SMS USER'S GUIDE

Tom Henderson

Dan Schaffer

Mark Govett

Jacques Middlecoff

Leslie Hart

Advanced Computing Branch
Aviation Division
NOAA/Forecast Systems Laboratory
325 Broadway
Boulder, Colorado 80303

April 2004
SMS Software Version: 2.8

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

TABLE OF CONTENTS

1 Introduction

This document describes the Scalable Modeling System (SMS) and shows how SMS directives can be used to parallelize a serial Fortran program for distributed or shared memory machines. SMS is intended for use with programs that perform computations on regular gridded data sets. The primary application area thus far has been weather, ocean, and climate models. SMS has been used to parallelize models that use explicit finite difference approximation (FDA) or spectral transform methods. SMS is general enough to be useful for parallelizing similar programs in other application areas.

Before reading this document, the reader should first read the companion overview paper, "The Scalable Modeling System: A Directive-Based Parallelization Tool for Distributed and Shared Memory Computers", published in Parallel Computing and available on the SMS website. It is assumed that the reader of this User's Guide is familiar with the concepts and terms introduced in the overview document. The reader should also be familiar with basic parallel processing concepts such as distributed and shared memory, message latency and bandwidth, the Single Program Multiple Data (SPMD) programming model, and dependence analysis. The overview document describes these concepts briefly and contains references for further reading. After reading this User's Guide, the reader should have a good understanding of the steps that need to be taken to parallelize a serial program using SMS directives. If more detailed information about any directive is needed, the reader should refer to the companion reference document, "SMS Reference Manual".

1.1 Organization of this Document

The SMS User's Guide begins by introducing the SMS directives in their simplest form. Section 2 introduces the most fundamental SMS directives with simple example programs that use the method of explicit finite difference approximation. This section also introduces other SMS directives that are useful in transform-based programs such as spectral numerical weather prediction (NWP) models. The remaining sections describe in detail how the SMS directives are used in more complex situations. Section 3 explains how to divide work among multiple processes by the method of data decomposition and how to parallelize loops. Additional loop index translations needed during parallelization are described in Section 4. Sections 5, 6 and 7 cover further details about the inter-process communication directives introduced in Section 2. Section 8 describes a method by which parallelization can be done incrementally. Section 9 addresses periodic boundary conditions. Section 10 describes SMS support for mesh refinement (nesting) and coupling between different grids. Section 12 discusses parallel I/O. Directives that control program termination are dealt with in Section 14. Section 15 discusses debugging tools. Sections 16 and 17 explain how to build and run parallel SMS programs.

1.2 Terms and Conventions

Throughout most of this document, the term "process" is used instead of "processor" or "CPU". "Process" is slightly more general because it is possible to run more than one process on a single "processor" (and this may actually make sense on some types of CPU's that provide direct hardware support for multi-threaded applications). However, on most machines there will be a one-to-one mapping of processes to processors.

Fortran source code will appear in `courier` font. When program source code appears inside the main body of text, it will also be *`italicized`*. Large blocks of code will include line numbers to simplify discussions. Commands will also appear in `courier` font and will be preceded by a generic command line prompt, ">>”. The results of commands will appear in `courier` font as well. Warning messages output by SMS will be **`courier bold`**. File names will appear in *italics* when not in code examples or command lines. SMS directives will appear in **`bold`** in code examples. When directive arguments appear in the text they will be ***`courier font, bold and italicized`***. Sometimes example code will be a slightly modified version of a previous example. In that case, the changed lines will be highlighted.

2 Getting Started

2.1 Basic Parallelization Steps

The first step in any parallelization effort is to understand the performance characteristics of the serial program. Program components that take little time to run may not need to be parallelized at all. The second step is to perform dependence analysis to identify the places in the code where inter-process communication may be required. Dependencies will be discussed as relevant SMS directives are introduced. A strategy for dividing the work among the processes must then be chosen. SMS uses the method of domain decomposition in which portions of large arrays, and their associated computations, are assigned to each process. The dependence analysis is used to help pick optimal decompositions that will minimize inter-process communication. The final step is to add SMS directives to parallelize the code.

To build the parallel program, the Parallelizing Pre-Processor (PPP, a component of SMS) is first run to translate the source code with directives into new parallel source code. The translated source code is then compiled and linked with the SMS library to produce an executable program that can be run on multiple processes. The *smsRun* command is used to run the parallel program. The debugging features of SMS can then be used to test the parallel program.

SMS supports ANSI standard Fortran77 and Fortran90 language features such as full array assignment, allocatable arrays, namelist, pointer, include, do-enddo, automatic arrays, and while statements. Partial support of modules is also offered (modules may contain variable declarations but not subroutines). Both fixed and free format source code is accepted. If free format is used, SMS directives must be placed in column 1. By convention, all SMS directives mentioned in the document text are preceded by "!SMS$" or "CSMS$ for fixed formatted codes. For free format, only the "!SMS$" is permitted. The code examples will contain the full directive names. A more detailed description of supported language features can be found at the following SMS web site:

http://www-ad.fsl.noaa.gov/ac/Fortran90_Language_Support.html

2.2 A Very Simple Program

Below is a simple Fortran program that prints a message on the screen:

```
program basic_ex1
print *,'HELLO'
end
```

If this program were stored in a file named *basic_ex1.f*, it could be built using the following command:

```
>> f90 -o basic_ex1 basic_ex1.f
```

The above command assumes that the Fortran compiler is named “f90”. When run, the program produces the expected output on the screen:

```
>> basic_ex1
 HELLO
```

This program is simple enough that a parallel version can be built directly without adding any SMS directives. To build with SMS, first run the Parallel Pre-Processor (PPP) to convert the print statements into parallel print statements:

```
>> ppp basic_ex1.f
```

The above command assumes that the SMS environment variable has been correctly set and that $SMS*/bin* is in the current path. For example, if SMS is located in the directory */usr/local/sms/* then (assuming a c-shell environment) the SMS environment variable should be set as follows:

```
>> setenv SMS /usr/local/sms
```

The path could be modified using a command like this:

```
>> set path= ( $SMS/bin $path )
```

See Section 17.3 for details about setting other environment variables used by SMS. SMS translates the serial code in *basic_ex1.f* into parallel code and places the result in file *basic_ex1_sms.f*. Depending on the configuration of SMS, other temporary files may also be created. The next step is to compile *basic_ex1_sms.f* and link it to the SMS library.

```
>> f90 -c -I$SMS/include basic_ex1_sms.f
>> f90 -o basic_ex1_sms -I $SMS/include basic_ex1_sms.o -L$SMS/lib \
   -lsms -lmpi
```

The above example assumes common behavior for f90 options "-I" (specify path for include files) and "-L" (specify path for libraries). Some Fortran compilers handle these options in slightly different ways. Note that link argument "-lmpi" links to the Message Passing Interface (MPI) library. SMS uses MPI to perform underlying low-level inter-process communication on most supported machines. Some machines may require different linkers or linker arguments to link to their MPI libraries.

The next step is to run the parallel program:

```
>> smsRun -np 1 basic_ex1_sms
```

The `smsRun` command shown above runs program *basic_ex1_sms* on 1 process. The output written to the screen will look something like this:

```
SMS:: Program started: 1999:12:02::15:55:22
SMS:  BITWISE EXACT reductions will NOT be used.
 HELLO
SMS:: Program complete, exiting: 1999:12:02::15:55:22 Elapsed Time = 0
sec.
```

All output lines beginning with "SMS::" are diagnostic messages from the SMS run-time system. The first and last output lines are time-stamps printed by SMS when a program begins and when it ends. These time-stamps are a useful guide for measuring wall-clock run times. The second text line is another message from SMS that indicates default behavior of reduction operations to be discussed in Section 7.2. Henceforth, diagnostic messages from SMS will usually be omitted for brevity. The remaining line contains the text that was output when this program was run as a serial Fortran code.

The program can be run on 3 processes using the *smsRun* command like this:

```
>> smsRun -np 3 basic_ex1_sms
```

The following text appears on the screen:

```
 HELLO
```

This looks just like the run made on one process. Why? By default, SMS prints only one message per Fortran print (or write) statement to mimic the behavior of the original serial code as closely as possible. SMS also provides other "parallel print" modes, as described in Section 2.3.2 and in detail in Section 12.2.

By default, the smsRun command creates some files in the /tmp directory. On some machines, this directory is not visible to all nodes participating in the parallel run. In these cases, the location of the temporary directory must be overridden by specifying the SMS_TEMPDIR environment variable. For example:

```
setenv SMS_TEMPDIR $HOME/tmp
```

2.3 Simple Computation on a Regular Grid

Example 2-1 illustrates a very simple code that initializes an array, performs a simple computation, and prints results on the screen. It consists of two parts: include file *basic.inc* and source file *basic_ex2.f.*

```
[Include file:  basic.inc]

      integer im, jm
      common /sizes_com/ im, jm


[Source file:  basic_ex2.f]

      program basic_ex2
      include 'basic.inc'
      im = 10
      jm = 10
      call compute
      end

      subroutine compute
      include 'basic.inc'
      integer i, j, xsum
      integer x(im,jm)
      do 100 j=1,jm
      do 100 i=1,im
        x(i,j) = 1
  100 continue
      xsum = 0
      do 200 j=1,jm
      do 200 i=1,im
        xsum = xsum + x(i,j)
  200 continue
      print *,'xsum = ',xsum
      return
      end
```

Example 2-1: A simple serial code to initialize an array and print a global sum.

This program initializes array *x*, sums the elements of *x*, and prints the result on the screen as shown below:

```
>> basic_ex2
 xsum = 100
```

Notice that this program uses automatic (dynamically allocated) arrays instead of traditional Fortran77 static array declarations. The SMS directives support both dynamic and static memory allocation schemes. Examples with dynamic memory allocation are shown first because they are slightly simpler. Static allocation examples appear in Section 3.3.

2.3.1 Parallelization by Domain Decomposition

Programs such as this one that involve computations on regular grids are often best parallelized using the method of domain decomposition. Arrays and the computations performed on them are "decomposed" (divided up) among the processes as evenly as possible. For example, Figure 2-1, Figure 2-2, and Figure 2-3 show how array `x` might be decomposed in the `i` dimension over one, two and three processes.

Note that the sub-domains of array `x` become smaller as the number of processes increases. These sub-domains are referred to as "local" arrays because they cannot be accessed by other processes on a distributed memory machine. In SMS terms, the original array `x` in the serial code is sometimes referred to as a "global array". Indices used to access a global array are called "global indices" while indices used to access a local array are called "local indices". Similarly, sizes of the dimensions of a global array are called "global sizes" and sizes of the dimensions of a local array are called "local sizes". For dynamic memory code, the local and global indices are identical. We will see in Section 3.3 that the global and local indices differ from each other for static memory codes.

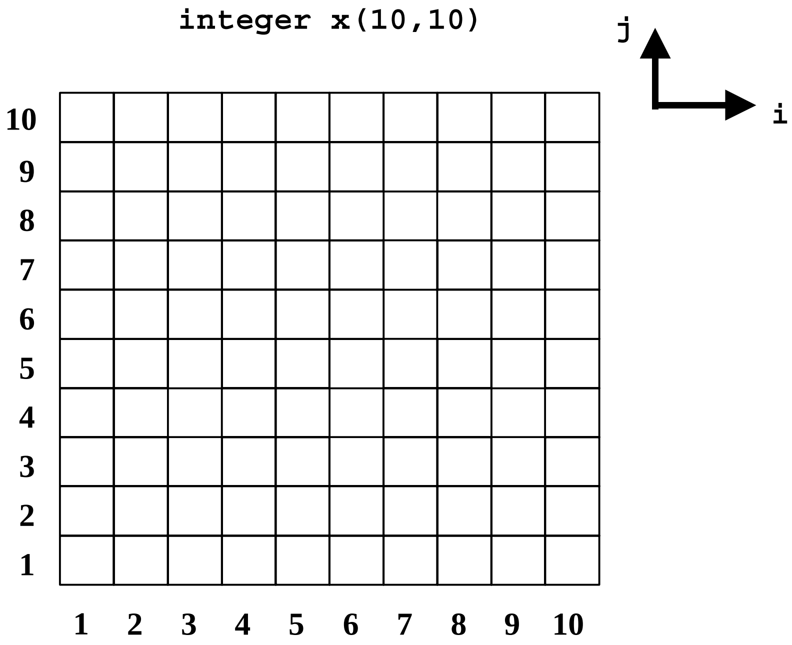


Figure 2-1: A graphical representation of a non-decomposed 10 by 10 integer array.

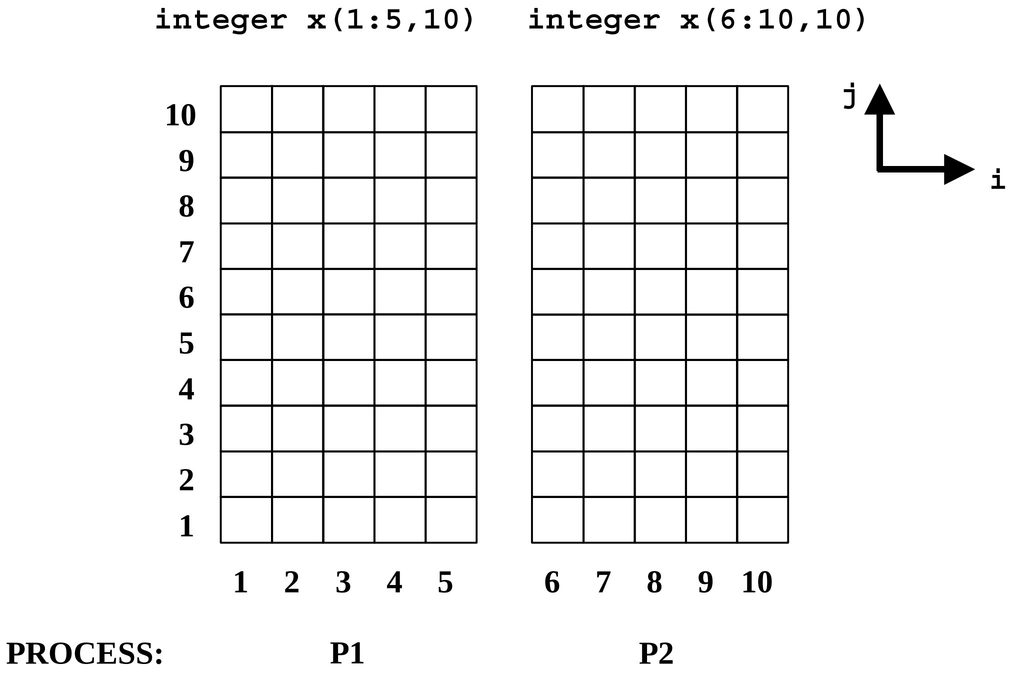


Figure 2-2: An illustration of a 10 by 10 array decomposed over two processes. These integer arrays are now local arrays declared by each process. When dynamic memory is used, global addressing is used to access local array elements. Thus, on process P2, the first dimension ranges from 6 to 10.

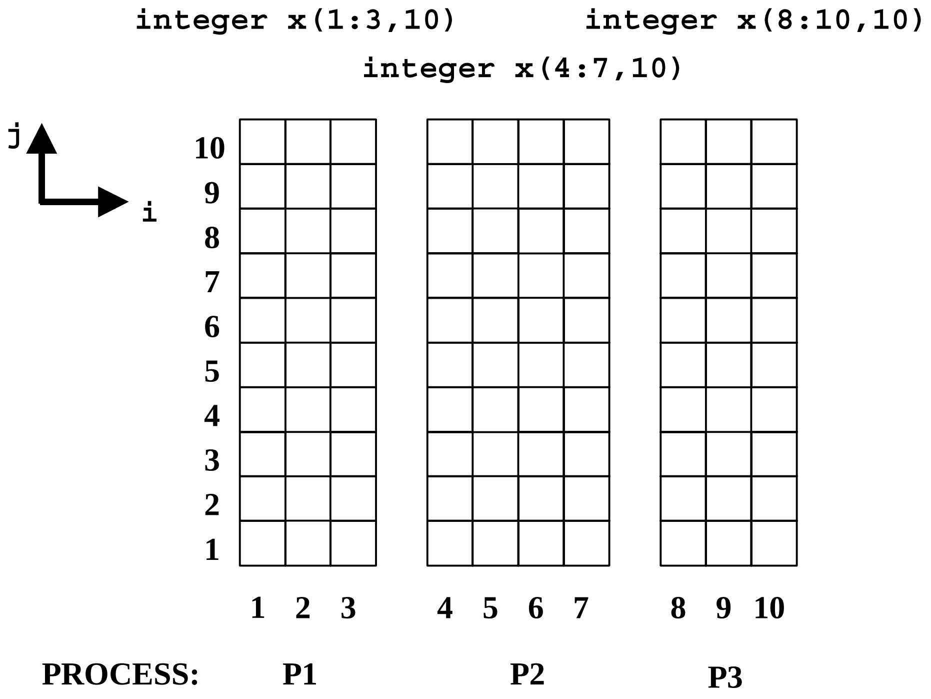


Figure 2-3: A 10 by 10 array decomposed over three processes. In this example, the locally declared size of process P2 is larger than the sizes of P1 or P3.

In this program, domain decomposition of array *x* requires three basic steps. First, the way in which *x* will be decomposed must be described. For this simple example, we choose to decompose only in the *i* dimension (decompositions of two dimensions are discussed in Section 3.2). Second, the declarations of array *x* should be modified to reflect smaller local sizes. Finally, the start and stop indices of each relevant loop must be changed to reflect the smaller range of local indices. These three steps are accomplished using four SMS directives. The DECLARE_DECOMP and CREATE_DECOMP directives are used to describe a single decomposition. Array declarations are modified using the DISTRIBUTE directive while loop start and stop indices are changed using the PARALLEL directive. These directives have been inserted into the serial program as shown in Example 2-2:

```
[Include file:  basic.inc]
      integer im, jm
      common /sizes_com/ im, jm
CSMS$DECLARE_DECOMP(DECOMP_I, 1)
```

```
[Source file:  basic_ex2.f]

      program basic_ex2
      include 'basic.inc'
      im = 10
      jm = 10
CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
      call compute
      end

      subroutine compute
      include 'basic.inc'
      integer i, j, xsum
CSMS$DISTRIBUTE(DECOMP_I, 1) BEGIN
      integer x(im,jm)
CSMS$DISTRIBUTE END
CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
      do 100 j=1,jm
      do 100 i=1,im
        x(i,j) = 1
  100 continue
      xsum = 0
      do 200 j=1,jm
      do 200 i=1,im
        xsum = xsum + x(i,j)
  200 continue
CSMS$PARALLEL END
      print *,'xsum = ',xsum
      return
      end
```

Example 2-2: A simple serial code with comment-based SMS directives added.

Notice that each of the SMS directives begins with five characters "**CSMS$**" which makes it a Fortran comment. This is true for all SMS directives. Also, note that both the DISTRIBUTE and PARALLEL directives come as BEGIN-END pairs. When an SMS directive appears in this form, its scope consists of all lines of code between the "BEGIN" and "END" directives. Some SMS directives, such as TRANSFER (Section 6) and REDUCE (Section 7) may be used either alone or as a BEGIN-END pair. The text translation effects of a BEGIN-END directive pair do not extend into called subroutines.

The first directive, DECLARE_DECOMP, is used to give a name to the SMS decomposition that will be used to divide among the processes the work done in loops 100 and 200. Its first argument, ***DECOMP_I***, is the user-chosen name for the decomposition. Any valid Fortran variable name (up to 20 characters long) may be used to name a decomposition provided it does not conflict with any variable in the serial code. The second argument, ***1***, is an integer that indicates how many dimensions are decomposed. This argument is omitted if static memory allocation is used (see Section 3.3) or if the decomposed arrays have non-unit lower bounds (see Section 3.6).

Next, the CREATE_DECOMP directive is used to describe what kind of decomposition ***DECOMP_I*** will be. The first argument is the decomposition name ***DECOMP_I*** specified in the DECLARE_DECOMP directive. The second argument, ***<im>***, describes the decomposition as a 1-dimensional decomposition where the number of data points is the global size of the original serial dimension. The last argument, ***<0>***, indicates that this decomposition will have no halo regions (halo thickness = 0). Halo regions are introduced later in this section and are described in detail in Section 5.1.

The third directive, DISTRIBUTE, associates array *x* with the decomposition ***DECOMP_I***. The second argument is used to indicate how array dimension(s) correspond to the dimensions of the decomposition named ***DECOMP_I***. In this simple one-dimensional decomposition, ***1*** indicates that the first dimension of the array *x* will be decomposed as described by the single dimension of the SMS decomposition named

DECOMP_I. The distinction between "dimension of an array" and "dimension of an SMS decomposition" will become more clear in the two-dimensional decomposition examples shown later in Section 3.2.

The DISTRIBUTE directive does two things. First, it identifies array declarations that will be translated to use local sizes. In the above example program, the DISTRIBUTE directive will cause the declaration of *x* to be translated to a local declarations such as one of those shown in Figure 2-1, Figure 2-2, or Figure 2-3 (depending on the number of processes). The second task of DISTRIBUTE is to provide information about how each array is decomposed to other SMS directives and to support automatic parallelization of some operations (such as unformatted I/O). These features are described in detail in later sections.

Finally, the PARALLEL directive identifies loops that must be modified to span the smaller local arrays during translation. The second argument, ***<i>***, indicates that loops with loop index *i* should be translated to span the decomposed dimension of array *x*. For example, if the program in Example 2-2 is run on two processes then *i* loops 100 and 200 will span local indices 1 through 5 on each process.

Building the SMS parallel code is a bit more complicated than the previous example due to the presence of the include file that contains a directive. Two commands are now needed. The first translates the include file:

```
>> ppp --header basic.inc
```

The "--header" option to the PPP command indicates that the file is an include file and must be translated differently than a standard Fortran source file. In the command above, include file *basic.inc* will be translated into new SMS include file *basic.inc.SMS*. The second command requires PPP option "--Finclude" to translate the Fortran source file:

```
>> ppp --Finclude=basic.inc basic_ex2.f
```

The "--Finclude" option to the PPP command indicates that file *basic.inc* is an include file that has been translated by PPP. During translation of source file *basic_ex2.f*, inclusions of this file will be translated from

```
      include 'basic.inc'
```

to

```
      include 'basic.inc.SMS'
```

to ensure that the translated include file is used.

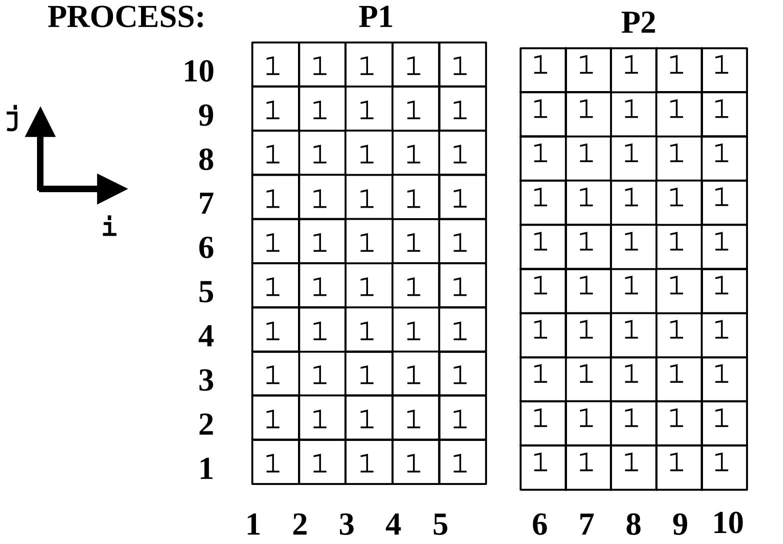
Running this program on one process produces the expected result.

```
>> smsRun -np 1 basic_ex2_sms
 xsum =  100
```

However, when this program is run on two and three processes, the values of *xsum* differ from the serial run.

```
>> smsRun -np 2 basic_ex2_sms
 xsum =  50

>> smsRun -np 3 basic_ex2_sms
 xsum =  30
```

Why did the parallel program produce incorrect results? The answer lies in the computations made in loop 200. In this loop, all of the elements of array *x* are summed and the result is placed in variable *xsum*. However, when the program is run on two or three processes, each process sums only its own local sub-domain of *x* as illustrated in Figure 2-4, and Figure 2-5. To reproduce the result of the original serial code, we will need the REDUCE (see Section 2.3.3) directive.



	1	2	3	4	5
10	1	1	1	1	1
9	1	1	1	1	1
8	1	1	1	1	1
7	1	1	1	1	1
6	1	1	1	1	1
5	1	1	1	1	1
4	1	1	1	1	1
3	1	1	1	1	1
2	1	1	1	1	1
1	1	1	1	1	1

	6	7	8	9	10
10	1	1	1	1	1
9	1	1	1	1	1
8	1	1	1	1	1
7	1	1	1	1	1
6	1	1	1	1	1
5	1	1	1	1	1
4	1	1	1	1	1
3	1	1	1	1	1
2	1	1	1	1	1
1	1	1	1	1	1

$$\mathtt{xsum} = \sum_{i}\sum_{j} \mathtt{x(i,j)}$$

P1: `xsum = 50` **P2:** `xsum = 50`

Figure 2-4: Each process sums its local portion of the array x.

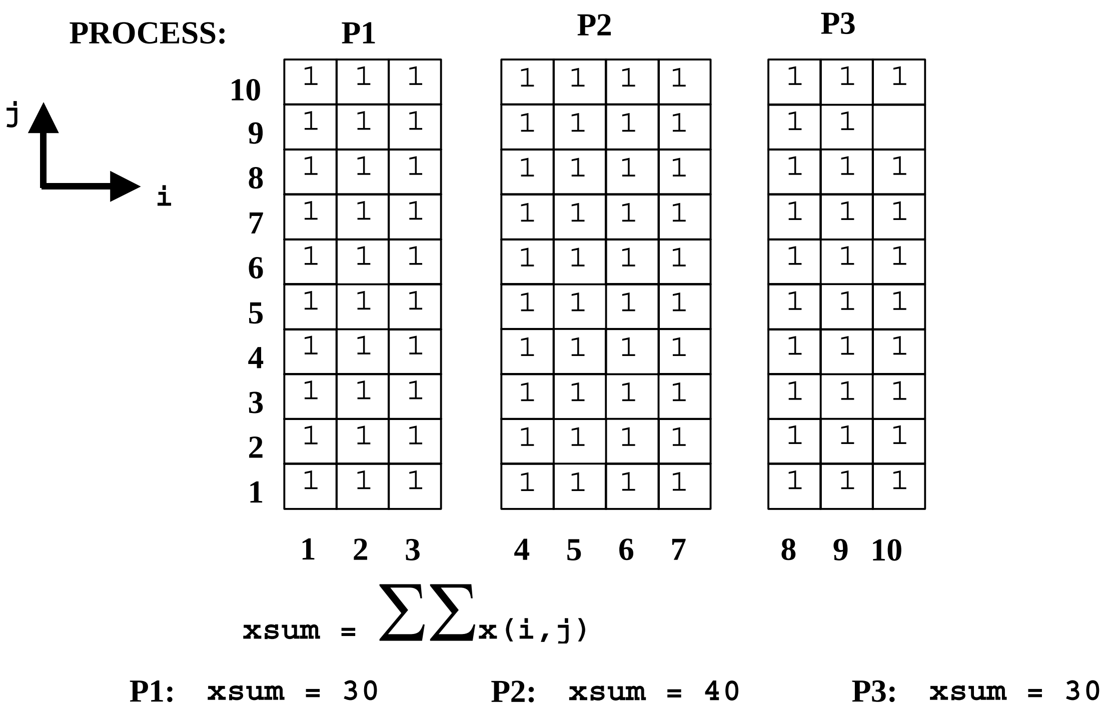


Figure 2-5: In this example, local sums are produced on each of the three processes.

2.3.2 Parallel Printing

In SMS, by default, only one process will print a message when a print statement is encountered. Therefore, the value of *xsum* printed is the value of *xsum* computed locally only on the printing process. We can see the value of *xsum* on every process by changing the default print behavior with the PRINT_MODE directive. The print statement on line 26 of the program in Example 2-2 would be modified as shown below:

```
CSMS$PRINT_MODE(ASYNC) BEGIN
      print *,'xsum = ',xsum
CSMS$PRINT_MODE END
```

This PRINT_MODE directive changes the print mode from the default mode to "asynchronous" mode. When a print statement is encountered in asynchronous print mode, each process will print a message to the screen. When run on two processes, the following results are printed:

```
>> smsRun -np 2 basic_ex2_sms
 xsum =  50
 xsum =  50
```

Clearly, each process has computed the correct sum for its local half of array *x*. When run on three processes we may see any of the following results:

```
>> smsRun -np 3 basic_ex2_sms
 xsum =  40
 xsum =  30
 xsum =  30

>> smsRun -np 3 basic_ex2_sms
 xsum =  30
 xsum =  40
 xsum =  30

>> smsRun -np 3 basic_ex2_sms
 xsum =  30
 xsum =  30
 xsum =  40
```

In the asynchronous print mode, the messages printed by each process may come out in any order. Another parallel print mode supported by SMS is the "ORDERED" print mode which preserves process order. Section 12.2 describes the SMS print modes in more detail.

2.3.3 Reduction

We have seen that each process has computed the correct sum for its local sub-domain of array *x*. To reproduce the same result as the original serial code, the local sums must be added together as shown in Figure 2-6. In more general terms, the computed value of *xsum* depends on all of the values of array *x*. This is known as a "global dependence" because the result of the computation depends on every element of global array *x*.

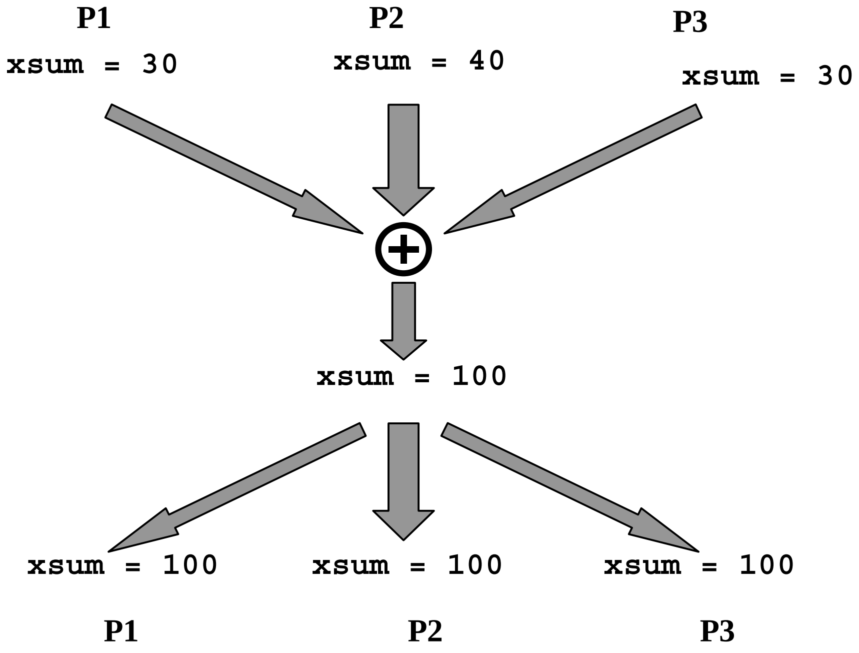


Figure 2-6: In this example, the reduction gathers the local sums, computes a global sum and then broadcasts the result out to the processes.

The REDUCE directive is used to resolve this dependence. To compute a global sum, insert the following line immediately before the print statement on line 26 of Example 2-2:

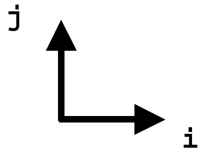
```
CSMS$REDUCE(xsum,SUM)
```

The REDUCE directive performs communications necessary to reduce the local values of a variable on each process to a single value that is identical on all processes. A specified operator is used to combine the values from each process. The first argument indicates that *xsum* is the name of the variable to be reduced. The second argument, ***SUM***, specifies that the local values of *xsum* will be summed during reduction. Reductions are described in more detail in Section 7. The parallel program now produces the expected results when run on various numbers of processes (assuming the PRINT_MODE directives used in Section 2.3.2 are removed):

```
>> smsRun –np 2 basic_ex2_sms
 xsum =  100
>> smsRun –np 3 basic_ex2_sms
 xsum =  100
```

2.4 Boundary Initialization

In Example 2-2, all elements of array *x* were initialized to the same value. Usually, it is desirable to initialize array elements differently depending on their location. This occurs often in models where elements near the model boundaries may be treated differently than other array elements. Example 2-3 below shows a variant of subroutine *compute* from Example 2-2 (changes are highlighted) that sets elements on the array boundaries where *i=1* or *i=im* to 2 and all other elements to 1. This is illustrated in Figure 2-7.



10	2	1	1	1	1	1	1	1	1	2
9	2	1	1	1	1	1	1	1	1	2
8	2	1	1	1	1	1	1	1	1	2
7	2	1	1	1	1	1	1	1	1	2
6	2	1	1	1	1	1	1	1	1	2
5	2	1	1	1	1	1	1	1	1	2
4	2	1	1	1	1	1	1	1	1	2
3	2	1	1	1	1	1	1	1	1	2
2	2	1	1	1	1	1	1	1	1	2
1	2	1	1	1	1	1	1	1	1	2
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

$$\mathbf{xsum} = \sum_{i}\sum_{j} \mathbf{x(i,j)}$$

$$\mathbf{xsum = 120}$$

Figure 2-7: An illustration of a boundary initialization where edge point values are different than interior points.

```
      subroutine compute
      include 'basic.inc'
      integer i, j, xsum
CSMS$DISTRIBUTE(DECOMP_I, 1) BEGIN
      integer x(im,jm)
CSMS$DISTRIBUTE END
CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
      do 100 j=1,jm
      do 100 i=1,im
        x(i,j) = 1
  100 continue
      do 110 j=1,jm
        x( 1,j) = 2
        x(im,j) = 2
  110 continue
      xsum = 0
      do 200 j=1,jm
      do 200 i=1,im
        xsum = xsum + x(i,j)
  200 continue
CSMS$PARALLEL END
```

```
22  CSMS$REDUCE(xsum,SUM)
23        print *,'xsum = ',xsum
24        return
25        end
```

Example 2-3: Boundary initialization requires special handling.

When the serial version of Example 2-3 is run, the following results are printed on the screen:

```
>> basic_ex3
 xsum =  120
```

However, when the parallel code is run on more than one process, results are unpredictable:

```
>> smsRun -np 2 basic_ex3_sms
<core dump>
```

The reason for these erroneous results can be seen by examining new loop 110 in detail. Line 14 in loop 110 contains the following statement:

```
        x(im,j) = 2
```

This statement will perform the following assignments:

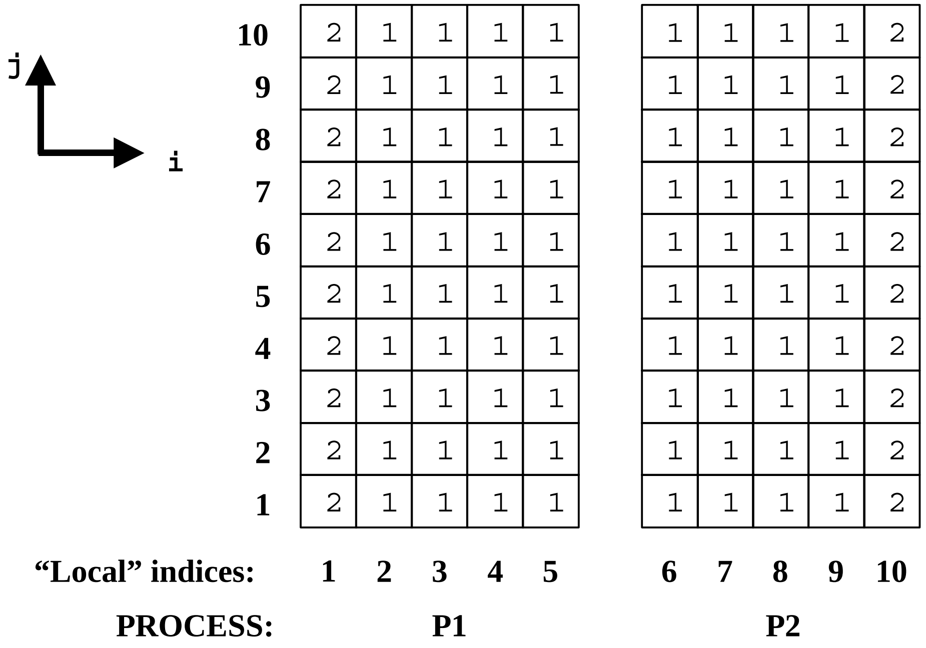
```
        x(10, 1) = 2
        x(10, 2) = 2
...
        x(10,10) = 2
```

On process 1 of a 2 process run, array *x* is dimensioned *x(1:5,1:10)* (see Figure 2-2) so *x(10,10)* is out of bounds. The behavior of any program that performs such assignments is unpredictable. Similarly, line 13 causes an out-of-bounds assignment on process 2.

To address this problem, the assignment statements must be modified so they are only executed on the processes that contain the specified global indices in their local sub-domains. The GLOBAL_INDEX directive solves these problems as shown below:

```
       do 110 j=1,jm
CSMS$GLOBAL_INDEX(1) BEGIN
         x( 1,j) = 2
         x(im,j) = 2
CSMS$GLOBAL_INDEX END
   110 continue
```

The GLOBAL_INDEX directive ensures the enclosed statements are only executed on the appropriate processes. Now only process 1 will execute line 13 and only process 2 will execute line 14. The argument in the GLOBAL_INDEX directive, ***1***, indicates that these translations will be applied to array indices that correspond to the first (and in this case only) decomposed dimension. In this case, the decomposed dimension corresponds to the *i* dimension of array *x*. (The concept of "decomposed dimension" is explained in detail in Section 3.) The effects of the GLOBAL_INDEX directives on the assignments of *x(1,j)* and *x(im,j)* are shown for the two process case in Figure 2-8.



10	2	1	1	1	1	1	1	1	1	2
9	2	1	1	1	1	1	1	1	1	2
8	2	1	1	1	1	1	1	1	1	2
7	2	1	1	1	1	1	1	1	1	2
6	2	1	1	1	1	1	1	1	1	2
5	2	1	1	1	1	1	1	1	1	2
4	2	1	1	1	1	1	1	1	1	2
3	2	1	1	1	1	1	1	1	1	2
2	2	1	1	1	1	1	1	1	1	2
1	2	1	1	1	1	1	1	1	1	2
“Local” indices:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
PROCESS:			**P1**					**P2**		

Figure 2-8: GLOBAL_INDEX is used to initialize the boundaries of the array *x*.

Now when the parallel code is run, results match the serial code:

```
>> smsRun -np 2 basic_ex3_sms
 xsum =  120
>> smsRun -np 3 basic_ex3_sms
 xsum =  120
```

2.5 A Simple Explicit FDA Program

The following example is an explicit FDA program that solves Laplace's equation on a two-dimensional surface with fixed boundaries using Jacobi relaxation. On a two-dimensional surface, Laplace's equation takes the form:

$$\frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y} = 0$$

A simple approach is to discretize the two-dimensional space and use and explicit finite difference approximation to the derivatives to seek a numerical solution. The discrete equation is:

$$4*f(i,j) - f(i-1,j) - f(i+1,j) - f(i,j-1) - f(i,j+1) = 0$$

The initial state is *f* on the boundaries. The boundaries are constant and non-periodic. The above equation is solved for *f(i,j)* iteratively until it converges. The solution is said to converge when the difference between successive solutions is less than a specified threshold. The difference between values of *f(i,j)* in two successive iterations is the following:

$$df(i,j) = (1/4) * (f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1)) - f(i,j)$$

Using the method of Jacobi relaxation, the value of *f(i,j)* during an iteration is calculated from the value of *f(i,j)* computed in the previous iteration as follows:

$$fnew(i,j) = fold(i,j) + df(i,j)$$

In Example 2-4 below, boundary elements of array *f* are initially set to 2.0 (lines 25-31). Laplace's equation is then solved and diagnostic messages are printed on the screen. Previously described SMS directives have already been inserted.

```
[Source file:  laplace.f]
 1        program laplace
 2        include 'basic.inc'
 3        im = 10
 4        jm = 10
 5  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
 6        call laplace
 7        end
 8
 9        subroutine laplace
10        include 'basic.inc'
11        integer i, j, iter
12        real max_error
13        real tolerance
14        parameter (tolerance = 0.001)
15  CSMS$DISTRIBUTE(DECOMP_I, 1) BEGIN
16        real f(im,jm), df(im,jm)
17  CSMS$DISTRIBUTE END
18  CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
19        do 100 j=1,jm
20        do 100 i=1,im
21          f(i,j) = 0.0
22    100 continue
23        do 110 j=1,jm
24  CSMS$GLOBAL_INDEX(1) BEGIN
25          f( 1,j) = 2.0
26          f(im,j) = 2.0
27  CSMS$GLOBAL_INDEX END
28    110 continue
29        do 120 i=1,im
30          f(i, 1) = 2.0
31          f(i,jm) = 2.0
32    120 continue
33        iter = 0
34        max_error = 2.0 * tolerance
35  C main iteration loop...
36        do while ((max_error .gt. tolerance) .and. (iter .lt. 1000))
37          iter = iter + 1
38          max_error = 0.0
39          do 200 j=2,jm-1
40          do 200 i=2,im-1
```

```
41            df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1))
42       &                - f(i,j)
43    200   continue
44          do 300 j=2,jm-1
45          do 300 i=2,im-1
46            if (max_error .lt. abs(df(i,j))) then
47              max_error = abs(df(i,j))
48            endif
49    300   continue
50  CSMS$REDUCE(max_error, MAX)
51          do 400 j=2,jm-1
52          do 400 i=2,im-1
53            f(i,j) = f(i,j) + df(i,j)
54    400   continue
55        enddo
56  CSMS$PARALLEL END
57        print *, 'Solution required ',iter,' iterations'
58        print *, 'Final error = ', max_error
59
60        return
61        end
```

Example 2-4: Serial code plus directives illustrate a parallel solution to Laplace's equation. This solution, using a one-dimensional decomposition, produces incorrect results.

Notice that the REDUCE directive generates the global maximum error from the local maxima on each process.

When the serial program is run, the following messages are printed on the screen:

```
>> laplace
 Solution required  85  iterations
 Final error =  9.9968910E-4
```

When the parallel program is run on more than one process, results are incorrect:

```
>> smsRun –np 2 laplace_sms
 Solution required  45  iterations
 Final error =  9.9253654E-4

>> smsRun –np 3 laplace_sms
 Solution required  131  iterations
 Final error =  9.9420547E-4
```

Why do results change for different numbers of processes? The answer lies in the computations made on lines 41 and 42:

```
  df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1)) - f(i,j)
```

Here, each *df(i,j)* is computed from *f(i-1,j)*, *f(i+1,j)*, *f(i,j-1)*, f*(i,j+1)*, and *f(i,j)*. This type of dependence is called an "adjacent dependence" because the computation at point *(i,j)* depends on data at adjacent (or "nearby") points. Adjacent dependencies are often represented graphically using a "stencil" as shown in Figure 2-9.

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

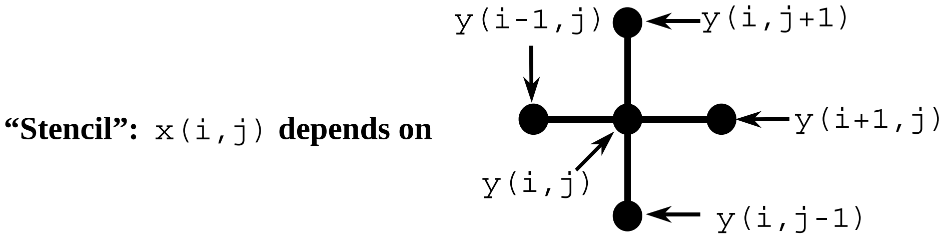


Figure 2-9: This five-point stencil illustrates the dependencies of the array *y* on the computation of *x*.

In Figure 2-10 stencils have been overlaid on graphical representations of the sub-domains assigned to each process during a run made on three processes. The stencil centered at point *(2,2)* on process P1 illustrates that computations at this grid point require values from points *(2,2)*, *(2,1)*, *(1,2)*, *(2,3)*, and *(3,2)*. These array elements are all inside the local sub-domain of process P1. Similarly, computations at point *(5,8)* depend only on array elements inside the local sub-domain of process P2. However, computations on sub-domain boundaries cannot be performed so easily. For example, the stencil centered at point *(7,5)* on process P2 depends on the element at point *(8,5)* which is located in the local sub-domain of process P3. Similarly, the stencil centered at point *(8,2)* on process P3 requires an element from process P2. The results of the parallel program above are incorrect because no data is sent between processes to resolve the adjacent dependence in loop 200.

It is possible to solve this problem by sending single data points between processes. However, on high-latency machines, sending messages that contain only one array element is very inefficient compared to sending messages that contain many array elements. The most common approach to handle adjacent dependencies is to create "halo" or “ghost” regions to store these data as shown in Figure 2-11. Each halo point corresponds to an interior point of a neighboring process. For example, in Figure 2-11, halo point (8,5) in process P2 corresponds to interior point (8,5) in process P3. When data in these regions are needed, the halo regions are updated by swapping columns (or larger blocks) of data between processes as shown in Figure 2-12. This form of inter-process communication is called "exchange" and is supported by the EXCHANGE directive.

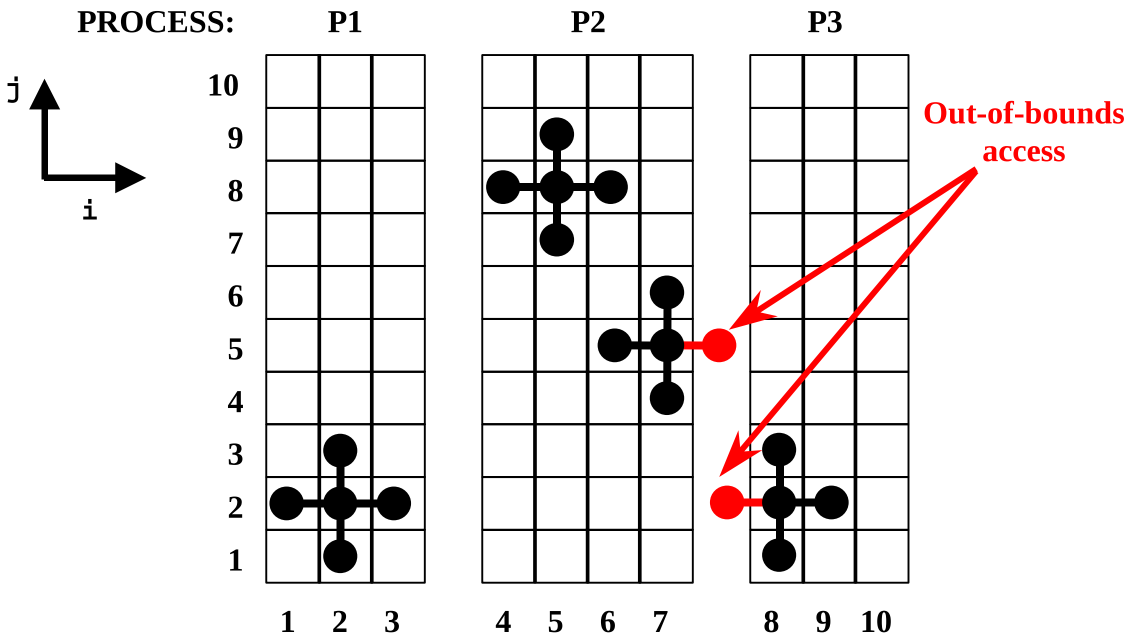


Figure 2-10: Illustration of how an adjacent dependence causes out-of-bounds data references on processes P2 and P3.

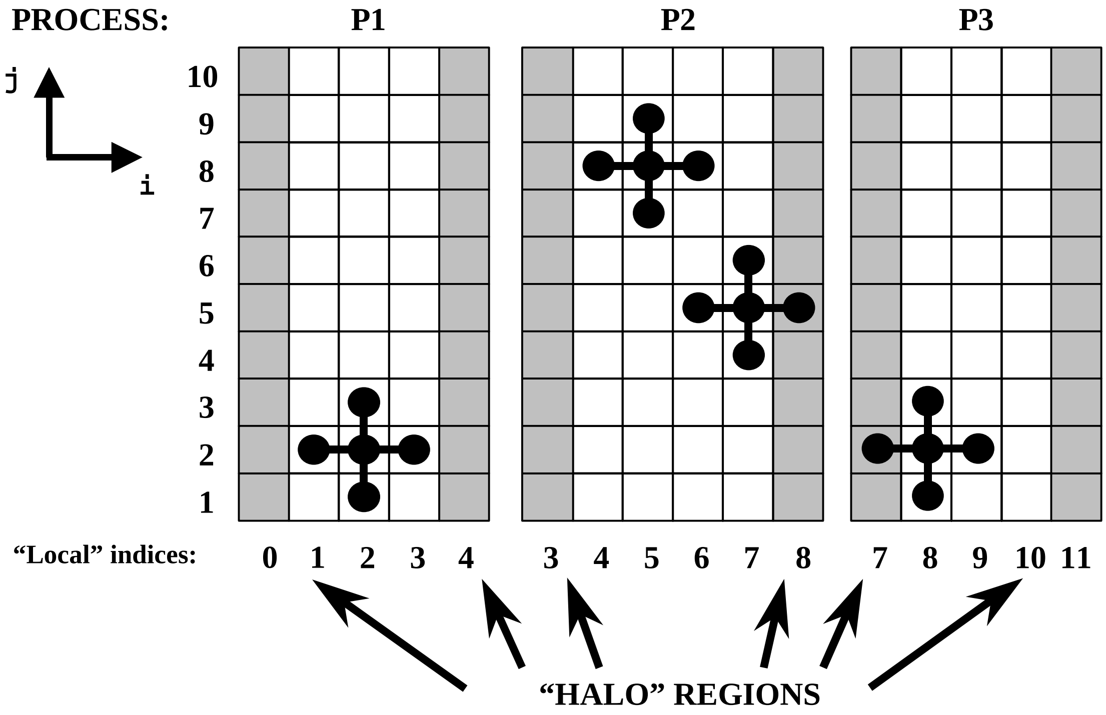


Figure 2-11: Halo regions eliminate the out-of-bounds array references. Notice the distinction between interior points (in white) and halo points (in gray). The local indices of the halo points on the domain edges actually lie outside the serial domain range (1 to 10). These edge halo points are only used for problems that have periodic boundary conditions as described in Section 9.

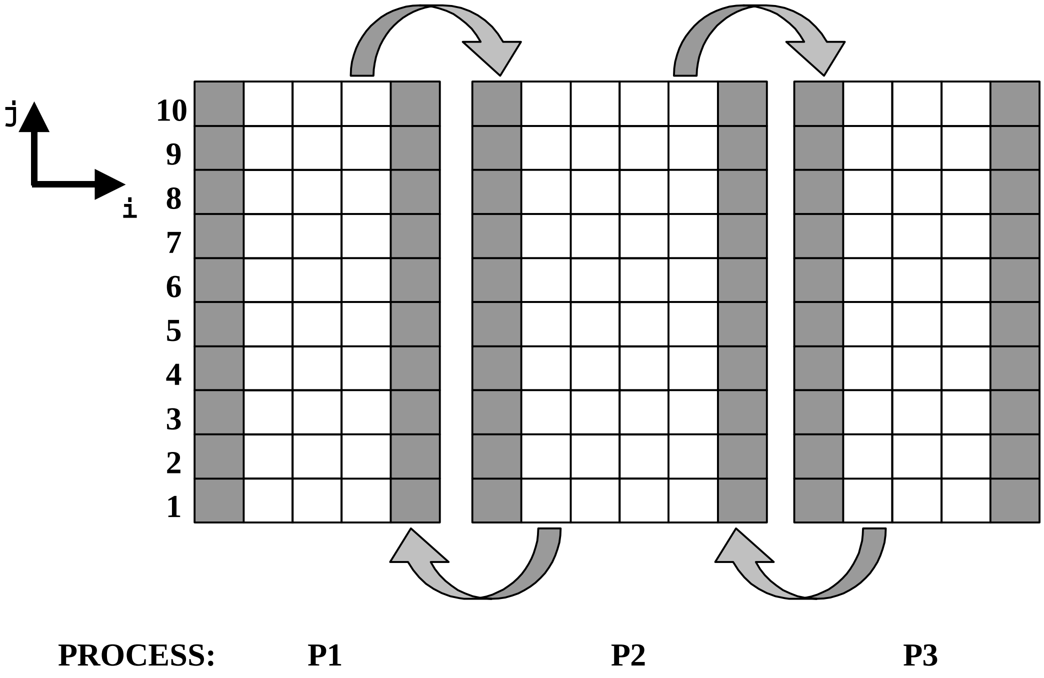


Figure 2-12: Halo regions are updated by exchanging data between adjacent processes.

The laplace program in Example 2-4 can be corrected by modifying line 5 to specify one halo point

```
CSMS$CREATE_DECOMP(DECOMP_I, <im>, <1>)
```

and by adding the following directive before line 39:

```
CSMS$EXCHANGE(f)
```

The third argument of the CREATE_DECOMP directive has been changed to ***<1>***. This indicates that all arrays decomposed using ***DECOMP_I*** will have a halo region one point thick added in the first decomposed dimension (the *i* dimension in this case). The only argument of the EXCHANGE directive is the name of the variable (*f)* to be exchanged. The directive is placed immediately before loop 200 to ensure that halo regions of *f* are updated prior to the computations that need them. The EXCHANGE directive is described in more detail in section 5.1.

Now the parallel program produces the correct results on more than one process:

```
>> smsRun -np 2 laplace_sms
 Solution required  85  iterations
 Final error =  9.9968910E-4

>> smsRun -np 3 laplace_sms
 Solution required  85  iterations
 Final error =  9.9968910E-4
```

2.6 Writing Output to Disk

The Laplace solver in Example 2-4 would be more useful if the final state of array *f* could be written to disk. This is easily done by adding the following code fragment immediately before the *return* statement (line 60) in subroutine *laplace*:

```
      open(10, file='f.out', form='unformatted')
      write(10) f
      close(10)
```

When the serial program is run, file *f.out* is written. For the SMS parallel program, no additional directives are required to handle this output. By default, SMS automatically generates *f.out* in exactly the same format as the serial program, for any number of processes. However, SMS can also produce other file formats as discussed in Section 12.1.

2.7 Using Multiple Decompositions

So far, we have seen how to parallelize a program that only requires a single domain decomposition. However, many programs require the use of different decompositions at different times to run efficiently in parallel. The TRANSFER directive provides the means to transform arrays between decompositions. Spectral models are prime candidates for application of TRANSFER (see Section 6.2).

In this section, we present a simple case where two different decompositions are needed. In Example 2-5, the statement at line 42 contains a dependence called a "recurrence relation". In this statement, an update to *x(i,j)* depends on *x(i,j-1)* which was updated in the previous loop iteration. SMS does not provide directives that directly support parallelization of a recurrence relation if the array dimension is decomposed. Since the second (*j*) dimension of *x* is decomposed, SMS cannot handle this statement. Similarly, the statement at line 63 prevents decomposition in *i*. One solution, shown in Example 2-5, is to decompose *x* in *i* and *y* in *j*.

```
[transfer.inc]
 1        integer im, jm
 2        common /sizes_com/ im, jm
 3
 4  CSMS$DECLARE_DECOMP(DECOMP_I, 1)
 5  CSMS$DECLARE_DECOMP(DECOMP_J, 1)
 6

[transfer.f]
 1        program TRANSFER1
 2        implicit none
 3
 4        include 'transfer.inc'
 5
 6        integer i
 7        integer j
 8
 9        im = 60
10        jm = 90
```

```

CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
CSMS$CREATE_DECOMP(DECOMP_J, <jm>, <0>)

      call DO_IT

      end



      subroutine DO_IT
      include 'transfer.inc'

CSMS$DISTRIBUTE(DECOMP_I, 1) BEGIN
      real x(im,jm)
CSMS$DISTRIBUTE END

CSMS$DISTRIBUTE(DECOMP_J, 2) BEGIN
      real y(im,jm)
CSMS$DISTRIBUTE END

C BEGIN

      x = 1.0

CSMS$PARALLEL(DECOMP_I, <i>) BEGIN

C dependence in the j dimension that
C SMS does not provide directives to parallelize
      do j = 2, jm
        do i = 1, im
          x(i,j) = x(i,j) + x(i,j-1)
        end do
      end do
CSMS$PARALLEL END

CSMS$TRANSFER(<X, Y>) BEGIN
      do j = 1, jm
        do i = 1,im
          y(i,j) = x(i,j)
        end do
      end do
CSMS$TRANSFER END



CSMS$PARALLEL(DECOMP_J, <j>) BEGIN

C dependence in the i dimension that
C SMS does not provide directives to parallelize
      do j = 1, jm
        do i = 2, im
          y(i,j) = y(i,j) + y(i-1,j)
        end do
      end do
CSMS$PARALLEL END

      open(10,file='f1',form='unformatted')
```

```
69        write(10) y
70        close(10)
71
72        return
73        end
```

Example 2-5: A simple SMS parallel program that requires two data decompositions due to recurrence relations in *i* for array *x* and *j* for array *y*.

Example 2-5 contains two DECLARE_DECOMP and CREATE_DECOMP directives. The DISTRIBUTE directive at line 24 uses ***DECOMP_I*** to decompose *x* in *i*. The DISTRIBUTE directive at line 28 uses ***DECOMP_J*** to decompose *y* in *j*. The TRANSFER directive at line 47 causes SMS to replace the serial code between the BEGIN and END TRANSFER directives (a simple copy) with communication that re-distributes (transposes) the data among the processes as illustrated in Figure 2-13. *x* is referred to as the source array of the TRANSFER directive and *y* is referred to as the destination array.

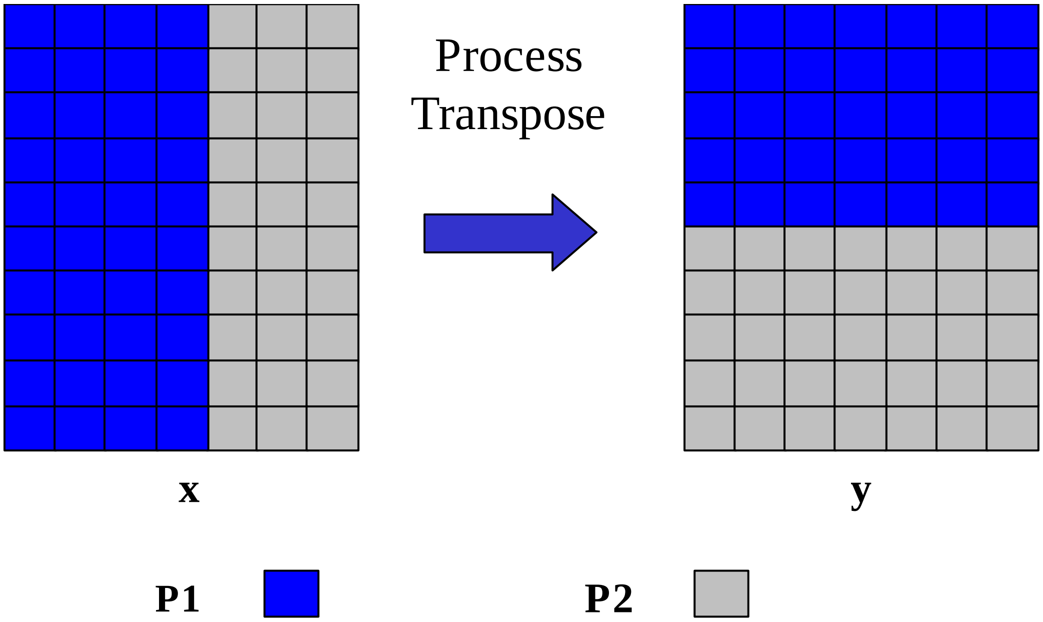


Figure 2-13: An illustration of the data movement required between processes P1 and P2 for a transposition operation.

3 Decomposing Arrays and Parallelizing Loops

3.1 Choosing Decompositions

In order to choose domain decompositions that will yield optimal performance, the dependencies of arrays on one another must be analyzed. Usually, several decomposition options are possible. Decompositions of 3D arrays supported by SMS are shown in Figure 3-1. Dependence analysis is used to help pick optimal decompositions that will minimize inter-process communication. Typical explicit FDA models will be optimally decomposed in one or both of the horizontal dimensions as illustrated by "a", "b", or "d" of Figure 3-1. All of these decompositions may be used by spectral models which are described in Section 6.2.

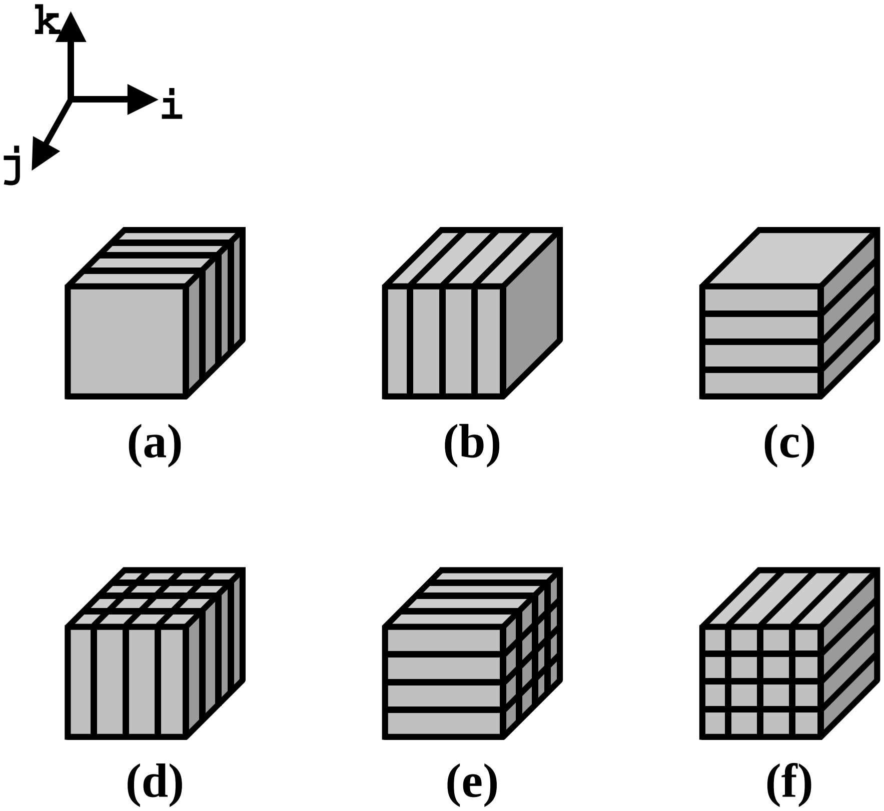


Figure 3-1: Decompositions of three-dimensional arrays supported by SMS.

Other issues to consider when selecting decompositions are the architecture of the machine on which the program will most likely be run and how many processes will be available. For vector machines, it is best to leave the inner dimension non-decomposed when possible to maximize vector lengths. On cache-based machines, it may be best to decompose the inner dimension instead. For example, in Figure 3-1, decomposition "a" would preserve long vector lengths while decomposition "b" would not. In addition, when the number of processes available is larger than the number of grid points in the single decomposed dimension, two dimensions should be decomposed.

3.2 Two-Dimensional Decompositions

The full power of the DECLARE_DECOMP, CREATE_DECOMP, DISTRIBUTE, and PARALLEL directives becomes more apparent when two dimensions are decomposed. Consider the following example:

```
[Include file:  decomp_ex1.inc]

      integer im, jm, km
      common /sizes_com/ im, jm, km
CSMS$DECLARE_DECOMP(DECOMP_IJ, 2)


[Source file:  decomp_ex1.f]

      program decomp_ex1
      include 'decomp_ex1.inc'
      im = 15
      jm = 10
      km = 2
CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
      call compute
      end

      subroutine compute
      include 'decomp_ex1.inc'
      integer i, j, k
CSMS$DISTRIBUTE(DECOMP_IJ, 1, 2) BEGIN
      integer z(im,jm,km)
CSMS$DISTRIBUTE END
      integer zsum
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do 100 k=1,km
      do 100 j=1,jm
      do 100 i=1,im
        z(i,j,k) = 1
  100 continue
      zsum = 0
      do 200 k=1,km
      do 200 j=1,jm
      do 200 i=1,im
        zsum = zsum + z(i,j,k)
  200 continue
CSMS$PARALLEL END
CSMS$REDUCE(zsum, SUM)
      print *,'zsum = ',zsum
      return
      end
```

Example 3-1: An SMS program that uses a two dimensional decomposition.

When run, the serial version of this program prints the following:

```
>> decomp_ex1
 zsum =          300
```

Directives CREATE_DECOMP, DISTRIBUTE, and PARALLEL now have more complex arguments than in the simple examples from Section 2.3. The second argument to CREATE_DECOMP, ***<im, jm>***, indicates that the decomposition named ***DECOMP_IJ*** has two decomposed dimensions and that the global size of the first decomposed dimension is *im* and the global size of the second decomposed dimension is *jm*. The third argument, ***<0,0>***, indicates that ***DECOMP_IJ*** has no halo regions in either decomposed dimension.

The second argument to DISTRIBUTE, ***1***, indicates that the first dimension of array *z* is decomposed as described by the first decomposed dimension of ***DECOMP_IJ.*** The third argument, ***2***, indicates the second dimension of array *z* is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The third dimension of array *z* will not be decomposed. This is decomposition "d" in Figure 3-1. More details about DISTRIBUTE can be found in Section 3.4.1.

The second argument to PARALLEL, ***<i>***, is used to identify loop indices for loops spanning the first decomposed dimension of ***DECOMP_IJ***. Similarly, the third argument, ***<j>,*** is used to identify loop indices for loops spanning the second decomposed dimension of ***DECOMP_IJ***. The PARALLEL directive will translate both *i* and *j* dimensions of loops 100 and 200 to local loop bounds.

When this code is run on 2 or 3 processes, we see the expected results:

```
>> smsRun -np 2 decomp_ex1_sms
SMS:  Using default process layout (2 x 1) for decomposition decomp_ij
 zsum =  300
>> smsRun -np 3 decomp_ex1_sms
SMS:  Using default process layout (3 x 1) for decomposition decomp_ij
 zsum =  300
```

Note that SMS prints an additional diagnostic message for two-dimensional decompositions. This message describes how many processes are assigned to each decomposed dimension, which can be useful for debugging or performance analysis. For brevity, this message will not be shown again.

3.3 Using Statically Allocated Memory

When dynamic memory allocation is used, SMS automatically sets local array sizes at run-time. However, static memory codes require the local array sizes to be declared by the programmer. In addition, the local and global indices differ (Figure 3-2 below), often necessitating conversions between the two (see Section 4).

Example 3-2 illustrates a program using static memory allocation. In this example, the DECLARE_DECOMP directive requires a new second argument, ***<(im/2)+1, jm/2>.*** This informs SMS that the decomposition named ***DECOMP_IJ*** has two decomposed dimensions and specifies declared local sizes for each. The declared sizes will be used to translate declarations of static arrays enclosed by DISTRIBUTE directives. For instance, *z* will have a size of *(im/2 +1, jm/2, km)* in the translated version of the code in Example 3-2.

```
[Include file:  decomp_ex4.inc]

 1       integer im, jm, km
 2       parameter (im = 15, jm = 10, km = 2)
 3 CSMS$DECLARE_DECOMP(DECOMP_IJ, <(im/2)+1, jm/2>)

[Source file:  decomp_ex4.f]

 4       program decomp_ex4
 5       include 'decomp_ex4.inc'
 6 CSMS$DISTRIBUTE(DECOMP_IJ, 1, 2) BEGIN
 7       integer z(im,jm,km)
 8 CSMS$DISTRIBUTE END
 9       integer zsum, i, j, k
10 CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
11 CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
12       do 100 k=1,km
13       do 100 j=1,jm
14       do 100 i=1,im
15         z(i,j,k) = 1
16   100 continue
17       zsum = 0
18       do 200 k=1,km
19       do 200 j=1,jm
20       do 200 i=1,im
21         zsum = zsum + z(i,j,k)
22   200 continue
23 CSMS$PARALLEL END
24 CSMS$REDUCE(zsum, SUM)
25       print *,'zsum = ',zsum
26       end
```

Example 3-2: An SMS program that uses static memory allocation requires the local sizes be declared in the DECLARE_DECOMP directive. In this example, these local sizes are: *(im/2)+1* **and** *jm/2***.**

In static memory cases, where the number of processes assigned to a decomposed dimension does not evenly divide the global size of that dimension, the declared local sizes specified in the DECLARE_DECOMP directive must be set for the process(es) that use(s) the most memory. For a 4-process run, the term ***(im/2)+1*** (Example 3-2, line 3) ensures there will be sufficient local memory for all processes even though two require local arrays of size *(8,5,2)* while the other two require arrays of size *(7,5,2)*. Figure 3-2 illustrates this point.

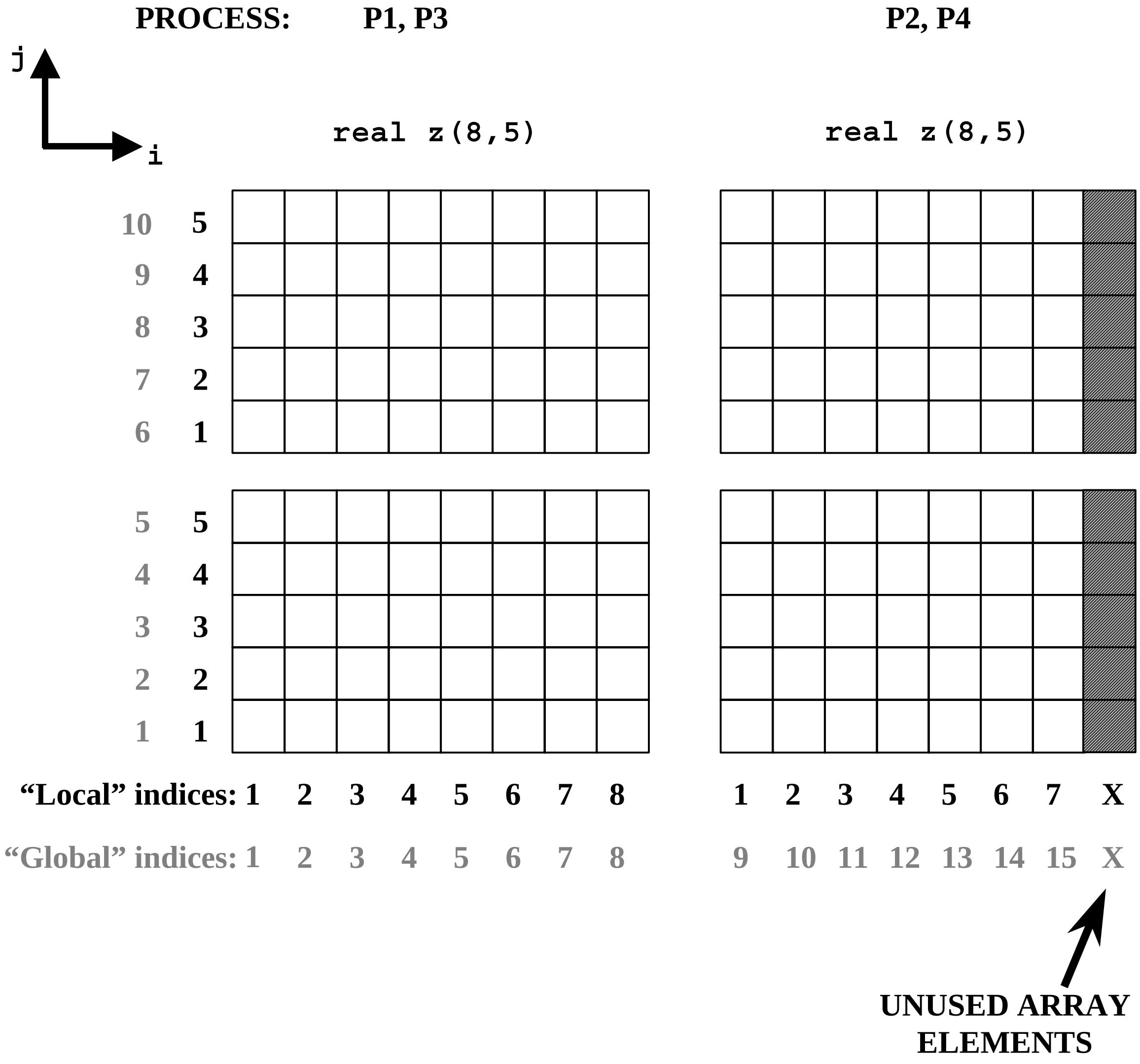


Figure 3-2: For static memory allocation, the size of the decomposed arrays is set in the DECLARE_DECOMP directive based on the number of processes that will be used to run the program. Sometimes all the memory declared will not be used as illustrated for processes P2 and P4. Processes P2, P3, and P4 have local indices that are different from the corresponding global indices of array `z`. (Non-decomposed dimension "k" is not shown.)

A run on 4 processes yields the correct results. A run made on 8 processes also works. Why? In this case, SMS assigns processes as shown in Figure 3-3. The largest local array sizes required on any process for the eight-process run are*(4,5,2)*. So the declared local array sizes are big enough to hold the translated arrays and the program runs as expected. However, it wastes memory because only half of each declared array is ever used *(1:4,*,*)*.

In addition to wasting memory, performance of the 8-process run might not be optimal on a cache-based machine because the data used in each array are scattered over a block of memory twice the needed size. This will likely result in more cache misses and may significantly degrade performance. Further, this effect becomes more severe as the number of processes increases. For example, if the program were run on 32 processes, the largest local array sizes required on any process would be only *(2,3,2)*. Therefore, it is especially important to declare arrays using the smallest possible sizes for large numbers of processes.

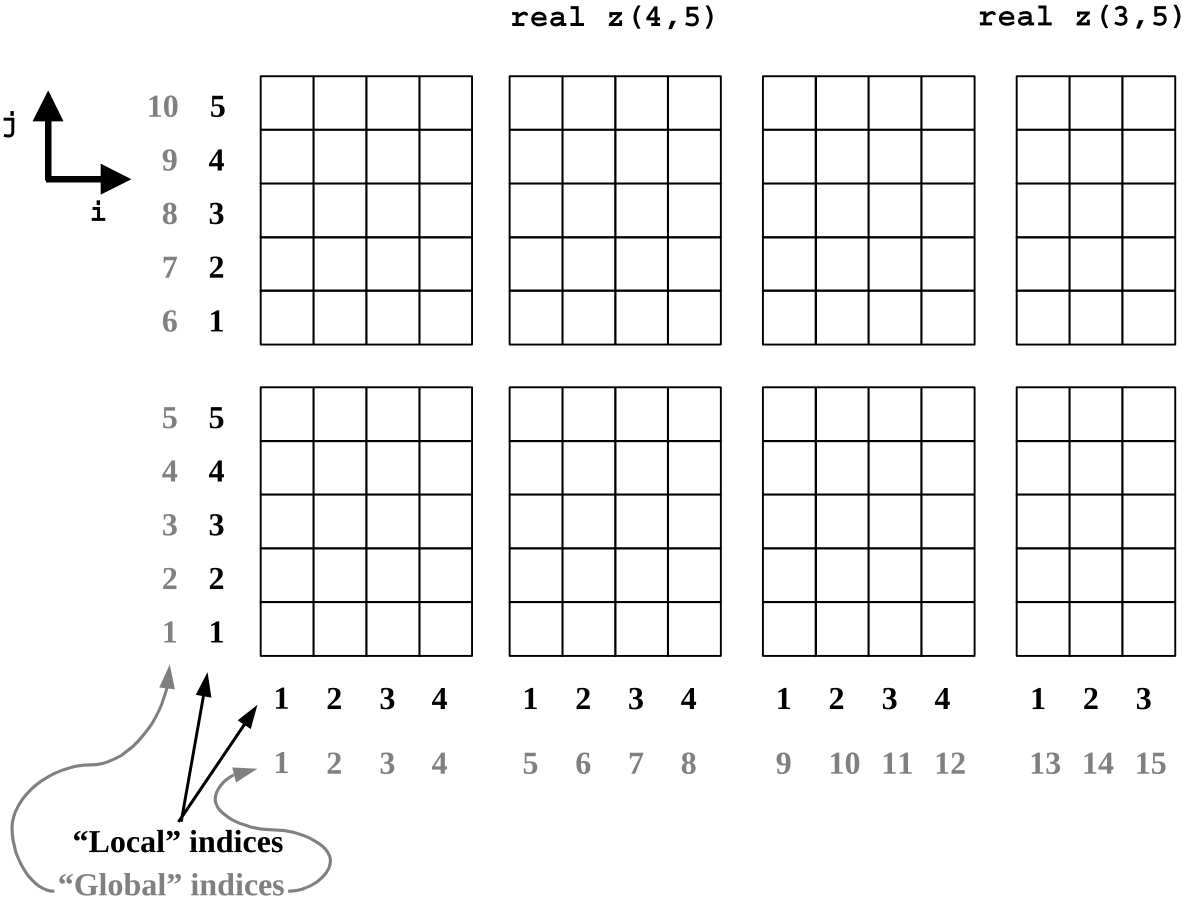


Figure 3-3: Memory layout for 8-process run.

Running Example 3-2 on 2 processes produces the following:

```
>> smsRun -np 2 decomp_ex4_sms
 Process:   1 Error at: ./decomp_ex4_sms.f:10.1
 Process:   1 Error status=   -2202 : USER DECLARED STATIC ARRAY IS TOO
SMALL.
 Process:   1 Aborting...
```

What happened? By default, the two processes are distributed along the *i* dimension so the largest local array sizes required on any process for the two-process run is *(8,10,2)*. However, the DECLARE_DECOMP directive set local array sizes to *((im/2)+1,jm/2,km) = (8,5,2)* which is too small for the two process run (see Figure 3-2). SMS detects this error at run time, prints the error messages, and aborts the program.

To provide sufficient memory for the local arrays in a two-process run, we can modify the sizes in the DECLARE_DECOMP directive as follows:

```
CSMS$DECLARE_DECOMP(DECOMP_IJ, <(im/2)+1, jm>)
```

If the following DECLARE_DECOMP directive were used

```
CSMS$DECLARE_DECOMP(DECOMP_IJ, <im, jm>)
```

all translated arrays would be declared full-size. This code could then be run on any number of processes (provided each process has enough memory). This is very useful during debugging because it allows comparison of results for runs made on different numbers of processes. Once debugging is complete, the DECLARE_DECOMP directives should be changed to minimize memory use.

Determining the proper local sizes for static memory models that need EXCHANGE directives will be discussed in Section 5.1.3.

3.4 More about DISTRIBUTE

3.4.1 Further Detail on DISTRIBUTE Syntax

This section explains the distinction between “dimension of an array” and “dimension of an SMS decomposition”. The DISTRIBUTE directive can decompose several types of arrays as shown the in the following code fragments:

```
CSMS$DISTRIBUTE(DECOMP_IJ, 1, 2) BEGIN
      integer x(im,jm,km)
CSMS$DISTRIBUTE END
```

Here, the first dimension of array *x* is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the second dimension of array *x* is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The third dimension of array *x* is not decomposed.

```
CSMS$DISTRIBUTE(DECOMP_IJ, 1, 3) BEGIN
      real a(im,km,jm)
CSMS$DISTRIBUTE END
```

The numbers *1* and *3* refer to array dimensions. The order in which they appear determines the dimensions of the decomposition to which they refer. Here, the first dimension of array *a* is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the third dimension of array *a* is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The second dimension of array *a* is not decomposed.

```
CSMS$DISTRIBUTE(DECOMP_IJ, 3, 2) BEGIN
      real b(km,jm,im), avg
CSMS$DISTRIBUTE END
```

Here, the third dimension of array *b* is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the second dimension of array *b* is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***. The first dimension of array *b* is not decomposed. *avg* is not decomposed at all because it is a scalar variable.

The user can also specify how variables are distributed by using variable name tags instead of dimension numbers. For example,

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real x(im, jm, km)
CSMS$DISTRIBUTE END
```

again indicates the first dimension of array *x* is decomposed as described by the first decomposed dimension of ***DECOMP_IJ*** and the second dimension of array *x* is decomposed as described by the second decomposed dimension of ***DECOMP_IJ***.

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real b(km, jm, im)
CSMS$DISTRIBUTE END
```

indicates that the third dimension of *b* is distributed based on the first decomposed dimension and the second dimension of *b* is distributed based on the second decomposed dimension.

Using this syntax, it is possible to enclose the last two arrays inside the same distribute directive:

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real x(im, jm, km)
      real b(km, jm, im)
CSMS$DISTRIBUTE END
```

3.4.2 Using DISTRIBUTE to Define Decomposed Boundary Arrays

Regional weather forecast and ocean models often require boundary condition data. A code segment handling western boundary conditions might look as shown in Example 3-3.

```
      subroutine UPDATE_BOUNDARIES(u)
      integer, parameter :: im = 10
      integer, parameter :: jm = 20
      integer, parameter :: km = 30
csms$declare_decomp(dh, 2)

csms$distribute(dh, 1, 2) begin
```

```
 8       real u(im, jm, km)
 9 csms$distribute end
10
11 csms$distribute(dh, , 1) begin
12       real ubw(jm, km)
13 csms$distribute end
14
15       open(10, file='west_bdy', form='unformatted')
16       read(10) ubw
17       close(10)
18
19 csms$parallel(dh, , <j>) begin
20 csms$global_index(1) begin
21       do k = 1, km
22         do j = 1, jm
23           u(1, j, k) = (u(1, j, k) + ubw(j,k))/2.0
24         end do
25       end do
26 csms$global_index end
27 csms$parallel end
28
29       return
30       end
```

Example 3-3: Subroutine showing how boundary condition arrays can be handled in SMS.

The DISTRIBUTE statement on line 11 defines an unusual kind of decomposed array. Its first dimension is decomposed according to the second dimension of decomposition *dh* but none of the array dimensions are decomposed based on the first decomposed dimension. When distribution of an array does not involve all decomposed dimensions, the distribution is called a "slice" and the array is referred to as a "sliced array". Since the exact manner in which sliced arrays are distributed is somewhat poorly defined, care must be taken when using them. Limitations of sliced arrays are described in detail in Section 13. The PARALLEL directive syntax on line 19 will be discussed in Section 3.5.

3.5 More About PARALLEL

The PARALLEL directive will translate serial loops correctly provided the upper and lower loop bounds are valid global indices. For example, the *i* and *j* loops below would all be correctly translated:

```
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do 100 k=1,km
      do 200 j=3,jm-2
      do 200 i=3,im-2
        z(i,j,k) = x(i,j,k) + y(i,j,k)
  200 continue

      do 210 j=1,2
      do 210 i=1,im
        z(i,j,k) = 0
  210 continue

      do 220 j=jm-1,jm
      do 220 i=1,im
        z(i,j,k) = 0
  220 continue
```

```
      do 230 j=1,jm
      do 230 i=1,2
        z(i,j,k) = 0
  230 continue

      do 240 j=1,jm
      do 240 i=im-1,im
        z(i,j,k) = 0
  240 continue

  100 continue
CSMS$PARALLEL END
```

In this code fragment, notice that the translated version of loop 210 would only be executed on processes that contain global indices *j=1* or *j=2*. The PARALLEL directive ensures that other processes will skip loop 210. Similar translations will occur for the other loops.

Recall the syntax seen on lines 19-22 of Example 3-3.

```
CSMS$PARALLEL(dh, , <j>) BEGIN
      do k = 1, km
        do j = 1, jm
```

It indicates that no enclosed loops correspond to the first decomposed dimension but any loops that use index *j* correspond to the second decomposed dimension and should be translated.

There is no run-time performance penalty for using a PARALLEL directive because processes are not synchronized. Also, PARALLEL directives may enclose any valid Fortran executable statements. Therefore, a program that uses only one decomposition will usually require no more than one BEGIN-END pair of PARALLEL directives for each program unit (subroutine, function, or main program).

In tagging loop indices to be translated, some care is required. First, indices can sometimes be used for non-decomposed loops as well as for loops that span decomposed dimensions. This is the case in the following fragment:

```
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do 200 k=1,km
      do 200 j=1,jm
      do 200 i=1,im
        z(i,j,k) = x(i,j,k) + y(i,j,k)
  200 continue
      do 500 i=1,3
        call smooth(z)
  500 continue
CSMS$PARALLEL END
```

In this case, loop 500 is used to repeatedly call subroutine *smooth* which performs some computations on decomposed array *z*. This loop should NOT be translated because *i* is being used as an iteration count, not as an index into a decomposed dimension. This is easily fixed either by using a loop index other than `i` or `j` in loop 500 or by moving the PARALLEL END directive to exclude loop 500.

Second, make sure that all loops manipulating decomposed arrays are enclosed inside PARALLEL directives. During translation, PPP will issue a warning message whenever it finds a loop that is not enclosed by PARALLEL directives if that loop contains a decomposed array:

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

3.6 Arrays with Non-Unit Lower Bounds

When arrays in the serial code are declared with non-unit lower bounds, the SMS decomposition must reflect this fact. Consider the following variant of Example 3-1:

```
[Include file:  decomp_ex6.inc]

      integer im, jm, km
      common /sizes_com/ im, jm, km
CSMS$DECLARE_DECOMP(DECOMP_IJ : <0,0>)

[Source file:  decomp_ex6.f]

      program decomp_ex6
      include 'decomp_ex6.inc'
      im = 15
      jm = 10
      km = 2
CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
      call compute
      end

      subroutine compute
      include 'decomp_ex6.inc'
      integer i, j, k
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      integer z(0:im-1,0:jm-1,0:km-1), zsum
CSMS$DISTRIBUTE END
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do 100 k=0,km-1
      do 100 j=0,jm-1
      do 100 i=0,im-1
        z(i,j,k) = 1
  100 continue
      zsum = 0

      do 200 k=0,km-1
      do 200 j=0,jm-1
      do 200 i=0,im-1
        zsum = zsum + z(i,j,k)
  200 continue

CSMS$PARALLEL END
CSMS$REDUCE(zsum, SUM)
      print *,'zsum = ',zsum
      return
      end
```

In this program, array *z* is declared so the first index (lower bound) is zero in each dimension instead of the Fortran default of one. The bounds of loops 100 and 200 now start at zero. The only difference between the directives in this example and those in Example 3-1 is DECLARE_DECOMP. The new final argument, *<0,0>* indicates that array declarations have a lower bound of zero in both decomposed dimensions.

3.7 Aligned Decompositions

Sometimes, arrays with different global sizes are aligned so a given coordinate corresponds to the same physical grid point in all of them. This may occur in atmospheric models based on staggered grids where the wind arrays are slightly different than the mass (temperature, pressure) arrays. Sometimes, sizes of aligned arrays can be very different. One example of this case is a coupled ocean-ice model where the ice model arrays are only defined in the northern latitudes. Aligned arrays are illustrated in Figure 3-4.

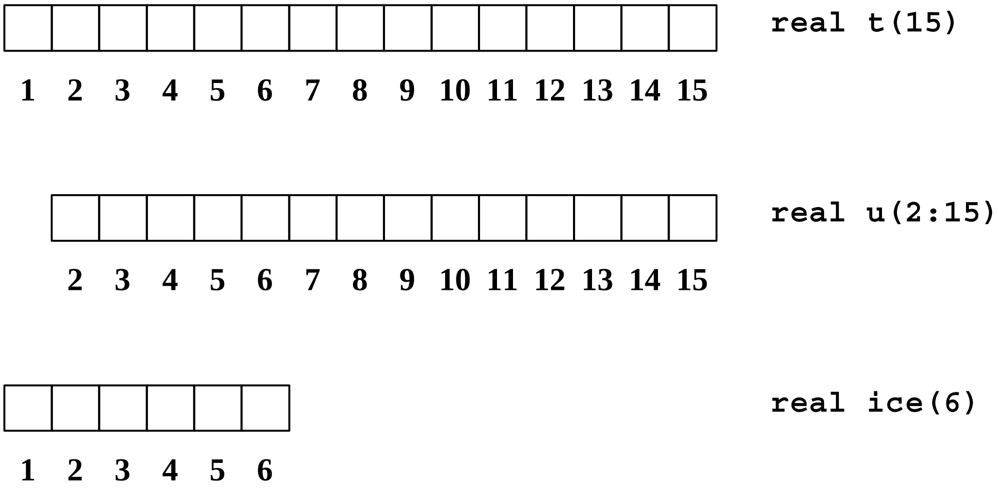


Figure 3-4: Aligned arrays *t*, *u* and *ice* in serial code. Arrays *u* and *ice* are aligned with "parent" array *t*. Arrays *t* and *ice* have significantly different sizes.

SMS supports aligned arrays by allowing decompositions to be aligned. A decomposition is said to be "aligned" with another decomposition if all of the global indices in each dimension of the aligned decomposition correspond to identical global indices in a second "parent" decomposition. In particular, identical global indices in both decompositions will always be assigned to the same process. Example 3-4 shows how the DECLARE_DECOMP and CREATE_DECOMP directives can be used to create aligned decompositions. These decompositions are illustrated in Figure 3-5.

```
      program decomp_ex8
      integer im, im_ice
      parameter (im= 15, im_ice = 6)
CSMS$DECLARE_DECOMP(dh_parent, <im/3 + 2>)
CSMS$DECLARE_DECOMP(dh_aligned_u : <2>, ALIGNED = dh parent)
CSMS$DECLARE_DECOMP(dh_aligned_ice: ALIGNED = dh_parent)
      integer i
CSMS$DISTRIBUTE(dh_parent, 1) BEGIN
```

```
      integer t(im)
CSMS$DISTRIBUTE END
CSMS$DISTRIBUTE(dh_aligned_u, 1) BEGIN
      integer u(2:im)
CSMS$DISTRIBUTE END
CSMS$DISTRIBUTE(dh_aligned_ice, 1) BEGIN
      integer ice(im_ice)
CSMS$DISTRIBUTE END
      integer isum

CSMS$CREATE_DECOMP(dh_parent, <im>, <1>)
CSMS$CREATE_DECOMP(dh_aligned_u, <im-1>)
CSMS$CREATE_DECOMP(dh_aligned_ice, <im_ice>)
CSMS$PARALLEL(dh_parent, <i>) BEGIN
      do 100 i=1,im
        t(i) = 1
  100 continue
      do 200 i=2,im
        u(i) = 2
  200 continue
CSMS$EXCHANGE(u)
      do 300 i=1,im_ice
        ice(i) = t(i) + u(i+1)
  300 continue
      isum = 0
      do 400 i=1,im_ice
        isum = isum + ice(i)
  400 continue
CSMS$PARALLEL END
CSMS$REDUCE(isum, SUM)
      print *,'isum = ',isum
      end
```

Example 3-4: An SMS program that uses aligned decompositions.

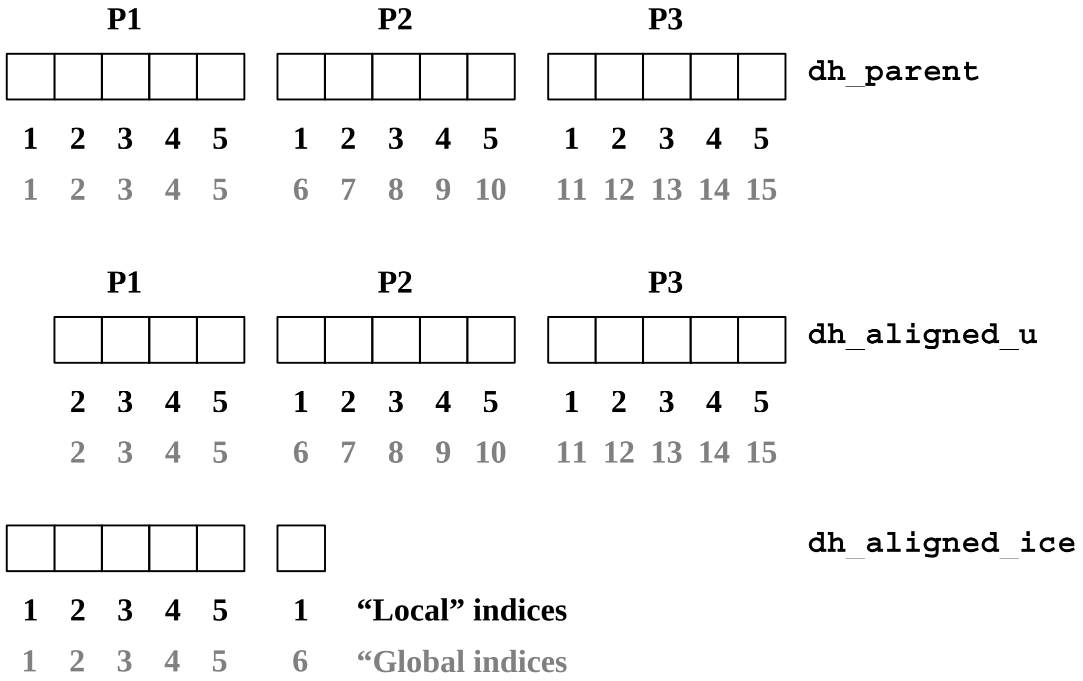


Figure 3-5: Memory layout for parent decomposition *`dh_parent`* and aligned decompositions *`dh_aligned_u`* and *`dh_aligned_ice`* when three processes are used. Halo regions are not shown.

The optional "ALIGNED" keyword is used in the DECLARE_DECOMP directive to indicate that the declared decomposition is aligned with another "parent" decomposition. Any number of decompositions may be aligned with the same parent. A parent decomposition must not be aligned. Also, every global index of any aligned decomposition must be also be a valid global index in its parent. In the DECLARE_DECOMP directive, declared local size is not specified for an aligned decomposition that uses statically allocated memory. SMS automatically uses the parent's declared local size in this case. (The above conditions ensure that an aligned decomposition will never require more local storage than its parent.) Also, aligned arrays can have different lower bounds than their parent as shown on line 5 of Example 3-4. Note that the "ALIGNED" option always comes last when it is used with the lower bounds option. Lower bounds can be omitted for an aligned decomposition if they are identical to the parent's lower bounds as shown in line 6.

The syntax of the CREATE_DECOMP directive is also slightly different for aligned decompositions. The halo thickness is left out because SMS automatically uses the parent's halo thickness. Index scrambling is not currently supported for aligned decompositions or for the parent of an aligned decomposition.

Note that the parent decomposition is used in the PARALLEL directive in Example 3-4. It is always correct to use the parent decomposition in the PARALLEL directive. In fact, aligned decompositions should only appear in DECLARE_DECOMP, CREATE_DECOMP, and DISTRIBUTE directives.

Finally, it is possible to create invalid decompositions when using aligned arrays. This will occur whenever any process has an interior region that ends up being smaller than the halo region. (SMS does not support this because halo update communication is inefficient.) For example, if line 19 of Example 3-4 were modified to increase the halo sizes of all three decompositions to two then the interior of process P2 would be smaller than the halo thickness for decomposition *dh_aligned_ice* (see Figure 3-5). SMS will detect this error at run time and print the following message:

```
Aligned decomposition causes an interior to be smaller than the halo.
```

The simplest solution to this problem is to use a process configuration file to adjust the parent decomposition so the interior size in the aligned decomposition is increased (see Section 10.1).

4 Translating Array Indices

We have seen that, in static memory models, local and global indices are different whenever more than one process is used so conversions between them are required. TO_GLOBAL (Section 4.1) and TO_LOCAL (Section 4.2) provide support for these conversions. In addition, it is sometimes desirable to generate process-local loop start and end indices and array sizes to simplify parallelization of subroutines in both dynamic and static memory codes. The TO_LOCAL (Section 4.3) directive does this as well. Finally, boundary condition calculations must be restricted to processes containing boundary points. GLOBAL_INDEX (Section 4.4) handles these cases.

4.1 Translating Local Indices to Global Indices

For a static memory code, when a loop has been translated using the PARALLEL directive, the value of the index is now process-local as illustrated in Figure 3-2. If the intent of the program is to access the global value, this index will need to be translated back to a global value. This point is illustrated in Example 4-1.

```
 1       program tran_index1
 2       implicit none
 3       integer i, j
 4       integer, parameter :: im = 5
 5       integer, parameter :: jm = 3
 6 CSMS$DECLARE_DECOMP(DECOMP_IJ, <im, jm>)
 7
 8 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
 9       integer x(im,jm)
10 CSMS$DISTRIBUTE END
11
12 CSMS$CREATE_DECOMP(DECOMP_IJ, <im, jm>, <0,0>)
13
14 CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
15
16       do 100 j=1,jm
17       do 100 i=1,im
18           x(i,j) = (100 * i) + j
19   100 continue
20
21 CSMS$SERIAL BEGIN
22       do j = 1, jm
23         write(*,'(16i5)') (x(i,j),i=1,im)
24       end do
25 CSMS$SERIAL END
26 CSMS$PARALLEL END
27
28       end
```

Example 4-1: An SMS parallel program that incorrectly initializes array *x* in loop 100.

This program initializes array *x* in loop 100 (lines 16-19). Each element of array *x* is then printed on the screen. When the serial code is run, the following is printed on the screen:

```
>> tran_index1
```

```
  101  201  301  401  501
  102  202  302  402  502
  103  203  303  403  503
```

The same result is seen when the SMS version is run on one process. However, the results are incorrect when two processes are used:

```
>> smsRun -np 2 tran_index1_sms
  101  201  301  101  201
  102  202  302  102  202
  103  203  303  103  203
```

Why are the results incorrect? The PARALLEL directive has translated the *i* and *j* indices used to compute *x* in loop 100 using local indices. However, correct operation requires that *x* be initialized using global indices as in the original serial code. The solution is to use the TO_GLOBAL directive to translate the local indices to global indices. In this case, line 18 would be replaced with the following code:

```
CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
        x(i,j) = (100 * i) + j
CSMS$TO_GLOBAL END
```

The first argument in the TO_GLOBAL directive, ***<1,i>,*** indicates that array index *i* is an index in the first decomposed dimension. The second argument, ***<2,j>,*** indicates that array index *j* is an index in the second decomposed dimension. All occurrences of indices *i* and *j* inside the TO_GLOBAL directives that are not array references will be converted to their global equivalents in the first and second decomposed dimensions, respectively. Running the new parallel code on various numbers of processes will now yield the same result as the serial run.

Note that the TO_GLOBAL directive must appear within a PARALLEL directive. Directives TO_LOCAL and GLOBAL_INDEX, introduced later in this section, also have this restriction. Also note that since *x* is decomposed, the SERIAL directive is required to handle the write statement on line 24 as will be explained in Section 8.

4.2 Translating Global Indices to Local Indices Inside Loops

Sometimes, indices that have been translated to global values need to be translated back to local values to be used as indices into decomposed arrays in a static memory code. The TO_LOCAL directive is used for this translation. Consider the following code fragment that uses computed indices to avoid out-of-bounds references:

```
CSMS$PARALLEL(DECOMP_IJ, <i>, <j>) BEGIN
      do j=1,jm
        do i=1,im
CSMS$TO_GLOBAL(<1,i>) BEGIN
CSMS$TO_LOCAL(<1,im1,ip1>) BEGIN
          im1 = max( 1,i-1)
          ip1 = min(im,i+1)
CSMS$TO_LOCAL END
CSMS$TO_GLOBAL END
```

```
          x(i,j) = y(i,j) - y(im1,j) - y(ip1,j)
        end do
      end do
CSMS$PARALLEL END
```

The max and min functions compare index *i* with global index values *1* and *im*. Therefore, the TO_GLOBAL directive must be used. The TO_GLOBAL directive will convert *i-1* and *i+1* to global values so *ip1* and *im1* will be computed as global indices. However, *ip1* and *im1* are then used as indices into decomposed array *x*, so they must be converted back from global to local values to avoid out-of-bounds array references for multi-process runs. The TO_LOCAL directive shown accomplishes this. The first argument in the TO_LOCAL directive, ***<1,im1,ip1>,*** indicates that array indices *im1* and *ip1* are both used in loops that span the first decomposed dimension. In this example, occurrences of either index in code enclosed by the TO_LOCAL directives that are not array references will be converted to their local equivalents in the first decomposed dimension.

4.3 Using TO_LOCAL to Generate Process-Local Sizes and Loop Bounds

In many models, large sections of code contain no dependencies that require communications (typically weather model physics routines). If the array bounds and loop limits are passed into these routines, SMS provides a means to parallelize them without inserting directives into the code. Example 4-2 shows such a case.

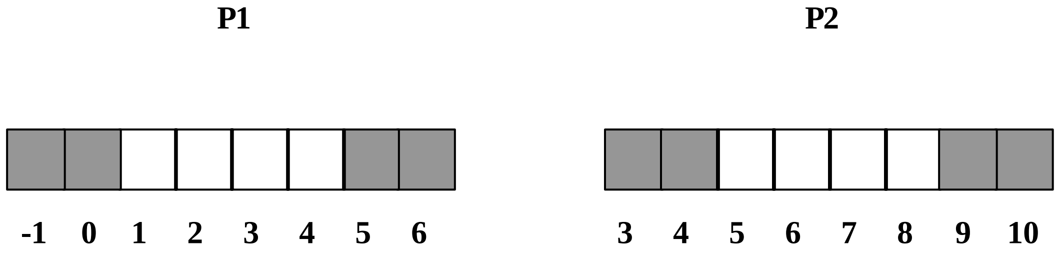
```
      program AVOID_DIRECTIVES
      implicit none
      integer i
      integer, parameter :: im = 8

CSMS$DECLARE_DECOMP(dh, 1)

     integer istart, iend

CSMS$DISTRIBUTE(dh, 1) BEGIN
      integer, allocatable :: x(:), y(:)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(dh, <im>, <2>)

      allocate(x(im))
      allocate(y(im))
      x = 0.0

CSMS$PARALLEL(dh,<i>) BEGIN

      do i=1,im
CSMS$TO_GLOBAL(<1,i>) BEGIN
        x(i) = i
CSMS$TO_GLOBAL END
      end do

      y = 0.0


csms$to_local(<1, istart : lbound>, <1, iend : ubound>) begin
```

```
32       istart = 1
33       iend   = im - 1
34 csms$to_local end
35
36       call physics(x, lbound(x,1), ubound(x,1), istart, iend, y)
37
38 CSMS$SERIAL BEGIN
39       write(*,'(8i5)') (y(i),i=1,im)
40 CSMS$SERIAL END
41
42 CSMS$PARALLEL END
43       end
44
45       subroutine physics(arr_in, i_mem_start, i_mem_end,
46      &                   dim1_start, dim1_end,
47      &                   arr_out)
48       implicit none
49       integer i_mem_start, i_mem_end
50       integer arr_in(i_mem_start:i_mem_end)
51       integer dim1_start, dim1_end
52       integer arr_out(i_mem_start:i_mem_end)
53
54       integer i
55       do i = dim1_start, dim1_end
56         arr_out(i) = 2.0*arr_in(i)
57       end do
58       return
59       end
```

Example 4-2: Sample code that shows how TO_LOCAL can be used to pass local array bounds and start/end loop limits to subroutines so that no directives need to be added to them.

Program *AVOID_DIRECTIVES* calls subroutine *physics* (line 36), passing the arrays *x* and *y*, the starting and ending addresses of those arrays, and the starting and ending loop limits (*istart*, *iend*) over which the loops in *physics* will span. The TO_LOCAL directive at lines 31-34 converts the dimensions and loop limits to their process local values. During source code translation, the syntax **<1, istart : lbound>** causes replacement of instances of *istart* with the local index of the first interior point for the first decomposed dimension for the given process. Figure 4-1 shows all the sizes and bounds for this case, assuming the program is run on 2 processes.

The result is that, inside subroutine *physics*, *i_mem_start*, *i_mem_end*, *dim1_start*, *dim1_end*, *dim2_start*, and *dim2_end* all have the correct process-local values. Consequently, subroutine *physics* produces the right answer for any process decomposition, even though it contains no SMS directives.



Process	i_mem_start	i_mem_end	dim1_start	dim1_end
P1	-1	6	1	4
P2	3	10	5	8

Figure 4-1: Process layout and memory bounds of arrays *`arr_in`* and *`arr_out`* and loop bounds for a 2-process run of Example 4-2.

4.4 Using Global Index to Handle Boundary Conditions

Consider the following code fragment that is enclosed in a PARALLEL directive but is not inside a loop:

```
      id = 5
      jd = 4
      x(id,jd) = 10
```

The following use of TO_LOCAL would be incorrect:

```
CSMS$TO_LOCAL(<1,id>,<2,jd>) BEGIN
      id = 5
      jd = 4
CSMS$TO_LOCAL END
      x(id,jd) = 10
```

The translation of *`id`* and *`jd`* from global values to process-local values will work fine on the process that "owns" global point *`(5,4)`*. However, the translation will be erroneous on processes that do not own global point *`(5,4)`* because there is no valid local equivalent of these global coordinates on those processes. In order to restrict the execution of these statements to the process that owns the data, the GLOBAL_INDEX directive must be used as shown below:

```
      id = 5
      jd = 4
CSMS$GLOBAL_INDEX(1,2) BEGIN
      x(id,jd) = 10
CSMS$GLOBAL_INDEX END
```

The GLOBAL_INDEX directive ensures that the execution of the enclosed assignment statement will only be permitted on the process that owns the global point *(id,jd)*. In addition, if index translation is needed, *id* and *jd* will be translated to process-local equivalents. The first argument in the directive, ***1***, indicates that all array indices corresponding to the first decomposed dimension are affected. The second argument, ***2***, indicates that all array indices corresponding to the second decomposed dimension are affected.

Consider the following example that initializes the boundaries of an array that is decomposed in two dimensions:

```
 1       subroutine compute(im,jm)
 2       integer im,jm
 3 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
 4       integer x(im,jm)
 5 CSMS$DISTRIBUTE END
 6       integer i, j
 7 CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
 8       do 100 j=1,jm
 9       do 100 i=1,im
10         x(i,j) = (100 * i) + j
11   100 continue
12       do 110 j=2,jm-1
13 CSMS$GLOBAL_INDEX(1) BEGIN
14         x( 1,j) = 0
15         x(im,j) = 0
16 CSMS$GLOBAL_INDEX END
17   110 continue
18       do 120 i=2,im-1
19 CSMS$GLOBAL_INDEX(2) BEGIN
20         x(i, 1) = 0
21         x(i,jm) = 0
22 CSMS$GLOBAL_INDEX END
23   120 continue
24 CSMS$GLOBAL_INDEX(1,2) BEGIN
25       x( 1, 1) = 0
26       x(im, 1) = 0
27       x( 1,jm) = 0
28       x(im,jm) = 0
29 CSMS$GLOBAL_INDEX END
30
31 CSMS$PARALLEL END
32
33 CSMS$SERIAL BEGIN
34       print *,'ARRAY x:'
35       print *, x
36 CSMS$SERIAL END
37       return
38       end
```

Example 4-3: An SMS subroutine that illustrates the use of the GLOBAL_INDEX directive to initialize array boundaries.

This subroutine initializes array *x* as in previous examples (lines 8-11). It is assumed this is a dynamic memory code so TO_GLOBAL is not required. It then proceeds to set the boundary values of *x* to zero in lines 12 through 28. Three pairs of GLOBAL_INDEX directives handle the necessary translations. The first pair deals with global indices *1* and *im* in loop 110 while the second pair deals with global indices *1* and *jm* in loop 120.

The third pair handles global indices in the four assignment statements on lines 25 through 28. In each case, the enclosed statements are only executed on the appropriate processes. The SERIAL directive on line 33 will be discussed in Section 8.

When the serial and parallel codes are run, the following is printed on the screen (assuming values of `im` and `jm` as in previous examples):

```
 ARRAY x:
    0     0     0     0     0
    0   202   302   402     0
    0     0     0     0     0
```

4.5 Using GLOBAL_INDEX With Aligned Decompositions

Care must be taken when using aligned decompositions in cases where the size of an aligned dimension is significantly smaller than the size of the corresponding dimension in the parent. Problems can occur when handling boundaries of aligned arrays using GLOBAL_INDEX. For example, consider the following code inserted after loop 300 in Example 3-4:

```
  CSMS$GLOBAL_INDEX(<dh_aligned_ice: 1>) BEGIN
        ice(5) = ice(6)
  CSMS$GLOBAL_INDEX END
```

This code will fail at run time because it assumes that global indices *5* and *6* reside on the same process. However, when three processes are used, these global indices reside on different processes as illustrated in Figure 3-5. The simplest solution to this problem is to use a process configuration file to adjust the parent decomposition so the global indices reside on the same process in the aligned decomposition (see Section 10.1).

Finally, notice that the optional syntax that allows a decomposition to be explicitly specified is used in the GLOBAL_INDEX directive above. Whenever GLOBAL_INDEX is used with an aligned array inside a PARALLEL directive that specifies the parent decomposition, (as on line 22 of Example 3-4), the aligned decomposition must be explicitly specified. This requirement will be relaxed in a future release of SMS.

5 Handling Adjacent Dependencies

5.1 Further Details on EXCHANGE

In Section 2.5, we saw how the EXCHANGE directive was used to implement communications needed to resolve adjacent dependencies for a simple one-dimensional decomposition. In this sub-section, we expand on that discussion by examining the treatment of two-dimensional decompositions and larger stencils, and by discussing other miscellaneous details about EXCHANGE.

5.1.1 Using EXCHANGE in the Case of Two-Dimensional Decompositions

We begin by modifying the Laplace program (Example 2-4) introduced in Section 2.5 so that a two dimensional decomposition is used. Two-dimensional data decompositions allow parallel programs to scale to a large number of processes.

```
      program basic_ex_2d_decomp
      include 'basic.inc'
      im = 10
      jm = 10
CSMS$CREATE_DECOMP(DECOMP_I, <im,jm>, <1,1>)
      call laplace
      end

      subroutine laplace
      include 'basic.inc'
      integer i, j, iter
      real max_error
      real tolerance
      parameter (tolerance = 0.001)
CSMS$DISTRIBUTE(DECOMP_I, 1, 2) BEGIN
      real f(im,jm), df(im,jm)
CSMS$DISTRIBUTE END
CSMS$PARALLEL(DECOMP_I,<i>, <j>) BEGIN
      do 100 j=1,jm
      do 100 i=1,im
        f(i,j) = 0.0
  100 continue
      do 110 j=1,jm
CSMS$GLOBAL_INDEX(1) BEGIN
        f( 1,j) = 2.0
        f(im,j) = 2.0
CSMS$GLOBAL_INDEX END
  110 continue
      do 120 i=1,im
CSMS$GLOBAL_INDEX(2) BEGIN
        f(i, 1) = 2.0
        f(i,jm) = 2.0
CSMS$GLOBAL_INDEX END
  120 continue
      iter = 0
      max_error = 2.0 * tolerance
C main iteration loop...
      do while ((max_error .gt. tolerance) .and. (iter .lt. 1000))
        iter = iter + 1
        max_error = 0.0
```

```
CSMS$EXCHANGE(f)
        do 200 j=2,jm-1
        do 200 i=2,im-1
          df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) +
     &                          f(i,j+1)) - f(i,j)
  200 continue
        do 300 j=2,jm-1
        do 300 i=2,im-1
          if (max_error .lt. abs(df(i,j))) then
            max_error = abs(df(i,j))
          endif
  300 continue
CSMS$REDUCE(max_error, MAX)
        do 400 j=2,jm-1
        do 400 i=2,im-1
          f(i,j) = f(i,j) + df(i,j)
  400 continue
      enddo

CSMS$PARALLEL END
      print *, 'Solution required ',iter,' iterations'
      print *, 'Final error = ', max_error

      return
      end
```

Example 5-1: Two-dimensional decomposition version of Example 2-4

The CREATE_DECOMP directive now lists two decomposed dimensions (with global sizes *im* and *jm*). The halo width for each dimension is 1 in this case. As discussed in Section 3.2, the DISTRIBUTE, PARALLEL, and GLOBAL_INDEX directives are modified to handle the 2-D decompositions. Although the communication patterns required to support 2-dimensional decompositions are more complex than the 1-dimensional case, SMS hides all of these details. Thus, the EXCHANGE directive is unchanged. Figure 5-1 shows some sample stencils overlaid on a 3x3 process decomposition of the problem. The stencil centered at global coordinate *(3,2)* only requires P1 communicate with P2. However, the stencil centered at global coordinate *(4,4)* requires P5 communicate with both P2 and P4. Figure 5-2 and Figure 5-3 show the full communications pattern for a 2-D exchange. Notice that the corner halo points of the center process are filled with data from the corresponding corner processes in the drawings.

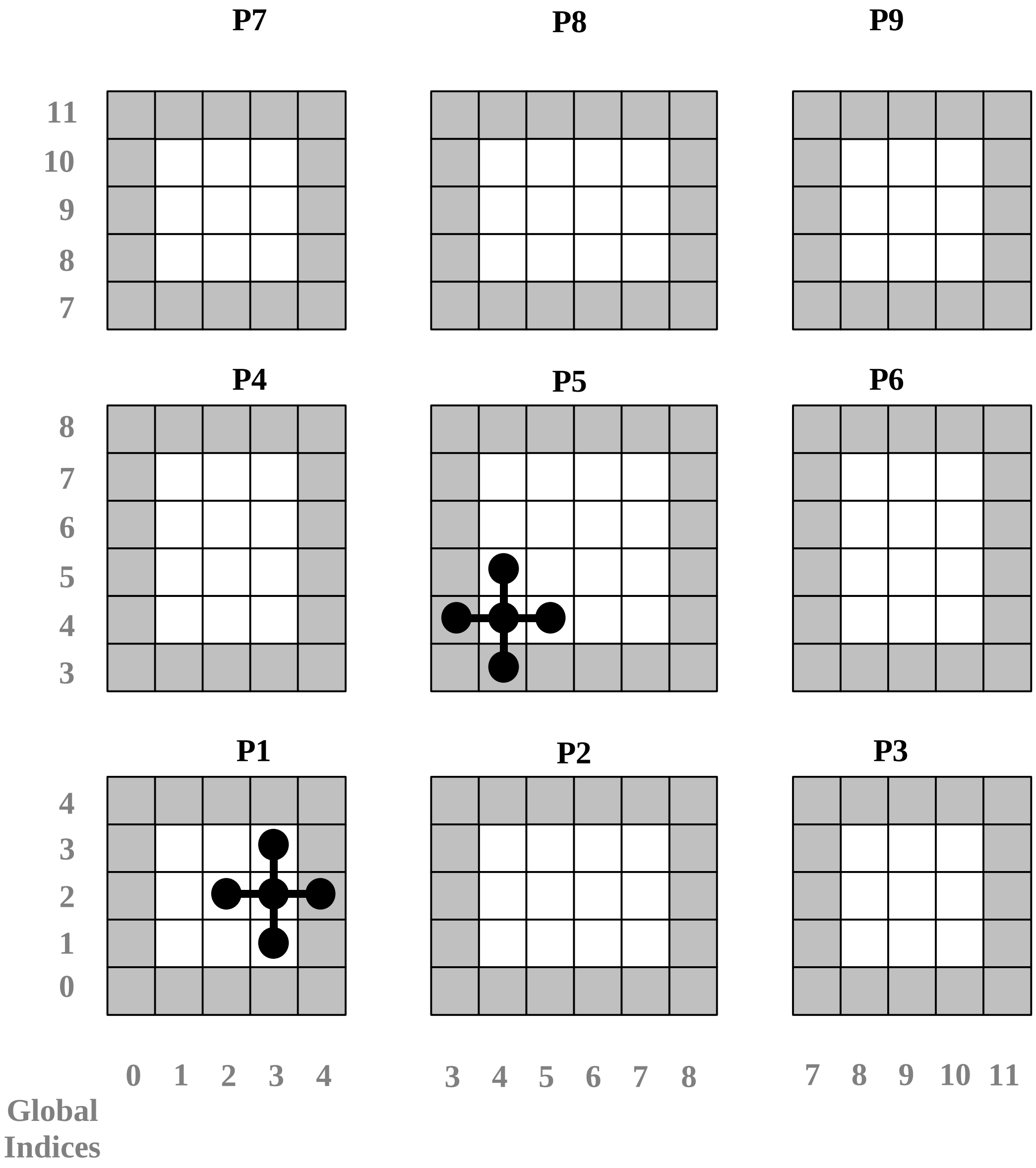


Figure 5-1: Sample stencils overlaid on a 3x3 process decomposition for the Laplace problem. The halo regions are the shaded areas. The white boxes are referred to as the "interior" of each process's sub-domain.

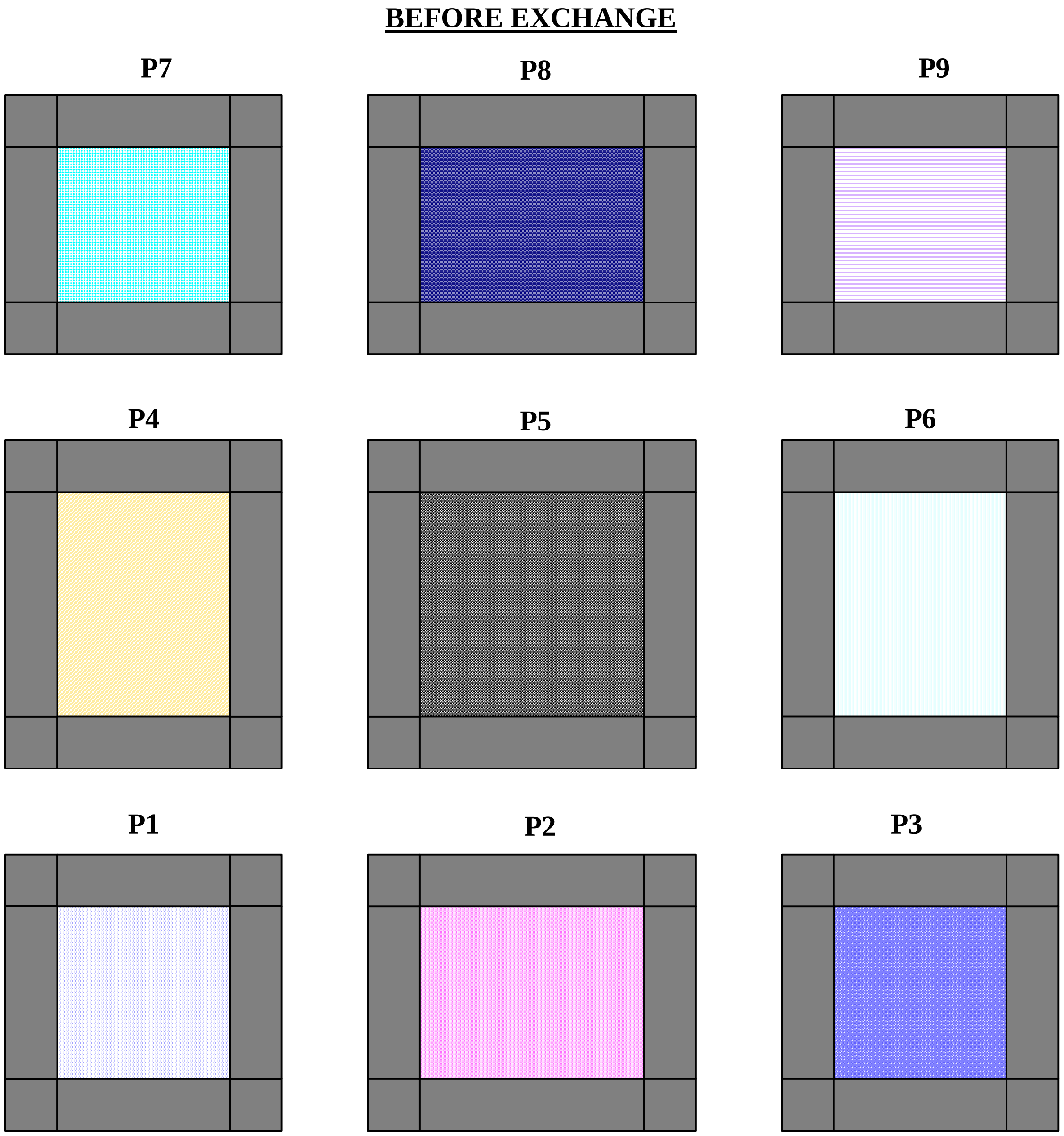


Figure 5-2: Schematic of how data are distributed among 9 processes just prior to an exchange operation. The big boxes contain the interior data. The boxes on the edges are the halo regions.

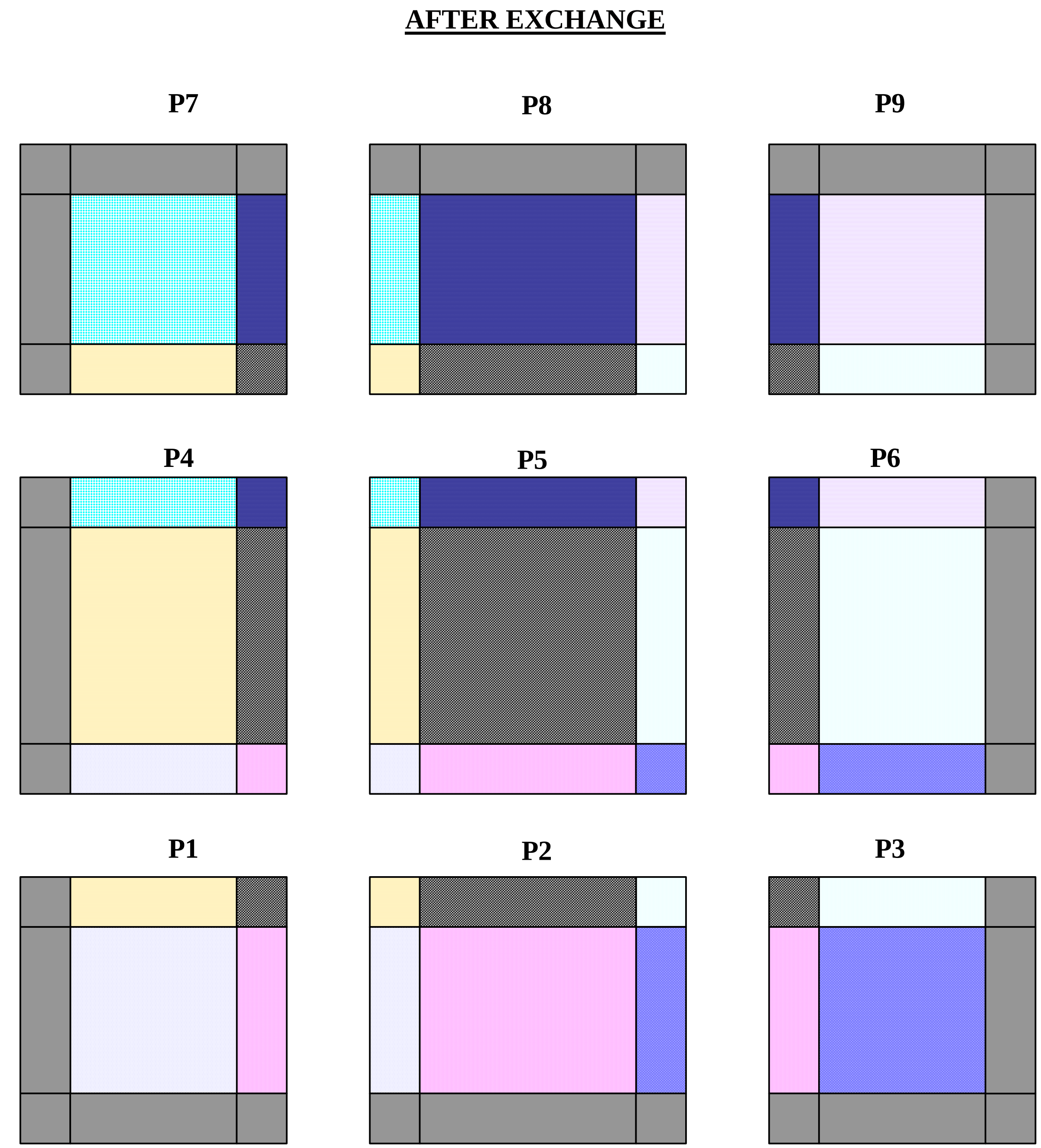


Figure 5-3: Illustration of the data distribution just after a 2-dimensional exchange for a problem with non-periodic boundaries.

5.1.2 Larger Stencils

In Figure 2-10, the widths of the stencil for the calculation of `df` in the laplace program are one point in each direction. Since this is the only computation in Laplace requiring "exchange", it is clear that the halo widths specified by CREATE_DECOMP must be 1 in the `i` dimension (line 5). However, suppose we modify Example 2-4 by adding additional calculations of x that step 2 points into the halo region (lines 60-64 in Example 5-2 below).

```
 1        program basic_ex_halo2
 2        include 'basic.inc'
 3        im = 10
 4        jm = 10
 5  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <2>)
 6        call laplace
 7        end
 8
 9        subroutine laplace
10        include 'basic.inc'
11        integer i, j, iter
12        real max_error
13        real tolerance
14        parameter (tolerance = 0.001)
15  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
16        real f(im,jm), df(im,jm)
17  CSMS$DISTRIBUTE END
18  CSMS$PARALLEL(DECOMP_I,<i>) BEGIN
19        do 100 j=1,jm
20        do 100 i=1,im
21          f (i,j) = 0.0
22          df(i,j) = 0.0
23    100 continue
24        do 110 j=1,jm
25  CSMS$GLOBAL_INDEX(1) BEGIN
26          f( 1,j) = 2.0
27          f(im,j) = 2.0
28  CSMS$GLOBAL_INDEX END
29    110 continue
30        do 120 i=1,im
31          f(i, 1) = 2.0
32          f(i,jm) = 2.0
33    120 continue
34        iter = 0
35        max_error = 2.0 * tolerance
36  C main iteration loop...
37        do while ((max_error .gt. tolerance) .and. (iter .lt. 1000))
38          iter = iter + 1
39          max_error = 0.0
40  CSMS$EXCHANGE(f)
41          do 200 j=2,jm-1
42          do 200 i=2,im-1
43            df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1))
44       &                      - f(i,j)
45    200 continue
46          do 300 j=2,jm-1
47          do 300 i=2,im-1
48            if (max_error .lt. abs(df(i,j))) then
49              max_error = abs(df(i,j))
50            endif
51    300 continue
52  CSMS$REDUCE(max_error, MAX)
53          do 400 j=2,jm-1
54          do 400 i=2,im-1
55            f(i,j) = f(i,j) + df(i,j)
56    400 continue
57        enddo
58
59  CSMS$EXCHANGE(df)
60        do j = 1, jm
```

```
61           do i = 3, im-2
62             f(i,j) = f(i,j) + 2.0*df(i,j) - df(i-2,j) - df(i+2,j)
63           end do
64         end do
65
66   CSMS$PARALLEL END
67         print *, 'Solution required ',iter,' iterations'
68         print *, 'Final error = ', max_error
69
70         end
```

Example 5-2: Modified version of Example 2-4 with additional code that has a stencil width of 2 in the *i* direction.

For the calculations starting at line 60, the width of the stencil is 2 in the `i` direction as shown in Figure 5-4.

```
df2(i,j) = 2.0*df(i,j) - df(i-2,j) - df(i+2,j)
```

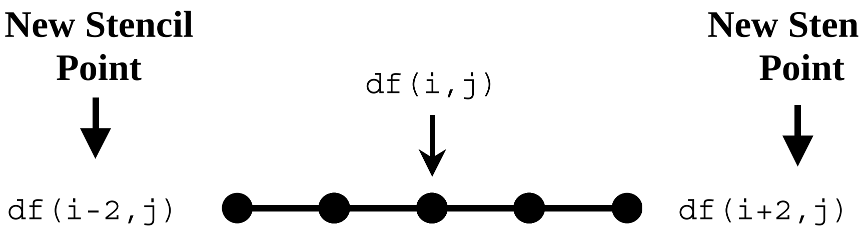


Figure 5-4: Modified stencil for additional calculations in Example 5-2. This time the stencil width is 2 in the *i* direction.

This program now has two calculations involving the same dimension of the same decomposition with different stencil widths. SMS handles this by requiring the programmer to make the halo width of the decomposition equal to the larger of the two widths. It is up to the programmer to determine the width of the largest stencil required in each dimension for every decomposition. The CREATE_DECOMP directive (line 5) shows the correct halo width specification (***<2>***).

5.1.3 Exchanges in Static Memory Models

For static memory models that require exchanges, the process-local array sizes specified in the DECLARE_DECOMP directive must be large enough to include the halo regions. In the program fragment below, the halo size is one. Since halo regions are on each side, the declared local array size is the global size (`im`) divided by the number of processes (4) plus 2 to accommodate the halo regions and plus 1 since 4 does not divide 30 evenly.

```
      program STATIC_MEMORY_EXCHANGE
      implicit none
      integer im
      parameter(im = 30)
      integer jm
      parameter(jm = 5)
CSMS$DECLARE_DECOMP(my_dh, <im/4 + 2 + 1>)
```

5.1.4 Miscellaneous

Another point about EXCHANGE is that, for both static and dynamic memory models, the number of processes used must be small enough to ensure the size of the smallest interior region is greater than the halo width in each decomposed dimension. SMS will issue the following run-time error message if this condition is violated:

```
Process:  0 Error status=   -2100 MSG IS: NNT_DECOMP_ERR
```

Also, we point out that EXCHANGE automatically implements the process synchronization required for the parallel code to produce correct results. A process scheduled to receive data from a neighbor will wait until the data have fully arrived before proceeding with the next set of calculations. A side effect of this synchronization is that the EXCHANGE directive cannot be used inside a decomposed loop because the number of iterations may not be the same on every process, causing deadlock.

5.2 Performance Optimizations

In this section, some optimizations are described that can be employed to reduce the number of exchanges and the amount of data exchanged in a parallel SMS program.

5.2.1 Limited-Thickness Exchanges

Choosing a single halo width could mean some data are communicated unnecessarily. The exchange at line 40 in Example 5-2 is an example of such inefficiency. The stencil of the computations in loop 200 is still one point wide in the *i* direction. However, since the halo width of *f* is now 2 in this dimension, one extra halo point on each side for each *j* index will be communicated unnecessarily. This extra communication can be eliminated by using a variant of the EXCHANGE directive that only exchanges part of the halo region:

```
CSMS$EXCHANGE(f <1,1>)
```

This option to EXCHANGE tells SMS to exchange only the first halo point in the lower and upper halo regions.

If we were to modify Example 5-2 to use a two dimensional decomposition, the CREATE_DECOMP directive would look as follows:

```
CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <2,1>)
```

Now, the maximum stencil width is 2 in the first decomposed dimension (for the exchange at line 59) and 1 in the second decomposed dimension (for the exchange at line

40). If the exchange at line 40 only requires 1 point in each direction, it could be optimally written as:

```
CSMS$EXCHANGE(f <1,1> <1,1>)
```

Since the exchange at line 59 is only needed to update points in the *i* dimension, it would optimally be written as:

```
CSMS$EXCHANGE(df <2,2> <0,0>)
```

5.2.2 Aggregating Exchanges

The program SLOW in Example 5-3, uses a statically declared one-dimensional decomposition (line 10) to distribute the arrays *a*, *b* and *c*. In this example, a halo thickness of one is defined by CREATE_DECOMP (line 24). After a series of iterations (line 39) a global sum is produced with the REDUCE directive (line 63).

```
 1        program SLOW
 2        implicit none
 3        integer im
 4        parameter(im = 30)
 5        integer jm
 6        parameter(jm = 5)
 7        integer iterations
 8        parameter(iterations = 3)
 9
10    CSMS$DECLARE_DECOMP(my_dh, <im/3 + 2>)
11
12    CSMS$DISTRIBUTE(my_dh, <im>) BEGIN
13          real a(im)
14          real b(im,jm)
15          real c(im,jm)
16    CSMS$DISTRIBUTE END
17
18          real ysum
19
20          integer i
21          integer j
22          integer iter
23
24    CSMS$CREATE_DECOMP(my_dh, <im>, <1>)
25
26          ysum = 0.0
27          b = 0.0
28          c = 0.0
29
30          do j = 1, jm
31
32    CSMS$PARALLEL(my_dh, <i>) BEGIN
33            do i = 1, im
34    CSMS$TO_GLOBAL(<1, i>) BEGIN
35              a(i) = real(3*i + 2 + j)
36    CSMS$TO_GLOBAL END
37            end do
38
39            do iter = 1, iterations
40
41    CSMS$EXCHANGE(a)
42
```

```
43            do i = 2, im-1
44              b(i,j) = a(i+1) + a(i-1)
45              c(i,j) = b(i,j) + c(i,j)
46            end do
47
48  CSMS$EXCHANGE(b)
49  CSMS$EXCHANGE(c)
50
51            do i = 2, im-1
52              a(i) = b(i+1,j) + b(i-1,j) + c(i+1,j) - c(i-1,j)
53            end do
54
55          end do
56
57          do i = 2, im - 1
58            ysum = ysum + a(i)
59          end do
60
61        end do
62
63  CSMS$REDUCE(ysum, SUM)
64
65        print *, 'ysum is ', ysum
66  CSMS$PARALLEL END
67        end
```

Example 5-3: A sub-optimal version of a program parallelized using SMS.

SMS provides the capability to aggregate the exchanges of multiple variables. If lines 48-49 are replaced with

```
CSMS$EXCHANGE(b,c)
```

then SMS will combine the communications of the corresponding halo regions of `b` and `c` as shown in Figure 5-5. Since the number of messages sent is halved, performance on high-latency machines will improve.

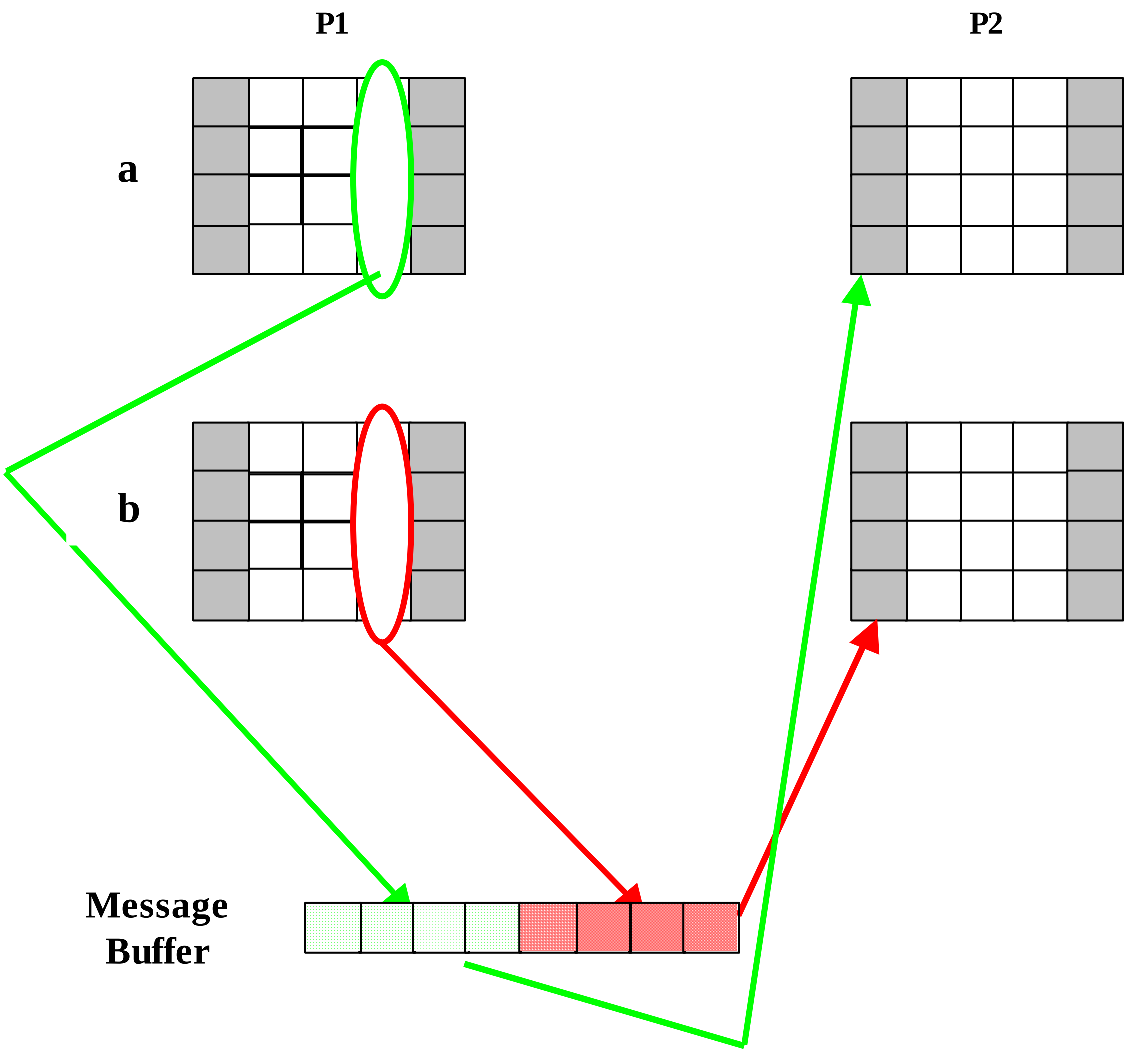


Figure 5-5: An illustration of how communications are aggregated to reduce latency for a portion of the exchange of `a` and `b`. The last column of process P1's variables are communicated as a single message to P2 where they are unpacked into the corresponding halo regions.

5.2.3 Exchanging Array Sections

Sometimes, it is not necessary to exchange an entire array. For example, in the following code fragment an adjacent dependence may only apply to some of the vertical levels of a 3D array:

```
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      real x(im,jm,km)
      real y(im,jm,km)
CSMS$DISTRIBUTE END
```

```
CSMS$EXCHANGE(x <0,1> <0,0>)
      do 100 k = 1,2
      do 100 j = 1,im
      do 100 i = 1,im
        y(i,j,k) = x(i+1,j,k) - x(i,j,k)
  100 continue
```

However, the exchange directive will exchange array *x* at all *k* levels even though the dependence exists only for *k=1* and *k=2*. This exchange directive can be optimized using standard Fortran array syntax as shown below:

```
CSMS$EXCHANGE(x(:,:,1:2) <0,1> <0,0>)
```

Now, only the *k=1* and *k=2* levels of array *x* will be exchanged. Note that use of array section syntax will only improve performance for subsections in non-decomposed dimensions.

5.2.4 Trading Communications for Computations Using HALO_COMP

Example 5-3 can be further optimized by trading communication for redundant computations in the halo region as is briefly discussed in the SMS overview paper. This is done using the HALO_COMP directive to modify the ranges of parallel loops to include computations in the halo regions. These extra computations can eliminate the need for some exchanges.

Figure 5-6, Figure 5-7, and Figure 5-8 illustrate how redundant computations work. Without the HALO_COMP directive, *b* and *c* are only computed in interior points using stencils like that shown in Figure 5-6. Halo regions of *b* and *c* must then be updated via an exchange for *a* to be properly computed as shown in Figure 5-7. A computation one step into the halo region (Figure 5-8) requires that *a* have a halo size of two instead of one. Since process P1 now computes points such as *b(4,2)* and *c(4,2)*, the computation of *a(3,2)* shown in Figure 5-7 can proceed without having exchanged *b* and *c*. However, extra computations are done since process P2 must also perform exactly the same computation for its corresponding interior points *b(4,2)* and *c(4,2)*,

Use of the HALO_COMP directive in this example reduces latency because the exchanges of *b* and *c* are no longer required. In addition, communication bandwidth is reduced. Although the amount of data communicated by the exchange of *a* has doubled, this is more than offset by the elimination of the exchanges of *b* and *c*.

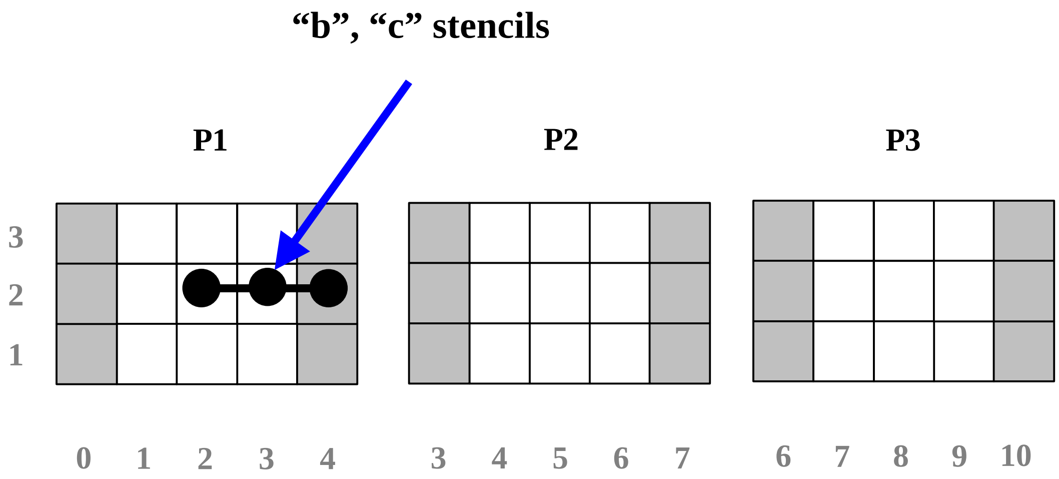


Figure 5-6: Memory layout of `a` (assuming `im=9`, `jm=3`) with sample stencil for calculations of `b` and `c` overlaid.

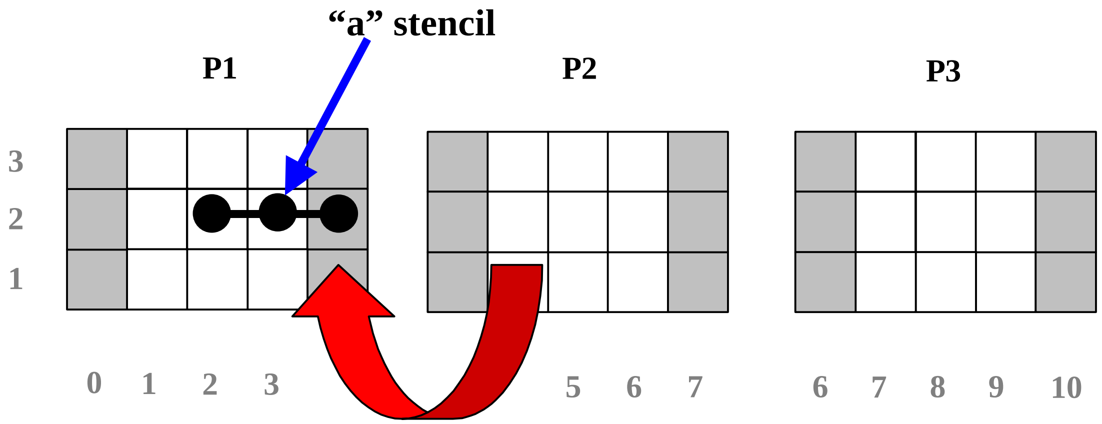


Figure 5-7: Memory layout of `b` and `c` with sample stencil for calculation of `a` overlaid. The halo regions of `b` and `c` must be updated via exchange for the calculation of `a` to be executed correctly.

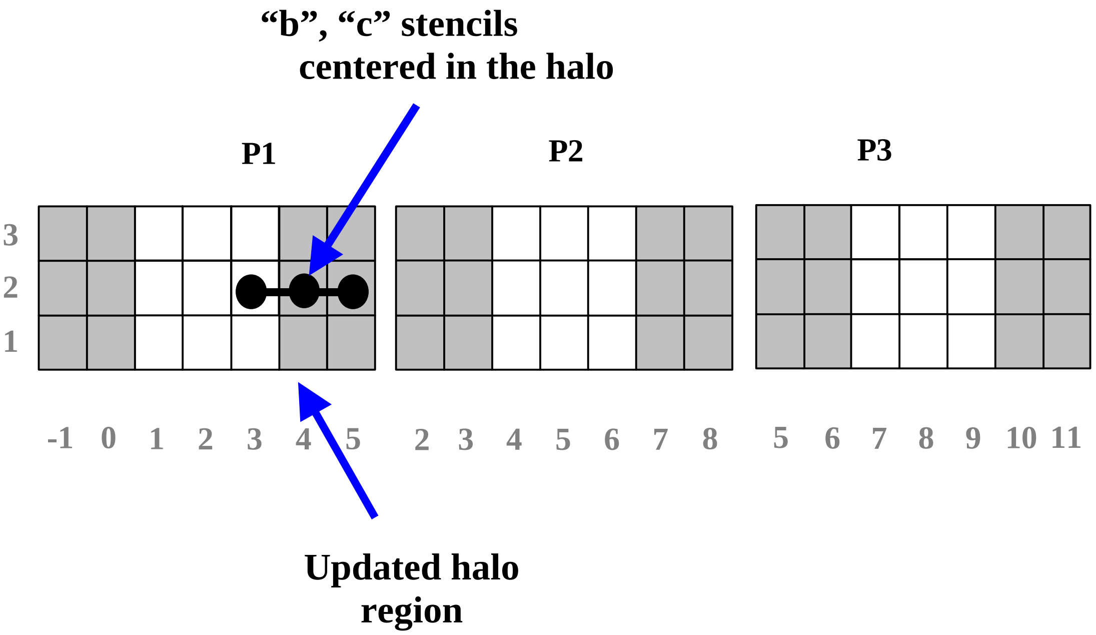


Figure 5-8: Modified memory layout of a with new sample stencil centered in the halo region. The computation of point b(4,2) and c(4,2) effectively updates the halo regions of b and c so that the computation of a in Figure 5-7 can be made without an exchange.

A net improvement in performance by this technique will only be realized if the cost of the additional computation by each process is less than the cost of exchanging b and c. Whether or not the code runs faster will, in general, depend on the communication patterns in the program, the number of processes used, and the target hardware platform.

A version of Example 5-3 that implements redundant calculations is shown in Example 5-4. The HALO_COMP directive on line 43 tells SMS that the enclosed loop should be executed 1 step into the halo region in each direction. This updates b and c sufficiently to satisfy the dependencies in the loop at lines 52-54. DECLARE_DECOMP and CREATE_DECOMP have been modified to accommodate the new halo size of 2. The exchanges of b and c have been eliminated.

```
 1          program FASTER
 2          implicit none
 3          integer im
 4          parameter(im = 30)
 5          integer jm
 6          parameter(jm = 5)
 7          integer iterations
 8          parameter(iterations = 3)
 9
10    CSMS$DECLARE_DECOMP(my_dh, <im/3 + 4>)
11
12    CSMS$DISTRIBUTE(my_dh, <im>) BEGIN
13          real a(im)
14          real b(im,jm)
```

```
      real c(im,jm)
CSMS$DISTRIBUTE END

      real ysum

      integer i
      integer j
      integer iter

CSMS$CREATE_DECOMP(my_dh, <im>, <2>)

      ysum = 0.0
      b = 0.0
      c = 0.0

      do j = 1, jm

CSMS$PARALLEL(my_dh, <i>) BEGIN
        do i = 1, im
CSMS$TO_GLOBAL(<1, i>) BEGIN
          a(i) = real(3*i + 2 + j)
CSMS$TO_GLOBAL END
        end do

        do iter = 1, iterations

CSMS$EXCHANGE(a)

CSMS$HALO_COMP(<1,1>) BEGIN
          do i = 2, im-1
            b(i,j) = a(i+1) + a(i-1)
            c(i,j) = b(i,j) + c(i,j)
          end do
CSMS$HALO_COMP END



          do i = 2, im-1
            a(i) = b(i+1,j) + b(i-1,j) + c(i+1,j) - c(i-1,j)
          end do

        end do

        do i = 2, im - 1
          ysum = ysum + a(i)
        end do

      end do

CSMS$REDUCE(ysum, SUM)

      print *, 'ysum is ', ysum

CSMS$PARALLEL END

      end
```

Example 5-4: A version of Example 5-3 that has been optimized by trading communications for redundant calculations in the halo region.

5.2.5 Using HALO_COMP and TO_LOCAL To Make Subroutines Do Redundant Computations

We saw in Section 4.3 how the TO_LOCAL directive can be used to parallelize subroutines without requiring directives inside the subroutine code. The approach works by making the subroutines operate on the interior of the process local arrays. Now, suppose we want those called routines to do redundant computations in the halo region to avoid communication. Example 5-5 shows a modified version of Example 4-2, illustrating how this is done.

```
     program tran_index7
     implicit none
     integer i
     integer, parameter :: im = 8

CSMS$DECLARE_DECOMP(dh, 1)

    integer istart, iend

CSMS$DISTRIBUTE(dh, 1) BEGIN
     integer, allocatable :: x(:), y(:), z(:)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(dh, <im>, <2>)

     allocate(x(im))
     allocate(y(im))
     allocate(z(im))
     x = 0.0
CSMS$PARALLEL(dh,<i>) BEGIN

CSMS$HALO_COMP(<1,1>) BEGIN
     do i=1,im
CSMS$TO_GLOBAL(<1,i>) BEGIN
       x(i) = i
CSMS$TO_GLOBAL END
     end do

     y = 0.0


CSMS$TO_LOCAL(<1, istart : lbound>, <1, iend : ubound>) BEGIN
     istart = 1
     iend   = im - 1
CSMS$TO LOCAL END
CSMS$HALO_COMP END

     call physics(x, lbound(x,1), ubound(x,1), istart, iend, y)

     do i = 1, im - 1
       z(i) = y(i) + y(i+1)
     end do

CSMS$SERIAL BEGIN
     write(*,'(8i5)') (z(i),i=1,im)
CSMS$SERIAL END

CSMS$PARALLEL END
     end
```

Example 5-5: Modified version of Example 4-2 that passes lower and upper bounds into subroutine *physics* so that it does redundant computations for one point in the halo region for each dimension and for each direction.

Since the calculations of `istart` and `iend` are now contained within both a TO_LOCAL and HALO_COMP directive, the effect is to change the lower and upper bounds passed to the `physics` so that it will do redundant computations for one point in the halo region for each direction. Figure 5-9 shows the new table of lower and upper bounds (compare to the table in Figure 4-1). Now, following the call to `physics`, variable `y` contains valid data one point into the halo region. Consequently, an EXCHANGE directive is not need prior to the loop at lines 39-41.

Process	i_mem_start	i_mem_end	dim1_start	dim1_end
P1	-1	6	1	5
P2	3	10	4	8

Figure 5-9: Table of memory and computational bounds for Example 5-5. Compare the *dim1_start* and *dim1_end* values to those the table in Figure 4-1. The memory start and end values are unchanged.

6 Handling Complex Dependencies Using TRANSFER

Section 2.7 introduced the TRANSFER directive and explained how it could be used to handle complex dependencies in more than one dimension (see Example 2-5). In Section 6.1, we show how TRANSFER can be used when either the source or destination array are non-decomposed. In Section 6.2, we examine how TRANSFER can be applied to the parallelization of spectral models. TRANSFER is also used for inter-grid interpolation as described in Section 11.2.

Like EXCHANGE, TRANSFER automatically implements the process synchronization required for the parallel code to produce correct results. A side effect of this synchronization is that the TRANSFER directive cannot be used inside a decomposed loop because the number of iterations may not be the same on every process, causing deadlock.

6.1 Further Details about TRANSFER

While TRANSFER can be used to generate communications to transpose arrays decomposed in one or more dimensions, it can also be used when either the source or destination arrays are not decomposed. If the destination array is not decomposed but the source is, then the TRANSFER directive effectively implements a “gather” of the source into the destination as illustrated in Figure 6-1. After the transfer, the entire array is replicated on each process. Since the local data for each process must be communicated to all other processes, this operation can be quite expensive.

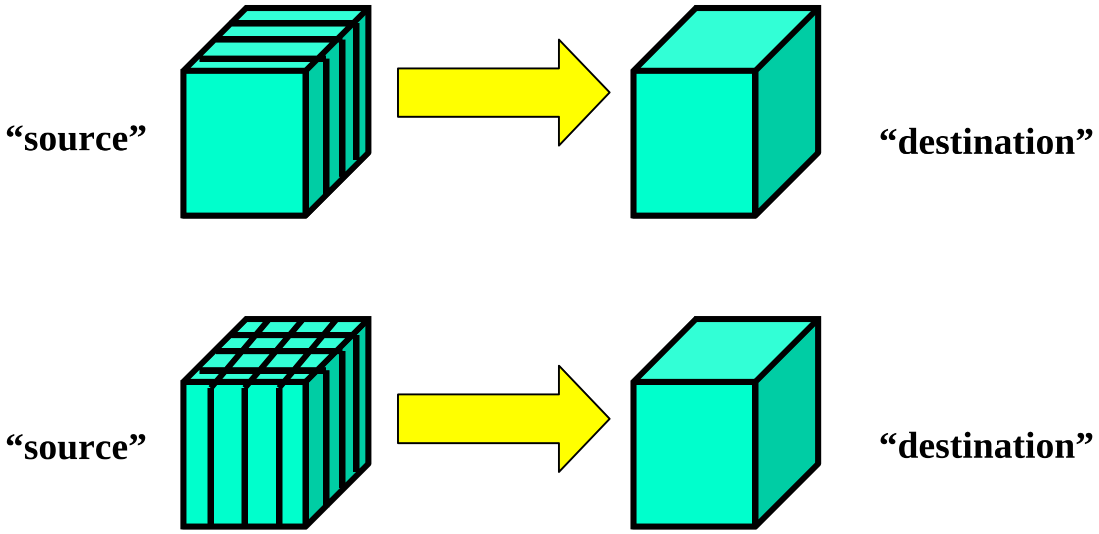


Figure 6-1: Schematic of the behavior of TRANSFER when the source array is decomposed and the destination array is NOT decomposed. The effect is to “gather” the process-local data from the source array into the globally-sized destination array.

If the source array is not decomposed but the destination array is decomposed then TRANSFER performs an "extract" of data from the source into the destination as shown in Figure 6-2. Note that no communication is needed in this case since each process has access to all needed data to begin with. Note that the SERIAL directive also performs “gather” and “extract” operations and is often easier to use than TRANSFER (see Section 8).

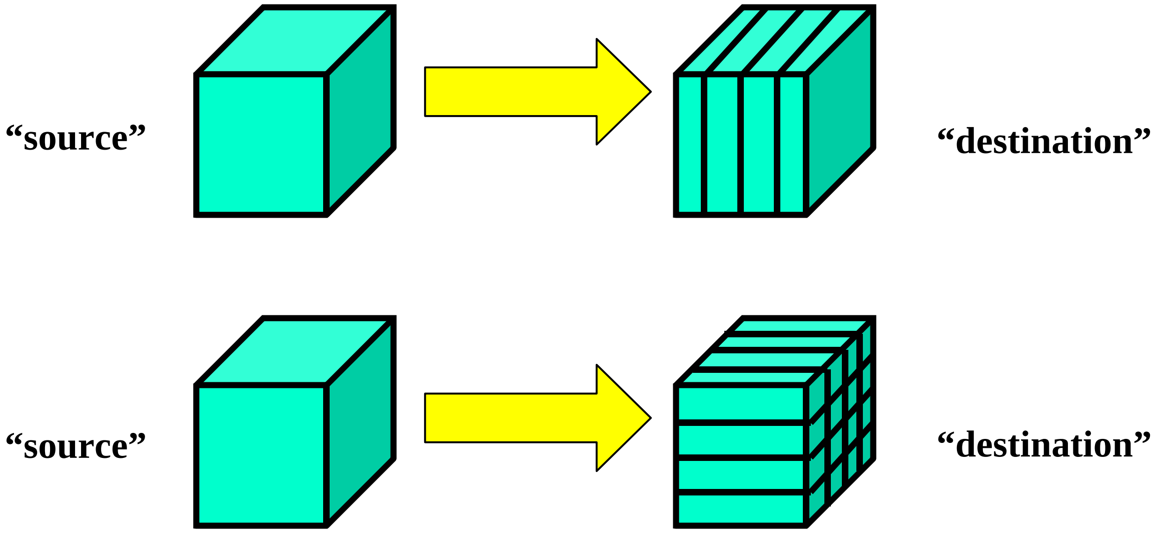


Figure 6-2: Schematic of the behavior of TRANSFER when the source array is NOT decomposed and the destination array is decomposed. The effect is to “extract” the appropriate data from the globally sized source array into the process-local destination array.

As in the case of EXCHANGE, TRANSFERs can be aggregated as follows to reduce latency:

```
CSMS$TRANSFER(<source1, destination1>, <source2, destination2>) BEGIN
      Serial code to be replaced...
CSMS$TRANSFER END
```

Note that, for all TRANSFER directives, the type and rank of the source and destination arrays must be the same. However, the array sizes may differ.

6.2 Applying TRANSFER to Spectral Models

Many spectral models have multiple phases of computation that repeat in a fixed pattern. Phases often have different optimal decompositions. Therefore, performance may be maximized by using multiple decompositions and transferring between them. Consider the case of one-dimensional decompositions for these models. The physical parameterizations contain complex dependencies in the vertical. This makes it efficient to decompose in one of the horizontal dimensions. At the same time, many computer system vendors provide highly optimized assembly FFT libraries that far out-perform anything that can be done with hand-tuned Fortran code.

Taking advantage of this serial code requires decomposing in a dimension other than `i`. Typically, the data are decomposed in the `j` dimension during physics and FFT computations (decomposition "a" in Figure 3-1). The Legendre transformations contain complex dependencies in the `j` dimension. Therefore, a second decomposition in either `i` (decomposition "b" in Figure 3-1) or `k` (decomposition "c" in Figure 3-1) is needed for optimal performance during these calculations. The TRANSFER directive provides the means to transpose the data from decomposition "a" to ("b" or "c") and back again. For large numbers of processes, 2D decompositions are needed to ensure that all processes have work to do. In this case, physics computations may be done using decomposition “d”, FFT computations may be done using decomposition “e”, and Legendre transform computations may be done using decomposition “f”.

7 Handling Global Dependencies Using REDUCE

In Section 2.3, we saw how the REDUCE directive was used to implement communication needed to do global summations and maxima. In this section we examine the REDUCE directive in more detail. In addition to giving more examples of the directive, Section 7.1 also shows that the form of REDUCE introduced in Section 2.3.3 (which will be referred to as "Standard Reductions") does not necessarily produce the bit-wise exact same answer as the serial code for global summations of floating point numbers. Section 7.2 introduces a second form of REDUCE called "Bit-wise Exact" that does produce the bit-wise exact same answer, regardless of the number of processes. Although quite useful for debugging, the second form has the drawback that it runs slowly.

Like EXCHANGE, both forms of REDUCE automatically implement the process synchronization required for the parallel code to produce correct results. A side effect of this synchronization is that the REDUCE directive cannot be used inside a decomposed loop because the number of iterations may not be the same on every process, causing deadlock.

7.1 More on Standard Reductions

In addition to global summations and maxima, the REDUCE directive can be used to generate global minima and to reduce arrays as seen in Example 7-1. Global minima are generated by specifying the keyword ***MIN*** (line 52). Also notice that reductions can be aggregated in the same way as exchanges (line 50). One of the variables reduced is the non-decomposed array *xsum* (line 50). The summation of *xsum* looks like the following:

```
Xsum_global(1) = Xsum_local$_1$ (1) + Xsum_local$_2$ (1) + ...
Xsum_global(2) = Xsum_local$_1$ (2) + Xsum_local$_2$ (2) + ...
      .
      .
      .
```

where *Xsum_local(j)* is the value of process-local *xsum(j)* on process P and *Xsum_global* is the value of *xsum* after the global summation is complete.

```
     1        program REDUCTIONS
     2        implicit none
     3        include 'basic.inc'
     4
     5        im = 50
     6        jm = 2
     7
     8  CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0>)
     9
    10        call DO_THEM
    11
    12        end
    13
```

```
      subroutine DO_THEM
      implicit none
      include 'basic.inc'

CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
      real x(im,jm)
      real y(im,jm)
CSMS$DISTRIBUTE END

      real xsum(jm)
      real ysum
      real xmin
      real xmax

      integer i
      integer j

      open (10, file='reduce_data', form='unformatted')
      read (10) x, y
      close(10)

CSMS$PARALLEL(DECOMP_I, <i>) BEGIN
      xsum = 0.0
      ysum = 0.0
      xmax = -999.0
      xmin = 999.0

      do j = 1, jm
        do i = 1, im
          xsum(j) = xsum(j) + x(i,j)
          ysum = ysum + y(i,j)
          xmax = max(xmax, x(i,j))
          xmin = min(xmin, x(i,j))
        end do
      end do

CSMS$REDUCE(xsum, ysum, SUM)
CSMS$REDUCE(xmax,       MAX)
CSMS$REDUCE(xmin,       MIN)

      print *
      print *, 'Global values'
      do j = 1, jm
        write(*,100) j, xsum(j)
      end do
      write(*,150) ysum
      write(*,200) xmax
      write(*,300) xmin

 100  format('j ', i2, ' xsum = ', F13.5)
 150  format('ysum = ', F13.5)
 200  format('xmax = ', F13.5)
 300  format('xmin = ', F13.5)

CSMS$PARALLEL END

      return
      end
```

Example 7-1: Program showing additional examples of how the REDUCE directive can be used.

If we were to modify Example 7-1 so that the *j* dimension is also decomposed and were to make *xsum* a decomposed variable, then the reduction of *xsum* would FAIL. This would happen because SMS does not currently support reductions that produce decomposed variables. (This would require doing the reduction over a subset of the processes.)

When run with 2 processes, program REDUCTIONS yields the following results:

```
 Global values
j  1 xsum =    1258.28589
j  2 xsum =    1310.71448
ysum =   -2464.28540
xmax =     100.00000
xmin =    -100.00000
```

However, when run with 4 processes, the results are:

```
 Global values
j  1 xsum =    1258.28577
j  2 xsum =    1310.71436
ysum =   -2464.28613
xmax =     100.00000
xmin =    -100.00000
```

Notice that the values for *xsum* and *ysum* are slightly different between the 2 and 4 process runs. We will now see why this is the case.

7.2 Bit-wise Exact Reductions

The differences in results in Example 7-1 are due to round-off error in the floating-point addition. The numbers are added in a different order in the 4-process case as compared to the 2-process case because the sums are first computed locally before being combined (see Section 2.3.3). In weather and climate models (which are non-linear systems), if the global sums feed back into the main model equations, these slight errors can grow and propagate; potentially yielding completely different model predictions for runs with differing numbers of processes.

For debugging purposes, it is useful to avoid these round-off errors. In fact, this is necessary for correct operation of the COMPARE_VAR debugging feature (see Section 15.1). To do this, SMS offers a form of REDUCE that produces the bit-wise exact same answer for any number of processes. Example 7-2 below shows how this works.

```
      program EXACT_REDUCTIONS
      implicit none
      include 'basic.inc'

      im = 50
      jm = 2

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

      call DO_THEM

      end

```

```
14        subroutine DO_THEM
15        implicit none
16        include 'basic.inc'
17
18  CSMS$DISTRIBUTE(DECOMP_I, <im>) BEGIN
19        real x(im,jm), y(im,jm)
20  CSMS$DISTRIBUTE END
21
22        real ysum
23
24        integer i
25        integer j
26
27        open (10, file='reduce_data', form='unformatted')
28        read (10) x, y
29        close(10)
30
31  CSMS$PARALLEL(DECOMP_I, <i>) BEGIN
32
33  CSMS$REDUCE(ysum, SUM) BEGIN
34        ysum = 0.0
35        do j = 1, jm
36          do i = 1, im
37            ysum = ysum + y(i,j)
38          end do
39        end do
40  CSMS$REDUCE END
41
42        print *
43        print *, 'Global values'
44        write(*,150) ysum
45
46   150  format('ysum = ', F13.5)
47
48  CSMS$PARALLEL END
49
50        return
51        end
```

Example 7-2: Program illustrating the bit-wise exact form of the REDUCE directive.

The modified REDUCE syntax can be see on lines 33 and 40. The syntax requires a BEGIN and END directive. SMS replaces the calculations between the REDUCE BEGIN and END with code that gathers each process's piece of *y* into a globally-sized (replicated) variable and then sums the result in the correct order. Conceptually, the generated parallel code would be:

```
call GATHER(y, y_global)
ysum = 0.0
do j = 1, jm
  do i = 1, im
    ysum = ysum + y_global(i,j)
  end do
end do
```

The "gather" operation is done in the same way as TRANSFER was used to gather variables as discussed in Section 6.1. Since the gather operation requires communicating the entire contents of *y* to all processes, this form of global sum is significantly less efficient than the "standard" form of reduction. In that case, only the process-local scalar sums are communicated to all the processes.

The bit-wise exact form of the REDUCE directive will only produce exact sums if an environment variable called SMS_BITWISE is set to the value EXACT. Running EXACT_REDUCE in a c-shell environment might look as follows:

```
>> setenv SMS_BITWISE EXACT
>> smsRun -np 2 exact_reduce
SMS:  BITWISE EXACT reductions will be used when requested.
 Global values
ysum =    -2464.28418
```

Notice that the message printed by SMS regarding reductions now indicates that bit-wise exact reductions will be used.

If SMS_BITWISE is NOT set to EXACT then the effect of the REDUCE directive is the same as in the "standard" reduction; each process computes a local sum of y and the resulting scalars are summed across the processes.

```
>> setenv SMS_BITWISE INEXACT
>> smsRun -np 2 exact_reduce
SMS:  Standard reductions will be used.
 Global values
ysum =    -2464.28540
```

In summary, the "bit-wise exact" form of global summation is valuable for testing purposes, particularly for non-linear systems. Its use is required for correct operation of the COMPARE_VAR debugging feature (see Section 15.1). However, for long model runs, when optimal performance is important, the "standard" form of REDUCE will likely be more appropriate because it is much faster. The programmer can use the bit-wise exact form of REDUCE in the code and then decide at run-time, with the SMS_BITWISE environment variable, which reduction to use.

8 Incremental Parallelization Using SERIAL

The SERIAL directive is useful when other SMS directives cannot be easily applied to a piece of serial code or when efficient performance is not critical. One example is initialization. For long model runs, the effects of inefficient code during initialization become negligible. Diagnostic print messages are another case. If the user can turn off diagnostic messages when high performance is needed then the presence of inefficient parallel code that generates these messages does not pose a problem. Another use of SERIAL is for incremental parallelization of a large code. Using this technique, SERIAL directives are inserted around large sections of code. The directives are then removed one-by-one as each section is parallelized. This simplifies testing and debugging because each parallel code section can be tested and debugged separately.

8.1 Improving the Performance of SERIAL

Any code segment enclosed by SERIAL BEGIN and SERIAL END directives is called a "serial region". When SMS encounters a serial region, it automatically gathers all decomposed arrays on a single process, executes the enclosed code segment on the process, and scatters all decomposed arrays back to all processes (see Figure 8-1). In addition, all non-decomposed variables are broadcast to all the processes. By default, SMS gathers/scatters all decomposed variables and broadcasts all non-decomposed variables referenced in the serial region. These communications cause the code to run even more slowly than the original serial code. To improve performance, the user can specify variables to be scattered, gathered, or broadcast using keywords "***IN***", "***OUT***", or "***INOUT***" (when both operations are required). Also, when no communication is required, the "***IGNORE***" keyword can be used to suppress communication.

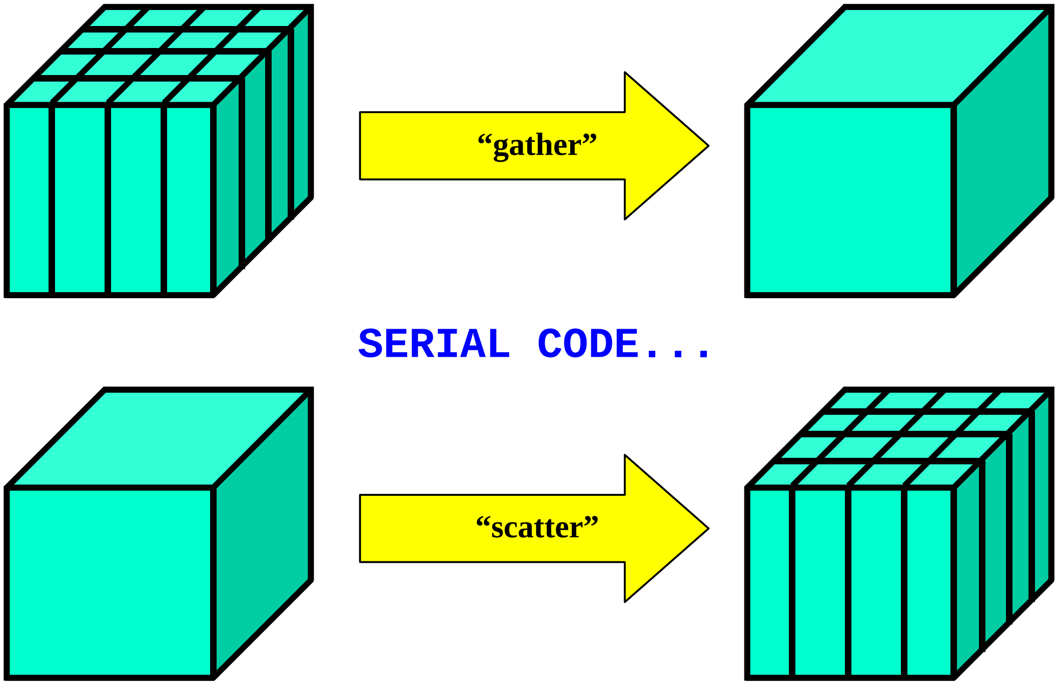


Figure 8-1: Gather and scatter of decomposed data at the beginning and end of a serial region.

In Example 8-1, *x* and *y* are decomposed while *z* is not. The subroutine calls at lines 39-40 read in *x* and *z* using C language routines (the C code is not shown). Since C routines cannot be handled by SMS, the SERIAL directive is used to generate code that gathers *x* and *y* into global variables. A single process then executes the code at lines 39-41. Finally, the generated code scatters *x* and *y* and broadcasts the value of *z*. Scalar variables *im* and *jm* are not broadcast since the default is set to “***IGNORE***”. When high performance is desired, the user can avoid this poorly performing code segment by setting *ENABLE_DIAGS* to *.FALSE.*.

```
[Include file: serial.inc]
      1      integer im,jm
      2      common /sizes_com/ im,jm
      3 CSMS$DECLARE_DECOMP(DECOMP_IJ, 2)

[Source file: serial1.f]
      1       program SERIAL
      2
      3       include 'serial.inc'
      4
      5       integer i
      6       integer j
      7
      8       im = 5
      9       jm = 4
     10
     11 CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <0,0>)
     12
     13       call DO_IT
     14
     15       end
```

```
16
17       subroutine DO_IT
18       include 'serial.inc'
19
20 CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
21       real x(im,jm)
22       real y(im,jm)
23 CSMS$DISTRIBUTE END
24       real z
25       logical ENABLE_DIAGS
26       ENABLE_DIAGS = .true.
27
28       open(10, file='yin', form='unformatted')
29       read(10) y
30       close(10)
31
32 C Some parallel computations
33 C          .
34 C          .
35 C          .
36
37       if (ENABLE_DIAGS) then
38 CSMS$SERIAL(<x, y, INOUT>, <z, OUT> : DEFAULT=IGNORE) BEGIN
39         call READ_2D_ARRAY_USING_C(x, im, jm)
40         call READ_SCALAR_USING_C(z)
41         print *, 'y(2,2), z ', y(2,2), z
42 CSMS$SERIAL END
43       end if
44 C More parallel calculations
45             .
46             .
47       return
48       end
```

Example 8-1: A sample program showing how the SERIAL directive can be used to generate correct parallel code in a simple fashion when other SMS directives will not suffice.

8.2 Limitations of SERIAL

Some care must be taken when using the SERIAL directive. First, SMS does not perform inter-procedural analysis so calling a routine that uses common block variables from within a serial region can produce erroneous results. Suppose we insert the following code after line 38 in Example 8-1:

```
call sub1
```

Further suppose *sub1* looks like this:

```
subroutine sub1
real xc
common /com1/ xc
xc = 2.0
return
end
```

SMS has no way of knowing that *xc* has to be broadcast before the end of the serial region because it does no inter-procedural analysis. A solution here would be to include */com1/* in subroutine *DO_IT* and specify *xc* as an "***OUT***" variable in the directive:

```
CSMS$SERIAL(<x, y, INOUT>, <xc, z, OUT> : DEFAULT=IGNORE) BEGIN
```

Second, care must be taken when constants are passed into subroutines that use SERIAL. In Example 8-2, the constant *2* is passed to subroutine *DO_IT*. Since *DO_IT* calls a C routine that uses dummy argument *n*, a SERIAL directive would normally be required to handle this. However the SERIAL directive generates a broadcast of dummy argument *n*, which will attempt to modify its value. Since *n* is the constant *2*, the result will be unpredictable (with luck, a core dump). This problem can be corrected by using the "***IGNORE***" keyword to ensure that the SERIAL directive will not attempt to broadcast *n*.

```
      program SERIAL

      include 'serial.inc'

      integer i
      integer j

      im = 5
      jm = 4

CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <0,0>)

      call DO_IT(2)

      end

      subroutine DO_IT(n)

      integer n

CSMS$SERIAL BEGIN
      call c_routine(n)
CSMS$SERIAL END

      return
      end
```

Example 8-2: Code where use of the SERIAL directive generates parallel code that fails to run properly.

Care must also be taken when using SERIAL in combination with statements that alter execution sequence (such as IF, GOTO, etc.). To avoid problems, program execution should only enter a serial region from the statement immediately before the SERIAL BEGIN directive. Similarly, program execution should only exit a serial region from the statement immediately before the SERIAL END directive.

Finally, like EXCHANGE, SERIAL automatically implements the process synchronization required for the parallel code to produce correct results. A side effect of this synchronization is that the SERIAL directive cannot be used inside a decomposed loop because the number of iterations may not be the same on every process, causing deadlock.

9 Periodic Boundary Conditions

By default, SMS assumes that decomposed dimensions have non-periodic boundary conditions. However, global FDA models have periodic boundary conditions in the east-west direction. Also, idealized test cases often use periodic boundary conditions in more than one dimension (such as in both horizontal dimensions). SMS supports periodicity in any decomposed dimension.

9.1 Using CREATE_DECOMP to Specify Periodic Boundaries

SMS allows the user to specify periodic boundary conditions using the optional ***PERIODIC*** keyword in the CREATE_DECOMP directive as shown in Example 9-1. This option causes exchanges to fill in the halo regions on the outer model boundaries as shown in Figure 9-1.

```
     program laplace_periodic
     implicit none

     integer im
     parameter(im=100)
     integer jm
     parameter(jm=99)

CSMS$DECLARE_DECOMP(dh : <0,0>)

     real global_error
     real tolerance
     parameter (tolerance = 0.002)
     integer iter
     integer i
     integer j

CSMS$DISTRIBUTE(dh, 1, 2) BEGIN
     real, allocatable ::  f(:,:)
     real, allocatable ::  df(:,:)
CSMS$DISTRIBUTE END


CSMS$CREATE_DECOMP(dh, <im+2, jm+2>, <2,2> : <PERIODIC, PERIODIC>)

     allocate(f (0:im+1, 0:jm+1))
     allocate(df(0:im+1, 0:jm+1))

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

CSMS$SERIAL BEGIN
     do j = 0, jm+1
       do i = 0, im + 1
         f(i,j) = sqrt(real(i+j)) - sqrt(real(i)) - sqrt(real(j))
       end do
     end do
CSMS$SERIAL END

     iter = 0
     global_error = 1.0

     do while ((global_error .gt. tolerance) .and. (iter .lt. 1000))
```

```
       iter = iter + 1

       global_error = 0.0

CSMS$EXCHANGE(f)
CSMS$GLOBAL_INDEX(1) BEGIN
      do j = 1, jm
        f(0   ,j) = f(im, j)
        f(im+1,j) = f(1 , j)
      end do
CSMS$GLOBAL_INDEX END

CSMS$GLOBAL_INDEX(2) BEGIN
      do i = 1, im
        f(i,0   ) = f(i, jm)
        f(i,jm+1) = f(i, 1 )
      end do
CSMS$GLOBAL_INDEX END

       do j = 1, jm
         do i = 1, im
           df(i,j) = 0.25*(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1))
    &                - f(i,j)
         end do
       end do

       do j = 1, jm
         do i = 1, im
           if (global_error .lt. abs(df(i,j))) then
             global_error = abs(df(i,j))

           end if
         end do
       end do


       do j = 1, jm
         do i = 1, im
           f(i,j) = f(i,j) + df(i,j)
         end do
       end do

CSMS$REDUCE(global_error, MAX)

     end do

     print *, 'Ended with iter : ', iter
     print *, 'Global_error: ', global_error

CSMS$PARALLEL END

     end
```

Example 9-1: Version of the Laplace program with periodic boundary conditions.

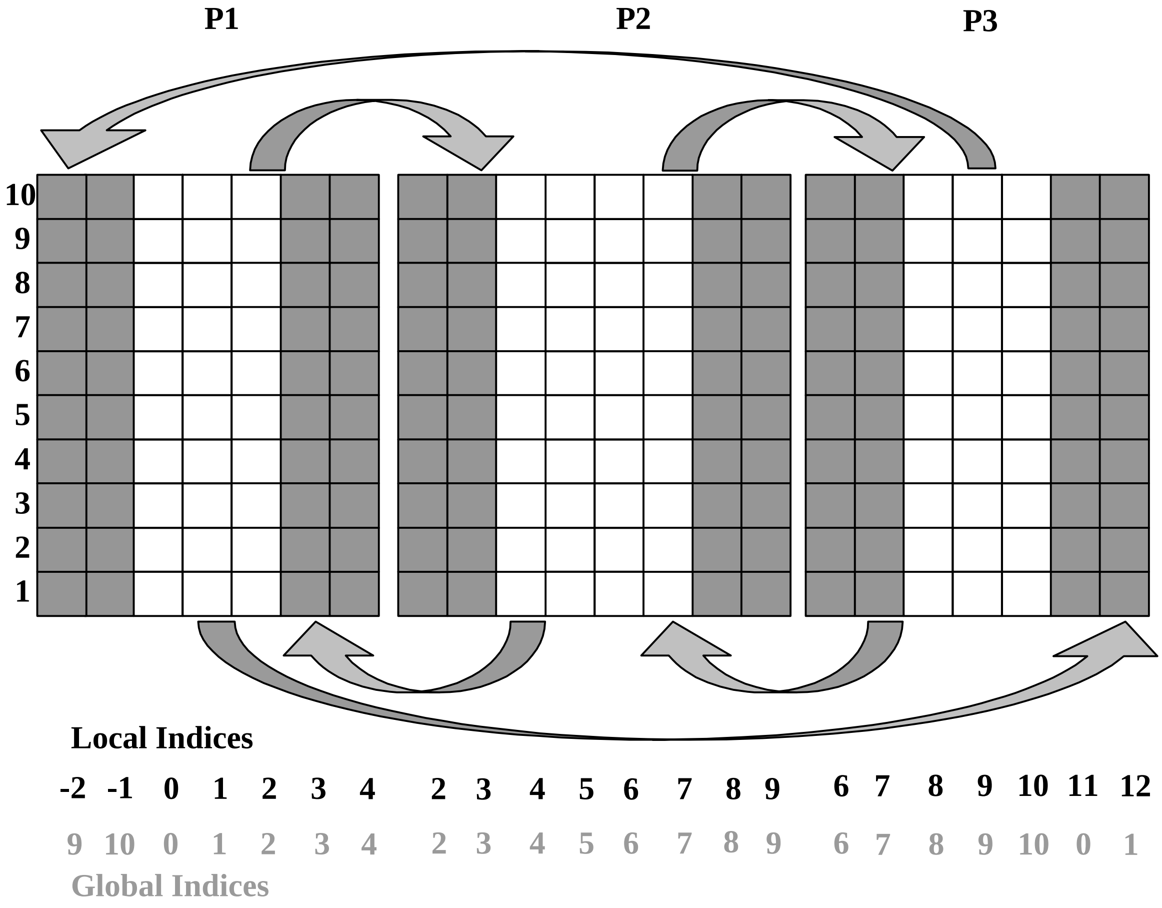


Figure 9-1: Exchanges in a periodic model.

With the model boundary halo regions properly updated, the periodic boundary conditions shown in lines 48-60 will produce the correct answer. Notice that even in the dynamic memory case, the local and global indices are not identical as shown in Figure 9-1. With `im` set to 9, local index –1 on process P1 corresponds to global index `im+1` (10). Local index 11 on process P3 corresponds to global index 0. However, explicit index translation using TO_LOCAL and TO_GLOBAL is still not needed in a periodic code that uses dynamic memory because index differences only occur at the periodic boundaries where GLOBAL_INDEX will automatically handle any needed translations.

It is also possible to mix periodic and non-periodic boundaries by using the ***NONPERIODIC*** keyword in the CREATE_DECOMP directive. In the example below, the first dimension of decomposition `dh` will be periodic and the second dimension will be non-periodic.

```
CSMS$CREATE_DECOMP(dh, <im+2, jm+2>, <2,2> : <PERIODIC, NONPERIODIC>)
```

9.2 Periodic Boundary Optimization

Notice in Example 9-1 that two halo points are specified even though the maximum stencil size is 1. The extra halo point is needed for the periodic boundary condition due to the mapping between boundary points and halo points. For example, in the boundary condition statement:

f(i,0) = f(i, jm)

the interior point `im` maps to the first halo point on the left-hand-size (local index –2). However, in this example, no interior points map to halo point –1. Thus, it is possible to eliminate the second halo point. To do this, the user can optimize the handling of periodic boundary conditions via the following syntax:

```
CSMS$CREATE_DECOMP(dh, <im+2, jm+2>, <2,2> :
CSMS$>             <PERIODIC(1)(IM),PERIODIC(1)(JM)>)
```

This variation indicates that, for the 1st decomposed dimension, the lowest global index appearing on the right-hand-side of any boundary condition is `1` (Example 9-1: line 50). The highest such point is `im` (line 51). Analogously, the lowest and highest indices for the 2nd decomposed dimension are `1` and `jm`, respectively (lines 57 & 58). Notice that the halo size in the directive is now 1 instead of 2. This optimization decreases the communication bandwidth for the periodic boundary halo updates and improves cache re-use overall in the model because of the reduced number of halo points required.

10 Mitigating Static Load Imbalances

Ideally, each process will have exactly the same amount of work to do. In practice, most models have computations that vary spatially so some processes may have more work than others. This is commonly known as load imbalance. Load imbalances slow down a parallel program because processes with less work are forced to wait for processes with more work to catch up. Load imbalances can be fixed (static) or variable (dynamic) in time. SMS provides tools to reduce static load imbalances. A future SMS release will provide methods to mitigate dynamic load imbalances.

Several types of static load imbalances are found in weather and ocean models. One type is due to the fact that edge processes may have boundary condition calculations while other processes do not. SMS allows local regions with different sizes to be specified for each process at run time. By giving more points to interior processes, the imbalance can be mitigated. This approach can also be used to reduce load imbalance in ocean models where computations are skipped for land grid points (see Section 10.1).

Global atmospheric models can have longitudinal imbalances due to differences in the computations required for day/night points and latitudinal imbalances stemming from summer/winter (or tropical/temperate) computational differences. SMS provides means to mitigate both of these effects using a technique called index scrambling. This technique is described in Section 10.2.

10.1 Controlling Process Layout

By default, SMS uses a pre-determined set of rules to decide how data points are assigned to each process and how many processes are allocated to each decomposed dimension. Roughly speaking, the data points are distributed evenly among the processes along a given decomposed dimension as seen in Figure 2-2 and Figure 2-3. In addition, processes are divided among the decomposed dimensions so as to minimize the amount of data moved during an EXCHANGE operation.

SMS also provides a mechanism for the user to specify the assignment of data points using a process configuration file in the form of a Fortran namelist. For a given decomposition, the user defines how many processes are assigned to each decomposed dimension. SMS will then choose how many data points are assigned to each process as before. The user can also optionally define how many data points are assigned to each process. Example 10-1 shows how process configuration works.

```
[file config.f]
      program CONFIGURE
      integer, parameter :: im = 12
      integer, parameter :: jm = 12
CSMS$DECLARE_DECOMP(dh1, 2)
CSMS$DECLARE_DECOMP(dh2, 2)
CSMS$CREATE_DECOMP(dh1, <im,jm>, <0,0>)
CSMS$CREATE_DECOMP(dh2, <im,jm>, <0,0>)
CSMS$DISTRIBUTE(dh1, 1, 2) begin
```

```
       real, allocatable :: a1(:,:)
CSMS$DISTRIBUTE end

CSMS$DISTRIBUTE(dh2, 1, 2) begin
       real, allocatable :: a2(:,:)
CSMS$DISTRIBUTE end
   .
   .
   .

[file my_config]
&decomp
decomp1_name = 'dh1',
decomp1_nps = 2 2,
decomp2_name = 'dh2',
decomp2_nps = 4 1,
decomp2_ddim1_sizes = 2 4 4 2/
```

Example 10-1: Sample program and associated configuration file illustrating how the user can tell SMS how to distribute the data among the processes.

The program in the example defines two decompositions. The configuration file, *my_config*, specifies how the data are distributed among the processes. In this case, it indicates that two processes will be assigned to each decomposed dimension of decomposition dh1. However, the user leaves it up to SMS to determine how many data points are assigned to each process. Figure 10-1 shows the memory layout of *a1*.

For *dh2*, the user specifies that the 4 processes will be assigned to the first decomposed dimension. The user further specifies how many points are assigned to each process. Figure 10-2 shows the memory layout of *a2*.

The user tells SMS to use configuration file *my_config* as follows:

```
>> smsRun -cf my_config my_program
SMS: For processor layout of decomposition dh1, using config file :
my_config
SMS: For processor layout of decomposition dh2, using config file :
my_config
```

Notice that the *smsRun* command does not specify how many processes are requested. SMS figures out how many are needed from the configuration file. Also notice the diagnostic message from SMS that describes which decompositions are defined in the configuration file. For brevity, this message will not be shown again.

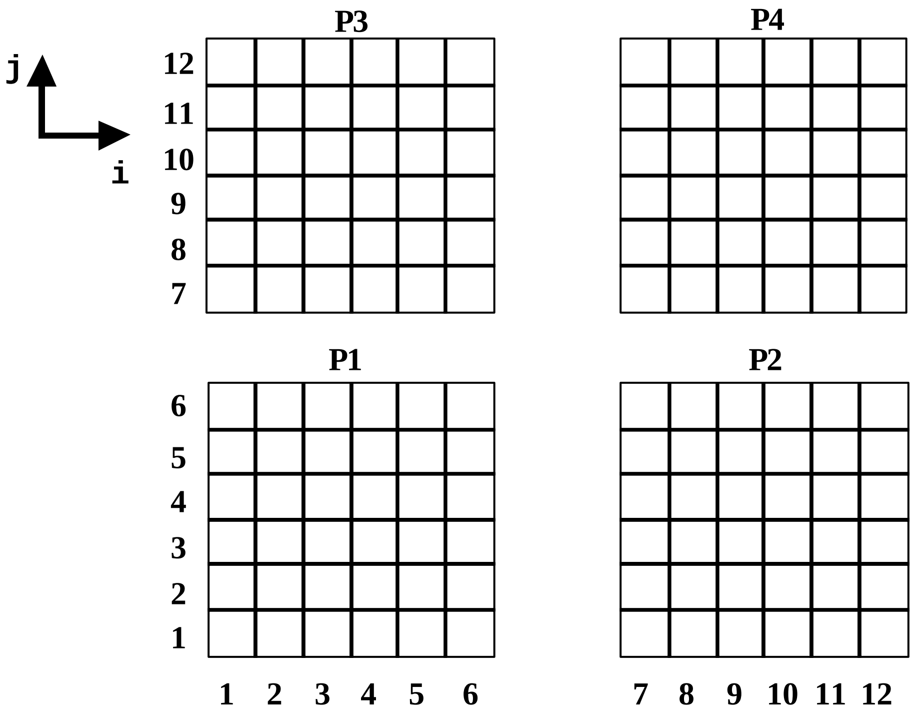


Figure 10-1: Memory layout of array *`a1`* in program *`CONFIGURE`*.

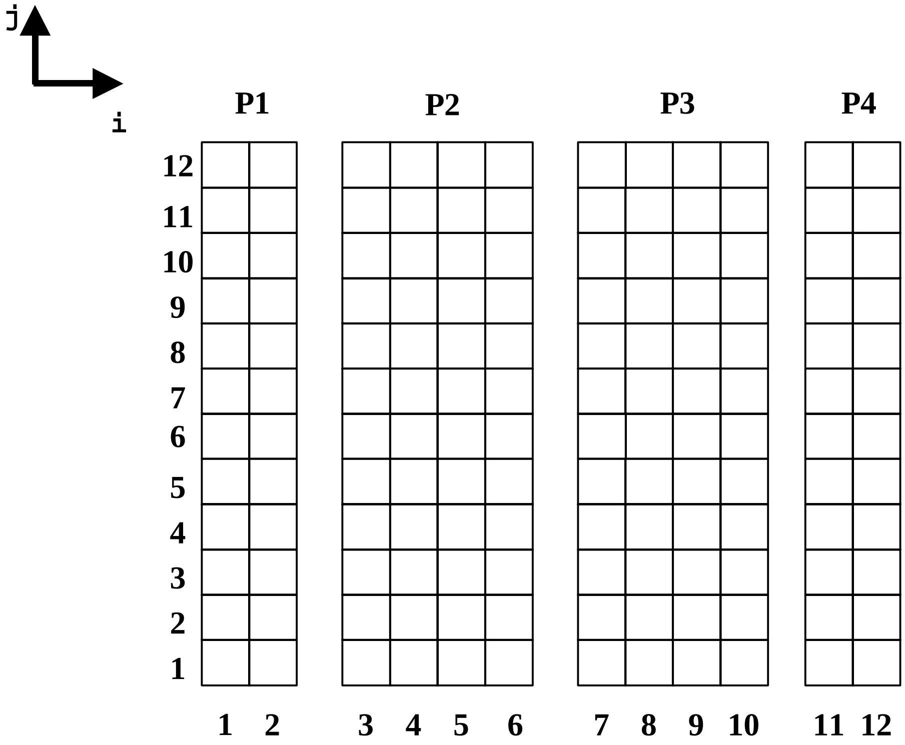


Figure 10-2: Memory layout of array *`a2`* in program *`CONFIGURE`*.

SMS also supports decompositions in which the number of local data points on each process is allowed to vary in two decomposed dimensions. Such “irregular” decompositions may be arbitrarily complex so long as each process has a single rectangular region and every point is owned by one and only one process. The code fragment below shows a configuration file for an irregular decomposition. This decomposition is illustrated in Figure 10-3. Irregular decompositions are useful for reducing complex static load imbalances such as those found in an ocean model where no computations occur at land points.

```
&decomp
decomp1_name = 'dh_irregular',
decomp1_nps = 7,
decomp1_ddim1_proc_starts =  1  5  9  1  5  1  5,
decomp1_ddim1_proc_ends   =  4  8 10  4  8  4 10,
decomp1_ddim2_proc_starts =  1  1  1  5  4  7  8,
decomp1_ddim2_proc_ends   =  4  3  7  6  7 10 10/
```

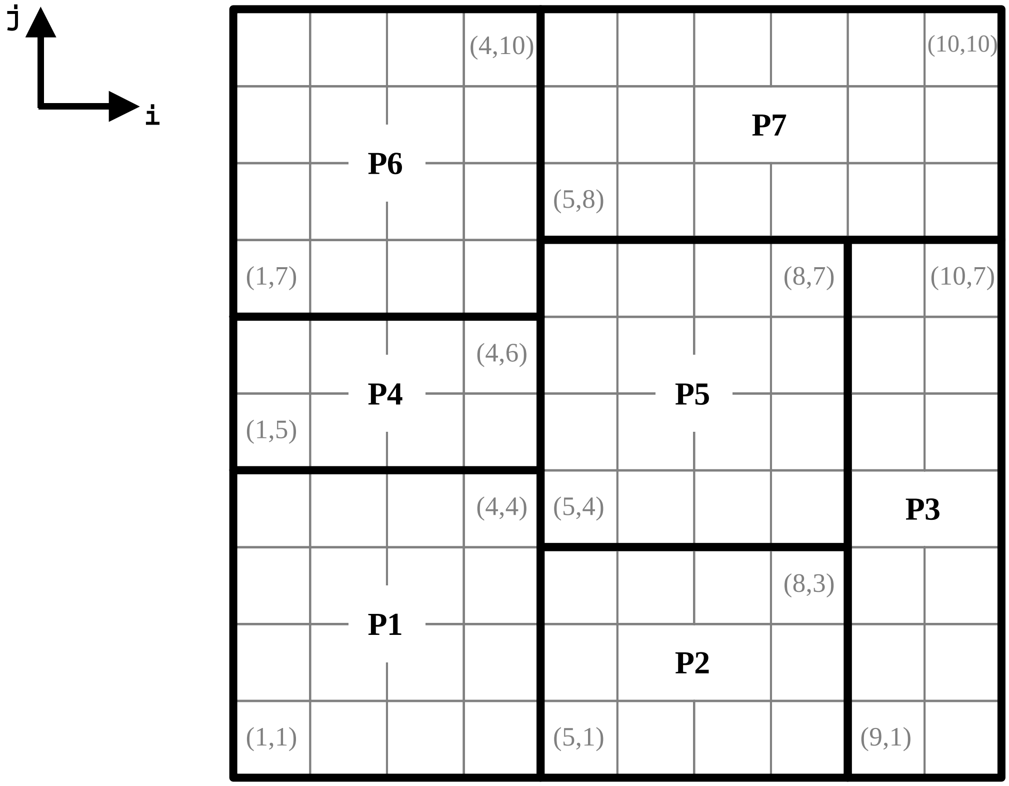


Figure 10-3: An “irregular” decomposition.

10.2 Index Scrambling

Index scrambling, accessible via an option to the CREATE_DECOMP directive, moves adjacent row/column pairs to other processes. Figure 10-4 shows how the data might be distributed following longitudinal scrambling. Due to the complexity of these redistributions, they are not permitted for decompositions that have adjacent dependencies (because the communications generated by EXCHANGE directives would perform poorly).

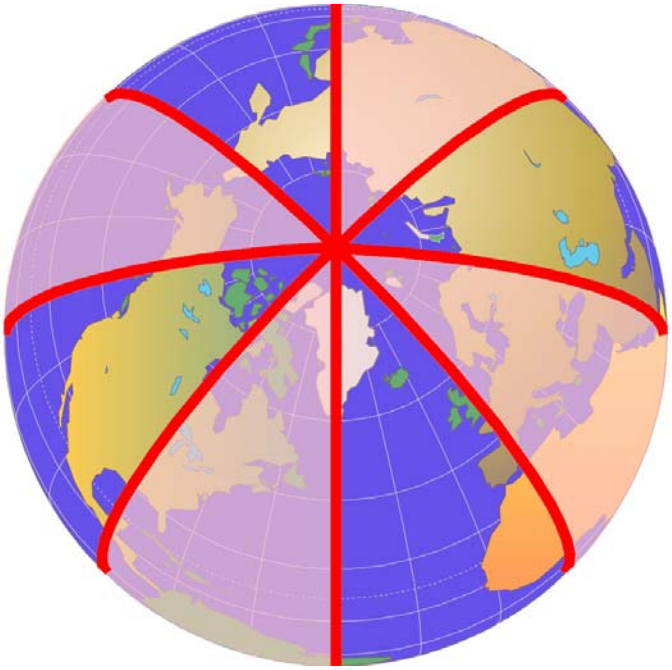


Figure 10-4: Longitude scrambling is used to reduce load imbalances due to computational differences stemming from day/night cycles in a global weather model. In this case, the model is run using 2 processes. One process has the brightly colored segments; the other has the shaded segments. The effect is to give each process half of the day points and half of the night points.

To use index scrambling, a fourth argument is added to the CREATE_DECOMP as shown in the code fragments below:

```
CSMS$CREATE_DECOMP(DECOMP_J, <jm>, <0> : <SCRAMBLE_LAT_STRATEGY>)

CSMS$CREATE_DECOMP(DECOMP_I, <im>, <0> : <SCRAMBLE_LON_STRATEGY>)
```

In the first case, argument ***<SCRAMBLE_LAT_STRATEGY>*** indicates that the first decomposed dimension of *DECOMP_J* will be scrambled using a method appropriate for balancing load among latitudes in a global model. In the second case, argument ***<SCRAMBLE_LON_STRATEGY>*** indicates that the first decomposed dimension of ***DECOMP_I*** will be scrambled using a method appropriate for balancing load among longitudes in a global model. No other code changes are required to use the scrambling feature. For this reason, it is convenient to add index scrambling as a performance optimization once debugging of the non-scrambled parallel code is complete.

11 Nesting and Coupling: Transfer-Interpolation

In some programs it is necessary to interpolate data from one grid onto another grid. This kind of "inter-grid interpolation" occurs, for example, in nested atmospheric codes that use mesh refinement techniques to improve resolution in critical areas. Data from a coarse "parent" grid may be used to compute boundaries of a fine "child" grid (see Figure 11-1). Conversely, data from a fine "child" grid may be used to compute overlapping values in its coarse "parent" (see Figure 11-2). Also, it may be necessary to reconcile overlapping portions of two "sibling" grids (see Figure 11-3). In addition, inter-grid interpolation is used to couple models that are based on different grids (see Figure 11-4). A common example is coupling of atmospheric and oceanic models.

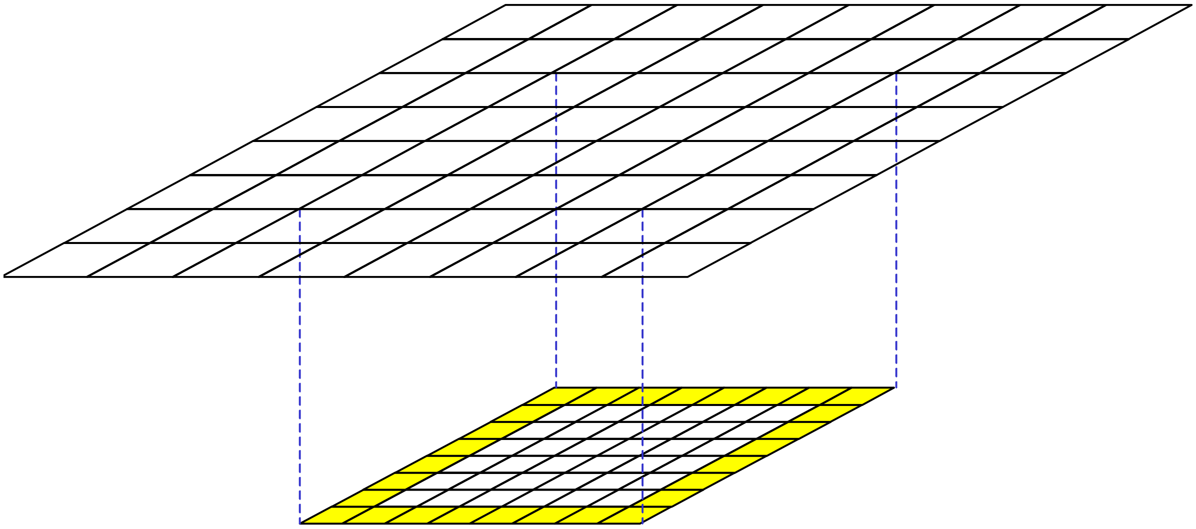


Figure 11-1: Computing boundary points in a high-resolution "child" grid from points in a low-resolution "parent" grid.

In a parallel program, inter-grid interpolations often require inter-process communication that can be quite complex. SMS encapsulates this complexity by combining both interpolation and communication into a single "*transfer-interpolation*" operation. SMS transfer-interpolation can be used to parallelize any inter-grid interpolation that uses weighted sums of values in a source grid to compute values in a destination grid.

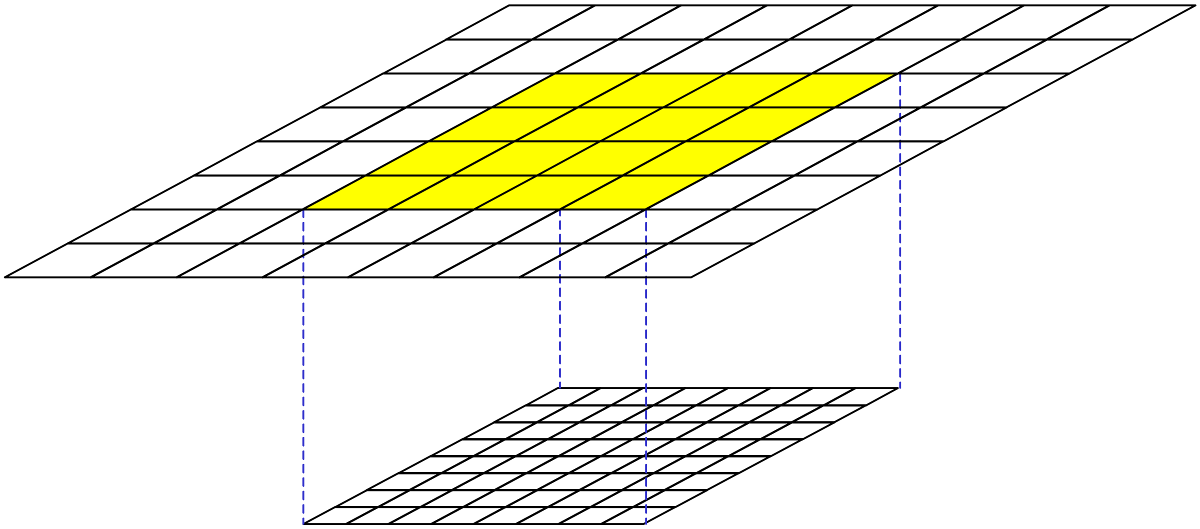


Figure 11-2: Computing overlapping points in a low-resolution “parent” grid from points in a high-resolution “child” grid.

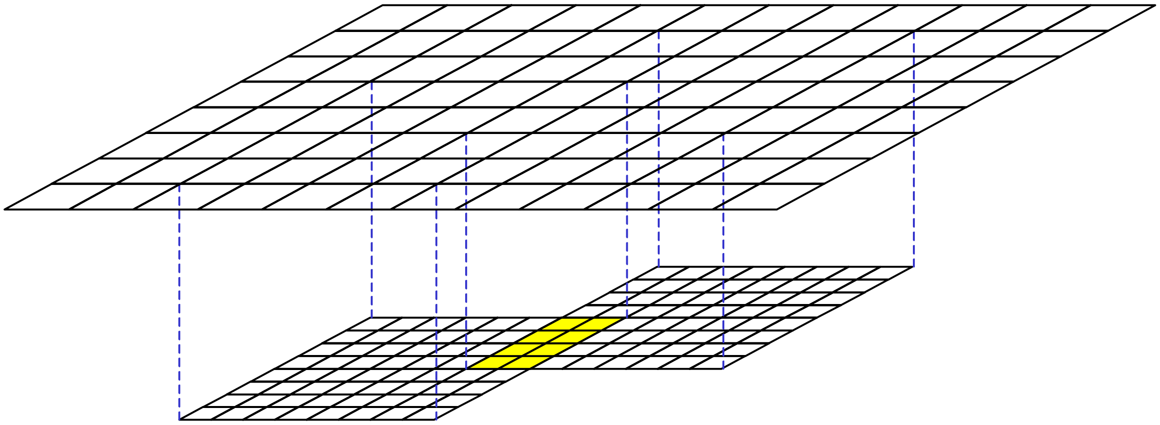


Figure 11-3: Reconciling overlapping points in “sibling” grids.

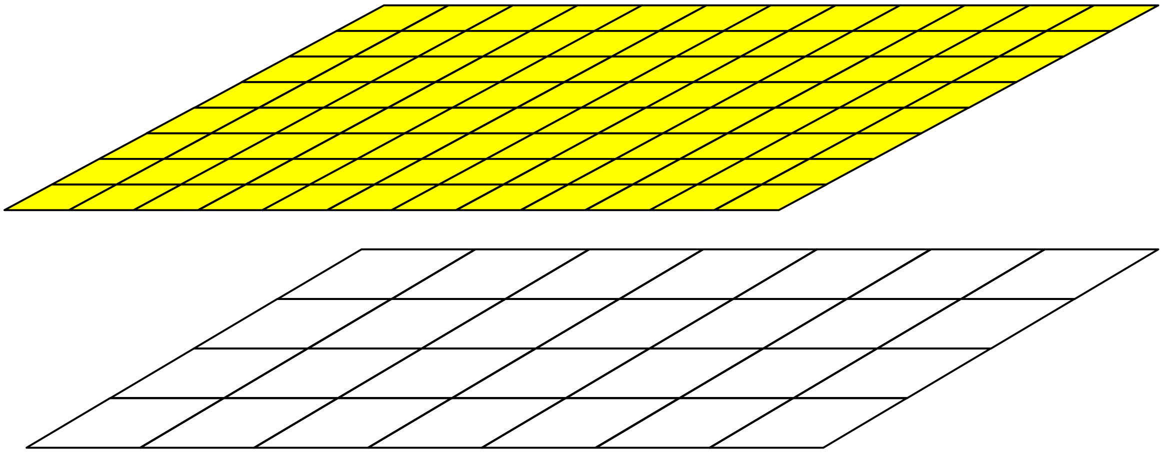


Figure 11-4: Coupling different grids.

11.1 Using SET_TRANSFER_INTERPOLATE to Define Interpolations

The SET_TRANSFER_INTERPOLATE directive is used to specify any inter-grid interpolation scheme that can be expressed as weighted sums. Before SET_TRANSFER_INTERPOLATE can be used, the serial code must be transformed so that source grid coordinates, destination grid coordinates, and weights are each stored in arrays. Code in this form will be referred to as “*stencil-oriented*”.

11.1.1 Transforming Serial Code to “Stencil-Oriented” Form

Example 11-1 shows a simple stencil-oriented serial code fragment that is used to compute elements of fine-grid array *X_FINE* from weighted sums of elements of coarse-grid array *X_COARSE*. In this code, *NUM_FINE_POINTS* is the number of elements to be computed in array *X_FINE*. *MAX_STENCIL_POINTS* is the maximum number of elements from *X_COARSE* that will be used to compute any element in *X_FINE* by weighted sum. The coordinates of each element of *X_FINE* to be computed are stored in array *fine_indices*. The coordinates of each element of *X_COARSE* that are used to compute each element of *X_FINE* are stored in array *stencil_indices*. The corresponding weights are stored in array *stencil_weights*. (Initializations of *fine_indices* , *stencil_indices* , and *stencil_weights* are not shown). The actual weighted-sum computations are performed in the loop beginning at line 13. Figure 11-5 illustrates details of the weighted sum used to compute one of the elements of *X_FINE*.

```
1       real X_COARSE(imc,jmc)
2       real X_FINE(imf,jmf)
3
4       integer    fine_indices(2,NUM_FINE_POINTS)
5       integer stencil_indices(2,MAX_STENCIL_POINTS,
6      &                          NUM_FINE_POINTS)
7       real      stencil_weights(MAX_STENCIL_POINTS,
8      &                          NUM_FINE_POINTS)
9
```

```
10 ...Initialize fine_indices, stencil_indices, and stencil_weights...
11
12 C NOTE "stencil-oriented" form of serial interpolation code
13       do ifp = 1,NUM_FINE_POINTS
14         i = fine_indices(1,ifp)
15         j = fine_indices(2,ifp)
16         X_FINE(i,j) = 0.0
17         do icp = 1,MAX_STENCIL_POINTS
18           X_FINE(i,j) = X_FINE(i,j) +
19      &                    (stencil_weights(icp,ifp) *
20      &          X_COARSE(stencil_indices(1,icp,ifp),
21      &                   stencil_indices(2,icp,ifp)))
22         enddo
23       enddo
```

Example 11-1: Serial inter-grid interpolation code in "stencil-oriented" form.

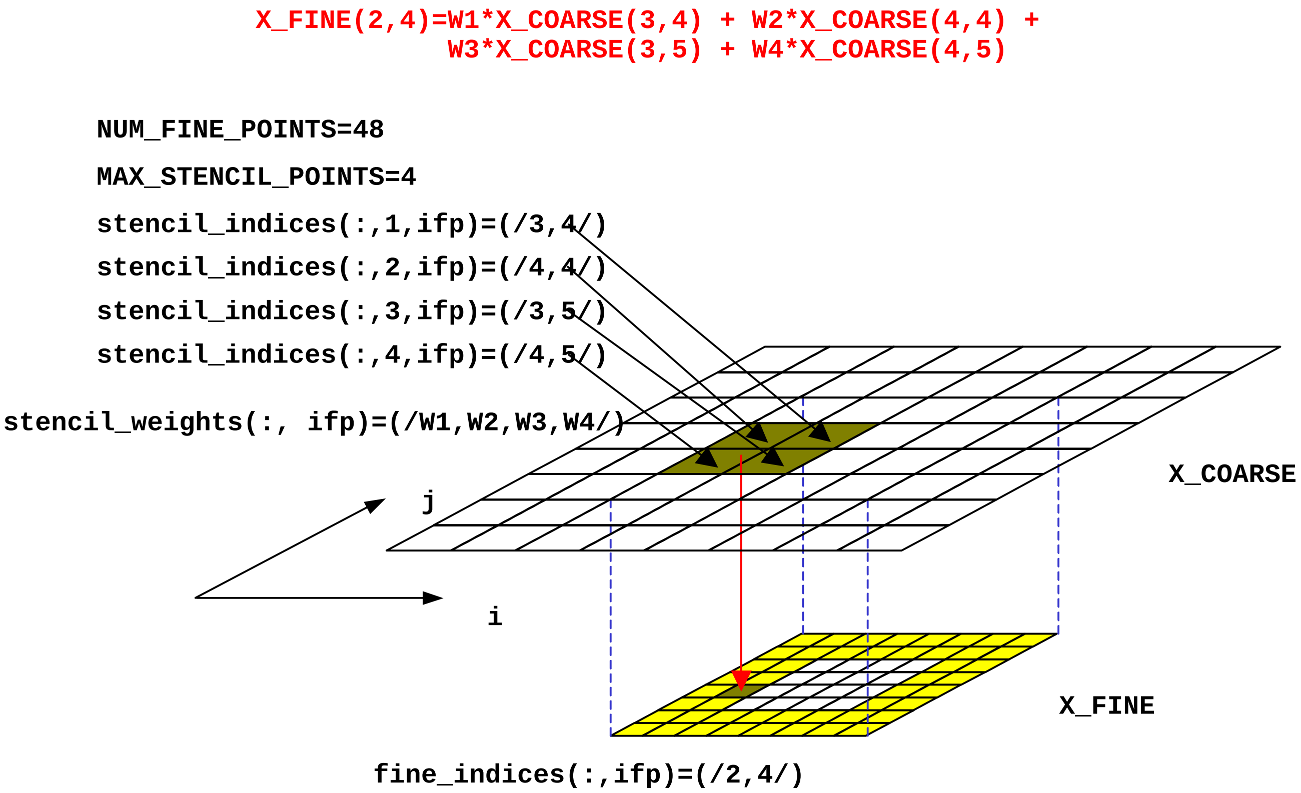

X_FINE(2,4)=W1*X_COARSE(3,4) + W2*X_COARSE(4,4) +
W3*X_COARSE(3,5) + W4*X_COARSE(4,5)
NUM_FINE_POINTS=48
MAX_STENCIL_POINTS=4
stencil_indices(:,1,ifp)=(/3,4/)
stencil_indices(:,2,ifp)=(/4,4/)
stencil_indices(:,3,ifp)=(/3,5/)
stencil_indices(:,4,ifp)=(/4,5/)
stencil_weights(:, ifp)=(/W1,W2,W3,W4/)
j
i
X_COARSE
X_FINE
fine_indices(:,ifp)=(/2,4/)

Figure 11-5: Detail of coarse-to-fine interpolation computations. Integer *ifp* is the loop index used on line 13 of Example 11-1.

11.1.2 Defining an Interpolation

Once the serial code is in "stencil-oriented" form, the SET_TRANSFER_INTERPOLATION directive can be used to define inter-grid interpolations. Example 11-2 shows how this is done for the inter-grid interpolation introduced in Example 11-1. Arrays *X_COARSE* and *X_FINE* are distributed using decompositions *dhCoarse* and *dhFine*, respectively. The SET_TRANSFER_INTERPOLATION directive begins on line 17.

```
 1 CSMS$DISTRIBUTE(dhCoarse, <imc>, <jmc>) BEGIN
 2       real X_COARSE(imc,jmc)
 3 CSMS$DISTRIBUTE END
 4 CSMS$DISTRIBUTE(dhFine, <imf>, <jmf>) BEGIN
 5       real X_FINE(imf,jmf)
 6 CSMS$DISTRIBUTE END
 7
 8       integer    fine_indices(2,NUM_FINE_POINTS)
 9       integer stencil_indices(2,MAX_STENCIL_POINTS,
10      &                          NUM_FINE_POINTS)
11       real      stencil_weights(MAX_STENCIL_POINTS,
12      &                          NUM_FINE_POINTS)
13
14 CSMS$PARALLEL(dhFine, <i>, <j>) BEGIN
15 ...Initialize fine_indices, stencil_indices, and stencil_weights...
16
17 CSMS$SET_TRANSFER_INTERPOLATION( dhCoarse, dhFine, 2,
18 CSMS$>  NUM_FINE_POINTS, fine_indices, MAX_STENCIL_POINTS,
19 CSMS$>  stencil_indices, stencil_weights, INTERP_C_F)
20
21 CSMS$TRANSFER( <X_COARSE, X_FINE : INTERP_C_F> ) BEGIN
22 C NOTE “stencil-oriented” form of serial interpolation code
23       do ifp = 1,NUM_FINE_POINTS
24         i = fine_indices(1,ifp)
25         j = fine_indices(2,ifp)
26         X_FINE(i,j) = 0.0
27         do icp = 1,MAX_STENCIL_POINTS
28           X_FINE(i,j) = X_FINE(i,j) +
29      &                    (stencil_weights(icp,ifp) *
30      &          X_COARSE(stencil_indices(1,icp,ifp),
31      &                   stencil_indices(2,icp,ifp)))
32         enddo
33       enddo
34 CSMS$TRANSFER END
```

Example 11-2: The SET_TRANSFER_INTERPOLATION directive is used to define the inter-grid interpolation introduced in Example 11-1. The TRANSFER directive performs the computations and inter-process communications required to interpolate between grids.

The first two arguments in the SET_TRANSFER_INTERPOLATION directive are the names of the decompositions used by the source and destination arrays. The third argument is the rank of the interpolation, which cannot be larger than the rank of the source or destination arrays. The fourth argument is the number of elements in the destination array that will be computed during interpolation and the fifth argument is a list of coordinates for each element. The maximum number of elements from the source array that will be used to compute any element of the destination array is indicated by the sixth argument. The seventh argument is an array of lists of coordinates of elements from the source array that will be used to compute each element of the destination array. The weights used to interpolate each element of the destination array are indicated by the eighth argument. Finally, the last argument is a user-defined name for the interpolation. This name is used again in the TRANSFER directive to select the interpolation scheme. Finally, the code that initializes `fine_indices`, `stencil_indices`, and `stencil_weights` must appear inside a PARALLEL directive (see line14 of Example 11-2).

Note that any array dimension may be interpolated whether or not it is decomposed. The independence of interpolated dimensions and decomposed dimensions allows full flexibility to support any weighted-sum interpolation scheme with any decomposition. However, to minimize inter-process communication, interpolation of non-decomposed dimensions should be decoupled and handled separately whenever possible (see Example 11-4).

11.2 Using TRANSFER to Interpolate Between Grids

The TRANSFER directive has special syntax for inter-grid interpolations as shown on line 21 of Example 11-2. The source and destination arrays (***X_COARSE*** and ***X_FINE***) are enclosed in angle brackets (see Section 6). In addition, a third argument (***INTERP_C_F***) references the name of the interpolation, defined by the SET_TRANSFER_INTERPOLATION directive (line 19 of Example 11-2). Note that the serial code enclosed between the TRANSFER “BEGIN” and “END” must include all interpolation computations to ensure that the serial code works correctly. During source code translation, code between the “BEGIN” and “END” is replaced with its parallel equivalent.

Transfer-interpolation operations are aggregated to reduce latency in the same way aggregation is done for standard transfers. Note that aggregation of multiple interpolations is done even if the source or destination arrays are not decomposed in the same way or if different interpolations are used. Aggregation of transfer-interpolation operations is illustrated in Example 11-3 for a simple case where arrays *Y_COARSE* and *Y_FINE* are decomposed like *X_COARSE* and *X_FINE*. Aggregation is achieved by combining the transfer-interpolations into a single TRANSFER directive (lines 1 and 2).

```
 1 CSMS$TRANSFER( <X_COARSE, X_FINE : INTERP_C_F>,
 2 CSMS$>         <Y_COARSE, Y_FINE : INTERP_C_F>) BEGIN
 3       do ifp = 1,NUM_FINE_POINTS
 4         i = fine_indices(1,ifp)
 5         j = fine_indices(2,ifp)
 6         X_FINE(i,j) = 0.0
 7         Y_FINE(i,j) = 0.0
 8         do icp = 1,MAX_STENCIL_POINTS
 9           X_FINE(i,j) = X_FINE(i,j) +
10      &                     (stencil_weights(icp,ifp) *
11      &          X_COARSE(stencil_indices(1,icp,ifp),
12      &                    stencil_indices(2,icp,ifp)))
13           Y_FINE(i,j) = Y_FINE(i,j) +
14      &                     (stencil_weights(icp,ifp) *
15      &          Y_COARSE(stencil_indices(1,icp,ifp),
16      &                    stencil_indices(2,icp,ifp)))
17         enddo
18       enddo
19 CSMS$TRANSFER END
```

Example 11-3: This form of the TRANSFER performs inter-grid interpolations to compute elements of both *X_FINE* and *Y_FINE*. When a single TRANSFER directive is used for multiple interpolations, messages are aggregated. This will improve performance on some machines by reducing message latency. Note that the serial code enclosed by the TRANSFER directive must perform interpolation computations for all destination arrays.

In some circumstances, it is convenient to use an interpolation for computations on arrays with higher rank. For example, suppose that arrays *X_COARSE* and *X_FINE* from

Example 11-2 each had a third, non-decomposed dimension with size *km*. Furthermore, suppose that interpolation between *X_COARSE* and *X_FINE* was independent of the new third dimension. The same interpolation, *INTERP_C_F* could still be used as shown in Example 11-4. The syntax ***INTERP_C_F(1,2)*** in the TRANSFER directive on line 1 indicates that the first two dimensions of the arrays correspond to the first two dimensions of the interpolation. SMS will automatically replicate interpolation computations in the third array dimension.

```
 1 CSMS$TRANSFER( <X_COARSE, X_FINE : INTERP_C_F(1,2)> ) BEGIN
 2       do k = 1,km
 3         do ifp = 1,NUM_FINE_POINTS
 4           i = fine_indices(1,ifp)
 5           j = fine_indices(2,ifp)
 6           X_FINE(i,j,k) = 0.0
 7           do icp = 1,MAX_STENCIL_POINTS
 8             X_FINE(i,j,k) = X_FINE(i,j,k) +
 9      &                      (stencil_weights(icp,ifp) *
10      &                X_COARSE(stencil_indices(1,icp,ifp),
11      &                         stencil_indices(2,icp,ifp),k))
12           enddo
13         enddo
14       enddo
15 CSMS$TRANSFER END
```

Example 11-4: Interpolations may be used for arrays with higher rank. Here, interpolation of 3D array *X_FINE* from 3D array *X_COARSE* is done using 2D decomposition *INTERP_C_F* (defined on lines 14-16 of Example 11-2).

11.3 Using SET_NEST_LEVEL to Switch Between Grids

In the examples shown so far, the source and destination decompositions have had different names (*dhCoarse* and *dhFine* in Example 11-2). However, it is often desirable to define a single decomposition name and use it to describe both the source and destination decompositions. In Example 11-5, subroutine *SOLVE* is called twice, once with *X_COARSE* (line 29) and once with *X_FINE* (line 36). However, *X_COARSE* and *X_FINE* are decomposed differently. To avoid passing SMS-specific decomposition information through the interface of subroutine *SOLVE* (which would unnecessarily complicate the serial code), SMS allows a single decomposition name to refer to more than one decomposition. Individually, these component decompositions are called "*nests*".

In Example 11-5, decomposition *dh* is declared to have two nests by using the syntax "***dh(2)***" in the DECLARE_DECOMP directive (line 4). The nests are then referred to individually using "***dh(1)***" (coarse nest) and "***dh(2)***" (fine nest) on lines 10, 11, 21, and 24. SMS directives inside subroutine SOLVE do not explicitly reference a specific nest (***dh(1)*** or ***dh(2)***). Instead, the nest level is selected before each call to SOLVE by SET_NEST_LEVEL directives on lines 28 and 35. Unlike most other SMS directives, the effects of the SET_NEST_LEVEL directive carry over into called subroutines. The selected nest will remain selected until another SET_NEST_LEVEL directive is encountered.

```
 1 [include file params.inc]
```

```
      integer imc,jmc,imf,jmf
      common /sizes_com/ imc,jmc,imf,jmf
CSMS$DECLARE_DECOMP(dh(2), 2)


[main program source file]
C ...
      include 'params.inc'
CSMS$CREATE_DECOMP(dh(1), <imc, jmc>, <3,3>)
CSMS$CREATE_DECOMP(dh(2), <imf, jmf>, <3,3>)
C ...
CSMS$SET_TRANSFER_INTERPOLATION( dh(1), dh(2), 2,
CSMS$>  NUM_FINE_POINTS, fine_indices, MAX_STENCIL_POINTS,
CSMS$>  stencil_indices, stencil_weights, INTERP_C_F)


[subroutine source file]
C ...
      include 'params.inc'
CSMS$DISTRIBUTE(dh(1), <imc>, <jmc>) BEGIN
      real X_COARSE(imc,jmc)
CSMS$DISTRIBUTE END
CSMS$DISTRIBUTE(dh(2), <imf>, <jmf>) BEGIN
      real X_FINE(imf,jmf)
CSMS$DISTRIBUTE END

CSMS$SET_NEST_LEVEL( dh, 1 )
      CALL SOLVE(X_COARSE,imc,jmc)

CSMS$TRANSFER( <X_COARSE, X_FINE : INTERP_C_F> ) BEGIN
C ...
CSMS$TRANSFER END

CSMS$SET_NEST_LEVEL( dh, 2 )
      CALL SOLVE(X_FINE,imf,jmf)


[another subroutine source file]
      SUBROUTINE SOLVE(X,im,jm)
      include 'params.inc'
CSMS$DISTRIBUTE(dh, <im>, <jm>) BEGIN
      real X(im,jm)
CSMS$DISTRIBUTE END
      integer i,j
CSMS$PARALLEL(dh, <i>, <j>) BEGIN
C ...
CSMS$PARALLEL END
      END
```

Example 11-5: Subroutine *SOLVE* must be called with either *X_COARSE* or *X_FINE* as the first argument. Since these two arrays are decomposed differently, the best way to do this is to create a single decomposition with two “nests”, dh(2). The SET_NEST_LEVEL directive selects which nest is used.

12 I/O

A powerful feature of SMS is its ability to support most types of I/O without directives. The fact that communication patterns for I/O of decomposed and non-decomposed arrays differ is hidden from the programmer. SMS automatically generates the communication needed to read or write data to or from disk in the same sequence as the serial code would have done it regardless of the number processes used. Unformatted I/O is discussed in Section 12.1 while formatted I/O is covered in Section 12.2. Printing in a parallel program often requires additional decisions from the programmer. SMS allows the programmer to make these decisions by providing several print modes that are introduced in Section 12.2.2. Finally, Section 12.3 discusses methods for improving I/O performance.

12.1 Unformatted I/O

Figure 12-1 illustrates dependencies for read and write of a simple one-dimensional decomposed array. During a read, data from a single file must be parceled out to each process. This type of communication pattern is called "scatter". During a write, data from each process must be combined in the “proper order” and written to disk. This type of communication pattern is a different form of "gather" than the form used for TRANSFER and bit-wise exact REDUCE. Instead of gathering data into a global variable that is replicated in memory on all processes, data is gathered into a single file on disk. "Proper order" means the data must be read from or written to disk in the same sequence that the serial code uses. Though it appears quite simple in Figure 12-1, the data reorganization required to match serial ordering in files can be quite complex, especially for two-dimensional decompositions or when the decomposed arrays have halo regions (Figure 12-2). Additionally, when variables being input have halo regions associated with them, these regions are automatically updated by SMS.

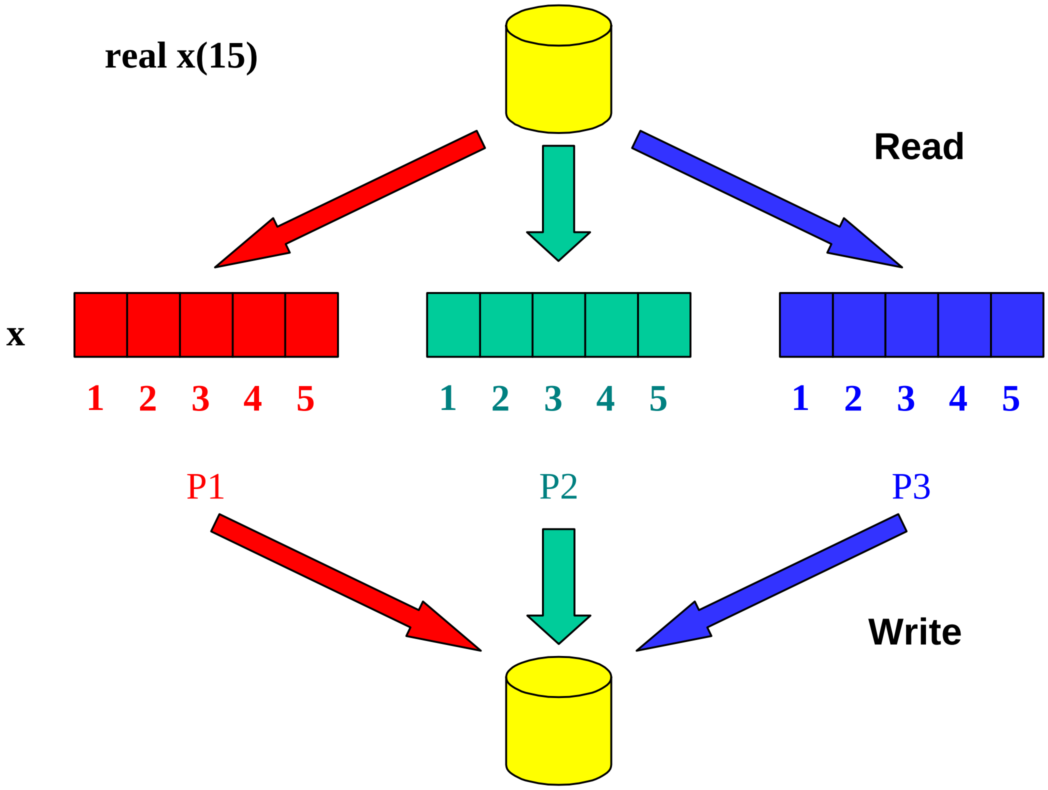


Figure 12-1: Conceptual schematic of the input and output of a decomposed array. On input, one process reads the global data from disk. The appropriate sections of the global array are then "scattered" to each process. On output, the decomposed data are gathered into a global array and then written to disk. The underlying implementation may use a more efficient scheme on some machines.

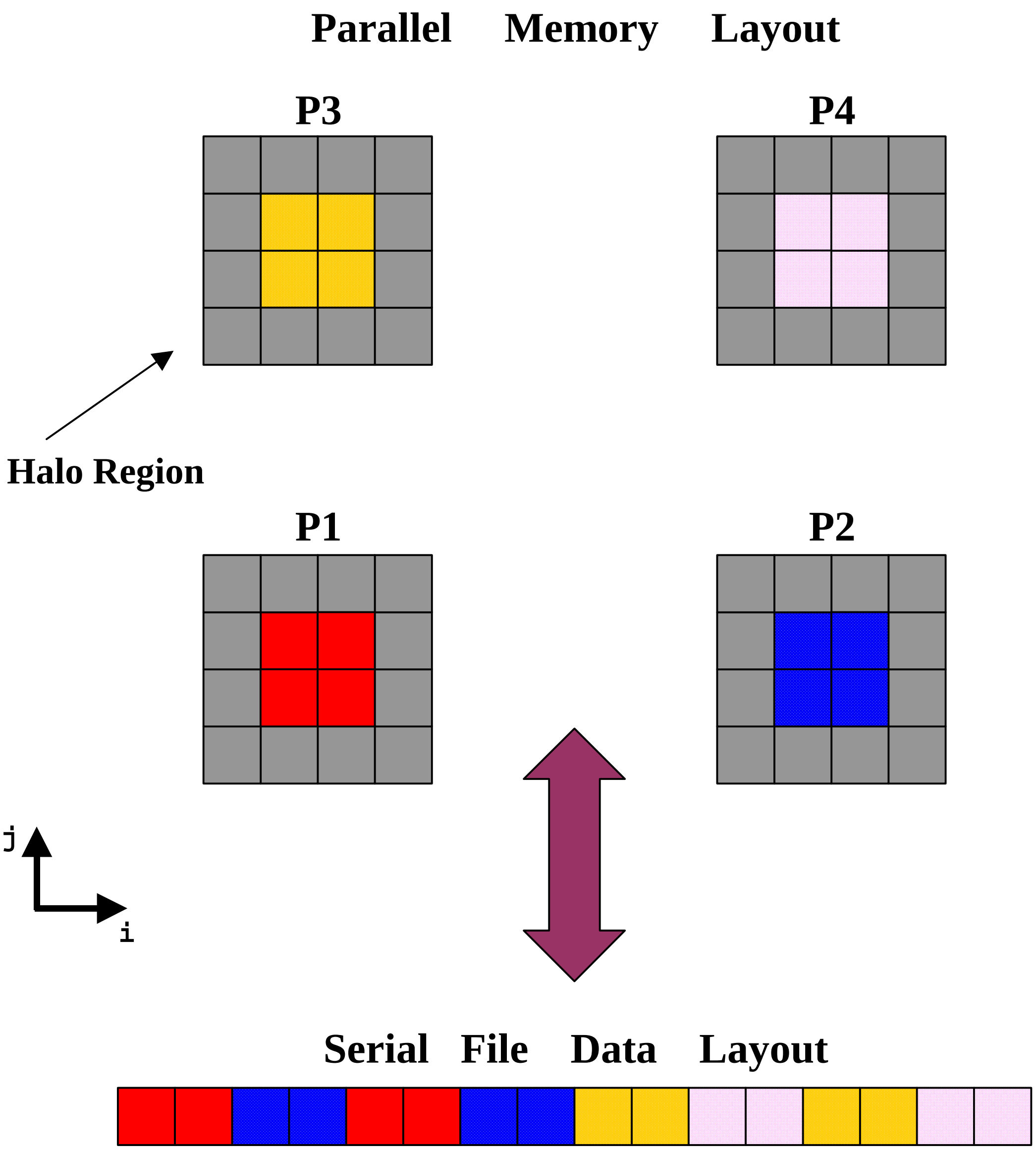


Figure 12-2: Conceptual schematic of the re-ordering required to write and read two-dimensionally decomposed data to disk in the same order as the serial code would write it. Special care has to be taken to write the only the interior of each process-local domain and not the halo data. The halo regions are filled during the read operations.

Figure 12-3 illustrates dependencies for read and write of a non-decomposed variable. During a read, a copy of data from a single file must be sent to each process. This type of communication pattern is called "broadcast". During write, it is only necessary to write data from a single process because each process should have an identical copy of the variable.

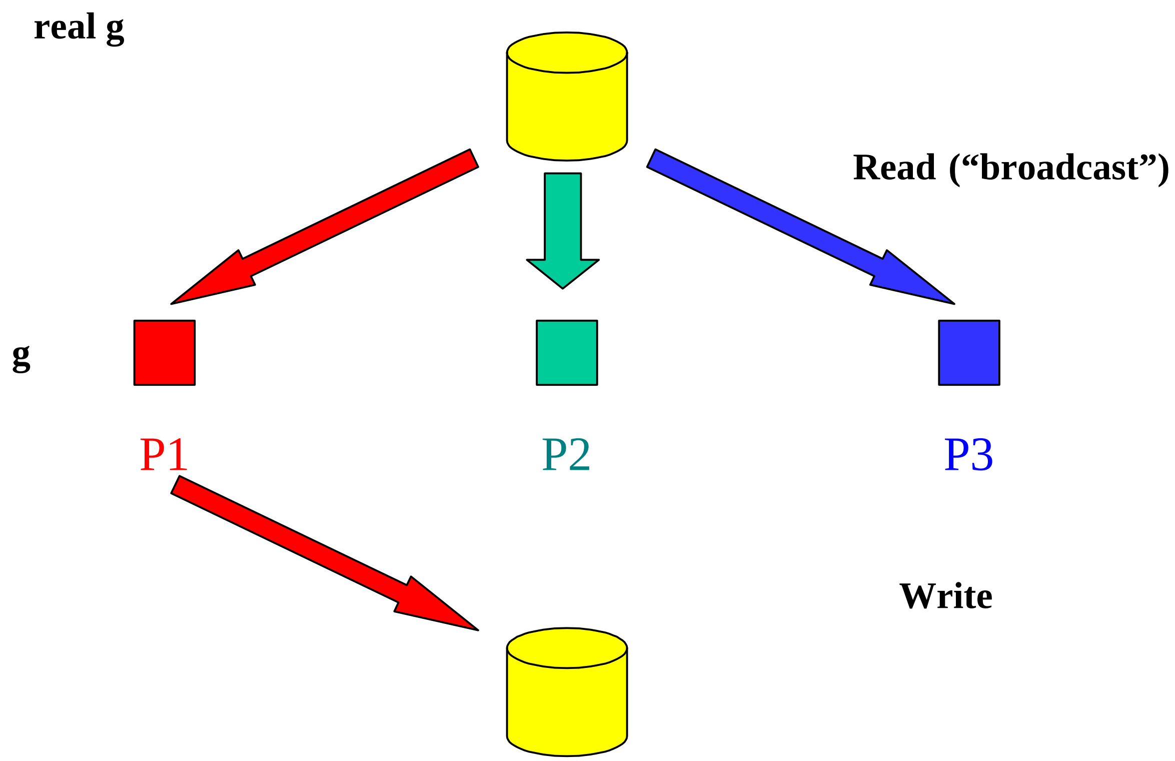


Figure 12-3: Schematic of the input and output of a non-decomposed array. On input, one process reads the data from disk. The data are then replicated on all other processes. On output, a single process writes the data to disk.

Example 12-1 demonstrates unformatted I/O of both decomposed and non-decomposed variables. SMS automatically translates the read (line 32) and write (line 28) statements for both decomposed arrays *x* and *y* and non-decomposed scalar *scale* to the appropriate parallel I/O operations. When automatically generating parallel I/O operations, SMS uses information in the DISTRIBUTE directives to determine how to scatter, gather, or broadcast data. Notice that any types of decomposed or non-decomposed variables can be mixed in a single write or read statement. It is not necessary to reorganize existing serial read or write statements to take advantage of automatic parallelization by SMS.

```
[Include file:  io.inc]

 1        integer im, jm
 2        common /sizes_com/ im, jm
 3  CSMS$DECLARE_DECOMP(DECOMP_IJ, 2)
```

```
[Source file:  binary.f]

 1        program binary_io
 2        include 'io.inc'
 3        im = 10
 4        jm = 5
```

```
CSMS$CREATE_DECOMP(DECOMP_IJ, <im,jm>, <1,0>)
      call write_data
      end

      subroutine write_data
      include 'io.inc'
      integer i, j
      real scale
CSMS$DISTRIBUTE(DECOMP_IJ, <im>, <jm>) BEGIN
      integer x(im,jm), y(im,jm)
CSMS$DISTRIBUTE END
CSMS$PARALLEL(DECOMP_IJ,<i>,<j>) BEGIN
      do j=1,jm
        do i=1,im
CSMS$TO_GLOBAL(<1,i>, <2,j>) BEGIN
          x(i,j) = (100 * i) + j
          y(i,j) = mod(i,2)
CSMS$TO_GLOBAL END
        end do
      end do
CSMS$PARALLEL END
      scale = -1.0
      open (17,file='io1_out.dat',form='unformatted')
      write (17) x, y, scale
      close (17)

      open (18,file='io1_out.dat',form='unformatted')
      read  (18) x, y, scale
      close (18)
      return
      end
```

Example 12-1: This program does output and input of both decomposed and non-decomposed data. No additional directives are required for the correct I/O to be performed, regardless of the number of processes.

By default, SMS assumes unformatted files are stored in native FORTRAN binary format. The default behavior can be modified using the following environment variables:

```
SMS_READ_FORMAT
SMS_WRITE_FORMAT
```

The currently available (case insensitive) formats are:

```
IBM
SUN
SGI
FUJITSU
HP
DEC
COMPAQ
IA32
MPI
MPI_EXTERNAL
EXTERNAL
SMS
```

In some cases, file formats with different names are actually the same format. For example, *SGI* and *SUN* are really the same. Note that *MPI*, *MPI_EXTERNAL*, *EXTERNAL*, and *SMS* all refer to the portable MPI I/O external format. The advantage to using this format is that any file written by an SMS program may be read by any other SMS program on any other machine. This is true regardless of the number of processes used on either machine because SMS preserves serial data ordering. To convert data files from one format to another, simply write a serial program that reads and writes the data, compile and link with SMS and then set the above environment variables appropriately.

12.2 Formatted I/O

SMS supports the formatted input and output of non-decomposed data without directives. Also supported is unformatted I/O via namelist of both decomposed and non-decomposed data. Formatted I/O of decomposed variables by means other than namelist is not currently supported, so code segments that include this kind of I/O must be enclosed by a SERIAL directive (see Section 8). In a future release of SMS, this use of SERIAL will become optional.

12.2.1 Formatted Input

Formatted input including namelists is handled automatically by SMS. The user does not need to add any directives. The only caveat is that input variables cannot be decomposed arrays unless a namelist is used. In this case, a work-around is to enclose the formatted read statements within a SERIAL directive. Since formatted reads typically occur infrequently during the course of a model run, this approach usually does not incur a significant performance penalty.

12.2.2 Formatted Output

The simple task of printing a message on the screen becomes complicated in a parallel program. Consider the following simple print statement:

```
      print *,'HELLO'
```

There are no clear standard definitions of what will appear on the screen when a "parallel" print statement is executed. Will each process print a separate message? Will the separate messages appear on different lines on the screen? Will all processes be forced to wait until the print is complete before useful work can continue? If the statement were executed on three processes, we might see any of the following output:

```
HELLO

HELLO
HELLO
HELLO

HHHEEELLLLLLOOO

HELLHEHLEOLOLLO
```

During the brief history of parallel computing, each of these possibilities has been implemented on at least one parallel machine.

SMS simplifies this situation by providing three "print modes" that allow the user to control the behavior of parallel print. The modes are default, ASYNC, and ORDERED. These modes are selected using the PRINT_MODE directive. The PRINT_MODE directive may only be used for formatted output to stdout. This may be accomplished either by using *PRINT* statements or *WRITE* statements when the unit number is either *6* or *. (Note that units 0, 5, and 6 should not be opened in an SMS program because many Fortran compilers behave strangely when these units are connected to named files.) The print modes are illustrated in Example 12-2.

```
      program print_modes
      implicit none
      integer, parameter :: im = 12
      integer xmax, i

CSMS$DECLARE_DECOMP(dh, 1)

CSMS$DISTRIBUTE(dh, 1) BEGIN
      integer, allocatable :: x(:)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(dh, <im>, <0>)
      allocate(x(im))

CSMS$SERIAL BEGIN
      do i = 1, im
        x(i) = i
      end do
CSMS$SERIAL END

CSMS$PARALLEL(dh,<i>) BEGIN
      xmax = 0
      do i = 1,im
        xmax = max(xmax,x(i))

        if (x(i) .ge. 12) then
CSMS$PRINT_MODE(ASYNC) BEGIN
          print *,'WARNING:  x .ge. 12 !!  '
CSMS$PRINT_MODE END
        endif

      end do
CSMS$PARALLEL END

CSMS$PRINT_MODE(ORDERED) BEGIN
CSMS$INSERT      print *,'DEBUG:  local maximum value = ',xmax
CSMS$PRINT_MODE END

CSMS$REDUCE(xmax,MAX)

      write( *,900) 'maximum value = ',xmax
  900 format(a, i4)
      end
```

Example 12-2: Program that illustrates the various print modes supported by SMS.

When the serial code versions *print_modes* is run, the following is printed on the screen:

```
>> print_modes
```

```
 WARNING:  x .ge. 12 !!
maximum value =   12
```

When the parallel code is run on 1 process, the following is printed on the screen:

```
>> smsRun -np 1 print_modes_parallel
 WARNING:  x .ge. 12 !!
 DEBUG:  local maximum value = 12
maximum value =   12
```

For 4 processes:

```
>> smsRun -np 4 print_modes_parallel
 WARNING:  x .ge. 12 !!
 DEBUG:  local maximum value =  3
 DEBUG:  local maximum value =  6
 DEBUG:  local maximum value =  9
 DEBUG:  local maximum value = 12
maximum value =   12
```

The write statement on line 41 in Example 12-2, is printed using the default print mode. The default print mode is used for any print statement that is not enclosed by a PRINT_MODE directive. It will cause the parallel code to print the same messages as the serial code in most cases. Only one system-dependent designated process (the "root" process) will execute the print statement; the others will skip it and can immediately continue with useful computations.

The print statement on line 36 is executed using the ORDERED print mode. The ORDERED print mode may only be selected using the PRINT_MODE directive. This mode causes one message to be printed on the screen for each process and guarantees that the messages always appear in the same order. It is most useful for debugging purposes. However, in order to guarantee message ordering, no process can continue until all processes have executed the print statement. This means care must be taken that all processes will ALWAYS execute an ordered print or the program will hang. For example, suppose we use the ORDERED print mode at line 27:

```
      if (x(i) .ge. 12) then
CSMS$PRINT_MODE(ORDERED) BEGIN
        print *,'WARNING:  x .ge. 12 !!  '
CSMS$PRINT_MODE END
        endif
```

In this case, we see the same results for the one-process run. However, the four-process run produces the following results:

```
>> smsRun -np 4 print_modes_parallel
 DEBUG:  local maximum value =  3
 DEBUG:  local maximum value =  6
 DEBUG:  local maximum value =  9
 WARNING:  x .ge. 12 !!
```

In this case, the program hangs (deadlocks) before the final message can be printed because the warning print statement is now an ordered-mode print that has been executed by only one process. The program will wait forever for the other processes to enter this print statement. The default print mode is also inappropriate here because the warning message would not be printed if point 12 were not on the root process. Deadlock would not occur, but the warning message would not be printed.

The ASYNC (asynchronous) mode is the proper mode to use in cases like the printed warning statement on line 28 because there is no guarantee that all processes will execute the print statement. In this mode, one message will appear on the screen for each process that executes the print statement. Like the default mode, there is no process synchronization during asynchronous prints. As a result, ordering of print statements may vary from one run to the next when ASYNC mode is used. Like the ORDERED mode, the ASYNC print mode may only be selected using the PRINT_MODE directive. For example, suppose we use the ASYNC mode for line 35 instead of ORDERED.

```
CSMS$PRINT_MODE(ASYNC) BEGIN
CSMS$INSERT      print *,'DEBUG:  local maximum value = ',xmax
CSMS$PRINT_MODE END
```

Running with four processes two times might produce the following results:

```
>> smsRun -np 4 print_modes_parallel
 DEBUG:  local maximum value =  3
 DEBUG:  local maximum value =  6
 DEBUG:  local maximum value =  9
 WARNING:  x .ge. 12 !!
 DEBUG:  local maximum value = 12
maximum value =   12

>> smsRun -np 4 print_modes_parallel
 DEBUG:  local maximum value =  6
 DEBUG:  local maximum value =  3
 WARNING:  x .ge. 12 !!
 DEBUG:  local maximum value =  9
 DEBUG:  local maximum value = 12
maximum value =   12
```

Note that the ASYNC-mode prints can appear in any order and can even appear out-of-order with other non-ASYNC-mode prints. This can be confusing in some cases. Also, the ASYNC mode does not work properly when the SMS program is being run in "serverless" mode (see Section 12.3.2).

12.3 I/O Performance Tuning

This section discusses methods for optimizing SMS I/O performance.

Please note that the I/O components of SMS are currently being re-designed to improve performance and simplify optimization options. Most of the environment variables that have been available to tune I/O performance will become obsolete in a future release. Discussion of these "deprecated" features has been omitted from this version of the SMS Users Guide. Please refer to an earlier version for discussion of deprecated features.

12.3.1 The SMS Server Process

By default SMS designates an additional process, the “server” process, to handle all formatted and unformatted I/O operations. This allows computations to be done concurrent with I/O operations and can improve the overall performance of an SMS program. Figure 12-4 illustrates a program run using four compute processes and an SMS server process.

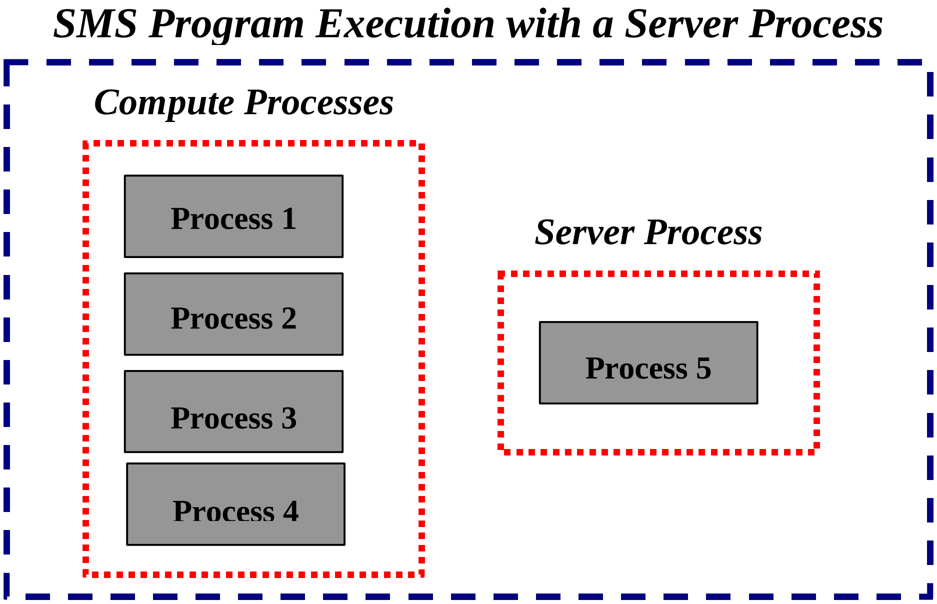


Figure 12-4: In this example, four processes are requested to run an SMS program. By default, an additional process, the “server” process, will be used by SMS for I/O operations.

12.3.2 Serverless I/O

For small numbers of processors (approximately less than 8 depending on the amount of I/O), it may be beneficial to combine the server process functions with one of the computational processes. This type of operation is called serverless I/O and is illustrated in Figure 12-5.

If serverless I/O is used, the I/O functions that would normally be run on a separate process will be combined with one of the compute processes. Serverless SMS can be requested through an environment variable given by the command:

```
>> setenv SMS_SERVER_MODE serverless
```

On most machines, for small numbers of processors, where there is a one-to-one correspondence between processes and processors, serverless I/O will improve performance by making one more processor available to do computations. However, when large numbers of processors are used, program execution will usually be faster with a server.

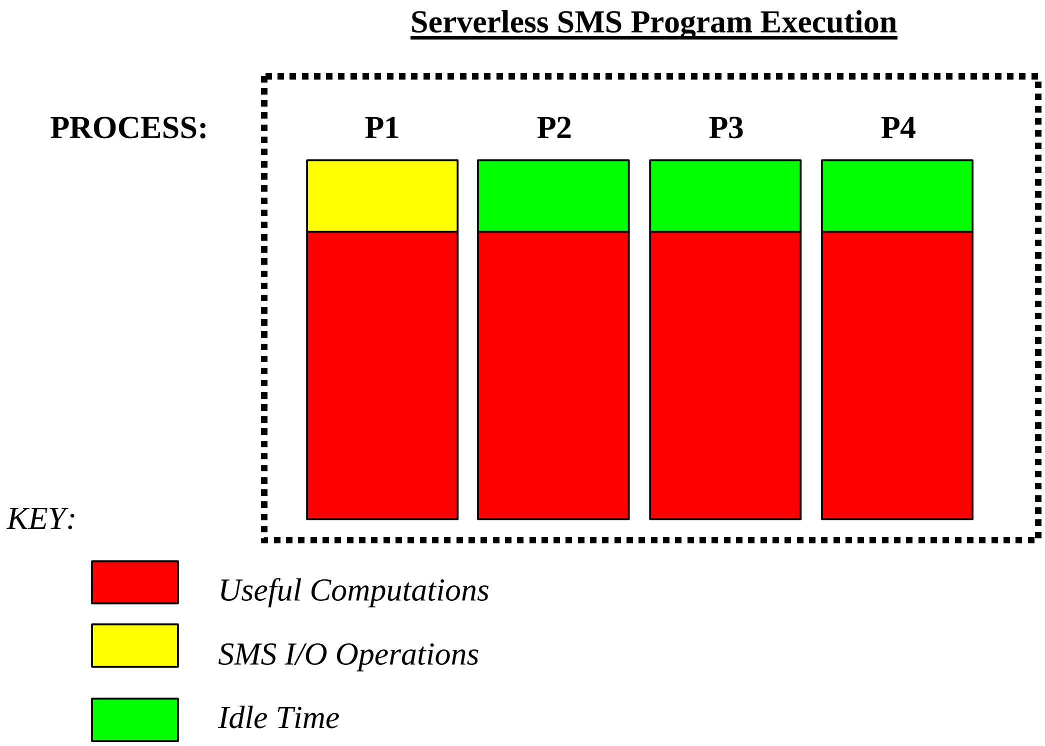


Figure 12-5: An illustration of four SMS processes used to run a program without a server process. In this example, process P1 must handle both program computations and SMS server functions that include I/O operations. While these operations occur, the other processes will be idle.

12.3.3 The FLUSH_OUTPUT Directive

During write operations when a server process is present, the server buffers the data to be output, re-orders any decomposed data into serial order, and then writes to disk in large blocks. By default, any write to disk will be delayed until the buffer is full or the file is closed. When this happens, buffers are "flushed" and their contents written to disk in large blocks.

The environment variable SMS_CLOSE_MODE can be set to “require-flush” for improved user control of when SMS output buffers are flushed. Then, performance improvement can be gained by controlling when these buffers are flushed using the FLUSH_OUTPUT directive. This directive instructs the SMS I/O server process to flush the buffers immediately. The FLUSH_OUTPUT directive should be placed after an output cycle is finished so the output will be done while the next cycle of computations is being done.

The following code fragment shows how this directive can be used:

```
      open  (20,file='main_fields.dat',form='unformatted')
      write (20) u,v,w,p,t
      close (20)

      open  (30,file='moisture.dat',form='unformatted')
      write (30) qs,qi,qr,qg,qw
      close (30)
CSMS$FLUSH_OUTPUT
c useful computation ...
```

Example 12-3: Proper placement of a FLUSH_OUTPUT directive.

In this example, two files are written. Useful computations will proceed while data is simultaneously re-ordered and written to disk.

13 “Sliced” Arrays

When distribution of an array does not involve all decomposed dimensions, the distribution is called a “slice” and the array is referred to as a “sliced array”. Currently, the exact manner in which a sliced array is distributed is poorly defined in SMS, so there are some limitations on their use. One limitation is demonstrated in Example 13-1. The DISTRIBUTE statement on line 6 of defines a sliced array whose first dimension is decomposed according to the second dimension of decomposition *dh*. The first decomposed dimension of *dh* is not involved in the distribution.

```
 1       program slice
 2 csms$declare_decomp(dh, 2)
 3 csms$distribute(dh, 1, 2) begin
 4       integer, allocatable :: u(:,:)
 5 csms$distribute end
 6 csms$distribute(dh, , 1) begin
 7       integer, allocatable :: ubw(:)
 8 csms$distribute end
 9       im = 4
10       jm = 4
11 csms$create_decomp(dh, <im,jm>, <0,0>)
12       allocate(u(im,jm))
13       allocate(ubw(jm))
14       ubw = 0
15 csms$serial begin
16       do j = 1, jm
17         do i = 1, im
18           u(i,j) = (100 * i) + j
19         enddo
20       enddo
21       print *, 'After initialization, u(:,:) = '
22       do j = 1, jm
23         write(*,'(4i5)') (u(i,j),i=1,im)
24       enddo
25 csms$serial end
26 csms$parallel(dh, , <j>) begin
27       do j = 1, jm
28 csms$global_index(1) begin
29         ubw(j) = u(1, j)
30 csms$global_index end
31       end do
32 csms$print_mode(ordered) begin
33       print *, 'on each process:  ubw = ', ubw
34 csms$print_mode end
35 csms$serial begin
36       print *, 'inside SERIAL, ubw = ', ubw
37 csms$serial end
38 csms$print_mode(ordered) begin
39       print *, 'after SERIAL, on each process:  ubw = ', ubw
40 csms$print_mode end
41 csms$parallel end
42       end
```

Example 13-1: This program illustrates limitations of sliced arrays.

When this program is run as a serial code, the following output is printed:

```
After initialization, u(:,:) =
101  201  301  401
102  202  302  402
103  203  303  403
104  204  304  404
on each process:  ubw =   101  102  103  104
inside SERIAL, ubw =   101  102  103  104
after SERIAL, on each process:  ubw =   101  102  103  104
```

When the SMS parallel code is built, ppp prints a warning message:

```
"./slice_sms.f.tmp", line 39:39 WARNING: Decomposed slice array should
be used carefully.  See the SMS Users Guide for more information.
```

When the SMS program is run on four processes, then following output is printed:

```
After initialization, u(:,:) =
101  201  301  401
102  202  302  402
103  203  303  403
104  204  304  404
on each process:  ubw =   101  102
on each process:  ubw =     0    0
on each process:  ubw =   103  104
on each process:  ubw =     0    0
inside SERIAL, ubw =     0    0    0    0
after SERIAL, on each process:  ubw =     0    0
after SERIAL, on each process:  ubw =     0    0
after SERIAL, on each process:  ubw =     0    0
after SERIAL, on each process:  ubw =     0    0
```

Notice that the values of *ubw* are correct before the SERIAL directive but incorrect afterwards. This is due to the fact that SMS cannot tell if *ubw* has not been correctly initialized on all processes. The GLOBAL_INDEX directive on line 28 causes *ubw* to be initialized only on processes that contain global index 1 in the first decomposed dimension. Had *ubw* been initialized outside of a GLOBAL_INDEX directive, the SERIAL directive would have worked as expected. As is the case with non-decomposed variables, SMS expects sliced arrays to be replicated across all processes. For sliced arrays, replication must occur across processes that share the same global indices in the remaining decomposed dimension. For example, *ubw* would be replicated if *ubw(j)* were the same on every process that contains global index *j* in its interior region. The SERIAL directive will not work for sliced arrays that are not replicated. The I/O and the REDUCE, TRANSFER, and COMPARE_VAR directives have the same limitation. (There are plans to extend the DISTRIBUTE directive to support non-replicated sliced arrays. These will be implemented in a future release of SMS if there is sufficient interest from users.)

14 Program Termination

Parallel programs using the SMS run-time system require special handling to ensure all processes exit normally. An SMS control process is often used to manage the child processes that are spawned through the *smsRun* command to execute a program. Two types of program termination are supported by SMS: a normal exit and an abort. When a program exits normally, the SMS control process will wait until every processes' computations, communications and I/O are complete before exiting. A program abort will not guarantee the completion of outstanding operations or an orderly termination of processes.

14.1 Automatic Code Generation for Termination

By default, SMS automatically generates code to abort whenever a Fortran "stop" statement is encountered. SMS also generates a normal exit whenever a program "end" statement is encountered. Consider the following program:

```
program main

do ii=0, num_iter
  call time_steps(ii,status)
  if (status .eq.  ABORT) then
    print *,’ Model Run failed at iteration: ‘,ii
    stop
  endif
enddo

print *,’ Model Run Successfully Completed’
stop
end
```

Example 14-1: Automatic Code Generation by SMS will cause this program to always abort.

Since the Fortran "stop" appears before the line before the end program statement, SMS will generate code to abort the parallel run. During code translation the following warning message will appear when source contains a Fortran stop statement:

```
WARNING: Program abort detected.
```

Since the intent of the original code in this case is to exit normally from the program, two actions can be taken to ensure this happens in the SMS-generated source. Either the second "stop" statement (above the "end") should be removed, or the EXIT directive should be used as illustrated in the next section.

14.2 EXIT Directive

EXIT is used to control the run-time behavior of an SMS program. This directive, when inserted just before a "stop" statement, will instruct SMS to generate code to exit rather than abort. The proper placement of this directive is illustrated in Example 14-2. In this example, SMS will generate an ABORT at line 7 and a normal exit at line 12.

```
      program main

      do ii=0, num_iter
        call time_steps(ii,status)
        if (status .eq. ABORT) then
          print *,’ Model Run failed at iteration: ‘,ii
          stop
        endif
      enddo

      print *,’ Model Run Successfully Completed’
CSMS$EXIT
      stop
      end
```

Example 14-2: Using `CSMS$EXIT` to override automatic translations

14.3 MESSAGE Directive

MESSAGE, is used to send a message to the user at run-time and optionally terminate execution of the program when it is encountered. This directive is useful when the user wishes to avoid unnecessary parallelization of code they believe is never executed. Three run-time actions are available to the user of MESSAGE: ABORT, terminates execution after writing the given message to stderr, WARN writes the given text to stderr, and INFORM writes the text to stdout.

```
      if (condition_ever_met) then
CSMS$MESSAGE(ABORT,'COMPS: THIS CODE HAS NOT BEEN PARALLELIZED BY SMS')
          call comps(a,b,c,d,NX,NY)
      endif
```

Example 14-3: Using MESSAGE to output run-time messages.

In this example, the programmer believes the subroutine *comps* is never executed so rather than parallelizing it, MESSAGE is used. Since ABORT is specified, SMS will terminate the execution of this program after the message is output to stderr.

15 Debugging

SMS provides two directives that aid the debugging process. COMPARE_VAR enables the user to compare the values of variables between simultaneous runs of an SMS program on different numbers of processes. This helps the user quickly pinpoint the source of an error that causes solutions to diverge. Sometimes, errors are due to missing or incorrectly placed exchanges. The CHECK_HALO directive helps identify these cases by flagging halo regions that are not updated as expected.

15.1 Using COMPARE_VAR To Find Parallelization Errors

Example 15-1 shows an application of the COMPARE_VAR directive. In this code, a one process run of the code yields:

```
 Running program check_var
 i, y(i)
           9   34.00000
          10   38.00000
          11   42.00000
          12   46.00000
```

However, a 2 process run yields a different answer:

```
 Running program check_var
 i, y(i)
           9   34.00000
          10   17.00000
          11   23.00000
          12   46.00000
```

```
     program check_var
     parameter (IM = 20)
CSMS$DECLARE_DECOMP(dh, 1)
CSMS$DISTRIBUTE(dh, 1) BEGIN
     real, allocatable :: x(:)
     real, allocatable :: y(:)
CSMS$DISTRIBUTE END

CSMS$CREATE_DECOMP(dh, <im>, <1>)

     print *, 'Running program check_var'
     allocate(x(im))
     allocate(y(im))
     x = 0.0

CSMS$PARALLEL(dh, <i>) BEGIN
     do i = 1, im
       x(i) = i*2 - 1
     end do

     y = 0.0

     do i = 2, im-1
       y(i) = x(i-1) + x(i+1)
     end do
```

```
26
27
28 CSMS$SERIAL BEGIN
29      print *, 'i, y(i)'
30      do i = 9, 12
31        print *, i, y(i)
32      end do
33 CSMS$SERIAL END
34
35 CSMS$PARALLEL END
36      end
```

Example 15-1: Application of the COMPARE_VAR directive.

To track down the problem, the user can insert a COMPARE_VAR directive at line 26 as follows:

```
CSMS$COMPARE_VAR(y(2:im-1), 'after y assignment')
```

Then the user can run the code as follows:

```
>> setenv SMS_SERVER_MODE serverfull
>> smsRun -np 1 check_var -np 2 check_var -cv
```

This tells SMS to simultaneously launch a 1-process and 2-process run of the program `check_var` and compare results ("*-cv*"). When the code generated by the COMPARE_VAR directive is reached, each run gathers *y* into a global equivalent, exchanges its values with the other run, and then compares them in the specified range (2:im-1). (If no range is specified, all elements of *y* are compared.) If the variable is a scalar or a non-decomposed array then it is immediately compared; no gather operation is required. In this example, since the variables differ, the program terminates with the following error message:

```
NP=1:  Running program check_var
NP=2:  Running program check_var
  COMPARE_VAR failed :y    after y assignment
 Variable values for first, second run:    38.00000       17.00000
Incorrect at indices =    10
```

The difference occurs because an exchange is needed prior to the loop starting at line 23. The character string, "after y assignment", helps pinpoint the location of the difference. The error message also identifies the name of the variable that is in error in case more than one variable is specified in the directive. It also prints out the global array location and variable values of the first point in the array that differs. Both runs immediately exit when an error is found. Notice that the print statement at line 11 appears twice. One instance is labeled with "NP=1" to indicate it came from the one-process run; the other is labeled with "NP=2" for the two-process run.

Several additional points should be made about COMPARE_VAR. First, if the user runs the program in the standard fashion:

```
smsRun -np 2 check_var
```

then the result is the same as if the COMPARE_VAR directive had not been included in the code at all. Second, when running with COMPARE_VAR enabled, any output to files is turned off to avoid generation of incorrect data that could occur when the two runs simultaneously write to the same file. Third, SMS must be run with a server process for the comparison to be made properly. Fourth, if the user wishes to use configuration files while doing a comparison run, the SMS launch line looks as follows:

```
smsRun –cf config1 check_var –cf config2 check_var -cv
```

Fifth, the user should avoid putting COMPARE_VAR directives inside a decomposed loop. For example, consider the following code fragment:

```
CSMS$PARALLEL(dh, <i>) BEGIN
      do i = 1, 2
        x(i) = x(i) + y(i)
CSMS$COMPARE_VAR(x, “Compare 1”)
      end do
```

Suppose simultaneous runs on 1 and two processes are compared. Each process in the 2 process run will call COMPARE_VAR once. However, the single process in the 1 process run will execute the COMPARE_VAR code twice. This will cause a hang or possibly an error message if a subsequent COMPARE_VAR directive appears in the code. Sixth, as a reminder, bitwise-exact reductions should be enabled when using COMPARE_VAR. Otherwise, round-off differences in summations will cause COMPARE_VAR to indicate the presence of a spurious error. Seventh, more than one variable can be compared in one directive. For example:

```
CSMS$COMPARE_VAR(x, y, “Compare 2”)
```

It is also possible to use COMPARE_VAR to verify that the parallel and serial code solutions are identical. To do this, translate and compile the code as before:

```
>> ppp prog.f
>> f90 prog_sms.f –o par_code
```

Next, translate the code with the --CompareOnly option and compile:

```
>> ppp --CompareOnly prog.f
>> f90 prog_sms.f –o serial_code
```

The –CompareOnly flag tells SMS to ignore all directives except for COMPARE_VAR. I/O statements are also left un-translated. Finally, the two executables can be simultaneously launched as follows:

```
>> smsRun –np 1 serial_code –np 2 par_code -cv
```

As before, when a difference is found, it will be flagged and the programs will terminate. WARNING: this use of COMPARE_VAR may not work on all systems!

15.2 Debugging Adjacent Dependencies: CHECK_HALO

The analysis of adjacent dependencies in a serial code and the process of accurately placing EXCHANGE and HALO_COMP directives are highly prone to error. To help the user track down such errors, the CHECK_HALO directive and associated SMS_CHECK_HALO environment variable can be used to check if all or part of a halo variable is up-to-date. Suppose, in Example 5-4, the user forgot to include the HALO_COMP directives on lines 43 and 48. When the program is run, it does not produce the correct answer for `ysum`. The user can observe that the loop on lines 52-54 requires one point of the lower and upper halo regions of `b` and `c` be up-to-date. To check this assumption, the following directive can be added at line 51:

```
CSMS$CHECK_HALO(b<1,1>, c<1,1>, 'LOOP 52')
```

If the SMS_CHECK_HALO environment variable is set to "ON", the generated code checks if the afore-mentioned halo points are up-to-date. In this case, since the halo regions are not up-to-date, the SMS program will generate the following error message and terminate:

```
LOOP 52 Halo check failed for var : b
```

Suppose the HALO_COMP directives are included as shown on lines 43 and 48. This time the check passes so no error messages are generated and the program continues. Suppose the user includes the HALO_COMP directives on lines 43 and 48 and specifies the CHECK_HALO directive as follows:

```
CSMS$CHECK_HALO(b, c, 'LOOP 52')
```

This form of the directive tells SMS to check the entire halo region. Since, for the lower and upper halo regions, only the inner layer of halo points is up-to-date, the program will terminate with the same error message.

The CHECK_HALO directive can be added to the code on a permanent basis. When SMS_CHECK_HALO is “ON”, CHECK_HALO adds costly communication. However, if the SMS_CHECK_HALO environment variable is set to something other than "ON" then the halo checks are skipped and the CHECK_HALO directives do not degrade performance. If, after a code change, the program generates the wrong answer, the halo checks can be turned back on to help isolate the problem. Also, note that except for the character string, the syntax for CHECK_HALO is identical to EXCHANGE.

16 Building a Parallel Program

16.1 Overview

This section describes how to use the PPP to translate Fortran code into SMS parallel source code. Output files, named automatically by PPP, will be introduced in Section 16.2. Several command line options to PPP are described in Section 16.3. In Section 16.4, a simple makefile is described which can be used to build a serial or SMS parallel code. In addition, various relevant compiler and linker options are discussed in this section. Building incorrect parallel source using PPP can result in both syntactic and semantic errors that must be corrected. Section 16.5 will discuss how to interpret these PPP generated messages. Finally, Section 16.6 will describe possible compiler errors due to namespace conflicts from PPP-generated source code.

16.2 PPP-Generated Output Files

Output files generated by PPP are named automatically. Include files will be named by appending ".SMS" to the original file name (e.g. *params.h* becomes *params.h.SMS*). All other source files will be named by appending "_sms" to the body of the original filename (e.g. *main.f* becomes *main_sms.f*). Intermediate files are also generated during the code translation process. These files, appended with the suffix ".tmp", remain after PPP translation. When errors are detected in the code during code parallelization, PPP messages will be generated that reference these intermediate files (see Example 16-7). However, any corrections should still go into the original file from which translated code was generated by PPP.

16.3 Building SMS Parallel Source Code

The transformation of Fortran code into parallel SMS code requires the use of PPP. PPP translations are based on both its analysis of the original code and the SMS directives that were inserted into the code. This section describes how to use PPP to create parallel code at the command line, defines what code generation options are available, and gives some examples.

16.3.1 PPP Command Line Options

All PPP code translations are managed through a command line script called **ppp**. A single file can be processed at a time and no inter-procedural analysis is done. PPP is invoked by: *ppp [options] filename*. Command line options currently available are:

```
--checkfirst            A useful optimization to avoid PPP processing of
                        files that do not require translation.  This
                        option can be used to allow more flexible use of
                        suffix rules (see Section 16.4).  If no I/O
                        statements or directives are found, no PPP
```

processing is done and the following message is output:
File has no directives - SKIPPING PPP PROCESSING

--comment Leave replaced lines in the code as Fortran comments. This can be useful for debugging the parallel code. Note: the string used to comment out the original code is **"C-PPP"**.

--CompareOnly Only translate COMPARE_VAR directives. Note that I/O statements will not be translated (see Section15.1).

--ExtendedSource Allow valid Fortran source to extend beyond 72 characters.

--Fcommon *file* Name of an optional include file that is not part of the original source code. Typically it will contain data decomposition directives (see Example 16-4).

--Finclude *file* Name of an included file to be parallelized that is referenced in the source file being translated by PPP (see Example 16-2).

--FixedFormat source code is Fortran fixed format.

--Flookup *file* Name of file containing mapping of module names to source code files containing them (see Example 16-5).

--Fmodule *file* Name of module that is not part of original source code. Typically, it will contain data decomposition directives (see Example 16-5).

--FreeFormat input file is Fortran free format.

--Fvisible *files* Name(s) of file(s) to be made visible to PPP in order to correctly translate the current file. This option is only required for a series of interdependent include files (see Example 16-3).

--header Indicates that the file that is about to be translated is a Fortran include file.

--help Prints the command line options

--IncludePath *path* Include file search path. Similar to -I F77/F90 compiler option

--r8 Indicates that an SMS program to be run on a machine whose normal default is 4 byte real numbers will be, instead, compiled so that the default is 8 byte real numbers.

--smsDebug Indicates that formatted output statements to stdout will use SMS output routines to provide process specific information.

--Verbose *level* Controls the output of PPP diagnostic and code analysis messages. Errors, Warnings and Notes are output based on the verbose level. (see Example 16-8).

16.3.2 Examples

Example 16-1 shows how to build a parallel version of an include file:

```
>> ppp --header  params.h

[params.h]

      parameter(nx=50, ny=50)

CSMS$DECLARE_DECOMP(decomp, <nx, ny>)

C global variable declarations ...
```

Example 16-1: Building any Fortran include file requires the --header option.

Example 16-2 shows how to use the parallel version of an include file when translating an executable code file. Since the translation of *params.h* will result in an SMS parallel version of this file (*params.h.SMS*), we use the --Finclude option to ensure this include file reference will be changed in the parallel version of *dynamics.f.*

```
>> ppp --Finclude=params.h --comment dynamics.f
 [dynamics.f]

      program dynamics
      include 'params.h'
c   Fortran code ...
      end

                              GENERATED PARALLEL PSEUDO CODE

[dynamics_sms.f]
      program dynamics
C-PPP      include 'params.h'
      include 'params.h.SMS'
c    Fortran code
      end
```

Example 16-2: The –Finclude option is used to specify the Fortran include file params.h which is referenced in the file (dynamics.f) being translated. This ensures the parallel (translated) include file will be referenced in the translated output of dynamics.f.

Example 16-3 illustrates the use of the --Fvisible option. In this example, the file "variables.h" requires information about the data decompositions listed in "params.h" to correctly translate the declarations "a" and "b" enclosed within the DISTRIBUTE directive. In particular, the array dimensions `nx, ny` and `nz` must be translated to process local sizes using information provided by DECLARE_DECOMP. The --Fvisible option is used is used to make params.h "visible" to variables.h.

```
>> ppp --header params.h
>> ppp --Fvisible=params.h --header variables.h
>> ppp --Finclude=params.h --Finclude=variables.h main.f

[params.h]

      parameter(nx=50, ny=50)
CSMS$DECLARE_DECOMP(decomp, <nx, ny>)

C global variable declarations ...

[variables.h]
CSMS$DISTRIBUTE(decomp, nx, ny) BEGIN
      real a(nx, ny, nz)
      real b(nx, ny, nz)
CSMS$DISTRIBUTE END


[main.f]
      program main
      include 'params.h'
      include 'variables.h'
c other code
      end
```

Example 16-3: The --Fvisible option is used when inter-dependent include files must be translated.

In Example 16-1, the DECLARE_DECOMP was added to an include file that already existed (*params.h*). If the user prefers to insert the SMS directives into a separate "directives" file, the option --Fcommon is used instead of --Finclude. Example 16-4 illustrates the --Fcommon option.

```
>> ppp --header directives.inc
>> ppp --Fcommon=directives.inc dynamics.f


[directives.inc]
CSMS$DECLARE_DECOMP(decomp, 2)

[dynamics.f]
      program main
c more Fortran code
      end


                     GENERATED PARALLEL PSEUDO CODE
      program main
      include 'directives.inc.SMS'
      include 'params.h'
c more Fortran code
      end
```

Example 16-4: In this example DECLARE_DECOMP, defined in "directives.inc", is required by "dynamics.f" when the parallel executable is built. It is not needed for a serial build.

Example 16-5 shows how to handle Fortran 90 modules in SMS. The DECLARE_DECOMP directive is inserted into a separate "directives" module in a way analogous to the include file in Example 16-4. In addition, the main program uses a module needed by the serial code.

```
[decomp.F]
      module decomp
CSMS$DECLARE_DECOMP(dh, 2)
      end module decomp


[m1_module.F]
      module m1
CSMS$DISTRIBUTE(dh, 1, 2) BEGIN
      real, allocatable :: u(:,:,:)
CSMS$DISTRIBUTE END
      end module m1


[main.F]
      program USE_MODULES
      use m1
      integer, parameter :: im = 10
      integer, parameter :: jm = 20
      integer, parameter :: km = 30

CSMS$CREATE_DECOMP(dh, <im,jm>, <1,1>)

      allocate(u(im,jm,km))
CSMS$PRINT_MODE(ORDERED) BEGIN
```

```
      print *, size(u,1), size(u,2), size(u,3)
CSMS$PRINT_MODE END
      end
```

```
[my_lookup_file]
decomp                  decomp.F
m1                      m1_module_cpp.f
```

Example 16-5: Code and module look-up file illustrating how SMS handles Fortran 90 modules. See text for explanation.

To handle this code correctly, the decomposition module file must be translated first:

```
>> ppp decomp.f
```

Notice that the "--header" option is NOT used in this case in contrast to the case where an include file is translated.

Next, the module file is translated. Since the module file contains reference to the decomposition, "dh", the decomposition module must be made visible to SMS during translation. This is done with the "–Fmodule=decomp" option. This option tells SMS to insert "use decomp" in the translated version of the module file. SMS must also know the mapping of modules to the files that contain them. In this case, module decomp is contained in file *decomp.F*. The lookup table specifies that mapping. The file containing this look-up table must also be specified when translating *m1_module.F*. The full command is :

```
>> ppp –Fmodule=decomp  --Flookup=my_lookup_file m1_module.F
```

Finally, the command line for translating *main.F* is:

```
>> ppp –Fmodule=decomp  --Flookup=my_lookup_file  main.F
```

16.4 Building SMS Programs

A simple makefile is presented to aid the user in building an SMS program. The environment variable "SMS" must be set to the location where the SMS software has been installed. This can either be done explicitly in the Makefile (line 5) or via *setenv* prior to running make (e.g. *setenv SMS pathname*).

```
 1    # standard make file used to build serial or SMS parallel
 2    # executables
 3    #
 4    .SUFFIXES:  .f .o
 5    SMS = /usr/local/sms
 6
 7    # system-specific compilation flags (for a Compaq Alpha EV67)
 8    FC = mpif90
 9    FFLAGS = -O4 –arch host –tune host -fixed -I$(SMS)/include
10
11    # SMS link libraries
12    LIBS = -L$(SMS)/lib -lsms -lmpi
13
```

```
14    # PPP specific options set here
15    PPP = $(SMS)/bin/ppp
16    PPP_FLAGS = --Finclude=params.h --Finclude=variables.h --comment \
17    --checkfirst
18    PPP_HEADER_FLAGS = --header --comment
19
20    # include files
21    INCLUDES = params.h variables.h globals.h
22    PINCLUDES = ${INCLUDES:.h=.h.SMS}
23
24    # object files
25    OBJS = file1.o file2.o file3.o
26
27    TARGET = par_prog
28
29    # executable
30    $(TARGET):  $(PINCLUDES) $(OBJS)
31          $(FC) -o $(TARGET) $(OBJS) $(FFLAGS) $(LIBS)
32
33    # suffix rules
34    .f.o: $(PINCLUDES)
35          $(PPP) $(PPP_FLAGS) $*.f
36          $(FC) -c $(FFLAGS) $*_sms.f
37          /bin/mv –f $*_sms.o $*.o
38
39    # include file translations
40    params.h.SMS:     params.h
41          $(PPP) $(PPP_HEADER_FLAGS) params.h
42
43    variables.h.SMS:  variables.h params.h
44          $(PPP) $(PPP_HEADER_FLAGS) --Fvisible=params.h variables.h
45
46    clean:
47          /bin/rm –f $(TARGET) *_sms.f *.SMS *.o *.tmp
```

Example 16-6: A makefile for an SMS program.

16.4.1 Makefile Compiler and Linker Options

The Fortran compiler flags (FLAGS on line 9) are set for a Compaq Alpha EV67. Other systems will require different options. A makefile provided in the SMS distribution ($SMS/lib/makefile.header) gives recommended compilation flags (found in variable STD_OPT_FLAGS) that should be used when modifying FFLAGS for a different target machine.

16.4.2 Include File Handling

Include files are listed in the makefile variable INCLUDES. Parallel include files (line 22) translated using SMS are built using the explicit targets *params.h.SMS* and *variables.h.SMS* (lines 40-44*)*. Notice the PPP command to build *variables.h.SMS* (line 44) contains the --Fvisible option in addition to the standard PPP flags defined by: PPP_HEADER_FLAGS at line 18. Since *variables.h.SMS* requires information from *params.h* for proper translation, this option is required (see Example 16-3).

PPP_FLAGS (lines 16-17) lists the include files that are translated by PPP via the --Finclude option. This option is required to ensure any references to these files in Fortran source will be modified to their parallel filename (see Example 16-2).

16.4.3 Building the Executable

To build the SMS parallel executable “par_prog” using this makefile, simply run make:

```
>> make
```

Translated source code is written to the file “`file1_sms.f`”.

16.5 PPP Error Reporting

Two types of errors are reported by PPP: parsing errors and semantic errors. Parsing errors must be corrected before further translations of the input file are permitted. Semantic errors are reported as errors, warnings or notes. These messages can be controlled through the --verbose option of PPP (see Section 16.5.2).

16.5.1 Parsing Errors

Parsing errors occur when PPP cannot resolve the Fortran source code to the grammar defined by the SMS directives (refer to the SMS Reference Manual), the Fortran 77 language, and currently supported Fortran 90 syntax. Further details about language extensions supported by SMS can be found at:

http://www-ad.fsl.noaa.gov/ac/Fortran90_Language_Support.html

The parser currently supports statements or SMS directives that are up to 500 characters in length. Multiple statement lines are collapsed and white space is removed before statements are parsed. Statements longer than 500 characters will not be parsed correctly.

The form of a parsing error message is:

```
<filename> <line> <column> <error type> <message>

filename     - name of file being parsed
line         - line number
column       - column number in which error occurred
error type   - types are:
                 ERROR, WARNING, NOTE
message      - diagnostic message
```

An example of a PPP-generated parsing error is shown in Example 16-7.

```
 1 CSMS$DECLARE_DECOMP(spec_dh,<jtrun>)
 2 CSMS$DISTRIBUTE(spec_dh, jtrun) BEGIN
 3       real*8 cc(jtrun), bb(jtrun)
 4 CSMS$DISTRIBUTE END
 5
 6 CSMS$PARALLEL(spec_dh, m) BEGIN
 7       do 3 m=2, jtrun, 2
 8         cc(m) = cc(m) + bb(m)
 9     3 continue
10
11 C CSMS$PARALLEL END is missing
12
13       end
```

Example 16-7: Code that generates a PPP parsing error.

PPP generates the following error message:

```
"Loops_sms.f.tmp" 13 501 ERROR:  Syntax error
"Loops_sms.f.tmp" 13 501 NOTE  Parsing resumed here
```

This message indicates the parser failed in the file *Loops_sms.f.tmp* at line 13 column 501. A parsing error occurring at column 501 indicates no resolution of the statement to the grammar by the end of the line. In the example, the parser expects a PARALLEL END directive before the end of the file. Naturally, the error should be corrected in the original file (*Loops.f*) rather than the PPP-generated file.

16.5.2 PPP Diagnostic Messages

Three levels of diagnostic messages are reported by PPP. A PPP ERROR is reported when a section of code targeted for translation contains a syntax or semantic error. A PPP WARNING is reported when PPP suspects that it may generate an incorrect translation. A PPP NOTE identifies a place where a particular type of transformation occurred or SMS limitation was detected. By default, all PPP ERROR messages will be output. Control of diagnostic messages is handled through the PPP command line option: "*--verbose = <value>*". Three verbose options are supported:

```
value       message domain

   1        PPP ERRORS only (DEFAULT)
   2        PPP ERRORS and WARNINGS only
   3        PPP ERRORS, WARNINGS and NOTES
```

While the error messages should always be addressed, warning messages may also be useful for detecting potential problems. For example, the code segment in Example 16-8 below causes PPP to generate the following important warning message:

```
./IO.f.tmp” 11 13 WARNING: This variable, decomposed by CSMS$DISTRIBUTE,
is being used outside of a parallel region.
```

This warning message indicates a problem on line 11, column 13 of the PPP-generated file *IO.f.tmp* (which is not shown). The variable *cc* was defined to be a distributed array (using DISTRIBUTE) but is being referenced outside a parallel region (PARALLEL). Further explanation on the use of these directives can be found in Section 2.3.

```
>> ppp --verbose=2 IO.p
```

```
CSMS$DISTRIBUTE(dh, m, n) BEGIN
      real cc(m,n)
CSMS$DISTRIBUTE END

      do i = 1, m
        do j = 1, n
          cc(i,j) = 0.0
        enddo
      enddo

c   more code ...
```

Example 16-8: Code that generates a WARNING because the decomposed variable "cc" is being used outside of a parallel region.

16.6 Compilation Errors

During the translation process PPP generates new variables for some translations. PPP variables are either automatically generated or defined explicitly by PPP. Explicitly defined names will always contain a double underscore in their name (e.g. ppp__status). To avoid compiler errors due to name space conflicts, avoid using variable names with double underscores in them. For example, the serial code cannot contain a declaration of a variable named *ppp__status* because PPP translation explicitly declares a variable named *ppp__status* for its own use. A compilation error would result because two variables would be declared with the same name.

17 Running an SMS Program

17.1 Introduction

Once a program has been translated into SMS parallel code (Section 16.3) and linked to the appropriate libraries (see Section 16.4), it can be run on one or more processes using the SMS program launcher *smsRun*. The standard syntax for *smsRun* is:

```
>> smsRun -np numprocs execname
```

By default, SMS uses an additional server process to perform I/O operations, and provide overall management and control services for the other processes (see Figure 12-4). For example, to run the executable *test* with two processes and one server process, the user would enter:

```
>> smsRun -np 2 test
```

It is possible to take advantage of the idle compute cycles available on the server process by setting SMS environment variable *SMS_SERVER_MODE* to *serverless*. This will permit computational and management functions to co-exist in a single process. This option is beneficial when only a small number of processes are available. However, as the numbers of processes grow, the cost of performing both server functions and computations will limit the performance of the other dependent processes.

Figure 12-4 assumes a single process is run on each processor. However, SMS permits the user to request more processes (using *smsRun*) than available processors. For example if *my_program* was run with 20 processes

```
>> smsRun -np 20 my_program
```

on a system with only 16 processors, five processors would contain two processes, one would contain the server process, and the rest would each contain a single process designated to run the program. This is a bad idea because performance will suffer whenever multiple processes are scheduled on a single processor on most machines.

17.2 Optional Command Line Arguments

Several optional arguments to *smsRun* are supported. The *-cf* option is used to gain more control over process layout as described in Section 10.1. In the example below, program *my_program* is run using the process layout specified in configuration file *my_config*:

```
>> smsRun -cf my_config my_program
```

The *-cv* option is used to assist debugging by turning on COMPARE_VAR directives as described in Section 15.1. In the first example below, program *my_program* is run simultaneously on 1 and 9 processes. Arrays specified in COMPARE_VAR directives are compared on-the-fly. If a difference is found, execution is halted and an error message is printed. In the second example, different executables are loaded (useful for comparing a serial run to a parallel run or for testing static memory codes that use minimum memory) and the second program uses a configuration file.

```
>> smsRun -np 1 my_program -np 9 my_program -cv
>> smsRun -np 1 my_program1 -cf my_config9 my_program9 -cv
```

Another option, *-sms-*, allows the user to specify machine-specific arguments to the underlying communication layer (e.g. MPI, SHMEM) directly. All arguments that follow this option will be ignored by SMS and passed directly to the communications software. The following command illustrates a way to pass the run-time option *-mpi_special* to the underlying program launcher (which is *mpirun* when SMS is built using MPI):

```
>> smsRun -np 3 my_program -sms- -mpi_special
```

17.3 Run-time Environment Variables

Several environment variables can also be set to control the run-time behavior of SMS. The following environment variables are available:

```
SMS_BITWISE       - Set to "EXACT" to use bit-wise exact reductions
                    (see Section 7.2)
SMS_CHECK_HALO    - Set to "ON" to enable checks of halo regions
                    specified by CHECK_HALO directives (see
                    Section 15.2).
SMS_CLOSE_MODE    - Use to improve output performance (see
                    Section 12.3.3).
SMS_IO_FORMAT     - Use to specify file format for files that are read
                    or written by SMS.
SMS_READ_FORMAT   - Use to specify file format for files that are read
                    by SMS programs (see Section 12.1).
SMS_SERVER_MODE   - Set to "SERVERLESS" to avoid using a server process
                    (see Section 12.3.2).
SMS_WRITE_FORMAT  - Use to specify file format for files that are
                    written by SMS programs (see Section 12.1).
SMS_XFERMODE      - Use to choose optimal communication patterns for
                    TRANSFER.  Options are: "LOGN" and "ORIGINAL".
```

17.4 Run-time Error Messages

When an error occurs in an SMS program, execution will usually terminate and SMS will generate an informational message describing the source file name, line number, and a brief summary of the problem. A complete set of SMS run-time error messages is available at the following SMS web site:

http://www-ad.fsl.noaa.gov/ac/SMS_Messages.html

When the code in Example 3-2 is run with 2 processes, the following error message is generated (as seen in Section 3.3):

```
 Process:   1 Error at: ./decomp_ex4_sms.f:10.1
 Process:   1 Error status=   -2202 : USER DECLARED STATIC ARRAY IS TOO
SMALL.
 Process:   1 Aborting...
```

The first line of the error message indicates the file name and location within the file where the problem occurred. PPP-generated code frequently uses sub-numbering to handle multiply generated calls to SMS routines that stem from the same line of original code. In this example, a run-time error was detected by SMS at line 17 in code generated by the directive CREATE_DECOMP that can be found in temporary file: *decomp_ex4_sms.f.tmp* (not shown).

The second line gives the SMS error message. The error messages reflect the incorrect sizing of the decomposition ***decomp***, declared by DECLARE_DECOMP and initialized by CREATE_DECOMP.

Once the problem is understood corrections to the code can be made. These corrections should go into the original file (in this case *decomp_ex4.f*) not in the temporary file where the problem was detected and probably diagnosed. Once changes are made, PPP can be executed to re-translate the input file from which a fresh executable can be built and tested.