは、OMP_NESTED は無効となっております。

  1. 環境変数 OMP_NESTED=TRUE と OMP_MAX_ACTIVE_LEVELS=n の二つを必ず指定すること。Fortran の API を利用する場合は、ネスト構造の到達前に以下のような指定が必要です。

  ```
  --- 環境変数での指定の場合 ---
  export OMP_NESTED=TRUE
  export OMP_MAX_ACTIVE_LEVELS=2
  --- API を使用 ---
  use omp_lib
  call omp_set_nested(.true.)
  call omp_set_max_active_levels(2) ! (一例)
  ```

  2. orphaned nested parallelism の構成の場合に、ネスト実行が有効となります。例えば、

workshare 実行する部分が orphaned な別のルーチン配下で並列実行するような形を言います（parallel region 配下で call したルーチン内で omp do を行う形態)。要は、parallel directive が「別の」ルーチン（親ルーチン）で指定されている必要があります。

```
subroutine foo()
use omp_lib
!$OMP PARALLEL NUM_THREADS(6)
print *, "Hi from foo ", OMP_get_thread_num()
!$OMP END PARALLEL
end subroutine foo

program test1
use omp_lib

call OMP_set_num_threads(12)
call OMP_set_nested(.true.)
call OMP_set_max_active_levels(2)

print *, omp_get_max_active_levels()
print *, omp_get_nested()

!$OMP PARALLEL NUM_THREADS(2)

  print *, "Hello", OMP_get_thread_num()
  call foo()       ! orphaned routine --> nested parallel OK

!$OMP PARALLEL NUM_THREADS(4)
    print *, "Hi ", OMP_get_thread_num()   ! No nested using PGI compiler
!$OMP END PARALLEL

!$OMP END PARALLEL
end program test1

$ pgf90 -mp -Minfo nest.f90
foo:
       3, Parallel region activated
       4, Parallel region terminated
test1:
      18, Parallel region activated
      23, Parallel region activated
      24, Parallel region terminated
$ a.out
           2
  T
 Hello             0
 Hello             1
 Hi from foo               0      ここは nest される
 Hi from foo               2
 Hi from foo               3
 Hi from foo               4
 Hi from foo               1
 Hi from foo               5
 Hi from foo               0
 Hi from foo               2
 Hi from foo               1
```

```
Hi from foo             4
Hi from foo             5
Hi from foo             3
Hi              0       これは nest されない
Hi              0
```

- **OMP_DYNAMIC** 環境変数は現在のところ、利用できません。
- **MPSTKZ** 環境変数は並列リージョンの各スレッド・ローカルのスタックサイズを増加させるために使用します。setenv MPSTKZ 8M 等の方法で設定し、M は MB を意味します。Linux のデフォルトサイズは、多くの場合 2MB に設定されております。並列実行時に、セグメンテーション・フォールト等の問題が出た場合は、まず、このサイズを増加させてください。これでも問題が解決されなければ、Linux 上のハード・リミットが設定されている可能性があります。以下の記事「自動並列・OpenMP 並列実行時の Segmentation fault の対処法」をご参考にしてください。
- **OMP_WAIT_POLICY** (Proposed OpenMP 3.0 Feature)環境変数が、**PGI 7.0** から導入されました。この環境変数値は並列スレッドのアイドルの方法を指定するもので、特に、アイドル時、spin (busy wait) か sleep のどちらを使用するかを指定するものです。値は ACTIVE あるいは PASSIVE となります。この挙動は、自動並列化によって生成されるスレッドに対しても適用されます。並列実行時のスレッドがアイドルする局面は、バリア同期のとき、あるいはクリティカル・リージョンに入ったとき、あるいは、並列リージョン間のシリアル動作の場合等があります。この中でバリア同期は、常に MP_SPIN 環境変数によって指定される間隔で sched_yield システムコールによる busy wait を使用します。シリアルリージョンの実行中、待機しているスレッドは、バリア(ACTIVE) を使用した busy wait の状態か、あるいはオペレーティングシステム内の mutex(PASSIVE) を使用した politely wait のどちらかを選択でき、これは、OMP_WAIT_POLICY をセットすることにより可能となります。デフォルトは、ACTIVE です。

  ACTIVE をセットした時は、アイドルしているスレッド(busy wait)は 100%CPU を使用します。このメカニズムは、頻繁に並列リージョンに入り、出たりするような並列プログラムにとっては、並列リージョンに入る際、速い動作が可能となるため並列オーバーヘッドが小さくなる利点があります。一方、PASSIVE の場合は、次の並列リージョンに入るまで（スレッドが再起動されるまで)、CPU 時間を消費しない、オペレーティングシステム内の mutex 上での wait 動作となります。これは、シリアルリージョンが比較的長い場合とか、マルチユーザ環境で CPU リソースを共有しているような場合に設定します。
- **OMP_STACKSIZE** (Proposed OpenMP 3.0 Feature) 環境変数とスタックサイズ API が、**PGI 7.0** より新たに生成されるスタック（子スタック）のサイズを制御するためにサポートされました。API の関数としては、omp_set_stack_size と omp_get_stack_size が実装されております。この環境変数は、新たにスレッドが生成される度にこの値が反映され、システムのデフォルト値を上書きします。数字の後に B,K,M,G のサイズ識別子を付加することができます。B=Byte,K=Kbyte,M=Megabyte,G=Gigabayte の意味です。
- **OMP_MAX_ACTIVE_LEVELS** 変数は現在、無効となっております。一般には、ネスト並列性を有効（無効）にします。 PGI 8.0 以降で有効です。
- **OMP_THREAD_LIMIT** 変数は、デフォルト 256 です。プログラムで使用可能な最大スレッド数の絶対数です。PGI 8.0 以降で有効です。

■ 並列性能に効果が期待されるコンパイル・オプション

```
pgfortran {-mp|-Mconcur} -Munsafe_par_align -Mnontemporal –Mmovnt
```

- **-M[no]unsafe_par_align** オプションは、並列化ループでの配列の参照において、その配列の最初の要素が「整列」されている場合、「整列移動（aligned moves)」を行うことが安全であるとみなします。 NOTE: 但し、このオプションは、コンパイラがその安全性を疑った場合でも、「整列移動」で行うコードを生成しますので、その計算結果の検証は必ず行ってください。このオプションは、特に、 STREAM Benchmark やメモリ・インテンシブなメモリアクセスを含むループブロックの並列化で効果を発揮しますが、標準利用のオプションとして推奨するものではありません。
- **-Mnontemporal** オプションは、non-temporal ストア並びにプリフェッチの生成を強制するオプションです。-fastsse と共に使用します。性能が向上する場合があります。
- **-M[no]movnt** オプションは、これまで使用してきた-Mnontemporal を置き換えるものです。(PGI 6.1 以降)

■ プログラムの自動並列化コンパイル例と実行例

```
（コンパイル）
 $ pgfortran -fastsse -Mconcur -Minfo vadd.f
 vector_op:
      4, Parallel code generated; block distribution
         Unrolling inner loop 8 times
 loop:
     18, Parallel code activated if loop count >= 100; block distribution
         Generating sse code for inner loop
         Generated prefetch instructions for 3 loads
（実行）
 $ export NCPUS=2
```

あるいは、

```
 $ export OMP_NUM_THREAD=2

 $ /bin/time a.out
 -1.000000 -771.000 -3618.000 -6498.00 -9999.00
   8.00user 0.55system 0:04.27 elapsed 200%CPU
```

■ 自動並列・OpenMP 並列実行時の Segmentation fault の対処法

この Segmentation fault の問題は、自動並列化あるいは OpenMP 並列化を行った際のスレッド実行時の問題です。共有メモリ上の複数の CPU を使用した並列実行では、実行時に並列プロセス上に存在する「スタック領域」を利用します。このスタック領域は、各 OS のビルド時に default として設定されておりますが、このデフォルトの stack size の領域を上回って使用される場合、このエラーが生じます。即ち、実行プログラムの配列サイズが大きくなった場合に、実行プロセスのスタックサイズも比例して大きなものを利用しますので、これを超えたためのエラーです。対処法としては、まず、上記の「OpenMP/自動並列化実行時に設定する環境変数」の中の **MPSTKZ** 環境変数および、**OMP_STACKSIZE** 環境変数でスレッドライブラリのサイズを変更してみて下さい。これでも駄目な場合は、ログインしたシェル環境の中で、stack size を明示的に大きく指定する方法をとらなければなりません。これは、PGI の制約ではなく、一般的なスレッド方式の並列処理上で付きまとう、よく知られた問題です。以下の方法でそのサイズを変更してください。

ログインシェルが B シェル系(sh,bsh)の場合と C シェル(csh,tcsh)系の場合とでコマンドが異なります。

B シェル系の場合のサイズを変更する　build-in command は **ulimit** です。
C シェル系の場合のサイズを変更する　build-in command は **limit** です。

現在の stacksize に限らず、プロセスのパラメータのサイズを見るには、

【B シェル系】

```
$ ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
file size (blocks, -f) unlimited
max locked memory (kbytes, -l) unlimited
max memory size (kbytes, -m) unlimited
open files (-n) 1024
pipe size (512 bytes, -p) 8
stack size (kbytes, -s) unlimited
cpu time (seconds, -t) unlimited
max user processes (-u) 16379
virtual memory (kbytes, -v) unlimited
```

スタックサイズのものだけを見たい場合、

```
$ ulimit -s
unlimited
```

【C シェル系】

```
$ limit
cputime unlimited
filesize unlimited
datasize unlimited
stacksize unlimited
coredumpsize 0 kbytes
memoryuse unlimited
vmemoryuse unlimited
descriptors 1024
memorylocked unlimited
maxproc 16379
```

スタックサイズのものだけを見たい場合、

```
$ limit stacksize
stacksize unlimited
```

となります。もし、デフォルトで、stacksize が unlimited となっている場合、この unlimited と言うのは無制限と言う意味ではありますが、実際は、これが無制限と言う振る舞いをしないため、明示的な数字で指定する必要があります。
従って、最大上限スタックサイズを明示的な数字で指定してください。以下の例では、16384 KByte を指定する場合の例です。指定する数字の単位は、KByte です。また、1024 の倍数にしてください。

【B シェル系】
$ ulimit -s 16384
（確認する）
$ ulimit -s
16384

（注意）B シェル系の場合、一度指定したら、このシェル環境上ではこの上限サイズは変更できません。1 回のみです。たとえ変更しようとしても、エラーメッセージが表示されます。この場合は、一度、ログアウトして再度、ログインして同じような手続きを行ってください。

【C シェル系】
$ limit stacksize 16384
（確認する）
$ limit stacksize
stacksize 16384 kbytes

（注意）C シェル系の場合は、何度でも指定を変更できます。

以上のような操作で、シェル環境のスタックサイズの上限を適宜指定してください。どの値が適切かは、何度か試行錯誤をしていただくことになりますが、コンパイラ・オプション **–Mchkstk** を指定して並列実行すると、stack overflow の状態がエラーメッセージとして現れますのでこれを参考にすることもできます。この実行を行う前には、PGI_STACK_USAGE 環境変数の設定を行って下さい。

```
$ export PGI_STACK_USAGE=yes
$ pgfortran –fastsse –mp –Minfo –Mchkstk test.f90
$ ulimit –s 102400
$ time a.out
**ERROR: in routine cns3 there is a
stack overflow: thread 0, max 102388KB, used 0KB, request -1416120528B
```

■ NUMA アーキテクチャ上でのスレッド制御を行うためのマルチプロセッサ環境変数

PGI コンパイラは、他社のコンパイラにはない、AMD 社の Opteron やインテル社の Nehalem 以降の NUMA アーキテクチャ上での OpenMP スレッド並列を制御するための環境変数を提供します。これをマルチプロセッサ環境変数と言い、特にデュアルコア・プロセッサを含むシステムのスレッド並列制御に有効です。ここでは、この環境変数の機能について紹介します。

コンパイラ・オプションとして PGI 6.0-5 より、-mp=numa （numa サブオプション）が追加されました。これは、NUMA アーキテクチャを採用する **AMD** 並びに**インテル**のマルチ（コア）プロセッサシステム用のオプションです。インテル®のプロセッサシステムでは、nehalem アーキテクチャ以降で NUMA を採用しています。
-mp オプションは、 OpenMP の指示行を含むプログラムの並列処理モジュールを生成するオプションですが、その性能の向上のために、NUMA システムライブラリをリンクし、process(thread)-to-CPU スケジューリングの CPU affinity (親和度)等の機能を有効にするオプションとなります。NUMA アーキテクチャのシステムにおいて有効であり、現在、NUMA 対応 LinuxやMicorosoft Windows のOS 下でこの NUMA ライブラリをリンクすることが可能です。これによって、NUMA アーキテクチャに対するユーザ側での制御自由度が向上されております。最新の Intel Sandybridge プロセッサ上の Windows OS は、NUMA 対応となっていますので、この環境変数は有効です。特に、メモリ・バウンドな特性を有するプログラムでは、OpenMP 等によ

る並列実行を行う際に、MP_BIND の指定と MP_BLIST を変更することによって性能が大きく変わることがあります。

マルチコアプロセッサを含む SMP 環境でのスレッド制御を行うための新しいマルチプロセッサ (mp) 環境変数 (**MP_BIND**、**MP_BLIST**、**MP_SPIN)** が PGI 6.0-5 から追加されました。これらの環境変数は、スレッドの実行を別の CPU に移行(切り替え）させるかどうかの設定 (MP_BIND)、スレッドを物理的な CPU にバインドして固定させるための設定 (MP_BLIST)、バリア同期の待ち状態に入ったと時に、アイドル・スレッドがどの程度、プロセス・サイクルを消費するかを制御する設定 (MP_SPIN) を行うものです。

- **MP_BIND** 環境変数 － OpenMP スレッドを物理 CPU にバインド（結合）するために、MP_BIND 環境変数を導入しました。この環境変数を yes あるいは y とセットするとこの機能が有効となり、no あるいは n をセットするとこのバインド機能は無効となります。デフォルトは no です。これは実行時に OpenMP ランタイム・サポート・ライブラリによって解釈され、PGI コンパイラの挙動に対して影響を及ぼすものではありません。-mp=numa オプションをつけてコンパイルされたモジュールに対して有効です。この機能は、NUMA 対応の Linux や Micorosoft Windows の OS 下で、かつ、-mp=numa オプションをつけてコンパイルされたモジュールに対して有効です。最新の Intel Sandybridge プロセッサ上の Windows OS は、NUMA 対応となっていますので、この環境変数は有効です。特に、メモリ・バウンドな特性を有するプログラムでは、OpenMP 等による並列実行を行う際に、MP_BIND の指定と MP_BLIST を変更することによって性能が大きく変わることがあります。最初は、MP_BLIST を指定しないで、MP_BIND を yes として設定してから、性能を確かめて下さい。MP_BIND=y を設定すると、OS が物理的な CPU コアに、実行スレッドを 1 対 1 にバインドします。
- **MP_BLIST** 環境変数 － MP_BLIST 環境変数を新たに追加しました。スレッドと CPU の固定を定義するために使用され、例えば、MP_BLIST=3,2,1,0 の指定では、CPU 3,2,1,0 がそれぞれスレッド 0,1,2,3 にマッピングされます。-mp=numa オプションをつけてコンパイルされたモジュールに対して有効です。
- **MP_SPIN** 環境変数 － MP_SPIN 環境変数を新たに追加しました(6.0-5) 。並列リージョンで実行しているスレッドが、バリア同期に入った場合、セマフォ上でのスピン実行を行います。MP_SPIN は、sched_yield() システムコールが呼び出されるされる前に、セマフォを確認する時間（サイクル数）を指定するために使われます。sched_yield() システムコールは、他のプロセスに実行を許すために、スレッドを再スケジューリングするためのものですが、並列処理においては、他にスケジューリングされない方が望ましいですので、この環境変数でその時間を設定します。デフォルトは、1000000 (Linux)サイクルです。もし、MP_SPIN=-1 とセットした場合、セマフォ上で、sched_yield() コールせずに、スピンし全スレッドが同期するまで待ちます。

使用方法の詳細は、弊社ホームページ（http://www.softek.co.jp/SPG/Pgi/TIPS/numa_bind.html）をご覧ください。

2.4 コンパイル時の様々な実行情報を表示するためのオプション

PGI の F77, F2003, HPF, C, C++ のコンパイラを使用する際に、コンパイラのメッセージ、情報を得るための基本的なオプションを説明します。以下は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。

■ コンパイラが最適化を施した部分の情報を得る

```
$ pgfortran -fastsse -Mvect=prefetch -Minfo Test.f

    initia:
       195, Loop unrolled 8 times
       204, Loop unrolled 8 times
    fielde:
        56, Interchange produces reordered loop nest: 57, 56
        90, 1 loop-carried redundant expression removed with 3 operations
            and 4 arrays
       231, Unrolling inner loop 8 times
            Generated prefetch instructions for 3 loads
       239, Unrolling inner loop 8 times
            Generated prefetch instructions for 2 loads
    bounde:
       269, Unrolling inner loop 8 times
            Generated prefetch instructions for 2 loads
       360, Generating vector sse code for inner loop
       383, Generating vector sse code for inner loop
    fieldh:
       485, Unrolling inner loop 8 times
            Generated prefetch instructions for 4 loads
    boundh:
       515, Generating vector sse code for inner loop
            Generated prefetch instructions for 2 loads
    array:
       654, Unrolling inner loop 8 times
       665, Unrolling inner loop 8 times
       676, Unrolling inner loop 8 times
       705, Generating vector sse code for inner loop
       720, Generating vector sse code for inner loop
       735, Generating vector sse code for inner loop
       770, Generating vector sse code for inner loop
       775, Generating vector sse code for inner loop
    build_xxx:
       796, Generating vector sse code for inner loop
            Generated prefetch instructions for 3 loads
```

- **-Minfo** オプションは、コンパイル時の様々な情報を出力するためのオプションです。一般には、上記のとおりフラグなしで指定することで、ほとんどの最適化情報が得られます。
- **-Minfo=**[=flag [,flag...]] と言うフラグを指定すると表示する情報を制御できます。フラグの詳細は、PGI User's Guide を参照して下さい。-Minfo=all は全ての情報が表示されます。その他、以下の flag があります。

all　　以下のサブオプションをすべて指定したものと解釈します。

accel	PGI Accelerator 最適化に関する情報
ccff	common compiler feedback format で最適化情報をオブジェクトファイル追加
ftn	Fortran 特有な情報の有効化
lre	LRE 情報の有効化
inline	インライン化に関する情報
intensity	ループの計算密度の出力
ipa	IPA 最適化情報
loop	ベクトル化等のループに関する情報
mp	OpenMP 並列化に関する情報
par	並列化に関する情報
opt	最適化に関する情報
pfo	Profile Feed back 最適化に関する情報
time	コンパイル時間統計の出力
unroll	アンロール最適化情報
par	並列化の情報の有効化
pfo	プロファイル・フィードバックに関する情報の有効化
vect	ベクトル化に関する情報

■ コンパイル時に最適化が不可能であった部分のみの情報を得る

```
$ pgfortran -fastsse -Mneginfo Test.f

    main:
         67, Loop not vectorized: contains call
    initia:
        195, Loop not vectorized due to data dependency
    bound:
        306, Loop not vectorized: contains call
```

- **-Mneginfo**[=all,concur,loop] オプションを指定すると、最適化でが阻害された部分とその理由を表示します。

■ コンパイラメッセージの出力レベルを指示する

```
$ pgcc -c t.c                                  (デフォルトは全てのメッセージを出す)
  PGC-W-0119-void function main cannot return value (t.c: 13)   これは warning
  PGC/x86-64 Linux 12.4-0: compilation completed with warnings

$ pgcc -c -Minform=severe   t.c     (Severe、Fatal のみのメッセージを出す)
   PGC/x86-64 Linux 12.4-0: compilation completed with warnings
```

- **-Minform=level** オプションは、コンパイラのメッセージのレベルを指定します。

  -Minform=inform : inform、warn, severe, fatal の全てのメッセージを出力
  -Minform=warn : warn, severe, fatal メッセージを出力
  -Minform=severe : severe, fatal メッセージを出力
  -Minform=fatal : fatal メッセージのみ出力

■ 自動並列化を行った時の並列化情報メッセージ

```
$ pgfortran -fastsse -Mconcur -Minfo Test.f

  initia:
     195, Loop unrolled 8 times
     204, Parallel code activated if loop count >= 100; block distribution
          Loop unrolled 8 times
  fielde:
     230, Parallel code for non-innermost loop generated; block distribution
     231, Unrolling inner loop 8 times
     238, Parallel code for non-innermost loop generated; block distribution
     239, Unrolling inner loop 8 times
  bounde:
     269, Parallel code activated if loop count >= 100; block distribution
          Unrolling inner loop 8 times
     279, Parallel code activated if loop count >= 100; block distribution
     295, Parallel code activated if loop count >= 100; block distribution
     302, Parallel code activated if loop count >= 100; block distribution
     321, Parallel code activated if loop count >= 100; block distribution
     337, Parallel code activated if loop count >= 100; block distribution
     346, Parallel code activated if loop count >= 100; block distribution
     360, Parallel code activated if loop count >= 100; block distribution
          Generating vector sse code for inner loop
     369, Parallel code activated if loop count >= 100; block distribution
     383, Parallel code activated if loop count >= 100; block distribution
          Generating vector sse code for inner loop
  fieldh:
     484, Parallel code for non-innermost loop generated; block distribution
     485, Unrolling inner loop 8 times
  boundh:
     515, Parallel code activated if loop count >= 100; block distribution
          Generating vector sse code for inner loop
  array:
     653, Parallel code for non-innermost loop generated; block distribution
     654, Unrolling inner loop 8 times
     664, Parallel code for non-innermost loop generated; block distribution
     665, Unrolling inner loop 8 times
     675, Parallel code for non-innermost loop generated; block distribution
     676, Unrolling inner loop 8 times
     696, Parallel code for non-innermost loop generated; block distribution
     705, Parallel code activated if loop count >= 100; block distribution
          Generating vector sse code for inner loop
     711, Parallel code for non-innermost loop generated; block distribution
     720, Parallel code activated if loop count >= 100; block distribution
          Generating vector sse code for inner loop
     726, Parallel code for non-innermost loop generated; block distribution
     735, Parallel code activated if loop count >= 100; block distribution
```

```
        Generating vector sse code for inner loop
    770, Parallel code activated if loop count >= 100; block distribution
        Generating vector sse code for inner loop
    775, Parallel code activated if loop count >= 100; block distribution
        Generating vector sse code for inner loop
build_xxx:
    796, Parallel code activated if loop count >= 100; block distribution
        Generating vector sse code for inner loop
pml:
    829, Parallel code for non-innermost loop generated; block distribution
    850, Parallel code for non-innermost loop generated; block distribution
    871, Parallel code for non-innermost loop generated; block distribution
    892, Parallel code for non-innermost loop generated; block distribution
```

- **-Mconcur** オプションは自動並列化を行うオプションです。コンパイラが抽出した自動並列化箇所に対する並列コード生成の情報を出力しています。

■ ソースプログラムのリスティング・ファイルを作成する

```
$ pgfortran   -fastsse -Mlist -Minfo Test.f

   PGF90 (Version      13.8)          09/26/2013   14:52:53       page 1

   Switches: -noasm -nodclchk -nodebug -nodlines -noline -list
             -inform severe -opt 2 -nosave -object -noonetrip
             -depchk on -nostandard
             -nosymbol -noupcase

   Filename: stream.f

   (     1) *===============================================
   (     2) * Program: STREAM
   (     3) * Programmer: John D. McCalpin
   (     4) * RCS Revision: $Id: stream.f,v 5.6 2005/10/04 00:20:48 mccalpin Exp
   (     5) * mccalpin
   (     6) *===============================================
```

- **-Mlist** オプションを指定すると、**xxxx.lst** と言う名称でソース・リスティングファイルを作成します。また、コンパイル時に使用された最適化スイッチの内容も記されます。

■ ソースプログラムとその生成アセンブラのリスティングを対応付けて出力する

```
$ pgfortran   -fastsse -Manno -S Test.f
  あるいは、
  $ pgcc    -fastsse -Manno -Mkeepasm Test.c  (C の場合はーMkeepasm を入れる)

   .LB1_836:
   # lineno: 151
   #             DO 60 j = 1,n
   #                 a(j) = b(j) + scalar*c(j)
   #     60      CONTINUE
           movapd   %xmm0, %xmm1
```

```
        movapd   (%esi,%ecx), %xmm2
        movapd   16(%esi,%ecx), %xmm3
        subl     $8, %eax
        mulpd    %xmm1, %xmm2
        mulpd    %xmm1, %xmm3
        addpd    (%edi,%ecx), %xmm2
        addpd    16(%edi,%ecx), %xmm3
        movapd   %xmm2, (%edx,%ecx)
        movapd   32(%esi,%ecx), %xmm2
        movapd   %xmm3, 16(%edx,%ecx)
        mulpd    %xmm1, %xmm2
        mulpd    48(%esi,%ecx), %xmm1
        addpd    32(%edi,%ecx), %xmm2
        addpd    48(%edi,%ecx), %xmm1
        movapd   %xmm2, 32(%edx,%ecx)
        movapd   %xmm1, 48(%edx,%ecx)
        addl     $64, %ecx
        testl    %eax, %eax
        jg       .LB1_836
# lineno: 154
```

- **-Manno** はアセンブリ言語コードと共にソースコードを注釈するオプションです。-Manno -S の指定により、アセンブリ言語リスティング・ファイル **xxxx.s** が生成され、その中にソース・リストとそれに対するアセンブリ言語リストが両方表示される。
- **-S** オプションはリンケージ処理を行わない。このコマンド実行後、Test.s と言うアセンブリ言語リスティング・ファイルに、対応するソースコードも注釈される。
- **-Mkeepasm** は、生成されたアセンブリ言語のリスティングを **xxxx.s** ファイルに出力します。

■ 指定したコンパイル・オプションの意味を知る

```
$ pgfortran   -fastsse -flags -Minfo Test.f

   Reading rcfile /usr/pgi/linux86-64/13.8/bin/.pgfortranrc
   -fastsse                == -fast
   -fast                   Common optimizations; includes -O2 -Munroll=c:1
                           -Mnoframe -Mlre -Mautoinline
                           == -Mvect=sse -Mcache_align -Mflushz -Mpre
   -M[no]vect[=[no]altcode|[no]assoc|cachesize:<c>|[no]fuse|[no]gather|[no]idiom|levels:
   <n>|[no]partial|prefetch|[no]short|[no]simd|[no]sizelimit[:n]|[no]sse|[no]tile|[no]uniform
   ]
                           Control automatic vector pipelining
       [no]altcode         Generate appropriate alternative code for vectorized loops
       [no]assoc           Allow [disallow] reassociation
       cachesize:<c>       Optimize for cache size c
       [no]fuse            Enable [disable] loop fusion
       [no]gather          Enable [disable] vectorization of indirect array references
       [no]idiom           Enable [disable] idiom recognition
       levels:<n>          Maximum nest level of loops to optimize
       [no]partial         Enable [disable] partial loop vectorization via inner loop distribution
       prefetch            Generate prefetch instructions
       [no]short           Enable [disable] short vector operations
       [no]simd            Generate [don't generate] SIMD instructions
        128                Use 128-bit SIMD instructions
```

```
    256             Use 256-bit SIMD instructions
[no]sizelimit[:n]
                    Limit size of vectorized loops
[no]sse             Generate [don't generate] SSE instructions
[no]tile            Enable [disable] loop tiling
[no]uniform         Perform consistent optimizations in both vectorized and residual
loops; this may affect the performance of the residual loop
-Mcache_align       Align large objects on cache-line boundaries
-M[no]flushz        Set SSE to flush-to-zero mode
-M[no]pre           Enable partial redundancy elimination
```

- **-flags** オプションを指定すると、実際のコンパイルは行いませんが、指定したコンパイル・オプションの意味とそのフラグを表示します。
- 同様なオプションとして、**-help** がありますが、このオプションは PGI コンパイラが提供する全オプションのリストとその意味を標準出力に出力します。また、 -help -<option> を同時に指定すると、上記 -flags と同等な機能を提供します。

■ PGI コンパイラの内部手続き（コード生成、アセンブラ、リンケージ）の詳細を見る

```
$ pgcc -fastsse -v stream.c

    【PGI コンパイラによるコード生成フェーズ】

    /usr/pgi/linux86-64/13.8/bin/pgc stream.c -opt 2 -x 119 0xa10000 -x 122 0x40 -x 123
    0x1000 -x 127 4 -x 127 17 -x 19 0x400000 -x 28 0x40000 -x 120 0x10000000 -x 70 0x8000
    -x 122 1 -x 125 0x20000 -quad -vect 56 -y 34 16 -x 34 0x8 -x 32 8388608 -y 19 8 -y 35 0 -x
    42 0x30 -x 39 0x40 -x 199 10 -x 39 0x80 -x 34 0x400000 -x 149 1 -x 150 1 -x 59 4 -x 59 4
    -tp nehalem -x 120 0x1000 -astype 0 -x 121 1 -stdinc
    /usr/pgi/linux86-64/13.8/include:/usr/local/include:/usr/lib/gcc/x86_64-redhat-linux/4.4.6/
    include:/usr/lib/gcc/x86_64-redhat-linux/4.4.6/include:/usr/include -def unix -def __unix
    -def __unix__ -def linux -def __linux -def __linux__ -def __NO_MATH_INLINES -def
    __x86_64 -def __x86_64__ -def __LONG_MAX__=9223372036854775807L -def
    '__SIZE_TYPE__=unsigned long int' -def '__PTRDIFF_TYPE__=long int' -def __THROW=
    -def __extension__= -def __amd_64__amd64__ -def __k8 -def __k8__ -def __SSE__ -def
    __MMX__ -def __SSE2__ -def __SSE3__ -def __SSSE3__ -predicate '#machine(x86_64)
    #lint(off) #system(posix) #cpu(x86_64)' -cmdline '+pgcc stream.c -fastsse -fast
    -Mvect=sse -Mcache_align -Mflushz -Mpre -v' -x 123 0x80000000 -x 123 4 -x 119 0x20 -def
    __pgnu_vsn=40406 -alwaysinline /usr/pgi/linux86-64/13.8/lib/libintrinsics.il 4 -autoinl 10 -x
    168 100 -x 174 8000 -x 14 0x200000 -x 120 0x200000 -x 70 0x40000000 -x 9 1 -x 42
    0x14200000 -x 72 0x1 -x 136 0x11 -quad -x 119 0x10000000 -x 129 0x40000000 -x 129 2
    -x 164 0x1000 -asm /tmp/pgcciMZcyDmKLlcg.sm
    PGC/x86-64 Linux 13.8-0: compilation completed with informational messages

    【アセンブラ(as)でオブジェクトを作成するフェーズ】

    /usr/pgi/linux86-64/13.8/bin/pgsmart -agg 0x62000020 -o /tmp/pgcciMZcyiWjTW3N.s
    /tmp/pgcciMZcyDmKLlcg.sm

    /usr/bin/as /tmp/pgcciMZcyiWjTW3N.s -o /tmp/pgcciMZcylX7_l0T.o
```

【リンカ ld でリンケージするフェーズとそのオプション】

/usr/bin/ld /usr/lib64/crt1.o /usr/lib64/crti.o /usr/pgi/linux86-64/13.8/lib/trace_init.o /usr/lib/gcc/x86_64-redhat-linux/4.4.6/crtbegin.o /usr/pgi/linux86-64/13.8/lib/initmp.o -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/pgi/linux86-64/13.8/lib/pgi.ld -L/usr/pgi/linux86-64/13.8/lib -L/usr/lib64 -L/usr/lib/gcc/x86_64-redhat-linux/4.4.6 /tmp/pgcciMZcylX7_l0T.o -rpath /usr/pgi/linux86-64/13.8/lib -lpgmp -lnuma -lpthread -lnspgc -lpgc -lm -lgcc -lc -lgcc /usr/lib/gcc/x86_64-redhat-linux/4.4.6/crtend.o /usr/lib64/crtn.o （リンクされるライブラリの順序等の確認が可能)

- -v オプション（あるいは、-#）を指定すると、PGI コンパイラの内部手続き（コード生成、アセンブラ、リンケージ）の各コマンドのエコーバックが得られます。また、各コマンドにはその詳細全オプションが出力されます。
- このオプションが有効な場面は、リンケージにおけるライブラリのリンク状況を知ることができるため、例えば静的リンク時におけるトラブルとして多い、未解決なライブラリがどのような順番でリンクされているか等の確認ができます。

■ 標準 Fortran に準拠していない構文の確認

```
$ pgfortran -c -Mstandard fftpde.F
PGF90-W-0170-F90 extension: c-style #line directive  (fftpde.F: 1)
PGF90-W-0173-F90 extension: nonstandard use of data type length specifier (fftpde.F: 17)
PGF90-W-0173-F90 extension: nonstandard use of data type length specifier (fftpde.F: 30)
  0 inform,   3 warnings,   0 severes, 0 fatal for fftpde

$ pgf77 -c -Mstandard stream.f
PGFTN-W-0171-F77 extension: nonstandard statement type DO (stream.f: 419)
PGFTN-W-0171-F77 extension: nonstandard statement type ENDDO (stream.f: 424)
```

- **-Mstandard** オプションを指定すると、標準 Fortran に準拠していない構文（拡張構文を含む）が使用された場合、その内容を警告メッセージとして出力します。

■ オプションヘルプ(-help) でオプションの意味を調べる

```
$ pgfortran -help=group
Switch Classifications:  (オプションスイッチのカテゴリを表示)
overall         Overall switches
opt             Optimization switches
debug           Debugging switches
prepro          Preprocessor switches
asm             Assembler switches
linker          Linker switches
language        Language-specific switches
target          Target-specific switches
other           Other switches

例えば、カテゴリ linker(リンケージ処理）に関するオプションスイッチを表示
$ pgfortran -help=linker  (Linux 版の場合の表示)
```

```
  Linker switches:
  -acclibs              Append Accelerator libraries to the link line
  --[no-]as-needed      Passed to linker; only set DT_NEEDED for the following shared libraries if
they are used
  -Bdynamic             Passed to linker; specify dynamic binding
  -Bstatic              Passed to linker; specify static binding
  -Bstatic_pgi          Use to link static PGI libraries with dynamic system libraries; implies
-Mnorpath
  -Bsymbolic            Passed to linker; specify symbolic binding
  -cudalibs             Link with CUDA-enabled libraries
  -g77libs              Include g77 or gfortran library when linking
  -L<libdir>            Passed to linker; Add directory to library search path
  -l<lib>               Passed to linker; Add library name to library search list
  -m                    Passed to linker; display link map
  -Mcudalib[=cublas|cufft|curand|cusparse]
                        Add appropriate versions of the CUDA-optimized libraries
  -M[no]eh_frame        Add link flags to preserve exception-handling frame information
  -Mlfs                 Link with library directory for large file support
  -Mmpi=mpich|sgimpi|mpich1|mpich2|mvapich1
                        Use default or specified MPI communication libraries
      mpich             Use default or MPIDIR-specified MPICH v3.0 libraries
      sgimpi            Use default or MPI_ROOT-specified SGI MPI libraries
      mpich1            DEPRECATED: Use MPIDIR to specify MPICH1 libraries
      mpich2            DEPRECATED: Use MPIDIR to specify MPICH2 libraries
      mvapich1          DEPRECATED: Use MPIDIR to specify MVAPICH1 libraries
  -Mnorpath             Don't add -rpath paths to link line
  -Mnostartup           Do not use standard linker startup file
  -Mnostdlib            Do not use standard linker libraries
  -Mscalapack           Add Scalapack libraries
  -pgc++libs            Append gnu compatible C++ libraries to the link line
  -pgcpplibs            Deprecated: Append C++ libraries to the link line
  -pgf77libs            Append pgf77 libraries to the link line
  -pgf90libs            Append pgf90 libraries to the link line
  -R<ldarg>             Passed to linker; just link symbols from object, or add directory to run
time search path
  -r                    Generate relocatable object; passed to linker
  -rpath <dir>          Passed to linker; add directory to runtime shared library search path
  -s                    Passed to linker; strip symbol table from exectable
  -shared               Used to produce shared libraries; implies -fpic
  -soname <soname>      Passed to linker; sets DT_SONAME field to the specified name
  -u<undef>             Passed to linker; generate undefined reference
  --[no-]whole-archive
                        Passed to linker; includes all objects in subsequent archives
  -Wl,<arg>             Pass argument to linker
  -YC,<complibdir>      Change compiler library directory
  -YL,<stdlibdir>       Change standard library directory
  -Yl,<linkdir>         Change linker directory
  -YS,<startupdir>      Change startup object directory
```

- **-help** オプションで、オプションのカテゴリ別に整理されたオプション・フラグの意味を表示します。

2.5 オブジェクトのリンク時に使用するオプション

PGI の F77, F2003, HPF, C, C++ のコンパイラを使用してプログラムを開発する際に、リンク時の有用なオプションを以下に示します。以下の機能は、pgfortran 以外のコマンドでも同じです。

PGI オブジェクトのバージョン間の互換性

PGI コンパイラで作成されたオブジェクトファイル、ライブラリは、各 PGI バージョン間で互換性がない場合があります。互換性がない以前のバージョンのオブジェクトに関しては、再コンパイルが必要となります。また、PGI 6.1 以降においても、-Mipa オプションを付して作成されたオブジェクトも再コンパイルが必要となります。

■ リンク時における外部ライブラリの指定

```
(Linux/OS X)
  pgfortran -fastsse -o a.out test.f -L/opt/lib -lacml64
(Windows)
   pgfortran -Minfo -fastsse -L"C:/Program Files (x86)/Intel/.../mkl/lib/intel64" ¥
   mkl_intel_lp64.lib mkl_pgi_thread.lib mkl_core.lib -mp pgi_mm.f -o pgi_mm.exe
```

- **-L** オプションは、明示的に外部ライブラリを指定する際の検索ライブラリ・パスを指定するものです。上記の例では、/opt/lib 配下のライブラリを探せと言う指示になります。なお、明示的なライブラリの指定では、リンク時に、システムライブラリより上位でリンク処理を行います。なお、-L の指定では、そのパス名をブランクを入れずに続けて記述する必要があります。
- **-l** は、リンクすべきライブラリ名を指定します。上記の例では、libacml64.a と言うライブラリをリンクせよと言う指示になります。ライブラリ・ファイル名の上位の 'lib' と末尾の '.a' を記述する必要はありません。-L、-l 共に、その後続文字列は空白を入れずに指定します。
- (windows)ライブラリ・ファイル名(***.lib)をそのまま、コマンドリストに記述します。

■ 動的な shared library を含まない実行モジュールを作成する （Linux、Windows）

```
pgfortran -fastsse -Wl,-Bstatic test.f (あるいは、-Bstatic でもよい)
```

- 一般に Linux 上の実行モジュールの形式は、システム的に共通な shared library は動的なものとしてリンクされ、実行時に必要なライブラリを読み込んで実行する形式となっています。一般的な shared library の名前のコンベンションは、libpgc.so と言った .so と言うサフィクスが付けられています。
- 同様に、Windows 環境においても実行時に Dynamic Link Library(DLL) を必要とする動的リンク形式で動作する場合が多いです。Windows 版の PGI 7.0 までは、実行形式モジュールが、PGI の提供する DLL ファイル(pg.dll) と Microsoft(R) のマルチスレッド対応のランタイムライブラリを必要としましたが、PGI 7.1 以降は、静的ライブラリ形式の実行モジュールの生成がデフォルトとなりました。また、明示的に、静的ライブラリをリンクする際は、**-Bstatic** オプションを指定します。また、その逆に、DLL を必要とする実行モジュールを生成する際は、**-Bdynamic** を指定します。Windows では、これらのオプションは、コンパイル時並びにリン

ク時の両方に指定する必要があります。

- コンパイラをインストールしたシステム上で実行する場合は、PGI コンパイラが提供する動的ライブラリを自動的に取り込みますが、作成した実行モジュールを PGI コンパイラがインストールされていない別の Linux システム上で動かそうとした場合、PGI の動的ライブラリ（libpgc.so 等）が存在しないため動作しません。そこで、実行モジュールを生成する際に、動的ライブラリのリンクではなく静的なライブラリ（実態のあるライブラリ）を全てリンクしておきたいと言う要求が出てきます。 **-Wl,-Bstatic** は、静的にライブラリをリンクし、どこでも動作可能な実行モジュールを作成するためのオプションです。但し、この実態のある静的ライブラリをリンクしますので、モジュールのサイズは大きくなります。まお、PGI 5.1 リリース以降は、同様なコンパイル・オプションとして、**-Bstatic** が提供されております。

- Linux において、**-mcmodel=medium** オプションを使用してコンパイルしたオブジェクトは、静的なリンクはできません。これは、Linux86-64 プログラミングモデル上での制約であり、PGI コンパイラ特有の制約ではありません。

(注意) 静的なリンク方法で、リンク時に未解決な参照ルーチンがあるというメッセージでエラーになる場合があります。これに関する記事、またリンク操作に関する裏技等に関しては、弊社のホームページ（TIPS 情報）にて、詳細に説明しています。
（参考 URL）http://www.softek.co.jp/SPG/Pgi/TIPS/link.html

■ PGI 専用ライブラリのみ静的にリンクし、Linux システムライブラリは動的にリンク（Linux）

```
pgfortran -fastsse –Bstatic_pgi test.f
```

- PGI が提供する専用ライブラリは静的リンクで行い、Linux のシステム依存ライブラリは、ダイナミックにリンクさせるためのオプションです。これは PGI 6.2 から新設されたオプションです。このオプションで生成された実行モジュールは、他の Linux システム（OS の glibc バージョン等が同じもの）においても、PGI が提供する専用シェアードライブラリをコピーすることなく実行できる利点があります。

■ 32bit Linux で 1GB を越えるメモリを使用する Fortran プログラムを実行したい （Linux）

```
pgfortran -fastsse -Wl,-Bstatic test.f （あるいは、-Bstatic でもよい）
```

- **32bit Linux OS** では、一般に、使用可能なメモリ空間の上限は 2GB ですが、実際のデフォルトでは 1GB までのユーザ・メモリ空間を利用できる実行モジュールしか作成できません。しかし、コンパイル・オプション "-Wl,-Bstatic" を用いて静的ライブラリを用いる実行モジュールを作成した場合、1 GB を超えるプログラムの実行モジュールを作成することが可能です。本オプションは、コンパイラが暗黙に用いる動的ライブラリを用いず、強制的に全て静的リンクします。いわゆる、1GB の壁は、動的ライブラリをメモリマップ上に置くポイントが、1GB ポインタであることによる問題です。動的ライブラリを用いない静的リンクにしますと、この制約がなくなります。
  なお、1GB を越えるメモリ量を扱う Fortran プログラムを動かすためには、マシン自体にも十分なメモリ量（物理メモリ＋swap メモリで 1GB 以上）を確保することも必要です（プログラムが必要とするメモリ量よりも、物理メモリ＋swap メモリ量が少ないと正常に動作しません）。

■ リンケージ・マップとオブジェクト間のクロスリファレンスを出力する

```
pgfortran -fastsse -m -Wl,'-cref' test.f (Linux)
pgfortran -fastsse -m test.f (Windows)
```

- **-m** オプションを付加することにより、ld コマンドによるリンケージ・マップが出力され、**-Wl,'-cref'** オプションにより、オブジェクト間のクロスリファレンスが標準出力に出力されます。。Windows の場合は、当該マップファイルが、*****.map と言うファイル名で保存されます。

■ Windows におけるスタックサイズの調整

```
pgfortran -fastsse -Wl,-stack:N test.f
(PGI 7.0 以降)
pgfortran -fastsse -stack={ (reserved bytes)[,(committed bytes)] }{, [no]check } test.f
```

- Windows x64 上では、以下のコンパイル・オプションでスタックサイズを指定します。N は、必要とする任意のスタックサイズ値を指定します。
  -Wl,-stack:N
- PGI 7.0 以降では、Windows のみに有効な以下の -stack オプションが新設されました。

  -stack={ (reserved bytes)[,(committed bytes)] }{, [no]check }

  -stack を指定しない場合のデフォルトは、以下のとおりです。

  Win32 ： -stack:2097152,2097152、2MB がデフォルトで設定されます。
  Win64 ： デフォルトの設定はありません。

  例： pgfortran -stack=524288,262144,nocheck myprog.f

  プログラムの実行において、全体でトータル 524,288 stack bytes (512KB)の領域を予約し、各ルーチンのために OS が割り当てるスタックエリアとして 262,144 stack bytes (256KB)を指定します。また、ルーチンに入る際のスタックの初期化を行わないように nocheck 引数をコンパイラに指示すると言う指定の方法です。

  reserved bytes ：プログラムで使用するトータル・スタックサイズ値を指定
  committed bytes ：各ルーチンのために OS が割り当てるスタックエリアのサイズを指定
  デフォルトは、4096 バイト。
  [no]check ：ルーチンに入る際にスタックの初期化を行うコードを生成するか、
  行わないコードの生成かの指示を行う。”check”がデフォルトです。

  スタック初期化コードは、「ルーチンのスタックサイズ」が committed bytes を超えたときに必要とされます。指定した committed bytes が reserved bytes に等しいか、あるいは、各ルーチンで必要とされるスタックバイト数に等しいとき、stack=nocheck オプションを指定して、スタック初期化コードの生成を抑止することができます。こうすることによって、コンパイラは十分なコミットスタックスペースを指定しているものと理解し、プログラムはそれ自身のスタックサイズの管理を必要とされません。

  （注意）
  -stack=(reserved bytes),(committed bytes) はリンク時のオプションです。-stack=[no]check は、コンパイル時のオプションです。

■ ライブラリ、リンケージに関する補足情報

PGI コンパイラに限らず、一般的なコンパイラでは、アセンブル機能とリンク機能は、GNU のユーティリティを使用します（Linux の場合）。一般に、プログラムのコンパイルから実行モジュールの生成までは、以下の内部的な手続きが行われています。PGI コンパイラは以下の手続きを自動的に行っています。以下の手続きの内容と各コマンドのオプションを知るための方法は、こちらで説明しています。

① コンパイル&コード生成 (pgi のコンパイラコマンド)
② アセンブラによるオブジェクト生成 (GNU as コマンド)
③ リンカによるライブラリ等のリンクを行って実行モジュールの生成（GNU ld コマンド）

上記の手続きの中で、①, ② に関してはユーザが直接操作することができませんが、③ のリンケージの処理に関しては、ユーザ独自で GNU の ld コマンドのオプションを操作して、リンク時の問題等に対応することが可能です。「リンク時における問題」の多くは、必要とするライブラリが見つからない問題、同じ関数・サブプログラムを別のライブラリ・ファイルからリンクしたい等の対処が多いため、リンケージに関する基本的な知識を要します。ここでは、リンク時に必要となるシステムライブラリとリンケージに関する基本的な技術情報を提供します。

● ランタイム・ライブラリに関する基礎知識

まず始めに、実行モジュールにリンクされる、ライブラリに関する基礎的な事項について説明します。ユーザプログラム(オブジェクト）にリンクされるライブラリには、大きく以下の二つのライブラリ形式があります。

・スタティック（静的）ライブラリ
・シェアード（共有）ライブラリ

二つ目の共有ライブラリの範疇には、一般にダイナミックライブラリと言う形式のものもありますが、この共有ライブラリの中の特殊な例です。スタティックライブラリは、リンク時に実行モジュールの中に組み込まれます。モジュールサイズは大きくなりますが、別のシステム上で動作させたい場合は、全て必要なライブラリが組み込まれているため、そのままの形で動作します。一般に、ファイル名形式として、**lib***.a** と言う **.a** の名称がつけられております。スタティックライブラリを作成するためには、一般に、ar コマンドでオブジェクト(*.o)をまとめ、 ranlib コマンドでリンクを高速化する先頭子をつけ、strip コマンドでシンボルテーブルを削除します。
一方、シェアード（共有）ライブラリは、プログラムの実行時にシステム上に備えている、そのライブラリをロードして使用するためのものであり、システムライブラリのバージョンが異なったり、システム上に存在しない場合は、実行モジュールが動作しない場合があります。一般に、ファイル名形式として、**lib***.so** と言う **.so** の名称がつけられております。シェアードライブラリの一つである、ダイナミックライブラリとは、実行時に **load/unload** することが可能な「位置独立なライブラリ形式」のものを言います。これは、リンク時のオプションに **-fPIC** を指定して作成されております。
一般の実行モジュールは、ほとんどが「シェアード（共有）ライブラリ」をリンクするとの前提で生成され、実行時に必要とされるシステム共有ライブラリと PGI の場合は、PGI 用の共有ライブラリがロードされる形式になっています。このデフォルトのリンク方式を使用する場合には、「リンク時における問題」はほとんど発生しませんが、明示的にスタティック(静的)ライブラリをリンクする際に未解決なオブジェクト・ルーチンが存在する場合や作為的にあるライブラリを組み込みたい等の操作をした場合、リンク時の問題が発生します。

● リンク時におけるライブラリの組込み順序

デフォルトでは、リンカはまず、シェアード（共有）ライブラリを検索し、必要となるルーチンを組み込もうとします。次に、スタティック(静的)なライブラリの中から探し出し、実行モジュールに組み込みます。従って、一般のコンパイル(リンク)オプションでは、ある配慮を行わなければ、常に、シェアード（共有）ライブラリが組み込まれると言うことになります。この点を理解しておくが、「リンクする順序」と共に重要なポイントとなります。
ちなみに、実行モジュールの中でどのような dynamic shared library が使用されているかは、以下のコマンド (ldd) で確かめることができます。以下の例では、PGI 関連の libpgc.so だけでなく、システムに備えられたシェアード（共有）ライブラリもリンクされるべき実行モジュールであることが分かります。特に libpgc.so ファイルは、PGI をインストールしたマシン上でしか存在しません。例えば、他のシステムでこのような実行モジュールを動作させるためには、この libpgc.so ファイルを予めコピーしておくことが必要となります。

```
$ ldd a.out
      libc.so.6 => /lib/i686/libc.so.6 (0x4002d000)
      libpgc.so => /usr/pgi/linux86/5.2/lib/libpgc.so (0x4015d000)
      libm.so.6 => /lib/i686/libm.so.6 (0x40172000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

一方、スタティックなライブラリをリンクして作成された実行モジュールでは、ldd コマンドでどのように表示されるでしょうか？　以下のように表示されます。

```
$ ldd a.out
      not a dynamic executable
```

リンカが暗黙で行うシステム上のライブラリ検索順序は上記に述べたとおりですので、これを作為的に変更することによって、所望とするライブラリ・ルーチンを先に組み込むことができます。一般に、リンカ（コマンド）のオプションの指定で、ライブラリの検索順位を明示的に指定することができます。とは言っても、この操作は簡単で、単にリンクするライブラリ名をコマンド列の中で先に指定するだけの操作となります。このことを理解すれば、後は試行錯誤で、リンクエラーが起きないように所望とするライブラリを組み込むことができます。複数のライブラリ間で呼び出し/参照の順序の依存関係がある場合、ライブラリ名をコマンド列で指定する順序は、必ず以下のようにしなければなりません（順序はセンシティブです）。

例えば、　ユーザのスタティックライブラリ **libA.a** の中の関数が、**libB.a** 中の関数を必要とする場合の例を記します。

libA.a ---> **libB.a**　（共有ライブラリの場合は、**libA.so** ---> **libB.so**）

リンカのコマンド列では、以下のような順番でリンク検索順序を指定します。ライブラリ(*.a) 内の参照の解決は、コマンド行に記されたものを順番に、1 回限りの検索を行います。特に大事なことは、コマンド行上で先行するアーカイブ・ライブラリを、後続の入力ファイル（ライブラリを含む）内の新しい参照の解決に使用することはありません。このリンケージにおける暗黙の決まりごとが、コマンド行のライブラリ名指定の順序に関係します。以下の例では、最初に test.o の未解決の参照をコマンド行の次以降に記された *.a を順番に検索して解決すれば問題なく、test.o を実行モジュールにできますが、これで全て OK とは言えません。次のコマンド行に出てくる libA.a 内部の未解決参照があるかどうか、リンカはチェックします。リンカは、コマンド行の libA.a の右側に記述されているライブラリのみしか、未解決参照シンボルを検索しません。もし、その前にアーカイブ名記

述されていたとしても、例えば、この場合 libB.a がコマンド行の前に記述されていても、「検索」自体を行いません。常に、コマンド行を 1 回限り、常に右にあるものを検索するのみです。

pgcc test.o -Bstatic -L{directory} **-lA -lB**
（ -lB -lA の順番では libB.a の関数が未解決になります）

あるいは、明示的にライブラリ名を指定します。

pgcc test.o-Bstatic **libA.a libB.a**

共有ライブラリ（libA.so、libB.so）の場合は、

pgcc test.o -L{directory} **-lA -lB** あるいは、pgCC test.C **libA.so libB.so**

● 静的にリンクする場合のコンパイル・オプション

PGI コンパイラでは、リンカに静的リンケージを行うことを指示するために、**-Bstatic** と言うオプションがあります。基本的には、この指定によってリンカは**静的ライブラリを先に検索しようとします**。但し、-Bstatic オプションを指定して実行モジュールを作成する際に、リンカのレベルで、「未解決な参照ルーチン」が存在し、実行モジュールが作成できないエラーが生じる場合があります。これは、PGI コンパイラ（ld リンカ）のデフォルトのライブラリの検索順序が適当でない場合に生じます。

-Bstatic とは逆に、明示的に dynamic shared library の扱いでリンクするように指示するためのオプションも存在します。これは、**-Bdynamic** オプションを指定することで実現しますが、リンカに明示的にライブラリのリンク方法を指定するには、例えば、コマンド列に以下のような記述を行います。**-Bdynamic** 以降に記述された「ライブラリ」がダイナミックに、**-Bstatic** 以降に記述された「ライブラリ」がスタティックにリンクすることを指示するものです。

```
$ pgf90 test.f -Bdynamic -lc -lm -lgcc -lc -lgcc -Bstatic -lpgftnrtl -lnspgc –lpgc
```

なお、弊社ホームページの http://www.softek.co.jp/SPG/Pgi/TIPS/link.html に、リンク操作の具体例を説明した「リンク操作の裏技」と言う記事がありますので、詳細は、こちらをご参照下さい。

2.6 プログラムを開発・検証する際に便利なオプション

PGI の F77, F2003, HPF, C, C++ のコンパイラを使用してプログラムを開発・検証する際に、有用なオプションを以下に示します。いずれもプログラムのデバッグを行うときに有効なオプションです。

■ 実行時に配列境界のチェック （Bounds check） を行う

```
pgfortran -fastsse -Mbounds test.f

あるいは
pgfortran -fastsse -C test.f
```

- **-Mbounds** あるいは **–C** オプションは、実行時に配列の境界チェックを有効にするようにオブジェクトを生成するためのものです。プログラムの配列境界の妥当性を確認するデバッグ時に非常に有効です。例えば、配列境界外のアクセスを行った場合、以下のような形式で出力されます。

  【Fortran の場合】
  PGFTN-F-Subscript out of range for array a (add.f: 211)
  subscript=3, lower bound=1, upper bound=2, dimension=2

  【C の場合】
  PGC-F-Subscript out of range for array ST2 (test.c: 385)
  subscript=2, upper bound=1, dimension=1

■ 浮動小数点演算での例外処理としてトラップを掛けプログラムを終了する

```
pgfortran -fastsse -Ktrap=fp test.f
```

- **-Ktrap** オプションを使うことで、浮動小数点演算の例外処理方式の指定が可能です。default は例外が起きても実行を続行しますが、このトラップ処理を入れることで、例外が起きた時点でプログラムを終了させることが可能です。
- **-Ktrap=<flags>** 形式で、適用する例外処理のフラグを指定することで、細かな操作が可能となります。以下のフラグがありますが、これはプロセッサの例外処理マスクに相当します。

  inv : invalid operation
  denorm : denormalized operand
  divz : divide-by-zero
  ovf : overflow
  unf : underflow
  inexact : precision
  （現在 inexact は、PGI ではサポートしていない、多数の exception が発生する可能性あり）
  fp : inv,divz,ovf と等価
  none : 全てのトラップを抑止します（PGI 6.2 以降）

■ 浮動小数点演算の方式を IEEE 754 Standard に厳密に準拠するコードを生成

```
pgfortran -fastsse -Kieee test.f
```

- **-Kieee** オプションを使うことで、浮動小数点演算の処理方式を厳密に IEEE 754 に準拠した計算方式での実行モジュールの作成を行います。この場合、コンパイラが行う最適化のいくつかが実施されません。このオプションの別の側面は、プログラムの結果精度に関する誤差感度の評価に使用できます。
- **-Kieee** オプションをリンクステップで指定した場合、システムによってはより精度の高い数学ライブラリがリンクされる場合があります。
- **IEEE 754 Standard** に準拠した演算とは、例えば、演算に伴う右辺、左辺のメモリ・ロード/ストアを IEEE に準拠した形で厳密に行い、最適化等による copy propagetion （変化しない変数、定数の演算上の置き換え）等を行わない計算方式などを言います。また、除算 a/b では、コンパイル時に a*(1/b) といった近似演算も無効とされます。さらに、内部組み込み関数への浮動小数点の引数の引渡しでは、丸めが行われます。

■ x87 レジスタ（スタック）のビット長の制御 （Intel px/p5/p6/piii CPU のみ用限定）

```
pgfortran -fastsse -pc 64 test.f
```

- -pc オプションは、浮動小数点演算に x87 演算スタックを使用する旧プロセッサ (Pentium III 等) において有効です。具体的には、CPU target が px/p5/p6/piii のみの場合の add, subtract, multiply, divide, square root の演算に影響します。Pentim 4 以上のプロセッサでは、スカラ浮動小数点演算に、デフォルトでは x87 スタックを使用しません。この -pc オプションを使用するときには、強制的に -tp p6 等でクロスコンパイルを行い、実行することが必要です
- **-pc** オプションを使うことで、x87 アーキテクチャ上のレジスタビット長の使用精度の制御が可能です。x87 のレジスタは 80bit 長の長さを有します。PGI のデフォルトは、80bit そのままのビット長でレジスタ IN/OUT 処理を行います。このオプションを指定しない場合のデフォルトは PGI では、80bit ビット長を使用します。
- **-pc {80|64|32}** 形式で、ビット長を指定することで、そのビット数でのレジスタ IN/OUT 処理を行うように指定できます。-pc 64 では、64bit を使用した Standard IEEE 倍精度の処理を明示的に指定できます。-pc 32 の場合も同様です。
- このオプションの別の側面は、プログラムの結果精度に関する誤差感度の評価に使用できます。ビット長を変更することで、計算結果にどの程度の差異が相対的に生じるか評価することで、プログラム実装の精度安定性の見識を得ることが可能です。
- -pc { 64 | 32 } を指定する場合、main program のコンパイルは、必ず、このオプションを指定してください。

■ 浮動小数点演算の数値結果に影響を与える最適化の有効/無効化

```
pgfortran -fastsse -Mvect=assoc  test.f
```

- **-Mvect=assoc** オプションは、より高度な最適化を行うためにループ内の処理を変換（ループ分割、ループのネストの変更、演算の結合等）を許可するオプションで、浮動小数点演算の丸め誤差によって引き起こる計算結果の違いが起きても良いことを指示するものです。ベクトル化に

おける最適化手法では、数式的(数学的）には正しい組み換え(変換）を行いますが、計算機のような有限桁で計算を行う場合は、計算順序の変更により仮数部の桁落ち、あるいは、丸め誤差等が発生する副作用を伴います。これを有効化する場合に assoc フラグを使用します。

- **-Mvect=noassoc** は、上記の最適化手法を disable にするオプションです。これは、デフォルトです。 このオプションの別の側面は、プログラムの結果精度に関する誤差感度の評価に使用できます。

```
pgfortran -Mlre[=array|assoc|noassoc] test.f
```

- **-Mlre** オプションは、loop-carried redundant removal と言う最適化手法です。loop-carried とは、ループ iteration 処理内と言う意味で、この中で共通数式(冗長数式)をまとめる、あるいは冗長な配列の参照を削除する最適化となります。浮動小数点演算の丸め誤差によって引き起こる計算結果の違いが生じる副作用があります。
  - ◆ array： 個々の配列要素の参照を冗長性削減の対象として扱う。デフォルトは、2 以上のオペランドを含む冗長式のみが対象となる。
  - ◆ assoc：冗長性削減の対象を増やすことができる、演算式の再結合を許す最適化。結果の差異が生じる可能性がある。
  - ◆ noassoc：上記を許さない最適化

- **-Mnolre** は、上記の最適化手法を disable にするオプションです。これは、デフォルトです。但し、-fastsse 複合オプションを使用した場合、-Mlre は有効となっています。

■ 緩い精度の内部組込関数、演算

```
-M[no]fprelaxed=[div,order,recip,rsqrt,sqrt]
```

内部組込み関数 （div/sqrt/rsqrt) の計算において、緩い精度で行うことをコンパイラに指示します。性能は向上しますが、計算精度は劣ります。 (PGI 6.1 以降） デフォルトは、-Mnofprelaxed。PGI 6.2 以降、細かな制御を行うためのサブオプションを導入し、サブオプションは以下のとおりです。

div ： 緩い精度で除算処理を行う。
recip ： 逆数近似を緩い精度で行う（PGI 9.0 以降)。
noorder ：a*b+a*c を a*(b+c)と変換する方式等、演算の順序の変更をしない。
order ：a*b+a*c を a*(b+c)と変換する方式も含め、演算の順序の変更を許す。
rsqrt ： 緩い精度で sqrt の逆数近似（1/sqrt）の処理を行う
sqrt ： 緩い精度で sqrt の処理を行う

なお、サブオプションを付加しない場合（-Mfprelaxed のみ）は、そのターゲットプロセッサに応じて、顕著な性能向上が行える処理に、緩い精度での処理を行うかを選択し適用されます。

```
-M[no]fpapprox[=div|sqrt|rsqrt]
```

-M[no]fpapprox[=div|sqrt|rsqrt]は、特定の浮動小数点演算において、低精度近似を使用して実行します。(PGI 7.1 以降）**このオプションは結果の差異が生じる可能性がありますので、十分注意して使用してください。**

div ： 浮動小数点除算近似
sqrt ： 浮動小数点平方根近似

rsqrt　　：浮動小数点逆数平方根近似

デフォルトでは、-Mfpapprox は使用されません。もし、サブオプションを指定しない-Mfpapprox のみの場合は、上記の全てのサブオプションが指定されたものとして扱います。

■ ファイル I/O における内部エンディアン方式の変換

```
pgfortran -fastsse -byteswapio test.f
```

- アンフォーマット Fortran データファイルの入出力時にビッグエンディアン (big- Endian) からリトルエンディアン (little-endian) にあるいはその逆に、バイトをスワップ します。生成された実行モジュールは、自動的に read/write 処理中において、このエンディアン変換を行います。
- RISC/UNIX システム、あるいはハイエンドのシステムで採用されているビッグエンディアン形式のデータファイルから x86 あるいは AMD64 で採用されているリトルエンディアン形式のデータファイルにプログラム実行時に自動変換する際に有効です。
- データ変換時の仮定として、アンフォーマット・シーケンシャルファイル並びにダイレクト・アクセスファイルのレイアウトが両システム上で同一であることと、内部表現形式が IEEE 形式であることを仮定しています。

■ プログラム・ルーチン間のコールグラフの出力（PGI 6.1 新機能、Linux のみ）

● コールグラフ対応実行モジュールの作成

```
pgfortran -fastsse -Mipa=cg -o a.out test.f
```

● コールグラフの出力(pgicg コマンドを使用)

```
$ pgicg -graph a.out （実行順番のコール・フロー）

laplce_
. opnfil_ (opnfil.f:4) [74]
. . pgf90io_src_info [15 18 23 26]
. . pgf90io_open [15 23]
. . pgf90io_ldw_init [18 26]
. . pgf90io_ldw [18 26]
. . pgf90io_ldw_end [18 26]
. pgf90io_src_info [76 83]
. pgf90io_ldw_init [76 83]
. pgf90io_ldw [76 83]
. pgf90io_ldw_end [76 83]
. input_ (input.f:11) [80]
. . pgf90io_src_info [18 20 25 27 37 38 39 45]
. . pgf90io_ldr_init [18 25]
. . pgf90io_ldr [18 25]
. . pgf90io_ldr_end [18 25]
. . pgf90io_ldw_init [20 27]
. . pgf90io_ldw [20 27]
. . pgf90io_ldw_end [20 27]
. . pgf90io_fmtw_init [37 38 39 45]
. . pgf90io_fmt_write [37 38 39 45]
. . pgf90io_fmtw_end [37 38 39 45]
. header_ (header.f:8) [87]
. . pgf90io_src_info [12 13 14 15]
. . pgf90io_fmtw_init [12 13 14 15]
. . pgf90io_fmtw_end [12 13 14 15]
```

```
. . pgf90io_fmt_write [13 14 15]
. init_ (init.f:16) [100]
. timer_ (timer.c:8) [104 113]
. . gettimeofday [13]
. solve_ (solve.f:14) [106]
. error_ (error.f:8) [109]
. swap_ (swap.f:8) [110]
. verify_ (verify.f:18) [117]
. epilog_ (epilog.f:13) [121]
. . pgf90io_src_info [33 35 36 37 39 41 45 47 50 51]
. . pgf90io_fmtw_init [33 35 36 37 39 41 45 47 50 51]
. . pgf90io_fmtw_end [33 35 36 37 39 41 45 47 50 51]
. . pgf90io_fmt_write [35 36 37 41 50 51]
. clsfil_ (clsfil.f:3) [124]
. . pgf90io_src_info [11 12]
. . pgf90io_close [11 12]

$ pgicg -callers a.out

clsfil_ (clsfil.f:3) called by laplce_
epilog_ (epilog.f:13) called by laplce_
error_ (error.f:8)   called by laplce_
gettimeofday           called by timer_ (timer.c:8)
header_ (header.f:8) called by laplce_
init_ (init.f:16)    called by laplce_
input_ (input.f:11) called by laplce_
laplce_
opnfil_ (opnfil.f:4) called by laplce_
pgf90io_close          called by clsfil_ (clsfil.f:3)

pgf90io_ldr            called by input_ (input.f:11)
pgf90io_ldr_end        called by input_ (input.f:11)
pgf90io_ldr_init       called by input_ (input.f:11)
pgf90io_ldw            called by input_ (input.f:11) laplce_ opnfil_ (opnfil.f:4)
pgf90io_ldw_end        called by input_ (input.f:11) laplce_ opnfil_ (opnfil.f:4)
pgf90io_ldw_init       called by input_ (input.f:11) laplce_ opnfil_ (opnfil.f:4)
pgf90io_open           called by opnfil_ (opnfil.f:4)
solve_ (solve.f:14) called by laplce_
swap_ (swap.f:8)       called by laplce_
timer_ (timer.c:8)   called by laplce_
verify_ (verify.f:18) called by laplce_
```

● pgicg utility のその他のオプションのヘルプ
pgicg –help

- プログラム内の各ルーチンが実行時にどのような順番でコールされるかと言う「コールグラフ」の出力が可能です。また、ルーチン間の呼び出し依存関係を出力します。(Linux 版のみ)

■ ループ内の演算密度（Intensity） の表示 (Linux 版/PGI 7.2 以降)

```
pgfortran -fastsse -Minfo=intensity test.f
```

- Minfo=intensity ーループ内の「演算密度」 (Computational Intensity) を表示します。各ループレベルまでの情報を表示しようとしますが、デフォルトではコンパイル時の静的解析できる

最内側ループの情報が表示されます。その外側のループレベルの情報は、一度実行した後に、正確な数字が表示できます。演算密度とは、一般にループ内の演算数とメモリのロード・ストア数との比率を表し、演算とメモリ参照のバランスを見るための指標です。このような情報はパフォーマンス・チューニングにおいて特に重視されます。

- ループ内の演算が浮動小数点演算である場合、演算密度は、浮動小数点演算総数を浮動小数点データのメモリ・ロードとストアの総和で割った比率として定義します。
- ループ内の演算が整数演算である場合、演算密度は、整数演算総数を整数データのメモリ・ロードとストアの総和で割った比率として定義します。

以下は、Linux 上で実施した例です。

```
 (PGF90 (Version      10.1)          01/29/2010   11:33:44       page 6
(  292)    do loop=1,nn
(  293)       gosa= 0.0
(  294)       do k=2,kmax-1
(  295)          do j=2,jmax-1
(  296)             do i=2,imax-1
(  297)                s0=a(I,J,K,1)*p(I+1,J,K) &amp;
(  298)                     +a(I,J,K,2)*p(I,J+1,K) &amp;
(  299)                     +a(I,J,K,3)*p(I,J,K+1) &amp;
(  300)                     +b(I,J,K,1)*(p(I+1,J+1,K)-p(I+1,J-1,K) &amp;
(  301)                                 -p(I-1,J+1,K)+p(I-1,J-1,K)) &amp;
(  302)                     +b(I,J,K,2)*(p(I,J+1,K+1)-p(I,J-1,K+1) &amp;
(  303)                                 -p(I,J+1,K-1)+p(I,J-1,K-1)) &amp;
(  304)                     +b(I,J,K,3)*(p(I+1,J,K+1)-p(I-1,J,K+1) &amp;
(  305)                                 -p(I+1,J,K-1)+p(I-1,J,K-1)) &amp;
(  306)                     +c(I,J,K,1)*p(I-1,J,K) &amp;
(  307)                     +c(I,J,K,2)*p(I,J-1,K) &amp;
(  308)                     +c(I,J,K,3)*p(I,J,K-1)+wrk1(I,J,K)
(  309)                ss=(s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
(  310)                GOSA=GOSA+SS*SS
(  311)                wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
(  312)             enddo
(  313)          enddo
(  314)       enddo----------------------

~/Himeno> pgf90 -fastsse -Minfo=intensity himenoBMTxp.f90
jacobi:
jacobi:
     292, Intensity = [symbolic], and not printable, try the -Mpfi -Mpfo options
     294, Intensity = [symbolic], and not printable, try the -Mpfi -Mpfo options
     295, Intensity = [symbolic], and not printable, try the -Mpfi -Mpfo options
     296, Intensity = 1.06
```

コンパイル時に情報が特定できないループ情報（外側ループの情報）は、1 度実行する。
まず、-Mpfi オプションを付してコンパイル&リンクします。

```
~/Himeno>   pgf90 -fast -Minfo=intensity himenoBMTxp.f90<b> -Mpfi</b>
~/Himeno>   ./a.out （実行）
```

実行後、pgfi.out と言う統計情報ファイルができます。これを元に再度、フィードバックコンパイル（-Mpfo）を行うと、以下のように外側ループの Intensity が出力されます。

```
~/Himeno>   pgf90 -fast -Minfo=intensity himenoBMTxp.f90<b> -Mpfo</b>
jacobi:
```

```
      292, Intensity = 12.37
      294, Intensity = 1.06
      295, Intensity = 1.06
      296, Intensity = 1.06
```

2.7 異なる CPU Target のモジュールを作成するオプション（クロスコンパイル）

PGI の F77, F2003, C, C++ のコンパイラを使用する際に、異なる CPU ターゲット用のクロスコンパイルを行う際のオプションを説明します。以下は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。

現在の x86 系のプロセッサ（x64 とも言う）は、ハードウェア的に全て 64 ビット対応となっています。しかし、その上で動作する OS の種別が 32bit-OS か 64bit-OS かにより、そのシステムが 32bit か 64bit かと言う区別をしております。32bit-OS 上では、そこで生成される用実行バイナリは、32 ビット用のバイナリのみとなります。一方、64bit-OS 上では、デフォルトでは、64 ビット実行バイナリが生成されますが、クロスコンパイル機能で、32 ビット用の実行バイナリも作成することができます。例えば、64bit-OS 上で、-tp nehalem-32 と指定すれば、32 ビット用 Nehalem CPU に最適化された実行バイナリが作成されます。クロスコンパイル機能は、こうしたプロセッサやビットモードが異なるシステム上で動作させるための最適バイナリを作成する機能です。さらに、PGI 製品は、マイクロ・アーキテクチャが異なる Intel® 64 と AMD64 の各プロセッサ用に最適化したコード･ブロックや GPU 用のコードを全て一つの実行バイナリの中に生成できる「PGI Unified Binary™」と言うユニークな特長を有しています。

■ クロスコンパイル機能オプション

```
$ pgfortran -tp sandybridge-64 -fastsse xx.f
  （Intel SandyBridge 64bit プロセッサ用に最適化してバイナリを作成する例）

 PGI 14.1 以降は、以下の指定方法を推奨
 (target 名と -m32(32bit binary) あるいは -m64 (64bit bainary)の二つを指定する)

      -tp={target 名} -m32|-m64

  $ pgfortran -tp haswel -m64 -fastsse   xx.f90 (Haswell の 64bit 用バイナリ作成）
  $ pgfortran -tp haswel -m32 -fastsse   xx.f 90(Haswell の 32bit 用バイナリ作成）

 Intel sandybridge と AMD Shaghai プロセッサ用の PGI Unified Binary を作成
  $ pgcc -tp sandybridge-64,shaghai-64 -fastsse   xx.c
```

- **-tp** オプションは、-tp にそれぞれの CPU タイプに応じた target 名を記述することでクロスコンパイル機能（その CPU に特化した最適化を行ったモジュールを作成する）を使用することができます。この CPU ターゲット名は、以下の表に示します。**-tp** の後に各 **target** を指定し、そのアーキテクチャに沿ったコードを生成します。ターゲットのデフォルトは、コンパイルを実行する際のシステムの「プロセッサ・タイプ」にターゲットが設定されます。そのシステム（プロセッサ）がサポートする全てのマシン・インストラクションを使用してコードが生成されます。

target	プロセッサの説明	備考
k7	AMD Athlon Processor	
k8-32	AMD Opteron/Athlon64 32-bit mode	PGI 5.0 以降
k8-64	AMD Opteron/Athlon64 AMD64 64-bit mode	PGI 5.0 以降

k8-64e	AMD Opteron/Turion/Athlon64 AMD64 64-bit mode Revision E/F 以降(SSE3 機構あり)	PGI 6.1 以降
barcelona-32	AMD Opteron/Quad-Core 32-bit mode	PGI 7.0-3 以降
barcelona-64	AMD Opteron/Quad-Core 64-bit mode	PGI 7.0-3 以降
shanghai-32	AMD Opteron/Quad-Core 32-bit mode	PGI 8.0-1 以降
shanghai-64	AMD Opteron/Quad-Core 64-bit mode	PGI 8.0-1 以降
istanbul-32	AMD Opteron/six-Core 32-bit mode	PGI 9.0-1 以降
istanbul-64	AMD Opteron/six-Core 64-bit mode	PGI 9.0-1 以降
bulldozer-32	AMD Opteron/siz-Core Bulldozer 32-bit mode	PGI 11.9 以降
bulldozer-64	AMD Opteron/siz-Core Bulldozer 64-bit mode	PGI 11.9 以降
piledriver-32	AMD Piledriver : 32-bit mode	PGI 13.1 以降
piledriver-64	AMD Piledriver : 64-bit mode	PGI 13.1 以降
haswell-32	Intel Core i7/i5/i3 (Haswell) 32-bit mode	PGI 14.1 以降
haswell-64	Intel Core i7/i5/i3 (Haswell) 64-bit mode	PGI 14.1 以降
ivybridge-32	Intel Core i7/i5/i3 (Ivy Bridge) 32-bit mode	PGI 14.1 以降
ivybridge-64	Intel Core i7/i5/i3 (Ivy Bridge) 64-bit mode	PGI 14.1 以降
sandybridge-32	Intel Core i7/i5/i3 (Sandy Bridge) 32-bit mode	PGI 11.6 以降
sandybridge-64	Intel Core i7/i5/i3 (Sandy Bridge) 64-bit mode	PGI 11.6 以降
nehalem-32	Intel Core i7/i5/i3 (Nehalem) 32-bit mode	PGI 9.0-1 以降
nehalem-64	Intel Core i7/i5/i3 (Nehalem) 64-bit mode	PGI 9.0-1 以降
core2-64	Intel Core 2 Intel 64 64-bit mode	PGI 6.2 以降
core2-32	Intel Core 2 32-bit mode	PGI 6.2 以降
penryn-32	Intel Penryn(Quad Core) 32-bit mode	PGI 7.2 以降
penryn-64	Intel Penryn(Quad Core) 64-bit mode	PGI 7.2 以降
p7-64	Intel Xeon/Pentium 4 Intel64 64-bit mode	PGI 5.2 以降
p7	Intel P7 Xeon/Pentium 4 32-bit mode	
p6	Intel P6 Pentium (Pentium Pro, II, III)	
p5	Intel Pentium	
px	Intel generic Pentium	
x64	-tp p7-64,k8-64 と同等	PGI 6.1 以降

特に、特に、AMD 社の AMD64 （Opetron 系）とインテル社の Intel(R) 64 (Pentium, Xeon 系）は、互換性を有するものですが、各プロセッサのマイクロ・アーキテクチャの違いにより、その最適化方法が異なります。例えば、AMD64 用に生成されたコードは、Intel(R)64 マシン上では十分な性能を発揮できません。また、その逆で、Intel(R)64 用に生成されたコードも AMD64 マシン上では性能が大幅に落ちる場合があります。PGI コンパイラは、AMD64 並びに、Intel(R)64 のプロセッサのそれぞれに最適化し、一つの実行バイナリファイルとして生成できる **PGI Unified Binary** 機能を有しています。これは、他社製コンパイラにはない機能となります。PGI コンパイラは、最新のテクノロジーを有するインテル社のプロセッサも、AMD 社のプロセッサにも十分な最適化を行える商用コンパイラです。

■ PGI 7.0 以降の新機能（複数のターゲットに対する最適化 Unified Binary）

Multiple PGI Unified Binary Targets － -tp スイッチは、コンマ（,）で区切られた複数のプロセッサ Target リストを指定する形態をサポートしました。二つ以上の **64-bit ターゲット**に対

しての PGI Unified Binary としての最適化（例: -tp sandybridge-64,shaghai-64,core2-64 と指定すると三つのターゲットに最適化されたコードを生成します）を指示することができます。

マルチターゲット指定の例

```
$ pgcc -tp sandybridge-64,shaghai-64,core2-64 -fastsse xx.c
```

また、PGI Unified Binary 指示用のディレクティブ、プラグマが提供されております。これらは、コンパイラに対して、一つ以上のターゲット用の Unified Binary 最適化コードを関数、サブルーチン、ファイル全体に対して生成するように指示するものです。ディレクティブを指示した場合は、特別のコマンドライン・オプションは必要ありません。
Fortran ディレクティブの書式は、以下のとおりです。

```
!pgi$[g|r| ] pgi tp [target]...
```

ここで、ディレクティブが有効となるスコーピングの範囲は、g(global)、r(routine)、あるいは空白で指示します。デフォルトは、r(routine)単位となります。例えば、

```
!pgi$g pgi tp sandybridge-64 piledriver-64 nehalem-64
```

は、全体のソースファイルに対して（g(global)でスコーピングを指示している）、sandybridge-64、piledriver-64、nehalem-64 用の最適化を施した Unified Binary を生成すると言う意味となります。
C/C++のプラグマの書式は、以下のとおりです。

```
#pragma routine tp sandybridge-64 piledriver-64 nehalem-64
```

これは、次の function/routine に対して、k8_64、p7_64、core2_64 用の Unified Binary を生成すると言う意味となります。

■ クロスコンパイルした実行モジュールに関する留意点

クロスコンパイル機能を利用して実行モジュールを作成する場面は、作成した実行モジュールを別のマシン(システム)で動作させるような場合となります。その場合の注意事項は、弊社ホームページ上の「別のマシンで実行モジュールを動かす」（http://www.softek.co.jp/SPG/Pgi/TIPS/another.html）と言う記事で説明しておりますが、一番簡単な方法として、必要とするライブラリを静的にリンク（-Bstatic オプション）する方法をお勧めします。但し、-mcmodel=medium を伴って静的ライブラリをリンクすることはできません。これは、コンパイラの制約ではなく、64 ビットプログラミングモデル上の制約となります。

```
pgfortran -tp nehalem,sandybridge –m64 -fastsse -Bstatic xx.f90
```

■ PGI がサポートするプロセッサとそのハードウェア最適化機能

以下の表は、PGI コンパイラがサポートしているプロセッサとそのハードウェア機構について纏めたものです。PGI コンパイラは、それぞれのプロセッサ用にハードウェア機構を十分に利用して最適化を行います。

Processors supported by PGI 2017									
Brand	CPU	<target>	Mem Addressing	Floating point HW					
				SSE1	SSE2	SSE3	SSSE3	SSE4	ABM SSE4a
AMD	Piledriver	Piledriver-64	64-bit	Yes	Yes	Yes	No	Yes	Yes
AMD	Bulldozer	bulldozer-64	64-bit	Yes	Yes	Yes	No	Yes	Yes
AMD	Isutanbl	istanbul-64	64-bit	Yes	Yes	Yes	No	Yes	Yes
AMD	Shanghai	shanghai-64	64-bit	Yes	Yes	Yes	No	No	Yes
AMD	Balcelona	barcelona-64	64-bit	Yes	Yes	Yes	No	No	Yes
AMD	Opteron/Athlon64	k8-64	64-bit	Yes	Yes	Yes	No	No	No
AMD	Opteron Rev E/F	k8-64e	64-bit	Yes	Yes	Yes	No	No	No
AMD	Turion64	k8-64e	64-bit	Yes	Yes	Yes	No	No	No
Intel	Core i7/i5/i3 (Haswell)	haswell-64	64-bit	Yes	Yes	Yes	Yes	Yes	Yes
Intel	Core i7/i5/i3 (Ivy Bridge)	ivybridge-64	64-bit	Yes	Yes	Yes	Yes	Yes	Yes
Intel	Core i7/i5/i3 (Sandy Bridge)	sandybridge -64	64-bit	Yes	Yes	Yes	Yes	Yes	Yes
Intel	Core i7(nehalem)	nehalem-64	64-bit	Yes	Yes	Yes	Yes	Yes	Yes
Intel	Penryn	penryn-64	64-bit	Yes	Yes	Yes	Yes	Yes	No
Intel	Core 2	Core2-64	64-bit	Yes	Yes	Yes	Yes	Yes	No
Intel	P4/Xeon Intel 64	p7-64	64-bit	Yes	Yes	Yes	Yes	No	No

2.8 MPI プログラムを開発時に使用するオプション

本項では、PGI コンパイラとツールを利用して、MPI プログラムの開発を行うために使用されるコンパイル・オプション、並列デバッガ等の使用方法について説明します。MPI の使用に関しては、PGI User's Guide の 6 章に詳細に説明されておりますので、これも併せてご参照ください。

現在のシステムは、1 プロセッサに複数のマルチコアを搭載しているため、MPI のプログラム開発も 1 ノード上の複数の並列プロセスを使用して開発できるようになりました。このように、1 ノード内のローカルなプロセス環境で MPI 並列の開発ができることから、 PGI の Workstation/Server ライセンスにおいても、ローカルノード上での MPI 用のデバッガ、プロファイラを含めた開発環境を提供します。なお、ローカル並びにリモートノードも含めた MPI の開発環境は、 PGI CDK ライセンス製品で可能となります。

■ 各 PGI 製品の MPI プログラム開発環境

PGI Professional Network Floating 製品（Linux 64 ビット版）は、PGI 16.10 以降、Open MPI 1.10.x がデフォルトで実装されます。MPICH 3.2 と MVAPICH 2.1 のオプショナルライブラリもインストールすることができます。提供するオープンソース・ソフトウェアとしては、従来の PGI CDK 製品の構成と同じものとなります。

PGI Workstation / Server / Community / Professional Node-locked 製品（Linux 64 ビット版）は、PGI 16.1 以降、以前の MPICH version 3 に代えて Open MPI 1.10.x をバンドルしました。 PGI 14.1 以降 PGI 15.10 までは、MPICH version 3 over Ethernet をバンドルしていました(MPI-3 規格準拠ライブラリ)。PGI 7.1～PGI 13.10 までは、、MPICH-1 ライブラリをバンドルし、インストール時にライブラリ一式を実装していました。PGI 14.1 以降をお持ちのお客様は、MPICH v3 の並列デバッギング並びにプロファイリングの機能が使用できます。PGI 7.1 ～ PGI 13.10 のライセンスを有するお客様は、 MPICH1 プログラムの MPI 並列デバッギング並びにプロファイリングの機能が使用できます。その他の MPICH2 や Open MPI 等のソフトウェアは、ご自身で構築していただく必要があります。自身で構築した MPI ライブラリの並列デバッグ機能はありません。

PGI Cluster Development Kit (CDK) 製品（Linux 版）には、PGI CDK 16.1 以降、Open MPI 1.10.x がデフォルトで実装されます。MPICH 3.2 と MVAPICH 2.1 のオプショナルライブラリもインストールすることができます。PGI CDK 14.1 以降 PGI CDK 15.10 までは、MPICH3 over Ethernet がバンドル実装されました。その他、別途プリビルドされた、Open MPI 1.8.4 over Ethernet と MVAPICH2 2.0 over Infiniband が提供され、必要に応じてインストールすることができます。以前の PGI CDK 13.10 までは、 MPICH1、MPICH2、高速通信媒体 InfiniBand 用の MVAPICH-1.1 ライブラリがバンドルされており、これらの MPI ライブラリを実装して使用することができました。PGI CDK では、これらの MPI ライブラリが使用でき、さらに、MPI 並列デバッガとプロファイラを利用することができます。

PGI Workstation / Server / Community / Professional 製品（Windows 64 ビット版）は、 Microsoft(R) HPC Pack SDK で提供されている、MS-MPI ライブラリをコンパイル・オプションレベルで使用できるようになっております。PGI 7.1 以降の Windows 版では、業界で初めて、MS-MPI 対応の MPI 並列デバッガとプロファイラを提供しました。PGI コンパイラを使用することによって、ローカルなノード上で、MPI 開発環境を即座に構築することが可能です。PGI 2013 から、MS-MPI ライブラリが PGI ソフトウェアの中にバンドルされており、PGI をインストール時に MS-MPI ライブラリが同時に実装されます。

PGI Workstation / Community Edition（OS X 64 ビット版）は、PGI 14.1 以降、MPICH3 をバンドルしております。PGI 16.1 以降も引き続き MPICH3 をバンドルしております。

■ PGI Workstation/Server 製品にバンドルされた MPICH 環境のカスタマイズ

PGI Workstation/Server 製品における PGI コンパイラと共に実装された MPICH 環境は、デフォルトでは、インストールしたシステム上（ローカル上）でのみ MPI 実行ができるようになっております。一般に MPI プログラムの実行は、リモートノードを含めた分散ノード環境で並列実行を行いますので、MPICH 環境のカスタマイズが必要となります。その一例として、並列実行に参加するノード名を定義した、machines.LINUX と言うファイルを変更する必要があります。machines.LINUX ファイルの中に、MPI 実行に使用されるホスト名を定義すると、mpirun コマンドは、このファイルに定義されたホスト名を並列計算用のノードとして順番に使用します。machines.LINUX ファイルは、PGI 環境では以下のディレクトリに置かれております。$PGI は、PGI をインストールしたディレクトリ名で、デフォルトは/opt/pgi となります。なお、PGI CDK 製品では、インストール時に必要とする並列計算用のノード名をセットしますので、以下のようなカスタマイズは必要ありません。以下のパス名の例は、**PGI 201**7**(15.x)**の場合の例です。

（64 ビット Linux 環境） $PGI/linux86-64/**17.x**/mpi/mpich/share/machines.LINUX

このファイルの中に、以下のような形態で MPI 並列実行に供される「ホスト名」を 1 行ずつ指定（変更）します。 以下の例では、ローカル並びにリモートの 3 台のホスト名が photon26/27/28 と言う名称で、その名前に続くコロン以降の数字は、そのホストに搭載されている「プロセッサ総コア数」を指定した例です。なお、これらのホストの Linux のバージョンは、glibc ライブラリの互換性の問題により同じであることが必要です。

```
photon26:2
photon27:4
photon28:4
```

■ MPI ライブラリをリンクするためのコンパイル・オプション

PGI 2016 以降、Linux 用には Open MPI ライブラリがバンドルされました。コンパイル時のコマンドは mpif90/mpicc 等のドライバコマンドを使用します。PGI 2014 ～ PGI 2015 では、PGI Workstation/Server ライセンスにおいは、以下の -Mmpi= オプションが使用できます。なお、MPI の実行はローカルな 1 台のシステム内での実行となります（クラスタ・ノード間での実行はできません）。また、Open MPI(Linux 版)、MPICH3(OS X 版)、MS-MPI(Windows 版）以外の MPI ライブラリは実装されておりませんので、他のライブラリが必要な場合はご自身で MPI ライブラリを実装する必要があります。

PGI CDK for Linux 製品は、同様に Open MPI が付属しており、mpiexec の実行は、ローカルノード上だけでなく、クラスタ・ワイドで実行出来ます。オプション MPI ライブラリとして、PGI 用のプリ・コンパイルされた MVAPICH2 と MPICH3 が付属しておりますが、これは別途、インストールする必要があります。

MPI ソフトウェア	OS	-Mmpi= オプション	バンドルしている PGI 製品	適用バージョン
MPICH1	Linux	-Mmpi=mpich1	PGI Linux 版に付属	PGI 13.10 以前
MPICH2	Linux	-Mmpi=mpich2	PGI CDK に付属	PGI 13.10 以前
MPICH3	Linux/OS X	-Mmpi=mpich	Linux/OS X 版に	PGI 14.1 以降

			付属	
Open MPI	Linux	mpif90/mpic ラッパー使用	Linux 版に付属	PGI 16.1 以降
Open MPI	Linux CDK	mpif90/mpicc ラッパー使用	PGI CDK に付属	PGI 14.1 以降
MVAPICH1	Linux CDK	-Mmpi=mvapich1	PGI CDK に付属	PGI 13.10 以前
MVAPICH2	Linux CDK	mpif90/mpicc ラッパー使用	PGI CDK 版に付属	PGI 14.1 以降
SGI MPI	Linux or CDK	-Mmpi=sgimpi	--	PGI 13.5 以降
MS-MPI	Windows	-Mmpi=msmpi	PGI 2013 から PGI Windows 版に付属	PGI 7.1 以降 使用可能
MPICH3	OS X	Mmpi=mpich	OS X 版に付属	PGI 14.1 以降

● **PGI Workstation and PGI Server** 製品 (PGI 16.1 以降)

(Linux 版)

mpif90/mpicc/mpic++ -fastsse -Minfo {file_name} (Open MPI ライブラリ使用)

pgfortran/pgcc/pgc++ -fastsse -Minfo {file_name} **-Mmpi=sgimpi**

(SGI MPI ライブラリを別途実装した場合)

(OS X 版)

pgfortran/pgcc/pgc++ -fastsse -Minfo {file_name} **-Mmpi=mpich**

(MPICH3 ライブラリ使用)

(Windows 版 MPMSI)

pgfortran -fastsse -Minfo test.f90 -Mmpi=msmpi

pgcc -fastsse -Minfo test.c -Mmpi=msmpi

（なお、C++コンパイラは終息しました）

● **PGI Workstation and PGI Server** 製品 (PGI 14.1 ～ 15.10)

(Linux / OS X 版)

pgfortran/pgcc/pgc++ -fastsse -Minfo test.f90 -**Mmpi=mpich**

(MPICH3 ライブラリ使用)

pgfortran/pgcc/pgc++ -fastsse -Minfo test.f90 -Mmpi=sgimpi

(SGI MPI ライブラリを別途実装した場合)

なお、OS X 版の pgc++ コマンドは PGI 15.1 から提供開始。それ以前のバージョンでは、pgCC を使用する。 他の Open MPI を使用したい場合は、ご自身でオープンソースからビルドが必要

(Windows 版 MPMSI：PGI 2013 からバンドルされた)

pgf95/pgcc/pgCC -fastsse -Minfo test.f -**Mmpi=msmpi**

● **PGI Workstation and PGI Server** 製品 (PGI 8.0 ～13.10)

(Linux 版)

pgfortran/pgcc/pgcpp -fastsse -Minfo test.f90 –Mmpi=mpich1 (MPICH-1 ライブラリ使用)

pgfortran/pgcc/pgcpp -fastsse -Minfo test.f90 -Mmpi=sgimpi (SGI-MPI ライブラリ使用)

他の MPICH2、Open MPI を使用したい場合は、ご自身でオープンソースからビルドが必要

(Windows 版 MS-MPI：Microsoft(R) HPC Pack 2008 SDK 実装済みのシステム上)

pgfortran/pgcc/pgcpp -fastsse -Minfo test.f90 -Mmpi=msmpi

● PGI Cluster Development Kit (CDK) 製品

(PGI CDK 16.1 以降) 適切な PATH 環境変数を設定すること

pgfortran/pgcc/pgc++ -fastsse -Minfo test.f90 -**Mmpi=mpich** (MPICH3 ライブラリ使用)

mpif90/mpicc/mpic++ -fastsse -Minfo test.f90 (MVAPICH2 ライブラリ使用の場合、wrapper コマンド使用)

mpif90/mpicc/mpic++ -fastsse -Minfo test.f90 (Open MPI ライブラリ使用の場合、wrapper コマンド使用)

(PGI CDK 14.1 ~ 15.10)

pgfortran/pgcc/pgc++ -fastsse -Minfo test.f90 -Mmpi=mpich (MPICH3 ライブラリ使用)

mpif90/mpicc -fastsse -Minfo test.f90 (MVAPICH2 ライブラリ使用の場合、wrapper コマンド使用)

mpif90/mpicc -fastsse -Minfo test.f90 (Open MPI ライブラリ使用の場合、wrapper コマンド使用)

(PGI CDK 7.1 以降)

pgfortran/pgcc/pgcpp -fastsse -Minfo test.f90 -Mmpi=mpich1 (MPICH-1 ライブラリ使用)

pgfortran/pgcc/pgcpp -fastsse -Minfo test.f90 -Mmpi=mpich2 (MPICH-2 ライブラリ使用)

pgfortran/pgcc/pgcpp-fastsse -Minfo test.f 90-Mmpi=mvapich1 (MVAPICH-1 ライブラリ使用)

(PGI CDK 7.0 以前)

pgfortran/pgcc/pgcpp -fastsse -Minfo test.f90 -Mmpi (MPICH-1 ライブラリ使用)

pgfortran/pgcc/pgcpp -fastsse -Minfo test.f90 -Mmpi2 (MPICH-2 ライブラリ使用)

- PGI 16.1 以降の Linux 版では、Open MPI がデフォルト使用されるため、コンパイルには、mpif90/mpicc/mpic++ のラッパーを使用して下さい。適切な PATH 環境変数の設定が必要です。このラッパーコマンドを使用する場合は、-Mmpi オプションは必要ありません。
- Open MPI 以外はラッパーコマンドを使用しないで、-Mmpi=mpich3 等のオプションを使うことも出来ます。この指定により、PGI コンパイラ製品に実装された MPI ライブラリを使用し、当該 MPI のインクルードファイルを取り込み、リンク時には適切な MPI ライブラリをリンクできます。

  Linux システム上では、このオプションは、コンパイル時に -I$MPIDIR/include をセットし、リンク・フェーズでは、-L$MPIDIR/lib をコマンドライン上に挿入します（予め環境変数 MPIDIR が設定されております）。指定されたサブオプション (mpich1,mpich2,mpich,mvapich1,mvapich2 等) は、MPICH-1 、MPICH-2、MPICH-3、MVAPICH-1、MVAPICH2 通信ライブラリを選択するものです。MPICH ライブラリを実装している「ベース・ディレクトリ（$MPIDIR)」は、サイトのコンパイラ初期設定ファイル siterc に設定されております。PGI が提供するプリ・コンパイル MPICH ライブラリは、コンパイラのインストール時に自動的にセットされますが、ユーザ自身がビルドした、MPICH ライブラリの場合でも、MPIDIR のそのベース・ディレクトリのパス名をセットすることにより、上記のようなオプションで MPI ライブラリを使用できます（詳細は、PGI User's Guide をご覧下さい)。なお、siterc ファイルは、例えば、$PGI/linux86-64/{version}/bin 配下に存在します。MPICH2 の場合も同様に、環境変数 MPI2DIR と言うものがありますが、ご自身で実装した MPICH2 の場合は、このオプションではなく、mpif90/mpicc 等のコマンドを使用することを推奨します。
- Windows(R) システム上では、-Mmpi=msmpi オプションにより、コンパイル時に -I$(CCP_INC)/include をセットし、リンク・フェーズでは、-L$(CCP_LIB**)/lib をコマンドライン上に挿入します。CCP_HOME、CCP_SDK 環境変数は、必ずセットされていなければなりませんが、これは一般に、Microsoft(R) HPC pack 2008 SDK をインストールするとき、この変数は、MSMPI ディレクトリの場所にセットされます。ご自身でセットする必要はありません。PGI 14.1 以降は、マイクロソフト社の MS-MPI の環境変数は、MSMPI_INC、MSMPI_LIB32、MPSMPI_LIB64 に変更されております。

■ MPI を使用する際、2GB 以上の配列オブジェクトが存在する場合

2GB 以上の配列オブジェクトを有する MPI プログラムのコンパイルの方法に関しては、以下の URL をご覧下さい。

http://www.softek.co.jp/SPG/Pgi/TIPS/opt_mpi.html#mcmodel

■ MPI プログラムの並列実行方法

```
(Open MPI コンパイル、リンク)
  mpif90  -fastsse -o mpihello mpihello.f
(MPICH v3 コンパイル、リンク)
  pgfortran -fastsse -o mpihello mpihello.f -Mmpi=mpich
(MPI 並列実行)
  mpirun -np 4 ./mpihello
  Hello world!  I'm node            1
  Hello world!  I'm node            3
  Hello world!  I'm node            0
  Hello world!  I'm node            2
-------------------------------------------------------------------------------------------------------
(Open MPI コンパイル、リンク： PGI CDK)
  mpif90 -o mpihello mpihello.f
  mpicc -o myname myname.c
(MPICH v3 コンパイル、リンク： PGI CDK)
  pgfortran -o mpihello mpihello.f -Mmpi=mpich
  pgcc -o myname myname.c -Mmpi=mpich
(MPI 並列実行)
  photon27:> mpirun -np 4 ./mpihello（二つのノードの 4CPU を使用して mpihello 並列実行）
  Hello world!  I'm node            1
  Hello world!  I'm node            3
  Hello world!  I'm node            0
  Hello world!  I'm node            2
  photon27:> mpiexec -n 4 ./myname   （二つのノードの 4CPU を使用して myname 並列実行）
  My name is photon27
  My name is photon26
  My name is photon26
  My name is photon27
```

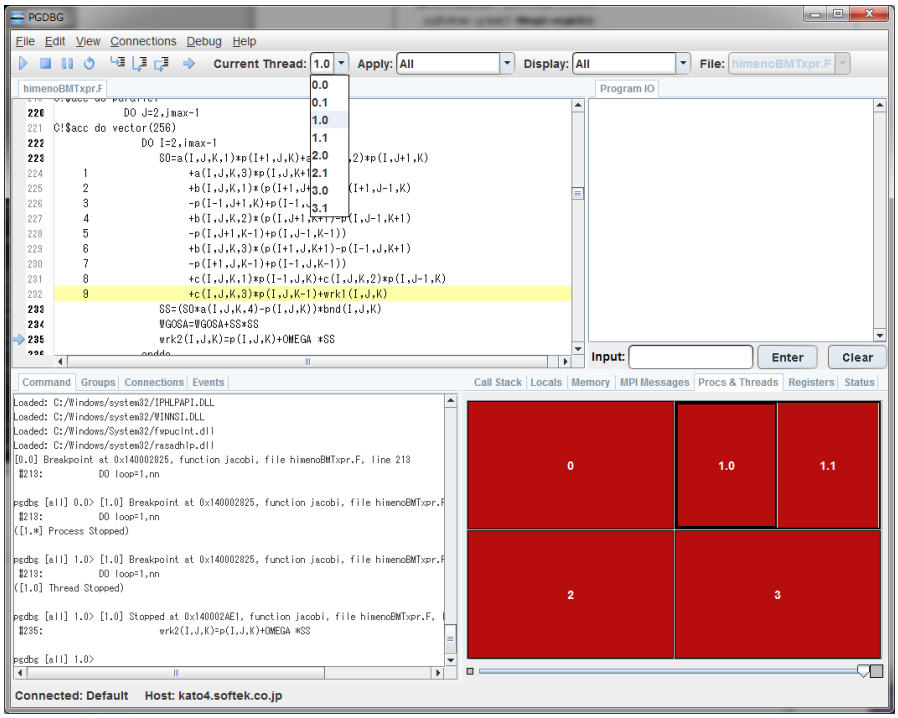
- MPI 並列実行時のプロセス数に制約はありません。
- 上記の mpihello.f と myname.c のソースプログラムは、以下の URL にあります。
  http://www.softek.co.jp/SPG/Pgi/TIPS/mpich.html#mpi_program

■ MPI 対応 PGDBG 並列デバッガの利用　(PGI Workstation / Server ライセンス)

```
(Open MPI コンパイル、リンク)
   mpif90 -g  test.f   (PGI 16.1 以降)
(MPICH コンパイル、リンク)
   pgfortran -g test.f –Mmpi=mpich  (PGI 14.1 ～15.10)
   pgfortran -g test.f –Mmpi=mpich1  (PGI 8.0 ～13.10)

(ローカルノード上で動作させるための machinefile の設定)
 ローカル・マシン名を photon27 とする。"photon27"で 4 プロセス用分を記述
   photon27:> cat hostfile
   photon27
```

```
  photon27
  photon27
  photon27
（PGDBG for Open MPI デバッガの起動　PGI 16.1～）
  pgdbg -mpi -np 4 ./a.out

（PGDBG for MPICH-3 デバッガの起動　PGI 14.1～15.10）
  pgdbg -mpi -n 4 ./a.out

（PGDBG for MPICH-1 デバッガの起動　PGI 8.0 ～13.10 ）
  mpirun -np 4 -dbg=pgdbg -machinefile hostfile ./a.out
```

- PGI Workstation/Server ライセンスの場合は、MPI 並列デバッギング機能は、PGI コンパイラをインストールしたローカル上(同一システム上)での 16 プロセスまでの MPICH/open MPI プロセス・デバッギングに制約されます。
- 一方、PGI CDK 製品は、ローカル並びにリモート MPI プロセスに対応し、その上限プロセス数は、購入ライセンスの「PGI CDK プロセス数」まで対応します。
- PGI デバッガの詳細は、PGI Debugger User Guide をご参照ください。



4 ローカル・プロセスの MPI プログラムのデバッギングの様子

■ MPI 対応 PGDBG 並列デバッガの利用　(PGI CDK ライセンス)

```
(Open MPI コンパイル、リンク)
    mpif90 -g test.f    (PGI 16.1 以降)
(MPICH v3 コンパイル、リンク)
   pgfortran -g test.f90 -Mmpi=mpich

(PGDBG for MPICH v3 /Open MPI デバッガの起動)
   pgdbg -mpi -n 4 ./a.out

(MVAPICH-2, Open MPI コンパイル、リンク)
   mpif90 -g test.f90 (他、mpicc、mpic++ コマンドあり)

(PGDBG for MVAPICH-2, Open MPI デバッガの起動)
   pgdbg –mpi=<launcher_path> -n 8 ./a.out          (8 プロセスの MPI デバッギング)
```

- PGI CDK ライセンスでは、ローカル並びにリモート MPI プロセスに対応し、その上限プロセス数は、購入ライセンスの「PGI CDK プロセス数」まで対応します。
- MVAPICH2/Open MPI を使用したデバッグの起動時の-mpi への引数に関して、MPI プログラム起動 launcher である mpiexec が、環境変数 PATH にて参照出来る場合は、-mpi のみの引数でよいですが、launcher コマンドが参照出来なければ、フルパス名で当該 Launcher を指定して下さい。なお、 PGI CDK ソフトウェアがインストールされ、ライセンスされているときにのみ、このオプションでデバッガが起動できます。

■ MPI 対応 PGPROF 並列プロファイラの利用 (PGI Workstation/Server、PGI CDK ライセンス)

```
(MPICH v3 コンパイル、リンク, PGI 14.1～15.10)
 pgfortran -fastsse -Minfo -Mprof=mpich,func test.f

(MPICH 実行後、pgprof の起動)
 mpiexec -n 4 ./a.out
 pgprof
```

PGI Workstation/Server ライセンスの場合は、以下の環境で行う

```
  (ローカルノード上で動作させるための machinefile の設定)
  ローカル・マシン名を photon27 とする。"photon27"で 4 プロセス用分を記述
  photon27:> cat hostfile
  photon27
  photon27
  photon27
  photon27

(MPICH 実行後、pgprof の起動)
  mpiexec -n 4 -machinefile hostfile ./a.out
  pgprof
```

- MPI 対応のプロファイラの機能は、PGI 15.10 リリースまでの機能です。PGI 16.1 以降は、本機能は提供されていません。
- PGI Workstation/Server ライセンスの場合は、MPI 並列プロファイリング機能は、PGI コン

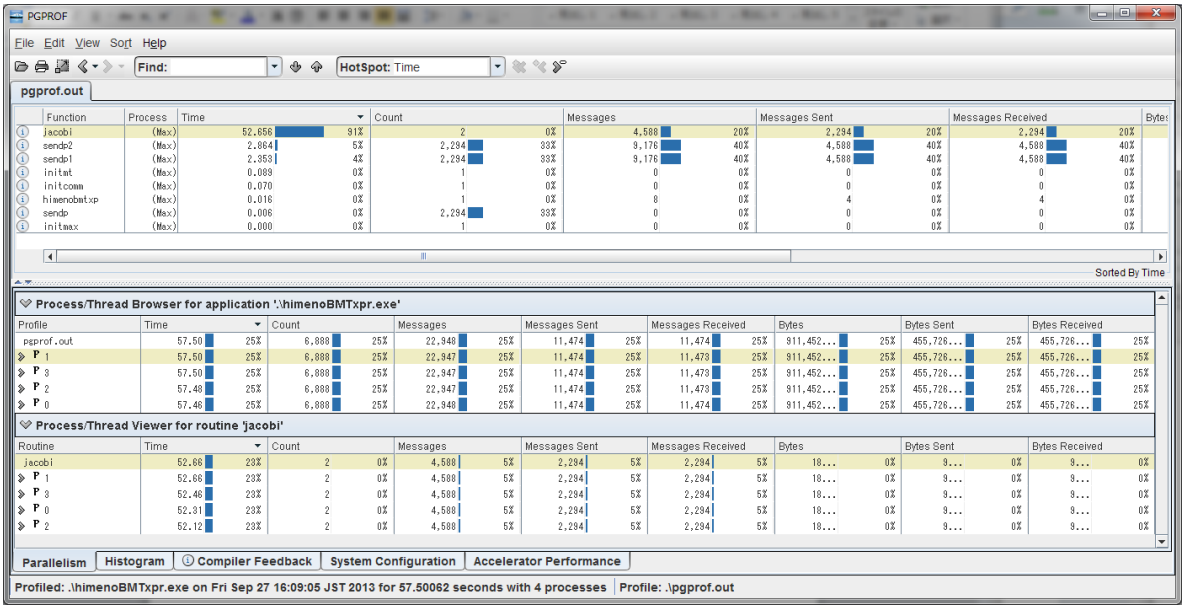
パイラをインストールしたローカル上 (同一システム上) での 16 プロセスまでの MPICH-1 プロセスの実行プロファイルに制約されます。PGI CDK ライセンスでは、ローカル並びにリモート MPI プロセスに対応し、その上限プロセス数は、購入ライセンスの「PGI CDK プロセス数」まで対応します。

- -Mprof[=[mpich |msmpi] -Mprof オプションに mpich (Linux/OS X) あるいは、msmpi (Windows)を指定し、この後、コンマに引き続きプロファイルの機能 (func|line|time)を指定する。mpich を指定した場合は、MPICH v3 用のプログラムを対象とする。msmpi を指定した場合は、Windows 上の MS-MPI を使用したプログラムを対象とする。

- PGI Profiler for MPI に関する詳細な説明は、PGI User's Guide 6.4 項 と PGI Profiler User Guide をご参照ください。

以下は、PGI 13.10 までのコンパイラで有効です。

- mpich1 サブオプションは、プロファイル用 MPICH-1 ライブラリを使用します。-Mmpi=mpich1 を包含します (PGI 14.1 以降廃止)。
- mpich2 サブオプションは、プロファイル用 MPICH-2 ライブラリを使用します。-Mmpi=mpich2 を包含します (PGI 14.1 以降廃止)。
- mvapich1 サブオプションは、プロファイル用 MVAPICH-1 ライブラリを使用します。-Mmpi=mvapich1 を包含します (PGI 14.1 以降廃止)。



MPICH プログラムのプロファイリングの様子

3 PGI アクセラレータ・コンパイラ製品の機能 (GPGPU 用のコンパイル機能)

PGI アクセラレータ・コンパイラ製品は、NVIDIA 社の GPU / GPGPU とその CUDA 開発環境を実装したシステム上で、GPU を活用するためのコンパイラを含めたプログラム開発環境を提供します。「アクセラレータ」とは、特別の目的で CPU にアタッチして使用する協調プロセッサ（GPU）であり、時間の掛かる計算部分を CPU の演算機構からそのデータと実行部分のカーネルをオフロードするために使用されます。

PGI アクセラレータ・コンパイラの機能は、Fortran と C99、C++の三つの言語に対応しており、NVIDIA 社の GPU を備えた、全てのインテル(R)プロセッサ並びに AMD のプロセッサベースのシステム上で動作し、Linux、Windows、Apple の各プラットフォームに対応しています。PGI アクセラレータ・コンパイラ製品の機能とその構成を以下の図に示しました。本コンパイラ製品は、インテル(R)プロセッサ並びに AMD のプロセッサのマルチコアに対応した、従来のコンパイラ製品の上位の製品系列に位置づけられ、従来のホスト側の Fortran/C/C++コンパイラ環境がベースとなります。PGI アクセラレータ・コンパイラ製品の機能としては、

(1)「コンパイラ指示行によるプログラミングが可能な OpenACC プログラミング機能」
(2)「CUDA API を利用して明示的にプログラミングを行うための PGI CUDA Fortran 拡張機能」

の両方のモデルに対するコンパイル機能を備えます。一般に前者のようなコンパイラ指示行のみの挿入で行うような言語開発モデルを Implicit Programming Model と称し、また、後者のような開発者が明示的に CUDA API を使用してコーディングするような言語モデルを Explicit Programming Model と言います。PGI アクセラレータ・コンパイラ製品は、この両方のプログラミング機能を提供します。

3.1 GPU /アクセラレータ対応 PGI コンパイラの二つのプログラミングモデル

PGI アクセラレータ・コンパイラを使用して、GPU / GPGPU 対応の並列プログラミングを行う方法は、上述したように「二つの方法」が用意されております。具体的なコードを示し、簡単に概略を説明します。

(1) **PGI Accelerator Programming Model(OpenACC)** の例（コンパイラ指示行による）

一つは、スレッド並列用の標準規約 OpenMP のようなコンパイラ指示行により、その並列領域を指定してコンパイラに GPU（アクセラレータ用）並列コードを生成させる方法です。これは、PGI 社が提唱、公開した規約である「PGI Fortran & C Accelerator Programming Model」を発展させた OpenACC 規約による指示行をプログラムに挿入するだけで、ユーザは高級言語レベルで host + accelerator 用のコードを作成できます。そのプログラムの一例を以下に示します。

```
      subroutine saxpy( n, a, x, y )
      real, dimension(*) :: x,y
      real :: a
      integer :: n, i
!$acc kernels
            do i = 1, n
                  y(i) = a * x(i) + y(i)
            end do
!$acc end kernels
      end subroutine
```

(2) **PGI CUDA Fortran** プログラミングの例（明示的な CUDA API の利用）

CUDA は NVIDIA 社の GPU のアーキテクチャですが、NVIDIA 社からの CUDA 開発環境は、CUDA C として知られている拡張 C コンパイラとツール群のみが提供されております。CUDA C は、C/C++高級言語上から GPU のために CUDA API を使用して明示的にプログラミングすることができるものです。PGI 社と NVIDIA 社は共同で CUDA Fortran の開発を行い、CUDA C と同等な機能を PGI Fortran 95/Fortran 2003 コンパイラに実装しました。すなわち、Fortran 上の CUDA 関数呼び出しと言語拡張を行うことにより、GPU への汎用数値演算処理カーネルをマッピングすることや、x64 プロセッサと GPU 間のデータの移動と配置を明示的に制御できます。PGI CUDA Fortran コンパイラは、ネイティブ Fortran 環境で CUDA C と同等レベルの制御と最適化を実現し、PGI CUDA Fortran と CUDA C/C++の相互運用を可能にします。

```
! GPGPU kernel definition
        attributes(global) subroutine ksaxpy( n, a, x, y )
        real, dimension(*) :: x,y
        real, value :: a
        integer, value :: n, i
                i = (blockidx%x-1) * blockdim%x + threadidx%x
                if( i <= n ) y(i) = a * x(i) + y(i)
        end subroutine
! Host subroutine
        subroutine solve( n, a, x, y )
        real, device, dimension(*) :: x, y
        real :: a
        integer :: n
! call the kernel
                call ksaxpy<<<n/64, 64>>>( n, a, x, y )
        end subroutine
```

3.2 PGI アクセラレータ・プログラミングモデルの使用

本項の詳細は、http://www.softek.co.jp/SPG/Pgi/TIPS/opt_accel.html に説明しています。また、OpenACC に関するガイドは、http://www.softek.co.jp/SPG/Pgi/OpenACC/index.html にありますので、詳細なプログラミングの方法に関してはこのガイドをご参照ください。

■ OpenACC コンパイル・オプション

ここでは、 OpenACC で記述されたプログラムをコンパイルする際のコンパイル・オプションについて説明します。PGI の F2003, C99, C++ のコンパイラを使用して、PGI アクセラレータ用のコンパイルを行うオプションの例を示します。以下は、pgfortran（pgf95、あるいは、pgf90 も同じコンパイラです）を使用した場合の例ですが、C/C++言語用の pgcc /pgc++コンパイラのオプションの設定方法も、同様です。なお、コマンドライン上でリンク時にも、必ず、コンパイル時に指定したものと同じ -acc オプションを指定することが必要です。また、ターゲット・アクセラレータ用に細かなサブオプションを指定するために、-acc に併せて、–ta オプションも指定することが出来ます。

● Fortran コンパイルの一例

```
pgfortran -fast -Minfo –acc -ta=tesla test.f
pgfortran -fast -Minfo=accel –acc   test.f90
```

● C99/C++ コンパイルの一例
pgcc/pgc++ -fast -Minfo **–acc -ta=tesla** test.c
pgcc/pgc++ -fast -Minfo=accel **-acc** test.c
(PGI16.1 以降、Windows 版 C++コンパイラは終息しました)

PGI 16.10 以降、正式に CUDA 8.0 toolkit をサポートしました。Pascal (CC6.0) 用の executable を作成する場合は、CUDA 8.0 ライブラリを必要とする場合がありますので、必ず、-ta=tesla,cuda8.0 あるいは、-ta=tesla,cc60 のオプションを明示的に付けてコンパイルしてください。PGI 17.1 以降のコンパイラのデフォルトは CUDA 7.5 ライブラリをリンクする形となっております。

- -Minfo=accel ： このオプションを指定すると、コンパイラがアクセラレータ領域を GPU カーネルに翻訳できたかどうかについて、コンパイラのメッセージとして出力します。-Minfo のみの指定では、その他の最適化情報も併せてメッセージとして出力します。

- -acc[=[no]autopar|[no]required|strict|verystrict] : OpenACC ディレクティブを認識するオプションです。コンパイル時だけではなく、リンク時にも必要です。

  -acc=[no]autopar は、OpenACC parallel 構文内の自動並列化を行う[行わない]を指定します。(PGI 13.6 以降)
  -acc=[no]required はアクセラレータ・コードを生成出来なかった場合、コンパイルエラーとする(default) (PGI 14.1 以降、但し、PGI 15.1 で廃止)
  -acc=strict は、non-OpenACC accelerator ディレクティブが見つかった場合、warning を出します。
  -acc=verystrict は、non-OpenACC accelerator ディレクティブが見つかった場合、エラーメッセージを出し、コンパイルを終了します。

- PGI 14.1 以降、-ta オプションの指定方法が変更されました。 PGI 14.1 以降 AMD の Radeon GPU ボードも OpenACC 対応となったため、以下のように、NVIDIA 社と AMD 社の二つのメーカの通称ボード名で、OpenACC コンパイルの「ターゲットの識別」を行います。さらに、各ターゲットに対する細かなオプションを指定できます。(デフォルトは -ta=tesla,host です)

  -ta=tesla(,tesla_suboptions) ： NVIDIA 社の GPU ボード対応の OpenACC コンパイルを行います。従来の -ta="nvidia" の "nvidia"名 は今後、廃止される予定です。
  -ta=radeon(,radeon_suboptions) ： AMD 社の Radeon GPU/APU ボード対応の OpenACC コンパイルを行います。

  (PGI 13.10 以前の指定方法) -ta=nvidia(,nvidia_suboptions) ： PGI アクセラレータディレクティブあるいは、OpenACC ディレクティブを認識するオプションです。-ta は、ターゲット・アーキテクチャを意味します。PGI 13.10 までは、「nvidia」 のみとなります。Fortran における !$acc ディレクティブ、C における #pragma acc ディレクティブをコンパイラに認識させ、ターゲットへの細かなオプションを使用するために、以下のサブオプションを使用できます。

  -ta=tesla - NVIDIA アクセラレータをターゲットとして選択します。さらに、以下の tesla(nvidia) 用のサブオプションがあります。このサブオプションは、カンマ（,）で区切って複数のものを指定することができます。

サブオプション	nvidia 用 機能
Analysis	ループの解析のみ行い、コードの生成を行いません。 (PGI 13.10 以降廃止)

cc10	compute capability 1.0 のコードを生成 (PGI 14.1 以降廃止)
cc11	compute capability 1.1 のコードを生成 (PGI 14.1 以降廃止)
cc12	compute capability 1.2 のコードを生成 (PGI 14.1 以降廃止)
cc13	compute capability 1.3 のコードを生成 (PGI 14.1 以降廃止)
cc1x	compute capability 1.x のコードを生成(PGI 15.1 以降廃止)
cc1+	compute capability 1.x, 2.x, 3.x のコードを生成 (PGI 14.1 以降)、(PGI 15.1 以降廃止)
cc20	compute capability 2.0 のコードを生成 (PGI 10.4 以降) (PGI 14.1 以降廃止)
cc2x	compute capability 2.x のコードを生成 (PGI 10.4 以降)
cc2+	compute capability 2.x, 3.x のコードを生成 (PGI 14.1 以降)
fermi	cc2x と同じ (PGI 13.1 以降)
fermi+	cc2+と同じ (PGI 14.1 以降)
cc30	compute capability 3.0 のコードを生成 (PGI 12.8 以降) (PGI 14.1 以降廃止)
cc35	compute capability 3.5 のコードを生成 (PGI 13.1 以降) (PGI 14.1 以降廃止)
cc3x	compute capability 3.x のコードを生成 (PGI 12.8 以降)
cc3+	compute capability 3.x (=cc3x) 以上のコードを生成 (PGI 14.1 以降)
kepler	cc3x と同じ (PGI 13.1 以降)
kepler+	cc3+と同じ (PGI 14.1 以降)
cc50	compute capability 5.0 (=cc50) (PGI 15.7 以降)
Cc60	compute capability 6.0 (=cc60) (PGI 16.9 以降)
charstring	GPU カーネル内で文字列の使用を制限付きで使用する(PGI 15.1 以降)
cuda2.3 or 2.3	PGI にバンドルされた CUDA toolkit 2.3 バージョンを使用 (PGI 10.4 以降)
cuda3.0 or 3.0	PGI にバンドルされた CUDA toolkit 3.0 バージョンを使用 (PGI 10.4 以降)
cuda3.1 or 3.1	PGI にバンドルされた CUDA toolkit 3.1 バージョンを使用 (PGI 10.8 以降)
cuda3.2 or 3.2	PGI にバンドルされた CUDA toolkit 3.2 バージョンを使用 (PGI 11.0 以降)
cuda4.0 or 4.0	PGI にバンドルされた CUDA toolkit 4.0 バージョンを使用 (PGI 11.6 以降)
cuda4.1 or 4.1	PGI にバンドルされた CUDA toolkit 4.1 バージョンを使用 (PGI 12.2 以降)
cuda4.2 or 4.2	PGI にバンドルされた CUDA toolkit 4.2 バージョンを使用 (PGI 12.6 以降)
cuda5.0 or 5.0	PGI にバンドルされた CUDA toolkit 5.0 バージョンを使用 (PGI 13.1 以降)
cuda5.5 or 5.5	PGI にバンドルされた CUDA toolkit 5.5 バージョンを使用 (PGI 13.9 以降)
cuda6.0 or 6.0	PGI にバンドルされた CUDA toolkit 6.0 バージョンを使用 (PGI 14.4 以降)
cuda6.5 or 6.5	PGI にバンドルされた CUDA toolkit 6.5 バージョンを使用

	(PGI 14.9 以降)
cuda7.0 or 7.0	PGI にバンドルされた CUDA toolkit 7.0 バージョンを使用 (PGI 15.4 以降)
cuda7.5 or 7.5	PGI にバンドルされた CUDA toolkit 7.5 バージョンを使用 (PGI 15.9 以降)
Cuda8.0 or 8.0	PGI にバンドルされた CUDA toolkit 8.0 バージョンを使用 (PGI 15.9 以降)
[no]debug	デバイスコード内にデバッグ情報を生成する[しない] (PGI 14.1 以降)
fastmath	fast math ライブラリを使用
[no]flushz	GPU 上の浮動小数点演算の flush-to-zero モードを制御。デフォルトは noflushz。(PGI 11.5 以降)
[no]fma	fused-multiply-add 命令を生成する[しない]（-O3 ではデフォルト）
keep	kernel バイナリファイル(.bin)、kernel ソースファイル(.gpu)、portable assembly(.ptx)ファイルを保持し、各々ファイルとして出力する (PGI 13.10 以降)
keepbin	kernel バイナリファイルを保持し、ファイル(.bin)として出力する (PGI 13.10 以降廃止)
keepgpu	kernel ソースファイルを保持し、ファイル(.gpu)として出力する (PGI 13.10 以降廃止)
keepptx	GPU コードのための portable assembly(.ptx)ファイルを保持し、ファイルとして出力する (PGI 13.10 以降廃止)
[no]lineinfo	GPU line information を生成する(PGI 15.1 以降)
[no]llvm	llvm ベースのバックエンドを使用してコードを生成する。(PGI 15.1 以降) 64-bit 上では LLVM バックエンドを使う [使わない]。なお、PGI 15.1 以降はデフォルト llvm を使うように変更された。リンク時にエラーが生じる場合、nollvm を試すことをお勧めする。
maxregcount:n	GPU 上で使用するレジスタの最大数を指定。ブランクの場合は、制約が無いと解釈する
mul24	添字計算に、24 ビット乗算を使用 (GT200 系、CC 1.3 のみ)
nofma	fused-multiply-add 命令を生成しない
noL1	グローバル変数をキャッシュするためのハードウェア L1 データキャッシュの使用を抑止する (PGI 13.10 以降)
loadcache:L1 loadcache:L2	グローバル変数をキャッシュするためにハードウェア L1 あるいは L2 データキャッシュを使用する。但し、アーキテクチャ上、有効とならない GPU がある (PGI 14.4 以降)
pin	デフォルトを pin ホストメモリ(割付）としてセットする(PGI 14.1~15.10)
pinned	デフォルトを pin ホストメモリ(割付）としてセットする。pin ホストメモリ(割付）としてセットする。プログラムのアロケート時に pinned メモリを割り付けるように変更した (PGI 16.1 以降)
time	アクセラレータ領域の単純な時間情報を集積するためにプロファイル・ライブラリをリンクする。このオプションは、PGI 13.1 以降廃止されました。この代わりに、プロファイルを環境変数

	PGI_ACC_TIME に 1 をセットすることにより実行後プロファイル情報が出力されます。
[no]required	アクセラレータ・コードを生成出来なかった場合、コンパイルエラーとする[しない] (default)(PGI 15.1 以降廃止)
[no]rdc	異なるファイルに配置されたデバイスルーチンをそれぞれ分割コンパイルし、リンクが出来るようにする。cc2x 以降、CUDA 5.0 以降の機能を使用する。(PGI 13.1 以降 + CUDA 5.0 以降) (PGI 14.1 は以降デフォルト)
[no]unroll	自動的に最内側ループのアンローリングを行う (default at -O3) (PGI 14.9 以降)
managed	CUDA managed Memory を使用する
beta	ベータ版機能のコード生成（生成コード内の 128-bit ロード・ストアオペレーションを有効化）(PGI 15.7 以降)
[no]wait	ホスト側での実行継続を行う際に、各カーネルが終了するまで待つ。nowait は待たない。 (PGI 10.8 以降)
safecache	cache directive 内での可変長の配列セクションの使用を許す。但し、そのサイズは CUDA shared memory 内に収まるものでなければならない。(PGI 16.5 以降)

-ta=radeon - AMD アクセラレータをターゲットとして選択します。さらに、以下の radeon 用のサブオプションがあります。このサブオプションは、カンマ（,）で区切って複数のものを指定することができます）(PGI 14.1 以降)。

サブオプション	AMD -ta=radeon のサブオプション
buffercount:n	データをアロケートする際の OpenCL バッファの最大数をセットする
keep	kernel ファイルを保持する
[no]lineinfo	GPU line information を生成する(PGI 15.1 以降)
[no]llvm	llvm/SPIR ベースのバックエンドを使用してコードを生成する。(PGI 15.1 以降) 64-bit 上では LLVM/SPIR バックエンドをデフォルトとして使う [使わない]
[no]unroll	自動的に最内側ループのアンローリングを行う (default at -O3)
spectre	Radeon Spectre アーキテクチャ用コードを生成
tahiti	Radeon Tahiti アーキテクチャ用コードを生成 (default)
capeverde	Radeon capeverde アーキテクチャ用コードを生成
spir	LLVM/SPIR バックエンドを 64-bit モードでフォルトとして使う (PGI 15.1 以降)

-ta=nvidia(radeon),**host** - ターゲットとして、host を選択する。nvidia オプションとの組み合わせで使用されます。アクセラレータ領域をホスト側で実行するようにコンパイルする。このオプションは、GPU が実装されていないシステムでも動作するような実行バイナリとなる PGI Unified Binary コードを生成します。この host サブオプションは、上記の nvidia 専用のサブオプションを指定した後に指定します（host は、最後に指定します）。

■ PGI アクセラレータ用実行時の環境変数

ACC_DEVICE : ACC_DEVICE 環境変数は、プログラムの実行モジュールが一つ以上の異

なるデバイスタイプを使用して実行できるように生成されていた場合（PGI Unified Binary)、「アクセラレータ・リージョン」を実行する際に使用するデフォルトのデバイスタイプを指定するものです。この環境変数の値は、コンパイラ・リリースのおける実装時に定義されていますが、現在、NVIDIA(nvidia) と HOST(host) が定義されています。
例：

```
export ACC_DEVICE=NVIDIA
setenv ACC_DEVICE NVIDIA
```

ACC_DEVICE_NUM ： ACC_DEVICE_NUM 環境変数は、「アクセラレータ・リージョン」を実行する際に使用するデフォルトのデバイス番号を指定するものです。環境変数の値は、0～正の整数でなければなりません。システム内に複数の GPU デバイスが実装されている場合、その論理番号が 0 から順番に付されて管理されています。pgaccelinfo コマンドを実行すると、各 GPU デバイスのロプロパティが論理番号順に表示できます。0 を指定した場合、システム実装時のデフォルトが使用されます。
例：

```
export ACC_DEVICE_NUM=1
setenv ACC_DEVICE_NUM 1
```

ACC_NOTIFY（PGI 13.10 まで)： ACC_NOTIFY 環境変数は、Kernel がアクセラレータ上で実行された際に、そのイベントを標準出力としてショートメッセージで印字するために使用されます。環境変数の値は、負の整数であってはなりません。0 を指定するとこの機能を抑止します（デフォルト)。0 以外の正数の場合は、カーネルを実行する毎に、ショートメッセージを標準出力に印字します。
例：

```
export ACC_NOTIFY=1
setenv ACC_NOTIFY 1
```

(メッセージ一例)

```
launch kernel file=/home/***/GPGPU/OpenMP/jacobi4.F function=jacobi line=229
device=1 grid=2500 block=128x4
```

PGI_ACC_NOTIFY - （PGI 14.1 以降新設）PGI_ACC_NOTIFY 環境変数は、ビットマスクとして利用する整数定数をセットして、デバイス上の実行イベントの情報を出力するためのものです。整数値 1 をセットすると Kernel launch のイベントを標準出力として出力します。整数値 2 は、データ転送のイベントの出力、整数値 4 の場合は、region の entry/exit 情報、整数値 8 は、デバイス上の wait/sync のイベントを出力します。0 を指定するとこの機能を抑止します（デフォルト)。
例：

```
export ACC_NOTIFY=2
setenv ACC_NOTIFY 2
```

(メッセージ一例)

```
upload CUDA data file=acc_f2a.f90 function=main line=37 device=0 variable=a
bytes=4000000
download CUDA data file=acc_f2a.f90 function=main line=41 device=0 variable=r
bytes=4000000
```

PGI_ACC_TIME ： PGI_ACC_TIME 環境変数は、実行後に簡易プロファイル情報を標準出力に出力するために使用されます。PGI 12.x まで有効であった -ta=time の "time" sub-option は廃止されました。環境変数の値は、負の整数であってはなりません。0 を指定するとこの機能を

抑止します(デフォルト)。0 以外の正数の場合は、プロファイル情報を標準出力に印字します。(PGI 13.1 以降) なお、この機能を有効にするには、LD_LIBRARY_PATH 環境変数に、PGI のライブラリ・パスを設定する必要があります。具体的には、64 ビット環境では、$PGI/linux86-64/{バージョン番号}/lib をセット、32 ビット環境では、$PGI/linux86/{バージョン番号}/lib をセットして、当該プログラムを実行します。
例 :

```
export PGI_ACC_TIME=1
setenv PGI_ACC_TIME 1
```

PGI_ACC_BUFFERSIZE - (PGI 14.1 以降新設) PGI_ACC_BUFFERSIZE 環境変数は、NVIDIA デバイスにおけるホストとデバイス間のデータ転送で使用される pinned buffer(Pinned memory 上) のサイズを指定するものである。

PGI_ACC_CUDA_GANGLIMIT - (PGI 14.1 以降新設) PGI_ACC_CUDA_GANGLIMIT 環境変数は、NVIDIA デバイスにおける、kernel によって起動される gang (CUDA thread block) の最大数を指定するものである。

PGI_ACC_DEV_MEMORY - (PGI 14.1 以降新設) PGI_ACC_DEV_MEMORY 環境変数は、AMD Radeon デバイスにおける、アロケートされる OpenCL のバッファの最大値を指定するものである。この最大値は、ターゲット・デバイスによって制限される場合がある。

ACC_NUM_CORES - ACC_NUM_CORES 環境変数は、-ta=multicore オプションで作成された マルチコア CPU 上で動作する OpenACC プログラムを動作させる際の利用するコア数を指定するものです。

ACC_BIND - ACC_BIND 環境変数は、-ta=multicore オプションで作成された マルチコア CPU 上で動作する OpenACC プログラムを動作させる際、デフォルトでセットされます。ACC_BIND の設定挙動は、OpenMP における MP_BIND の挙動と同様です。

■ PGI アクセラレータのコンパイル事例

▸ 使用例 1 一般的な GPU アクセラレータ用のコンパイル・オプション

```
$ pgfortran -fast -Minfo=accel -acc f1.f90
main:
      21, Generating copyin(a(1:n))
          Generating copyout(r(1:n))
      22, Loop is parallelizable
          Accelerator kernel generated
          22, !$acc do parallel, vector(256)

$ pgcc -fast -Minfo=accel –acc c1.c
main:
      23, Generating copyin(a[0:n-1])
          Generating copyout(r[0:n-1])
      25, Loop is parallelizable
          Accelerator kernel generated
          25, #pragma acc for parallel, vector(256)
```

▸ 使用例 2 GPU 実行処理時のプロファイルデータを出力

以下の例は、Linux 上での状況を示したものです。"a.out" と言う実行モジュール名は、Linux のデフォルト名です。Windows 上では、以下の例の場合、デフォルトでは f1.exe、c1.exe と言う名前の実行バイナリとなります。

```
$ export PGI_ACC_TIME=1
$ pgfortran -fast -Minfo=accel –acc f1.f90
$ ./a.out
Accelerator Kernel Timing data
  main  NVIDIA  devicenum=0
    time(us): 249
    21: compute region reached 1 time                (GPU 計算領域に 1 回入った)
        21: data copyin reached 1 time               (ホストからデバイスへデータコピー)
            device time(us): total=79 max=79 min=79 avg=79
        22: kernel launched 1 time                   (カーネル計算領域に 1 回入った)
            grid: [782]  block: [128]                (グリッドサイズ 782、ブロックサイズ 128)
            device time(us): total=96 max=96 min=96 avg=96 (カーネル実行時間)
            elapsed time(us): total=106 max=106 min=106 avg=106 (経過時間)
        25: data copyout reached 1 time              (デバイスからホストへデータコピー)
            device time(us): total=74 max=74 min=74 avg=74

$ pgcc -fast -Minfo=accel -acc c1.c
$ ./a.out
Accelerator Kernel Timing data
  main  NVIDIA  devicenum=0
    time(us): 397
    23: compute region reached 1 time
        23: data copyin reached 1 time
            device time(us): total=74 max=74 min=74 avg=74
        25: kernel launched 1 time
            grid: [782]  block: [128]
            device time(us): total=252 max=252 min=252 avg=252
            elapsed time(us): total=262 max=262 min=262 avg=262
        28: data copyout reached 1 time
            device time(us): total=71 max=71 min=71 avg=71
```

■ PGI アクセラレータ・プログラミングモデルの既知の制限事項

PGI アクセラレータ・プログラミングモデルを使用して GPU 並列領域内の特性として制約を受ける条件には、次のようなものがあります。

- GPU アクセラレータに処理をオフロード（処理を依頼）するループのネスト（多重ループ）領域は、必ず「rectangular」の形態であること。 特に、triangular ループ、あるいは、以下の例のようなネストした多重ループを成している一方のループの値が、他方のループ・インデックスの上限・下限値を決めるようなループは、サポートされない。これは、NVIDIA GPU のアーキテクチャに依存した制約である。

  例：

  for (j=0; j<n; J++)

  for (i=0; i<j; i++)

{some code}

- GPU アクセラレータに処理をオフロード（処理を依頼）するループの中で配列にアクセスするために使われている「ポインタ」は、C99 言語の 'restrict' 属性を有した宣言を行わなければならない。
  あるいは、safeptr としたプログラムとして構成するか、あるいは、当該アクセラレータ用のループを含む全体のプログラムファイルに対して -Msafeptr オプションを付加してコンパイルすることでも代替できる。ただし、これらのアプローチは、副作用を伴うため、実行結果の検証を行う必要がある。
- 少なくとも、オフロードの対象となるループのいくつかは、同期を伴わない、あるいは、イテレーション間の依存性がない完全なデータ並列の特性を有すること。 こうしたループは、NVIDIA GPU 内のマルチプロセッサ間で処理の分割が可能となる。また、ネスト内の一つ以上のループは、同期処理を必要とするようなベクトル処理を行うループとすることができる。例えば、多くのケース、リダクション演算は OK である。こうしたループは、NVIDIA GPU 内の一つのマルチプロセッサの中の複数のプロセッサによりベクトル処理ができる。また、ネスト内の一つ以上のループは、シーケンシャル実行することもできるが、こうしたループは、一つのスレッド・プロセッサ内で順番に実行される。-e.g. 最内側ループのように 。
- 配列のインデックスを計算する演算（gather あるいは scatter のようなもの）は、避けるべきである。ループ・ネスト内にこうした演算がある場合は、ループの並列化あるいはベクトル化を阻止する「依存性」としてコンパイラが認識する。PGI Accelerator コンパイラの今後のリリースでは、ディレクティブに"independent" 節を導入し、こう言ったループの依存性が存在しないことを明示的に指示できるようにする予定である。この結果、こうしたループも並列化が可能となる。
- GPU アクセラレータに処理をオフロード（処理を依頼）するループの中では、ポインタ演算はできない。

3.2 PGI CUDA Fortran の使用

本項の詳細は、http://www.softek.co.jp/SPG/Pgi/TIPS/opt_cudaF.html に説明しています。

■ PGI CUDA Fortran コンパイル・オプション

PGI CUDA Fortran 構文を含むプログラムのファイル名は、***.cuf と言う名称にします。コンパイラは、この名称のファイルを CUDA Fortran であると認識し、-Mcuda オプションを付けなくてもコンパイルできます。また、***.CUF と cuf サフィックスを大文字とすると、コンパイラは、CUDA Fortran プログラムで、かつ、cpp 形式のプリプロセッシング処理を行うべきプログラムであると認識します。もちろん、Fortran のファイル名のサフィックスを *.f、*.F、*.f90、*.F90 と言った従来の慣習である名称としても良いですが、この場合は、コマンドラインに必ず、オプション -Mcuda を明示的に記述しなければなりません。

● CUDA Fortran ファイル名一例
.cuf、.CUF（CUDA Fortran プログラムファイルであることを明示）
.f、.F、*.f90、*.F90、*.f95、*.F95 （従来の慣習名）

PGI pgfortran、(pgf95 あるいは、pgf90 は同じコンパイラです）コンパイラを使用して、CUDA Fortran プログラムをコンパイルするためのオプションの例を示します。なお、CUDA Fortran は、デフォルトでは F90 以降の「自由記述形式」のファイルと見なしてコンパイルしますので、もし旧 F77 時代の「固定記述形式（7 カラムから実行文と言った形式）」の場合は、必ず、-Mfixed と言うオプションを付けて、コンパイラに指示する必要があります。以下は、pgfortran、(pgf95 あるいは、

pgf90 は同じコンパイラです）を使用した場合の例です。

PGI 16.10 以降、正式に CUDA 8.0 toolkit をサポートしました。Pascal (CC6.0) 用の executable を作成する場合は、CUDA 8.0 ライブラリを必要とする場合がありますので、必ず、-Mcuda=cuda8.0 あるいは、-Mcuda=cc60 のオプションを明示的に付けてコンパイルしてください。PGI 17.1 以降のコンパイラのデフォルトは CUDA 7.5 ライブラリをリンクする形となっております。

● CUDA Fortran コンパイルの一例
pgfortran **test.cuf** （最適化なし）
pgfortran -O2 test.cuf （最適化あり）
pgfortran -O2 -Mfixed test.cuf （最適化あり、ソースは F77 固定記述形式）
pgfortran -O2 **–Mcuda=cuda7.5,Kepler** test.f90
● エミュレーションモード
pgfortran -O2 **-Mcuda=emu** test.cuf

- -Mcuda ： このオプションを指定すると、コンパイラは、一般的な Fortran 構文だけでなく CUDA Fortran 構文を解釈するコンパイラモードとなります。CUDA Fortran プログラムをコンパイルし、必要なライブラリをリンクすします。なお、リンク時においてもこのオプションが必要です。

  -Mcuda[=以下のサブオプション] があります。このサブオプションは、カンマ（,）で区切って複数のものを指定することができます。

サブオプション	機能
emu	エミュレーションモードでコンパイルします。これは、GPU 用のコード生成は行わず、ホスト側でエミュレーション実行可能なコードを生成します。一般に、デバッグ時に使用します。CUDA Fortran の " device code (kernel)" は、ホスト上で実行出来るコードで生成され、ホスト側の pgdbg デバッガを使用できます。
cc10	compute capability 1.0 のコードを生成 (PGI 14.1 以降廃止)
cc11	compute capability 1.1 のコードを生成 (PGI 14.1 以降廃止)
cc12	compute capability 1.2 のコードを生成 (PGI 14.1 以降廃止)
cc13	compute capability 1.3 のコードを生成 (PGI 14.1 以降廃止)
cc1x	compute capability 1.x のコードを生成(PGI 15.1 以降廃止)
cc1+	compute capability 1.x, 2.x, 3.x のコードを生成 (PGI 14.1 以降)、(PGI 15.1 以降廃止)
cc20	compute capability 2.0 のコードを生成 (PGI 10.4 以降) (PGI 14.1 以降廃止)
cc2x	compute capability 2.x のコードを生成 (PGI 10.4 以降)
cc2+	compute capability 2.x, 3.x のコードを生成 (PGI 14.1 以降)
fermi	cc2x と同じ (PGI 13.1 以降)
fermi+	cc2+と同じ (PGI 14.1 以降)
cc30	compute capability 3.0 のコードを生成 (PGI 12.8 以降) (PGI 14.1 以降廃止)
cc35	compute capability 3.5 のコードを生成 (PGI 13.1 以降) (PGI 14.1 以降廃止)
cc3x	compute capability 3.x のコードを生成 (PGI 12.8 以降)
cc3+	compute capability 3.x (=cc3x) 以上のコードを生成 (PGI 14.1

	以降)
kepler	cc3x と同じ (PGI 13.1 以降)
kepler+	cc3+と同じ (PGI 14.1 以降)
cc50	compute capability 5.0 (=cc50) (PGI 15.7 以降)
cc60	compute capability 6.0 (=cc60) (PGI 16.9 以降)
charstring	GPU カーネル内で文字列の使用を制限付きで使用する(PGI 15.1 以降)
cuda2.3 or 2.3	PGI にバンドルされた CUDA toolkit 2.3 バージョンを使用 (PGI 10.4 以降)
cuda3.0 or 3.0	PGI にバンドルされた CUDA toolkit 3.0 バージョンを使用 (PGI 10.4 以降)
cuda3.1 or 3.1	PGI にバンドルされた CUDA toolkit 3.1 バージョンを使用 (PGI 10.8 以降)
cuda3.2 or 3.2	PGI にバンドルされた CUDA toolkit 3.2 バージョンを使用 (PGI 11.0 以降)
cuda4.0 or 4.0	PGI にバンドルされた CUDA toolkit 4.0 バージョンを使用 (PGI 11.6 以降)
cuda4.1 or 4.1	PGI にバンドルされた CUDA toolkit 4.1 バージョンを使用 (PGI 12.2 以降)
cuda4.2 or 4.2	PGI にバンドルされた CUDA toolkit 4.2 バージョンを使用 (PGI 12.6 以降)
cuda5.0 or 5.0	PGI にバンドルされた CUDA toolkit 5.0 バージョンを使用 (PGI 13.1 以降)
cuda5.5 or 5.5	PGI にバンドルされた CUDA toolkit 5.5 バージョンを使用 (PGI 13.9 以降)
cuda6.0 or 6.0	PGI にバンドルされた CUDA toolkit 6.0 バージョンを使用 (PGI 14.4 以降)
cuda6.5 or 6.5	PGI にバンドルされた CUDA toolkit 6.5 バージョンを使用 (PGI 14.9 以降)
cuda7.0 or 7.0	PGI にバンドルされた CUDA toolkit 7.0 バージョンを使用 (PGI 15.4 以降)
cuda7.5 or 7.5	PGI にバンドルされた CUDA toolkit 7.5 バージョンを使用 (PGI 15.9 以降)
Cuda8.0 or 8.0	PGI にバンドルされた CUDA toolkit 8.0 バージョンを使用 (PGI 15.9 以降)
fastmath	fast math ライブラリを使用 (PGI 10.4 以降)
[no]flushz	GPU 上の浮動小数点演算の flush-to-zero モードを制御。デフォルトは noflushz。(PGI 11.5 以降)
keepbin	kernel バイナリファイルを保持し、ファイル(.bin)として出力する
keepgpu	kernel ソースファイルを保持し、ファイル(.gpu)として出力する
keepptx	GPU コードのための portable assembly(.ptx)ファイルを保持し、ファイルとして出力する
[no]lineinfo	GPU line information を生成する(PGI 15.1 以降)
[no]llvm	llvm ベースのバックエンドを使用してコードを生成する。(PGI 15.1 以降) 64-bit 上では LLVM バックエンドを使う [使わない]。なお、PGI 15.1 以降はデフォルト llvm を使うように変更された。リンク時にエラーが生じる場合、nollvm を試すことをお勧めする。

maxregcount:n	GPU 上で使用するレジスタの最大数を指定。ブランクの場合は、制約が無いと解釈する
nofma	fused-multiply-add 命令を生成しない
noL1 noL1cache	グローバル変数をキャッシュするためのハードウェア L1 データキャッシュの使用を抑止する (PGI 13.10 以降)
loadcache:L1 loadcache:L2	グローバル変数をキャッシュするためにハードウェア L1 あるいは L2 データキャッシュを使用する。但し、アーキテクチャ上、有効とならない GPU がある (PGI 14.4 以降)
ptxinfo	コンパイル時に PTXAS 情報メッセージを表示する(PGI 11.0 以降)
[no]rdc	Fortran Module 内の device routine など、異なるファイルに配置されたデバイスルーチンをそれぞれ分割コンパイルし、リンクが出来るようにする。CUDA 5.0 以降の機能を使用します。(PGI 13.1 以降 + CUDA 5.0 以降) (PGI 14.1 以降デフォルト)
[no]unroll	自動的に最内側ループのアンローリングを行う (default at -O3) (PGI 14.9 以降)

（注意） -Mcuda=emu によるエミュレーションモードでの実行は、実際の GPU 上で実行する状況を完全に再現できないことに注意して下さい。特に、エミュレーションモードでは、一時に「シングル」スレッド・ブロックしか実行できません。この状態では、メモリアクセス（転送）時に生じるメモリレース等によるエラーは再現できません。また、ホスト CPU 側の浮動小数点演算器と内部組込関数が使用されるため、GPU デバイス上の演算器等を使用した結果に較べ、若干の数値的差異が起こりえます。

■ PGI CUDA Fortran プログラム例

CUDA Fortran の簡単なプログラム例とコンパイル・実行例を示します。以下のプログラムは、デバイス上の配列(int_d)を宣言して、GPU 上で整数配列に値を入れ、ホスト側の配列に戻して印字すると言う単純なものです。しかし、CUDA Fortran 構文の基本的なものが使用されている例です。

```
module cudamod
    use cudafor
    implicit none

    contains
    !   kernel subprogram
    attributes(global) subroutine test1 (intdat,n)
        integer :: it, ib
        integer, value :: n
        integer, device :: intdat(n)
        !---------------------------
        it = threadidx%x
        iib = (blockidx%x-1) * 16
        intdat(it+ib) = it + ib
        !---------------------------
    end subroutine test1
end module cudamod

program cuda_device
```

```
    use cudafor
    use cudamod
    implicit none
    integer, parameter :: n=64
    integer :: int_h(n), ist
    ! デバイス側の配列データを allocatable で宣言
    integer, device, allocatable, dimension(:)  :: int_d

    int_h = 0
    allocate(int_d(n))

    ! カーネル（デバイスプログラム）を起動する
    call test1 <<<n/16,16>>> (int_d, n) !pass arguments
    ist = cudathreadsynchronize()  ! 同期ポイント

    ! デバイス側の配列データをホスト側に戻す
    int_h = int_d
    print *,'int_h = ',int_h
    deallocate(int_d)
end program cuda_device
```

▸ コンパイル&実行

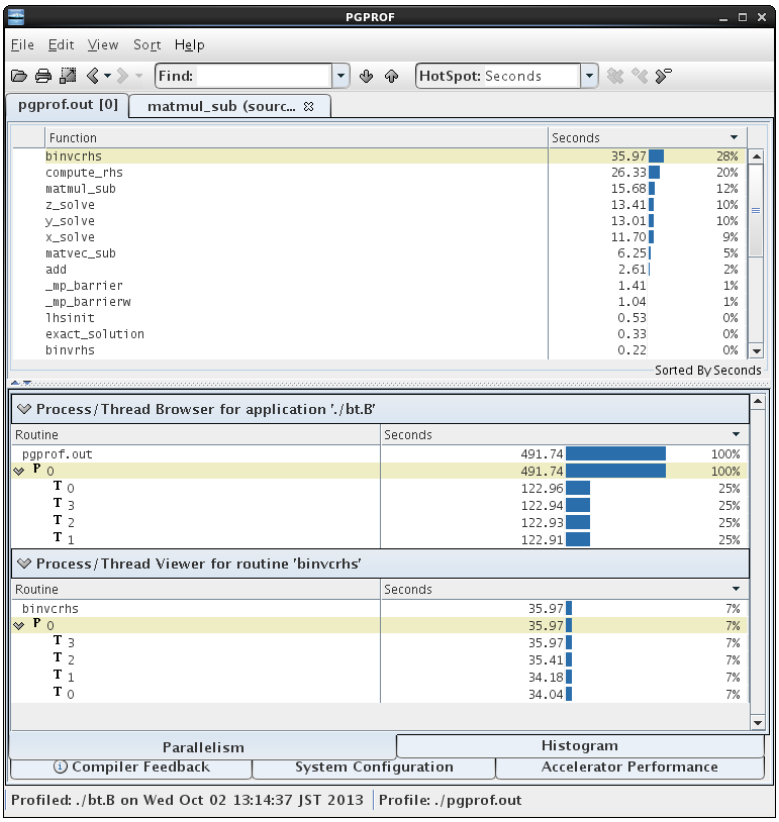
```
$ pgfortran -O2 -Minfo -Mcuda -o test.exe test.cuf
cuda_device:
      28, Memory zero idiom, loop replaced by call to __c_mzero4
$ test.exe
 int_h =           1           2           3           4           5
           6           7           8           9          10          11
          12          13          14          15          16          17
          18          19          20          21          22          23
          24          25          26          27          28          29
          30          31          32          33          34          35
          36          37          38          39          40          41
          42          43          44          45          46          47
          48          49          50          51          52          53
          54          55          56          57          58          59
          60          61          62          63          64
```

4 PGI ツール（プロファイラ、デバッガ）の使用

PGI 社の並列化アプリケーション開発支援ツールとして、並列デバッギング・ツールと並列プロファイリング・ツールを提供します。これらのユーティリティは、全ての PGI Workstation 製品、PGI Server 製品、PGI CDK 製品に無償でバンドルされています。

■ PGPROF 性能解析プロファイラ（OpenMP and MPI）

PGPROF は、Linux/Windows/OS X システム上で対話型によって簡単に使用できる、シリアル実行プログラムあるいは OpenMP スレッドプログラム/MPI マルチプロセスのための性能解析ユーティリティです。 これは、32 ビットあるいは 64 ビットのマルチコア・プロセッサを搭載する 1CPU システムあるいは SMP システム上で動作し、OpenMP のスレッドプログラム・MPI プログラムのプロファイリング機能を提供します。プロファイリング機能は、関数・サブルーチンレベルだけではなく、ソースコードレベルにおいても提供します。OpenMP 対応のプロファイル機能は、PGI Workstation/PGI Server 製品（ローカルな 1 台のノード上での MPI プロセスのプロファイリングが可能）、PGI CDK 製品（リモートノードを含めたクラスタ上の MPI プロセスのプロファイリングが可能）に付属します。また、PGI Workstation/Server Windows 版では、Microsoft(R) Windows 上で実行する MS-MPI 並列プログラムのプロファイリング機能を提供します。

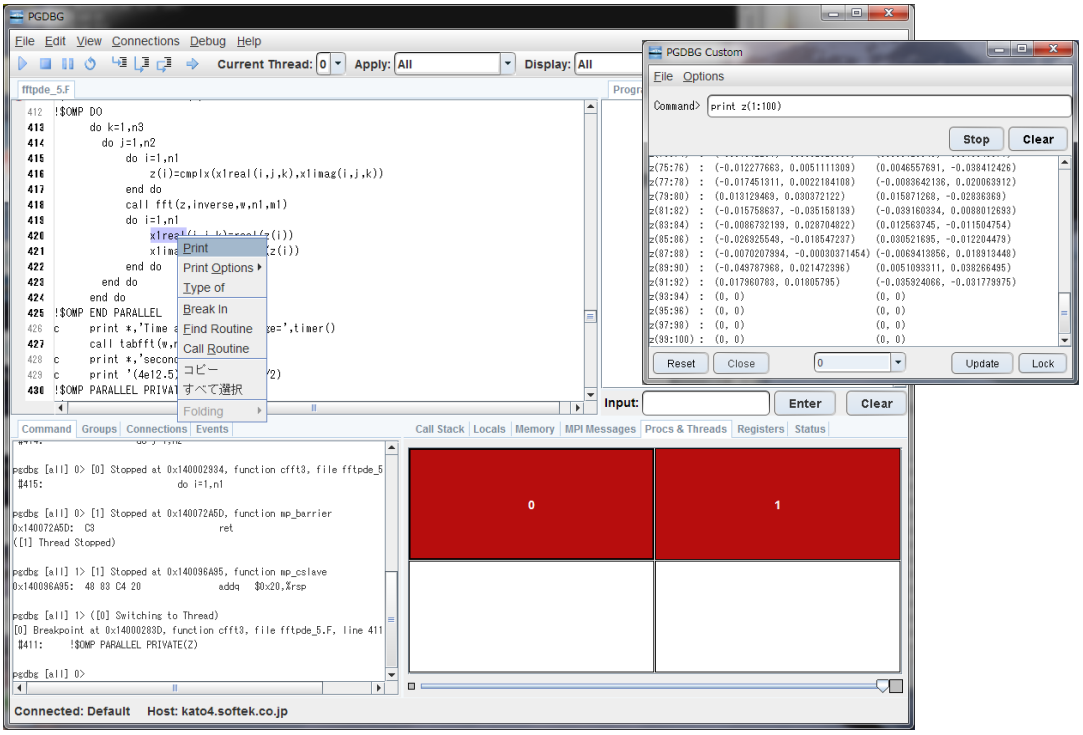


GUI ベースのプロファイリング表示

■ PGDBG シンボリック・デバッガ（OpenMP and MPI）

PGDBG は、Linux/Windows/OS X システム上で、シリアル実行プログラムあるいは OpenMP スレッドプログラム・MPI プロセスのデバッグを行うためのユーティリティです。これ

は、32 ビットあるいは 64 ビットのマルチコアプロセッサを搭載するマルチコア CPU システム上で動作し、OpenMP/MPI のプログラムのシンボリック・デバッグ機能を提供します。OpenMP 対応のデバッガは、PGI Workstation/PGI Server/PGI CDK 製品に付属します。また、MPI プロセス対応のデバッギング機能も、 PGI Workstation/PGI Server 製品（ローカルな 1 台のノード上での MPI プロセスのデバッグが可能）、PGI CDK 製品（リモートノードを含めたクラスタ上の MPI プロセスのデバッグが可能）に付属します。また、 PGI Workstation/Server Windows 版では、Microsoft(R) Windows 上で実行する MS-MPI 並列プログラムのデバッグ機能を提供します。



GUI ベースのデバッギング作業画面

4.1 PGI 性能解析プロファイラ（PGPROF）の使用

PGI 製品における F77, F2003, C, C++ のコンパイラを使用して、性能解析プロファイラ PGPROF を使用する場合のコンパイル・オプションの例を示します。以下は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。なお、プロファイラとは、実行時間のボトルネック等を調べるためのツールであり、各ルーチン単位、ステートメント単位での実行時間等を分析することができます。

以下に説明する機能は、PGI 15.10 リリースまで有効であったものです。PGI 16.1 以降は、GPU プロファイリングに対応した OpenMP/OpenACC 用のプロファイラとして、全く異なるプロファイラ機能を提供します。詳細は、ドキュメントサイト上の「PGPROF® Quick Start Guide」をご覧ください。

「プロファイラ PGPROF の使用方法」に関しては、以下の弊社ページも参考にしてください。

https://www.softek.co.jp/SPG/Pgi/TIPS/opt_prof.html

■ プロファイルを行うためのコンパイル・オプションの設定

```
pgfortran -Mprof -fastsse -Minfo test.f
pgfortran -Mprof={func|lines} -fastsse –Minfo test.f
```

gprof スタイルのプロファイルデータ(**gmon.out**) の取得の場合は、

```
pgfortran -pg -fastsse -Minfo test.f
```

- **-Mprof** オプションは、性能解析プロファイラ PGPROF を使用するための実行時間データをサンプルするための機構を実行モジュールに実装するためのオプションです。
- **-Mprof={func|lines}** のフラグを指定することができます。何も指定しない場合のデフォルトは、func です。func は関数・サブルーチンレベルのプロファイリングを行う場合のもので、一方 lines は statement 単位でのプロファイリングが可能となります。 その他のサブオプションもありますので、詳細は、PGI Profiler User Guide を参照してください。
- OpenMP でのスレッド単位のプロファイリングでは、-Mprof=func の関数レベルのオプションでは、並列スレッドのプロファイルは採取できません。データ取得単位の小さな statement 単位の -Mprof=lines を指定してください。
- これで生成された実行モジュールを実行すると、実行後、pgprof.out と言うファイルが生成されます。PGPROF ユーティリティはこのファイルを使用してプロファイリング結果を表示します。


上記のオプションはコンパイル時だけでなく、リンケージ時においても指定する必要があります。特に Makefile 等で、コンパイルフェーズとリンク・フェーズを分けて行う場合は、リンケージのオプションにも同じように指定してください。

既知の制約事項

- **-Mprof=func** 並びに**-mcmodel=medium** と **-mp** を一緒に使用した場合、生成された実行モジュールはセグメンテーション・フォールトが生じます。これらのオプションを同時に使用することは避けてください。

■ pgcollect プロファイリング・ユーティリティ

pgcollect と言うユーティリティを使用してプロファイリングを行う方法について、以下の URL で説明しております。手軽にプロファイリングが可能なため、pgcollect ユーティリティを使用したプロファイリングを推奨します。詳細は、以下の Web ページをご覧ください。

ソフテック技術コラム「プログラムのプロファイルを取得する」
http://www.softek.co.jp/SPG/Pgi/TIPS/public/accel/gpu-accel1.html

■ プロファイラ PGPROF を使用する

PGPROF は X-window 使用して GUI ベースで分析が可能です。そのための、環境変数等の準備を行います。もし、X-window を利用しない場合は、コマンド・モード（-text オプション）での使用が可能です。

PGPROF を動かすシステム上のユーザのログインシェル（.bashrc or .cshrc 等）に
X を表示するマシン名 **DISPLAY** 環境変数を定義します。
（例）
$ **DISPLAY**=mypc.softek.co.jp:0.0 ; export DISPLAY

表示するクライアント（X Server 側）では、X-window を立ち上げ、
外部マシンからのプロトコルを受け入れるためのコマンドを発行します。

$ **xhost +**

X-window の Server/Client 方式を取らず、同一のマシンで行う場合は、上記の操作を同一のマシン上で行ってください。

プログラムを実行した directory へ移動して、実行後生成された pgprof.out が
存在することを確認します。次のコマンドで、pgprof を立ち上げます。

$ **pgprof pgprof.out**

これで、X-window 上に プロファイルの状況が表示されます。

■ プロファイラ結果を解析する

OpenMP でコーディングされた FFT プログラムを使用して、プロファイルの取得からその解析の様子をデモンストレーションしてみましょう。GUI ベースで行う際は、視覚的に様々な情報が見ることができますが、ここでは、コマンドベースで pgprof を使用した際の操作の様子を示します。なお、使用した並列数は、4 スレッドの場合の結果を示します。コマンド入力の様子を追跡してください。

```
$ ls
fftpde_5.F

$ pgf90 -fastsse -Mprof=func -Minfo -mp fftpde_5.F (プロファイル用オプション)
$ export OMP_NUM_THREADS=4   (4 スレッド計算)
$ time a.out    （実行）
FORTRAN STOP

real     0m1.298s
user     0m1.109s
sys      0m0.018s
$ ls
a.out   fftpde_5.F   pgprof.out    （pgprof.out が生成された）

$ unset DISPLAY   （DISPLAY 環境変数を解除）

$ pgprof –text pgprof.out
PGPROF 15.5-0
```

```
The Portland Group - PGI Compilers and Tools
Copyright (c) 2015, NVIDIA CORPORATION.  All rights reserved.

Datafile  : pgprof.out
Processes : 1
Threads   : 4
```

pgprof> p　（まず、情報の出力 **p** は print）

```
Profile output - Thu Jun 04 16:39:53 JST 2015
Program                          : /home/kato/PGI/FFT/fftpde/STEPS/a.out
Datafile                         : pgprof.out
Process                          : 0
Total Time for Process           : 0.308520 secs
Sort by max time
Select all
                  Routine      Source  Line
  Calls  Time(%)    Name        File   No.

      7       38   cfft3  fftpde_5.F   363
229,376       30     fft  fftpde_5.F   481
      1       24  fftpde  fftpde_5.F     1
    129        8  vranlc  fftpde_5.F   283
     21        0  tabfft  fftpde_5.F   550
      2        0   timer  fftpde_5.F   563
    465        0  randlc  fftpde_5.F   209
```

pgprof> time raw（時間を%表示から実時間表示へ）
pgprof> d cost　（cost も表示する、**d** は display）
pgprof> p　（この状態で情報出力、**p** は print ）

```
Profile output - Thu Jun 04 16:39:53 JST 2015
Program                          : /home/kato/PGI/FFT/fftpde/STEPS/a.out
Datafile                         : pgprof.out
Process                          : 0
Total Time for Process           : 0.308520 secs
Sort by max time
Select all
                            Routine      Source  Line
  Calls   Time(s)   Cost(s)    Name        File   No.

      7  0.116323  0.207456   cfft3  fftpde_5.F   363
229,376  0.091508  0.364899     fft  fftpde_5.F   481
      1  0.075138  0.30852   fftpde  fftpde_5.F     1
    129  0.026071  0.103467  vranlc  fftpde_5.F   283
     21  0.00007   0.00007   tabfft  fftpde_5.F   550
      2  0.000043  0.000043   timer  fftpde_5.F   563
    465  0.000038  0.000073  randlc  fftpde_5.F   209
```

pgprof> s thread　（S は統計情報の出力選択、各スレッド情報も出力する）
pgprof> p　（この状態で情報出力）

```
Profile output - Thu Jun 04 16:39:53 JST 2015
Program                          : /home/kato/PGI/FFT/fftpde/STEPS/a.out
Datafile                         : pgprof.out
Process                          : 0
```

```
Total Time for Process            : 0.308520 secs
Sort by max time
Select all
```

（スレッド毎のプロファイル統計データ）
並列化がきれい に行われていることがわかる

Calls	Time(s)	Cost(s)	Routine Name	Source File	Line No.
7	0.116323	0.207456	cfft3	fftpde_5.F	363
7	0.116323	0.207456	cfft3	fftpde_5.F	363
7	0.116323	0.207456	cfft3	fftpde_5.F	363
7	0.116323	0.207456	cfft3	fftpde_5.F	363
57,344	0.091063	0.091063	fft	fftpde_5.F	481
57,344	0.091063	0.091063	fft	fftpde_5.F	481
57,344	0.091063	0.091063	fft	fftpde_5.F	481
57,344	0.091063	0.091063	fft	fftpde_5.F	481
1	0.075138	0.30852	fftpde	fftpde_5.F	1
1	0.075138	0.30852	fftpde	fftpde_5.F	1
1	0.075138	0.30852	fftpde	fftpde_5.F	1
1	0.075138	0.30852	fftpde	fftpde_5.F	1
33	0.025845	0.025845	vranlc	fftpde_5.F	283
33	0.025845	0.025845	vranlc	fftpde_5.F	283
33	0.025845	0.025845	vranlc	fftpde_5.F	283
33	0.025845	0.025845	vranlc	fftpde_5.F	283
21	0.00007	0.00007	tabfft	fftpde_5.F	550
21	0.00007	0.00007	tabfft	fftpde_5.F	550
21	0.00007	0.00007	tabfft	fftpde_5.F	550
21	0.00007	0.00007	tabfft	fftpde_5.F	550
2	0.000043	0.000043	timer	fftpde_5.F	563
2	0.000043	0.000043	timer	fftpde_5.F	563
2	0.000043	0.000043	timer	fftpde_5.F	563
2	0.000043	0.000043	timer	fftpde_5.F	563
81	0.000038	0.000038	randlc	fftpde_5.F	209
81	0.000038	0.000038	randlc	fftpde_5.F	209
81	0.000038	0.000038	randlc	fftpde_5.F	209
81	0.000038	0.000038	randlc	fftpde_5.F	209
1	0.075138	0.30852	fftpde	fftpde_5.F	1
465	0.000038	0.000073	randlc	fftpde_5.F	209
2	0.000043	0.000043	timer	fftpde_5.F	563
21	0.00007	0.00007	tabfft	fftpde_5.F	550
7	0.116323	0.207456	cfft3	fftpde_5.F	363
129	0.026071	0.103467	vranlc	fftpde_5.F	283
229,376	0.091508	0.364899	fft	fftpde_5.F	481
21	0.000053	0.000053	tabfft	fftpde_5.F	550
2	0.000032	0.000032	timer	fftpde_5.F	563

pgprof> quit（終了）

■ GNU 形式のプロファイルデータ (gmon.out) を取得する方法　（Linux のみ）

PGI の pgprof ユーティリティを使用しないで、GNU 形式のプロファイルデータを取得する方法を説明します。PGI コンパイラには、GNU 互換のプロファイルデータを取得するために、**-pg オプション**が用意されています。これを付けて生成された実行モジュールは、実行後、GNU 互換のプロファイル結果データを生成します。プロファイルデータは、GNU のデフォルトである gmon.out フ

ァイルに記録されます。以下のようにオプションを設定し、実行してください。

```
$ pgf90 -fastsse -Minfo -mcmodel=medium -pg -o a.out (source files)
$ a.out （実行）
```

実行後、**gmon.out** が生成されます。
次に、GNU ユーティリティの一つである、**gprof** を使用してプロファイル結果を表示します。

プロファイルを表示するためのコマンド列は、以下の例のようになります。

```
$ gprof a.out gmon.out
```

注意する点は、GNU 形式のプロファイル結果は、各ルーチンの実行時間に関して、実時間表示ではありません。プロファイリングデータを「サンプリング形式」で取得しているためです。一方、PGI のプロファイルは実時間を表示します。

gprof の出力例を以下に示します。各ルーチンの実行回数と実行時間等が現れますので、少なくともどのルーチンが時間を消費しているかは理解できるはずです。また、CALL Graph も出力できますので、別の利用用途としても利用可能です。

```
$ gprof a.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 41.35      0.43     0.43   229262    0.00     0.00  fft_
 35.58      0.80     0.37        7    0.05     0.11  cfft3_
 18.27      0.99     0.19        1    0.19     1.04  MAIN_
  4.81      1.04     0.05      129    0.00     0.00  vranlc_
  0.00      1.04     0.00      465    0.00     0.00  randlc_
  0.00      1.04     0.00       21    0.00     0.00  tabfft_
  0.00      1.04     0.00        2    0.00     0.00  timer_

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
```

```
            function is profiled, else blank.

name        the name of the function.  This is the minor sort
            for this listing. The index shows the location of
            the function in the gprof listing. If the index is
            in parenthesis it shows where it would appear in
            the gprof listing if it were to be printed.

                        Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.96% of 1.04 seconds

index % time    self  children    called     name
                0.19    0.85       1/1           main [2]
[1]    100.0    0.19    0.85       1         MAIN_ [1]
                0.37    0.43       7/7           cfft3_ [3]
                0.05    0.00     129/129         vranlc_ [5]
                0.00    0.00     465/465         randlc_ [6]
                0.00    0.00       2/2           timer_ [8]
-----------------------------------------------
                                                <spontaneous>
[2]    100.0    0.00    1.04                  main [2]
                0.19    0.85       1/1           MAIN_ [1]
-----------------------------------------------
                0.37    0.43       7/7           MAIN_ [1]
[3]     76.9    0.37    0.43       7         cfft3_ [3]
                0.43    0.00  229262/229262      fft_ [4]
                0.00    0.00      21/21          tabfft_ [7]
-----------------------------------------------
                0.43    0.00  229262/229262      cfft3_ [3]
[4]     41.3    0.43    0.00  229262          fft_ [4]
-----------------------------------------------
                0.05    0.00     129/129         MAIN_ [1]
[5]      4.8    0.05    0.00     129          vranlc_ [5]
-----------------------------------------------
                0.00    0.00     465/465         MAIN_ [1]
[6]      0.0    0.00    0.00     465          randlc_ [6]
-----------------------------------------------
                0.00    0.00      21/21          cfft3_ [3]
[7]      0.0    0.00    0.00      21          tabfft_ [7]
-----------------------------------------------
                0.00    0.00       2/2           MAIN_ [1]
[8]      0.0    0.00    0.00       2          timer_ [8]
-----------------------------------------------
```

4.2 PGI デバッガ（PGDBUG）の使用

PGI 製品における F77, F2003, C, C++ のコンパイラを使用して、シンボリック・デバッガ PGDBG を使用する場合のコンパイル・オプションの例を示します。以下は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。

■ デバッガを使用するためのシンボリック・デバッグ情報を生成するオプション

```
$ pgfortran -g test.f
$ pgfortran -gopt test.f
```

- **-g** オプションは、シンボリック・デバッグ情報をオブジェクトファイル上に生成するためのものです。これによって、シンボリック情報が含まれた PGDBG あるいは Etnus TotalView 等のデバッガを使用する際の実行モジュールが生成されます。
- **-g** オプションを指定すると、最適化レベルのオプション -O のレベル値が 0 になります。
- **-gopt** オプションは、デバッグ情報を生成する-g オプションの機能を変更するものです。一般に、−g オプションでは、デフォルトで最適化レベルを下げて実行モジュールが生成されますが、本オプションは、最適化されたコードのデバッギングが可能とするような方法でモジュールが生成されます。-gopt はシンボリック・デバッグ情報をオブジェクトモジュールの中に付加し、さらに、-g が指定されない時と同じ最適化コードを生成するように、コンパイラに対して指示するものです。(PGI 6.1 以降)　但し、デバッギングの本来の形は、-O0 で行うことですので、-gopt で問題がある場合は、-g –O0 のオプションレベルでデバッグを行ってください。


PGI 5.1 以降のシンボリック・デバッグ情報のフォーマットは、新しい共通フォーマットである DWARF2 フォーマットで生成されています。他のデバッガでも利用できる互換性の高いフォーマットです。PGI コンパイラは旧フォーマットである DWARF1 でもデバッグ情報を生成可能です。この場合は、pgf90 -g -Mdwarf1 と言うオプション設定を行います。

■ デバッガ PGDBG を使用する

PGDBG は X-window 使用して GUI ベースで分析が可能です。そのための、環境変数等の準備を行います。もし、X-window を利用しない場合は、コマンド・モードでの使用が可能です。

PGDBG を動かすシステム上のユーザのログインシェル（.bashrc or .cshrc 等）に
X を表示するマシン名 **DISPLAY** 環境変数を定義します。

（例）
$ **DISPLAY**=mypc.softek.co.jp:0.0 ; export DISPLAY

表示するクライアント（X Server 側）では、X-window を立ち上げ、外部マシンから
のプロトコルを受け入れるためのコマンドを発行します。

$ **xhost +**

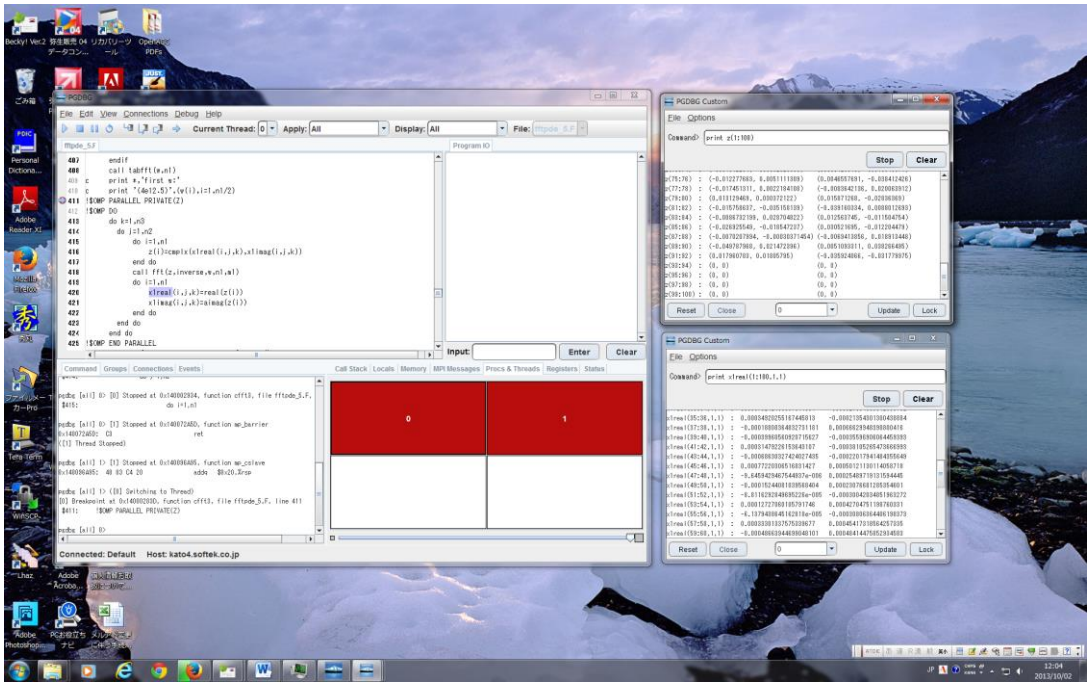

X-window の Server/Client 方式を取らず、同一のマシンで行う場合は、上記の操作を同一のマシン上で行ってください。

デバッグしたい実行モジュールの存在する directory へ移動して、
次のコマンドで、pgdbg を立ち上げます。

$ pgdbg test.exe

これで、X-window 上に デバッガの GUI が表示されます。

PGDBG の操作等は、PGI TOOL'S Guide をご参考にしてください。本稿では割愛します。



GUI ベースのマルチスレッド PGDBG デバッギングの状況

5　例題によるチュートリアル

本章では、例題を提示し実際に PGI コンパイラを適用する方法等に関して説明します。

5.1 行列積のプログラムを例に PGI コンパイラを使用する

KEYWORD	： プリプロセス処理、行列積性能、AMD ACML ライブラリ、DGEMM
	： DGEMM ベンチマーク、OpenMP 並列化、F90 組込み関数 MATMUL
	： 時間計測方法

ここでは、行列積の計算を行うためのプログラムを例示し、PGI コンパイラの使用法、ACML ライブラリの使用法、行列積計算の性能等に関して説明します。ここで、例示したプログラムは、行列積の処理プログラムを以下の三つの方法で記述したものです。

① 一般的な Fortran プログラムでコーディング例（シリアル版と OpenMP 並列版）
② F90 の組込み関数（MATMUL）を使用した例
③ AMD ACML ライブラリを使用する例（シリアル版と OpenMP 並列版）

本プログラムは、Fortran 上でプリプロセス・ディレクティブを組込み、それぞれの機能プログラムがコンパイル・動作できるようになっています。ここでは、これらを実現するために、一連の処理方法とその実行性能について説明します。ソースファイルは、以下のものを使用します。なお、このファイルは弊社のホームページ上からダウンロード可能です。
（http://www.softek.co.jp/SPG/Pgi/TIPS/tutorial/matrix.html）

使用ソースファイル　：　matmul.F　（Fortran90 ベース）
時間計測ルーチン　　：　wallclock.c（下記で述べる second() 関数）
（Elapsed time ベース：並列処理時の時間計測に便利、CPU 時間ベースではない）

■ Fortran でのプリプロセス処理

C 言語でのプリプロセス処理は、一般的に使用される方法ですが、Fortran ベースにおいても使用できます。基本的な構文は、C 言語対応のものに倣いますが、よく使用されるものは #IF DEFINED 句であり、コンパイル・オプションで -D に定義された「フレーズ」をもとに真偽を判定し、翻訳すべきソースファイルを生成します。以下の例では、その選択肢のフレーズとして、「MATMUL」、「BLAS」、「指定しない」のパターンで判断され、プリプロセス後のソースが生成されます。

【プリプロセス処理を行うコンパイル・オプション】

プリプロセス処理を明示的にコンパイラに知らせるためには、ソースファイル名の接尾子を .f (.f90, f95) 形式ではなく、 **.F**（.F90, F95）大文字形式に変更します。この形式のファイル名は、全てプリプロセス処理の対象のファイルとしてコンパイラは認識します。さらに「フレーズ」を指定するには、 C 言語系と同じように、**-D** オプションに続く文字列で指定します。

```
$ pgfortran -fastsse -Minfo -DBLAS matmul.F
```

```
#if defined (MATMUL)          (-DMATMUL が指定された場合、以下の文を採用)
! F90 Intrinsic function (MATMUL)
        c = matmul(a, b)

#elif defined (BLAS)          (-DBLAS が指定された場合、以下の文を採用)
! AMD ACML(LAPACK) Library
        call dgemm('N','N', m, p, n, 1.0d0, a, m, b, n, 0.0d0, c, n)

#else                         (-D を指定しない場合、以下の文を採用)
! Source coding
!$omp parallel default(shared)
!$omp do   scedule(runtime)
        do j = 1, p
           do i = 1, m
              c(i,j) = 0.0
           enddo
        enddo
        do i = 1, m
!$omp do   scedule(runtime)
           do ii = 1, n
              arow(ii) = a(i,ii)
           enddo
!$omp do   scedule(runtime)
           do j = 1, p
              do k = 1, n
                 c(i,j) = c(i,j) + arow(k) * b(k,j)
              enddo
           enddo
        enddo
!$omp end parallel

#endif
```

ソースファイルのプリプロセス処理後の編集された中間ソースファイルを作成するためには、以下のオプション（-E あるいは-F）も指定します。

```
$ pgfortran -fastsse -Minfo -DBLAS -E matmul.F
   （プリプロセス処理後、編集された内容が標準出力に出力される）
$ pgfortran -fastsse -Minfo -DBLAS -F matmul.F
   （プリプロセス処理後、編集された内容が matmul.f と言うファイルに出力される）
```

■ Elapsed time （経過時間）を測定する関数

プログラムの中のある部分の実行時間を測定する場合、その対象部分の前後に時間計測ルーチンを挟んで、経過時間を測定しなければならないことがあります。ここでは、よく使用される時間関数について、説明します。一般に、1 CPU（シリアル）計算の実行時間を測定する場合には、CPU 時間測定の関数を使用しますが、並列処理の時間計測においては、複数の CPU が消費した CPU 時間の総和ではなく、実際の経過時間を測り、性能向上の度合いを確かめる必要があります。並列処理性能は、経過時間（Elapsed time）をもとに判断する必要があるため、ここでは、その関数の使用法を説明します。この方法は、1 CPU 計算（シリアル計算）における時間計測でも使用できます。

ここで説明する時間計測の方法は、以下の三つです。その使用法については、matmul.F ファイルの内容を見てお確かめください。

表 5 – 1　経過時間を測定する時間関数

	関数名	時間精度	使用可能言語	対応 OS	説明
①	second	マイクロ秒	F77/F90/C/C++	Linux	システム時間関数 (gettimeofday) 使用
②	system_clock	ミリ秒	F90	Linux/Windows	Fortran90 の組込み関数
③	omp_get_wtime	マイクロ秒	F77/F90/C/C++	Linux/Windows	OpenMPの組込み関数

① matmul.F 上では、-D マクロを指定しない場合使用される （wallclcok.c を参照）
② matmul.F 上では、-DSYSCLK を指定する場合使用される
③ matmul.F 上では、-DOMPTIME を指定する場合使用される（PGI 6.1 以降で使用可能）

なお、FORTRAN77 のプログラムでは、②は使用できないため、①あるいは③をお利用ください。例題の matmul.F 上では、以下のようなコーディングを行い、使用する時間関数のマクロ名を指定して選択している。

```
#if defined (SYSCLK)
! Elapsed time : Using F90 system_clock      : Milli-second
      call system_clock(count=clock0)

#elif defined (OMPTIME)                      : Micro-second
! Elapsed time : OpenMP build-in function
      time0 = omp_get_wtime()
#else
! Elapsed time : SECOND() C function         : Micro-second
      time0 = second()
#endif
```

● second() 関数を使用する際のプログラム上の設定

```
real*8 second                    second() 関数名も倍精度宣言
real*8 time0, time1, time2       一時的に時間を保存する変数も倍精度
real*8 t, walltime

time0=second()                   この時点での経過時間を取得

  {計算ブロック}

time1=second()                   この時点での経過時間を取得
walltime = (time1 - time0)       経過時間（秒単位）
```

second() 関数を含む wallclock.c の C 言語ファイルのオブジェクトを Fortran と結合するには、以下のコマンドで行います。PGCC コンパイラも使用できる環境であれば、pgfortran のコマンドライン上に C ファイルを指定すれば自動的に C のコンパイルも行います。

```
$ pgfortran -fastsse -Minfo matmul.F wallclcok.c
```

もし、Fortran の言語コンパイラしかない場合は、 gcc を使用してコンパイルした後、pgfortran コマンドでリンクします。

```
$ gcc -c wallclock.c
$ pgfortran -fastsse -Minfo matmul.F wallclock.o
```

● system_clock 関数を使用する際のプログラム上の設定

```
integer hz, clock0, clock1, clock2        整数宣言（Integer*4 のみ）
real*8 t, walltime

call system_clock(count_rate=hz)    時間採取回数＝時間精度（1000）
print *, "system_clock resolution: ", real(1.d0/hz)

call system_clock(count=clock0)       この時点でのクロックを取得

  {計算ブロック}

call system_clock(count=clock1)       この時点でのクロックを取得
t = (clock1 - clock0)
walltime = real(t) / real(hz)                 経過時間（秒単位）
```

● OpenMP の omp_get_wtime 関数を使用する際のプログラム上の設定

```
real*8 omp_get_wtime, omp_get_wtick   OpenMP 関数名を倍精度宣言

print *, "OMP_Wtime resolution: ", omp_get_wtick()   時間関数の精度を表示

time0 = omp_get_wtime()                  この時点での経過時間を取得

  {計算ブロック}

time1 = omp_get_wtime()                  この時点での経過時間を取得
walltime = (time1 - time0)                    経過時間（秒単位）
```

■ 行列積計算の三つの方法

ここでは、行列積(1000 x 1000)を計算するための方法として、以下の三つの方法を採用した場合の実行性能を比較した結果を示します。題材 matmul.F プログラムを PGI 15.5 でコンパイルして得られた性能結果を 表 5-2 に示しました。また、各計算の実際のコマンド・ログは以下に示しました。

表 5-2　行列積の性能(1000 x 1000)

計算手法		-D マクロ	1コア性能 (MFLOPS)	4コア性能 (MFLOPS)	実効性能 対ピーク性能
①	Fortran プログラムでコーディング（OpenMP）	指定しない	3015	7663	18%
②	F90組込み関数を使用する（1core 計算のみ）	MATMUL を指定	5265	–	13%
③	ACMLライブラリ LAPACK の使用	BLAS を指定	10784	36642	88%

表 **5-3** 使用システム**(64-bit Intel Core i7** の仕様**)**

プロセッサ	Intel(R) Core(TM) i7 CPU 920 (1プロセッサ/4コア)
Clock	2.6 GHz
最大浮動小数点性能(ピーク性能)	41.6 GFLOPS
L2 cache	8MB
使用 OS	CentOS 6.5
使用コンパイラ	PGI 15.5

● 行列積計算を OpenMP ソースコーディングした場合の性能

行列積の計算を Fortran ソースコーディングしたものを OpenMP で並列化します。その際の 1 スレッド計算、4 スレッド計算時の性能を求めます。

【ソース内容】

```
! Source coding
!$omp parallel default(shared)
!$omp do scedule(runtime)
          do j = 1, p
             do i = 1, m
                c(i,j) = 0.0
             enddo
          enddo
          do i = 1, m
!$omp do schedule(runtime)
             do ii = 1, n
                arow(ii) = a(i,ii)
             enddo
!$omp do scedule(runtime)
             do j = 1, p
                do k = 1, n
                   c(i,j) = c(i,j) + arow(k) * b(k,j)
                enddo
             enddo
          enddo
!$omp end parallel

-----------------------------------------------------------------
$ pgf90 -fastsse -Minfo -mp matmul.F wallclock.o
matmul.F:
matmul_time:
     29, Loop interchange produces reordered loop nest: 30,29
         2 loops fused
         Generated vector sse code for the loop
     30, Loop not fused: function call before adjacent loop
         2 loops fused
     34, Loop interchange produces reordered loop nest: 35,34
     52, Loop not vectorized/parallelized: too deeply nested
```

```
     64, Parallel region activated
     66, Parallel loop activated with static block schedule
     67, Memory zero idiom, loop replaced by call to __c_mzero8
     71, Barrier
         Loop not vectorized/parallelized: too deeply nested
     73, Parallel loop activated with static block schedule
         Generated vector sse code for the loop
     77, Barrier
     78, Parallel loop activated with   runtime schedule schedule
         Loop not vectorized/parallelized: contains call
     79, Generated an alternate version of the loop
         Generated vector sse code for the loop
         Generated 2 prefetch instructions for the loop
     83, Barrier
         Parallel region terminated
    100, Loop not vectorized/parallelized: contains call

$ export OMP_NUM_THREADS=1              (1 スレッド)
$ a.out
  Elapsed time =     0.6629619598388672         (sec)
  M =            1000 , N =            1000 , P =            1000
  MFLOPS =       3015.255959008352
  c(1,1) =       1000.000000000000

$ export OMP_SCHEDULE="GUIDED,20"  (OpenMP ランタイムスケジューリング)
$ export OMP_NUM_THREADS=4              (4 スレッド)
$ a.out
  Elapsed time =     0.2607220411300659         (sec)
  M =            1000 , N =            1000 , P =            1000
  MFLOPS =       7667.169186523676
  c(1,1) =       1000.000000000000
```

● F90 組込み関数 MATMUL を使用した場合の性能 （1 コアのみ）

【ソース内容】
```
! F90 Intrinsic function (MATMUL)
        c = matmul(a, b)
```

```
$ pgf90 -fastsse -Minfo matmul.F -DMATMUL wallclock.o
matmul.F:
matmul_time:
     29, Loop interchange produces reordered loop nest: 30,29
         2 loops fused
         Generated vector sse code for the loop
     30, Loop not fused: function call before adjacent loop
         2 loops fused
     34, Loop interchange produces reordered loop nest: 35,34
     52, Loop not vectorized/parallelized: contains call
    100, Loop not vectorized/parallelized: contains call$ a.out
$ a.out
 Elapsed time =     0.3679704666137695          (sec)
 M =            1000 , N =            1000 , P =            1000
 MFLOPS =       5432.501196076145
 c(1,1) =       1000.000000000000
```

● ACML ライブラリ/LAPACK パッケージの DGEMM ルーチンを使用した場合の性能

ここでは、AMD 社が提供している OpenMP 版の ACML ライブラリを使用して並列性能の測定を行います。PGI コンパイラには、ACML ライブラリがバンドルされております。
この詳細と使用法に関しては、弊社ホームページの
http://www.softek.co.jp/SPG/Pgi/TIPS/acml.html をご覧ください。

なお、ACML ライブラリは、PGI 15.10 リリースまでバンドルされておりましたが、ライブラリの開発が終息したため、PGI 16.1 以降にはバンドルされません。その代わり、OpenBLAS project source に基づいた最適化 BLAS/LAPACK を提供しました。使用方法に関しては、以下の URL のページをご参照ください。

「OpenBLAS project source に基づいた BLAS/LAPACK ライブラリ」

【ソース内容】
```
! AMD ACML(LAPACK) Library
        call dgemm('N','N', m, p, n, 1.0d0, a, m, b, n, 0.0d0, c, n)
```

```
$ pgf90 -fastsse -Minfo -mp matmul.F -DBLAS wallclock.o   -lacml_mp
                                                （並列版 ACML ライブラリを指定する）
matmul.F:
matmul_time:
     29, Loop interchange produces reordered loop nest: 30,29
         2 loops fused
         Generated vector sse code for the loop
     30, Loop not fused: function call before adjacent loop
         2 loops fused
```

```
     34, Loop interchange produces reordered loop nest: 35,34
     52, Loop not vectorized/parallelized: contains call
    100, Loop not vectorized/parallelized: contains call
```

PGI 6.2 以降においては、コンパイラに付属した OpenMP 対応 ACML をリンクすることが可能です。-mp と -lacml オプションを付加した場合、OpenMP 対応の ACML を参照するようになります。なお、PGI 7.0 以降の場合、OenMP 対応の ACML は、明示的に -lacml_mp を指定することが必要です。

(PGI 6.2 の場合 ：Linux & Windows 64bit)
$ pgf90 -fastsse -Minfo -mp wallclock.o matmul.F -DBLAS **-lacml**

(PGI 7.0 以降の場合 ：Linx & Windows)
$ pgf90 -fastsse -Minfo **-mp** wallclock.o matmul.F -DBLAS **-lacml_mp**

(Windows 32 ビット版の場合は、-Munix を付加する)
$ pgf90 -fastsse -Minfo -mp **-Munix** wallclock.o matmul.F -DBLAS **-lacml_mp**

$ **export OMP_NUM_THREADS=1** (1 スレッド)
$ a.out
 Elapsed time = 0.1850744485855103 (sec)
 M = 1000 , N = 1000 , P = 1000
 MFLOPS = 10801.05879162676
 c(1,1) = 1000.000000000000

$ **export OMP_NUM_THREADS=4**(4 スレッド)
$ a.out
 Elapsed time = 5.4553985595703125E-002 (sec)
 M = 1000 , N = 1000 , P = 1000
 MFLOPS = 36642.60233550101
 c(1,1) = 1000.000000000000

付録　PGI 技術情報リンク、コンパイラ・オプション一覧

次頁に、PGI コンパイラの主なオプションに意味に関して纏めました。ご参考にしてください。

なお、PGI コンパイラに関する技術情報（TIPS）、FAQ 等につきましては、弊社のホームページにも情報を提供しております。以下の URL にてご覧ください。

（ソフテック PGI 製品トップ）
http://www.softek.co.jp/SPG/Pgi/

（PGI 製品サポートセンター）
http://www.softek.co.jp/SPG/Pgi/support.html

（PGI 技術情報）
http://www.softek.co.jp/SPG/Pgi/tips.html

（PGI コンパイラ・オプションの使用法）
http://www.softek.co.jp/SPG/Pgi/comp-tips.html

（PGI FAQ）
http://www.softek.co.jp/SPG/Pgi/pgi-faq.html

（ソフテック PGI テクニカル情報・コラム）
http://www.softek.co.jp/SPG/Pgi/TIPS/para_guide.html

（主要アプリケーションのポーティングガイド）
http://www.pgroup.com/resources/tips.htm

（PGI コンパイラ最新リリース情報）
http://www.softek.co.jp/SPG/Pgi/new_release.html

（PGI ソフトウェアダウンロードサイト）
http://www.softek.co.jp/SPG/ftp.html

（ソフテック版日本後インストールの手引き・アーカイブ）
http://www.softek.co.jp/SPG/Pgi/inst.html

PGI Compiler Option 一覧

PGI コンパイラのコンパイル・オプション (2017年2月版 PGI 2017対応)

PGI の F77, F2003, C, C++ のコンパイラを使用する際のオプションを以下に示しました。以下は、pgfortran を使用した場合の例ですが、コンパイラのオプションの設定方法は、他の言語コンパイラでも同じです。なお、各オプションの詳細は、PGI User's Guide をお読みください。

```
コマンド名 -[options] [path] filename

(Fortran言語 : 例)   pgf95/pgf90/pgfortran は全て同じコンパイラです。
  pgf95 -fastsse -Minfo=all -L/opt/lib -lmylib test.f
  pgf90 -fastsse -Minfo=all -L/opt/lib -lmylib test.f
  pgfortran -fastsse -Minfo=all -L/opt/lib -lmylib test.f

(C11言語 : 例)
  pgcc  -fastsse -Mipa=fast,inline -L/opt/lib -lmylib test.cpp

(PGI C++03 for Windows) PGI 2016以降、Windows 版の C++ コンパイラは廃止されました (終息)。
  pgCC(pgcpp)  -fastsse -Mipa=fast,inline -L/opt/lib -lmylib test.cpp
  PGI 16.1 以降、Windows 版の C++ コンパイラ処理系の提供はありません。

(PGI C++14言語 for GNU 4.8 ABI 互換、Linux/macOS/OpenPOWER 版) 一例
  pgc++ -fastsse -Mipa=fast,inline  test.cpp
    PGI 16.1 以降 GNU 4.8 互換、5.1 ABI対応(GNU GCC 5.1以上搭載されている場合)
    PGI 15.4 以降 GNU 4.9 互換、4.9 ABI対応(GNU GCC 4.9が搭載されている場合)
    PGI 15.1 以降 Apple OS X でもサポート開始
    PGI 14.7 以降 GNU 4.8 ABI互換
    PGI 13.2～14.7以降 GNU ABI互換 (Linux版)
```

必要とする各オプションを *-[option]* 形式でブランクを空けて指定します。また、-M オプションは、最適化オプションを詳細に指定するものであり、-M に引き続きブランクを空けずにフラグを指定します。なお、-M にさらに、サブ・フラグがある場合は、-M[flag]={subflag} の形式で指定します。**サブ・フラグを指定しない場合は、コンパイラの default 設定のサブ・フラグが使用されます。**

[options]　各コンパイル・オプションを指定する。指定順序は基本的に制約はない
　　但し、ライブラリパス等の順序は重要であり、その順位で反映される
[path]　　リンカへのライブラリ等のパスを指定する
[filename] ソースファイル、オブジェクト・ファイル、アセンブリ・ファイル等を指定する

コンパイル・オプションの概要について以下の表に纏めました。表の中の**「白抜き」の行は、最適化において、よく使用されるオプション**を表しています。

- PGI コンパイル・オプションの説明
- -M オプション (最適化詳細オプション) の各種フラグについての説明
- C、C++ 特有のコンパイル・オプション

PGI Compiler のコンパイル・オプション

オプション	記述
-#	コンパイラ手続きの呼出し情報を表示します。
-###	ドライバコマンドを表示しますが、実行しません (-dryrunと同じ)。
-**acc**[=strict|verystrict]	OpenACC用ディレクティブを認識し、GPU用のコードの生成を行います。(リンク時オプションにも必要)　PGI 12.6 以降 -acc=[no]autopar は、OpenACC parallel構文内の自動並列化を行う[行わない] (PGI 13.6 以降)。 -acc=[no]required はアクセラレータ・コードを生成出来なかった場合、コンパイルエラーとする(default) (PGI 14.1 以降)、(PGI 15.1 以降廃止) -acc=strict は、non-OpenACC accelerator ディレクティブが見つかった場合、warning を出す。-acc=verystrict は、non-OpenACC accelerator ディレクティブが見つかった場合、エラーメッセージを出し、コンパイルを終了する。

	-acc=sync は、async cluase を無視します。 -acc=[no]wait は各デバイス kernel の終了を待つか待たないかを指示します。
-Bdynamic	(Linux) 明示的にコンパイラのドライバが、shared object library をリンクするように指示する。 (PGI 7.1 以降) Windows にも対応。 PGI のランタイムライブラリの DLL をリンクして実行バイナリを生成します。このオプションは、コンパイル時とリンク時の両方で必要です。
-Bstatic	(Linux) 明示的に、PGI runtime static libraries を使用して静的リンクを行うように指示する。 (PGI 7.1 以降) Windows にも対応。Windows 上では、実行形式モジュールにリンクされる全てのファイルは、同じオプションでコンパイル/リンクされなければなりません。また、本オプションは、コンパイル時にも必要です。Windows のデフォルトは、-Bstatic となりました。
-Bstatic_pgi	(Linux only) PGI 用の share library を静的にリンクし、システム依存のライブラリは、ダイナミック・ローディングする形式の実行モジュールを生成する（PGI 6.2 以降）。 このオプションは、-Mnorpath 機能を含む。
-byteswapio	(Fortran only) アンフォーマット Fortran データ･ファイルの入出力時にビッグエンディアンからリトルエンディアンにあるいはその逆に、バイトをスワップ します。生成された実行モジュールは、自動的に read/write 処理中にこのエンディアン変換を行います。
-C	実行時の配列の境界チェックを有効にする実行モジュールを作成するように指示する。
-c	アセンブリフェーズの後で止まり、オブジェクトコードを filename.o にセーブ。
-cudalibs	リンク時に、CUDA ランタイム API ライブラリ群をリンクする
-D<arg>	プリプロセッサマクロを定義します。
-d[D|I|M|N]	**【PGI 7.0 以降　新設】** プリプロセッサからの追加情報を出力させるためのものです。 　-dD：ソースファイルからマクロと値をプリントします。 　-dI：インクルードファイル名をプリントします。 　-dM：前以って定義された、コマンドラインマクロを含むマクロと値をプリントします。 　-dN：ソースファイルからマクロ名をプリントします。
-dryrun	コンパイル手続き上のドライバコマンドを表示しますが,、実行しません。
-drystdin	(PGI 7.2新設）標準インクルード・ディレクトリを出力して終了します。
-E	プリプロセスフェーズの後で止まり、標準出力にプリプロセスされたファイルを表示。 (PGI 7.0 以降) pgcc -E は、.h ファイルの前処理を行うようになりました。
-F	（pgf77、pgfortranとpghpfのみ）プリプロセスフェーズの後で止まり、プリプロセスされたファイルを　filename.f　にセーブ。
-f	無視されます。
-fast	一般的に最適化セット・フラグ。 x86 並びに AMD64 ターゲットに対するフラグ -O2 -Munroll -Mnoframe -Mlre と 同等。PGI のバージョンによって異なるため、pgf90 -fast -help でオプションの内容を確認すること。 **【PGI 7.0 以降の C/C++ 環境】** C/C++コンパイラは、-fast あるいは -fastsse の複合オプションの中に、-Mautoinline 機能も有効になるように変更されました。 **【PGI 7.0 以降の 64 ビット環境】** 64 ビットシステムのターゲットに対して、-fast オプションは、従来の -fastsse オプションと同じ機能を有するオプションに変更しました。新しい -fast オプションは、SSE 命令を伴うベクトル化、キャッシュ整列、flushz（SSEのflush-to-zeroモード）機能を有効にします。従来の -fast と等価な機能として、**-nfast** と言うオプションが新設されました。
-fastsse	SSE/SSE2 インストラクションを有するマシンターゲットへの一般的な最適化フラグセット。 x86 並びに AMD64 ターゲットに対するフラグ、-O2 -Munroll -Mnoframe -Mscalarsse -Mvect=sse -Mchache_align -Mflushz と同等 **【PGI 7.0 以降の C/C++ 環境】** C/C++コンパイラは、-fast あるいは -fastsse の複合オプションの中に、-Mautoinline 機能も有効になるように変更されました。
-flags	有効なドライバオプションとその内容を表示します。この場合は、コンパイルの実行は行われません。
-fpic	（Linux only) 他のコンパイラとの互換性を有するポジション独立のコードを生成します。Dynamic Shared Library を作成する際に使用することができる。-mcmodel=medium と共には使用できません。
-fPIC	（Linux only) -fpicと同じ。

-G	リンカに共有オブジェクトファイル（ダイナミック・リンク・ライブラリ）を生成するように指示する
-g	オブジェクトモジュールにデバッグ情報を含ませます。
-gopt	オブジェクトモジュールにデバッグ情報を含ませます。最適化されたコードのデバッギングが可能とするような方法でモジュールが生成されます。-goptはシンボリックデバッグ情報をオブジェクトモジュールの中に付加し、さらに、-gが指定されない時と同じ最適化コードを生成するように、コンパイラに対して指示するものです。
-g77libs	（Linux only) g77 によって生成されたオブジェクトファイルを pgfortran を使用してコンパイルされたメインプログラムにリンクする場合、このオプションを指定することで、g77 でコンパイルされたプログラム内で使用している未解決な g77 サポートライブラリを検索できるようにします。
-help	ドライバが認識する全てのオプションを標準出力に表示します。また、オプションとサブオプションの内容も表示します。 -help と他のコンパイルオプションを同時に付けた場合、そのオプションの意味・内容を表示します。
-I<dirname>	ディレクトリを #include ファイルのためのサーチパスに加えます。
-i	リンカに渡されます。
-i2	2 バイトとしてINTEGER変数を扱います。（明示的にバイト数を指定しない整数宣言変数）
-i4	4 バイトとしてINTEGER変数を扱います。（明示的にバイト数を指定しない整数宣言変数）
-i8	8 バイトとしてINTEGER変数を扱い、INTEGER*8 オペレーションに 64ビットを使います。
-i8storage	INTEGER変数を 4 バイトとして扱うが、ストアする際に 8 バイトワード(64bit)としてストアする。
-K<flag>	特別なセマンテイックをコンパイルに指示します。<flag> は多種類あるため、詳細は、User's Guide を参照。例えば、IEEE 754 に準拠するように浮動小数点演算を行う、あるいは、**浮動小数点演算の例外処理の方式等の指定**が可能です(default は例外が起きても実行続行する)。 ieee / noieee ：厳密なIEEE 754に準拠する浮動小数点演算 pic ：ポジション独立のコードを生成 trap=[subflag] ：例外が生じた場合、実行を停止させます。 trap=none ： 全てのトラップを抑止（PGI 6.2) （例） **pgfortran -Ktrap=fp test.f**
-L<dirname>	ライブラリ・ディレクトリを指定します。これをライブラリ・サーチ・パスに加えます。
-l<library>	指定された<library>ライブラリをロードします。
-M<pgflag>	コード生成と最適化の各種のフラグ<pgflag>を指定します。フラグの指定方法は、-M<pglag>,<pgflag>, ... or -M<pgflag>=xxxx
-m	標準出力にリンクマップを表示します。
-m32	デフォルトのプロセッサタイプとして、32ビットコンパイラを使用することをコンパイラに指示する。（PGI 10.3 新設）
-m64	デフォルトのプロセッサタイプとして、64ビットコンパイラを使用することをコンパイラに指示する。（PGI 10.3 新設）
-module <moduledir>	(F90/F95/HPF only) ディレクトリ<moduledir>にモジュールファイル（.mod）を保存/検索します。
-mcmodel=medium	（linux86-64のみ） linux86-64 環境において、medium memory model をサポートするコードを生成します。（2GB 超えのプログラム） このオプションは、 **linux86-64（64bit Linux) のみ**となります。 Win64/osx86-64 では使用できません。
-mp[=align,[no]numa allcores,bind]	ユーザーによって挿入された共有メモリ並列プログラミングディレクティブを解釈、処理します。 **align** サブオプションは、並列化と SSE によるベクトル化の両方が適用されるループにおいて、ベクトル化のためのアライメント（整列）を最大化するようなアルゴリズムを使用して、OpenMP スレッドにループ回数を割り当てるようにするものです。この機能は、このような特性を帯びた多くのループがプログラムに存在する時に性能が向上します。しかし、一方、各ループの中で、非常に大きなタスク・ワークを含むループで、そのループ長が相対的に短いプログラムにおいては、結果としてロードバランスの問題が生じて大きく性能を落とす場合がありますので注意が必要です。このオプションを適用し性能を確認してから使用してください。（このalign サブオプションは、**PGI 6.1** 以降のオプションです） **-mp=nonuma** (libnumaライブラリをリンクしない) PGI 6.1 以降 PGI 6.2 から libnumaを有しないシステムには、 その stub（仲介）ライブラリを提供する。

	（**PGI 8.0 以降のサブオプション**） allcores　環境変数OMP_NUM_THREADS あるいは NCPUSにセットしていない場合、すべての有効なコアを使用する（リンク時に指定すること） bind　スレッドをコアあるいはプロセッサにバインドする（リンク時に指定すること）
-noswitcherror	コマンドライン上に、コンパイラに有効なオプションではないものが指定された場合、エラーで終了させる代わりに警告レベルに変更する。この挙動は、コンパイラのサイト初期設定ファイル　siterc ファイルに set NOSWITCHERROR=1 を指定することでも可能となります。（PGI 7.0-4 以降）　siterc ファイルは、一般に $PGI/linux86{-64}/{version}/bin の配下にあります。 PGI 7.1 以降は、未知のオプションが指定された場合、コンパイル・エラーとなります。
-**O***level*	コード最適化レベルを指定します。level は 0、1、2、 3 あるいは 4。 0：各ステートメントに対し基本ブロックを生成します。しかし、スケジューリング並びにグローバルな最適化は行いません。 1：基本ブロック内でのスケジューリング並びにいくつかのレジスタ・割当の関する最適化を行います。しかし、グローバルな最適化は行いません。 2：全ての上記　レベル 1 の最適化を行います。さらに、基本ブロック間の制御フローとデータフロー解析を実施し、グローバル最適化を行います。導入変数の削除や問題のないループの移動、グローバルレジスタの割り当て等のグローバル最適化を行います。 3：アグレッシブなグローバル最適化を行います。全てのレベル 1, 2 の最適化だけでなく、効果のあるなしに関わらず、スカラの置き換え、より積極的な最適化を行います。 4：4 レベルの最適化は、浮動小数点演算式の中で不変変数に対する巻上げ最適化を行うようになります。(PGI 7.0 以降で新設）(PGI 7.1) algebraic transformations と レジスタ・アロケーション最適化を追加しました。 (PGI 2013以降)　上記 -O2 の機能の中に SIMD ベクトルコード生成（-Mvect=simd)、キャッシュアラインメント、冗長性の排除等の最適化機能も含まれました。これらの追加最適化は -O3、-O4 でも引き継がれます。 また、-O のみ指定した場合は、上記、レベル 2 の最適化であるが、SIMD ベクトル最適化は行わない形態となります(PGI 2012 以前の -O2 と同等な最適化となります）。
-o	オブジェクトファイルの名前を指定します。
-nomp	(PGI 11.0以降）PGI 11.0 から、リンク時のオプションとして、常に-mpオプション（マルチスレッドライブラリ）がデフォルトして付加されます。これは、リンケージの時の動作ですので、コンパイル時の動作ではありません。もし、このデフォルトを変更したい場合は、新しいオプション -nomp をリンク時に指定して下さい。
-pgc++libs	PGF77 あるいは pgfortran あるいは、 pgcc でオブジェクトをビルドする際に、PGC++ ランタイムライブラリ群をリンクするために使用します。(pgf77 あるいは pgfortran、 pgccで指定する）
-pgcpplibs	PGF77 あるいは pgfortran あるいは、 pgcc でオブジェクトをビルドする際に、PGCPP ランタイムライブラリ群をリンクするために使用します。(pgf77 あるいは pgfortran、 pgccで指定する）(PGI 16.1以降廃止）
-pgf77libs	PGF77 でコンパイルされたオブジェクトを C あるいは C++ のメインプログラムにリンクする際に、PGF77 ランタイムライブラリをリンクするために使用します。（ pgcc あるいは pgCC で指定する）（PGI 6.0～）
-pgf90libs	pgfortran でコンパイルされたオブジェクトを F77 あるいは、C、C++ のメインプログラムにリンクする際に、pgfortran ランタイムライブラリをリンクするために使用します。（pgf77 あるいは、pgcc、pgc++/pgCC で指定する）（PGI 6.0～）
-P	（pgccとpgc++）プリプロセスフェーズの後で止まり、プリプロセスされたファイルをfilename.iにセーブします。
-pc	(CPU target が、px/p5/p6/piii のみ)　浮動小数点計算時の x86 アーキテクチャ上のレジスタビット長の使用精度の制御を行います。プログラムの誤差感度の評価に有効です。 -pc 32　:　単精度 (32bit) -pc 64　:　倍精度　(64bit) -pc 80　:　x87 natice　(80bit) このモードがデフォルトです -Kieee も参照のこと（厳密の IEEE 754 準拠）
-pg	gprof-style のサンプルベースのプロファイルデータを生成する。生成されたプロファイルデータ gmon.out ファイルは、pgprof で分析可能となる。
-Q	コンパイラステップの変化を選択します。

-R<directory>	(Linux only) リンカへ渡されます。リンク時の共有オブジェクトファイルのサーチパスの中に<directory> を入れます。これは、環境変数 LD_LIBRARY_PATH の内容を変えるものではありません。
-r	リロケータブルなオブジェクトファイルを作成します。
-r4	DOUBLE PRECISION 変数を REAL と解釈します。
-r8	REAL 変数を DOUBLE PRECISION と解釈します。
-rc file	ドライバのスタートアップファイルの名前を指定します。
-rdynamic	pgc++コンパイラにリンカーへのオプションとして -export-dynamicを適用するように指示するスイッチ (PGI 16.1以降)
-S	コンパイルフェーズの後で止まり、アセンブリ言語コードを filename.s にセーブします。
-s	オブジェクトファイルからシンボルテーブル情報を除去します。
-shared	(Linux only) リンカへ渡されます。共有オブジェクトファイルを生成するようにリンカに指示します。
-show	コンパイラ起動時の各ドライバの設定パラメータ、引数の詳細を表示します。
-silent	警告メッセージをプリントしません。
-soname <library.so>	(Linux only) shared オブジェクトを生成する時、library.so（一例） と言うシェアードライブラリを内部の DT_SONAME フィールドへセットするようにリンカーに指示します。
-stack=nocheck	(PGI 7.1 以降） (Windows only) -stack オプションは、Windows 上で自動ランタイムスタック拡張を行わないようにすることができるように変更されました。もし、researve と commit サブオプションが、十分なスタック量を確保できるようにセットされたなら、自動的な拡張チェックは必要ありませんし、スタックのチェックを避けることができます。デフォルトは、-stack=checkです。Win64 では、デフォルトの researve 値あるいは commit 値はありません。Win32 では、researve、commit それぞれのデフォルト値は、2,097,152byteです。
-time	様々なコンパイルステップの実行時間を表示します。
-ta=nvidia(,suboption),host -ta=radeon (,suboptions),host -ta=multicore	(PGI 2010以降、pgfortranとpgcc に有効) (PGI 13.1 pgcpp でも利用可能) OpenACC 用のターゲット・アーキテクチャを意味します。PGI 13.10 以前は、-ta=nvidia でしたが、PGI 14.1 以降 AMD の Radeon GPU ボードも OpenACC 対応となったため、以下のように、NVIDIA 社と AMD 社の二つのメーカの通称ボード名で、OpenACC コンパイルの「ターゲットの識別」を行います。さらに、各ターゲットに対する細かなオプションを指定できます。(デフォルトは -ta=tesla,host です) コンパイル方法の詳細はこちらへ **-ta=tesla** ： NVIDIA アクセラレータをターゲットとして選択します。さらに、以下の nvidia 用のサブオプションがあります。

サブオプション	NVIDIA -ta=tesla(nvidia) のサブオプション
analysis	ループの解析のみ行い、コードの生成を行いません。）(PGI 13.10以降廃止)
cc10	compute capability 1.0 のコードを生成 (PGI 14.1以降廃止)
cc11	compute capability 1.1 のコードを生成 (PGI 14.1以降廃止)
cc12	compute capability 1.2 のコードを生成 (PGI 14.1以降廃止)
cc13	compute capability 1.3 のコードを生成 (PGI 14.1以降廃止)
cc1x	compute capability 1.x のコードを生成 (PGI 15.1以降廃止)
cc1+	compute capability 1.x, 2.x, 3.x のコードを生成 (PGI 14.1以降)、 (PGI 15.1以降廃止)
tesla	cc1x と同じ(PGI 13.1以降)、 (PGI 15.1以降廃止)
tesla+	cc1+ と同じ (PGI 14.1以降)、 (PGI 15.1以降廃止)
cc20	compute capability 2.0 のコードを生成 (PGI 10.4以降) (PGI 14.1以降廃止) (PGI15.5以降復活)
cc2x	compute capability 2.x のコードを生成 (PGI 10.4以降)
cc2+	compute capability 2.x, 3.x のコードを生成 (PGI 14.1以降)
fermi	cc2xと同じ (PGI 13.1以降)
felmi+	cc2+と同じ (PGI 14.1以降)

cc30	compute capability 3.0 のコードを生成 (PGI 12.8以降) (PGI 14.1以降廃止) (PGI15.5以降復活)
cc35	compute capability 3.5 のコードを生成 (PGI 13.1以降) (PGI 14.1以降廃止) (PGI15.5以降復活)
cc3x	compute capability 3.x のコードを生成 (PGI 12.8以降)
cc3+	compute capability 3.x (=cc3x) 以上のコードを生成 (PGI 14.1以降)
kepler	cc3xと同じ (PGI 13.1以降)
kepler+	cc3+と同じ (PGI 14.1以降)
cc50	compute capability 5.0 のコードを生成 (PGI 15.7以降)
cc60	compute capability 6.0 のコードを生成 (PGI 16.10以降)
charstring	GPUカーネル内で文字列の使用を制限付きで使用する (PGI 15.1以降)
cuda2.3 or 2.3	PGIにバンドルされた CUDA toolkit 2.3 バージョンを使用 (PGI 10.4以降)
cuda3.0 or 3.0	PGIにバンドルされた CUDA toolkit 3.0 バージョンを使用 (PGI 10.4以降)
cuda3.1 or 3.1	PGIにバンドルされた CUDA toolkit 3.1 バージョンを使用 (PGI 10.8以降)
cuda3.2 or 3.2	PGIにバンドルされた CUDA toolkit 3.2 バージョンを使用 (PGI 11.0以降)
cuda4.0 or 4.0	PGIにバンドルされた CUDA toolkit 4.0 バージョンを使用 (PGI 11.6以降)
cuda4.1 or 4.1	PGIにバンドルされた CUDA toolkit 4.1 バージョンを使用 (PGI 12.2以降)
cuda4.2 or 4.2	PGIにバンドルされた CUDA toolkit 4.2 バージョンを使用 (PGI 12.6以降)
cuda5.0 or 5.0	PGIにバンドルされた CUDA toolkit 5.0 バージョンを使用 (PGI 13.1以降)
cuda5.5 or 5.5	PGIにバンドルされた CUDA toolkit 5.5 バージョンを使用 (PGI 13.9以降)
cuda6.0 or 6.0	PGIにバンドルされた CUDA toolkit 6.0 バージョンを使用 (PGI 14.4以降)
cuda6.5 or 6.5	PGIにバンドルされた CUDA toolkit 6.5 バージョンを使用 (PGI 14.9以降)
cuda7.0 or 7.0	PGIにバンドルされた CUDA toolkit 7.0 バージョンを使用 (PGI 15.4以降)
cuda7.5 or 7.5	PGIにバンドルされた CUDA toolkit 7.5 バージョンを使用 (PGI 15.9以降)
cuda8.0 or 8.0	PGIにバンドルされた CUDA toolkit 8.0 バージョンを使用 (PGI 16.10以降)
[no]debug	デバイスコード内にデバッグ情報を生成する[しない] (PGI 14.1 以降)
fastmath	fast mathライブラリを使用
[no]flushz	GPU上の浮動小数点演算の flush-to-zero モードを制御。デフォルトはnoflushz。 (PGI 11.5以降)
[no]fma	fused-multiply-add命令を生成する[しない] (default at -O3)
keep	kernelバイナリファイル(.bin)、kernelソースファイル(.gpu)、portable assembly(.ptx)ファイルを保持し、各々ファイルとして出力する (PGI 13.10以降)
keepbin	kernelバイナリファイルを保持し、ファイル(.bin)として出力する) (PGI 13.10以降廃止)

keepgpu	kernelソースファイルを保持し、ファイル(.gpu)として出力する (PGI 13.10以降廃止)
keepptx	GPUコードのためのportable assembly(.ptx)ファイルを保持し、ファイルとして出力する) (PGI 13.10以降廃止)
[no]lineinfo	GPU line informationを生成する (PGI 15.1 以降)
[no]llvm	llvmベースのバックエンドを使用してコードを生成する。PGI 15.1 以降は、64-bit上ではLLVMバックエンドをデフォルトとして使う [使わない]
maxregcount:n	GPU上で使用するレジスタの最大数を指定。ブランクの場合は、制約が無いと解釈する
mul24	添字計算に、24ビット乗算を使用 (GT200系、CC 1.3のみ) (PGI 13.10以降廃止)
noL1	グローバル変数をキャッシュするためのハードウェア L1 データキャッシュの使用を抑止する (PGI 13.10以降)
loadcache:L1 loadcache:L2	グローバル変数をキャッシュするためのハードウェア L1 or L2 データキャッシュを使用する (PGI 14.4以降)但し、アーキテクチャ上、有効とならない GPU がある
pin	デフォルトを pin ホストメモリ(割付）としてセットする(PGI 14.1以降、PGI16.1廃止)
pinned	プログラムのアロケート時に pinned メモリを割り付けるよう（PGI 16.1以降)
time	アクセラレータ領域の単純な時間情報を集積するためにプロファイル・ライブラリをリンクする。このオプションは、PGI 13.1 以降廃止されました。この代わりに、プロファイルを環境変数 PGI_ACC_TIME に 1 をセットすることにより実行後プロファイル情報が出力されます。
[no]required	アクセラレータ・コードを生成出来なかった場合、コンパイルエラーとする [しない] (default) (PGI 14.1 以降）、 (PGI 15.1以降廃止)
[no]rdc	異なるファイルに配置されたデバイスルーチンをそれぞれ分割コンパイルし、リンクが出来るようにする。cc2x以降、CUDA 5.0 以降の機能を使用する。(PGI 13.1以降 + CUDA 5.0 以降) (PGI 14.1 は以降デフォルト)
[no]unroll	自動的に最内側ループのアンローリングを行う (default at -O3) (PGI 14.9以降)
managed	CUDA managed Memory を使用する
beta	ベータ版機能のコード生成（生成コード内の 128-bit ロード・ストアオペレーションを有効化）(PGI 15.7以降)
[no]wait	ホスト側での実行継続を行う際に、各カーネルが終了するまで待つ。nowaitは待たない。）(PGI 13.10以降廃止)
safecache	cache directive 内での可変長の配列セクションの使用を許す。但し、そのサイズは CUDA shared memory 内に収まるものでなければならない。(PGI 16.5以降)

-ta=tesla**,host** ： host は、アクセラレータがターゲットとして存在しないコード生成も行う。アクセラレータ領域をホスト側で実行するようにコンパイルする。PGI Unified Binaryコードを生成する。

-ta=radeon - AMD アクセラレータをターゲットとして選択します。さらに、以下の radeon 用のサブオプションがあります。このサブオプションは、カンマ（,）で区切って複数のものを指定することができます） (PGI 14.1 以降)。

サブオプション	AMD -ta=radeon のサブオプション
buffercount:n	データをアロケートする際のOpenCLバッファの最大数をセットする
capeverde	Radeon Cape Verde アーキテクチャ用のコードを生成

keep	kernel ファイルを保持する
[no]lineinfo	GPU line informationを生成する (PGI 15.1 以降)
[mo]llvm	llvm/SPIRベースのバックエンドを使用してコードを生成する。PGI 15.1 以降は、64-bit上ではLLVM/SPIRバックエンドをデフォルトとして使う [使わない]
[no]required	アクセラレータ・コードが生成出来なかった場合、コンパイルエラーとする [しない] (default)、(PGI 15.1 以降廃止)
[no]unroll	自動的に最内側ループのアンローリングを行う (default at -O3) (PGI 14.9以降)
capeverde	Radeon capeverdeアーキテクチャ用コードを生成
spectre	Radeon Spectre アーキテクチャ用コードを生成
tahiti	Radeon Tahiti アーキテクチャ用コードを生成
spir	LLVM/SPIRバックエンドをデフォルトとして使う (PGI 15.1 以降)

-ta=radeon**,host** ： host は、アクセラレータがターゲットとして存在しないコード生成も行う。アクセラレータ領域をホスト側で実行するようにコンパイルする。PGI Unified Binaryコードを生成する。

-ta=multicore - ホスト上のマルチコアCPU上で並列動作するように OpenACC プログラムをコンパイルします。（PGI 15.10 以降）

-tp <target>

ターゲットプロセッサのタイプを指定し、そのアーキテクチャに沿ったコードを生成します。 **ターゲットの default は、コンパイルを実行するシステムの「プロセッサ・タイプ」にターゲットが設定されます。コンパイルの方法は、こちらへ**

amd64	AMD64 Processor	PGI6.0以前
athlon	AMD Athlon Processor	PGI6.0以前
athlonxp	AMD Athlon XP Processor	PGI6.0以前
k8-32	AMD Athlon64/Opteron 32-bit mode	
k8-64	AMD Athlon64/Opteron 64-bit mode	
k8-64e	AMD Opteron Rev.E/F Turion 64-bit mode	PGI6.1以降
barcelona	AMD barcelona/Quad-Core AMD64	PGI 7.0-3 以降
barcelona-32	AMD barcelona/Quad-Core AMD64 32-bit mode	PGI 7.0-3 以降
barcelona-64	AMD barcelona/Quad-Core AMD64 64-bit mode	PGI 7.0-3 以降
shanghai	AMD shanghai/Quad-Core AMD64	PGI 8.0以降
shanghai-32	AMD shanghai/Quad-Core AMD64 32-bit mode	PGI 8.0以降
shanghai-64	AMD shanghai/Quad-Core AMD64 64-bit mode	PGI 8.0以降
istanbul	AMD istanbul/six-Core AMD64	PGI 9.0以降
istanbul-32	AMD istanbul/six-Core AMD64 32-bit mode	PGI 9.0以降
istanbul-64	AMD istanbul/six-Core AMD64 64-bit mode	PGI 9.0以降
bulldozer	AMD bulldozer AMD64	PGI 11.9以降
bulldozer-32	AMD bulldozer AMD64 32-bit mode	PGI 11.9以降
bulldozer-64	AMD bulldozer AMD64 64-bit mode	PGI 11.9以降
piledriver	AMD Piledriver AMD64	PGI 13.1以降

	piledriver-32 / AMD Piledriver AMD64 32-bit mode / PGI 13.1以降 piledriver-64 / AMD Piledriver AMD64 64-bit mode / PGI 13.1以降 piii / Intel PentiumIII with SSE1 only p6 / Intel Pentium Pro, II, III, AthlonXP p7 / Intel Pentium 4/Xeon 32-bit mode px / Intel generic x86 mode p7-64 / Intel Xeon/Pentium4 EM64T 64-bit mode / PGI5.2以降 core2 / Intel Core 2 (Duo) 32-bit mode / PGI6.2以降 core2-64 / Intel Core 2 (Duo) EM64T 64-bit mode / PGI6.2以降 penryn / Intel Penryn 32-bit mode / PGI7.2以降 penryn-64 / Intel Penryn 64-bit mode / PGI7.2以降 nehalem / Intel Core i7/i5/i3(Nehalem) / PGI9.0以降 nehalem-32 / Intel Core i7/i5/i3(Nehalem) 32-bit mode / PGI9.0以降 nehalem-64 / Intel Core i7/i5/i3(Nehalem) 64-bit mode / PGI9.0以降 sandybridge / Intel Core i7/i5/i3(SandyBridge) / PGI11.6以降 sandybridge-32 / Intel Core i7/i5/i3(SandyBridge) 32-bit mode / PGI11.6以降 sandybridge-64 / Intel Core i7(SandyBridge) 64-bit mode / PGI11.6以降 ivybridge / Intel Core i7/i5/i3(IvyBridge) / PGI14.1以降 ivybridge-32 / Intel Core i7/i5/i3(IvyBridge) 32-bit mode / PGI14.1以降 ivybridge-64 / Intel Core i7(IvyBridge) 64-bit mode / PGI14.1以降 haswell / Intel Core i7/i5/i3(Haswell) / PGI14.1以降 haswell-32 / Intel Core i7/i5/i3(Haswell) 32-bit mode / PGI14.1以降 haswell-64 / Intel Core i7(Haswell)　以降 64-bit mode / PGI14.1以降 x64 / AMD64/EM64Tの両方に対応可能とした最適化を施こす Unified Bynary の生成（-tp p7-64,k8-64 と同意） / PGI6.1以降 **【PGI 7.0 以降の 64 ビット環境】** -tp オプションは、コンマ（、）で区切られた複数の64ビット・ターゲットを記述する方式を採用しました。以前のバージョンでは、これは一つのターゲットのみの記述方式でした。もし、複数のターゲットが指定された場合、Unified binary は、各ターゲットに対して最適化されたコードを生成します。
-[no]traceback	環境変数 PGI_TERM のスイッチにより異常終了時のトレースバックの処理を制御できますが、その際に必要なデバッグ情報を加えます。なお、Fortranコンパイラのデフォルトはonであり、C/C++ のデフォルトは OFF として設定されています。
-U symbol	プリプロセッサマクロを #undef します。
-u symbol	リンカーにとって未定義なものとしてシンボルテーブルを symbol で初期化します。未定義シンボルは、アーカイブライブラリ上の最初のメンバーのローディングを引きおこします。
-V{Release_Number}	バージョンメッセージ、及び、他の情報を表示します。-V に続けてシステムにインストールしている過去のバージョンを指定した場合、デフォルトバージョンではなく、そのバージョンのコンパイラを使用してコンパイルされます。 （例）　pgfortran -V5.2 test.f (PGI 7.1 以降)　プロセッサ名をプリントするようになりました。例えば、Core 2 Duo上でコンパイルすると-Vオプションは、-tp core2-64と表示します。
-v	コンパイラ、アセンブラ、及び、リンカフェーズ呼出しを表示します。
-W	引数を特定のフェーズ（コンパイル、アセンブラ、リンカ）に渡します。 -W{0,1,l}, <option>,<option> 形式：0　はコンパイラ、1 はアセンブラ、l はリンカ
-w	警告メッセージを表示しません。

-M オプションの各種フラグ

pgflag	記述	カテゴリ
allocatable=[95/03]	【PGI 7.0 以降 pgfortranで　新設】 -Mallocatable= オプションは、コンパイラがメモリ割付 (allocatable) に係る意味合いを制御します。デフォルトは Fortran 95 に準拠します。 =03 オプションは、Fortran 2003 に準拠します。	Fortran95 言語
anno	アセンブリコードと共にソースコードを注釈する。-Manno -S の指定により、アセンブラ・リスティング・ファイル xxxx.s の中にソース・リストとそれに対するアセンブラアセンブラ・リストが両方表示される。	その他
[no]asmkeyword	（pgccとpgc++）コンパイラが　C/C++ ソースファイルにの中に asm キーワードの挿入を許すかどうかを指定。asm キーワードの構文は以下のとおり。 　asm("statement") ; statement はアセンブラ言語による文であり、ダブル・クォツで囲むことが必要。	C / C++言語
[no]autoinline[=levels:n | maxsize:n | totalsize:n]	（PGI 6.2 以降）最適化オプション -O2 以上において、C/C++ コンパイラはインラインキーワードで定義されたもの、あるいはクラス実体(class body)で定義された関数をインライン化する。-Mnoautoinline は、このインライン化を抑止する。levels:n は、インラインの段数（レベル）の数の制約値を指定します。そのデフォルトは4です。 （PGI 2010 以降の C/C++ コンパイラ）　-O2 オプション時にインライン化をコンパイラに指示する。 levels:n　：インラインを行うレベル階層を最大 n まで行うことを指示。デフォルトは10 です。 maxsize:n ： nサイズを超える関数のインラインを行わない。デフォルトは100。 totalsize:n ： インライン対象が、n サイズ時にインラインを止めることを指示。 デフォルトは800。	インライン C / C++言語
[no]backslash	（pgf77、pgfortranとpghpfのみ） backslash キャラクタが quote された文字列において　escape キャラクタとして扱うかを決定。	Fortran言語
[no]**bounds**	実行時の配列の境界チェックを有効にするか、無効するかを指定。プログラムのデバッグ時に非常に有効である。例えば、配列境界外のアクセスを行った場合、以下のような形式で出力される。 　PGFTN-F-Subscript out of range for array a (a.f: 2) 　subscript=3, lower bound=1, upper bound=2, dimension=2	その他
[no]builtin	（pgccとpgc++）数学サブルーチンのビルトインサポート（選択された数学ライブラリルーチンをインライン化する）を用いてコンパイルする[しない]。	最適化
byteswapio	FortranアンフォーマットデータのI/O時にバイトオーダをスワップ（リトルエンディアンからビッグエンディアンに、またその逆）する。	その他
cache_align	可能な限り、16バイト以上のデータオブジェクトをキャッシュラインに整列させる。特に、SSE/SSE2 のベクトル化を行う際に有効（必須）である。	最適化
chkfpstk (32bit only)	関数の開始時と、関数またはサブルーチン呼び出しから戻った後での x86 FPスタックの内部の一貫性についてチェック。 実行時に環境変数 PGI_CONTINUE=verbose のセットを行うと警告メッセージが出る。32bit のみに有効で、64ビット環境では無視される。	その他
chkptr	（pgfortranとpghpfのみ） NULLポインタについてチェック。	その他
chkstk	パラレル領域のエントリー時と、パラレル領域の開始前にエントリ上のスタックの利用可能なスペースをチェック。多くのプライベートな変数が宣言されるときに有益。 （PGI 7.1 以降）　-Mchkstk オプションでコンパイルされたプログラムは、スタック high-water mark の情報を収集できるように	その他

<table>
<tr><td></td><td>指示できます（Windows版のみ）。もし、環境変数 PGI_STACK_USAGE が実行時にセットされた場合、スタックの high-water mark が実行終了時に印字されます。</td><td></td></tr>
<tr><td>concur[=flag[,flag,...]]</td><td>ループの自動並行化を有効にします。複数のプロセッサにより並列化できるループの並列性を確認し、可能な限り並列化する（共有メモリマルチCPUシステムのみで有効）。以下のサブ・フラグがありますので、詳細は User's Guide を参照のこと。
altcode:n / noaltcode
dist:block / dist:cyclic
cncall
assoc/noassoc
[no]innermost (最内側ループの並列化)　PGI 6.1以降
nonuma (libnumaライブラリをリンクしない) PGI 6.1 以降
（PGI 8.0 以降）
allcores　環境変数OMP_NUM_THREADS あるいは NCPUSにセットしていない場合、すべての有効なコアを使用する　（リンク時に指定すること）
bind　　スレッドをコアあるいはプロセッサにバインドする（リンク時に指定すること）</td><td>最適化</td></tr>
<tr><td>cpp=[option]</td><td>後続のコンパイル手続きを行わずに、PGI cppライクのプリプロセッサを実行する。このオプションは、makefileの中に含む各ルーチンの依存情報を生成する際に有効です。optionは、以下に示す一つあるいは複数の文字列（m, md, mm, mmd）からなる。もし、これらの複数のオプションが指定された場合は、最後にリストされたオプションのみが有効となる。
<table>
<tr><td>m</td><td>: makefile dependenciesをstdoutに出力する。</td></tr>
<tr><td>md</td><td>: makefile dependenciesをfilename.dと言うファイルに出力します。ここでfilenem.dとは、コンパイルする入力ファイル名のルート部分の名前が採用される。</td></tr>
<tr><td>mm</td><td>: makefile dependenciesをstdoutに出力しますが、システムincludeファイルは無視する。</td></tr>
<tr><td>mmd</td><td>: makefile dependenciesをfilename.dと言うファイルに出力します。ここでfilenem.dとは、コンパイルする入力ファイル名のルート部分の名前が採用される。なお、ステムincludeファイルは無視する。</td></tr>
<tr><td>[no]comment</td><td>: プリプロセス処理の出力のコメントを残す（さない）。</td></tr>
<tr><td>[suffix:]
<suff></td><td>: makefile dependenciesを含むファイルの添字として<suff>を使用する</td></tr>
</table></td><td>その他</td></tr>
<tr><td>cray</td><td>（pgf77、pgfortranとpghpfのみ）Cray Fortran（CF77）互換性を強制。</td><td>最適化</td></tr>
<tr><td>cuda=[option]</td><td>（pgfortranのみ、PGI 2010以降、アクセラレータ製品のみ） コンパイラは、一般的な Fortran 構文だけでなく CUDA Fortran 構文を解釈するコンパイラモードとなる。CUDA Fortran プログラムをコンパイルし、必要なライブラリをリンクします。なお、リンク時においてもこのオプションが必要です。以下のサブオプションを有する。このサブオプションは、カンマ（,）で区切って複数のものを指定する。
<table>
<tr><td>サブ
オプション</td><td>nvidia用 機能</td></tr>
<tr><td>emu</td><td>エミュレーションモードでコンパイルします。これは、GPU 用のコード生成は行わず、ホスト側でエミュレーション実行可能なコードを生成します。一般に、デバッグ時に使用します。CUDA Fortran の " device code (kernel)" は、ホスト上で実行出来るコードで生成され、ホスト側の pgdbg デバッガを使用できます。</td></tr>
<tr><td>cc10</td><td>compute capability 1.0 のコードを生成 (PGI 13.10以降廃止)</td></tr>
</table></td><td>CUDA Fortran言語</td></tr>
</table>

cc11	compute capability 1.1 のコードを生成 (PGI 13.10以降廃止)
cc12	compute capability 1.2 のコードを生成 (PGI 13.10以降廃止)
cc13	compute capability 1.3 のコードを生成 (PGI 13.10以降廃止)
cc1x	compute capability 1.x のコードを生成 (PGI 15.1以降廃止)
cc1+	compute capability 1.x, 2.x, 3.x のコードを生成 (PGI 14.1以降), (PGI 15.1以降廃止)
tesla	compute capability 1.x (=cc1x) のコードを生成 (PGI 13.1以降), (PGI 15.1以降廃止)
cc20	compute capability 2.0 のコードを生成 (PGI 10.4以降) (PGI 13.10以降廃止) (PGI15.5以降復活)
cc2x	compute capability 2.x のコードを生成 (PGI 10.4以降)
cc2+	compute capability 2.x, 3.x のコードを生成 (PGI 14.1以降)
felmi	compute capability 2.x (=cc2x) のコードを生成 (PGI 13.1以降)
felmi+	cc2+と同じ (PGI 14.1以降)
cc30	compute capability 3.0 のコードを生成 (PGI 12.8以降) (PGI 13.10以降廃止) (PGI15.5以降復活)
cc35	compute capability 3.5 のコードを生成 (PGI 13.1以降) (PGI 13.10以降廃止) (PGI15.5以降復活)
cc3x	compute capability 3.x のコードを生成 (PGI 12.8以降)
kepler	compute capability 3.x (=cc3x) のコードを生成 (PGI 13.1以降)
cc50	compute capability 5.0 のコードを生成 (PGI 15.7以降)
cc60	compute capability 6.0 のコードを生成 (PGI 16.10以降)
charstring	GPUカーネル内で文字列の使用を制限付きで使用する(PGI 15.1 以降)
cuda2.3 or 2.3	PGIにバンドルされた CUDA toolkit 2.3 バージョンを使用 (PGI 10.4以降)
cuda3.0 or 3.0	PGIにバンドルされた CUDA toolkit 3.0 バージョンを使用 (PGI 10.4以降)
cuda3.1 or 3.1	PGIにバンドルされたCUDA toolkit 3.1 バージョンを使用 (PGI 10.8以降)
cuda3.1 or 3.1	PGIにバンドルされたCUDA toolkit 3.2 バージョンを使用 (PGI 11.0以降)
cuda4.0 or 4.0	PGIにバンドルされた CUDA toolkit 4.0 バージョンを使用 (PGI 11.6以降)
cuda4.1 or 4.1	PGIにバンドルされた CUDA toolkit 4.1 バージョンを使用 (PGI 12.2以降)
cuda4.2 or 4.2	PGIにバンドルされた CUDA toolkit 4.2 バージョンを使用 (PGI 12.6以降)

	cuda5.0 or 5.0 — PGIにバンドルされた CUDA toolkit 5.0 バージョンを使用 (PGI 13.1以降)	
	cuda5.5 or 5.5 — PGIにバンドルされた CUDA toolkit 5.5 バージョンを使用 (PGI 13.9以降)	
	cuda6.0 or 6.0 — PGIにバンドルされた CUDA toolkit 6.0 バージョンを使用 (PGI 14.4以降)	
	cuda6.5 or 6.5 — PGIにバンドルされた CUDA toolkit 6.5 バージョンを使用 (PGI 14.9以降)	
	cuda7.0 or 7.0 — PGIにバンドルされた CUDA toolkit 7.0 バージョンを使用 (PGI 15.4以降)	
	cuda7.5 or 7.5 — PGIにバンドルされた CUDA toolkit 7.5 バージョンを使用 (PGI 15.9以降)	
	cuda8.0 or 8.0 — PGIにバンドルされた CUDA toolkit 8.0 バージョンを使用 (PGI 16.10以降)	
	fastmath — fast mathライブラリを使用 (PGI 10.4以降)	
	[no]flushz — GPU上の浮動小数点演算の flush-to-zero モードを制御。デフォルトはnoflushz。 (PGI 11.5以降)	
	keepbin — kernelバイナリファイルを保持し、ファイル(.bin)として出力する	
	keepgpu — kernelソースファイルを保持し、ファイル(.gpu)として出力する (PGI 10.3新設)	
	keepptx — GPUコードのためのportable assembly(.ptx)ファイルを保持し、ファイルとして出力する	
	maxregcount:n — GPU上で使用するレジスタの最大数を指定。ブランクの場合は、制約が無いと解釈する	
	[no]lineinfo — GPU line informationを生成する(PGI 15.1 以降)	
	[no]llvm — 64-bit上ではLLVMバックエンドをデフォルトとして使う [使わない]	
	nofma — fused-multiply-add命令を生成しない (PGI 10.4以降)	
	noL1 noL1cache — グローバル変数をキャッシュするためのハードウェア L1 データキャッシュの使用を抑止する (PGI 13.10以降)	
	loadcache:L1 loadcache:L2 — グローバル変数をキャッシュするためのキャッシュを選択する（Kepler K40以降） (PGI 14.4以降)	
	ptxinfo — コンパイル時にPTXAS情報メッセージを表示する(PGI 11.0以降)	
	[no]rdc — Fortran Module 内の device routine など、異なるファイルに配置されたデバイスルーチンをそれぞれ分割コンパイルし、リンクが出来るようにする。CUDA 5.0 以降の機能を使用します。(PGI 13.1以降 + CUDA 5.0 以降)(PGI 14.1 以降デフォルト)	
	[no]unroll — 自動的に最内側ループのアンローリングを行う (default at -O3) (PGI 14.9以降)	
cudalib[=cublas|cufft|curand|cusparse]	指定した NVIDIA CUDAライブラリをリンクする。	リンク
cudax86	(PGI 11.5新設、pgc++/pgcpp のみ) CUDA C++ プログラムを PGI C++ コンパイラでコンパイルして、この実行バイナリをインテルやAMDの x86 プロセッサ上で実行できる PGI CUDA C for Multi-core x86 機能を有効にする。	C++言語

[no]daz	IEEE 754 正規化されていない数字（内部表現）に対して、zero セットすることを許可する（しない）オプション。(PGI6.0) このオプションは、メインプログラムに対して適用しなければ有効とならない。 PGI 6.2 以降、64ビット EM64T の場合は、-Mdazがデフォルトとし、AMD64 の場合は、-Mnodaz がデフォルトとなる。	最適化
[no]dclchk	（pgf77、pgfortranとpghpfのみ）全てのプログラム変数が宣言されていることを前提としてチェックする（しない）。	Fortran言語
[no]defaultunit	（pgf77、pgfortranとpghpfのみ）どのようにアステリスクキャラクタ“*”が（I/O ユニット 5 と 6 の状態に関係なく）標準入力、および、標準出力と関連して扱われるかを決定。	Fortran言語
[no]depchk	潜在的なデータ依存性が実際に存在することをコンパイラに指示してチェックを行う。一方、nodepchk は依存性がないことをコンパイラに指示する（もし、この場合、存在した場合は不正確な結果となる）。	最適化
[no]dlines	（pgf77、pgfortranとpghpfのみ）コンパイラが実行可能なステートメントとしてカラム1に“D”を含む行を扱うかどうかを決定。	Fortran言語
dll	（Windows only) ランタイムライブラリの DLL バージョンとリンクする。 (PGI 7.0以降）-Mdll オプションは、-D_DLL 機能を含意します。-D_DLL は、プリプロセッサ・シンボル _DLL を定義するものです。 ((PGI 7.1以降）-Mdll オプションが削除されました。その代わり、-Mdynamic オプションを使用します。	その他
dollar,*char*	コンパイラがドル記号コードをマップする際の文字(*char*)を指定。ドル記号を名前として使用することを許す。ANSI C は許さない。	Fortran言語
[no]dse	【PGI 7.0 以降】 参照しない変数の保存を排除（dead store eliminations）する最適化を有効[無効]にするオプションです。これは、C++プログラムのようなパフォーマンス向上のために、関数呼び出しをインライン化することが多い場合に有効となります。	最適化
dwarf1 | dwarf2 | dwarf3	DWARF1 あるいは DWARF2、DWARF3 フォーマットのいずれかのデバッグ情報を生成する。デフォルトは、 DWARF2 である。-g とともに使用する。	コード生成
nodwar	(PGI 8.0以降）デバッグ情報を生成バイナリに付け加えないように指示する。	コード生成
eh_frame, noeh_frame	(PGI 2010以降）リンカーに、executable内のen_frameのcall frame を保持/非保持することを指示する。 (注意) このオプションは、システムunwindライブラリを持つ、最新のLinux、Windowsシステムでのみ有効です。	コード生成
extend	（pgf77、pgfortranとpghpfのみ）コンパイラは、132 カラムソースコードを受け付けます。デフォルトでは 72 カラムコードを受け付けます。	Fortran言語
extract[=*flag[, flag,...]]*	関数エキストラクタを起動。コマンドライン上で指定されたファイルから関数を抽出し、指定した外部 directory へその関数ファイルを生成、追加する。 インライン（-Minline） とともに使用する場合が多い。以下のフラグがありますので、詳細は User's Guide を参照のこと。 name:func size:number lib:dirname	インライン化
fcon	（pgccとpgCCのみ）浮動小数点定数を倍精度型の代わりに、 float 型として扱うようにコンパイラに指示。	C / C++言語
fixed	（pgfortranとpghpfのみ）F77スタイルの固定フォーマットのソースであると認識する	Fortran言語
[no]flushz	SSE/SSE2 を flush-to-zero モードにセットする。浮動小数点のアンダーフローが生じた場合、これを 0 にセットする。このオプションは、メインプログラムに対して適用しなければ有効とならない。	最適化
free	（pgfortranとpghpfのみ）コンパイラは F90 形式のフリーフォーマットのソースコードであると仮定する。	コード生成
[no]func32	32 Byte 上で全ての関数をアライン（整列）させる。	その他

[no]fpapprox[=div|sqrt|rsqrt]	(PGI 7.1 以降) 特定の単精度浮動小数点演算を低精度近似方を使用して実行します。のオプションは結果の差異が生じる可能性がありますので、十分注意して使用してください。 div：浮動小数点除算近似 sqrt：浮動小数点平方根近似 rsqrt：浮動小数点逆数平方根近似 デフォルトでは、-Mfpapprox は使用されません。もし、サブ・オプションを指定しない -Mfpapprox のみの場合は、上記の全てのサブ・オプションが指定されたものとして扱います。 (PGI 8.0　追加)　-Mnofpapprox　：低い精度の浮動小数点演算を使用しないように指示する。	最適化
[no]fpmisalign	(PGI 7.1 以降) AMD barcelonaプロセッサに対して、16-byte境界に整列されていないアドレスを持つメモリ・オペランドのベクトル演算命令の使用を許可します。デフォルト設定は、Barcelonaを含めて、全てのプロセッサにおいて-Mnofpmisalignです。本オプションは、-tp barcelona-64あるいは、-tp barcelonaの設定時、あるいは、barceona上でコンパイルされたときにのみ効果があります。また、このオプションでコンパイルされたコードは、barcelonaプロセッサ上だけで実行できるものとなりますのでご注意下さい。	最適化
[no]fprelaxed= [div,order,rsqrt,sqrt:recip, intrinsic]	いくつかの内部組込み関数 (div/sqrt/rsqrt) の計算において緩い精度で行うことをコンパイラに指示する。性能は向上するが、計算精度は劣る。(PGI 6.1 以降)　デフォルトは、-Mnofprelaxed。 PGI 6.2 以降、細かな制御を行うためのサブオプションを導入。サブオプションは以下のとおりです。 -Mfprelaxed=[div,rsqrt,sqrt] div：緩い精度で除算処理を行う。 order：a*b+a*cをa*(b+c)と変換する方式も含め、演算の順序の変更 noorder：上記 order を行わない。 rsqrt：緩い精度でsqrtの逆数近似（1/sqrt）の処理を行う sqrt：緩い精度でsqrtの処理を行う。 なお、サブオプションを付加しない場合（-Mfprelaxedのみ）は、そのターゲットプロセッサに応じて、顕著な性能向上が行える処理に緩い精度での処理を行うかを選択し適用される。 （PGI 9.0 新設） recip：緩和した精度で逆数近似 （PGI 13.1 新設） intrinsic：緩和した精度の組み込み関数を使用	最適化
[no]i4	（pgf77、pgfortranとpghpfのみ）どのようにコンパイラがINTEGER 変数を扱うかを決定。i4 の場合、INTEGER*4、noi4 の場合は、INTEGER*2 として扱う。	最適化
iface=unix | cref | mixed_str_len_arg | nomixed_str_len_arg	（PGI 7.2 新設　Windowsのみ）サブオプション-MifacはFortranのための呼び出しルール（コンベンション）を調整するものです。 unix (32bit only) - Use UNIX calling conventions＋語末のアンダースコアがあるタイプ cref - Use CREF calling conventions＋ 語末のアンダースコアがないタイプ mixed_str_len_arg -文字列の長さをその対応する引数の直後に置くタイプ nomixed_str_len_arg - 文字列の長さを引数リストの最後に置くタイプ。	Fortran言語
[no]idiom	ループ内でidiom認識（パターン認識）を行う [抑止する] (PGI 15.1以降)	最適化
info[=*flag[,flag,...]*]	コンパイル時に最適化並びにコード生成に関するコンパイル・メッセージを標準出力に表示する。以下のサブ・フラグがありますので、詳細は User's Guide を参照のこと。 all inline ipa loop mp opt time unroll （PGI 7.2新設） intensity －ループ内の「演算密度」(Computational Intensity)を表示します。デフォルトは、最内側ループの情報が表示されます。演算密度とは、一般にループ内の演算数とメモリのロード・ストア数との比率を表し、演算とメモリ参照のバランスを見るための	その他

	指標です。このような情報はパフォーマンス・チューニングにおいて特に重視されます。 ・ ループ内の演算が浮動小数点演算である場合、演算密度は、浮動小数点演算総数を浮動小数点データのメモリロードとストアの総和で割った比率として定義します。 ・ ループ内の演算が整数演算である場合、演算密度は、整数演算総数を整数データのメモリロードとストアの総和で割った比率として定義します。 （PGI 8.0 以降新設） all 以下のサブオプションをすべて指定したものと解釈します。 -Minfo=accel,inline,ipa,loop,lre,mp,opt,par,unified,vect accel アクセレータ情報の有効化 ccff オブジェクトファイルに最適化情報を追加します ftn Fortran特有な情報の有効化 hpf HPF特融な情報の有効化 information inline インライン情報の有効化 lre LRE情報の有効化 par 並列化の情報の有効化 pfo プロファイル・フィードバックに関する情報の有効化 vect ベクトル化の情報の有効化 （PGI 9.0 新設） accel アクセラレータ領域をGPU Kernel に翻訳することが成功したかどうかの情報を示す	
inform*[=level*]	指定した *level* 以上のエラー・メッセージをを表示するように指示。 fatal ： fatal error messages. severe ： severe and fatal error messages. warn ： warning, severe and fatal error messages inform ： all error messages (inform, warn, severe and fatal)	その他
inline [=*func* | *filename.ext* | *number* | *levels:number*],...	関数のインライン展開を行う。以下のサブ・フラグがありますので、詳細は User's Guide を参照のこと。 except:func ： IPA(-Mipa) のインライン機能にも影響する。 [name:]func filename.ext Number levels:number (PGI 17.1廃止） totalsize:n、maxsize:n （PGI 17.1以降） PGI 7.1以降） 配列の形態(Array shape) が一致しない場合でも Fortran におけるインライン処理を許可（抑止）する。-Mconcur あるいは -mp の場合を除いたデフォルトは、-Minline=noreshape。-Mconcur あるいは -mp の場合のデフォルトは、Minline=reshape。	インライン化
instrument [=functions]	(PGI 9.0以降、linux86-64 にみ) Common Compiler Feedback Format (CCFF)を使用して、 PGI コンパイラは、どのようにプログラムの最適化を行ったら良いか、あるいは、特定の最適化がなされないのか等の関数レベルの instrument 情報をオブジェクトに保持することを可能とする。-Minstrument=functions の指定も -Minstrumen tと同じ意味なる。このオプションは、-Minfo=ccff -Mframe の二つを指定したことと同意です。	その他
ipa[=*flags*]	関数、サブルーチン間のグローバルな最適化を行うために、内部手続き間の最適化を行うように指示。version 5.2 から 1パスで行うことが可能。この ipa は同時に -O2 のレベルで行うことを前提にしている。以下のサブ・フラグ *flags* の詳細は、User's Guide を参照のこと。**一般的には、-Mipa=fast を指定すると良い**。 [no]align [no]arg [no]const Interprocedural constant propagation [no]cg except:<func> [no]f90ptr fast force [no]globals inline:<n> inline ipofile	最適化

	[no]keepobj [no]libc [no]libinline [no]libopt [no]localarg main:<func> noerror [no]ptr [no]pure required safe:[<function>|<library>] [no]safeall [no]shape summary [no]vestigial -Mipa Default enables constant propagation **複合フラグ fast の意味は** -Mipa=align,arg,const,f90ptr,shape,globals,localarg,ptr **PGI6.0 New flag:** -Mipa[=…,safe:<libname>,safeall,…] – IPA機能を使用してコンパイルして生成していない、ライブラリ名 libname の中のプログラムユニットへの呼び出しが安全であると仮定する、あるいは呼び出し側におけるIPA最適化を禁止しないことをコンパイラに指示するためのオプションです。-Mipa=safeall は実行モジュールの中にリンクされる全てのライブラリが安全であることコンパイラに指示します。 **PGI 6.1 New flag:** -Mipa=cg スイッチを設定することで、プログラムのコール・グラフ情報を出力できるようになりました。これは、新規に提供されたpgicgコマンド・ユーティリティを使用して、出力可能です。 -Mipa=except:<func> – IPA最適化において、インラインすべきでない関数funcを指定します。-Mipa=inlineと共に指定します。デフォルトは、内部的に検出されたすべての関数がインライン対象となります。 **PGI 6.2 New flag:** -Mipa=[no]libc は、システム標準Cライブラリ内で、あるルーチンへの呼び出しを最適化するために使用します。-fastオプション時のデフォルトは-Mipa=libcです。 nolibcは、その機能を抑止します。 **PGI 7.1 New flag:** -Mipa=[no]reshape は、配列の形態(Array shape) が一致しない場合でも Fortran におけるインライン処理を許可（抑止）します。 **PGI 7.2 New flag:** -Mipa=jobs:<n>– jobs:[n] サブオプションを指定できるようになりました。このサブオプションは、並列に n ジョブで再コンパイルを行うように指示するものです。 **PGI 9.0 New flag:** -Mipa=nopfo は、プロファイル・フィードバック情報の引用回数情報を無視する。このサブオプションは、inlineサブオプションの次に指定されているときのみ有効。-Mipa=inline,nopfo は、IPA手続きに対して、PFO情報が有効な状態において、インラインされる関数を決める際に、PFO情報を無視するように伝えます。 **PGI 13.1 New flag:** -Mipa=reaggregation IPA guided structure reaggregation を行います。自動的に struct の要素の並べ替え、あるいは、メモリやキャッシュの利用向上のために struct を substruct に分離する等の処理を行います。	
noipa	内部手続き間解析と最適化機能を抑制します。機能複合オプションの後に、このオプションを指定した場合、他の機能に関しては影響せず、IPA最適化のみを抑制することができます。(PGI 6.0)	最適化
[no]iomutex	（pgf77、pgfortranとpghpfのみ）クリティカルセクションがFortran I/Oコールの周辺で生成されるかどうかを決定。	Fortran言語
[no]large_arrays	(**64bit環境,**) 配列添え字（インデックス）を 64 ビット整数で扱えるように変更します。この意味は、必要に応じて、64 ビット整数変数あるいは定数が、インデックスの計算において使用されることを意味します。但し、コンパイラが暗黙に 32 ビット整数から 64 ビット整数に変更することによって、思わぬところで副作用が現れるかもしれないことに注意してください。一般に、64 ビット・アドレッシングが必要なインデックス変数は、明示的に 64 ビット整数宣言をすることが最も安全な方法で、これ行ってかつ、このオプ	コード生成

<table>
<tr><td></td><td>ションを指定することを推奨します。
さらに、Linux 環境下では 2GB を超える「単一の静的なデータオブジェクト」を扱うことができるコードを生成します。PGI 5.2 の場合は、pgfortran, PGF77, PGCC でサポートします。一般的には、-mcmodel=medeium と同時に使用します。PG 6.0 以降では、2GB以上の単一の静的なデータオブジェクトをサポートするために指定する有効化（無効化）フラグです。pgfortran, PGF77, PGCC, PGC++ の言語でサポートします。なお、このオプションは、PGI 6.0 から　Linux の -mcmodel=medium の複合オプションの中に加えられました。2GB以上の単一の静的なデータオブジェクトを使用するアプリケーションでは必要とされるオプションです。</td><td></td></tr>
<tr><td>largeaddressaware[=no]</td><td>(PGI 7.2新設：Windowsのみ)　Windows x64 用に 2GB 以上のアドレス・インデックシングを Windows のリンカーへ指示します。（RIP-relative　addressingを使用する）。デフォルトは、no で、direct addressing 形式となります。</td><td>その他</td></tr>
<tr><td>[no]loop32</td><td>(PGI 7.1 以降)
barcelona上での 32-byte 境界上にある最内側ループを整列します。barcelona上で 32-byte 境界で整列されている場合、小さなループは性能が向上する可能性があります。しかし、実際には、ほとんどのアセンブラが、まだ効果的なパッディング(padding)を実装していません。その結果、このオプションで遅くなる可能性もあります。Barcelonaに対して最適化されたアセンブラを有するシステム上でこのオプションを使用してください。デフォルトは、-Mnoloop32 です。</td><td>最適化</td></tr>
<tr><td>lsf</td><td>(32-bit Linux) 32ビットシステム上で 2GB 以上のファイル　I/O を扱うためのライブラリをリンクする。</td><td>環境</td></tr>
<tr><td>lre[=array | assoc | noassoc]
[no]lre</td><td>ループ内での冗長性を削除する最適化の有効化 [無効化]。
array　：個々の配列要素の参照を冗長性削減の対象として扱う。デフォルトは、2以上のオペランドを含む冗長式のみが対象となる。
assoc　：冗長性削減の対象を増やすことができる、演算式の再結合を許す最適化。結果の差異が生じる可能性がある。
noassoc：上記を許さない最適化</td><td>最適化</td></tr>
<tr><td>keepasm</td><td>アセンブリファイルを保持するようにコンパイラに命令。ファイル名は、<filenema>.s 。</td><td>その他</td></tr>
<tr><td>[no]list</td><td>コンパイラがリスティング・ファイルを作成するかどうかを指定。ファイル名は、<filenema>.lst　。</td><td>その他</td></tr>
<tr><td>[no]m128</td><td>(PGI 9.0 新設　pgcc のみ)
__m128, __m128d, __m128iデータ型を認識するためオプション</td><td>その他</td></tr>
<tr><td>makedll[=export_all]</td><td>(Windows only) Dynamic Link Library (DLL) を生成する。=export_all は、DLL内の全ての関数をエクスポートする。Winodws 上での DLL の作成に関しては、PGI User's Guide の 8 章を参考のこと。
(PGI 7.1 以降)　-Mmakedll オプションは、-Mdynamic オプションを内包します。</td><td>その他</td></tr>
<tr><td>makeimpdll[=export_all]</td><td>(Windows only) DLL を生成することなしに、import ライブラリを生成する。=export_all は、DLL内の全ての関数をエクスポートする。</td><td>その他</td></tr>
<tr><td>makeimplib</td><td>(Windows only：PGI 7.0 以降) DLLを生成することなしに、インポートライブラリを生成します。これは、まだ、それ自身のDLLライブラリが構築される前に、DLLのためにインポートライブラリを生成したい時に使用します。</td><td>その他</td></tr>
<tr><td>mpi[=option]</td><td>(MPI 使用可能ライセンスのみ：PGI 7.1 以降)
プログラムのビルドに使用する MPI ライブラリの指定を行う。
<table>
<tr><th>使用するライブラリ</th><th>コンパイル・リンクに必要なオプション</th></tr>
<tr><td>MPICH1</td><td>-Mmpi=mpich1 は　PGI 13.10 以前有効、PGI 14.1以降廃止、必要な場合は、MPIDIR環境変数にそのディレクトリをセット</td></tr>
<tr><td>MPICH2</td><td>-Mmpi=mpich1 は　PGI 13.10 以前有効、PGI 14.1以降廃止、必要な場合は、MPIDIR環境変数にそのディレクトリをセット</td></tr>
</table></td><td>コード生成</td></tr>
</table>

	MPICH v3 -Mmpi=mpich (PGI 14.1 以降) MS-MPI -Mmpi=msmpi （Windows) MVAPICH1 (CDK) -Mmpi=mvapich1 は PGI 13.10 以前有効、PGI 14.1以降廃止、必要な場合は、MPIDIR環境変数にそのディレクトリをセット MVAPICH2 (CDK) MVAPICH2 の mpif90,mpicc等のラッパーを使用する Open MPI (CDK) Open MPI の mpif90,mpicc等のラッパーを使用する SGI MPI -Mmpi=sgimpi （PGI 13.5 以降）	
neginfo[=*flags*]	なぜ、最適化が行われないかに関する情報を生成するようにコンパイラに指示。 all : 全てのメッセージ出力 concur : 自動並列化できない理由 loop : メモリ階層型の最適化ができない理由 (PGI 8.0 以降) all 以下のサブオプションをすべて指定したものと解釈します。 -Mneginfo=accel,inline,ipa,loop,lre,mp,opt,par,vect accel アクセレータ情報の有効化 ftn Fortran特有な情報の有効化 hpf HPF特有な情報の有効化 information inline インライン情報の有効化 ipa IPA i情報の有効化 lre LRE情報の有効化 mp OpenMP情報の有効化 opt 最適化の情報の有効化 par 並列化の情報の有効化 pfo プロファイル・フィードバックに関する情報の有効化 vect ベクトル化の情報の有効化	その他
names=lowercase|uppercase	Fortran外部関数名の大文字/小文字を指定する。Lowercaseの場合は、小文字を使用すると言う意味となり、upperecaseは大文字を使用すると言う意味となる。（PGI 7.2新設）	その他
noframe	関数の真のスタック・フレームポインタのセットアップ処理を消去するように指示。このオプションを有効化すると traceback 機能を使用することができない。	最適化
nomain	（pgf77、pgfortranとpghpfのみ）リンクステップ時に、Fortran のメインプログラムを呼ぶオブジェクトファイルを含めない形でリンクする。C プログラムと Fortran プログラムのオブジェクトの混在したものをリンクする時、C プログラムにメインプログラムが存在している場合で、かつ pgf77, pgfortran でリンクする時に使用する。	コード生成
[no]movnt	non-temporal ストア並びにプリフェッチの生成を強制するオプション。これまで使用してきた -Mnontemporal を置き換えるものです。(PGI 6.1)	コード生成
nontemporal	non-temporal ストア並びにプリフェッチの生成を強制するオプション。	最適化
noopenmp	-mp オプションと同時に使用した場合、強制的に OpenMP directives を無視するようにコンパイラに指示する。但し、SGI スタイルの並列 directive は解釈する。	その他
[no]prefetch （PGI5.2まで) [no]prefetch [=d:<m>[,n:<p>[,{nta | t0 |w}]]] (PGI6.0以降)	prefetch インストラクションの生成を有効化/無効化する。-Mvect（−fastsse) オプションと共に使用する。 **PG 6.0 以降**では、メモリデータのプリフェッチ命令を生成することを有効化（無効化）します。このオプションは、-Mvect あるいは、-Mfastsse（-Mvectを含む複合オプション）と組み合わせて使用する場合のみ有効です。新たなサブオプションである、d:<m> **距離サブフラグ**は、アクセスしようとするデータの先にある m キャッシュラインの長さをプリフェッチするようにコンパイラに指示します。n:<p> **数サブフラグ**は、プリフェッチが使用されている場所において、最大 p プリフェッチ命令まで出すことができるようにコンパイラに指示するものです。また、新たな nta | t0 | w の各サブオプションは、プリフェッチのために、prefetchnta、prefetch0、prefetchw 命令を使うようにコンパイラに指示するものです。なお、prefetchw は、IA32 あるいは EM64T プロセッサ	最適化

	ではサポートしません。	
nopgdllmain	(Windows only) デフォルトの DllMain() を DLL の中に含んでいるモジュールをリンクしない。このフラグは、pgfortran による DLL の構築に対して適用される。	その他
norpath	(Linux only) PGI の 共有ライブラリ・オブジェクトを含むディレクトリパス名を -rpath オプションをリンク時のコマンド行に付加しない。（デフォルトは -Mrpath で含む）	その他
nosgimp	-mp オプションと同時に使用した場合、強制的に SGI スタイルの並列 directive を無視するようにコンパイラに指示する。但し、OpenMP directives は解釈する。	その他
nostartup	（pgf77、pgfortranとpghpfのみ）標準のスタートアップルーチンをリンクしない。	環境
nostddef	標準のプリプロセッサマクロを認識しないようにコンパイラに指示。	環境
nostdinc	インクルードファイルの標準の場所を検索しないようにコンパイラに指示。	環境
nostdlib	標準のライブラリをリンクしないようにリンカに指示。	環境
[no]onetrip	（pgf77、pgfortranとpghpfのみ）各 DOループが少なくとも 1 回実行させるかさせないかの指示。	言語
novintr	イディオム認識を抑制し、最適化されたベクトル関数の呼び出しを行う。	最適化
pfi	-Mpfo 最適化オプションを含む後続のコンパイル時において使用されるプロファイルとデータ・フィードバック情報を集めるための実行モジュールを生成するためのオプションです。-Mpfi を伴う実行モジュールはこの情報を集積するためのオーバヘッドが発生するため、実行時間が多く掛かります。(PGI 6.0) （PGI 7.2 新設） -Mpfi[=indirect] -Mpfiオプションは、間接的(indirect)な関数呼び出しターゲットを保持することを指示する	最適化
pfo	強化されたブロック・リオーダリング機能を含む特定の性能最適化を有効にするために、pgfi.outプロファイル・フィードバック・トレースファイルのデータを使用して最適化を行います。(PGI 6.0) （PGI 7.2 新設） -Mpfo[=indirect|nolayout]　Indirect サブオプションは、間接的な関数呼び出しのインライン化を有効にするもので、nolayout は、動的なコード配置を抑止する	最適化
pre[=all]、nopre	（PGI 7.2新設）サブオプションを付けない -Mpre オプションは、一部の冗長部削除を有効にする。サブオプション all を付けた場合、よりアグレッシブな pre 処理を行う。 （PGI 9.0 以降） =all サブオプションが廃止された。 (PGI 2010以降) -Mnopre 冗長部削除の最適化を抑止する。	最適化
preprocess	cpp 形式の前処理をアセンブラ言語と Fortran ソースファイル上で行う。 代わりとして -cpp オプションを新設 （PGI 17.1)	その他
prof[=flags[,flags,.]]	プロファイルオプションをセット。関数レベルと、行レベルのプロファイリングがサポートされます。-Mprof=func、あるいは -Mprof=lines を指定する。これは、リンク時にも指定が必要である（特に Makefile 等でコンパイルとリンク処理を別々に行う際に注意） **PGI 6.0 New feature:** プロファイル・オプションをセットします。-ql, -qp, -pg スイッチは、通常、プロファイルのために使用されますが、プロファイリングのデフォルトの方法を再セット（上書き）するために以下のオプションを指定します。詳細は、PGI User's Guide をご覧ください。 **dwarf**：サードパーティのプロファイリング・ツールによって、ソース相関を有効にするため、DWARF 情報を生成する。 **func**：PGI スタイルの関数レベルのプロファイリングを実行する **hwcts**：ハードウェア・カウンタを用いた PAPI ベースのプロファイリングを使用する場合に指定する （linux x86-64ベースのシステムのみ） **lines**：PGI スタイルのソースレベルのプロファイリングを実行する	コード生成

	time：サンプリングベースのインストラクション・ベースのプロファイリングを実行 （PGI 7.1 以降） プロファイルするアプリケーションにリンクするための MPI ライブラリ名を、-Mprof オプションに指定する。 使用するライブラリ / コンパイル・リンクに必要なオプション MPICH1 / -Mmpi=mpich1 は　PGI 13.10 以前有効、PGI 14.1以降廃止、-Mprof=mpich1,{func|lines|time} MPICH2 / -Mmpi=mpich1 は　PGI 13.10 以前有効、PGI 14.1以降廃止、-Mprof =mpich2,{func|lines|time} MPICH v3 / -Mprof=mpich,{func|lines|time} (PGI 14.1 以降) MS-MPI / -Mprof =msmpi,{func|lines}　（Windows) MVAPICH1 (CDK) / -Mmpi=mvapich1 は　PGI 13.10 以前有効、PGI 14.1以降廃止、-Mprof=mvapich1,{func|lines|time} MVAPICH2 (CDK) / MVAPICH2 の mpif90,mpicc等のラッパーを使用する -profile={profcc|proffer} -Mprof ={func|lines|time} Open MPI (CDK) / Open MPI の mpif90,mpicc等のラッパーを使用する -Mprof ={func|lines|time} SGI MPI / -Mprof =sgimpi,{func|lines|time}（PGI 13.5 以降） （PGI 8.0以降） [no]ccff：CCFF情報の有効化 [無効化]	
[no]propcond	(PGI 7.1 新設) equality conditionalsから派生するassertions からのconstant propagation 最適化を有効にします。これは、デフォルトで有効となります。	最適化
[no]r8	（pgf77、pgfortranとpghpfのみ）コンパイラが REAL 変数と定数をDOUBLE PRECISION に変換する（しない）。	最適化
[no]r8intrinsics	（pgf77、pgfortranとpghpfのみ）コンパイラが 組込み関数の CMPLX and　REAL 型を DCMPLX and DBLEとして扱す（扱わない）	最適化
[no]recursive	（pgf77、pgfortranとpghpfのみ）ローカル変数をスタックに割当てます（割当てません）。これは再帰を可能にします。SAVEされた、データ初期化された、または、namelist メンバは、このスイッチの設定に関係なく常にスタティックに割当てられます。	コード生成
[no]reentrant	コンパイラがコードをリエントラントとしない最適化を回避するかどうかを指定。	コード生成
[no]ref_externals	（pgf77、pgfortranとpghpfのみ） EXTERNAL 文に現れる名前の参照を強制（強制しない）。	コード生成
safeptr[=*flags*]	（pgcc と pgCC のみ）ポインタと配列の間のデータ依存関係を以下のサブフラグの内容でオーバライドするようにコンパイラに指示します。以下のサブ・フラグ *flags* の詳細は、User's Guide を参照のこと。 all　all is safe arg　Argument pointers are safe auto　Local pointers are safe dummy　Argument pointers are safe local　Local pointers are safe static　Static local pointers are safe global Global pointers are safe -Msafeptr All pointers are safe	最適化

safe_lastval	スカラがループの後で使用され、しかし、ループの全ての反復に関しては定義されない場合、コンパイラはデフォルトではループを並列化しません。しかし、このオプションは、コンパイラにループを並列化することが安全であると告げます。特定のループについて、全てのスカラの最後に計算された値がループの並列化を安全にします。	コード生成
[no]**save**	（pgf77、pgfortranとpghpfのみ）コンパイラが全てのローカルな変数がSAVEステートメントと同等な状況に強いるように仮定するかどうかを決定。	Fortran言語
[no]**scalarsse**	スカラの浮動小数点演算において、xmm レジスタを使用した SSE/SSE2 のインストラクションを使用するか否かを指示。このオプションは、-tp { p7 / p7-64 / k8-32 / k8-64 以降の target } 時に有効。	最適化
scalapack	(PGI 14.1 以降 Linux / OS X 版のみ）バンドルされた MPICH 3.0.4 と共に使用し、分散メモリ用の LAPACK ライブラリ（ScaLapak) のリンクを有効にする。	ライブラリ
schar	（pgccとpgc++） "plain" character を signed char ととして扱う。--uchar を参照。	C / C++言語
[no]second_underscore	（pgf77、pgfortranとpghpfのみ） Fortran のグローバルなシンボル名が、既にその名前の suffix にアンダースコアを有しているものが存在している場合に、内部シンボル名として 2 つめのアンダースコアを加えます（加えません）。Fortran Module 間のシンボル名の競合が起きる場合にも便利です。また、g77プログラムとのリンク時に有効です。	コード生成
[no]signextend	コンパイラがサインビットを拡張するかどうかを指定します。	コード生成
[no]single	（pgccとpgc++） float パラメータを double パラメータキャラクタに変換するかどうかを指示。	C / C++言語
nosizelimit sizelimit:n	ベクトライザにループ中のステートメント数に拘らず、全てのループに対してベクトル化最適化の対象とするように指示する。 PGI 6.2 から nosizelimit が、デフォルトとなった。一方、そのステートメントのサイズは、-Mvect=sizelimit:n （nはループ内のステートメントの数）によって制限される。	最適化
[no]smart	AMD64専用　post-pass instruction スケジューリングを行うか否かのスイッチ。(デフォルトは no)	最適化
[no]smartalloc[=option]	メインルーチン中に最適化された mallopt ルーチンのコールを加えます。これを有効にするためには、Fortran、C、C++のメインプログラムを含むファイルをコンパイルする際に、このオプションを付する必要がある。デフォルトは、-Mnosmartalloc。（PGI 6.2 以降） **PGI 7.1 New feature:** -Msmartalloc オプションは、Linux 並びに Windows 上での large TLBs をサポートするために強化されました。このオプションは、最適な malloc ルーチンを有効にするために、メイン・プログラムをコンパイルする際に使用することが必要です。サブ・オプション huge は、シングルプログラムで使用される大きな 2MB ページを有効にするために指定します。これは、実行するために必要な TLB エントリ数を削減する効果があります。このオプションは、AMDの Barcelona やインテル(r)の Core2 システムで特に有効です。古いプロセッサ・アーキテクチャでは、TLB エントリの数が少ないため、大きな効果は期待できない可能性もあります。サポートするサブ・オプションは、以下のとおりです。 huge　　：huge page のランタイムライブラリをリンクします。 huge:<n>　：使用されるページの数の限度を n に設定します。 hugebss　：huge page の中に BSS セクションを置きます。 huge　サブ・オプションは、それ自身、必要とされる huge page をアロケートしようとします。Huge page の数は、:n サブ・オプションで制限を設けることができ、あるいは、環境変数 PGI_HUGE_SIZE でも設定できます。Hugebss は、プログラムの初期化されていないデータセクションを huge page の中に置きます。 (PGI 8.0 以降) hugebss　：　hugeページの中にBSSセクションを置く (PGI 9.0 新設)	環境

	nohuge： -Msmartalloc=hugeを上書き（無効化）するサブオプション	
[no]stack_arrays	自動配列(Automatic Array)をスタック上に配置します。(PGI 13.1 以降) -Mnostack_arrays は、従来通り自動配列をヒープ上に配置します。従来からの互換性を維持するために-Mnostack_arrays がデフォルトです。	Fortran言語
standard	（pgf77、pgfortranとpghpfのみ） ANSI 標準に適合しないソースコードを検出します。 （PGI 7.1 以降） -Mstandard は、-Mbackslash を内包しました。これは、-Mstandard が現れたときにバックスラッシュ・エスケープ・シーケンスを認識することを禁止します。例えば、バックスラッシュは標準的なキャラクタとして扱います。	Fortran言語
[no]stride0	（pgf77、pgfortranとpghpfのみ）コンパイラは、増分がゼロであるかもしれない誘導変数を含むループのために代替のコードを生成します（生成しません）。	コード生成
[no]traceback	環境変数 $PGI_TERMを使用することにより、ランタイム traceback のためにデバッグ情報が追加されました。また、デフォルトでのトレースバック機能は、f77、f90/f95では有効となっておりますが、C/C++では無効となっています。コンパイラへの初期設定ファイル siterc あるいは、.mypg*rc ファイルに TRACEBACK=OFF をセットすることで、デフォルトのレースバック機能を無効にすることができます。反対に OFF の代わりに ON と指定することによって、有効にすることができます。	最適化制御
uchar	（pgccとpgc++） "plain character" を unsigned char として扱う。-- scharも参照。	C/C++言語
unix	（pgf77、pgfortran for Win32) Fortran サブプログラムに対して、UNIX の呼び出し、名前のコンベンションを使用することを指示。	コード生成
[no]unixlogical	（pgf77、pgfortranとpghpfのみ）論理値 .TRUE. と .FALSE. が、unixlogical 非ゼロ（TRUE）、ゼロ（FALSE） と決定されるかどうかを決定します。デフォルトの unixlogical では、none-zero 値がTRUE で、0 の値が FALSE です。nounixlogical は、VMS convention スタイルを使用する。	Fortran言語
[no]**unroll**[=*flags*]	アンロール展開を制御。-Munroll=flags と言う形態でサブフラグを設定できる。以下のサブ・フラグ *flags* の詳細は、User's Guide を参照のこと。 c：***m*** n：***u*** **PGI 7.1 New feature:** -M[no]unroll[=c:<n>|n:<n>|m:<n>] ： マルチ・ブロックを持ったループをアンロールする機能を追加しました。特に、条件文を伴ったこのようなループで、アンロールできるようになりました。新しいオプション -Munroll="m" は、この機能を制御するために導入されました。 n:<n>：シングル・ブロックを n 回アンロール m:<n>：マルチ・ブロックを n 回アンロール デフォルトでは、-Munroll=m は有効となっておりません。また、-Munroll=m の場合のデフォルトの n 値は 4 です。	最適化
[no]upcase	（pgf77、pgfortranとpghpfのみ）コンパイラがプログラム識別子に大文字を許すかどうかを決定します。upcase の場合、大文字も識別されます。デフォルトは、noupcase で全てが小文字として識別されます。特に、リンク時のモジュール名の識別において重要です。	Fortran言語
unsafe_par_align	並列化ループでの配列の参照において、その配列の最初の要素が「整列」されている限り、「整列移動（aligned moves)」を行うことは安全であるとみなします。 NOTE: このオプションは、コンパイラがその安全性を疑った場合でも、「整列移動」で行うコードを生成します。このオプションは、特に、 STREAM Benchmark やメモリ・インテンシブなメモリアクセスを含むループブロックの並列化で効果を発揮します。	最適化
vect	コードベクタライザを起動。プログラムのベクトル化を行います。-Mvect の指示だけでも良い。以下のサブ・フラグ *flags* の詳細は、User's Guide を参照のこと。 altcode:***n*** / noaltcode ： 代替スカラコードの生成 assoc / noassoc ： ループの結合の許可	最適化

	cachesize:***n*** ： cache tileing の最適化における cache size の仮定 nosizelimit ： 全てのループに対して、そのソースコード数の制限なしで、ベクトル化の適用を行うように指示する prefetch ： ベクトル可能なコードの可能な限りの prefetch操作 smallvect[:n] ： 最大のベクトル長の定義 sse ： SSE/SSE2 インストラクションの使用によるベクトル化 (PGI 7.1 以降) gether ： 配列のgather（ギャザー）間接参照を有するループのベクトル化ができるようになりました。コンパイラのデフォルトは、-Mvect=gatherです。 (PGI 7.2 新設) partial ： 最内側ループの分離によるループのベクトル化を有効にするように指示するサブオプション （PGI 8.0 以降） [no]short ： 短いベクトル演算を有効化[無効化] -Mvect=short は、ループ外のスカラコードから生じる、あるいは、ループ・イテレーションの中から生じる短ベクトル演算のためのパックSSE演算の生成を有効化します。 (PGI 11.6 新設) simd:{128|256} ： SIMD命令とデータを使用してベクトル化する際、そのデータ幅を 128bit / 256bit のどちらを使用するかを選択する。256bit を使用できるかはプロセッサに依存する。	
novector	ベクトル化を抑制します。-fastsse のような機能複合オプションの後に、このオプションを指定した場合、他の機能に関しては影響せず、ベクトル化のみを抑制することができます。(PGI 6.0)	最適化
novintr	コンパイラに、イディオム認識を実行しないように指示する、あるいは、手製の最適化ベクトル関数を導入することを指示する。	最適化
varargs	(pgf77 と pgfortran のみ) Fortran ユニットに対して、C ルーチンが vararg 型のインタフェースを有すると仮定する場合に指定します。	コード生成
writable-strings	(pgcc/pgc++/pgCC: PGI 7.2新設）書き込み可能なデータセグメント内に string constant をストアできるようにします。（注意）既存の-Xt並びに-Xsは、本オプションを含む	

-C と C++ 特有のオプション

オプション	記述
-alias=[ansi|traditional]	(PGI 7.1 以降) C、C++プログラムにおける、「型」ベースのポインタ・エイリアス規則に基づき、最適化方法を選択します。 ansi：ANSI C型ベースのポインタの一義化(disambiguation)を使用した最適化を有効化 traditional：型ベースのポインター義化を無効にする C コンパイラでは、デフォルトは -alias=ansi で、C++ においては、-alias=traditionalとしています。
-A	(pgc++) プログラムが Proposed ANSI C++ に合致していることを指示する
--no_alternative_tokens	(pgc++) 代替トークンの認識を Enable/disable する。 These are tokens that make it possible to write C++ without the use of the , , [,], #, &, and ^ and characters. The alternative tokens include the operator keywords (e.g., and, bitand, etc.) and digraphs. デフォルトは、..no_alternative_tokens.
-B	C ソース内における // を使用したC++ 形式のコメントを許可する。
-b	(pgc++) cfront2.1 互換でコンパイルを行う
-b3	(pgc++) cfront3.0 互換でコンパイルを行う。 See -babove.
-c11	C11言語を使用する (PGI 15.1以降)
-c1x	C11言語を使用する (PGI 15.1以降)
-c89	(pgccのみ） C ソース言語として、 C89 standard (C89) を使用する（PGI 6.2 以降） PGI 6.1 以前のデフォルト
-c8x	(pgccのみ） -c89 と同じ機能
-c99	(pgccのみ） C ソース言語として、 C99 standard (C99) を使用する（PGI 6.2 以降 のデフォルト）

-c9x	(pgccのみ) -c99 と同じ機能
--c++11	(pgc++のみ) C++11 言語を認識する(-std=c++11と同じ)
-[no]compress_names	(PGI 7.1 以降) C++ マングル名を 1024 キャラクタにフィットするように圧縮します。高度にネストされたテンプレート・パラメータは、非常時長い関数名が作成されるようになります。これらの長い名前は、古いアセンブラでは問題を引き起こす原因になります。現在のデフォルトは、-no_compress_names です。全ての C++ユーザコードは、このスイッチを使用する際に、再度コンパイルされなければなりません。PGI によって提供されるライブラリは、-compress_namesと共に動作します。
--[no]bool	(pgc++) bool の認識をするかどうかを指示する。デフォルトは、 --bool.
--[no]builtin	数学関数ルーチンをビルトインでコンパイルするかどうかを指示する。選択された数学ライブラリルーチンをコンパイル時にインライン化する。デフォルトは、--builtin. コンパイルオプションは -M[no]builtin
--cfron t _2 . 1	(pgc++) cfront version 2.1互換でコンパイルするかどうかを指示する。
--cfront_3.0	(pgc++) cfront version 3.0 互換でコンパイルするかどうかを指示する。
--c++[arg]	(PGI 2013以降、 pgc++のみ)C++ の規格を指定する。c++**14**、c++**11**、C++**0x**、C++**03**のいずれかを指定。例えば、c++11 は C++11 機能を解釈する。
--create_pch filename	(pgc++) filename を伴った プリコンパイルされたヘッダーファイルを生成する。
--dependencies	(pgc++)makefile 依存性を標準出力に出力する （-M を参照）。
--dependencies_to_ file *filename*	(pgc++)makefile 依存性を *filename* ファイルに出力する。
--diag_error tag	(pgc++)指定されたダイアグ・メッセージの標準的なエラーレベルの内容を tag を使用して上書きする。
--diag_remark tag	(pgc++) 指定されたダイアグ・メッセージの標準的なエラーレベルの内容を tag を使用して上書きする。
--diag_suppress tag	(pgc++) 指定されたダイアグ・メッセージの標準的なエラーレベルの内容を tag を使用して上書きする。
--diag_warning tag	(pgc++) 指定されたダイアグ・メッセージの標準的なエラーレベルの内容を tag を使用して上書きする。
--display_error_number	(pgc++) 生成されたダイアグ・メッセージの中にエラーメッセージ番号を表示する。
--enumber	(pgc++) C++ front-end error の数の上限を指定した数にセットする。
--[no_]exceptions	(pgc++) Disable/enable例外処理のサポートを許可するかどうかを指示する。 デフォルトは、 --exceptions このオプションは PGI 17.1 以降廃止
--gnu	(pgc++、PGI 2013以降) GNU 互換 C++ コンパイルモード。GNU C++ コンパイラとの互換性を維持するために、gnu ライブラリをリンクする。PGI 13.1 より、この機能を提供するために GNU互換 C++コンパイラ（コマンド名 pgc++)も別提供した。 PGI 17.1 以降廃止
--gnu_version	(pgc++、PGI 2010以降) コンパイル時に使用するGNU C++互換性をセットする。デフォルトは、最新のバージョン番号がセットされる。使用例 gnu 4.8.2 の場合： --gnu_version 040802。
--gnu_extensions	(pgc++) Linux system header files をコンパイルする必要がある “include next” のような GNU 拡張を許す。
--instantiation_dir	(pgc++) If --one_instantiation_per_object is used, define dirnameas the instantiation directory.
--[no]llalign	(pgc++) 整数の境界で、 long long integersの整列を行うかどうかを指示する。デフォルトは --llalign.
-M	make 依存性リストを生成する。
-MD	make 依存性リストを生成する。
-MD,filename	(pgc++) make 依存性リストを生成して、それらを filename へ出力する。
--microsoft_version	(pgCCのみ、PGI 2010以降) コンパイル時に使用するMicrosoft C++互換性をセットする。デフォルトは、最新のバージョン番号がセットされる。使用例： --microsoft_version 1.5。
--one_instantiation_per_object	(pgc++) 各 template instantiation (function or static data member) を個々のオブジェクトファイル上に置く。
--optk_allow_dollar_in_id_chars	(pgc++) 識別子としてドル記号を許す。

--pch	(pgc++) 自動的にプリコンパイルされたヘッダファイルを使用する、あるいは生成することを指示する。
--pch_dir directoryname	(pgc++) プリコンパイルされたヘッダファイルの置かれたディレクトリをサーチパスに加える。
--[no_]pch_messages	(pgc++) 現在のコンパイルフェーズで、プリコンパイルされたヘッダファイルが生成され/使用されたかと言うメッセージを表示するかどうかを指示する。
--pedantic	(pgc++)　PGI 8.0 新設　含まれたシステムヘッダーファイルに関わる警告メッセージを印刷
+p	(pgc++) 全ての anachronistic construct を抑制する。
-P	プリプロセスフェーズの後で止まり、プリプロセスされたファイルを filename.i　にセーブ。
--preinclude=<filename>	(pgc++) コンパイル時の始めにインクルードされるファイルの名前を指定する。このオプションは、システム依存のマクロ、型をセットする時に使われる。
--prelink_objects	(pgc++) このオプションが指定された場合、テンプレート・ライブラリにしようとするオブジェクト・セットのために、template　instantiations を作成する。
-std=c++11	(pgc++のみ)　C++11 言語を認識する(--c++11と同じ)
-t [arg]	テンプレート関数の instantiation を制御する。[arg] は以下の引数が存在する。 all local none used
--use_pch filename	(pgc++) 現在のコンパイルフェーズで、指定された名前のプリコンパイルされたヘッダファイルを使用する。
--[no_]using_std	(pgc++) 標準ヘッダーファイルがインクルードされた時に、std namespace の使用を暗黙に使用するか否かを指示する。
-X	(pgc++) クロス・リファレンス情報を生成し、指定されたファイルに書き込む。
-Xm	(pgc++) 名前として $ を許す。 PGI 7.1 以降、このオプションは削除されました。現在、ほとんどの場合、ドルサインは許可されております。
-Xs	(PGI 7.1以降)　C/C++ において、レガシーな標準モードを使用する。 これは、-alias=traditional オプションを内包します。
-Xt	(PGI 7.1以降)　C/C++ において、レガシーな移行モードを使用する。 これは、-alias=traditional オプションを内包します。
-xh	(pgc++) 例外処理を enable にする。
--zc_eh	(PGI 7.1以降) ゼロ・オーバーヘッド例外領域を生成します。本オプションは、実際の例外処理が起こるまで、例外ハンドリングのコストを遅らせる措置を行います。多くの例外領域を有しながら、あまり例外が起こらないプログラムでは、このためのコンパイル・オプションによって、ランタイム性能の向上に繋がるかもしれません。デフォルトは、--zc_eh を使用しませんが、その代わり、setjmp と longjmp と共に例外ハンドリングを実装する -sjlj_eh を使用します。このオプションは、PGI C++ の以前のバージョンでコンパイルされた C++ コードにも互換性があります。--zx_eh オプションは、libgcc_eh 内のシステム unwind ラ イブラリを提供している新しい Linux システムと Windows 上でのみ有効です。 このオプションは、PGI 2011(11.0)以降、C++コンパイラのデフォルトとなりました。 PGI 17.1 以降廃止
-*suffix* (see -P)	(pgc++) -E、-F、-P の機能で指定された中間ファイルをセーブする。

PGI 6.0 以降でのC++のテンプレートのインスタント化の変更について

C++ テンプレートのインスタント化は、32-bit 並びに 64-bit Linux システムにおいて変更されました。新しい方法では、全てのテンプレート参照を解決するためにGNUリンカーを使用し、重複を回避することでテンプレートの使用の単純化に大きな効果を発揮します。この新しい方法は、PGI コンパイラの前のバージョンと互換性はありません。 C++ プログラムを PGC++ 6.0 用に移行するためには、全ての C++ プログラムを再コンパイルすることが必要です、また、makefile 上の全てのテンプレート・インスタント化フラグを削除することが必要で

す。次のテンプレートのインスタント化に関係するコマンドオプションが削除する対象となります。

-one_instantiation_per_object
-instantiation_dir
-instantiate
-[no]auto_instantiation
-prelink_objects
-Wc, -tlocal
-Wc, -tused
-Wc, -tall