Scalable Modeling System (SMS) Reference Manual

National Oceanic and Atmospheric Administration
Forecast Systems Laboratory
Advanced Computing Branch
325 Broadway
Boulder, Colorado 80303

Mark W. Govett
Leslie Hart
Tom Henderson
Dan Schaffer

February 2000
SMS Version 2.0

Table of Contents

Introduction

This document describes all of the Scalable Modeling System (SMS) directives required to translate regular grid finite difference model and spectral model Fortran codes into parallel versions that can be run on a variety of shared and distributed memory machines. A component of SMS, called the Parallel Pre-Processor (PPP), is used to translate the SMS directives and serial Fortran code into parallel Fortran source. Further information about SMS is described in the overview document: ***SMS: A Directive-Based Parallelization Tool for Shared and Distributed Memory High Performance***. In addition, the ***SMS Users Guide***, details information on how to use SMS to parallelize Fortran 77 codes. It is highly recommended that these documents be read before this ***SMS Reference Manual*** is used.

In this reference manual, each directive is described in detail with sections defining the syntax, describing the directive, its limitations, notes, error messages, and code examples. The syntax section provides a description of each parameter and the permitted grammar types. These types are defined in the next section titled: Documentation Notes.

In order to clarify code translations, generated output code is occasionally presented. This output is NOT the actual code generated by PPP; it is pseudo-code and should be used for informational purposes only.

Documentation Notes

I. Grammar Types

1. (expr) an expression: any combination of integers, or variable names (including array references) and the binary operators: */+-
   Examples: { 1, a, a+1, a(1)+1 }

2. (expr-) an in item 1 EXCEPT array references are not permitted (eg. a(1))
   Examples: { 1, a, a+1 }

3. (expr+) in addition to item 1, phrases of the form LowerBound : UpperBound are permitted, where both bounds can be expressions
   Examples: { 1, a+1, 1:nx, a(1):b(1) }

4. (int) any numeric integer value
   Examples: {1, 2, 3 }

5. (key) a specific keyword must be given
   Examples: { Treat_As_Complex }

6. (string) any quoted text
   Examples: { “this is a string” }

7. (var) any variable name
   Examples: { nx, ny, a, b }

8. (var|int) a variable or integer as defined in items 4 and 7 are permitted

II. Syntax Notes

1. PPP directives longer than 72 characters must be continued on the next line using the PPP line continuation characters: **CSMS$>**

Example:

```
CSMS$CREATE_DECOMP( decomp,
CSMS$>    <global_size1, global_size2, global_size3>,
CSMS$>    <halo_1, halo_2, halo_3>
```

CSMS$CREATE_DECOMP

SYNTAX

CSMS$CREATE_DECOMP (dh[(nest)],
< global1 [,global2 [,global3]]>,
< halo1 [,halo2 [,halo3]]>
[,< lb_strat1 [,lb_strat2 [,lb_strat3]]>])

Required Fields:

decomp	(var)	name of data decomposition
global1	(expr)	global size of the 1st decomposed dimension
halo1	(expr)	thickness of the halo region for the 1st decomposed dimension

Optional Fields:

nest	(expr-)	decomposition nest index
global2	(expr)	global size of the 2nd decomposed dimension
global3	(expr)	global size of the 3rd decomposed dimension
halo2	(expr)	thickness of the halo region for the 2nd decomposed dimension
halo3	(expr)	thickness of the halo region for the 3rd decomposed dimension
lb_strat1	(key)	load balancing strategy for the 1st decomposed dimension
lb_strat2	(key)	load balancing strategy for the 2nd decomposed dimension
lb_strat3	(key)	load balancing strategy for the 3rd decomposed dimension

_____ Load Balancing Strategies are:
SCRAMBLE_LAT_STRATEGY
SCRAMBLE_LON_STRATEGY
SCRAMBLE_WAVENUM_STRATEGY

SEMANTICS

This directive initializes the data structures, declared by CSMS$DECLARE_DECOMP, that are necessary to support data movement, local and global address translation, do-loop transformations, and data decomposition by SMS. These structures are initialized at run-time based on the number of processors specified at the command line (using smsRun). Once decompositions are defined and initialized,

CSMS$DISTRIBUTE is used to determine how each array is divided up among the processors.

This directive should be inserted at the beginning of the executable portion of the program. Each dimension specified by the global size arguments (*global1, 2, 3*) will be decomposed by SMS. Both static and dynamic memory allocation are supported (see Examples 1 and 2). If dynamic memory allocation is used, CSMS$CREATE_DECOMP cannot appear after a CSMS$DISTRIBUTE because the sizes of decomposition structures, determined at run-time, cannot be assigned before they are initialized (see Example 3).

Nesting is supported by SMS. If multiple nests are declared (eg. two declared nests implies CSMS$DECLARE_DECOMP(dh(2)), then a directive must be specified to initialize EACH NEST (eg. CSMS$CREATE_DECOMP(dh(1), CSMS$CREATE_DECOMP(dh(2)). Consult the SMS Users Guide for more information and examples on data decomposition using SMS.

LIMITATIONS

1. Any two of the first three dimensions can be decomposed.

2. The maximum global size and halo thickness permitted for each dimension is 1000 and 6 respectively. In a future upgrade, these values can be optionally set to other values at the PPP command line.

MESSAGES

ERRORS:

This data decomposition structure does not exist.

A data decomposition with this name has not been defined by a corresponding CSMS$DECLARE_DECOMP directive that is in scope.

This index scrambling technique is not supported.

The only types of index scrambling supported are:
scramble_wavenum_strategy
scramble_lat_strategy
scramble_lon_strategy

The number of arguments differs from the definition.

The number of dimensions specified must match the number of dimensions given for this decomposition in the corresponding CSMS$DECLARE_DECOMP directive.

The number of halo arguments differs from the number of global size values.

There must be a one to one correspondence between the global size (global1,2,3) and halo thickness (halo1,2,3) arguments.

RELATED DIRECTIVES

CSMS$DECLARE_DECOMP
CSMS$DISTRIBUTE
CSMS$TO_LOCAL

NOTES

1. Halo widths for a scrambled dimension must be zero.

EXAMPLES

Example 1: Static Memory Allocation
Example 2: Dynamic Memory Allocation
Example 3: Limitation for the Dynamic Case

Example 1: Static Memory Allocation

For a fortran program with the statically allocated array u(nx,ny,nz), a decomposition structure is created to decompose the array U over the available processors. The locally declared size of the decomposed array should be large enough to run on the minimum expected number of processors (see Example 1 of CSMS$DECLARE_DECOMP). The declared local sizes are determined at run-time based on the number of processors and the global sizes given by CSMS$CREATE_DECOMP. The example below declares local array sizes large enough to run on 16 processors (4 by 4).

```
      program static

      parameter(NX=512, NY=512, NZ=64)

CSMS$DECLARE_DECOMP(dh, <NX/4+(2*3)+1, NY/4+(2*3)+1>)
CSMS$DISTRIBUTE(dh, nx, ny) BEGIN
```

```
      real u(nx,ny,nz)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP( dh, <nx, ny>,<3, 3>)

      end
```

Example 2: Dynamic Memory Allocation

This program creates automatic arrays whose decomposed sizes depend on the number of processors specified at run-time. Note that this type of memory allocation is an extension to f77 and is not a feature of all compilers. It is supported by all f90 compilers however.

Since the subroutine arguments *nx* and *ny* are translated to the actual local sizes (CSMS$TO_LOCAL), no directives are needed inside the subroutine *model* assuming it is embarrassingly parallel.

```
      program dynamic
      integer nx,ny,nz
      namelist /domain_size/ nx,ny,nz
CSMS$DECLARE_DECOMP(dh)

      open(1,file='domain_size.nl')
      read(1,domain_size)
      close(1)

CSMS$CREATE_DECOMP( dh, <nx,ny>,<0,0> )
CSMS$TO_LOCAL(<dh : 1, nx>,<dh : 2, ny>:SIZE) BEGIN
      call model(nx,ny,nz)
CSMS$TO_LOCAL END
      end

-------------------------------------------------------------
      subroutine model(n1,n2,n3)
      integer n1,n2,n3
      real u(n1,n2,n3)
C
      do i=1,n1
        do j=1,n2
c           more code ..
        enddo
      enddo

      return
      end
```

Example 3: Limitation for the Dynamic Case

This example illustrates a requirement that a dynamically allocated data decomposition be initialized by CSMS$CREATE_DECOMP **BEFORE** it is used by the CSMS$DISTRIBUTE

directive. This is because structures initialized by CSMS$CREATE_DECOMP are used to size the decomposed array dimensions identified by CSMS$DISTRIBUTE.

```
      program simple_comp

      call compute(100)
      end
.................................................
      subroutine compute(im)
      integer im

CSMS$DECLARE_DECOMP(decomp)

CSMS$DISTRIBUTE(decomp, <im>) BEGIN
      integer x(im), y(im), z(im)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(decomp, <im>, <0>)

CSMS$PARALLEL(decomp,<i> ) BEGIN

c     executable code ...

CSMS$PARALLEL END

      return
      end
```

Re-coded Input File

CSMS$DECLARE_DECOMP and CSMS$CREATE_DECOMP are moved into the main routine to insure CSMS$DISTRIBUTE will receive the correct local size values inside *compute()*. Note: a common block is used to communicate the local array sizes between routines in code generated by CSMS$DECLARE_DECOMP. This explains why CSMS$DECLARE_DECOMP appears in both routines.

```
      program simple_comp

CSMS$DECLARE_DECOMP(decomp)
CSMS$CREATE_DECOMP(decomp, <100>, <0>)
      call compute(100)
      end
.................................................
      subroutine compute(im)
      integer im
CSMS$DECLARE_DECOMP(decomp)

CSMS$DISTRIBUTE(decomp, <im>) BEGIN
      integer x(im), y(im), z(im)
CSMS$DISTRIBUTE END

CSMS$PARALLEL(decomp,<i> ) BEGIN

c     executable code ...
```

```
CSMS$PARALLEL END
      return
      end
```

CSMS$DECLARE_DECOMP

SYNTAX

CSMS$DECLARE_DECOMP (decomp[(num)],
[,<declared1 [,declared2 [,declared3]]>]
[:<lower1, [,lower2 [,lower3]] >])

Required Fields:

decomp	(var)	name of the data decomposition

Optional Fields:

num	(expr-)	number decompositions nests
declared1	(expr+)	declared local size of the 1st decomposed dimension
declared2	(expr+)	declared local size of the 2nd decomposed dimension
declared3	(expr+)	declared local size of the 3rd decomposed dimension
lower1	(expr-)	lower bound of the 1st decomposed dimension
lower2	(expr-)	lower bound of the 2nd decomposed dimension
lower3	(expr-)	lower bound of the 3rd decomposed dimension

SEMANTICS

This directive defines decomposition structures necessary to support data movement, local and global address translation, do-loop transformations and data decomposition. These structures are then filled at run-time based on information provided by CSMS$CREATE_DECOMP and applied to decomposed arrays using the CSMS$DISTRIBUTE directive. These decomposed arrays are defined with unit lower bounds unless stated explicitly (using *lower1,2,3*) by this directive (see Example 1).

This directive must be in scope of all references to arrays, parallel regions and I/O statements that require access to the generated decomposition structures. Typically, CSMS$DECLARE_DECOMP is placed in an include file that either already exists or has been created specifically for the parallel model. These file dependencies must be indicated by all modules that need them using the ppp command line options --Finclude or --Fcommon respectively (see the SMS Users Guide: Building A Parallel Program).

This directive was designed to support both dynamic and static memory allocation. If **static allocation** is used, the user **MUST** specify local sizes (*declared1,2,3* fields above) that are large enough to handle the number of processors with which the user plans to run the program. To minimize memory use and optimize cache re-use, multiple statically allocated programs can be built with different declared size values for differing numbers of processors. At run time, the actual local sizes required on each processor will be computed. If dynamic allocation is used, declared size values **MUST NOT** be present - SMS will calculate the local sizes required automatically.

LIMITATIONS

1. Any two of the first three dimensions can be decomposed.

MESSAGES

ERRORS:

The number of arguments differs from the number of lower bounds values.

If lower bounds values (lower1,2,3) are listed, we require the number of arguments be equal to the number of declared local size (declared1,2,3) arguments given (for STATIC memory allocation only).

RELATED DIRECTIVES

CSMS$CREATE_DECOMP
CSMS$DISTRIBUTE
CSMS$PARALLEL
CSMS$LOCAL_SIZE

NOTES

1. This directive must be in scope of all directives that require access to the declared decomposition (decomp). Typically, this directive is placed in a file that is included by all modules that need it; however, it can also be inserted directly into existing source. See Example 1.

EXAMPLES

Example 1: Non-Unit Lower Bounds Support

Static memory allocation is used in this example because the declared local sizes (*NX_LOCAL*, *NY_LOCAL*) are given in CSMS$DECLARE_DECOMP. Further, the original arrays are defined with lower bounds of zero. We include the non-unit lower bound values in CSMS$DECLARE_DECOMP. This information is then used by CSMS$DISTRIBUTE to preserve the lower bounds of the decomposed arrays.

To improve the clarity of this example, constants were defined using CSMS$INSERT directive to define the local sizes, number of processors and halo thicknesses of each dimension. These computations could also have been stated explicitly as parameters in the directive (eg. CSMS$DECLARE_DECOMP(my_decomp, <NX+1/4+2*3+1,...>), avoiding the insert declarations.

The size declarations (*NX_LOCAL*, *NY_LOCAL*) will be explained in detail. The first terms (*NX+1/nprocs_x*, *NY+1/nprocs_y*) indicate the minimum number of processors permitted will be 12 because 4 (nprocs_x) and 3 (nprocs_y) processors are required for each dimension to satisfy computations on the array "a". The second terms (thick_x, thick_y) indicate sufficient storage is allowed for halo thickness of 3 in both directions (indicated by 2*3). Finally, an additional unit of storage is included to account for round off errors on varying values of NX and NY in the third term.

Note: the number of processors assigned to each dimension (*nprocs_x*, *nprocs_y*) is determined at run-time based on the number of processors assigned to the program and the global dimensions of the decomposed arrays. See the SMS Users Guide for a more detailed discussion.

```
      integer nx,ny
      parameter(nx=64, ny=48)

CSMS$INSERT       parameter(nprocs_x=4, nprocs_y=3)

CSMS$INSERT       parameter(thick_x=2*3, thick_y=2*3)

CSMS$INSERT       parameter(NX_LOCAL = (NX+1)/nprocs_x+thick_x+1)
CSMS$INSERT       parameter(NY_LOCAL = (NY+1)/nprocs_y+thick_y+1)

CSMS$DECLARE_DECOMP(my_decomp, <NX_LOCAL, NY_LOCAL>:<0,0>)

CSMS$DISTRIBUTE (my_decomp, NX, NY) BEGIN
      real a(0:NX,0:NY)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(my_decomp,<NX+1,NY+1>, <0,0>)
```

```
CSMS$PARALLEL (my_decomp,NX,NY) BEGIN
      do i=0, NX
        do j=0, NY

          a(i,j) = 0.0

c     more data parallel computations  ...

        enddo
      enddo
CSMS$PARALLEL EN
```

CSMS$DISTRIBUTE

SYNTAX

CSMS$DISTRIBUTE ([<] decomp [(nest)],
[<tags1>] [,<tags2> [,<tags3>]]
[: TREAT_AS_COMPLEX][>]
[, <decomp2, ...>]) BEGIN

CSMS$DISTRIBUTE END

Required Fields:

decomp	(var)	name of the data decompostion

Optional Fields

nest	(expr-)	decomposition nest index
tags1	(var|int)	list of variables that identify the 1st decomposed dimension.
tags2	(var|int)	list of variables that identify the 2nd decomposed dimension
tags3	(var|int)	list of variables that identify the 3rd decomposed dimension
TREAT_AS_COMPLEX		keyword specifies real declarations should be treated as their complex equivalent (see note 2)
decomp2		another decomposition couplet - with options as defined above. This is useful for nesting. See Example 3.

SEMANTICS

This directive links the data decomposition structures, defined by CSMS$DECLARE_DECOMP and created by CSMS$CREATE_DECOMP, with arrays targeted for decomposition. CSMS$DISTRIBUTE is the heart of PPP handling of decomposed data in the user's program. Two fundamental aspects of distributed arrays are handled: (1) identifying arrays that are decomposed and how they are decomposed, and (2) replacing the sizes of the decomposed array dimensions with their corresponding processor local sizes.

Variables listed in the directive *(tags1,2,3*) identify which array dimensions are decomposed. Further, a mapping between each decomposed array dimension and the dimensions of the data decomposition is also determined. This mapping is used to insure correct handling of each array for exchanges,

transfers, I/O operations, local and global address translations, and reductions.

Replacing the array's decomposed dimensions with processor local sizes depend on identifying the memory allocation scheme that was used (given by CSMS$DECLARE_DECOMP). For static allocation, the declared local size, specified in CSMS$DECLARE_DECOMP *(declared1,2,3*) will be used. In the dynamic case the size of each decomposed dimension, computed at run-time by SMS, will be used.

If dynamic memory allocation is used, CSMS$CREATE_DECOMP cannot appear after a CSMS$DISTRIBUTE because the sizes of decomposition structures, determined at run-time, cannot be assigned before they are initialized. See Example 3 in CSMS$CREATE_DECOMP for more details.

The optional keyword TREAT_AS_COMPLEX applies to real arrays that store the real and imaginary parts of complex numbers in adjacent memory locations. If this keyword is used, all declarations contained in the enclosed directive will be treated as though they are complex values by SMS (eg. for I/O, transfers, reductions, exchanges). See Example 2 for more details.

LIMITATIONS

1. The translation of assumed size declarations (eg. a(nx,*)) are not supported.

MESSAGES

ERRORS

The name given for this decomposition has not been defined or is not in scope.

The data decomposition name given by parameter decomp (above), must have been defined by CSMS$DECLARE_DECOMP and be in scope.

CSMS$DISTRIBUTE can only be used for declarative statements.

This directive was designed to operate on array declarations. It cannot be used for subroutine or function declarations, assignment statements, etc.

WARNINGS

This array, decomposed by CSMS$DISTRIBUTE, is being used outside of a parallel region.

Typically, a decomposed array (defined by the CSMS$DISTRIBUTE), is used within a declared parallel region (CSMS$PARALLEL).

RELATED DIRECTIVES

CSMS$DECLARE_DECOMP
CSMS$CREATE_DECOMP

NOTES

1. Brackets: <> are not required when a single tag (tag1, 2, 3) is used to specify a decomposed dimension. For example CSMS$DISTRIBUTE(dh,nx,ny) can be used, where nx relates to the first dimension and ny to the second.

2. An array is not decomposed (1) if it is outside the scope of a CSMS$DISTRIBUTE directive or (2) none of its dimensions are declared with one of the tags specified in the enclosed distribute directive.

EXAMPLES

Example 1: Array Declaration Examples
Example 2: TREAT_AS_COMPLEX Array Handling
Example 3: Multiple data Decompositions

Example 1: Array Declaration Examples

In the example below, all of the arrays EXCEPT *x5* will be decomposed according to data decomposition *dh*. The variable *x5* will not be decomposed (or translated) because none of its dimensions matches the tags listed in CSMS$DISTRIBUTE. Generated parallel output illustrates these points.

```
CSMS$DECLARE_DECOMP( dh, <im/2+1, jm/2+1>)
CSMS$DISTRIBUTE( dh, im, jm) BEGIN
      real x0(im,5)
      real x1(im,jm)
      real x2(3,im,jm)
      real x3(jm,3,im)
      real x4(jm,im)
      real x5(10,20,km)
```

```
CSMS$DISTRIBUTE END
```

.....

GENERATED PARALLEL PSEUDO-CODE

Each of the arrays listed is decomposed differently as shown in the output generated. For example, array *x3*'s first dimension is associated with the second dimension of the data decomposition: *dh*. The second dimension is not decomposed and the third is associated with the first dimension of the decomposition. The translated output assigns the appropriate declared size parameter that is defined by output generated by CSMS$DECLARE_DECOMP.

.....

```
C CSMS$DISTRIBUTE( dh, im, jm) BEGIN
      real x0( Declared_Size_1, 5)
      real x1( Declared_Size_1, Declared_Size_2)
      real x2( 3, Declared_Size_1, Declared_Size_2)
      real x3( Declared_Size_2,3, Declared_Size_1)
      real x4( Declared_Size_2, Declared_Size_1)
      real x5( 10,20,km)
C CSMS$DISTRIBUTE END
```

Example 2: TREAT_AS_COMPLEX Array Handling

A common method for the treatment of complex variables is to store the real and imaginary parts into adjacent real value pairs. When TREAT_AS_COMPLEX is used, all real arrays within the enclosed CSMS$DISTRIBUTE will be handled as though they are complex variables.

In this example, the following two declarations will be treated exactly the same way by the SMS run-time system. The inner most dimension of the real declaration is twice the size of the complex declaration. However, the user's treatment of these different declarations only requires the TREAT-AS_COMPLEX keyword. No other changes are required by the user. SMS takes care of all the details required to treat these real-valued pairs as complex variables.

```
CSMS$DECLARE_DECOMP(my_dh, <jtrun>)
CSMS$DISTRIBUTE(my_dh, <jtrun>) BEGIN
      complex a(jtrun, lev, my)
CSMS$DISTRIBUTE END

CSMS$DISTRIBUTE(my_dh, <jtrun> : TREAT_AS_COMPLEX) BEGIN
      real ar(jtrun*2, lev, my)
CSMS$DISTRIBUTE END
```

Example 3: Multiple Data Decompositions

The parallelization of nested finite difference approximation (FDA) models frequently requires multiple grids be in scope. In this example, a fine and a coarse mesh decomposition are dynamically allocated and initialized using CSMS$DECLARE_DECOMP and CSMS$CREATE_DECOMP. These decompositions are used by CSMS$DISTRIBUTE to decompose the fine and coarse variables *fldfm* and *fldcm* respectively. The scratch arrays *cwork* and *fwork* are similarly decomposed.

After coarse grid computations are done, CSMS$TRANSFER is used to transfer data to the fine grid. Refer to the SMS Users Guide for more information on nesting.

Note: Multiple decompositions are expressed in a single CSMS$DISTRIBUTE directive to avoid re-ordering the coarse and fine mesh array declarations. In this scenario, two CSMS$DISTRIBUTE directives would be used: one for the coarse mesh variables and the other for the fine mesh ones.

```
      parameter (maxgrds=2)
      parameter (mcm=512, ncm=512, mfm=512, nfm=512)
CSMS$DECLARE_DECOMP(dh(maxgrds), <mcm/4, ncm/4>)

CSMS$DISTRIBUTE(<dh(1),mcm,ncm>, <dh(2),mfm,nfm>) BEGIN
      real fldcm (mcm,ncm,kk)
      real fldfm (mfm,nfm,kk)

      real cwork (mcm, ncm)
      real fwork (mfm, nfm)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(dh(1),<mcm, ncm>, <0,0>)
CSMS$CREATE_DECOMP(dh(2),<mfm, nfm>, <0,0>)

c     coarse grid computations ...

C     transfer results to the fine grid
CSMS$TRANSFER(<fldcm, fldfm>)

C     fine grid computations ...
```

CSMS$EXCHANGE

SYNTAX

CSMS$EXCHANGE (
Var1 [< lower1:upper1, lower2:upper2, lower3:upper3>]
[,Var2 [<...>]])

Required Fields:

Var1	(var)	variable to be exchanged.

Optional Fields:

Var2	(var)	another variable to be exchanged
dim1	(var|int)	thickness of halo to be exchanged in the first decomposed dimension.
dim2	(var|int)	thickness of halo to be exchanged in the second decomposed dimension.
dim3	(var|int)	thickness of halo to be exchanged in the third decomposed dimension.

NOTE: dim1, dim2, and dim3 assume the thickness of upper and lower halos to be exchanged is the same. If you wish to exchange halo data with different lower and upper the field dim1, dim2 or dim3 can be replaced with:

lower:upper

for any decomposed dimension.

SEMANTICS

This directive communicates with neighboring processors to update halo or ghost regions. Only the halo regions are updated; if other local data must be moved between variables that are decomposed differently (or not decomposed) then CSMS$TRANSFER should be used. If multiple arrays are exchanged within a single directive, the exchanges are aggregated (combined) to improve performance. Information provided by CSMS$DISTRIBUTE is used to generate the correct communication code for each variable exchanged.

LIMITATIONS

1. This directive will not work for variables whose interior region is smaller than the halo thickness (defined

by CSMS$CREATE_DECOMP) in a given dimension.

2. This directive does not work with decomposition dimensions containing scrambled indices.

MESSAGES

ERRORS

Cannot exchange a variable that has not been decomposed.

A variable must be decomposed in order for its local data to be exchanged with neighboring processors

RELATED DIRECTIVES

CSMS$CREATE_DECOMP
CSMS$CREATE_PACKED_SPEC_DECOMP
CSMS$DISTRIBUTE
CSMS$HALO_COMP
CSMS$TRANSFER

NOTES

EXAMPLES

Example 1: Dependent Loop Computations

In this example, we have chosen to do exchanges after every dependent loop. This technique is useful on machines having (relatively) low latencies and high band-widths for inter-processor communication. Redundant computations are avoided, but communication is needed before every loop. This code will scale well for large numbers of processors provided inter-processor communication latency is low and bandwidth is high.

Another approach that is useful on high latency machines is to eliminate the second exchange by doing redundant computations in the halo region of the variable *wk1*. Example 1 in CSMS$HALO_COMP illustrates this approach.

```
      subroutine smooth(x)

C Include all stuff not passed in (irrelevant to example).
```

```
      include 'everything.h'

CSMS$DISTRIBUTE(DH_GRID, nx, ny) BEGIN
      real x(nx,ny)
C Local declaration
      real wk1(nx,ny)
CSMS$DISTRIBUTE END

C Exchange variable x to update its halo region.
CSMS$EXCHANGE(x)

CSMS$PARALLEL (DH_GRID, <i>, <j>) BEGIN
C Smoother computations.
      do 20 j=2,ny-1
      do 20 i=2,nx-1
        wk1(i,j)=0.5*x(i,j)+0.125*
     &           (x(i-1,j)+x(i+1,j)+x(i,j-1)+x(i,j+1))
   20 continue
CSMS$EXCHANGE(wk1)

      do 40 j=2,ny-1
      do 40 i=2,nx-1
        x(i,j)=0.5*wk1(i,j)+0.125*
     &       (wk1(i-1,j)+wk1(i+1,j)+wk1(i,j-1)+wk1(i,j+1))
   40 continue
CSMS$PARALLEL END

c more computations ...
```

CSMS$EXIT

SYNTAX

CSMS$EXIT [([status])]

Required Fields:
none.

Optional Fields:

status	(int)	exit status reported by SMS when the program terminates.

SEMANTICS

The termination of a program using SMS can either abort or exit normally. A normal exit from SMS will insure proper and orderly process termination. The SMS control process will wait until every processor's computations, communications and I/O are complete before exiting. An abort from SMS will terminate all processes immediately, regardless of state.

PPP automatically generates code to either abort or exit in two different ways. Code that tells SMS to **abort** (PPP_ABORT) is generated whenever a "stop" statement is encountered in the Fortran source. Code to terminate a program normally (PPP_EXIT) is generated by PPP whenever an "end" program statement is encountered in the source code.

This directive modifies the above default behavior from a program abort to a normal exit when CSMS$EXIT appears at the statement **prior** to the Fortran stop statement. For more information about SMS process control, refer to the SMS Users Guide.

LIMITATIONS

NONE

MESSAGES

NONE

RELATED DIRECTIVES

NOTES

EXAMPLES

Example 1: Program Termination

By default, code is automatically generated by PPP to PPP_ABORT whenever a stop statement is encountered unless CSMS$EXIT appears in the previous statement. In this example, the CSMS$EXIT is used to insure a normal program termination if the abort flag is false.

```
      program main

      logical abort

      abort = .false.
      call model( params, abort)

      if (abort .eq. .true.) stop

CSMS$EXIT
      stop
      end
```

CSMS$FLUSH_OUTPUT

SYNTAX

CSMS$FLUSH_OUTPUT

Required Fields

NONE.

SEMANTICS

CSMS$FLUSH_OUTPUT allows compute processes to continue model computations concurrent with data being written to disk. This directive should be used whenever the application has a long period of computations before the next I/O statement is reached. See the SMS Users Guide for more details on the use of I/O cache processors.

This directive is never required; it is only used to improve performance. Further, this directive has no effect on the file contents; output is exactly the same with or without this directive.

LIMITATIONS

NONE

MESSAGES

NONE

RELATED DIRECTIVES

NONE

NOTES

Since read and ASCII output statements interact with the I/O subsystem via the SMS server process, they should be avoided until the cache processors are finished servicing binary output operations resulting from the CSMS$FLUSH_OUTPUT.

If you are not using I/O cache processors, the following environment variables may be useful:
SMS_RBS - size of block for input (in Bytes)

SMS_RBC - number of blocks for input (default is 16)
SMS_WBS - size of blocks for output (in Bytes)
SMS_CLOSE_MODE - set to "require_flush" to defer output until CSMS$FLUSH_OUTPUT is encountered.

If input files are too large to fit in the physical memory of one processor, the following environment variable is useful:

SMS_RAN_RSTYLE - set to "one-var" to instruct SMS to read one input variable at a time.

The recommended values for these variables are:

SMS_RBS - size of the file /15 (size of the largest variable if "one-var" is used).
SMS_RBC - 16
SMS_WBS - size of the file
SMS_CLOSE_MODE - do not require-a flush

All of these variables are optimization hints to the SMS I/O sub-system. If values are not specified, SMS will do the best job it can and will make correct progress at the expense of performance. De-optimization values can result in extremely poor I/O performance. Refer to the SMS Users Guide for more information about optimizing parallel I/O operations using SMS.

EXAMPLES

Example 1: Overlap Computations with Model Output

This code segment outputs binary data to disk every "iout1" time steps though the routine "output". This code segment will only work efficiently if the SMS write operation prior to the CSMS$FLUSH_OUTPUT are complete before more binary I/O are done. Otherwise, all processes containing outstanding I/O requests must wait for write operations to complete before computations can resume.

```
c main model time-stepping loop
      do istep = 1, numsteps

c ... model computations ...
        call compute

c some binary output to disk
        if (mod(istep,iout1).eq.0) then
          call output
CSMS$FLUSH_OUTPUT
```

```
  endif
enddo
```

CSMS$GLOBAL_INDEX

SYNTAX

```
CSMS$GLOBAL_INDEX (
        dim1 [>]
     [, dim2 [>] ] ) BEGIN

CSMS$GLOBAL_INDEX END
```

Required Fields:

dim1	(int)	decomposed dimension for which array references should be treated as global.

Optional Fields:

dim2	(int)	another decomposed dimension for which array references should be treated as global.

SEMANTICS

This directive is used to translate indices of decomposed arrays from global references to processor-local references. Typical uses of this directive are for model boundary handling, and the output of diagnostic messages on elements of decomposed arrays.

The name of the data decomposition is determined either explicitly (*decomp1*, *decomp2*) or by using the default decomposition defined by an enclosing parallel (CSMS$PARALLEL) region. The relationship between the array dimension and the decomposition dimension declared global by this directive (*dim1*, *dim2*) is determined by CSMS$DISTRIBUTE. See the examples for more details.

LIMITATIONS

1. A maximum of two dimensions is supported.
2. The translation of decomposed array indices that are themselves arrays, is not currently permitted. In a future update we plan to support these translations.
3. Read, write, and print statements that occur within the scope of a CSMS$GLOBAL_INDEX directive will not be translated. See Example 4 for more details.
4. Else clauses of an if-then-else must relate to the same

local data given as tested by the if-conditional. See Example 2 for more details.

5. Nested CSMS$GLOBAL_INDEX directives are not permitted.

MESSAGES

ERRORS

No data decomposition has been specified for this directive.

Either the data decomposition must be given explicitly in the global_index (decomp1, decomp2), or this directive must be contained within an enclosing parallel region (CSMS$PARALLEL).

CSMS$GLOBAL_INDEX translations of array indices on I/O statements are not supported.

These types of translations are not currently supported. See Example 4 for a coding alternative.

The translation of this indirect array reference is not currently supported.

The translation of decomposed array indices that are themselves arrays, is not currently permitted. In a future update we plan to support these translations.

Non-unit stride loops within the scope of a global_index directive are not currently translated.

We plan to update PPP to support these loops.

RELATED DIRECTIVES

CSMS$DISTRIBUTE
CSMS$INSERT
CSMS$PARALLEL
CSMS$REMOVE

NOTES

EXAMPLES

Example 1: Boundary Initialization
Example 2: Conditional Statement Handling
Example 3: Handling of Decomposed Dimensions

Example 4: Translation of I/O statements

Example 1: Boundary Initialization

A typical use of this directive is to initialize the boundaries of decomposed arrays. In this example, *u* and *v* are both decomposed in two dimensions by CSMS$DISTRIBUTE. To initialize the east-west global boundaries of the decomposed arrays *u* and *v*, CSMS$GLOBAL_INDEX(1) is used. Similarly, CSMS$GLOBAL_INDEX(2) is used to initialize the north-south boundaries (not shown).

Each processor contains a sub-region of the decomposed arrays *u* and *v*. The generated code will insure that only those processors that contain the east-west global boundaries will execute the assignment statements.

```
CSMS$DISTRIBUTE(dh,m,n) BEGIN
      real u(m,n,k)
      real v(m,n,k)
CSMS$DISTRIBUTE END

CSMS$PARALLEL (dh, i,j) BEGIN
CSMS$GLOBAL_INDEX(1) BEGIN
      do j=1, n
        v(1,j,k) = v(1,j,k) - vav
        u(m,j,k) = u(m,j,k) - vav
      enddo
CSMS$GLOBAL_INDEX END
CSMS$PARALLEL END
```

Example 2: Conditional Statement Handling

It is important to insure that all decomposed array references within an if-else statement containing a CSMS$GLOBAL_INDEX directive, are local to the processor where computations are done. For example, an out of bounds reference would be generated if *ek(m,j,k)* were referenced within the if loop because that data would not be local to the processors that contain the first row of the global array *ek*.

```
CSMS$DISTRIBUTE(dh,m,n) BEGIN
      real ek(m,n,k)
      real em(m,n,k)
      real u3(m,n,k)
CSMS$DISTRIBUTE END

CSMS$PARALLEL (dh,m,n) BEGIN
      do j=1, n
CSMS$GLOBAL_INDEX(1) BEGIN
```

```
          if(u3(1,j,k) .eq. 0) then
            ek(1,j,k) = 2.0*ek(2,j,k)+ek(3,j,k)
            em(1,j,k) = 2.0*em(2,j,k)+em(3,j,k)
          else
            ek(1,j,k) = ek(2,j,k)
            em(1,j,k) = ek(2,j,k)
          endif
CSMS$GLOBAL_INDEX END
        enddo
CSMS$PARALLEL END
```

Example 3: Handling of Decomposed Dimensions

In this example we illustrate the binding of the decomposed dimension given by the CSMS$GLOBAL_INDEX directive with the data decomposition as it applies to the arrays. Viewing the CSMS$DISTRIBUTE tagged variables (m,n), we notice that *avz* and *zsfc* are decomposed in both dims but *zygeo* is only decomposed in the 2^{nd} (j) dimension. This will affect the translation of the assignment statement as illustrated by the generated parallel code.

Proper translation requires that each variable in the assignment statement be checked to see if it is decomposed in the first dimension (specified by GLOBAL_INDEX(1)) as defined by CSMS$DISTRIBUTE. Using this decomposition information, PPP translates only the first index of *avz* and *zsfc*, leaving the other variables alone.

```
CSMS$DISTRIBUTE(dh,m,n) BEGIN
      real avz(m,n)
      real zygeo(n,k)
      real zsfc(m,n)
CSMS$DISTRIBUTE END
      real sigma(k)

CSMS$PARALLEL(dh,,j) BEGIN
      if(condition_met)then
        do k=1,k
          do j=1,n
CSMS$GLOBAL_INDEX(1) BEGIN
            zygeo(j,k)=sigmma(k)*avz(1,j)+zsfc(1,j)
CSMS$GLOBAL_INDEX END
          enddo
        enddo
      endif

c     More code ...
CSMS$PARALLEL END
```

Example 4: Translation of I/O statements

CSMS$GLOBAL_INDEX translations of array indices for I/O statements are not currently supported. These translations are complicated by the requirement that statements often need to be broken into multiple fragments because no single processor has all of the data in its local memory. In the event a format descriptor exists, it must be broken up too.

```
      parameter(nx=16,ny=16)
csms$declare_decomp(dh,nx_a,ny_a)

csms$distribute(dh,nx,ny) begin
       real a(nx,ny)
csms$distribute end

csms$parallel(dh) begin

csms$global_index( 1, 2 ) begin
      print a(1,1),a(1,ny),a(nx,ny),a(nx,1)
csms$global_index end

csms$parallel end
```

.....

Re-coded Input File

Two re-coding techniques can be used to resolve this print statement. A reduction operation can be done to gather the corner points by each processor and then output by a single print statement. This technique is illustrated in Example 3 of CSMS$REDUCE.

Another approach is to use CSMS$INSERT and CSMS$REMOVE to re-code this segment. As illustrated below, the print statement is broken into four statements (one for each corner point). Do-loops are used and translated (if the loop variables appear in CSMS$PARALLEL) to insure only those processors containing the array references will output them. Further, asynchronous output is required to avoid processor deadlock (see CSMS$PRINT_MODE for more details).

Note: While the first approach will guarantee the same output as the original code, it will execute more slowly because of the extra communications required by the reduction.

.....

```
      parameter(nx=16,ny=16)
csms$declare_decomp(dh, <nx,ny> )

csms$distribute(dh, nx, ny) begin
       real a(nx,ny)
csms$distribute end
```

```
csms$parallel (dh, i, j) begin

csms$remove begin
      print *,a(1,1),a(1,ny>,a(nx,ny),a(nx,1)
csms$remove end

csms$print_mode (sync) begin
csms$insert       do i=1,1
csms$insert         do j=1,1
csms$insert           print *,a(I,j)
csms$insert         enddo
csms$insert       enddo

csms$insert       do i=1,1
csms$insert         do j=ny,ny
csms$insert           print *,a(i,j)
csms$insert         enddo
csms$insert       enddo

csms$insert       do i=nx,nx
csms$insert         do j=ny,ny
csms$insert           print *,a(i,j)
csms$insert         enddo
csms$insert       enddo

csms$insert       do i=nx,nx
csms$insert         do j=1,1
csms$insert           print *,a(i,j)
csms$insert         enddo
csms$insert       enddo

csms$print_mode end
csms$parallel end
```

CSMS$HALO_COMP

SYNTAX

**CSMS$HALO_COMP (< lower1, upper1> [<,lower2, upper2>
[,lower3, upper3>]]) BEGIN**

CSMS$HALO_COMP END

Required Fields:

lower1	(expr-)	number of steps in the halo region computations will be done for the lower (left-most) local boundary in the 1st decomposed dimension.
upper1	(expr-)	number of steps in the halo region computations will be done for the upper (right)local boundary in the 1st decomposed dimension

Optional Fields:

Lower2	(expr-)	as above for the lower boundary of the 2nd decomposed dimension.
Upper2	(expr-)	as above for the upper boundary of the 2nd decomposed dimension
Lower3	(expr-)	as above for the lower boundary of the 3rd decomposed dimension.
Upper3	(expr-)	as above for the upper boundary of the 3rd decomposed dimension .

SEMANTICS

This directive is used to control the number of steps into the halo or ghost region that computations will be done. The default behavior (when CSMS$HALO_COMP is not used) is to avoid doing any computations in the halo region. Computations in the halo region are called "redundant computations" because each halo point corresponds to an interior point of a neighboring processor's local memory. As a result, computations on a halo point will be done on at least two processors. The benefit of doing these extra computes is that communication between processors can be reduced leading to an overall improvement of performance on many machines.

LIMITATIONS

NONE

MESSAGES

ERRORS

CSMS$HALO_COMP can only be used within an active parallel region.

Must be bounded by CSMS$PARALLEL BEGIN / END.

RELATED DIRECTIVES

CSMS$DECLARE_DECOMP
CSMS$DISTRIBUTE
CSMS$EXCHANGE
CSMS$PARALLEL

NOTES

1. This directive must be used within the scope of a parallel region.

2. The minimum number of steps into the halo region permitted is zero; the maximum number is the halo thickness (defined by CSMS$CREATE_DECOMP) for the dimension of the decomposition (given by CSMS$PARALLEL)

EXAMPLES

Example 1: Trading Computations for Communication

This example illustrates an SMS method to trade-off redundant computations in the halo region for reduced communications. CSMS$HALO_COMP(<1,1>,<1,1>) indicates redundant computations will be done one step into the halo region of the first two decomposed dimensions. This eliminates the requirement that wk1 be updated before it is required. After loop 40 the array *x* is valid only in the interior (all halo points are in need of an update via an exchange).

Note: This approach is useful on machines that have (relatively) high latencies for inter-processor communication. Communication is needed only once at the beginning of subroutine smooth. Redundant computations will limit the scalability for large numbers of processors however. Refer to the SMS Users Guide for more information.

CSMS$EXCHANGE (Example 1) illustrates this same code segment with no redundant computations in the halo region. Instead an extra CSMS$EXCHANGE is required to update *wk1* before the 40 loop given below.

```
      subroutine smooth(x)
C Include all stuff not passed in (irrelevant to example).
      include 'everything.h'
CSMS$DISTRIBUTE(DH_GRID, <nx, ny>) BEGIN
C Subroutine arguments.
      real x(nx,ny)
C Local declarations.
      real wk1(nx,ny),wk2(nx,ny)
CSMS$DISTRIBUTE END

C Exchange variable x to update its halo region.
CSMS$EXCHANGE(x)

CSMS$PARALLEL (DH_GRID, <i>, <j>) BEGIN
C Smoother computations.
CSMS$HALO_COMP(<1,1>, <1,1>) BEGIN
      do 20 j=2,ny-1
      do 20 i=2,nx-1
        wk1(i,j)=0.5*x(i,j)
     &          +0.125*(x(i-1,j)+x(i+1,j)+x(i,j-1)+x(i,j+1))
   20 continue
CSMS$HALO_COMP END

C no exchange is required here due to redundant comps in the halo.

CSMS$HALO_COMP END
      do 40 j=2,ny-1
      do 40 i=2,nx-1
        x(i,j)=0.5*wk1(i,j)
     &         +0.125*(wk1(i-1,j)+wk1(i+1,j)+wk1(i,j-1)+wk1(i,j+1))
   40 continue
CSMS$PARALLEL END
```

CSMS$IGNORE

SYNTAX

CSMS$IGNORE BEGIN
CSMS$IGNORE END

Required Fields:
NONE

SEMANTICS

This directive informs PPP about sections of code that should not be translated but should still be retained in the code.

LIMITATIONS

NONE

MESSAGES

NONE

RELATED DIRECTIVES

CSMS$INSERT
CSMS$REMOVE

NOTES

NONE

EXAMPLES

Example 1: File Format Conversion

Binary fortran files are not portable across operating systems due to non-standard record handling. As a result, we normally convert these files to MPI external IO format, a standard that SMS uses for binary I/O. In this code example, we do not want PPP to translate the original

open/read/close statements because we need to read the Fortran binary formatted input file.

In the generated output, PPP only converts the fortran write statements and "ignores" the read statements. Write statements are translated into calls to SMS library routines that will output these binary data into the MPI external IO format. The transformed code should be linked with the single process version of the SMS library and run on one processor. Refer to the SMS Users Manual for more information on I/O operations.

```
CSMS$IGNORE BEGIN
C read Fortran unformatted binary file
      open(18, file ='binary.dat',form='unformatted')
      read(18) a,b,c
      close (18)
CSMS$IGNORE END

C write SMS-format binary file
      open (18, file='binary.SMS_dat',form='unformatted')
      write (18) a,b,c
      close (18)
```

Example 2: Collapsed Loop Computations

In this example, we compute the vector length of the first two decomposed dimensions and store the result in "mn". We then use this value as a stop loop bound on the collapsed computation found in the first loop. In this case, we do not want PPP to touch the "i" loop so the CSMS$IGNORE is used.

Note: In almost all cases, better performance can be achieved by re-writing the collapsed loops in their standard 2 or 3 dimensional form. Loop collapsing is an outdated hand-optimization that usually does not improve performance.

Further, if collapsed loops are used on decomposed arrays that also contain halos, it may not be possible to avoid doing computations on the un-initialized values in these regions. Again, we recommend re-writing these loops.

```
      subroutine comp(m,n,kk)

CSMS$DECLARE_DECOMP(dh)
CSMS$DISTRIBUTE (dh,m,n) BEGIN
      real fldin (m,n,kk)
CSMS$DISTRIBUTE END
```

```
CSMS$PARALLEL(dh, i, j) BEGIN
CSMS$TO_LOCAL( <1,m>, <2,n> :SIZE) BEGIN
      mn = m * n
CSMS$TO_LOCAL END


      do k=1,kk
CSMS$IGNORE BEGIN
        do i=1,mn
          fldin (i,1,k)=0.0
        enddo
CSMS$IGNORE END
      enddo
c
      do i = 1,m
        do j = 1, n

c          array computations ...

        enddo
      enddo
CSMS$PARALLEL END
```

CSMS$INSERT

SYNTAX

CSMS$INSERT *"line of code"*

Required Fields:
NONE
Optional Fields:
NONE

SEMANTICS

This directive allows users to insert code to be parallelized by PPP. Each line that is inserted must be prefaced by CSMS$INSERT and adhere to Fortran 77 fixed format rules. A single space separator is required for statement labels and and additional six spaces for non-labelled fortran statements.

LIMITATIONS

NONE

MESSAGES

NONE

RELATED DIRECTIVES

CSMS$REMOVE
CSMS$IGNORE

NOTES

EXAMPLES

Example 1: Output File Redirection

In this example, CSMS$INSERT and CSMS$REMOVE are used to select input data files based on whether a parallel run (using SMS) is executed. Different files are used because SMS uses the MPI external data format for binary I/O. While

this format is portable, standard Fortran binary files are not.

```
CSMS$REMOVE BEGIN
      open(36,file='myfile',form='unformatted')
CSMS$REMOVE END

CSMS$INSERT       open(36,file='myfile.SMS',form='unformatted')
      read(36) u,v,w,p,t,qv
      close(36)
```

CSMS$MESSAGE

SYNTAX

CSMS$MESSAGE (Action, "Text")

Required Fields:

Action	(key)	message action - options are: ABORT, WARN, INFORM
Text	(string)	quoted string to be output

SEMANTICS

This directive is used to inform the user at run-time of code segments that are being executed by SMS that may be problematic for PPP. This is useful when code segments do not appear to be executed. Unable to consult the code author, an alternative to spending lots of time rewriting their code is to simply ABORT with a message from SMS if the code ever is executed. This serves as a useful alternative to simply removing the code with an CSMS$REMOVE or reworking the code segment.

Three message actions are available. ABORT is the only action that will halt execution of the program. WARN and INFORM will only write the given message (*Text*) to stderr or stdout respectively.

LIMITATIONS

NONE

MESSAGES

ERRORS

Supported message actions are: WARN, ABORT, AND INFORM.

RELATED DIRECTIVES

CSMS$INSERT
CSMS$PRINT_MODE
CSMS$REMOVE

NOTES

1. The type of output (eg. ASYNC, ROOT, etc.) produced from this directive will depend on the enclosing CSMS$PRINT_MODE specification. By default, asynchronous output will be used.

EXAMPLES

Example 1: Unsupported Code

In this example, we observe a periodic boundary initialization where *nx* is the right-hand bound for the first dimension. This segment cannot be parallelized without rewriting the code to separate the left-side and right-side decomposed array references on *ekm*.

Rather than modify this code, we use CSMS$MESSAGE to abort in the event this code segment is executed. At that point, this code can be reworked.

```
CSMS$DISTRIBUTE (dh,nx,my) BEGIN
      real ekm(nx,my,kk)
CSMS$DISTRIBUTE END

c     condition_ever_met set by other computations in the model

CSMS$PARALLEL(dh,i,j) BEGIN
      if (condition_ever_met) then
CSMS$MESSAGE(ABORT,'This code is not supported by sms')

        do k=1, kk
          do j=1,n
            ekm(1,j,k)=ekm(nx,j,k)

C           more periodic array references ...

          enddo
        enddo
      endif
c
c .. other code ...

CSMS$PARALLEL END
```

CSMS$PARALLEL

SYNTAX

CSMS$PARALLEL (decomp [(nest)]
[,< ivars > [,< jvars >] [,< kvars >]) BEGIN

CSMS$PARALLEL END

Required Fields:

decomp	(var)	name of the data decomposition

Optional Fields:

nest	(expr-)	decomposition nest index
ivars	(var)	comma separated list of variables used to reference arrays decomposed by *decomp* in the 1st decomposed dimension
jvars	(var)	comma separated list of variable used to reference arrays decomposed by *decomp* in the 2nd decomposed dimension
kvars	(var)	comma separated list of variables used to reference arrays decomposed by *decomp* in the 3rd decomposed dimension

SEMANTICS

Defines a region over which parallel computations will be done on each processor's local data, as defined by the given data decomposition (*decomp*). All do-loops inside a parallel region that reference the specified loop variables *(ivars, jvars, kvars*) will be translated. This directive also provides a default data decomposition context for other PPP directives used within the scope of the parallel region.

LIMITATIONS

1. Explicit nesting of parallel regions is not permitted.

MESSAGES

ERRORS

Nesting of CSMS$PARALLEL directives is not permitted.

It is likely that a CSMS$PARALLEL END directive is needed.

CSMS$PARALLEL END directive requires a corresponding CSMS$PARALLEL BEGIN.

Parallel Begin / End pairs are required.

The number of dimensions given exceeds the rank of the data decomposition.

The number of dimensions specified by the parallel region arguments (ivar1, jvar1, kvar1) cannot exceed the number of dimensions in the data decomposition (decomp) defined by CSMS$DECLARE_DECOMP. Be sure to separate each dimension's parallel variables with brackets <> (see note 2).

The decomposition given for this directive has not been defined or is not in scope.

The decomposition name, given by decomp, must have been declared by CSMS$DECLARE_DECOMP and be visible to this CSMS$PARALLEL directive.

NOTES

Non-unit stride do-loop detected.

Negative unit stride do-loop detected.

RELATED DIRECTIVES

CSMS$DECLARE_DECOMP
CSMS$CREATE_DECOMP
CSMS$DISTRIBUTE
CSMS$HALO_COMP
CSMS$TO_LOCAL
CSMS$TO_GLOBAL
CSMS$GLOBAL_INDEX
CSMS$REDUCE

NOTES

1. Brackets (<>) can be omitted if a single variable is listed for that dimension (ivar1, jvar1, kvar1). Example 1 illustrates this point.

EXAMPLES

Example 1: Decomposed Loop Generation
Example 2: Non-Unit Stride Loop Handling

Example 1: Decomposed Loop Generation

In this example the only loops that are translated into local start and stop array references are for those variables that are listed in the CSMS$PARALLEL directive. In this case we have specified *i* for the first decomposed dimension, and *j* for the second. The *k* loop is not translated.

```
CSMS$DISTRIBUTE(decomp, i, j) BEGIN
      real f(m,n)
CSMS$DISTRIBUTE END

CSMS$PARALLEL (decomp, i, j) BEGIN
      do k=1, levs
        do j=1, n
          do i=1, m
            f(i,j)= 0.0
          enddo
        enddo
      enddo
CSMS$PARALLEL END
```

.....

GENERATED PARALLEL PSEUDO-CODE

Translation vectors (eg. *i_start*, *i_stop*), created at run-time by CSMS$CREATE_DECOMP, are used to define the processor local start and stop loop bounds. Factors that affect these translation vectors are (1) the data decomposition, (2) the data decomposition dimension, (3) the number of steps into the halo region computations are done (CSMS$HALO_COMP), and (4) the decomposition nest level (if any).

.....

```
CSMS$PARALLEL (decomp, i,j) BEGIN
      do k = 1, levs

        do j= j_start(1), j_stop(n)
          do i = i_start(1), i_stop(m)

            f(i,j)= 0.0
          enddo
        enddo
      enddo
CSMS$PARALLEL END
```

Example 2: Non-unit Stride Loop Handling

A non-unit stride of 2 is used in this example. PPP translation will insure only globally addressed even numbered computations are done on each processor.

```
CSMS$DISTRIBUTE(decomp, jlen) BEGIN
      real*8 cc(jlen), bb(jlen)
CSMS$DISTRIBUTE END

CSMS$PARALLEL(decomp, m) BEGIN
      do 3 m=2, jlen, 2
        cc(m) = cc(m) + bb(m)
3     continue
CSMS$PARALLEL END
```

CSMS$PRINT_MODE

SYNTAX

CSMS$PRINT_MODE (mode) BEGIN
CSMS$PRINT_MODE END

Required Fields:
mode (key) print mode to be used for output.

Options are:
ASYNC -asynchronous output
ORDERED -ordered output
ROOT -root node output

SEMANTICS

This directive applies to standard output of strings. Four output mode options are available:

ASYNC Each processor prints its string when this statement is reached. The processors do not synchronize. Message ordering may differ from one run to the next.

ORDERED In fixed order, each processor prints its string. This mode is typically used for debugging. The processors synchronize so deadlock will occur if one or more processors do not execute the output statement

ROOT The designated root processor (node zero) prints its string. The processors do not synchronize.

There is also a default print mode. If the user does not specify a print mode using CSMS$PRINT_MODE, the environment variable SMS_PUTS_MODE is examined. SMS_PUTS_MODE can be set to any of the above modes at run-time. If it is not visible or set to something else then the mode reverts to the ROOT node`.

LIMITATIONS

NONE

MESSAGES

NONE

RELATED DIRECTIVES

CSMS$MESSAGE

NOTES

1. This directive also controls the output of messages generated by the CSMS$MESSAGE directive.

EXAMPLES

Example 1: Code Segment where ASYNC Mode is Required.

The processors that output *tstart* will depend on the environment variable SMS_PUTS_MODE. If SMS_PUTS_MODE is not set, only the ROOT node will output this statement.

The second write statement is only printed on a process where *x(i,j)* is greater than *thresh*. Since all processors may not satisfy this condition, asynchronous output (ASYNC) is required or a deadlock may occur. For more information about process control, refer to the SMS Users Guide.

```
      write (*,6000) tstart
 6000 format (' Model start time = ',f8.3)

CSMS$PARALLEL( dh, i, j) BEGIN
      do j = 1, ny
        do i = 1, nx
          if (x(i,j).gt.thresh) then
CSMS$PRINT_MODE(ASYNC) BEGIN
            write (*,6001) x(i,j)
CSMS$PRINT_MODE END

 6001 format (' Convergence error, x =  ',f11.3)
          endif
        enddo
      enddo
C...
```

CSMS$REDUCE

SYNTAX

Standard Reduction

CSMS$REDUCE(Var [,Vars], Function)

Required Fields:

Var	(var)	variable to be reduced
Function	(key)	type of reduction operation - supported functions are: MAX, MIN, SUM

Optional Fields:

Vars	(var)	other variables to be reduced

Bit-wise Exact Reduction

CSMS$REDUCE (<SrcVar1, DestVar1 [,Rdims]> [,<SrcVar2, DestVar2 [,Rdims]>][,...] [, SUM]) BEGIN

CSMS$REDUCE END

Required Fields:

SrcVar1	(var)	array variable to be reduced
DestVar1	(var)	variable where reduction result will be stored
Function	(var)	type of reduction operation - supported functions are: SUM

Optional Fields:

Rdims	(var)	array dimensions over which reduction operations will be done
SrcVar2	(var)	another array variable to be reduced
DestVar2	(var)	variable where result of SrcVar2 reduction will be stored
SUM	(key)	bit-wise exact sum

SEMANTICS

A reduction is used to determine a global MAX, MIN or SUM over all the processors. Two types of reductions are supported: standard and bit-wise exact reductions.

The **standard reduction** directive performs the given function on non-decomposed variables. The reduced variable will be stored in the variable being reduced (in place reduction).

The **bit-wise exact reduction** is used to insure exactly the same floating point SUM regardless of the number of processors. Since floating point arithmetic is not associative (due to round-off errors), this reduction is useful to insure precise parallel results. The variables to be reduced must be decomposed arrays and should be real or complex data types where round-off errors are an issue.

By default, all dimensions of the given array will be reduced unless reduction dimensions are explicitly stated (by *Rdims*). The size of those dimensions that are not reduced must match the corresponding dimensions of the destination array that will store the result. Example 1 illustrates this point.

LIMITATIONS

1. Standard reductions do not operate on decomposed arrays.

2. Reduction of complex types has not been implemented yet.

3. Implicitly typed variables cannot be reduced. This will be corrected in a future release.

4. Bit-wise reductions can be done on any of the first three decomposed dimensions.

MESSAGES

ERRORS

Bit-wise Reductions are allowed over any of the first 3 dimensions.

Any of the first three dimensions can be reduced. See Example 1.

Reductions are only permitted for "integer", "real" or "double" types.

These are the only types that can currently be reduced.

Source bit-wise reduction variables must be decomposed.

Source bit-wise exact reduction variables must be

arrays and they must be decomposed.

Supported standard reduction functions are MIN, MAX, and SUM.

These function names are case insensitive.

Supported bit-wise reduction functions are: SUM

This variable's type/precision is not currently supported.

*This refers to the fortran language extension that allows a precision modifier to be added to the type (eg. integer*4, logical*2,...). Currently PPP only supports the type/precision specifiers: real*4 and real*8.*

RELATED DIRECTIVES

CSMS$DISTRIBUTE
CSMS$PARALLEL

NOTES

1. If no reduction dimension (*Rdims*) is specified, the first three dimensions will be reduced.

2. PPP assumes all code within the CSMS$REDUCE is a reduction as specified in the directive. If it is not, then results will vary wildly between the bit-wise and standard reductions.

EXAMPLES

Example 1: Bit-wise Exact Sums over User Specified Dimensions

In this example we are reducing *pkc* over dimensions one and three. The size the array in the remaining dimension must match the declared size of the destination array *pksum*. In this case *pksum* contains two elements and this matches the size of the second dimension of *pkc*.

The code between the begin and end reduce will be replaced with a global sum that will insure exactly the same result regardless of the number of processors.

```
csms$distribute(grid_dh,<my>) begin
      real*8 pkc(nx,2,my)
csms$distribute end
      real*8 pksum(2)


csms$reduce( <pkc, pksum,1,3>,SUM) begin
      pksum = 0.0
      prsum = 0.0

        do j = 1, my
        do l = 1, 2
        do i = 1, nx
          pksum(1) = pksum(1) + pkc(i,l,j)
        enddo
        enddo
        enddo
csms$reduce end
```

Example 2: Standard Reductions

Standard reductions only operate on non-decomposed variables - in this case we compute a max, min and mean from the variable *dout*. Local values are stored in *dmax, dmin* and *dmean* and then reductions are done to get global results.

```
CSMS$DISTRIBUTE (dh, len) BEGIN
      real dout(len)
CSMS$DISTRIBUTE END

      real dmax, dmin, dmean

      dmax=dout(1)
      dmin=dout(1)
      dmean=0.0
CSMS$PARALLEL (dh, I) BEGIN
      do i=1,len
        dmax=max(dout(i),dmax)
        dmin=min(dout(i),dmin)
        dmean=dmean+dout(i)
      enddo
CSMS$PARALLEL END

CSMS$REDUCE (dmean, SUM)
CSMS$REDUCE (dmin, MAX)
CSMS$REDUCE (dmin, MIN)
      dmean=dmean/float(len)
      print *,dmax,dmin,dmean
```

Example 3: I/O Statement Output

In this example, we wish to output the corner points of a decomposed array. Two possibilities exist to handle this situation. First, we break the output statement into multiple print statements and hand code in local loops

(CSMS$GLOBAL_INDEX - Example 4).

Second, we can use CSMS$REDUCE to gather the corner points on each processor and then output the results. This parallelization requires we create a variable (*pts*) to hold the corner points. Using CSMS$GLOBAL_INDEX, the previously initialized values of *pts* are updated on those processors holding a local copy of each corner point. Finally we gather the corner points using the reduction operator: SUM.

```
CSMS$INSERT       real pts(4)
CSMS$DISTRIBUTE(dh,nx,ny) BEGIN
      real a(nx,ny)
CSMS$DISTRIBUTE END

CSMS$INSERT       do i=1,4
CSMS$INSERT         pts(i) = 0.0
CSMS$INSERT       enddo

CSMS$PARALLEL (dh, i, j) BEGIN

CSMS$REMOVE BEGIN
      write(6,'(4f4.2)') a(1,1),a(1,ny),a(nx,ny),a(nx,1)
CSMS$REMOVE END

CSMS$GLOBAL_INDEX (1,2) BEGIN
CSMS$INSERT       pts(1) = a(1,1)
CSMS$INSERT       pts(2) = a(1,ny)
CSMS$INSERT       pts(3) = a(nx,ny)
CSMS$INSERT       pts(4) = a(nx,1)
CSMS$GLOBAL_INDEX END

CSMS$REDUCE( pts, SUM)
CSMS$INSERT       write(6,'(4f4.2)') (pts(i),i=1,4)
```

CSMS$REMOVE

SYNTAX

CSMS$REMOVE BEGIN
CSMS$REMOVE END

Required Fields:
NONE

SEMANTICS

This directive removes all the code between the CSMS$REMOVE BEGIN and CSMS$REMOVE END. If the ppp command line option "--comment" is used, these code are simply commented out, otherwise the removed code will not appear in the translated output. Often this directive is used in conjunction with CSMS$INSERT to modify code segments that are undesirable, not parallelizable, or problematic for PPP.

LIMITATIONS

NONE

MESSAGES

ERRORS

CSMS$REMOVE END found at line <line #> without a matching CSMS$REMOVE BEGIN.

RELATED DIRECTIVES

CSMS$IGNORE
CSMS$INSERT

NOTES

EXAMPLES

See the example in CSMS$INSERT.

CSMS$TO_GLOBAL

SYNTAX

```
CSMS$TO_GLOBAL (
       < dim1,vars1>
     [,< dim2,vars2>]
     [,< dim3,vars3>] ) BEGIN

CSMS$TO_GLOBAL END
```

Required Fields:

dim1	(int)	decomposition dimension
vars1	(var)	comma separated list of variables that should be converted to global values in the given dimension

Optional Fields:

dim2	(int)	another data decomposition dimension
vars2	(var)	comma separated list of variables that should be converted to global values in the given dimension
dim3	(int)	another data decomposition dimension
vars3	(var)	comma separated list of variables that should be converted to global values in the given dimension

SEMANTICS

This directive converts variables,used as array indices, from processor local to their corresponding global value. Conversions are done on scalar entities only; no arrays or array references will be translated. For the direct translation of array indices, refer to the directive CSMS$GLOBAL_INDEX.

LIMITATIONS

1. No arrays or array indices will be translated. See csms$global_index to handle these translations.

MESSAGES

ERRORS

No data decomposition has been specified for this directive.

Either the data decomposition must be specified directly using the parameter decomp, or this directive must be within the scope of a csms$parallel region.

RELATED DIRECTIVES

CSMS$DISTRIBUTE

NOTES

1. This directive must be inside an enclosing parallel region.

EXAMPLES

Example 1: Global Indices Inside Loops

In this example, the loop index "j" has been translated to a local value by CSMS$PARALLEL. Since the if-conditional requires a global value of *j* in the 2nd decomposed dimension, CSMS$TO_GLOBAL(<2,j>) is used.

```
C sequential code with directives
CSMS$DISTRIBUTE(dh, <IM_WORLD>, <JM_WORLD>) BEGIN
      real x(IM_WORLD,JM_WORLD)
      real y(IM_WORLD,JM_WORLD)
      real z(IM_WORLD,JM_WORLD)
CSMS$DISTRIBUTE END

CSMS$PARALLEL(dh, <i>, <j>) BEGIN

      do j=1, JM_WORLD
        do i=1, IM_WORLD
CSMS$TO_GLOBAL(<2,j>) BEGIN
          if (j .gt. 5) then
CSMS$TO_GLOBAL END
            x(i,j) = y(i,j)
          else
            x(i,j) = z(i,j)
          endif
        enddo
      enddo
CSMS$PARALLEL END
```

CSMS$TO_LOCAL

SYNTAX

```
CSMS$TO_LOCAL (
        < dim1, vars1>
     [, < dim2, vars2>
     [, < dim3, vars3>]]
     [: SIZE]) BEGIN

CSMS$TO_LOCAL END
```

Required Fields:

dim1	(int)	decomposition dimension
vars1	(var)	comma separated list of variables that should be converted to local values

Optional Fields:

dim2	(int)	another data decomposition dimension
vars2	(var)	comma separated list of variables that should be converted to global values
dim3	(int)	another data decomposition dimension
vars3	(var)	comma separated list of variables that should be converted to global values
SIZE	(key)	to request the locally declared array size

SEMANTICS

This directive converts a variable to array size from global to processor local values. Conversions are done on scalar entities only; no arrays or array references will be not translated.

If the SIZE keyword is used, the variable is converted to the locally declared size of the given decomposed dimension. This modifier is often used in computing the local size of an array in the given decomposed dimension. See Example 2 for more details.

LIMITATIONS

1. Array indices are not be translated.

MESSAGES

ERRORS

No data decomposition has been specified for this directive.

Either the decomposition (decomp) must be specified explicitly, or this directive must be contained within an enclosing parallel region (CSMS$PARALLEL).

The only option permitted for this directive is: "SIZE"

NOTES

To_Local: An output array index was translated.

RELATED DIRECTIVES

CSMS$DISTRIBUTE

NOTES

1. The SIZE modifier is applied to all local variables listed in the directive.

2. This directive must be inside an enclosing parallel region.

EXAMPLES

Example 1: Local Computations are Required
Example 2: Using the SIZE Keyword
Example 3: Local Length Computations

Example 1: Local Computations are Required

In this example, we require local computations within a defined parallel region. The loop index "i" has been translated into a local variable by CSMS$PARALLEL. We use CSMS$GLOBAL to reference a global maximum. Once computed, this maximum must then be converted back to a local value for use as the decomposed array index: *ilocal*.

```
CSMS$DISTRIBUTE(decomp, m) BEGIN
      real x(m)
      real y(m)
CSMS$DISTRIBUTE END

CSMS$PARALLEL(decomp,i) BEGIN
      do i = 1, m
CSMS$TO_GLOBAL(<1,i>) BEGIN
CSMS$TO_LOCAL(<1, ilocal>) BEGIN
      ilocal = max(1,i-1)
CSMS$TO_LOCAL END
CSMS$TO_GLOBAL END
        x(i) = y(ilocal)
      enddo
CSMS$PARALLEL END
```

Example 2: Using the SIZE Keyword

The SIZE modifier to this directive will get the actual size of the locally defined array for the given decomposed dimension. This local size is often passed as an argument to a subroutine as illustrated in the example below.

In this case, we use the SIZE modifier of CSMS$TO_LOCAL to pass the declared local sizes of the arrays used in the subroutine *compute*. Since all references to the arrays "*a*" and "*b*" within subroutine *compute* are local no parallelization of this routine is required.

```
CSMS$DISTRIBUTE(decomp,nx,ny) BEGIN
      real a(nx,ny,kk)
      real b(nx,ny,kk)
CSMS$DISTRIBUTE END

CSMS$PARALLEL(decomp) BEGIN
CSMS$TO_LOCAL(<1,nx>,<2,ny>:size) BEGIN
      call compute(a,b,nx,ny)
CSMS$TO_LOCAL END
CSMS$PARALLEL END

c -----------------------------------------------------
      subroutine compute(a,b,m,n)

      integer m,n
      real a(m,n),b(m,n)

      do i=1,m
        do j=1,n

c           local computations

        enddo
      enddo
```

Example 3: Local Length Computations

In this example, we compute the local vector length of the first two decomposed dimensions and store it in *mn*. This value is used as a local upper bound for the collapsed *i* loop computation. IMPORTANT NOTE: The *i* loop does not require translation in this case because *fldin* is a local array; no parallel region (CSMS$PARALLEL) is required.

Note: Loop collapsing is an outdated hand-optimization that should be rewritten to their standard two or three dimensional form. Refer to Example 2 from CSMS$IGNORE for more details on the dangers of using collapsed loops.

```
      subroutine comp(m,n,kk)

CSMS$DISTRIBUTE(dh,m,n) BEGIN
      real fldin  (m,n,kk)
CSMS$DISTRIBUTE END

CSMS$PARALLEL(dh) BEGIN
CSMS$TO_LOCAL(<dh:1,m>,<dh:2,n>:SIZE) BEGIN
      mn = m * n
CSMS$TO_LOCAL END
CSMS$PARALLEL END

        do i=1,mn
          fldin (i,1,k)=0.0
        enddo
```

CSMS$TRANSFER

SYNTAX

Multi-line option

```
CSMS$TRANSFER (
      < SrcVar1, DestVar1 >
     [< SrcVar2, DestVar2>)  BEGIN

CSMS$TRANSFER END
```

Single-line option

```
CSMS$TRANSFER (
  < SrcVar1, DestVar1 >
 [< SrcVar2, DestVar2> )
```

Required Fields:

SrcVar1	(var)	variable to be transferred
DestVar1	(var)	destination variable to receive the transfer

Optional Fields:

SrcVar2	(var)	another variable to be transferred
DestVar2	(var)	destination variable to receive the transfer

SEMANTICS

This directive is used to move data arrays from a source to a destination decomposition. Source or destination arrays may be decomposed or non-decomposed. The source (SrcVar1) and destination (DestVar1) arrays must be the same type and rank but may have different sizes. Additional pairs of arrays do not need to be similar to any other pairs. However, array pairs that are the same type and rank as other pairs will be aggregated to reduce communication latency.

Transfers between decompositions should be used when the benefits of resolving data dependencies are greater than the communication costs associated with transferring between data decompositions.

Both single and multi-line transfer directives are

supported. Both generate the same SMS transfer code however, **the multi-line transfer will remove all statements between the begin and end transfer directives**. This option is useful when existing statements that copy between grids should be replaced. Example 1 illustrates this point.

LIMITATIONS

1. The source and destination variables must be different arrays.

2. For any dimension, data may be scrambled before or after the transfer but not both. This means you cannot transfer between packed data decompositions.

MESSAGES

ERROR

Vector lengths must be the same for Grid Ratio and Offset fields.

If grid ratio and offsets are used, they must contain the same number of dimensions.

Closing CSMS$TRANSFER must be supplied.

Nesting of transfer directives is not permitted. In addition, all transfers must be completed by the end of a program unit (function, subroutine or program).

RELATED DIRECTIVES

CSMS$DECLARE_DECOMP
CSMS$CREATE_DECOMP
CSMS$DISTRIBUTE

NOTES

EXAMPLES

Example 1: Spectral Model Transfers

Data transfers between grids are useful in spectral models for transformations between fourier and grid space. The multi-line transfer option is used because the parallel transfer replaced the original code that originally copied data between the array pairs <*fi, fj*> and <dfi, dfj>.

This code segment illustrates the computation being broken up due to data dependencies in both the 1st (*i*) and 2nd (*j*) dimensions. One dimensional data decompositions are defined for each dimension (*dhi* and *dhj*). CSMS$TRANSFER is used to transpose between these decompositions so that efficient parallel computations can be done.

```
      subroutine spectral_comp

CSMS$DISTRIBUTE(dhj,<jm>) BEGIN
      real fj(im,jm,km)
      real dfj(im,jm,km)
CSMS$DISTRIBUTE END

CSMS$DISTRIBUTE(dhi,<im>) BEGIN
      real fj(im,jm,km)
      real dfj(im,jm,km)
CSMS$DISTRIBUTE END

      do while (global_error .gt. tolerance)

        call compute_j_dep(fi,dfi)
CSMS$TRANSFER(<fi,fj>, <dfi, dfj> BEGIN
        do k= 1, km
          do j = 1, jm
            do i = 1, im
              fj(i,j,k) = fi(i,j,k)
              dfj(i,j,k) = dfi(i,j,k)
            enddo
          enddo
        enddo
CSMS$TRANSFER END

        call compute_i_dep(fj,dfj)
CSMS$TRANSFER(<fj,fi>, <dfj, dfi> BEGIN
        do k= 1, km
          do j = 1, jm
            do i = 1, im
              fi(i,j,k) = fj(i,j,k)
              dfi(i,j,k) = dfj(i,j,k)
            enddo
          enddo
        enddo
CSMS$TRANSFER END

c         more code ....
      enddo
```

Automatic Code Translations

In some cases PPP translates code without requiring directives. Currently, there are two areas in which automatic translation of the code are done: I/O, and program termination statements. This section will highlight these areas in detail.

Input / Output Statements

Code generation for I/O relies on CSMS$DISTRIBUTE directive to identify variables that are decomposed. Variables that are not contained within this directive or are in the directive but contain no matching variable tags will be treated as NON-DECOMPOSED entities. See the documentation on CSMS$DISTRIBUTE for further details on tagging.

Currently all Fortran read, write, open, close and print statements are automatically translated by PPP with the following EXCEPTIONS:

a. Implied do-loops for unformatted output.
   eg. write(5) (a(i),i=1,5)

b. Unformatted output of single elements of a decomposed arrays:
   eg. write(5) a(5)

c. end, err options are not handled
   eg. open(5,err=50,end=100,file='test.dat')

e. formatted input/ output strings longer than 256 characters
   (see example 1).

Each of these exceptions will generate a PPP error.

The slash format specifier is automatically removed from all I/O statements (eg. format(/,i5) will become: format(i5))

MESSAGES

ERRORS:

This real type is not currently supported.

*The only real types PPP currently supports are: real, real*4 and real*8*

Unformatted Implied-Do statements are not currently

supported.

*Due to the complexity in the code generation required, PPP does not currently translate these statements. (Eg. print *,(a(i),i=1,nx)).*

Note: PPP is able to translate formatted implied-do statements; however the output must be less than the 512 character length currently allowed.

Maximum format string length exceeded.

The number of characters permitted for formatted I/O cannot exceed 512.

Unformatted I/O of decomposed arrays is not currently supported.

The PPP handling of these statements has not been developed (see Example 1).

NOTES:

Unsupported array declaration.

PPP cannot handle assumed size arrays because some translations must know the number of elements (see Example 2).

Slash format descriptor removed

The SMS run-time system cannot currently handle the slash format descriptor so it is removed. If you wish to retain the original output formatting you will need to break the output into multiple statements.

EXAMPLES

Example 1: Decomposed array output
Example 2: Assumed size arrays

Example 1: Decomposed array output

SMS manages the I/O of decomposed arrays in parallel. PPP translation of read and write statements currently handles the output of full arrays only. A future upgrade will handle individual array elements.

In the example below, the first "write" statement will generate an error; all other statements are translated correctly by PPP.

```
CSMS$DISTRIBUTE(dh,nx,ny) BEGIN
      real a(nx,ny,nz)
CSMS$DISTRIBUTE END
      real levs(lm)

      read(5) a
      do k=1,lm
        write(4) a(1,1,k)
        print *,'pressure level = ',levs(k)
      enddo
```

Example 2: Assumed size arrays

This type of implied array declaration is not supported in PPP. A warning message is used because the statement may not affect code translations. In this example the handling of "string" does not require translation by PPP so the message can be ignored.

Note: In a future upgrade, PPP will output a warning message only when it DOES affect a code translation.

```
subroutine puts(proc,handle,string,status)
integer proc, handle, status
character*(*) string

print *,string
return
```