Star-P® Software Development Kit (SDK) Tutorial and Reference Guide

Release 2.5.1

8/22/07

COPYRIGHT

Copyright© 2004-2007, Interactive Supercomputing, Inc. All rights reserved. Portions Copyright© 2003-2004 Massachusetts Institute of Technology. All rights reserved.

Trademark Usage Notice

STAR-P® and the "star p" logo are registered trademarks of Interactive Supercomputing, Inc. MATLAB® is a registered trademark of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders. ISC's products are not sponsored or endorsed by The Mathworks, Inc. or by any other trademark owner referred to in this document.

Contents

About the Star-P SDK 1
Introduction 1
Prerequisites 2
Key server concepts 2
The SDK Package 2
The gateway function 2
User-defined functions 2
Key datatypes 3
Key concepts on the client 4
Berkeley Unified Parallel C (UPC) Support 5

Data Parallelism using the Star-P SDK 7
Amesos *p: Using the Star-P SDK to Interface with High-performance Numerical Libraries. 7
Introduction 7
Implementation overview 8
Declaring the SDK solver function 11
Validating the inputs 11
Defining the linear system to be solved 13
Solving and checking for errors 15
Returning the solution to the client. 16
Registering the solver function with the Star-P runtime 17
Compiling and linking the SDK package 17
Client-side implementation. 18
Loading the solver into the Star-P client. 18
Implementing the dispatcher function. 18
Overloading the "\" operator for sparse matrices 19
Testing the sparse solver 20
Design, implementation and troubleshooting hints 20
Design and implementation guidelines. 20
When things go terribly wrong 21

Task Parallelism using the Star-P SDK 23
Introduction 23
Server-side Task Parallel Functions and Wrapper Classes 24
Wrapper class descriptions 24
pearg_t Class Functions 25
Public Functions. 25
Returns. 27
ppevalc_module_t Class Functions 29
Member Functions 30
Example using Star-P SDK for Task Parallel Functions. 30

Client-side Task Parallel Functions for the Star-P SDK 32
Loading a compiled C++ module on the Star-P server 32
Calling a compiled C++ function using ppeval 33
Unloading a compiled C++ module from the Star-P server 33
Application Example 33
Implementation Overview. 34
Code Structure 35
Code Changes 36
Modifying the Existing Routines 36
Including Star-P Header Files 38
Creating New Routines to Use the Star-P SDK Linkages 38
Preserving C++ Data between ppeval Calls. 40
Compiling Your Routines into an SDK Package. 40
Loading Your SDK Package into Star-P 41
Summary 41
Other Considerations 42
Changing a Reduction to Fit this Approach 42
Complete Original Code. 42
Complete Final Code 46

Star-P SDK Class Lists 53
Star-P SDK for Data Parallel Operations 53
starp_sdk_t Class Reference 53
Detailed Description 54
Constructor & Destructor Documentation. 57
Member Function Documentation 57
starp_value_t Class Reference 59
Scalar Constructors 61
Array Constructors 61
Conversion Operators 61
Friends 62
Classes 62
Detailed Description 62
Constructor & Destructor Documentation. 62
Member Function Documentation 63
Friends And Related Function Documentation. 66
matrix_t Class Reference. 66
Detailed Description 67
Constructor and Destructor Documentation 67
Member Function Documentation 67
Friends and Related Function Documentation 70
matrix_t Member List 71
layout_desc_t Class Reference 71
Detailed Description 73
Constructor & Destructor Documentation. 74
Member Function Documentation 75
Friends And Related Function Documentation. 81
Member Data Documentation 81
type_desc_t Class Reference 82

Detailed Description 83
Constructor & Destructor Documentation. 83
Member Function Documentation 83
Friends And Related Function Documentation. 84
Member Data Documentation 85
type_desc_t Member List 85
matrix_factory_t Class Reference 86
Detailed Description 87
Member Function Documentation 87
Friends and Related Function Documentation 90
matrix_factory_t Member List. 91
initializer_t Struct Reference 91
Detailed Description 91
Member Function Documentation 92
PPcsr Struct Reference 93
Detailed Description 93
Member Data Documentation 93
Star-P SDK for Task Parallel Operations 94
ppeval C Engine Class List 94
ppeval C Engine File List 95
ppeval C Engine Class Documentation 95
pearg_t Class Reference 95
Classes 98
ppevalc_module_t Class Reference 101
ppevalc_module_t::private_state Struct Reference 104

List of All Class Members 105
- a - 105
- c - 105
- d - 105
- e - 106
- f - 106
- g - 106
- h - 106
- i - 106
- l - 107
- m - 107
- n - 108
- o - 108
- p - 109
- r - 109
- s - 109
- t - 110
- u - 110
- z - 110
- ~ - 110

INTERACTIVE supercomputing

Chapter 1

About the Star-P SDK

Introduction

The Star-P[1] Software Development Kit (SDK) enables users to extend the Star-P server by interfacing it with custom serial and parallel libraries written in C, C++ and Fortran.

- On the **server side**, the SDK provides a high-level C++ API for creating and operating on distributed dense and sparse matrices, as well as invoking user defined C++ functions. It also features a user-friendly distributed error handling mechanism. The API therefore serves as a glue for conveniently passing data back and forth from the Star-P server runtime and a user's custom serial or parallel code.
- On the **client side**, the SDK provides a number of utility commands for loading and unloading user packages (for data parallel operations) and modules (for task parallel operations) on the server and invoking user-defined functions.

This document provides an overview of implementing user-defined functions in Star-P. This chapter contains a brief glossary of SDK-related concepts. The next chapter contains a tutorial on data parallel operations with the Star-P SDK that discusses the creation of an interface of Star-P to Trilinos, a third-party suite of distributed-memory sparse solvers. The following chapter provides instruction on interfacing Star-P to a serial C++ function and invoking it in a task-parallel manner using `ppeval`. The final chapter contains a reference of all the classes in the C++ API.

1. Trademark Usage Notice: STAR-P® and the "star" logo are registered trademarks of Interactive Supercomputing, Inc. MATLAB® is a registered trademark of The MathWorks, Inc. Other product or brand names are trade-marks or registered trademarks of their respective holders. ISC's products are not sponsored or endorsed by The MathWorks, Inc. or by any other trademark owner referred to in this document.

Prerequisites

In order to create functions using the SDK a user must possess:

- access to the server machine where the Star-P server is installed
- ability to compile Fortran, C and C++ programs on the server
    - The Star-P server is compiled using version 9.1.045 of the Intel(R) C++ compiler using the option to generate code compatible with GCC 3.3. In general, it is recommended that you also compile the SDK functions using a similar setup as it ensures that the generated libraried will be binary compatible with the Star-P server.
- access to supporting header files and libraries such as MPI
    - On the SGI Altix platform, it is recommended that you compile custom MPI code against the SGI MPT (Message Passing Toolkit) library. On the EM64-T platforms, it is recommended that you compile against the Intel MPI library
- ability to make the resulting shared objects accessible to the Star-P server

In order to use the functions created through the SDK on the user needs:

- a MATLAB® client
- the absolute path to the compiled shared library on the server containing the user-defined functions, and
- the names of the functions contained in the user's custom shared library

Key Concepts

This section briefly describes the main SDK programming concepts on the server side.

There are primarily two types of extensions that can be written using the Star-P SDK: a data parallel *package* and a task parallel *module*. A package extension is useful when writing code that requires access to global matrix data structures and is typically used when interfacing with parallel libraries. On the other hand, a module extension is used when the function needs access to only a single slice of data at a time and is typically used when interfacing with a serial library in an embarrasingly parallel fashion.

The following sub-sections provide an overview of the important concepts in writing the two forms of the SDK extensions.

The Task-Parallel Module

Each task-parallel module consists of:

1. A number of user-defined functions written according to a specified signature. Each evaluation of these functions only passes in the local data corresponding to a single slice of each input (if it is split), or the entire input (if it is broadcast)

2. A single gateway function that simply maps each of the user-defined functions to a unique name. The user-written function can then be invoked on the client-side using ppeval

User-defined functions

Each user-defined function must be written according to the following signature:

```
static void function_name (ppevalc_module_t& module,
                           const pearg_vector_t& inputs,
                           pearg_vector_t& outputs)
```

The first argument is an object containing module-level information, the second argument is a vector of inputs to the user-defined function and the last argument is a vector of outputs from each function evaluation.

The gateway function

The single gateway function in the module is named `ppevalc_module_init` and is called only once when the module is loaded. The primary purpose of the gateway is to map each of the user-defined functions to a unique name through which they can be invoked on the client.

Key datatypes

The following are the key datatypes when writing task parallel modules. For more details on using these, please refer to the class documentation in Chapter 4.

Datatype	**Brief description**	**Key functionality**
`ppevalc_module_t`	Base module class	Contains global state information about the module instance; registers user-defined functions in ppevalc_module_init; provides error handling functionality
`pearg_t`	Wrapper object for inputs and outputs	Wraps slice vectors, broadcast inputs and scalars.

The Data Parallel Package

A Star-P SDK package is a user-provided shared library containing:

1. A number of user-defined functions written according to a specified signature, and
2. A single gateway function, again written according to a specified signature

User-defined functions

There can be any number of user-defined functions within a single SDK package, but they must be written according to the following signature:

```
static void function_name (starp_sdk_t& sdk,
                           const starp_value_vector_t& inputs,
                           starp_value_vector_t& outputs)
```

The first argument is an opaque variable containing state information about the Star-P server; the second argument is a list of inputs to the user-defined function and the third argument is a list of outputs (which is initially empty).

The gateway function

Each SDK package contains a single function named `starpInit` that gets invoked when a user-defined package is loaded from the client. The primary purpose of this function is to map each user-defined SDK function to a unique function name. It is this unique name that is used by the client to invoke the corresponding function on the server.

Key datatypes

The following are the key datatypes on the server. For more details on using these, please refer to the class documentation in Chapter 4.

Datatype	Brief description	Key functionality
`starp_sdk_t`	Base SDK class	Contains global state information about the server instance; registers user-defined functions in starpInit; provides error handling functionality
`starp_value_t`	Wrapper object for inputs and outputs	Wraps distributed matrices, local matrices and scalars.

`matrix_t`	Distributed matrix class	Describes a distributed dense or sparse Star-P matrix. Provides methods for getting matrix data (either from the local process or all the processes), data redistri-bution, etc.
`layout_desc_t`	Size/distribution of a matrix	Contains information about the global dimensions of the matrix as well as local dimensions on each process
`type_desc_t`	Datatype of matrix elements	Contains information about the underlying datatype as well as its counterpart in MPI
`storage_desc_t`	Local storage of a matrix	Describes how a matrix is stored locally on each server process: either in dense, column major order, or sparse, compressed row representation, etc.
`matrix_factory_t`	Factory class to create new matrices	Enables the convenient construc-tion of distributed dense and sparse matrices, either using pre-defined constructors or a user-supplied initilization routine; also enables cloning the data in existing distributed matrices.

Key concepts on the client

This section briefly describes the commands on the client for operating on user-defined packages and invoking custom SDK functions.

Command	Key functionality	Example
`pploadpackage`	Causes the server to load an SDK package, invoke **`starpInit`** and register the user-defined functions and makes them available for use on the client.	`pploadpackage('/tmp/mypack-age.so')`
`ppunloadpackage`	Causes the server to un-load a previously loaded SDK package.	`ppunloadpackage('/tmp/mypack-age.so')`
`ppinvoke`	Executes a user-defined function (with the name registered on the server) using the specified distributed and front-end inputs	`x = ppinvoke('my_func',a,b,c)`
`ppevalcloadmodule`	Causes the server to load an SDK task parallel module, invoke **`ppevalcloadmodule`** register the user-defined functions, and makes them available for use on the client.	`sym_name = ppevalcloadmodule('module_name', 'sym_name');`

`ppevalcunloadmodule`	Causes the server to unload a previously loaded task-parallel module.	`ppevalcunloadmodule(sym_name);`
`ppeval` `ppevalsplit`	Executes a user-defined function, using the symbolic module name registered on the server and a passed member function, with the input arguments that are passed for the member function	`output_arg = ppeval('C://sym_name:foo', input_args);`

INTERACTIVE supercomputing

Chapter 2

Data Parallelism using the Star-P SDK

Amesos *p: Using the Star-P SDK to Interface with High-performance Numerical Libraries

Introduction

One of the primary aims of the Star-P SDK is to facilitate the easy integration of existing third-party parallel numerical libraries into the Star-P framework, such as

- ScaLAPACK and
- SuperLU.

This allows users to interactively develop applications on the client, yet leverage existing domain specific libraries with minimal implementation effort.

In this section, the following is demonstrated:
1. The use of the Star-P SDK to interface with an external suite of direct solvers
2. Techniques for safe interoperability of data types between Star-P and external libraries
3. Overloading operators on the client side for `ddense`, `dsparse` and `ddensend` objects
4. Tips for easily performing inconvenient data transformations
5. Hints for implementing, building and debugging your interface to other libraries

The discussion focus is providing an interactive interface to Trilinos, a suite of parallel linear and non-linear direct and iterative sparse solvers from Sandia National Laboratories. In particular, a basic interface to Amesos is illustrated. Amesos is a Trilinos package that provides a common interface to a large number of parallel sparse direct solvers such as

- SuperLU,
- MUMPS and
- PARDISO.

More information on the Trilinos project can be found at http://software.sandia.gov/trilinos and the solvers available through Amesos are listed at http://software.sandia.gov/trilinos/packages/amesos/index.html.

To simplify the example further, the interface from Amesos to KLU is presented. KLU is a sparse direct solver by Timothy Davis and Ekanathan Palamadai of the University of Florida. This solver is particularly suited for very sparse matrices arising in applications such as circuit simulation. More information on KLU can be found at
http://www.cise.ufl.edu/research/sparse/klu

Implementation overview

The implementation essentially consists of the two major steps:

1. Building the interface from Star-P to the library on the server side and registering the function with the Star-P server runtime system, and
2. Loading the server-side interface on the client and optionally, mapping an appropriate `ddense`, `ddsparse` or `ddensend` operator to the newly loaded function.

Each of these main steps is discussed in the following subsections.

Server-side implementation

The implementation on the server side consists of writing a function with a specified signature that:

- validates the input arguments,
- copies the data from a `dsparse` left-hand side and a `ddense` right-hand side passing it into a Trilinos data structure that defines a sparse linear system,
- invokes the solver and returns the solution back to the client. The SDK function also traps errors from the solver (such as a singular system) and returns these to the client.

The subsequent sections discuss these steps in more detail. The complete source code for the example can be found in the file `SDK_Amesos.cc` in your Star-P installation on the server under the `sdk/amesos_star-p/src` sub-directory.

Initial setup

The Star-P SDK currently provides a C++ API. A typical Star-P extension (also denoted as a *package*) consists of

- a number of SDK functions that must conform to a particular signature and
- a single gateway function that servers use to make these SDK functions available to the Star-P runtime.

The initial setup for writing our SDK interface to Amesos consists of creating a C++ source file (`SDK_Amesos.cc` in this case) using a text editor such as EMACS or VI or an IDE such as KDevelop or Eclipse.

The first step is to include a number of header files:

- `Starp.h` containing declarations for the various Star-P SDK classes,
- `mpi.h` containing function signatures and constants specific to MPI and
- a few header files containing class and function declarations specific to Trilinos and Amesos.

```
#define NO_SDK_DEPRECATED_METHODS TRUE

#include <Starp.h>
#include <mpi.h>
#include <Epetra_MpiComm.h>
#include <Epetra_MultiVector.h>
#include <Amesos.h>
```

Notice the use of the `NO_SDK_DEPRECATED_METHODS` macro. Defining this macro *before* including `Starp.h` disables backward compatibility with the previous release of the SDK. If you wish to use the previous release of the SDK, you must not define this macro in your implementation.

Note: ***Note to C and Fortran users:*** While only the C++ language is supported by the SDK interface at this point, the functions that you implement can call modules written in other languages. Therefore, it is possible to implement a simple C++ wrapper function around your custom module that takes in the input arguments from (and passes back outputs to) Star-P and simply calls your custom C function or Fortran subroutine for performing the underlying computation. Note however that when a Fortran subroutine is invoked from a C or C++ function, it is necessary to rename it in a platform and compiler-specific manner. Please consult the documentation from your compiler vendor on the name mangling scheme used by your compiler.

Figure 2-1 Select operations involved in using a library function to overload "\" for distributed sparse matrices.

Client side	Server side (SDK_Amesos.cc)	Comments
LoadAmesos.m:		Load package on the server side
`pploadpackage(SDK_Amesos.so)`		
	`starpInit ( … ) register_func(sdk,"SDK_Amesos", SDK_Amesos);`	Register “SDK_Amesos’

LoadAmesos.m: `path( … )`		Allow Star-P to find **@dsparse/mldivide.m** to overload "\" for **dsparse**objects
RunAmesos.m:		Calls
`x = A \ b ;`		@dsparse/mldivide.m
@dsparse/mldivide.m: `x =SDK_Amesos(A, b) ;`		Calls **@dsparse/SDK_Amesos.m**
@dsparse/SDK_Amesos.m: `x = ppin-voke('SDK_Amesos',A, b);`		Call SDK_Amesos on server
	`SDK_Amesos( …) {`	Solver SDK Function
	`A = inArgs.at(0).distributed_value();`	Get input matrix **A** from first parameter passed to **ppinvoke.**
	`data_a = A -> local_sparse_data() ;`	Get pointer to the sparse data structure for **A.**
	`b = inArgs.at(1).distributed_value()`	Get right-hand side ***b*** from second parameter passed to **ppinvoke.**
	`data_b = b -> local_dense_data <starp_double_t> ()`	Get pointer to data in ***b.***
	`x = matrix_factory.clone (b)`	Create ***x*** by copying ***b.***
	`data_x = x -> local_dense_data<starp_double_t> ();`	Get pointer to data in ***x***
	`:`	
	`:`	Compute ***x*** based on **A**
	`:`	and ***b*** using Amesos
	`outArgs.push_back(x);`	Return ***x*** back to client
	`}`	

Declaring the SDK solver function

The solver function for this example is SDK_Amesos and is declared with the following signature:

```
static void
SDK_Amesos (starp_sdk_t& sdk, const starp_value_vector_t& inArgs,
starp_value_vector_t& outArgs)
```

Note the use of the `static` keyword in the function declaration. This makes the solver function visible only to other functions in the same source file, in particular, the gateway function discussed later.

The first argument to the function is a `starp_sdk_t` reference containing global state information about the Star-P server, the second argument is a constant reference to the list of input arguments and the third argument is a reference to the list of output arguments. The output argument list is initially empty.

> **Note:** In the server SDK, all datatypes (be they locally-replicated matrices, distributed matrices, scalars or strings) are wrapped in objects of type `starp_value_t`. The type `starp_value_vector_t` is simply a C++ standard library `vector` of `starp_value_t` objects. Operating with `starp_value_list_t` objects therefore requires a modest understanding of the standard library `vector` class which may be found for instance, in the book “C++ Standard Library: A Tutorial and Reference” by N. Josuttis or online at http://www.sgi.com/tech/stl/Vector.html.

Validating the inputs

It is first necessary to check the input arguments for possible errors. For instance, the `SDK_Amesos` function accepts only two inputs: a sparse coefficient matrix which must be real and square and a right-hand side which must be real, dense and row-distributed.

Checking for the number of input arguments (two in this case) is handled as:

```
ThrowIfFalse (inArgs.size() == 2, "SDK_Amesos requires exactly two
arguments.\n);
```

Notice that since inArgs is simply an STL `vector`, the number of input element can be obtained using its size method; the `ThrowIfFalse` function simply throws a `runtime_error` if its first argument is false.

Similarly, the check for the first argument being sparse and real is performed using the following lines of code:

```
matrix_ptr_t matrix_A = inArgs.at(0).distributed_value();

ThrowIfFalse (matrix_A->storage_desc() == PP_STORAGE_CSR,
              "First argument to SDK_Amesos must be a sparse matrix.

ThrowIfFalse (matrix_A->data_type_is<starp_double_t>(),
              "SDK_Amesos does not support COMPLEX matrices at this
              time.\n");
```

`inArgs.at(0)` returns the first input argument as a `starp_value_t`. If a `starp_value_t` object wraps a distributed matrix, the `distributed_value` method returns a smart pointer to it; if the underlying object is not a distributed dense or sparse matrix, then the distributed_value method throws an invalid_argument exception.

Note: To find if a `starp_value_t` object wraps a distributed matrix, use the `is_distributed` method of the `starp_value_t` class.

For a distributed matrix, the storage descriptor, `storage_desc`, is a constant representing the internal storage of the matrix on each Star-P process. The constant `PP_STORAGE_CSR` corresponds to a sparse matrix stored in the compressed sparse row format (discussed in more detail in the next section).

The check for a square matrix is again similarly performed as follows:

```
starp_size_vector_t global_sizes_A =
matrix_A->layout_desc().global_size_array();

int global_rows_in_A = static_cast<int> (global_sizes_A[0]);
int columns_in_A     = static_cast<int> (global_sizes_A[1]);
ThrowIfFalse (global_rows_in_A == columns_in_A,
        "The first argument to SDK_Amesos must be a square matrix.\n")
```

In the above code segment, the layout descriptor, `layout_desc` describes how the matrix is laid out among the various Star-P processes; the method `global_size_array` in this class returns a list containing the dimensions of the matrix (the first element representing the number of rows, the second representing the number of columns, etc.). The dimensions of the matrix are represented using values of type `starp_size_t` and a `starp_size_vector_t` is simply a standard C++ vector or `starp_size_ts`.

Finally, we illustrate an important check to ensure that the distributed data passed to the SDK function can be safely passed to Amesos. The reason for performing this check is that on machines with a 64-bit architecture, `starp_size_t` is an unsigned 64-bit integer; this allows for the representation of large arrays with more than 232 non-zero elements. However, many parallel libraries including Amesos represent matrix dimensions using int, which on many architectures is a signed 32-bit integer. Therefore, passing in 64-bit numbers to such libraries can result in silent overflows that can in turn produce incorrect answers. Therefore, the

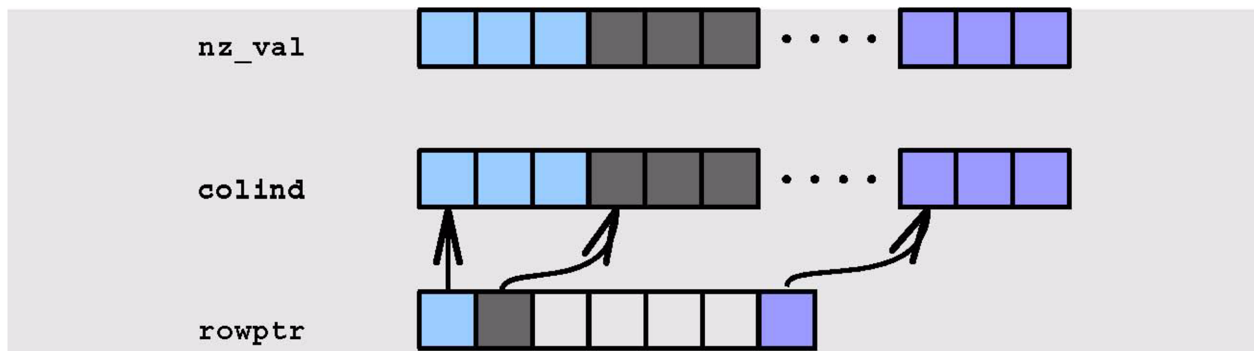
following code segment checks to make sure that the rows and columns of the sparse matrix as well as the number of non-zeros can be represented using 32-bit integers:

```
const PPcsr *sparse_data = (PPcsr *) matrix_A->local_sparse_data() ;

starp_size_t local_idx_nnz = sparse_data->nnz_loc ;
int          local_int_nnz = static_cast<int> (local_idx_nnz) ;
starp_size_t global_idx_nnz;
int          global_int_nnz;

MPI_Allreduce(&local_idx_nnz, &global_idx_nnz, 1, MPI_STARP_SIZE_T,
MPI_SUM, mpi_comm);
MPI_Allreduce(&local_int_nnz, &global_int_nnz, 1, MPI_INT         ,
MPI_SUM, mpi_comm);

ThrowIfFalse (static_cast<starp_size_t> (global_rows_in_A) ==
global_sizes_A[0],            "Too many rows for this integer type


ThrowIfFalse (static_cast<starp_size_t> (global_int_nnz) ==
global_idx_nnz,

      "Matrix is too large for this integer type./n);
```

The method local_sparse_data returns a pointer to the underlying compressed sparse row data structure. The field, `nnz_loc` returns the number of local non-zero entries. Observe that it is of type `starp_size_t` which represents a 64-bit integer on 64-bit architectures. We then cast this number down to a 32-bit integer (which might overflow locally), and perform a global summation of both the 32-bit and 64-bit values using the `MPI_Allreduce` function; on completion of this call, the results are stored in the variables `global_int_nnz` and `global_idx_nnz` respectively. When the global number of non-zero entries computing using the two techniques is different, then clearly the matrix cannot be represented using 32-bit indices and an error is returned back to the client.

Defining the linear system to be solved

Before describing the creation of the linear system, we briefly describe the storage format for distributed sparse matrices in Star-P. Globally, the sparse matrix is distributed such that each process gets a certain number of contiguous rows. Locally on each Star-P process, the sparse matrix is stored in the *compressed sparse row* format illustrated in the following figure:



For a matrix with `m_loc` rows and `nnz_loc` non-zero values, the array `nz_val` of length `nnz_loc` represents the non-zero entries in the matrix such that the entries from a higher-numbered row always occur after the entries in a low-numbered one. Each element of the array colind, also of length `nnz_loc` contains the index of each corresponding non-zero value. Finally, each element of the array rowptr of length m_loc contains offsets into the first non-zero entry on each row (which the cumulative sum of non-zero entries in that row. The structure `PPcsr` discussed previously simply encapsulates the various fields describing the layout of the sparse matrix.

The first step to creating the linear problem in Amesos is to create an `Epetra_Map` object (which is similar to the layout descriptor in Star-P):

```
int             num_globals       = -1;
const int       starting_index    = 0 ;
MPI_Comm mpi_  comm  = starpSDK.communicator() ;
Epetra_MpiComm epetra_comm (mpi_comm);

Epetra_Map epetra_map (num_globals, sparse_data->m_loc,starting_index,
epetra_comm);
```

Next, a compressed sparse row matrix of type `Epetra_CrsMatrix` is created and the position and values of the non-zero entries using the `InsertGlobalValues` method is indicated. A call to the `FillComplete` method actually performs the creation of the sparse matrix.

```
bool use_statically_allocated_storage = true;
Epetra_CrsMatrix epetra_matrix(Copy, epetra_map, &entries_per_row[0],
                                use_statically_allocated_storage);

vector<int> int_column_index (max_num_entries);

for (int i = 0; i < local_rows ; i++) {
    for (int j = 0; j < entries_per_row[i]; j++) {
        int_column_index[j] = static_cast<int> (
idx_column_index[row_pointers[i] + j] );     }

    epetra_matrix.InsertGlobalValues (sparse_data->fst_row+i,
entries_per_row[i],
                              &values[row_pointers[i]],
&int_column_index[ 0 ]);
}

epetra_matrix.FillComplete() ;
```

Then, the right-hand side is created and the space is allocated for the solution using by creating `Epetra_MultiVector` objects that directly map on to `ddense` objects in Star-P.

Note: The solution is initially set to the right-hand side.

```
double *data_b = (double *)
matrix_B->local_dense_data<starp_double_t>() ;
Epetra_MultiVector epetra_vector_B (Copy, epetra_map, data_b,
local_rows_in_B, cols_in_B);
Epetra_MultiVector epetra_vector_X (epetra_vector_B);
```

Finally, an Amesos data structure is created, an `Epetra_LinearProblem` representing the sparse system of equations to solve and choosing the solver (KLU in this example):

```
Epetra_LinearProblem epetra_linearproblem(&epetra_matrix,
&epetra_vector_X, &epetra_vector_B);
Amesos Factory;
Amesos_BaseSolver* amesos_solver = Factory.Create("Amesos_Klu",
epetra_linearproblem);
```

Solving and checking for errors

The next step in our example is the actual solution of the system of equations. This proceeds in three steps: The first step (symbolic factorization) analyzes the sparsity structure of the matrix, reorders the elements to reduce the number of non-zero elements during the factorization step and allocates internal storage for the factors themselves. This step is performed using the `SymbolicFactorization` method in Amesos.

```
if ((ret = amesos_solver->SymbolicFactorization()) != 0) {
  os << "Failure during Amesos::SymbolicFactorization: error code "
     << ret << "

  throw os.str() ;
}
```

Symbolic factorization can fail if the matrix is structurally singular and this error must be propagated back to the client.

After the symbolic factorization step, we perform forward Gaussian elimination on the reordered sparse matrix and compute the entries in the L and U factors. This step, denoted as numerical factorization is performed using the `NumericalFactorization` method in Amesos.

```
if ((ret = amesos_solver->NumericFactorization()) != 0) {
  os << "Failure during Amesos::NumericFactorization: error code "
     << ret << "

  throw os.str() ;
}
```

Numerical factorization can fail if the matrix is badly scaled or is close to singular (i.e., has a condition number >> 1). This error must again be returned to the client.

Finally, to compute the final solution, you must back solve the a triangular system of equations (called back substitution). This is done using the `Solve` method in Amesos:

```
if ((ret = amesos_solver->Solve()) != 0) {
  os << "Failure during Amesos::Solve: error code " << ret << "

  throw os.str() ;
}
```

While it is very rare for the back substitution step to fail, it is still prudent to check for errors and propagate them back to the client.

Returning the solution to the client

If the solution step is successful, the result can be safely returned to the client. This is a relatively straightforward process consisting of obtaining a Star-P `matrix_factory_t` instance, creating a `ddense` object, `matrix_X` and initializing it using the `ExtractView` method of `Epetra_MultiVector` objects.

```
matrix_factory_t MatFactory = sdk.matrix_factory ();
matrix_ptr_t matrix_X = MatFactory.clone (matrix_B);

double *sdk_data_x = (double*)
matrix_X->local_dense_data<starp_double_t> ();
double *epetra_data_x ;
int lda_x ;
epetra_vector_X.ExtractView(&epetra_data_x, &lda_x);

int local_size_B = lda_x * cols_in_B ;

for (idx_t i = 0; i < local_size_B ; ++i) {
  sdk_data_x[i] = epetra_data_x[i] ;
}
```

Once the `ddense` object has been initialized, it must be sent back to the client using the add_output_matrix method in the `StarpSDK` object.

```
starpSDK.add_output_matrix (matrix_X);
```

Note that since the argument list is initially empty, we use the `push_back` method of the standard vector class to append the matrix to the end of the list.

Registering the solver function with the Star-P runtime

Each user-defined package in Star-P must contain a gateway function, `starpInit` that gets invoked each time the package is loaded. This function can be interpreted as a “constructor” for the package: it can

- initialize user-specified persistent state objects,
- create MPI resources such as custom data types,
- load external libraries, etc.

However, the most common use for the gateway is to make other user-defined functions such as `SDK_Amesos` available to the Star-P runtime. This is done using `register_func` method of the `starp_sdk_t` class:

```
extern "C" bool starpInit (starp_sdk_t &sdk)
{
  sdk.register_func("SDK_Amesos", &SDK_Amesos);

  return true;
}
```

The `register_func` method binds a user-defined function to a name (“`SDK_Amesos`”) that can be used to invoke it from the client side using `ppinvoke` (discussed later).

Note that the gateway function must have the same name and signature specified in this example and must use the `extern “C”` linkage directives to disable name mangling by the C++ compiler. Failure to follow this convention will not cause a compile error, but will trigger a runtime error during loading of the package.

Compiling and linking the SDK package

Compiling the SDK example requires modification of the supplied `Makefile`. For this particular case, you will only need to modify the `TRILINOS_PREFIX` and `SDK_PREFIX` variables to point to the installation directory of Trilinos and the Star-P SDK. The Star-P server installation comes with a pre-compiled version of Trilinos library available under the `sdk/examples/amesos/server/TrillinosPrefix` subdirectory under the installation root directory.

Simply running `make` on the same directory as the source file will create a dynamic shared object file, `SDK_Solver.so`.

If you are unable to compile the provided example after making the necessary changes, please contact your Interactive Supercomputing customer service representative.

Client-side implementation

Loading the solver into the Star-P client

Once the shared object has be successfully compiled on the server, it is necessary to load it using the `ppioadpackage` command on the client to make the SDK function available to Star-P. The only option to ppioadpackage is the absolute path on the server of the dynamic shared object file (in our case, `SDK_Amesos.so`).

```
%  SDK_Amesos_path should point to the shared library file built on the
server
%

SDK_Amesos_path = ;
pploadpackage (SDK_Amesos_path);

%
%  MyPath should point to the directory that this file resides in on
client
%
global MyPath;
MyPath = '/usr/local/starp/sdk/examples/amesos/client' ;
```

In addition, it might also be necessary to setup the MATLAB® path to point to any new wrapper functions that in turn invoke SDK functions on the server. A script for performing both these operations is `LoadAmesos.m` and can be found under the `matlab/sdk/amesos` subdirectory of your Star-P client installation.

Implementing the dispatcher function

Once the SDK function has been loaded, we write a simple wrapper that invokes our SDK function, `SDK_Amesos`. The invocation is done using the `ppinvoke` command that must be passed the name of the SDK function (the second argument to the `register_func` method in the `starpInit` function on the server) and all the expected arguments. The wrapper function, `SDK_Amesos` below calls the solver function and passes in the matrix and the right-hand side and stores the resulting `ddense` answer in the output variable.

```
function x = SDK_Amesos( A, b )

    x = ppinvoke ('SDK_Amesos', A, b );
```

Note that since the server function incorporates extensive error-handling of the inputs, the wrapper function itself is quite basic.

Overloading the “\” operator for sparse matrices

The next step in our client-side implementation is to overload the backslash(\) operator for sparse matrices so that it calls our SDK solver instead of the default solver installed with Star-P. This is done by first creating a sub-directory named `@dsparse` under the directory specified in the `MyPath` variable and adding this to the *top* of your MATLAB® path. Then, we create a function named `mldivide.m` (for matrix left divide) under this subdirectory that gets invoked each time the expression “A\b” is called with a `dsparse` left-hand side matrix.

Our custom mldivide function simply checks for a global variable `DsparseSolver`; if this variable is set to `Amesos_KLU`, we call our custom solver function (`SDK_Amesos`, implemented in Step 2), else, we simply revert back to the default sparse solver.

```
function x = mldivide(A,b)

    global DsparseSolver
    global MyDsparsePath

    if ~isempty( DsparseSolver ) &&
~(strcmp(DsparseSolver,'Amesos_KLU') || ...
         (strcmp(DsparseSolver,'superlu') )
         fprintf('Sparse Solver %s not supported, reverting to the
default method

                  DsparseSolver)
    end

    if ( strcmp(DsparseSolver,'Amesos_KLU') && isreal(A) )
         x = SDK_Amesos( A, b ) ;
    else
         rmpath( MyDsparsePath )
         x = mldivide(A,b) ;
         path( MyDsparsePath, path )
    end
```

Testing the sparse solver

The final step in our example is to test the entire implementation. This can be done interactively from the MATLAB® prompt after running `LoadAmesos` as follows:

```
>> N = 1000;
>> density = 0.01;
>> A = sprandn(N*p, N, density);
>> b = randn(N*p, 1);
>> % First call the default sparse solver
>> x_default = A \ b;
>> % Now call KLU
>> global DsparseSolver; DsparseSolver = 'Amesos_KLU' ;
>> x_klu = A \ b;
>> % Compute the relative error
>> err = norm(x_default ñ x_klu)/norm(x_default) * 100
```

or by running the `LoadAndRunAmesos` script in the `matlab/sdk/amesos_star-p` subdirectory.

Design, implementation and troubleshooting hints

In this section, we briefly review a few guidelines for designing and debugging your own package. Parallel programs (yes, even those written with the Star-P SDK!) are notorious hard to debug and following some of the principles outlined here will make the task of implementing a Star-P extension as painless as possible.

Design and implementation guidelines

1. Place a lot of emphasis on *modular* design, where each component can be developed and tested as independently as possible. In particular, try and keep the underlying functionality of your function as independent of the SDK as possible; as far as possible the SDK function should only server to wrap the actual computational subroutine. Debug your computational routines using a simple, standalone test container and ensure that it functions as desired. Then, when you implement the SDK wrapper, you will have to debug on the interfaces from Star-P to your function.
2. Follow the principle of *test-based development*. Write unit tests before, not after, completing the implementation.
3. Pointers are the main cause of memory errors. Instead, use STL containers such as list and vector. If you must allocate memory dynamically, wrap it in a smart pointer; the SDK ships with a simple reference-counted smart pointer class (see `smart_ptr.h`)
4. For error handling, use exceptions instead of error codes. Simply throwing an exception inside your SDK function causes it to be properly propagated back to the client with no additional work. The exceptions thrown should be derived from the **std::exception** class.
5. When implementing communication using MPI, avoid the use of blocking communication. For point-to-point message exchanges, use non-blocking sends and receives or the `MPI_Sendrecv` primitives; these automatically handle the scheduling of messages and will not result in deadlocks inherent in naïvely implemented MPI codes.

6. When transferring array sections between processes, use the vector data type facility in MPI instead of implementing your own packing and unpacking routines.
7. `pploadpackage` has no effect if a package with the same name is already loaded. Use `ppunloadpackage` to unload the previously loaded package before re-loading it with `pploadpackage`.
8. When rebuilding an SDK package while the server is running, the existing **.so** file should be deleted before creating the new **.so** file. If you directly overwrite the existing **.so** file while a starpserver instance is running, this could cause the server to crash the next time the SDK package is accessed in any way. An easy way to ensure that this rule is followed is to add a **`rm -f mypackage.so`** command to the Makefile rule which creates the **.so** file. For example:

```
mypackage.so: $(OBJECTS)
        rm -f mypackage.so
        $(LINK) -o mypackage.so
```

   The **-f** option makes it so the **`rm`** command will execute without an error even if **mypackage.so** doesn't already exist.
9. Keep your SDK function simple by transforming as much data as possible on the client *before* calling the function. For example, if your function requires the data to be distributed in a particular manner or requires all the data for the distributed matrix to reside only on one process, use the `reshape` command on the client as follows:

```
% Create a row distributed matrix A
App = randn(100*p, 100, 100);
% Distributed App along the columns and call your SDK function
Bpp = reshape(App, 100, 100*p, []);
Xpp = ppinvoke('my_func_1',Bpp);
%Now make all data reside only on the root process
Cpp = reshape(App, 1*p,[]);
%Call the second SDK function
Ypp = ppinvoke('my_func_2', double(size(App)), Cpp);
```

When things go terribly wrong

Finding the “black box”

The first step to tracing the problem is to locate the log file on the server that contains all output from the Star-P standard out and standard error streams. This log file is stored in the `.starpworkgroup` subdirectory in the home directory of the user who launched the Star-P process on the server (for example, if you launched Star-P on the server as `joeuser`, the log files can be found on the server under `/home/joeuser/.starpworkgroup`). The log files are of the form `starpsession.N.log`, where `N` is the process ID of the server process. The most recent log file can be obtained by typing “`ls -ltd * | head`”.

If the Star-P server process was terminated because of a *segmentation violation* error (SIG_SEGV), you must perform the following steps to ensure that the server log file contains useful debugging information: First, you must compile your source with debugging turned on (for most compilers, this corresponds to appending a "`-g`" flag to the compile line in the `Makefile`). Second, you must ensure that the `MPI_COREDUMB_DEBUGGER` environment variable is set in your `.bashrc` or `.cshrc` file to point to the debugger to be used to generate the core dump stack trace. By default, the Intel® debugger, `idb` is invoked provided it is in your path. If you do not have the Intel® debugger or wish to use another debugger such as `gdb` for generating the stack trace, please refer to the help page for the `mpirun` command (type `man mpirun` in your command prompt on the server and type `/MPI_COREDUMP_DEBUGGER` to go to the relevant section).

Debugging your code using a debugger

If you have tracked down a server segmentation violation to an SDK function written by you, the next step is to try and narrow down the error in the source by stepping through your code inside a debugger. This consists of the following steps:

1. First and foremost, reproduce the error using *as few* Star-P server processes as possible (ideally one or two). It can be highly unproductive to simultaneously step through 16 debugging sessions!
2. Ensure your SDK function is compiled with debugging turned on (see previous item)
3. On the client side, load your package using the `pploadpackage` command. Steps 4 through 7 must be performed on the server.
4. Obtain the process ID of the `starpserver` processes. This can be done by logging into the server using the user name used to start Star-P and typing "`ps ax`" on the command line. You should then see the Star-P server processes as follows:
5. The first column shows the process IDs that are of interest. Select the last NP set of IDs, were NP is the number of Star-P processes launched (In the above example, `NP = 2` and the IDs to note are 27191 and 27192). Notice that the PIDs occur consecutively: they map directly to the ranks of the MPI processes (in our case, rank 0 has a PID of 27191 and rank 1 has a PID of 27192).the processes you must type "`gdb --pid = NNN`" where NNN must be substituted for each process ID selected.Attach the debugger to the Star-P processes selected in step 4. For example to attach gdb to rank 0, one would type "`gdb --pid =27191`".
6. The debugger will now load up all the symbols in the Star-P server process, including ones from your packages (provided you remembered to use pploadpackage before calling gdb). You can now set breakpoints in your code as desired.
7. Once you have set all the breakpoints, type "`cont`" to continue execution
8. Switch back to the client and run your package function as usual using `ppinvoke`.
9. Switch back to the debugger processes on the server. Once the program flow reaches your code, the debugger processes will automatically stop at the breakpoint in your code. You can then examine variables, step into functions, as normal. Please refer to the gdb documentation for more details on using the debugger (type "`info gdb`" on the command prompt).

Chapter 3

Task Parallelism using the Star-P SDK

Introduction

Besides incorporating data parallel libraries via the Star-P SDK, the SDK can also be useful in parallelizing existing C++ programs that have certain types of available parallelism, without resorting to MPI. The parallelism must be of the type that is referred to as task parallelism or embarrassing parallelism. This means the parallel tasks are completely independent of each other, and have no communication or synchronization while they are active. The approach described below allows for reuse of the vast majority of the code of the existing program, which is often important when code is maintained as part of a major stand-alone application. This approach also provides the performance benefits of parallelism, typically through simple additions at the very high level language (VHLL, e.g., MATLAB or Python) level.

To use the Star-P SDK interface for task parallel functionality, you must be aware of both the server side C++ class definitions necessary to wrap exisiting C++ code or create new C++ code and the VHLL functions necessary to call your code from the client. On the server side, there are two wrapper classes, `pearg_t` and `ppevalc_module_t`, that are used to hold input and output arguments and provide an interface between your code and the starpserver respectively. On the client side, your newly defined C++ modules are loaded to the server using `ppevalcloadmodule`. By using either `ppeval` or `ppevalsplit`, with a special argument sequence that denotes the calling of a compiled C++ function, you can then use your C++ functions in task parallel from within a VHLL environment as you would any other MATLAB or Python function. When you are finished with a particular task parallel module, you can unload that module from the server using `ppevalcunloadmodule`.

This chapter begins by providing a description of the contents of the two server side wrapper classes for the Star-P task parallel SDK, showing some simple examples of their use. Next, the client side interfaces for loading, calling and unloading your compiled modules are described. Finally, an extended example is included showing the modifications necessary interfacing an original application code (for finite-element analysis) into the Star-P task parallel SDK.

Server-side Task Parallel Functions and Wrapper Classes

Wrapping existing codes using the Star-P SDK requires only two new classes:

- `pearg_t` - A class that holds input and output arguments to ppeval/ppevalsplit function calls. It supports three types of values or element types.
  - `char` - which represents a string
  - `double` - which means it can be an array of doubles
  - `double complex` - which means a complex double constructs complex data from real and imaginary data
- `ppevalc_module_t` - This class provides the interface for ppeval modules to interact with the starpserver runtime environment.

Basically, the use of these wrapper classes is just a matter of getting the input arguments to your compiled C++ functions and passing your arguments back to your VHLL client. When writing a C++ function for use by `ppeval`, everything that is placed between the input arguments and output arguments of a function is up to you. This means that the function you write must have the ability to be called in an embarrassingly parallel manner.

Wrapper class descriptions

On the server, `ppeval` uses the following classes:

- pearg_t - A class to hold input and output arguments to `ppeval` C++ functions. This class contains the following constructors:
  - scalar constructors
  - uninitialized and initialized array constructors

  This class also contains a data accessor which returns a pointer to the underlying data.
- `ppevalc_module_t` - A class that provides the interface for `ppeval` modules to interact with the starpserver runtime. This class contains constructors and deconstructors.

Each of these classes has an associated header file that describes the main body of code to other modules. These header files are

- `pearg_t.h`
- `ppevalc_module_t.h`

pearg_t Class Functions

Instances of this class have shallow copy, reference counted semantics, so it can be passed around by value. The data held is only freed when the last copy goes out of scope. To make a deep copy, use the `clone()` method.

Input Arguments

```
pearg_t ()
```

Creates a null `pearg_t`.

```
pearg_t clone () const
```

Creates a deep copy of this `pearg_t`.

```
void disown_data ()
```

Make this `pearg_t` disown data it holds.

Public Functions

Scalar Constructors

Creates an object initialized from the input argument.

```
pearg_t (starp_double_t v)
        (starp_dcomplex_t v)
        (std::string const &v)
```

The following table describes the parameters for the scalar constructors.

Table 1: Scalar Constructor Parameters

Type	Description
starp_double_t	An array of doubles
starp_complex_t	A complex double constructs complex data from real and imaginary data.
std:: string const &v	An alpha numeric character string.

Uninitialized Array Constructors

Create an array object of the given dimensions and type. Memory for the array is allocated and managed by this object. The memory is uninitialized. Use `data()` to get a pointer to the memory.

The following table describes the parameters for the uninitialized array constructors.

Table 2: Uninitialized Array Constructors

Type	Description
element_type	• CHAR • DOUBLE DOUBLE_COMPLEX
starp_size_t	• length • num_rows - number of rows • num_col - number of columns
starp_size_vector_t	Dimension size

Initialized Array Constructors

Create an array object backed by the array provided in argument v.

- If is_owner is true, then this object takes ownership of the memory pointed to by argument v and frees up the memory by using delete[ ].
- If is_owner is false, then it is the caller's responsibility to free the memory.

The following table describes the parameters for initialized array constructors.

Table 3: Initialized Array Constructor Parameters

Type	Description
element_type	• CHAR • DOUBLE DOUBLE_COMPLEX
starp_size_t	• length • num_rows - number of rows • num_col - number of columns
starp_size_vector_t	Dimension size
bool	TRUE if the pearg_t is null, meaning it was constructed with no argument constructor, therefore, it has no value.

Returns

```
bool is_null() const
```

Returns true if this pearg_t is null. It means that is was constructed with the null argument constructor and, therefore, has no value.

```
element_type_t element_type () const
```

Returns the object element type.

```
starp_size_t element_size () const
```

Return the size of one element in bytes of `element_type`.

```
starp_size_vector_t const & size_vector () const
```

Returns a vector containing the size of each dimension.

```
starp_size_t number_of_elements () const
```

Returns the total number of elements in the argument.

```
bool is_vector () const
```

Returns true if this argument is one-dimensional or two-dimensional with one of the dimensions of size equal to 1.

```
std::string const & string_data () const
```

Returns a string value if this is type STRING.

Data Accessor

Returns a pointer to the underlying data. The template type parameter must match the `element_type` for the object.

```
template<class ElementType> ElementType * data ()
template<class ElementType> ElementType const * data () const
```

Example:

```
starp_double_t *p = arg.data<starp_double_t();
```

Constructors and Deconstructors: Input

```
pearg_t::pearg_t () [inline]
```

Creats a null `pearg_t`. The `is_null` method will return true if this constructor is used.

Member Function: Input

```
pearg_t pearg_t::clone () const
```

Creates a deep copy of this `pearg_t`.

The copy constructor and assignment operator for this class make shallow copies, which internally refer to the same data. Use this method when you want to make a separate copy of a `pearg_t`.

```
template<class ElementType> ElementType const* pearg_t::data () const
template<class ElementType> ElementType* pearg_t::data ()
void pearg_t::disown_data () [inline]
```

Make this `pearg_t` disown the data that it holds. It will not free up memory for its argument data in it destructor. This is useful if you want to get a pointer to the data using `data ()`, and hold on to the pointer after the `pearg_t` object has gone out of scope.

Returns

```
starp_size_t pearg_t::element_size () const
```

Returns the size of one element in bytes of element_type.

```
element_type_t pearg_t::element_type () const [inline]
```

Returns the element type of this object.

```
bool pearg_t::is_null () const [inline]
```

Returns true if this `pearg_t` is null, meaning it was constructed with the no argument constructor and, therefore, has no value.

```
bool pearg_t::is_vector () const [inline]
```

Returns true if this argument is one dimensional, or two dimensional with one of the dimensions of size equal to 1.

```
starp_size_t pearg_t::number_of_elements () const [inline]
```

Returns the total number of elements in this argument. For strings, this will be the string length.

```
starp_size_vector_t const& pearg_t:: size_vector () const [inline]
```

Returns a vector containing the size of each dimension.

```
std::string const& pearg_t::string_data () const
```

Returns a string value if this is of type STRING.

> **Note:** Throws `ppevalc_exception_t` if `element_type` is not STRING.

ppevalc_module_t Class Functions

This class provides the interface for ppeval modules to interact with the starpserver runtime.

```
#include <ppevalc_module_h>
```

Public Member Functions: Input

```
void add_function (std::string const &func_name,
ppevalc_user_function_t func)
```

Registers a new function with the module.

Public Member Functions: Return

```
std::ostream & log
```

Returns a reference to an ostream which could be used by user functions to log messages.

Member Functions

Function

```
void ppevalc_module_t::add_functions (std::string const & funct_name
ppevalc_user_function_t func)
```

Description

Registers a new function with the module. The function pointer is accessbile using the `get_function` method.

Parameters

The following table describes the parameters:

Table 4: Function Parameters

Type	Description
`func_name`	A string to retrieve the associated function pointer. Later it can be used as a look-up key with the `get_function` method. If another function pointer was previously registered with the same name, it will be replaced.
`func`	The function pointer to associate with "`name`".

Function

```
std::ostream& ppevalc_module_t::log() [inline]
```

Returns

Returns a reference to an `ostream` which should be used by user functions to log messages.

Example using Star-P SDK for Task Parallel Functions

Below is an example that implements the cumulative sum function in C++. It demonstrates how you can

- pass input argument data directly to an external function, and,
- have the external data write directly into an output argument without making extra copies of the data.

For output arguments, this requires that the external function in question supports being instructed where to write its result. In this example, the C++ standard library's partial sum function is used.

```
static void cumsum(ppevalc_module_t& module, pearg_vector_t const&
inargs, pearg_vector_t& outargs)
{
    // check input and output arguments
    if (inargs.size() != 1)
    {
        throw ppevalc_exception_t("Expected 1 input argument");
    }
if (!inargs[0].is_vector())
{
    throw ppevalc_exception_t("Expected input arguments to be a
vector");
}
if (inargs[0].element_type() != pearg_t::DOUBLE)
{
    throw ppevalc_exception_t("cumsum only supports arrays of
doubles");
}
if (outargs.size() != 1)
{
    throw ppevalc_exception_t("Expected 1 output argument");
}
// create an output argument of the same type and shape as the input
argument pearg_t outarg(inargs[0].element_type(),
inargs[0].size_vector());

// call external function, telling to read its input args directly
// from inargs, and telling it to write its result directly into the
// outarg

starp_double_t const* indata = inargs[0].data<starp_double_t>();
starp_double_t *outdata = outarg.data<starp_double_t>();

std::partial_sum(indata, indata + inargs[0].number_of_elements(),
outdata);

outargs[0] = outarg;
}
```

Client-side Task Parallel Functions for the Star-P SDK

Using the Star-P SDK to load, invoke, and unload compiled C++ code for task parallel functionality involves the use of three different sets of functions.

- `ppevalcloadmodule` - This function loads a compiled C++ module to the server
- `ppeval/ppevalsplit` - These are the standard client side interfaces for all task parallel functionality. This section provides specific instructions for using ppeval or ppevalsplit to call compiled C++ codes. For more extensive information on using ppeval for task parallelism in VHLL environments, see the “Star-P® MATLAB® Programming Guide”.
- `ppevalcunloadmodule` - This function unloads a previously loaded compiled C++ module from the server.

Loading a compiled C++ module on the Star-P server

To load a previously compiled C++ package (a dynamic shared object (DSO) or .so file), use the function `ppevalcloadmodule`. This function is called in the followiing manner with two input arguments and one output argument:

```
sym_name = ppevalcloadmodule('module_name','sym_name');
```

- Input
  - `'module_name'` - the pathname of a valid DSO package that is located on the server machine.
  - `'sym_name'` - the symbolic name that you choose to assign to the module. It is used whenever the module is invoked within a ppeval call from the client.
- Output
  - `sym_name` - the output returns the symbolic module name.

This function can also be called with only a single input argument as follows:

```
sym_name = ppevalcloadmodule('module_name');
```

- Input
  - '`module_name`' - the pathname of a valid DSO package that is located on the server machine. The unused second argument defaults to '`module_name`' with any leading path removed and any file extension also removed.
- Output
  - ‘`sym_name`' - the output returns the symbolic module name.

Calling a compiled C++ function using ppeval

When using ppeval to run a compiled C++ code that has previously been loaded to the server, there are some slight differences from calling ppeval by passing a standard VHLL function. The calling syntax of `ppeval` for compiled code is the following:

```
output_arg = ppeval('C://sym_name:foo',input_args);
```

The first argument is a string that includes a `C://` directive, the symbolic name of the module, and the function which you desire to run in task parallel. The `C://` prefix may look almost like a Windows device name, but in fact it denotes that the routine to be called is written in C++, as opposed to a VHLL code. The second argument, and any further arguments that would need to be passed to the function, are treated in the same manner as any other ppeval call. These arguments need to have dimensions such that they can be either distributed or broadcast across the available processors. For more information on the required properties of these later input arguments, see “Task Parallelism with Star-P and MATLAB” in the “Star-P® MATLAB® Programming Guide”.

Unloading a compiled C++ module from the Star-P server

While you are creating and debugging anything within your module, you will have to unload the module before loading a new version. Unloading a previously loaded module can be performed using the `ppevalcunloadmodule` function as follows:

```
ppevalcunloadmodule(sym_name);
```

This function takes a single input argument, which is the symbolic name that was defined when loading the module via `ppevalcloadmodule`.

Application Example

Here we provide an extended application example for porting an existing application code for task parallel operation in Star-P. Note that this example uses Star-P solely for the expression of parallelism. With the exception of the number of bodies found in the input file, none of the data read in or created by the existing C++ routines are made into native Star-P variables that could be operated on by other Star-P functions. That simplifies the modification of this particular example code for parallelism, but limits the ability for the example C++ routines to be augmented by other Star-P functionality.

In this section, we demonstrate the following:

1. The changes necessary to an existing C++ program to interface it with the Star-P SDK, including the linkage between C++ and Star-P data types.
2. The Star-P client code needed to exercise the C++ routines in parallel.
3. Tips for enabling different C++ routines to communicate with each other.

4. The fine points to be understood for building proper C++ routines for the Star-P SDK interface.

The code used for this illustration is from a workflow involving finite-element analysis. It takes an input file that has descriptions of the bodies that are components of the analysis (e.g., number of joints, gravitational acceleration, components of reaction forces, Euler parameters, and translational and rotational acceleration), all per time-step, and writes a distinct file per body with all that information.

Not all types of parallelism are appropriate for this approach. If the parallelism is expressed via a loop, the iterations must be completely independent of communication, including not only through immediate variables but also through data structures that are malloced. For instance, if the loop has a variable in it that carries some output from one iteration into the next iteration, that would not fit with this approach. (Also, see “Changing a Reduction to Fit this Approach” for a description of simple changes that can make routines that don’t quite fit this approach work properly.) Star-P does not do any analysis of the C++ code to ensure that it truly is appropriate; rather, it depends on your knowledge of the code and your assertion that the code is completely independent.

Implementation Overview

The implementation shown here has two major steps:

1. Split the original `main.C` routine into its two major parts, one of which reads the input file, and the other of which loops over the number of bodies writing a file for each, and make each of these a function callable via the SDK. In the resulting parallel version, the input file is read redundantly on all processors, and then the work of writing the per-body files is divided among the processors. This code executes on the server.
2. Write a Star-P routine, in MATLAB, that calls the C++ routines in parallel. This code executes on the client.

See the sections “Complete Original Code” and “Complete Final Code” to see the overall changes presented below.

Code Structure

The original file `main.C` had the following code structure (full code in “Complete Original Code”)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "simulation.h"

int main ( int argc, char** argv )
{
    FILE *out_file;
    //  […]
    int i,j,k,l;

    simulation* simultn =// POINT 1
           new simulation("test.in", 3, 1, 0);

    char** bodies = simultn->get_body_list();
    int num_bodies = simultn->get_num_bodies();

    /* loop that should be parallelized */
    for (i=0; i<num_bodies; i++)// POINT 2
    {
        bodyInfo* bi = simultn->get_body_info(bodies[i], error);
       //  […]
        char * path = "./";
        strcpy(out_file_name, path);
        strcat(out_file_name, bodies[i]);
        strcat(out_file_name, ".dat");

        printf("\n Output file =%s\n", out_file_name);
        out_file = fopen(out_file_name, "w");

        int numTimeSteps = simultn->get_num_time_steps();
        //{1} Total number of time steps
        fprintf(out_file, "%d\n", numTimeSteps);
        //{2} Gravitational acceleration
        double* vals = simultn->get_grav_acceleration();
        fprintf(out_file, "%.5f %.5f %.5f\n", vals[0], vals[1],
vals[2]);
        delete [] vals;
        //  […]
```

```
         fclose(out_file);
        delete bi;
        free(bodies[i]);
    }// END POINT 2
    delete [] bodies;
    delete simultn;

    printf("\n Done generating .dat files!\n");
    return(1);
}
```

The first major portion of the code, denoted by "`POINT 1`", is the call to the `simulation` constructor, which reads the input file specified by the first parameter and creates the in-memory data structures representing that input data. The program passes its data between the two major portions via in-memory data structures. For the second portion to have all the necessary data, each processor that will participate in the second portion must read all the input data into its own memory, redundantly with all the other processors. (One can imagine a more efficient approach where each processor would read only the portions of the input file that it will ultimately process, but that requires significantly more invasive changes to the C++ code, which we chose to avoid here. Also, the I/O time is less than 10% of the execution time, so it does not dominate the performance.) The second major portion of the code is the `for` loop, encompassed by "`POINT 2`" and "`END POINT 2`"and including almost the rest of the original routine, where the data for each of the bodies in the input file is written to a distinct file, and there are no returned values within the program. This portion was changed to execute in an embarrassingly parallel mode.

Code Changes

Modifying the Existing Routines

Conceptually, we want to change the existing code to look something like (leaving out details for simplicity):

```
int main ( int argc, char** argv )
{
    simulation* simultn =
           new simulation("test.in", 3, 1, 0);

    char** bodies = simultn->get_body_list();
    int num_bodies = simultn->get_num_bodies();

    /* loop that should be parallelized */
    for (i=0; i<num_bodies; i++)    {
        loop_body(simultn, i);
    }
```

```
    delete [] bodies;
    delete simultn;

    printf("\n Done generating .dat files!\n");

    return(1);
```

The key change here is that all the code inside the primary `for` loop is now inside a subroutine, which can be called in parallel. This change by itself is not enough to enable the code to run in parallel, because the `main` routine here is still in C++, which does not provide a simple way to distribute the iterations of the loop across the processor cores of a distributed memory computer. To make this second change, we need to change the simplified `main` routine to run from the Star-P client. (We use the MATLAB language below, but Python could also be used.

```
nbody = peval('c://body:SDK_read_body_list',...
              '/home/demo/spr/codes/TOY/test.in');
nbody = max(nbody);
ign =  ppeval('c://body:SDK_loop_body',0:(nbody-1)*p);
```

The first and last lines of code use the Star-P `ppeval` construct, which denotes parallel execution of the routine specified by the first argument, with the input argument(s) specified. The first line calls a yet-to-be-defined routine (`SDK_read_body_list`) that provides the SDK linkages to the `read_body_list routine`. The input argument is the name of the input file, which is broadcast to all participating processors. Because the only input argument is a string (the file name), which by default is broadcast, there is no splitting of work that happens with this call; all the processors execute the same code redundantly and return the number of bodies found in the input file, which should be identical on all processors. The second line reduces this vector of identical values to a scalar for determining the loop bounds in the next line. The third line calls another yet-to-be-defined routine (`SDK_loop_body`) that provides the SDK linkages to the `loop_body` routine. The input argument is a row vector of body numbers that are to be processed by the parallel iterations. Since the underlying C++ code numbers bodies from 0, the iteration counts have to reflect that, instead of the 1-based numbering common to MATLAB. Since the input argument is a row vector, Star-P will split the work across the available processors. The real output of loop_body is the per-body files, but currently the `ppeval` construct requires at least one return value, which is assigned to the MATLAB-conventional `ign` (ignore) variable.

Now let's look at the `read_body_list` and loop_body routines. They look very similar to portions of the original code. `read_body_list`, for instance, primarily consists of the new of a simulation object. There is one important addition to this code, however, having to do with the fact that what used to be parts of the single main function are now distinct routines (`read_body_list` and `loop_body`), and hence passing data between them cannot be done with stack variables. The only piece of data that must be explicitly passed is the pointer

to the simulation object, which is done through a global (defined outside any function) C++ variable.

```
simulation* sim_addr;
int read_body_list (const char *simulation_name)
{
        simulation* simultn =
           new simulation(simulation_name, 3, 1, 0);
        sim_addr = simultn;
        return(simultn->get_num_bodies());
}
```

Similarly, the loop_body routine looks like the remainder of the original `main` routine, without the creation of the simulation object (done above) and without the for loop, which is now effectively done within Star-P, and with the address of the simulation object passed through `sim_addr` as described above. The code is shown in Complete Final Code below.

Including Star-P Header Files

The Star-P SDK defines a set of C++ data types and linkage routines to simplify the process of linking in your own routines. Those are captured in the include files pearg.h and ppevalc_module.h. For the purposes of this code, the following lines were added before the new routines written to link between the Star-P SDK and the modified routines from the existing code.

```
#define NO_SDK_DEPRECATED_METHODS TRUE
#include <pearg.h>
#include <ppevalc_module.h>
```

Creating New Routines to Use the Star-P SDK Linkages

The `SDK_read_body_list` routine uses the SDK structures that are defined and implemented by Star-P. Thus, you'll note that even though the MATLAB call above to `SDK_read_body_list` has only a single input argument, the actual definition of `SDK_read_body_list` has 3 arguments. The first of these is the opaque `ppevalc_module_t` object, and then the 2nd and 3rd are the input and output argument lists (inargs and outargs, respectively). Handling these input and output argument lists, in light of the arguments of the underlying routines to be called, is the primary function of the SDK linkage routine. In the case of `SDK_read_body_list`, the routine does basic error-checking on the number of expected arguments, gathers the filename argument string from inargs, ensuring that it is passed as a C++ string, and calls the underlying

`read_body_list` routine, whose return value (`nbody`) is then placed into the output arguments directly, since it's a scalar. (Returning an array or string would be more involved.)

```
static void SDK_read_body_list(ppevalc_module_t& module,
pearg_vector_t const& inargs, pearg_vector_t& outargs)
{
        if (inargs.size() != 1) {
                throw ppevalc_exception_t
                      ("expected 1 input argument");
        }
        if (outargs.size() != 1) {
                throw ppevalc_exception_t
                      ("expected 1 output argument");
        }
        string indata = inargs[0].string_data();
        int nbody = read_body_list(indata.c_str());
        outargs[0] = nbody;
}
```

Similarly, the routine processes its input and output arguments as dictated by the needs of the situation. Note that the specifics of querying the input arguments are different depending on whether it is a scalar, an array, or a string. Note also that all Star-P data is by default double-precision floating-point data, so any use of that data in another format (integer, in this case) requires explicit casting. The output argument is hard-coded since it's only there because Star-P requires an output argument.

```
static void SDK_loop_body(ppevalc_module_t& module, pearg_vector_t
const& inargs, pearg_vector_t& outargs)
{
        if (inargs.size() != 1) {
                throw ppevalc_exception_t
("expected 1 input argument");
        }
        if (inargs[0].number_of_elements() != 1) {
                throw ppevalc_exception_t
("expected input arg to be scalar");
        }

        const starp_double_t* p = inargs[0].data<double>();
        int indata = (int)p[0];
        loop_body(indata);

        // hard-coded output value
        outargs[0] = 1;
}
```

In addition to the specific routines that link to each of your routines, you need to create a routine that informs Star-P, at the time you load your package (see Loading Your SDK Package into Star-P below), about the routines in the package that you want to be visible to Star-P. For the example, that routine will look like the following. Note that the

`ppevalc_module_init` name is expected by Star-P, by convention, and hence needs to be used unchanged.

```
extern "C"
void ppevalc_module_init(ppevalc_module_t& module)
{
  module.add_function("SDK_read_body_list",
    &SDK_read_body_list);
  module.add_function("SDK_loop_body", &SDK_loop_body);
}
```

Preserving C++ Data between ppeval Calls

As described briefly above in Modifying the Existing Routines, passing data among different C++ routines plugged in via the SDK requires some care. In this example, the original code passed the address of the simulation object by a local variable, which worked by virtue of all the code being within the single main routine. When converting this to multiple routines, each of which can be called in parallel, the address of the simulation object has to be passed through a variable that persists on each processor. One example of this is a global C variable.

It's important to remember that data stored in your C++ package is unknown to the rest of Star-P, which will have no access to your variables. This can be a good thing or a bad thing. On the positive side, it means that Star-P will not change your variables when you weren't expecting it, and your variables will be preserved exactly as you left them on the previous call to one of your routines. On the negative side, your variables will have little means of interaction with other Star-P functions or variables. If it's important to have your routines and their input/output variables interact more closely with other existing Star-P functions, you may want to consider the techniques used in "Amesos *p: Using the Star-P SDK to Interface with High-performance Numerical Libraries".

Compiling Your Routines into an SDK Package

Once you have all your routines written as described above, you need to compile them into a package (a dynamic shared object (DSO), or `.so` file) that can be loaded into Star-P. The

following routines must typically be made to a makefile or build script to work properly with the Star-P SDK.

- Include the location of the Star-P SDK header files in the compile line for each module that uses the header files described in Including Star-P Header Files. If the install location is in the shell variable `STARP`, then the following should be added to the appropriate compile commands.

```
-I $(STARP)/generic/sdk/include -I$(STARP)/sdk/include
```

- The compile commands must also generate 64-bit-safe and position-independent code, i.e.

```
-DPPINDEX64 -fpic
```

- The link command needs to include the location of the built-in Star-P linkage routines, the option to create a .so, and the math library, i.e.

```
-L$(STARP)/ia64_linux/sdk/lib -shared -lm
```

Loading Your SDK Package into Star-P

So far, we have walked through the steps of creating the routines that execute on the Star-P client, the SDK linkage routines between Star-P and your routines that do real work, and those real work routines themselves. Once you have done that and have created the package, you need to tell Star-P about your package. In this example, that is done via the following command to the MATLAB Star-P client.

```
body = ppevalcloadmodule('SDK_body_compute.so','body');
```

The `ppevalcloadmodule` command takes two arguments, the first of which is the pathname to the DSO containing the package, and the second of which is a user-chosen name by which the module will be known in later ppeval calls. The line above would need to precede the code that executes the SDK routines (also shown above in Modifying the Existing Routines). The body name used as the second argument to ppevalcloadmodule must match the module part of the routine name passed to ppeval (e.g., `c://body:SDK_read_body_list` below). The `c://` prefix may look almost like a Windows device name, but in fact denotes that the routine to be called is written in C++, as opposed to MATLAB or Python.

```
nbody = ppeval('c://body:SDK_read_body_list',...
               '/home/demo/spr/codes/TOY/test.in');
nbody = max(nbody);
ign =  ppeval('c://body:SDK_loop_body',0:(nbody-1)*p);
```

Once you've created your package, you might load and execute it as simply as shown. While you're creating and debugging your package, you will have to unload the package before loading a new version.

```
ppevalcunloadmodule(body);
```

Summary

In this example we have shown how to take an existing C++ program, separate it into major chunks that can be run in parallel, and then fit those chunks into the Star-P SDK interface and run them in parallel. This example depended on each processor's reading the entire input, on which the work distribution step counted to achieve parallelism.

Other Considerations

Changing a Reduction to Fit this Approach

The introduction to this chapter states clearly that the parallelism to be exploited via this approach must be completely independent of communication and synchronization between iterations of a parallel loop. That may appear restrictive, but there are a number of techniques that can be used to convert a code that almost, but not quite, matches this approach to match it fully.

Consider the example below, where the routine `compute_value`, which is presumed to be computationally expensive, is called for each element of an array (and assume that it only depends on the specific value it's passed, and doesn't change other internal or global variables that would be used on subsequent iterations), and the return values from the routine are summed. By the rules of this approach, this code snippet does not work.

```
double sum = 0.0;
for (i=o; i<max; i++) {
    sum += compute_value(array[i]);
}
```

But consider a small change to this snippet, shown below. The new version places the return values from `compute_value` into an array indexed by the iteration number. This change allows the first loop below to be completely independent and suitable for running in parallel by the methods of this chapter. The second loop sums those return values into the single value, preserving the semantics of the original code. The second loop is not suitable for parallel execution, but if `compute_value` is the time-consuming portion of the computation, that is not a significant drawback. (Note that in practice, the second loop would need to reside

ultimately in the client code, where it would have access to the results of all the iterations, which it would not have if it resided in the server code, as shown here.)

```
double retvals[max];
for (i=o; i<max; i++) {
    retval[i] = compute_value(array[i]);
}
double sum = 0.0;
for (i=o; i<max; i++) {
    sum += retval[i];
}
```

Complete Original Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "simulation.h"

typedef char Boolean;
#define TRUE    1
#define FALSE   0

int main ( int argc, char** argv )
{
        FILE *out_file;
        char out_file_name[80];
        char Line[300];
        char target_line[300];
        char f_name_str_1[200];
        char s0[40], s1[40], s2[40], s3[40], s4[40];
        char temp_s0[40], temp_s1[40];
        int time_steps_index = 0;
        int euler_index = 0;
        int trans_acc_index = 0;
        int rot_vel_index = 0;
        int rot_acc_index = 0;
        char id_string[30];
        int i,j,k,l;

        const char *gink = "test.in";
        simulation* simultn =
           new simulation(gink, 3, 1, 0);

        int error = 0;

        char** bodies = simultn->get_body_list();
        int num_bodies = simultn->get_num_bodies();

        /* loop that should be parallelized */
```

```
for (i=0; i<num_bodies; i++)
{
bodyInfo* bi = simultn->get_body_info(bodies[i], error);

if (error == -1)
{
        printf("Body name entered is incorrect!\n");
        delete simultn;
        exit(1);
}
else if (error == -2)
{
        printf("No information for body found!\n");
        delete simultn;
        exit(1);
}
else if (error == -3)
{
        printf("Transformation matrix is singular!\n");
        delete simultn;
        exit(1);
}
else if (error == -4)
{
        printf("*.fms doesn't exist!\n");
        delete simultn;
        exit(1);
}

//initialize
char * path = "./";

strcpy(out_file_name, path);
strcat(out_file_name, bodies[i]);
strcat(out_file_name, ".dat");

printf("\n Output file =%s\n", out_file_name);
out_file = fopen(out_file_name, "w");

int numTimeSteps = simultn->get_num_time_steps();
//{1} Total number of time steps
fprintf(out_file, "%d\n", numTimeSteps);

//{2} Gravitational acceleration
double* vals = simultn->get_grav_acceleration();
fprintf(out_file, "%.5f %.5f %.5f\n", vals[0], vals[1], vals[2]);
delete [] vals;

//{3} Total number of joints //aaa
int numFrames = bi->get_num_frames();
fprintf(out_file, "%d\n", numFrames);

//{4} Time steps
```

```
//vals = bi->get_time_steps();
vals = simultn->get_time_steps();
for (j=0; j<numTimeSteps; j++)
        fprintf(out_file, "%.5f\n",vals[j]);
delete [] vals;
fputc('\n', out_file);

//{5} Assembly reference frame names
char** frameNames = bi->get_frame_names();
for (j=0; j<numFrames; j++)
{
        fprintf(out_file, "%s ", frameNames[j]);
}
fputc('\n', out_file);
for (j = 0; j<numFrames; j++)
        free(frameNames[j]);
delete [] frameNames;

//{6} Six components of reaction forces

int numOfTargetBodies = bi->get_num_target_bodies();
vals = bi->get_reaction_forces();
for (j=0; j<numTimeSteps; j++)
{
    for (l=0; l<numOfTargetBodies; l++)
    {
        for (k=0; k<5; k++)
            fprintf(out_file, "%.6e ",
                vals[6*(numTimeSteps*l+j)+k]);
        fprintf(out_file,"%.6e\n",vals[6*(numTimeSteps*l+j)+5]);
    }
    fprintf(out_file, "\n");
}
delete[] vals;

//{7} Euler parameters

vals = bi->get_euler_params();
for (j=0; j<numTimeSteps; j++)
        fprintf(out_file,"%.6e %.6e %.6e %.6e\n",vals[4*j],
                vals[4*j+1],vals[4*j+2],vals[4*j+3]);
delete[] vals;
fputc('\n', out_file);

//{8} Translational acceleration

vals = bi->get_trans_acc();
for (j=0; j<numTimeSteps; j++)
        fprintf(out_file,"%.6e %.6e %.6e\n",vals[3*j],
                vals[3*j+1],vals[3*j+2]);
delete[] vals;
fputc('\n', out_file);
```

```
        //{9} Rotational velocity

        vals = bi->get_rot_vel();
        for (j=0; j<numTimeSteps; j++)
                fprintf(out_file,"%.6e %.6e %.6e\n",vals[3*j],
                        vals[3*j+1],vals[3*j+2]);
        delete[] vals;
        fputc('\n', out_file);

        //{10} Rotational acceleration

        vals = bi->get_rot_acc();
        for (j=0; j<numTimeSteps; j++)
                fprintf(out_file,"%.6e %.6e %.6e\n",vals[3*j],
                        vals[3*j+1],vals[3*j+2]);
        delete[] vals;

        fclose(out_file);
        delete bi;

        free(bodies[i]);
        }
        delete [] bodies;
        delete simultn;

        printf("\n Done of generating .dat files!\n");
        return(1);

}
```

Complete Final Code

```
---------------myload.m (loads the new package into Star-P)------------------
body = ppevalcloadmodule('/home/demo/codes/SDK_body_compute.so','body')

--------toy.m (executes the new SDK routines from the Star-P client)---------
nbody = ppeval('c://body:SDK_read_body_list', '/home/demo/codes/test.in');
nbody = max(nbody);
ign =  ppeval('c://body:SDK_loop_body',0:(nbody-1)*p);

---SDK_body_compute.C (modified original routines plus new SDK linkage code)--
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "simulation.h"

typedef char Boolean;
#define TRUE    1
#define FALSE   0

simulation* sim_addr;
```

```
using namespace std;

int read_body_list (const char *simulation_name)
{
//      FILE *out_file;
//      char out_file_name[80];
//      char Line[300];
//      char target_line[300];
//      char f_name_str_1[200];
//      char s0[40], s1[40], s2[40], s3[40], s4[40];
//      char temp_s0[40], temp_s1[40];
//      int time_steps_index = 0;
//      int euler_index = 0;
//      int trans_acc_index = 0;
//      int rot_vel_index = 0;
//      int rot_acc_index = 0;
//      char id_string[30];
        int i,j,k,l;

        simulation* simultn =
           new simulation(simulation_name, 3, 1, 0);
        sim_addr = simultn;

//      int error = 0;
//
//      char** bodies = simultn->get_body_list();
//      int num_bodies = simultn->get_num_bodies();
//
//      /* loop that should be parallelized */
//      for (i=0; i<num_bodies; i++)
//      {
//      bodyInfo* bi = simultn->get_body_info(bodies[i], error);
//
//              [...]
//
//      free(bodies[i]);
//      }
//      delete [] bodies;
//      delete simultn;
//
//      printf("\n Done of generating .dat files!\n");
        return(simultn->get_num_bodies());

}

int loop_body(int i)
{
        FILE *out_file;
        char out_file_name[80];
        char Line[300];
        char target_line[300];
        char f_name_str_1[200];
        char s0[40], s1[40], s2[40], s3[40], s4[40];
```

```
        char temp_s0[40], temp_s1[40];
        int time_steps_index = 0;
        int euler_index = 0;
        int trans_acc_index = 0;
        int rot_vel_index = 0;
        int rot_acc_index = 0;
        char id_string[30];
        int j,k,l;

//      simulation* simultn =
//         new simulation("test.in", 3, 1, 0);
        simulation* simultn = sim_addr;

        int error = 0;

        char** bodies = simultn->get_body_list();
        int num_bodies = simultn->get_num_bodies();

//      /* loop that should be parallelized */
//      for (i=0; i<num_bodies; i++)
//      {
        bodyInfo* bi = simultn->get_body_info(bodies[i], error);

        if (error == -1)
        {
                printf("Body name entered is incorrect!\n");
                delete simultn;
                exit(1);
        }
        else if (error == -2)
        {
                printf("No information for body found!\n");
                delete simultn;
                exit(1);
        }
        else if (error == -3)
        {
                printf("Transformation matrix is singular!\n");
                delete simultn;
                exit(1);
        }
        else if (error == -4)
        {
                printf("*.fms doesn't exist!\n");
                delete simultn;
                exit(1);
        }

        //initialize
        char * path = "./";

        strcpy(out_file_name, path);
        strcat(out_file_name, bodies[i]);
```

```
strcat(out_file_name, ".dat");

printf("\n Output file =%s\n", out_file_name);
out_file = fopen(out_file_name, "w");

int numTimeSteps = simultn->get_num_time_steps();
//{1} Total number of time steps
fprintf(out_file, "%d\n", numTimeSteps);

//{2} Gravitational acceleration
double* vals = simultn->get_grav_acceleration();
fprintf(out_file, "%.5f %.5f %.5f\n", vals[0], vals[1], vals[2]);
delete [] vals;


//{3} Total number of joints //aaa
int numFrames = bi->get_num_frames();
fprintf(out_file, "%d\n", numFrames);

//{4} Time steps
//vals = bi->get_time_steps();
vals = simultn->get_time_steps();
for (j=0; j<numTimeSteps; j++)
        fprintf(out_file, "%.5f\n",vals[j]);
delete [] vals;
fputc('\n', out_file);

//{5} Assembly reference frame names
char** frameNames = bi->get_frame_names();
for (j=0; j<numFrames; j++)
{
        fprintf(out_file, "%s ", frameNames[j]);
}
fputc('\n', out_file);
for (j = 0; j<numFrames; j++)
        free(frameNames[j]);
delete [] frameNames;

//{6} Six components of reaction forces

int numOfTargetBodies = bi->get_num_target_bodies();
vals = bi->get_reaction_forces();
for (j=0; j<numTimeSteps; j++)
{
    for (l=0; l<numOfTargetBodies; l++)
    {
        for (k=0; k<5; k++)
            fprintf(out_file, "%.6e ",
                vals[6*(numTimeSteps*l+j)+k]);
        fprintf(out_file,"%.6e\n",vals[6*(numTimeSteps*l+j)+5]);
    }
    fprintf(out_file, "\n");
}
```

```
        delete[] vals;

        //{7} Euler parameters

        vals = bi->get_euler_params();
        for (j=0; j<numTimeSteps; j++)
                fprintf(out_file,"%.6e %.6e %.6e %.6e\n",vals[4*j],
                        vals[4*j+1],vals[4*j+2],vals[4*j+3]);
        delete[] vals;
        fputc('\n', out_file);

        //{8} Translational acceleration

        vals = bi->get_trans_acc();
        for (j=0; j<numTimeSteps; j++)
                fprintf(out_file,"%.6e %.6e %.6e\n",vals[3*j],
                        vals[3*j+1],vals[3*j+2]);
        delete[] vals;
        fputc('\n', out_file);

        //{9} Rotational velocity

        vals = bi->get_rot_vel();
        for (j=0; j<numTimeSteps; j++)
                fprintf(out_file,"%.6e %.6e %.6e\n",vals[3*j],
                        vals[3*j+1],vals[3*j+2]);
        delete[] vals;
        fputc('\n', out_file);

        //{10} Rotational acceleration

        vals = bi->get_rot_acc();
        for (j=0; j<numTimeSteps; j++)
                fprintf(out_file,"%.6e %.6e %.6e\n",vals[3*j],
                        vals[3*j+1],vals[3*j+2]);
        delete[] vals;

        fclose(out_file);
        delete bi;

        free(bodies[i]);
        return(1);
        }
//      delete [] bodies;
//      delete simultn;

//      printf("\n Done of generating .dat files!\n");
//      return(1);

//}

//
//  Modified from example on pp. 60-61 of Star-P User Guide (rel 2.4.1).
```

```
//
#define NO_SDK_DEPRECATED_METHODS TRUE
#include <pearg.h>
#include <ppevalc_module.h>

static void SDK_read_body_list(ppevalc_module_t& module, pearg_vector_t const&
inargs, pearg_vector_t& outargs)
{
        if (inargs.size() != 1) {
                throw ppevalc_exception_t("expected 1 input argument");
        }
        if (outargs.size() != 1) {
                throw ppevalc_exception_t("expected 1 output argument");
        }

        string indata = inargs[0].string_data();

        int nbody = read_body_list(indata.c_str());

        outargs[0] = nbody;
}
static void SDK_loop_body(ppevalc_module_t& module, pearg_vector_t const&
inargs, pearg_vector_t& outargs)
{
        if (inargs.size() != 1) {
                throw ppevalc_exception_t("expected 1 input argument");
        }
        if (inargs[0].number_of_elements() != 1) {
                throw ppevalc_exception_t("expected input argument to be a
scalar");
        }

        const starp_double_t* p = inargs[0].data<double>();
        int indata = (int)p[0];

        loop_body(indata);

        // hard-coded output value
        outargs[0] = 1;
}

extern "C"
void ppevalc_module_init(ppevalc_module_t& module)
{
  module.add_function("SDK_read_body_list", &SDK_read_body_list);
  module.add_function("SDK_loop_body", &SDK_loop_body);
}
```

INTER*CTIVE supercomputing

Chapter 4

Star-P SDK Class Lists

This chapter contains information on the contents of classes for the Star-P SDK related to both data parallel operation and task parallel operation.

Star-P SDK for Data Parallel Operations

Here are the classes, structs, unions and interfaces with brief descriptions:

initializer_t Struct Reference	A virtual base class users override to allow matrix factory to initialize matrices in an arbitrary way
layout_desc_t Class Reference	The layout descriptor describes how a matrix is distributed out among the processes in a MPI communicator
matrix_factory_t Class Reference	A factory class for creating new matrices
matrix_t Class Reference	A user-level interface for accessing and manipulating distributed matrices
PPcsr Struct Reference	
smart_ptr< T >	A reference counted smart_ptr class for scalar objects
smart_ptr_array< T >	A reference counted smart pointer class for arrays of objects
starp_sdk_t Class Reference	StarP SDK classes provide encapsulation of StarP runtime for custom user code
starp_value_t Class References	Holds an input or output argument of a user function
type_desc_t Class Reference	Identifies the type of data contained in a distributed matrix; provides access to the MPI properties of the elements of the matrix -- their size and type

starp_sdk_t Class Reference

StarP SDK classes provide encapsulation of StarP runtime for custom user code.

```
#include <StarpSDK.h>
```

Detailed Description

Public Member Functions	
	starp_sdk_t (PPServer *server, SdkPackage *package)
	~**starp_sdk_t** ()
bool	**is_root** () const Check to see if current process is root.
int	**my_rank** () const Return the MPI rank of the current process.
int	**num_processes** () const Return the number of processes in this MPI group.
int	**root_rank** () const Return the rank of the root process.
MPI_Comm	**communicator** () const Get the ID if the MPI communicator this process is associated with.
matrix_factory_t	matrix_factory
	Create a new matrix factory.
void	**set_error** (std:: string const &msg)
	Indicate crucial error.
void	**reset_error** ()
	Reset the error state to no-error.
bool	**is_error_set** () const
	Return true if error state is set.
std::string	**last_error** () const
	Return the last error message string set with set_error, or an empty string if none.
void	**register_func** (char *fname, SdkFunc fptr) Register user function with the server.
PPServer &	**server**()

Star-P SDK classes provide encapsulation of Star-P runtime for custom user code.

Star-P SDK classes allow custom user code running on the server to receive inputs from StarP; inspect and alter properties of matrices; create and modify new matrices, and return results back to Star-P.

The two types of data contained in the matrices made available by SDK are `starp_double_t` for Real values and `starp_dcomplex_t` for complex. All functions that provide direct access to the data are templated on these two types.

Here are the main concepts of the SDK:

1. Each user function that uses the SDK begins with the following code:

```
extern "C"
void test_redist (PPServer &inServer, PPArgList &inArgs,
                  PPArgList &outArgs)
 {
      starp_sdk_t sp = starp_sdk_t (inServer, inArgs, outArgs);
      ...
 }
```

2. **`starp_sdk_t`** class is responsible for all input and output argument passing as well as error reporting, for example:

```
sp.get_input_matrix(0);
        sp.set_error ("Incorrect number of inputs");
```

3. Star-P matrices are available via a smart pointer matrix_ptr class

4. A matrix has a number of objects describing its properties, these objects are available via

```
        // sparse or dense
        storage_desc_t sd = mA->storage_desc();
        // row or column distribuetd dimensions and
       // distribution dimension
        layout_desc_t = mA->layout_desc();
        // datatype mapping to MPI datatypes
        type_desct_t = mA->type_desc();
```

5. Data access is templated on the type of data contained in a matrix, the two currently supported types are starp_double_t and starp_dcomplex_t, for example:

```
        if (mA->data_type_is<starp_double_t>()) {
                ELTYPE *aData = mA->local_dense_data<starp_double_t>();
                ...
        } else {
                starp_dcomplex_t *aData =
mA->local_dense_data<starp_dcomplex_t>();
                ...
        }
```

You can test for a data type and specify in what form to get the data.

6. Exceptions are used extensively by the SDK to guard against invalid user actions. When a user uses an invalid index, process rank or datatype, SDK code throws an std::exception or a subclass of it. Such an exception should be handled in the try/catch block surrounding the user code, for example

```
        try {
                ... SDK code here
        } catch (exception& e) {
                sp.set_error (e.what());
        }
```

7. Once an error is set in the body of a function, no additional return parameters can be added; trying to do so will result in an exception being raised.

8. To register user functions with a server, they need to be included in a starpInit call as in

```
        extern "C" int starpInit (PPPackage *pack) {
                starp_sdk_t::register_func(pack, "sdk_add_anyd",
&ex_add_anyd);
                return 0;       // Return 0 to indicate no error
        }
```

9. To call the registered function from MATLAB, use a `ppinvoke()` command, for example

```
       ppinvoke ('sdk_add_anyd', a1, a2);
```

Definition at line 962 of file **`StarpSDK.h`**.

Constructor & Destructor Documentation

```
starp_sdk_t::starp_sdk_t(PPServer &  server
                         SdkPackage* package
                         )
```

```
starp_sdk_t::~starp_sdk_t()
```

Member Function Documentation

```
MPI_Comm starp_sdk_t::communicator (   ) const
```

Get the ID if the MPI communicator this process is associated with.

Returns:

MPI Communicator ID

```
bool starp_sdk_t::is_error_set ( ) const
```

Return true if error state is set.

The error state can be set by calling set_error(), and can be reset to no error by calling reset_error()

```
bool starp_sdk_t::is_root ( ) const
```

Check to see if current process is root.

Returns:

- true is MPI process rank is zero,
- false otherwise

```
std::string starp_sdk_t::last_error  (    )  const
```

Return the last error message string set with set_error, or an empty string if none.

```
matrix_factory_t starp_sdk_t::matrix_factory  (    )
```

Create a new matrix factory.

Returns:

An instance of a matrix factory initialized to work with the calling StarP server

```
int starp_sdk_t::my_rank  (    )  const
```

Return the MPI rank of the current process.

```
int starp_sdk_t::num_processes ( ) const
```

Return the number of processes in this MPI group.

```
void starp_sdk_t::register_func ( char * fname,
                                  SdkFunc fptr
                                )
```

Register user function with the server.

Such a function must have a signature in a form of

```
void (*PPFuncPtr)(starp_sdk_t& sdk, PPArgList &inArgs, PPArgList &out-
Args)
```

Parameters:

pack

An opaque Star-P package object passed in by the calling server

fname

String identifying the name of the function to the clients, in particular the "ppinvoke" MATLAB command.

fptr

A function pointer to the function being registered. This function has to be called and it is always coded in the following form

```
extern "C" int starpInit (PPPackage *pack)
{
  starp_sdk_t::register_func(pack, "sdk_add_anyd", &ex_add_anyd);
  starp_sdk_t::register_func(pack, "other_func", &identity_meta);
  ...
  return 0;        // Return 0 to indicate no error
}
```

```
void starp_sdk_t::reset_error  (    )
```

Reset the error state to no-error.

```
int starp_sdk_t::root_rank  (    )  const
```

Return the rank of the root process.

```
PPServer& starp_sdk_t::server ( ) [inline]
```

Definition at line 1051 of file **`StarpSDK.h`**.

```
void starp_sdk_t::set_error  (  std::string const &  msg )
```

Indicate crucial error.

All return argumnts previously set in this function are cleared; adding any further return arguments within the same function results in a runtime error exception.

Parameters:

s

Error string

The documentation for this class was generated from the following file:

- **StarpSDK.h**

starp_value_t Class Reference

Holds an input or output argument of a user function.

```
#include <StarpSDK.h>
```

Public Member Functions	
	`starp_value_t ()` Don't use the default constructor, it's just here so starp_value_t objects can be stored in STL containers.
	`starp_value_t` `(std::string const &val)` Constructs a string value.
	`starp_value_t` `(char const *str)`

	Constructs a string value from a null-terminated character array.
	starp_value_t (matrix_ptr_t val) Construct a starp_value_t which wraps a distributed matrix.
bool	**is_distributed** () const Return true if this wraps a matrix_t.
	is_scalar () const
bool	Return true if this is a scalar or array of length 1.
	is_string () const
bool	Return true if this is a string value.
	is_array () const
bool	Return true if this is a local array value.
	is_real () const
bool	Return true if this is a local scalar or array whose elements are not complex.
	element_type_is () const
template<class T> bool	Test if the the element type of a contained value is of a specific type.
	number_of_elements () const
size_t	Return the number of elements in this value if it's not a distributed value.
matrix_ptr_t	**distributed_value** () const Get distributed value.
std::string	**string_value** () const Get string value.
template<class T> T	**scalar_value** () const Get scalar value.
template<class T> T	const * **array_value** () const Get array value.

Scalar Constructors

Constructors for starp_value_t objects holding a scalar value.

	`starp_value_t (starp_double_t val)`
	`starp_value_t (starp_dcomplex_t val)`

Array Constructors

Constructs a value which wraps a local array. If isowner is true, then this object will free the memory of the array using delete[] when the last reference to this starp_value_t goes out of scope.

Parameters:

val

The array to wrap

nelems

The number of elements in array

isowner

If true, then this object will free the memory pointed by val when it goes out of scope.

	starp_value_t (**starp_double_t** *val, size_t nelems, bool isowner=true)
	starp_value_t (**starp_dcomplex_t** *val, size_t nelems, bool isowner=true)

Conversion Operators

Conversion operators which offer another way to get the same functionality as the *_value() methods.

	operator starp_double_t () const
	operator starp_dcomplex_t () const
	operator idx_t () const
	operator std::string () const
	operator starp_double_t const * () const
	operator starp_dcomplex_t const * () const

Friends

class	SdkPackage

Classes

struct	impl_t

Detailed Description

Holds an input or output argument of a user function.

starp_value_t objects have shallow copy reference-counted semantics. They can be passed around by value, but copies of a starp_value_t object refer to the same internal data.

Definition at line 748 of file **StarpSDK.h.**

Constructor & Destructor Documentation

```
starp_value_t::starp_value_t   (     )   [inline]
```

Don't use the default constructor, it's just here so starp_value_t objects can be stored in STL containers.

Definition at line 753 of file StarpSDK.h.

```
starp_value_t::starp_value_t   ( starp_double_t val )
```

```
starp_value_t::starp_value_t   (  starp_dcomplex_t  val   )
```

```
starp_value_t::starp_value_t   (  std::string const &  val   )
```

Constructs a string value.

```
starp_value_t::starp_value_t   (  char const *  str   )
```

Constructs a string value from a null-terminated character array.

```
starp_value_t::starp_value_t   (  matrix_ptr_t  val   )
```

Construct a starp_value_t which wraps a distributed matrix.

```
starp_value_t::starp_value_t  (  starp_double_t *  val,
                                 size_t            nelems,
                                 bool              isowner = true
                              )
```

```
starp_value_t::starp_value_t  (  starp_dcomplex_t *  val,
                                 size_t              nelems,
                                 bool                isowner = true
                              )
```

Member Function Documentation

```
starp_dcomplex_t const * starp_value_t::array_value< starp_dcomplex_t >
(     )  const [inline]
```

Get array value.

Use like this:

```
starp_double_t *dp = val.array_value<starp_double_t>();
```

Returns:

A pointer to the data contained

Exceptions:

invalid_argument

If is_array() is false

Definition at line 928 of file StarpSDK.h.

```
matrix_ptr_t starp_value_t::distributed_value  (     )  const
```

Get distributed value.

Returns:

A pointer to a matrix_t

Exceptions:

invalid_argument

If is_distributed() is false

```
template<class T>
bool starp_value_t::element_type_is  (    )  const [inline]
```

Test if the the element type of a contained value is of a specific type.

This method always returns false if this is a distributed value; in that case, call distributed_value() and query the matrix_t to find its element type.

Use like this:

```
if (val.element_type_is<double>()) { do_something(); }
```

Definition at line 813 of file StarpSDK.h.

```
bool starp_value_t::is_array  (    )  const
```

Return true if this is a local array value.

```
bool starp_value_t::is_distributed  (    )  const
```

Return true if this wraps a matrix_t.

```
bool starp_value_t::is_real  (    )  const
```

Return true if this is a local scalar or array whose elements are not complex.

```
bool starp_value_t::is_scalar  (    )  const
```

Return true if this is a scalar or array of length 1.

```
bool starp_value_t::is_string  (    )  const
```

Return true if this is a string value.

```
size_t starp_value_t::number_of_elements  (    )  const
```

Return the number of elements in this value if it's not a distributed value.

For a distributed value this method returns 1.

```
starp_value_t::operator idx_t  (    )  const
```

```
starp_value_t::operator starp_dcomplex_t  (    )  const
```

```
starp_value_t::operator starp_dcomplex_t const *  (    )  const
```

```
starp_value_t::operator starp_double_t  (    )  const
```

```
starp_value_t::operator starp_double_t const *  (    )  const
```

```
starp_value_t::operator std::string  (    )  const
```

```
idx_t starp_value_t::scalar_value< idx_t >  (    )  const [inline]
```

Get scalar value.

Use like this:

```
starp_double_t d = val.scalar_value<starp_double_t>();
```

Returns:

Return the scalar value of this

Exceptions:

invalid_argument

  If is_scalar() is false

Definition at line 904 of file StarpSDK.h.

```
std::string starp_value_t::string_value  (    )  const
```

Get string value.

Returns:

The string value of this object

Exceptions:

invalid_argument

  If is_string() is false

Friends And Related Function Documentation

```
friend class SdkPackage [friend]
```

Definition at line 873 of file **StarpSDK.h**.

The documentation for this class was generated from the following file:

- StarpSDK.h

matrix_t Class Reference

A user-level interface for accessing and manipulating distributed matrices.

```
#include <StarpSDK.h>
```

Public Member Functions	
	~**matrix_t**()
const **layout_desc_t** &	**layout_desc** () const
	Get layout descriptor for this matrix.
const **type_desc_t** &	**type_desc** () const
	Get type descriptor for this matrix.
const **storage_desc_t** &	**storage_desc** () const
	Get storage descriptor for this matrix.
template<class T>	
T *	**local_dense_data** () const throw (std::runtime_error)
	Get dense data stored in the matrix.
template<class T>	
void	**global_dense_data** (T *buf) const throw (std::runtime_error)
PPcsr *	**local_sparse_data** () const throw (std::runtime_error)
	Get sparse data stored in the matrix.
idx_t	**local_data_size** () const
	Get the total number of elements stored in this matrix.

template<class T>	
bool	**data_type_is** () const
	Check to see if the type of data stored in the matrix can be read out using the type T specified in a template parameter.
template<class T>	
void	**convert_data_to_type** () throw (std::runtime_error)
	Modify data in the matrix by changing it to the datatype specified by the template parameter T.
void	**redistribute** (uint_t new_dist_dimension) throw (std::invalid_argument)
	Modify the distribution of data in the matrix by specifying the new dimension to be used for distribution.
PPMatrixID	**matrix_id**() const
Friends	
class	starp_sdk_t
class	matrix_factory_t
class	starp_value_t
std::ostream &	**operator<<** (std::ostream &s, const **matrix_t** &m)
	output helper

Detailed Description

A user-level interface for accessing and manipulating distributed matrices.

Definition at line 459 of file **StarpSDK.h**.

Constructor and Destructor Documentation

```
matrix_t::~matrix_t()
```

Member Function Documentation

```
template<class T>
void matrix_t::convert_data_to_type (  ) throw (std::runtime_error)
```

Modify data in the matrix by changing it to the datatype specified by the template parameter T.

The supported values for T are `starp_double_t` and `starp_dcomplex_t`.

Exceptions:

std::runtime_error is thrown if an unsupported type T is specified

```
template<class T>
bool matrix_t::data_type_is (   ) const
```

Check to see if the type of data stored in the matrix can be read out using the type T specified in a template parameter.

The supported values of T are `starp_double_t` and `starp_dcomplex_t`.

Returns:

- **true** if **type_desc()**.is_valid_conversion<T>() is true;
- **false** otherwise.

```
template<class T>
void matrix_t::global_dense_data  (  T *  buf   )  const throw
(std::runtime_error)
```

```
const layout_desc_t& matrix_t::layout_desc (   ) const
```

Get layout descriptor for this matrix.

Returns:

Descriptor for layout of this matrix

```
idx_t matrix_t::local_data_size (   ) const
```

Get the total number of elements stored in this matrix.

Returns:

Number of elements stored in the matrix

```
template<class T>
T* matrix_t::local_dense_data (   ) const throw (std::runtime_error)
```

Get dense data stored in the matrix.

Returns:

An array of type T (either `starp_double_t` or `starp_dcomplex_t`)

Exceptions:

std::runtime_error if the matrix does not have dense data (not PP_STORAGE_DENSE) or the template type specified does not match the type of data stored in the matrix.

```
PPcsr* matrix_t::local_sparse_data (   ) const throw
(std::runtime_error)
```

Get sparse data stored in the matrix.

Returns:

A pointer to a **PPcsr** structure that holds the elements of a sparse matrix

Exceptions:

std::runtime_error if the matrix does not have sparse data (not PP_STORAGE_CSR) or the template type specified does not match the type of data stored in the matrix.

See also:

- storage_desc()
- data_type_is()

```
PPMatrixID matrix_t::matrix_id  (     )  const [inline]
```

Definition at line 550 of file **`StarpSDK.h`**.

```
void matrix_t::redistribute ( uint_t new_dist_dimension  ) throw
(invalid_argument)
```

Modify the distribution of data in the matrix by specifying the new dimension to be used for distribution.

This method currently supports 2D matrices

Parameters:

new_dist_dimension

The dimension along which to redistribute the data

Exceptions:

std::invalid_argument is thrown if the matrix has more than 2D.

```
const storage_desc_t& matrix_t::storage_desc (   ) const
```

Get storage descriptor for this matrix.

Returns:

Storage descriptor for this matrix.

```
const type_desc_t& matrix_t::type_desc (   ) const
```

Get type descriptor for this matrix.

Returns:

Descriptor for the datatype of this matrix

Friends and Related Function Documentation

```
friend class matrix_factory_t [friend]
```

Definition at line 558 of file **StarpSDK.h**.

```
std::ostream& operator<<  (  std::ostream &   s,
                              const matrix_t &   m
                          )  [friend]
```

Output helper.

```
friend class starp_sdk_t [friend]
```

Definition at line 557 of file **StarpSDK.h**.

```
friend class starp_value_t [friend]
```

Definition at line 559 of file **StarpSDK.h**.

The documentation for this class was generated from the following file:

- StarpSDK.h

matrix_t Member List

This is the complete list of members for **matrix_t**, including all inherited members.

convert_data_to_type()	**matrix_t**	
data_type_is() const	**matrix_t**	
layout_desc() const	**matrix_t**	
local_data_size() const	**matrix_t**	
local_dense_data() const	**matrix_t**	
local_sparse_data() const	**matrix_t**	
operator<<(ostream &s, const matrix_t &m)	**matrix_t**	[friend]
redistribute(uint_t new_dist_dimension)	**matrix_t**	
storage_desc() const	**matrix_t**	
type_desc() const	**matrix_t**	

layout_desc_t Class Reference

The layout descriptor describes how a matrix is distributed out among the processes in a MPI communicator.

```
#include <StarpSDK.h>
```

Public Member Functions	
	layout_desc_t (const starp_size_vector_t &global_size_array, uint_t distributed_dimension, MPI_Comm comm=MPI_COMM_WORLD) throw (std::out_of_range) Create a new layout descriptor given the dimension of the matrix and the direction of its distribution.
const starp_size_vector_t&	**global_size_array** () const
uint_t	**distributed_dimension** () const

idx_t	**dimension_count** () const
bool	**has_data** (uint_t process_rank) const throw (std::out_of_range) Determine if a given process has a hyperslab of data of non-zero thickness.
uint_t	procs_with_data () const
	Returns the number of processes in the communicator that contain non-empty hyperslabs of the matrix.
starp_size_t	**local_size_array** (uint_t process_rank) const throw (std::out_of_range)
starp_size_t	local_size() const
starp_size_t	global_size() const
const starp_size_vector _t &	local_size_array (uint_t process_rank) const throw (std::out_of_range)
	Get dimensions of a hyperslab on a particular processor.
idx_t	**start_idx** (uint_t process_rank) const throw (out_of_range)
	For a given process, get the starting index of a hyperslab along the distributed dimension.
idx_t	**end_idx** (uint_t process_rank) const throw (out_of_range) For a given process, get the ending index of a hyperslab along the distributed dimension.
bool	**has_data** () const
	Determine if this process has a hyperslab of data of non-zero thickness.
const starp_size_vec tor_t	**local_size_array** () const
	Get the dimensions of a hyperslab on this processor.
idx_t	**start_idx** () const
	Get the starting index of a hyperslab on current processor along the distributed dimension.
idx_t	**end_idx** () const
	Get the ending index of a hyperslab on current processor along the distributed dimension.
bool	**dimensions_equal** (const **layout_desc_t** &layout) const
	Check to see if sizes along all dimensions are equal for two descriptors.

bool	**distributions_equal** (const **layout_desc_t** &layout) const
	Check to see if two descriptors are distributed alng the same dimension.
Protected Attributes	
uint_t	m_comm_size
uint_t	m_comm_rank
starp_size_vector_t	m_global_size_array
starp_size_t	M_global_size
uint_t	m_dimension_count
uint_t	m_distributed_dimension
uint_t	m_procs_with_data
std:vec-tor<starp_size_vector_t>	m_local_size_arrays
std:vec-tor<starp_size_t>	m_local_sizes
std:vec-tor<starp_ssize_t>	m_start_idxs
std:vec-tor<starp_ssize_t>	m_end_idxs
Friends	
class	**matrix_t**
bool	**operator==** (const **layout_desc_t** &ld1, const **layout_desc_t** &ld2)
	Logical equality of two layout descriptors.
std::ostream &	**operator<<** (ostream &s, const **layout_desc_t** &ld)
	output helper

Detailed Description

The layout descriptor describes how a matrix is distributed out among the processes in a MPI communicator.

In Star-P, a N-dimensional matrix is distributed along a given dimension such that each process gets a contiguous, non-overlapping (N-1)-dimensional hyperslab along that dimension. Cyclic and block-cyclic distributions are currently unsupported.

Currently, the layout descriptor distributes a matrix according to the ScaLAPACK convention for two-dimensional block distributions (cf. http://www.netlib.org/scalapack/slug/node82.html).

Definition at line 219 of file StarpSDK.h.

Constructor & Destructor Documentation

```
layout_desc_t::layout_desc_t ( const starp_vector_size_t &
global_size_array,
                               uint_t           distributed_dimension,
                               MPI_Comm         comm = MPI_COMM_WORLD
                             ) throw (std::out_of_range)
```

Create a new layout descriptor given the dimension of the matrix and the direction of its distribution.

Parameters:

global_size_array

an ntuple holding the size of the matrix in each dimension

distributed_dimension

the number of the dimension along which the data is distributed across processors

comm

the id of the communicator used to distribute the matrix data, MPI_COMM_WORLD by default

Exceptions:

std::out_of_range

is thrown if distributed dimension is either negative or is greater than the number of dimensions specified in the global_size_array.

Member Function Documentation

```
idx_t layout_desc_t::dimension_count (  ) const [inline]
```

Returns:

Number of dimensions of a matrix associated with this layout descriptor.

Definition at line 256 of file StarpSDK.h.

Check to see if sizes along all dimensions are equal for two descriptors.

```
bool layout_desc_t::dimensions_equal ( const layout_desc_t & layout
 ) const
```

Parameters:

layout

The descriptor to compare against

Returns:

- **true** if the number and the size of all dimensions are equal in both arrays,
- **false** otherwise.

```
uint_t layout_desc_t::distributed_dimension (  ) const [inline]
```

A zero-based index of a distributed dimension of a matrix associated with this layout descriptor.

Definition at line 248 of file StarpSDK.h.

References m_distributed_dimension.

```
bool layout_desc_t::distributions_equal ( const layout_desc_t & layout
 ) const
```

Check to see if two descriptors are distributed along the same dimension.

Parameters:

layout

The descriptor to compare against

Returns:

- **true** if the distributed dimensions are equal in both arrays,
- **false** otherwise.

```
idx_t layout_desc_t::end_idx (   ) const
```

Get the ending index of a hyperslab on current processor along the distributed dimension.

Returns:

A zero-based index into the distributed dimension indicating the position of the last layer of the slab along that dimension.

If the processor has no data, return -2.

Definition at line 385 of file StarpSDK.h.

References m_comm_rank, and m_end_idxs.

```
idx_t layout_desc_t::end_idx ( uint_t process_rank  ) const throw
(std::out_of_range) [inline]
```

For a given process, get the ending index of a hyperslab along the distributed dimension.

An invariant that applies to start and end indices is that the thickness of hyperslab on any processor is equal to (end_idx - start_idx + 1).

Parameters:

process_rank

Number of a processor

Returns:

A zero-based index into the distributed dimension indicating the position of the last layer of the slab along that dimension. If the processor has no data, return -2.

Exceptions:

out_of_range

Is thrown if the process_rank is either negative or is greater than the number of processes in the communicator.

See also:

has_data()

Definition at line 346 of file StarpSDK.h.

References m_end_idxs.

```
starp_size_t layout_desc_t::global_size_array (  ) const [inline]
```

Definition at line 300 of file StarpSDK.h.

References m_global_size.

```
const starp_size_t layout_desc_t::global_size_array (  ) const [inline]
```

Returns:

size_vector describing the dimensions of a matrix associated with this layout descriptor

Definition at line 240 of file StarpSDK.h.

References m_global_size_array.

```
bool layout_desc_t::has_data (  ) const
```

Determine if this process has a hyperslab of data of non-zero thickness.

Returns:

true if this process has a hyperslab of non-zero thickness

Exceptions:

std::*out_of_range*

Is thrown if distributed dimension is either negative or is greater than the number of dimensions in this descriptor.

Definition at line 354 of file StarpSDK.h.

```
bool layout_desc_t::has_data ( uint_t process_rank  ) const throw
(std::out_of_range) [inline]
```

Determine if a given process has a hyperslab of data of non-zero thickness.

Typically, all processes in a communicator have data, however in some degenerate cases,some high-ranking processes may end up holding no data. For example, a 5x5 matrix distributed over 4 processors will have the following distribution: [2, 2, 1, 0].

Parameters:

process_rank

Number of a process

Returns:

true if that process has a non-zero thickness of the hyperslab

Exceptions:

std::out_of_range

Is thrown if distributed dimension is either negative or is greater than the number of dimensions in this descriptor.

Definition at line 274 of file StarpSDK.h.

References m_local_sizes.

```
starp_size_t layout_desc_t::local_size_array (  ) const [inline]
```

Definition at line 295 of file StarpSDK.h.

References m_comm_rank, and m_local_sizes.

```
starp_size_t layout_desc_t::local_size_array ( uint_t process_rank
) const throw (std::out_of_range) [inline]
```

Definition at line 290 of file StarpSDK.h.

References m_local_sizes.

```
const starp_size_vector_t& layout_desc_t::local_size_array() const
[inline]
```

Get dimensions of a hyperslab on this processor.

Returns

an size_vector that represents the dimensions of data available on this processor.

Definition at line 364 of file StarpSDK.h.

References m_comm_rank, and m_local_size_arrays.

```
const starp_size_vector_t& layout_desc_t::local_size_array() const
[inline]
```

Get dimensions of a hyperslab on a particular processor.

Parameters:

process_rank

Number of a process.

Returns:

An size_vector that represents the dimensions of data available on processor indicated by process_rank.

Exceptions:

std::out_of_range

Is thrown if distributed dimension is either negative or is greater than the number of dimensions in this descriptor.

Definition at line 313 of file StarpSDK.h.

References m_local_size_arrays.

```
uint_t layout_desc_t::procs_with_data ( ) const [inline]
```

Returns the number of processes in the communicator that contain non-empty hyperslabs of the matrix.

Definition at line 285 of file StarpSDK.h.

References m_procs_with_data.

```
idx_t layout_desc_t::start_idx (  ) const [inline]
```

Get the starting index of a hyperslab on current processor along the distributed dimension.

Returns:

A zero-based index into the distributed dimension indicating the position of the first layer of the slab along that dimension. If the processor has no data, return -1.

Definition at line 376 of file StarpSDK.h.

References m_comm_rank, and m_start_idxs.

```
idx_t layout_desc_t::start_idx ( uint_t process_rank  ) const throw
(std::out_of_range) [inline]
```

For a given process, get the starting index of a hyperslab along the distributed dimension.

An invariant that applies to start and end indices is that the thickness of hyperslab on any processor is equal to (end_idx - start_idx + 1).

Parameters:

process_rank

Number of a processor

Returns:

A zero-based index into the distributed dimension indicating the position of the first layer of the slab along that dimension. If the processor has no data, return -1.

Exceptions:

std::out_of_range

Is thrown if the process_rank is either negative or is greater than the number of processes in the communicator.

See also:

`has_data()`

Definition at line 331 of file StarpSDK.h.

References m_start_idxs.

Friends And Related Function Documentation

```
friend class matrix_t [friend]
```

Definition at line 435 of file StarpSDK.h.

```
std::ostream& operator<< ( std::ostream & s,
                           const layout_desc_t & ld
                          ) [friend]
```

output helper

```
bool operator== ( const layout_desc_t & ld1,
                  const layout_desc_t & ld2
                ) [friend]
```

Logical equality of two layout descriptors.

Parameters:

ld1

First descriptor to compare against

ld2

The second descriptor

Returns:

- **true** if both distributed dimensions and distributions are equal in both arrays,
- **false** otherwise.

See also:

`dimensions_equal`

`distributions_equal`

Member Data Documentation

```
uint_t layout_desc_t::m_comm_rank [protected]
```

Definition at line 423 of file StarpSDK.h.

Referenced by end_idx(), has_data(), local_size(), local_size_array(), and start_idx().

```
uint_t layout_desc_t::m_comm_size [protected]
```

Definition at line 422 of file StarpSDK.h.

```
uint_t layout_desc_t::m_dimension_count [protected]
```

Definition at line 426 of file StarpSDK.h.

Referenced by dimension_count().

```
uint_t layout_desc_t::m_distributed_dimension [protected]
```

Definition at line 427 of file StarpSDK.h.

Referenced by `distributed_dimension()`.

type_desc_t Class Reference

Identifies the type of data contained in a distributed matrix; provides access to the MPI properties of the elements of the matrix.

List of all members.

Public Member Functions	
MPI_Datatype	**`mpi_type`** `() const`
	Get the corresponding MPI datatype.
size_t	**`element_size`** `() const`
	Get the size of the corresponding MPI datatype.
template<class T> bool	**`is_valid_conversion`** `() const`
	Indicate whether matrix data tagged by this datatype can be read as a specified type T.
Static Public Attributes	
type_desc_t *const	**`REAL`**
	An immutable static value representing the properties of an starp_double_t.
type_desc_t *const	`COMPLEX`
	An immutable static value representing the properties of a starp_dcomplex_t.

Protected Member Functions	
	type_desc_t (MPI_Datatype type_rep)
void	**init** ()
bool	**is_a (type_desc_t** *const type_p) const
Protected Attributes	
MPI_Datatype	m_type_rep
bool	m_init
size_t	m_type_size
Friends	
class	StarpSDK
class	matrix_t
bool	**operator==** (const **type_desc_t** &td1, const **type_desc_t** &td2)
ostream &	**operator<<** (ostream &s, const **type_desc_t** &td)
	helper function to dump the values of a type descriptor

Detailed Description

Identifies the type of data contained in a distributed matrix; provides access to the MPI properties of the elements of the matrix -- their size and type.

Definition at line 117 of file StarpSDK.h.

Constructor & Destructor Documentation

```
type_desc_t::type_desc_t  (  MPI_Datatype  type_rep  )  [explicit, protected]
```

Member Function Documentation

```
size_t type_desc_t::element_size (  ) const
```

Get the size of the corresponding MPI datatype.

Returns:

The size (in bytes) for the MPI datatype that can be used to represent elements of a matrix associated with this type descriptor

```
void type_desc_t::init  (    )  [protected]
```

```
bool type_desc_t::is_a  (  type_desc_t *const   type_p   )  const [pro-
tected]
```

```
bool type_desc_t::is_valid_conversion< starp_dcomplex_t >  (    )
const [inline]
```

Indicate whether matrix data tagged by this datatype can be read as a specified type T.

The supported values of T are starp_double_t and starp_dcomplex_t. This function is guaranteed to evaluate to **true** for

```
type_desc_t::REAL->is_valid_conversion<starp_double_t>();
type_desc_t::COMPLEX->is_valid_conversion<starp_dcomplex_t>();
```

Returns:

- **true** if templated datatype T is a valid conversion target for a matrix associated with this type descriptor,
- **false** otherwise.

```
MPI_Datatype type_desc_t::mpi_type (   ) const
```

Get the corresponding MPI datatype.

Returns:

A valid MPI datatype that can be used to represent elements of a matrix associated with this type descriptor

Friends And Related Function Documentation

```
friend class matrix_t [friend]
```

Definition at line 183 of file **StarpSDK.h**.

```
std::ostream& operator<<  (  std::ostream &  s,
                             const type_desc_t &  td
                           )  [friend]
```

Helper function to dump the values of a type descriptor

```
bool operator= = ( const type_desc_t & td1,
                   const type_desc_t & td2
                 ) [friend]
```

Returns:

true if both MPI datatypes and sizes match for the two given type descriptors

```
friend class starp_sdk_t [friend]
```

Definition at line 182 of file StarpSDK.h.

Member Data Documentation

```
type_desc_t* const type_desc_t::COMPLEX [static]
```

An immutable static value representing the properties of a starp_dcomplex_t.

Definition at line 127 of file StarpSDK.h.

```
bool type_desc_t::m_init [protected]
```

Definition at line 174 of file StarpSDK.h

```
size_t type_desc_t::m_type_size [protected]
```

Definition at line 175 of file StarpSDK.h.

```
type_desc_t* const type_desc_t::REAL [static]
```

An immutable static value representing the properties of an starp_double_t.

Definition at line 123 of file StarpSDK.h.

The documentation for this class was generated from the following file:

- **StarpSDK.h**

type_desc_t Member List

This is the complete list of members for **type_desc_t**, including all inherited members.

COMPLEX	**type_desc_t**	[static]

element_size() const	**type_desc_t**
is_valid_conversion() const	**type_desc_t**
mpi_type() const	**type_desc_t**
operator<<(ostream &s, const type_desc_t &td)	**type_desc_t** [friend]
operator==(const type_desc_t &td1, const type_desc_t &td2)	**type_desc_t** [friend]
REAL	**type_desc_t** [static]

matrix_factory_t Class Reference

A factory class for creating new matrices.

#include <StarpSDK.h>

Public Member Functions	
template<class T> matrix_ptr_t	**create_matrix** (const **layout_desc_t** &layout, const storage_desc_t &storage, initializer_t *init_closure, const idx_t nnz=-1) const throw (std::invalid_argument)
	Create a new matrix with data elements of template type T (either `starp_double_t` or `starp_dcomplex_t`) with elemens initilized using a custom class.
void	**remove_matrix (matrix_ptr_t** matrix) const throw (std::invalid_argument) Deallocate a distributed matrix.
matrix_ptr_t	**clone** (**matrix_ptr_t** matrix) const Deep copy a matrix including all the data.
matrix_ptr_t	**clone (matrix_ptr_t** matrix, **uint_t** new_dist_dimension) const Deep copy of matrix, including all data, with a different distribution.
template<class T> matrix_ptr_t	**zeros** (const **layout_desc_t** &layout) const Create a new matrix with data elements of template type T (either `starp_double_t` or `starp_dcomplex_t`) populated with zeros.

template<class T> matrix_ptr_t	**ones** (const **layout_desc_t** &layout) const Create a new matrix with data elements of template type T (either starp_double_t or starp_dcomplex_t) populated with real ones.
template<class T> matrix_ptr_t	**rand** (const **layout_desc_t** &layout) const Create a new matrix with data elements of template type T (either starp_double_t or starp_dcomplex_t) populated with random numbers.
matrix_ptr_t	**create (PPMatrixID** id) const
Friends	
class	**starp_sdk_t**

Detailed Description

A factory class for creating new matrices.

All matrices created by this class are returned as matrix_ptr_t objects that do reference counting and garbage collection. A user must obtain a factory from the SDK object.

Definition at line 631 of file **StarpSDK.h**.

Member Function Documentation

```
matrix_ptr_t matrix_factory_t::clone  (  matrix_ptr_t  matrix,
                                          uint_t  new_dist_dimension
                                       )  const
```

Deep copy of matrix, including all data, with a different distribution.

Parameters:

matrix

Matrix to be copied

new_dist_dimension

The dimension along which the copy will be distributed

Returns:

Copy of input matrix.

```
matrix_ptr_t matrix_factory_t::clone ( matrix_ptr_t matrix ) const
```

Deep copy a matrix including all the data.

Parameters:

matrix

A matrix to be copied

Returns:

A smart pointer to a new matrix initialized with dimension, distribution and a copy of the data of the input matrix.

```
matrix_ptr_t matrix_factory_t::create(PPMatrixID id) const
```

```
template<class T>
matrix_ptr_t matrix_factory_t::create_matrix ( const layout_desc_t &
layout,
                                                 const storage_desc_t &
storage,
                                                 initializer_t *
init_closure,
                                                 const idx_t nnz = -1
                                               )  const throw
(std::invalid_argument)
```

Create a new matrix with data elements of template type T (either `starp_double_t` or `starp_dcomplex_t`) with elemens initilized using a custom class.

Parameters:

layout

Layout descriptor indicating the dimensions and distribution of a matrix

storage

Storage descriptor indicating whether the matrix is dense or sparse

init_closure

A subclass of initializer_t that has methods to populate the matrix data

nnz

Number of non-zero elements, optional, applies to sparse matrices only

Returns:

A smart pointer to a new matrix initialized according to parameters passed in

Exceptions:

std::invalid_argument is thrown if number of non-zero elements is specified for a dense matrix; a number of non-zero elements is NOT specified for a sparse matrix; or if an ND matrix is requested with sparse storage.

```
template<class T>
matrix_ptr_t matrix_factory_t::ones ( const layout_desc_t & layout )
const
```

Create a new matrix with data elements of template type T (either `starp_double_t` or `starp_dcomplex_t`) populated with real ones.

Parameters:

layout

Layout descriptor indicating the dimensions and distribution of a matrix

Returns:

A smart pointer to a new matrix initialized with ones

```
template<class T>
matrix_ptr_t matrix_factory_t::rand  ( const layout_desc_t & layout )
const
```

Create a new matrix with data elements of template type T (either `starp_double_t` or `starp_dcomplex_t`) populated with random numbers.

If a matrix of `starp_dcomplex_t` is requested, both the real and complex parts of each element are populated with random numbers.

Parameters:

layout

Layout descriptor indicating the dimensions and distribution of a matrix

Returns:

A smart pointer to a new matrix initialized with random values.

```
void matrix_factory_t::remove_matrix( matrix_ptr_t  matrix ) const
throw (std::invalid_argument)
```

Deallocate a distributed matrix.

After this method returns, the distributed object referred to by matrix will no longer be valid.

Parameters:

The matrix to be removed.

Exceptions:

If matrix refers to a matrix which has already been removed.

```
template<class T>
matrix_ptr_t matrix_factory_t::zeros ( const layout_desc_t & layout )
const
```

Create a new matrix with data elements of template type T (either `starp_double_t` or `starp_dcomplex_t`) populated with zeros.

Parameters:

layout

Layout descriptor indicating the dimensions and distribution of a matrix

Returns:

A smart pointer to a new matrix initialized with zeros

Friends and Related Function Documentation

```
friend class starp_sdk_t [friend]
```

Definition at line 729 of file **StarpSDK.h**.

The documentation for this class was generated from the following file:

- StarpSDK.h

matrix_factory_t Member List

This is the complete list of members for **matrix_factory_t**, including all inherited members.

`clone(matrix_ptr matrix) const`	matrix_factory_t	
`create_matrix(const layout_desc_t &layout, const storage_desc_t &storage, initializer_t *init_closure, const idx_t nnz=-1) const`	matrix_factory_t	
`ones(const layout_desc_t &layout) const`	matrix_factory_t	
`rand(const layout_desc_t &layout) const`	matrix_factory_t	
`zeros(const layout_desc_t &layout) const`	matrix_factory_t	

initializer_t Struct Reference

A virtual base class users override to allow matrix factory to initialize matrices in an arbitrary way.

```
#include <StarpSDK.h>
```

Public Member Functions	
`virtual`	`~initializer_t()`
virtual void	`init (starp_double_t *const data, const idx_t length)=0` An intializer function for starp_double_t data.
virtual void	`init (starp_dcomplex_t *const data, const idx_t length)=0` An intializer function for starp_dcomplex_t data.

Detailed Description

A virtual base class users override to allow matrix factory to initialize matrices in an arbitrary way.

User-specified subclasses of this class are passed as one of the parameters to a matrix factory. The factory allocates storage for the matrix data and then calls one of the overriden member functions to initialize its contents.

Constructor & Destructor Documentation

```
virtual initializer_t::~initializer_t [inline, virtual]
```

Definition at line 600 of file StarpSDK.h.

Member Function Documentation

```
virtual void initializer_t::init ( starp_dcomplex_t *const data,
                                   const idx_t     length
                                 ) [pure virtual]
```

An intializer function for `starp_dcomplex_t` data.

Parameters:

data

pointer to complex data area that needs to be initialized

length

number of data elements that need to be initialized

```
virtual void initializer_t::init ( starp_double_t *const data,
                                   const idx_t     length
                                 ) [pure virtual]
```

An intializer function for ELTYPE data.

Parameters:

data

pointer to the real data area that needs to be initialized

length

number of data elements that need to be initialized

```
     // sample initializer that zeros out the data
     virtual void init (starp_double_t* const data, const idx_t length)
{
      for (int i=0; i<length; i++) {
        data[i] = 0;
      }
```

The documentation for this struct was generated from the following file:

- **StarpSDK.h**

PPcsr Struct Reference

```
#include <Starp.h>
```

Public Attributes	
idx_t	nnz_loc
idx_t	m_loc
idx_t	fst_row
void *	nzval
idx_t *	rowptr
idx_t *	colind

Detailed Description

Definition at line 68 of file **Starp.h**.

Member Data Documentation

```
idx_t* PPcsr::colind
```

Definition at line 75 of file **Starp.h**.

```
idx_t* PPcsr::fst_row
```

Definition at line 71 of file **Starp.h**.

```
idx_t* PPcsr::m_loc
```

Definition at line 70 of file **Starp.h**.

```
idx_t* PPcsr::nnz_loc
```

Definition at line 69 of file **Starp.h.**

```
idx_t* PPcsr::nzval
```

Definition at line 72 of file **Starp.h.**

```
idx_t* PPcsr::rowptr
```

Definition at line 73 of file **Starp.h.**

The documentation for this struct was generated from the following file:

- Starp.h

Star-P SDK for Task Parallel Operations

ppeval C Engine Class List

Here are the classes, structs unions and interfaces with brief descriptions:

`pearg_t` Class Reference	A class to hold input and output arguments to `ppeval` C++ functions
`ppevalc_module_t` Class Reference	This class provides the interface for `ppevalc` modules to interact with the starpserver runtime
`ppevalc_module_t::private_state`	

ppeval C Engine File List

Here is a list of all files with brief descriptions:

- pearg.h
- ppevalc.h
- ppevalc_module.cc
- ppevalc_module.h

ppeval C Engine Class Documentation

pearg_t Class Reference

```
#include <pearg.h>
```

Detailed Description

Instances of this class have shallow copy, reference-counted semantics, so they can be passed around by value. The data they hold is only freed when the last copy goes out of scope. To make a deep copy, use the `clone()` method.

Public Types

```
enum element_type_t {CHAR, DOUBLE, DOUBLE_COMPLEX}
```

Public Member Functions

```
peart_t()
```

Creates a null `pearg_t`.

```
pearg_t clone() const
```

Create a deep copy of this `pearg_t`.

```
bool is_null() const
```

Returns true if this `pearg_t` is null, meaning it was constructed with the no-argument constructor and therefore has no value.

```
element_type_t element_type() const
```

Return the element type of this object.

```
starp_size_t element_size() const
```

Returns the size of one element in bytes of `element_type`.

```
starp_size_vector_t const & size_vector () const
```

Returns a vector containing the size of each dimension.

```
starp_size_t number_of_elements () const
```

Returns the total number of elements in this argument.

```
bool is_vector () const
```

Returns true if this arg is one-dimensional, or two-dimensional with one dimension size equal to 1.

```
bool is_scalar () const
```

Returns true if `number_of_elements()` == 1

```
std::string const & string_data() const
```

Returns a string value if this of type `STRING`.

```
void * raw_data()
void disown_data()
```

Make this `pearg` disown the data it holds.

Scalar Constructors

Create an object initialized from the input argument.

```
pearg_t(starp_double_t v)
pearg_t(starp_dcomplex_t v)
pearg_t(std::string const &v)
```

Uninitialized Array Constructors

Create an array object of the given dimensions and type.

Memory for the array is allocated and managed by this object. The memory is uninitialized. Use `data()` to get a pointer to the memory.

```
pearg_t(pearg_t::element_type_t element_type, starp_size_t length)

pearg_t(pearg_t::element_type_t element_type, starp_size_t num_rows,
        starp_size_t num_cols)

pearg_t(pearg_t::element_type_t element_type, starp_size_vector_t
        const &dim_sizes)
```

Initialized Array Constructors

Create an array object backed by the array provided in argument `v`.

If `is_owner` is true, then this object takes ownership of the memory pointed to by `v` and frees it using `delete[]`. If `is_owner` is false, then it is the caller's responsibility to free the memory.

```
pearg_t (starp_double_t *v, starp_size_t length, bool is_owner=true)

pearg_t (starp_double_t *v, starp_size_t num_rows, starp_size_t
         num_cols, bool is_owner=true)

pearg_t (starp_double_t *v, starp_size_vector_t const &dim_sizes, bool
         is_owner=true)

pearg_t (starp_dcomplex_t *v, starp_size_t length, bool is_owner=true)

pearg_t (starp_dcomplex_t *v, starp_size_t num_rows, starp_size_t
         num_cols, bool is_owner=true)

pearg_t (starp_dcomplex_t *v, starp_size_vector_t const &dim_sizes,
         bool is_owner=true)
```

Data Accessor

Return a pointer to the underlying data.

The template type parameter must match the element_type for this object. Example usage:

```
starp_double_t *p = arg.data<starp_double_t>()
```

```
template<class ElementType> ElementType * data ()
template<class ElementType> ElementType const * data () const
```

Classes

```
struct impl_t
```

Member Enumeration Documentation

```
enum pearg_t::element_type_t
```

Enumerator:

- CHAR
- DOUBLE
- DOUBLE_COMPLEX

Constructor & Destructor Documentation

```
pearg_t::pearg_t () [inline]
```

Creates a null `pearg_t`.

The `is_null()` method will return true if this constructor is used.

```
pearg_t::pearg_t (starp_double_t v)

pearg_t::pearg_t (starp_dcomplex_t v)

pearg_t::pearg_t (std::string const & v)

pearg_t::pearg_t (pearg_t::element_type_t element_type, starp_size_t
                  length)
pearg_t::pearg_t (pearg_t::element_type_t element_type, starp_size_t
                  num_rows, starp_size_t num_cols)

pearg_t::pearg_t (pearg_t::element_type_t element_type,
                  starp_size_vector_t const & dim_sizes)

pearg_t::pearg_t (starp_double_t * v, starp_size_t length, bool
                  is_owner = true)

pearg_t::pearg_t (starp_double_t * v, starp_size_t num_rows,
                  starp_size_t num_cols, bool is_owner = true)

pearg_t::pearg_t (starp_double_t * v, starp_size_vector_t const &
                  dim_sizes, bool is_owner = true)
```

```
pearg_t::pearg_t (starp_dcomplex_t * v, starp_size_t length, bool
                  is_owner = true)

pearg_t::pearg_t (starp_dcomplex_t * v, starp_size_t num_rows,
                  starp_size_t num_cols, bool is_owner = true)

pearg_t::pearg_t (starp_dcomplex_t * v, starp_size_vector_t const &
                  dim_sizes, bool is_owner = true)
```

Member Function Documentation

```
pearg_t pearg_t::clone () const
```

Create a deep copy of this `pearg_t`.

The copy constructor and assignment operator for this class make shallow copies, which internally refer to the same data. Use this method when you want to make a completely separate copy of a `pearg_t`.

```
bool pearg_t::is_null () const [inline]
```

Return true if this `pearg_t` is null, meaning it was constructed with the no-argument constructor, and therefore has no value.

Most other methods of this object will segfault if called on a null pearg.

```
element_type_t pearg_t::element_type () const [inline]
```

Return the element type of this object.

```
starp_size_t pearg_t::element_size () const
```

Return the size of one element in bytes of element_type.

```
starp_size_vector_t const& pearg_t::size_vector () const [inline]
```

Return vector containing the size of each dimension.

```
starp_size_t pearg_t::number_of_elements () const [inline]
```

Return the total number of elements in this argument.

For strings this will be the length of the string.

```
bool pearg_t::is_vector () const [inline]
```

Return true if this arg is one-dimensional, or two-dimensional with one dimension size equal to 1.

```
bool pearg_t::is_scalar () const [inline]
```

Return true if number_of_elements() == 1.

```
std::string const& pearg_t::string_data () const
```

Return string value if this is of type STRING.

Exceptions:

`ppevalc_exception_t` if `element_type` is not STRING

```
template<class ElementType> ElementType* pearg_t::data () [inline]
```

```
template<class ElementType> ElementType const* pearg_t::data () const
[inline]
```

```
void* pearg_t::raw_data ()
```

```
void pearg_t::disown_data () [inline]
```

Make this pearg disown the data it holds.

This means it won't free the memory for its argument data in its destructor. This is useful if you want to get a pointer to the data using data(), and hold on to the pointer after the pearg_t object has gone out of scope.

The documentation for this class was generated from the following file:

* pearg.h

ppevalc_module_t Class Reference

```
#include <ppevalc_module.h>
```

Detailed Description

This class provides the interface for ppevalc modules to interact with the starpserver runtime.

Public Member Functions

```
ppevalc_module_t (std::string const &name)
```

Constructs a new module called `name`.

```
virtual ~ppevalc_module_t ()

std::string const & get_name () const
```

Get the name of the module as passed to the constructor.

```
void add_function (std::string const &func_name,
                   ppevalc_user_function_t func)
```

Register a new function with the module.

```
ppevalc_user_function_t get_function (std::string const &func_name)
                                      const
```

Retrieve a function pointer by name.

```
std::ostream & log ()
```

Returns a reference to an ostream which should be used by user functions to log messages.

Static Public Member Functions

```
static smart_ptr< ppevalc_module_t > load_module (std::string const
&module_name, std::string const &file_name)
```

Load a module from a shared library file.

Classes

```
struct private_state
```

Constructor & Destructor Documentation

```
ppevalc_module_t::ppevalc_module_t (std::string const & name)
```

Constructs a new module with name name .

Parameters:

`name` The name of the new module.

```
ppevalc_module_t::~ppevalc_module_t () [virtual]
```

Member Function Documentation

```
std::string const& ppevalc_module_t::get_name () const [inline]
```

Get the name of the module as passed to the constructor.

```
void ppevalc_module_t::add_function (std::string const & func_name,
ppevalc_user_function_t func)
```

Register a new function with the module.

The function pointer will be accessible using the `get_function()` method.

Parameters:

`func_name` A string, which can later be used as a lookup key with the get_function method to retrieve the associated function pointer. If another function pointer was previously registered with the same name, it will be replaced.

`func` The function pointer to associate with `name`.

```
ppevalc_user_function_t ppevalc_module_t::get_function (std::string
const & func_name) const
```

Retrieve a function pointer by name.

Parameters:

`func_name` A function name previously registered via `add_function()`.

Returns:

The function pointer associated with `func_name` , or `NULL` if no function was registered with that name.

```
std::ostream& ppevalc_module_t::log () [inline]
```

Returns a reference to an ostream which should be used by user functions to log messages.

```
ppevalc_module_ptr_t ppevalc_module_t::load_module (std::string const &
module_name, std::string const & file_name) [static]
```

Load a module from a shared library file.

The shared library must contain an exported function named ppevalc_module_init , with a signature matching ppevalc_module_init_function_t . A new ppevalc_module_t object will be created and passed to the module init function, and the init function can register module functions with the module object. The new module object is then returned.

Parameters:

`module_name` The name of the new module. This is used to initialize the name property of the `ppevalc_module` object.

`file_name` The path to a shared library file containing the module init function.

Returns:

A pointer to the newly create module on success.

Exceptions:

`ppevalc_exception_t` on error loading the module.

The documentation for this class was generated from the following files:

- ppevalc_module.h
- ppevalc_module.cc

ppevalc_module_t::private_state Struct Reference

Public Attributes

```
ACE_SHLIB_HANDLE shlib_handle
```

Member Data Documentation

```
ACE_SHLIB_HANDLE ppevalc_module_t::private_state::shlib_handle
```

The documentation for this struct was generated from the following file:

- ppevalc_module.cc

Chapter 5

List of All Class Members

Here is a list of all class members with the classes they belong to:

- a -

array_value() : starp_value_t (see, starp_value_t Class Reference on page 59)

- c -

clone() : matrix_factory_t (see, matrix_factory_t Class Reference on page 86)

colind : PPcsr (see, PPcsr Struct Reference on page 93)

communicator() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

COMPLEX : type_desc_t (see, type_desc_t Class Reference on page 82)

convert_data_to_type() : matrix_t (see, matrix_t Class Reference on page 66)

create() : matrix_factory_t (see, matrix_factory_t Class Reference on page 86)

create_matrix() : matrix_factory_t (see, matrix_factory_t Class Reference on page 86)

- d -

data_type_is() : matrix_t (see, matrix_t Class Reference on page 66)

dimension_count() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

dimensions_equal() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

distributed_dimension() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

distributed_value() : starp_value_t (see, starp_value_t Class Reference on page 59)

distributions_equal() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

- e -

element_size() : type_desc_t (see, type_desc_t Class Reference on page 82)

element_type_is() : starp_value_t (see, starp_value_t Class Reference on page 59)

end_idx() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

- f -

fst_row : PPcsr (see, PPcsr Struct Reference on page 93)

- g -

global_dense_data() : matrix_t (see, matrix_t Class Reference on page 66)

global_size() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

global_size_array() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

- h -

has_data() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

- i -

init() : initializer_t (see, initializer_t Struct Reference on page 91), type_desc_t (see, type_desc_t Class Reference on page 82)

is_a() : type_desc_t (see, type_desc_t Class Reference on page 82)

is_array() : starp_value_t (see, starp_value_t Class Reference on page 59)

is_distributed() : starp_value_t (see, starp_value_t Class Reference on page 59)

is_error_set() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

is_real() : starp_value_t (see, starp_value_t Class Reference on page 59)

is_root() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

is_scalar() : starp_value_t (see, starp_value_t Class Reference on page 59)

is_string() : starp_value_t (see, starp_value_t Class Reference on page 59)

is_valid_conversion() : type_desc_t (see, type_desc_t Class Reference on page 82)

- l -

last_error() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

layout_desc() : matrix_t (see, matrix_t Class Reference on page 66)

layout_desc_t() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

local_data_size() : matrix_t (see, matrix_t Class Reference on page 66)

local_dense_data() : matrix_t (see, matrix_t Class Reference on page 66)

local_size() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

local_size_array() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

local_sparse_data() : matrix_t (see, matrix_t Class Reference on page 66)

- m -

m_comm_rank : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_comm_size : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_dimension_count : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_distributed_dimension : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_end_idxs : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_global_size : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_global_size_array : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_init : type_desc_t (see, type_desc_t Class Reference on page 82)

m_loc : PPcsr

m_local_size_arrays : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_local_sizes : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_procs_with_data : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_start_idxs : layout_desc_t (see, layout_desc_t Class Reference on page 71)

m_type_rep : type_desc_t (see, type_desc_t Class Reference on page 82)

m_type_size : type_desc_t (see, type_desc_t Class Reference on page 82)

matrix_factory() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

matrix_factory_t : matrix_t (see, matrix_t Class Reference on page 66)

matrix_id() : matrix_t (see, matrix_t Class Reference on page 66)

matrix_t : layout_desc_t (see, layout_desc_t Class Reference on page 71), type_desc_t (see, type_desc_t Class Reference on page 82)

mpi_type() : type_desc_t (see, type_desc_t Class Reference on page 82)

my_rank() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

- n -

nnz_loc : PPcsr (see, PPcsr Struct Reference on page 93)

num_processes() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

number_of_elements() : starp_value_t (see, starp_value_t Class Reference on page 59)

nzval : PPcsr (see, PPcsr Struct Reference on page 93)

- o -

ones() : matrix_factory_t (see, matrix_factory_t Class Reference on page 86)

operator *() : smart_ptr_array< T >, smart_ptr< T >

operator bool() : smart_ptr< T >

operator idx_t() : starp_value_t (see, starp_value_t Class Reference on page 59)

operator starp_dcomplex_t() : starp_value_t (see, starp_value_t Class Reference on page 59)

operator starp_dcomplex_t const *() : starp_value_t (see, starp_value_t Class Reference on page 59)

operator starp_double_t() : starp_value_t (see, starp_value_t Class Reference on page 59)

operator starp_double_t const *() : starp_value_t (see, starp_value_t Class Reference on page 59)

operator std::string() : starp_value_t (see, starp_value_t Class Reference on page 59)

operator->() : smart_ptr_array< T >, smart_ptr< T >

operator<< : matrix_t (see, matrix_t Class Reference on page 66), layout_desc_t, type_desc_t (see, type_desc_t Class Reference on page 82)

operator=() : smart_ptr_array< T >, smart_ptr< T >

operator== : layout_desc_t (see, layout_desc_t Class Reference on page 71), type_desc_t (see, type_desc_t Class Reference on page 82)

operator[]() : smart_ptr_array< T >

- p -

procs_with_data() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

ptr() : smart_ptr_array< T >, smart_ptr< T >

- r -

rand() : matrix_factory_t (see, type_desc_t Class Reference on page 82)

REAL : type_desc_t (see, type_desc_t Class Reference on page 82)

redistribute() : matrix_t (see, matrix_t Class Reference on page 66)

register_func() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

release() : smart_ptr_array< T >, smart_ptr< T >

remove_matrix() : matrix_factory_t (see, matrix_factory_t Class Reference on page 86)

reset() : smart_ptr_array< T >, smart_ptr< T >

reset_error() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

root_rank() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

rowptr : PPcsr (see, PPcsr Struct Reference on page 93)

- s -

scalar_value() : starp_value_t (see, starp_value_t Class Reference on page 59)

SdkPackage : starp_value_t (see, starp_value_t Class Reference on page 59)

server() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

set_error() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)

smart_ptr : smart_ptr< T >

smart_ptr_array : smart_ptr_array< T >

starp_sdk_t : starp_sdk_t (see, starp_sdk_t Class Reference on page 53), matrix_factory_t, matrix_t (see, matrix_t Class Reference on page 66), type_desc_t (see, type_desc_t Class Reference on page 82)

starp_value_t : starp_value_t (see, starp_value_t Class Reference on page 59), matrix_t (see, matrix_t Class Reference on page 66)

start_idx() : layout_desc_t (see, layout_desc_t Class Reference on page 71)

storage_desc() : matrix_t (see, matrix_t Class Reference on page 66)

string_value() : starp_value_t (see, starp_value_t Class Reference on page 59)

- t -

type_desc() : matrix_t (see, matrix_t Class Reference on page 66)

type_desc_t() : type_desc_t (see, type_desc_t Class Reference on page 82)

- u -

use_count() : smart_ptr_array< T >, smart_ptr< T >

- z -

zeros() : matrix_factory_t (see, matrix_factory_t Class Reference on page 86)

- ~ -

~initializer_t() : initializer_t (see, initializer_t Struct Reference on page 91)

~matrix_t() : matrix_t (see, matrix_t Class Reference on page 66)

~smart_ptr() : smart_ptr< T >

~smart_ptr_array() : smart_ptr_array< T >

~starp_sdk_t() : starp_sdk_t (see, starp_sdk_t Class Reference on page 53)