

A Translation of a Fortran Program into a Fortress Program

Kento Emoto

emoto@ipl.t.u-tokyo.ac.jp

Graduate School of Information Science and Technology, University of Tokyo

Abstract

This report shows a strategy for program translation from Fortran to Fortress, and points out some issues about the translation, through an example translation of a practical Fortran program into a Fortress program, which implements the BiCGSTAB method.

1 Introduction

Fortress [ACH⁺08, For08] is a new programming language designed for high-performance computing (HPC) with high programmability. Fortress features include implicit parallelism, mathematical syntax, type inference, and definition of large parts of the language in its own libraries. Its slogan is “do for Fortran what Java did for C.”

The purpose of the experimental translation is to validate the easiness of program translation from Fortran to Fortress, and to point out issues about the translation. We especially focus on the easiness of parallelization owing to the translation.

This report shows a strategy for program translation from Fortran to Fortress, and points out some issues about the translation, through an example translation of a practical Fortran program into a Fortress program, which implements the BiCGSTAB method [vdV92].

2 Translation to Fortress

This section summarizes the translation of a Fortran program for the BiCGSTAB method into a Fortress program. Complete codes of these Fortran and Fortress programs are listed in Appendix A.

The following points should be kept in mind during the translation: shifts of index spaces of arrays, a distinction between variables to be updated and those not to be updated, and a distinction between loops to be parallelized and those not to be parallelized. Considering these points, we translate the Fortran program according to the following strategy.

- Index spaces are shifted in order not to use negative indices. Also, they should be started from zero to use linear algebraic operations defined in the Fortress library.
- A linear algebraic operation in Fortran is replaced with a corresponding linear algebraic operation in Fortress.
- A loop used to write independent operations with a uniform expression is replaced with a for-loop with generators.
- A loop to iterate a certain process in order is replaced with a while-loop.

The first point about index spaces is a workaround for the restrictions on indices in Fortress, which is discussed in Section 3. The second point makes the translated Fortress program clear. The last two points are for making the translated program a correct parallel program.

```

subroutine probmv(ap, ae, aw, an, as, at, ab, b, c, imax, jmax, kmax)
  !Computes the product c of a band matrix A and a vector b
  !(ap, ae, aw, an, as, at, ab) are the bands of A
  implicit none
  integer, intent(in) :: imax, jmax, kmax
  real(8), intent(in), dimension(imax, jmax, kmax) :: ap, ae, aw, an, as, at, ab
  real(8), intent(in) :: b(-imax*jmax+1:imax*jmax*kmax+imax*jmax)
  real(8), intent(out) :: c(-imax*jmax+1:imax*jmax*kmax+imax*jmax)
  integer i, j, k, m
  do k=1, kmax
  do j=1, jmax
  do i=1, imax
  m=i+imax*(j-1)+imax*jmax*(k-1)
  c(m) = ap(i, j, k)*b(m) +aw(i, j, k)*b(m-1) +ae(i, j, k)*b(m+1) &
    +as(i, j, k)*b(m-imax) +an(i, j, k)*b(m+imax) &
    +ab(i, j, k)*b(m-imax*jmax) +at(i, j, k)*b(m+imax*jmax)
  end do
  end do
  end do
end subroutine

```

Figure 1: The Fortran routine probmv to take a product of a band matrix and a vector.

The target Fortran program consists of three parts: a routine to take a product of a band matrix and a vector, a routine implementing the main loop of the BiCGSTAB method, and a sample driver to use the BiCGSTAB method. We report translations of the former two parts. ¹

2.1 Translation of the Band Matrix-Vector Product

This section shows translation of the Fortran routine probmv (Figure 1) to take a product of a band matrix and a vector. Figure 2 shows the translated Fortress function.

2.1.1 Fortran Routine probmv

The Fortran routine probmv takes the following arguments.

- ap, ae, aw, an, as, at, ab
Each of these arrays stores a band of the band matrix. Index spaces of these arrays are 3D corresponding to the physical space. The size of each array is (imax, jmax, kmax), in which imax, jmax, and kmax are also given in the arguments.
- b
A vector to be multiplied with the band matrix². Elements in the region (1:imax*jmax*kmax) have corresponding point in the physical space and are physically meaningful, while other imax*jmax points on each edge are physically meaningless but used to simplify the loop in the program. This vector is a linearized array of a 3D array with the same index space as bands of the matrix. The relation between a linear index m and a 3D index (i,j,k) is given by $m=i+imax*(j-1)+imax*jmax*(k-1)$.
- c
A vector to which the product is stored. The index space of this vector is the same as the vector b.

¹We modified the original Fortran program slightly to simplify loops in the program. See Appendix B for details.

²In this report, we will call a 1D array a vector when it is used as a mathematical vector in the Fortran program.

```

MatrixVectorProduct[[nat imax,nat jmax,nat kmax]]
(Ap, Ae, Aw, An, As, At, Ab, b, c) : () =
do
  for k ← 0 # kmax do
    for j ← 0 # jmax do
      for i ← 0 # imax do
        m = i + imax j + imax jmax k + imax jmax
        cm := Ap[i, j, k] bm + Aw[i, j, k] bm-1 + Ae[i, j, k] bm+1 + As[i, j, k] bm-imax + An[i, j, k] bm+imax
          + Ab[i, j, k] bm-imaxjmax + At[i, j, k] bm+imaxjmax
      end
    end
  end
end

```

Figure 2: The Fortress function `MatrixVectorProduct` translated from the Fortran routine `probmv` (Figure 1)

- `imax`, `jmax`, `kmax`
These integers specify the size of the 3D index space.

The vector `c` is output. The others are input.

The body of Fortran routine `probmv` is a simple nested loop. The loop iterates its body over the 3D index space (`1:imax`, `1:jmax`, `1:kmax`). In each point of the index space, its body takes the inner product of the vector `b` and the seven elements in a row of the band matrix, and then it stores the result in the output vector `c`. The computations carried out over the iteration space are independent from each other.

2.1.2 Fortress Function `MatrixVectorProduct`

First, we rewrite the arguments. The translated Fortress function `MatrixVectorProduct` takes the same arguments as the Fortran routine `probmv`, except that `imax`, `jmax`, and `kmax` are given as static parameters (`[[nat imax,nat jmax,nat kmax]]`) of the function. This is because the index space is usually determined statically and fixed during the execution of the program. It receives as dynamic parameters other arguments: seven arrays `Ap`, `Ae`, `Aw`, `An`, `As`, `At` and `Ab` to store the band matrix, vector `b` to be multiplied with the matrix, and vector `c` to which the product is stored. Types of the arguments are not specified in the program, because they are inferred by Fortress.

Next, we translate the tree-nested loop in the body. Because the computations carried out by the loop body are independent, we simply replace do-loops with for-loops. For example, `do k=1,kmax ... end do` is replaced with `for k ← 0 # kmax do ... end` (`0 # kmax` generates a sequence of `kmax` integers from 0). The only non-trivial point is that the indices were changed in order that they start from 0, to avoid some restrictions introduced by non-zero based indexing (see Section 3 for details).

Then, we translate the body expression of the loop. Keeping the structure of the expression, we replaced the product operator `*` with a space (i.e., juxtaposition operator), index-access `()` with `[]`, and the assign operator with `:=`. It is worth noting that in Fortress assignments to updatable variables are denoted by operator `:=`, and bindings to not updatable variables are denoted by operator `=`.

These are the steps of the translation. The translation is systematic and easy.

Finally, we mention the parallelization of the program. The for-loops used in the translation of the nested loop execute its body in parallel, because they use generators such as `0 # kmax`. Therefore, the parallelization was done by replacing do-loops with for-loops, and was very easy.

2.2 Translation of the Main Loop of BiCGSTAB

This section shows translation of the Fortran routine `bicgstab.band` (Figure 3) that implements the main loop of the BiCGSTAB method. Figure 4 shows the translated Fortress function.

This routine takes a band-matrix A and a vector b , and return a solution vector x such that $Ax = b$. To compute the solution x , it uses matrix-vector multiplications, inner products, and algebraic operations on vectors repeatedly in the main loop.

2.2.1 Fortran Routine `bicgstab_band`

The Fortran routine `bicgstab_band` takes the following arguments.

- `ap, ae, aw, an, as, at, ab`
These arrays are the same as those in the routine `probmv`.
- `x`
The solution vector of the linear equation. Similar to the vectors in the routine `probmv`, the index space of this vector is extended with zeros.
- `b`
The given vector of the linear equation. Similar to the vectors in the routine `probmv`, the index space of this vector is extended with zeros.
- `imax, jmax, kmax`
These integers specify the size of the 3D index space.
- `it`
The number of iterations of the main loop.
- `rr`
The residual at the end of the main loop.

Here, `x`, `it`, and `rr` are output. The others are input.

The Fortran routine `bicgstab_band` first allocates vectors `r`, `r0`, `p`, `y`, `e`, `v`, and `c` used in the main loop. These vectors have the extended index space similar to `x` and `b`. The routine then computes the residual vector $r = b - Ax$ and initializes `c1`, `p`, and `r0`, to start the main loop. The main loop uses matrix-vector products (`probmv`), inner products (`dot_product`), linear algebraic operations on vectors such as $e = r - \alpha p$, and other scalar operations. The program exits the main loop when the residual `rr` becomes less than `er0`.

2.2.2 Fortress Function `BiCGSTABband`

First we translate the arguments of the routine. The translated Fortress function returns the pair of the number of iterations and the residual, and takes the other arguments of the Fortran routine `bicgstab_band`. The parameters `imax`, `jmax`, and `kmax` to specify the index space are given as static parameters, similar to the Fortress function `MatrixVectorProduct`. It receives as dynamic parameters other arguments: seven arrays `Ap`, `Ae`, `Aw`, `An`, `As`, `At` and `Ab` to store the band matrix, vector `x` to which the solution is stored, and the given vector `b` of the linear equation. Types of the arguments are not specified in the program, because they are inferred by Fortress.

Next, we translate the allocation of vectors into declaration of variables. Because these variables are to be updated in the main loop, we declare them as updatable variables (since we want to update two vectors given as the arguments, those vectors x_2 and b_2 are re-declared as x and b). We use vector type `Vector[$\mathbb{R}64, (imax\ jmax\ kmax) + (2\ jmax\ imax)$]` for them to use linear algebraic operations and inner products on vectors³. Here, `[[$\mathbb{R}64, (imax\ jmax\ kmax) + (2\ jmax\ imax)$]]` are static parameters to mean that they have $(imax\ jmax\ kmax) + (2\ jmax\ imax)$ elements of 64bit real numbers.

Then, we translate the main loop. The main loop must be exited when the residual becomes small enough to finish the computation. To this end, we have to specify the region of such escaping by a label-expression in Fortress. Thus, the translated program has the label-expression labeled with `MainLoop`.

³Fortress defines 1D arrays with mathematical operations, such as linear algebraic operations and inner products, as vectors.

```

subroutine bicgstab_band (ap , ae , aw , an , as , at , ab , x , b , imax , jmax , kmax , it , rr )
implicit none
! Solves Ax=b using Bi-CGSTAB method
integer , intent (in) :: imax , jmax , kmax
real (8) , intent (in) , dimension (imax , jmax , kmax) :: ap , ae , aw , an , as , at , ab
real (8) , intent (in) , dimension (-imax*jmax+1:imax*jmax*kmax+imax*jmax) :: b
real (8) , intent (inout) , dimension (-imax*jmax+1:imax*jmax*kmax+imax*jmax) :: x
real (8) , intent (out) :: rr
integer , intent (inout) :: it
integer itrmax
real (8) alp , bet , c1 , c2 , c3 , ev , vv , er0
real (8) , allocatable :: r (:) , r0 (:) , p (:) , y (:) , e (:) , v (:) , c (:)
allocate (r (-imax*jmax+1:imax*jmax*kmax+imax*jmax) , &
r0 (-imax*jmax+1:imax*jmax*kmax+imax*jmax) , &
p (-imax*jmax+1:imax*jmax*kmax+imax*jmax) , &
y (-imax*jmax+1:imax*jmax*kmax+imax*jmax) , &
e (-imax*jmax+1:imax*jmax*kmax+imax*jmax) , &
v (-imax*jmax+1:imax*jmax*kmax+imax*jmax) , &
c (-imax*jmax+1:imax*jmax*kmax+imax*jmax))
itrmax = 300
!itrmax = nmax
!The upper bound of the number of iterations
er0 = 1.0d-6
!the feasible error to stop the iteration
call probmv (ap , ae , aw , an , as , at , ab , x , c , imax , jmax , kmax)
!computes c = Ax
!i.e. , r = b - matmul(a , x)
r = b - c
c1=dot_product (r , r)
if (c1 < er0) return
p = r
r0 = r
do it=1,itrmax
call probmv (ap , ae , aw , an , as , at , ab , p , y , imax , jmax , kmax)
!i.e. , y = matmul(a , p)
c2 = dot_product (r0 , y)
alp = c1/c2
e = r -alp*y
call probmv (ap , ae , aw , an , as , at , ab , e , v , imax , jmax , kmax)
!i.e. , v = matmul(a , e)
ev = dot_product (e , v)
vv = dot_product (v , v)
c3 = ev/vv
x = x + alp*p + c3*e
r = e - c3*v
rr = dot_product (r , r)
write (* , *) it , ' residual=' , rr
if (rr<er0) exit
c1 = dot_product (r0 , r)
bet = c1/(c2*c3)
p = r + bet*(p-c3*y)
end do
deallocate (r , r0 , p , y , e , v)
end subroutine

```

Figure 3: The Fortran routine bicgstab_band that implements the main loop of the BiCGSTAB method.

```

BiCGSTABband[[nat imax,nat jmax,nat kmax]](Ap, Ae, Aw, An, As, At, Ab, x2, b2) : (Z32, R64) =
do
  x : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := x2
  b : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b2
  r : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b.copy()
  r' : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b.copy()
  p : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b.copy()
  y : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b.copy()
  e : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b.copy()
  v : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b.copy()
  c : Vector[[R64, (imax jmax kmax) + (2 jmax imax)]] := b.copy()
  (* we dont need so many iterations because the matrix is a band matrix. *)
  itrmax = imax jmax kmax
  er0 = 1.0 10-4
  MatrixVectorProduct[[imax, jmax, kmax]](Ap, Ae, Aw, An, As, At, Ab, x, c)
  r := b - c
  c1 : R64 := r · r
  label MainLoop
    if (c1 < er0) then exit MainLoop with () end
    p := r
    r' := r
    it : Z32 := 1
    rr : R64 := 0
    while it ≤ itrmax do
      MatrixVectorProduct[[imax, jmax, kmax]](Ap, Ae, Aw, An, As, At, Ab, p, y)
      c2 = r' · y
      alp = c1/c2
      e := r - alp y
      MatrixVectorProduct[[imax, jmax, kmax]](Ap, Ae, Aw, An, As, At, Ab, e, v)
      ev = e · v
      vv = v · v
      c3 = if (vv = 0) ∧ (ev = 0) then 1 else ev/vv end
      x := x + alp p + c3 e
      r := e - c3 v
      rr := r · r
      println(it " residual = " rr)
      if (rr < er0) then exit MainLoop with (it, rr) end
      c1 := r' · r
      bet = c1/(c2 c3)
      p := r + bet(p - c3 y)
      it += 1
    end
  (it, rr)
end MainLoop
end

```

Figure 4: The Fortress function BiCGSTABband translated from the Fortran routine bicgstab.band (Figure 3).

```

(* * A ReadableArray1[[T, b0, s0]] is an arbitrary 1 - dimensional array
   whose s0 elements are of type T, and whose lowest index is b0.

   The natural order of all generators is from b0 to b0 + s0 - 1. * *)
trait ReadableArray1[[T, nat b0, nat s0]]
    extends { Indexed1[[s0]], Rank1, ReadableArray[[T, Z32]] }
    comprises { ImmutableArray1[[T, b0, s0]], Array1[[T, b0, s0]] }
    getter size(): Z32 = s0
    (* * omitted * *)
end

```

Figure 5: A part of the definition of 1D arrays. The parameter b_0 for the lower bound of indices is declared as a non-negative integer.

The do-loop of the main loop is replaced with a while-loop, because the do-loop is used to iterate its body in order. Basically, we think that for-loops in Fortress should be used for parallel computation with generators, and while-loops should be used for sequential iterations ⁴.

The translation of the main loop is straightforward. Calls of the routine `probmv` are replaced with calls of `MatrixVectorProduct`. For linear algebraic operations on vectors, the operator `*` for scalar multiplication is replaced with a space, and the assignment operator is replaced with `:=`. Scalar operations are translated by the similar replacements. Inner products `dot_product` are replaced with the inner product operator `.`. The escape from the loop is translated by replacing `exit` with `exit MainLoop with (it, rr)`, in which `with (it, rr)` means the loop is escaped with the return values `(it, rr)`.

These are the steps of the translation. The translation is systematic and easy.

Finally, we mention the parallelization of the program. Although the main loop itself is not parallelizable, some parts in its body can be parallelized. In fact, matrix-vector products are parallelized by calls of `MatrixVectorProduct`, and linear algebraic operations on vectors and inner products are parallelized because the standard library provides their parallel implementations. There is no room to parallelize the other parts in the main loop. Therefore, the parallelization was done by the systematic translation, and was very easy.

3 Issues on Translation to Fortress

This section discusses two issues about indices of arrays in the translation from Fortran to Fortress. These issues appear now (late March 2009) in the current Fortress implementation.

3.1 Negative Index of Arrays Is Not Allowed in Fortress

An array in Fortress receives the lower bound of its index space as a static parameter (Figure 5). However, the parameter type is `nat`, and it does not allow any negative lower bound. Thus, no negative index of arrays is allowed in Fortress. This restriction requires us to shift the index spaces in order that all index spaces start from non-negative integers. Such shifts of index spaces coerce us into modification of various program codes, such as ranges of loops and expressions with index accesses, as well as declarations of arrays. This modification makes the translation difficult.

For example, we often want to use negative index to put sentinels or default values in the negative region of the index space. Let's consider the following loop. It computes a sum of an element and the preceding element of the array `b` (its index is from 0 to 9).

```

for i ← 0 # 10 do
  if i = 0 then ci = bi

```

⁴We can use another approach, i.e., for-loops with sequential generators such as `for it ← seq(0 # itmax) do ... end`. This approach, however, have some overheads owing to sequentialization of generators.

```

        else  $c_i = b_{i-1} + b_i$ 
    end
end
l = c9

```

Since the index of the array b is valid only for the range from 0 to 9, the element b_{-1} is not present and the loop uses the conditional branch to avoid the access to b_{-1} . But a conditional branch in a loop body reduces efficiency. To improve the efficiency of the program, we want to simplify the loop as follows⁵, introducing a sentinel (a default value) $b_{-1} = 0$.

```

(* this is not correct! *)
b[-1] = 0
for i ← 0 # 10 do
     $c_i = b_{i-1} + b_i$ 
end
l = c9

```

But we cannot use negative indices now, we have to shift the index by one. The result is shown below.

```

b0 = 0
for i ← 1 # 10 do
     $c_i = b_{i-1} + b_i$ 
end
l = c10

```

The shift of the index requires modifications of the range of loop (0 # 10 to 1 # 10) and the index access ($l = c_9$ to $l = c_{10}$). These modifications make the translation difficult and increase the chance of embedding bugs.

3.2 Non Zero-based Indexing Prohibits Us From Using Algebraic Operations on Vectors in Fortress

Fortress provides traits Vector and Matrix for vectors and matrices with linear algebraic operations on them. But the index of a vector is restricted to start from zero, according to its definition shown in Figure 6. Therefore, when a Fortran program uses vectors with non-zero based indexing and linear algebraic operations on them, we have to either shift the index spaces to use linear algebraic operations provided by Fortress, or write codes for element wise computation to perform the linear algebraic operations.

Let's consider a translation of the following Fortran routine. This routine adds x and y scaled with s , stores the result into z , and finally assigns the head of z into e .

```

subroutine work(xmax)
implicit none
integer , intent(in) :: xmax
real(8) , dimension(:), allocatable :: x, y, z
real(8) s, e
allocate(x(xmax), y(xmax), z(xmax))
s=10
x=1
y=2

z = x + s * y
e = z(1)
end subroutine

```

⁵We can take another approach for this example, excluding the first case ($i = 0$) from the loop and computing it outside the loop. However, if the program uses nested loops, this approach would reduce productivity because of many code duplications.

```

trait Vector[[T extends Number, nat s0]]
  extends { AnyVector, Array1[[T, 0, s0]], AdditiveGroup[[Vector[[T, s0]]] ] }
  excludes { AnyMultiplicativeRing }
  opr +(self, v: Vector[[T, s0]]): Vector[[T, s0]] =
    ivmap[[T]](fn (i: Z32, e: T): T => e + v.get(i))
  opr -(self, v: Vector[[T, s0]]): Vector[[T, s0]] =
    ivmap[[T]](fn (i: Z32, e: T): T => e - v.get(i))
  opr -(self): Vector[[T, s0]] = map[[T]](fn (e: T): T => -e)
  scale(t: T): Vector[[T, s0]] = map[[T]](fn (v) => t v)
  pmul(v: Vector[[T, s0]]): Vector[[T, s0]] =
    ivmap[[T]](fn (i: Z32, e: T): T => e v.get(i))
  dot(v: Vector[[T, s0]]): T =
    Σ[(i, me_i) ← self.indexValuePairs] me_i v.get(i)
end

```

Figure 6: Trait Vector implements vector operations in Fortress. The index starts from 0.

It is worth noting that the expression `x(nmax)` in the allocation is an abbreviation of `x(1:nmax)`. Arrays in this program use one-based indexing and linear algebraic operations such as `z = x + s * y` at the same time. Therefore, this program needs non-straightforward modification to be translated into a Fortress program.

One approach to the translation is to shift the index spaces and use algebraic operations on vectors. The translated program is shown below.

```

work[[nat xmax]](): () = do
  x = array1[[R64, xmax]]().fill(fn (i: Z32): R64 => 1.0)
  y = array1[[R64, xmax]]().fill(fn (i: Z32): R64 => 2.0)
  s = 10
  z = x + s y
  e = z0
end

```

The advantage of this approach is that the translated program is clear owing to the use of linear algebraic operations on vectors. On the other hand, the shift of the index spaces requires modifications on various parts in the program. For example, the index access in the last expression of the above program is changed from 1 to 0. Moreover, it would be difficult to shift index spaces of many vectors similarly, when they essentially have different lower bounds of indices.

Another approach to the translation is to write codes for element wise computation corresponding to linear algebraic operations, without shifting the index spaces. The translated program is shown below.

```

work[[nat xmax]](): () = do
  x = array1[[R64, xmax]]().shift(1).fill(fn (i: Z32): R64 => 1.0)
  y = array1[[R64, xmax]]().shift(1).fill(fn (i: Z32): R64 => 2.0)
  z = array1[[R64, xmax]]().shift(1)
  s = 10
  zi := xi + s yi, i ← x.indices()
  e = z1
end

```

Linear algebraic operations can be replaced with element wise computations using generators. Fortress provides a concise notation for such computations like `zi := xi + s yi, i ← x.indices()`. Here, `x.indices()` is a generator to produce all indices of the array `x`. This notation means that the expression before the comma is evaluated against each value generated by `i ← x.indices()`, i.e., all indices of the array `x`. This approach

Table 1: Specifications of machines used in the experiment.

Name	CPU	Memory	OS	Java	Fortress
Machine1	Intel®Xeon®E5430 x2 (8 cores)	8GB	Linux 2.6.24	JDK 1.6.0_07	svn r3580
Machine2	Intel®Xeon®X7460 x4 (24 cores)	16GB	Linux 2.6.18	JDK 1.6.0_13	svn r3580

Table 2: Execution time and speedup on Machine1.

FORTRESS_THREADS	1	2	3	4	5	6	7	8
Execution time	272.2	153.3	109.9	101.1	97.8	111.1	109.1	112.0
Speedup	1.00	1.78	2.48	2.69	2.78	2.45	2.49	2.43

does not shift any index space, we do not need to modify other program codes. However, the translated element wise computations make the program slightly dirty.

4 Experiment Results

We measured execution time of the translated Fortress program on two parallel machines. It is worth noting that currently a Fortress program is executed by an interpreter [For08] and its absolute execution time is meaningless until a compiler will become available.

Table 1 specifications of machines used in the experiment.

Tables 2 and 3 list execution time and speedup of the translated Fortress program shown in Appendix A. Figures 7 and 8 show speedup of the program. The environmental variable FORTRESS_THREADS specifies the number of threads used in the execution of the Fortress program.

The tables and figures show good speedup for small numbers of threads. However, the speedup is limited for over five threads. This limitation is also observed for the following simple parallel loop.

```

run(args : String...) = do
  f(i) = do
    j : Z32 := 0
    while j < 100000 do
      j += 1
    end
  end
end
st = nanoTime()
for i ← 1:512 do
  f(i)
end
et = nanoTime()
t = et - st
ss = t/1000000000.0
println("elapsed time " t " ns ( " ss " s)")
end

```

Therefore, we think that this limitation of parallelization is the performance limitation (including the saturation of the memory bandwidth caused by interpreter execution) of the current interpreter.

5 Conclusion

In this report, we translated a Fortran program into a Fortress program according to the following strategy.

Table 3: Execution time and speedup on Machine2.

FORTRESS_THREADS	1	2	3	4	5	6	7	8	9	10	11	12
Execution time	304.8	185.4	134.8	106.6	92.4	84.8	81.3	82.0	82.4	81.9	83.1	82.8
Speedup	1.00	1.64	2.26	2.86	3.30	3.59	3.75	3.72	3.70	3.72	3.67	3.68
FORTRESS_THREADS	13	14	15	16	17	18	19	20	21	22	23	24
Execution time	82.4	83.9	83.0	84.4	85.1	85.5	86.8	87.8	93.6	90.0	91.5	92.0
Speedup	3.70	3.63	3.67	3.61	3.58	3.57	3.51	3.47	3.26	3.39	3.33	3.31

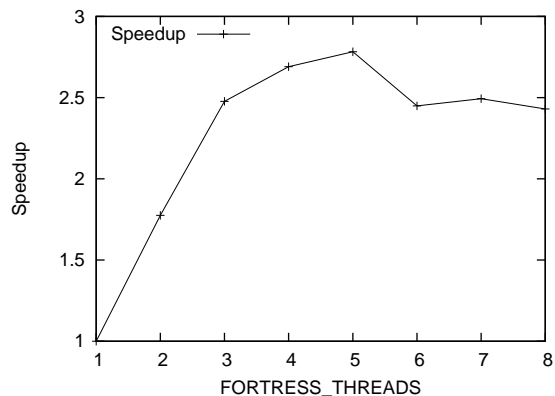


Figure 7: Speedup on Machine1.

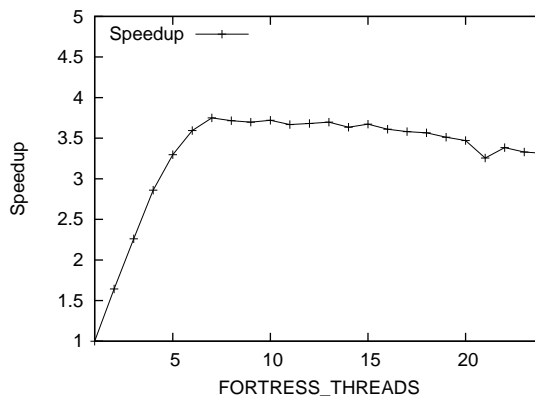


Figure 8: Speedup on Machine2.

- Index spaces are shifted in order not to use negative indices. Also, they should be started from zero to use linear algebraic operations defined in the Fortress library.
- A linear algebraic operation in Fortran is replaced with a corresponding linear algebraic operation in Fortress.
- A loop used to write independent operations with a uniform expression is replaced with a for-loop with generators.
- A loop to iterate a certain process in order is replaced with a while-loop.

The following points should be kept in mind during the translation: shifts of index spaces of arrays, a distinction between variables to be updated and those not to be updated, and a distinction between loops to be parallelized and those not to be parallelized.

The translation is systematic, except for shifts of index spaces. Moreover, the systematic translation achieves parallelization at the same time. The effective of the parallelization was shown by the experiment results.

There are two issues about indices in the translation form Fortran to Fortress. One is that no negative index is allowed in Fortress. The other is that linear algebraic operations in Fortress' standard library are provided only for vectors with zero-based indexing. These points requires shifts of index spaces, and that makes the translation hard. These points would be solved in the future Fortress library.

Acknowledgments

We would like to thank kind researchers for providing us with the Fortran code. Also, we would like to thank Mr. Hanabusa for giving us this good opportunity of interesting research.

References

- [ACH⁺08] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukeyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0. <http://research.sun.com/projects/plrg/fortress.pdf>, 2008.
- [For08] Project Fortress. The reference interpreter for the Fortress language. <http://projectfortress.sun.com/Projects/Community>, 2008.
- [vdV92] Henk A. van der Vorst. Bi-cgstab: a fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992.

A Complete Codes

A.1 Fortran Program

```
! Modified to use a single loop with the the general expression in probmv.
! Each vector has extra imax*jmax elements on both edges
!

! program bicgstab
! A sample program to use the bicgstab routine.
! subroutine make_vector(ap,ae,aw,an,as,at,ab,b,imax,jmax,kmax)
! A subroutine to make sample matrix and vector.
! subroutine bicgstab_band(ap,ae,aw,an,as,at,ab,x,b,imax,jmax,kmax,it,rr)
! A subroutine to solve a linear equation of a band matrix with BiCGSTAB
! subroutine probmv(ap,ae,aw,an,as,at,ab,b,c,imax,jmax,kmax)
! A subroutine to take a product of a band-matrix and a vector

program bicgstab
implicit none
integer ,parameter :: imax=160
integer ,parameter :: jmax=160
integer ,parameter :: kmax=160
real(8),dimension (:, :, :), allocatable :: ap, ae, aw, an, as, at, ab
real(8),dimension (:), allocatable :: b, x
real(8) rr
integer nmax, i, it
!+++++
!Given a matrix A and a vector b, this program find the vector x s.t. Ax=b.
!Variables used in the program:
!imax,jmax,kmax: each of them indicates the size of the dimension.
!ap,ae,aw,an,as,at,ab: bands of the matrix A.
!rr: the residual
!it: the number of iteration during the BiCGSTAB method
!+++++

!*****
! Measuring computation time
integer*4 CNT_RATE,CNT_MAX,it0, it1, itb, ite
double precision tick
call system_clock(count_rate=CNT_RATE,count_max=CNT_MAX)
tick=1.0d0/CNT_RATE
write(*,*) 'count_rate=',CNT_RATE, 'count_max=',CNT_MAX
write(*,'(A,E10.3)') 'counter tick= ',tick, ' seconds'
call system_clock(it0)
!*****

allocate (ap ( imax , jmax , kmax ) , ae ( imax , jmax , kmax ) , aw ( imax , jmax , kmax ) , &
an ( imax , jmax , kmax ) , as ( imax , jmax , kmax ) , at ( imax , jmax , kmax ) , ab ( imax , jmax , kmax ))
allocate ( b ( - imax * jmax + 1 : imax * jmax * kmax + imax * jmax ) , &
x ( - imax * jmax + 1 : imax * jmax * kmax + imax * jmax ))

x = 1.0d0+1.0d-2
do i=-imax*jmax+1,0
x(i)=0
end do
```

```

do i=imax*jmax*kmax+1,imax*jmax*kmax+imax*jmax
x(i)=0
end do

!initial solution
call make_vector(ap,ae,aw,an,as,at,ab,b,imax,jmax,kmax)
!making sample matrix and vector
call bicgstab_band(ap,ae,aw,an,as,at,ab,x,b,imax,jmax,kmax,it,rr)
!calling BiCGSTAB method

!do i=-imax*jmax+1,imax*jmax*kmax+imax*jmax
!write(*,*) i,' ',x(i)
!end do

!do i=1,nmax
!write(*,*) i,x(i)
!end do
write(*,*) 'iteration=',it
write(*,*) 'residual=',rr
deallocate(ap,ae,aw,an,as,at,ab,b,x)

!*****
!measuring computation time
call system_clock(it1)
write(*,*) 'init: a',(it1-it0)*tick
!*****

end program

subroutine make_vector(ap,ae,aw,an,as,at,ab,b,imax,jmax,kmax)
implicit none
integer, intent(in)::imax,jmax,kmax
real(8),dimension(imax,jmax,kmax),intent(out)::ap,ae,aw,an,as,at,ab
real(8),dimension(-imax*jmax+1:imax*jmax*kmax+imax*jmax),intent(out)::b
integer i,j,k,m,n
n = imax*jmax*kmax
b=0
do m=1,n
b(m) = 2.0d0
end do

do k=1,kmax
do j=1,jmax
do i=1,imax
m=i+imax*(j-1)+imax*jmax*(k-1)
if((i>1).and.(i<imax))then
b(m) = b(m) + 2.0d0
else
b(m) = b(m) + 1.0d0
endif

if((j>1).and.(j<jmax))then
b(m) = b(m) + 2.0d0
else
b(m) = b(m) + 1.0d0
endif

if((k>1).and.(k<kmax))then
b(m) = b(m) + 2.0d0
else
b(m) = b(m) + 1.0d0
endif

end do
end do
end do
ap = 2.0d0
ae = 1.0d0
aw = ae
an = ae
as = ae
at = ae
ab = ae

```

```

do j=1,jmax
do k=1,kmax
aw(1,j,k) = 0
ae(imax,j,k) = 0
end do
end do
do i=1,imax
do k=1,kmax
as(i,1,k) = 0
an(i,jmax,k) = 0
end do
end do
do i=1,imax
do j=1,jmax
ab(i,j,1) = 0
at(i,j,kmax) = 0
end do
end do

end subroutine

subroutine bicgstab_band(ap,ae,aw,an,as,at,ab,x,b,imax,jmax,kmax,it,rr)
implicit none
! Solves Ax=b using Bi-CGSTAB method
integer, intent(in):: imax,jmax,kmax
real(8), intent(in), dimension(imax,jmax,kmax):: ap,ae,aw,an,as,at,ab
real(8), intent(in), dimension(-imax*jmax+1:imax*jmax*kmax+imax*jmax):: b
real(8), intent(inout), dimension(-imax*jmax+1:imax*jmax*kmax+imax*jmax):: x
real(8), intent(out):: rr
integer, intent(inout):: it
integer itrmax
real(8) alp,bet,c1,c2,c3,ev,vv,er0
real(8), allocatable:: r(:),r0(:),p(:),y(:),e(:),v(:),c(:)
allocate(r(-imax*jmax+1:imax*jmax*kmax+imax*jmax),&
r0(-imax*jmax+1:imax*jmax*kmax+imax*jmax),&
p(-imax*jmax+1:imax*jmax*kmax+imax*jmax),&
y(-imax*jmax+1:imax*jmax*kmax+imax*jmax),&
e(-imax*jmax+1:imax*jmax*kmax+imax*jmax),&
v(-imax*jmax+1:imax*jmax*kmax+imax*jmax),&
c(-imax*jmax+1:imax*jmax*kmax+imax*
itrmax = 300
!itrmax = nmax
!The upper bound of the number of iterations
er0 = 1.0d-6
!the feasible error to stop the iteration
call probmv(ap,ae,aw,an,as,at,ab,x,c,imax,jmax,kmax)
!computes c = Ax
!i.e., r = b - matmul(a,x)
r = b - c
c1=dot_product(r,r)
if(c1 < er0) return
p = r
r0 = r
do it=1,itrmax
call probmv(ap,ae,aw,an,as,at,ab,p,y,imax,jmax,kmax)
!i.e., y = matmul(a,p)
c2 = dot_product(r0,y)
alp = c1/c2
e = r -alp*y
call probmv(ap,ae,aw,an,as,at,ab,e,v,imax,jmax,kmax)
!i.e., v = matmul(a,e)
ev = dot_product(e,v)
vv = dot_product(v,v)
c3 = ev/vv
x = x + alp*p + c3*e
r = e - c3*v
rr = dot_product(r,r)
write(*,*) it,' residual=',rr
if (rr<er0) exit
c1 = dot_product(r0,r)
bet = c1/(c2*c3)
p = r + bet*(p-c3*y)
end do
deallocate(r,r0,p,y,e,v)
end subroutine

```

```

subroutine probmv(ap,ae,aw,an,as,at,ab,b,c,imax,jmax,kmax)
!Computes the product c of a band matrix A and a vector b
!(ap,ae,aw,an,as,at,ab) are the bands of A
implicit none
integer, intent (in) :: imax,jmax,kmax
real (8), intent (in), dimension (imax,jmax,kmax) :: ap,ae,aw,an,as,at,ab
real (8), intent (in) :: b(-imax*jmax+1:imax*jmax*kmax+imax*jmax)
real (8), intent (out) :: c(-imax*jmax+1:imax*jmax*kmax+imax*jmax)
integer i,j,k,m
do k=1,kmax
do j=1,jmax
do i=1,imax
m=i+imax*(j-1)+imax*jmax*(k-1)
c(m) = ap(i,j,k)*b(m) +aw(i,j,k)*b(m-1) +ae(i,j,k)*b(m+1) &
+as(i,j,k)*b(m-imax) +an(i,j,k)*b(m+imax) &
+ab(i,j,k)*b(m-imax*jmax) +at(i,j,k)*b(m+imax*jmax)
end do
end do
end do
end subroutine

```

A.2 Translated Fortress Program

```

component BiCGSTAB2cleaned
export Executable
run(args : String ...) : () = do
  println("BiCGSTAB")
  testrun[[16,16,16]]()
end

(* Makes sample matrix A and vector b s.t. the answer of A x = b is x = 1 . *)
initalization[[nat imax,nat jmax,nat kmax]]() =
do
  Ap = array3[[R64,imax,jmax,kmax]]()
  Ae = array3[[R64,imax,jmax,kmax]]()
  Aw = array3[[R64,imax,jmax,kmax]]()
  An = array3[[R64,imax,jmax,kmax]]()
  As = array3[[R64,imax,jmax,kmax]]()
  At = array3[[R64,imax,jmax,kmax]]()
  Ab = array3[[R64,imax,jmax,kmax]]()
  Ap.fill(fn (i : Z32, j : Z32, k : Z32) : R64 => 2.0)
  Ae.fill(fn (i : Z32, j : Z32, k : Z32) : R64 => 1.0)
  Aw.fill(fn (i : Z32, j : Z32, k : Z32) : R64 => 1.0)
  An.fill(fn (i : Z32, j : Z32, k : Z32) : R64 => 1.0)
  As.fill(fn (i : Z32, j : Z32, k : Z32) : R64 => 1.0)
  At.fill(fn (i : Z32, j : Z32, k : Z32) : R64 => 1.0)
  Ab.fill(fn (i : Z32, j : Z32, k : Z32) : R64 => 1.0)

  for j ← 0 # jmax do
    for k ← 0 # kmax do
      Aw[0,j,k] := 0.0
      Ae[imax-1,j,k] := 0.0
    end
  end

  for i ← 0 # imax do
    for k ← 0 # kmax do
      As[i,0,k] := 0.0
      An[i,jmax-1,k] := 0.0
    end
  end

  for i ← 0 # imax do
    for j ← 0 # jmax do
      Ab[i,j,0] := 0.0
      At[i,j,kmax-1] := 0.0
    end
  end

  b = array1[[R64,(imax jmax kmax) + (2 jmax imax)]]()
  x = array1[[R64,(imax jmax kmax) + (2 jmax imax)]]()

  x.fill(fn (i : Z32) : R64 => 0.0)
  b.fill(fn (i : Z32) : R64 => 0.0)

  for i ← 0 # imax do

```

```

for j ← 0 # jmax do
  for k ← 0 # kmax do
    m = i + imax j + imax jmax k + imax jmax
    bm := 2.0
    if (i > 0) ∧ (i < imax - 1) then
      bm := bm + 2.0
    else
      bm := bm + 1.0
    end
    if (j > 0) ∧ (j < jmax - 1) then
      bm := bm + 2.0
    else
      bm := bm + 1.0
    end
    if (k > 0) ∧ (k < kmax - 1) then
      bm := bm + 2.0
    else
      bm := bm + 1.0
    end
  end
end
end
(* the answer of Ax=b is x = 1. *)
(Ap, Ae, Aw, An, As, At, Ab, b, x)
end
(* A sample driver for the BiCGSTAB method. *)
testrun[nat imax, nat jmax, nat kmax]() : () =
do
  (Ap, Ae, Aw, An, As, At, Ab, b, x) = initialization[imax, jmax, kmax]()
  st = nanoTime()
  BiCGSTABband[imax, jmax, kmax](Ap, Ae, Aw, An, As, At, Ab, x, b)
  et = nanoTime()
  t = et - st
  ss = t/1000000000.0
  println("elapsed time " t " ns ( " ss " s)")
end
(* BiCGSTAB method for band matrix A and vector b. *)
BiCGSTABband[nat imax, nat jmax, nat kmax](Ap, Ae, Aw, An, As, At, Ab, x2, b2) : (Z32, R64) =
do
  x : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := x2
  b : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b2
  r : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b.copy()
  r' : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b.copy()
  p : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b.copy()
  y : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b.copy()
  e : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b.copy()
  v : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b.copy()
  c : Vector[R64, (imax jmax kmax) + (2 jmax imax)] := b.copy()
  (* we dont need so many iterations because the matrix is a band matrix. *)
  itrmax = imax jmax kmax
  er0 = 1.0 10-4
  MatrixVectorProduct[imax, jmax, kmax](Ap, Ae, Aw, An, As, At, Ab, x, c)
  r := b - c
  c1 : R64 := r · r
label MainLoop
  if (c1 < er0) then exit MainLoop with () end
  p := r
  r' := r
  it : Z32 := 1
  rr : R64 := 0
  while it ≤ itrmax do
    MatrixVectorProduct[imax, jmax, kmax](Ap, Ae, Aw, An, As, At, Ab, p, y)
    c2 = r' · y
    alp = c1/c2
    e := r - alp y
    MatrixVectorProduct[imax, jmax, kmax](Ap, Ae, Aw, An, As, At, Ab, e, v)
    ev = e · v
    vv = v · v
    c3 = if (vv = 0) ∧ (ev = 0) then 1 else ev/vv end
    x := x + alp p + c3 e
    r := e - c3 v
    rr := r · r
    println(it " residual = " rr)
  end

```

```

        if (rr < ero) then exit MainLoop with (it, rr) end
        c1 := r' . r

        bet = c1/(c2 c3)
        p := r + bet(p - c3 y)

        it += 1
    end
    (it, rr)
end MainLoop
end

(* A product of a 7-band matrix and a vector. *)
MatrixVectorProduct[[nat imax, nat jmax, nat kmax]]
(Ap, Ae, Aw, An, As, At, Ab, b, c) : () =
do
  for k ← 0 # kmax do
    for j ← 0 # jmax do
      for i ← 0 # imax do
        m = i + imax j + imax jmax k + imax jmax
        c_m := Ap[i, j, k] b_m + Aw[i, j, k] b_{m-1} + Ae[i, j, k] b_{m+1} + As[i, j, k] b_{m-imax} + An[i, j, k] b_{m+imax}
              + Ab[i, j, k] b_{m-imaxjmax} + At[i, j, k] b_{m+imaxjmax}
      end
    end
  end
end
end
end
end

```

B Simplification of Loops in the Original Fortran Program

The provided Fortran program (Figure B.1) is modified to simplify the split loops in the routine probmv. The modification consists of the following three points.

- Extension of index spaces of vectors.
- Unification of split loops.
- Elimination of intermediate data structures.

The provided program splits loops in the routine for the matrix-vector product. This is because a unified loop may access elements outside the index space. Therefore, to avoid such access violations and overheads introduced by conditional branches, the unified loop is split into several loops in the provided program.

The split loops, however, make the translation hard and disturb a compiler's automatic parallelization. Thus, we unified the loop by extending the index spaces with zeros, and started the translation from the modified program (shown in Appendix A). Also, we removed unnecessary intermediate data structures used to translate indices in the provided program.

As a result of the simplification, the modified program achieved two times faster sequential execution than the provided program. Moreover, the Intel Fortran compiler succeeded in its automatic parallelization of the modified program so that the parallelized program achieved the same speedup as the hand-parallelized program with OpenMP.

B.1 The Original Fortran Program with Split Loops

```

subroutine probmv(ap, ae, aw, an, as, at, ab, b, c, imax, jmax, kmax)
!Computes the product c of a band matrix A and a vector b
!(ap, ae, aw, an, as, at, ab) are the bands of A
implicit none
integer, intent(in) :: imax, jmax, kmax
real(8), intent(in), dimension(imax, jmax, kmax) :: ap, ae, aw, an, as, at, ab
real(8), intent(in) :: b(imax*jmax*kmax)
real(8), intent(out) :: c(imax*jmax*kmax)
integer i, j, k, imax1, jmax1, kmax1, m
real(8), allocatable :: phi(:, :, :), x(:, :, :)
allocate(phi(imax, jmax, kmax), x(imax, jmax, kmax))
imax1 = imax-1
jmax1 = jmax-1

```

```

kmax1 = kmax-1
do k=1,kmax
do j=1,jmax
do i=1,imax
m=i+imax*(j-1)+imax*jmax*(k-1)
phi(i,j,k) = b(m)
end do
end do
end do
!translating the linear index to 3D index: b(m) -> phi(i,j,k)
!The main loop is splitted into several blocks to avoid use of IF statements

k=1
!j+-----
j=1
!i+-----
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

!i-----

do j=2,jmax1
!i+-----
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +as(i,j,k)*phi(i,j-1,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

!i-----

end do

j=jmax
!i+-----
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +as(i,j,k)*phi(i,j-1,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+at(i,j,k)*phi(i,j,k+1)

!i-----
!j-----

do k=2,kmax1
!j+-----
j=1

```

```

!i+++++
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

!i-----

do j=2,jmax1
!i+++++
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

!i-----

end do

j=jmax
!i+++++
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+as(i,j,k)*phi(i,j-1,k) &
+ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

!i-----
!j-----

end do

k=kmax
!j+++++
j=1
!i+++++
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1)

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1)

!i-----

do j=2,jmax1

```

```

!i+++++
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1) +ae(i,j,k)*phi(i+1,j,k) &

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
+ab(i,j,k)*phi(i,j,k-1)

!i-----
end do

j=jmax
!i+++++
i=1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +as(i,j,k)*phi(i,j-1,k) +ae(i,j,k)*phi(i+1,j,k) &
+ab(i,j,k)*phi(i,j,k-1) &

do i=2,imax1
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
+as(i,j,k)*phi(i,j-1,k) &
+ab(i,j,k)*phi(i,j,k-1)

end do
i=imax
x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) &
+as(i,j,k)*phi(i,j-1,k) &
+ab(i,j,k)*phi(i,j,k-1)

!i-----
!j-----

!general form
!x(i,j,k) = ap(i,j,k)*phi(i,j,k) +aw(i,j,k)*phi(i-1,j,k) +ae(i,j,k)*phi(i+1,j,k) &
! +as(i,j,k)*phi(i,j-1,k) +an(i,j,k)*phi(i,j+1,k) &
! +ab(i,j,k)*phi(i,j,k-1) +at(i,j,k)*phi(i,j,k+1)

do k=1,kmax
do j=1,jmax
do i=1,imax
m=i+imax*(j-1)+imax*jmax*(k-1)
c(m) = x(i,j,k)
end do
end do
end do
deallocate(phi)
end subroutine

```