Fortranの引数渡しを理解する

岩下 英俊
高性能Fortran推進協議会（HPFPC）
理化学研究所 計算科学研究機構
2016/11/25

内容

- はじめに
- FORTRAN77の引数インタフェース
- Fortran90で追加された引数インタフェース
- 引数渡しの性能を上げる
- まとめ

Fortranの引数渡し

- 引数渡し
  - 実引数（呼出し側）と仮引数（手続側）の組合せはいろいろ
    - 例

```
real a(10), b(10), c(10)
call  foo( a, b(6), c, 10 )

subroutine foo( x, y, z, n )
real x(10), y(5), z(n)
```

```
real d(10,10), e(10,10), f(10,10)
call  foo( d, e, f(2:9,2:9) )

subroutine foo( u, v, w )
real u(10,*), v(:,:), w(8,8)
```

  - Interfaceブロック（引用仕様宣言）や他の明示的引用仕様によって、引数渡しの方法が選択される。
    - e → v(:,:) には、明示的引用仕様が必須
    - f(2:9, 2:9) → w(8,8) は、明示的引用仕様の有無で動作が違う。
- 文法と標準的実装の両方を理解すれば：
  - プログラム表現の幅が広がる。
  - 性能低下を防ぐことができる。

引数渡しの種類

- 大きく2つ
  - FORTRAN77までのインタフェース（F77 I/F）
    - （仕様）実引数から仮引数へ、配列要素順序で対応付けられる。サイズは実引数側が大きければ問題なし。
    - （実装）先頭アドレスが渡るだけ、と思えばよい。
  - Fortran90で追加されたインタフェース（F90 I/F）
    - （仕様）実引数の形状を仮引数に引継ぐことができる（形状引継ぎ配列）。
    - （実装）先頭アドレスだけでなく、仮引数のindex計算のために必要な情報（dope vector）が裏で渡される。
    - （実装）実引数が不規則で不連続な配列であるとき、受渡し/復帰時にデータがpack/unpackされることがある。

FORTRAN77の引数インタフェース

- 基本ルール
- サイズが呼出し時に決まるとき
- 配列形状の変更
- 列ベクトルの受渡し
- 部分配列の受渡し

基本ルール

- 変数（variable）と式（expression）の違い
  - Fortran文法では、配列も配列要素も部分配列も変数
  - 備考：Fortranの代入文は、

    変数 = 式

- 実引数は、変数または式。変数は書き戻し可。式は不可。

親変数	実引数	仮引数	説明	
real a	a	real x	変数の受け渡し。書き戻し可	
real a	a*1.0	real x	式の受け渡し。書き戻し不可	
real a	(a)	real x	式の受け渡し。文法的には書き戻し不可	
real b(10)	b	real y(10)	配列変数の基本的な受け渡し	書き戻し可
real b(10)	b(1)	real y(10)	配列要素→配列変数の受け渡し	
real b(10)	b(2)	real y(8)	配列の一部の区間を使用する。	

サイズが呼出し時に決まるとき(1/2)

- サイズを仮引数などで渡し、配列の宣言形状とする。
  - Common変数かモジュール変数で渡すこともできる

呼出し側	呼ばれ側	備考
real a(10, 20) … call foo(a, 10, 20)	subroutine foo(x, n1, n2) integer n1, n2 real x(n1, n2)	n1=10, n2=20が推奨だが、n1*n2≦200なら可
integer m, n common /foo_interf/ m, n real a(10, 20) m=…; n=… call foo(a)	subroutine foo(x) integer n1, n2 common /foo_interf/ n1, n2 real x(n1, n2)	
use sizes real a(10, 20) n1=…; n2=… call foo(a)	subroutine foo(x) use sizes real x(n1, n2)	モジュール定義: module sizes integer n1, n2 end module

サイズが呼出し時に決まるとき(2/2)

- 最終次元に限りサイズを省略できる(大きさ引継ぎ配列)
  - 仮引数の最終次元には * を書く
  - 仮引数最終次元の下限を指定することもできる

呼出し側	呼ばれ側	説明
real a(10, 20), b(30) … call foo(a, 10, b)	subroutine foo(x, n1, y) integer n1 real x(n1, *), y(*)	コンパイラは仮引数の最終次元の大きさを知る必要がない
同上	subroutine foo(x, n1, y) integer n1 real x(0:n1-1, 0:*), y(0:*)	0-boundaryにしたい場合 • “0:*-1” とは書かない • “0:” と書くと別の意味

引数渡しの形状変更は推奨される？

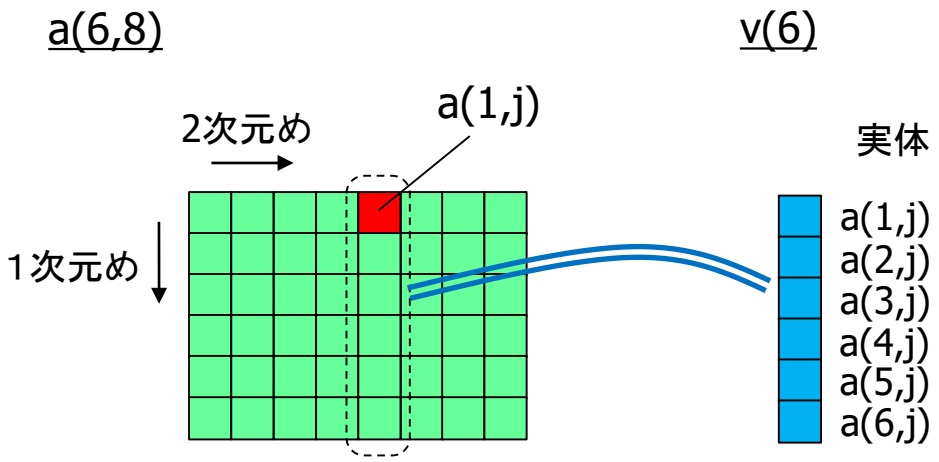
- プログラムを簡潔にし、高速化できることもある
  - 例: FFTE（www.fft.jp）の1次元FFTでは、本来1次元であるデータを、インデックス計算を簡単にするため局所的に2次元や3次元の配列として扱っている。

呼出し側	呼ばれ側
SUBROUTINE PZFFT1D(A,B,...,N,NPU) COMPLEX*16 A(*),B(*),... ... CALL PGETNXNY(N,NX,NY,NPU) ... CALL PZFFT1D0(...,A,...,B,...,NX,NY,NPU)	SUBROUTINE PZFFT1D0(...,AXPY,...,BXYP,...,NX,NY,NPU) COMPLEX*16 AXPY(NX/NPU,NPU,*) COMPLEX*16 BXYP(NX/NPU,NY/NPU,*) ... !$OMP DO !! OpenMPによるループ並列化 DO 30 J=1,NNY !! NNY=NY/NPU DO 20 K=1,NPU DO 10 I=1,NNX !! NNX=NX/NPU BXYP(I,J,K)=AXPY(I,K,J) !! ノード内で転置 10 CONTINUE 20 CONTINUE 30 CONTINUE !$OMP BARRIER !$OMP MASTER CALL MPI_ALLTOALL(BXYP, ...) !! ノード間で転置 !$OMP END MASTER

よくある例：列ベクトルの受渡し

- 行列の列ベクトルをサブルーチンに渡すとき

親変数	実引数	仮引数	説明
real a(6,8)	a(1, j)	real v(6)	F77 i/f での列の受け渡し（参照渡し）

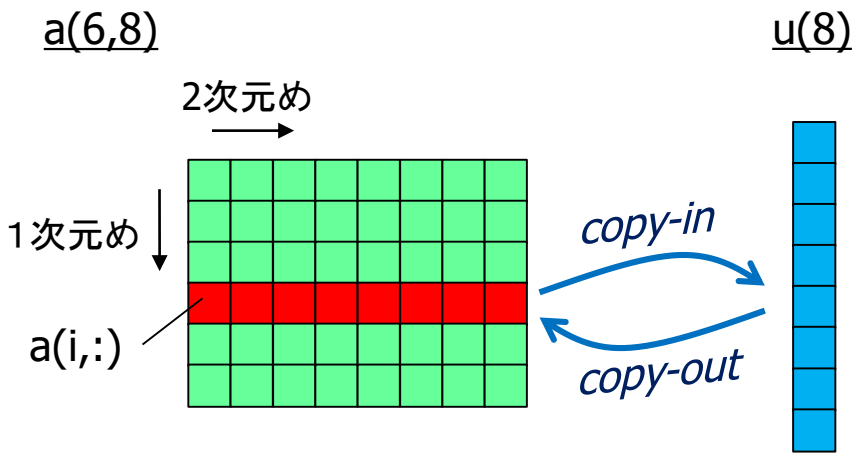


- 一般に、配列中の1要素を渡せば、要素の並び順でそれ以降の配列要素が参照可能

F77 i/fで部分配列を渡すと

- 不連続な部分配列は、必ずコピー渡し（copy-inとcopy-out）
- 連続な場合には、多くの実装ではコピーにならない（実装依存）

親変数	実引数	仮引数	説明
real a(6,8)	a(i, :)	real u(8)	明示的引用仕様がなければコピー渡し
real a(6,8)	a(:, j)	real v(6)	連続区間なので、コピー渡しか否かは実装依存



copy-in
- システムが確保した連続領域に値をpack

copy-out
- 値の書き戻し(unpack)

- 不連続な部分配列の参照渡しは、F90 i/fで可能

Fortran90で追加されたインタフェース

- 形状とは
- 形状引継ぎ配列
- 明示的引用仕様
- Interfaceブロック

「形状」の考え方

- 形状とは、「次元数」と、各次元の「大きさ」
  - 変数の宣言形状
    - スカラ変数は常に0次元
    - 配列変数の宣言形状は、宣言した次元数と各次元のサイズ
      - 例：宣言が a(10, 2:99, 100) なら、3次元、大きさ [10, 98, 100]
  - 部分配列と式の形状（F90以降）
    - スカラ変数・配列要素の引用は常に0次元
    - 配列変数の引用の形状は、その変数の宣言の形状と同じ。
    - 部分配列の形状は、そのスカラ添字以外を次元としてカウント
      - 例：同じaで a(10, 2:2, ix(1:10)) なら、2次元、大きさ [1, 10]
    - 配列式の形状は、演算のオペランドの形状と同じ
      - a + b は、a,bの一方が配列なら、他方は同じ形状の配列かスカラでなければならない。
      - 例：a + 1.0 の形状はaの形状と同じ。

形状引継ぎ配列(1/2)

- 実引数の形状を受け継ぐ仮引数(assumed-shape array)
  - 例

```
subroutine bar(x)
  real x(:,:)
  do j = 1, size(a,2)
    do i = 1, size(a,1)
      … x(i,j) …
end subroutine
```

色々な引数で呼べる

```
real a(10,10), b(10,10)
…
call bar( a )
call bar( a(1:9; 1:9) )
call bar( a(:, idx(1:10)) )
call bar( a+matmul(a,b) )
```

```
real w(10,20,30)
…
do j = 1,n
  call bar( w(:, j, :) )
end do
```

- (実装)形状の情報(dope vector)は、隠し引数で渡る。
  - コンパイラは、引数が形状引継ぎ配列かどうかを、明示的引用仕様(後述)から知る。
- 参照渡し/コピー渡しはコンパイラが選択
  - 添字にベクトル添字が1つでもあると、コピー渡し
  - 三つ組とスカラだけなら、多くの実装では参照渡し

形状引継ぎ配列(2/2)

- 性能は、コピー渡し(copy-in/out)と比べて
  - index計算のオーバヘッド増が少し
    - a(i,j,k) のアドレス = base + c1*i + c2*j + c3*k
      - 形状明示配列のとき、baseのみ呼び出し側から継承。
      - 形状引継ぎ配列のとき、baseと c1, c2, c3 を呼び出し側から継承。
  - 不連続アクセスに起因するオーバヘッド(キャッシュミスなど)が大
    - 形状明示配列のとき c1=1。ループ do i が最内側ならキャッシュ効率がよい。
    - 形状引継ぎ配列では、「不連続かもしれない」と考えて様々な最適化が低下
  - copy-in/outのコストがないので、大きな配列では有利か

明示的引用仕様とは

- 明示的引用仕様(explicit interface)とは、以下の3つ
  - 外部手続(従来の関数・サブルーチン)に対しては、Interfaceブロック
  - 内部手続に対しては、それ自体
  - モジュール手続に対しては、それ自体

内部手続barと、呼出し側foo

```
subroutine foo
  real a(10), b(10)
  ...
  call bar(a, b)
  ...
contains
  subroutine bar(x, y)
    real x(10), y(:)
    ...
  end subroutine
end subroutine
```

モジュール手続barと、呼び出し側foo

```
module mod_bar
contains
  subroutine bar(x, y)
    real x(10), y(:)
    ...
  end subroutine
end subroutine
```

```
subroutine foo
  use mod_bar
  real a(10), b(10)
  ...
  call bar(a, b)
  ...
end subroutine
```

- 形状引継ぎ配列をもつ手続を使うためには、必ず必要
  - (実装)呼び出し側の翻訳時、コンパイラは、手続の仮引数が形状引継ぎか否かを知って、呼出しのコードを変える

Interfaceブロック

- Interfaceブロック（引用仕様宣言）とは
  - 手続の名前、仮引数の名前と特性、関数結果の特性などの宣言
- Interfaceブロックが書かれていない外部手続は
  - どの引数も、形状引継ぎ配列でない（F77 i/fでよい）と判断される

Interfaceブロックの簡単な作り方

```
subroutine foo(a, b)
  real a, b
  interface
    subroutine bar(a, b)
      use params
      real a(n1,n2), b(:,:)
    end subroutine
  end interface
  …
  call bar(a, b)
  …
  return
end subroutine
```

```
subroutine bar(a, b)
  use params
  real a(n1,n2), b(:,:)
  …
  …          } 実行部
  return
end subroutine
```

実行部、DATA文、ENTRY文、
FORMAT文、文関数定義文
以外をコピー

モジュールにすれば共有できる

```
module bar_mod
  interface
    subroutine bar(a, b)
      use params
      real a(n1,n2), b(:,:)
    end subroutine
  end interface
end module
```

引数渡しの性能を上げる

- 引数渡しで起こる性能低下
- 回避法（1）、（2）、（3）

引数渡しで起こる性能低下

- 多くは、コンパイラの最適化を止めるか難しくてしまった場合
  - データ依存解析の前提を壊してしまうケース（重大）
    - 原因例：仮引数の不必要なポインタ/割付け配列宣言

    → ベクトル化/並列化、プリフェッチ、software pipeliningなどが止まる。
  - 連続な配列が連続でないように見えるケース（中間）
    - 原因例：形状引継ぎ配列の使い過ぎ

    → index計算やプリフェッチが非効率化。loop collapsingできない。
  - 本来は定数なのに変数に入れて持ち回るケース（軽微）
    - 原因例：引数配列の形状、ループ回転数などの不必要な変数化

    → コードの動的切り分けが増加。変数のindex計算が共通化できない。

- 回避するには
  (1) 仮引数をなるべくポインタ/割付け配列にしない。
  (2) 形状引継ぎ配列とF77インタフェースを使い分ける。
  (3) 定数はモジュールで共有し変数化を避ける。

(1) 仮引数をポインタ/割付け配列にしない

- ポインタ/割付け配列による性能低下
  - ポインタがループ内にあると、他の変数とのaliasの可能性から、ベクトル化/並列化、プリフェッチ、software pipeliningなどが止まる。
  - 割付け配列による性能低下は、ポインタほどではないが、ベースアドレスが一定でないためプリフェッチなどに影響する。
- 回避方法
  - 手続内（子孫を含む）でポインタ/割付け配列である必要がないなら、仮引数のPOINTER/ALLOCATABLE宣言を外す。

※ ポインタ/割付け配列を使っても、性能を諦める必要はない！

呼出し側

```
real, pointer :: a(:), b(:)
allocate ( a(N), b(N) )
…
call sub1( a, b, N )
```

手続側、悪い例
pointerである必要は？

```
subroutine sub1( a, b, n )
real, pointer :: a(:), b(:)
…
do i = 1, n
   a(i) = b(i)
```

手続側、良い例
非pointerと同じ性能が出る

```
subroutine sub1( a, b, n )
real :: a(n), b(n)
…
do i = 1, n
   a(i) = b(i)
```

(2) 形状引継ぎとF77 i/fの使い分け
形状引継ぎ配列を使いたいケース

- 手続きの中身に比べてcopy in/outのコストが大きい場合
  - 例：親変数が袖付きの巨大な配列で、袖を除く部分を引数渡し

袖の幅に関わらず使えるサブルーチン

```
real u(N1,N2)          !! 袖なし
real v(N1+2,N2+2)      !! 両側1枚袖
call do_body(u)
call do_body(v(2:N1+1,2:N2+1))
```

copyなし ⇄ copyなし

```
subroutine do_body(a)
  real a(:,:)
  ...
end subroutine
```

- wrapperとして
  - 例：引数渡しの煩わしさを軽減したライブラリの作成

利用者用の楽なインタフェース（形状引継ぎ）

copyなし

```
subroutine foo(a)
real:: a(:, :)
...
call foo_core( a, size(a,1), size(a,2) )
```

コア部分（形状明示配列を使って高速化）

copyあるかも

```
subroutine foo_core( a, n1, n2 )
real:: a(n1, n2)
...
do j=1,n2
  do i=1,n1
    ...
```

(2) 形状引継ぎとF77 i/fの使い分け
F77 i/f を使いたいケース

- copy in/outを積極的に利用
  - 例：MPI通信のデータのpack/unpackをFortranシステムに任せる。

```
real a(10,20), b(10,20)
…
call MPI_Allreduce( a(i, :), b(i, :), 20, MPI_REAL, MPI_SUM, … )
```

(1) a(i,:)からbuffer1にcopy-in
(2) buffer1からbuffer2へMPI_Allreduce
(3) buffer2からb(i, :)にcopy-out

  - MPI_Send/Recvでも同様に可能
- ただしMPI_Isend/Iredvなどnonblocking通信では使えない
  - MPIライブラリから復帰するとバッファ領域が解放されてしまうため

(3) 定数はモジュールで持ち回る

- 重要なパラメタは、モジュールで定数宣言して手続間で共有
- 性能を出すためには、忘れずにPARAMETER属性の宣言を

モジュール

```
module params
  integer, parameter ::    &
      n1=10, n2=20
end module
```

呼出し側

```
use params
real a(n1, n2)
…
call foo(a)
```

手続側

```
subroutine foo(x)
  use params
  real x(n1, n2)
  …
  do j = 1,n2
    do i = 1,n1
```

- ありがちな、良くない例

Includeファイル params.h

```
common /params/n1,n2
```

初期化手続の中で

```
include 'params.h'
n1=10
n2=20
```

呼出し側

```
include 'params.h'
real a(n1, n2)
…
call foo(a, n1, n2)
```

手続側

```
subroutine foo(x, n1, n2)
real x(n1, n2)
  …
  do j = 1,n2
    do i = 1,n1
```

まとめ

- F77インタフェース
  - 基本ルール：「変数」には書き戻しでき、「式」にはできない
  - 引数のサイズが呼出し時に決まるとき、引数で渡す
  - 引数の形状の変更は、うまく使えることもある。
  - 使用例として、列ベクトルの受渡しと、部分配列のcopy-in/out
- F90インタフェース
  - 宣言された配列、部分配列、式はすべて「形状」をもつ
  - 形状引継ぎ配列は、実引数から形状を引き継いだ仮引数
  - 形状引継ぎ配列の使用には、明示的引用仕様が必要
  - Interfaceブロックは、外部手続きの明示的引用仕様
- 引数渡しの性能を上げる
  - 引数周りの性能低下の原因の多くは、コンパイラ最適化のダウン
  - 仮引数のPOINTER/ALLOCATABLE宣言は、極力回避せよ
  - 形状引継ぎ配列と形状明示配列は、うまく使い分けよ
  - 定数でよいものは変数化せず、モジュールで持ち回れ

時間の都合で省略した関連項目

- オプショナル引数・キーワード引数
  - 手続の引数の数や位置が自由になります。
- 総称名と個別名
  - 引数の違う個別名手続が、共通の総称名で呼び出せます。
- INTENT(IN/OUT/INOUT)
  - コンパイラに対し、引数の参照・定義の有無を知らせます。
- CONIGUOUS宣言
  - コンパイラに対し、変数が連続であることを知らせます。