TclMPI - Tcl Bindings for MPI

0.6

Generated by Doxygen 1.7.1

Fri May 18 2012 22:54:50

Contents

Chapter 1

TclMPI User's Guide

This page describes Tcl bindings for MPI. This package provides a shared object that can be loaded into a Tcl interpreter to provide additional commands that act as an interface to an underlying MPI implementation. This allows to run Tcl scripts in parallel via mpirun or mpiexec similar to C, C++ or Fortran programs and communicate via wrappers to MPI function call.

The original motivation for writing this package was to complement a Tcl wrapper for the LAMMPS molecular dynamics simulation software, but also allow using the VMD molecular visualization and analysis package in parallel without having to recompile VMD and using a convenient API to people that already know how to program parallel programs with MPI in C, C++ or Fortran.

1.1 Compilation and Installation

The package currently consist of a single C source file which needs to be compiled for dynamic linkage. The corresponding commands for Linux and MacOSX systems are included in the provided makefile. All that is required to compile the package is an installed Tcl development system and a working MPI installation. Since this creates a dynamically loaded shared object (DSO), both Tcl and MPI have to be compiled and linked as shared libraries (this is the default for Tcl and OpenMPI on Linux, but your mileage may vary). As of May 15 2012 the code has been tested only on 32-bit and 64-bit x86 Linux platforms with OpenMPI.

To compile the package adjust the settings in the Makefile according to your platform, MPI and Tcl installation. For most Linux distributions, this requires installing not only an MPI and Tcl package, but also the corresponding development packages, e.g. on Fedora you need openmpi, openmpi-devel, tcl, and tcl-devel and their dependencies. Then type make to compile the tclmpi.so file. With make check you can run the integrated unittest package to see, if everything is working as expected.

To install you can create a directory, e.g. /usr/local/libexec/tclmpi, and copy the files tclmpi.so and pkgIndex.tcl into it. If you then use the command set auto_path [concat /usr/local/libexec/tclmpi $auto_path] in your .tclshrc or .vmdrc, you can load the tclmpi wrappers on demand simply by using the command package require tclmpi.

1.2 Software Development and Bug Reports

The TclMPI code is maintained using git for source code management, and the project is hosted on github at `https://github.com/akohlmey/tclmpi` From there you can download snapshots of the development and releases, clode the repository to follow development, or work on your own branch through forking it. Bug reports and feature requests should also be filed on github at through the issue tracker at: `https://github.com/akohlmey/tclmpi/issues.`

Chapter 2

TclMPI Developer's Guide

This document explains the implementation of the Tcl bindings for MPI implemented in TclMPI. The following sections will document how and which MPI is mapped to Tcl and what design choices were made.

2.1 Overall Design and Differences to the MPI C-bindings

To be consistent with typical Tcl conventions all commands and constants in lower case and prefixed with tclmpi::, so that clashes with existing programs are reduced. This is not yet set up to be a proper namespace, but that may happen at a later point, if the need arises. The overall philosophy of the bindings is to make the API similar to the MPI one (e.g. maintain the order of arguments), but don't stick to it slavishly and do things the Tcl way wherever justified. Convenience and simplicity take precedence over performance. If performance matters that much, one would write the entire code C/C++ or Fortran and not Tcl. The biggest visible change is that for sending data around, receive buffers will be automatically set up to handle the entire message. Thus the typical "count" arguments of the C/C++ or Fortran bindings for MPI is not required, and the received data will be the return value of the corresponding command. This is consistent with the automatic memory management in Tcl, but this convenience and consistency will affect performance and the semantics. For example calls to tclmpi::bcast will be converted into ∗two∗ calls to MPI_Bcast(); the first will broadcast the size of the data set being sent (so that a sufficiently sized buffers can be allocated) and then the second call will finally send the data for real. Similarly, tclmpi::recv will be converted into calling MPI_Probe() and then MPI_Recv() for the purpose of determining the amount of temporary storage required. The second call will also use the MPI_SOURCE and MPI_TAG flags from the MPI_Status object created for MPI_Probe() to make certain, the correct data is received.

Things get even more complicated with with non-blocking receives. Since we need to know the size of the message to receive, a non-blocking receive can only be posted, if the corresponding send is already pending. This is being determined by calling MPI_Iprobe() and when this shows no (matching) pending message, the parameters for the receive will be cached and the then MPI_Probe() followed by MPI_Recv() will be called as part of tclmpi::wait. The blocking/non-blocking behavior of the Tcl script should be very close to the corresponding C bindings, but probably not as efficient.

2.2 TclMPI Support Functions

Several MPI entities like communicators, requests, status objects cannot be represented directly in Tcl. For TclMPI they need to be mapped to something else, for example a string that will uniquely identify this entity and then it will be translated into the real object it represents with the help of the following support functions.

2.2.1 Mapping Communicators

MPI communicators are represented in TclMPI by strings of the form "::tclmpi::comm%d", with "%d" being replaced by a unique integer. In addition, a few string constants are mapped to the default communicators that are defined in MPI. These are tclmpi::comm_world, tclmpi::comm_self, and tclmpi::comm_null, which represent MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_NULL, respectively.

Internally the map is maintained in a simple linked list which is initialized with the three default communicators when the plugin is loaded and where new communicators are added at the end as needed. The functions mpi2tcl_comm and tcl2mpi_comm are then used to translate from one representation to the other while tclmpi_add_comm will append a new communicator to the list.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Data Structure Documentation

5.1 tclmpi_comm Struct Reference

Collaboration diagram for tclmpi_comm:

Data Fields

- const char ∗ label
- MPI_Comm comm
- int valid
- tclmpi_comm_t ∗ next

5.1.1 Detailed Description

Linked list entry to map MPI communicators to strings.

5.1.2 Field Documentation

5.1.2.1 MPI_Comm tclmpi_comm::comm

MPI communicator corresponding of this entry

5.1.2.2 const char∗ tclmpi_comm::label

String representing the communicator in Tcl

5.1.2.3 tclmpi_comm_t∗ tclmpi_comm::next

Pointer to next element in linked list

5.1.2.4 int tclmpi_comm::valid

Non-zero if communicator is valid

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

- tcl_mpi.c

5.2 tclmpi_req Struct Reference

Collaboration diagram for tclmpi_req:

Data Fields

- const char ∗ **label**
- void ∗ **data**
- int **len**
- int **type**
- int **source**
- int **tag**
- MPI_Request ∗ **req**
- MPI_Comm **comm**
- tclmpi_req_t ∗ **next**

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

- tcl_mpi.c

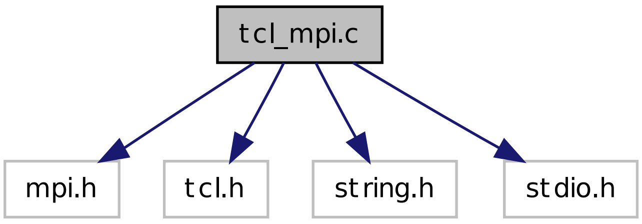
Chapter 6

File Documentation

6.1 tcl_mpi.c File Reference

```
#include <mpi.h>
#include <tcl.h>
#include <string.h>
#include <stdio.h>
```

Include dependency graph for tcl_mpi.c:



Data Structures

- struct tclmpi_comm
- struct tclmpi_req

Defines

- #define **TCLMPI_INVALID** -1
- #define **TCLMPI_NONE** 0
- #define **TCLMPI_AUTO** 1

- #define **TCLMPI_INT** 2
- #define **TCLMPI_INT_INT** 3
- #define **TCLMPI_DOUBLE** 4
- #define **TCLMPI_DOUBLE_INT** 5

Typedefs

- typedef struct tclmpi_comm tclmpi_comm_t
- typedef struct tclmpi_req **tclmpi_req_t**

Functions

- static const char ∗ tclmpi_add_comm (MPI_Comm comm)
- static const char ∗ mpi2tcl_comm (MPI_Comm comm)
- static MPI_Comm tcl2mpi_comm (const char ∗label)
- static const char ∗ **tclmpi_add_req** ()
- static tclmpi_req_t ∗ **tclmpi_find_req** (const char ∗label)
- static int **tclmpi_del_req** (tclmpi_req_t ∗req)
- static int **tclmpi_datatype** (const char ∗type)
- static int **tclmpi_errcheck** (Tcl_Interp ∗interp, int ierr, Tcl_Obj ∗obj)
- static int **tclmpi_commcheck** (Tcl_Interp ∗interp, MPI_Comm comm, Tcl_Obj ∗obj0, Tcl_Obj ∗obj1)
- static int **tclmpi_typecheck** (Tcl_Interp ∗interp, int type, Tcl_Obj ∗obj0, Tcl_Obj ∗obj1)
- int **TclMPI_Init** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Finalize** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Abort** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Comm_size** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Comm_rank** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Comm_split** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Barrier** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Bcast** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Allreduce** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Send** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Isend** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Recv** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Irecv** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Probe** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Iprobe** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **TclMPI_Wait** (ClientData nodata, Tcl_Interp ∗interp, int objc, Tcl_Obj ∗const objv[ ])
- int **Tclmpi_Init** (Tcl_Interp ∗interp)

Variables

- static tclmpi_comm_t ∗ **first_comm** = NULL
- static tclmpi_comm_t ∗ **last_comm** = NULL
- static int **tclmpi_comm_cntr** = 0
- static MPI_Comm **MPI_COMM_INVALID**
- static tclmpi_req_t ∗ **first_req** = NULL
- static int **tclmpi_req_cntr** = 0
- static char **tclmpi_errmsg** [MPI_MAX_ERROR_STRING]
- static int **tclmpi_init_done** = 0

6.1.1 Detailed Description

6.1.2 Typedef Documentation

6.1.2.1 typedef struct tclmpi_comm tclmpi_comm_t

Linked list entry type for managing MPI communicators

6.1.3 Function Documentation

6.1.3.1 const char ∗ mpi2tcl_comm (MPI_Comm *comm*) `[static]`

Translate an MPI communicator to its Tcl label.

Parameters

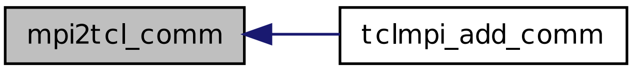
comm an MPI communicator

Returns

the corresponding string lable or NULL.

This function will search through the linked list of known communicators until it finds the (first) match and then returns the string label to the calling function. If a NULL is returned, the communicator does not yet exist in the linked list and a new entry will have to be added.

Here is the caller graph for this function:



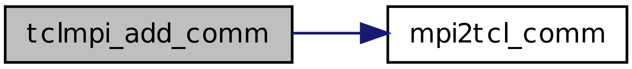
6.1.3.2 MPI_Comm tcl2mpi_comm (const char ∗ *label*) `[static]`

translate Tcl label to MPI communicator

6.1.3.3 const char ∗ tclmpi_add_comm (MPI_Comm *comm*) `[static]`

add communicator to translation table, if not already present

Here is the call graph for this function:



Index