ADAPTOR 6.1 Release Notes

Thomas Brandes, Resi Höver-Klier
Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
Tel.: +49-2241-142492, Fax: +49-2241-142181
e-mail: brandes@gmd.de

Abstract

ADAPTOR (Automatic DAta Parallelism TranslatOR) is a public domain High Performance Fortran (HPF) compilation system developed at the SCAI institute (GMD) during the last years. The tool transforms data parallel programs written in Fortran with array extensions, parallel loops, and layout directives into programs with explicit message passing.

These release notes describe the latest achievements incorporated within the new ADAPTOR version 6.1.

1 What is new in ADAPTOR 6.1

- ADAPTOR 6.1 does not include the support of any new language features but the existing ones are now more general, especially for HPF programs that use derived data types and pointers.
- The support of task parallelism has improved and its use is safer than in the previous version.
- ADAPTOR now provides automatic loop parallelization and identifies automatically scalar `NEW` and `RESIDENT` variables (see section 4).
- ADAPTOR now includes more appropriate, helpful error and warning messages to make it easier for users to debug programs and to tune them.
- ADAPTOR is also available for the NEC Cenju-4, a version of the DALIB is now also available for EPX (communication API of Embedded Parix).

With the new version of ADAPTOR only a new users guide is provided [BHK98b]. The installation guide [BHK98a] and the programmers guide [Bra98a] have not been updated.

2 New Supported Language Features of HPF

The latest version ADAPTOR 6.1 does not support any more language features than the previous one.

2.1 Derived Data Types

It is now rather safe to use distribution directives for components of derived data types. Also components with the `POINTER` attribute can have mapping directives, but should be used like arrays with the `ALLOCATABLE` attribute.

```
      integer, parameter :: N = 10
      type GRID
         real, dimension (N,N) :: A
         real, dimension (:,:), pointer :: B
!hpf$    distribute (*,block)  :: A, B
      end type

      type (GRID), dimension(20)  :: SEQUENCE
```

More critical is the mapping of arrays of derived data types when the components contain arrays:

```
      type GRID
         real, dimension (N,N) :: A
         real, dimension (:,:), pointer :: B
      end type

      type (GRID), dimension(20)  :: SEQUENCE
!hpf$ distribute (block) :: SEQUENCE           ! attention
```

The whole ADAPTOR translation is based of the idea that every processor contains a descriptor of every array and knows about the sizes and the mapping of these arrays. If now the array `SEQUENCE` is mapped, the array descriptors for the components `A` and `B` will only exist on one processor. Not every processor has an array descriptor of all existing arrays. So the other processors will not know about the sizes of the components when they are not mapped to this processor. This might cause problems in certain situations that are not fixed completely yet.

2.2 Task Parallelism

The features for HPF task parallelism have been improved. This includes especially the functionality of the HPF task library.

It is now also possible to couple different HPF programs. Distributed arrays can be exchanged between different tasks just like serial data in usual message passing programs.

Furthermore, ADAPTOR supports irregular processor subsets. This results in more flexibility when using task parallelism.

```
!hpf$ processors P (6)
      integer, dimension (4) :: IND = (/ 1, 3, 4, 6 /)
      real, dimension (N)    :: A1, A2
!hpf$ distribute A1 (block) onto P
!hpf$ distribute A2 (block) onto P(IND)
      ...
!hpf$ on (P(IND))
         call TASK (A2, N)
```

A more detailed description of the support for HPF task parallelism can be found in [Bra98b].

2.3 The `RANGE` Directive

The `RANGE` directive itself was already avaialable in the previous version of ADAPTOR. It could be used to restrict the possible mappings of dummy arrays whose mapping was inherited.

```
      subroutine sub (A, N)
      integer, intent (in)  :: N
      real, dimension (N,N) :: A
!hpf$ inherit A
!hpf$ range (block(),block()) :: A
```

The `RANGE` directive can now also be used to specify the possible mappings of arrays that have the `DYNAMIC` attribute. This becomes very important for global arrays in modules. Subprograms using this module have now more information about possible mappings of the arrays within this module.

```
      module DATA
      integer, parameter  :: N = 100
      real, dimension (N,N) :: A
!hpf$ dynamic :: A
!hpf$ range (block(),block()) :: A
!hpf$ distribute (*,block) :: A      ! initial distribution
      ...
```

Every subprogram using the module `DATA` has now more information about possible mappings of the array `A`.

2.4 The NODESCRIPTOR Directive

ADAPTOR will create for every array an array descriptor at runtime. This includes also small fixed sized arrays. In the following example, there will be an array descriptor for A and B for every component of `SEQUENCE`.

```
      type GRID
         real, dimension (3,3) :: A, B
         integer :: N1, N2
      end type

      type (GRID), dimension(20)  :: SEQUENCE
```

In other words, there will be 20 array descriptors for A and 20 for B at runtime. ADAPTOR now provides the `NODESCRIPTOR` directive to avoid this overhead.

```
      type GRID
         real, dimension (3,3) :: A, B
!adp$    nodescriptor :: A, B
         integer :: N1, N2
      end type

      type (GRID), dimension(20)  :: SEQUENCE
```

In the following situations, the array descriptor must exist and the `NODESCRIPTOR` directive would be invalid:

- Every mapped array and every array with the `DYNAMIC` attribute of HPF must have a descriptor at runtime.
- Every dynamic array (allocatable array or automatic array) must have a descriptor.
- Arrays with the `POINTER`, `TRACE`, `SHARED` or `SHADOW` attribute must have a descriptor.
- If a section of an array is passed to a subprogram, the array must have a descriptor.
- An array used in certain array operations (e.g. `CSHIFT`) must have a descriptor.

The absence of an array descriptor can also improve the performance when calling a pure routine working on small arrays. It avoids the creation of descriptors for every call and the matching of actual and dummy descriptors.

```
      real, dimension F1(6,N)
      real, dimension X1(3,N)
!hpf$ distribute F1 (*,block)
```

```
!hpf$ align X1 (J,I) with F1 (*,I)
      real F(6), X(3)
      ...
!hpf$ independent, new (F, X)
      do I = 1, N
         F = F1(:,I)
         call CALC (X, F)
         X1(:,I) = X
      end do
      ...
      pure subroutine CALC (X, F)
      real X(3), F(6)
!hpf$ nodescriptor :: X, F
      ...
      end subroutine CALC
```

The `NODESCRIPTOR` directive should be used very carefully. This feature has not been tested very carefully yet.

3 Support of Shared Memory Architectures

Within ADAPTOR, we want also to support shared memory architectures, e.g. SUN Enterprise, SGI Origin, Convex, NEC SX/4 or NEC SX/5. Unfortunately, this support is rather restricted.

- Shared arrays between SPMD processes use System V shared memory segments that are usually available on these architectures. This feature is no more supported in the public domain version.
- ADAPTOR can compile HPF programs to Fortran programs containing shared memory parallelization directives by `-1 -mp=xxx`. This feature causes a lot of problems and is not well tested.
- There is a DALIB version that uses shared memory segments for message passing `target_communication=SHM`. This version should work, but can have problems with large messages. Therefore we recommend to use the DALIB version for MPI and to use an MPI implementation that is based on shared memory and therefore more efficient.

In summary, we recommend to use the HPF programming model on these shared memory architectures just like on distributed memory architectures. We hope to have better and more stable support in future versions.

4 Automatic Loop Parallelization

ADAPTOR 6.1 has now capabilities for the automatic detection of `NEW` and `REDUCTION` variables in `DO` loops and the automatic identification of `INDEPENDENT DO` loops. The automatic loop parallelization is switched on by setting the following flag for the source-to-source translation fadapt:

```
-auto
```

Note: Instead of using automatic loop parallelization features, it is always possible to use the corresponding HPF directives directly in the source code.

4.1 `NEW` Variables

In a first step, ADAPTOR identifies `NEW` variables. Every scalar variable in a `DO` loop that is defined in every iteration before it is used becomes a `NEW` variable for the corresponding `DO` loop. If ADAPTOR can verify that the scalar variable does not live anymore at the end of the `DO` loop, it will print an info message. If ADAPTOR cannot verify that the last value of the scalar variable if no more needed, it assumes that the last value is no more used and prints a warning message.

```
do I = 1, N
   X = 2.0 * float(I)
   A(I) = X
end do
X = 1.0
```

```
      do I = 1, N
!hpf$ new (X), begin
        X = 2.0 * float(I)
        A(I) = X
!hpf$ end new
      end do
      X = 1.0
```

```
INFO: X becomes a NEW variable
```

ADAPTOR 6.1 is not able to make a full data flow analysis for live and dead variables in the presence of jump statements. Therefore it is not always possible to identify that a scalar variable does not live anymore at the end of a loop. In this case, ADAPTOR makes it also to a `NEW` variable and prints a warning message.

```
do I = 1, N
   if (A(I) .gt. 0) then
      X = A(I)
     else
      X = -A(I)
   end if
   A(I) = 2.0 * X
end do
goto 15
```

```
      do I = 1, N
!hpf$ new (X)
         if (A(I) .gt. 0) then
            X = A(I)
           else
            X = -A(I)
         end if
         A(I) = 2.0 * X
!hpf$ end new
      end do
```

```
                                     goto 15
```

```
WARNING: X becomes a NEW variable (no last val assumed)
```

If the value of the variable `X` is used after the `DO` loop (`X` lives at the end of the loop), the code might produce wrong results as the processors have not consistent values.

Beside the detection of `NEW` variables for `DO` loops, ADAPTOR also identifies scalar `NEW` variables within `ON` constructs.

```
!hpf$ on home (A(1)), begin       !hpf$ on home(A(1)), new(X), begin
      X = A(1) * B(1)                   X = A(1) * B(1)
      C(1) = 2.0 * X                    C(1) = 2.0 * X
!hpf$ end on                      !hpf$ end on
```

ADAPTOR makes a scalar variable only to a `NEW` variable if it is sure that the variable does not live anymore at the end of the `ON` construct. By this way, ADAPTOR knows later that it is not necessary to broadcast the value of `X` from the executing processor (or processor subset) to all the other processors.

4.2 `REDUCTION` Variables

In a next step, ADAPTOR identifies scalar `REDUCTION` variables in `DO` loops.

```
                                   !hpf$ reduction(X)
      do I = 1, N                        do I = 1, N
         X = X + A(I) * B(I)                X = X + A(I) * B(I)
      end do                             end do
```

```
INFO: X becomes a REDUCTION variable
```

ADAPTOR uses the same algorithm for the identification of `REDUCTION` variables that is used to verify the correct use of such variables appearing in a `REDUCTION` directive. `MIN` and `MAX` reductions programmed with `IF` statements will also be identified.

```
                                   !hpf$ reduction(X)   ! maxval
      do I = 1, N                        do I = 1, N
         if (A(I) .gt. X) then              if (A(I) .gt. X) then
            X = A(I)                           X = A(I)
         end if                             end if
      end do                             end do
```

```
INFO: X becomes a REDUCTION variable
```

, **Note**: ADAPTOR does not identify array variables as reduction variables though array variables can also be used in the `REDUCTION` directive. If it did, it could happen that there would be a reduction over a whole array instead of a single element. In the following example, it would not be useful to have a reduction over the whole array `X` in the inner loop as only a single element is used for the reduction. Unfortunately, the variable `X` is not a reduction variable for the outer loop as it is initialized within the loop. Only after loop distribution, that is not supported and done by ADAPTOR, a reduction over the whole array `X` would be useful.

```
      do J = 1, N
         X(J) = 0.0
         do I = 1, N
            X(J) = X(J) + A(I,J) * B(I,J)
         end do
      end do
```

4.3 Independent DO Loops

ADAPTOR identifies `DO` loops as `INDEPENDENT`, if it can prove that there are no loop carried dependences for the iterations of the loop. It uses a kind of Banerjee's equation to verify the absence of loop carried dependences.

```
      do J = K+1, N
         do I = K+1, N
            A(I,J) = A(I,J) - A(I,K) * A(K,J)
         end do
      end do

! becomes

!hpf$ independent
      do J = K+1, N
!hpf$ independent
         do I = K+1, N
            A(I,J) = A(I,J) - A(I,K) * A(K,J)
         end do
      end do
```

For the dependence analysis it is important that `NEW` and `REDUCTION` variables have already been identified before as these kind of variables cannot cause any dependences within the loop.

4.4 Induction Variables

Currently, ADAPTOR provides no support for recognizing induction variables in `DO` loops. Therefore, the following loop will not be parallelized automatically.

```
      K = 0
      do I = 1, N
```

```
      K = K + 1
      A(I) = B(K) + C(I)
   end do
```

The integer variable `K` is neither a `NEW` variable nor a `REDUCTION` variable. The `DO` loop will not be classified as an `INDEPENDENT` loop. The user has to rdo eplace the induction `K = K + 1` with a corresponding assignment `K = I` to get an `INDEPENDENT` loop.

```
   do I = 1, N
      K = I
      A(I) = B(K) + C(I)
   end do
```

5 New Compiler Flags

The new version ADAPTOR 6.1 provides some more flags to drive the compilation of HPF programs.

5.1 Compiler Flags in the Source Code

In certain situations, it is very convenient to specify compilation flags directly in the HPF source code. It will allow the same compile command in a `makefile` though the different source files should be compiled with different flags. It also can make sure that a certain source file is really compiled with the appropriate flags.

The compilation flags can be set in the source code by using the following command line directive:

```
!ADP$ FLAGS -auto -free -static -safety 0
      subroutine sub (...)
      ...
      end subroutine
```

Note: The `FLAGS` directive should only be used once in a source code as further directives will overwrite the previous flags. It is also not possible to use different flags for different subroutines within one source code.

5.2 Static Arrays

Many Fortran compilers allow fixed sized arrays to be considered as static arrays. They will allocate local variables to a static storage area. Uninitialized local elements are cleared to zero. Also common arrays will get a static storage area.

```
program P
parameter (N=100)
common /DATA/ U(N,N), V(N,N)
...
end

subroutine sub
parameter (N=100)
real A(N,N)
...
end
```

In this example code, the `COMMON` arrays `U` and `V` and the local array `A` can be considered as static arrays.

Fixed sized arrays remain fixed sized arrays during the source-to-source translation of ADAPTOR as long as these arrays are not mapped and have not the `DYNAMIC` attribute. But if they are mapped by the HPF mapping directives, ADAPTOR transforms them to allocatable arrays as the size on one processor will be known only at runtime. In the final code generation, allocatable arrays become pseudo-dynamic arrays (out of range addressing of dynamically allocated memory with the `malloc` command of C) and can be no more static.

ADAPTOR provides now a compiler flag to let the fixed sized arrays unchanged.

```
-static
```

Attention: Fixed sized arrays will use the full memory on every processor even if the arrays are mapped.

5.3 Warning and Info Messages

Due to the demand of many users that want more feedback of the source-to-source translation, ADAPTOR prints now more information and warning messages. This includes especially infos about the automatic parallelization and about the use of shadow areas for mapped arrays. Warnings will inform the user where the translation was not done as probably intended by the user (e.g. redistributions) or where ADAPTOR assumes certain properties that are very likely but might result in wrong code if they are not given (PURE subroutine calls in `INDEPENDENT` loops, no last value for scalar variables in `DO` loops, etc.).

These kind of information and warning messages can be suppressed with the following compiler flags:

```
-w             ! suppresses warning messages
-noinfo        ! suppresses information messages
```

6 Fixed Bugs

Though ADAPTOR is already a rather stable compilation system, some bugs and inconveniences of the last version have now been fixed in ADAPTOR 6.1.

6.1 Alignment of Pseudo-Dynamic Arrays

Dynamic arrays and mapped fixed sized arrays become pseudo-dynamic arrays during the final code generation of the source-to-source translation of ADAPTOR.

```
      integer, parameter :: N = 100
      real A(N,N)                      ! becomes dynamic array
!hpf$ distribute A(block,block)        ! due to the distribute directive
      ...
      A(I,J) = 1.0

      real, dimension (:,:), allocatable :: A
      ...
      call dalib_local_sizes (A_DSP, MY_LB1, MY_UB1, MY_LB2, MY_UB2)
      allocate (A(MY_LB1:MY_UB1,MY_LB2:MY_UB2))
```

ADAPTOR translates allocatable and automatic arrays to pseudo-dynamic arrays that allow the compilation of the generated code by a FORTRAN 77 compiler. A small one-dimensional array is used to access dynamically allocated memory on the heap via out-of-range addressing. The multi-dimensional indexes are linearized.

```
      real A(1:2)                          ! fixed size array for addressing
      integer :: A_ZERO, A_DIM1, A_DIM2
      ...
      call dalib_allocate (A_DSP, A)       ! allocates A on the heap
                                           ! via the malloc routine
      ...
      A(A_ZERO+A_DIM1*J+I) = 1.0           ! out of boundary addressing
```

The value of `A_ZERO` stays for the difference between the addesses of the static Fortran array `A` and the memory allocated on the heap divided by the size for one element. In other words, the difference between the two addresses must be a multiple of the size for one element. While this is usually the case for data types where one element has 4 bytes (e.g. `INTEGER`, `REAL`), problems arise for 8 bytes data types (e.g. `DOUBLE PRECISION`). If the dynamic data is allocated at an address which is a multiple of 8, but the static array has an address that is not a multiple of 8, the following error message appeared in previous versions:

```
array_access: static addr = -1073743036, dyn addr = 134750504
              diff = 1208493540 is not multiple of size = 8
```

```
SERIOUS ERROR in DALIB (Distributed Array LIBrary)
==================================================

UNIT  : CALC_PI
ERROR : alignment problem, use flag -f for compilation

stop execution
   ...
```

Certain Fortran compilers provided flags to force the alignment of static arrays at 8-byte boundaries, but not all of them (especially the GNU Fortran compiler on LINUX systems). Also arrays of derived data types caused these runtime errors.

The new ADAPTOR version 6.1 has no more this problem. An internal offset is used to make the difference of the addresses a multiple of the element size. The penalty is the allocation of some more bytes and some more data in the internal array descriptors.

6.2 Loop Fusion

Loop fusion is an optimization heavily used by ADAPTOR. Unfortunately, the previous versions had the following problems:

```
J = NY
do K = 1, NZ
   X(J,K) = TMP(J,K,3)
end do
do K = 1, NZ
   do J = NY-1,1,-1
      X(J,K) = TMP(J,K,2)*X(J+1,K)
   end do
end do
```

ADAPTOR fused the two `K` loops which was not correct due to the dependences of the variable `J`.

```
J = NY
do K = 1, NZ
   X(J,K) = TMP(J,K,3)
   do J = NY-1,1,-1
      X(J,K) = TMP(J,K,2)*X(J+1,K)
   end do
end do
```

Another problem was due to the fact that ADAPTOR did not rename correctly the indices of loops. The following two `J` loops were fused incorrectly.

```
do J = 1, N
   B(I) = 1.0
end do
do I = 1, N
   do J = 1, N
      A(I,J) = 1.0
   end do
end do
```

```
do J = 1, N
   B(I) = 1.0
   do J = 1, N
      A(J,J) = 1.0
   end do
end do
```

Both problems have been fixed and ADAPTOR is now hopefully safe for all kinds of loop fusion.

6.3 Realignment

The `REALIGN` directive caused serious runtime errors in the following situation:

```
      integer, parameter :: N = 10
      real A(N,N,N), B(N,N,N)
!hpf$ dynamic :: A, B
!hpf$ distribute B(*,*,block)
!hpf$ align A with B
      ...
!hpf$ redistribute B(*,block,*)
!hpf$ realign A with B
      ...
```

The redistribution of B destroyed the old array descriptor of B to which the array A was still aligned. The realignment of A resulted in runtime errors.

The runtime system now counts references to array descriptors of arrays or templates that might be no more existent and destroys them only if there is no more any reference to it.

7 Tuning of HPF Programs

This section summarizes some important issues that should be considered to get good performance out of HPF programs with ADAPTOR. These issues have been identified when looking at typical HPF programs of our users that have been translated with ADAPTOR and where the performance was poor.

7.1 Overhead of HPF Programs

Many users complain about the overhead introduced by the HPF directives and by the necessary modifications of the original program. The compiled HPF programs run often on a single node much slower than the original Fortran program.

This overhead is of course very crucial for the acceptance of HPF. Within the ADAPTOR project, we paid much attention to this problem and analyzed this overhead very carefully. Some of the typical overheads are described in the following sections.

At this point it should be noted that ADAPTOR also allows the compilation of an HPF program for a single node (**gmdhpf -1 <program.hpf>**). This version of the program contains no communication at all. It can be used to verify the overhead that comes from array operations and other code modifications. Any overhead here will also appear when the program is compiled to an SPMD program. Tuning of the program at this level is simpler and eliminates already a lot of performance problems.

7.2 Loop Ordering

Most Fortran programmers know that nearly on all machines it is better to run the innermost loop for the first array index.

```
       real, dimension (N,N) :: A, B
       integer I, J
       do J = 1, N
          do I = 1, N
             A(I,J) = A(I,J) + B(I,J)
          end do
       end do
```

ADAPTOR takes care of this for array operations.

```
       A = A + B

!      will be translated to

       do J = 1, N
          do I = 1, N
             A(I,J) = A(I,J) + B(I,J)
          end do
       end do
```

But ADAPTOR does not change the order of nested loops in `FORALL` statements. So the following loop results in poor performance:

```
       forall (I=1:N, J=1:N) A(I,J) = A(I,J) + B(I,J)

!      will be translated to

       do I = 1, N
          do J = 1, N
             A(I,J) = A(I,J) + B(I,J)
          end do
       end do
```

ADAPTOR does not any loop interchanging, so users should be very careful about the loop ordering. Performance problems of this kind can already be identified when compiling HPF programs for a single node.

7.3 SHADOW Areas

ADAPTOR creates automatically shadow areas for distributed arrays if it is appropriate. Especially for stencil operations, shadow areas increase the performance of the program dramatically.

```
      real, dimension (N,N) :: A, B
!hpf$ distribute (block,block) :: A, B
      ...
!hpf$ independent
      do I = 2, N-1
!hpf$ independent
         do J = 2, N-1
            A(I,J) = f(B(I,J),B(I,J-1),B(I,J+1),B(I-1,J),B(I+1,J))
         end do
      end do
```

So ADAPTOR will insert the following directive automatically:

```
!hpf$ shadow B(1:1,1:1)
```

Unfortunately, there are still some situations that cause performance problems.

- If the array B is an use associated array and the corresponding module is not in the same compilation unit (separate compilation), ADAPTOR cannot insert a shadow area. A corresponding warning message should be considered and the `SHADOW` directive should be inserted by hand directly in the source file of the module.

- If the array B is a dummy array, ADAPTOR inserts the shadow. But if the actual argument has not enough shadow, a copy-in and copy-out operation will be necessary that decreases the performance. ADAPTOR propagates the `SHADOW` directives from dummy arguments to actual argument, but can do this only within one compilation unit.

ADAPTOR now gives warnings in corresponding situations and the user should act on them to get good performance.

7.4 Serial Loops

Any serial loop over a distributed array gives very poor code performance. The innermost statements will be executed by all processors that can imply a broadcast of single elements that are used in the statements.

```
      real, dimension (N,N) :: A, B
!hpf$ distribute (block,block) :: A, B
      ...
      do I = 2, N-1
         do J = 2, N-1
            A(I,J) = f(B(I,J),B(I,J-1),B(I,J+1),B(I-1,J),B(I+1,J))
         end do
      end do
```

ADAPTOR would take this code and generate the following (pseudo-) SPMD program:

```
      do I = 2, N-1
         do J = 2, N-1
            X1 = broadcast (B(I,J-1))
            X2 = broadcast (B(I,J+1))
            X3 = braodcast (B(I-1,J))
            X4 = broadcast (B(I+1,J))
            if (is_local (A (I,J)) then
               A(I,J) = f(B(I,J),X1,X2,X3,X4)
            end if
         end do
      end do
```

The best way to increase the performance is to insert the `INDEPENDENT` directive or to switch on the automatic parallelization. But nevertheless, there are some situations where ADAPTOR will serialize the loop and produce programs with a very low performance:

- ADAPTOR cannot always identify `INDEPENDENT` loops automatically. Here the user has to insert this property by inserting the corresponding directive.
- If a variable is used like a `NEW` or `REDUCTION` variable but this is not identified automatically or the corresponding directive is not given, ADAPTOR will serialize the loop. In the following example, the loop is serialized to make sure that X contains the correct value at the end of the loop.

```
!hpf$ independent
      do I = 1, N
         if (A(I) .gt. 0.0) then
            X = A(I)
```

```
            A(I) = X * (X - 1.0)
         end if
      end do
```

In such situations, the use of the `NEW` directive will increase the performance dramatically. If the last value of X is really important, the program should use the `REDUCTION` directive:

```
      X = 0.0
!hpf$ independent, new(Y), reduction(X)
      do I = 1, N
         Y = A(I)
         if (Y .gt. 0.0) then
            X = max (X, Y)
            A(I) = Y * (Y - 1.0)
         end if
      end do
```

- ADAPTOR is not always able to extract communication out of parallel loops. A typical example is the following loop:

```
!hpf$ distribute (block) :: A

!hpf$ independent, new(K)
      do I = 2, N
         K = I - 1
         A(I) = A(I) + A(K)
      end do
```

  In such situations, it is very helpful for ADAPTOR to replace the index `K` directly with the expression `I-1` (forward substitution). One of the next versions of ADAPTOR will do this automatically.

- If an independent loop is serialized, ADAPTOR gives a warning message. The user should analyse the intermediate files (use the flag `-G`) very carefully to identify the problem. At first, it should be verified that ADAPTOR has chosen a good home for the loop iterations. If not, the `HOME` directive should be used. Then it should be verified whether ADAPTOR could really extract all necessary communication. If this was not possible, the user can do this very often by himself. He has to create a temporary array aligned to the loop iterations and to define it with the non-local values.

References

[BHK98a] T. Brandes and R. Höver-Klier. ADAPTOR Installation Guide (Version 6.0). Technical documentation, GMD, June 1998. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/iguide.ps`.

[BHK98b] T. Brandes and R. Höver-Klier. ADAPTOR User's Guide (Version 6.1). Technical documentation, GMD, December 1998. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/uguide.ps`.

[Bra98a] T. Brandes. ADAPTOR Programmer's Guide (Version 6.0). Technical documentation, GMD, June 1998. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/pguide.ps`.

[Bra98b] T. Brandes. Implementation and Evaluation of Nested Task and Data Parallelism for High Performance Fortran within the ADAPTOR Compilation System. Working paper (unpublished), GMD, 1998.