[大規模科学計算システム]

GDB を用いたデバッグ入門

山下 毅† 小野 敏† 伊藤 英一†

†東北大学 情報部 情報基盤課

1 はじめに

計算機を利用した数値計算シミュレーションは，コンピュータ性能が向上するとともに，ユーザーの高いニーズにも答えられるようになってきています．それとともにユーザーアプリケーションは複雑なアルゴリズムが組まれ，ユーザーがコーディングするソースファイルは大量で複雑になっています．コーディングが終了した後，コンパイルエラーを全て修正しいざ実行となると，バグによる実行時エラーの発生に悩まされた経験がほとんどの方にあるでしょう．機能の拡張や，ソースファイルの管理を柔軟にするために，ソースファイルはサブルーチン毎に分割されることが一般的です．しかしながら実行時エラーはコンパイルエラーと異なり，エラー位置を特定するための情報が乏しいので，ソースファイルが分割されているとバグの箇所を特定するために時間がかかってしまう，という悪影響もあります．そこで本稿では，サブルーチンが多数ある長いソースファイルや分割されたソースファイルでも，エラー位置の特定とバグの修正（デバッグ）にかかる時間を短縮するための便利なソフト”デバッガ”の初歩的な使用方法を紹介します．

2 GDB とは

GDB(GNU Project debugger) [1] は，Unix プログラマの間で最も一般的に使われているデバッガ（プログラムの不具合，バグの発見や修正を支援するソフトウェア）です．GDB は GNU プロジェクトにより開発されたオープンソースウェアです．関連するソフトウェアとして，グラフィカルユーザインタフェースシステムの DDD や，統合開発環境（IDE）の Eclipse からも GDB の機能を使用することが出来ますが，本稿ではコマンドラインからデバッグを行う GDB の初歩的な操作方法について解説します．

2.1 GDB を使うメリット

デバッガを使用しないでも，もちろんデバッグ作業は可能です．C,C++ では printf 文や std 文，Fortran では print 文や write 文といった，「出力文」を使って，エラー位置の特定や変数値の確認ができます．出力文を使用したデバッグの場合，エラー位置の特定に多くの時間を費やした経験があると思います．それに対して，デバッガを使用した場合，プログラムリストの任意の場所に

「ブレークポイント」を設定して，プログラムを一時停止することでエラー位置の特定ができ，また任意の変数の値を逐一確認することができます．特にサブルーチンやソースファイルが大量である場合，デバッガの使用はエラーを引き起こすプログラムの矛盾などのバグの発見と修正に威力を発揮します．

2.2 サポート言語

GDB は C, C++, D, Objective-C, Fortran, Java, OpenCL C, Pascal, assembly, Modula-2, Ada の各言語に対応しています [1]．サイバーサイエンスセンターでは並列コンピュータ (gen.isc.tohoku.ac.jp) にインストールされており，C, C++ および Fortran のデバッグに使用することができます*1．今回は並列コンピュータ上での Fortran のデバッグについて解説します．

3 GDB を使用したデバッグ作業例

3.1 例題プログラム

デバッグ作業を行う例として，リスト 1, 2 に Fortran のプログラムリストを示しました．このプログラムは，配列の各要素同士の掛け算を行いその結果を別の配列に代入します。その後，配列の一部分と，処理にかかった時間を出力します．なおこのプログラムはコンパイルは成功しますが，実行時にエラーとなるバグが 4 つ含まれています．

リスト 1　メインプログラム:main.f90

```
1       program main
2
3       implicit none
4       integer            :: I, J                          ! 変数型の定義
5       integer,parameter  :: M=10000, N=10                 ! 配列サイズの設定
6       double precision   :: A(M,N), B(M,N), C(M,N)        ! 配列の定義
7       real               :: start_time, end_time          ! 変数型の定義
8
9       call clock(start_time)                              ! システム時刻の取得（開始時刻）
10      do J=1,N
11        do I=1,M
12          A(I,J)=I*J*1.0D0                                ! 配列の初期値設定
13          B(I,J)=sqrt(I*J*1.0D0)
14        enddo
15      enddo
16      call sub(A,B,C,M,N)                                 ! サブルーチンの呼び出し
17      call clock(end_time)                                ! システム時刻の取得（終了時刻）
18      write(*,'(5F10.3)') C(1:5,100)                      ! 計算後の配列の値を表示
19      write(*,*)'user time = ',end_time-start_time        ! 経過時間の表示
20      write(*,*)'Program has been finished normally.'     ! 正常終了時のメッセージ表示
21
22      end program
```

*1 ベクトル型スーパーコンピュータ (SX-9) 上では NEC 製のデバッガ，DBX/PDBX が動作します．SSH クライアントプログラムを使用し super.isc.tohoku.ac.jp にログイン後，今回解説した操作方法と同様に使用することができます．本稿では DBX/PDBX 向けの解説は省略します．

リスト 2　サブルーチン:sub.f90

```
1       subroutine sub(A,B,C,M,N)
2
3       implicit none
4       integer           :: I, J, M, N               ! 変数型の定義
5       double precision :: A(M,N), B(M,N), C(M,N)  ! 配列の定義
6
7       do J=1,M
8         do I=1,M
9           C(I,J)=A(I,J)*B(I,J)                     ! 配列の各要素同士の掛け算
10        enddo
11      enddo
12
13      end subroutine
```

3.2　通常のコンパイルと実行

始めに，以下のコマンドで通常のコンパイルを行ってみます．

```
$ f95 -o sample main.f90 sub.f90
```

コンパイラからのエラーメッセージは無く，正常にコンパイルできたようです．以下のコマンドで会話型形式で実行してみます[*2]．

```
$ ./sample
```

すると，「セグメントエラー」というエラーメッセージを出して，プログラムは正常終了しませんでした．このエラーメッセージだけでは，プログラムのどの位置にエラーがあるのかすぐには分かりません．では，GDB を使って実際にデバッグしてみます．

3.3　GDB を用いたデバッグ手順

3.3.1　デバッグ用コンパイル

GDB を使ってデバッグするためには，コンパイル時に-g オプションを付けます

```
$ f95 -g -o sample main.f90 sub.f90
```

3.3.2　GDB の起動

GDB を起動[*3]します．このとき，コンパイルした実行ファイル名 (sample) を引数にします．

[*2] このプログラムの実行時間は gen 上で数秒です．もし終了しない場合は Ctrl-C コマンドで強制終了するか，それでも終了しない場合は他の端末から以下のコマンドで強制終了してください．
$ ps aux | grep 利用者 ID
(ユーザーが実行中のプロセス ID が表示されます．)
$ kill -9 (sample が実行されているプロセス ID)

[*3] 全てのデバッグ作業が終わるまで gdb は終了しません．ソースファイルを編集する別の端末を開いてください．

```
$ gdb sample
```

3.3.3 プログラムの実行

プロンプトが (gdb) に変わり，GDB のコマンド待ち状態になります．始めにプログラムを GDB 上で実行してみます．コマンドは run (r)（カッコ内は省略形．以下同じ）です．

```
(gdb) run
```

会話型形式で実行した場合と同じように，以下のエラーメッセージを出してプログラムは正常終了しませんでした．

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000047fddd in __intel_new_memset ()
```

3.3.4 エラー位置の特定

別端末にエディタソフトでソースファイルを開いて内容を確認することもできますが，GDB ではプログラムをコマンド list(l) により表示させることができます．list コマンドの後に，関数名 (main または sub)，またはソースファイル名:行番号（main.f90:行番号または sub.f90:行番号）を指定すると，指定した行番号の前後 10 行分のリストが表示されます．

```
(gdb) list main
(gdb) l main.f90:12
```

コマンドを入力せずに Enter キーを押すと，続く 10 行分のリストが表示されます．[*4]

次に，プログラムの実行を中断させるブレークポイントを設定して，プログラムのどの行でエラーが発生しているかを特定します．ブレークポイントの設定にはコマンド breakpoint(b) を使い，行番号や関数名で指定します．口安として，main.f90 内の各 CALL 文の前にブレークポイントを設定してみます．

```
(gdb) list main
(gdb) brake 9
(gdb) b 16
(gdb) b 17
```

[*4] GDB ではコマンドを入力せずに Enter キーを押すと，直前のコマンドが繰り返されます．また，カーソルキーの上を押すと，過去に入力したコマンドをヒストリー入力することが出来ます．Tab キーによるコマンド補完もできます．

設定されたブレークポイントをコマンド info(i) を使って確認します．

```
(gdb) info breakpoint
```

以下のように設定したブレークポイントを確認できます．

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x00000000004031be	in main at main.f90:9
2	breakpoint	keep	y	0x0000000000403260	in main at main.f90:16
3	breakpoint	keep	y	0x0000000000403297	in main at main.f90:17

ブレークポイントを設定したので，プログラムを再実行します．

```
(gdb) r
```

プログラムの始めから実行する確認メッセージが出ますので，”y”を入力します．プログラムリスト 9 行目の call clock(start_time) の前で実行が一時停止します．

```
Breakpoint 1, sample () at main.f90:9
9                 call clock(start_time)
```

コマンド continue(c) で次のブレークポイントまで実行を進めます．プログラムリスト 16 行目で一時停止します．さらに次のブレークポイントまで実行を進めます．17 行目で一時停止します．さらに次のブレークポイントまで実行を進めます．

```
(gdb) continue
(gdb) c
(gdb) (Enter キーのみ)
```

17 行目を実行したときに，先程と同じエラーメッセージで実行が異常終了しました．17 行目の call clock(end_time) でエラーが発生しているようです．サブルーチン clock はコンパイラの組み込み関数なので，インテル Fortran コンパイラーのライブラリリファレンス [2] を参照すると，clock の戻り値はサイズ 8 の char 型配列とのことです[*5]．サブルーチン clock の引数に実数型で宣言した変数 (start_time,end_time) を渡していたので，エラーが発生したようです．そこで，プログラムリスト main.f90 の 9 行目と 17 行目のサブルーチン clock を，実数型で CPU 時間を返すサブルーチン cpu_time に変更します．別端末で main.f90 のソースファイルを修正し，変更を保存します．

[*5] 現在時刻を 00:00:00 の文字列で返します．

3.3.5 再コンパイルと再実行

先程と同様に，-g オプションを付けて再コンパイルします．GDB はデバッグ対象の実行ファイルが更新されたことを自動的に検知しますので，GDB を立ち上げ直す必要はありません．GDB 上で更新前のプログラムが実行中だったので，Ctrl-C で実行を中断します．更新後のプログラムが読み込まれたことを確認するため，プログラムリストを表示してみます．ファイル名を指定すると，そのファイルに関して list コマンドが有効になります．list コマンドの後に行番号を指定するだけでその前後の 10 行が表示されます．

```
(gdb) l main.f90:9
(gdb) l 17
```

ブレークポイントの設定は引き継がれていますので，コマンド delete(d) で全てのブレークポイントを削除します．

```
(gdb) delete
```

delete コマンドのみ入力すると全てのブレークポイントが削除されます．ブレークポイントを個別に削除する場合は，ブレークポイントの番号を指定します．確認のメッセージが表示されますので，y を入力します．ブレークポイントを 18 行目の write 文の前に設定して，run コマンドでプログラムを再実行します．

```
(gdb) b 18
(gdb) r
```

18 行目で一時停止しました．この行以前でのエラーは解消されたようです．コマンド next(n) で 1 行づつ実行を継続します．

```
(gdb) next
```

以下のエラーメッセージで実行が停止しました．

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000044dec1 in cvt_ieee_t_to_text_ex ()
```

18 行目の write 文にエラーがありそうなので，この行を詳しく見てみます．write 文で配列 C の一部分の値を表示させようとしていますが，宣言された配列サイズは 10000 × 10 ですが，配列 C の 2 次元目の 100 番目にアクセスしようとしています．そこで、プログラムリスト main.f90 の 18 行目の C(1:5,100) を C(1:5,10) と修正して再度-g オプションを付けてコンパイルし，GDB で再実行してみます．今度は GDB 上で正常に終了し，結果が表示されました．

3.3.6 変数値の確認

今まではデバッグ用のオプション-g を付けてコンパイルしていましたので，通常のコンパイルを行い，会話型形式で実行してみます[*6]．

```
$ f95 -o sample main.f90 sub.f90
$ ./sample
```

「セグメントエラー」により異常終了しました．メインルーチン main.f90 にはエラーがなさそうなので，サブルーチン sub.f90 について調査してみます．再度-g オプションを付けてコンパイルします．GDB は立ち上がったままですので，sub.f90 のプログラムリストを表示します．

```
(gdb) l sub.f90:1
```

この操作でブレークポイントを設定する対象ファイルが sub.f90 になります．

GDB 上で実行した場合では，サブルーチン sub の実行ではエラーの発生はありませんでしたので，ブレークポイントを行列計算後の 12 行目に設定します．

```
(gdb) b 12
(gdb) r
```

DO ループの変数 I,J の値を確認してみます．実行中の変数の値はコマンド print(p) で確認できます．引数に確認したい変数を記述します．DO ループの変数 I,J の値を確認します．

```
(gdb) print I
(gdb) p J
```

I は配列サイズ +1 の 1001 で終了しています．J も 1001 となっていますが，これは配列サイズの 10 を超えています．プログラムリスト sub.f90 の 7 行目の DO ループを見ると，DO J=1,M となっています．N と M を間違えているようです．プログラムリスト sub.f90 の 7 行目を DO J=1,M を DO J=1,N と修正して通常のコンパイルと実行をしてみます．今度は正常に実行が終了しました．これで 4 箇所のバグを修正するデバッグ作業が終了しました．

3.3.7 デバッガの終了

GDB を終了させるコマンドは quit(q) です．確認のメッセージが出ますので y を入力します．

```
(gdb) quit
```

[*6] オプション-g を付けたデバッグ用のコンパイルはコンパイラによる最適化が一部行われません．プログラムによっては本番の実行に用いると実行速度が遅くなることがあります．

表 1　今回使用した GDB コマンド

コマンド	省略形	引数	動作
gdb		実行ファイル名	GDB の起動
run	r	コマンドライン引数	デバッグ対象ファイルの実行
list	l	file[:line]/line	プログラムリストの表示
break	b	[file:]function/[file:]line	ブレークポイントの設定
info	i	feature	feature の情報を表示
		breakpoints(b)	ブレークポイント一覧を表示
continue	c		ブレークポイント停止後に実行を再開
next	n	[count]	ブレークポイント停止後に次の行を実行
delete	d	[breakpoints][range...]	ブレークポイントを削除
quit	q		GDB を終了

4　おわりに

本稿で使用した GDB コマンドを表 1 に示しました．GDB はマルチスレッドでのデバッグや，動的メモリの参照など様々な機能を持つ高機能なデバッガソフトです．本稿で解説した使い方はごく基本的なものですので，より便利で効率的な使い方を知りたい方は，下記に代表的な参考文献を上げましたのでご参照下さい．効率的なプログラム開発の手助けとして，ぜひデバッガソフトをご活用下さい．

参考文献

[1] GDB: The GNU Project Debugger, http://www.gnu.org/software/gdb/documentation/
[2] インテル Fortran ライブラリ・リファレンス, http://jp.xlsoft.com/documents/intel/for_lib.pdf
[3] 実践 デバッグ技法 GDB, DDD, Eclipse によるデバッキング, Norman Matloff, Peter Salzman, 訳:杣川愛三, オライリー・ジャパン (2009)
[4] Debug Hacks デバッグを極めるテクニック＆ツール, 吉岡弘隆, 大和一洋, 大岩尚宏, 安部東洋, 吉冂俊輔, オライリー・ジャパン (2009)
[5] GDB ハンドブック, Arnold Robbins, 訳:千住治郎, オライリー・ジャパン (2009)
[6] GDB を使った実践的デバッグ手法, CQ 出版株式会社 (2007)