C言語入門

目 次

ここでは，Unix 系 OS 上でのプログラムの作成について最低限知っておくべき事項をまとめる．研究室では Linux ディストリビューションの一つである Ubuntu を採用しているため，動作確認は Ubuntu 上で行っているが，Unix 系 OS では同じようにコンパイル実行できると考えている．

1　プログラムのコンパイル

C 言語は人間が読みやすいようにで記述されたものであり，コンピュータに理解させるためには機械語に翻訳しなければならない．この機械語への変換を一般にコンパイルという．Unix 系 OS では標準で gcc が搭載されているので，まず gcc によるコンパイルについて述べる．

1.1　単一ファイルに書かれたプログラムのコンパイル

例として以下のプログラムをファイル (test01.c) に作成する．

```
#include <stdio.h>
extern int main(int argc, char **argv)
{
    printf("Hello World!\n");
}
```

コンパイルは以下のように行う (#はプロンプトである)．

```
# gcc -o test01.out test01.c
```

「-o test01.out」は実行形式の名前を test01.out にするという指定である．

この実行結果は以下のようになる．

```
# ./test01.out
Hello World!
```

理工系のプログラムでは sin，cos などの関数を利用する場合があるが，基本的な関数に関しては数学ライブラリとして用意されている．例として以下のプログラムをファイル (test02.c) に作成する．

```
#include <stdio.h>
#include <math.h>
extern int main(int argc, char **argv)
{
    printf("cos(pi) = %lf\n", cos(M_PI));
}
```

コンパイルは以下のように行う (#はプロンプトである)．

```
# gcc -o test02.out test02.c  -lm
```

先ほどとの違いは，プログラムの初めに math.h をインクルードし，コンパイル時に 「-lm」をコンパイルオプションに加えて数学ライブラリをリンクすることである．

この実行結果は以下のようになる．

```
# ./test02.out
cos(pi) = -1.000000
```

1.2　複数ファイルに分けて書かれたプログラムのコンパイル

例として以下のプログラムをファイル (test02a.c, test02b.c) に作成する．

```
#include <stdio.h>
extern double func(double x);
```

```
extern int main(int argc, char **argv)
{
    double y;
    y = func(2.0);
    printf("y = %lf\n", y);
}
```

```
extern double func(double x)
{
    return x*x;
}
```

コンパイルは以下のように行う．必要なファイルを並べて記述するだけである．

```
#  gcc -o test03.out test03a.c test03b.c
```

この実行結果は以下のようになる．

```
# ./test03.out
y = 4.000000
```

しかしながら，大きなプログラムを作っているときに，ひとつのファイルを書き換える度に全てのファイルをコンパイルするのは非効率的である．この場合，個々のファイルをコンパイルしておいて，それらをリンクして実効形式を作成するのが一般的である．そのときの手順を以下に示す．

```
#  gcc -c test03a.c
#  gcc -c test03b.c
#  gcc -o test03.out test03a.o test03b.o
```

まず,「-c」オプションによりコンパイルだけを行いオブジェクト test03a.o，test03b.o を作成し，これらのオブジェクトファイルをリンクして test03.out を作成している．

例えば，プログラムの仕様変更により test03b.c を

```
#include <math.h>
extern double func(double x)
{
    return sin(x);
}
```

と書き換えた場合，(test03a.c はコンパイル済みとして) コンパイルは以下のように行う

```
#  gcc -c test03b.c
#  gcc -o test03.out test03a.o test03b.o -lm
```

sin 関数を利用することにしたので，コンパイル時に「-lm」により数学ライブラリを合わせてリンクしている．

1.3 make の利用

前小節の例では短いプログラムだったためプログラムをファイル分けする恩恵はあまりなく，2 つのファイルしかなかったため分割コンパイルの恩恵も少ない．ただし，大きなプログラムを作成するときには，可読性の観点から機能ごとにファイル分けすることは重要であり，分割コンパイルの恩恵も大きくなる．その一方で，ひとつ

ひとつのファイルを手作業でコンパイルしてリンクするのは煩わしい作業である．make を利用すると，Makefile に書かれた情報をもとに，この煩わしい作業を簡単に行うことができる．以下に Makefile の例を示す (以下の内容を Makefile という名前で保存する．行頭のインデントはスペースではなくタブであることに注意する).

```
PROG = test03.out
SRCS = test03a.c test03b.c
OBJS = $(SRCS:.c=.o)
CC = gcc
CFLAGS = -O3
LIBS = -lm
all: $(PROG)
$(PROG) : $(OBJS)
    $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
clean :
    /bin/rm $(PROG) $(OBJS)
```

このファイル (Makefile) を作成するとプログラムのコンパイルは以下のように行える．

```
# make
gcc -O3   -c -o test03a.o test03a.c
gcc -O3   -c -o test03b.o test03b.c
gcc -O3 -o test03.out test03a.o test03b.o -lm
```

先に示した手順を make コマンド一つで行える．Makefile の前半の「=」で結ばれている行は変数への値の代入である.「:」の前に書かれたものがターゲットであり，ただ make としたときは all が指定されたことになる.「:」で関係付けられている行はファイルの依存関係を示していて，左に書かれたものが右に書かれたものに依存していることを示している．make 実行時にファイルの作成された時刻を比較して，目的のファイルが依存関係にあるファイルよりも後に作成されていれば作成し直す必要はないが，先に作成されているならば依存関係にあるファイルの変更を反映するために目的のファイルを作り直す．ターゲットが指定されたときに行うことは次の行に書かれている (先頭はタブであることが必須).

例として，ターゲット all が指定されると， $(PROG) すなわち test03.out を作成しいようとする．このとき (次の行から) test03.out は $(OBJS) すなわち test03a.o と test03b.o に依存していることがわかる．また，test03a.o と test03b.o は暗黙の依存関係によりそれぞれ test03a.c と test03b.c に依存していることがわかっている．したがって，make コマンドの実行後以下のことが自動で行われる．

1. test03.out が test03a.o と test03b.o に依存していることを知る．
2. test03a.o が test03a.c に test03b.c が test03b.o に依存していることを知る．
3. それ以上の依存関係はないことを知る．
4. test03a.o と test03a.c の作成時間を調べ，test03a.c の方が新しければ test03a.o を作り直す．そうでなければ何もしない．
5. test03b.o と test03b.c の作成時間を調べ，test03b.c の方が新しければ test03b.o を作り直す．そうでなければ何もしない．
6. test03.out と test03a.o, test03b.o の作成時間を調べ test03.out よりも後に作られたファイルが 1 つでもあれば，その変更を反映するために test03.out を作り直す．そうでなければ何もしない．

以上の手順により本当にコンパイルし直す必要のあるファイルを自動で見極めて効率的にコンパイルをしてくれる．ここで示した Makefile には clean というターゲットが用意されていて make clean とするとターゲット clean に指定されたことを実行する．ここでは以下のようにオブジェクト (test03a.o, test03b.o) と実効形式 (test03.out) を消去している．

```
# make clean
/bin/rm test03.out test03a.o test03b.o
```

ここで示した Makefile の補足説明を以下に記す.

```
OBJS = $(SRCS:.c=.o)
```

は $(SRCS) で指定されたファイルの拡張子 .c を .o に書き換えたものを代入することを意味している.

test03a.o と test03b.o の作り方は Makefile の中には書かれていないように見えるが，暗黙の作成方法により作成される．その際

```
CFLAGS = -O3
```

のように CFLAGS に値を代入しておくことで，gcc によるコンパイルのときにオプションとして指定される (make 実行後の動作を確認すること)．この CFLAGS は予約された変数であり，この名前で使うこと．CC も予約後であり，作成方法を指定していないとき，CC に設定されたコンパイラが用いられる.

```
    $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
```

の「$@」はターゲットを表し，この場合には $(PROG) すなわち test03.out である.

ここで記載したことは make の機能のごく一部であり，さらに高度な機能については各自勉強されたい．参考までに，make はプログラムのコンパイル以外の用途にも使うことができる.

make コマンドは Makefile を自動的に読み込むが，これは makefile とい名前で作成しても良い．makefile と Makefile が両方あるときには makefile が優先される．また，計算機システムごとに Makefile を変更する必要がある場合には Makefile.FreeBSD, Makefile.Linux などとそれぞれのシステムに対して Makefile を作成しておき

```
make -f Makefile.FreeBSD
```

のように読み込むファイルを指定することもできる.

1.4 コンパイルに関する他の事項

1.4.1 strip

コンパイル後の実行形式は通常大きなファイルになる．通常，コンパイル後の実行形式にはデバッグなどのためにシンボルテーブルが付属している．strip コマンドを用いてこのシンボルテーブルを削り落とすことでファイルサイズを縮小できる．以下のことを確かめよ.

```
# ls -l test03.out
-rwxr-xr-x 1 tsuji tsuji 8673  8月 20 21:55 test03.out*
# strip test03.out
# ls -l test03.out
-rwxr-xr-x 1 tsuji tsuji 6288  8月 20 22:52 test03.out*
```

strip 実行前に比べてファイルサイズが小さくなっていることがわかる．この例は小さなプログラムなため効果が小さいが，大きなプログラムではその効果がより顕著になる．とは言え，最近はディスク容量が大きいので気にするほどでもないか...

プログラムが完成した後のオブジェクトファイル (*.o) は消去しておくほうがディスク容量を節約できるが，これも今となってはそれほど気にするほどのことでもないか...

1.4.2 ライブラリの作成

複数のプログラムで共通に使われるような関数はライブラリを作成しておいて，コンパイル時に既に作成しておいたライブラリをリンクすると便利である．複数のプログラムで使われている関数の改良を行うときに，高速化あるいは高機能化したその関数をいちいち各プログラムのディレクトリに改めてコピーする必要がない．ただし，関数の仕様を勝手に変えられてしまうとその関数のリンク時にエラーが生じたりするので注意が必要である．そのため，ライブラリにバージョン番号を付けて古いライブラリも残していたりする．

ライブラリの作り方の例を以下に示す．まず，test05a.c，test05b.c，test05c.c という以下の 3 つのファイルを作成する．

```
#include <test05.h>
extern int main(int argc, char **argv)
{
    if (argv[1][0] == 'A') funcA();
    else funcB();
}
```

```
#include <stdio.h>
extern int funcA()
{
    printf("Function A is used\n");
}
```

```
#include <stdio.h>
extern int funcB()
{
    printf("Function B is used\n");
}
```

さらに以下の test05.h を作る．

```
extern int funcA();
extern int funcB();
```

ライブラリの作成は以下のように行う．

```
# gcc -c test05b.c
# gcc -c test05c.c
# ar r libtest05bc.a test05b.o test05c.o
ar: libtest05bc.a を作成しています
```

ライブラリに登録する関数のオブジェクトを gcc -c により作成し，ar コマンドによりライブラリ libtest05bc.a を作る．ライブラリに登録されているオブジェクトは以下のように確認することができる．

```
# ar t libtest05bc.a
test05b.o
test05c.o
```

このライブラリをプログラム本体にリンクするときは以下のように行う．

```
# gcc -I. -L. -o test05 test05a.c -ltest05bc
```

「-I」はヘッダファイル (test05.h) の置き場所を教えており「.」はカレントディレクトリを表す.「-L」はライブラリ (libtest05.a) の置き場所を教えており「.」はカレントディレクトリを表す．静的ライブラリの名前は「lib◯◯.a」とすることが決まっていて，リンク時には「-l◯◯」という形 (lib を -l に置き換え，.a を削除した形) でリンクする．この実行結果は以下のようになる．

```
# ./test05.out A
Function A is used
# ./test05.out B
Function B is used
```

引数 A が与えられたときは funcA が呼ばれ，引数 B が与えられたときは funcB が呼ばれている．

1.4.3 静的リンクと動的リンク

静的リンクではプログラムコンパイル時に全ての関数を結合して一体化し，プログラム実行時には必要な全ての関数をメモリ上にロードする．一方，動的リンクではプログラムの実行時に必要になったときに初めてその関数をメモリ上にロードする．

先に示した例は静的リンクの例であり，一度コンパイルが完了すると，オブジェクトファイル (test05a.o，test05b.o，test05c.o) やライブラリファイル (libtest05.a) は不要でありこれを削除してもプログラムの動作には関係せず，下記のことが確かめられる．

```
# rm test05?.o libtest05bc.a
# ./test05.out A
Function A is used
# ./test05.out B
Function B is used
```

次に，動的リンクのための共有ライブラリの作成とリンクの例を示す．ここでは，既に作成済みの test05a.c，test05b.c，test05c.c，test05.h を使う．まず，共有ライブラリの作成は以下のように行う．

```
# gcc -shared -fPIC -o libtest05b.so test05b.c
# gcc -shared -fPIC -o libtest05c.so test05c.c
```

ここでは，関数ごとにライブラリを作成している．ライブラリのリンクは以下のようにする．

```
# gcc -I. -L. -o test05.out test05a.c -ltest05b -ltest05c
```

プログラムの実行は以下のように行う (csh 系の場合)

```
# setenv LD_LIBRARY_PATH .
# ./test05.out A
Function A is used
#./test05.out B
Function B is used
```

1 つめのコマンドはシェルにライブラリの置き場所を教えているものであり，デフォルトのライブラリが置いてあるディレクトリに置く場合には必要ない．見た目の動作は静的リンクの場合と同じに見える．ここで，libtest05c.so を消去してプログラムを実行してみると以下のようになる．

```
# rm libtest05c.so
# ./test05.out A
./test05.out: error while loading shared libraries: libtest05c.so: cannot open shared object file: N
```

```
# ./test05.out B
./test05.out: error while loading shared libraries: libtest05c.so: cannot open shared object file: N
```

ライブラリが見つからないというエラーが出ている．動的リンクであるので「./test05.out A」実行時には libtest05c.so 内の funcB という関数は必要ないのでエラーにならないかと思ったが駄目なようである．「./test05.out B」のときには funcB が必要で実行時にロードできずにエラーになるのは当然である．

共有ライブラリはひとつにまとめて

```
# gcc -shared -fPIC -o libtest05bc.so test05b.c test05c.c
# gcc -I. -L. -o test05.out test05a.c -ltest05bc
```

とすることもできる．ところで，ディレクトリ内に libtest05bc.a と libtest05bc.so があるときにどちらのライブラリをリンクするかが問題になるが，そのときはデフォルトでは libtest05bc.so の共有ライブラリを動的リンクする．静的リンクをしたい場合には -static オプションを付けて以下のようにする．

```
# gcc -I. -L. -static -o test05.out test05a.c -ltest05bc
```

1.4.4 デバッガの利用

プログラムが正常にコンパイルできたように見えても，それは文法エラーがないということで，正しく動作することは保証されていない．実行時エラーの原因となるものをバグ，これををなくすようにプログラムを修正する作業をデバッグともいうが，大きなプログラムを開発しているときに実行時エラーが大きなプログラムになるとどこにバグが存在するかを調べるのは大変な作業である．これを支援するものにデバッガがあり，Unix 系 OS では通常 gdb というデバッガが使える．これを利用するためにはプログラムコンパイル時に -g オプションを指定する．あるいは make を用いるときに

```
CFLAGS = -g
```

としておけば自動的に gcc でのコンパイルの際に -g オプションが付けられる．

例として以下のプログラム (test05.c) をコンパイルして実行すると (たぶん) 実行時エラーが生じる．

```
#include <stdio.h>

extern int main(int argc, char **argv)
{
    int i, n = 1000;
    int a[5];

    a[0] = 0;
    for (i = 0; i < n; i++) {
        a[i] = a[i-1]+i;
    }
    printf("%d\n", a[n-1]);
}
```

デバッガを用いるためには以下のようにする．

```
# gcc -g -o test05.out test05.c -lm
# gdb test05.out
```

その後 (gdb) というプロンプトが現れたら，以下のように r を入力しプログラムを走らす (r は run の意味)．

```
(gdb) r
Starting program: /home/tsuji/Here/Lab/test05.out
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400569 in main (argc=1, argv=0x7fffffffe438) at test05.c:10
10 a[i] = a[i-1]+i;
(gdb)
```

ここれは test05.c の 10 行目にエラーがあることがわかる．さらに i の値を調べるために，以下のように print i と入力するとその値が 820 であることがわかり，サイズ 5 の配列 a[ ] を壊していることがわかる．

```
(gdb) print i
$1 = 820
```

本来 i = 5 のときにエラーになって欲しいところであるが，何事もないように動作しているように見えることがある．そのメモリ領域が他で使われていなければ問題が生じないように見えるが，他のデータ領域を上書きしてしまうため思わぬところでおかしな動作を起こしたりする．

デバッガの終了は以下のように quit コマンドを入力する．

```
(gdb) quit
```

2　C 言語の基礎知識

2.1　入出力関数

2.1.1　printf 関数等で出力を行うときの書式の制御

小数点以下の何桁出力するかを指定するとき，通常は以下のように書く．

```
// test10A.c
#include <stdio.h>
#include <math.h>
extern int main(int argc, char **argv)
{
    printf("pi = %10.5lf\n", M_PI);
}
```

この実行結果は以下のとおりである．全部で 10 桁で小数点以下 5 桁であるので，整数部分は 4 桁になる．このため,「=」の後に半角スペースが 1+3 = 4 個出力されている．

```
# ./test10A.out
pi =    3.14159
```

この小数点以下の桁数をプログラムをコンパイルし直さないで変更するには，例えば以下のようにする．

```
// test10B.c
#include <stdio.h>
#include <math.h>
extern int main(int argc, char **argv)
{
    int n1 = atoi(argv[1]);
    int n2 = atoi(argv[2]);
    char format[256];
    sprintf(format, "pi = %%%d.%dlf\n", n1+n2+1, n2);
    printf(format, M_PI);
}
```

「%%d」などとすると整数値を代入することになるため,「%」を出力したいときには「%%」とする．つまり書式をプログラム中で作成している．この実行結果は以下のとおりであり，第 1 引数が整数部の桁数，第 2 引数が小数点以下の桁数を与えている．

```
# ./test10B.out 1 8
pi = 3.14159265
# ./test10B.out 1 12
pi = 3.141592653590
```

例えば連続するファイルにファイル番号を付けるとき

```
// test10C.c
#include <stdio.h>
#include <math.h>
extern int main(int argc, char **argv)
{
    int i, n = 10;
    FILE *fp;
```

```
    char f_name[256];
    for (i = 0; i <= n; i++) {
        sprintf(f_name, "a%d.data", i);
        fp = fopen(f_name, "w");
        fprintf(fp, "%d\n", i);
        fclose(fp);
    }
}
```

とすると，実行後にできるファイルは

```
# ls
a0.data  a10.data  a3.data  a5.data  a7.data  a9.data
a1.data  a2.data   a4.data  a6.data  a8.data
```

となり，a1.data の次が a10.data になってしまう．ファイル名を a00.data, a01.data, ···，a10.data のようにすると ls コマンドで見たときにも順番に並ぶようになる．このためには例えば以下のようにする．

```
// test10D.c
#include <stdio.h>
#include <math.h>
#include <string.h>
extern int main(int argc, char **argv)
{
    int i, n = atoi(argv[1]), k;
    FILE *fp;
    char f_name[256], format[256], *p;

    k = (int)log10(n)+1;
    for (i = 0; i <= n; i++) {
        sprintf(format, "a%%%dd.data", k);
        sprintf(f_name, format, i);
        while ((p = strchr(f_name, ' ')) != NULL) *p = '0';
        fp = fopen(f_name, "w");
        fprintf(fp, "%d\n", i);
        fclose(fp);
    }
}
```

引数で与えた個数のファイルを作る．「k = (int)log10(n)+1」により最大の桁数を調べ，ファイル番号をその桁数で付ける．このときその桁に満たない番号では先頭に空白が入ることになるので，空白を 0 で埋めてからファイルをオープンしている．このプログラムを引数 100 で実行してできるファイルは以下の通りである．

```
# ls
a000.data  a013.data  a026.data  a039.data  a052.data  a065.data  a078.data  a091.data
a001.data  a014.data  a027.data  a040.data  a053.data  a066.data  a079.data  a092.data
a002.data  a015.data  a028.data  a041.data  a054.data  a067.data  a080.data  a093.data
a003.data  a016.data  a029.data  a042.data  a055.data  a068.data  a081.data  a094.data
a004.data  a017.data  a030.data  a043.data  a056.data  a069.data  a082.data  a095.data
a005.data  a018.data  a031.data  a044.data  a057.data  a070.data  a083.data  a096.data
a006.data  a019.data  a032.data  a045.data  a058.data  a071.data  a084.data  a097.data
```

```
a007.data  a020.data  a033.data  a046.data  a059.data  a072.data  a085.data  a098.data
a008.data  a021.data  a034.data  a047.data  a060.data  a073.data  a086.data  a099.data
a009.data  a022.data  a035.data  a048.data  a061.data  a074.data  a087.data  a100.data
a010.data  a023.data  a036.data  a049.data  a062.data  a075.data  a088.data
a011.data  a024.data  a037.data  a050.data  a063.data  a076.data  a089.data
a012.data  a025.data  a038.data  a051.data  a064.data  a077.data  a090.data
```

2.1.2 ファイルが存在するかの確認

2.2 並列計算

2.2.1 OpenMP

比較的容易に並列計算を実現することができる．円周率 π の計算例を以下に示す．

```
// pi.c
#include <stdio.h>
#include <stdlib.h>

extern int main(int argc, char **argv)
{
    int i;
    int N = atoi(argv[1]);
    double x, dx, pi, sum = 0.0;

    dx = 1.0/N;
    #pragma omp parallel for reduction(+:sum) private(x)
    for (i = 0; i < N; i++) {
        x = (i+0.5)*dx;
        sum += 4.0/(1.0+x*x);
    }
    pi = sum*dx;
    printf("pi = %16.14lf\n", pi);
}
```

このプログラムは

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

であることを利用して，左辺を数値積分することにより円周率 π の近似値を求める．プログラム中の N が分割数であり，この値を大きくすることで計算精度が改善されるが，それだけ計算量が多くなる．プログラム中の「#pragma ...」という行が OpenMP の並列化を行うための指示文である．ここでは for 文を並列化していて，reduction(+:sum) を指定することで，各スレッドごとに sum が作られ，スレッドごとに計算され，for 文を抜ける際にその総和を sum に代入する．private(x) は変数 x がスレッドごとに作られることを指示する．以下にコンパイル実行の例を示す．最初に OpenMP 並列化しない場合のコンパイル・実行，次に OpenMP 並列化したコンパイル・実行の例を示している．計算時間を測るために time コマンドを用いている．

```
# gcc -O3 -o pi.out pi.c -lm
# time ./pi.out 1000000000
pi = 3.14159265358997
4.895u 0.003s 0:04.90 99.7% 0+0k 0+0io 0pf+0w
```

```
# gcc -fopenmp -O3 -o pi.out pi.c -lm
# time ./pi.out 1000000000
pi = 3.14159265358977
7.167u 0.000s 0:01.15 622.6% 0+0k 0+0io 0pf+0w
```

上記の結果を見ると CPU 時間は並列化しない方が短いが，OpenMP 並列化した場合には CPU 稼働率が約 600%であり，実質的なユーザの待ち時間は OpenMP 並列化した方が圧倒的に短いことが実際に走らせてみるとわかる．

OpenMP 並列化にはいくつかの方法があり，円周率 π の計算プログラムを以下のように作ることもできる．ここでは 4 スレッドを用いることを指定し，変数 i, x, をスレッドごとに作り，スレッドごとの総和を配列 sum[ ] に格納した後，その総和を求めている．

```
// pi-1.c
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 4

extern int main(int argc, char **argv)
{
    int i;
    int N = atoi(argv[1]);
    double dx, pi, sum[NUM_THREADS];

    dx = 1.0/N;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i; double x;
    int id = omp_get_thread_num();
    sum[id] = 0.0;
    for (i = id; i < N; i += NUM_THREADS) {
        x = (i+0.5)*dx;
        sum[id] += 4.0/(1.0+x*x);
    }
}
    for (i = 0; i < NUM_THREADS; i++) pi += sum[i]*dx;
    printf("pi = %16.14lf\n", pi);
}
```

この実行結果は以下のとおりであり，並列化はされているようだが恩恵は少ない．

```
# gcc -fopenmp -O3 -o pi-1.out pi-1.c -lm
# time ./pi-1.out 1000000000
pi = 3.14159265358977
12.655u 0.000s 0:03.55 356.3% 0+0k 0+0io 0pf+0w
```

一方，以下のように for 文を並列化すると並列化の恩恵が得られる．

```
// pi-2.c
```

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

#define NUM_THREADS 4

extern int main(int argc, char **argv)
{
    int i;
    int N = atoi(argv[1]);
    double dx, pi, sum[NUM_THREADS];

    dx = 1.0/N;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i; double x;
    int id = omp_get_thread_num();
    sum[id] = 0.0;
#pragma omp for
    for (i = id; i < N; i += NUM_THREADS) {
        x = (i+0.5)*dx;
        sum[id] += 4.0/(1.0+x*x);
    }
}
    for (i = 0; i < NUM_THREADS; i++) pi += sum[i]*dx;
    printf("pi = %16.14lf\n", pi);
}
```

```
# gcc -fopenmp -O3 -o pi-2.out pi-2.c -lm
# time ./pi-2.out 1000000000
pi = 0.78539816326619
4.870u 0.000s 0:01.36 358.0% 0+0k 0+0io 0pf+0w
```

また，以下のように並列化を行うこともできる．

```
// pi-3.c
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 4

extern int main(int argc, char **argv)
{
    int i;
    int N = atoi(argv[1]);
    double dx, pi, sum;

    dx = 1.0/N;
omp_set_num_threads(NUM_THREADS);
```

```
#pragma omp parallel
{
    int i; double x;
    int id = omp_get_thread_num();
    #pragma omp parallel private (x, sum,i)
    { id = omp_get_thread_num();
        for (i = id, sum = 0.0; i < N; i += NUM_THREADS) {
            x = (i+0.5)*dx;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum*dx;
    }
}
    printf("pi = %16.14lf\n", pi);
}
```

```
# gcc -fopenmp -O3 -o pi-3.out pi-3.c -lm
# time ./pi-3.out 1000000000
pi = 3.14159265658986
6.309u 0.003s 0:01.87 336.8% 0+0k 0+0io 0pf+0w
```