CSlib Users Manual

8 Aug 2018 version

http://cslib.sandia.gov - Sandia National Laboratories

Copyright (2018) Sandia Corporation. This software and manual is distributed under the GNU Lesser General Public License.

Table of Contents

Manual for the Client/Server messaging library (CSlib) 1
- 8 Aug 2018 version 1
- Quick tour 1
- Introduction 3
- Features 5
- Client/server model of communication 5
- Use cases for the CSlib 6
- Running client/server apps 7
- Build the CSlib 9
- Test programs 10
- Build the test apps 11
- Run the test apps 12
- API for the CSlib 13
- Language requirements 14
  - Header files and modules to include 14
  - Compatible data types 15
  - Strings in C/C++ versus Fortran 16
  - Contiguous memory for 2d, 3d, etc arrays 17
  - Ordering of 2d, 3d, etc arrays in C/C++ versus Fortran 17
  - Splitting an MPI communicator in different languages 18
- Create and destroy the library 19
- Exchange messages 22
  - Send and recv methods 23
  - Pack and unpack a single value 25
  - Pack and unpack a string 27
  - Pack and unpack multiple datums 28
  - Pack and unpack multiple datums in parallel 30
  - Pack and unpack of multidimensional arrays 33
  - Who owns the data memory 36
  - Persistence of data 36
  - No matching requirement for pack and unpack methods 36
  - Synchronizing methods 37
  - How pack_parallel() and unpack_parallel() organize their data 37
  - Referencing a vector containing array data 38
  - Distributing data for pack_parallel() and unpack_parallel() 38
- Query message statistics 39
- More Python details 40
  - Building the CSlib for use from Python" 41
  - Loading the CSlib from Python at runtime 41
  - "Parallel python via mpi4py 42
  - Sending, receiving Python lists, Numpy arrays, ctypes vectors 42
- Error messages 43

Manual for the Client/Server messaging library (CSlib)

8 Aug 2018 version

The Client/Server messaging library (CSlib) is a lightweight implementation of the client/server model suitable for coupling two scientific applications runnning in serial or parallel.

- Quick tour
- Introduction
- Features
- Client/server model of communication
- Use cases for the CSlib
- Running client/server apps
- Build the CSlib
- Test programs
- Build the test apps
- Run the test apps
- API for the CSlib
  - language details
  - create and destroy the library
  - exchange messages
  - query statistics
- More Python info
- Error messages

PDF file of this manual, generated by htmldoc

The CSlib is an open-source code, distributed freely under the terms of the modified Berkeley Software Distribution (BSD) License. See this page for more details.

It was developed at Sandia National Laboratories, a US Department of Energy facility, for use in coupling scientific applications running on desktop machines to large supercomputers. Funding for the CSlib came from DOE's Office of Advanced Scientific Computing Program (OASCR) under the auspices of its Exascale Computing Program (ECP).

The author of the CSlib is Steve Plimpton, who can be contacted at sjplimp at sandia.gov.

Additional info is available at these links:

- CSlib website = documentation and tarball downloads
- GitHub site = clone or report bugs, pull requests, etc
- Questions, bugs, suggestions: email to sjplimp at sandia.gov or post to GitHub

IMPORTANT NOTE: the GitHub site will not be stood up until ~Aug 2018.

CSlib documentation

Quick tour

You can quickly try out the CSlib if you

- build the CSlib

- build the test programs that use the CSlib
- run the test programs

You can do this step-by-step (below), or do all these steps in one command from the test dir:

```
% cd cslib/test
% sh Run_serial.sh      # for serial apps
% sh Run_parallel.sh    # for parallel apps
% sh Run_mixed.sh       # for mixed apps (language, serial/parallel, etc)
```

These shell scripts build and run in all languages (C++, C, Fortran, Python), and for serial and parallel (via MPI). This includes a socket mode via the ZeroMQ (ZMQ) library. So if support for some of the options is not available on your system, apps may not build or run. In which case you can try the 3-step process below.

You can examine the source code of any of the apps in the test dir to see the logic and syntax for using the CSlib from your application. They invoke every method provided by the CSlib.

Build the CSlib with one of these make commands. This should produce one or more lib*.so files. If you're stuck, read this section.

```
% cd cslib/src
% make                # build 2 shared CSlibs: parallel & serial
% make zmq=no         # ditto if you don't have the ZeroMQ lib on your system
% make shlib_serial   # just the serial lib if you don't have MPI on your system
```

Build the test programs. This should produce pairs of executable files that start with "client" and "server". If you're stuck, read this section.

```
% cd cslib/test
% make all           # build all the apps (CSlib and simple)
% make serial        # build the CSlib serial apps, don't need MPI
% make parallel      # build the CSlib parallel apps, except fortran
% make f90           # build the CSlib fortran apps, serial & parallel
```

Run the test programs from the test dir. Launch the client app in one window, the server app in another. It doesn't matter which is launched first. At the end, you should see output from the client for the effective messaging bandwidth, and an indication if any errors occurred. The server should just silently exit. If you're stuck, read this section. The test apps are explained in more detail in this section.

These are the command-line args for the client and server apps:

```
mode = file or zmq                          # for serial apps
mode = file or zmq or mpi/one or mpi/two    # for parallel apps
Nlen = length of vectors to exchange
Niter = number of exchanges
dtype = Python data type: 1 for list, 2 for Numpy, 3 for ctypes
```

IMPORTANT NOTE: You must specify the same mode and Nlen for both apps. For mode = mpi/one, launch both apps via a single mpirun command, not two commands in two windows, for example:

```
% mpirun -np 2 client_parallel mode Nlen Niter : -np 4 server_parallel mode Nlen
```

Note that to run the Python apps, you must enable Python to find the CSlib shared library and src/cslib.py wrapper script. To run the parallel Python apps, you must have mpi4py installed in your Python. See this section for details on both these topics.

Serial C++, C, Fortran, Python:

```
% client mode Nlen Niter    # launch in one window
% server mode Nlen          # launch in second window

% client_c mode Nlen Niter
% server_c mode Nlen

% client_f90 mode Nlen Niter
% server_f90 mode Nlen

% python client.py mode Nlen Niter dtype
% python server.py mode Nlen dtype
```

Parallel C++, C, Fortran, Python:

```
% mpirun -np 2 client_parallel mode Nlen Niter
% mpirun -np 4 server_parallel mode Nlen

% mpirun -np 2 client_parallel_c mode Nlen Niter
% mpirun -np 4 server_parallel_c mode Nlen

% mpirun -np 2 client_parallel_f90 mode Nlen Niter
% mpirun -np 4 server_parallel_f90 mode Nlen

% mpirun -np 2 python client_parallel.py mode Nlen Niter dtype
% mpirun -np 4 python server_parallel.py mode Nlen dtype
```

Mix and match langauges and serial vs parallel. A serial and parallel app can only use the CSlib together for mode = file or zmq. If mode = mpi/two is used, both apps MUST be launched with mpirun, even if one or both of them only runs on a single processor.

```
% client mode Nlen Niter
% server_c mode Nlen

% client mode Nlen Niter
% python server.py mode Nlen

% client mode Nlen Niter
% mpirun -np 4 python server.py mode Nlen dtype

% mpirun -np 2 client_parallel mode Nlen Niter
% mpirun -np 4 server_parallel_f90 mode Nlen dtype

% mpirun -np 2 client_parallel mode Nlen Niter
% mpirun -np 4 python server_parallel.py mode Nlen dtype
```

CSlib documentation

Introduction

The Client/Server library (CSlib) tries to make it easy to couple two stand-alone applications (apps) together via a simple messaging paradigm, namely the client/server model. It was designed with scientific codes in mind, but could be used by any pair of apps that match the client/server model as described in this section.

The library hides the details of the low-level messaging from the apps. Both apps may be serial, both may be parallel (via MPI), or one can be serial and the other parallel. This is useful in at least a few scenarios:

1. One or both of the codes is not a library, so the other code cannot link to it and call it directly.
2. You wish to run the codes in parallel on different numbers of processors. If the first code calls the other code as a library, and you run in parallel with MPI, then both codes run on all the processors. This can be worked around, but the details can be ugly.
3. You want to create a generic protocol (format and content of messages) that allows one kind of code A to couple to another kind of code B. There may be multiple codes A and/or multiple codes B that could be coupled interchangeably. The CSlib enables a single protocol to be defined between all A codes and all B codes. By contrast if all the B codes are libraries, each A code would need to include code to call the various library interfaces, which are likely different.
4. You want to run the two codes on different machines in different geographic locations. The socket mode of messaging can do this.

To use the library, one of the apps acts as a "client", the other as a "server", as explained in this section. Thus you need to add client or server logic to an app to call the CSlib appropriately and send/receive messages. This is often easier to do than reconfiguring a code to run as a library. If it is impossible to add new logic to an existing code (e.g. no access to source, or others will not have access to source code changes you make), you may be able to still write a simple wrapper on that code (e.g. via Python). The wrapper can call the CSlib and communicate with the other code. The wrapper can create input files for the black-box code, launch it, and parse its output.

It may make sense for the same app to be able to function as either a client or server, depending on what kind of problem is being run. In this scenario, the same code could be launched twice, once as a client and once as a server. The two instances of the same code can then exchange messages with each other.

The CSlib can be used to couple N > 2 codes together. Simply instantiate the library once for each pair of codes you couple. For example, if N = 4, and each code needs to communicate with the other 3, the CSlib could be instantiated 6 times (4*3/2). A pair of codes can also both instantiate the CSlib twice (or more), each acting once as a client and once as a server. The two codes can then exchange messages with each other on different communication "channels" as needed.

As described in the next section, the CSlib performs its messaging between two parallel codes in a simple manner, using one processor of each. Thus it is not designed to efficiently couple codes when each is running on 100s or 1000s of processors, and where they frequently exchange large amounts of data. This requires higher bandwidth parallel data transfers, where many or all processors of each code exchange data with each other simultaneously via MPI.

Solutions to that problem typically use more specialized libraries and require the two codes be linked together as one application. Here are references to papers and software on this topic:

- Tinker-toy parallel programming
- A Parallel Rendezvous Algorithm for Interpolation Between Multiple Grids, S. J. Plimpton, B. Hendrickson, J. Stewart, J Parallel and Distributed Computing, 64, 266-276 (2004). (abstract)
- Data Transfer Kit (DTK)
- Adaptable IO System (ADIOS)

At some point we may extend the CSlib to include options for a more parallel or high-bandwidth form of code coupling, either wrapping these tools directly or using conceptual ideas they implement.

CSlib documentation

Features

The CSlib currently supports 4 messaging protocols or "modes" of code-to-code coupling:

- file = via files
- zmq = via sockets
- mpi/one = via MPI, where two apps are launched with one mpirun command
- mpi/two = ditto but launched with two mpirun commands

The file mode sends messages by writing/reading binary files.

The zmq mode is implemented using the open-soure ZeroMQ socket library, referred to in this manual as ZMQ.

The two mpi modes send messages via the distributed-memory message passing interface (MPI). They differ only in how the two applications (apps) are launched via MPI's mpirun or mpiexec command and in how the MPI_COMM_WORLD communicator and processor ranks are assigned to each app.

From the app's persepective (client or server code) all these modes function identically. The same calls are made to the CSlib, other than arguments to the constructor that chooses the messaging mode. For the "mpi/one" mode each application may also want to split the MPI communicator (MPI_COMM_WORLD) so that each app runs in its own communicator. This is discussed in this section.

In the CSlib, a "message" has a numeric ID and can contain zero or more "fields". A field contains zero or more datums of a single "type". A type is an integer or floating-point value with 32-bit or 64-bit precision. Character strings are also supported. Arbitrary byte sequences may be supported in a future release.

When a client or server code is run in parallel as a distributed-memory app, using MPI, only one of its processors participates in the CSlib messaging. In MPI parlance, the proc 0 of one code communicates with the proc 0 of the other, either by files, via a socket, or via MPI sends and receives.

However, all processors in each app call the CSlib functions simultaneously. They can format their messages in one of 2 ways. For sending, all processors can provide a copy of the same send data. Or if the data is distributed across the processors, each can provide its portion to the send() call. Likewise when the recv() is called, all processors can receive a copy of the entire message. Or if the data is meant to be distributed, each can receive its portion. The CSlib communicates data via MPI within the client and server apps to accomplish this.

CSlib documentation

Client/server model of communication

The client/server model is a messaging paradigm, used ubiquitously by web-based applications, where one or more clients (e.g. your phone) communicate with a server (e.g. a website). In the context of the CSlib, it can used to couple two scientific applications (apps) together where one app acts as the "client" and the other as the "server". The client sends messages to the server requesting it perform specific computations and return the results.

These are key points to understand about the client/server model, as it is implemented in the CSlib. The test apps, discussed in this section, illustrate all these points, so they are useful to look at to see how to morph your app so it can run as a client and/or server.

1. At run time, the client and server codes (apps) must use the same mode of communication (obviously), i.e. they agree to communicate by files, by sockets, or via MPI. The mode setup is performed by the

initial call to the library, made by both the client and server apps. For each mode there are arguments for the instantiation which must be consistent between the client and server.

2. All messaging follows a single pattern, repeated as many times as needed:
    - client sends a request message
    - server receives the request message
    - server sends a response message
    - client receives the response message
3. This means the server cannot initiate a send. It also means the client cannot send several messages in a row, then wait for a response. Likewise the server cannot send several messages in a row as a response, then wait for a new request.
4. When the client is done, it should tell the server there are no more requests by sending a final message. The server should respond to the client. The client should wait to receive that final message from the server.
5. To send a message, the client or server performs two operations. The first is to call send() with a message ID and # of fields. The second is to call pack() or pack_parallel() once for each field. Pack() is used if all processors have a copy of the same field data. Pack_parallel() is used if each sending processor owns a portion of the field data, e.g. each proc owns a subset of grid cells or particles. In the latter case, the message that is sent to the other app is the union of all the processor's portions.
6. To receive a message, the client or server performs two operations. The first is to call recv() which returns with a message ID, # of fields, and info about each field. The second is to call unpack() or unpack_parallel() once for each field (if desired). Unpack() returns a pointer to all the field data to every processor (even if it was sent via pack_parallel()). Unpack_parallel() copies the portion of the field data each receiving processor owns into memory owned by the app.
7. A recv() call is always blocking. It will not return until the message from the other app has arrived and the CSlib has stored the message data internally. A send() call is not blocking for the *file* mode. The file of send data is written and the call returns, whether the other app reads the file immediately or not. A send() call for the other modes (zmq, mpi/one, mpi/two) may or may not be blocking. It depends on the size of the message and the protocols that ZMQ and MPI use for sending/receiving large messages. If the message is large, the sender may block until the receiver is ready to receive the data.
8. You need to pay attention to whether the memory for various library calls is owned by the client or server apps or by the CSlib. It's different for sends versus receives. In the case of a send() or pack(), the app owns the memory. The CSlib makes an internal copy of the data to send it as a message to the other app. Thus as soon as the send() call returns, the app can re-use its memory. In the case of a recv() or unpack(), the CSlib owns the memory because it allocated it to receive the message from the other app. The CSlib simply returns pointers to that internal memory to the app. The app can examine the values or it can make its own copy if desired. In the case of unpack_parallel() the portion of returned field data is copied into the app's memory.
9. Note that preceeding paragraph implies something important. The CSlib re-uses its internal memory for every new send or receive call, for both per-field info and per-field data. This means the app cannot expect the pointers to received data returned from the recv() and unpack() calls to remain valid, once it begins to call send(), pack(), or pack_parallel() to send the next message. The app MUST make its own copy of the recv() or unpack() data if persistence is required. This is true even if the app simply wishes to in-place modify a vector of field data to return it to the other app. To do this, the app should make a copy of the vector, modify the copy, then send the copy. We may add alernate memory usage options to the CSlib in the future.

CSlib documentation

Use cases for the CSlib

Here we describe different ways of using the CSlib in either serial or parallel. Note that a pair of client and server apps can use the library in different ways, one can be a serial app, the other a parallel app.

Also note that "running an app in parallel" and "using the CSlib in parallel" does not preclude running the app on a single processor. Here we are talking about whether the app and the CSlib are built with MPI support or not. MPI itself operates perfectly fine on a single processor.

- It the app is serial (does not use or know about MPI), you must use the CSlib in serial, since there is no way for the app to pass it an MPI communicator. This means you can only use the file or zmq modes of messaging. You cannot use an MPI mode of messaging (mpi/one or mpi/two), even if the other app uses the CSlib in parallel.
- If the app is parallel, you should normally use the CSlib in parallel. If both apps are parallel, you can use any of the messaging modes the CSlib provides, as described in this section.
- We say "normally" in the preceeding paragraph, because nothing prevents a parallel app from instantiating the CSlib serially (w/out passing an MPI communicator). In that case, only proc 0 should normally instantiate and make calls to the CSlib. The app will be responsible for having proc 0 communicate message data to its other processors as needed, instead of the CSlib doing it.
- We say "normally" for a 2nd time in the preceeding paragraph, because you could also use the CSlib in either of the following ways. If all procsessors in one app instantiate the CSlib in serial, then each will look for a partner client or server to exchange messages with. If the other app does the same thing, then pairs of processors, one in each app, could exchange messages with each other. Or if two processors in the same app instantiate the CSlib in serial, one could act as a client, the other as a server, and they could exchange messages. In the latter case you don't need two client and server apps; a single app is performing both functions. Again, all of the messaging must use the file or zmq mode, not an MPI mode, because the CSlib is being used serially.
- Finally, two parallel apps can use multiple instances of the CSlib in parallel. Each app can split their processors into sub-communicators. Each sub-commuicator in one app can pair with a sub-communicator in the other app. If sub-communicators in each app instantiate the CSlib, they can exchange messages with a sub-communicator in the other app using any of the messaging modes the CSlib provides. Similarly, a single parallel app, acting as both client and server, can use the CSlib in parallel. If the app splits its MPI communicator into two (or more) sub-communicators, pairs of sub-communicators can each instantiate the CSlib. One sub-communicator acts as a client, the other as a server, and they can exchange data in any of the messaging modes.

IMPORTANT NOTE: The CSlib must be built differently for use in serial versus parallel. Build it with no MPI support for serial, and with MPI support for parallel. This section explains how to build in both ways, which will produce libraries with different names, so an invididual app can link against either.

TODO: We should think about eventually building with MPI support but allowing serial use (no MPI communicator).

CSlib documentation

Running client/server apps

To run a client/server model requires launching two codes (apps), often at (nearly) the same time. Here are examples of how to do this on different platforms.

Desktop machine:

You can launch the client and server apps in different terminal windows like this:

```
% client args ...              # serial executable (client) in one window
% server args ...              # ditto for server in other window

% python client.py args ...    # two serial Python scripts
```

```
% python server.py args ...

% mpirun -np 2 client args ...          # two parallel executables
% mpirun -np 4 server args ...

% mpirun -np 2 python client.py args ...  # two parallel Python scripts
% mpirun -np 4 python server.py args ...

% client args ...                        # serial executable in one window
% mpirun -np 4 server args ...           # parallel executable in the other
```

You can also launch both executables in the same window, if you append a "&" character to launch the first app in the background, like this:

```
% mpirun -np 2 client args ... &         # two parallel executables in same window
% mpirun -np 4 server args ...
```

If the two apps use the "mpi/one" messaging mode, they must be launched together using a single mpirun command like this:

```
% mpirun -np 2 client args ... : -np 4 server args
```

If the two apps use the "mpi/two" messaging mode, they must be both be launched with an mpirun command, even if one or both them run on a single processor. This is so that MPI can figure out how to connect both MPI processes together to exchange MPI messages between them.

IMPORTANT NOTE: In any of these launch scenarios, for any mode of messaging, it does not matter whether you launch the client or server app first. All the CSlib messaging modes will block when the CSlib is instantiated, until a connection is made with the other app. An exception is the file mode where each app will block at its first recv() until the other app performs a matching send().

Two desktop machines:

TODO: Need to flesh out this section with examples. Only file and zmq mode are relevant.

- talk about port IDs for zmq mode and geographically distinct machines - give example port IDs?
- file mode requires a common file system (may not be possible when geographically distinct)

Cluster or supercomputer:

On these machines you typically run from a single window if nodes were requested interactively, or from a batch script in a queueing system (Slurm, MOAB, etc). In both cases the launch comands listed above for a "Desktop machine" should work, so long as a "&" character is used at the end of the first command (launching the first app), so that both apps can effectively be launched at the same time.

- TODO: need to give examples of OpenMPI syntax for task
- placement
- TODO: talk about port IDs for zmq mode, how to ID them on a cluster
- TODO: talk about file mode requiring a common file system

Note that on a cluster or a supercomputer (or even on a single node, i.e. a desktop), the mpirun command gives you control over which nodes and cores an application is launched on. Since you are running two apps (client and server), you should think about how you want to use the available compute resources. If you have P total MPI tasks (typically P = nodes * cores/node), you could do any of the following:

- run both the client and server on all P tasks
- run the client on P1 tasks and the server on P2 = P - P1 tasks
- run the client on P1 < P tasks and the server on P tasks (or vice versa)

If the computation the client performs is tiny compared to what the server computes, it makes sense to run the server on more processors. Or vice versa. If the client does nothing while waiting for the server to respond, and ditto while the server waits for a request from the client, then it may make sense to run one of the client and server on all processors and the other on some (or all) of the processors. Or at least to run proc 0 of both the client and server on the same core, so they can exchange messages quickly, and because neither will be computing on the core at the same time.

IMPORTANT NOTE: When launching a parallel client or server app that uses the CSlib in parallel you obviously use the mpirun command (or equivalent). When running a parallel app on a single processor, you may be used to launching it using just the executable name (without mpirun). However, when using the CSlib in one of its mpi modes, you MUST usee mpirun to launch both apps. This is because you are launching two apps (the client and server) and relying on their respective mpirun environments to enable each app to find the other and send/receive MPI messages between them.

IMPORTANT NOTE: If a client or server app crashes or hangs (and you kill it), e.g. due to a bug in how you call the CSlib, then temporary files may be left on your system. These will have the names or prefixes specified when the apps instantiated the CSlib. You should delete these files before running again, otherwise the existing files may produce unpredictable behavior in a subsequent run.

CSlib documentation

Build the CSlib

The CSlib can be built with MPI support (for parallel use by parallel apps), or without MPI support (for serial use by serial apps). It can be built as a shared or static library. It can be built with or without socket support via the ZeroMQ (ZMQ) library.

All of this is done from the src dir. There is a single Makefile with various targets and options. Type "make help" to see this list:

```
make                    default = shlib
make shlib              build 2 shared CSlibs: parallel & serial
make lib                build 2 static CSlibs: parallel & serial
make all                build 4 CSlibs: shlib and lib
make shlib_parallel     build shared parallel CSlib
make shlib_serial       build shared serial CSlib
make lib_parallel       build static parallel CSlib
make lib_serial         build static serial CSlib
make ... zmq=no         build w/out ZMQ support
```

Try the default shared library build first, by just typing "make". Shared libraries are preferred since you will not need to include additional libraries like ZMQ when you link the CSlib with an app. You can build static libraries if they are required on a large parallel machine. Note that a shared library is required to use the CSlib from a Python app. It's useful to have both a parallel and serial version of the CSlib so that either parallel and serial apps can be run.

If you only need a parallel shared library, type "make shlib_parallel". If you do not have MPI on your system, type "make shlib_serial".

If you don't have the ZMQ library on your system, you can append "zmq=no" to any of the listed make commands. The resulting library will not allow use of mode zmq for messaging via sockets.

Note that you may wish to build the CSlib multiple times. For example, if the client app is a serial program that does not use MPI, it will link to the CSlib with no MPI support, while if the server app runs in parallel, it will link to the CSlib with MPI support.

If the build is successful, one or more of these library files are created:

```
libcsmpi.so         # shared lib with MPI suppport
libcsnompi.so       # shared lib with no MPI support
libcsmpi.a          # static lib with MPI support
libcsnompi.a        # static lib with no MPI support
```

The provided Makefile assumes the MPI and ZMQ header files, compilers, libraries, etc are in your path. If that is not the case or you wish to use different compilers or flags, you can create/edit a new Makefile, e.g. Makefile.mine, and invoke it as follows:

```
make -f Makefile.mine shlib ...
```

IMPORTANT NOTE: When building a client or serial app that uses the CSlib library in parallel, the app will also be built with MPI, since it passes an MPI communicator to the CSlib. You MUST insure the same MPI library is used for building both the CSlib and client/server apps. Otherwise you wlll get errors, either at build time (when the app is built) or at run time (because the MPI used by the CSlib will not be properly initialized by the app). Insuring use of the same MPI library can be non-trivial when a system has multiple versions of MPI, or the CSlib and apps are built at different times or by different people.

CSlib documentation

Test programs

There are several pairs of test programs in the test dir. These use the CSlib:

client.cpp, server.cpp	serial client/server in C++
client_c.cpp, server_c.cpp	serial c/s in C
client_f90.f90, server_f90.f90	serial c/s in Fortran 90
client.py, server.py	serial c/s in Python
client_parallel.cpp, server_parallel.cpp	parallel c/s in C++
client_parallel_c.cpp, server_parallel_c.cpp	parallel c/s in C
client_parallel_f90.f90, server_parallel_f90.f90	parallel c/s in Fortran 90
client.parallel_py, server_parallel.py	serial c/s in Python

These are stand-alone apps which do not use the CSlib:

simple_client_zmq, simple_server_zmq	c/s in native ZMQ
simple_client_mpi_one, simple_server_mpi_one	c/s in native MPI via single mpirun
simple_client_mpi_two, simple_server_mpi_two	c/s in native MPI via two mpiruns

Codes in the first table (CSlib test apps) illustrate how to use the CSlib from different languages, for both serial and MPI parallel applications. Codes in the second table (simple test apps) do not use the CSlib. They are bare-bones codes that call the zeroMQ (ZMQ) and MPI libraries directly. They can be useful for debugging if the

CSlib tests do not build or work properly on your system for some reason.

The CSlib test apps written in C++ instantiate the CSlib as a C++ class. The C apps are actually written in (vanilla) C++ but call the CSlib through its C interface, just as a true C program would. The Fortran programs also call the CSlib through its C interface, using a Fortran iso_c_binding interface file (src/cslib_wrap.f90 or test/cslib_wrap.f90). The Python scripts use the Python interface in src/cslib.py), which in turn wraps the C interface of the CSlib.

The serial CSlib test apps must each be run on a single processor. The parallel CSlib test apps can be run via mpirun on any number of processors, including a single processor. Parallel client and server apps can run on different numbers of processors. You can run a serial client app with a parallel server app, or vice versa. You can run a client app in one language with a server app in another language.

All the CSlib test apps are functionally identical. They exercise the complete set of library calls for the CSlib. They can be run in any of the messaging modes (except the serial apps cannot use an MPI mode). The client sends a set of vectors and scalars to the server. The server receives them, increments the vectors and scalars, and sends them back to the client. The client receives them, and updates the data in its vectors and scalars. This is repeated N times after which the client sends an all-done message to the server. The vector length is an input to both the client and server; the iteration count N in an input to the client. The Python apps also have an input for which Python data type to use for vectors (lists or tuples, Numpy array, ctypes vectors), since the CSlib Python wrapper (src/cslib.py) will work with all 3.

At the end of a run, both the client and server app should exit cleanly. The client prints stats and timing info about the run, including the effective messaging bandwidth. It also prints an error metric on the max difference for any element of any of the vectors or scalars, between the acutal versus expected value. If it is non-zero and tiny, there may have been some accumulated round-off error for floating point arithmetic. If it is not tiny, there is likely a bug in the test program or the CSlib.

CSlib documentation

Build the test apps

This is done from the test dir after building the CSlib, as explained in this section.

As explained in this section, the test dir contains pairs of client/server programs which use the CSlib (CSlib test apps), as well of pairs of simple client/server apps which call the ZMQ and MPI libraries directly.

There is a single Makefile for building all of them. Type "make" to see this list.

```
make all           build all the CSlib and simple apps
make serial        build the CSlib serial apps, except fortran
make parallel      build the CSlib parallel apps, except fortran
make f90           build the CSlib fortran apps, serial & parallel
make simple        build the simple apps
make pair          build a single pair of c/s apps (list below)

single pairs of test apps:
  C++: serial_c++, parallel_c++
  C:   serial_c, parallel_c
  F90: serial_f90, parallel_f90
single pairs of simple apps:
  simple_zmq, simple_mpi_one, simple_mpi_two
```

Try the "make all" build first. If you have MPI on your system and the requisite compilers, everything should just build. If not, you can use the other targets to avoid building certain executables. Or you can type commands like this if you only want to build specific pairs of client/server apps.

```
make serial_c++
make parallel_c
make simple_mpi_one
```

When builds are successful, you should have pairs of client and server executable files with the corresponding names in the test dir.

IMPORTANT NOTE: To build the serial CSlib apps, you must link against the serial shared CSlib library. To build the parallel CSlib apps, you must link against the parallel shared CSlib library. As explained in this section, you can build both these versions of the CSlib by simply typing "make" in the src dir.

IMPORTANT NOTE: If you only built static versions of the CSlib, and you built the CSlib with ZMQ support, you will need to include the ZMQ library in the link lines in this Makefile (or a similar Makefile you use to build your own apps).

The Makefile assumes that standard compilers and MPI wrappers are in your path. If that is not the case or you wish to use different compilers or flags, you can create/edit a new Makefile, e.g. Makefile.mine, and invoke it as follows:

```
make -f Makefile.mine target ...
```

CSlib documentation

Run the test apps

After you have built the CSlib and built the test apps, you can run them by following the directions in the quick tour, which explains the command-line arguments used by both the client and server apps. Both the quick tour and running client/server apps sections explain the syntax for launching both apps together, either from two terminal windows or a single window, or on a parallel machine.

We explain some additional details here.

You can perform a series of test runs via the shell scripts provided in the test dir:

```
% cd cslib/test
% sh Run_serial.sh      # for serial apps
% sh Run_parallel.sh    # for parallel apps
% sh Run_mixed.sh       # for mixed apps (language, serial/parallel, etc)
% sh Run_simple.sh      # for simple apps
```

which first build the CSlib, then build a subset of test apps, then perform a series of runs in serial or parallel using all the langauges.

As explained in "this section", after a successful run both apps should exit cleanly, the error count should be zero, and the effective messaging bandwidth of the run should be shown.

You can also run any pairing of a single client and single server app together from the command line. Note that the two apps can be in different languages, the two apps can be run on different processor counts, or one can be serial and the other parallel. For the latter, you must use mode = "file" or "zmq"; modes "mpi/one" and "mpi/two"

require both apps be parallel. The Run_mixed.sh scripts gives several examples of this.

The quick tour section gives many examples of the syntax for running a client and server app from two terminal windows. The *.sh scripts illustrate the syntax for running both apps from one window, by simply appending a "&" character to run the first app in the background.

Note that for mode = "mpi/one", a single mpirun command launches both apps, so this mode must be used from one terminal window.

Note that for mode = "mpi/two", each app must be launched by a mpirun command, even if they run on a single processor. This is so that MPI can figure out how to connect both MPI processes together to exchange MPI messages between them.

If the Python apps fail to launch or give errors, see this section for details on how to setup Python to work with the CSlib on your machine.

The simple apps do not use the CSlib and may be useful to test if the ZeroMQ (ZMQ) library and MPI library work as expected on your system. You can run two of them like this, from different terminal windows.

```
% simple_client_zmq 10 &       # 10 = # of times to loop
% simple_server_zmq

% mpirun -np 3 simple_client_mpi_two 10 &    # 10 = # of times to loop
% mpirun -np 4 simple_server_mpi_two
```

And the third like this, from one terminal window

```
% mpirun -np 3 simple_client_mpi_one 10 : -np 4 simple_server_mpi_one
```

Again, 10 = # of times to loop.

The simple ZMQ example is adpated from the ZeroMQ guidebook web page (C version).

CSlib documentation

API for the CSlib

The CSlib is a C++ class, so it can be used directly from C++. It can also be used by any langauge which can call C functions, since a C-style interface to the library is provided. A Fortran 2003 iso_c_binding interface file and a Python wrapper are also provided, both of which use the C interface.

The APIs for all these languages are conceptually identical, they just differ in syntax.

The C++, C, Fortran, and Python apps in the test dir illustrate use of this API. They call all of the functions listed below. These apps are described in this section.

When running a parallel client or server app, all the processors in the app should typically make the same calls at the same time to the CSlib. Several of the CSlib methods are effectively synchronizing (as noted on their doc pages), meaning they perform internal MPI communication between the processors in a single app.

Here is the short list of all the CSlib functions, from the C++ class header file. Click on a link for an explanation, including example code for how to call the CSlib from various languages.

Language requirements:

This page describes how to use the CSlib from apps written in different languages.

Create and destroy:

```
CSlib(int csflag, char *mode, char *txt, MPI_Comm *pworld);
~CSlib();
```

Send a message:

```
void send(int msgID, int nfield);
void pack_int(int id, int value);
void pack_int64(int id, int64_t value);
void pack_float(int id, float value);
void pack_double(int id, double value);
void pack_string(int id, char *string);
void pack(int id, int ftype, int flen, void *data);
void pack_parallel(int id, int ftype, int nlocal, int *ids, int nper, void *data);
```

Receive a message:

```
int recv(int &nfield, int *&fieldID, int *&fieldtype, int *&fieldlen);
int unpack_int(int id);
int64_t unpack_int64(int id);
float unpack_float(int id);
double unpack_double(int id);
char *unpack_string(int id);
void *unpack(int id);
void unpack_parallel(int id, int nlocal, int *ids, int nper, void *data);
```

Query statistics:

```
int extract(int flag);
```

CSlib documentation

Language requirements

This page describes language specific issues to be aware of when writing client or server apps in different languages.

- Header files and modules to include
- Compatible data types
- Strings in C/C++ versus Fortran
- Contiguous memory for 2d, 3d, etc arrays
- Ordering of 2d, 3d, etc arrays in C/C++ versus Fortran
- Splitting an MPI communicator in different languages

Header files and modules to include

Include these lines in any app file that makes calls to the CSlib.

C++:

```
#include "cslib.h"              // found in src dir
using namespace CSLIB_NS;
```

This is the CSlib class header file and its namespace If you prefer not to use the namespace, you can prepend the namespace to the call that instantiates the CSlib and the pointer it returns, e.g.

```
CSLIB_NS::CSlib *cs = new CSLIB_NS::CSlib(csflag,mode,str,&world);
```

C:

```
#include "cslib_wrap.h"          // found in src dir
```

This is the C-style wrapper on the CSlib class, found in src.

Fortran (2003 or later):

Include these lines in any program element (main program, subroutine, function, etc) which calls the CSlib:

```
USE iso_c_binding       ! standard Fortran module
USE cslib_wrap          ! found in src or test dir
```

The iso_c_binding module defines the C-style data types and functions used in calls to CSlib. The cslib_wrap module defines how Fortran calls the C interface to the CSlib. The code for the interface is in src/cslib_wrap.f90 file (also test/cslib_wrap.f90). That file must be compiled with the app; you can simply copy it into the directory with other app source files.

Python:

```
import cslib as CS      # found in src dir
from mpi4py import MPI
```

The first line references the file src/cslib.py which defines a Python CSlib class which wraps the C interface to the CSlib. The second line is only necessary if you are running a parallel Python application, so that you can pass an MPI communicator to the CSlib.

See more details in this section on how to insure these two lines will work in your Python.

Compatible data types

For the data types the CSlib supports, these are the equivalent native data types in different languages. If sending a message with this type of data, you need to be sure both the client and server app agree on how to declare the data.

Data type	C/C++	Fortran	Python/Numpy	Python/ctypes
32-bit integer	int	integer(4)	intc	c_int
64-bit integer	int64_t	integer(8)	int64	c_longlong
32-bit floating point	float	real(4)	float32	c_float
64-bit floating point	double	real(8)	float	c_double
string	char *	character(c_char)	N/A	N/A

Note that a character string in Fortran can also be declared as "character(len=32)" for sending. But for receiving, declare it as "character(c_char), pointer" and use the c_f_pointer() function to convert the received C-style string into the correct Fortran length. This is discussed below.

Note that regular Python does not require declaration of data types. You can assign values like this:

- a = 5
- a = 3000000000
- a = 1.5
- a = 1.5e-200
- a = "hello"

and Python knows whether it is an integer (any precision), floating point, or string, and will send/receive it to/from the CSlib correctly. When using Numpy and ctypes within Python, it is different. When you allocate a vector or multidimentional array of numeric data types, you must use the type identifiers above to declare the data correctly. See the Python test codes in the test dir for examples.

Read more about Numpy data types here:

https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html

Read more about ctypes data types here:

https://docs.python.org/2/library/ctypes.html

Strings in C/C++ versus Fortran

Character strings in C or C++ are NULL-terminated. This means the string "hello" requires 6 bytes of storage. Strings in Fortran are not NULL-terminated. So "hello" uses only 5 bytes of storage. Or it may be assigned like this:

```
character(len=32) :: string
string = "hello"
```

in which case 27 space characters are added to the end, to fill a 32-character string.

Internally, the CSlib stores strings in the C/C++ format with a trailing NULL. This means a Fortran app that sends a string must pass it to the CSlib in that format, via the pack_string() method described in this section. For example,

```
character(len=32) :: str = "hello
call cslib_pack_string(cs,1,trim(str)//c_null_char)
```

The trim() function trims trailing space (if any). A c_null_char is appended.

Likewise a Fortran app that receives a string will receive it in the C format, via the unpack_string() method described in this section. For examples,

character(c_char), pointer :: str(:) => null() ptr = cslib_unpack_string(cs,1) call c_f_pointer(ptr,str,[fieldlen(1)-1])

The c_f_pointer() function assigs the C pointer to the string to the Fortran variable str, and sets its lenght to one less than the received fieldlen(). This removes the trailing NULL.

Contiguous memory for 2d, 3d, etc arrays

As explained in this section and this section data in multidimensional arrays can be passed as an argument to pack() or pack_parallel() for sending (or received by unpack_parallel()), but only if the data is stored contiguously in memory.

For Fortran and Python (Numpy arrays or ctypes arrays) this is the case. However for C/C++ it may not be the case, depending on how the app creates a multidimensional array. The data in this kind of 2d Nx3 array:

```
int a10003;
```

is stored contiguously. But its dimensions must be specified at compile time.

A "double **" pointer to an Nx3 2d array in C, is more flexible. It can be allocated at run time. But it may not store the 3N values in contiguous memory, depending on how it was allocated. It may be a pointer to N pointers, each of which points at 3 contiguous values. But the N 3-vectors are not contiguous with each other.

The example code in this section and this section assumes the allocation for a 2d array is done contiguously (and later freed) via methods like this. Note that in malloc2d() there are two mallocs, one for 3N doubles (the data), the other for N pointers. A reference like &a**00** then points to the data as a 1d chunk and can be passed to pack() or unpack_parallel().

```
double **a = malloc2d(1000,3);
free2d(a);

double **malloc2d(int n1, int n2)

  int nbytes = n1*n2 * sizeof(double);
  double *data = (double *) malloc(nbytes);
  nbytes = n1 * sizeof(double *);
  double **array = (double **) malloc(nbytes);

  int n = 0;
  for (int i = 0; i <n1; i++)
    array**i** = &data**n**;
    n += n2;

  return array;


void free2d(double **array)

  if (array == NULL) return;
  free(array**0**);
  free(array);
```

Similar mallocs/frees can be written for 3d, 4d, etc arrays.

Ordering of 2d, 3d, etc arrays in C/C++ versus Fortran

Multidimensional arrays in Fortran store their data contiguously in memory with the leftmost index varying fastest, and the rightmost index varying slowest. Thus for this array:

```
integer(4) :: a(2,2)
```

the order of the elements in memory is a(1,1), a(2,1), a(1,2), a(2,2).

For C/C++ it is the oppposite (assuming the array data is stored in contiguous memory as described above, and not as pointers to pointers). The rightmost index varies fastest, the leftmost index varies slowest. So for this array:

```
int a22
```

the order of the elements in memory is a**00**, a**01**, a**10**, a**11**.

Since the CSlib treats multidimensional arrays as simply a chunk of contiguous memory, if you wish to send an array from a C/C++ app to a Fortran app, or vice versa, you must declare them in the 2 languages consistently. For example, an array declared as Nx3 in C/C++ should be declared as 3xN in Fortran.

Splitting an MPI communicator in different languages

As explained in this section, if the CSlib is used with messaging *mode* = "mpi/one", then both the client and server app are running within the same MPI communicator which spans both the client and server processors, typically MPI_COMM_WORLD. That spanning communicator is passed as an argument when the CSlib is created (instantiated) by the app. A *pworld* argument is also passed which is the sub-communicator for the processors the client or server app is running on. To create the two sub-communicators, both apps must execute a MPI_Comm_split() before instantiating the CSlib. Note that the MPI_Comm_split() call is synchronous; all processors in both the client and server apps must call it at the same time. Here is example code for this operation in each language. The line with the call to CSlib or cslib is what instantiates the library with the two communicators as arguments.

C++:

```
int me,nprocs,csflag;
MPI_Comm world,partial,both;

MPI_Init(&argc,&argv);
world = MPI_COMM_WORLD;
MPI_Comm_rank(world,&me);
csflag = 0;                     // 0 for client, 1 for server
MPI_Comm_split(world,csflag,me,&partial);
both = world;
world = partial;                // reset app's world,me,nprocs to sub-communicator
MPI_Comm_rank(world,&me);
MPI_Comm_size(world,&nprocs);
cs = new CSlib(csflag,mode,&both,&world);

MPI_Comm_free(&world);          // free the sub-commuicator when app exits
```

C:

```
identical to C++, except for this final line:
cslib_open(cslib,mode,&both,&world,cs);
```

Fortran:

```
integer :: me,nprocs,csflag,partial,ierr
integer, target :: world,both

call MPI_Init(ierr)
world = MPI_COMM_WORLD
call MPI_Comm_rank(world,me,ierr)
```

```
csflag = 0;                 ! 0 for client, 1 for server
call MPI_Comm_split(world,csflag,me,partial,ierr)
both = world
world = partial             ! reset app's world,me,nprocs to sub-communicator
call MPI_Comm_rank(world,me,ierr)
call MPI_Comm_size(world,nprocs,ierr)
call cslib_open_fortran_mpi_one(0,trim(mode)//C_null_char, &
                                c_loc(both),c_loc(world),cs)

call MPI_Comm_free(world,ierr)  ! free the sub-commuicator when app exits
```

Python:

```
from mpi4py import MPI    # this performs MPI Init
world = MPI.COMM_WORLD
me = world.rank
csflag = 0                # 0 for client, 1 for server
partial = world.Split(csflag,me)
both = world
world = partial           # reset app's world,me,nprocs to sub-communicator
me = world.rank
nprocs = world.size
cs = CSlib(0,mode,both,world)

world.Free()              # free the sub-commuicator when app exits
```

CSlib documentation

Create and destroy the library

These calls create an instance of the CSlib and later destroy it.

API:

```
CSlib(int csflag, char *mode, void *ptr, MPI_Comm *pworld);
~CSlib();
```

The ptr argument is different for different modes, as explained below.

C++:

```
#include "cslib.h"                  // header file for CSlib

int csflag = 0;                     // 0 for client, 1 for server
char *mode = "file";                // "file" or "zmq" or "mpi/one" or "mpi/two", see below
char *ptr = "tmp.couple";           // filename or socket ID or MPI communicator, see below

MPI_Comm world = MPI_COMM_WORLD;    // MPI communicator

CSlib *cs = new CSlib(csflag,mode,ptr,NULL);     // serial usage
CSlib *cs = new CSlib(csflag,mode,ptr,&world);   // parallel usage

delete cs;
```

Note the following details of the C++ syntax:

- To use the CSlib in serial, the final argument is NULL. To use it in parallel, the final argument is a

pointer to an MPI communicator, NOT the communicator itself.
- The ptr argument is always a pointer, but points to different things depending on the mode. More details below.

C:

```
#include "cslib_wrap.h"                   // header file for C interface to CSlib
```

Same argument declarations and notes as above for C++.

void *cs; // pointer to CSlib instance, used in all calls

```
cslib_open(csflag,mode,ptr,NULL,cs);      // serial usage
cslib_open(csflag,mode,ptr,&world,cs);    // parallel usage

cslib_close(cs);
```

Fortran:

use iso_c_binding ! C-style data types and functions for Fortran use CSlib_wrap ! F90 interface to CSlib

```
type(c_ptr) :: cs                          ! pointer to CSlib instance, used in all calls
integer :: csflag = 0
character(len=32) :: mode = "file"
character(len=32) :: txt = "tmp.couple"
integer, target :: world,both
world = MPI_COMM_WORLD

call cslib_open_fortran(csflag,trim(mode)//c_null_char, &
          trim(txt)//c_null_char,c_null_ptr,cs)              ! serial usage
call cslib_open_fortran(csflag,trim(mode)//c_null_char, &
          trim(txt)//c_null_char,c_loc(world),cs)            ! parallel usage

call cslib_open_fortran_mpi_one(csflag,trim(mode)//c_null_char, &
          c_loc(both),c_loc(world),cs)                       ! parallel usage for mode = mpi/one

call cslib_close(cs)
```

Note the following details of the Fortran syntax:

- The mode and txt strings are passed as NULL-terminated C-style strings by trimming them (remove trailing space) and appending a c_null_char.
- To use the CSlib in serial, a NULL if passed for the pworld argument as a c_null_ptr. To use the CSlib in parallel, the MPI communicator is passed as a pointer via c_loc(world).
- Using the CSlib with the mpi/one mode requires calling a different method. Two MPI communicators are passed, as explained below.

Python:

```
from cslib import CSlib                # Python wrapper on CSlib
from mpi4py import MPI              # Python wrapper on MPI

csflag = 0
mode = "file"
ptr = "tmp.couple"
world = MPI.COMM_WORLD
```

```
cs = CSlib(csflag,mode,ptr,None)        # serial usage
cs = CSlib(csflag,mode,ptr,world)       # parallel usage

del cs                                  # not really needed, Python cleans up
```

Here are more details about the CSlib constructor arguments.

IMPORTANT NOTE: The client and server apps must specify all the following arguments consistently in order to communicate. The CSlib has no way to detect if this is done incorrectly, since the two apps run as separate executables. The apps will likely just hang or even crash if that happens.

The *csflag* argument determines whether the app is the client or the server. Use *csflag* = 0 if the app is a client, *csflag* = 1 if the app is a server.

Mode can be one of 4 options. Both apps must use the same *mode*. The choice of *mode* determines how the *ptr* argument is specified.

For *mode* = "file" = the 2 apps communicate via binary files. The *ptr* argument is a filename which can include a path, e.g. ptr = "subdir/tmp.couple" or *ptr* = "/home/user/tmpdir/dummy1". Files that begin with this filename will be written, read, and deleted as messaging is performed. The filename must be the same for both the client and server app. For parallel apps running on a large parallel machine, this must be a path/filename visible to both the client and server, e.g. on the front-end or on a parallel filesystem. It should not be on a node's local filesystem if the proc 0 of the two apps run on different nodes. If the CSlib is instantiated multiple times (see this section for use cases), the filename must be unique for each pair of client/server couplings.

For *mode* = "zmq" = the 2 apps communicate via a socket. The *ptr* argument is different for the client and server apps. For the client app, ptr is the IP address (IPv4 format) or the DNS name of the machine the server app is running on, followed by a port ID for the socket, separated by a colon. E.g.

```
ptr = "localhost:5555"        # client and server running on same machine
ptr = "192.168.1.1:5555"      # server is 192.168.1.1
ptr = "deptbox.uni.edu:5555"  # server is deptbox.uni.edu
```

For the server, the socket is on the macjhine the server app is running on, so ptr is simply

```
ptr = "*:5555"
```

where "*" represents all available interfaces on this machine, and 5555 is the port ID. Note that the port (5555 in this example) must be specified as the same value by the client and server apps.

Note that the client and server can run on different machines, separated geographically, so long as the server accepts incoming TCP requests on the specified port.

TODO: need to answer these Qs

- what values can port ID have?
- need port examples for a big cluster, how to find node IP addresses
- can 2 servers run on same machine with different ports?

If the CSlib is instantiated multiple times, (see this section for use cases), the port ID (5555 in this example) must be unique for each pair of client/server couplings.

For *mode* = "mpi/one" or *mode* = "mpi/two", the 2 apps communicate via MPI.

For "mpi/one", the *ptr* argument is a pointer to the MPI communicator that spans both apps, while the *pworld* argument (discussed below) is a pointer to the MPI communcator for just the client or server app. This means the two apps must split the spanning communicator before instantiating the CSlib, as discussed below. The same as for *pworld* discussed below, the *ptr* argument is NOT the MPI communicator itself, but a pointer to it.

For "mpi/two", the *ptr* argument is a filename created (then deleted) for a one-time small exchange of information by the client and server to setup an MPI inter-communicator for messaging between them. As with the file mode, the filename can contain a path, and the same filename must be specified by the client and server. On a large parallel machine, this must be a path/filename visible to both the client and server. If the CSlib is instantiated multiple times (see this section for use cases), the filename must be unique for each pair of client/server couplings.

For more discussion on how to specify the *mode* and *ptr* arguments when running client and server apps in different scenarios (on a desktop, on two different machines, on a cluster or supercomputer), see this section.

The *pworld* argument determines whether the app uses the CSlib in serial versus parallel. Use *pworld* = NULL for serial use. For parallel use, the app initializes MPI and runs within an MPI communicator. Typically this communicator is MPI_COMM_WORLD, however depending on how you intend to call and use the CSlib it could be a sub-communicator. The latter is always the case if *mode* = "mpi/one". See the important note below.

Pworld is passed to the CSlib as a pointer to the communicator the app is running on. Note that *pworld* is NOT the MPI communicator itself, rather a pointer to it. This allows it to be specified as NULL for serial use of the CSlib. Thus if MPI_COMM_WORLD is the communicator, the app can do something like this, as shown in the C++ code example above:

```
MPI_Comm world = MPI_COMM_WORLD;
CSlib *cs = new CSlib(csflag,"zmq",port,&world);
```

IMPORTANT NOTE: If *mode* = "mpi/one" then both the client and server app are running within the same MPI communicator which spans both the client and server processors, typically MPI_COMM_WORLD. That spanning communicator is passed as the *ptr* argument, as dicsussed above. In this case, *pworld* is the sub-communicator for the processors the client or server app is running on. To create the two sub-communicators, both apps must invoke MPI_Comm_split() before instantiating the CSlib. Note that the MPI_Comm_split() call is synchronous; all processors in BOTH the client and server apps must call it.

Example code for doing this in various langautes is given in this section.

When an app has completed its messaging, it should destroy its instance of the CSlib, freeing the memory it has allocated internally. The syntax for this operation is shown in the code examples at the top of this page.

When using *mode* = "mpi/one", the app should also free the sub-communicator it created, as in the code examples given in this section for splitting the MPI communicator.

CSlib documentation

Exchange messages

Sending a message occurs in two stages. An initial call to send() sets a message ID and field count. A field is zero or more datums of the same data type (int, double, etc). Then pack() methods are called, once for each field. The message is not sent to the other application until the last field has been packed.

Receiving a message also occurs in two stages. An initial call to recv() returns a message ID, field count, and info about each of the fields (id, data type, length) in the message. A field is zero or more datums of the same datat type (int, double, etc). Then unpack() methods are called, typically once for each field. Fields can be unpacked in any order, or not at all.

These are code examples for the various CSlib methods used to exchange messages. Code for the sending app and receiving app are shown together:

- Send and recv methods
- Pack and unpack a single value
- Pack and unpack a string
- Pack and unpack multiple datums
- Pack and unpack multiple datums in parallel
- Pack and unpack of multidimensional arrays

Additional details on the methods:

- Who owns the data memory
- Persistence of data
- No matching requirement for pack and unpack methods
- Synchronizing methods
- How pack_parallel() and unpack_parallel() organize their data
- Referencing a vector containing array data
- Data distributions for pack_parallel() and unpack_parallel()

Send and recv methods

API:

```
void send(int msgID, int nfield);
int recv(int &nfield, int *&fieldID, int *&fieldtype, int *&fieldlen);
```

C++ sending app:

```
int msgID = 1;
int nfield = 10;
cs->send(msgID,nfield);    // cs = object created by CSlib()
```

C++ receiving app:

```
int nfield;
int *fieldID,*fieldtype,*fieldlen;
int msgID = cs->recv(nfield,fieldID,fieldtype,fieldlen);
```

C sending app:

```
void *cs;        // ptr returned by cslib_open()
int msgID = 1;
int nfield = 10;
cslib_send(cs,msgID,nfield);
```

C receiving app:

```
void *cs;
```

```
int nfield;
int *fieldID,*fieldtype,*fieldlen;
int msgID = cslib_recv(cs,&nfield,&fieldID,&fieldtype,&fieldlen);
```

Fortran sending app:

```
type(c_ptr) :: cs      ! ptr returned by cslib_open()
integer :: msgID = 1
integer :: nfield = 10
call cslib_send(cs,msgID,nfield)
```

Fortran receiving app:

```
type(c_ptr) :: cs
integer :: msgID,nfield
type(c_ptr) :: ptr,pfieldID,pfieldtype,pfieldlen
integer(c_int), pointer :: fieldID(:) => null()
integer(c_int), pointer :: fieldtype(:) => null()
integer(c_int), pointer :: fieldlen(:) => null()

msgID = cslib_recv(cs,nfield,pfieldID,pfieldtype,pfieldlen)
call c_f_pointer(pfieldID,fieldID,[nfield])
call c_f_pointer(pfieldtype,fieldtype,[nfield])
call c_f_pointer(pfieldlen,fieldlen,[nfield])
```

Python sending app:

```
msgID = 1
nfield = 10
cs.send(msgID,nfield)     # cs = object created by CS.open()
```

Python receiving app:

```
msgID,nfield,fieldID,fieldtype,fieldlen = cs.recv()
```

Comments:

For the send() method, the *msgID* is an integer identifier chosen by the sender, so the receiver can interpret the message accordingly. It can be positive, negative, or zero. *Nfield* is the number of data fields in the message. Nfield can be >= 0.

Nfield is returned by recv(). FieldID, fieldtype, and fieldlen are vectors of length nfield. They contain the field ID, field data type, and field length for each field (explained below). These vectors are allocated by the CSlib, which just returns pointers to the vectors. The values in those vectors are those specified by the sender, in the pack() methods that follow.

For the C cslib_recv(), note that the address of nfield, fieldID, fieldtype, fieldlen are passed.

For the Fortran cslib_recv(), pfieldID, pfieldtype, pfieldlen are returned as pointers. The c_f_pointer() function converts them to Fortran vectors of length nfield.

Pack and unpack a single value

A single value is a 32-bit integer, 64-bit integer, 32-bit floating point value, or a 64-bit floating point value.

API:

```
void pack_int(int id, int value);
void pack_int64(int id, int64_t value);
void pack_float(int id, float value);
void pack_double(int id, double value);

int unpack_int(int id);
int64_t unpack_int64(int id);
float unpack_float(int id);
double unpack_double(int id);
```

C++ sending app:

```
int id = 1;          // different for each pack call
int oneint = 1;
int64_t oneint64 = 3000000000;
float onefloat = 1.5;
double onedouble = 1.5;

cs->pack_int(id,oneint);       // cs = object created by CSlib()
cs->pack_int64(id,oneint64);
cs->pack_float(id,onefloat);
cs->pack_double(id,onedouble);
```

C++ receiving app:

int id = fieldID[0]; // different for each unpack call

```
int oneint = cs->unpack_int(id);
int64_t oneint64 = cs->unpack_int64(id);
float onefloat = cs->unpack_float(id);
double onedouble = cs->unpack_double(id);
```

C sending app:

```
void *cs;        // ptr returned by cslib_open()
int id = 1;
int oneint = 1;
int64_t oneint64 = 3000000000;
float onefloat = 1.5;
double onedouble = 1.5;

cslib_pack_int(cs,id,oneint);
cslib_pack_int64(cs,id,oneint64);
cslib_pack_float(cs,id,onefloat);
cslib_pack_double(cs,id,onedouble);
```

C receiving app:

```
void *cs;
int id = fieldID[0];

int oneint = cslib_unpack_int(cs,id);
int64_t oneint64 = cslib_unpack_int64(cs,id);
```

```
float onefloat = cslib_unpack_float(cs,id);
double onedouble = cslib_unpack_double(cs,id);
```

Fortran sending app:

```
type(c_ptr) :: cs      ! ptr returned by cslib_open()
integer(4) :: id = 1
integer(4) :: oneint = 1
integer(8) :: oneint64 = 3000000000
real(4) :: onefloat = 1.5
real(8) :: onedouble = 1.5

call cslib_pack_int(cs,id,oneint)
call cslib_pack_int64(cs,id,oneint64)
call cslib_pack_float(cs,id,onefloat)
call cslib_pack_double(cs,id,onedouble)
```

Fortran receiving app:

```
type(c_ptr) :: cs
integer(4) :: oneint
integer(8) :: oneint64
real(4) :: onefloat
real(8) :: onedouble
integer(4) :: id = fieldID(1)

oneint = cslib_unpack_int(cs,id)
oneint64 = cslib_unpack_int64(cs,id)
onefloat = cslib_unpack_float(cs,id)
onedouble = cslib_unpack_double(cs,id)
```

Python sending app:

```
id = 1
oneint = 1
oneint64 = 3000000000
onefloat = 1.5
onedouble = 1.5

cs.pack_int(id,oneint)            # cs = object created by CS.open()
cs.pack_int64(id,oneint64)
cs.pack_float(id,onefloat)
cs.pack_double(id,onedouble)
```

Python receiving app:

```
id = fieldID[0]

oneint = cs.unpack_int(id)
oneint64 = cs.unpack_int64(id)
onefloat = cs.unpack_float(id)
onedouble = cs.unpack_double(id)
```

Comments:

For all the pack() methods, *id* is an identifier the sending app chooses for each field it packs. These can normally just be 1,2,3,...,N for N fields, but that is not required. Each id can be positive, negative, or zero. For the unpack() methods, the receiving app can access the list of field IDs from the fieldID vector returned by recv().

Pack and unpack a string

API:

```
void pack_string(int id, char *string);
char *unpack_string(int id);
```

C++ sending app:

```
txt = (char *) "hello world";
int id = 1;

cs->pack_string(id,txt);       // cs = object created by CSlib()
```

C++ receiving app:

```
int id = fieldID[0];

char *txt = cs->unpack_string(id);
```

C sending app:

```
void *cs;        // ptr returned by cslib_open()
int id = 1;
txt = (char *) "hello world";

cslib_pack_string(cs,id,txt);
```

C receiving app:

```
void *cs;
int id = fieldID[0];
char *txt = cslib_unpack_string(cs,id);
```

Fortran sending app:

```
type(c_ptr) :: cs      ! ptr returned by cslib_open()
character(len=32) :: txt = "hello world"
integer(4) :: id = 1

call cslib_pack_string(cs,id,trim(txt)//c_null_char)
```

Fortran receiving app:

```
type(c_ptr) :: cs
type(c_ptr) :: ptr
character(c_char), pointer :: txt(:) => null()
integer(4) :: id = fieldID(1)

ptr = cslib_unpack_string(cs,id)
call c_f_pointer(ptr,txt,[fieldlen(1)-1])
```

Python sending app:

```
txt = "hello world"
id = 1
```

```
cs.pack_string(id,txt)    # cs = object created by CS.open()
```

Python receiving app:

```
id = fieldID[0]

txt = cs.unpack_string(id)
```

Comments:

The memory for the string returned by unpack_string() is owned by the CSlib. It just returns a pointer to the app.

For Fortran, the trim() function removes trailing space. A c_null_char is appended to the pack string to make it a NULL-terminated C-style string. A pointer to the string is returned by cslib_unpack_string(). The c_f_pointer() function converts it to a Fortran style string of length N-1 to remove the trailing NULL.

Pack and unpack multiple datums

The pack() and unpack() methods are for exchanging multiple datums of the same data type, which can represent contiguous 1d vectors or multidimensional arrays (2d, 3d, etc) of datums. The set size can be one, i.e. a single value. Here we give examples for vectors. The syntax for declaring multidimensional arrays in the native langauges are different, so that is discussed below.

Use these methods if all an app's processors send/receive a copy of the entire field. Use the parallel versions (below) if the field data is distributed across the sending or receiving app processors.

The syntax for 64-bit floating point datums (float64) is shown here. The comments indicate what to change for 32-bit integers (int32), 64-bit integers (int64), or 32-bit floating point (float32).

API:

```
void pack(int id, int ftype, int flen, void *data);
void *unpack(int id);
```

C++ sending app:

```
int id = 1;
int ftype = 4;           // 1 for int32, 2 for int64 = 2, 3 for float32
int flen = 100;
double *sdata = (double *) malloc(flen*sizeof(double));  // double -> int for int32, int64_t for int

cs->pack(id,ftype,flen,sdata);
```

C++ receiving app:

```
int id = fieldID[0]

double *rdata = (double *) cs->unpack(id);     // double -> int for int32, int64_t for int64, float
```

C sending app:

```
void *cs;          // ptr returned by cslib_open()
int id = 1;
int ftype = 4;         // 1 for int32, 2 for int64 = 2, 3 for float32
```

```
int flen = 100;
double *sdata = (double *) malloc(flen*sizeof(double));     // double -> int for int32, int64_t for

cslib_pack(cs,id,ftype,flen,sdata);
```

C receiving app:

```
void *cs;
int id = fieldID[0];

double *rdata = (double *) cslib_unpack(cs,id);     // double -> int for int32, int64_t for int64, f
```

Fortran sending app:

```
type(c_ptr) :: cs      ! ptr returned by cslib_open()
integer(4) :: id = 1
integer(4) :: ftype = 4         ! 1 for int32, 2 for int64 = 2, 3 for float32
integer :: flen = 100
real(8), allocatable, target :: sdata(:)     ! real(8) -> integer(4) for int32, integer(8) for int64
allocate(sdata(flen))

call cslib_pack(cs,id,ftype,flen,c_loc(sdata))
```

Fortran receiving app:

```
type(c_ptr) :: cs
integer(4) :: id = fieldID(1)
integer(4) :: flen = fieldlen(1)
type(c_ptr) :: ptr
real(c_double), pointer :: rdouble(:) => null()      ! real(c_double) -> integer(c_int) for int32, i

ptr = cslib_unpack(cs,id)
call c_f_pointer(ptr,rdata,[flen])
```

Python sending app:

```
id = 3
ftype = 4      # 1 for int32, 2 for int64 = 2, 3 for float32
flen = 100
dtype = 1,2,3   # to send Python list, Numpy array, ctypes vector

if dtype == 1:                      # Python list
 sdata = flen*[0]
elif dtype == 2:                    # Numpy 1d array
 sdata = np.zeros(flen,np.float)    # float -> intc for int32, int64 for int64, float32 for float32
elif dtype == 3:                    # ctypes vector
 sdata = (ctypes.c_double * flen)()   # c_double -> c_int for int32, c_longlong for int64, c_float f

cs.pack(id,ftype,flen,sdata)    # cs = object created by CS.open()
```

Python receiving app:

```
id = fieldID[0]
dtype = 1,2,3     # to return Python list, Numpy 1d array, ctypes vector

rdata = cs.unpack(id,tflag=dtype) # tflag is optional, default = 3 for ctypes
```

Comments:

For pack(), *ftype* specifies the type of datums in the field:

- ftype = 1 = 32-bit integers
- ftype = 2 = 64-bit integers
- ftype = 3 = 32-bit floating point
- ftype = 4 = 64-bit floating point

For pack(), *flen* is the number of datums in the field. Flen can be 0 or more.

The final data argument for pack() is a pointer to contiguous chunk of memory that stores the *flen* datums. If *flen* = 0, data can be NULL. If *flen* = 1, and a scalar values is passed, then the argument should be the address of the scalar value, since the CSlib interprets it as a vector. Note that data can be passed as a pointer to any data type, (int, int64_t, float, double, etc). The library treats it as a void pointer and casts it to the appropriate type to match the ftype argument.

Pack() can be passed a pointer to any data type, (int, int64_t, float, double, etc). The CSlib treats it as a void pointer and casts it to the appropriate type to match the ftype value.

Unpack() returns a "void *" pointer to the field data. The app casts the pointer to the appropriate data type.

There is no allocation of memory by the app for calls to unpack(). The CSlib owns the memory for the data and just returns a pointer to the app.

For the Fortran unpack(), the function c_f_pointer() must be called to map the returned C pointer to a Fortran variable. The final argument of c_f_pointer is a "shape" argument with the length of the vector. This is known by the receiving app as a value in the fieldlen vector returned by recv().

For the Python pack(), data can be passed as a Python list (or tuple), a Numpy array, or a ctypes vector. Examples for how to create these data structures are shown above. All of these data types are accessed the same in subsequent Python code, e.g. rdata[13].

The Python unpack() takes a final optional argument tflag (for type flag). It can be specified as 1,2,3 to return the vector as a Python list, Numpy array, or ctypes vector. The default is tflag = 3 = ctypes vector.

Pack and unpack multiple datums in parallel

Similar to pack() and unpack(), the pack_parallel() and unpack_parallel() methods are for passing a contiguous list of datums which can represent 1d vectors or multidimensional arrays (2d, 3d, etc) of datums. Here we give examples for vectors. The syntax for declaring multidimensional arrays is different, so those are discussed below.

Use these methods if the field data is distributed across the sending or receiving app processors. Use the pack() and unpack() methods above if all an app's processors send/receive a copy of the entire field.

The syntax for 64-bit floating point datums (float64) is shown here. The comments indicate what to change for 32-bit integers (int32), 64-bit integers (int64), or 32-bit floating point (float32).

API:

```
void pack_parallel(int id, int ftype, int nlocal, int *ids, int nper, void *data);
void unpack_parallel(int id, int nlocal, int *ids, int nper, void *data);
```

C++ sending app:

```
int id = 1;
int ftype = 4;           // 1 for int32, 2 for int64 = 2, 3 for float32
int nlocal = 100;        // could be different for each proc
int nper = 1;            // for a vector (see multidimensional arrays below)
int *ids = (int *) malloc(nlocal*sizeof(int));  // this proc's datum IDs
double *sdata = (double *) malloc(nlocal*sizeof(double));    // double -> int for int32, int64_t for

cs->pack_parallel(id,ftype,nlocal,ids,nper,sdata);
```

C++ receiving app:

```
int id = fieldID[0];
int nlocal = 200;
int nper = 1;
int *ids = (int *) malloc(nlocal*sizeof(int));
double *rdata = (float *) malloc(nlocal*sizeof(double));    // double -> int for int32, int64_t for

cs->unpack_parallel(id,nlocal,ids,nper,rdata);
```

C sending app:

```
void *cs;        // ptr returned by cslib_open()
int id = 1;
int ftype = 4;    // 1 for int32, 2 for int64 = 2, 3 for float32
int nlocal = 100;
int nper = 1;
int *ids = (int *) malloc(nlocal*sizeof(int));
double *sdata = (double *) malloc(nlocal*sizeof(double));     // double -> int for int32, int64_t fo

cs->pack_parallel(id,ftype,nlocal,ids,nper,sdata);
```

C receiving app:

```
void *cs;
int id = fieldID[0];
int nlocal = 200;
int nper = 1;
int *ids = (int *) malloc(nlocal*sizeof(int));
double *rdata = (float *) malloc(nlocal*sizeof(double));     // double -> int for int32, int64_t for

cslib_unpack_parallel(cs,id,nlocal,ids,nper,rdata);
```

Fortran sending app:

```
type(c_ptr) :: cs      ! ptr returned by cslib_open()
integer(4) :: id = 1
integer(4) :: ftype = 4      ! 1 for int32, 2 for int64 = 2, 3 for float32
integer :: nlocal = 100
integer :: nper = 1
integer(4), allocatable, target :: ids(:)
real(8), allocatable, target :: sdata(:)      ! real(8) -> integer(4) for int32, integer(8) for int6
allocate(ids(nlocal))
allocate(sdata(nlocal))

call cslib_pack_parallel(cs,id,ftype,nlocal,c_loc(ids),1,c_loc(sdata))
```

Fortran receiving app:

```
type(c_ptr) :: cs
integer(4) :: id = fieldID(1)
integer :: nlocal = 200
integer :: nper = 1
integer(4), allocatable, target :: ids(:)
real(8), allocatable, target :: rdata(:)     ! real(8) -> integer(4) for int32, integer(8) for int64
allocate(ids(nlocal))
allocate(rdata(nlocal))

cslib_unpack_parallel(cs,id,nlocal,c_loc(ids),nper,c_loc(rdata))
```

Python sending app:

```
id = 1
ftype = 4         # 1 for int32, 2 for int64 = 2, 3 for float32
nlocal = 100
nper = 1
dtype = 1,2,3   # to send Python list, Numpy array, ctypes vector

ids = nlocal*[0]

if dtype == 1:                      # Python list
 sdata = nlocal*[0]
elif dtype == 2:                    # Numpy 1d array
 sdata = np.zeros(nlocal,np.float)   # float -> intc for int32, int64 for int64, float32 for float32
elif dtype == 3:                    # ctypes vector
 sdata = (ctypes.c_double * nlocal)()    # c_double -> c_int for int32, c_longlong for int64, c_floa

cs.pack_parallel(id,ftype,nlocal,ids,nper,sdata)
```

Python receiving app:

```
id = fieldID[0]
nlocal = 200
nper = 1
dtype = 1,2,3     # to return Python list, Numpy array, ctypes vector
ids = nlocal*[0]

rdata = cs.unpack_parallel(id,nlocal,ids,nper,tflag=dtype)    # tflag is optional, default = 3 for c
```

Comments:

For these parallel methods, *nlocal* is the number of elements of global data each processor owns, where an "element" is *nper* datums. See the next section on multidimensional arrays for code examples where nper > 1.

Each element has a unique global ID from 1 to Nglobal, where Nglobal = the sum of *nlocal* across all processors. *Ids* is a 1d integer vector of length *nlocal* containting the subset of global IDs the processor owns. See code examples at the end of this page for how processors might partition the data and initialize their *nloca* and *ids*.

The data argument for pack_parallel() and unpack_parallel() is a pointer to a contiguous 1d vector. In this case, it stores the nper*Nlocal datums owned by the processor. The first *nper* values are the datums for the first element in the *ids* vector, the next Nper values are the datums for the second element, etc.

Unlike for unpack(), the memory for the field data returned by unpack_parallel() is stored by the app; it is not stored internal to the CSlib. This means the app must allocate the memory before calling unpack_parallel().

For the Python code, see the comments above in this section for the use of dtype and Python lists, Numpy arrays,

ctypes vectors.

Pack and unpack of multidimensional arrays

Multidimensional arrays (2d,3d,etc) can be passed as field data using the same pack(), unpack(), pack_parallel(), unpack_parallel() described above. But ONLY if their numeric values are stored in contiguous memory. We illustrate in this section how to declare and use such arrays with the 4 pack/unpack methods.

For C/C++, a malloc2d() function is used here to allocate contiguous memory for the array data. Code for this function is given in this section. It's just an example of how to create appropriate multidimensional arrays for use with the CSlib. It's not needed for Fortran or Python, because their multidimensional arrays use contiguous memory.

Note that if one of the client or server apps is written in Fortran, but the other is not, and you send an array of data from one to the other, you should insure data is ordered consistently in the two arrays. This issue is also discussed in this section.

The syntax for 64-bit floating point datums (float64) is shown here. The comments indicate what to change for 32-bit integers (int32), 64-bit integers (int64), or 32-bit floating point (float32). It should also be clear how to adapt the syntax below for 3d, 4d, etc arrays.

API:

```
void pack(int id, int ftype, int flen, void *data);
void pack_parallel(int id, int ftype, int nlocal, int *ids, int nper, void *data);

void *unpack(int id);
void unpack_parallel(int id, int nlocal, int *ids, int nper, void *data);
```

C++ sending app:

```
int id = 1;
int ftype = 4;       // 1 for int32, 2 for int64 = 2, 3 for float32
int flen = 1000;
int nlocal = 100;
int nper = 3;
int *ids = (int *) malloc(nlocal*sizeof(int));
double **sarray = (double **) malloc2d(flen,nper); // double -> int for int32, int64_t for int64, fl
double **sarraylocal = (double **) malloc2d(nlocal,nper);
```

cs->pack(id,ftype,nper*flen,&sarray[0][0]); // note nper*flen
cs->pack_parallel(id,ftype,nlocal,ids,nper,&sarraylocal[0][0]);

C++ receiving app:

```
int id = fieldID[0];
int nlocal = 200;
int nper = 3;
int flen = fieldlen[0] / nper;    // fieldlen = nper*flen
int *ids = (int *) malloc(nlocal*sizeof(int));
double **rarray = (double **) malloc2d(flen,nper);  // double -> int for int32, int64_t for int64, f
double **rarraylocal = (double **) malloc2d(nlocal,nper);

double *rvec = (double *) cs->unpack(id);  // double -> int for int32, int64_t for int64, float for
memcpy(&rarray[0][0],rvec,nper*flen*sizeof(double));  // copy into 2d array if desired
```

```
cs->unpack_parallel(id,nlocal,ids,nper,&rarraylocal[0][0]);
```

C sending app:

```
void *cs;        // ptr returned by cslib_open()
int id = 1;
int ftype = 4;      // 1 for int32, 2 for int64 = 2, 3 for float32
int flen = 1000;
int nlocal = 100;
int nper = 3;
int *ids = (int *) malloc(nlocal*sizeof(int));
double **sarray = (double **) malloc2d(flen,nper); // double -> int for int32, int64_t for int64, fl
double **sarraylocal = (double **) malloc2d(nlocal,nper);
```

cslib_pack(cs,id,ftype,nper*flen,&sarray[0][0]);
cslib_pack_parallel(cs,id,ftype,nlocal,ids,nper,&sarraylocal[0][0]);

C receiving app:

```
void *cs;
int id = fieldID[0];
int nlocal = 200;
int nper = 3;
int flen = fieldlen[5] / nper;
int *ids = (int *) malloc(nlocal*sizeof(int));
double **rarray = (double **) malloc2d(flen,3);  // double -> int for int32, int64_t for int64, floa
double **rarraylocal = (double **) malloc2d(nlocal,3);

double *rvec = (double *) cslib_unpack(cs,id);  // double -> int for int32, int64_t for int64, float
memcpy(&rarray[0][0],rvec,nper*flen*sizeof(double));
cslib_unpack_parallel(cs,id,nlocal,ids,nper,&rarraylocal[0][0]);
```

Fortran sending app:

```
type(c_ptr) :: cs      ! ptr returned by cslib_open()
integer(4) :: id = 1
integer(4) :: ftype = 4      ! 1 for int32, 2 for int64 = 2, 3 for float32
integer(4) :: flen = 1000
integer :: nlocal = 100
integer :: nper = 3
integer(4), allocatable, target :: ids(:)
real(8), allocatable, target :: sarray(:,:),sarraylocal(:,:)  ! real(8) -> integer(4) for int32, int
allocate(ids(nlocal))
allocate(sarray(nper,flen))   ! note 3xN, not Nx3
allocate(sarraylocal(nper,nlocal))

call cslib_pack(cs,id,ftype,flen,c_loc(sarray))
call cslib_pack_parallel(cs,id,ftype,nlocal,c_loc(ids),nper,c_loc(sarraylocal))
```

Fortran receiving app:

```
type(c_ptr) :: cs
integer(4) :: id = fieldID(1)
integer :: nlocal = 200
integer :: nper = 3
integer(4) :: flen = fieldlen(1) / nper
integer(4), allocatable, target :: ids(:)
type(c_ptr) :: ptr
real(c_double), pointer :: rarray(:) => null()  ! real(c_double) -> integer(c_int) for int32, intege
real(8), allocatable, target :: rarraylocal(:,:)   ! real(8) -> integer(4) for int32, integer(8) for
allocate(ids(nlocal))
```

```
allocate(rarraylocal(nper,nlocal))

ptr = cslib_unpack(cs,id)
call c_f_pointer(ptr,rarray,[nper,flen])
call cslib_unpack_parallel(cs,id,nlocal,c_loc(ids),nper,c_loc(rarraylocal))
```

Python sending app:

```
id = 1
ftype = 4             # 1 for int32, 2 for int64 = 2, 3 for float32
flen = 1000
nlocal = 100
nper = 1
dtype = 1,2,3   # to send Python list, Numpy array, ctypes vector

ids = nlocal*[0]

if dtype == 1:                      # Python list of lists
 sarray = [nper*[0] for i in range(flen)]
 sarraylocal = [nper*[0] for i in range(nlocal)]
elif dtype == 2:                    # Numpy 2d arrays
 sarray = np.zeros((flen,nper),np.float)   # float -> intc for int32, int64 for int64, float32 for f
 sarraylocal = np.zeros((nlocal,nper),np.float)
elif dtype == 3:                    # ctypes 2d arrays
 sarray = ((ctypes.c_double * nper) * flen)()  # c_double -> c_int for int32, c_longlong for int64,
 sarraylocal = ((ctypes.c_double * nper) * nlocal)()

cs.pack(id,ftype,flen,sarray)
cs.pack_parallel(id,ftype,nlocal,ids,nper,sarraylocal)
```

Python receiving app:

```
id = fieldID[0]
nlocal = 200
nper = 1
flen = fieldlen[0] / nper
dtype = 1,2,3     # to create Python list, Numpy array, ctypes vector from returned data
ids = nlocal*[0]

rvec = cs.unpack(id,tflag=dtype)   # tflag is optional, default = 3 for ctypes
rveclocal = cs.unpack_parallel(id,nlocal,ids,nper,tflag=dtype)
if dtype == 1:                 # Python list of lists
 rarray = [rvec[i:i+nper] for i in range(0,len(rvec),nper)]
 rarraylocal = [rveclocal[i:i+nper] for i in range(0,len(rveclocal),nper)]
elif dtype == 2:               # Numpy arrays
 rarray = np.reshape(rvec,(flen,nper))
 rarraylocal = np.reshape(rveclocal,(nlocal,nper))
```

Comments:

Which return vecs (not arrays), which require data copies.

For Python, when unpacking to a Python list (dtype=1), a 1d Python list is returned by unpack() and unpack_parallel(). The lists can be restructured into a 2d list of lists, as shown above. This requires a data copy (as did the creation of the 1d Python list).

For Python, when unpacking to a Numpy array (dtype=2), a 1d Numpy vector is returned by unpack() and unpack_parallel() since the CSlib does not know the dimensionality of the array desired by the caller. But the Numpy reshape() function can then be used to morph the data into a multidimensional array (without a copy).

For Python, when unpacking to ctypes (dtype=3), a 1d ctypes vector is returned by unpack() and unpack_parallel().

TODO: should show how to convert (copy?) ctypes vec to ctypes array

---

Who owns the data memory

In this context, "owns" means allocates memory to store the data and later frees it.

For all the pack methods used when sending a message, the app owns the data memory.

For the recv() method, the CSlib owns the 3 returned field vectors: fieldID, fieldtype, fieldlen.

For the unpack() and unpack_string() methods, the CSlib owns the data. Only a pointer is returned to the app.

For the unpack_parallel() method, the app owns the data. It allocates the data before calling unpack_parallel(). The CSlib fills in values for the datums owned by each processor.

All of these rules hold for C/C++ and Fortran apps. In Fortran, data owned by the CSlib is not copied into Fortran vectors, arrays, strings in the code examples above. Instead, the variables are declared as Fortran pointers, and they point to the memory allocated inside the CSlib.

For Python apps, the rules are sightly different, because of the support for different Python data types (Python lists, Numpy arrays, ctypes vectors). A returned string is copied into a Python string, owned by the app, If data owned by the CSlib is returned as a Python list, it is copied and the app owns the list. If data owned by the CSlib is returned as a Numpy array or ctypes vector, no copy is performed. The Numpy or ctypes object points to memory allocated inside the CSlib. If new Python data structures are created after the unpack(), e.g.

Note that in an efficiency sense, all of this means that data received as a message by the CSlib is not copied when invoking methods where the CSlib owns the data. For methods where the app owns the data, a copy is performed.

Persistence of data

IMPORTANT NOTE: The CSlib reuses its internal data buffers for every send and receive. Thus if the CSlib owns the memory for data that is received/unpacked by the app (as discussed in the previous sub-section), the pointers returned to the app will NOT be valid once the app issues the next send() and pack() calls. The app MUST make its own copy of the data if persistence is required. This is true even if the app simply wishes to in-place modify a vector of received field data to send it back to the other app. To do this, the app should make a copy of the data, modify the copy, then send the copy.

As explained in the previous sub-section, this applies to the field vectors returned by recv(), the string returned by unpack_string(), and the data returned by unpack(). It does not apply to the data returned by unpack_parallel(), because the app owns that memory.

No matching requirement for pack and unpack methods

All the code examples above show the sending and receiving app using matching pack() and unpack() methods for the same field data. However, in general that is not required.

Say an app sends a field with a single integer via pack_int(). The receiving app can receive that field data via unpack_int(). But it could also use unpack(), as follows in C++:

int id = 1; int *vec = (int *) cs->unpack(id); // length of vec will be 1 value = vec[0];

Or a parallel app could even use unpack_parallel() as follows, assuming only one proc sets nlocal = 1 and ids.

```
int id = 1;
int nlocal = 1;          // other procs set nlocal = 0
int nper = 1;
int *ids = NULL;
int *rvec = NULL;
if (nlocal) {
  ids = (int *) malloc(nlocal*sizeof(int));
  ids[0] = 1;
  rvec = (int *) malloc(nlocal*sizeof(double));
}
cs->unpack_parallel(id,nlocal,ids,nper,rvec);
value = rvec[0];
```

Similarly, one app can use pack() to send a field, and the other app could use unpack_parallel() to receive the field. Or vice versa. This is because the internal message format within the CSlib is independent of how the message was packed. This section below gives details on how the field data is organized for pack_parallel() and unpack_parallel().

Also note, that if both the sending and receiving app are parallel, and use pack_parallel() and unpack_parallel() for the same field, there is no requirement that the two apps distribute the field data in the same way. In fact two apps may be running on different numbers of processors so identical distributions are not even possible.

Synchronizing methods

In this context, synchronizing refers to how a parallel app uses the CSlib. If a CSlib method is synchronizing it means all processors in a parallel app must call it at the same time (or the app will pause until all its processors make the call). Non-synchronizing calls do not need to be called at the same time (or at all) by individual processors.

The recv() call is synchronizing. All processors must call it together because the returned data is communicated to all of them.

None of the unpack() calls are synchronizing.

None of the pack() calls are synchronizing, except for pack_parallel(). All processors must call it together since they will communicate to gather data for the message.

The send() call is not technically synchronizing in the following sense. The normal way to use the send() and pack() methods to send a message from a parallel app is to have all processors call the same methods. However, if pack_parallel() is not used to pack any of the fields for a particular message it is OK for the app to only invoke the send() and pack() calls from its processor 0. However, there is no harm in having all processors invoke the send() and pack() calls, assuming they all have a copy of the send data.

How pack_parallel() and unpack_parallel() organize their data

When using pack_parallel(), the field data sent to the other app will be the union of the data portions from all the processors. Thus if Nglobal = sum of Nlocal, it will be nper*Nglobal in length. The library uses the *ids* vectors containing sets of unique global IDs from each of the processors to order the data elements from 1 to Nglobal in the send message. As discussed above, the receiving app processors can access the field data in one of two ways. Either via the unpack() method, which gives each processor access to the entire globally ordered vector with all

nper*Nglobal datums. Or via the unpack_parallel() method, which fills a local vector provided by the receiving app with nper*Nlocal datums for the elements corresponding to the *ids* vector provided by each receiving processor.

This means that for a serial app, a pack_parallel() call sends the same volume of data as a pack() call, but the data that is sent will be reordered from 1 to Nglobal using the list of sending *ids*. Likewise for a serial app, an unpack_parallel() call receives the same volume of data as an unpack() call, except that the data will be owned by the app (not the CSlib) and will be ordered from 1 to Nglobal using the list of receiving *ids*.

Referencing a vector containing array data

In some of the code examples above for multidimensional arrays, data is returned from the CSlib via an unpack methods as a vector. As shown, it can be copied into a 2d array for the app to use. However, to avoid the copy, you can also simply index the contiguous vector values using I,J indices with the appropriate 1d offset.

For example, assume the 1d vector contains Nx*Ny values representing a 2d Nx by Ny array. The vector values have a C-style ordering where the 1st row of values in the array appear first in the vector and are followed by the 2nd row, etc. Then in C syntax:

```
array[i][j] = vector[i*ny + j];
```

In Fortran syntax, the vector would be indexed the same but array would be declared and indexed as Ny by Nx:

```
real(8) :: array(ny,nx)
array(j,i) = vector(i*ny + j)
```

Distributing data for pack_parallel() and unpack_parallel()

These 2 methods allow data to be sent or received which is distributed across the processors of a parallel app.

As a concrete example, say that the xyz coordinates of 1000 particles are spread across 4 processors with global IDs from 1 to Nglobal = 1000. Each processor owns a subset of the particles, indexed by its *ids* vector. One proc has Nlocal = 230 particles (any subset of the global IDs), another has 240, another 250, and another 280. The coords are stored locally in a 2d Nlocal x 3 array of doubles (ftype = 4), so they can be accessed as coords[i][j], where I = 0 to nlocal-1, J = 0,1,2. Thus nper = 3. Each processor could then call pack_parallel() as follows in C++. This assumes the data within the 2d coords array is stored contiguously:

```
cs->pack_parallel(1,4,nlocal,ids,3,&coords[0][0]);
```

The following code examples in different languages illustrate another way a parallel app could distribute the elements of a vector of length Nglobal with global IDs from 1 to Nglobal. The vector elements are split evenly across the Nprocs processors in strided fashion. Nlocal = # of datums owned by processor *me*. *Ids* = list of Nlocal global IDs owned by processor *me*.

C++ or C:

```
MPI_Init(&argc,&argv);
MPI_Comm world = MPI_COMM_WORLD;
int me,nprocs;
MPI_Comm_rank(world,&me);
MPI_Comm_size(world,&nprocs);

int nglobal = 1000;
int nlocal = nglobal/nprocs;
```

```
if (me <nglobal % nprocs) nlocal++;

int *ids = (int *) malloc(nlocal*sizeof(int));
int m = 0;
for (int i = me; i <nglobal; i += nprocs)
 ids[m++] = i+1;
```

Fortran:

```
Integer :: world,me,nprocs,ierr,nglobal,nlocal
integer(4), allocatable, target :: ids(:)

call MPI_Init(ierr)
world = MPI_COMM_WORLD
call MPI_Comm_rank(world,me,ierr)
call MPI_Comm_size(world,nprocs,ierr)

nglobal = 1000
nlocal = nglobal/nprocs
if (me <mod(nglobal,nprocs)) nlocal = nlocal + 1

allocate(ids(nlocal))
m = 1
do i = me+1,nglobal,nprocs
 ids(m) = i
 m = m + 1
enddo
```

Python:

```
from mpi4py import MPI
world = MPI.COMM_WORLD
me = world.rank
nprocs = world.size

nglobal = 1000
nlocal = nglobal/nprocs
if me <nglobal % nprocs: nlocal += 1

ids = nlocal*[0]
m = 0
for i in range(me,nglobal,nprocs):
 ids[m] = i+1
 m += 1
```

CSlib documentation

Query message statistics

The CSlib stores a running count of how many messages the app has sent and received since the library was instantiated. These two values can be queried by the app whenever desired.

TODO: maybe a cummulative count of message sizes as bigint should be added

API:

```
int extract(int flag);
```

C++:

CSlib *cs = CSlib(...);

```
int nsend = cs->extract(1);
int nrecv = cs->extract(2);
```

or more simply:

```
int nsend = cs->nsend;
int nrecv = cs->nrecv;
```

C:

```
void *cs;
cslib_open(...,&cs)

int nsend = cslib_extract(cs,1);
int nrecv = cslib_extract(cs,2);
```

Fortran:

```
type(c_ptr) : cs
call cslib_open_fortran(...,cs)

integer :: nsend,nrecv
nsend = cslib_extract(cs,1)
nrecv = cslib_extract(cs,2)
```

Python:

```
import cslib as CS
cs = CS.open(...)

nsend = cs.extract(1)
nrecv = cs.extract(2)
```

CSlib documentation

More Python details

The CSlib can be used from a client or server app written in Python using a provided Python wrapper on the C interface to the CSlib. Currently only Python 2.7 is supported. At some point we'll make a Python 3 compatible version of the wrapper.

This section describes a few additional requirements when using Python as well as a few options Python enables:

Building the CSlib for use from Python Accessing the CSlib from Python at runtime Parallel Python via mpi4py Sending, receiving Python lists, Numpy arrays, ctypes vectors

---

Building the CSlib for use from Python"

A Python script loads the CSlib at runtime, which requires the CSlib to be built as a shared library, either for parallel or serial use. This can be done from the src directory by typing "make shlib". See this section for more details and options.

Loading the CSlib from Python at runtime

The Python interface to the CSlib is a single file src/cslib.py that implements a CSlib Python class. An instance of the class can be created in a Python app as follows, where the arguments to open() are explained in this section:

```
from CSlib import CS
cs = CSlib.open(...)
```

If either of these lines gives an error, take the following steps, and try again.

The error is likely because Python cannot find src/cslib.py and a shared library build of the CSlib (libcs.so or libcsnompi.so), which is also in the src dir.

The simplest way to fix this, is to define (or extend) two environment variables. If you do this in your shell start-up script you only need to do it once. These examples use my local path for the CSlib; alter for your system accordingly:

For bash, type these lines or add them to ~/.bashrc:

```
export PYTHON_PATH="$PYTHON_PATH:/home/sjplimp/cslib/src"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/sjplimp/cslib/src"
```

For csh or tcsh, type these lines or add them to ~/.cshrc:

```
setenv PYTHONPATH $PYTHONPATH:/home/sjplimp/cslib/src
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/sjplimp/cslib/src
```

After editing, do this in an already-existing terminal window, so the path changes take effect:

```
% source .bashrc
% source .cshrc
```

Alternatively, you can extend your PYTHON_PATH and LD_LIBRARY_PATH when you run a client or server Python script, before you import and instantiate the CSlib:

```
% python
>>> import sys,os
>>> sys.path.append("/home/sjplimp/cslib/src")   # path = PYTHON_PATY
>>> sys.path                # to verify PYTHON_PATH has been extended
>>> os.environ("LD_LIBRARY_PATH") += ":home/sjplimp/cslib/src"
>>> os.environ("LD_LIBRARY_PATH") # to verify LD_LIBRARY_PATH has been extended
```

Alternatively, you can copy the files (cslib.py and libcs*.so) to someplace your current PYTHON_PATH and LD_LIBRARY_PATH variables already point to.

"Parallel python via mpi4py

For an app (Python or not) to use the CSlib in parallel, it must pass an MPI communicator to the CSlib. Such an app also typically makes MPI calls in the app itself to work in parallel. To do both of these in Python, use the mpi4py Python package.

To see if mpi4y is already part of your Python type this line:

>>> from mpi4py import MPI

If it works you should be able to run this test.py script like this: mpirun -np 4 python test.py

```
from mpi4py import MPI
world = MPI.COMM_WORLD
me = world.rank
nprocs = world.size
print "Me %d Nprocs %d" % (me,nprocs)
```

and get 4 lines of output "Me N Nprocs 4", where N = 0,1,2,3.

If the import fails, you need to install mpi4py in your Python. Here are two ways to do it.

If you are using anaconda for your Python package management, simply type the following which will download and install it:

```
conda install mpi4py
```

If not, you can download mpi4py from its web site http://www.mpi4py.org, unpack it, and install it via pip:

```
pip install mpi4py
```

Once installed, the test.py example script above should run.

IMPORTANT NOTE: When mpi4py is installed in your Python, it compiles an MPI library (e.g. inside conda) or uses a pre-existing MPI library it finds on your system. This MUST be the same MPI library that the CSlib and the client and server apps are built with and link to, i.e. if other apps are not written in Python. If they do not match, you will typically get run-time MPI errors when your client and server apps run.

You can inspect the path to the MPI library that mpi4py uses like this:

```
% python
>>> import mpi4py
>>> mpi4py.get_config()
```

Sending, receiving Python lists, Numpy arrays, ctypes vectors

Python has several ways to represent a 1d vector or multi-dimensional array (2d, 3d, etc) of numeric values. Once created, all of them can be accessed with the same syntax, e.g. sdouble[500] = 3.4 for the 1d vector, or value = sarray[500][2] for the 2d array.

Python list or tuple:

nlen = 1000 sint = (1,4,3,8,6) # 5-length tuple (values cannot be changed) sdouble = nlen*[0.0] # 1000-length 1d vector sarray = [3*[0] for i in range(nlen)] # 1000x3 2d array (list of lists)

Numpy arrays:

import numpy as np nlen = 1000 sdouble = np.zeros(nlen,np.float) # 1000-length vector sarray = np.zeros((nlen,3),np.float) # 1000x3 2d array

ctypes vectors or arrays:

import ctypes sdouble = (nlen * ctypes.c_double)() # 1000-length vector sarray = (nlen * (3 * ctypes.c_double))() # 1000x3 2d array

The Numpy arrays and ctypes vectors store numeric data internally as contiguous chunks of memory, which Python lists or tuples do not. For numeric operations in Python the Numpy arrays are typically most efficient, since you can call Numpy functions which operate on the vectors or arrays with C code. The Numpy arrays and ctypes vectors have the additional advantage that when you send and receive them with the CSlib, their data does not need to be copied into C-compatible data structures.

Data in all 3 of these formats, including a Python tuple, can be sent as a field in a message via the pack() methods described in this section. The cslib.py wrapper detects which format the data is in, and converts it via ctypes to a C-style vector in order to call the C interface to the CSlib. For a Python list or tuple this requires coping the data into a ctypes vector before calling the library. For the Numpy and ctypes formats, no copy is needed.

When a field is unpacked from a received message via the unpack() methods described in this section, you can choose to have the data returned to the Python app in any of the 3 formats. This is done by using an optional final argument "tflag" to the unpack() and unpack_parallel() methods.

- tflag = 1 = Python list
- tflag = 2 = Numpy 1d vector
- tflag = 3 = ctypes 1d vector (default)

For tflag=1, this requires coping the data into a Python list. For the Numpy and ctypes formats, no copy is needed.

CSlib documentation

Error messages

The CSlib will print an error message and halt when an error condition is detected. Here is more info about these errors. The first field of the messages is which CSlib method triggered the error.

```
constructor(): CSlib invoked with MPI_Comm but built w/out MPI support
constructor(): CSlib invoked w/out MPI_Comm but built with MPI support
```

A serial app (which passes a NULL to the constructor of the CSlib) must be linked with a serial version of the CSlib. Likewise a parallel app (which passes an MPI communicator to the constructor of the CSlib) must be linked with a parallel version of the CSlib. See this section for how to build the CSlib for serial or parallel use.

```
constructor(): No mpi/one mode for serial lib usage
constructor(): No mpi/two mode for serial lib usage
```

If the CSlib is being used in serial, the "mpi/one" and "mpi/two" modes of messaging cannot be used.

```
constructor(): Invalid client/server arg
constructor(): Unknown mode
```

The csflag or mode args to the CSlib constructor are incorrect.

```
constructor(): Library not built with ZMQ support
constructor(): Server could not make socket connect
```

These are possible errors when mode = "zmq" is used. The first error means the socket mode (zmq) of messaging cannot be requested if the CSlib was built without the ZeroMQ (ZMQ) library. See this section for more info. The second error occurs is the ZMQ library is unable to connect the socket between the client and server apps. Check that the port string was specified correctly for both apps.

```
send(): Invalid nfield
send(): Message header size exceeds 32-bit integer limit
send(): Could not open send message file
```

Nfield must be >= 0. If it is too large (100s of millions), the 2nd error can occur. The 3rd error occurs when mode = "file" and a message file cannot be opened. Check the syntax of the path/file that was specified correctly and that the apps have read/write access to the file.

```
pack(): Reuse of field ID
pack(): Invalid ftype
pack(): Invalid flen
```

The same field ID cannot be packed twice. Valid ftypes are 1,2,3,4. Flen must be >= 0.

```
pack_parallel(): Reuse of field ID
pack_parallel(): Invalid ftype
pack_parallel(): Invalid nlocal
pack_parallel(): Invalid nper
```

The same field ID cannot be packed twice. Valid ftypes are currently 1,2,3,4. Nlocal must be >= 0. Nper must be >= 1.

```
pack(): Message size exceeds 32-bit integer limit
```

The size of the entire send message buffer must be <= 2^31 bytes, which is around 2 billion bytes.

```
recv(): Could not open recv message file
```

The error occurs when mode = "file" and a message file cannot be opened. Check the syntax of the path/file that was specified correctly and that the apps have read/write access to the file.

```
unpack_int(): Unknown field ID        # ditto for int64, float, double, string
unpack_int(): Mis-match of ftype     # ditto for int64, float, double, string
unpack_int(): Flen is not 1                # ditto for int64, float, double, but not string
```

The received message does not conatin the requested field ID. The requested data type (int, int64, float, double, string) does not match the data type of the sent field. The length of the sent field must be one to use these unpack methods. Except for unpack_string() which can retrieve a string of any length.

```
unpack(): Unknown field ID
```

The received message does not conatin the requested field ID.

```
unpack_parallel(): Unknown field ID
unpack_parallel(): Invalid nlocal
unpack_parallel(): Invalid nper
```

The received message does not conatin the requested field ID. Nlocal must be >= 0. Nper must be >= 1.

`extract(): Invalid flag`

Valid flags are currently 1 or 2.

---

malloc(): Failed to allocate N bytes realloc(): Failed to reallocate N bytes

These are interal errors within the CSlib. A memory allocation or reallocate of N bytes failed. This could be because the allocation was too large, or because the system does not have sufficient memory.