SMS Training Course

Dan Schaffer
NOAA Forecast Systems Laboratory
December 2004

Outline

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

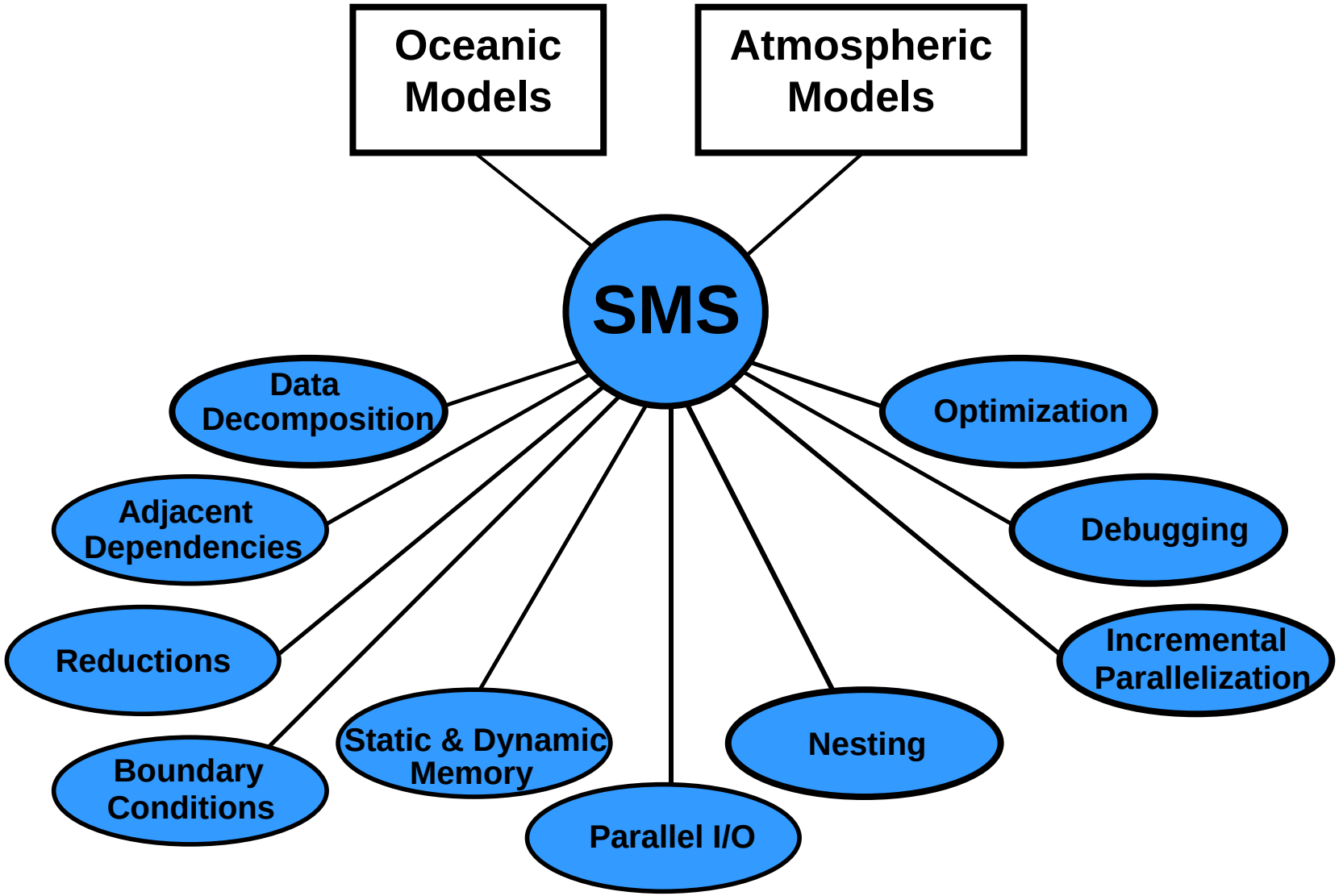
Overview

- Extensive documentation available at:

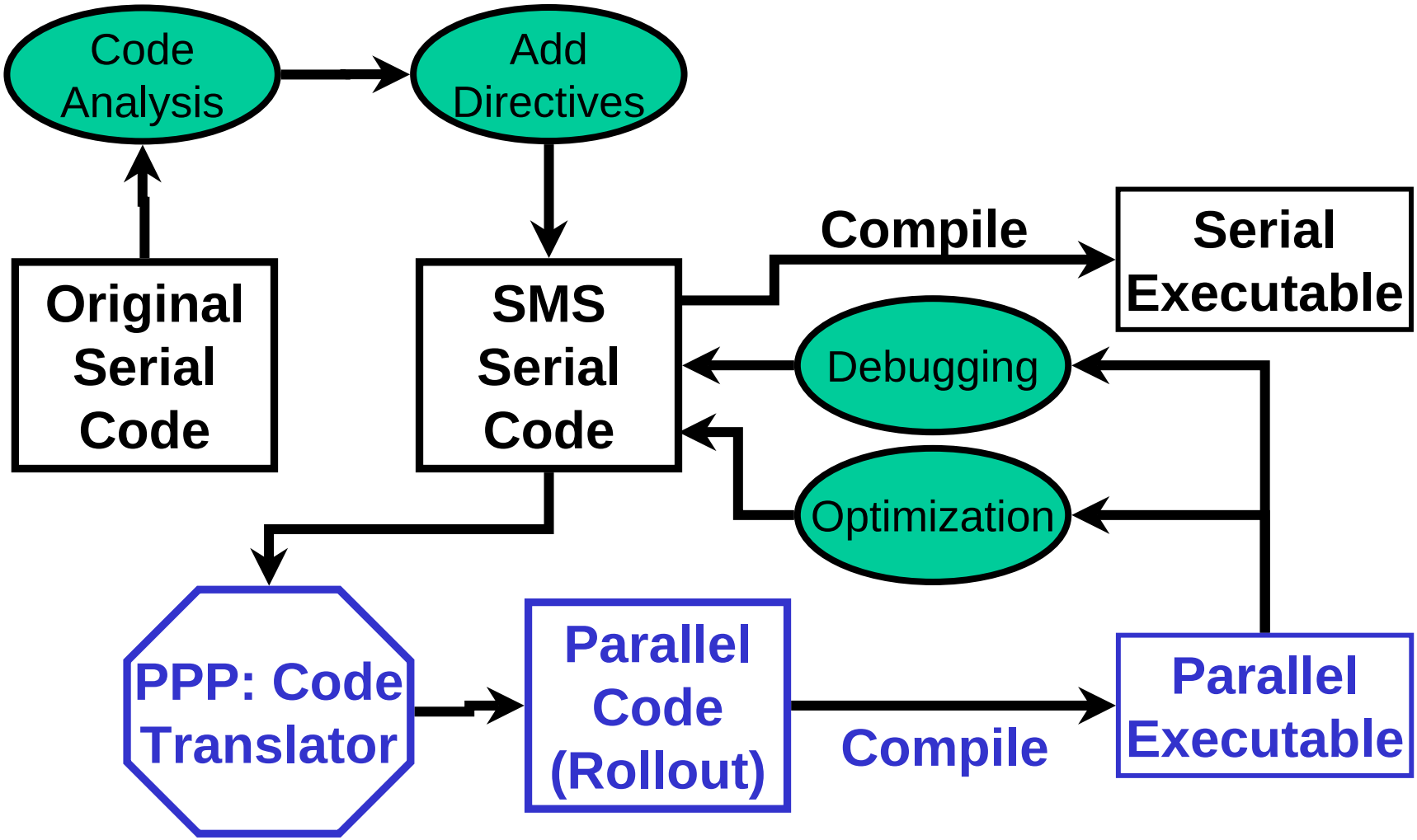
www-ad.fsl.noaa.gov/ac/sms.html

- Web-site includes a link to the SMS User's Guide. Latest public version is 2.8. I have provided a PDF version of version 2.9 by email to Mark

Overview


Oceanic Models
Atmospheric Models
SMS
Data Decomposition
Adjacent Dependencies
Reductions
Boundary Conditions
Static & Dynamic Memory
Parallel I/O
Nesting
Optimization
Debugging
Incremental Parallelization

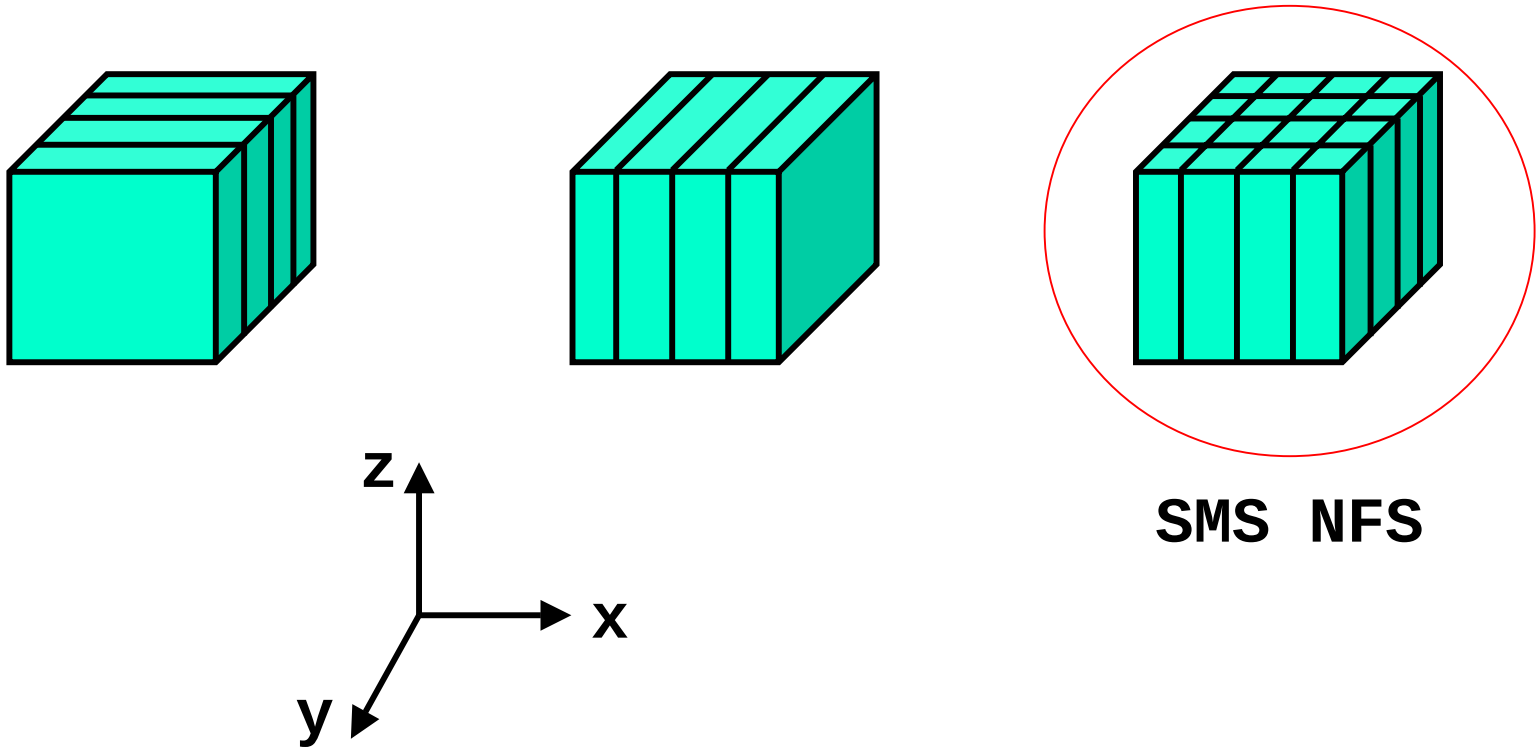
Code Parallelization With SMS



SMS Serial Code

```
        integer IM, i
        parameter(IM = 15)
CSMS$DECLARE_DECOMP(my_dh, 1)
CSMS$DISTRIBUTE(my_dh, <IM>) BEGIN
        real x(IM), y(IM), xsum
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP (my_dh, <IM>, <2>)
        open (10, file = 'x_in.dat', form='unformatted')
        read (10) x
CSMS$PARALLEL(my_dh, <i>) BEGIN
        do 100 i = 3, 13
          y(i) = x(i) - x(i-1) - x(i+1) - x(i-2) - x(i+2)
  100   continue
CSMS$HALO_UPDATE(y)
CSMS$CHECK_HALO(y, 'before loop 200')
        do 200 i = 3, 13
           x(i) = y(i) + y(i-1) + y(i+1) + y(i-2) + y(i+2)
  200   continue
CSMS$COMPARE_VAR(x, 'after loop 200')
        xsum = 0.0
        do 300 i = 1, 15
           xsum = xsum + x(i)
  300   continue
CSMS$REDUCE(xsum, SUM)
CSMS$PARALLEL END
        print *,'xsum = ',xsum
```

Typical FDA Decompositions



Overview

- SMS provides directives for:
  - Decompositions and Array bound, Loop Transformations
    - **`CSMS$DECLARE_DECOMP`**
    - **`CSMS$CREATE_DECOMP`**
    - **`CSMS$DISTRIBUTE`**
    - **`CSMS$PARALLEL`**

Overview

- SMS provides directives for:
  - Resolving adjacent dependencies
    - **`CSMS$HALO_UPDATE`**
  - Resolving global dependencies
    - **`CSMS$REDUCE`**
  - Partial parallelization
    - **`CSMS$SERIAL`**
    - **`CSMS$TO_LOCAL`**

Overview

- SMS provides directives for:
  - Boundary conditions
    - **`CSMS$GLOBAL_INDEX`**
  - Formatted output
    - **`CSMS$PRINT_MODE`**
  - Debugging
    - **`CSMS$COMPARE_VAR`**
    - **`CSMS$CHECK_HALO`**

Overview

- SMS provides directives for:
  - Nesting
    - **`CSMS$SET_NEST_LEVELS`**
    - **`CSMS$SET_TRANSFER_INTERPOLATE`**
    - **`CSMS$TRANSFER`**
  - Performance optimization
    - **`CSMS$HALO_COMP`**

Overview

- Supported machines
  - SGI O3K, Altix
  - SUN UltraSparc
  - IBM SP
  - DEC Alpha
  - HP Superdome
  - LINUX Clusters

Overview

- SMS accepts ANSI FORTRAN77 code plus FORTRAN90 syntax including:
  - **`real*8`**, **`integer*8`**, (etc.)
  - do/end do
  - while statement
  - namelists
  - Whole array assignments

```
real x(10)
x = 1.0
```

  - Include files

Overview

- More FORTRAN90 syntax:
  - Allocatable arrays
  - Pointers
  - F90 variable declarations

    ```
    integer, intent(in) :: i1(:)
    ```
  - Modules (partial support)
  - Fixed and free format

Overview

- SMS directives
  - Start with **`CSMS$`** **`(`**`!SMS$` **`for free format)`**
  - Case insensitive
  - May be no more than 72 characters per line
  - Can continue to another line: **`CSMS$>`**
  - May be inserted into FORTRAN include/module files
  - Can apply to multiple lines using begin/end syntax

Outline

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

Decomposition Directives

- Overview
  - **`CSMS$DECLARE_DECOMP`** makes user defined decomposition "name" visible
  - **`CSMS$CREATE_DECOMP`** initializes SMS data structures
  - **`CSMS$DISTRIBUTE`** specifies how each array is decomposed

Decomposition Directives

- **`CSMS$DECLARE_DECOMP`**
  - Functionality
    - A declarative (not executable) directive
    - Makes decomposition "name" visible to other directives
      - Declares internal SMS data structures
    - Tells SMS number of decomposed dimensions (at most 2 supported currently)
    - Only one needed per decomposition

Decomposition Directives

- **`CSMS$DECLARE_DECOMP`**
  - Syntax, dynamic memory

```
CSMS$DECLARE_DECOMP(dh, num_decomposed_dims)
```

  - Static memory discussed in the user's guide

Decomposition Directives

- **`CSMS$CREATE_DECOMP`**
  - Functionality
    - An “executable” directive that should be placed at the program beginning
    - Tells SMS
      - Global sizes of decomposition dimensions
      - Maximum halo widths (sizes of largest stencil)
    - Initializes internal SMS data structures
    - Generates process local memory layout

Decomposition Directives

- **CSMS$CREATE_DECOMP**
  - Syntax:

```
CSMS$CREATE_DECOMP(dh,
CSMS$>    <ddim1_global_size, ddim2_global_size>,
CSMS$>    <ddim1_halo_width,  ddim2_halo_width>)
```

ddim1_global_size is the global size of the 1st decomposed dimension

ddim2_global_size is the global size of the 2nd decomposed dimension

ddim1_halo_width should be set to the maximum stencil size over the entire model for the 1st decomposed dimension (also for ddim2_halo_wdith)

Decomposition Directives

- **`CSMS$DISTRIBUTE`**
  - Functionality
    - A declarative (not executable) directive
    - Specifies how arrays are decomposed
    - Replaces array declarations and allocations with process local equivalents
    - Associates arrays with decompositions:
      - Communication directives such as **`HALO_UPDATE`**, **`SERIAL`**, **`CHECK_HALO`**, **`COMPARE_VAR`**
      - Used for Fortran I/O
    - Many needed per decomposition

Decomposition Directives

- **CSMS$DISTRIBUTE**
  - Syntax:

```
CSMS$DISTRIBUTE(dh, <ddim1_tag>, <ddim2_tag>) BEGIN
  <serial code array declarations>
CSMS$DISTRIBUTE END
```

Simple Decomposition Example: Dynamic Memory

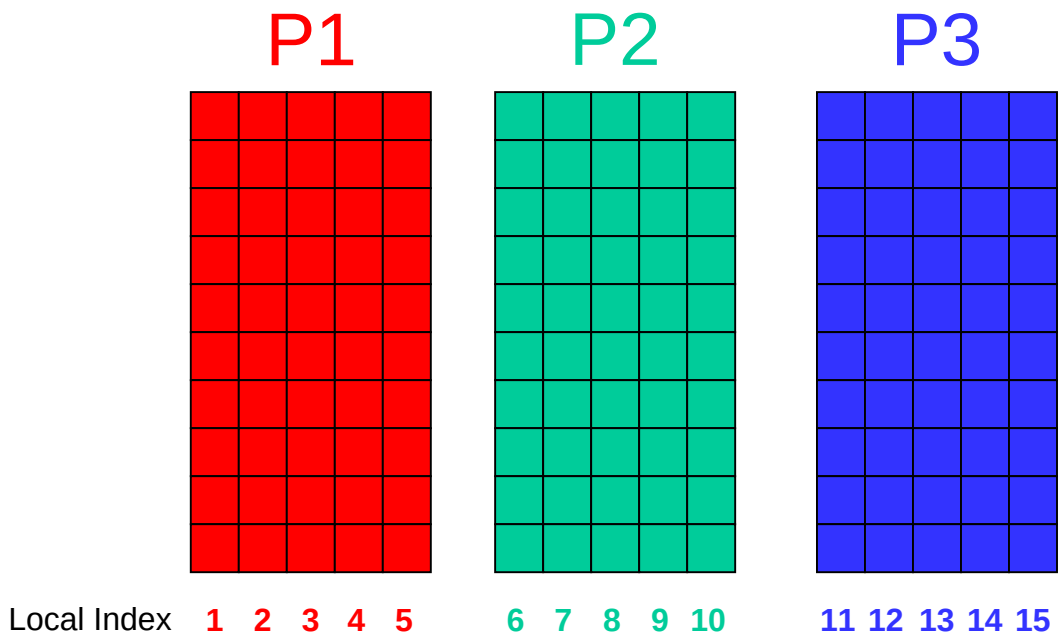
- 2D dynamic memory array decomposed in the first dimension

```
CSMS$DECLARE_DECOMP(my_dh, 1)

      integer, parameter :: IM = 15
      integer, parameter :: JM = 10

CSMS$DISTRIBUTE(my_dh, <IM>) BEGIN
      real :: x(IM,JM)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(my_dh, <IM>, <0>)
```

Simple Decomposition Example: Dynamic Memory

```
CSMS$DISTRIBUTE(my_dh, <IM>) BEGIN
C Rollout
c-SMS       real :: x(IM,JM)
      real x(local_start:local_end, JM)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(my_dh, <IM>, <0>)
      call SMS_DECOMP(…)
```

Simple Decomposition Example



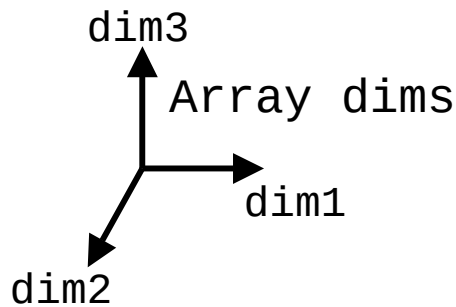
Serial Declaration: **`real x(15,10)`**

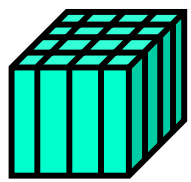
Parallel Declaration, P1: **`real x( 1: 5,10)`**
Parallel Declaration, P2: **`real x( 6:10,10)`**
Parallel Declaration, P3: **`real x(11:15,10)`**

2-D Decomposition Example

- Two decomposed dimensions, 3D array
  - 1st and 2nd dimensions of **x1** are decomposed using the 1st and 2nd dimensions of my_dh

```
CSMS$DECLARE_DECOMP(my_dh, 2)
CSMS$DISTRIBUTE(my_dh, <IM>, <JM>) begin
      real :: x1(IM,JM,KM)
CSMS$DISTRIBUTE end

C Rollout
      real x1(start1:end1, start2:end2, KM))
```





PE Layout

Loop Transformation Directive

- **`CSMS$PARALLEL`**
  - Tells SMS where/how loops need to be translated into their parallel equivalents
  - Defines a “parallel region”
    - Provides default decomposition “context” for other directives
  - User specifies loop index variables to guide translation

Loop Transformation Directive

- **CSMS$PARALLEL**
  - Syntax:

```
CSMS$PARALLEL(dh, <ddim1_loop_var,
CSMS$>             ddim2_loop_var>) BEGIN
   < CODE >
CSMSPARALLEL END
```

Loop Transformation

```
C “dh” is a 1D decomposition
CSMS$PARALLEL(dh, <i>) BEGIN
...
      do i=1,15
...
CSMS$PARALLEL END


C conceptual rollout
C P1 -->
      do i=1,5
C P2 -->
      do i=6,10
C P3 -->
      do i=11,15
```

Loop Transformation

```
C “dh” is a 1D decomposition
CSMS$PARALLEL(dh, <i>) BEGIN
...
      do i=14,15
...
CSMS$PARALLEL END


C conceptual rollout
C P1, P2 -->
      do NOTHING
C P3 -->
      do i=14,15
```

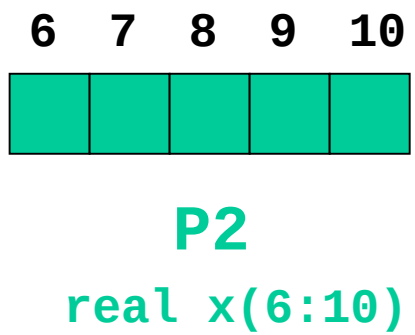
Boundary Conditions

- **`CSMS$GLOBAL_INDEX`**
  - Functionality
    - Ensures that assignment statements execute only on appropriate process(es)
  - Syntax:

```
CSMS$GLOBAL_INDEX(ddim1, ddim2)
```

Boundary Conditions

```
C Array x is decomposed
      real x(15)
CSMS$GLOBAL_INDEX(1) BEGIN
      x(8) = 0.0
CSMS$GLOBAL_INDEX END
```




```
C Conceptual rollout
C P1 -->
      do NOTHING
C P2 -->
      x(8) = 0.0
C P3 -->
      do NOTHING
```

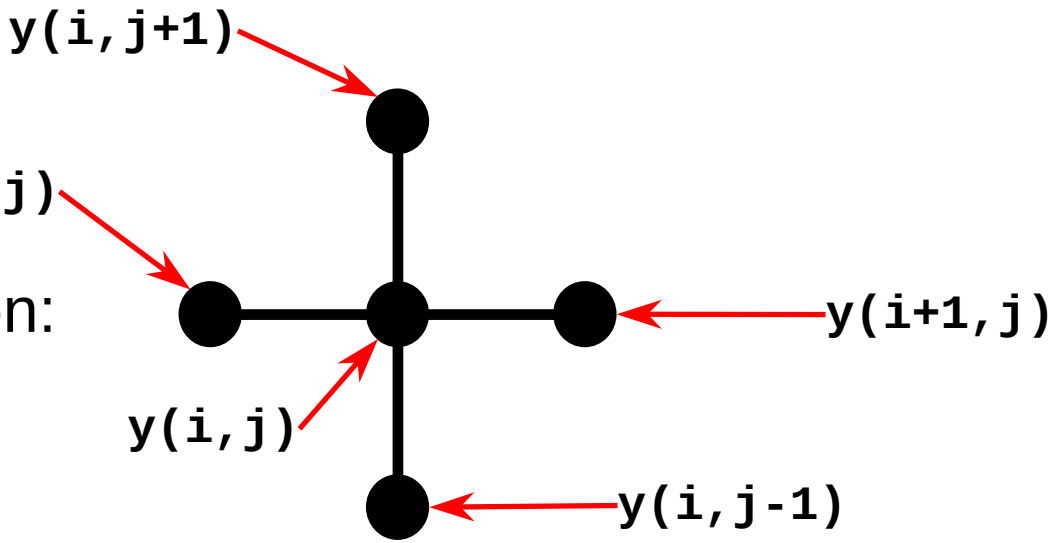
Outline

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

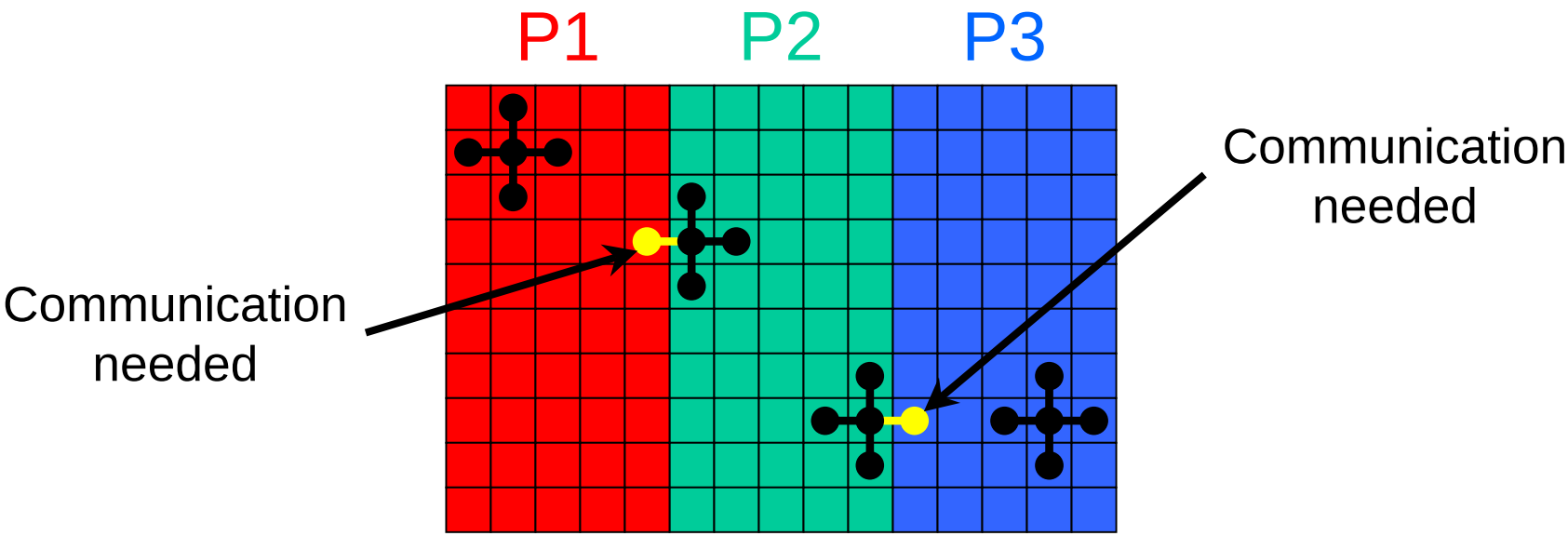
Adjacent Dependencies

```
x(i,j) = y(i,j) + y(i+1,j) + y(i-1,j) + y(i,j-1) + y(i,j+1)
```

"Stencil": `x(i,j)` depends on:

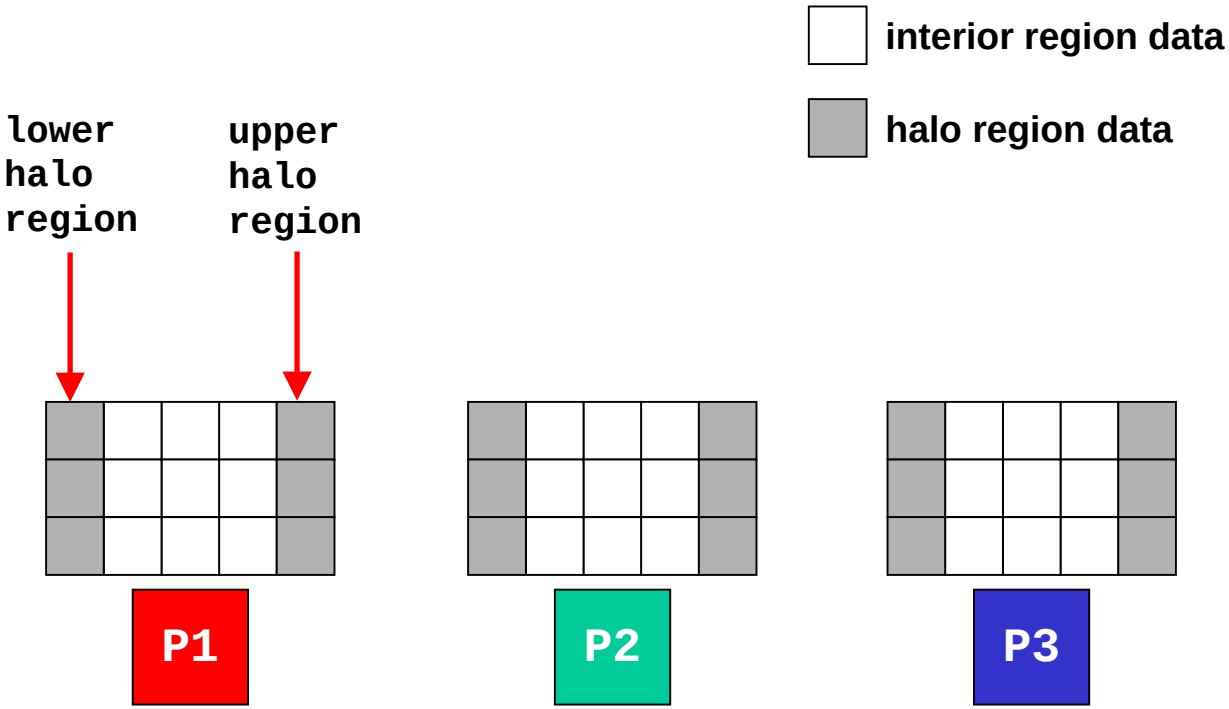


Off-Process References Due To Adjacent Dependencies



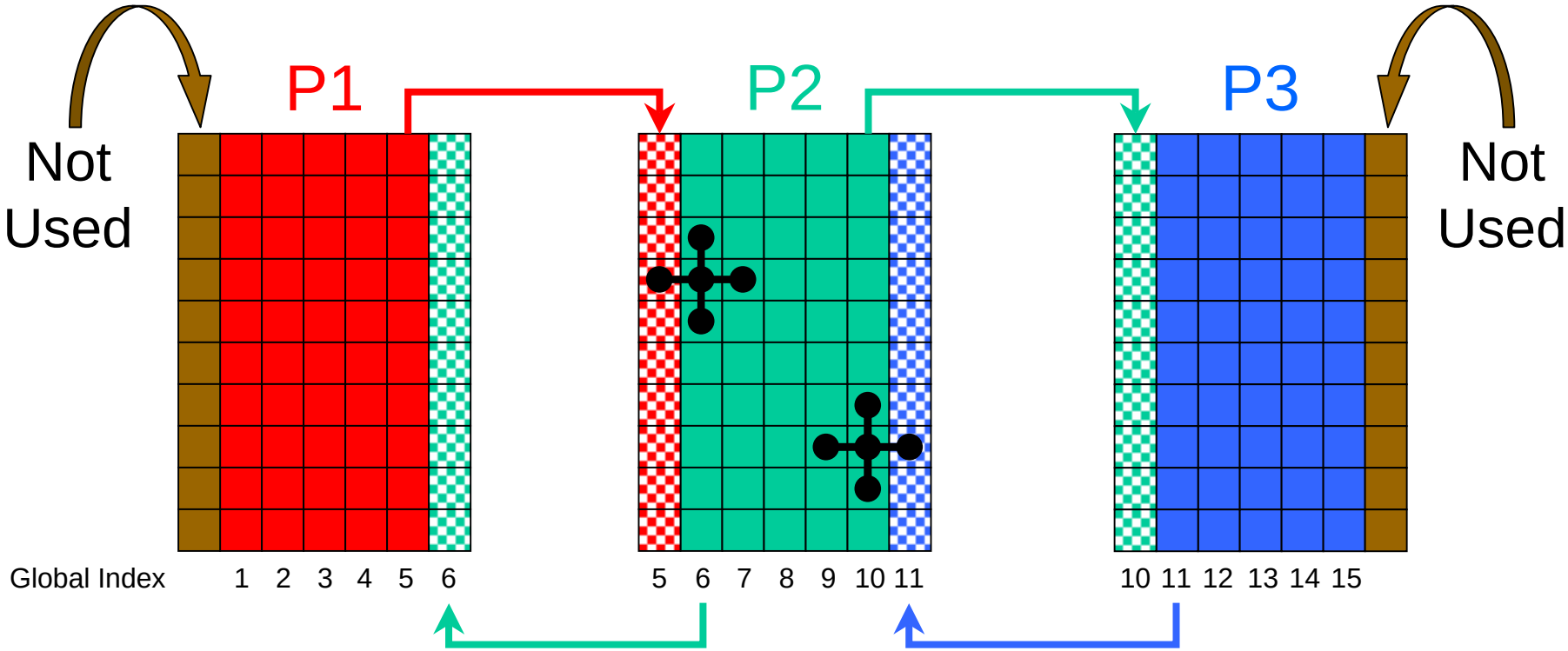

```
y(i,j) = x(i,j) + x(i+1,j) + x(i-1,j)
                + x(i,j+1) + x(i,j-1)
```

Some Terminology



Halo Region Update

Halo Thickness = 1



Halo Update Directive

- **`CSMS$HALO_UPDATE`**
  - Functionality
    - HALO_UPDATE data with neighboring processes to enable finite difference calculations
    - Sends appropriate data to neighbors
    - Stores received data in halo regions
    - HALO_UPDATEs are aggregated together to reduce latency: **`CSMS$HALO_UPDATE(x,y)`**
    - Other optimizations discussed later
  - Syntax:

```
CSMS$HALO_UPDATE(arr1, arr2, ...)
```

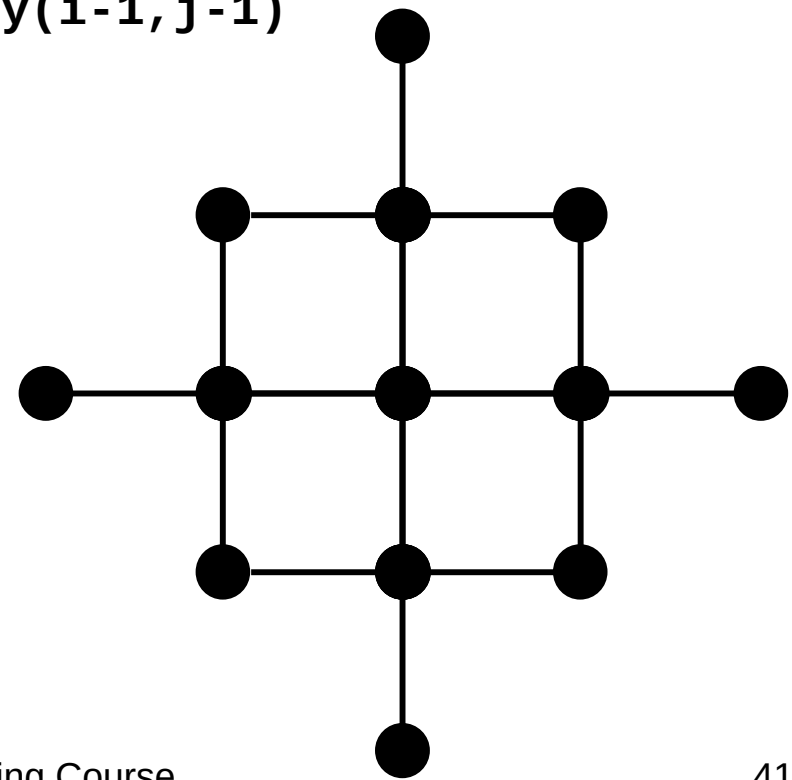
1-D Halo Update Example
Halo Thickness 1

```
      integer, parameter :: IM=15
      integer, parameter :: JM=10
CSMS$DECLARE_DECOMP(my_dh, 1)
CSMS$DISTRIBUTE(my_dh, <IM>) BEGIN
      real x(IM, JM)
CSMS$DISTRIBUTE END
CSMS$CREATE_DECOMP(my_dh, <IM>, <1>)
CSMS$HALO_UPDATE(x)  ! does the communication
C Rollout
      call SMS_HALO_UPDATE(x)
    do j = 1, JM
      do i = 1, IM
        y(i,j) = x(i,j) + x(i+1,j) + x(i-1,j)
                     + x(i,j+1) + x(i,j-1)
```

Adjacent Dependencies

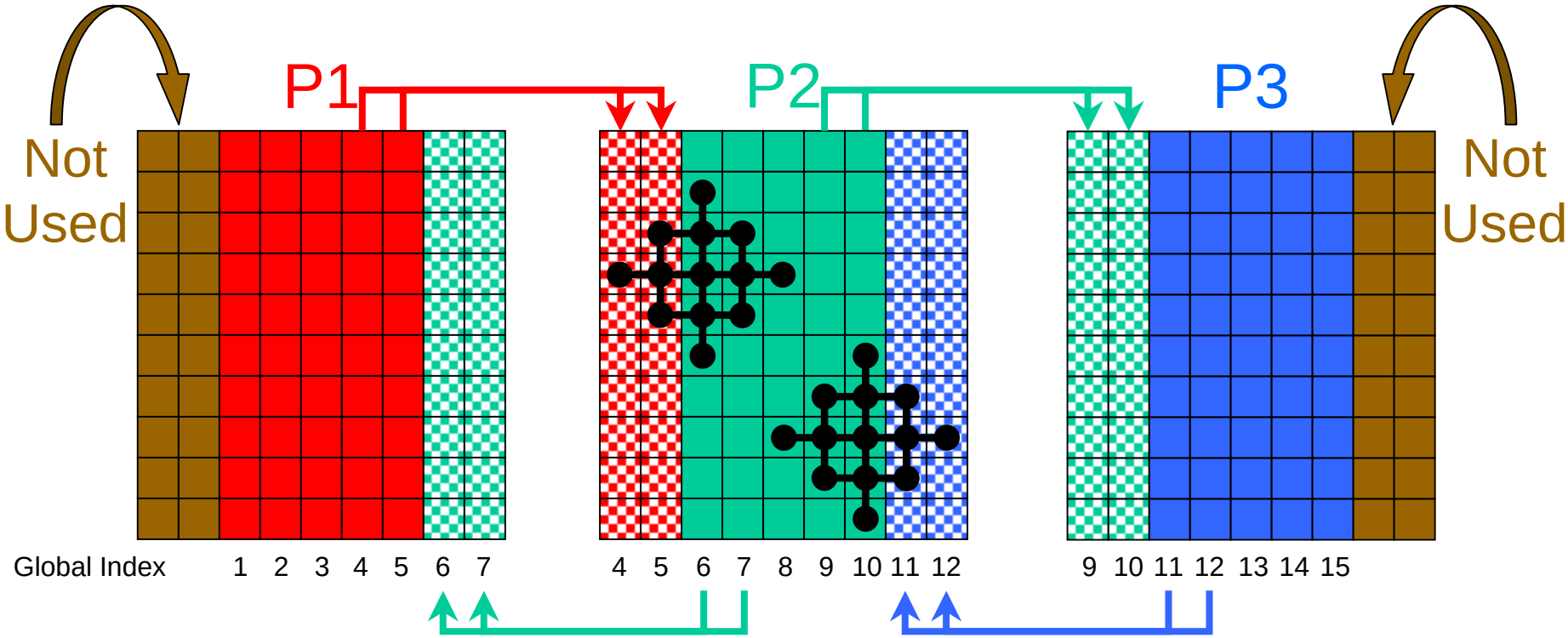
```
x(i,j) = y(i,j) + y(i+1,j) + y(i+2,j) + y(i,j+1) + y(i,j+2)
                + y(i-1,j) + y(i-2,j) + y(i,j-1) + y(i,j-2)
                + y(i+1,j+1) + y(i+1,j-1)
                + y(i-1,j+1) + y(i-1,j-1)
```

"Stencil": `x(i,j)` depends on:



Halo Region Update

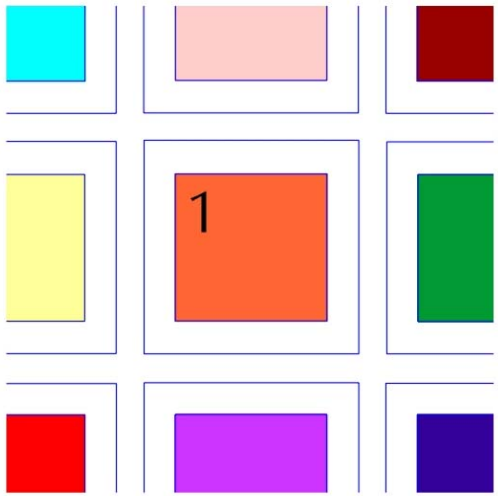
Halo Thickness = 2



1-D Halo Update Example
Halo Thickness 2

```
      integer, parameter :: IM=15
      integer, parameter :: JM=10
CSMS$DECLARE_DECOMP(my_dh, 1)
CSMS$DISTRIBUTE(my_dh, <IM>) BEGIN
      real x(IM, JM)
CSMS$DISTRIBUTE END
CSMS$CREATE_DECOMP(my_dh, <IM>, <2>)
CSMS$HALO_UPDATE(x)

      do j = 1, JM
        do i = 1, IM
          y(i,j) = x(i,j) + x(i+1,j) + x(i-2,j)
                      + x(i,j+1) + x(i,j-2)
```

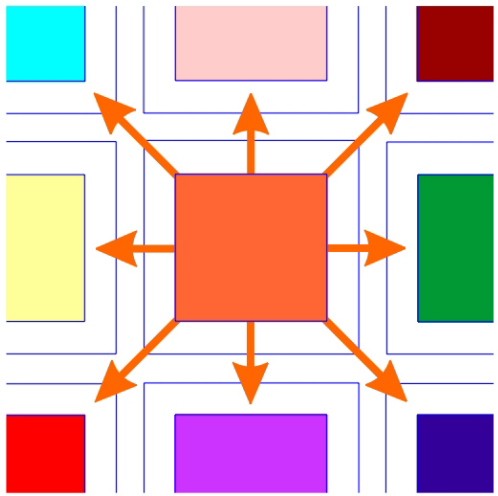
2D Halo Update



"Before"

SMS Directive: **CSMS$HALO_UPDATE**

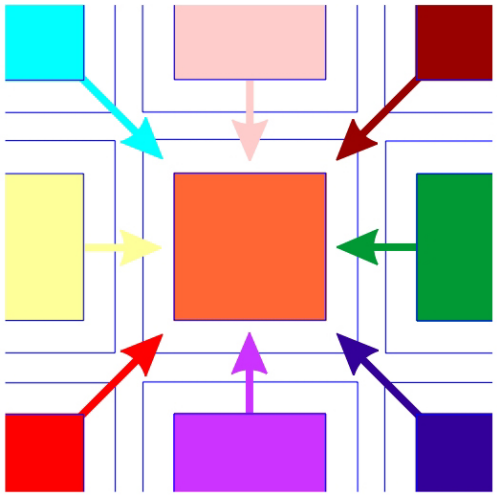
2D Halo Update



"Sends"

SMS Directive: **CSMS$HALO_UPDATE**

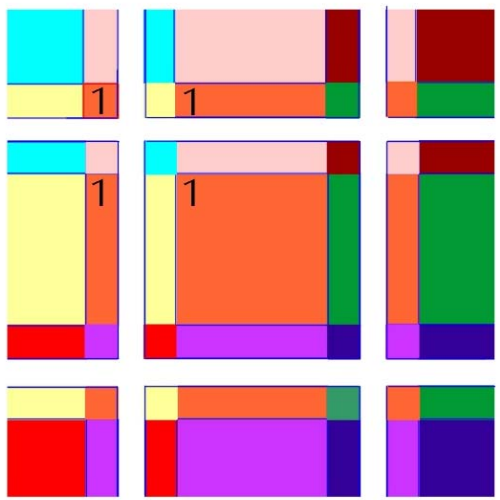
2D Halo Update



"Receives"

SMS Directive: **CSMS$HALO_UPDATE**

2D Halo Update



"After"

SMS Directive: **CSMS$HALO_UPDATE**

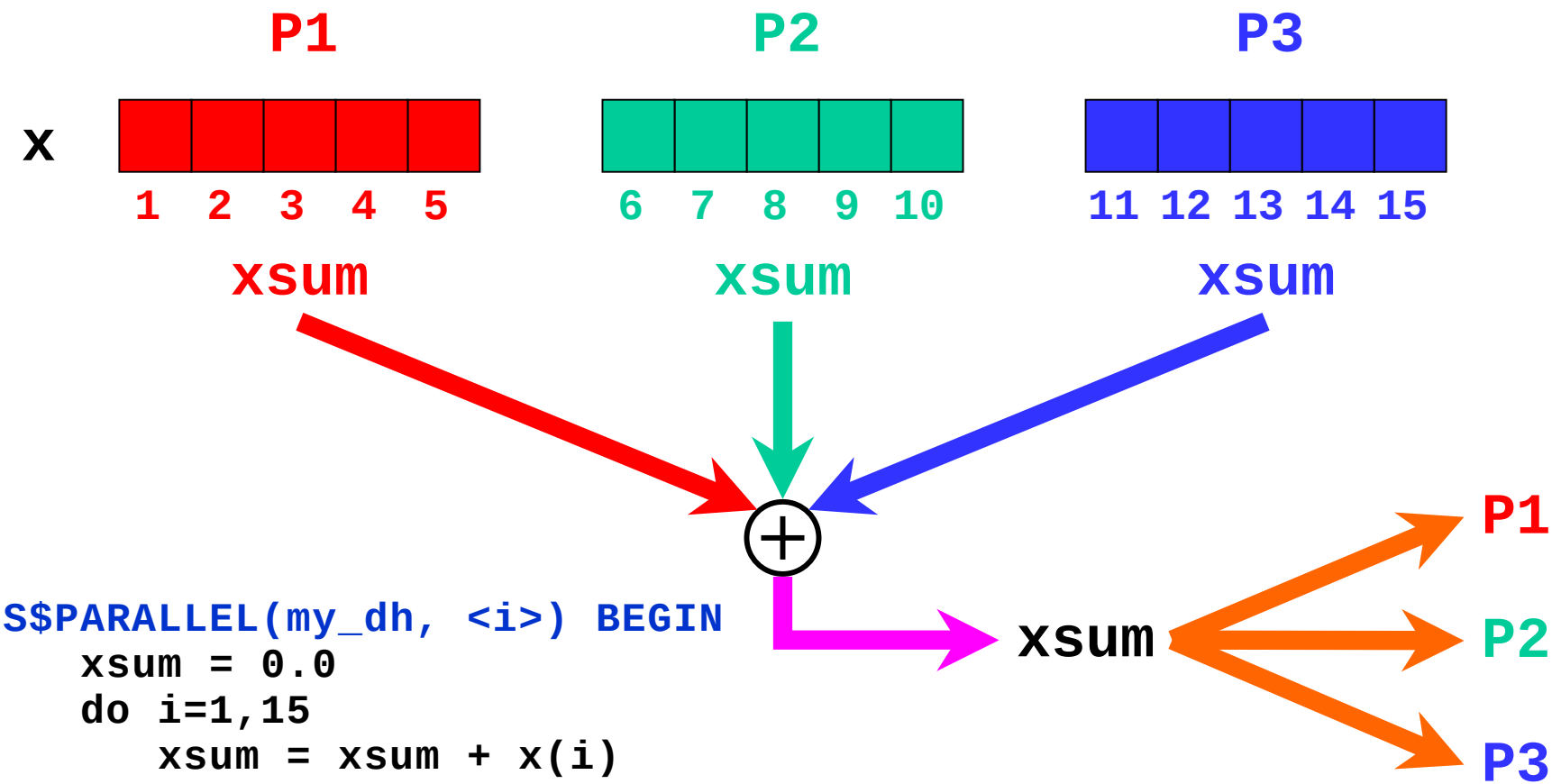
2-D Halo Update Example
Halo Thickness 2

```
      integer, parameter :: IM=15
      integer, parameter :: JM=10
CSMS$DECLARE_DECOMP(my_dh, 2)
CSMS$DISTRIBUTE(my_dh, <IM>, <JM>) BEGIN
      real x(IM, JM)
CSMS$DISTRIBUTE END
CSMS$PARALLEL(my_dh, <IM>, <JM>) BEGIN
CSMS$CREATE_DECOMP(my_dh, <IM,JM>, <2,2>)
CSMS$HALO_UPDATE(x)
     do j = 1, JM
       do i = 1, IM
         y(i,j) = x(i,j) + x(i+1,j) + x(i-2,j)
                + x(i,j+1) + x(i,j-2)
```

Global Dependencies

- When you need a global maximum, minimum or summation across all processes then use the directive **`CSMS$REDUCE`**

“Inexact” Summation




```
CSMS$PARALLEL(my_dh, <i>) BEGIN
      xsum = 0.0
      do i=1,15
         xsum = xsum + x(i)
      enddo
CSMS$REDUCE(xsum, SUM)
CSMS$PARALLEL END
```

Reduction Directives

- **`CSMS$REDUCE`**
  - Functionality
    - Determine MAX, MIN or SUM over all processes via communication
    - Reduction done "in place“
    - Reductions are aggregated together to reduce latency
  - General Syntax:

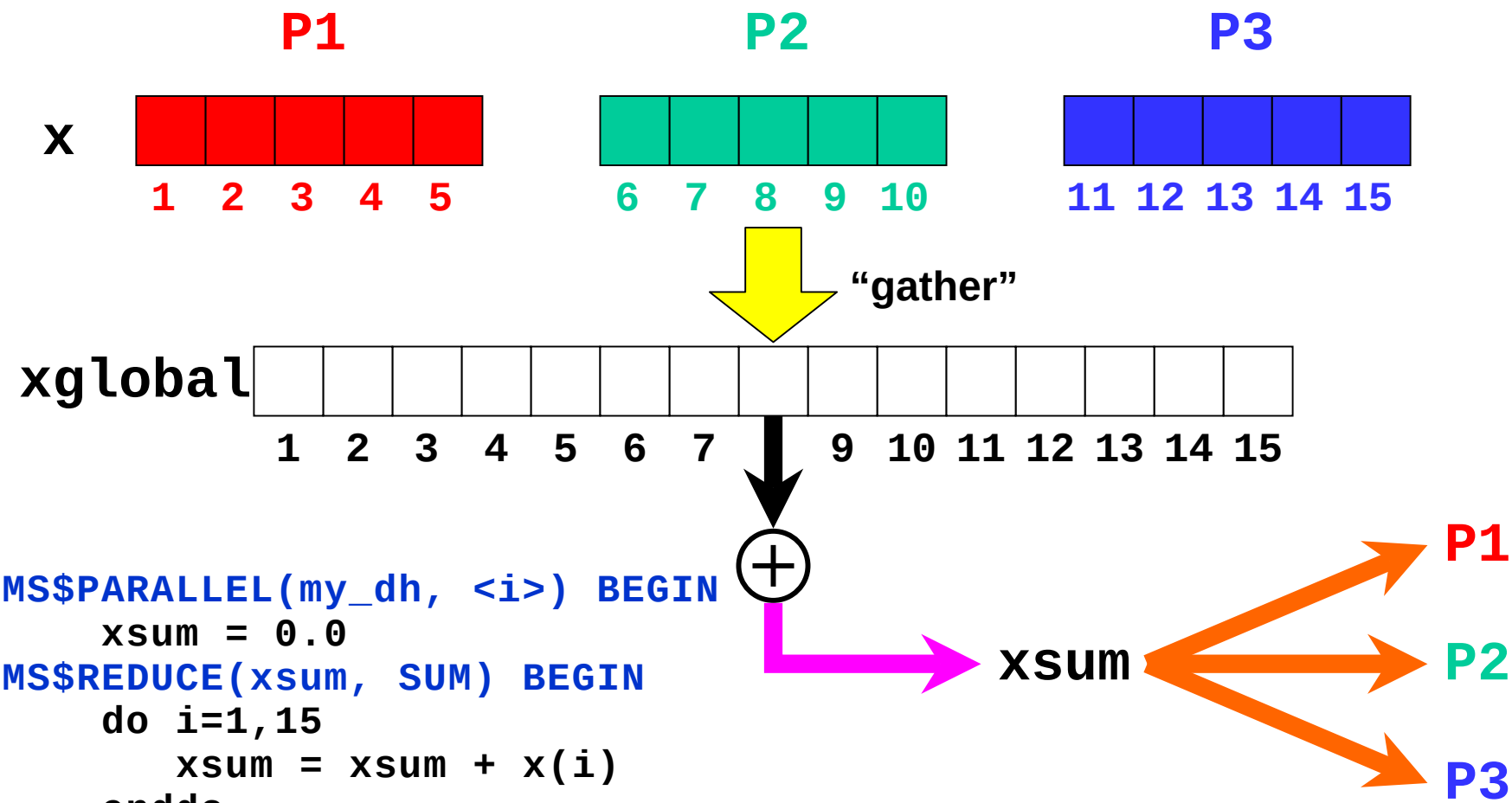
```
CSMS$REDUCE(non_decomposed_var1,non_decomposed_var2, OP)
```

  - “OP” = “SUM”, “MAX”, or “MIN”

Reductions (continued)

- Summation details
  - Compute "local sum" on each process
  - Sum "local sums" over processors to get final result
  - Results differ slightly for different numbers of processes with floating-point data
    - Floating-point addition is NOT associative
- Use **`BEGIN`**-**`END`** form of the **`CSMS$REDUCE`** directive to get “bitwise-exact” results

Bitwise-exact Summation




```
CSMS$PARALLEL(my_dh, <i>) BEGIN
      xsum = 0.0
CSMS$REDUCE(xsum, SUM) BEGIN
      do i=1,15
         xsum = xsum + x(i)
      enddo
CSMS$REDUCE END
CSMS$PARALLEL END
```

Bitwise-exact Summation

```
C Rollout
CSMS$PARALLEL(my_dh, <i>) BEGIN
      xsum = 0.0
CSMS$REDUCE(xsum, SUM) BEGIN
c-SMS      do i=1,15
c-SMS         xsum = xsum + x(i)
c-SMS      enddo
      call SMS_GATHER(x,x_global)
      do I = 1, 15
        xsum = xsum + x_global(i)
      end do
      call SMS_BROADCAST(xsum)
CSMS$REDUCE END
CSMS$PARALLEL END
```

Bitwise-exact Summation

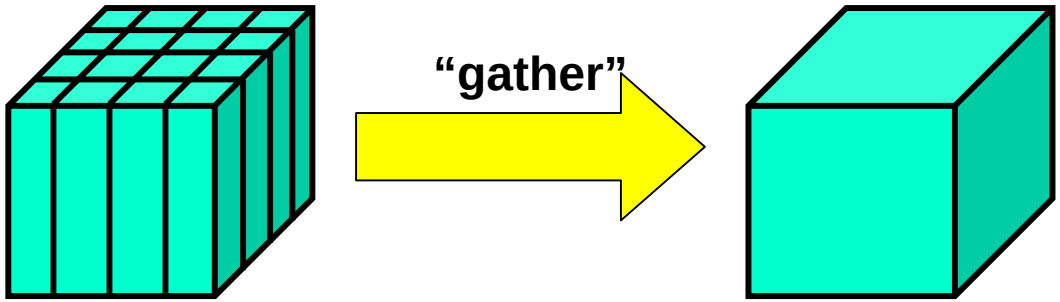
- **CSMS$REDUCE**
  - Functionality
    - Use this form when debugging because you can check if model solutions are bitwise exactly what you expect
    - Addresses floating point addition round-off error
    - Result is independent of number of processes
    - Performance penalty
    - Activated when environment variable SMS_BITWISE is EXACT
    - Otherwise, reverts to standard behavior
  - Syntax for bitwise exact form:

```
CSMS$REDUCE(non_decomposed_var, SUM) BEGIN
    CODE
CSMS$REDUCE END
```

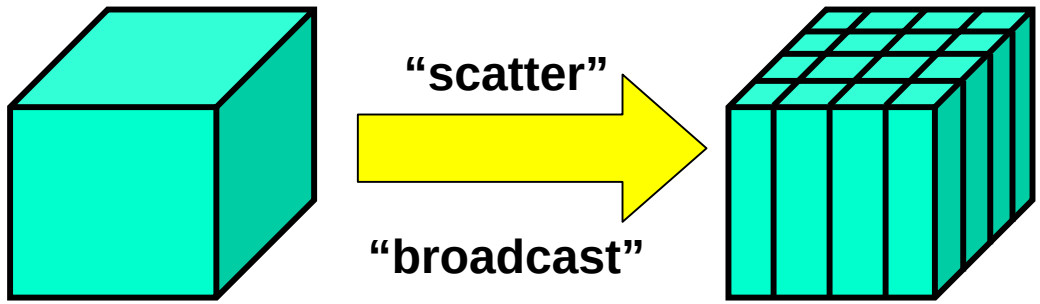
Outline

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

Partial Parallelization: SERIAL




```
CSMS$SERIAL BEGIN
      SERIAL CODE...
CSMS$SERIAL END
```



Partial Parallelization: SERIAL

- **`CSMS$SERIAL`**
  - Functionality
    - Code between BEGIN and END executed on one designated “root” process
    - Useful when
      - other SMS directives cannot easily be applied to a piece of serial code and/or
      - Efficient performance is not critical (i.e. during initialization)

Partial Parallelization: SERIAL

- **CSMS$SERIAL**
  - Syntax:

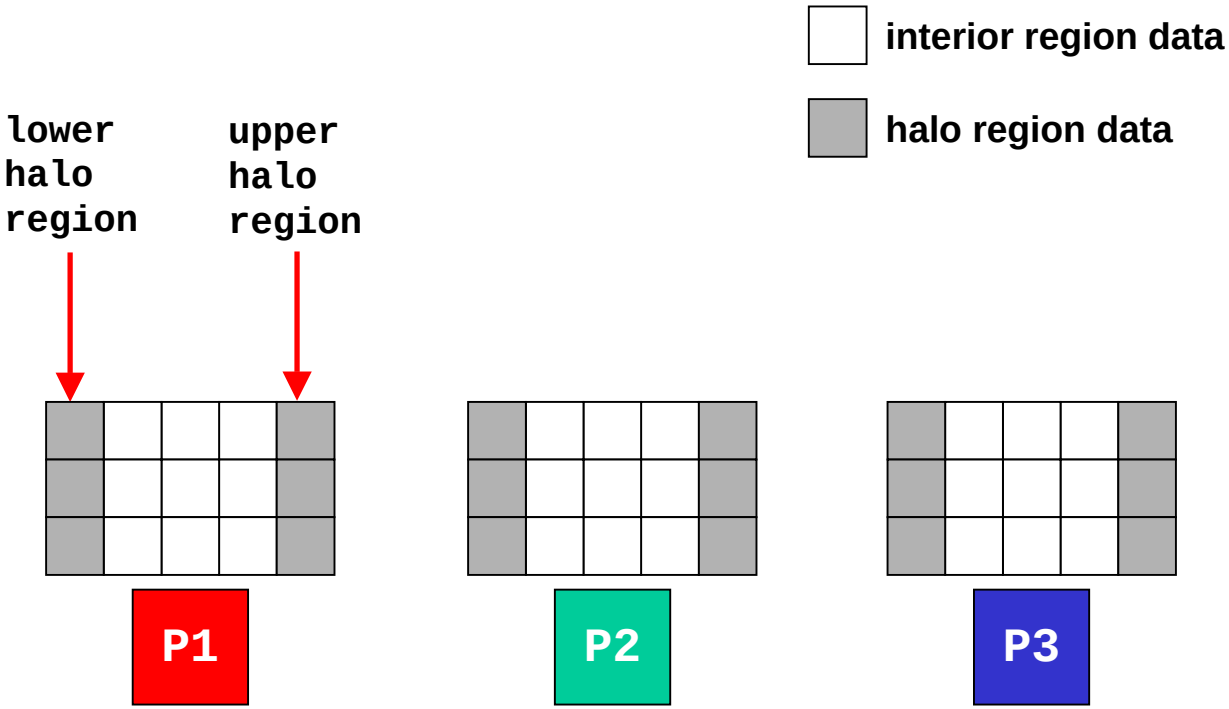
```
CSMS$SERIAL([communication specifications]) BEGIN
  Serial Code
CSMS$SERIAL END
```

  - Use “communication specifications” to improve performance
    - Discussed later…

Partial Parallelization: SERIAL

```
CSMS$DISTRIBUTE(my_dh, <IM>) BEGIN
        real x(IM)
        real y(IM)
CSMS$DISTRIBUTE END
        y = 2.0
CSMS$SERIAL BEGIN
        open(10, file='f0', form='unformatted')
        open(11, file='f1', form='unformatted')
        do ind = 4, 5
          read (10) x(ind)
          write(11) y(ind)
        end do
        close(10)
        close(11)
CSMS$SERIAL END
```

Terminology Reminder



Partial Parallelization: TO_LOCAL(interior)

- In NFS and other models there are subroutines which are:
  - Hard to parallelize
    - In NFS we want a 2-dimensional decomposition
    - But this decomposition cannot be applied to collapsed loops and arrays such as in CUPKUO
  - Have no dependencies (“embarrassingly parallel”)

Partial Parallelization: TO_LOCAL(interior)

- In such cases, just the interior (no halo points) of decomposed arrays can be passed to these subroutines
- The correct process local array sizes must also be passed
- CSMS$TO_LOCAL with the “interior” option handles this case

Partial Parallelization: TO_LOCAL(interior)

```
program to_local
integer, parameter :: IM = 15
integer, parameter :: JM = 10
integer IMJM

real x(IM,JM)

IMJM = IM*JM
call EMBARRASSINGLY_PARALLEL(X, IMJM)

end

subroutine EMBARRASSINGLY_PARALLEL(X, IMJM)
integer IMJM
real X(IMJM)
do I = 1, IMJM
  X(I) = 0.0
end do
```

Partial Parallelization: TO_LOCAL(interior)

```
      program to_local
      integer, parameter :: IM = 15
      integer, parameter :: JM = 10
      integer IMJM
CSMS$DECLARE_DECOMP(my_dh, 2)
CSMS$DISTRIBUTE(my_dh, <IM>, <JM>) BEGIN
      real x(IM,JM)
CSMS$DISTRIBUTE END
CSMS$TO_LOCAL(my_dh : <1,IM:interior>, <2,JM:interior>) BEGIN
      IMJM = IM*JM
      call EMBARRASSINGLY_PARALLEL(X, IMJM)
CSMS$TO_LOCAL END
      end

      subroutine EMBARRASSINGLY_PARALLEL(X, IMJM)
      integer IMJM
      real X(IMJM)
      do I = 1, IMJM
        X(I) = 0.0
      end do
```

Partial Parallelization: TO_LOCAL(interior)

```
Rollout

CSMS$TO_LOCAL(my_dh : <1,IM:interior>, <2,JM:interior>) BEGIN

      IMJM = DECOMP_DIM_1_LOCAL_SIZE * DECOMP_DIM_2_LOCAL_SIZE

      call EMBARRASSINGLY_PARALLEL(
     & X(DDIM1_INTERIOR_START:DDIM1_INTERIOR_END,
     &   DDIM2_INTERIOR_START:DDIM2_INTERIOR_END), IMJM)

CSMS$TO_LOCAL END

      end
```

Outline

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

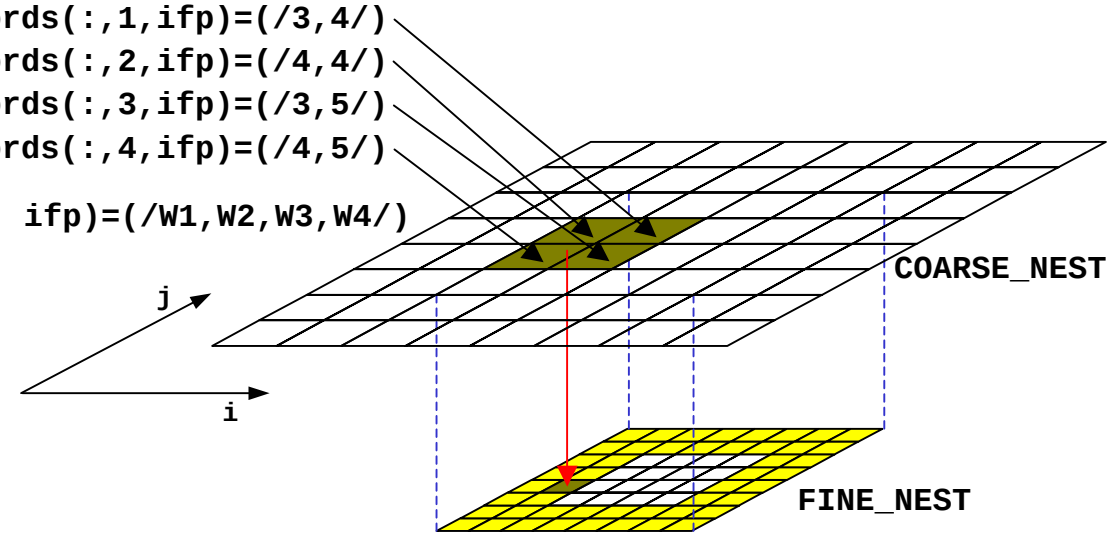
Nesting Overview

- SMS handles nested models for fixed nest locations (no moving nests currently)
- Nesting is supported in both directions (parent to child and child to parent) for multiple nests
- It is assumed that nested grid interpolation is linear, i.e. child grid points are computed by a linear combination of parent grid points
- The serial code must implement these linear combinations in a particular way shown a bit later

Nesting Interpolation Example

```
FINE_COORDS(2,4)=W1*COARSE_COORDS(3,4) + W2*COARSE_COORDS(4,4) +
                 W3*COARSE_COORDS(3,5) + W4*COARSE_COORDS(4,5)
```

```
NUM_FINE_POINTS=48
MAX_STENCIL_POINTS=4
coarse_coords(:,1,ifp)=(/3,4/)
coarse_coords(:,2,ifp)=(/4,4/)
corase_coords(:,3,ifp)=(/3,5/)
coarse_coords(:,4,ifp)=(/4,5/)
coarse_weights(:, ifp)=(/W1,W2,W3,W4/)
```




```
fine_coords(:,ifp)=(/2,4/)
```

Nesting Serial Code - Setup

```
        coarse_weights = 0.0

        ifp = 0
        do j=1,jm_fine
          do i=1,im_fine
            ifp = ifp + 1
            fine_coords(1,ifp) = i
            fine_coords(2,ifp) = j

C Set up coarse points needed for interpolation...
            coarse_coords(1,1,ifp) = ioffset + ((i-1)/3)
            coarse_coords(2,1,ifp) = joffset + ((j-1)/3)
            .
            .
C Compute weights...
            if ((mod(i,3).eq.1).and.(mod(j,3).eq.1)) then
              coarse_weights(1,ifp) = 0.67
            endif

            if ((mod(i,3).eq.2).and.(mod(j,3).eq.1)) then
            .
            .

          end do
        end do

        num_fine_points   = ifp
```

Nesting Serial Code - Interpolate

```
 do k = 1, km
   do ifp=1,num_fine_points
     nest_fine(fine_coords(1,ifp),fine_coords(2,ifp),k)= 0.0

     do icp=1,max_coarse_points ! 4 for bi-linear interpolation

       if (coarse_weights(icp,ifp).ne.0.0) then
         nest_fine(fine_coords(1,ifp),fine_coords(2,ifp),k)=
&        nest_fine(fine_coords(1,ifp),fine_coords(2,ifp),k)+
&          (coarse_weights(icp,ifp) *
&           nest_coarse(coarse_coords(1,icp,ifp),
&             coarse_coords(2,icp,ifp),k))
       endif
     end do
   end do
 end do
```

Nesting Directives

- **CSMS$DECLARE_DECOMP(grid_dh(3),2)**
  - **Specify 3 nests for the decomposition**
  - **This is a 2-dimensional decomposition as we have seen before**

- **CSMS$CREATE_DECOMP(grid_dh(1),<im1,jm1>, <1,1>)**
- **CSMS$CREATE_DECOMP(grid_dh(2),<im2,jm2>, <1,1>)**
- **CSMS$CREATE_DECOMP(grid_dh(3),<im3,jm3>, <1,1>)**
  - **Create the decompositions with the appropriate nest sizes**

Nesting Directives

- **CSMS$SET_NEST_LEVELS(2,3)**
  - **In models such as NFS, a single subroutine handles nesting interpolation between any pair of nests**
  - **SMS provides a means for directives to refer to parameterized (instead of absolute) nest levels**
  - **This example says that the parameterized nest levels are 2 and 3**

Nesting Directives

```
CSMS$DISTRIBUTE(grid_dh($1), <im_coarse>, <jm_coarse>) begin
   real x_coarse(im_coarse,jm_coarse)
```

- **In this example, x_coarse is distributed based on the decomposition for the 1st of the parameterized nest levels (nest level 2)**

```
CSMS$DISTRIBUTE(grid_dh($2), <im_fine>, <jm_fine>) begin
   real x_fine(im_fine,jm_fine)
```

- **x_fine is distributed based on the decomposition for the 2nd of the parameterized nest levels (nest level 3)**

Nesting Directives

```
CSMS$PARALLEL(grid_dh($2), <i>, <j>) begin
   do j = 1, jm_fine
     do i = 1, im_fine
```

- **The loops here are translated based on the 2nd of the parameterized nest levels (nest level 3)**

Nesting Directives

```
csms$set_transfer_interpolation( grid_dh($1), grid_dh($2), 2,
csms$> num_fine_points, fine_coords, max_coarse_points,
csms$> coarse_coords, coarse_weights, INTERP )
```

- This directive stores away inside SMS the indices, weights and communication patterns needed to interpolate nest grid points
- grid_dh($1) and grid_dh($2) are the (parameterized) decompositions
- The number “2” indicates that this is a 2-dimensional interpolation (although it can be executed symmetrically over multiple levels as is shown in the user’s guide)
- num_fine_points are the total number of fine points that have to be computed

Nesting Directives

```
csms$set_transfer_interpolation( grid_dh($1), grid_dh($2), 2,
csms$> num_fine_points, fine_coords, max_coarse_points,
csms$> coarse_coords, coarse_weights, INTERP )
```

- max_coarse_points (4 in our example) is the number of coarse points used to compute each fine point
- fine_coords is the array of coordinates of fine grid points that are to be computed
- coarse_coords is the array of coordinates of the coarse grid used to compute the fine points
- coarse_weights are the weights used in the linear combination
- INTERP is an SMS handle that will be referenced by the actual TRANSFER directive (see below)

Nesting Directives

```
CSMS$TRANSFER( <t_coarse, t_fine : INTERP     >
CSMS$>                    <q_coarse,q_fine: INTERP>) BEGIN
c-SMS    Serial Interpolation code here
     call sms_transfer_interpolate(t_coarse, …)
CSMS$TRANSFER END
```

- PPP replaces the serial interpolation code with SMS library calls that interpolate nest_coarse to nest_fine using the stored away interpolation scheme

Using SMS

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

SMS Unformatted I/O

- No directives required except :
  - Non-FORTRAN I/O such as DMS, use **`CSMS$SERIAL`**

SMS Formatted I/O

- Handled automatically except:
  - Special “print” behaviors (handle with **`CSMS$PRINT_MODE`**)

Printing

- What does parallel print mean?

```
print *,'hello'
>> hello
>> hellohellohello
>> hhelheellollloo
>> hello
   hello
   hello
```

Printing

- SMS supports several print behaviors (print modes)
- Default print mode (“ROOT”) mimics serial code
  - One printed message per print statement
- Use **`CSMS$PRINT_MODE`** to select other print modes
  - “ASYNC”
    - Data-dependent prints

Data-Dependent Printing (Asynchronous Print Mode)

- "ASYNC" print mode
  - Every process prints
  - No ordering or synchronization
    - Print order may vary from one run to the next
  - Use for prints that depend on decomposed data

```
CSMS$PARALLEL(dh, <i>) BEGIN
      do i = 1, IM
        if (x(i) .eq. 0.0) then
CSMS$PRINT_MODE(async) BEGIN
          print *, 'ERROR:  x(i) is 0!'
CSMS$PRINT_MODE END
```

  - With default print mode, error messages from processes other than the "ROOT" process will not be printed

Using SMS

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

Using PPP

- ppp (Command-line options) file
  - Source file

    ppp *source_file.f*

    generates

    source_file_sms.f

PPP options

- **--includepath=path**
  - **Tell PPP where to look for include files**
  - **Analogous to Fortran -I option**
  - **Required: --includepath=$SMS/include**

- **$SMS is an environment variable that refers to the directory containing the SMS library you are using**

PPP options

- **Suppose you put your `DECLARE_DECOMP` directive into module “my_module” and want it to be visible to all files that are run through PPP. The option –Fmodule=my_module will cause PPP to add “use my_module” to any subroutine processed by the PPP command**

PPP options

```
[decomp.f]
      module decomp_module
CSMS$DECLARE_DECOMP(my_dh, 1)
      end module decomp_module

[sub.f]
      subroutine sub
      use my_module
CSMS$DISTRIBUTE(my_dh, <IM>) begin
      real x(IM)
CSMS$DISTRIBUTE end
       return
       end
```

```
ppp decomp_module.f  --> decomp_module_sms.f

ppp --Fmodule=decomp_module sub.f --> sub_sms.f
```

PPP options

- --r8
  - Tell PPP to assume all variables of type real are 8 bytes
  - Variables declared with an explicit kind (i.e. real(kind=4) or real*4) will be left as is.

PPP options

--comment

- Lines of code that PPP would delete are, instead, left in as comments

```
csms$parallel(my_dh, <i>) begin
c-SMS      do i = 1,IM
      do i = start1,end1
         u(i) = 0.0
      end do
csms$parallel end
```

--CompareOnly

- Tells PPP to only translate COMPARE_VAR directives (discussed later)

Compiling, Linking

- Compiling

```
pgf90 –c –I $SMS/include sub1_sms.f
pgf90 –c –I $SMS/include sub2_sms.f
```

- Linking

```
mpif90 –o par_code sub1_sms.o
  sub2_sms.o –L $SMS/lib  -lsms
```

Running

- Running

  smsRun -np 2 par_code

- **par_code can be**
  - **An SMS binary executable**
  - **A shell script that sets some environment variables and then executes an SMS binary executable. For example:**

```
[par_code]
setenv CWBNFSL N04011212serial
sms_binary_nfs.exe
```

Using SMS

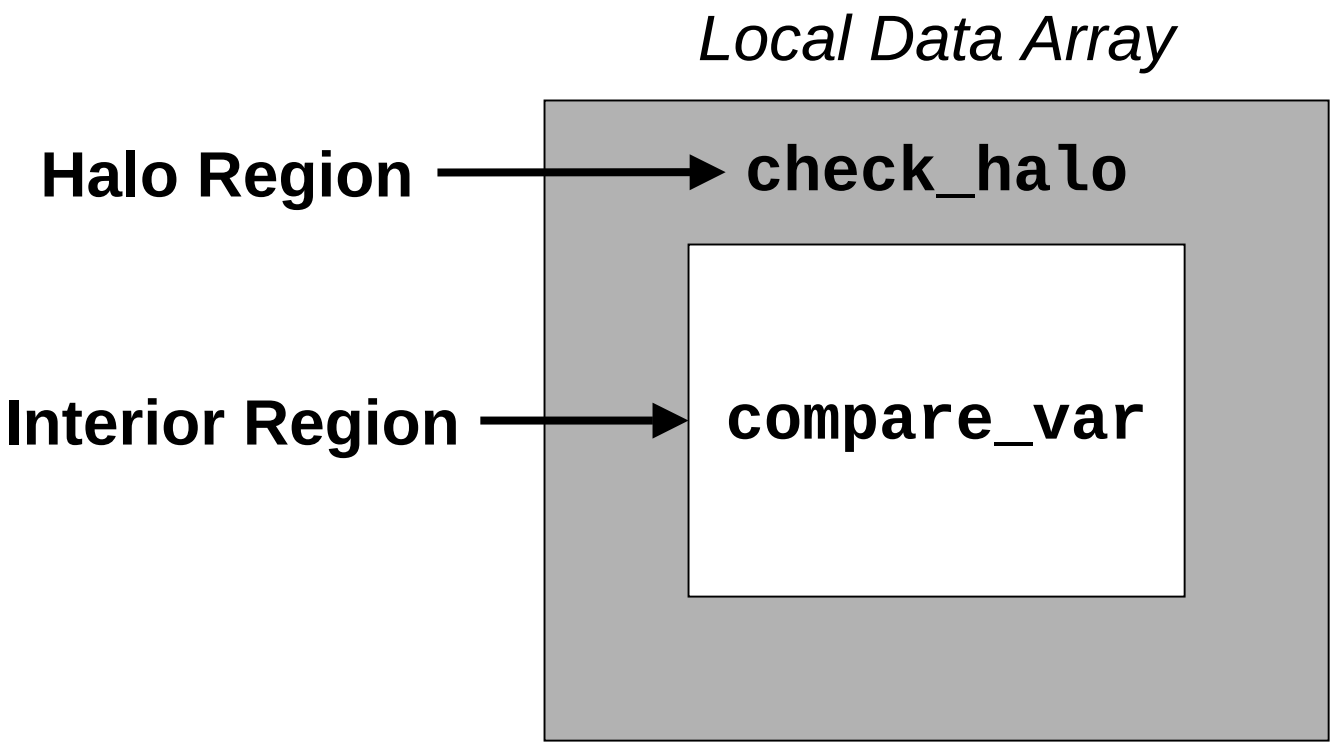
- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

Debugging Procedure

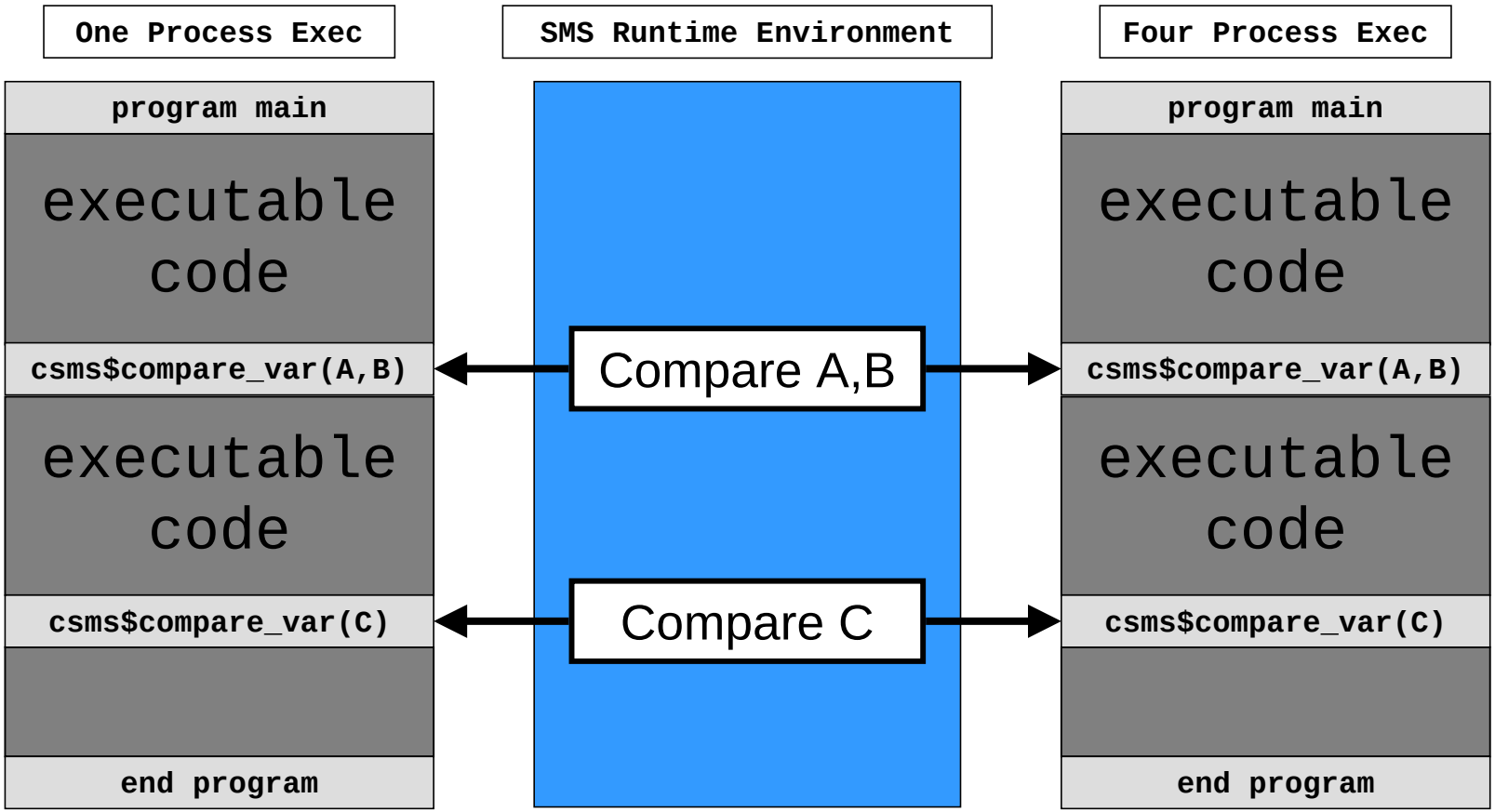
- Establish serial baselines
- Verify parallel reproducibility
  - One process
  - Multiple processes
- Use “bitwise-exact" comparison
  - Round-off error can mask bugs

SMS Debugging Directives

Insert directives any place in the code to ensure array values are correct



CSMS$COMPARE_VAR



Debugging Directives

- **CSMS$COMPARE_VAR**
  - Functionality
    - **Verifies that variable has same values for 2 separate runs**
    - **Checks interior (not halo) of variable**
    - **variables can only be a simple type (real, integer, etc.)**
  - Syntax:

```
CSMS$COMPARE_VAR(y, 'At Here')
```

    - **If variable does not have same values; error msg and termination:**

```
compare_var failed : y   At Here
Values for first, second run : 38.0   17.0
Incorrect at indices   10  13   2
```

Debugging Directives

- Two execution modes
  - One program
    - **smsRun –np 1 par_code –np 4 par_code –cv**
    - “-cv” says execute the COMPARE_VAR directives
  - Two programs (i.e. serial and SMS parallel codes)
    - SMS parallel code translated and compiled normally
    - Serial code translated as follows:

      ppp --CompareOnly my_prog.f

      (This only translates COMPARE_VAR directives)

    - **smsRun –np 1 serial_code –np 1 par_code –cv**

Debugging Directives

– **CSMS$COMPARE_VAR** example

```
      subroutine s1(x,y)
      integer, parameter :: im=10
csms$distribute(my_dh, 1) begin
      real x(im)
      real y(im)
csms$distribute end

csms$parallel(my_dh, <i>) begin
      do i = 2, im – 1
        y(i) = x(i-1) + x(i)
      end do
csms$compare_var(y(2:im-1), 'After I loop')
```

Debugging Directives

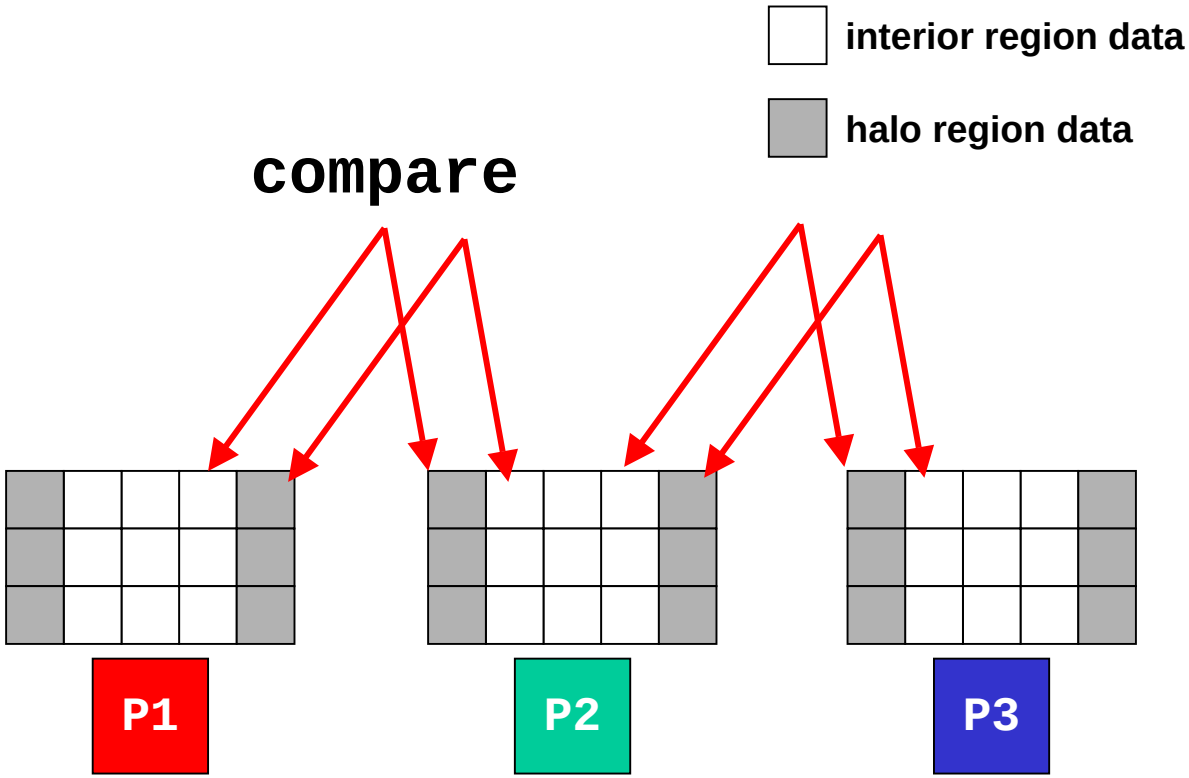
- **CSMS$CHECK_HALO**
  - Functionality
    - **Verifies that variable halo region is up-to-date**
    - **If halo not up-to-date; error msg and termination:**

      Halo check failed for var: x Here
    - **Degrades performance**
    - **Enabled when SMS_CHECK_HALO is ON**
  - Syntax:

```
CSMS$CHECK_HALO(x, 'Here')
```

    - **Syntax similar to HALO_UPDATE**
    - **User also specifies string that indicates where check being made**

CSMS$CHECK_HALO



Debugging Directives

- **CSMS$CHECK_HALO** example

```
      subroutine s1(x,y)
      integer, parameter :: im=10
csms$distribute(my_dh, 1) begin
      real x(im)
      real y(im)
csms$distribute end

csms$parallel(my_dh, <i>) begin

csms$check_halo(x, 'Before i loop')
      do i = 2, im – 1
        y(i) = x(i-1) + x(i)
      end do
```

Using SMS

- Overview
- Decompositions, array/loop transformations
- Resolving adjacent and global dependencies
- Partial parallelization
- Nesting
- I/O
- Building and running an SMS program
- Debugging support
- Performance optimization

Performance Optimization

- Combine communications (aggregation)
- Optimize halo updates
- Trade communication for redundant computation

  `CSMS$HALO_COMP`
- Optimize **`CSMS$SERIAL`**

Communication Aggregation

- Combine communication operations
  - Reduces number of messages
    - Fewer messages = less message start-up time (latency)
- Big speed-up on high-latency machines
- Works with **`CSMS$HALO_UPDATE`**, **`CSMS$TRANSFER`**, and **`CSMS$REDUCE`**

Communication Aggregation: Before

```
CSMS$HALO_UPDATE(z)
      do 100 i=3,13
         y(i) = z(i) - z(i-1) - z(i+1) - z(i-2) - z(i+2)
  100 continue
CSMS$HALO_UPDATE(a)
      do 200 i=3,13
         x(i) = a(i) + a(i-1) + a(i+1) + a(i-2) + a(i+2)
  200 continue
```

Communication Aggregation: After

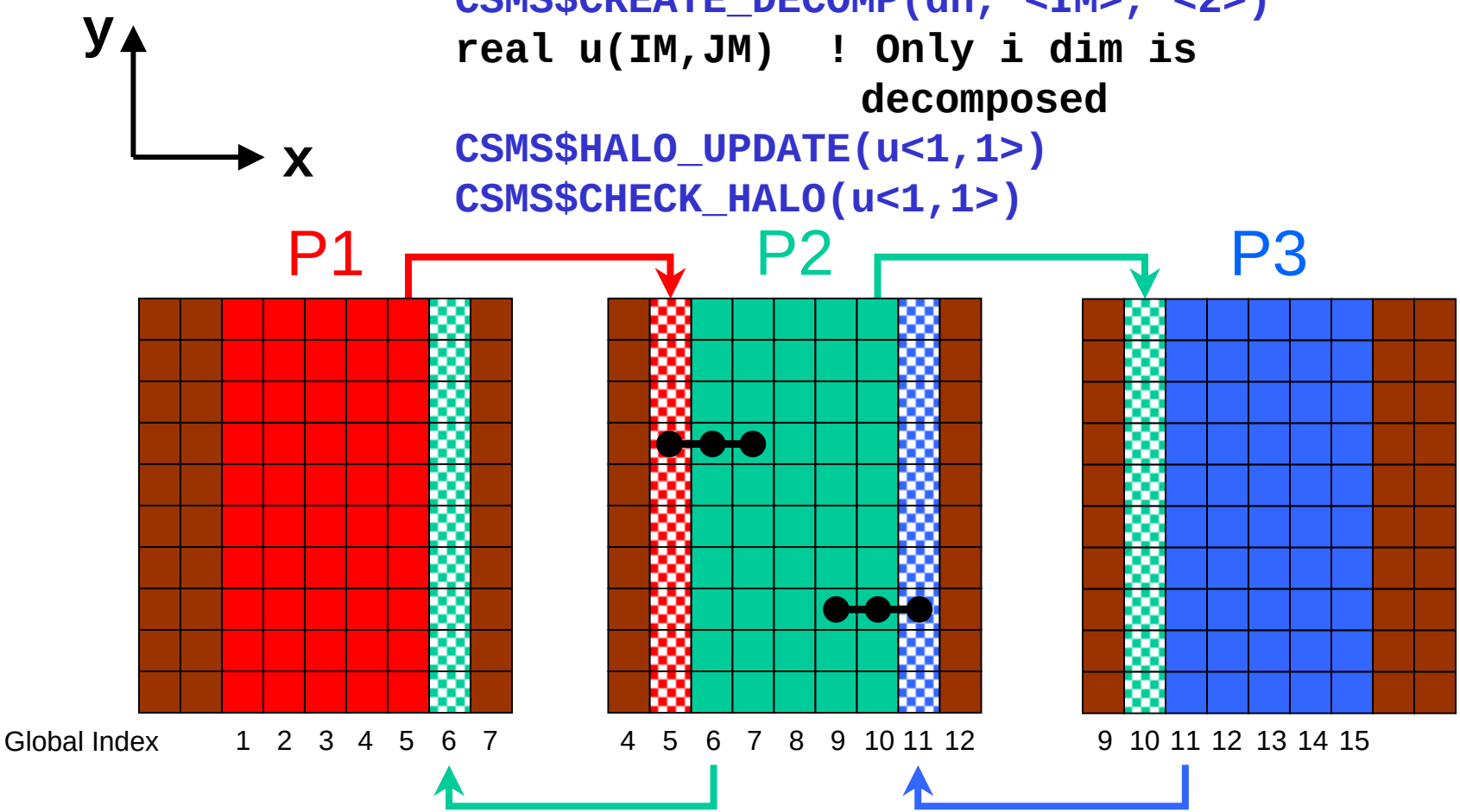
```
CSMS$HALO_UPDATE(a, z)
      do 100 i=3,13
         y(i) = z(i) - z(i-1) - z(i+1) - z(i-2) - z(i+2)
  100 continue
      do 200 i=3,13
         x(i) = a(i) + a(i-1) + a(i+1) + a(i-2) + a(i+2)
  200 continue
```

Halo Update Optimizations

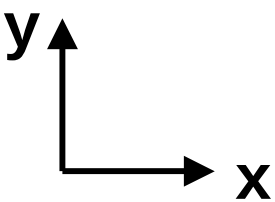
- Partial halo updates
  - Update inner layers of halo only
- Array section halo updates
  - Limit HALO_UPDATE to selected region in non-decomposed dimensions
- Both reduce amount of data sent

Partial HALO_UPDATE (Limited Halo Thickness)

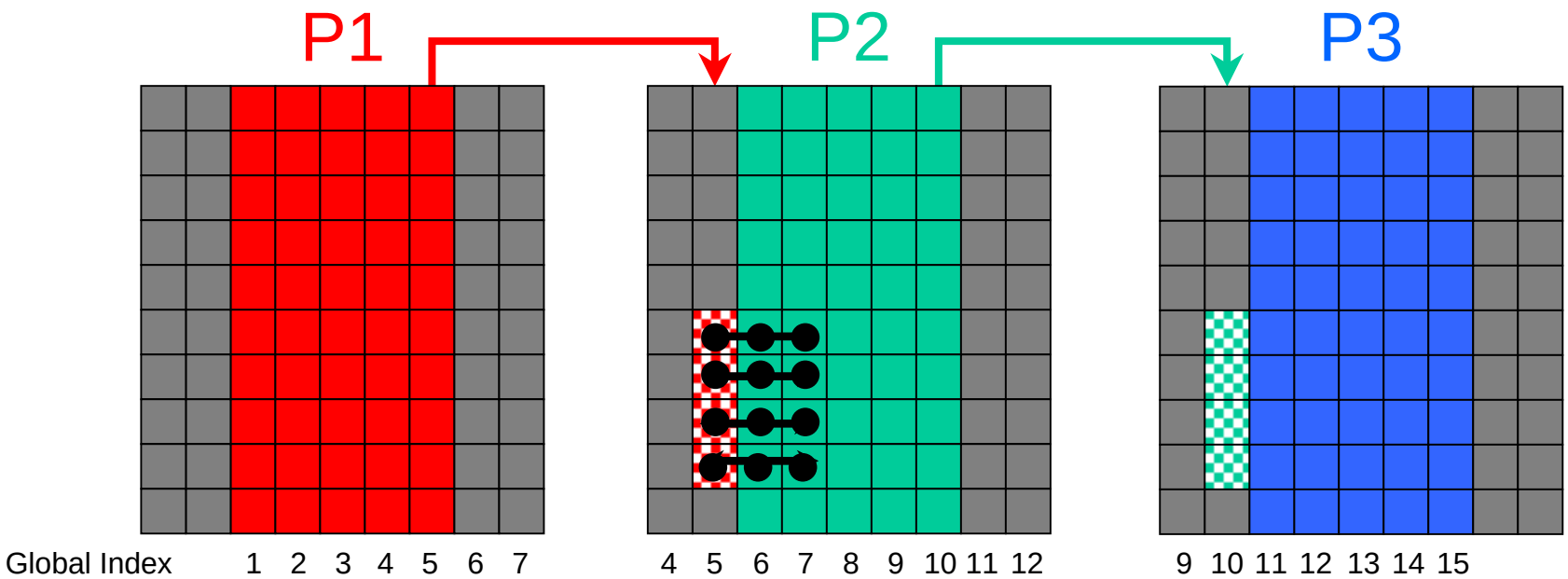
```
CSMS$CREATE_DECOMP(dh, <IM>, <2>)
real u(IM,JM)  ! Only i dim is
                 decomposed
CSMS$HALO_UPDATE(u<1,1>)
CSMS$CHECK_HALO(u<1,1>)
```



Partial HALO_UPDATE (Array Section and Limited Halo)




```
CSMS$CREATE_DECOMP(dh, <IM>, <2>)
CSMS$HALO_UPDATE(u(:,2:5)<1,0>)
CSMS$CHECK_HALO (u(:,2:5)<1,0>)
do j = 2,5
  do i = 1,IM
    v(i,j) = u(i-1,j) + u(i,j)
```



Trade Communication For Redundant Computation

- **CSMS$HALO_COMP**
  - Functionality
    - Computation in the “halo” regions
    - Reduces the number of HALO_UPDATEs
      - Can reduce latency and amount of data communicated
    - Halo computation is “redundant”
      - Identical computation is done in the "interior" of a neighboring process
    - Performance tradeoff
      - Measure performance to evaluate tradeoff

Trade Communication For Redundant Computation

- **CSMS$HALO_COMP**
  - Syntax:

```
CSMS$HALO_COMP(<ddim1_lower, ddim1_upper>,
CSMS$>         <ddim2_lower, ddim2_upper>) BEGIN
   Serial Loops
CSMS$HALO_COMP END
```

ddim1_lower is the number of halo points in the lower halo region for which computations should be executed for the first decomposed dimension

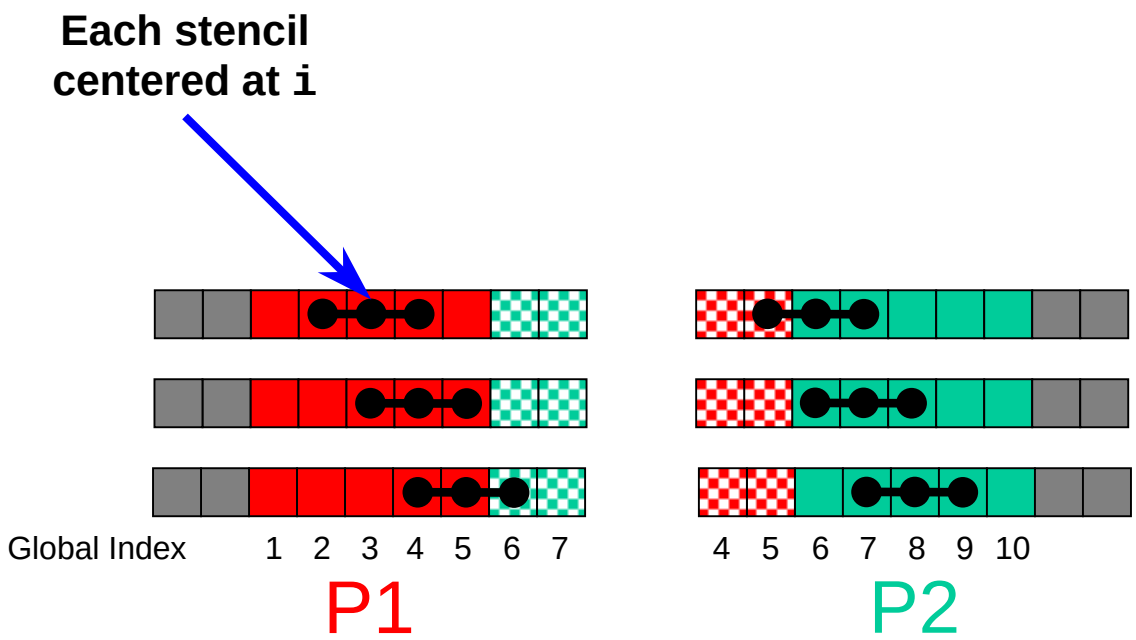
Analogously for ddim1_upper, ddim2_lower, ddim2_upper

Example Without CSMS$HALO_COMP

```
CSMS$PARALLEL(my_dh,<i>) BEGIN

CSMS$HALO_UPDATE(a<1,1>)
        do 150 i=3,8
          y(i) = a(i) - a(i+1) - a(i-1)
          z(i) = a(i) + a(i+1) - a(i-1)
  150   continue

CSMS$HALO_UPDATE(y<1,1>, z<1,1>)
        do 250 i=3,8
          x(i) = y(i)*z(i) + y(i+1)*z(i-1) + y(i-1)*z(i+1)
  250   continue
CSMS$PARALLEL END
```

Example Without `CSMS$HALO_COMP`

- Iterations for loops 150 and 250 are the same:

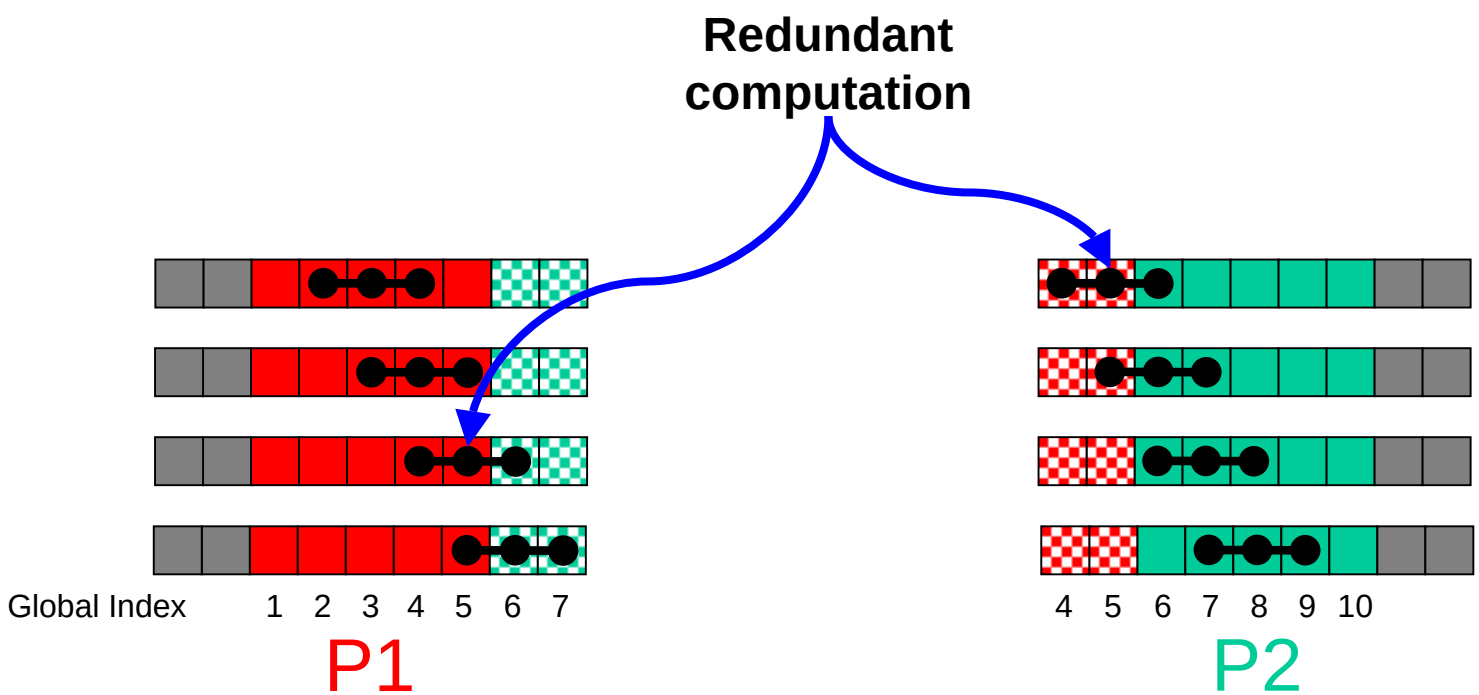


Example With CSMS$HALO_COMP

```
CSMS$PARALLEL(my_dh,<i>) BEGIN

CSMS$HALO_UPDATE(a<2,2>)
CSMS$HALO_COMP(<1,1>) BEGIN
        do 150 i=3,8
          y(i) = a(i) - a(i+1) - a(i-1)
          z(i) = a(i) + a(i+1) - a(i-1)
  150   continue
CSMS$HALO_COMP END

C Do not need CSMS$HALO_UPDATE(y<1,1>, z<1,1>)
        do 250 i=3,8
          x(i) = y(i)*z(i) + y(i+1)*z(i-1) + y(i-1)*z(i+1)
  250   continue
CSMS$PARALLEL END
```

Example With `CSMS$HALO_COMP`

- Iterations for loop 250 remain the same
- Iterations for loop 150 change as follows:



More Efficient Version of CSMS$SERIAL

```
C Assume X and Y are decomposed

CSMS$SERIAL(<x, in>, <y, out> : default=ignore) BEGIN
      do i = 1, IM
        y(i) = x(i)
      end do

CSMS$SERIAL END
```

In this efficient version, x is gathered, y is scattered but there is no other communication

With normal SERIAL, additionally “y” would be gathered, “x” would be scattered and “i” would be broadcast

Questions?