TclMPI

Overview

This page describes Tcl bindings for MPI. With 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 and will allow to run Tcl scripts in parallel via `mpirun` or `mpiexec` similar to C, C++ or Fortran programs. The main motivation for writing this package is to allow using the VMD molecular visualization and analysis package in parallel without having to recompile VMD for MPI support and to complement a Tcl wrapper for the LAMMPS molecular dynamics simulation software.

Contents

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`.

Software Development and Bug Reports

Source code management for this project is hosted on github at https://github.com/akohlmey/tclmpi and you can clone 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 https://github.com/akohlmey/tclmpi/issues.

Similarities and Differences to the MPI C-bindings

Commands and constants are prefixed with `::tclmpi::` to reduce clashes with existing programs and all lowercase to stay consistent with normal Tcl commands. This may be expanded at a later point into a full fledged namespace, if the need arises. The overall philosophy of the bindings is to make the API similar to the MPI one, but don't stick to it slavishly and do things *the Tcl way* wherever possible and justified. Convenience and simplicity take precedence over performance. If performance matters that much, use the C/C++ or Fortran bindings but not Tcl. The biggest visible change is that sending data around, receive buffers will be automatically set up to handle the entire message, thus the typical "count" arguments of regular C/C++ or Fortran MPI bindings is not required and the received data will be the return value of the corresponding command. This convenience has consequences on the semantics, so calls to `::tclmpi::bcast` will be converted to *two* calls to `MPI_Bcast()`: the first will broadcast the size of the data set being sent (so that sufficiently sized buffers can be allocated) and the second will actually send the data. Similarly, `::tclmpi::recv` will be converted into calling `MPI_Probe()` and then `MPI_Recv()` for the same purpose, the latter call will also use the MPI_SOURCE and MPI_TAG flags from the `MPI_Status` object created for `MPI_Probe()`. For consistency with this only the `::tclmpi::allreduce` reduction has been implemented. 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, otherwise the parameters to the receive are cached and then the receive only called as part of `::tclmpi::wait`. As a consequence the experienced blocking/non-blocking behavior of the Tcl script should be **very** close to the corresponding C bindings, but probably not as efficient.

Completeness of the Implementation

TclMPI deliberately only implements a rather small subset of the MPI-1 and MPI-2 API. In many cases the missing APIs are not used a lot or won't make a significant difference, e.g. in case of buffered send and receives, since the Tcl to C interface requires an implicit buffering already. In a few cases, the design choices of the Tcl bindings implementation makes a certain call impossible to interface without significantly changing the semantics. Suggestions for improvements or requests for interfaces to additional MPI APIs are always welcome.

Data types

TclMPI currently supports three data types. The `::tclmpi::auto` is usually recommended for convenience, `::tclmpi::int` or `::tclmpi::double` should only be needed for TclMPI calls that perform mathematical operation (like `::tclmpi::allreduce`) or when large quantities of data have to be transferred and the time for serialization and data transfer becomes significant.

- `::tclmpi::auto`
  The Tcl object will be serialized (as a string) and then transferred and rebuilt. This data type can send **any** Tcl object that can be serialized (e.g. nested lists and mixed data types).
- `::tclmpi::int`
  The Tcl object (list or number) will be converted to integer and then transferred as integer array. This data type can send variable length lists of integers or single integers. **All** data will be converted to flat list of integers. List elements that cannot be directly converted into an integer (strings, lists) will be represented by 0.

- `::tclmpi::double`
  The Tcl object (list or number) will be converted to double precision floating point and then transferred as double precision floating point array. This data type can send variable length lists of doubles or single numbers. **All** data will be converted to a flat list of floating point numbers and thus data that cannot be converted into floating point (strings, lists) will be represented by 0.0.

Communicators

In TclMPI communicators are represented by string constants, which are converted inside the TclMPI code to match the specific representation of the MPI library. A the following communicators are predefined:

- `::tclmpi::comm_world`
  This is the equivalent of MPI_COMM_WORLD and is the communicator that contains all processes at the time of running `::tclmpi::init`.
- `::tclmpi::comm_self`
  This is the equivalent of MPI_COMM_SELF and is the communicator that contains the current rank only.
- `::tclmpi::comm_null`
  This is the equivalent of MPI_COMM_NULL and is the communicator that contains no processes.

TclMPI Command Reference

::tclmpi::init <argv>

This command initializes the MPI environment. Needs to be called before any other TclMPI commands. MPI can be initialized at most once, so calling `::tclmpi::init` multiple times is an error. Argument is the content of the `$argv` variable, which is automatically created by Tcl. Return value is a modified version of `$argv` that is processed by the underlying MPI library to remove all MPI implementation specific entries.

::tclmpi::finalize

This command closes the MPI environment and cleans up all MPI states. After calling this function, no more TclMPI commands including `::tclmpi::init` may be used. All processes much call this routine before exiting. This command takes no arguments and has no return value.

::tclmpi::abort <errorcode>

This command makes a best attempt to abort all tasks sharing the communicator and exit with the provided error code. Only one task needs to call `::tclmpi::abort`. This command terminates the program, so there can be no return value.

::tclmpi::comm_size <comm>

This function returns the number of processes involved in a communicator.

::tclmpi::comm_rank <comm>

This command returns the unique rank of the current process in a particular communicator. Rank is an integer between 0 and the size of a communicator.

::tclmpi::comm_split <comm> <color> <key>

This function partitions the processes involved in the provided communicator into disjoint subgroups, one for each value of `color`. All processes with the same value of `color` will form a new communicator. The `key` value determines the relative ranks of the processes in the new communicator with the current rank on the original communicator being used as tiebreaker. `color` has to be a non-negative integer or `::tclmpi::undefined`. The function returns the new communicator, or `::tclmpi::comm_null` in case of a color value of `::tclmpi::undefined`. This is a collective call, i.e. all processes of the communicator have to call it.

::tclmpi::barrier <comm>

This command blocks the calling process until all members of the communicator have called it and thus synchronizes all processes. There is no return value.

::tclmpi::bcast <data> <type> <root> <comm>

This command broadcasts the provided `data` object (list or single number or string) from the process with rank `root` on the communicator `comm` to all processes. The `data` argument has to be present on all processes but will be ignored on all but the rank `root` process. The data resulting from the broadcast will be stored in the return value of the command. This is important when the data type is not `::tclmpi::auto`, since using those data types may incur an irreversible conversion.

::tclmpi::allreduce <data> <type> <op> <comm>

This command performs a global reduction operation `op` on the provided `data` object across all processes participating in the communicator `comm`. If data is a list, then the reduction will be done across each respective entry of the same list index. The result is distributed to all processes and used as return value of the command. This command only supports the data types `::tclmpi::int` and `::tclmpi::double`. The following reduction operations are supported: ::tclmpi::max (maximum), ::tclmpi::min (minimum), ::tclmpi::sum (sum), ::tclmpi::prod (product), ::tclmpi::land (logical and), ::tclmpi::band (bitwise and), ::tclmpi::lor (logical or), ::tclmpi::bor (bitwise or), ::tclmpi::lxor (logical exclusive or), ::tclmpi::bxor (bitwise exclusive or), ::tclmpi::maxloc (max value and location), ::tclmpi::minloc (min value and location).

::tclmpi::send <data> <type> <dest> <tag> <comm>

This function performs a regular <u>blocking</u> send to process rank `dest` on communicator `comm`. The choice of data type determines how data is being sent (see above for details) and the corresponding receive has to use the exact same data type, or the transfer will fail.

::tclmpi::isend <data> <type> <dest> <tag> <comm>

This function performs a <u>non-blocking</u> send to process rank `dest` on communicator `comm`. The choice of data type determines how data is being sent (see above for details) and the corresponding receive has to use the exact same data type, or the transfer may fail or lead to inconsistent data. The non-blocking send returns immediately while the transfer is still in progress and returns a request handle of the form `::tclmpi::req#`, with `#` being a unique integer number. This request is best stored in a variable and needs to be handed to a `::tclmpi::wait` call to wait for completion and release all associated allocated resources with the communication request.

::tclmpi::recv <type> <source> <tag> <comm> ?status?

This procedure provides a <u>blocking</u> receive operation, i.e. it only returns after the message is received in full. The received data will be passed as return value. The `type` argument has to match that of the corresponding send command. Instead of a specific `source` rank, the constant `::tclmpi::any_source` can be used and similarly `::tclmpi::any_tag` as `tag`, to not select on source rank or tag, respectively. The (optional) status argument would be the name of a variable in which the resulting information will be stored in the form of an associative array. The associative array has the entries MPI_SOURCE (rank of sender), MPI_TAG (tag of message), COUNT_CHAR (size of message in bytes), COUNT_INT (size of message in `::tclmpi::int` units), COUNT_DOUBLE (size of message in `::tclmpi::double` units).

::tclmpi::irecv <type> <source> <tag> <comm>

This procedure provides a non-blocking receive operation, i.e. it returns immediately. The call does not return any data but a request handle of the form `::tclmpi::req#`, with `#` being a unique integer number. This request is best stored in a variable and needs to be handed to a `::tclmpi::wait` call to wait for completion and transfer the data pass it to the calling code as return value of the wait call. The `type` argument has to match that of the corresponding send command. Instead of a specific `source` rank, the constant `::tclmpi::any_source` can be used and similarly `::tclmpi::any_tag` as `tag`, to not select on source rank or tag, respectively. The (optional) status argument would be the name of a variable in which the resulting information will be stored in the form of an associative array. The associative array has the entries MPI_SOURCE (rank of sender), MPI_TAG (tag of message), COUNT_CHAR (size of message in bytes), COUNT_INT (size of message in `::tclmpi::int` units), COUNT_DOUBLE (size of message in `::tclmpi::double` units).

::tclmpi::probe <source> <tag> <comm> ?status?

This function allows to check for an incoming message on the communicator `comm` without actually receiving it. Nevertheless, this call is blocking, i.e. it won't return unless there is actually a message pending that matches the requirements of `source` rank and message `tag`. Instead of a specific `source` rank, the constant `::tclmpi::any_source` can be used and similarly `::tclmpi::any_tag` as `tag`, to accept send requests from any source rank or tag, respectively. The (optional) `status` argument would be the name of a variable in which the resulting information will be stored in the form of an associative array. The associative array has the entries MPI_SOURCE (rank of sender), MPI_TAG (tag of message), COUNT_CHAR (size of message in bytes), COUNT_INT (size of message in `::tclmpi::int` units), COUNT_DOUBLE (size of message in `::tclmpi::double` units).

::tclmpi::iprobe <source> <tag> <comm> ?status?

This function allows to check for an incoming message on the communicator `comm` without actually receiving it. Unlike ::tclmpi::probe, this call is non-blocking, i.e. it will return immediately and report whether there is a message pending or not in its return value (1 or 0). Instead of a specific `source` rank, the constant `::tclmpi::any_source` can be used and similarly `::tclmpi::any_tag` as `tag`, to not select on source rank or tag, respectively. The (optional) `status` argument would be the name of a variable in which the resulting information will be stored in the form of an associative array. The associative array has the entries MPI_SOURCE (rank of sender), MPI_TAG (tag of message), COUNT_CHAR (size of message in bytes), COUNT_INT (size of message in `::tclmpi::int` units), COUNT_DOUBLE (size of message in `::tclmpi::double` units).

::tclmpi::wait <request> ?status?

This function takes a communication request created by a non-blocking send or receive call and waits for its completion. In case of a send, it will merely wait until the matching communication is completed and any resources associated with the request can be releaseed. If the request was generated by a non-blocking receive call, ::tclmpi::wait will hand the received data to the calling routine in its return value. The (optional) `status` argument would be the name of a variable in which the resulting information will be stored in the form of an associative array. The associative array has the entries MPI_SOURCE (rank of sender), MPI_TAG (tag of message), COUNT_CHAR (size of message in bytes), COUNT_INT (size of message in `::tclmpi::int` units), COUNT_DOUBLE (size of message in `::tclmpi::double` units).

Examples

The following section provides some simple examples using TclMPI to recreate common MPI example programs.

Hello World

This is the TclMPI version of "hello world".

```
#!/usr/bin/tclsh
# hello world a la TclMPI

# point Tcl to the directory with pkgIndex.tcl
# and load the TclMPI package.
set auto_path [concat $env(PWD)/.. $auto_path]
package require tclmpi 0.2

# initialize TclMPI
set argv [::tclmpi::init $argv]

set comm ::tclmpi::comm_world
set size [::tclmpi::comm_size $comm]
set rank [::tclmpi::comm_rank $comm]

puts "hello world, this is rank $rank of $size"

# close out TclMPI
::tclmpi::finalize
exit 0
```

Computation of Pi

This script uses TclMPI to compute the value of Pi from numerical quadrature of the integral from zero to 1 over 4/(1+x*x).

```
#!/usr/bin/tclsh
# compute pi
set auto_path [concat $env(PWD)/.. $auto_path]
package require tclmpi 0.2

# initialize MPI
set argv [::tclmpi::init $argv]

set comm ::tclmpi::comm_world
set size [::tclmpi::comm_size $comm]
set rank [::tclmpi::comm_rank $comm]
set master 0

set num [lindex $argv 0]
# make sure all processes have the same interval parameter
set num [::tclmpi::bcast $num ::tclmpi::int $master $comm]

# run parallel calculation
set h [expr {1.0/$num}]
set sum 0.0
for {set i $rank} {$i < $num} {incr i $size} {
  set sum [expr {$sum + 4.0/(1.0 + ($h*($i+0.5))**2)}]
}
set mypi [expr {$h * $sum}]

# combine and print results
set mypi [::tclmpi::allreduce $mypi ::tclmpi::double ::tclmpi::sum $comm]
if {$rank == $master} {puts "result: $mypi. relative error: [expr {abs(($mypi -
3.14159265358979)/3.14159265358979)}]"}

# close out TclMPI
::tclmpi::finalize
exit 0
```

Version History

Version 0.1

Initial test version, unreleased

Version 0.2

First public version. Supported operations: init, finalize, comm_rank, comm_size, comm_split, barrier, bcast, allreduce, send, recv, and probe.

Version 0.3

Added operation: abort. Added unit tests. Converted MPI errors into Tcl errors (so they can be handled with `catch`), argument testing code refactoring.

Version 0.4

Fixed several bugs, more unit tests, better checking of arguments and error reports for it. Update/bugfix to `pi.tcl` example.

Version 0.5

Fixed a few bugs and significantly cleaned up the argument handling. Added some non-blocking operations: isend, irecv, iprobe and wait. Added parallel unit tests and wrote a simple test harness for it. This version will be considered feature complete unless specific needs arise or requests are being made. For the foreseeable time, only bugfixes, documentation updates, and text & example programs will be added.

Version 0.6

Current development version. Added modified BSD-style license, bundle PDF of this webpage as documentation.