S3L 2.0 User's Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300 fax 650 969-9131

Part No.: 805-1557-10
Revision A, November 1997

Contents

Preface v

1. Introduction to S3L 1-1

1.1 S3L Overview 1-1

1.2 Contents of S3L 1-3

1.2.1 Core Scientific Library Routines 1-3

1.2.2 Auxiliary S3L Functions 1-5

2. S3L Arrays 2-1

2.1 Describing S3L Arrays 2-1

2.2 S3L Array Handles 2-3

2.2.1 Axis and Coordinate Numbering 2-3

2.2.2 Row / Column Axis Descriptions 2-3

2.3 Types of Array Distribution 2-5

3. S3L Data Types 3-1

3.1 Sun HPF Data Types 3-1

3.2 C and F77 Data Types 3-2

4. Multiple Instance 4-1

4.1 Defining Multiple Independent Data Sets 4-2

4.2 Rules for Data Axes and Instance Axes 4-3

4.3 Specifying Single-Instance vs. Multiple-Instance Operations 4-4
4.3.1 Example 1: Matrix-Vector Multiplication 4-4
4.3.2 Example 2: Fast Fourier Transforms 4-8

5. Using S3L 9

5.1 Creating a Program that Calls S3L Routines 9
5.1.1 Include the S3L Header File 10
5.1.2 Compiling and Linking 10
5.1.3 Executing S3L Programs 11
5.1.4 Restriction 11
5.2 The S3L Safety Mechanism 11
5.2.1 Synchronization 12
5.2.2 Error Checking and Reporting 12
5.3 Levels of Error Checking 12
5.4 Selecting a Safety Mechanism Level 13
5.4.1 Setting the S3L Safety Environment Variable 14
5.4.2 Setting the Safety Level from Within a Program 14
5.5 Online Sample Code and Man Pages 14
5.5.1 Sample Code Directories 14
5.5.2 Compiling and Running the Examples 15
5.5.3 Man Pages 16

A. Summary of S3L Routines A-1

Index Index-1

Preface

This manual describes the Sun™ Scientific Subroutine Library (S3L). It is directed to anyone developing Sun HPF applications, as well as to developers of message-passing C or Fortran 77 programs.

Acknowledgments

The 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.)

For S3L applications with message-passing components, ScaLAPACK accesses 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.

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

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

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface or Symbol	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. Command-line variable; replace with a real name or value.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be `root` to do this. To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

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

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
Sun HPC Software News	*Sun HPC Software 2.0 Release Notes*	801-1562-10
Sun HPC Software Programming	*Sun HPC Software User's Guide*	801-1554-10
Sun MPI Programming	*Sun MPI User's Guide*	805-1556-10
Sun HPF Programming	*Sun HPF 1.0 Guide*	805-1558-10
Prism Development Environment	*Prism User's Guide* *Prism Reference Manual*	805-1552-10 805-1552-10

Ordering Sun Documents

SunDocs[SM] is a distribution program for Sun Microsystems technical documentation. Contact SunExpress for easy ordering and quick delivery. You can find a listing of

available Sun documentation on the World Wide Web.

TABLE P-4 SunExpress Contact Information

Country	Telephone	Fax
Belgium	02-720-09-09	02-725-88-50
Canada	1-800-873-7869	1-800-944-0661
France	0800-90-61-57	0800-90-61-58
Germany	01-30-81-61-91	01-30-81-61-92
Holland	06-022-34-45	06-022-34-46
Japan	0120-33-9096	0120-33-9097
Luxembourg	32-2-720-09-09	32-2-725-88-50
Sweden	020-79-57-26	020-79-57-27
Switzerland	0800-55-19-26	0800-55-19-27
United Kingdom	0800-89-88-88	0800-89-88-87
United States	1-800-873-7869	1-800-944-0661
World Wide Web: `http://www.sun.com/sunexpress/`		

Sun Documentation on the Web

The `docs.sun.com` web site enables you to access Sun technical documentation on the World Wide Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email or fax your comments to us. Please include the part number of your document in the subject line of your email or fax message.

- Email: `smcc-docs@sun.com`
- Fax: SMCC Document Feedback
  1-650-786-6443

LSF Technical Support

LSF 3.0, a product of Platform Computing Corporation, is part of the Sun HPC Software 2.0 Foundation Package. As such, it is supported by Sun as part of Sun HPC Software 2.0.

Sun HPC Software includes LSF Base and LSF Batch. However, LSF JobScheduler and LSF MultiCluster are not included and, therefore, not supported by Sun.

Information Sources for PVM and PETSc

TABLE P-5 lists organizations and resources for information about the publicly available libraries PVM and PETSc. This information is subject to change.

TABLE P-5 Information Sources for PVM and PETSc

Product	Contact
PVM	Copyright holders: University of Tennessee, Oak Ridge National Laboratory, Emory University Electronic mail: `pvm@msr.epm.ornl.gov` Newsgroup: `comp.parallel.pvm` Web site: `http://www.epm.ornl.gov/pvm/pvm_home.html`
PETSc	Developed and supported by the Mathematics and Computer Science Division of the Argonne National Laboratory.

CHAPTER 1

Introduction to S3L

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

1.1 S3L Overview

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 Ultra™ HPC Systems, from a single processor on an SMP, through multiple processors on a stand-alone SMP, to a cluster of SMPs.

The chief advantages of S3L are summarized below.

- S3L is optimized for Sun Ultra HPC systems.
- S3L functions have a simple array syntax interface that is callable from Sun's parallel Fortran language, Sun HPF, as well as from message-passing C and F77 programs.
- S3L supports multiple instances.
- S3L is thread safe.
- S3L uses the Sun Performance Library™ for nodal computation.
- S3L is supported by Sun.
- S3L includes built-in diagnostics.

S3L provides an *array syntax* interface that supports both Sun HPF and message-passing programs written in C or F77.

Sun HPF programs call S3L functions directly, passing distributed HPF arrays as arguments. This direct interface is possible because array syntax support is built into both Sun HPF and S3L. This means each Sun HPF call to an S3L subroutine contains all the array layout and shape information needed by S3L to operate on the distributed array.

Because array syntax is not inherent in C and F77, S3L uses structures called *array handles* to extend its array syntax support to message-passing programs written in these languages. The message-passing programmer simply creates and destroys array handles in the calling program as needed. Note that S3L array handles are analogous to the array descriptors found in the public domain packages ScaLAPACK and PETSc.

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

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

S3L routines 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 BLAS1, BLAS2, BLAS3, LINPACK, LAPACK, FFTPACK, and VFFTPACK. Besides providing appropriate nodal support to S3L, routines from the Sun Performance Library can be called independently from any F77 user code running locally on a Sun Ultra HPC Server node.

Note – The Sun Performance Library is made available to S3L users as part of either WorkShop Compilers Fortran v4.2 or Performance WorkShop Fortran v3.0.

S3L routines operate on objects of various data types. However, this information is encoded in the array handle and is decoded at run time, allowing 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 illustrate correct use of all S3L functions. These examples can be used as templates in developing actual code. Separate examples are provided to demonstrate Sun HPF, C (with Sun MPI), and F77 (with Sun MPI) interfaces.

1.2 Contents of S3L

S3L consists of a set of *core* library functions—that is, the subroutines that perform the linear algebra, Fourier transform, and related operations—plus a set of auxiliary utilities.

The core library functions are introduced in Section 1.2.1 and the auxiliary utilities in Section 1.2.2. All S3L functions and utilities are listed in Appendix A and are described in their online man pages.

1.2.1 Core Scientific Library Routines

The S3L core routines consist of:

- Dense-matrix operations
  - *2–Norm* – The 2–norm routines compute the global 2–norm of a parallel array.
  - *Inner product* – The inner-product routines compute the global inner product over all axes of two source parallel arrays. The inner product is added to the destination. A routine that takes the conjugate of the second operand is provided for complex data.
  - *Outer product* – The outer-product routines compute one or more instances of an outer product of two vectors. The result is added to the destination. For complex data, a routine that takes the conjugate of the second operand is provided.
  - *Matrix-vector multiplication* – The matrix-vector-multiplication routines compute one or more instances of a matrix-vector product. The result is added to the destination, or is added to a second parallel array. For complex data, a routine that takes the conjugate of the matrix is provided.
  - *Matrix multiplication* – The matrix-multiplication routines compute one or more matrix products. Each routine add the result to the destination. Routines that take the transpose of either or both operand matrices (or, for complex data, the Hermitian of either matrix) are provided.
- LU-factorization and LU-solve routines
  - *LU-factorization routine* – For each m x n coefficient matrix A of a, this routine computes LU factorization using partial pivoting with row interchanges.
  - *LU-solve routine* – This routine uses the L and U factors produced by the LU-factorization routine to produce solutions to the system AX=B. B may represent one or more right-hand sides for each instance of the systems of equations.

    - *LU-invert routine* – For each m x m (square) instance of matrix A, this routine computes the inverse of A using the LU-factorization results of the `S3L_lu_factor` routine.
- Parallel 1D, 2D, and 3D FFTs
    - *Setup and deallocation of FFT handles* – Routines are provided to initialize and deallocate FFT handles for both complex and real data types. Separate routines are used for the two data types.
    - *Simple complex-to-complex, mixed-radix, forward and inverse FFT routines* – Performs the forward or inverse Fast Fourier Transform of a parallel array of type complex or double complex. Supports both power-of-two and arbitrary radix parameters.
    - *Detailed complex-to-complex FFT routine* – Allows independent specification along each data axis of the transform direction in a complex-to-complex FFT. Can improve performance over the simple FFT in some cases.
    - *Simple real-to-complex and complex-to-real FFT routines* – Perform the forward (real-to-complex) and inverse (complex-to-real) FFT operations on 1-, 2-, or 3-dimensional arrays.
- Parallel random number generators
    - *Fibonacci RNG setup and deallocation* – Routines to initialize and deallocate the state table of a lagged Fibonacci random number generator (LFG).
    - *Fibonacci RNG* – Uses an LFG to initialize a parallel array.
    - *Random LCG setup* – Routine to define the parameters used in the S3L linear congruential random number generator (LCG).
    - *Random LCG* – Uses a parallel LCG to produce random numbers that are independent of the array distribution.
- *Parallel sort* – Sorts a 1D parallel array.
- *Parallel transpose* – Performs a generalized transposition of a parallel array.
- *Copy array routine* – Copies the elements of one array onto another.
- *Initialize (or exit) S3L environment* – Sets up the S3L environment before (and deallocates after) an application uses S3L routines.

1.2.2 Auxiliary S3L Functions

S3L also includes a variety of functions that augment use of the core routines. These are summarized below.

- *Message-passing interface* – Allows message-passing programs to use S3L routines on parallel arrays.
- *Thread safety* – Supports thread-safe use of S3L routines in multithreaded operations.
- *Safety mechanism* – The S3L safety mechanism synchronizes nodes to allow tracking of S3L activity. It also performs error checking and reporting at different levels of detail.

CHAPTER 2

S3L Arrays

In S3L, the programmer is presented with a single conception of an array, called the *parallel array*. This term simply means that all processes executing the program in which the array is declared have a global view of the array. That is, no matter how the array has been distributed, S3L ensures that its layout is understood by all participating processes.

This simplified view of S3L arrays is convenient when an application is moved from a partition with *M* processes to a partition with *N* processes. Even though the arrays may have drastically different physical distributions from one run to another, no changes to the application are needed.

2.1 Describing S3L Arrays

In a multiprocess environment, where different parts of the array may be stored in different processes, there are many ways of describing how the array is distributed.

In the most general case, two numbers are needed to describe where each element of the array is stored:

- The process where the element is stored
- Its memory location in the given process

The size of such a handle would be of the order of the size of the total parallel array and, since every process would need to know where every array element is stored, the total memory requirements for such a handle would be of the order of the array size times the number of processes.

For array handles to be of a practical size, the space allocated to parallel array distributions must be restricted. In S3L, every axis (dimension) of a parallel array is distributed along a certain number of processes. This distribution is identified by the extent of the array along the particular axis, and the block size of the distribution.

If the array extent along a particular axis is n, then the set of array indices numbered 0 to n – 1 is partitioned into blocks. If the block size is b, block i contains the indices i * b to (i + 1) * b – 1. There are ⌈n/b⌉ such blocks, where ⌈⌉ denotes truncation toward the next larger integer.

Every block always contains b indices, except possibly for the last block, which may contain a smaller number of indices if n is not exactly divisible by the number of processes along the particular axis. Assume that the given axis is distributed over p processes, identified by the numbers 0 to p – 1.

In a block-cyclic distribution, the first block (block 0) is assigned to process 0, the second to process 1, etc. When the blocksize exceeds the number of processes along an axis, block p is assigned to process 0. In general, block i is assigned to process mod(⌈n/b⌉,p), where mod denotes the modulo operation.

In a slightly more general case, instead of assigning block 0 to process 0, it could be assigned, for example, to process q, 0 <= q < p. Block 1 would be assigned to process q + 1, etc. The number of the process to which block 0 is assigned is the *starting process* of the block-cyclic distribution.

By specifying the extent of the array, the block size, the number of processes, and the starting process of the block cyclic distribution along each axis, you are fully specifying the distribution of the array. Elements whose indices differ only along a single axis are stored to the appropriate processes as determined by the parameters of the block cyclic distribution along that axis.

Block-cyclic distribution is more general than a simple block distribution, where a given array axis is partitioned into exactly *p* blocks, where *p* is the number of processes along that axis. For an array axis whose extent is *n* and is to be distributed along *p* processes, the block distribution is equivalent to a block-cyclic distribution with blocksize ⌈*n*/*p*⌉.

The block-cyclic distribution scheme can also describe axes that are in fact local to a process—that is, all array elements whose coordinates differ only along the particular axis (direction) are in the same process. If the block size along an axis is equal to the corresponding extent of the array, then this axis becomes local.

Because S3L supports block-cyclic distribution of parallel arrays, it offers considerable flexibility in how the arrays can be distributed and in the algorithms used to perform the various calculations. For example, to compute the LU decomposition of a parallel array, it is more efficient to distribute the array in a block-cyclic fashion and to set the block size to a value that depends on the process characteristics (such as cache size). For other computations, such as FFTs, it is more efficient to have the array distributed in a pure block fashion, with all axes except the last being local to the processes.

Whenever it is more efficient to use a different array distribution than that of a given array, S3L internally redistributes the array to the new distribution. After the computation is done, S3L restores it to its original distribution.

2.2 S3L Array Handles

This section explains how S3L array handles are used to characterize parallel arrays—that is, it defines the kind of information about arrays that these handles convey.

In S3L, a parallel array is internally described by an S3L array handle. This handle defines the universal properties of the array—that is, a set of properties that belong to all processes associated with the parallel array. Examples of these general properties are type, size, and distribution.

An S3L array handle also defines a set of properties that are specific to individual processes. These *local* properties include such parameters as the extents of the local part (subgrid) of the array and its memory location within the local process.

An S3L array handle is always defined in all participating processes—that is, every process has access to the universal information needed to perform certain operations on the parallel array.

2.2.1 Axis and Coordinate Numbering

The Fortran and C interfaces follow different numerical-base conventions for numbering axes and coordinates in arrays; each adheres to its respective language conventions. Consequently, the following axis and coordinate numbering rules apply:

- The Fortran interface expects array-axis and coordinate numbering to be one-based. This means the lowest number for an axis of an array is axis 1. Likewise, the minimum value for variables that take coordinate values is 1.
- The C interface expects zero-based axis numbers and coordinates. Therefore, the lowest number for an axis of a C parallel array is axis 0. Likewise, variables that take coordinate values have 0 as the minimum value.

2.2.2 Row/Column Axis Descriptions

In descriptions of Fortran and C parallel arrays, row and column axes are distinguished as follows:

- "The axis that counts the rows," "the row axis," and *row_axis* refer to axis 1 in FIGURE 2-1 (Fortran) and axis 0 in FIGURE 2-2 (C).

- "The axis that counts the columns," "the column axis," and *column_axis* refer to axis 2 in FIGURE 2-1 (Fortran) and axis 1 in FIGURE 2-2 (C).

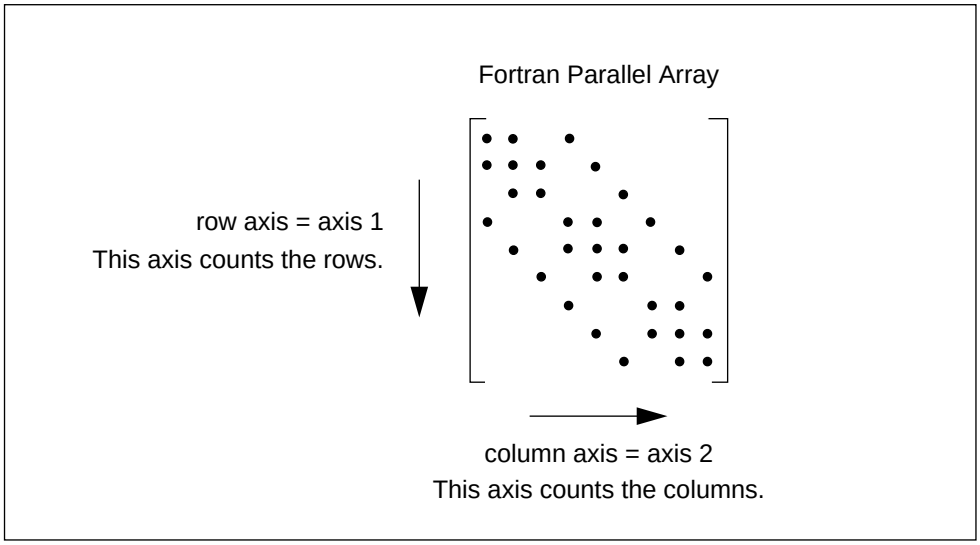

FIGURE 2-1 Row and Column Axes—Fortran

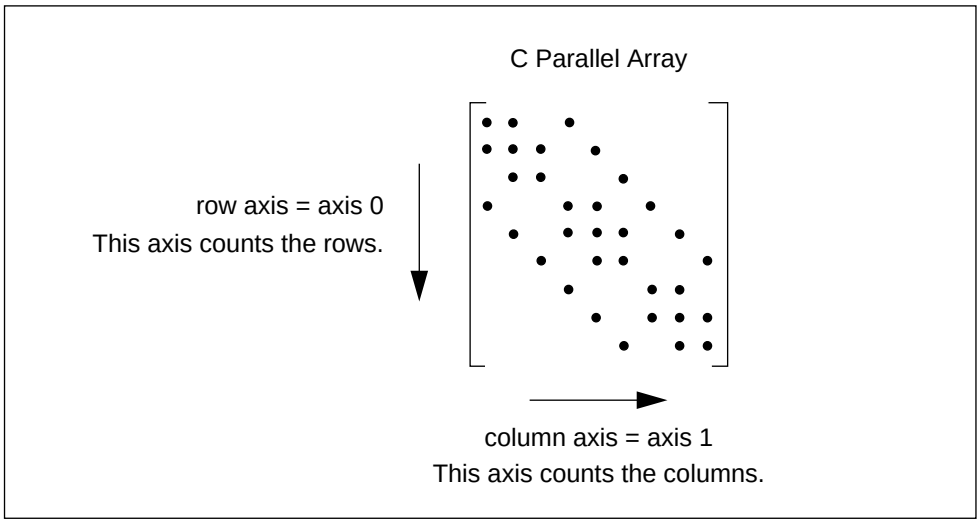

FIGURE 2-2 Row and Column Axes—C

2.3 Types of Array Distribution

Arrays passed to S3L routines by C or F77 message-passing programs can have block, cyclic, or block-cyclic distributions. Regardless of the type of distribution specified by the calling program, S3L will automatically select the distribution scheme that is most efficient for the routine being called. If that means S3L changes the distribution type internally, it will restore the original distribution scheme on the resultant array before passing it back to the calling program.

Arrays from C and F77 message-passing programs can also be undistributed. That is, all the elements of the array can be located on the same process—a *serial* array in the conventional sense.

Within Sun HPF programs, array distribution is specified separately for each axis of the array. Unless explicitly specified otherwise, each axis of a Sun HPF array is distributed in block fashion over the available processes. For example, the default distribution scheme for a 2D array is (block, block).

Currently, Sun HPF supports two alternatives to block distribution: block(*N*) and *collapsed*.

- A block(*N*) distribution is a more detailed from of block distribution. It allows the programmer to specify the number of axis subscripts in the *typical* block for a given axis. If the number of processes does not divide evenly into the number of indices in the block(*N*)-distributed axis, this number will be less at the end of the axis.
- A collapsed axis is one that is kept in a single block rather than being partitioned into blocks. Sun HPF replicates a collapsed axis onto every available process. Consequently, it is regarded as a purely local axis rather than a distributed parallel axis.

In Sun HPF documentation, an array that has all its axes collapsed is referred to as a *serial* array. A Sun HPF serial array is kept intact and replicated on all the available processes.

Note – S3L does not support serial arrays passed by Sun HPF programs. In other words, S3L requires that at least one axis of a Sun HPF array must be block- or block(*N*)-distributed.

CHAPTER 3

S3L Data Types

The C and F77 language interfaces implement different data types from those recognized by Sun HPF. This chapter describes these differences.

Note – For C and F77 message-passing applications, data type information is encoded in the array handle and is decoded at run time, allowing appropriate branching to occur during execution. Consequently, there is no need for separate routines with different names to implement the different data types. For each S3L function, a single routine supports all types.

3.1 Sun HPF Data Types

S3L supports the following data types for Sun HPF programs:

```
real*4
real*8
complex*8
complex*16
```

Place the following line at the top of any Sun HPF program unit that makes an S3L call.

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

3.2 C and F77 Data Types

TABLE 3-1 shows the data types supported for the C and Fortran 77 interfaces. Within each subroutine call, elements of all array arguments must match in data type, unless the argument descriptions indicate otherwise. In this table, *complex* refers to the data type `S3L_complex_t` and *dcomplex* refers to the data type `S3L_dcomplex_t`.

Place one of the following `#include` lines at the top of any C or F77 program unit that makes an S3L call:

```
#include <s3l/s3l-c.h>      C programs
#include <s3l/s3l-f.h>      F77 programs
```

TABLE 3-1 Data Types Supported by S3L for Fortran 77 and C

	Data Type				
Operation	**int**	**float**	**double**	**complex**	**dcomplex**
Inner product		x	x	x	x
2–norm		x	x	x	x
Outer product		x	x	x	x
Matrix vector multiplication		x	x	x	x
Matrix multiplication		x	x	x	x
LU factor		x	x	x	x
LU solve		x	x	x	x
LU invert		x	x	x	x
FFT setup				x	x
FFT deallocate setup				x	x
Simple complex-to-complex FFT				x	x
Detailed complex-to-complex FFT				x	x
Simple real-to-complex FFT				x	x
Simple complex-to-real FFT				x	x
RNG Fibonacci	x	x	x	x	x
RNG LCG	x	x	x	x	x
Sort	x	x	x		
Transpose	x	x	x	x	x
Copy array	x	x	x	x	x

CHAPTER 4

Multiple Instance

Most S3L routines support *multiple instances;* that is, they allow you to perform multiple independent operations on different data sets concurrently. TABLE 4-1 shows which operations currently support multiple instances.

TABLE 4-1 S3L Support for Multiple Instances

Operation	Instances	
	Single	Multiple
inner product	x	x
2–norm	x	x
outer product	x	x
matrix-vector multiplication	x	x
matrix multiplication	x	x
LU factor	x	x
LU solve	x	x
LU invert	x	x
simple complex-to-complex FFT	x	
detailed complex-to-complex FFT	x	x
inverse FFT	x	
simple real-to-complex FFT	x	
simple complex-to-real FFT	x	
RNG Fibonacci	x	
RNG LCG	x	
sort	x	
copy array	x	

4.1 Defining Multiple Independent Data Sets

To perform an S3L operation on multiple independent data sets concurrently, you must embed the multiple independent instances of each operand or result argument in a parallel array. The axes of the shape of the parallel array fall into two distinct groups:

- The *data axes* define the geometry of the individual instances of the operand or result.
- The *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.

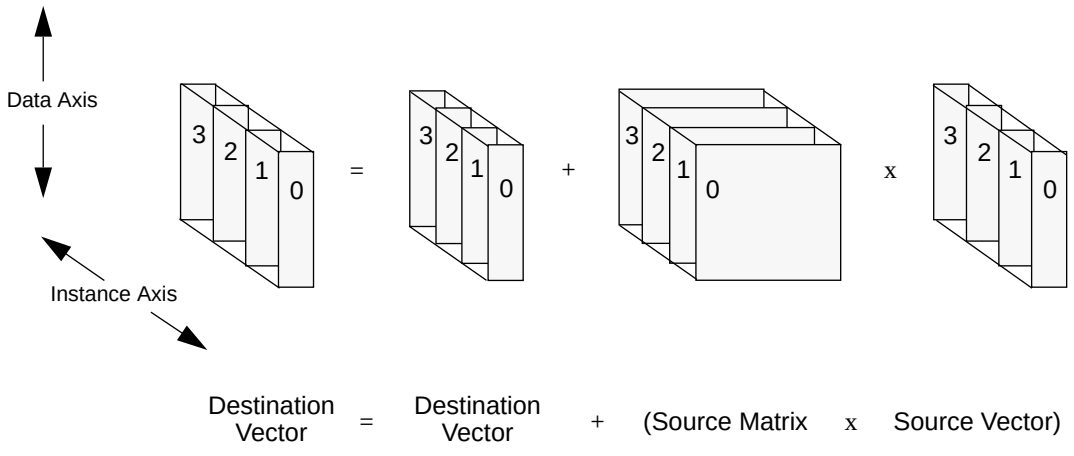

FIGURE 4-1 A Multiple-Instance Matrix-Vector Multiplication Problem

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.

4.2 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 axes of the parallel arrays, starting with axis 0 and 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. In some cases, 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, 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 S3L routines contain specific information about optimizing layouts.

Section 4.3, “Specifying Single-Instance vs. Multiple-Instance Operations,” illustrates these rules being applied in a matrix-vector multiplication example.

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

4.3 Specifying Single-Instance vs. Multiple-Instance Operations

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.

Section 4.3.1, "Example 1: Matrix-Vector Multiplication," explains the differences between single- and multiple-instance operation for the matrix-vector-multiplication routine. Section 4.3.2, "Example 2: Fast Fourier Transforms," discusses use of multiple instances in FFTs.

4.3.1 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 HPF form of this 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.

4.3.1.1 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 HPF can be performed by first defining the block-distributed arrays

```
    real, dimension(p, q) :: a
    real, dimension(q) :: x
    real, dimension(p) :: y
!hpf$ distribute a(block, block)
!hpf$ distribute x(block)
!hpf$ distribute y(block)
```

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 in F77 is

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

4.3.1.2 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.

```
    real, dimension(p, q, r) :: a
    real, dimension(q, r) :: x
    real, dimension(p, r) :: y
!hpf$ distribute a(block, block, block)
!hpf$ distribute x(block, block)
!hpf$ distribute y(block, block)
```

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 axes 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 will be evaluated:

- The values of *x_vector_axis* = 1 and *col_axis* = 2 indicate that the product **a**`(i, j, s0)` * **x**`(j, s0)` will be calculated for all `j`.
- 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 processors evaluates the *N*th instance at the same time—that is, in parallel.

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

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 (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. Use of this data layout is discussed below.

Multiple-instance operations are usually most efficient when each cell (all of the data axes) resides on one process. Use of such a layout scheme is discussed in this section. In addition, the use of several instance axes are illustrated. Declarations of arrays containing these axes can be done as

```
    real, dimension(p, q, k, m, n) :: a
    real, dimension(q, k, m, n) :: x
    real, dimension(p, k, m, n) :: y

!hpf$ distribute a (*, *, block, block, block)
!hpf$ distribute x (*, block, block, block)
!hpf$ distribute y (*, block, block, block)
```

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

To analyze the results of the call

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

we shall use `s0`, `s1`, and `s2` to denote the index along each of the three instance axes. For a given set of `s0`, `s1`, and `s2`, the following will be evaluated:

- The values of *x_vector_axis* = 1 and *col_axis* = 2 indicate that the product **`a(i, j, s0, s1, s2)`** * **`x(j, s0, s1, s2)`** will be calculated for all **`j`**.
- 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 `matvec_mult.hpf`. This can be found in the S3L examples directory `examples/dense_matrix_ops-hpf/`, the location of which is site-specific.

4.3.2 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. You can also specify axes along which no transform is performed, but address bits are reversed. The axes that are transformed or bit-reversed are the data axes, and define the cell; the axes along which you perform no transformation 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 S3L

This chapter explains how to implement calls to S3L routines into your Sun HPF, F77, or C program. The following topics are included:

- Creating a program that calls S3L routines
- Restriction
- The S3L safety mechanism
- Online sample code and man pages

S3L documentation includes sample online programs that demonstrate how to call each S3L routine. You are encouraged to experiment with these sample programs. Online man pages are also included for all S3L routines. Section 5.5, "Online Sample Code and Man Pages," explains how to find the program examples.

5.1 Creating a Program that Calls S3L Routines

▼ To use S3L routines in a program:

1. **Place calls to S3L routines into your code.**
2. **Include the appropriate header file in each program unit that calls S3L routines.**
   See Section 5.1.1, "Include the S3L Header File," for details.

3. **Use the appropriate compiler command to compile your code; include the S3L link switch on the command line.**

   See Section 5.1.2, “Compiling and Linking,” for details.

The remainder of this section describes the steps listed above more fully.

Note – S3L requires the presence of the Sun Performance Library routines and its associated license file. This library is not installed with the S3L and other Sun HPC Software. Instead, it is included as part of either the WorkShop Compilers Fortran v4.2 or Performance WorkShop Fortran v3.0 languages system.

Note – Use `libsunperf` 1.2 instead of `libsunperf` 1.1. It will provide better performance.

5.1.1 Include the S3L Header File

Place the appropriate include line at the top of any program unit that makes an S3L call. The correct include file is shown below for each S3L language interface is

- For S3L + Sun HPF

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

- For S3L + Sun MPI + F77

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

- For S3L + Sun MPI + C

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

This allows the program to access the header file that contains prototypes of the routines and defines the symbols and data types required by the interface.

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

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

If the file appears to be missing, consult your system administrator.

5.1.2 Compiling and Linking

Compile your program and link in S3L (along with any other libraries it needs).

Note – S3L provides a link-line switch that does more than just link in S3L subroutines. Depending on which compiler has been invoked, it also automatically links any other libraries needed to augment S3L, greatly simplifying the link line.

- For S3L + Sun HPF

% **hpf -o** *program program*.**hpf -ls3l**

- For S3L + MPI + F77

% **tmf77 -o** *program program*.**f -ls3l**

- For S3L + Sun MPI + C

% **tmcc -o** *program program*.**c -ls3l**

5.1.3 Executing S3L Programs

Execute a program that has been linked with S3L just as you would any other program compiled for running on a Sun HPC System. For example, to execute the program `a.out` interactively, simply enter

% **tmrun a.out**

Or, you can submit the program to a batch queue by entering

% **tmsub a.out**

Refer to the *Sun HPC User's Guide* for complete instructions on program execution on a Sun HPC System.

5.1.4 Restriction

S3L does not accept sections of parallel arrays. You cannot pass a section of an HPF array to a S3L routine. This restriction is imposed because array sections may not be appropriately aligned with the underlying process grid.

5.2 The S3L Safety Mechanism

The S3L safety mechanism offers two basic features: It synchronizes the parallel processes so that you can pinpoint the area of code that generated an error. It also performs error checking and reports errors at a user-selectable level of detail.

5.2.1 Synchronization

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

5.2.2 Error Checking and Reporting

The safety mechanism can perform error checking and generate run-time error information at multiple levels of detail. You can turn safety checking on at any level during all or part of a program. One level checks for errors in the usage and arguments of the S3L calls in your program; a more detailed level also checks for errors generated by internal 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).

5.3 Levels of Error Checking

The S3L safety mechanism has four selectable levels: 0, 2, 5, and 9. These levels are defined in TABLE 5-1.

At levels 2, 5, and 9, some safety mechanism error messages are displayed at the terminal when you run the program; other information appears in the backtrace when you use a debugger such as Prism.

TABLE 5-1 S3L Safety Mechanism Levels

0	Turns off the safety mechanism. Explicit synchronization and error checking are not performed. This level is appropriate for production runs of code that has already been thoroughly tested.
2	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	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	Checks for and reports all level 2 and level 5 errors, as well as errors generated by lower levels of code that were 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 performs all implemented error checking and exacts a very high performance price. It is appropriate for detailed debugging when a problem occurs.

5.4 Selecting a Safety Mechanism Level

You can select the desired S3L safety mechanism level in either of two ways:

- By setting the environment variable *S3L_SAFETY*
- By using the subroutine calls `S3L_get_safety` and `S3L_set_safety` in a program

These methods are described in Section 5.4.1, “Setting the S3L Safety Environment Variable,” and Section 5.4.2, “Setting the Safety Level from Within a Program.”

5.4.1 Setting the S3L Safety Environment Variable

The *S3L_SAFETY* environment variable takes a single argument, which can be the integer `0`, `2`, `5`, or `9`. For example, to select the highest level, enter:

```
% setenv S3L_SAFETY 9
```

One advantage of using the *S3L_SAFETY* environment variable is that you can set or change the safety level without recompiling your code.

5.4.2 Setting the Safety Level from Within a Program

To set the S3L safety level from within your program, include the following subroutine call. Specify the desired level in the integer argument *n*:

```
S3L_set_safety (n);
```

To see what S3L safety level is currently in effect, include the following call:

```
n = S3L_get_safety();
```

The advantage of using these calls from within a program is that you can set or obtain the safety level at any point within your code. However, you must recompile the code each time you change these calls.

5.5 Online Sample Code and Man Pages

5.5.1 Sample Code Directories

The online sample programs are located in subdirectories of the S3L examples directory. Separate Sun HPF, C, and F77 versions are provided. The generic relative path for these examples is

`examples/`*operation_class[–language_suffix]/example_name.language*

where `examples` is installed in a site-specific location.

operation_class is the name of the general class of S3L routines that are illustrated by the example.

The *–language_suffix* is used to denote either Sun HPF or F77 language-specific implementations of these general classes. Examples implemented in C are do not include a *–language_suffix*.

example_name.language is the name given to the example. The *language* extension is `.hpf`, `.c`, or `.f`. For example,

```
examples/dense_matrix_ops-hpf/outer_prod.hpf
```

is the Sun HPF version of a program example that illustrates use of `s3l_outer_prod` routines. The equivalent examples for C and F77 applications are

`examples/dense_matrix_ops/outer_prod.c`	(C with Sun MPI)
`examples/dense_matrix_ops-f/outer_prod.f`	(F77 with Sun MPI)

In addition to these routine-oriented examples, S3L provides an on-line application example as well. The directory `../examples/s3l/spectral-hpf` shows the use of S3L Fast Fourier Transform (FFT) routines to solve the Navier-Stokes equations in two dimensions with periodic boundary conditions. A result example is provided in a `data` subdirectory, and a visualization of the simulation output is provided in the `vizlab` subdirectory.

5.5.2 Compiling and Running the Examples

Each example subdirectory has a makefile. Each makefile references the file `../Make.simple` (for example, `examples/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 you with highly optimized executables. Information on optimization flags is best obtained from the documentation for the compiler of interest.

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,

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

will start your runs on 4 processes.

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

5.5.3 Man Pages

To read the online man page for a routine, enter

```
% man routine_name
```

APPENDIX **A**

Summary of S3L Routines

This appendix lists, in alphabetical order, the routines that make up the Sun Scientific Subroutine Library. Each listing shows the C and Fortran language syntax for their respective argument sequences. Detailed descriptions are for all these routines are available in their respective man pages..

Note – Some of the descriptions of linear algebra routines in the man pages include information about *numerical stability*. These references imply a conventional definition of numerical stability. That is, an algorithm is stable if the computed result is the exact solution of a slightly different problem. For example, if **A** is the input matrix, the computed result is the true result corresponding to the matrix **A**+**E**, where **E** is small in norm compared with **A**.[1] Most of the algorithms used by S3L are numerically stable in this sense. However, a few are only *conditionally stable*, which means that the numerical stability may depend on the condition number of the problem.

1. For a more formal definition, see G.H Golub and C. F. Van Loan, *Matrix Computations*, 2d ed. (Baltimore: Johns Hopkins University Press, 1989).

▼ 2-Norm

C Syntax

```
S3L_2_norm (z, x, x_vector_axis)

S3L_gbl_2_norm(a, x)
```

Fortran Syntax

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

S3L_gbl_2_norm(a, x, ier)
```

▼ Copy Array

C Syntax

```
S3L_copy_array(A, B)
```

Fortran Syntax

```
S3L_copy_array(A, B, ier)
```

▼ FFT Detailed

C Syntax

```
S3L_fft_detailed(a, setup_id, iflag, axis)
```

Fortran Syntax

```
S3L_fft_detailed(a,setup_id,iflag,axis,ier)
```

▼ FFT Free Setup

C Syntax

```
S3L_fft_free_setup(setup_id)
```

Fortran Syntax

```
S3L_fft_free_setup(setup_id,ier)
```

▼ FFT Setup

C Syntax

```
S3L_fft_setup(a, setup_id)
```

Fortran Syntax

```
S3L_fft_setup(a, setup_id, ier)
```

▼ FFT Simple

C Syntax

```
S3L_fft(a, setup_id)
```

Fortran Syntax

```
S3L_fft(a, setup_id, ier)
```

▼ Free RNG Fibonacci

C Syntax

```
S3L_free_rand_fib(setup_id)
```

Fortran Syntax

```
S3L_free_rand_fib(setup_id, ier)
```

▼ Get Safety

C Syntax

```
safety_level = S3L_get_safety()
```

Fortran Syntax

```
S3L_get_safety()
```

▼ Inner Product

C Syntax

```
S3L_inner_prod(z, y, x_vector_axis, y_vector_axis)
S3L_inner_prod_noadd(z, x, x_vector_axis, y_vector_axis)
S3L_inner_prod_addto(z, x, y, u, x_vector_axis,  y_vector_axis)
S3L_inner_prod_c1(z, x, y, x_vector_axis, y_vector_axis)
S3L_inner_prod_c1_noadd(z, x, y, x_vector_axis, y_vector_axis)
S3L_inner_prod_c1_addto(z, x, y, u, x_vector_axis,
y_vector_axis)
S3L_gbl_inner_prod(a, x, y)
S3L_gbl_inner_prod_noadd(a, x, y)
S3L_gbl_inner_prod_a, x, y, b)
S3L_gbl_inner_prod_c1(a, x, y)
S3L_gbl_inner_prod_c1_noadd(a, x, y)
S3L_gbl_inner_prod_c1_addto(a, x, y, b)
```

Fortran Syntax

```
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)

S3L_inner_prod_c1(z, x, y, x_vector_axis,  y_vector_axis, ier)

S3L_inner_prod_c1_noadd(z, x, y, x_vector_axis,
y_vector_axis,ier)

S3L_inner_prod_c1_addto(z, x, y, u, x_vector_axis,
y_vector_axis,ier)

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)

S3L_gbl_inner_prod_c1(a, x, y, ier)

S3L_gbl_inner_prod_c1_noadd(a, x, y, ier)

S3L_gbl_inner_prod_c1_addto(a, x, y, b, ier)
```

▼ Inverse FFT

C Syntax

```
S3L_ifft(a, setup_id)
```

Fortran Syntax

```
S3L_ifft(a, setup_id, ier)
```

▼ LU Deallocate

C Syntax

```
S3L_lu_deallocate(setup_id)
```

Fortran Syntax

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

▼ LU Factor

C Syntax

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

Fortran Syntax

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

▼ LU Invert

C Syntax

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

Fortran Syntax

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

▼ LU Solve

C Syntax

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

Fortran Syntax

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

▼ Matrix Multiplication

C Syntax

```
S3L_mat_mult(C, A, b, row_axis, col_axis)
S3L_mat_mult_noadd(C, a, b, row_axis, col_axis)
S3L_mat_mult_addto(C, a, b, D, row_axis, col_axis)
```

```
S3L_mat_mult_t1(c, a, b, row_axis, col_axis)
S3L_mat_mult_t1_noadd(c, a, b, row_axis, col_axis)
S3L_mat_mult_t1_addto(c, a, b, D, row_axis, col_axis)
S3L_mat_mult_h1(C, a, b, row_axis, col_axis)
S3L_mat_mult_h1_noadd(c, a, b, row_axis, col_axis)
S3L_mat_mult_h1_addto(c, a, b, D, row_axis, col_axis)
S3L_mat_mult_t2(c, a, b, row_axis, col_axis)
S3L_mat_mult_t2_noadd(c, a, b, row_axis, col_axis)
S3L_mat_mult_t2_addto(c, a, b, D, row_axis, col_axis)
S3L_mat_mult_h2(c, a, b, row_axis, col_axis)
S3L_mat_mult_h2-noadd(c, a, b, row_axis, col_axis)
S3L_mat_mult_h2_addto(C, a, b, D, row_axis, col_axis)
S3L_mat_mult_t1_t2(c, a, b, row_axis, col_axis)
S3L_mat_mult_t1_t2_noadd(c, a, b, row_axis, col_axis)
S3L_mat_mult_t1_t2_addto(c, a, b, D, row_axis, col_axis)
```

Fortran Syntax

```
S3L_mat_mult(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_noadd(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_addto(C, A, B, D, row_axis, col_axis, ier)
S3L_mat_mult_t1(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_t1_noadd(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_t1_addto(C, A, B, D, row_axis, col_axis, ier)
S3L_mat_mult_h1(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_h1_noadd(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_h1_addto(C, A, B, D, row_axis, col_axis, ier)
S3L_mat_mult_t2(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_t2_noadd(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_t2_addto(C, A, B, D, row_axis, col_axis, ier)
S3L_mat_mult_h2(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_h2_noadd(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_h2_addto(C, A, B, D, row_axis, col_axis, ier)
```

```
S3L_mat_mult_t1_t2(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_t1_t2_noadd(C, A, B, row_axis, col_axis, ier)
S3L_mat_mult_t1_t2_addto(C, A, B, D, row_axis, col_axis, ier)
```

▼ Matrix-Vector Multiplication

C Syntax

```
S3L_mat_vec_mult(y, A, x, y_vector_axis, row_axis, col_axis,
x_vector_axis)
S3L_mat_vec_mult_noadd(y, A, x, y_vector_axis, row_axis,
col_axis, x_vector_axis)
S3L_mat_vec_mult_addto(y, A, x, v, y_vector_axis, row_axis,
col_axis, x_vector_axis)
S3L_mat_vec_mult_c1(y, A, x, y_vector_axis, row_axis, col_axis,
x_vector_axis)
S3L_mat_vec_mult_c1_noadd(y, A, x, y_vector_axis, row_axis,
col_axis, x_vector_axis)
S3L_mat_vec_mult_c1_addto(y, A, x, v, y_vector_axis, row_axis,
col_axis, x_vector_axis)
```

Fortran Syntax

```
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)
S3L_mat_vec_mult_c1(y, A, x, y_vector_axis, row_axis, col_axis,
x_vector_axis, ier)
S3L_mat_vec_mult_c1_noadd(y, A, x, y_vector_axis, row_axis,
col_axis, x_vector_axis, ier)
S3L_mat_vec_mult_c1_addto(y, A, x, v, y_vector_axis, row_axis,
col_axis, x_vector_axis, ier)
```

▼ Outer Product

C Syntax

```
S3L_outer_prod(A, x, y, row_axis, col_axis, x_vector_axis,
y_vector_axis)

S3L_outer_prod_noadd(A, x, y, row_axis, col_axis, x_vector_axis,
y_vector_axis)

S3L_outer_prod_addto(A, x, y, B, row_axis, col_axis,
x_vector_axis, y_vector_axis)

S3L_outer_prod_c2(A, x, y, row_axis, col_axis, x_vector_axis,
y_vector_axis)

S3L_outer_prod_c2_noadd(A, x, *y, row_axis, col_axis,
x_vector_axis, y_vector_axis)

S3L_outer_prod_c2_addto(A, x, y, B, row_axis, col_axis,
x_vector_axis, y_vector_axis)
```

Fortran Syntax

```
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)

S3L_outer_prod_c2(A, x, y, row_axis, col_axis, x_vector_axis,
y_vector_axis, ier)

S3L_outer_prod_c2_noadd(A, x, y, row_axis, col_axis,
x_vector_axis, y_vector_axis, ier)

S3L_outer_prod_c2_addto(A, x, y, B, row_axis, col_axis,
x_vector_axis, y_vector_axis, ier)
```

▼ Random Fibonacci

C Syntax

```
S3L_rand_fib(a, setup_id)
```

Fortran Syntax

```
S3L_rand_fib(a, setup_id, ier)
```

▼ Random LCG

C Syntax

```
S3L_rand_lcg(a, iseed)
```

Fortran Syntax

```
S3L_rand_lcg(a, iseed, ier)
```

▼ Real FFT

C Syntax

```
S3L_rc_fft_setup(a, *setup_id)
S3L_rc_fft(a, setup_id)
S3L_cr_fft(a, setup_id)
S3L_rc_fft_free_setup(setup_id)
```

Fortran Syntax

```
S3L_rc_fft_setup(a, setup_id, ier)
S3L_rc_fft(a, setup_id, ier)
S3L_cr_fft(a, setup_id, ier)
S3L_rc_fft_free_setup(setup_id, ier)
```

▼ Set Safety

C Syntax

```
S3L_set_safety(n)
```

Fortran Syntax

```
S3L_set_safety(n)
```

▼ Setup Random Fibonacci

C Syntax

```
S3L_setup_rand_fib(setup_id, seed)
```

Fortran Syntax

```
S3L_setup_rand_fib(setup_id, seed, ier)
```

▼ Sort

C Syntax

```
S3L_sort(a)
```

Fortran Syntax

```
S3L_sort(a, ier)
```

▼ Transpose

C Syntax

```
S3L_trans(a, b, axis_perm)
```

Fortran Syntax

```
S3L_trans(a, b, axis_perm, ier)
```

Index

NUMERICS

2-norm, thumbnail 1-3

A

array coordinate numbering
  C and F77 conventions 2-3
array copy, thumbnail 1-4
array handles 1-2
auxiliary S3L functions 1-5
  message-passing interface, thumbnail 1-5
  safety mechanism, thumbnail 1-5
  thread safety, thumbnail 1-5
axis numbering
  C and F77 conventions 2-3

C

calling S3L routines
  summary 9
code examples
  running 15
code examples, path names 14
compiling S3L programs 10
copy array, thumbnail 1-4
core library routines 1-3

D

data types 3-1
  C and F77 3-2
  Sun HPF 3-1
dense matrix operations, summary 1-3

E

executing S3L programs 11

F

FFT setup and deallocation, thumbnail 1-4
FFT, detailed complex to complex, thumbnail 1-4
FFT, simple forward and inverse, thumbnail 1-4
FFT, simple real-to-complex, thumbnail 1-4
FFTs, 1D, 2D, 3D, summary 1-4
Fibonacci RNG, thumbnail 1-4

I

initialize S3L environment 1-4
inner product, thumbnail 1-3

L

LCG, setup and deallocation
  thumbnail 1-4
LCG, thumbnail 1-4
linear congruential RNG, thumbnail 1-4
linking S3L 10
LU factorization, thumbnail 1-3
LU invert, thumbnail 1-4
LU routines, summary 1-3
LU solve, thumbnail 1-3

M

man pages on-line 16
matrix multiplication, thumbnail 1-3
matrix-vector multiplication, thumbnail 1-3

message-passing interface, thumbnail 1-5
multiple instance 1-2
  list of S3L functions supporting 4-1
multiple instances
  cells, organizing data to form 4-3
  cells, size of 4-3
  cells, the basic unit 4-3
  data axes, rules for 4-3
  determining number of instances 4-3
  frames, cells embedded in 4-3
  frames, organizing data to form 4-3
  how to specify 4-2
  instance axes, rules for 4-3
  matrix vector multiplication example
    multiple instance 4-5
    single instance 4-4
multithread support 1-2

O

on-line examples
  running 15
on-line examples, path names 14
outer product, thumbnail 1-3

P

parallel arrays
  axis and coordinate numbering 2-3
parallel sort, thumbnail 1-4
parallel transpose, thumbnail 1-4

R

random LCG
  thumbnail 1-4
random number generators, summary 1-4
restrictions 11
RNG, Fibonacci, thumbnail 1-4
RNG, setup and deallocation, thumbnail 1-4
row, column axis descriptions 2-3
  C and F77 conventions 2-3

S

S3L array handles 2-3
S3L header files 10
  for C plus Sun MPI 10
  for F77 plus Sun MPI 10
  for Sun HPF 10
  standard header file path 10
S3L, contents 1-3
S3L, core library 1-3
S3L, features 1-1
S3L, overview 1-1
safety mechanism
  error checking 12
  setting environment variable 14
  setting via procedure call 14
  synchronization feature 12
safety mechanism, thumbnail 1-5
sample code
  running 15
sample code directory 14
single vs multiple instance
  calling sequence examples 4-4
single vs multiple instance operations
  calling sequence summary 4-4
  Fast Fourier Transforms example 4-8
  matrix vector multiplication example 4-4
sort routine, thumbnail 1-4
Sun Performance Library 1-2

T

thread safety, thumbnail 1-5
transpose routine, thumbnail 1-4