数 値 解 析 41－5
（1992. 6. 5）

IEEE浮動小数点数のハードウエアによる精度の違いについて

福井義成
（株）東芝，総合情報システム部

分散処理を行なう場合，浮動小数点数の表現方法が異なると色々と問題が起きる．さらに，浮動小数点数の表現方法がIEEE形式である計算機でも，システムによって結果が異なっている．IEEEの浮動小数点数の表現は規格で決っているが，規格で規定している以外の部分のインプリメント方法がシステムによって異なり，計算結果が違うことがある．各種のワークステーションやスーパーコンピュータをネットワークで結合し，分散処理を行なう場合，データ形式の互換性が重要となる．ここでは，現存するのシステムではIEEE形式の浮動小数点であっても結果が異なることに対する注意を喚起したい．

Hardware Dependent Precision Difference of IEEE Floating Point Numbers

Yoshinari Fukui

Total Information & Systems Division
TOSHIBA CORPORATION
580-1, Horikawa-cho, Saiwai-ku, Kawasaki 210, JAPAN

In distributed computing, problem arises when different floating point number formats are used on each computers. Even when IEEE standard floating point number is used in different systems, nonidentical results may be derived. The reason for these differences is the different implementations of floating point number format not specified in IEEE standard. Data format compatibility is important in distributed computing over network connected supercomputers and/or workstations. And cautions should be taken since IEEE standard floating point number format may have nonidentical values over different systems.

１．はじめに

ここでは，“ＩＥＥＥ浮動小数点数”の計算精度について述べる．筆者自身，ネットワークを使用した分散処理を行なう場合，浮動小数点数の表現方法について問題意識を持っていたところ，「ＩＥＥＥの浮動小数点数の計算機でも計算結果が異なるものがある．」という話を聞き，試したところ，本当にシステムによって結果が異なっていた．ＩＥＥＥの浮動小数点数の表現は規格（７５４，８５４）で決っているが，規格で規定している以外の部分のインプリメント方法がシステムによって異なり，計算結果が違うことがある．各種のワークステーションやスーパーコンピュータをネットワークで結合し，分散処理を行なう場合，データ形式の互換性が重要となる．

２．データ形式について

各種のワークステーションやスーパーコンピュータをネットワークで結合し，分散処理を行なう場合，相互にデータを交換する都合上，データ形式の互換性が重要である．通常，データ交換の対象となるものは以下の３つである．

（１）文字（数字，英字，漢字）
（２）整数
（３）浮動小数点数

データ形式が異なる場合の変換の方式には

（１）文字形式を介する
（２）バイナリで変換を行なう

等がある．「文字形式を介する」方法は，ＦＯＲＴＲＡＮの書式付きの入出力などを利用するものである．整数や浮動小数点数を文字形式に変換し，その後，ＡＳＣＩＩ－ＥＢＣＤＩＣ変換等を行えばよい．この方法は非常に簡便であるが，以下のような問題点がある．

（１）桁数をどの程度にするか（桁あふれ等）
（２）浮動小数点数の２進（１６進）－１０進変換の誤差（桁数に依存）
（３）処理時間がかかる（場合によっては本来の計算時間以上かかる）

「バイナリで変換を行なう」方法はＣＲＡＹなどで実現されており，ＦＯＲＴＲＡＮの書式なしの入出力ルーチンの中で文字，整数，浮動小数点数の変換を自動的に行なってくれるものである．変数の属性はコンパイラーが処理し，属性にあった入出力ルーチンへの呼び出しを生成してくれる．

整数の場合，１６ビット，３２ビット，６４ビットの整数などがある．また，負の値の

表現方法には２の補数表示，絶対値表示，１の補数表示などがあるが，現在では２の補数表示のみを考えればよいであろう．そのため，異なる表現同士の変換といっても，ビット長の変換だけでよいであろう．長い整数から短い整数に変換するためには，オーバーフローに注意する必要はあるが，長い整数の下の部分を取り出せばよい．逆に，短い整数から長い整数に変換するためには，符号ビットの拡張をし，長さを合わせればよい．

浮動小数点数の表現には

ＩＥＥＥ方式
ＩＢＭ方式
ＣＲＡＹ方式
ＶＡＸ方式
ＡＣＯＳ６方式（ＧＥ６３５）

等がある．精度についても３２ビット，６４ビット，１２８ビットなどがある．表現方法が異なる計算機の間でデータ交換を行なう場合，何等かの変換が必要となる．

複数の計算機の間で浮動小数点データを交換する場合，互いに同じ形式であれば問題ないが，異なる場合，

（１）変換誤差
（２）オーバーフロー／アンダーフロー

などの問題が生じる．

３．ＩＥＥＥ浮動小数点数のハードウエアによる精度の違い

システム（ハードウエア，コンパイラーなど）によって，同じＩＥＥＥ形式の浮動小数点数であっても，計算結果が異なることがある．同じ浮動小数点数の表現であっても，大規模な計算を行なった場合，コンパイラーが生成する演算順序が異なり，計算結果が微妙に（？）異なってしまうことは珍しいことではない．特に，ベクトル計算機を使用する時には，ベクトル化のためよくあることである．しかし，異なるシステムのＩＥＥＥ形式の場合は，非常に単純な演算でも結果が異なることがある．最初に試したプログラムを下記の示す（テストはすべてＦＯＲＴＲＡＮを使用して行なった．アセンブラ等は使用していない）．この例は n=3,30,300,... とし，１／ｎをｎ回加算するプログラムである．色々なシステムで計算を行なったところ，結果は２つに分類できた．（計算結果２でＳの値が１．０になっているにもかかわらず，誤差が出ているのはＳの２進１０進変換が１０進１７桁目を四捨五入しているためである．）

プログラム１．

```
      program ex01
      implicit real*8 (a-h,o-z)
      read(5,*) n, loop
      do 300 l=1,loop
        a=1.0d0/n
        s=0.0d0
        do 100 i=1,n
          s=s+a
  100   continue
        d=s-1.0d0
        write(6,6100) l, n, s, d
 6100   format(1x,i3,i10,1pd27.19,d27.19)
        n=n*10
  300 continue
      end
```

計算結果１．

loop	n	Ｓの値	誤差
1	3	1.0000000000000000000d+00	0.0000000000000000000d+00
2	30	9.9999999999999990000d-01	-1.1102230246251570000d-16
3	300	9.9999999999999610000d-01	-3.8857805861880480000d-15
4	3000	9.9999999999995640000d-01	-4.3631764867768650000d-14
5	30000	9.9999999999989960000d-01	-1.0036416142611420000d-13
6	300000	1.0000000000045930000d+00	4.5934367420841230000d-12
7	3000000	1.0000000000606620000d+00	6.0662141976308700000d-11

計算結果２．

loop	n	Ｓの値	誤差
1	3	1.0000000000000000000d+00	-5.5511151231257827000d-17
2	30	1.0000000000000000000d+00	-1.3877787807814457000d-17
3	300	1.0000000000000000000d+00	6.4184768611141862000d-17
4	3000	1.0000000000000000000d+00	-3.3393426912553537000d-17
5	30000	1.0000000000000004000d+00	3.5279938692678314000d-16
6	300000	9.9999999999999800000d-01	-1.9762295098979532000d-15
7	3000000	1.0000000000000380000d+00	3.7979168421298226000d-14

計算結果１と２では n= 3000000 の場合，１０進で約３桁も異なっている．演算の性質上

中間結果を保存するレジスターの長さが影響していると予想し，次のプログラムでのテストを行なった．このプログラムは１に2^{-n}を加算し，2^{-n}が小さくなり，積み残しが起きる状況をテストする． sub1 では$1+2^{-n}+2^{-n}$を計算し， sub2 では$2^{-n}+2^{-n}+1$を計算している．この順序通りに計算すれば， sub1 よりは sub2 のほうが有限桁計算の影響が出にくいと考えられる．

プログラム２．

```
      program ex02
      read(5,*) m, n
      call sub1(m,n)
      call sub2(m,n)
      end
      subroutine sub1(m,n)
      implicit real*8 (a-h,o-z)
      dimension  c(2)
      a=1.0d0
      b=1.0d0
      do 100 i=1,m
  100 b=b*0.5d0
      do 300 i=m,n
        do 200 j=1,2
  200   c(j)=b
        d=a+c(1)+c(2)
        write(6,6100) i, d, d
 6100   format(1x,i3,2x,z16)
  300 b=b*0.5d0
      end
      subroutine sub2(m,n)
          ↓
        d=  c(1)+c(2)+a
          ↓
      end
```

計算結果３．

$1+2^{-n}+2^{-n}$の結果

m	d の内部表現	
50	3ff0000000000008	
51	3ff0000000000004	
52	3ff0000000000002	
53	3ff0000000000000	←
54	3ff0000000000000	

$2^{-n}+2^{-n}+1$の結果

m	d の内部表現	
50	3ff0000000000008	
51	3ff0000000000004	
52	3ff0000000000002	
53	3ff0000000000001	←
54	3ff0000000000000	

計算結果４．

$1+2^{-n}+2^{-n}$の結果

m	d の内部表現	
50	3ff0000000000008	
51	3ff0000000000004	
52	3ff0000000000002	
53	3ff0000000000001	←
54	3ff0000000000000	

$2^{-n}+2^{-n}+1$の結果

m	d の内部表現	
50	3ff0000000000008	
51	3ff0000000000004	
52	3ff0000000000002	
53	3ff0000000000001	←
54	3ff0000000000000	

色々なシステムで計算を行なったところ，結果はやはり，計算結果３と４の２つに分類できた．計算結果３では，ＩＥＥＥ形式の浮動小数点数の倍精度の仮数部の長さは５２ビットであるため，2^{-53}のところで変化が起きている．計算結果３の$1+2^{-n}+2^{-n}$の場合，$1+2^{-n}$の演算で積み残しが発生していると考えられる．$2^{-n}+2^{-n}+1$の場合は$2^{-n}+2^{-n}$を先に演算し，結果的に$1+2^{-n+1}$と等価になるため，2^{-53}のところで積み残しは起きていない．計算結果４では2^{-53}のところで両者とも積み残しは起きていないので，中間結果を保存するレジスターが長いと想像できる．

以上の例では，計算結果は２つに分類できた．この２つの結果を出す環境は，今回のテスト（ＦＯＲＴＲＡＮを使用）では，以下のようであった．

（１）誤差が小さいケース
- ８０＊８７＋ＭＳ－ＦＯＲＴＲＡＮ
- ６８８８１＋Ｆ７７で最適化を行なった場合

（２）誤差が大きいケース
- ６８８８１＋Ｆ７７で最適化を行わない場合
- テストしたその他の環境

今回はＦＯＲＴＲＡＮ環境で行なった．この問題はレジスターの扱いによって大きく影響が出るため，アセンブラ等を使用した場合は今回の結果と異なる可能性もある．８０＊８７の場合は常に，拡張精度の演算を行なっているようである．６８８８１の場合，最適化を行なわないと加算のところでｆａｄｄｄという命令を生成しており，最適化を行なうとｆａｄｄｘという命令を生成している．ｆａｄｄｄは倍精度演算，ｆａｄｄｘは拡張精度演算を行なう．

４．長い中間レジスターを使用することの是非

今回の計算例では，長い中間レジスターを使用することは良い結果を与えている．また，内積計算の場合，長い中間レジスターを使用することによって非常に良い計算結果を得ることができることは良く知られている．しかし，長い中間レジスターを使用することが困った結果を与えることもある．次の例（プログラム３）はＡＣＯＳ６の計算例である．ＡＣＯＳ６は単精度の浮動小数点数は３６ビット（仮数部２７ビット）であるが，演算結果は仮数部７１ビットのレジスターに残る．

```
ZBZ02=ZB*ZB-(Z-Z0)**2
```

の計算では　ZB*ZB の結果は一度メモリーに格納されるため，２７ビットの精度になるが，(Z-Z0)**2 の結果は７１ビットのレジスターに残っており，これと２７ビットの数の差を取るため，ゼロにならない．ＡＣＯＳ６の場合は，浮動小数点レジスターが１つしかな

いため，メモリーへの格納が多くなり，この現象が起きやすい．ZBZ02 の値をそのまま条件分岐に使用すると，意図しない方向への分岐が発生することがある（ただし，浮動小数点演算で，このようなことが発生する可能性のある条件分岐を行なうこと自体問題であるが，計算機のなかでこのような処理が行なわれていることを知っているユーザばかりではないであろう）．

プログラム３．

```
      Z=0
      ZB=0.18996827E03
      Z0=0.18996827E03
      ZBZB=ZB*ZB
      Z0Z0=(Z-Z0)**2
      WRITE(6,1000) "ZBZB      =",ZBZB
      WRITE(6,1000) "(Z-Z0)**2=",Z0Z0
      ZBZ01=ZBZB-Z0Z0
      ZBZ02=ZB*ZB-(Z-Z0)**2
      WRITE(6,1000) "ZBZ01     = ",ZBZ01
      WRITE(6,1000) "ZBZ02     = ",ZBZ02
      STOP
 1000 FORMAT(" ",A10,E16.8)
      END
```

計算結果５．

```
 ZBZB      =  0.36087943E 05
 (Z-Z0)**2=  0.36087943E 05
 ZBZ01     =  0.
 ZBZ02     =  0.24581095E-06
```

５．おわりに

ここでテストしたどのシステムも３２／６４ビットの形式ではＩＥＥＥの規格に違反しているわけではないが，コンパイラーまで含めた環境で，拡張倍精度の使用方法に違いがあるようである．

もともと浮動小数点数の表現が異なるを意識しているシステム間で分散処理を行なう場合，浮動小数点数の違いに注意する人は多いであろうが，どれも“同じ”ＩＥＥＥ形式であるシステム間で分散処理を行なう場合，浮動小数点数の違いに注意する人はあまり多くないかもしれない．使用者の立場からは，精度がよいほうが望ましいが，それ以前の問題として，結果が同じであってほしいと望みたい（このことは，拡張倍精度の扱いかたと演算順序の両方に関係しますが）．ここでは，少なくとも現在のシステムではＩＥＥＥ形式の浮動小数点であっても結果が異なることに対する注意を喚起できればよいと考えます．

数値計算屋の立場からは，拡張倍精度をうまく利用できるほうが，望ましいと考えていますが，演算の高速性を追求することとは矛盾するのでしょうか？　長いレジスターの利用は演算順序にも大きく影響されます．最近の高度に発達した（？）最適化ＦＯＲＴＡＲＮコンパイラーを使用しているとプログラムを書く人間の意志が尊重されない（考えてい

る計算順序が尊重されない）ことがあり，歯がゆい思いをしています．やって欲しい部分だけの最適化と長いレジスターの活用がうまく制御できるようになってほしいと望んでいます．ユーザの立場からは，個々の演算が問題ではなく，計算全体の精度／速度が重要だからです．

参考文献

［１］別冊インターフェース，数値演算プロセッサ，ＣＱ出版社，１９８７．
［２］数値計算ガイド，日本サン・マイクロシステムズ（株）発行マニュアル，１９９１．