Sun™ S3L 4.0 Programming Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 U.S.A.
650-960-1300

Part No. 816-0652-10
August 2001, Revision A

Send comments about this document to: docfeedback@sun.com

Contents

Preface

This manual describes the Sun™ Scalable Scientific Subroutine Library (Sun S3L). It is directed to anyone developing message-passing C, C++, F77, or F90 programs.

Acknowledgments

The Sun S3L dense linear algebra routines make use of the ScaLAPACK library described in "ScaLAPACK: Linear Algebra Software for Distributed Memory Architectures," J. Demmel, J. Dongarra, R. van de Geijn, and D. Walker in *Parallel Computers: Theory and Practice*, Ed. by T. Casavant, P. Tvrdik, and F. Plasil. (IEEE Press, 1995, pp. 267-282.)

ScaLAPACK routines access the Sun MPI library through calls to the BLACS library described in "Two-dimensional Basic Linear Algebra Communications Subprograms," J. Dongarra and R. van de Geijn, in E*nvironments and Tools for Parallel Scientific Computing*, Ed. by J. Dongarra and B. Tourancheau (Elsevier Science Publisher B.V., 1993, pp. 31-40), in "Basic Linear Algebra Communication Subprograms: Analysis and Implementation Across Multiple Parallel Architectures," R.C. Whaley.

How This Book Is Organized

Chapter 1 provides an overview of the Sun S3L contents.

Chapter 2 identifies the data types that are supported by Sun S3L.

Chapter 3 discusses parallel arrays—their distribution and indexing in a Sun S3L context.

Chapter 4 discusses Sun S3L support for the multiple-instance paradigm.

Chapter 5 explains how to use Sun S3L routines in a message-passing program.

Chapter 6 describes how to set up the Sun S3L environment for use by a message-passing program.

Chapter 7 describes the Sun S3L routines that simplify the management of dense parallel arrays.

Chapter 8 describes the Sun S3L routines that create and free custom process grids.

Chapter 9 describes the Sun S3L routines that extract information about parallel arrays and process grids and that access the contents of the arrays.

Chapter 10 describes the Sun S3L routines that perform dense matrix operations.

Chapter 11 describes the Sun S3L routines that provide solutions to linear systems equations for real and complex general matrices.

Chapter 12 describes the Sun S3L routines that perform fundamental linear algebra operations on sparse matrices.

Chapter 13 describes the Sun S3L routines that provide solutions to sparse linear systems of the type A*x = B.

Chapter 14 describes the Sun S3L routines that perform FFT computations and the supporting routines that set up and deallocate the internal data structures used by the FFT routines.

Chapter 15 describes the Sun S3L routines that perform random number generation.

Chapter 16 provides a summary listing of Sun S3L routines that are not discussed in detail in this manual.

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures.

See either of the following for such information:

- AnswerBook2™ online documentation for the Solaris™ 2.x operating environment
- Other software documentation that you received with your system

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail`.
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

Shell	Prompt
C shell	%
C shell superuser	#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

Application	Title	Part Number
All	*Sun HPC ClusterTools™ 4 Product Notes*	816-0647-10
All	*Sun HPC ClusterTools 4 Performance Guide*	816-0656-10
Sun S3L Programming	*Sun S3L 4.0 Reference Manual*	816-0653-10
Sun MPI Programming	*Sun MPI 5.0 Programming and Reference Guide*	816-0651-10
Sun MPI Programming	*Sun HPC ClusterTools 4 User's Guide*	816-0650-10
Prism	*Prism 6.2 User's Guide*	816-0654-10
Prism	*Prism 6.2 Reference Manual*	816-0655-10

Accessing Sun Documentation Online

The `docs.sun.com`SM web site enables you to access a select group of Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

`http://docs.sun.com`

Ordering Sun Documentation

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at:

`http://www.fatbrain.com/documentation/sun`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number (816-0652-10) of your document in the subject line of your email.

CHAPTER 1

Introduction to Sun S3L

This chapter contains general information about the Sun Scalable Scientific Subroutine Library (Sun S3L).

Sun S3L Overview

Sun S3L provides a set of parallel and scalable functions and tools widely used in scientific and engineering computing. It can be used on all Sun HPC systems—from a single processor on an SMP, to multiple processors on a standalone SMP, to a cluster of SMPs.

The chief advantages offered by Sun S3L are summarized below:

- Sun S3L is optimized for Sun HPC systems.
- Sun S3L functions have a simple array syntax interface that is callable from message-passing programs written in C, C++, F77, or F90.
- Sun S3L supports multiple instances.
- Sun S3L is thread safe.
- Sun S3L uses the Sun Performance Library™ for nodal computation.
- Extensive and detailed programming examples are provided online.
- Sun S3L is supported by Sun.
- Sun S3L includes built-in diagnostics.

Sun S3L uses *array handles* to provide array syntax support to message-passing programs. Array handles, which are closely analogous to the array descriptors found in the public domain packages ScaLAPACK and PETSc, facilitate argument passing by encapsulating information about distributed arrays.

Sun S3L operates on multidimensional arrays of up to 31 dimensions. This means it implements the multiple-instance paradigm, where the same function is applied to multiple, disjoint data sets concurrently.

The Sun S3L user interface includes a communicator setup routine that enables Sun S3L functions to be used in multithreaded applications. This routine causes Sun S3L to establish an independent Sun MPI communicator and thread-safe data for each thread from which the routine is called.

Sun S3L routines implement the Sun Performance Library for nodal operations. This is a collection of libraries for dense linear algebra and Fourier transforms based on the standard libraries BLAS, LINPACK, LAPACK, FFTPACK, and VFFTPACK. Besides providing appropriate nodal support to Sun S3L, routines from the Sun Performance Library can be called independently from any user codes running locally on a Sun Ultra HPC Server node.

Note – The Sun Performance Library is available to Sun S3L users as part of the Forte™ Developer 6 products.

Sun S3L routines operate on objects of various data types. However, this information is encoded in the array handle and is decoded at runtime, enabling appropriate branching to occur during execution. Consequently, there is no need for separate routines with different names to implement the different data types. A single routine suffices for all types.

An extensive set of online examples illustrates correct use of all Sun S3L functions. These examples can be used as templates in developing actual code. Separate examples are provided to demonstrate C and Fortran interfaces.

Contents of Sun S3L

Sun S3L consists of a set of *core* library functions—that is, the routines that perform the linear algebra, Fourier transform, and other computational functions usually found in a mathematical routine library—plus a set of auxiliary utilities, referred to as the *toolkit* functions.

TABLE 1-1 and TABLE 1-2 list the Sun S3L computational and toolkit routines, respectively.

Note – Many Sun S3L computational routines support the corresponding ScaLAPACK application programming interfaces (APIs). See TABLE 1-3 for a list of these supported APIs.

Most of the computational and toolkit routines are discussed in later chapters of this programming guide. Detailed descriptions of all the Sun S3L routines are provided in the *Sun S3L Reference Manual*. They are also described in their online man pages.

TABLE 1-1 Sun S3L Core Mathematical Routines

Function	Description
Dense Matrix Operations	
`S3L_2_norm()`	Compute 2-norm of a vector.
`S3L_inner_prod()`	Compute inner product of two vectors.
`S3L_mat_mult()`	Compute product of two matrices.
`S3L_mat_vec_mult()`	Compute product of a matrix and vector.
`S3L_outer_prod()`	Compute outer product of two matrices.
Sparse Matrix Operations	
`S3L_declare_sparse()`	Create an S3L handle for an S3L sparse array.
`S3L_free_sparse()`	Free memory allocated to S3L sparse array.
`S3L_convert_sparse()`	Convert an array from one sparse format to another
`S3L_rand_sparse()`	Create an S3L array with random values and sparsity.
`S3L_matvec_sparse()`	Compute product of a sparse matrix and dense vector.
`S3L_read_sparse()`	Read sparse matrix from a file.
`S3L_write_sparse()`	Write sparse matrix to a file.
`S3L_print_sparse()`	Print all nonzero values from a sparse matrix.
Gaussian Elimination for Dense Systems	
`S3L_lu_factor()`	Perform LU factorization of a matrix.
`S3L_lu_invert()`	Compute inverse of square matrix instances of S3L array using `S3l_lu_factor()` results.
`S3L_lu_solve()`	Solve system of linear equations (AX=B) for square matrix instances of S3L array.
`S3L_lu_deallocate()`	Deallocate `S3L_lu_factor()` resources.
Walsh Transform	
`S3L_walsh()`	Compute discrete Walsh/Hadamard transform of 1D and 2D S3L arrays.
`S3L_walsh_setup()`	Prepare internal data structure for discrete Walsh/Hadamard transform.
`S3L_walsh_free_setup()`	Free memory allocated to Walsh/`Hadamard` transform.
Iterative Eigenpairs Computation	
`S3L_eigen_iter()`	Compute selected eigenpairs of dense or sparse matrices.

TABLE 1-1 Sun S3L Core Mathematical Routines *(Continued)*

Function	Description
Finite-Difference Stock Option Pricing	
`S3L_fin_fd_1D()`	Solve 1D Black-Scholes PDE to compute prices of vanilla and several exotic stock options.
`S3L_fin_fd_2D()`	Solve 2D Black-Scholes PDE to compute prices of vanilla and several exotic stock options.
Discrete Cosine Transform	
`S3L_dct_iv()`	Compute DCT Type IV of 1D, 2D, and 3D S3L arrays.
`S3L_dct_iv_setup()`	Prepare internal data structures for DCT Type IV operation.
`S3L_dct_iv_free_setup()`	Free memory allocated to DCT.
Discrete Sine Transform	
`S3L_dst()`	Compute DST of 1D, 2D, and 3D S3L arrays.
`S3L_dst_setup()`	Prepare internal data structures for DST.
`S3L_dst_free_setup()`	Free memory allocated to DST.
QR Array Factoring/Solving	
`S3L_qr_factor()`	Compute QR decomposition of a real or complex S3L array.
`S3L_get_qr()`	Extract Q and R arrays from a QR-decomposed S3L array.
`S3L_qr_solve()`	Compute the least-squares solution to an over-determined system of the form a*x=b.
`S3L_qr_free()`	Free memory allocated to QR decomposition.
Quadratic Programming Optimization	
`S3L_qp_attr_init()`	Initialize a set of QP attributes with default values.
`S3L_qp_attr_destroy()`	Destroy a specified set of QP attributes.
`S3L_qp_attr_set()`	Specify the type of solver to be used and amount of error output.
`S3L_qp()`	Solve linear/quadratic optimization problem.
Cholesky Solver	
`S3L_cholesky_factor()`	Perform Cholesky factorization for each square matrix in an S3L array.
`S3L_cholesky_solve()`	Solve a system of distributed linear equations of the form AX = B for each square matrix in an S3L array.
`S3L_cholesky_invert()`	Compute the inverse of each square matrix in an S3L array.

TABLE 1-1 Sun S3L Core Mathematical Routines *(Continued)*

Function	Description
Sparse Linear System Solvers	
Direct Method	
`S3L_sparse_solve()`	A direct solver for solving sparse linear systems of equations of the form A*x = y.
`S3L_sparse_solve_free()`	Free memory allocated to the direct solver.
Iterative Method	
`S3L_gen_iter_solve()`	An iterative solver for solving sparse linear systems of equations of the form A*x = y.
Sparse Linear Problem Solver	
`S3L_lp_sparse()`	Solve a linear/quadric optimization problem of the form `min c'*x`.
Fast Fourier Transforms	
`S3L_fft()`	Perform simple FFT on an S3L array.
`S3L_fft_detailed()`	Perform in-place forward or reverse FFT along a specified axis of an S3L array.
`S3L_ifft()`	Perform the inverse FFT on an S3L axis.
`S3L_rc_fft()`	Perform forward FFT of a real S3L array.
`S3L_cr_fft()`	Perform inverse FFT of a complex S3L array.
`S3L_fft_setup()`	Prepare internal structure for FFT operation.
`S3L_rc_fft_setup()`	Prepare internal data structure for real-to-complex (forward) FFT.
`S3L_cr_fft_setup()`	Prepare internal data structure for complex-to-real (inverse) FFT.
`S3L_fft_free_setup()`	Free memory allocated to FFT setup.
`S3L_rc_fft_free_setup()`	Free memory allocated to real-to-complex or complex-to-real FFT setup.
Structured Solvers	
`S3L_gen_band_factor()`	Perform LU factorization of an n x n general banded S3L array.
`S3L_gen_band_solve()`	Solve a banded system.
`S3L_gen_band_free_factors()`	Free resources allocated to factorization of general banded S3L array.
`S3L_gen_trid_factor()`	Compute factorization of a tridiagonal matrix.
`S3L_gen_trid_solve()`	Solve a tridiagonal system.
`S3L_gen_trid_free_factors()`	Free resources allocated to factorization of a tridiagonal matrix.
Dense Symmetric Eigenvalue Solver	
`S3L_sym_eigen()`	Find eigenvalues and, optionally, eigenvectors in Hermitian matrices.

TABLE 1-1 Sun S3L Core Mathematical Routines *(Continued)*

Function	Description
Condition Numbers	
`S3L_condition_number()`	Compute the condition numbers of one or more instances of a square S3L array.
Parallel Random Number Generators	
`S3L_setup_rand_fib()`	Initialize state table for the lagged Fibonacci random number generator (LFG).
`S3L_rand_fib()`	Initialize an S3L array with an LFG.
`S3L_rand_lcg()`	Initialize an S3L array with a Linear Congruential random number generator.
`S3L_free_rand_fib()`	Free memory allocated to the random number generator state table.
Least Squares Solver	
`S3L_gen_lsq()`	Find the least squares solution to an overdetermined system or the minimum norm solution to an underdetermined system.
Dense Singular Value Decomposition	
`S3L_gen_svd()`	Compute the singular value of an S3L array and, optionally, the right and/or left singular vectors.
Autocorrelation	
`S3L_acorr_setup()`	Set up conditions for computing the autocorrelation of a signal.
`S3L_acorr()`	Compute 1D or 2D autocorrelation of a signal.
`S3L_acorr_free_setup()`	Free memory allocated to a particular autocorrelation setup.
Convolution	
`S3L_conv_setup()`	Set up conditions for computing the convolution of a signal.
`S3L_conv()`	Compute 1D or 2D convolution of a signal.
`S3L_conv_free_setup()`	Free memory allocated to a particular convolution setup.

TABLE 1-1 Sun S3L Core Mathematical Routines *(Continued)*

Function	Description
Deconvolution	
`S3L_deconv_setup()`	Set up conditions for computing the deconvolution of an S3L array.
`S3L_deconv()`	Compute 1D or 2D deconvolution of an S3L array.
`S3L_deconv_free_setup()`	Free memory allocated to a particular deconvolution setup.
Grade Elements of an Array	
`S3L_grade_up()`	Grade all elements of an S3L array in ascending order.
`S3L_grade_down()`	Grade all elements of an S3L array in descending order.
`S3L_grade_detailed_up()`	Grade elements along one axis of an S3L array in ascending order.
`S3L_grade_detailed_down()`	Grade elements along one axis of an S3L array in descending order.
Sort Elements of an Array	
`S3L_sort()`	Sort all elements of a one-dimensional array in ascending order.
`S3L_sort_up()`	Sort all elements of a one-dimensional or multidimensional array in ascending order.
`S3L_sort_down()`	Sort all elements of a one-dimensional or multidimensional array in descending order.
`S3L_sort_detailed()`	Sort elements along one axis of an S3L array in either ascending or descending order using quicksort or radixsort algorithm.
`S3L_sort_detailed_up()`	Sort elements along one axis of an S3L array in ascending order.
`S3L_sort_detailed_down()`	Sort elements along one axis of an S3L array in descending order.
Parallel Transpose	
`S3L_trans()`	Perform generalized transposition of an S3L array.

TABLE 1-2 Sun S3L Toolkit Routines

Function	Description
Create/Exit Sun S3L Environment	
`S3L_init()`	Set up Sun S3L environment.
`S3L_exit()`	Leave Sun S3L environment.
Create S3L Array Handles	
`S3L_declare()`	Create S3L array handle (basic method).
`S3L_declare_detailed()`	Create S3L array handle (with control over more parameters).
`S3L_DefineArray()`	Declare S3L array (not recommended; for back-compatibility with Sun S3L 2.0 only).
Release S3L Array Handles	
`S3L_free()`	Release an S3L array (recommended).
`S3L_UndefineArray()`	Release an S3L array (for Sun S3L 2.0 only).
Control S3L Process Grids	
`S3L_set_process_grid()`	Define an S3L process grid.
`S3L_free_process_grid()`	Release resources allocated to a process grid.
Perform Operations on S3L Arrays	
`S3L_array_op1()`	Perform operation on array (one operand).
`S3L_array_op2()`	Perform operation on array (two operands).
`S3L_array_scalar_op2()`	Perform operation on array and scalar value.
`S3L_cshift()`	Perform circular shift along a specified axis.
`S3L_forall()`	Apply a user-defined function to some or all elements in an array.
`S3L_reduce()`	Perform a reduction function across an array.
`S3L_reduce_axis()`	Perform a reduction function along one axis of an array
`S3L_set_array_element()`	Set the value of an element of an S3L array.
`S3L_set_array_element_on_proc()`	Set the value of an element of an S3L array, using the value supplied on a specific process.
`S3L_get_array_element()`	Retrieve the value of an element of an S3L array.
`S3L_get_array_element_on_proc()`	Retrieve the value of an element of an S3L array, as supplied by a specified process.
`S3L_zero_elements()`	Set all elements in an S3L array to zero.

TABLE 1-2 Sun S3L Toolkit Routines *(Continued)*

Function	Description
Get Information About S3L Arrays	
`S3L_describe()`	Get information about an S3L array or process grid.
`S3L_get_attribute()`	Get the value of an S3L array attribute.
`S3L_read_array()`	Read an S3L array from a file.
`S3L_read_sub_array()`	Read part of an S3L array from a file.
`S3L_print_array()`	Print an S3L array to standard output.
`S3L_print_sub_array()`	Print part of an S3L array to standard output.
`S3L_write_array()`	Write an S3L array to a specified file.
`S3L_write_sub_array()`	Write part of an S3L array to a specified file.
Miscellaneous Tools	
`S3L_copy_array()`	Copy an S3L array into another S3L array.
`S3L_from_ScaLAPACK()`	Convert ScaLAPACK descriptor to S3L handle.
`S3L_to_ScaLAPACK()`	Convert S3L handle to ScaLAPACK descriptor.
`S3L_thread_comm_setup()`	Prepare S3L environment for thread-safe operation.
`S3L_set_safety()`	Set error-checking level for S3L operations.
`S3L_get_safety()`	Get S3L error-checking level.

TABLE 1-3 Supported ScaLAPACK APIs

Category	Routine
PBLAS 1,2,3	`p{s,d}dot`, `p{c,z}dotu`, `p{s,d}nrm2`, `p{sc,dz}nrm2`, `p{s,d}ger`, `p{c,z}geru`, `p{s,d,c,z}gemv`, `p{s,d,c,z}gemm`
LU factor, solve, inverse	`p{s,d,c,z}getrf`,`p{c,d,c,z}getrs`,`p{c,d,c,z}getri`
Tridiagonal solvers	`p{s,d,c,z}dttrf, p{s,d,c,z}dttrs`
Banded solvers	`p{s,d,c,z}gbsv, p{s,d,c,z}gbtrf, p{s,d,c,z}gbtrs`
Symmetric eigensolver	`p{s,d}syevx, p{c,z}heevx`
Singular Value Decomposition	`p{s,d,c,z}geqrf`
Least Squares Solver	`p{s,d,c,z}gels`

CHAPTER 2

Sun S3L Data Types

Data type information is encoded in the S3L array handle for both C and Fortran interfaces and is decoded at runtime. This allows appropriate branching to occur during execution, which makes it unnecessary to maintain separate routines with different names for each language interface.

TABLE 2-1 shows the data types supported for the various Sun S3L routines. TABLE 2-2 lists the C and Fortran language-specific data type equivalents.

Within each subroutine call, elements of all array arguments must match in data type, unless the argument descriptions indicate otherwise.

Place one of the following include lines at the top of any C or Fortran program unit that makes an S3L call:

C and C++ programs

```
#include <s3l/s3l-c.h>
```

F77 and F90 programs

```
include 's3l/s3l-f.h'
```

Note – For Sun S3L 2.0 (previously released version of Sun S3L), the S3L array handles for the F77 interfaces are of type `integer*4`. For subsequent releases, they are of type `integer*8`. Therefore, when porting F77 programs from Sun S3L 2.0 to a later version, be sure to change the array handle data type definitions accordingly. If you want your F77 program to be compatible across Sun S3L 2.0 and subsequent releases, you should insert `#ifdef` statements in appropriate places in the code.

TABLE 2-1 Array Data Types Supported for C/C++ and F77/F90

Operation	int	long integer	float	double	complex	double complex
2-norm			x	x	x	x
Autocorrelation			x	x	x	x
Convolve			x	x	x	x
Copy array	x	x	x	x	x	x
Circular shift	x	x	x	x	x	x
Declare array	x	x	x	x	x	x
Deconvolve			x	x	x	x
Define array	x	x	x	x	x	x
Describe array	x	x	x	x	x	x
Exit				– N/A –		
FFT, simple and detailed complex-to-complex					x	x
FFT, inverse					x	x
FFT, simple real-to-complex			x	x		
FFT, simple complex-to-real			x	x		
Forall	x	x	x	x	x	x
Free array handle	x	x	x	x	x	x
General band solver			x	x	x	x
General iterative solver			x	x	x	x
General least squares			x	x	x	x
General singular value decomposition (SVD)			x	x	x	x
General tridiagonal			x	x	x	x
Get array elements	x	x	x	x	x	x
Get array attributes	x	x	x	x	x	x
Grade up/down	x	x	x	x	x	x

TABLE 2-1 Array Data Types Supported for C/C++ and F77/F90 *(Continued)*

Operation	int	long integer	float	double	complex	double complex
Initialize S3L environment				– N/A –		
Inner product			x	x	x	x
LU factor			x	x	x	x
LU solve			x	x	x	x
LU invert			x	x	x	x
Matrix multiplication			x	x	x	x
Matrix vector multiplication			x	x	x	x
Matrix vector sparse			x	x	x	x
Outer product			x	x	x	x
Print array	x	x	x	x	x	x
Print sparse array			x	x	x	x
Read array	x	x	x	x	x	x
Read sparse array			x	x	x	x
Reduce	x	x	x	x	x	x
Reduce axis	x	x	x	x	x	x
RNG, lagged Fibonacci	x	x	x	x	x	x
RNG, linear congruential	x	x	x	x	x	x
RNG, sparse matrix			x	x	x	x
Set array elements	x	x	x	x	x	x
Set process grid				– N/A –		
Set safety				– N/A –		
Sort	x	x	x	x		
Thread communicator setup				– N/A –		
Symmetric eigenvalues, eigenvectors			x	x	x	x
Transpose	x	x	x	x	x	x
Write array	x	x	x	x	x	x
Zero elements	x	x	x	x	x	x

TABLE 2-2 Equivalent S3L, Fortran, and C Array Data Types

S3L Data Types	F77/F90 Data Types	C/C++ Data Types
S3L_integer	INTEGER*4	int
S3L_long_integer	INTEGER*8	long long
S3L_float	REAL*4	float
S3L_double	REAL*8	double
S3L_complex	COMPLEX*8	typedef struct { float real; float imag; } S3L_cmpx8
S3L_double_complex	COMPLEX*16	typedef struct cmpx16_s { float double real; float double imag; } S3L_cmpx16

CHAPTER 3

Sun S3L Arrays

This chapter discusses various issues related to parallel arrays in the Sun S3L context. This discussion is organized into the following sections:

Overview

Sun S3L distributes arrays on an axis-by-axis basis across multiple processes, enabling operations to be performed in parallel on different sections of the array. These distributed arrays are referred to in this manual as *S3L arrays*. They may also be referred to as S3L parallel arrays.

When a message-passing program passes an array to a Sun S3L routine, it can specify any of the following distribution methods for the array's axes:

- Block – The axis is divided into blocks and distributed across the processes, with each process receiving no more than one block.
- Block-cyclic – The axis is divided into smaller blocks and distributed across the processes in round-robin fashion.
- Local – The axis is placed as an undivided whole on a single process.

Regardless of the distribution scheme specified by the calling program, Sun S3L will, if necessary, automatically distribute the axes internally in a manner that is most efficient for the routine being called. If S3L changes the distribution method internally, it will restore the original distribution scheme on the resultant array before passing it back to the calling program.

S3L Array Attributes

A principal attribute of S3L arrays is *rank*—that is, the number of dimensions, or axes, the array has. For example, an S3L array with three dimensions is called a rank-3 array. S3L arrays can have up to 31 dimensions.

An S3L array is also defined by its *extents*, which is its length along each dimension, and its *type*, which refers to the data type of its elements. The following data types are defined for S3L arrays:

- `S3L_integer` (4-byte integer)
- `S3L_long_integer` (8-byte integer)
- `S3L_float` (4-byte floating point number)
- `S3L_double` (8-byte double precision floating point number)
- `S3L_complex` (8-byte complex number)
- `S3L_double_complex` (16-byte complex number)

The C and Fortran equivalents of these array data types are described in Chapter 2.

Array Indexing

Sun S3L routines that access specific locations within arrays use either zero-based or one-based indexing:

- Zero-based indexing is applied when the calling program uses the C-language interface.
- One-based indexing is applied when the calling program uses the Fortran-language interface.

S3L Array Handles

Each S3L array must be associated with a unique S3L array handle. This is a set of internal data structures that contains a full description of the array—that is, all the information needed to define both the global and local characteristics of the array. The global definition includes such information as the array's rank and how it is distributed. The local information includes its extents and its location in the local process memory. No matter how an array has been distributed, the associated S3L array handle ensures that its layout is understood by all MPI processes.

In C programs, S3L array handles are declared as type `S3L_array_t` and in Fortran programs as type `integer*8`.

Creating S3L Array Handles

TABLE 3-1 lists the routines that Sun S3L provides for creating S3L array handles.

TABLE 3-1 Sun S3L Routines for Creating Array Handles

Routine	Notes	Comments
`S3L_declare`	Dense	Allocates memory for a dense parallel array and returns an S3L array handle that describes the array.
`S3L_declare_detailed`	Dense	Same as `S3L_declare`, but this routine gives the user control over more array mapping parameters.
`S3L_declare_sparse`	Sparse	Allocates memory for a sparse parallel array and returns an S3L array handle that describes the array. This routine is used to set up sparse S3L arrays for various sparse matrix and sparse linear systems functions.
`S3L_read_sparse`	Sparse	Sets up a sparse matrix and reads sparse matrix data from a file into it. The nonzero values are mapped into the matrix in terms of the sparse data structure stored in the file.
`S3L_rand_sparse`	Sparse	Sets up a sparse matrix and populates it with random nonzero values in a sparsity pattern that is specified by arguments in the function argument list.

Detailed descriptions of `S3L_declare` and `S3L_declare_detailed` are provided in "Creating and Destroying Array Handles for Dense S3L Arrays" on page 57.

`S3L_declare_sparse`, `S3L_read_sparse`, and `S3L_rand_sparse` are described more fully in Chapter 12.

There are three other Sun S3L routines that also create S3L array handles, but they are meant for special-case situations. They are listed in TABLE 3-2.

TABLE 3-2 Routines for Creating S3L Array Handles in Special Cases

Routine	Notes	Comments
`S3L_convert_sparse`	Sparse	Converts a sparse array from one supported sparse format to another supported sparse format. There are four such supported sparse formats.
`S3L_from_ScaLAPACK_desc`	Dense	Converts a ScaLAPACK array descriptor to an S3L array handle.
`S3L_DefineArray`	Dense	This routine is an earlier version of `S3L_declare`, but its user interface is less efficient. It is retained only for compatibility with the Sun HPC 2.0 release of Sun S3L.

These routines are all described in the *Sun S3L Reference Manual*.

Deallocating S3L Array Handles

When an S3L array is no longer needed, use `S3L_free` to deallocate a dense array and `S3L_free_sparse` to deallocate a sparse array. This makes the memory resources available for other uses.

MPI Processes and S3L Process Grids

In a Sun MPI application, each process is identified by a unique *rank*. This is an integer in the range 0 to np-1, where np is the total number of MPI processes spawned by the application.

Note – This use of the term *rank* is unrelated to the rank of an S3L array. Process ranks correspond to MPI ranks as used in interprocess communication. An S3L array's rank refers to the number of dimensions the array has.

Sun S3L maps each S3L array onto a logical arrangement of MPI processes, referred to as a *process grid*. A process grid will have the same number of dimensions as the S3L array with which it is associated. The section of an S3L array that is on a given process is called a *subgrid*.

Sun S3L controls the ordering of the processes within the n-dimensional process grid. FIGURE 3-1 through FIGURE 3-3 illustrate this. These examples show how S3L might arrange eight processes in one- and two-dimensional process grids.

In FIGURE 3-1, the eight processes form a one-dimensional grid.

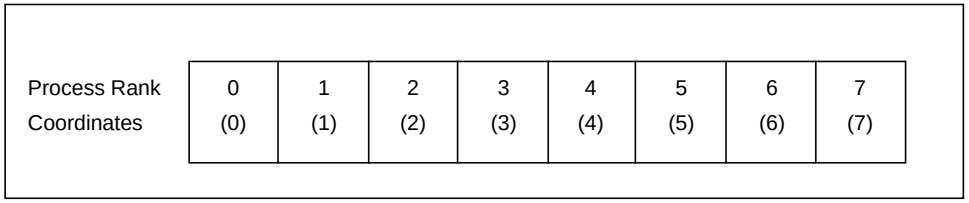


FIGURE 3-1 Eight Processes Arranged as a 1x8 Process Grid

FIGURE 3-2 and FIGURE 3-3 show the eight processes organized into rectangular 2x4 process grids.

Note that, although both process grids have 2x4 extents, they differ in their majorness attribute. This attribute determines the order in which the processes are distributed onto a process grid's axes or local subgrid axes. The two possible modes are:

- Column major – Processes are distributed along column axes first; that is, the process grid's row indices increase fastest.
- Row major – Processes are distributed along row axes first; the process grid's column indices increase fastest.

In FIGURE 3-2, subgrid distribution follows a column-major order. In FIGURE 3-3, process grid distribution is in row-major order.

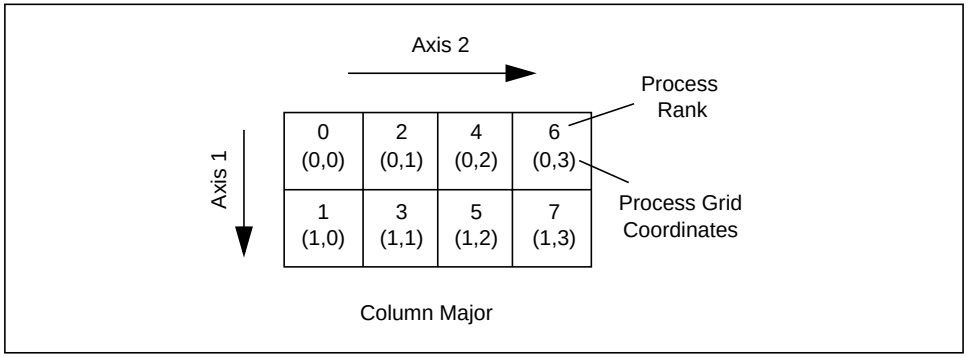


FIGURE 3-2 Eight Processes Arranged as a 2x4 Process Grid: Column-Major Order

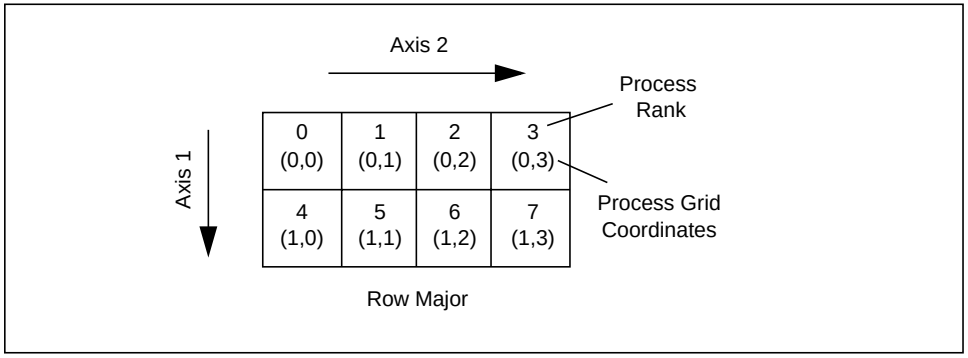


FIGURE 3-3 Eight Processes Arranged as a 2x4 Process Grid: Row-Major Order

---

Note – In these examples, axis numbers are one-based (Fortran-style). They would be zero-based for the C-language interface. Process ranks and process grid coordinates are always zero-based.

Creating Process Grids

By default, S3L will automatically assign a process grid of an appropriate shape and size whenever an S3L array handle is created. In choosing a default process grid, S3L always has the goal of producing as even a distribution of the S3L array as possible.

However, the programmer has the option of defining a process grid explicitly by calling the function `S3L_set_process_grid`. This enables the programmer to specify:

- The number of dimensions the process grid will have
- The order in which the axes are created: column major or row major
- The extent of each of the process grid's axes
- The list of processes to be included in the process grid

Upon exit, `S3L_set_process_grid` returns a process grid handle.

A process grid can be defined over the full set of processes being used by an application or over any subset of those processes. This flexibility can be useful when circumstances call for setting up a process grid that does not include all available processes.

For example, if an application will be running in a two-node cluster where one node has 14 CPUs and the other has 10, better load balancing may be achieved by defining the process grid to have 10 processes in each node.

When the process grid is no longer needed, you can deallocate its process grid handle by calling `S3L_free_process_grid`.

Detailed descriptions of `S3L_set_process_grid` and `S3L_free_process_grid` are provided in "Creating a Custom Process Grid" on page 71.

Distributing S3L Arrays

S3L array axes are distributed either locally or in a block-cyclic pattern. When an axis is distributed locally, all indices along that axis are made local to a particular process.

An axis that is distributed block-cyclically is partitioned into blocks of some *useful* size and the blocks are distributed onto the processes in a round-robin fashion:

- The first block goes to the first process, the second block to the second process, and so on. This continues until all processes have received an initial block.
- After the last process in the sequence has received its first block, the next block is sent to the first process, the block after that to the second process, and so on. This cycle is repeated until all elements in the axis have been distributed.

The definition of a useful block size will vary, depending in large part on the kind of operation to be performed. See the discussion of S3L array distribution in the *Sun HPC ClusterTools Performance Guide* for additional information about block-cyclic distribution and choosing block sizes.

A special case of block-cyclic distribution is block distribution. This involves choosing a block size that is large enough to ensure that all blocks in the axis will be distributed on the first distribution cycle—that is, no process will receive more than one block. FIGURE 3-4 through FIGURE 3-6 illustrate block and block-cyclic distributions with a sample 8x8 array distributed onto a 2x2 process grid.

In FIGURE 3-4 and FIGURE 3-5, block size is set to 4 along both axes and the resulting blocks are distributed in pure block fashion. As a result, all the subgrid indices on any given process are contiguous along both axes.

The only difference between these two examples is that process grid ordering is column-major in FIGURE 3-4 and row-major in FIGURE 3-5.

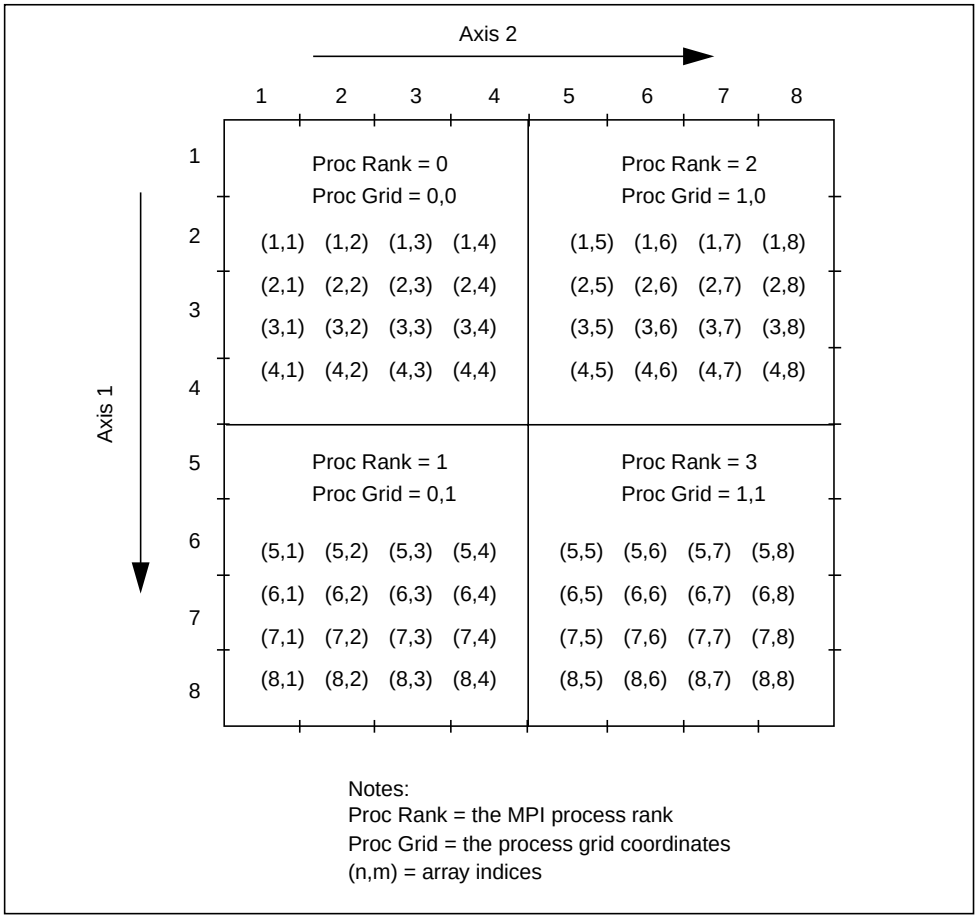


FIGURE 3-4 An 8x8 S3L Array Distributed on a 2x2 Process Grid Using Pure Block Distribution: Column-Major Ordering of Processes

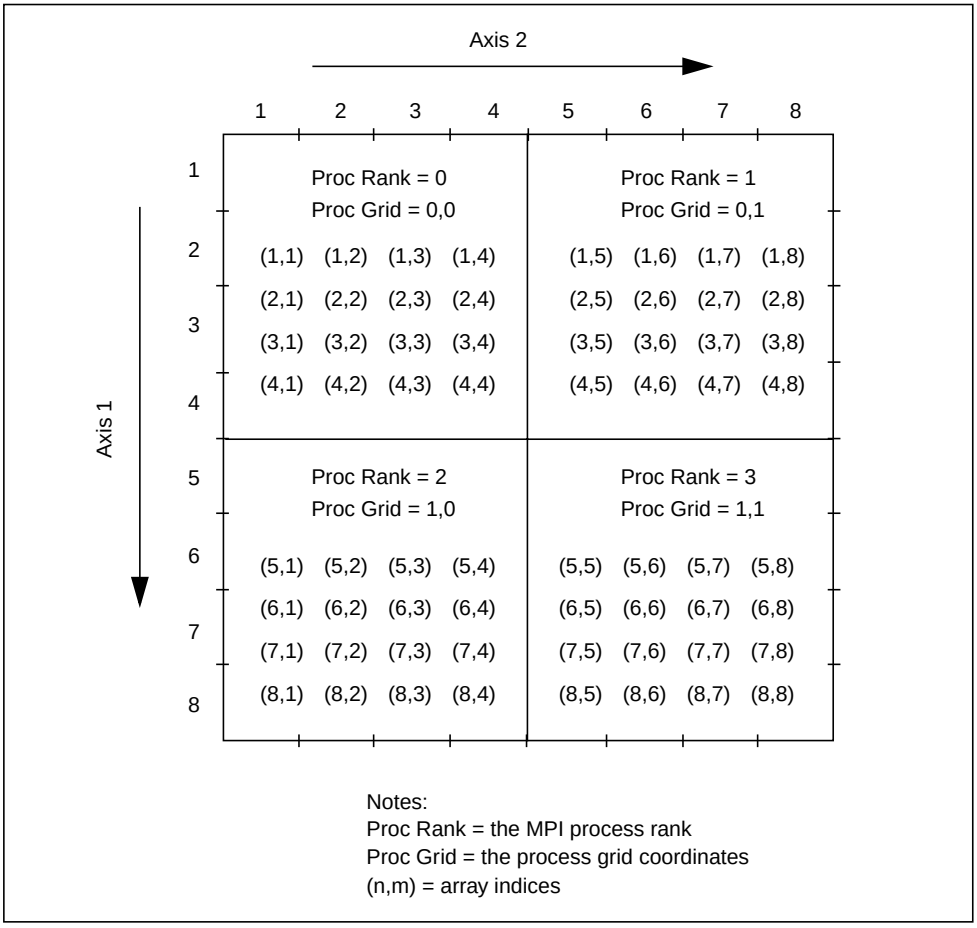


FIGURE 3-5 An 8x8 S3L Array Distributed on a 2x2 Process Grid Using Pure Block Distribution: Row-Major Ordering of Processes

FIGURE 3-6 shows block-cyclic distribution of the same array. In this example, the block size for the first axis is set to 4 and the block size for the second axis is set to 2.

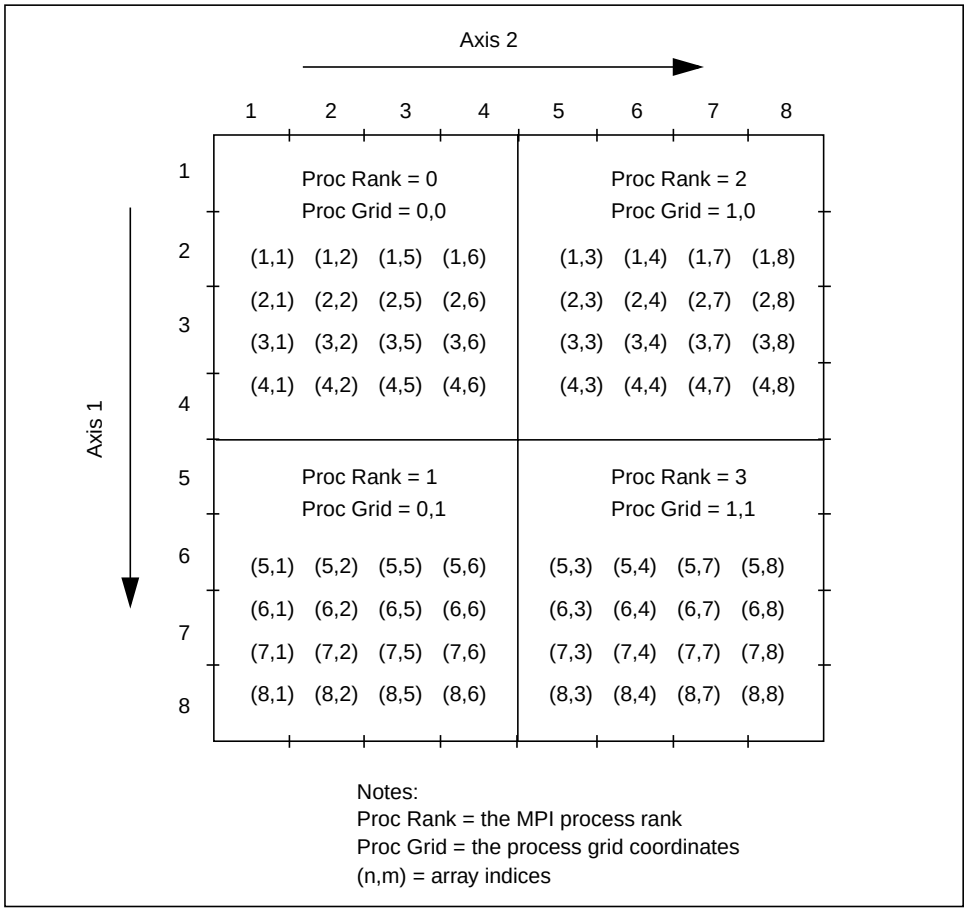


FIGURE 3-6 An 8x8 S3L Array Distributed on a 2x2 Process Grid Using Block-Cyclic Distribution: Column-Major Ordering of Processes

When no part of an S3L array is distributed—that is, when all axes are local—all elements of the array are on a single process. By default, this is the process with MPI rank 0. The programmer can request that an undistributed array be allocated to a particular process with the `S3L_declare_detailed` routine.

Although the elements of an undistributed array are defined only on a single process, the S3L array handle enables all other processes to access the undistributed array.

Examining the Contents of S3L Arrays

Printing S3L Arrays

The Sun S3L utilities `S3L_print_array` and `S3L_print_sub_array` can be used to print the values of a distributed S3L array to standard output.

`S3L_print_array` prints the whole array, while `S3L_print_sub_array` prints a section of the array that is defined by programmer-specified lower and upper bounds.

The values of array elements will be printed out in column-major order. This is referred to as Fortran ordering, where the leftmost axis index varies fastest.

Each element value is accompanied by the array indices for that value. This is illustrated by the following example.

`a` is a 4 x 5 x 2 S3L array that has been initialized to random double-precision values with a call to `S3L_rand_lcg`. A call to `S3L_print_array` will produce the following output:

```
    call s3l_print_array(a)
(1,1,1)    0.000525
(2,1,1)    0.795124
(3,1,1)    0.225717
(4,1,1)    0.371280
(1,2,1)    0.225035
(2,2,1)    0.878745
(3,2,1)    0.047473
(4,2,1)    0.180571
(1,3,1)    0.432766
...
```

For large S3L arrays, it is often a good idea to print only a section of the array rather than the entire array. This not only reduces the time it takes to retrieve the data, but it can be difficult to locate useful information in displays of large amounts of data. Printing selected sections of a large array can make the task of finding data of

interest much easier. This can be done using the function `S3L_print_sub_array`. The following example shows how to print only the first column of the array shown in the previous example:

```
      integer*4 lb(3),ub(3),st(3)

  c         specify the lower and upper bounds
  c         along each axis. Elements whose coordinates
  c         are greater or equal to lb(i) and less than or
  c         equal to ub(i) (and with stride st(i)) are
  c         printed to the output
      lb(1) = 1
      ub(1) = 4
      st(1) = 1
      lb(2) = 1
      ub(2) = 1
      st(2) = 1
      lb(3) = 1
      ub(3) = 1
      st(3) = 1
      call s3l_print_sub_array(a,lb,ub,st,ier)
```

The following output would be produced by this call:

```
(1,1,1)    0.000525
(2,1,1)    0.795124
(3,1,1)    0.225717
(4,1,1)    0.371280
```

If a stride argument other than 1 is specified, only elements at the specified stride locations will be printed. For example, the following sets the stride for axis 1 to 2:

```
st(1) = 2
```

which results in the following output:

```
(1,1,1)    0.000525
(3,1,1)    0.225717
```

Visualizing Distributed S3L Arrays With Prism

S3L arrays can be visualized with Prism, the debugger that is part of the Sun HPC ClusterTools suite. Before S3L arrays can be visualized, however, the programmer must instruct Prism that a variable of interest in an MPI code describes an S3L array.

For example, if variable `a` has been declared in a Fortran program to be of type `integer*8` and a corresponding S3L array of type `S3L_float` has been allocated by a call to `S3L_declare` or `S3L_declare_detailed`, the programmer should enter the following at the Prism command prompt:

```
type float a
```

Once this is done, Prism can print values of the distributed array:

```
print a(1:2,4:6)
```

Or it can assign values to it:

```
assign a(2,10) = 2.0
```

or visualize it:

```
print a on dedicated
```

CHAPTER 4

Multiple Instance

Most Sun S3L routines support *multiple instances.* That is, they enable you to perform multiple independent operations on different data sets concurrently. The multiple instance discussions in this chapter are organized into the following sections:

Defining Multiple Independent Data Sets

To perform a Sun S3L operation on multiple independent data sets in parallel, you must embed the multiple independent instances of each operand or result argument in a parallel array.

The shape of the parallel array is defined by two kinds of axes:

- *Data axes* define the geometry of the individual instances of the operand or result.
- *Instance axes* label the multiple instances.

FIGURE 4-1 illustrates this with an example of a matrix-vector multiplication operation in which four independent products are computed simultaneously. It shows how the destination and source vectors and the source matrix are organized with respect to the data and instance axes:

- The four destination vectors are embedded in a 2D parallel array with one data axis and one instance axis.
- The four source vectors are similarly embedded in another parallel array.
- The source matrices are embedded in a 3D parallel array.

The instances within each variable are labeled 0 through 3.

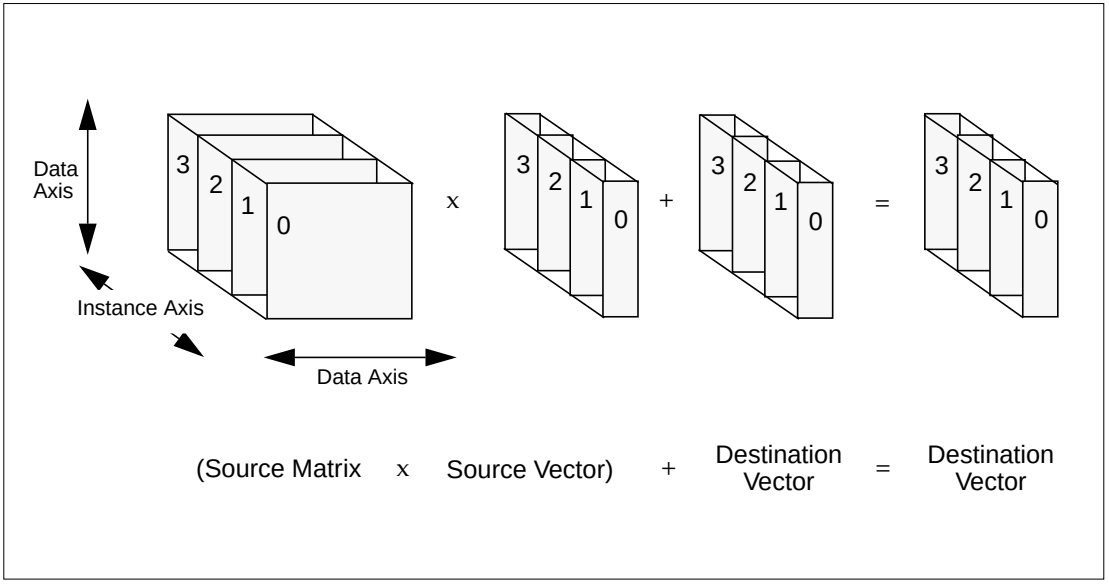


FIGURE 4-1 A Multiple-Instance Matrix-Vector Multiplication Problem

The logical unit on which the routine operates—sometimes called a *cell*—is defined by the data axes. The instance axes define the geometry of the *frame* in which the cells are embedded. The 3D parallel array shown in FIGURE 4-1 is a frame containing four 2-dimensional cells.

The product of the lengths of the instance axes is the total number of instances. The product of the lengths of the data axes is the size of the cell.

Rules for Data Axes and Instance Axes

When you organize your data to form cells and frames for a multiple-instance operation, apply the following rules:

- All parallel arrays involved in the operation must have the same number of instance axes.
- Counting up through the instance axes of the parallel arrays (excluding the data axes), corresponding instance axes must occur in the same order in each operand or result.

- The corresponding instance axes of the operands or results must have identical lengths. Certain routines also require that corresponding instance axes must also have identical layouts. The situations where identical layouts are required are identified in the applicable man pages.
- The lengths of the data axes must be defined so that the operation makes sense. For example, in matrix multiplication, the data axis lengths of the operand and result matrices must obey the standard rules for axis lengths in matrix multiplication. Specific requirements for data axis lengths are provided in the applicable man pages.
- Except where explicitly noted, Sun S3L supports all combinations of layouts for data axes and instance axes. Which layout will provide the best performance for any given operation depends largely on the nature of the operation.

In most cases, however, performance is best when the cells (that is, all of the data axes) are local to a processing element. Instance axes are typically defined as nonlocal axes. Some man pages for Sun S3L routines contain specific information about optimizing layouts.

"Specifying Single-Instance vs. Multiple-Instance Operations" on page 31 illustrates these rules being applied in a matrix-vector multiplication example.

Note – Most Sun S3L routines impose few or no restrictions on where the instance axes can occur in a parallel array.

Specifying Single-Instance vs. Multiple-Instance Operations

Sun S3L routines that support multiple instances have the same calling sequence for single-instance and multiple-instance operations. The methods for specifying single-instance and multiple-instance operations depend on which routine you are calling. The man pages for routines that are capable of multiple-instance operation contain specific information for their respective routines.

"Example 1: Matrix-Vector Multiplication" on page 32 explains the differences between single- and multiple-instance operation for the matrix-vector-multiplication routine.

"Example 2: Fast Fourier Transforms" on page 37 discusses use of multiple instances in FFTs.

Example 1: Matrix-Vector Multiplication

When you call the matrix-vector-multiplication routine, `S3L_mat_vec_mult`, the dimensionality of the arguments you supply determines whether the routine performs a single-instance or multiple-instance operation. The F77 form of this Sun S3L function is:

```
S3L_mat_vec_mult(y, a, x, y_vector_axis, row_axis, col_axis,
x_vector_axis, ier)
```

Note – The `S3L_mat_vec_mult` routine requires you to specify which axes you are using as data axes for each matrix or vector argument.

Single-Instance Operation

To perform a single-instance operation, specify each vector argument as a 1D parallel array and each matrix argument as a 2D parallel array. (Alternatively, you can declare these arguments to have more dimensions, but all instance axes must have length 1.)

For example, a single-instance operation in F77 can be performed by first defining the block-distributed arrays:

```
  integer*8 a, x, y
  integer*4 ext(2), axis_is_local(2)
  integer*4 ier

  axis_is_local(1) = 0
  axis_is_local(2) = 0

  ext(1) = p
  ext(2) = q

  call s3l_declare(a, 2, ext, S3L_float, axis_is_local,
 $    S3L_USE_MALLOC, ier)

  call s3l_declare(x, 1, ext, S3L_float, axis_is_local,
 $    S3L_USE_MALLOC, ier)

  call s3l_declare(y, 1, ext, S3L_float, axis_is_local,
 $    S3L_USE_MALLOC, ier)
```

and then using

```
call S3L_mat_vec_mult(y, a, x, 1, 1, 2, 1, ier)
```

Arrays `x` and `y` are 1D. The definitions of `x_vector_axis` = 1 and `col_axis` = 2 indicate that the product `a(i, j) * x(j)` will be evaluated for all values of `j`. These products will be summed over the first index of `a` (`row_axis` = 1), and the result added to the corresponding element in `y`. The equivalent code is

```
do i = 1, p
    sum = 0.0
    do j = i, q
        sum = sum + a(i, j) * x(j)
    enddo
enddo
```

Multiple-Instance Operation

To perform a multiple-instance operation, embed the multiple instances of each vector argument in a parallel array of rank greater than 1, and embed the multiple instances of each matrix argument in a parallel array of rank greater than 2.

For example, the simplest multiple-instance matrix-vector multiplication involves the definition of one instance axis.

```
 integer*8 a, x, y
 integer*4 ext(3), axis_is_local(3)
 integer*4 ier

 axis_is_local(1) = 0
 axis_is_local(2) = 0
 axis_is_local(3) = 0

 ext(1) = p
 ext(2) = q
 ext(3) = r

 call s3l_declare(a, 3, ext, S3L_float, axis_is_local,
$    S3L_USE_MALLOC, ier)

 ext(1) = q
 ext(2) = r

 call s3l_declare(x, 2, ext, S3L_float, axis_is_local,
$    S3L_USE_MALLOC, ier)
 ext(1) = p
```

```
  ext(2) = r

  call s3l_declare(y, 2, ext, S3L_float, axis_is_local,
 $    S3L_USE_MALLOC, ier)
```

In this code, all three arrays contain an instance axis of length `r`. In addition, each instance axis is the rightmost axis in the array declaration. In other words, the order of data axes and instance axes is the same in all three arrays. These axis definitions produce arrays whose geometries are outlined in FIGURE 4-1. In the illustration, `r` = 4.

Multiplication using these arrays is then performed by:

```
call S3L_mat_vec_mult(y, a, x, 1, 1, 2, 1, ier)
```

In analyzing the operations performed in this call, it is useful to define `s0`, the index along the instance axis. For a given value of `s0`, the following operations will be done:

- The product `a(i, j, s0) * x(j, s0)` will be calculated for all `j`. This is indicated by the values of the arguments `x_vector_axis` and `col_axis`, which are 1 and 2 respectively.
- The above product will be summed over `i`, the first index of `a` (`row_axis` = 1), and the result added to `y(i, s0)`.

These two operations will be performed for all 1 <= `s0` <= `r`. In other words, the matrix-vector multiplication will be evaluated for all instances:

```
y(:, s0) * a(:, :, s0) * x(:, s0)
```

The order in which these instances are evaluated depends on the layouts of the arrays. Since all arrays are block-distributed along all axes, it is possible for one set of processes to work on the first instance:

```
y(:, 1) = a(:, :, 1) * x(:, 1)
```

while another set of processes evaluates the *N*-th instance at the same time—that is, in parallel:

```
y(:, N) = a(:, :, N) * x(:, N)
```

The Importance of Data Layout

The extent of parallelism depends on the details of the data layouts, particularly on the mapping of the data and instance axes to the underlying process grid axes. The highest degree of parallelism is achieved when all data axes are local and all instance axes are distributed.

The use of local data axes forces each cell (that is, all data axes) to reside entirely in just one process. The use of distributed instance axes spreads the collection of cells over the process grid, resulting in better load-balancing among processes.

Multiple-instance operations are usually most efficient when each cell (all of the data axes) resides on one process. Local distribution of data axes is illustrated below, using an S3L array of rank 5, with the first two axes being the data axes and the other three being instance axes.

```
 integer*8 a, x, y
 integer*4 mat_ext(5), mat_axis_is_local(5)
 integer*4 vec_ext(4), vec_axis_is_local(4)
 integer*4 ier

 mat_axis_is_local(1) = 1
 mat_axis_is_local(2) = 1
 mat_axis_is_local(3) = 0
 mat_axis_is_local(4) = 0
 mat_axis_is_local(5) = 0

 mat_ext(1) = p
 mat_ext(2) = q
 mat_ext(3) = k
 mat_ext(4) = m
 mat_ext(5) = n

 call s3l_declare(a, 5, mat_ext, S3L_float, mat_axis_is_local,
$    S3L_USE_MALLOC, ier)

 vec_axis_is_local(1) = 1
 vec_axis_is_local(2) = 1
 vec_axis_is_local(3) = 0
 vec_axis_is_local(4) = 0

 vec_ext(1) = q
 vec_ext(2) = k
 vec_ext(3) = m
 vec_ext(4) = n

 call s3l_declare(x, 4, vec_ext, S3L_float, vec_axis_is_local,
$    S3L_USE_MALLOC, ier)

 vec_ext(1) = p
 vec_ext(2) = k
 vec_ext(3) = m
 vec_ext(4) = n
```

```
  call s3l_declare(y, 4, vec_ext, S3L_float, vec_axis_is_local,
 $    S3L_USE_MALLOC, ier)
```

The data axes are defined to be local to a process. Each array has three block-distributed instance axes. Note that the order of instance axes is the same in all three arrays.

Multiplication using these arrays is then performed by

```
call S3L_mat_vec_mult(y, a, x, 1, 1, 2, 1, ier)
```

The following is an analysis of the results of this multiple-instance matrix-vector operation. In this analysis, `s0`, `s1`, and `s2` are used to denote the index along each of the three instance axes. For a given value of `s0`, the following operations will be done:

- The product `a(i, j, s0, s1, s2) * x(j, s0, s1, s2)` will be calculated for all `j`. This is indicated by the values of the arguments `x_vector_axis` and `col_axis`, which are 1 and 2, respectively.
- This product will be summed over `i`, the first index of `a` (`row_axis` = 1), and the result added to `y(i, s0, s1, s2)`.

These two operations will be performed for all 1 <= `s0` <= `k`, 1 <= `s1` <= `m`, and 1 <= `s2` <= `n`. In other words, the matrix-vector multiplication will be evaluated for all instances:

```
y(:, s0, s1, s2) = A(:, :, s0, s1, s2) * x(:, s0, s1, s2)
```

However, unlike the previous example, the data axes in this case are local. This means that the evaluation of each instance does not involve any interprocess communication. Each process independently works on its own set of instances, using a purely local matrix-vector multiplication algorithm. These local algorithms are usually faster than their global counterparts, since no communication between processes is involved.

Source code for these operations is in the file `mat_vec_mult.f`. This can be found in the S3L examples directory:

`/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f`.

Note – `/opt/SUNWhpc` is the default location for the `examples` directory. If you cannot find the directory, it may be that your site is not using the default path.

Example 2: Fast Fourier Transforms

When calling the detailed complex-to-complex FFT routine, `S3L_fft_detailed`, you can supply a multidimensional parallel array and specify whether you want to perform a forward transform, an inverse transform, or no transform along each axis. The axes that are transformed are the data axes and define the cell. The axes along which no transformation is performed are the instance axes.

Note – The simple FFT routine, `S3L_fft`, performs a transform along each axis of the supplied parallel array. Consequently, it does not support multiple instances.

CHAPTER 5

Using Sun S3L

This chapter explains how to incorporate calls to S3L routines into your message-passing program. It covers the following topics:

Note – Sun S3L documentation includes sample online programs that demonstrate how to call each Sun S3L routine. You are encouraged to experiment with these sample programs. See "Accessing Online Program Examples and Man Pages" on page 48.

Incorporating S3L Function Calls Into Your Program

The basic steps required for incorporating Sun S3L function calls into an MPI program are relatively simple and can be summarized as shown in TABLE 5-1.

FIGURE 5-1 illustrates these steps with a simple program example. They are also discussed more fully in the following subsections.

TABLE 5-1 Summary of Basic S3L Functions

`include 's3l/s3l-f.h'` `include 's3l/s3l_errno-f.h'`	Place references to the appropriate S3L include files in the program's header section.
`S3L_init()`	Set up an S3L environment.
`S3L_thread_comm_setup()`	If the MPI program is multithreaded, use to set up support for thread-safe operation.
`S3L_set_safety()`	If in program development mode, this routine may be used to enable enhanced error checking. Alternatively, the `S3L_SAFETY` environment variable can be used for this purpose.
`S3L_declare()` `S3L_declare_sparse()`	Create S3L array handles for any dense and/or sparse parallel arrays that will be used by S3L functions.
`S3L_*(`*args*`)`	Place calls to S3L functions of interest.
`S3L_free()` `S3L_free_sparse()`	Deallocate array handles created earlier.
`S3L_exit`	Undo the S3L environment.

```
c
c Copyright (c) 2001, by Sun Microsystems, Inc.
c All rights reserved.
c
c This example shows how to create an S3L environment and then deallocate it when
c it is no longer needed. It also shows a simple MPI program that creates a 2D
c array and an S3L array handle for the array, fills the array with data, prints the
c array, and deallocates the array.

      program main

      implicit none
      include 's3l/s3l-f.h'
      include 'mpif.h'

      integer*4 nrow, ncol
      parameter (nrow = 8, ncol = nrow)

c global array descriptor

      integer*8 a

c Initialize S3L environment. Because MPI was not already initialized
c at time of s3l_init call, S3L will call mpi_init.

      call s3l_init(ier)

      call mpi_comm_rank(MPI_COMM_WORLD, me, ier)
      call mpi_comm_size(MPI_COMM_WORLD, np, ier)

c Set array extents (lengths of array axes).

      ext(1) = nrow
      ext(2) = ncol

c Specify distribution of array axes; 0 means both are block-distributed.

      axis_is_local(1) = 0
      axis_is_local(2) = 0

c Create S3L array handle for a.

      call s3l_declare(a, 2, ext, S3L_float, axis_is_local,
     $     S3L_USE_MALLOC, ier)

c Fill array a with zeros.

      call s3l_zero_elements(a, ier)

c Print contents of a to standard out.

      call s3l_print_array(a, ier)

c Deallocate array handle for a and exit s3l environment.

      call s3l_free(a, ier)
      call s3l_exit(ier)

      end
```

FIGURE 5-1 S3L_init and S3L_exit Program Example

Referencing Sun S3L Include Files

Place the appropriate include lines at the top of any program unit that makes an S3L call. The correct include files are as follows for both C and Fortran language interfaces:

C or C++

```
#include <s3l/s3l-c.h>
#include <s3l/s3l_errno-c.h>
```

F77 or F90

```
include 's3l/s3l-f.h'
include 's3l/s3l_errno-f.h'
```

The first line allows the program to access the header file containing prototypes of the routines and defines the symbols and data types required by the interface. The second line includes the header file containing error codes the routines might return.

If the compiler cannot find the Sun S3L include file, verify that a path to the directory does exist. The default path is:

```
/opt/SUNWhpc/include/
```

Setting Up the S3L Environment

Before a message-passing program can start using Sun S3L functions, every MPI process in the program must call `S3L_init`. This will create an S3L environment to handle calls from MPI processes. `S3L_init` also initializes the BLACS environment.

See Chapter 6 for detailed instructions on using `S3L_init`.

Enabling Thread-Safe Use of Sun S3L Routines

If your MPI program is multithreaded, you can use `S3L_thread_comm_setup` to create a safe environment for individual threads and sets of cooperating threads to call Sun S3L functions. `S3L_thread_comm_setup` creates the internal MPI communicators and data structures needed for thread-safe operation of Sun S3L routines.

Note – If `S3L_thread_comm_setup` is used, it must be the first S3L function called after `S3L_init`.

See "Setting Up Support for Thread-Safe Operation" on page 53 for additional information.

Using the Sun S3L Safety Mechanism

Sun S3L provides a user-controlled mechanism for increasing the level of error checking performed during S3L operations. This safety mechanism provides, in addition to the default level, three incrementally higher levels of error checking of S3L functions.

When either of the two highest error checking modes is specified, the safety mechanism also causes the MPI processes to be synchronized. This synchronization can simplify the task of isolating errors to particular processes.

The safety mechanism can be controlled either by setting the environment variable `S3L_SAFETY` or with a call to the `S3L_set_safety` function.

The safety mechanism also provides the inquiry function `S3L_get_safety`, which returns the current safety level.

See "Controlling the S3L Safety Mechanism" on page 54 for more information.

Creating S3L Array Handles for Dense Arrays

`S3L_declare` and `S3L_declare_detailed` each creates an S3L array handle that describes a dense S3L parallel array. Both support calling arguments that enable the user to specify:

- The array's rank (number of dimensions)
- The extent of each axis
- The array's data type
- Which axes, if any, will be distributed locally
- How memory will be allocated for the array

`S3L_declare_detailed` provides additional arguments, which may be used to control:

- The block sizes to be used in distributing array axes
- The starting address of the local subgrid
- Which processes contain the start of each array axis

See "Creating and Destroying Array Handles for Dense S3L Arrays" on page 57 for detailed instructions on using `S3L_declare` and `S3L_declare_detailed`.

If the simpler `S3L_declare` routine is used, S3L will automatically assign a process grid to the S3L array. The size of the process grid will be based on values supplied by the `S3L_declare` arguments, with the goal of distributing the array as evenly as possible across the processes.

If `S3L_declare_detailed` is used instead, the programmer will have the choice of either using the default process grid assigned by S3L or of specifying a custom process grid through a call to `S3L_set_process_grid`. This topic is discussed more fully in "MPI Processes and S3L Process Grids" on page 18 and in "Creating a Custom Process Grid" on page 71.

Creating S3L Array Handles for Sparse Arrays

Sun S3L provides the following functions for creating S3L array handles for sparse arrays:

- `S3L_declare_sparse`

- S3L_read_sparse
- S3L_rand_sparse

An overview of each is provided below. All are described more fully in Chapter 12.

Note – S3L_convert_sparse, which is used to convert an array from one sparse format to another, also creates an S3L array handle. But, it does so for the converted array, not for a newly created array. Since this discussion is about creating array handles for new sparse arrays, S3L_convert_sparse is not covered here. It is discussed in "Converting a Sparse Matrix From One Format to Another" on page 123.

Overview of S3L_declare_sparse

S3L_declare_sparse is similar in purpose to S3L_declare, except that it applies to sparse matrices in one of the following sparse formats:

- Coordinate
- Compressed Sparse Row
- Compressed Sparse Column

See "Supported Sparse Formats" on page 111 for detailed descriptions of these sparse formats.

Before calling S3L_declare_sparse, you must create three 1D arrays using S3L_declare or S3L_declare_detailed. Two of these arrays, designated row and col, hold mapping information (pointer and index values) that specify where the nonzero values are located in the sparse matrix. The third array, designated val, contains all the nonzero elements belonging to the sparse matrix.

See "Declaring a Sparse Matrix" on page 116 for more detailed descriptions of row, col, and val.

When calling S3L_declare_sparse, the following information is passed to the function through the argument string:

- The sparse format to be used
- The number of rows and columns in the sparse matrix
- The S3L array handles for row, col, and val

See "Declaring a Sparse Matrix" on page 116 for detailed descriptions of the arguments taken by S3L_declare_sparse.

Upon completion, S3L_declare_sparse returns an S3L array handle that describes the sparse array.

Overview of S3L_read_sparse

S3L_read_sparse reads sparse matrix data from an ASCII file and distributes the data to all participating processes. The data read from the file includes the nonzero values of the sparse matrix, as well as mapping information that specifies the row and column layout of the nonzero values in the sparse matrix.

The data must be stored in one of the following sparse formats:

- Coordinate
- Compressed Sparse Row
- Compressed Sparse Column
- Variable Block Row

When calling S3L_read_sparse, the following information is passed to the function:

- The sparse format to be used: Coordinate, Compressed Sparse Row, and so forth
- The number of rows and columns in the sparse matrix
- The number of nonzero elements in the sparse matrix
- The data type of the sparse array: S3L_float, S3L_double, S3L_complex, or S3L_double_complex
- The name of the file to be read
- The format of the data to be read from the file: ascii or ASCII

Upon completion, S3L_read_sparse returns an S3L array handle that represents the newly created sparse array.

See "Initializing a Sparse Matrix From a File" on page 117 for detailed descriptions of the arguments taken by S3L_read_sparse.

Overview of S3L_rand_sparse

S3L_rand_sparse enables you to create a sparse matrix without either supplying the nonzero data content or specifying the exact locations of the nonzero elements. It creates a global general sparse matrix, populating it with a set of random values. The sparse format can be any one of:

- Coordinate
- Compressed Sparse Row
- Compressed Sparse Column
- Variable Block Row

When calling S3L_rand_sparse, the following information is passed to the function:

- The sparse format to be used: Coordinate, Compressed Sparse Row, and so forth

- The type of random pattern to be used
- The number of rows and columns in the sparse matrix
- The desired density level; that is, the approximate percentage of elements in the sparse matrix that will be nonzero values. The density level must be expressed as a positive number no greater than 1.0
- The data type of the sparse array: `S3L_float`, `S3L_double`, `S3L_complex`, or `S3L_double_complex`
- A seed value for the random number generator
- If the Variable Block Row format is specified, two additional arguments are used to pass row and column indices. These specify where the blocks of nonzero values are located in the sparse matrix

Upon completion, `S3L_rand_sparse` returns an S3L array handle that represents the newly created sparse array.

See "Initializing a Sparse Matrix With Random Values" on page 119 for detailed descriptions of the arguments taken by `S3L_rand_sparse`.

Freeing S3L Array Handles for Dense and Sparse Arrays

Sun S3L provides separate functions for deallocating S3L array handles for dense and sparse arrays. This is necessary because sparse arrays involve more complex internal data structures, which require more steps to be deallocated.

Use `S3L_free` to deallocate a dense S3L array.

Use `S3L_free_sparse` to deallocate a sparse S3L array.

In either case, the only argument is the S3L array handle that describes the array.

Using the Sun S3L Link Switch

To link in Sun S3L at compile time, include the switch `-ls3l`. This automatically links in the Forte Developer 6 library as well.

Sun S3L requires the presence of the Sun Performance Library routines and its associated license file. This library is not installed with the Sun HPC ClusterTools components. Instead, it is included as part of the Forte Developer 6 compiler suite.

The following examples show the `-ls3l` in use with the various supported compilers:

C

```
% tmcc -dalign -o program program.c -ls3l
```

C++

```
% tmCC -dalign -o program program.cc -ls3l
```

F77

```
% tmf77 -dalign -o program program.f -ls3l
```

F90

```
% tmf90 -dalign -o program program.f90 -ls3l
```

Note – The `-dalign` option is needed because `libs3l` and `libsunperf` libraries are compiled with it.

Accessing Online Program Examples and Man Pages

Sample Code Directories

The online sample programs are located in subdirectories of the S3L `examples` directory. Separate C and Fortran versions are provided. The generic path for these examples is:

`/opt/SUNWhpc/examples/s3l/`*op_class*`[`*–lang_suffix*`]/`*ex_name.lang*

This shows the default location of the `examples` directory. If it is not in `/opt/SUNWhpc/`, ask your system administrator for the correct path.

op_class refers to the general class of operations to which the routines of interest belong.

Note – The *–lang_suffix* part of the name is used for Fortran implementations only. The *op_class* directory name does not include *–lang_suffix* for examples implemented in C.

---

ex_name.lang is the example's file name. The *lang* extension is `.c`, or `.f`. For example, the following is the Fortran version of a program example that illustrates use of `s3l_outer_prod` routines:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/outer_prod.f
```

The following shows the equivalent example for the C interface:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/outer_prod.c
```

Compiling and Running the Examples

Each example subdirectory has a makefile. Each makefile references the file `/opt/SUNWhpc/Make.simple`. If you are copying the example sources and makefiles to one of your own subdirectories, you should also copy `Make.simple` to your subdirectory's parent directory.

`Make.simple` contains definitions of compilers, compiler flags, and other variables that are needed to compile and run the examples. Note that the compiler flags in this file will *not* provide highly optimized executables. Information on optimization flags is best obtained from the documentation for the compiler being used.

Each makefile has several targets that are meant to simplify the compilation and execution of examples. If you want to compile the source codes and create all executables in a particular `example` directory, use the command, `make`.

If you wish to run the executables, enter `make run`. This command will also perform any necessary compilation and linking steps, so you need not issue `make` before entering `make run`.

By default, your executables will be run on two processes. You can change this by specifying the `NPROCS` variable on the command line. For example, the following will start execution on four processes:

```
% make run NPROCS=4
```

Executables and object files can be deleted by `make clean`.

Man Pages

To read the online man page for a Sun S3L routine, enter the following:

```
% man routine_name
```

CHAPTER 6

Setting Up the S3L Environment

This chapter describes how to prepare the S3L environment for use by an MPI application. Its contents are organized into the following sections:

Creating and Removing S3L Environments

Creating an S3L Environment

Before an application can start using Sun S3L functions, every process involved in the application must call `S3L_init` to prepare the S3L environment to handle calls from MPI processes. `S3L_init` also initializes the BLACS environment.

`S3L_init` tests the MPI library to verify that it is Sun MPI. If not, it returns the following error and terminates:

```
S3L error: invalid MPI. Please use Sun HPC MPI.
```

If the MPI layer is Sun MPI, `S3L_init` proceeds to:

- Initialize the S3L environment
- Initialize the BLACS environment
- Enable the Prism library to access Sun S3L operations

If the application calls `S3L_init` before initializing the MPI environment—that is, before it calls `MPI_init`—S3L will call `MPI_init` itself.

Note – If `S3L_init` calls `MPI_Init` internally, a subsequent call to `S3L_exit` (to undo the S3L environment) will result in an internal S3L call to `MPI_Finalize`. This will remove the MPI environment created by the S3L call to `MPI_Init`.

FIGURE 5-1 contains a program example that illustrates the use of `S3L_init`. It also illustrates `S3L_exit` as well as a few other simple S3L function calls.

`S3L_init` does not take any input arguments. If the call is made from a Fortran program, error status will be in `ier`.

Examples showing `S3L_init` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/copy_array.c
/opt/SUNWhpc/examples/s3l/utils-f/copy_array.f
```

Removing an S3L Environment

When an application is finished using Sun S3L functions, it must call `S3L_exit` to perform various cleanup tasks associated with the current S3L environment.

`S3L_exit` checks to see if the S3L environment is in the initialized state, that is, to see if `S3L_init` has been called more recently than `S3L_exit`. If not, `S3L_exit` returns the error value `S3L_ERR_NOT_INIT` and exits.

If S3L had initialized the MPI environment—that is, if `MPI_Init` had been called from within S3L rather than from the application, calling `S3L_exit` will cause S3L to call `MPI_Finalize`, which will remove the MPI environment created by the S3L call to `MPI_Init`.

See FIGURE 5-1 for an example of `S3L_exit` in use.

`S3L_exit` does not take any input arguments. If the call is made from a Fortran program, error status will be in `ier`.

Examples showing `S3L_exit` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/inner_prod.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/inner_prod.f
/opt/SUNWhpc/examples/s3l/utils-f/copy_array.f
```

Setting Up Support for Thread-Safe Operation

Sun S3L provides a setup utility that allows MPI applications containing multiple threads to safely call Sun S3L functions. This utility, `S3L_thread_comm_setup`, establishes the appropriate internal MPI communicators and data structures to support thread-safe use of S3L functions.

Note – The only Sun S3L routine that can be called before `S3L_thread_comm_setup` is `S3L_init`.

`S3L_thread_comm_setup` need not be invoked if Sun S3L functions are called from only one thread in the program.

However, when Sun S3L routines will be called from separate threads and/or sets of cooperating threads, each must call `S3L_thread_comm_setup` individually. This is necessary because a unique communicator must be used for each calling thread or set of cooperating threads.

The term *cooperating threads* refers to a set of threads that will be working on the same data. For example, one thread can initialize a random number generator, obtain a setup ID, and pass this to a cooperating thread, which will then use the random number generator.

Note – The user must ensure that the threads within a cooperating set are properly synchronized.

A unique communicator is required because S3L performs internal communications. For example, when `S3L_mat_mult` is called from a multithreaded program, the thread on one node needs to communicate with the appropriate thread on another node. This can be done only if a communicator that is unique to these threads has been previously defined and passed to the communications routines within S3L.

Note – Threads library functions are not available in F77. For this reason, no F77 interface is provided for `S3L_thread_comm_setup`.

`S3L_thread_comm_setup` has the following argument syntax:

```
S3L_thread_comm_setup(comm, ier)
```

`comm` specifies an MPI communicator that is congruent with, but not identical to, `MPI_COMM_WORLD`.

For detailed descriptions of the C bindings for this routine, see the `S3L_thread_comm_setup(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_thread_comm_setup` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/inner_prod_mt.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/matmult_mt.c
```

Controlling the S3L Safety Mechanism

Sun S3L includes an internal safety mechanism that can be useful during program development. This safety mechanism enables you to:

- Select different levels of error checking when debugging new code—see "Error Checking and Reporting" on page 54
- Force synchronization of MPI processes for easier error isolation—see "Synchronization" on page 55

Error Checking and Reporting

The safety mechanism can perform error checking and generate runtime error information at multiple levels of detail. You can turn safety checking on at any level during all or part of a program.

One safety level checks for errors in the usage and arguments of the Sun S3L calls in your program. A more detailed level also checks for errors generated by internal Sun S3L routines. Examples of errors found and reported by the safety mechanism include the following:

- A supplied or returned data element that should be numerical is not. For example, it is identified as a Not a Number (NaN), or as infinity. NaNs are defined in the IEEE Standard for Binary Floating-Point Arithmetic.
- The code generates a division by 0 (for example, because of bad data, a user error, or an internal software problem).

Note – For performance reasons, Sun S3L conducts most of its argument checking and error handling independently on each process. Consequently, when the safety mechanism is enabled and an error is detected, different processes may return different error values.

Synchronization

When a Sun S3L application executes on multiple processes, the processes are generally running asynchronously with respect to one another. The Sun S3L safety mechanism provides an interface for explicitly synchronizing the processes to each Sun S3L call made by your code. It traps and reports errors, indicating when the errors occurred relative to the synchronization points.

The S3L safety mechanism can be set to operate at any one of four levels, which are described in TABLE 6-1.

TABLE 6-1 S3L Safety Level Values

Safety Level	Description
`0`	The safety mechanism is turned off. Explicit synchronization and error checking are not performed. This level is appropriate for production runs of code that have already been thoroughly tested.
`2`	This level detects potential race conditions in multithreaded S3L operations on parallel arrays. To avoid race conditions, an S3L function locks all parallel array handles in its argument list before proceeding. This safety level causes warning messages to be generated if more than one S3L function attempts to use the same parallel array at the same time.
`5`	In addition to checking for and reporting level 2 errors, level 5 performs explicit synchronization before and after each call and locates each error with respect to the synchronization points. This safety level is appropriate during program development or during runs for which a small performance penalty can be tolerated.
`9`	This level checks for and reports all level 2 and level 5 errors, as well as errors generated by any lower levels of code called from within S3L. Performs explicit synchronization in these lower levels of code and locates each error with respect to the synchronization points. This level is appropriate for detailed debugging following the occurrence of a problem.

The Sun S3L safety mechanism can be controlled in either of two ways:

- By setting the environment variable `S3L_SAFETY`

- By using the call `S3L_set_safety` in a program

  `setenv S3L_SAFETY` *number*

  where *number* is one of: `0`, `2`, `5`, or `9`

The value of `S3L_SAFETY` is read in when `S3L_init()` is called. This value can be overridden at any point in the user's program by a call to `S3L_set_safety()`. When `S3L_set_safety()` is called, its value overrides `S3L_SAFETY` until the program completes.

If `S3L_set_safety()` is called again, the new safety level will override the previous setting. In other words, `S3L_set_safety()` can be called multiple times within a single program. The next time the program is run, the safety level specified by `S3L_SAFETY` will be reasserted

To check the current safety mechanism setting, call the companion function, `S3L_get_safety`. It will return the safety level value currently in effect. `S3L_get_safety` does not take any input arguments.

If the call is made from a Fortran program, error status will be in `ier`.

Examples showing `S3L_set_safety` and `S3L_get_safety` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/copy_array.c
/opt/SUNWhpc/examples/s3l/utils-f/copy_array.f
```

CHAPTER 7

Sun S3L Toolkit Routines for Managing Dense Arrays

This chapter describes a group of Sun S3L routines that are convenient for managing dense S3L arrays. The descriptions are organized into the following sections:

Creating and Destroying Array Handles for Dense S3L Arrays

`S3L_declare` and `S3L_declare_detailed` each creates an S3L array handle that describes an S3L array. Both support calling arguments that enable the user to specify:

- The array's rank (number of dimensions)
- The extent of each axis
- The array's data type
- Which axes, if any, will be distributed locally
- How memory will be allocated for the array

`S3L_declare_detailed` supplies additional arguments that allow more detailed control over array mapping. This additional control applies to:

- The starting address of the local subgrid. This value is used only if the programmer elects to allocate array subgrids explicitly by disabling automatic array allocation.
- The block size to be used in distributing the array along each axis. The programmer has the option of letting Sun S3L choose a default block size.
- Which processes contain the start of each array axis. Again, the programmer can let Sun S3L specify default processes. To use this option, the programmer must specify a particular process grid, rather than using one provided by Sun S3L.

`S3L_declare` and `S3L_declare_detailed` have the following argument syntax:

```
S3L_declare(A, rank, extents, type, axis_is_local, atype, ier)
S3L_declare_detailed(A, addr_a, rank, extents, type, blocksizes,
proc_src, axis_is_local, pgrid, atype, ier)
```

Upon exit, `A` contains an S3L array handle for the S3L array.

`addr_a` depends on the use of the `atype` argument. If `atype` is set to `S3L_DONOT_ALLOCATE`, `addr_a` will be taken as the starting address of the local (per process) portion of `A`. Otherwise, `addr_a` will be ignored. This argument is used only by `S3L_declare_detailed`.

`rank` specifies the number of dimensions the array will have.

`extents` is an integer vector whose length is equal to the number of dimensions in the array. Each element in `extents` specifies the extent of the corresponding array axis. The first element corresponds to the first axis, the second element to the second axis, and so forth. Axis indexing is zero-based for the C interface and one-based for the Fortran interface.

`type` specifies the array's data type, which must be one of the S3L data types listed in Chapter 2.

`blocksizes` specifies the block sizes to be used in distributing the array axes. To select default block sizes, set `blocksizes` to NULL (C/C++) or to –1 (F77/F90).

`proc_src` is a vector whose length is at least equal to the rank of `A`. Each element of `proc_src` corresponds to an axis of the process grid and is the index for the start of the array axis allocated to that process grid axis. In other words, each `proc_src` element specifies the index value of the process that contains the first data element of the corresponding array axis.

`axis_is_local` is an integer vector that contains one element for each axis of `A`. Each element specifies, with a `0` or `1` value, whether the axis it represents will be distributed across multiple processes (`0`) or locally to a single process (`1`).

Note – The `axis_is_local` argument is used only if a process grid handle is *not* specified in this call. See the description of the `pgrid` argument below.

Use the `pgrid` argument to request that S3L associate `A` with a particular process grid. To do this, supply the handle of the desired process grid for this argument. If the process grid already exists, you can acquire its handle by calling `S3L_get_attribute`. If it does not already exist, you can create it by calling `S3L_set_process_grid`.

If you want S3L to assign a default process grid to `A`, set `pgrid` to `NULL` (C/C++) or to `0` (F77/F90).

`atype` specifies how `A` will be allocated in memory. Use one of the following predefined values for this argument:

- `S3L_USE_MALLOC` – Uses `malloc()` to allocate the array subgrids.
- `S3L_USE_MEMALIGN64` – Uses `memalign()` to allocate the array subgrids and to align them on 64-byte boundaries.
- `S3L_USE_MMAP` – Uses `mmap()` to allocate the array subgrids. Array subgrids on the same SMP (node) will be in shared memory.
- `S3L_USE_SHMGET` – Uses `shmget()` to allocate the array subgrids. Array subgrids on the same SMP (node) will be in shared memory.

Notes

When `S3L_USE_MMAP` or `S3L_USE_SHMGET` is used on a 32-bit platform, the part of an S3L array owned by a single SMP cannot exceed 2 gigabytes.

When `S3L_USE_MALLOC` or `S3L_USE_MEMALIGN64` is used, the part of the array owned by any single process cannot exceed 2 gigabytes.

If these size restrictions are violated, an `S3L_ERR_MEMALLOC` will be returned. On 64-bit platforms, the upper bound is equal to the system's maximum available memory.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for these routines, see the `S3L_declare(3)` and `S3L_declare_detailed(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

S3L Declare Example

The following F77 program example illustrates the use of `S3L_declare` with a 1D, double-precision array of length 1000.:

```
include 's3l/s3l-f.h'
include 's3l/s3l-errno-f.h'
integer*4 local,ext,ier
integer*8 A
local = 0
ext = 1000
call s3l_declare(A,1,ext,S3L_double,local,S3L_USE_MALLOC,ier)
```

In this example, the array has only one axis, so `array_is_local` contains a single element, whose value is `0`, which means the axis will be distributed across multiple processes.

If the program containing this code is run on six processes, Sun S3L will associate a 1D process grid of length 6 with `A`. It will set the block size of the array distribution to `ceiling`(1000/6)=167. As a result, processes 0 though 4 will have 167 local array elements and process 5 will have 165.

If `array_is_local` had instead been set to `1`, the entire array would have been allocated to process 0.

Upon successful completion, `S3L_declare` returns an S3L array handle, which subsequent S3L calls can use as an argument to gain access to that array.

S3L Declare Detailed Example

The following example illustrates the extra control that `S3L_declare_detailed` provides over array handle creation:

```
include 's3l/s3l-f.h'
include 's3l/s3l_errno-f.h'
integer*8 A, pg_a
integer*4 ext_a(3), block_a(3), local_a(3), ier
ext_a(1) = 100
ext_a(2) = 100
ext_a(3) = 100
local_a(1) = 1
local_a(2) = 0
local_a(3) = 0
call s3l_declare_detailed(A,0,3,ext_a,S3L_double,block_a,
$     -1,local_a,pg_a,S3L_USE_MALLOC,ier)
```

In this example, `S3L_declare_detailed` creates an array handle for a 3D, double-precision array. The extent of each array axis is 100.

Each axis of S3L array `A` will be distributed, using block sizes specified in `block_a`.

Because `local_a(1)` is set to `1`, the first axis of `A` will be local to the first process in the process grid's first axis. The second and third axes of `A` will be distributed along the second and third axes of the process grid.

If `local_a(1)` had been set to `0` instead, it would have been distributed across all processes in the process grid's first axis. That is, all three array axes would have been distributed across multiple processes along their respective process grid axes.

Examples showing `S3L_declare` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/transpose/ex_trans1.c
/opt/SUNWhpc/examples/s3l/grade-f/ex_grade.f
```

Examples showing `S3L_declare_detailed` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/copy_array.c
/opt/SUNWhpc/examples/s3l/utils-f/copy_array.f
/opt/SUNWhpc/examples/s3l/utils/get_attribute.c
/opt/SUNWhpc/examples/s3l/utils-f/get_attribute.f
/opt/SUNWhpc/examples/s3l/utils/scalapack_conv.c
/opt/SUNWhpc/examples/s3l/utils-f/scalapack_conv.f
```

Converting Between ScaLAPACK Descriptors and S3L Array Handles

If your program includes arrays that are described by ScaLAPACK descriptors, you can use `S3L_from_ScaLAPACK_desc` to convert those descriptors into S3L array handles.

Sun S3L also provides a routine, `S3L_from_ScaLAPACK_desc`, for converting S3L array handles into ScaLAPACK array descriptors.

Converting From ScaLAPACK to S3L

`S3L_from_ScaLAPACK_desc` has the following argument syntax:

```
S3L_from_ScaLAPACK_desc(s3ldesc, scdesc, data_type, address, ier)
```

`s3ldesc` is the S3L array handle that `S3L_from_ScaLAPACK_desc` produces on exit.

`scdesc` is the ScaLAPACK array descriptor.

`data_type` specifies the data type of the array.

`address` holds the starting address of the existing array subgrid.

Note – In Fortran programs, `address` should be either a pointer or the starting address of a local array, as determined by the `loc(3F)` function.

If the call is made from a Fortran program, error status will be in `ier`.

Converting From S3L to ScaLAPACK

`S3L_to_ScaLAPACK_desc` has the following argument syntax:

```
S3L_to_ScaLAPACK_desc(s3ldesc, scdesc, data_type, address, ier)
```

`s3ldesc` contains the S3L array handle that is to be converted to a ScaLAPACK array descriptor.

`scdesc` contains the ScaLAPACK array descriptor that is produced on exit.

`data_type` specifies the data type of the S3L array. This must be one of the supported data types described in Chapter 2.

`address` holds the starting address of the existing array subgrid.

Note – In Fortran programs, `address` should be either a pointer or the starting address of a local array, as determined by the `loc(3F)` function.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for these routines, see the `S3L_from_ScaLAPACK_desc(3)` and `S3L_to_ScaLAPACK_desc(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_from_ScaLAPACK_desc` and `S3L_to_ScaLAPACK_desc` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/scalapack_conv.c
/opt/SUNWhpc/examples/s3l/utils-f/scalapack_conv.f
```

Freeing S3L Array Handles

When an S3L array is no longer needed, call `S3L_free` to deallocate the internal structures that describe the array. The routine has the following argument syntax:

```
S3L_free(a, ier)
```

`a` is the array handle to be deallocated.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_free(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_free` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/io/ex_print1.c
/opt/SUNWhpc/examples/s3l/io-f/ex_print1.f
```

Initializing an S3L Array From a File

`S3L_read_array` enables you to populate an S3L array with data read from a file. The process with MPI rank 0 reads the data from a local file and distributes the data to all processes that have local subgrids of the S3L array.

`S3L_read_sub_array` is similar to `S3L_read_array`, except it distributes the file data to specific sections of each axis of the S3L array. Lower and upper bound arguments are available, which allow the specification of the first and last indices along each axis. A stride length greater than 1 can also be specified for any axes to produce noncontiguous indexing along the affected axes.

For both functions, the format of the file must be either ASCII or binary.

`S3L_read_array` and `S3L_read_sub_array` have the following argument syntax:

```
S3L_read_array(a, filename, format, ier)
S3L_read_sub_array(a, lbounds, ubounds, strides, filename,
format, ier)
```

`a` is an S3L array handle describing the S3L array to be initialized. This array handle was returned by a previous call to either `S3L_declare` or `S3L_declare_detailed`.

`lbounds` is an integer vector that is used in calls to `S3L_read_sub_array`. Each element of `lbounds` corresponds to an axis of `a` and specifies the lower bound of the indices along the axis it represents. This lower-bound index marks the beginning of a subset of array elements along that axis that will be initialized. The default lower bound is

- 0 for the C interface
- 1 for the Fortran interface.

`ubounds` is an integer vector that is used in calls to `S3L_read_sub_array`. Each element of `ubounds` corresponds to an axis of `a` and specifies the upper bound of the indices along the axis it represents. This upper-bound index marks the upper end of a subset of array elements along that axis that will be initialized. The default upper bound is:

- axis extent –1 for the C interface
- axis extent for the Fortran interface

`strides` is an integer vector that is used in calls to `S3L_read_sub_array`. Each element of `strides` corresponds to an axis of `a` and specifies a stride length to be used in indexing along that axis. The default stride length is 1, which produces contiguous indexing.

`filename` is a scalar character variable that specifies the name of the file to be read.

`format` is a scalar character variable used to specify the format of the data to be read. The value can be either `ascii` or `binary`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for these routines, see the `S3L_read_array(3)` and `S3L_read_sub_array(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_read_array` and `S3L_read_sub_array` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/io/ex_io.c
/opt/SUNWhpc/examples/s3l/io-f/ex_io.f
```

Writing an S3L Array to a File

`S3L_write_array` causes the process with MPI rank 0 to write a distributed S3L array into a specified file. The file is local to the process with MPI rank 0.

`S3L_write_sub_array` writes a subset of the distributed S3L array to a specified file. This subset is defined by arguments that specify the lower and upper bounds of the section of each axis to be written. A stride length greater than 1 can also be specified for any axes to produce noncontiguous indexing along the affected axes.

For both functions, the format of the file must be either `ascii` or `binary`.

`S3L_write_array` and `S3L_write_sub_array` have the following argument syntax:

```
S3L_write_array(a, filename, format, ier)
S3L_write_sub_array(a, lbounds, ubounds, strides, filename,
format, ier)
```

`a` is an S3L array handle describing the S3L array to be written. This array handle was returned by a previous call to either `S3L_declare` or `S3L_declare_detailed`.

`lbounds` is an integer vector that is used in calls to `S3L_write_sub_array`. Each element of `lbounds` corresponds to an axis of `a` and specifies the lower bound of the indices along the axis it represents. This lower-bound index marks the beginning of a subset of array elements along that axis that will be written. The default lower bound is:

- 0 for the C interface
- 1 for the Fortran interface

`ubounds` is an integer vector that is used in calls to `S3L_write_sub_array`. Each element of `ubounds` corresponds to an axis of `a` and specifies the upper bound of the indices along the axis it represents. This upper-bound index marks the upper end of a subset of array elements along that axis that will be written. The default upper bound is:

- axis extent –1 for the C interface
- axis extent for the Fortran interface

`strides` is an integer vector that is used in calls to `S3L_write_sub_array`. Each element of `strides` corresponds to an axis of `a` and specifies a stride length to be used in indexing along that axis. The default stride length is 1, which produces contiguous indexing.

`filename` is a scalar character variable that specifies the name of the file to be written to.

`format` is a scalar character variable used to specify the format of the data to be written. The value can be either `ascii` or `binary`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for these routines, see the `S3L_write_array(3)` and `S3L_write_sub_array(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_write_array` and `S3L_write_sub_array` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/io/ex_io.c
/opt/SUNWhpc/examples/s3l/io-f/ex_io.f
```

Printing an S3L Array to Standard Output

`S3L_print_array` causes the process with MPI rank 0 to print the S3L array described by the array handle `a` to standard output.

`S3L_print_sub_array` prints a specified subset of the S3L array. This subset is defined by the `lbounds`, `ubounds`, and `strides` arguments. `lbounds` and `ubounds` specify the lower and upper boundaries for indexing along each axis. `strides` specifies the stride length to be used along each axis. It must be greater than zero.

`S3L_print_array` and `S3L_print_sub_array` have the following argument syntax:

```
S3L_print_array(a, ier)
S3L_print_sub_array(a, lbounds, ubounds, strides, ier)
```

`a` is an S3L array handle describing the S3L array to be printed. This array handle was returned by a previous call to either `S3L_declare` or `S3L_declare_detailed`.

`lbounds` is an integer vector that is used in calls to `S3L_print_sub_array`. Each element of `lbounds` corresponds to an axis of `a` and specifies the lower boundary of the indices along the axis it represents. This lower-bound index marks the beginning of a subset of array elements along that axis that will be printed. The default lower bound is:

- 0 for the C interface
- 1 for the Fortran interface

ubounds is an integer vector that is used in calls to S3L_print_sub_array. Each element of ubounds corresponds to an axis of a and specifies the upper boundary of the indices along the axis it represents. This index marks the upper end of a subset of array elements along that axis that will be printed. The default upper bound is:

- axis extent –1 for the C interface
- axis extent for the Fortran interface

strides is an integer vector that is used in calls to S3L_print_sub_array. Each element of strides corresponds to an axis of a and specifies a stride length to be used in indexing along that axis. The default stride length is 1, which produces contiguous indexing.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for these routines, see the S3L_print_array(3) and S3L_print_sub_array(3) man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing S3L_print_array and S3L_print_sub_array in use can be found in:

```
/opt/SUNWhpc/examples/s3l/io/ex_print1.c
/opt/SUNWhpc/examples/s3l/io/ex_io.c
/opt/SUNWhpc/examples/s3l/io-f/ex_io.f
```

Copying S3L Arrays

S3L_copy_array copies the contents of S3L array A into S3L array B, which must have the same rank, extents, and data type as A.

S3L_copy_array_detailed copies a specified subset of S3L array A into the corresponding section of S3L array B. The subset of A to be copied is defined along each axis by the indices:

```
lbA(i), <= j <= ubA(i), with strides stA(i), i=0, rank-1
```

where lbA and ubA are the lower- and upper-bound indices of axes of A and stA is the indexing stride to be used along axes of A.

The array section of B is defined along each axis by the indices:

```
lbB(i), <= j <= ubB(i), with strides stB(i), i=0, rank-1
```

where lbB, ubB, and stB are analogous to lbA, ubA, and stA.

S3L_copy_array and S3L_copy_array_detailed have the following argument syntax:

```
S3L_copy_array(A, B, ier)
S3L_copy_array_detailed(A, B, lbA, ubA, stA, lbB, ubB, stB,
perm, ier)
```

A is an S3L array handle describing the S3L array to be copied (the source array). This array handle was returned by a previous call to either S3L_declare or S3L_declare_detailed.

B is an S3L array handle describing the S3L array into which A is to be copied (the destination array). This array handle was returned by a previous call to either S3L_declare or S3L_declare_detailed.

B must have the same rank, extents, and data type as A.

lbA is an integer vector that is used in calls to S3L_copy_array_detailed. Each element of lbA corresponds to an axis of A and specifies the lower bound of the indices along the axis it represents. This lower-bound index marks the beginning of a subset of array elements along that axis that will be copied. The default lower bound is:

- 0 for the C interface
- 1 for the Fortran interface

ubA is an integer vector that is used in calls to S3L_copy_array_detailed. Each element of ubA corresponds to an axis of A and specifies the upper bound of the indices along the axis it represents. This upper-bound index marks the upper end of a subset of array elements along that axis that will be copied. The default upper bound is:

- axis extent –1 for the C interface
- axis extent for the Fortran interface

stA is an integer vector that is used in calls to S3L_copy_array_detailed. Each element of stA corresponds to an axis of A and specifies a stride length to be used in indexing along that axis. The default stride length is 1, which produces contiguous indexing.

lbB, ubB, and stB have the same meaning for the destination array B as lbA, ubA, and stA have for source array A.

perm is an integer vector that is used in calls to S3L_copy_array_detailed. The first element of perm controls whether the other elements will be evaluated. It does so in the following way: If the first element is NULL (C interface) or negative (Fortran interface), perm is ignored. Otherwise, the other elements are evaluated.

Each of the other elements of perm correspond to an axis of B and specify whether the axis it represents will be permuted.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for these routines, see the `S3L_copy_array(3)` and `S3L_copy_array_detailed(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_copy_array` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/copy_array.c
/opt/SUNWhpc/examples/s3l/utils-f/copy_array.f
```

Examples showing `S3L_copy_array_detailed` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/copy_array_det.c
/opt/SUNWhpc/examples/s3l/utils-f/copy_array_det.f
```

CHAPTER 8

Creating and Freeing Custom Process Grids

This chapter explains how to create and free a process grid with user-defined parameters. It contains the following sections:

Creating a Custom Process Grid

`S3L_set_process_grid` enables you to define various aspects of the process grid and associate it with an S3L array handle. It has the following argument syntax:

```
S3L_set_process_grid(pgrid, rank, majorness, grid_extents,
plist_length, process_list, ier)
```

Upon exit, `pgrid` contains the handle for the process grid.

`rank` specifies the number of dimensions the process grid is to have. This must be the same as the rank of the S3L array with which it will be associated.

`majorness` specifies the order in which execution will proceed within the process grid. Use one of the following predefined values for this argument:

`S3L_MAJOR_COLUMN`	Execution proceeds from leftmost axis to rightmost axis.
`S3L_MAJOR_ROW`	Execution proceeds from leftmost axis to rightmost axis.

For example, if (`i`,`j`) represents the indices for a process grid's first and second axes, specifying `S3L_MAJOR_COLUMN` will cause `i` to be the inner loop index and `j` to be the outer loop index. For `S3L_MAJOR_ROW`, the sequence would be the opposite.

`grid_extents` is an integer vector whose length equals the rank of the process grid. Each element in `grid_extents` specifies the extent of the corresponding axis of the process grid. Axis indexing is zero-based for the C interface and one-based for the Fortran interface.

`plist_length` specifies the length of the `process_list` argument, which is described below.

`process_list` is an integer array whose length is specified in `plist_length`. It contains a list of the processes that will be included in the process grid. For example, if your program is running on MPI processes 0 through 3, but you want to create a process grid for a particular S3L array consisting only of processes 1 and 3, set `plist_length` to 2 and have:

```
process_list[0] = 1
process_list[1] = 3
```

If `plist_length` is 0, `process_list` will be ignored. The process grid is then created using all available processes in `MPI_COMM_WORLD`.

Note – If the product of all grid extents is *N* and if a value greater than *N* is specified for `plist_length`, only the first *N* elements of `process_list` will be used.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_set_process_grid(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Set Process Grid Example

The following F77 example shows how to specify a two-dimensional process grid that is defined over a set of eight processes having MPI ranks 0 through 7. The process grid has extents of 2x4 and is assigned column-major ordering.

```
include 's3l/s3l-f.h'
integer*8 pg
integer*4 rank
integer*4 pext(2),process_list(8)
integer*4 i,ier

rank = 2
pext(1) = 2
pext(2) = 4
```

```
do i=1,8
    process_list(i)=i-1
end do
call s3l_set_process_grid(pg,rank,S3L_MAJOR_COLUMN,
    pext,8,process_list,ier)
```

Further examples showing `S3L_set_process_grid` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/scalapack_conv.c
/opt/SUNWhpc/examples/s3l/utils-f/scalapack_conv.f
```

Deallocating a Process Grid

`S3L_free_process_grid` frees the process grid handle returned by a previous call to `S3L_set_process_grid`. It has the following argument syntax:

```
S3L_set_process_grid(pgrid, ier)
```

`pgrid` is the process grid handle to be deallocated.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_free_process_grid(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_free_process_grid` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/scalapack_conv.c
/opt/SUNWhpc/examples/s3l/utils-f/scalapack_conv.f
```

CHAPTER 9

Extracting Information From S3L Arrays and Process Grids

Sun S3L provides several functions for extracting information about S3L arrays and process grids, as well as accessing the contents of the arrays. These functions are discussed in the following sections:

Extracting Descriptions of S3L Arrays and Process Grids

`S3L_describe` prints information about an S3L array or process grid to standard output. It has the following argument syntax:

```
S3L_describe(A, info_node, ier)
```

`A` is an S3L handle for either an S3L array or a process grid, whichever object is to be described.

`info_node` is a scalar integer variable that specifies the rank of the process from which the information is to be taken.

Note – Some array parameters, such as subgrid size and address values, will vary from process to process.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_describe(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_describe` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/scalapack_conv.c
/opt/SUNWhpc/examples/s3l/utils-f/scalapack_conv.f
```

Extracting S3L Array Attributes

Sun S3L provides a function, `S3L_get_attribute`, for extracting specific details about an S3L array's attributes. TABLE 9-1 lists the attributes that can be acquired with `S3L_get_attribute`.

TABLE 9-1 `S3L_get_attribute` Output Values

Attribute Name	Description
`S3L_ELEM_TYPE`	Returns the S3L data type of the elements of an S3L array or sparse matrix.
`S3L_ELEM_SIZE`	Returns the size (in bytes) of the elements of an S3L dense array or sparse matrix.
`S3L_RANK`	Returns the rank (number of dimensions) of an S3L dense array or sparse matrix.
`S3L_EXTENT`	If requesting an S3L array extent attribute, returns the extent of an S3L dense array or sparse matrix along the specified axis. If requesting a process grid extent attribute, it returns the number of processes over which a given axis of an array is distributed.
`S3L_BLOCK_SIZE`	Returns the block size of the block-cyclic distribution of an S3L dense array along the specified dimension.
`S3L_BLOCK_START`	Returns the index of the starting process of the block-cyclic distribution of an S3L dense array along the specified dimension.
`S3L_SGRID_SIZE`	Returns the subgrid size of the block-cyclic distribution of an S3L dense array along the specified dimension.
`S3L_AXIS_LOCAL`	Returns 0 if the axis is local (not distributed) and 1 if it is distributed.

TABLE 9-1 `S3L_get_attribute` Output Values *(Continued)*

Attribute Name	Description
`S3L_SGRID_ADDRESS`	Returns the starting address of the local subgrid (local per-process part) of an S3L dense array.
`S3L_MAJOR`	If requesting an S3L array attribute, returns the majorness of the elements in the local part of the array. If requesting a process grid attribute, returns the majorness (F77 or C) of the process grid. In either case, the majorness value may be either `S3L_MAJOR_COLUMN` (F77 major) or `S3L_MAJOR_ROW` (C major).
`S3L_ALLOC_TYPE`	Returns one of the predefined memory allocation types for dense S3L arrays. See the description of the `atype` argument in "Creating and Destroying Array Handles for Dense S3L Arrays" on page 57 for descriptions of the supported allocation types.
`S3L_SHARED_ADDR`	For dense S3L arrays that have been allocated in shared memory (single SMP case), returns the global starting address of the array. All processes can directly access all elements of such arrays without the need for explicit interprocess communication.
`S3L_PGRID_DESC`	Returns the process grid descriptor associated with an S3L dense array or sparse matrix.
`S3L_SCALAPACK_DESC`	For 1D and 2D S3L dense arrays, returns the ScaLAPACK array descriptor associated with the array.
`S3L_SPARSE_FORMAT`	For an S3L sparse matrix, returns the sparse format in which the matrix is stored.
`S3L_NONZEROS`	For an S3L sparse matrix, returns the number of nonzero elements of the matrix.
`S3L_RIDX_SGRID_ADDR`	For an S3L sparse matrix stored in the `S3L_SPARSE_COO` format, returns the starting address of an array of index sets containing the local row numbers that comprise each local submatrix. For an S3L sparse matrix stored in the `S3L_SPARSE_CSR` format, it returns the starting address of an array containing pointers to the beginning of each row of the local submatrix.
`S3L_CIDX_SGRID_ADDR`	For an S3L sparse matrix stored in either the `S3L_SPARSE_COO` or `S3L_SPARSE_CSR` format, returns the starting address of an array of index sets containing the global column numbers that comprise each local submatrix.
`S3L_NRZS_SGRID_ADDR`	For an S3L sparse matrix stored in either the `S3L_SPARSE_COO` or `S3L_SPARSE_CSR` format, `S3L_NZRS_SGRID_ADDR` returns the starting address of an array containing nonzero elements of the local submatrix.

TABLE 9-1 `S3L_get_attribute` Output Values *(Continued)*

Attribute Name	Description
`S3L_RIDX_SGRID_SIZE`	For an S3L sparse matrix stored in the `S3L_SPARSE_COO` format, returns the size of an array of index sets containing the local row numbers that comprise each local submatrix. For an S3L sparse matrix stored in the `S3L_SPARSE_CSR` format, returns the size of an array containing the pointers to the beginning of each row of the local submatrix.
`S3L_CIDX_SGRID_SIZE`	For an S3L sparse matrix stored in either the `S3L_SPARSE_COO` or `S3L_SPARSE_CSR` format, returns the size of an array of index sets containing the global column numbers that comprise each local submatrix.
`S3L_NRZS_SGRID_SIZE`	For an S3L sparse matrix stored in either the `S3L_SPARSE_COO` or `S3L_SPARSE_CSR` format, returns the size of an array containing nonzero elements of the local submatrix.
`S3L_COORD`	Returns the coordinate of the calling process in an S3L process grid, along the dimension given in axis.
`S3L_ON_SINGLE_SMP`	Returns 1 if an S3L process grid is defined on a single SMP and 0 if not.

`S3L_get_attribute` has the following argument syntax:

```
S3L_get_attribute(a, req_attr, axis, attr, ier)
```

`a` is a pointer to a descriptor of an unknown type.

`req_attr` is the predefined value that specifies the attribute to be accessed. See TABLE 9-1 for the list of values that this argument can have.

`axis` is a scalar integer variable that is used to access attributes that are axis-specific, such as extents or block sizes.

On exit, `attr` contains the attribute value.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_get_attribute(3)` man page or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_get_attribute` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/get_attribute.c
/opt/SUNWhpc/examples/s3l/utils-f/get_attribute.f
```

Obtaining and Setting Array Elements

Sun S3L provides the following routines for accessing specific array elements in a distributed array:

- `S3L_get_array_element`
- `S3L_set_array_element`
- `S3L_get_array_element_on_proc`
- `S3L_set_array_element_on_proc`

These functions locate the target elements by means of their global coordinates, which the programmer supplies.

`S3L_get_array_element`

Use `S3L_get_array_element` to obtain the value of a specific element from a distributed array. The argument `coor` specifies the global coordinates of the target element. The process that contains the array element specified in `coor` will return the value of that element in `val`.

Each process that contains a subgrid of the S3L array checks to see if the global coordinates in `coor` map to their local subgrid. The process that has the target element in its local subgrid stores the value of the element in the local variable, `val`. All other processes exit from `S3L_get_array_element` without modifying their local `val` variables.

FIGURE 9-1 illustrates this with a diagram of a 1 x 16 array that is distributed over a 1 x 4 process grid. `S3L_get_array_element` is used to obtain the value of the array element at global coordinate 6.

Note – For clarity, symbols such as '`!`' and '`@`' are used to represent the values stored in the various array elements.

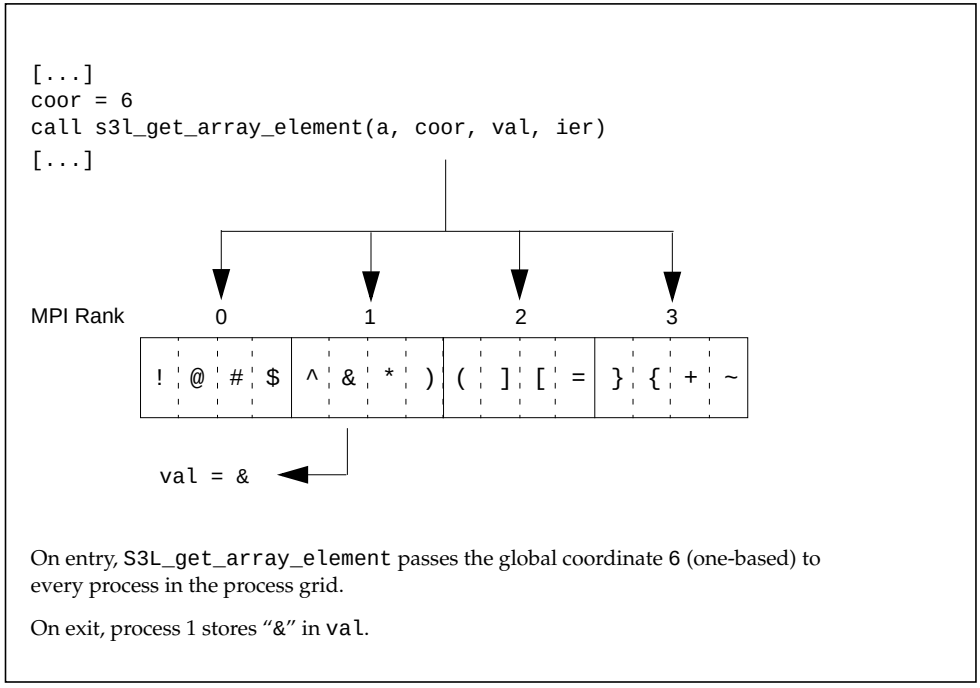


FIGURE 9-1 Illustration of `S3L_get_array_element` Use for a 1 x 16 S3L Array

`S3L_get_array_element` has the following argument syntax:

```
S3L_get_array_element(a, coor, val, ier)
```

`a` is an S3L array handle for the distributed array containing the element of interest.

`coor` is an integer vector that specifies the global coordinates of a particular element in the S3L array described by `a`.

`val` is an array variable. On exit, the `val` that is local to the process containing the target element will contain the value of the target element.

S3L_set_array_element

`S3L_set_array_element` assigns the value stored in `val` to the array element whose global coordinates are in the argument `coor`.

Each process that contains a subgrid of the S3L array will store the value to be assigned to the target element in a local copy of the `val` variable and will check to see if the global coordinates in `coor` map to the local subgrid. The process that has the target element in its local subgrid will assign the contents of `val` to the element. All other processes exit `S3L_set_array_element` without modifying any elements in their subgrids.

FIGURE 9-2 illustrates this with the same 1 x 16 S3L array and 1 x 4 process grid. In this case, the `S3L_set_array_element` call will assign the value `W` to the array element at global coordinate `6`. All processes in the process grid will call `S3L_set_array_element`, but only the process with MPI rank 1 will update a local array element—the element at local coordinate 2.

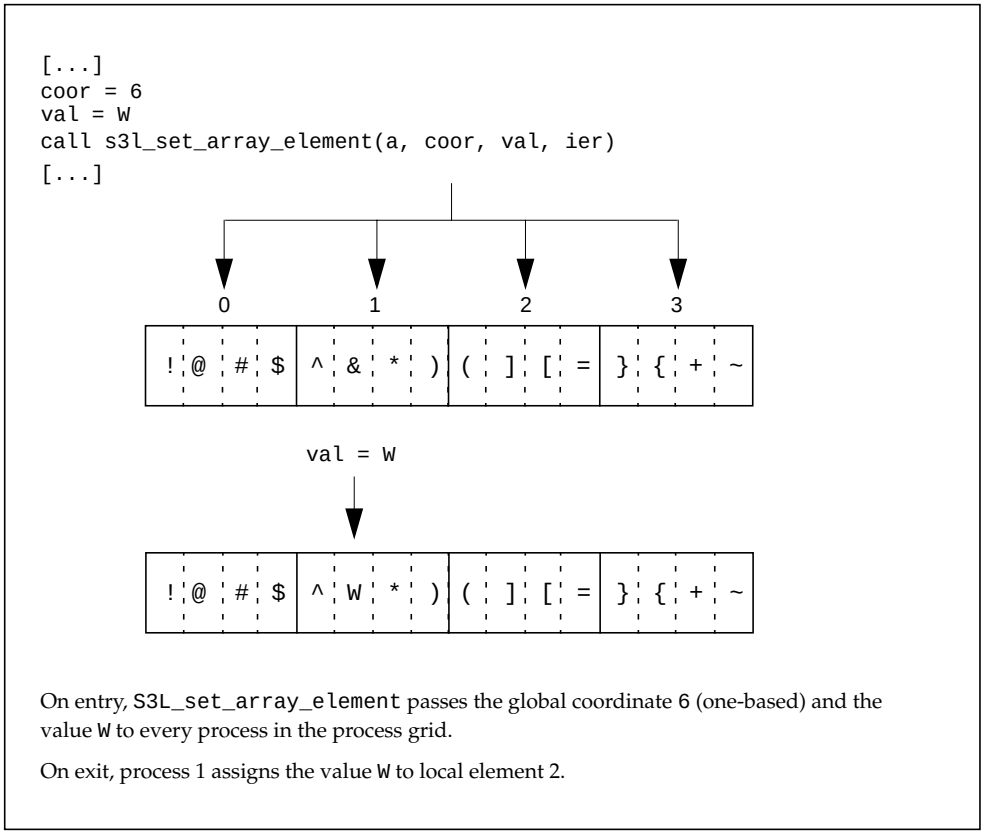


FIGURE 9-2 Illustration of `S3L_set_array_element` Use for a 1 x 16 S3L Array

`S3L_set_array_element` has the following argument syntax:

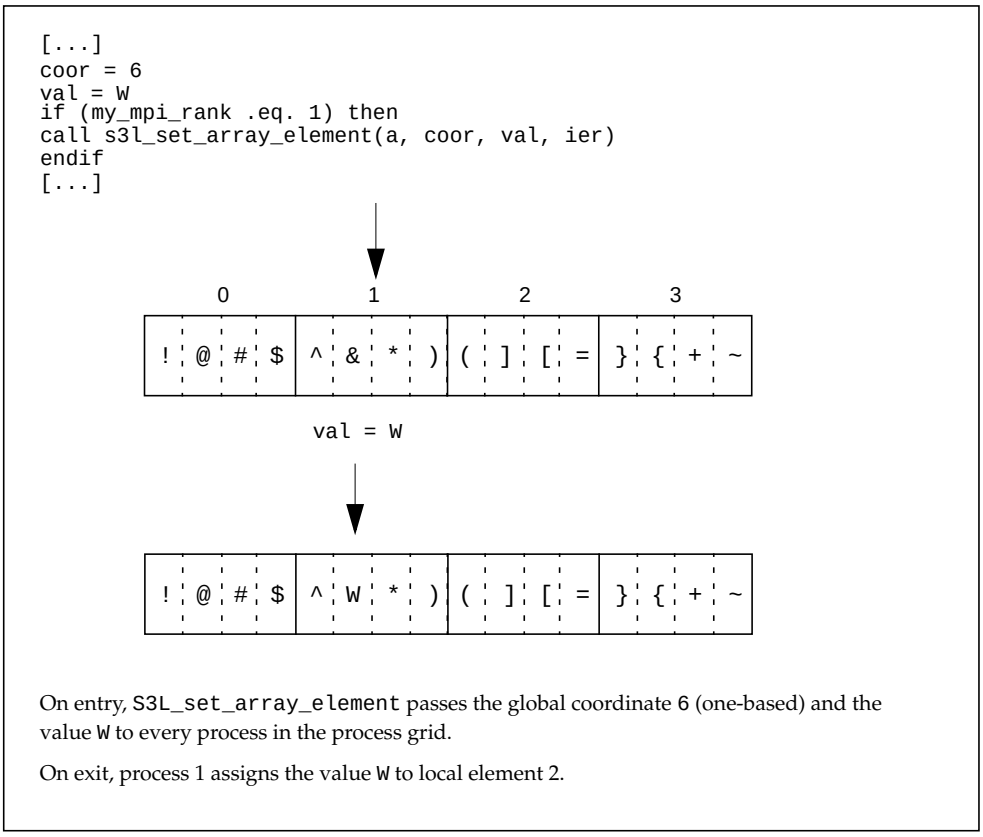
```
S3L_set_array_element(a, coor, val, ier)
```

`a` is an S3L array handle describing the distributed array that contains the element(s) of interest.

`coor` is an integer vector that specifies the global coordinates of one or more elements in the array described by `a`.

`val` is an array variable that contains the value(s) to be assigned to the element(s) specified in `coor`.

Note – If you know the MPI rank of the process that contains the array element to be set, you can limit execution of `S3L_set_array_element` to that process, avoiding the need for communication operations by the other processes. This is illustrated in FIGURE 9-3.



On entry, `S3L_set_array_element` passes the global coordinate `6` (one-based) and the value `W` to every process in the process grid.

On exit, process 1 assigns the value `W` to local element 2.

FIGURE 9-3 Illustration of `S3L_set_array_element` Use When the Element's Process Locality is Known

S3L_get_array_element_on_proc

S3L_get_array_element_on_proc is similar to S3L_get_array_element, except the target element(s) must be local to a specific MPI process. In other words, one or more target elements are specified by their global coordinates in coor and a particular process is specified in the argument pnum. The target process will assign the value(s) of any local array elements specified in coor to the local val variable. FIGURE 9-4 illustrates the use of S3L_get_array_element_on_proc.

The value in pnum is the MPI rank of the process, which is defined in the global communicator, MPI_COMM_WORLD.

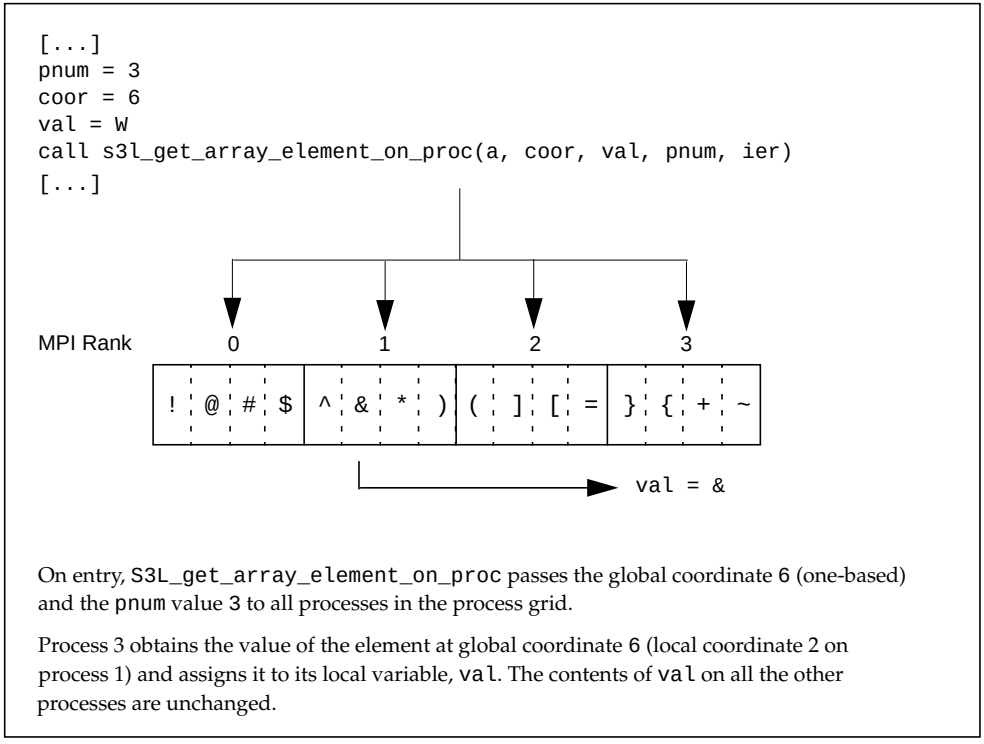


FIGURE 9-4 Illustration of S3L_get_array_element_on_proc for a 1 x 16 S3L Array

S3L_get_array_element_on_proc has the following argument syntax:

```
S3L_get_array_element_on_proc(a, coor, val, pnum, ier)
```

a is an S3L array handle describing the distributed array that contains the element(s) of interest.

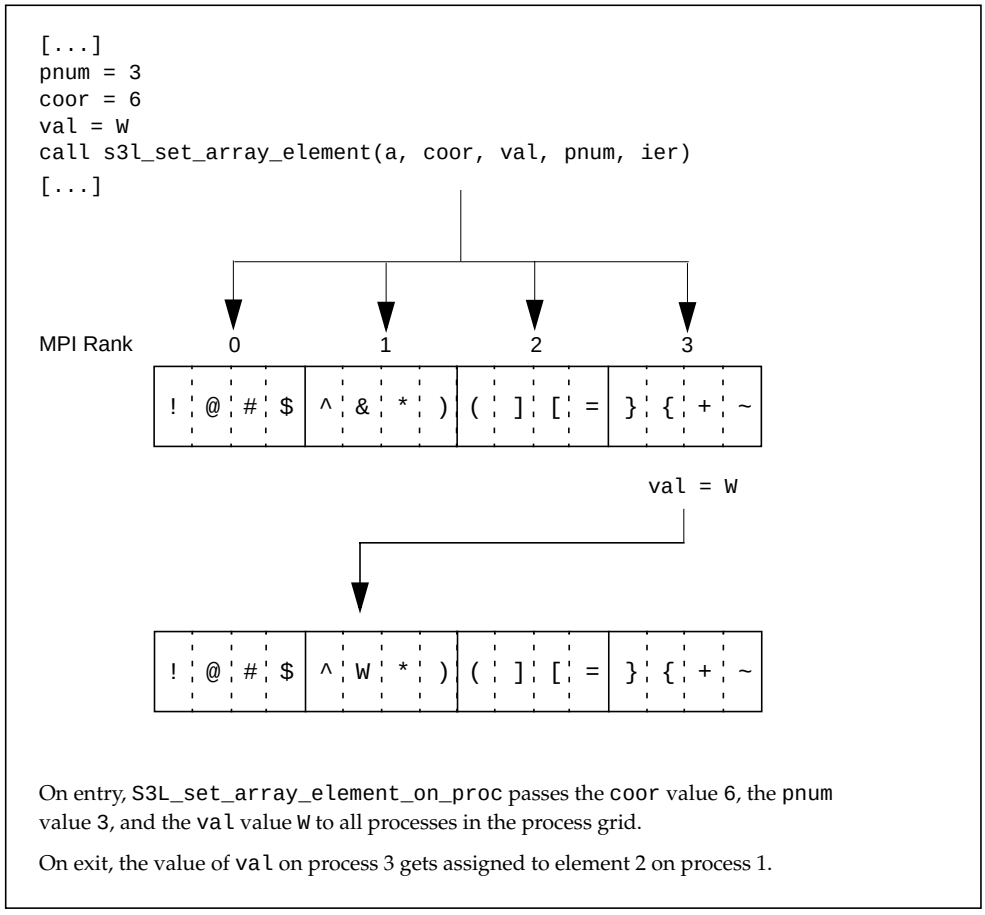
coor is an integer vector that specifies the global coordinates of one or more elements in the array described by a.

`val` is an array variable that contains value(s) obtained from the element(s) that are specified in `coor` and are local to the process whose rank is specified in `pnum`.

`pnum` is an integer variable that specifies the MPI rank of the target process.

S3L_set_array_element_on_proc

Use `S3L_set_array_element_on_proc` to set the value of one or more array elements that are local to a specific process. Pass the global coordinates of the target elements in `coor`, the value(s) to be assigned in `val`, and the MPI rank of the target process in `pnum`. FIGURE 9-5 shows `S3L_set_array_element_on_proc` in use.



On entry, `S3L_set_array_element_on_proc` passes the `coor` value 6, the `pnum` value 3, and the `val` value `W` to all processes in the process grid.

On exit, the value of `val` on process 3 gets assigned to element 2 on process 1.

FIGURE 9-5 Illustration of `S3L_set_array_element_on_proc` for a 1x16 S3L Array

S3L_set_array_element_on_proc has the following argument syntax:

```
S3L_set_array_element_on_proc(a, coor, val, pnum, ier)
```

a is an S3L array handle describing the distributed array that contains the element(s) of interest.

coor is an integer vector that specifies the global coordinates of one or more elements in the array described by a.

val is an array variable that contains the value(s) to be assigned to the element(s) specified in coor that are local to the process whose MPI rank is in pnum.

pnum is an integer variable that specifies the MPI rank of the target process.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for these routines, see the S3L_get_array_element(3), S3L_set_array_element(3), S3L_get_array_element_on_proc(3), and S3L_set_array_element_on_proc(3) man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing the S3L_get_array_element and S3L_set_array_element functions in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/cshift_reduce.c
/opt/SUNWhpc/examples/s3l/utils-f/cshift_reduce.f
/opt/SUNWhpc/examples/s3l/utils/copy_array.c
/opt/SUNWhpc/examples/s3l/utils-f/copy_array.f
```

Examples showing the S3L_get_array_element_on_proc and S3L_set_array_element_on_proc functions in use can be found in:

```
/opt/SUNWhpc/examples/s3l/utils/zero_elements.c
/opt/SUNWhpc/examples/s3l/utils-f/zero_elements.f
```

CHAPTER 10

Dense Matrix Routines

Sun S3L includes support for matrix-matrix and matrix-vector multiplication, inner- and outer-product computation, and 2-norm computation. The routines that support these operations are discussed the following sections:

Overview

The dense matrix routines, like most other Sun S3L routines, can be used in both single-instance or multiple-instance contexts. For example, the matrix-matrix multiplication routine can be used in either of the following ways:

- To simply multiply two 2D arrays
- To multiply many instances of the two arrays, which are embedded in another array of greater dimensionality

In the first case, the operation would be performed on a single process, with both arrays local to that process or on multiple processes, with the two arrays block-distributed across the processes.

In the second case, each instance of the multiplication operation would be performed on a different process, with each process having a pair of instances of the two arrays local to it.

All of the dense matrix routines operate on at least one S3L array, which would ordinarily be created by a call to `S3L_declare` or `S3L_declare_detailed`. See "Creating and Destroying Array Handles for Dense S3L Arrays" on page 57 for information on how to create and deallocate dense S3L arrays.

The balance of this chapter discusses the various Sun S3L dense matrix routines more closely.

Matrix-Matrix Multiplication

Sun S3L provides 18 versions of matrix multiplication routines. These are listed in TABLE 10-1.

TABLE 10-1 S3L Matrix-Matrix Multiplication Operations

Routine	Operation	Data Type
`S3L_mat_mult`	$C = C + AB$	Real or complex
`S3L_mat_mult_noadd`	$C = AB$	Real or complex
`S3L_mat_mult_addto`	$C = D + AB$	Real or complex
`S3L_mat_mult_t1`	$C = C + A^TB$	Real or complex
`S3L_mat_mult_t1_noadd`	$C = A^TB$	Real or complex
`S3L_mat_mult_t1_addto`	$C = D + A^TB$	Real or complex
`S3L_mat_mult_h1`	$C = C + A^HB$	Complex only
`S3L_mat_mult_h1_noadd`	$C = A^HB$	Complex only
`S3L_mat_mult_h1_addto`	$C = D + A^HB$	Complex only
`S3L_mat_mult_t2`	$C = C + AB^T$	Real or complex
`S3L_mat_mult_t2_noadd`	$C = AB^T$	Real or complex
`S3L_mat_mult_t2_addto`	$C = D + AB^T$	Real or complex
`S3L_mat_mult_h2`	$C = C + AB^H$	Complex only
`S3L_mat_mult_h2_noadd`	$C = AB^H$	Complex only
`S3L_mat_mult_h2_addto`	$C = D + AB^H$	Complex only
`S3L_mat_mult_t1_t2`	$C = C + A^TB^T$	Real or complex
`S3L_mat_mult_t1_t2_noadd`	$C = A^TB^T$	Real or complex
`S3L_mat_mult_t1_t2_addto`	$C = D + A^TB^T$	Real or complex

In each routine, two S3L arrays, represented by A and B, are multiplied. A third S3L array, represented by C, will hold the results of the operation. Other aspects of the operation vary from routine to routine as follows:

- Some routines replace the contents of C with the product of A and B. These routine names end with _noadd.
- Other routines add the product of A and B to the contents of a fourth S3L array, represented by D. These routine names end with _addto.
- All other routines add the product of A and B to the contents of C. These routines do not include either _noadd or _addto in their names.
- Some routines take the transpose of one or both operand matrices. This is indicated in the routine names by the strings _t1 and _t2, where:
  - _t1 indicates the transpose of the first factor array (A)
  - _t2 indicates the transpose of the second factor array (B)
- Some routines take the Hermitian of an operand matrix. It must contain complex data. This is indicated in the routine names by the strings _h1 and _h2, which follow the same naming pattern as _t1 and _t2.

The argument syntax for the matrix-matrix multiply routines is summarized below:

```
S3L_mat_mult(A, B, C, row_axis, col_axis, ier)
S3L_mat_mult_noadd(A, B, C, row_axis, col_axis, ier)
S3L_mat_mult_addto(A, B, C, D, row_axis, col_axis, ier)
```

A, B, C, and D are S3L array handles returned by earlier calls to S3L_declare or S3L_declare_detailed.

A and B represent the multiplication operand matrices. C represents the matrix that stores the result of the operation. A, B, and C must all have the same rank.

D is used only in the _addto class of routines, when its contents are added to the product of A and B. D must have the same shape as C.

Note – The argument D can be identical to C in all matrix multiply _addto routines, except t1_t2__addto (both A and B are transposed).

The contents of A and B are not changed in any of the matrix multiply routines. If D is distinct from C, its contents are not changed either. If D and C are the same variable, its contents are overwritten by the result of the matrix multiply operation.

row_axis is a scalar integer that specifies which axis of A, B, C, and D counts the rows of the embedded matrix or matrices. It must be nonnegative and less than the rank of C.

`col_axis` is a scalar integer that specifies which axis of `A`, `B`, `C`, and `D` counts the columns of the embedded matrix or matrices. It must be nonnegative and less than the rank of `C`.

For detailed descriptions of the Fortran and C bindings for the matrix-matrix multiply routines, see the `S3L_mat_mult(3)` man page or the corresponding descriptions in the *Sun S3L Reference Manual*.

For calls that do not transpose either matrix `A` or `B`, the variables conform correctly with the axis lengths for `row_axis` and `col_axis` shown in TABLE 10-2.

TABLE 10-2 Recommended `row_axis` and `col_axis` Values When Matrix `A` and Matrix `B` Are Not Transposed

Variable	`row_axis` Length	`col_axis` Length
A	p	q
B	q	r
C	p	r
D	p	r

For calls that transpose matrix `A` (A^T), the variables conform correctly with the axis lengths for `row_axis` and `col_axis` shown in TABLE 10-3.

TABLE 10-3 Recommended `row_axis` and `col_axis` Values When Matrix `A` Is Transposed

Variable	`row_axis` Length	`col_axis` Length
A	q	p
B	q	r
C	p	r
D	p	r

For calls that transpose matrix B (B^T), the variables conform correctly with the axis lengths for `row_axis` and `col_axis` shown in TABLE 10-4.

TABLE 10-4 Recommended `row_axis` and `col_axis` Values When Matrix B Is Transposed

Variable	`row_axis` Length	`col_axis` Length
A	p	q
B	r	q
C	p	r
D	p	r

For calls that transpose both A and B (A^TB^T), the variables conform correctly with the axis lengths for `row_axis` and `col_axis` shown in TABLE 10-5.

TABLE 10-5 Recommended `row_axis` and `col_axis` Values When Both Matrix A and Matrix B Are Transposed

Variable	`row_axis` Length	`col_axis` Length
A	q	p
B	r	q
C	p	r
D	p	r

A matrix multiply routine will use one of three algorithms, depending on various factors. The three candidate algorithms are:

- Broadcast-Multiply-Roll
- Cannon
- Broadcast-Broadcast-Multiply

Examples showing `S3L_mat_mult` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/matmult.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/matmult.f
```

Matrix-Vector Multiplication

Sun S3L provides six matrix-vector multiplication routines, which compute one or more instances of a matrix-vector product. For each instance, these routines perform the operations listed in TABLE 10-6.

Note – In these descriptions, conj[A] denotes the conjugate of A.

TABLE 10-6 S3L Matrix-Vector Multiplication Operations

Routine	Operation	Data Type
`S3L_mat_vec_mult`	y = y + Ax	Real or complex
`S3L_mat_vec_mult_noadd`	y = Ax	Real or complex
`S3L_mat_vec_mult_addto`	y = v + Ax	Real or complex
`S3L_mat_vec_mult_c1`	y = y + conj[A]x	Complex only
`S3L_mat_vec_mult_c1_noadd`	y = conj[A]x	Complex only
`S3L_mat_vec_mult_c1_addto`	y = v + conj[A]x	Complex only

In each matrix-vector routine, an S3L array, represented by `A`, is multiplied by a vector, represented by `x`. Another S3L array, represented by `y`, holds the results of the matrix-vector operation. Other aspects of the operation vary from routine to routine as follows:

- Some routines replace the contents of `y` with the product of `A` and `x`. Their names end with `_noadd`.
- Other routines add the product of `A` and `x` to the contents of another S3L array, represented by `v`, and replace the contents of `y` with the result. Their names end with `_addto`.
- The remaining routines add the product of `A` and `x` to the contents of `y`. These routines do not include either `_noadd` or `_addto` in their names.
- Some routines take the complex conjugate of `A`. This is indicated in the routine names by the string `_c1`.

The argument syntax for the matrix-vector routines is summarized below:

```
S3L_mat_vec_mult(y, A, x, y_vector_axis, row_axis, col_axis,
x_vector_axis, ier)
S3L_mat_vec_mult_noadd(y, A, x, y_vector_axis, row_axis,
col_axis, x_vector_axis, ier)
S3L_mat_vec_mult_addto(y, A, x, v, y_vector_axis, row_axis,
col_axis, x_vector_axis, ier)
```

y, A, x, and v are S3L array handles returned by earlier calls to `S3L_declare` or `S3L_declare_detailed`.

A and x represent the matrix and vector multiplication operands, respectively. y represents the array that stores the result of the matrix-vector operation.

v is used only in the `_addto` class of routines. Its contents are added to the product of A and x.

Note – The argument v can be identical to y in both routines that have `_addto` in their names.

---

y, A, x, and v must have the following rank and size relationships:

- x and y must have the same rank, which can be one or greater.
- The rank of A must be one greater than the rank of y.
- The instance axis of A must match the instance axis of y in length and order of declaration. This means, each matrix instance in A corresponds to a vector in y.
- v has the same rank and shape as y.

`y_vector_axis` is a scalar integer that specifies the axis of y and v along which the elements of the embedded vectors lie.

`row_axis` is a scalar integer that specifies which axis of y, A, x, and v counts the rows of the embedded matrix or matrices. It must be nonnegative and less than the rank of A.

`col_axis` is a scalar integer that specifies which axis of y, A, x, and v counts the columns of the embedded matrix or matrices. It must be nonnegative and less than the rank of A.

`x_vector_axis` is a scalar integer that specifies the axis of x along which the elements of the embedded vectors lie.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the matrix-vector multiply routines, see the `S3L_mat_vec_mult(3)` man page or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing S3L_mat_vec_mult in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/mat_vec_mult.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/matvec_mult.f
```

2-Norm Operations

The multiple-instance 2-norm routine, S3L_2_norm, computes one or more instances of the 2-norm of a vector. The single-instance 2-norm routine, S3L_gbl_2_norm, computes the global 2-norm of a parallel array.

For each instance z of z, the multiple-instance 2-norm routine performs one of the operations shown in TABLE 10-7.

TABLE 10-7 S3L Multiple-Instance 2-norm Operations

Operation	Data Type
$z = (x^T x)^{1/2} = \|\|x\|\|(2)$	Real
$z = (x^H x)^{1/2} = \|\|x\|\|(2)$	Complex

Upon successful completion, S3L_2_norm overwrites each element of z with the 2-norm of the corresponding vector in x.

The single-instance 2-norm routine performs the operations shown in TABLE 10-8.

TABLE 10-8 S3L Single-Instance 2-norm Operations

Operation	Data Type
$a = (x^T x)^{1/2} = \|\|x\|\|(2)$	Real
$a = (x^H x)^{1/2} = \|\|x\|\|(2)$	Complex

Upon successful completion, S3L_gbl_2_norm overwrites a with the global 2-norm of x.

The argument syntax for the single- and multiple-instance 2-norm routines are summarized below:

```
S3L_gbl_2_norm(a, x, ier)
S3L_2_norm(z, x, x_vector_axis, ier)
```

x and z are S3L array handles returned by earlier calls to S3L_declare or S3L_declare_detailed.

x represents a parallel array of rank 2 or greater and at least one nonlocal instance axis. It contains one or more instances of the vector x whose 2-norm will be computed.

z represents a parallel array that will contain the results of the multiple-instance 2-norm operation. Its rank must be one less than that of x.

a is a pointer to a scalar variable, which is the destination for the results of the single-instance 2-norm operation.

x_vector_axis is a scalar integer that specifies the axis of x along which the vectors lie.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for the 2-norm routine, see the S3L_2_norm(3) man page or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing S3L_2_norm in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/norm2.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/norm2.f
```

Inner-Product Operations

Sun S3L provides six multiple-instance inner-product routines, all of which compute one or more instances of the inner product of two vectors embedded in two parallel arrays. It also provides six single-instance inner product routines, all of which compute the inner product over all the axes of two parallel arrays.

The two sets of inner-product routines are discussed separately below.

Multiple-Instance Inner-Product Routines

The operations performed by the inner product routines are listed in TABLE 10-9.

TABLE 10-9 S3L Multiple-Instance Inner-Product Operations

Routine	Operation	Data Type
`S3L_inner_prod`	$z = z + x^T y$	Real or complex
`S3L_inner_prod_noadd`	$z = x^T y$	Real or complex
`S3L_inner_prod_addto`	$z = u + x^T y$	Real or complex
`S3L_inner_prod_c1`	$z = z + x^H y$	Complex only
`S3L_inner_prod_c1_noadd`	$z = x^H y$	Complex only
`S3L_inner_prod_c1_addto`	$z = u + x^H y$	Complex only

For each multiple-instance inner-product routine, array `x` contains one or more instances of the first vector in each inner-product pair, `x`. Likewise, array `y` contains one or more instances of the second vector in each pair, `y`.

In each multiple-instance inner-product routine, the inner products are computed for vectors embedded in two S3L arrays, represented by `x` and `y`. Another S3L array, represented by `z`, holds the results of the inner-product operation. Other aspects of the operation vary from routine to routine as follows:

- Some routines replace the contents of `z` with the inner products of `x` and `y`. Their names end with `_noadd`.
- Other routines add the inner-product results of `x` and `y` to the contents of another S3L array, represented by `u` and replace the contents of `z` with the result. Their names end with `_addto`.
- The remaining routines add the inner product of `x` and `y` to the contents of `z`. These routines do not include either `_noadd` or `_addto` in their names.
- Three routines take the transpose of the `x` array. Their names do not contain any special indication of this.
- The other three routines take the Hermitian of `x`, which must contain complex data. This is indicated in the routine names by the string `_c1`.

The argument syntax for the multiple-instance inner-product routines is summarized below:

```
S3L_inner_prod(z, x, y, x_vector_axis, y_vector_axis, ier)
S3L_inner_prod_noadd(z, x, y, x_vector_axis, y_vector_axis, ier)
S3L_inner_prod_addto(z, x, y, u, x_vector_axis, y_vector_axis,
ier)
```

z, x, y, and u are S3L array handles returned by earlier calls to S3L_declare or S3L_declare_detailed.

x and y represent the S3L arrays that contain the vector pairs from which the inner products will be computed. z represents the array that stores the results of the multiple-instance inner-product operations.

For some multiple-instance inner-product operations, the inner-product results are added to the contents of z. In other operations, the inner-product results simply replace the contents of z.

u is used only in the _addto class of routines. Its contents are added to the inner-product results computed from x and y.

z, x, y, and u must have the following rank and size relationships:

- x and y must be at least rank 1 arrays, must be of the same rank, and their corresponding axes must have the same extents. Additionally, x and y must both be distributed arrays—that is, each must have at least one axis that is nonlocal.
- Array z, which stores the results of the multiple-instance inner-product operations, must be of rank one less than that of x and y. Its axes must match the instance axes of x and y in length and order of declaration, and it must also have at least one axis that is nonlocal. This means each vector pair in x and y corresponds to a single destination value in z.
- Finally, x, y, and z must match in data type and precision.

x_vector_axis is a scalar integer that specifies the axis of x along which the elements of the embedded vectors lie.

y_vector_axis is a scalar integer that specifies the axis of y along which the elements of the embedded vectors lie.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for the multiple-instance inner-product routines, see the S3L_inner_prod(3) man page or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing S3L_inner_prod in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/inner_prod.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/inner_prod.f
```

Note – If each instance axis of x and y—that is, the axes along which the inner product will be taken—contains only a single vector, either declare the axes to have an extent of 1 or use the comparable single-instance inner-product routine, as described below.

Single-Instance Inner-Product Routines

The operations performed by the single-instance inner-product routines are listed in TABLE 10-10.

TABLE 10-10 S3L Single-Instance Inner-Product Operations

Routine	Operation	Data Type
`S3L_gbl_inner_prod`	$a = a + x^T y$	Real or complex
`S3L_gbl_inner_prod_noadd`	$a = x^T y$	Real or complex
`S3L_gbl_inner_prod_addto`	$a = b + x^T y$	Real or complex
`S3L_gbl_inner_prod_c1`	$a = a + x^H y$	Complex only
`S3L_gbl_inner_prod_c1_noadd`	$a = x^H y$	Complex only
`S3L_gbl_inner_prod_c1_addto`	$a = b + x^H y$	Complex only

The argument syntax for the single-instance inner-product routines is summarized below:

```
S3L_gbl_inner_prod(a, x, y, ier)
S3L_gbl_inner_prod_noadd(a, x, y, ier)
S3L_gbl_inner_prod_addto(a, x, y, b, ier)
```

`x` and `y` are S3L array handles returned by earlier calls to `S3L_declare` or `S3L_declare_detailed`. They represent the S3L arrays containing the vector pairs from which the inner-products will be computed.

`a` is a pointer to a scalar variable that is the destination for the results of the single-instance inner-product operations. For `S3L_gbl_inner_prod` and `S3L_gbl_inner_prod_c1`, `a` is also a source of values to be added to the inner products of `x` and `y`.

`b` is also a pointer to a scalar variable. It is used only in the `_addto` class of routines. Its contents are added to the inner-product results computed from `x` and `y`.

For detailed descriptions of the Fortran and C bindings for the single-instance inner-product routines, see the `S3L_inner_prod(3)` man page or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_inner_prod` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/inner_prod.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/inner_prod.f
```

Outer-Product Operations

Sun S3L provides six outer-product routines that compute one or more instances of an outer product of two vectors. For each instance, the outer-product routines perform the operations listed in TABLE 10-11.

Note – In these descriptions, y^T and y^H denote y transpose and y Hermitian, respectively.

TABLE 10-11 S3L Outer-Product Operations

Routine	Operation	Data Type
`S3L_outer_prod`	$A = A + xy^T$	real or complex
`S3L_outer_prod_noadd`	$A = xy^T$	real or complex
`S3L_outer_prod_addto`	$A = B + xy^T$	real or complex
`S3L_outer_prod_c2`	$A = A + xy^H$	complex only
`S3L_outer_prod_c2_noadd`	$A = xy^T$	complex only
`S3L_outer_prod_c2_noadd`	$A = B + xy^T$	complex only

In elementwise notation, for each instance `S3L_outer_prod` computes:

```
A(i,j) = A(i,j) + x(i) * y(j)
```

and `S3L_outer_prod_c2` computes

```
A(i,j) = A(i,j) + x(i) * conj[y(j)]
```

where conj[y(j)] denotes the conjugate of y(j).

The argument syntax for the outer-product routines is summarized below:

```
S3L_outer_prod(A, x, y, row_axis, col_axis, x_vector_axis,
y_vector_axis, ier)
S3L_outer_prod_noadd(A, x, y, row_axis, col_axis, x_vector_axis,
y_vector_axis, ier)
S3L_outer_prod_addto(A, x, y, B, row_axis, col_axis,
x_vector_axis, y_vector_axis, ier)
```

`A`, `x`, `y`, and `B` are S3L array handles returned by earlier calls to `S3L_declare` or `S3L_declare_detailed`.

x and y represent the S3L arrays that contain the vector pairs from which the inner-products will be computed. A represents the array that stores the results of the outer-product operations.

x contains one or more instances of the first source vector, x, embedded along the axis specified by axis x_vector_axis (see below).

y contains one or more instances of the second source vector, y, embedded along the axis specified by y_vector_axis (see below).

B is used only in the _addto class of routines. Its contents are added to the outer products computed from x and y.

A, x, y, and B must conform to the following rank and size relationships:

- A must be of rank 2 or greater.
- The rank of x and y must be one less than the rank of A.
- Array z, which stores the results of the multiple-instance inner-product operations, must be of rank one less than that of x and y. Its axes must match the instance axes of x and y in length and order of declaration. It must also have at least one axis that is nonlocal. This means each vector pair in x and y corresponds to a single destination value in z.
- Finally, x, y, and z must match in data type and precision.

row_axis is a scalar integer that specifies which axis of A and B counts the rows of the embedded matrix or matrices. It must be nonnegative and less than the rank of A.

col_axis is a scalar integer that specifies which axis of A and B counts the columns of the embedded matrix or matrices. It must be nonnegative and less than the rank of A.

x_vector_axis is a scalar integer that specifies the axis of x along which the elements of the embedded vectors lie.

y_vector_axis is a scalar integer that specifies the axis of y along which the elements of the embedded vectors lie.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for the outer-product routines, see the S3L_outer_prod(3) man page or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing S3L_outer_prod in use can be found in:

```
/opt/SUNWhpc/examples/s3l/dense_matrix_ops/outer_prod.c
/opt/SUNWhpc/examples/s3l/dense_matrix_ops-f/outer_prod.f
```

CHAPTER 11

General Linear Systems Solvers

Sun S3L includes routines that provide solutions to linear systems equations for real and complex general matrices. Both Gaussian elimination and Householder transformation methods are supported. These routines are discussed in the following sections:

Gaussian Elimination for Dense Systems

LU Factor Routine

`S3L_lu_factor` is used to decompose one or more matrices, `A`, into their LU factors using Gaussian elimination with partial pivoting. The resulting factors can then be used by `S3L_lu_solve` to solve the linear system `Ax = b` or by `S3L_lu_invert` to compute the inverse of `A`.

The LU factorization routine uses a parallel, block-partitioned algorithm based on the ScaLAPACK implementation. `S3L_lu_factor` uses an efficient pivoting scheme that involves fewer interprocess communication steps. Nodal computation makes use of the underlying Forte Developer 6 routines, chiefly for matrix multiplication operations.

For each M x N coefficient matrix `A` of `a`, `S3L_lu_factor` computes the LU factorization using partial pivoting with row interchanges.

The factorization has the form `A` = P x L x U, where P is a permutation matrix, L is the lower triangular with unit diagonal elements (lower trapezoidal if M > N), and U is the upper triangular (upper trapezoidal if M < N). L and U are stored in `A`.

In general, S3L_lu_factor performs most efficiently when the array is distributed using the same block size along each axis.

S3L_lu_factor behaves somewhat differently for 3D arrays, however. In this case, it applies nodal LU factorization on each M x N coefficient matrix across the instance axis. This factorization is performed concurrently on all participating processes.

You must call S3L_lu_factor before calling any of the other LU routines. The S3L_lu_factor routine performs on the preallocated parallel array and returns a setup ID. You must supply this setup ID in subsequent LU calls, as long as you are working with the same set of factors.

S3L_lu_factor has the following argument syntax:

```
S3L_lu_factor(a, row_axis, col_axis, setup_id, ier)
```

a is an S3L array handle returned by an earlier call to S3L_declare or S3L_declare_detailed. The array it represents contains one or more instances of the coefficient matrix A that is to be factored. Each coefficient matrix A is assumed to be dense, with dimensions M x N.

row_axis is a scalar integer that specifies which axis of a counts the rows of each coefficient matrix A.

col_axis is a scalar integer that specifies which axis of a counts the columns of each coefficient matrix A.

Note – row_axis and col_axis must not be equal.

---

setup_id is a scalar integer returned by a call to S3L_lu_factor. It can be used in calls to other LU routines to reference the computed LU factors.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for the LU factor routine, see the S3L_lu_factor(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_lu_factor in use can be found in:

```
/opt/SUNWhpc/examples/s3l/lu/lu.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu1.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu2.c
/opt/SUNWhpc/examples/s3l/lu-f/lu.f
/opt/SUNWhpc/examples/s3l/lu-f/ex_lu1.f
```

LU Solve Routine

For each square coefficient matrix A of the parallel a, S3L_lu_solve uses the LU factors computed by a previous call to S3L_lu_factor to solve a system of distributed linear equations AX = B.

Note – Throughout these descriptions, L^{-1} and U^{-1} denote the inverse of L and U, respectively.

A and B are corresponding instances within a and b, respectively. To solve AX = B, S3L_lu_solve performs forward elimination:

```
Let UX = C
A = LU implies that AX = B is equivalent to C = L-1B
```

followed by back substitution:

```
X = U-1C = U-1(L-1B)
```

To obtain this solution, the S3L_lu_solve routine performs the following steps:

- Applies L^{-1} to B.
- Applies U^{-1} to $L^{-1}B$.

Upon successful completion, each B is overwritten with the solution to AX = B.

In general, S3L_lu_solve performs most efficiently when the array is distributed using the same block size along each axis.

S3L_lu_solve behaves somewhat differently for 3D arrays, however. In this case, the nodal solve is applied on each of the 2D systems AX = B across the instance axis of a and is performed concurrently on all participating processes.

The input parallel arrays a and b must be distinct.

S3L_lu_solve has the following argument syntax:

```
S3L_lu_solve(b, a, setup_id, ier)
```

a and b are S3L array handles returned by earlier calls to S3L_declare or S3L_declare_detailed. They must be of the same type (real or complex) and precision.

a represents the parallel array that was factored by a previous call to S3L_lu_factor. Each coefficient matrix A in a is a dense, square (M x M) matrix. Use the same value of a that was used in the S3L_lu_factor call.

The instance axes of b must match those of a in order of declaration and extents.

Note – For the 2D case, if `b` consists of only one right-hand-side vector, it can be represented either as a vector or as an array of rank 2, with the number of columns set to 1. If it is represented as a rank 2 array, its elements will be counted along the axis specified in the argument `row_axis`. See below.

`setup_id` is the integer returned by the earlier call to `S3L_lu_factor` that computed the LU factors for array `a`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the LU solve routine, see the `S3L_lu_solve(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_lu_solve` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/lu/lu.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu1.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu2.c
/opt/SUNWhpc/examples/s3l/lu-f/lu.f
/opt/SUNWhpc/examples/s3l/lu-f/ex_lu1.f
```

LU Invert Routine

`S3L_lu_invert` uses the LU factorization generated by `S3L_lu_factor` to compute the inverse of each square (M x M) matrix instance `A` of the parallel array `a`. It does this by inverting U and then solving the system $A^{-1}L = U^{-1}$ for A^{-1}, where A^{-1} and U^{-1} denote the inverse of A and U, respectively.

For arrays with rank > 2, the nodal inversion is applied on each 2D slice of `a` across the instance axis and is performed concurrently on all participating processes.

`S3L_lu_invert` has the following argument syntax:

```
S3L_lu_invert(a, setup_id, ier)
```

`a` is an S3L array handle returned by an earlier call to `S3L_declare` or `S3L_declare_detailed`. Use the same value of `a` that was used in the `S3L_lu_factor` call.

`setup_id` is the integer returned by the earlier call to `S3L_lu_factor`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the LU invert routine, see the `S3L_lu_invert(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_lu_invert` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/lu/lu.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu1.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu2.c
/opt/SUNWhpc/examples/s3l/lu-f/lu.f
/opt/SUNWhpc/examples/s3l/lu-f/ex_lu1.f
```

LU Deallocate Routine

`S3L_lu_deallocate` frees up the memory allocated to the LU factored array that is associated with a `setup_id` returned by a previous `S3L_lu_factor` call.

`S3L_lu_deallocate` has the following argument syntax:

```
S3L_lu_deallocate(setup_id, ier)
```

`setup_id` is the integer returned by the earlier call to `S3L_lu_factor`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the LU deallocation routine, see the `S3L_lu_deallocate(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_lu_deallocate` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/lu/lu.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu1.c
/opt/SUNWhpc/examples/s3l/lu/ex_lu2.c
/opt/SUNWhpc/examples/s3l/lu-f/lu.f
/opt/SUNWhpc/examples/s3l/lu-f/ex_lu1.f
```

Householder Transformations

Computing QR Decomposition of S3L Arrays

`S3L_qr_factor` computes the QR decomposition of a real or complex S3L array. On exit, the Q and R factors are packed in array `a`.

`S3L_qr_factor` generates internal information related to the decomposition, such as the vector of elementary reflectors. It also returns a setup parameter, which can be used by subsequent calls to `S3L_qr_solve` to compute the least-squares solution to the system a*x = b, where a is an m x n array, with m > n, and b is an m x nrhs array.

`S3L_qr_factor` can be used for arrays with more than two dimensions. In such cases, the `axis_r` and `axis_c` arguments specify the row and column axes of 2D array slices whose QR factorization is to be computed.

When `a` is a 2D array, `axis_r` and `axis_c` should be set in the following manner:

QR factorization of	C/C++		F77/F90	
	axis_r	axis_c	axis_r	axis_c
a	0	1	1	2
transpose of a	1	0	2	1

Notes

`S3L_qr_factor` is more efficient when both dimensions of the input array are distributed block cyclically, using equal block sizes.

If least-squares solutions are to be found for multiple a*x = b systems, where all systems have the same matrix, the same QR factorization setup can be used by all the `S3L_qr_solve` instances. In other words, only one call to `S3l_qr_setup` is needed to support multiple QR solve operations so long as the matrix is the same in every case.

`S3L_qr_factor` has the following argument syntax:

```
S3L_qr_factor(a, axis_r, axis_c, setup, ier)
```

`a` is an S3L array handle for an array whose QR decomposition is to be computed. On exit, the contents of the array described by `a` are destroyed.

axis_r is an integer that is used to specify which axis will be treated as the row axis.

axis_c is an integer that is used to specify which axis will be treated as the column axis.

setup is an integer value returned by S3L_qr_factor on exit. This value is a unique identifier for the QR decomposition results and can be used by subsequent calls to S3L_qr_solve and S3L_get_qr.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_qr_factor(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_qr_factor in use can be found in:

```
/opt/SUNWhpc/examples/s3l/qr/ex_qr1.c
/opt/SUNWhpc/examples/s3l/qr-f/ex_qr1.f
```

Finding the Least-Squares Solution for a QR-Decomposed Array

S3L_qr_solve computes the least-squares solution to an overdetermined linear system of the form a*x = b. a is an m x n array, where m > n (overdetermined) and b is an m x nrhs array of the same type as a.

S3L_qr_solve uses the QR factorization results from a previous call to S3L_qr_factor for the computation. On exit, the first n x nrhs rows of b are overwritten with the least-squares solution of the system.

If a and b have more than two dimensions, the operation is performed in 2D slices over all of the arrays. These slices were specified by the row and column axis arguments, axis_r and axis_c, of the earlier S3L_qr_factor call.

Notes

For m > n, the single routine S3L_gen_lsq performs the same set of operations as the sequence: S3L_qr_factor, S3L_qr_solve, S3L_qr_free. However, if multiple least-squares solutions are to be found for a set of matrices that are all the same, the explicit sequence can be more efficient. This is because S3L_gen_lsq performs the full sequence every time it is called, even though the QR factorization step is needed only the first time.

In such cases therefore, the following sequence can be used to eliminate redundant factorization operations:

- `S3L_qr_factor`, `S3L_qr_solve`, `S3L_get_qr` for the first solution
- `S3L_qr_solve`, `S3L_get_qr` for the second and all subsequent solutions

`S3L_qr_solve` has the following argument syntax:

```
S3L_qr_solve(a, b, setup, ier)
```

`a` is an S3L array handle that describes an array containing a QR decomposition that was computed by an earlier call to `S3L_qr_factor`.

`b` is an S3L array handle for the array that, on exit, contains the solution to the least-squares problem in its first *n* rows.

`setup` is an integer value that was returned by an earlier call to `S3L_qr_factor`. It represents the internal QR factorization results from that QR decomposition.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_qr-solve(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_qr_solve` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/qr/ex_qr1.c
/opt/SUNWhpc/examples/s3l/qr-f/ex_qr1.f
```

Obtaining Q and R Arrays

`S3L_get_qr` extracts the Q and R arrays from the packed representation of a QR-decomposed S3L array. If S3L array `a` is of size m x n, the array `q` should be m x min(m,n) and `r` should be min(m,n) x n.

If either `q` or `r` is zero, it is assumed that the extraction of the corresponding array is not desired. `q` and `r` should not both be zero.

Arrays `a`, `q`, and `r` should all be of the same rank and be of the same data type.

If `a` has more than two dimensions, the QR factorization will have been performed in 2D slices, which were defined by the `S3L_qr_factor` arguments `axis_r` and `axis_c`. These axis numbers are included in the internal QR setup information referred to by the `setup` parameter.

The dimensions of `q` and `r` should have the appropriate lengths along `axis_r` and `axis_c`, as described for the 2D case. In addition, all other dimensions should have the same lengths as those of `a`.

S3L_get_qr has the following argument syntax:

```
S3L_get_qr(a, q, r, setup, ier)
```

a is an S3L array handle that describes an array containing a QR decomposition that was computed by an earlier call to S3L_qr_factor.

q is an S3L array handle for the array that, on exit, contains the orthonormal array produced by the QR decomposition.

r is an S3L array handle for the array that, on exit, contains an upper triangular array.

setup is an integer value that was returned by an earlier call to S3L_qr_factor. It represents the internal QR factorization results from that QR decomposition.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_get_qr(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_get_qr in use can be found in:

```
/opt/SUNWhpc/examples/s3l/qr/ex_qr1.c
/opt/SUNWhpc/examples/s3l/qr-f/ex_qr1.f
```

Freeing QR Factors

S3L_qr_free frees all internal resources associated with a particular QR factorization operation.

S3L_qr_free has the following argument syntax:

```
S3L_qr_free(setup, ier)
```

setup is an integer value that represents the internal QR factorization data structures that are to be freed.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_qr_free(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_qr_free in use can be found in:

```
/opt/SUNWhpc/examples/s3l/qr/ex_qr1.c
/opt/SUNWhpc/examples/s3l/qr-f/ex_qr1.f
```

CHAPTER 12

Basic Sparse Matrix Routines

This section describes various Sun S3L routines that perform fundamental linear algebra operations on sparse matrices. They are described in the following sections:

All of the sparse matrix functions work with both real and complex data types.

Supported Sparse Formats

A matrix is considered to be sparse if special techniques can be used to take advantage of the structure of the matrix (that is, the nonzero elements and their locations).

Many different ways of storing sparse matrices have been devised to realize these advantages, which can mean better efficiency in both memory use and arithmetic operations. The differences among these sparse storage schemes are mostly reflections of such issues as the amount of storage required, the degree of indirect addressing necessary to perform the kernel operations (such as matrix-vector product), and the suitability for a range of different processor architectures.

The Sun S3L sparse routines support the following widely used sparse formats:

- Coordinate (COO)
- Compressed Sparse Row (CSR)

- Compressed Sparse Column (CSC)
- Variable Block Row (VBR)

Note – `S3L_declare_sparse` does not support the Variable Block Row format directly, but `S3L_convert_sparse` can be used to convert sparse matrix data from Variable Block Row format to one of the other formats, which are all compatible with `S3L_declare_sparse`.

Coordinate Format

The Coordinate format consists of the following three arrays:

- `indx` – Integer array that contains the row indices of the sparse matrix `A`. `indx` receives its contents from the argument `row`.
- `jndx` – Integer array that contains the column indices of the matrix `A`. `jndx` receives its contents from the argument `col`.
- `val` – Floating-point array that stores the nonzero elements of the sparse matrix in any order. `val` receives its contents from the argument `val`.

To illustrate the Coordinate format, consider the following sample 4 x 6 sparse matrix:

```
  3.14       0          0         20.04       0          0
  0         27          0          0         -0.6        0
  0          0         -0.01       0          0          0
 -0.031      0          0          0.08       0        314.0
```

Using zero-based indexing, the contents of the `indx`, `jndx`, and `val` arrays might be as follows:

```
indx = (    3,    1,    0,     3,     2,    0,     1,    3 ),
jndx = (    5,    1,    3,     3,     2,    0,     4,    0 ),
val  = ( 314.0, 27.0, 20.04,  0.08, -0.01, 3.14, –0.6, -0.031 )
```

For example, the row and column indices 0 and 3 show that the value 20.04 belongs in the first row and fourth column of the sparse matrix.

Compressed Sparse Row Format

The Compressed Sparse Row format stores the sparse matrix values in the following three arrays:

- `ptr` – Integer array that contains pointers to the beginning of each row in `indx` and `val`. `ptr` receives its contents from the argument `row`.
- `indx` – Integer array that contains the column indices of the nonzero elements in `val`. `indx` receives its contents from the argument `col`.
- `val` – Floating-point array that stores the nonzero elements of the sparse matrix. `val` receives its contents from the argument `val`.

The contents of the `ptr`, `indx`, and `val` arrays might be as follows:

```
ptr  = ( 0, 2, 4, 5, 8 ),
indx = (  0,     3,     1,    4,    2,     0,     3,      5 ),
val  = ( 3.14, 20.04, 27.0, -0.6, -0.01, -0.031, 0.08, 314.0 )
```

For example, `ptr[1]` = `2` indicates that the first nonzero element in row 1 is stored in `val[2]` (= `val[ptr[1]]`), which is `27.0`.

Compressed Sparse Column Format

The Compressed Sparse Column format also stores the sparse matrix in three arrays, but the pointer and index references swap axes. In other words, the Compressed Sparse Column format can be viewed as the Compressed Sparse Row format for the transpose of the sparse matrix. In the Compressed Sparse Column format, the three internal arrays are:

- `ptr` – Integer array that contains pointers to the beginning of each column in `indx` and `val`. `ptr` receives its contents from the argument `row`.
- `indx` – Integer array that contains the row indices of the nonzero elements in `val`. `indx` receives its contents from the argument `col`.
- `val` – Floating-point array that stores the nonzero elements of the sparse matrix. `val` receives its contents from the argument `val`.

This matrix-transpose relationship can be seen by comparing the following values in the `ptr` and `indx` arrays with the corresponding arrays in the Compressed Sparse Row example:

```
ptr  = ( 0, 2, 3, 4, 6, 7, 8 ),
indx = (  0,     3,      1,    2,     0,    3,     1,     3 ),
val  = ( 3.14, -0.031, 27.0, -0.01, 20.04, 0.08, -0.6, 314.0 )
```

Note that, in the Compressed Sparse Column format, the nonzero elements in `val` are stored column by column, instead of row by row, as in the Compressed Sparse Column format.

For example, `ptr[5]` = `7` means that the first nonzero element of column 5 is stored in `val[7]` (= `val[ptr[5]]`), which is `314.0`, and its row index is stored in `indx[7]` (= `indx[ptr[5]]`), which is `3`.

Variable Block Row Format

The first three sparse matrix formats all provide natural layouts for point sparse matrices. However, for matrices with nonzero elements clustered in blocks, Variable Block Row (VBR) format offers a more efficient representation. For any block-structured matrices, such as those derived from a discretized partial differential equation, using VBR format can reduce the amount of integer storage, and the block representation of the matrix enables the numerical algorithms to perform the kernel matrix operations more efficiently on the block entries.

The data structure of the VBR format consists of the following six arrays:

`rptr`	Integer array. It contains the block row partitioning information—that is, the first row number of each block row.
`cptr`	Integer array. It contains the block column partitioning information—that is, the first column number of each block column.
`val`	Scalar array. It contains the block entries of the matrix.
`indx`	Integer array. It contains the pointers to the beginning of each block entry stored in `val`.
`bindx`	Integer array. It contains the block column indices of block entries of the matrix.
`bptr`	Integer array. It contains the pointers to the beginning of each block row in `bindx` and `val`.

To illustrate the Variable Block Row data layout, consider the following 6 x 8 sparse matrix with a variable block partitioning:

```
    0  1    2  3  4    5    6  7  8
   +------+---------+----+-------+
 0 | 1  2 |         |  3 |       |
 1 | 4  5 |         |  6 |       |
   +------+---------+----+-------+
 2 |      | 7  8  9 | 10 |       |
   +------+---------+----+-------+
 3 |      |         | 11 | 12 13 |
 4 |      |         | 14 | 15 16 |
 5 |      |         | 17 | 18 19 |
   +------+---------+----+-------+
 6
```

Using zero-based indexing, the matrix could be stored in VBR format, as follows:

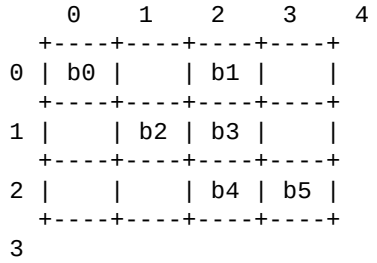
```
rptr  = (0, 2, 3, 6),
cptr  = (0, 2, 5, 6, 8),
bptr  = (0, 2, 4, 6),
bindx = (0, 2, 1, 2, 2, 3),
```

```
indx  = (0, 4, 6, 9, 10, 13, 19)
val   = (1.0, 4.0, 2.0, 5.0, 3.0, 6.0, 7.0, 8.0, 9.0,
         10.0, 11.0, 14.0, 17.0, 12.0, 15.0, 18.0,
         13.0, 16.0, 19.0)
```

In array `rptr`, `0`, `2`, `3`, and `6` are pointers to the boundaries of the block rows. Likewise in `cptr`, `0`, `2`, `5`, `6`, and `8` are pointers to the boundaries of the block columns.

In array `bptr`, `0`, `2`, `4`, and `6` are pointers to the location in `bindx` of the first nonzero block entry of each block row.

These block-based pointers are illustrated in the following figure, which represents the block structure of the original 6 x 8 sparse matrix. It shows the first block row with two nonzero blocks, one in block column 0 and the other in block column 2. The next nonzero block is at block row 1 and block column 1, and so forth. Block 6 is the outer boundary of the block rows.



In array `bindx`, `0`, `2`, `1`, `2`, `2`, and `3` are block column indices for each of the six nonzero block entries.

In array `indx`, `0`, `4`, `6`, `9`, `10`, `13`, and `19` point to the locations in `val` of the first nonzero block entry from each block row.

The last array, `val`, stores nonzero blocks `b0`, `b1`, ..., `b5` block by block with each block stored as a dense matrix in standard column-by-column form. Moreover, the starting location in `val` where the first element of each block gets stored is indexed by array `indx`.

The VBR data structure can be understood by analyzing the representation of block row 1 for example.

First, `bptr[1]` = `2` indicates that `b2`, the first nonzero block from block row 1 is from block column 1, as indicated by `bindx[2]` = `bindx[bptr[1]]` = `1`.

Second, `bptr[1]` = `2` also indexes into `indx`. That is, `indx[bptr[1]]` = `indx[2]` = `6` points to `val[6]` = `val[indx[bptr[1]]` = `val[indx[2]]`), where `6` is the location in `val` at which the first element of `b2`, `7.0`, is stored.

The next nonzero block in block row 1 is `b3`, its block column index is 2, as indicated by `bindx[bptr[1]+1]` = `bindx[3]` = `2`, and the first element of block `b3` is stored in `val[9]` (= `val[indx[bptr[1]+1]]` = `val[indx[3]]`), which is `10.0`.

Declaring a Sparse Matrix

The Sun S3L routine `S3L_declare_sparse` can be used to create a sparse matrix S3L array handle that describes an array that conforms to the Coordinate, Compressed Sparse Row, or Compressed Sparse Column format.

Note – A method for using `S3L_declare_sparse` to create an S3L array handle for an array with a Variable Block Row format is described later in "Converting a Sparse Matrix From One Format to Another" on page 123.

`S3L_declare_sparse` has the following argument syntax:

```
S3L_declare_sparse(A, spfmt, m, n, row, col, val, ier)
```

Upon exit, `A` contains an S3L array handle for the global general sparse matrix. This handle can be used in subsequent calls to other S3L sparse array functions.

`spfmt` indicates which sparse format is to be used in representing the sparse matrix. Its value can be any one of:

- `S3L_SPARSE_COO`
- `S3L_SPARSE_CSR`
- `S3L_SPARSE_CSC`

`m` indicates the number of rows in the sparse matrix.

`n` indicates the number of columns in the sparse matrix.

`row` is an integer parallel array of rank 1. Its length and content can vary, depending on which sparse format is used:

- `S3L_SPARSE_COO` – `row` is of the same size as arrays `col` and `val` and contains row indices of the nonzero elements in array `val`.
- `S3L_SPARSE_CSR` – `row` is of size `m`+1 and contains pointers to the beginning of each row in arrays `col` and `val`.
- `S3L_SPARSE_CSC` – `row` is of size `n`+1 and contains pointers to the beginning of each column in arrays `col` and `val`.

`col` is an integer global array of rank 1 with the same length as array `val`. Its use will vary, depending on which sparse format is used:

- `S3L_SPARSE_COO` – `col` contains column indices of the corresponding elements stored in array `val`.
- `S3L_SPARSE_CSR` – `col` contains column indices of the corresponding elements stored in array `val`.
- `S3L_SPARSE_CSC` – `col` contains row indices of the corresponding elements in S3L array `val`.

`val` is a parallel array of rank 1, containing the nonzero elements of the sparse matrix. The storage pattern varies, depending on which sparse format is used:

- `S3L_SPARSE_COO` – Nonzero elements can be stored in any order.
- `S3L_SPARSE_CSR` – Nonzero elements should be stored row by row, from row 1 to row `m`.
- `S3L_SPARSE_CSC` – Nonzero elements should be stored column by column, from column 1 to column `n`.

The length of `val` is `nnz` for all three formats. This parameter represents the total number of nonzero elements in the sparse matrix. The data type of array elements can be real or complex (single- or double-precision).

Note – Because `row`, `col`, and `val` are copied to the working arrays described earlier (`indx`, `jndx`, `ptr`, and `val`), they can be deallocated immediately following the `S3L_declare_sparse` call.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_declare_sparse(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_declare_sparse` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse2.c
```

Initializing a Sparse Matrix From a File

`S3L_read_sparse` reads sparse matrix data from an ASCII file and distributes the data to all participating processes. Upon successful completion, `S3L_read_sparse` returns an S3L array handle in `A` that represents the distributed sparse matrix.

S3L_read_sparse supports the following sparse matrix formats:

- S3L_SPARSE_COO
- S3L_SPARSE_CSR
- S3L_SPARSE_CSC
- S3L_SPARSE_VBR

MatrixMarket Notes

Under the S3L_SPARSE_COO format, S3L_read_sparse can also read data supplied in either of two Coordinate formats that are distributed by MatrixMarket (http://gams.nist.gov/MatrixMarket/). The two supported MatrixMarket formats are real general and complex general.

MatrixMarket files always use one-based indexing. Consequently, they can only be used directly by Fortran programs. For a C or C++ program to use a MatrixMarket file, it must call the F77 application program interface. This is illustrated by the program example:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse.c
```

S3L_read_sparse has the following argument syntax:

```
S3L_read_sparse(A, spfmt, m, n, nnz, type, fname, dfmt, ier)
```

Upon exit, A contains an S3L array handle for the global general sparse matrix. This handle can be used in subsequent calls to other S3L sparse array functions.

spfmt indicates which sparse format is to be used. Its value can be any one of:

- S3L_SPARSE_COO
- S3L_SPARSE_CSR
- S3L_SPARSE_CSC
- S3L_SPARSE_VBR

m indicates the number of rows in the sparse matrix.

n indicates the number of columns in the sparse matrix.

nnz indicates the number of nonzero elements in the sparse matrix.

type indicates the S3L data type of the sparse array. It must be one of:

- S3L_float
- S3L_double
- S3L_complex
- S3L_double_complex

fname is a scalar character variable that names the ASCII file that contains the sparse matrix data.

dfmt specifies the format of the data to be read from the file named by fname. Allowed values are ascii and ASCII.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_read_sparse(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_read_sparse in use can be found in:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse.c
/opt/SUNWhpc/examples/s3l/sparse-f/ex_sparse2.f
```

Initializing a Sparse Matrix With Random Values

If you want to create a sparse matrix but don't have a source of data for its contents, you can use S3L_rand_sparse to create a global general sparse matrix that is populated with a set of random values. You specify the sparsity pattern to be used, as well as density of nonzero values, as part of the S3L_rand_sparse call.

Note – S3L_rand_sparse is intended primarily as a convenient tool for creating sparse matrices in those programming situations where the actual data values are not important—for example, when trying out various sparse matrix sizes or sparsity patterns.

S3L_rand_sparse supports all four sparse formats.

S3L_rand_sparse has the following argument syntax:

```
S3L_rand_sparse(A, spfmt, stype, m, n, density, type, seed, ...,
ier)
```

Upon exit, A contains an S3L array handle for the global general sparse matrix. This handle can be used in subsequent calls to other S3L sparse array functions.

spfmt indicates which sparse format is to be used. Its value can be any one of:

- S3L_SPARSE_COO
- S3L_SPARSE_CSR
- S3L_SPARSE_CSC
- S3L_SPARSE_VBR

If S3L_SPARSE_VBR is specified, two additional arguments should also be supplied:

- rptr – An integer array of length m+1, such that rptr[i] is the row index of the first point row in the i-th block row.
- cptr – An integer array of length n+1, such that cptr[j] is the column index of the first column in the j-th block column.

  If used, the rptr and cptr arguments follow the seed argument (as indicated by the "..." entry in the syntax illustration above.
- stype specifies the type of random pattern to be used. The choices are:
  - S3L_SPARSE_RAND – Random pattern
  - S3L_SPARSE_DRND – Random pattern with a guaranteed nonzero diagonal
  - S3L_SPARSE_SRND – Random symmetric sparse array
  - S3L_SPARSE_DSRN – Random symmetric sparse array with a guaranteed nonzero diagonal
  - S3L_SPARSE_DSPD – Random symmetric positive definite sparse array

For all formats except VBR, m indicates the number of rows in the sparse matrix. For the S3L_SPARSE_VBR format, m denotes the number of block rows in the sparse matrix.

For all formats except VBR, n indicates the number of columns in the sparse matrix. For the S3L_SPARSE_VBR format, n denotes the number of block columns in the sparse matrix.

density is a positive number less than or equal to 1.0. It suggests the approximate density of the array. For example, if 0.1 is supplied as the density argument, approximately 10% of the array elements will have nonzero values.

type specifies the data type of the sparse array. It must be one of:

- S3L_float
- S3L_double
- S3L_complex
- S3L_double_complex

seed is an integer that is used internally to initialize the random number generators. It affects both the pattern and the values of the array elements. The results are independent of the number of processes on which the function is invoked.

If the call is made from a Fortran program, error status will be in ier.

Note – The number of nonzero elements generated will depend primarily on the combination of the `density` value and the array extents given by `m` and `n`. Usually, the number of nonzero elements will approximately equal `m * n * density`. The behavior of the algorithm may cause the actual number of nonzero elements to be somewhat smaller than `m * n * density`. Regardless of the value supplied for `density`, the number of nonzero elements will always be >= `m`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_rand_sparse(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_rand_sparse` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/iter/ex_iter.c
/opt/SUNWhpc/examples/s3l/iter-f/ex_iter.f
```

Writing a Sparse Matrix to a File

`S3L_write_sparse` causes the process with MPI rank 0 to write the global sparse matrix `A` into a file. The matrix data will be written in a user-specified format, which can be any one of:

- `S3L_SPARSE_COO` – Coordinate
- `S3L_SPARSE_CSR` – Compressed Sparse Row
- `S3L_SPARSE_CSC` – Compressed Sparse Column
- `S3L_SPARSE_VBR` – Variable Block Row

`S3L_write_sparse` has the following argument syntax:

```
S3L_write_sparse(A, spfmt, fname, dfmt, ier)
```

`A` is an S3L array handle for the global general sparse matrix to be written.

`spfmt` indicates which sparse format is to be used. Its value can be any one of:

- `S3L_SPARSE_COO`
- `S3L_SPARSE_CSR`
- `S3L_SPARSE_CSC`
- `S3L_SPARSE_VBR`

`fname` is a scalar character variable that names the file to which the sparse matrix data will be written.

`dfmt` is a scalar character variable that specifies the data file format to be used in writing the sparse matrix data. The allowed values are `ascii` and `ASCII`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_write_sparse(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_write_sparse` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse.c
/opt/SUNWhpc/examples/s3l/sparse-f/ex_sparse.f
```

Printing a Sparse Matrix to Standard Output

`S3L_print_sparse` prints all nonzero values of a global general sparse matrix and their corresponding row and column indices to standard output.

For example, the following 4 x 6 sample matrix:

```
  3.14       0          0          20.04       0          0
  0         27          0           0         -0.6        0
  0          0         -0.01        0          0          0
 -0.031      0          0           0.08       0        314.0
```

could be printed by a C program in the following manner.

```
 4 6 8
(0,0)  3.140000
(0,3)  200.040000
(1,1)  27.000000
(1,4)  -0.600000
(2,2)  -0.010000
(3,0)  -0.031000
(3,3)  0.080000
(3,5)  314.000000
```

The first line prints three integers, `m`, `n`, and `nnz`, which represent the number of rows, columns, and the total number of nonzero elements in the matrix, respectively.

If the matrix is represented in the `S3L_SPARSE_VBR` format, three additional integers are printed: `bm`, `bn`, and `bnnz`. These integers indicate the number of block rows and block columns and the total number of nonzero block entries.

Note that, in the previous example, the row and column indices are zero-based, which means that the call to `S3L_print_sparse` was made from a C program. If the call had used the Fortran interface, the row and column indices would be one-based, as shown below:

```
 4 6 8
(1,1)  3.140000
(1,4)  200.040000
(2,2)  27.000000
(2,5)  -0.600000
(3,3)  -0.010000
(4,1)  -0.031000
(4,4)  0.080000
(4,6)  314.000000
```

`S3L_print_sparse` has the following argument syntax:

```
S3L_print_sparse(A, ier)
```

`A` is an S3L array handle for the global general sparse matrix to be printed.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_print_sparse(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_print_sparse` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse.c
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse2.c
/opt/SUNWhpc/examples/s3l/sparse-f/ex_sparse.f
```

Converting a Sparse Matrix From One Format to Another

`S3L_convert_sparse` converts an S3L sparse matrix that is represented in one sparse format to a different sparse format. It supports all four of the Sun S3L sparse formats.

`S3L_convert_sparse` has the following argument syntax:

```
S3L_convert_sparse(A, B, spfmt, ..., ier)
```

A is an S3L array handle for the global general sparse matrix whose format is to be converted—that is, the source matrix.

On exit, A is the S3L array handle for the global general sparse matrix that resulted from the conversion.

B is an S3L array handle for the converted global general sparse matrix—that is, the destination matrix.

The next argument, spfmt, specifies the sparse format to be used for the destination matrix. The value of spfmt must be one of:

- S3L_SPARSE_COO
- S3L_SPARSE_CSR
- S3L_SPARSE_CSC
- S3L_SPARSE_VBR

If the spfmt value is S3L_SPARSE_VBR, you can control the block-partitioning structure by supplying the following additional arguments after spfmt:

bm	Scalar integer that indicates the number of block rows in the block sparse matrix
bn	Scalar integer that indicates the number of block columns in the block sparse matrix
rptr	Integer array of length bm + 1, such that rptr[i] is the row index of the first point row in the i-th block row
cptr	Integer array of length bn + 1, such that cptr[j] is the column index of the first column in the j-th block column.

Note – To use the Sun S3L internal blocking algorithm rather than controlling the block partitioning explicitly, specify these four arguments as NULL pointers (for C/C++) or set to –1 (for F77/F90).

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_convert_sparse(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_convert_sparse in use can be found in:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse1.c
/opt/SUNWhpc/examples/s3l/sparse-f/ex_sparse1.f
```

Computing a Sparse Matrix-Vector Product

S3L_matvec_sparse computes the product of a global general sparse matrix with a global dense vector. The sparse matrix is described by the S3L array handle A. The global dense vector is described by the S3L array handle x. The product is stored in the global dense vector described by the S3L array handle y.

The array handle A is produced by a prior call to one of the following routines:

- S3L_declare_sparse
- S3L_read_sparse
- S3L_rand_sparse
- S3L_convert_sparse

S3L_matvec_sparse has the following argument syntax:

```
S3L_matvec_sparse(y, A, x, ier)
```

y is a global array of rank 1, with the same data type and precision as A and x. Its length is equal to the number of rows in the sparse matrix. Upon completion, y contains the product of the sparse matrix A and the vector x.

A is an S3L array handle for the global general sparse matrix.

x is a global array of rank 1, with the same data type and precision as A and y and with a length equal to the number of columns in the sparse matrix.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_matvec_sparse(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_matvec_sparse in use can be found in:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse.c
/opt/SUNWhpc/examples/s3l/sparse-f/ex_sparse.f
/opt/SUNWhpc/examples/s3l/iter/ex_iter.c
/opt/SUNWhpc/examples/s3l/iter-f/ex_iter.f
```

Deallocating a Sparse Matrix Array Handle

`S3L_free_sparse` deallocates the memory reserved for a sparse matrix and the associated array handle.

`S3L_free_sparse` has the following argument syntax:

```
S3L_free_sparse(A, ier)
```

`A` is an S3L array handle for the parallel S3L array that was allocated by a previous call to `S3L_declare_sparse`, `S3L_read_sparse`, `S3L_rand_sparse`, or `S3L_convert_sparse`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_free_sparse(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_free_sparse` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse.c
/opt/SUNWhpc/examples/s3l/sparse/ex_sparse2.c
/opt/SUNWhpc/examples/s3l/iter/ex_iter.c
/opt/SUNWhpc/examples/s3l/sparse-f/ex_sparse.f
/opt/SUNWhpc/examples/s3l/iter/ex_iter.c
/opt/SUNWhpc/examples/s3l/iter-f/ex_iter.f
```

CHAPTER 13

Sparse Linear System Solvers

Sun S3L provides support for solving sparse linear systems of the type A*x = B. The discussion of these routines is organized into the following sections:

Solving Sparse Linear Systems by the Direct Method

`S3L_sparse_solve` solves a linear system of equations A*x = y, where A is a sparse S3L array and A and y are both single- or double-precision real parallel arrays.

When calling `S3L_sparse_solve` to solve a new (unfactored) sparse linear system, specify `S3L_FULL_FACTOR_SOLVE` as the first element of the `option` argument vector. This will cause `S3L_sparse_solve` to reduce fill by reordering the array and to perform symbolic and numeric factoring before solving the system. It will also return a `setup` value that identifies the internal setup created by the factoring process.

If the same linear system is to be solved again, but with a different right-hand side, specify `S3L_SOLVE_ONLY` as the first element of the `option` argument. Also specify the `setup` value returned by the `S3L_sparse_solve` call that factored the sparse array. The new solution will make use of the internal setup created by the earlier `S3L_sparse_solve` call.

If a previously factored sparse array contains new values, but the sparsity pattern has not changed, it can be solved without specifying `S3L_FULL_FACTOR_SOLVE`. Instead, specify `S3L_SAME_SPARSITY_SOLVE` and the previously returned `setup` value. This causes `S3L_sparse_solve` to perform numeric factorization on the sparse array and then solve the linear system.

When the internal setup for a linear system is no longer needed, the resources associated with it can be freed by calling `S3L_sparse_solve_free` and specifying the applicable `setup` value.

Note – The S+ message-passing direct sparse solver was developed by Kai Shen and Tao Yang of the University of California at Santa Barbara. S+ can be used for general (asymmetric) sparse matrices.

The Sun Performance Library direct solver solves a sparse linear system on a single process.

`S3L_sparse_solve` has the following argument syntax:

```
S3L_sparse_solve(A, y, options, roptions, setup, ier)
```

`A` is an S3L array handle that describes a parallel, single- or double-precision, real sparse array.

`y` is an S3L array handle that describes a parallel real array of rank 1 (a vector) or rank 2 (a matrix), which contains the right-hand side of the linear system A*x = y. On exit, `y` is overwritten with the solution of the system.

`options` is an array of rank 1 whose elements control `S3L_sparse_solve` behavior in the following ways:

`options[0] = S3L_FULL_FACTOR_SOLVE`	Perform fill-reducing reordering, symbolic factorization, numeric factorization, and solve the linear system.
`options[0] = S3L_SAME_SPARSITY_SOLVE`	Do only numeric factorization and solve the linear system. Use this option when the sparsity pattern of a previously factored array stays the same but has a new set of values.
`options[0] = S3L_SOLVE_ONLY`	Only solve the linear system, using a previously computed factorization.
`options[1] = S3L_SPLUS_SOLVER`	Use S+ sparse solver.
`options[1] = S3L_PERFLIB_SOLVER`	Use the Sun Performance Library 6.0 sparse solver.

If `options[1]` = `S3L_PERFLIB_SOLVER`, specify the following options as well:

`options[2] = S3L_NON_SYMMETRIC`	The sparse array has asymmetric structure and asymmetric values.
`options[2] = S3L_SYMMETRIC`	The sparse array has symmetric structure and symmetric values.
`options[2] = S3L_SYM_STRUCT`	The sparse array has symmetric structure but asymmetric values.
`options[3] = S3L_NO_PIVOT`	Do not use pivoting.
`options[3] = S3L_DO_PIVOT`	Use pivoting.

`roptions` is not currently used. It may be used in the future for specifying such parameters as a drop tolerance for pivoting, a threshold value for determining when a block is considered dense, and an amalgamation constant.

`setup` is an integer that is returned by `S3L_sparse_solve` upon completion. It describes the sparse linear solution resulting from this call.

Note – If the internal setup created by this call will be used for additional solutions of the linear system, subsequent calls to `S3L_sparse_solve` would use this setup value. It will also be used in a subsequent call to `S3L_sparse_solve_free` to free the internal data associated with this sparse linear system solution.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for this routine, see the `S3L_sparse_solve(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_sparse_solve` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/spsolve/ex_sparse_solve1.c
/opt/SUNWhpc/examples/s3l/spsolve-f/ex_sp_solve1.f
```

Solving Sparse Linear Systems by an Iterative Method

`S3L_gen_iter_solve` solves a linear system of equations of the form A*x = y, where A is a sparse S3L array and x and y are both single- or double-precision dense parallel arrays.

Given a general, square sparse matrix `A` and a right-hand side vector `b`, `S3L_gen_iter_solve` solves the linear system of equations `Ax = b`, using an iterative algorithm, with or without preconditioning.

The first three arguments to `S3L_gen_iter_solve` are S3L internal array handles that describe the global general sparse matrix `A`, the rank 1 global array `b`, and the rank 1 global array `x`.

The sparse matrix `A` is produced by a prior call to one of the following sparse routines:

- `S3L_declare_sparse`
- `S3L_read_sparse`
- `S3L_rand_sparse`
- `S3L_convert_sparse`

The global rank 1 arrays, `b` and `x`, have the same data type and precision as the sparse matrix `A`, and both have a length equal to the order of `A`.

Two local rank 1 arrays, `iparm` (integer array) and `rparm` (real array), provide user control over various aspects of `S3L_gen_iter_solve` behavior, including:

- Choice of algorithm to be used
- Type of preconditioner to use on `A`
- Flags to select the initial guess to the solution
- Maximum number of iterations to be taken by the solver
- If the restarted GMRES algorithm is chosen, selection of the size of the Krylov subspace
- Tolerance values to be used by the stopping criterion
- If the Richardson algorithm is chosen, selection of the scaling factor to be used

The options supported by the `iparm` and `rparm` arguments are described in the following subsections.

Note – `iparm` and `rparm` must be preallocated and initialized before `S3L_gen_iter_solve` is called. To enable the default condition for any parameter, set it to `0`. Otherwise, initialize the arguments with the appropriate parameter values, as described in the following subsections.

Algorithm

`S3L_gen_iter_solve` attempts to solve `Ax = b` using one of the following iterative solution algorithms. The choice of algorithm is determined by the value supplied for the parameter `iparm[S3L_iter_solver]`. The various options available for this parameter are listed and described in TABLE 13-1.

TABLE 13-1 `iparm[S3L_iter_solver]` Options

Option	Description
`S3L_bcgs`	BiConjugate Gradient Stabilized (Bi-CGSTAB)
`S3L_cgs`	Conjugate Gradient Squared (CGS)
`S3L_cg`	Conjugate Gradient (CG)
`S3L_cr`	Conjugate Residuals (CR)
`S3L_gmres`	Generalized Minimum Residual (GMRES) – default
`S3L_qmr`	Quasi-Minimal Residual (QMR)
`S3L_richardson`	Richardson method

Preconditioning

`S3L_gen_iter_solve` implements left preconditioning. That is, preconditioning is applied to the linear system `Ax = b` by:

$$Q^{-1} A = Q^{-1} b$$

where Q is the preconditioner and Q^{-1} denotes the inverse of Q. The supported preconditioners are listed in TABLE 13-2.

TABLE 13-2 `iparm[S3L_iter_pc]` Options

Option	Description
`S3L_none`	No preconditioning will be done (default).
`S3L_jacobi`	Point Jacobi preconditioner will be used. Note that this option is not supported when the sparse matrix `A` is represented under `S3L_SPARSE_VBR` format.
`S3L_bjacobi`	Block Jacobi preconditioner will be used. Note that this option is supported only when the sparse matrix `A` is represented under `S3L_SPARSE_VBR` format.
`S3L_ilu`	Use a simplified ILU(0)—the Incomplete LU factorization of level zero preconditioner. This preconditioner modifies only diagonal nonzero elements of the matrix. Note that this option is not supported when the sparse matrix `A` is represented under `S3L_SPARSE_VBR` format.

Convergence/Divergence Criteria

The `iparm[S3L_iter_conv]` parameter selects the criterion to be used for stopping computation. Currently, the single valid option for this parameter is `S3L_r0`, which selects the default criterion for both convergence and divergence. The convergence criterion is satisfied when:

`err` = `||rj||_2 / ||r0||_2` < epsilon

and the divergence criterion is met when:

`err` = `||rj||_2 / ||r0||_2` > 10000.0

where:

- `rj` and `r0` are the residuals obtained at iterations `j` and `0`.
- `||.||_2` is the 2-norm.
- epsilon is the desired convergence tolerance stored in `rparm[S3L_iter_tol]`.
- 10000.0 is the divergence tolerance, which is set internally in the solver.

Initial Guess

The parameter `iparm[S3L_iter_init]` determines the contents of the initial guess to the solution of the linear system as follows:

- `0` – Applies zero as the initial guess. This is the default.
- `1` – Applies the value contained in array `x` as the initial guess. For this case, the user must initialize `x` before calling `S3L_gen_iter_solve`.

Maximum Iterations

On input, the `iparm[S3L_iter_maxiter]` parameter specifies the maximum number of iterations to be taken by the solver. Set to `0` to select the default, which is 10000.

On output, `iparm[S3L_iter_maxiter]` contains the total number of iterations taken by the solver at the time of termination.

Krylov Subspace

If the restarted GMRES algorithm is selected, `iparm[S3L_iter_kspace]` specifies the size of the Krylov subspace to be used. The default is 30.

Stopping Criterion Tolerance

On input, `rparm[S3L_iter_tol]` specifies the tolerance values to be used by the stopping criterion. Its default is 10-8.

On output, `rparm[S3L_iter_tol]` contains the computed error, `err`, according to the convergence criteria. See the `iparm[S3L_iter_conv]` description for details.

Richardson Scaling Factor

If the Richardson method is selected, `rparm[S3L_rich_scale]` specifies the scaling factor to be used. The default value is 1.0.

Iteration Termination

`S3L_gen_iter_solve` terminates the iteration when one of the following conditions is met:

- The computation has satisfied the convergence criterion.
- The computation has diverged.
- An algorithmic breakdown has occurred.
- The number of iterations has exceeded the supplied value.

`S3L_gen_iter_solve` has the following argument syntax:

```
S3L_gen_iter_solve(A, b, x, iparm, rparm, ier)
```

`A` is an S3L array handle that describes a global general sparse matrix.

`b` is an S3L array handle that describes a global array of rank 1. It has the same data type and precision as `A`. `b` contains the right-hand side vector of the linear problem.

`x` is an S3L array handle that describes a global array of rank 1. It has the same data type and precision as `A` and `b`. At the start, `x` contains the initial guess for the solution to the linear system. Upon exit, `x` contains the converged solution. If the computation breaks down or diverges, `x` will contain the results of the most recent iteration.

`iparm` is an integer local array of rank 1 and length `s3l_iter_iparm_size`. At the start, `iparm` options have the following uses:

- `iparm[S3l_iter_solver]` – Specifies the iterative algorithm to be used. Set it to `0` to use the default solver GMRES. See the Description section for details.
- `iparm[S3l_iter_pc]` – Specifies the preconditioner to be used. Set it to 0 to use the default option, `S3L_none`.
- `iparm[S3l_iter_conv]` – Selects the criterion to be used for stopping the computation.
- `iparm[S3l_iter_init]` – Specifies the contents of the initial guess to the solution of the linear system.
- `iparm[S3l_iter_maxiter]` – Specifies the maximum number of iterations to be taken by the solver.
- `iparm[S3l_iter_kspace]` – Specifies the size of the Krylov subspace for restarted GMRES.

Upon exit, `iparm` contains the total number of iterations taken by the solver at the time of termination.

rparm is a real local array with the same precision as x and a length equal to S3L_iter_rparm_size. At the start, it provides the following options:

- rparm[S3L_iter_tol] – Specifies the tolerance values to be used by the stopping criterion. It has a default of 10-8.
- rparm[S3L_rich_scale] – Specifies the scaling factor to be used in the Richardson method. The default is 1.0.

Upon exit, rparm contains the value of err, according to the stopping criterion, as computed for the last iteration.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_gen_iter_solve(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_gen_iter_solve in use can be found in:

```
/opt/SUNWhpc/examples/s3l/iter/ex_iter.c
/opt/SUNWhpc/examples/s3l/iter-f/ex_iter.f
```

Deallocating a Sparse Linear System Solver

S3L_sparse_solve_free frees all internal data associated with the solution of a sparse linear system.

S3L_sparse_solve_free has the following argument syntax:

```
S3L_sparse_solve_free(setup, ier)
```

setup is an integer associated with a particular sparse linear solution. It was previously returned by the S3L_sparse_solve call that produced the solution.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for this routine, see the S3L_sparse_solve_free(3) man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing S3L_sparse_solve_free in use can be found in:

```
/opt/SUNWhpc/examples/s3l/spsolve/ex_sparse_solve1.c
/opt/SUNWhpc/examples/s3l/spsolve-f/ex_sp_solvef1.f
```

CHAPTER 14

Fast Fourier Transform Routines

This chapter discusses the routines Sun S3L provides for FFT operations. The discussion is organized into the following sections:

Overview

Sun S3L provides the following FFT computational routines:

- `S3L_fft()` – Compute forward FFTs along all axes of a complex or double-complex array.
- `S3L_ifft()` – Compute reverse FFTs along all axes of a complex or double-complex array.
- `S3L_fft_detailed()` – Compute forward or inverse FFTs along a specified axis of a complex or double-complex array.
- `S3L_rc_fft()` – Compute forward FFTs along all axes of a real array, producing a complex result.
- `S3L_cr_fft()` – Compute inverse FFTs along all axes of a complex array, producing a real result.

In addition to these FFT computational routines, the library includes routines for setting up and deallocating internal data structures that are used by the FFT routines. These FFT setup and deallocation routines are

- `S3L_fft_setup()` – Set up for computing FFTs of complex arrays
- `S3L_rc_fft_setup()` – Set up for computing FFTs of real arrays

- `S3L_fft_free_setup()` – Deallocate internal structures created by `S3L_fft_setup()`
- `S3L_rc_fft_free_setup()` – Deallocate internal structures created by `S3L_rc_fft_setup()`.

Use of the FFT setup, computational, and deallocation routines is discussed in the following sections.

Setting Up for FFT Operations

Before performing an FFT setup of an S3L array, you must call either `S3L_declare` or `S3L_declare_detailed` to allocate memory for that array. Each array declaration routine returns an S3L array handle that uniquely references the allocated array.

After allocating the target array, but before calling an FFT computational routine, call the appropriate FFT setup routine. You have two choices:

- If the target array is of type `S3L_complex` or `S3L_double_complex`, call `S3L_fft_setup`.
- If the array is of type `S3L_float` or `S3L_double`, call `S3L_rc_fft_setup`.

`S3L_fft_setup` and `S3L_rc_fft_setup` have the same argument syntax:

```
S3L_fft_setup(a, setup_id, ier)
S3L_rc_fft_setup(a, setup_id, ier)
```

`a` is the S3L array handle returned by an earlier call to `S3L_declare` or `S3L_declare_detailed`.

`setup_id` is the setup ID returned by this call to `S3L_fft_setup`.

If the call is made from a Fortran program, error status will be in `ier`.

`S3L_fft_setup` and `S3L_rc_fft_setup` both allocate memory for internal setup structures. These internal data structures will hold the twiddle factors to be used in subsequent FFT computations, as well as information about the size and layout characteristics of the target array. Memory is also allocated for temporary arrays that may be needed during the FFT operations.

Note – The setup routines do not examine or modify the contents of the parallel array associated with the array handle. They use only its rank, extents, and type information in creating their setup structures.

If you want to perform multiple FFT computations on arrays that all have the same size and type, you can reuse the setup ID returned by one call to `S3L_fft_setup` or `S3L_rc_fft_setup`. In other words, you would only need to make a single call to a setup routine if all the target arrays for subsequent FFT operations are of the same size and type.

For detailed descriptions of the Fortran and C bindings for these routines, see the `S3L_fft_setup(3)` and `S3L_rc_fft_setup(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_fft_setup` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/fft/fft.c
/opt/SUNWhpc/examples/s3l/fft/ex_fft1.c
/opt/SUNWhpc/examples/s3l/fft/ex_fft2.c
/opt/SUNWhpc/examples/s3l/fft-f/ex_fft.f
/opt/SUNWhpc/examples/s3l/fft-f/ex_fft1.f
```

Examples of `S3L_rc_fft_setup` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/rc_fft/rc_fft.c
/opt/SUNWhpc/examples/s3l/rc_fft-f/rc_fft.f
```

Using Sun S3L FFT Computational Routines

Sun S3L FFT routines are optimized for use in a message-passing context. The core consists of a matrix transposition module, combined with nodal single-process, single-thread FFTs. The node-level operations are performed using functions from the Sun Performance Library.

The Sun S3L FFT routines support parallel arrays with arbitrary distributions in a multiprocess environment. However, certain distributions are better for maximum efficiency. See the *Sun HPC ClusterTools Performance Guide* for advice on performance tuning of FFT operations.

Simple, Complex-to-Complex FFTs

`S3L_fft` and `S3L_ifft` compute, respectively, the forward and inverse Discrete Fourier Transforms (DFTs) of complex arrays of up to three dimensions. Both power-of-two and arbitrary radix FFTs are supported. The same FFT setup can be used for both forward and inverse FFT operations.

Note – In Sun S3L, the forward FFT is defined by a negative sign in the exponential factors and the inverse by a positive sign.

`S3L_fft` and `S3L_ifft` have the same argument syntax:

```
S3L_fft(a, setup_id, ier)
S3L_ifft(a, setup_id, ier)
```

`a` is the S3L array handle returned by an earlier call to `S3L_declare` or `S3L_declare_detailed`.

`setup_id` is the setup ID returned by an earlier call to `S3L_fft_setup`.

If the call is made from a Fortran program, error status will be in `ier`.

`S3L_fft` and `S3L_ifft` do not perform scaling. Consequently, if you call `S3L_fft` and then call `S3L_ifft`, the original data will be scaled by the product of the array extents.

For FFT computation of 1D arrays, the Cooley-Tuckey algorithm with stages (number of processes) and length/number of processes is used. Consequently, for 1D FFTs, the array size must be a multiple of the square of the number of processes.

A standard row-column algorithm is used for 2D and 3D FFTs.

When the target array has more than three dimensions or if you want more control over how the FFT operations are applied to the axes of an array, use `S3L_fft_detailed` instead. This routine is discussed next.

Detailed, Complex-to-Complex FFTs

`S3L_fft_detailed` uses the same setup as `S3L_fft` and `S3L_ifft`, but accepts two additional parameters. Its argument syntax is:

```
S3L_fft_detailed(a, setup_id, iflag, axis, ier)
```

`a` is the S3L array handle returned by an earlier call to `S3L_declare` or `S3L_declare_detailed`.

`setup_id` is the setup ID returned by an earlier call to `S3L_fft_setup`.

`iflag` specifies the direction of the FFT computation; 1 = forward and –1 = inverse.

`axis` specifies the axis along which the FFT is to be computed.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the `S3L_fft` and `S3L_fft_detailed` routines, see the `S3L_fft(3)` and `S3L_fft_detailed(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_fft` and `S3L_fft_detailed` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/fft/fft.c
/opt/SUNWhpc/examples/s3l/fft/ex_fft1.c
/opt/SUNWhpc/examples/s3l/fft/ex_fft2.c
/opt/SUNWhpc/examples/s3l/fft-f/fft.f
```

Real-to-Complex and Complex-to-Real FFTs

`S3L_rc_fft` computes the forward FFT of a real array of up to three dimensions. It accepts as input a parallel array containing real, single- or double-precision data. Upon completion, it overwrites the real contents of the array with the packed representation of a complex array.

`S3L_cr_fft` computes the inverse FFT of an array of up to three dimensions whose contents are the packed representation of a complex array.

These routines employ algorithms based on nonstandard extensions of the Cooley-Tuckey factorization and the Chinese Remainder Theorem. Both power-of-two and arbitrary radix FFTs are supported.

The nodal FFT functions upon which the parallel FFT is based are mixed radix with prime factors of 2, 3, 5, 7, 11, and 13. Performance will benefit when the size of the array is a product of powers of these factors. When the size of an array cannot be factored into these prime factors, a slower DFT will be used for the remainder.

Supported Array Sizes

`S3L_rc_fft` and `S3L_cr_fft` have the following array size requirements:

- 1D – The array size must be divisible by $4 \times p^2$, where p is the number of processors.
- 2D – Each of the array lengths must be divisible by 2 x p, where p is the number of processors.
- 3D – The first dimension must be even and must have a length of at least 4. The second and third dimensions must be divisible by 2 x p, where p is the number of processors.

Scaling

The real-to-complex and complex-to-real parallel FFTs do not perform scaling. Consequently, for a forward 1D real-to-complex FFT of a vector of length *n*, followed by an inverse 1D complex-to-real FFT of the result, the original vector is multiplied by *n*/2.

If the data fits in a single process, a 1D real-to-complex FFT of a vector of length *n*, followed by a 1D complex-to-real FFT, results in the original vector being scaled by *n*.

For a real-to-complex FFT of a 2D real array of size n x m, followed by a complex-to-real FFT, the original array is scaled by n x m.

Similarly, a real-to-complex FFT applied to a 3D real array of size n x m x k, followed by a complex-to-real FFT, results in the original array being scaled by n x m x k.

Complex Data Packed Representation

The following describes the complex data packing scheme used in the real-to-complex FFTs of 1D, 2D, and 3D arrays.

1D Real-to-Complex Fourier Transform

The periodic Fourier Transform of a real sequence X[i], i = 0, ..., N-1 is Hermitian (exhibits conjugate symmetry around its middle point).

If X[i],i = 0, ..., N-1 are the complex values of the Fourier Transform, then:

```
X[i] = conj(X[N-i]), i=1, ..., N-1          (eq. 1)
```

Consider for example the real sequence:

```
X =

0
1
2
3
4
5
6
7
```

Its Fourier Transform is:

```
X =

28.0000
-4.0000 + 9.6569i
-4.0000 + 4.0000i
-4.0000 + 1.6569i
-4.0000
-4.0000 - 1.6569i
-4.0000 - 4.0000i
-4.0000 - 9.6569i
```

As you can see:

```
X[1] = conj(X[7])
X[2] = conj(X[6])
X[3] = conj(X[5])
X[4] = conj(X[4]) (i.e., X[4] is real)
X[5] = conj(X[3])
X[6] = conj(X[2])
X[7] = conj(X[1])
```

Because of the Hermitian symmetry, only N/2+1 = 5 values of the complex sequence X need to be calculated and stored. The rest can be computed from (1).

Note that X[0] and X[N/2] are real valued so they can be grouped together as one complex number. In fact, S3L stores the sequence X as:

```
X[0]    X[N/2]
X[1]
X[2]

or

X =
28.0000 - 4.0000i
-4.0000 + 9.6569i
-4.0000 - 4.0000i
-4.0000 + 1.6569i
```

The first line in this example represents the real and imaginary parts of a complex number.

To summarize, in S3L, the Fourier Transform of a real-valued sequence of length N (where N is even) is stored as a real sequence of length N. This is equivalent to a complex sequence of length N/2.

2D Real-to-Complex Fourier Transform

The method used for 2D FFTs is similar to that used for 1D FFTs. When transforming each of the array columns, only half of the data is stored.

3D Real-to-Complex Fourier Transform

As with the 1D and 2D FFTs, no extra storage is required for the 3D FFT of real data, since advantage is taken of all possible symmetries. For an array `a`(M,N,K), the result is packed in complex `b`(M/2,N,K) array. Hermitian symmetries exist along the planes `a`(0,:,:) and `a`(M/2,:,:) and along dimension 1.

See the `rc_fft.c` and `rc_fft.f` program examples for illustrations of these concepts. The paths for these online examples are provided at the end of this section.

Argument Syntax

`S3L_rc_fft` and `S3L_cr_fft` have the same argument syntax:

```
S3L_rc_fft(a, setup_id, ier)
S3L_cr_fft(a, setup_id, ier)
```

`a` is the S3L array handle returned by an earlier call to `S3L_declare` or `S3L_declare_detailed`.

`setup_id` is the setup ID returned by an earlier call to `S3L_rc_fft_setup`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the `S3L_rc_fft` and `S3L_cr_fft` routines, see the `S3L_rc_fft(3)` and `S3L_cr_fft(3)` man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing `S3L_rc_fft` and `S3L_cr_fft` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/rc_fft/rc_fft.c
/opt/SUNWhpc/examples/s3l/rc_fft-f/rc_fft.f
```

Deallocating FFT Setups

When an FFT setup is no longer needed, call `S3L_fft_free_setup` to free up memory that was allocated by a prior call to `S3L_fft_setup`. Likewise, call `S3L_rc_fft_free_setup` to free memory that was allocated by a prior call to `S3L_rc_fft_setup`.

S3L_fft_free_setup and S3L_rc_fft_free_setup have the same argument syntax:

```
S3L_fft_free_setup(setup_id, ier)
S3L_rc_fft_free_setup(setup_id, ier)
```

setup_id is the setup ID returned by an earlier call to S3L_rc_fft_setup.

If the call is made from a Fortran program, error status will be in ier.

For detailed descriptions of the Fortran and C bindings for the S3L_fft_free_setup and S3L_rc_fft_free_setup routines, see the S3L_fft_free_setup(3) and S3L_rc_fft_free_setup(3) man pages or the corresponding descriptions in the *Sun S3L Reference Manual*.

Examples showing S3L_fft_free_setup in use can be found in:

```
/opt/SUNWhpc/examples/s3l/fft/fft.c
/opt/SUNWhpc/examples/s3l/fft/ex_fft1.c
/opt/SUNWhpc/examples/s3l/fft/ex_fft2.c
/opt/SUNWhpc/examples/s3l/fft-f/ex_fft.f
/opt/SUNWhpc/examples/s3l/fft-f/ex_fft1.f
```

Examples of S3L_rc_fft_free_setup are in:

```
/opt/SUNWhpc/examples/s3l/rc_fft/rc_fft.c
/opt/SUNWhpc/examples/s3l/rc_fft-f/rc_fft.f
```

CHAPTER 15

Parallel Random Number Generation Routines

Sun S3L offers two alternative methods for generating pseudo-random numbers for parallel arrays, lagged Fibonacci random number generator (LFG) and Linear Congruential random number generator (LCG). The routines associated with these two methods are discussed in the following sections:

Initialize Lagged Fibonacci State Table

`S3L_setup_rand_fib` allocates a set of LFG state tables and initializes them with the fixed parameters: l = 17, k = 5, m = 32, where:

- l is the table lag pointer
- k is the short lag pointer
- m is the width, in bits, of each table element

An LFG state table operates as a circular buffer, with the table lag and short lag values pointing into different locations in the table. As the two pointers step through the state table, a sequence of pseudo-random numbers is created. At each step in the cycle, the two values pointed at by table lag and short lag are added together, with that sum replacing the value created by the previous step.

The size of each state table is l x m (the product of the table lag and width parameters). Since `S3L_setup_rand_fib` uses 17 and 32 for these parameters, each state table will occupy 544 bits of memory.

The table lag and short lag values have been selected to provide optimal balance between the length of the random number generation *period* and the amount of memory used. A random number generator's period is the number of new random values that it generates before returning to the start of the sequence. Maximizing the period ensures that the random numbers generated for each node are from a different period of the LFG.

A Linear Multiplicative Generator (LMG) is used to initialize the noncritical elements of the state table. The LMG itself is initialized by a seed value that is entered as an argument to `S3L_setup_rand_fib`.

`S3L_setup_rand_fib` has the following argument syntax:

```
S3L_setup_rand_fib(setup_id, seed, ier)
```

`setup_id` is a scalar integer returned by `S3L_setup_rand_fib`. It can be used by subsequent calls to `S3L_rand_fib` to access the state tables.

`seed` is an integer value that is used to initialize the Linear Multiplicative Generator which, in turn, initializes the LFG state tables.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the LFG setup routine, see the `S3L_setup_rand_fib(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_setup_rand_fib` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/rand_fib/rand_fib.c
/opt/SUNWhpc/examples/s3l/rand_fib-f/rand_fib.f
```

Lagged Fibonacci Random Number Generator

`S3L_rand_fib` writes a pseudo-random number into each element of a parallel array, `a`, using a lagged Fibonacci random number generator (LFG). The random numbers are produced by the following iterative equation:

$$x[n] = (x[n-e] + x[n-k]) \% 2^m$$

The result of `S3L_rand_fib` depends on how the parallel array `a` is distributed.

The following summarizes the different value ranges that are used for the different supported data types:

- When the parallel array is of type integer, its elements are filled with nonnegative integers in the range 0 . . . 2^{31} -1.
- When the parallel array is single- or double-precision real, its elements are filled with random nonnegative numbers in the range 0 . . . 1.
- For complex arrays, the real and imaginary parts are initialized to random real numbers.

`S3L_rand_fib` has the following argument syntax:

```
S3L__rand_fib(a, setup_id, ier)
```

`a` is an S3L array handle that describes the parallel array to be initialized by the LFG.

`setup_id` is an integer value that is used to access the state table associated with the array `a`.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the LFG routine, see the `S3L_rand_fib(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_rand_fib` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/rand_fib/rand_fib.c
/opt/SUNWhpc/examples/s3l/rand_fib-f/rand_fib.f
```

Linear Congruential Random Number Generator

`S3L_rand_lcg` initializes a parallel array `a`, using a Linear Congruential random number generator (LCG). It produces random numbers that are independent of the distribution of the parallel array.

The random numbers are initialized by an internal iterative equation of the type:

$x[n] = (a*x[n-1] + c) \% 2^m$

The result of `S3L_rand_lcg` does not depend on how the parallel array `a` is distributed.

The following summarizes the different value ranges that are used for the different supported data types:

- When the parallel array is of type integer, its elements are filled with nonnegative integers in the range 0 . . . 2^{31} -1.
- When the parallel array is single- or double-precision real, its elements are filled with random nonnegative numbers in the range 0 . . . 1.
- For complex arrays, the real and imaginary parts are initialized to random real numbers.

`S3L_rand_lcg` has the following argument syntax:

```
S3L__rand_lcg(a, iseed, ier)
```

`a` is an S3L array handle that describes the parallel array to be initialized by the LCG.

`iseed` is an integer value. If positive, it is used as the initial seed for the LCG. If zero or negative, the LCG produces a sequence of random numbers that is a continuation of a sequence generated by a prior `S3L_rand_lcg` call.

If the call is made from a Fortran program, error status will be in `ier`.

For detailed descriptions of the Fortran and C bindings for the LCG routine, see the `S3L_rand_lcg(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_rand_lcg` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/rand_lcg/rand_lcg.c
/opt/SUNWhpc/examples/s3l/rand_lcg-f/rand_lcg.f
```

Deallocate LFG Setup

`S3L_free_rand_fib` frees memory allocated to the state table of a particular random number generator.

`S3L_free_rand_fib` has the following argument syntax:

```
S3L_free_rand_fib(setup_id, ier)
```

`setup_id` is an integer value associated with the state tables that are to be deallocated.

For detailed descriptions of the Fortran and C bindings for the LFG deallocation routine, see the `S3L_free_rand_fib(3)` man page or the corresponding description in the *Sun S3L Reference Manual*.

Examples showing `S3L_free_rand_fib` in use can be found in:

```
/opt/SUNWhpc/examples/s3l/rand_fib/rand_fib.c
/opt/SUNWhpc/examples/s3l/rand_fib-f/rand_fib.f
```

CHAPTER 16

Summary of Other Sun S3L Routines

This chapter provides a summary listing of routines that are in Sun S3L, but are not discussed in detail elsewhere in this manual. The discussion is separated into two sections, as follows:

For detailed descriptions of these routines, refer to the *Sun S3L Reference Manual*.

Other Computational Routines

Walsh Transform

The following Sun S3L routines support the computation of the discrete Walsh/Hadamard transform of 1D and 2D S3L arrays:

```
S3L_walsh_setup
S3L_walsh
S3L_walsh_free_setup
```

S3L computes the unordered Hadamard transform, whose matrix is a permutation of the ordered Hadamard and Walsh transforms. The transform can be computed either in place or out of place.

If computed in-place, the result is in `a` and in order. If it is computed out of place, the result is in `b` and out of order.

Iterative Eigenpairs

Sun S3L provides the following routine for computing selected eigenpairs of dense or sparse matrices using iterative methods:

```
S3L_eigen_iter
```

Eigenpair selection can be based on user-specified properties, such as largest magnitude. When computing eigenpairs in densely populated arrays, the multiple-instance paradigm can be used to shorten the time to solution.

Stock Option Pricing

The following Sun S3L routines compute the prices of vanilla and certain exotic stock option prices. Optional support for hedge statistics (Greeks) is also provided.

```
S3L_fin_fd_1D
S3L_fin_fd_2D
```

Both use fourth-order, unconditionally stable, oscillation-free finite-difference method to solve 1D and 2D Black-Scholes partial differential equations.

Discrete Sine and Cosine Transforms

Sun S3L provides routines for computing the discrete cosine transform Type IV and discrete sine transform of 1D, 2D, and 3D S3L arrays. The arrays have to be real (`S3L_float` or `S3L_double`).

The discrete cosine transform routines are:

```
S3L_dct_iv_setup
S3L_dct_iv
S3L_dct_iv_setup
```

The discrete sine transform routines are:

```
S3L_dst_setup
S3L_dst
S3L_dst_free_setup
```

Quadratic Programming Optimization

The following routines are provided for quadratic programming optimization:

```
S3L_qp
S3L_qp_attr_set
S3L_qp_attr_destroy
S3L_qp_attr_init
```

`S3L_qp` applies an interior point method to solve a linear/quadratic optimization problem. The other routines are used to define certain attributes, such as the type of solver to be used, and to release the array handle associated with the specified attributes.

Sparse Linear Problem Solver

Sun S3L also provides the following routine for solving linear/quadratic optimization problems for sparse S3L arrays.

```
S3L_lp_sparse
```

Cholesky Solver

Sun S3L provides the following routines for computing Cholesky factors and solving systems of distributed linear equations:

```
S3L_cholesky_factor
S3L_cholesky_solve
S3L_cholesky_invert
```

For each square matrix `A` in the parallel array `a`, `S3L_cholesky_factor` computes the Cholesky factorization. The factorization has the form A = U' x U, where U is an upper triangular matrix.

`S3L_cholesky_solve` uses the factors computed by `S3L_cholesky_factor` to solve a system of distributed linear equations of the form AX = B for each square matrix `A` in the parallel array `a`.

`S3L_cholesky_invert` computes the inverse of each square matrix instance `A` of the parallel array `a`. It does this by inverting the Cholesky factor U and then computing inverse(U) * inverse(U)'.

Structured Solvers

The following routines can be used to solve banded and triangular systems:

```
S3L_gen_band_factor
S3L_gen_band_solve
S3L_gen_band_free_factors
S3L_gen_trid_factor
S3L_gen_trid_solve
S3L_gen_trid_free_factors
```

`S3L_gen_band_factor` performs the LU factorization of an n x n general banded array and `S3L_gen_band_solve` solves the banded system represented by the factorization results. `S3L_gen_band_free_factors` frees the internal data structures associated with the banded matrix factorization.

`S3L_gen_trid_factor` factors a tridiagonal matrix whose diagonal is stored in a vector. `S3L_gen_trid_solve` solves a tridiagonal system using the factors supplied by an earlier call to `S3L_gen_trid_factor`. `S3L_gen_trid_free_factors` frees the internal data structures associated with a prior tridiagonal factorization.

Both `S3L_gen_band_factor` and `S3L_gen_trid_factor` operate on arrays of rank 3 and greater in 2D slices, using the multiple-instance method to reduce overall execution time.

Dense Symmetric Eigenvalue Solver

Sun S3L provides the following routine for finding selected eigenvalues and, optionally, eigenvectors of Hermitian matrices:

```
S3L_sym_eigen
```

Various controls can be placed on the selection process, such as specifying a range of values or range of indices within which the eigenvalues must lie.

Condition Numbers

The following routine can be used to compute the condition numbers of square arrays:

```
S3L_condition_number
```

Internal LU factorization is used in combination with a norm as specified by a calling argument. The norm can be either 1-norm or infinity norm.

Least-Squares Solver

Sun S3L provides the following routine for finding the least-squares solution of an overdetermined system (`m` >= `n`):

`S3L_gen_lsq`

For an underdetermined system (`m` < `n`), `S3L_gen_lsq` finds the minimum norm solution.

Dense Singular Value Decomposition

The following routine computes the singular value of a parallel array and, optionally, the right and/or left singular vectors:

`S3L_gen_svd`

Iterative Solver

The following routine uses an iterative algorithm, with or without preconditioning, to solve a linear system of equations of the form Ax = b:

`S3L_gen_iter_solve`

Autocorrelation

Sun S3L provides the following routines for computing the 1D or 2D autocorrelation of a signal represented by a parallel array:

`S3L_acorr_setup`
`S3L_acorr`
`S3L_acorr_free_setup`

Convolution

Sun S3L provides the following routines for computing the 1D or 2D convolution of a signal represented by a parallel array, using a filter contained in a second parallel array:

```
S3L_conv_setup
S3L_conv
S3L_conv_free_setup
```

Deconvolution

Sun S3L provides the following routines for computing the 1D or 2D deconvolution of a signal represented by a parallel array:

```
S3L_deconv_setup
S3L_deconv
S3L_deconv_free_setup
```

Grade Elements

Sun S3L includes a family of routines that computes the grade of the elements of a parallel array. Grading is done in either descending or ascending order and is done either across the whole array or along a specified axis. These routines are:

```
S3L_grade_up
S3L_grade_down
S3L_grade_detailed_up
S3L_grade_detailed_down
```

Sort Elements

Sun S3L includes a family of routines that sorts the elements of a parallel array. Sorting is done in either descending or ascending order and is done either across the whole array or along a specified axis. These routines are:

```
S3L_sort
S3L_sort_up
S3L_sort_down
S3L_sort_detailed
S3L_sort_detailed_up
S3L_sort_detailed_down
```

Parallel Transpose

The following routine performs a general permutation of the axes of a parallel array:

```
S3L_trans
```

Other Toolkit Routines

Perform Operations on Array Elements

Sun S3L includes a variety of routines that can be used to alter the values of individual elements or particular subsets of elements in parallel arrays.

The following routines perform operations that involve one or two operands:

```
S3L_array_op1
S3L_array_op2
S3L_array_scalar_op2
```

`S3L_array_op1` and `S3L_array_op2` both operate on elements contained in one (`op1`) or two (`op2`) arrays. `S3L_array_scalar_op2` operations involve an array and a scalar.

The following routine performs a circular shift of a specified distance along a specified axis of an array:

```
S3L_cshift
```

The following routine applies a user-defined function to some or all elements of an array:

`S3L_forall`

The following routines perform a predefined reduction function over all elements of an array or along a specified axis of an array:

`S3L_reduce`
`S3L_reduce_axis`

The following routine sets all elements in an array to zero:

`S3L_zero_elements`

Copy Arrays

The following routine copies the contents of one parallel array into a second parallel array:

`S3L_copy_array`