TX7/AzusA へのユーザプログラムの移植について

東北大学情報シナジーセンター・スーパーコンピューティング研究部
後藤 英昭

1 はじめに

情報シナジーセンターで平成 14 年 1 月より運用を開始した新汎用コンピュータ NEC TX7/AzusA は，ノードあたり 16 個のプロセッサを搭載しており，並列処理により非常に高速な演算が可能です．物理メモリはノードあたり 16GB ないし 32GB と広大で，研究室の PC やデスクトップワークステーションでは困難な非常に大きな問題も扱うことができます．また，オペレーティングシステムには Linux を採用しているので，研究室あるいは自宅で PC に Linux を導入しているユーザにとって非常に親しみ易く，扱い易いシステムになっています．より大きな問題を高速に解きたいとか，研究室の計算機資源が不足しているとか，あるいは並列処理の研究をしたいといった場合に，気軽にプログラムを持ち込んで走らせることができます．

アルゴリズム開発やシミュレーションなどを目的として Fortran や C 言語で書かれたプログラムは，一般的には非常にポータビリティ (移植性) が良く，多くの場合はソースを再コンパイルするだけで様々な計算機の上でうまく動作します．とりあえず，研究室の PC やワークステーション (WS) で動かしているプログラムを持ち込んで，そのままコンパイルして走らせてみて下さい．もし正常に動作したなら，一部の例外を除いて，そのまま研究を進めることができるでしょう．

本記事では，研究室の PC や WS から，あるいは情報シナジーセンターの旧並列コンピュータ (HP Exemplar X) からプログラムを移植する際にユーザがしばしば出くわす問題を紹介し，対策方法を示します．2 章ではプロセッサのエンディアンの違いに起因する，バイナリ形式のデータファイルの非互換性について紹介します．Fortran での対策は非常に簡単ですが，C/C++言語ではプログラマーが「エンディアンというもの」を意識する必要があります．3 章では 64 ビット環境における C/C++プログラミングの注意点を紹介します．センターの汎用コンピュータに限らず，64 ビットの UNIX 計算機全般に通じる内容です．4 章では汎用コンピュータで利用できる複数の C/C++コンパイラの使い分け方と，若干の性能評価結果を示します．なお，並列処理による高速化については他の記事に譲り，本記事ではとりあえずプログラムを移植して TX7/AzusA 上で正しく動くようにすることを目標とします．

2 バイナリデータとエンディアン

2.1 エンディアンとは

あなたは卵を食べる時に尖った方から割りますか．それとも大きく丸い方から割りますか．これは，Jonathan Swift の有名な物語 「Gulliver's Travels」 の中で，小人の国 Lilliput と Blefuscu の間で紛争の種

* この文書は，「東北大学情報シナジーセンター 大規模科学計算機システム広報 SENAC Vol.35 No.1 (2002.4), pp.47–61」 に掲載された原稿を元に，A4 判に整形しなおしたものです．

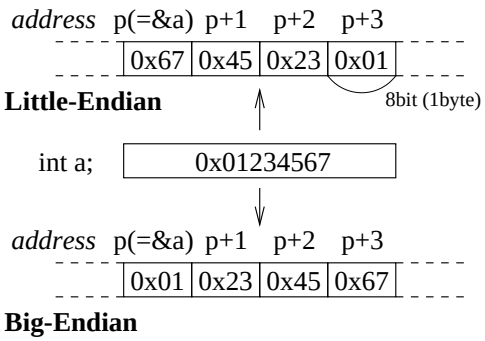

図 1: エンディアンによるメモリイメージの違い

となった流儀の違いです．丸い方から先に割る (break) 宗派は，物語の中で Big-Endian (ビッグ・エンディアン) と呼ばれています．計算機の世界でもプロセッサの構造 (アーキテクチャ) によって流儀の違いがあり，Swift の物語になぞらえて，Big-Endian と Little-Endian という用語が使われるようになりました [1].

現代のほとんどの商用計算機では，8 ビット (bit) を 1 バイト (byte) とし，メモリ上では 1 バイトごとにユニークなアドレスが与えられています．8 ビットでは $2^8 = 256$ 種類の数値しか表せないので，演算に用いる数値変数は計算機の内部では 32 ビットや 64 ビットで表され，それぞれ 4 バイトと 8 バイトのデータとしてメモリ上に置かれます．例えば，C 言語で 32 ビットの整数型変数

```
int a = 0x01234567;
```

があった時，この変数は 1 バイトごとに区切られて，0x01，0x23，0x45，0x67 という 4 個のバイト・データとしてメモリ上の連続した領域に格納されます．この時，アドレスの小さい方から大きい方へ向かって，下位バイト (LSB : Least-Significant Byte) である 0x67 から順に格納する流儀を Little-Endian と呼びます (図 1)．反対に上位バイト (MSB : Most-Significant Byte) である 0x01 から順に格納する流儀を Big-Endian と呼びます．

どちらのエンディアンになるかは，使用するプロセッサによって決まります．エンディアンを切り替えられるようなプロセッサも多いのですが，通常は電源投入直後にエンディアンが固定されてしまうので，オペレーティングシステムが変わってもいずれか一方のエンディアンだけが用いられます．近年 PC やワークステーションでよく使われているプロセッサとエンディアンのおおまかな対応を表 1 に示します．

エンディアンの違いは，プログラムの中で変数を使い，演算を行なっている場合は特に意識する必要はありません．エンディアンの違いが問題となるのは，データの入出力などの場合です．データをバイナリ形式でファイルに書き出したり，ファイルから読み込む場合には，エンディアンの違いに注意する必要があります．図 1 の例で，変数が格納されているメモリのイメージを，そのままファイルに書き出したとします．Little-Endian のプロセッサを搭載した計算機では，ファイル中のデータの並びは 0x67，0x45，0x23，0x01 の順になります．このファイルを Big-Endian のプロセッサを搭載した計算機で読み込むと，変数 a の値は 0x67452301 という誤った値になってしまいます．変数のメモリイメージをそのまま書き出す方法は，このようにエンディアンの相違による問題を引き起こすので，望ましくありません．次節より，Fortran と C/C++ 言語での対策方法を示します．

表 1: 様々なプロセッサとエンディアン

プロセッサ	endian	大規模科学計算システムの計算機
Sun SPARC	Big	
DEC (Compaq) Alpha	Little	
IBM POWER シリーズ	Big	
Motorola/IBM PowerPC	Big	
HP PA-RISC	Big	gen3 (NEC NX7000/460) 旧 par (HP Exemplar X)
MIPS MIPS32, MIPS64	Big	
Intel Itanium (IA-64)	Little	gen (NEC TX7/AzusA)
Intel Pentium, Xeon (IA-32, 俗称 x86)	Little	
NEC SX-4 system	Big	super (NEC SX-4)

2.2 Fortran の場合

Fortran では，書式なしでデータをバイナリ形式でファイルに保存した場合にエンディアンの影響を受けます．書式付きで保存したファイルではエンディアンの影響はありません．Fortran の言語プロセッサの多くでは，エンディアンの違いを吸収する仕組みが用意されているので，このような言語プロセッサを用いればエンディアンの対策は容易です．

本センターの汎用コンピュータ TX7/AzusA (ホスト名: gen) で，f95 コマンドを使用して Fortran プログラムをコンパイルした場合は，プログラム実行時にあらかじめ環境変数 `F_UFMTENDIAN` を設定しておくことによりエンディアンの切り替え (反転) が可能です．sh を使用している場合は

```
$ F_UFMTENDIAN=ALL ; export F_UFMTENDIAN
```

を実行し，csh を使用している場合は

```
% setenv F_UFMTENDIAN ALL
```

を実行することによって，バイナリファイルを Big-Endian のものとして扱うようにできます．環境変数が設定されているかどうかは，sh の場合でも csh の場合でも，

```
% echo $F_UFMTENDIAN
```

のようにして確認できます．

なお，本センターの汎用コンピュータでは，ユーザの標準の環境設定で `F_UFMTENDIAN=ALL` が指定されています．スーパーコンピュータ (SX-4) や旧並列コンピュータ (Exemplar X) で作成された Big-Endian のファイルを扱う場合は，環境変数の再設定は不要です．

もし PC などから Little-Endian のファイルを持ってきた場合は，環境変数 `F_UFMTENDIAN` をクリアしておく必要があります．環境変数をクリアするには，sh の場合は

```
$ unset F_UFMTENDIAN
```

とし，csh の場合は

```
% unsetenv F_UFMTENDIAN
```

のように操作します．

先の例では全ての入出力装置に対してエンディアンが切り替えられますが，一部の装置番号だけエンディアンを切り替える (反転させる) こともできます [2]．該当する装置番号に対する入出力操作のエンディアンを切り替えるには，

```
F_UFMTENDIAN n[,n]...   (nは装置番号)
```

という環境変数を設定しておきます．例えば装置番号 10，20，50 についてエンディアンを切り替えるには，sh を使用している場合は

```
$ F_UFMTENDIAN=10,20,50 ; export F_UFMTENDIAN
```

を実行し，csh を使用している場合は

```
% setenv F_UFMTENDIAN 10,20,50
```

を実行してからプログラムを実行します．詳しくは 「Intel(R) Fortran Compiler User's Guide [2]」の「Customizing Compilation Environment, Environment Variables」 をご覧ください [*1]．バイナリファイルのエンディアンが切り替わったかどうかは，

```
% od -t x1 filename
```

でファイルのダンプ表示を比較してみるとわかります．

環境変数 F_UFMTENDIAN によるエンディアンの切り替えは，本センターのスーパーコンピュータ SX-4 でもサポートされています [3]．

2.3 C/C++言語の場合

C/C++言語では，プログラマがエンディアンを意識したプログラミングを心がける必要があります．エンディアンに依存しないようなプログラムも書けるので，始めからそのように書いておけば，エンディアンを気にせずに異なるアーキテクチャの計算機とソースファイルやデータファイルを共用できるようになります．データファイルの共用を実現する手段として以下の二つが考えられます．

1. ファイルにエンディアンの情報を属性として持たせておき，プログラムがその情報を元に処理を切り替える．
2. ファイルのエンディアンをあらかじめ決めておく．

前者はプログラムが繁雑になるので，後者の方が楽です．ファイル中のバイト並びでどちらのエンディアンを採用するかを決めなければなりませんが，ネットワーク・バイトオーダに合わせて MSB 先行，すなわち Big-Endian にするのが広く好まれているようです．

[*1] ドキュメントは汎用コンピュータ (gen) の /opt/intel/compiler60/docs の下に置かれています．fcompindex.htm をウェブブラウザで閲覧してください．

```
int             a[N];
unsigned char   b[N * sizeof(int)];

int             i,j;
unsigned int    *pa;
unsigned char   *pb;

(配列 a[ ] に値を代入)

pa = (unsigned int  *)a;
pb = (unsigned char *)b;
for ( i=0 ; i<N ; i++ ){
    for ( j=0 ; j<sizeof(int) ; j++ ){
        *pb++ = (unsigned char)(*pa >> ((sizeof(int) - j - 1) * 8));
    }
    pa++;
}
                /* fp はファイルハンドル */
if ( N != fwrite((void *)b,sizeof(int),N,fp) ){
        goto Error;
}
```

図 2: Big-Endian によるバイナリデータの書き出し

```
int             a[N];
unsigned char   b[N * sizeof(int)];

int             i,j;
unsigned int    *pa;
unsigned char   *pb;

                /* fp はファイルハンドル */
if ( N != fread((void *)b,sizeof(int),N,fp) ){
        goto Error;
}

pa = (unsigned int  *)a;
pb = (unsigned char *)b;
for ( i=0 ; i<N ; i++ ){
    *pa = 0;
    for ( j=0 ; j<sizeof(int) ; j++ ){
        *pa |= (unsigned int)*pb++ << ((sizeof(int) - j - 1) * 8);
    }
    pa++;
}

(配列 a[ ] の値を利用)
```

図 3: Big-Endian によるバイナリデータの読み込み

int 型で長さが N の一次元配列 a[ ] にデータが格納されており，これをバイナリ形式でファイルに保存する例を示します．図 2 は，Big-Endian でファイルにデータを書き出すコードの例です．unsinged char 型で長さが N * sizeof(int) の一次元配列 b[ ] を出力バッファとして用意し，これに Big-Endian でデータのイメージを作成してから，fwrite() でファイルに書き込んでいます．このコードは，Big-Endian の計算機でコンパイルしても，Little-Endian の計算機でコンパイルしても，いずれも正しく Big-Endian でバイナリデータを書き出します．

図 3 は，Big-Endian でファイルからデータを読み込むコードの例です．unsinged char 型で長さが N * sizeof(int) の一次元配列 b[ ] を入力バッファとして用意し，これに一旦 fread() でデータを読み込んでから，計算機のアーキテクチャに合わせて配列 a[ ] にデータを格納しています．このコードは，

```
int a = 0x01;
if ( 0 == *(char *)&a ){
    /* Big-Endian */
}
```
図 4: エンディアンを調べるコードの例

Big-Endian の計算機でコンパイルしても，Little-Endian の計算機でコンパイルしても，いずれも正しく Big-Endian でバイナリデータを読み込みます．

図 2, 3 のどちらのプログラムでも，計算機のエンディアンを自動的に検出するコードを付け加えれば，ファイルのエンディアンとプロセッサのエンディアンが同じ計算機では変換処理をスキップして，配列 a[ ] に対してそのままファイル入出力の操作を行なわせることも可能です．そのようにプログラムを書き換えることは難しくないでしょう．参考までに，プロセッサのエンディアンを調べるコードの例を図 4 に示します．if 文が条件を満たせば，その計算機は Big-Endian です．

また，C 言語ではエンディアンを揃えるために htonl() や ntohl() といったライブラリ関数を使うこともできます．しかし，これらの関数が定義されているインクルードファイルの場所がオペレーティングシステムによって様々なので，ソースファイルの共用は難しくなります．

3 64ビット環境における C/C++プログラミング

3.1 64ビットの開発環境

汎用コンピュータ TX7/AzusA には Intel Itanium プロセッサが搭載されています．Itanium は IA-64 (IA: Intel Architecture) と呼ばれるプロセッサ・アーキテクチャ (構成) に基づいて設計・実装されたプロセッサの一つです．Itanium は 64 ビット CPU なので，Pentium 系の IA-32 のプロセッサと比べてレジスタの幅が倍 (64 ビット) になり，一度の演算でより大きな数値を扱うことができます．命令セットにおけるアドレス長も 32 ビットから 64 ビットに広がり，格段に大きなメモリ空間を扱えるようになっています．64 ビット CPU のこのような利点を生かすために，オペレーティングシステム (OS) やプログラミング環境も 64 ビットに対応させる必要があります．近年の WS や UNIX サーバの多くでは，64 ビット CPU が搭載され，64 ビット対応の OS が標準的に使われています．

本センターの汎用コンピュータでは，OS の IA-64 Linux はもちろん，C/C++ のコンパイラも 64 ビットに対応しており，64 ビットの開発環境が整っています．一方，Pentium 系の CPU を搭載した PC に Linux や FreeBSD などのいわゆる PC-UNIX を導入している場合は，C/C++コンパイラは 32 ビット対応です．少し古目の WS や UNIX サーバでも，32 ビットにしか対応していない開発環境が散見されます．32 ビット環境で作成された C/C++のプログラムを汎用コンピュータに持ち込んだ場合に，コンパイルエラーが発生したり，動作がおかしくなる場合があります．32 ビット環境と 64 ビット環境では C/C++の変数型の一部に違いがあり，このことがしばしばトラブルの元になっています．センターの汎用コンピュータを始め，様々な 64 ビット計算機と 32 ビット計算機との間でソースファイルを共用するためには，変数型の違いに影響を受けないようなプログラミングを心がける必要があります．

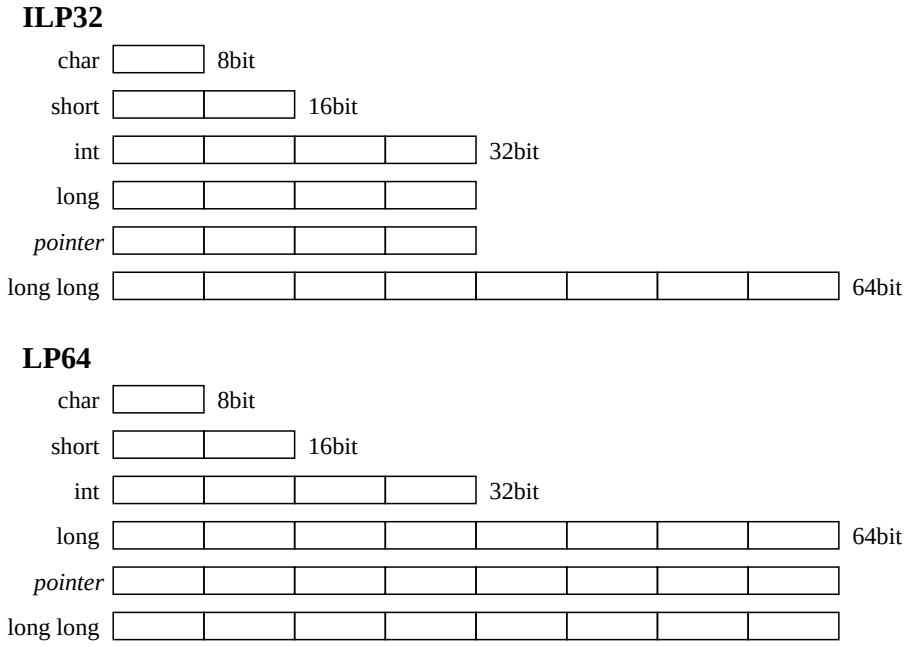

図 5: ILP32 / LP64 プログラミング・モデルにおける整数型データとポインタのサイズ

3.2 LP64 プログラミング・モデル

WS や PC の普及と共に 32 ビットの C/C++コンパイラが広く普及し，長期間に渡って使われていたせいか，C/C++言語の int 型，long 型，およびポインタ型の変数をいずれも 4 バイト (32 ビット) だと仮定して作られたプログラムが少なくないようです．その昔に PC で 16 ビットのコンパイラを使ったことがある人は，int 型が 2 バイトだったことを覚えているかもしれません．32 ビットの計算機で UNIX を使用している場合，一般に C/C++コンパイラはポインタ型を 32 ビットとして扱います．32 ビットのポインタでは $2^{32} = 4\text{GB}$ のアドレス空間しかアクセスできません．64 ビットの計算機では，先に述べたようにアドレス長が長くなっているので，ポインタ (アドレスに相当) のサイズも拡大する必要があります．

64 ビットの計算機システムを設計する際に，ポインタのサイズと合わせて，他のデータ型のサイズについても詳しく検討されました．主要な UNIX ベンダによる多くの議論の末に，64 ビット UNIX では「LP64 データ・モデル」を C 言語の標準とすることが，X/Open という標準団体で合意されました [4, 5]．このモデルは「LP64 プログラミング・モデル」とも呼ばれています．

LP64 プログラミング・モデルでは，long 型とポインタ型のみが 64 ビットに拡大されました (図 5)．LP64 の “L” は long，“P” は pointer を表します．int 型を 64 ビットに変更すると，新たに 32 ビットの型を作る必要があり，プログラムの移植性も悪くなることから，int 型は 32 ビットに据え置かれました．int 型，long 型，およびポインタ型を 32 ビットとして扱う従来のモデルは，ILP32 プログラミング・モデルと呼ばれています．

既に DEC (Compaq) Alpha を搭載した計算機で動いているプログラムは，本センターの汎用コンピュータでもほとんど変更せずに動かすことができるでしょう．Itanium と Alpha はいずれも 64 ビット CPU で，エンディアンも Little-Endian で共通しています．

スーパーコンピュータ SX-4 で提供されている C コンパイラは LP64 です．エンディアンの違いにさえ気を付ければ，それほど苦労せずにプログラムを汎用コンピュータに移植できるでしょう．Sun や IBM，SGI などの RISC プロセッサを搭載した WS で 64 ビットのコンパイラを使用している場合も同様です．

WS で 32 ビットのコンパイラを使用している場合や，PC で UNIX を使っている場合は，プログラムの書き換えが必要になる場合があります．

基本的には long 型やポインタ型を 4 バイトと仮定しないようなコードに書き換えればよく，そのようにすれば 32 ビットでも 64 ビットでもどちらの環境でも動くプログラムになります[*2]．

3.3 プログラミング上の注意点

32 ビット環境に慣れたプログラマーがしばしば犯す間違いを紹介しながら，32 ビット環境から 64 ビット環境へ移行する際に注意すべき代表的な点を紹介します．

3.3.1 ポインタを int 型変数に代入しない

ポインタを int 型変数に代入して操作してはいけません．

アセンブラを使った経験があるプログラマーなどは，C 言語のポインタを整数型変数に代入してアドレス計算を行なうようなコードを書くことがあります．例えば

```
/* wrong code */
long a[N];
long *p;
int  addr;
  ...
addr = (int)a;
addr += sizeof(long) * x;  /* a[x] */
p = (long *)addr;
  ...
```

のようなものです．このコードは ILP32 では正しく動きますが，LP64 ではポインタを int にキャストした時にアドレスの上位ビットが失われてしまい，Segmentation fault などを引き起こします．

C 言語では，データ領域を動的に確保するために， malloc() 関数がよく使われます．malloc() の戻り値の型が `void *` だということに注意すれば，戻り値を int 型で受けてはいけないということはすぐにわかります．malloc() や calloc() の戻り値は必ずポインタ型で受けるようにします．

3.3.2 long 型と int 型が混在した演算に気を付ける

ILP32 では sizeof(long) == sizeof(int) ですが，これを仮定してはいけません．特に，long 型と int 型が混在した演算では，途中結果をうっかり int 型変数に格納したりしないように，十分に気を付ける必要があります．

[*2] 当然のことですが，巨大なデータ領域が必要な処理は 32 ビット環境ではできないことがあります．

3.3.3 long 型を 4 バイト・データに使わない

long 型を 4 バイトと仮定してはいけません．

16 ビット環境の経験があるプログラマーは，int 型が 2 バイトになりうることを知っています．このため，長さが 4 バイトに決まっているデータ領域が必要な場合に，int 型を避けて，うっかり long 型を使ってしまうことがあります．LP64 では long 型が 8 バイトなので，期待した結果が得られなくなります．

決まった長さのデータ領域が必要な場合は，プログラムのどこか一か所で専用の変数型を定義し，残りの部分ではこの変数型を使うようにします．ILP32 や LP64 で 4 バイトの整数型を使いたいときは，例えば

```
typedef int int32;
```

のようにして，int32 型を使うようにします．int 型が 4 バイトではない環境にプログラムを持っていく場合でも，typedef の行を書き換えるだけで済みます．

3.3.4 printf() や sscanf() のフォーマットに注意する

32 ビット環境 (ILP32) に慣れたプログラマーは，しばしば

```
/* wrong code */
long a;
  ...
printf("%d\n",a);
```

のような間違ったコードを書きます．ILP32 では，sizeof(long) == sizeof(int) なので，このコードは正しく動きます．しかし，%d は int 型変数を示すフォーマット指定なので，LP64 では long 型変数 a の半分のビットしか関数 printf() に渡されないことになります．正しく

```
printf("%ld\n",a);
```

と書く必要があります．sscanf() などでも同様に，正しいフォーマット指定が必要です．

3.3.5 アラインメント (alignment) に気を付ける

多くの計算機では，アドレスの最下位ビットが 1 となる場所を先頭として 2 バイトの変数をメモリに格納することはできません．実際に格納できるとしても，メモリアクセスの効率が落ちるので，コンパイラは下位ビットが 0 となる場所に変数領域を割り当てます．同様に，4 バイトの変数は下位ビットが 01，10，11 となる場所には格納できず，00 となる場所に置かれます．このようにメモリ上でデータの境界が揃えられることをアラインメント (alignment) と呼びます．

例えば，下のような構造体があったとします．

```
struct Cell {
    char c;
    long a;
};

struct Cell cell;
```

構造体 cell がメモリ上の 0x12345670 番地に置かれているものと仮定します．変数 cell.c は 0x12345670 に置かれます．32 ビットの計算機で，ILP32 の場合は，cell.a は 4 バイトです．アラインメントの影響によって，cell.a は cell.c に続く領域 0x12345671–0x12345674 には置くことができず，4 バイト境界の 0x12345674–0x12345677 に置かれます．すなわち cell.c と cell.a は 4 バイトだけ離れた所に置かれます．

LP64 では cell.a が 8 バイトなので，8 バイト境界に置かれることが多く，先のコードを LP64 の環境でコンパイルすると cell.c と cell.a は 8 バイトだけ離れます．二つの変数の間隔を 4 バイトと決めつけているようなプログラムは，LP64 ではうまく動かないことになります．

アラインメントはコンパイラのオプションで変更できる場合があります．しかし，プログラムのポータビリティを高めるためには，できるだけアラインメントの影響を受けないようなコードを書くことが望ましいと言えます．

4 C コンパイラと実行環境

4.1 汎用コンピュータの C コンパイラ

情報シナジーセンターの汎用コンピュータ TX7/AzusA (gen) では，NEC/Intel C/C++コンパイラ (ecc, ecpc) と，GNU C/C++コンパイラ (gcc, g++) の 2 種類が利用できます．cc コマンドで ecc が，c++ コマンドで ecpc が起動されるようになっています．ecc は OpenMP [6] に対応しているので，プログラムの並列化が必要な場合は ecc を使うことになります．並列化の機能を使わない場合は，ecc と gcc のどちらを使うおうか悩むかも知れません． PC で Linux を使っている人は，gcc の方が馴染みがあって，gcc を使いたいと思うかもしれません．

Intel は IA-32 用に icc という C コンパイラを提供しています．PC の Linux では，icc の方が一般に gcc よりも高速な実行形式を生成することが知られています．言い替えれば，icc のほうが最適化機能が優れています．ecc は IA-64 用のコンパイラで，やはり gcc よりも最適化機能が優れているだろうと予想できます．

結論から先に述べると，ecc の方が一般に gcc よりもずっと高速な実行形式を生成します．並列処理が不要の場合でも，できるだけ ecc を使う方がよいと言えます．ecc と gcc で簡単な速度比較を行なったので，次節でその結果を紹介します．

4.2 ecc と gcc の速度比較

汎用コンピュータの主な用途を考慮すれば，数値計算のプログラムで速度を比較すべきでしょうが，残念ながら手元に手頃な C プログラムがなかったので，今回はファイル圧縮のプログラムを使いました．フリーソフトウェアの gzip と bzip2 を ecc と gcc でそれぞれコンパイルして，処理時間を測定しました．速度比較に用いたプログラムとソースパッケージ，コンパイル時の最適化オプションを表 2 に示します．使用したデータファイルの詳細を表 3 に示します．

実行時間の測定には /usr/bin/time を使い，例えば gzip なら

```
% /usr/bin/time gzip -c filename > /dev/null
```

のようにしてユーザ時間を求めました．ファイル入出力のオーバヘッドによる誤差を減らすために，ファイル出力はデータシンク (/dev/null) に捨てています．また，それぞれ最初の測定結果は捨てて，3 回連続して測定したユーザ時間の平均値を求めました．

表 2: 速度比較に用いたプログラムと最適化オプション

コマンド	最適化オプション	ソースパッケージ
gzip	-O2	gzip-1.3.2.tar.gz
gzcat	-O2	gzip-1.3.2.tar.gz
bzip2	-O2	bzip2-1.0.1.tar.gz

表 3: 速度比較に用いたデータファイル

名前	ファイルサイズ	ファイルの性質
100M.dat	100MB	全データ 0
netscape	13.16MB	netscape の実行形式 (バイナリ)
testimage.ppm	33.22MB	PPM 形式の 24bit フルカラー文書画像

測定結果を表 4 に示します．いずれのプログラムでもデータでも，ecc で生成された実行形式の方が gcc より高速なことがわかります [*3]．この他にも幾つかのプログラムを試していますが，ecc の方が数十パーセント，物によっては二倍近くも高速な実行形式を生成することがわかっています．単にコンパイラを変えるだけで処理速度の向上が見込めるので，ぜひ ecc を使ってみることをお奨めします．gcc の適用対象は，どうしても ecc でコンパイルが通らなかったり，ecc で速度が低下してしまうようなプログラムに限られるものと思われます．

4.3 IA-32 用バイナリの実行

Itanium は，ネイティブな IA-64 命令セットに加えて IA-32 の命令セットを持っており，IA-64 と IA-32 のバイナリを動的に切り替えて実行できます [7]．汎用コンピュータの IA-64 Linux でも IA-32 の互換モー

表 4: 処理時間 (ユーザ時間) の比較結果

コマンド	データ	処理時間の比 : (ecc)/(gcc)	速度向上比
gzip	100M.dat	0.67	+49%
	netscape	0.70	+43%
	testimage.ppm	0.66	+52%
gzcat	100M.dat	0.88	+14%
	netscape	0.69	+45%
	testimage.ppm	0.60	+67%
bzip2	100M.dat	0.83	+20%
	netscape	0.75	+33%
	testimage.ppm	0.84	+19%

[*3] gzip や bzip2 などの圧縮ユーティリティは，多くの利用者にとって有用なものなので，ecc でコンパイルされた高速版が汎用コンピュータ上に置かれています．/usr/local/bin にあるコマンドが高速版で，/usr/bin の方が IA-64 Linux の標準のコマンドです．

ドがサポートされており，PC の Linux のためにコンパイルされた Pentium 用のプログラムを，再コンパイルなしにそのまま動かすことができます．実際に，汎用コンピュータでは Acrobat Reader などの一部のアプリケーション・プログラムは IA-32 版が動いています．

IA-64 Linux の IA-32 互換モードは，あくまでソフトウェアの移行をスムーズにするためのものと考えられ，処理速度については期待できません．IA-32 用にコンパイルされた gzip と gzcat を試してみたところ，ecc でコンパイルされた IA-64 用の実行形式の 20〜40%の処理速度しか得られませんでした．従って，IA-64 Linux の IA-32 互換モードは，ソースファイルが入手できないプログラムや再コンパイルできないプログラム，あるいは処理速度があまり問題とならないような小柄なツールに，その利用を留めた方がよいでしょう．

5 おわりに

最近では，ポケットマネーでギガバイト級のメモリモジュール (PC 用) が買えます．PC に使われる CPU も非常に高速になったので，メモリを沢山挿して，Linux などの PC-UNIX をインストールして，PC を数値計算やアルゴリズムの開発に使うという例が増えています．しかし，あまり意識されていないようですが，PC 用に現在普及している UNIX では 1〜4GB の間にメモリ容量の壁があります．C コンパイラが ILP32 プログラミング・モデルに基づいている以上，1 プロセスあたり 4GB を超えるデータ領域を使うことは不可能です．少し大きな問題を扱おうとすると，すぐに 32 ビットの限界にぶち当たるでしょう．

少ないメモリで動くアルゴリズムを開発するという手段もありますが，新しいアルゴリズムの開発や実装が難しかったり，メモリ使用量の削減が困難だったり，あるいは単に計算結果が早く欲しいという場合も非常に多いものです．情報シナジーセンターの新しい汎用コンピュータ TX7/AzusA では，32GB という広大なメモリと 64 ビット対応の開発環境によって，大規模な計算が容易に実現できます．過去に試そうと思ったものの，メモリ使用量が多くて手を付けられなかったようなアイディアなどが死蔵されていたら，汎用コンピュータで試してみてはいかがでしょうか．

本記事では，ユーザプログラムを研究室の WS や PC から TX7/AzusA あるいは他の UNIX サーバに移植したり，プログラムを共用するのに，必要と思われる情報をまとめてみました．TX7/AzusA の利用やプログラムの移植作業に，または 64 ビット環境の入門に，本記事が微力でもお役に立てれば幸いです．

参考文献

[1] Danny Cohen, “On Holy Wars and a Plea for Peace,” USC/ISI IEN 137, Apr 1, 1980.

[2] “Intel(R) Fortran Compiler User’s Guide.”

[3] “SUPER-UX FORTRAN90/SX プログラミングの手引,” 日本電気, 1995. (G1AF07-3)

[4] 清兼義弘, “64 ビット UNIX&CDE,” 共立出版, 1997. (ISBN4-320-02870-8)

[5] 堀 俊晴, 清兼義弘, 日高和之, “64 ビット UNIX の効用 —RDB の検索速度が最大 250 倍に—,” 日経コンピュータ 1997 年 1 月 6 日号.
(http://tru64unix.compaq.co.jp/technical/64bit/nc970106/unix64.html)

[6] http://www.openmp.org/

[7] 池井 満, “IA-64 プロセッサ基本講座,” オーム社, 2000. (ISBN4-274-06376-3)