mpiDL ®

The Power of MPI in IDL

Version 2.4.0

Tech-X Corporation
5621 Arapahoe Avenue, Suite A
Boulder, CO 80303
http://www.txcorp.com
info@txcorp.com

Contents

1 Preface

Names of functions, parameters, and other code samples are denoted in a fixed font — for example, `ls -la`.

Terms requiring a specific definition will appear in *italics* on first use.

mpiDL was built on Red Hat Enterprise Linux WS release 3 using glibc 3.2 and gcc 3.3 along with MPICH 1.2.7, MPICH2 1.0.5, or OpenMPI 1.2.3. For support on other platforms (such as Solaris), please contact Tech-X.

2 Introduction to mpiDL

2.1 Overview

Many experiments and simulations generate large data sets that must be processed quickly. Scientists exploring fluid and particle dynamics, high-energy and plasma physics, biophysics, protein folding and medical imaging are challenged visualizing and analyzing complex data. In response, many scientists and developers rely on the Interactive Data Language (IDL) from ITT Visual Information Solutions to visualize and analyze these large data sets.

However, some analysis cannot be practically accomplished on a workstation or server with only symmetric multi-processing and multi-threading. Clusters offer cost-effective computing power, but IDL does not naturally take advantage of a parallel environment.

To bridge the gap between IDL and parallel computing, Tech-X Corporation has developed FastDL. With FastDL, scientists and developers can run IDL visualization and analysis applications in parallel on clusters, significantly shortening the time required to get results.

FastDL is made up of two parts: TaskDL and mpiDL. The former is a task-farming solution designed for problems where no communication between nodes of a cluster is required — for example, rendering frames for an animation or running the same data reduction procedure over a collection of data files. In contrast, mpiDL was developed for calculations that require communication between processors and allows for parallel algorithms to be implemented and used. It leverages the power of the industry-standard Message Passing Interface (MPI) with the ease of use and varied visualization and analysis capabilities of IDL.

Seasoned MPI users will quickly feel at home using mpiDL. Parallel programmers can write IDL programs that call MPI functions using the same approach they would use when writing C or Fortran programs. mpiDL also gives developers access to MPI's built-in, specialized parallel functionality. Developers who are new to parallel programming using explicit message passing can get up to speed quickly by modifying the mpiDL examples to create their own parallel IDL programs. The functions and procedures behave the same as other IDL routines, so there is no new syntax or style to learn.

mpiDL implements all of the MPI standard. This is accomplished by using IDL's dynamic loadable modules (DLMs). DLMs allows IDL functions to be created from shared libraries that can loaded and used at runtime. These libraries can be written in other languages (such as C or Fortran) and allow IDL to take advantage of compiled code. mpiDL uses DLMs to call MPI functions from IDL. mpiDL includes IDL wrapper functions for the low-level MPI functions. This allows users to have a intuitive, familiar interface for MPI in IDL without having to know low-level about the MPI implementation such as the C data types for various parameters.

For more specific information on the MPI standard, visit http://www.mpi-forum.org.

3 mpiDL Installation

Installation of mpiDL is straightforward.

1. Expand the installer with

   `tar -xvzf mpiDL-2.4.0.tar.gz`

2. Move into the resulting directory:

   `cd mpiDL-2.4.0-installer`

3. Run the install script with

   `sh mpidl-install.sh`

   and follow the prompts that will take you through the installation process. Note that you may need root access to install mpiDL in certain locations, such as `/usr/local/rsi/idl/products`.

Note: the 64 bit version of mpiDL has slightly different filenames, such as the following:

1. `mpiDL-2.4.0-64.tar.gz`
2. `mpidl-install-64.sh`
3. `README.64`

Finally, there are three environment variables that need to be set in a user's shell configuration so that IDL will be able to use mpiDL. These can be set in a user's `.bashrc` or `.cshrc` files, depending on the shell used. For shells other than bash or csh, consult the appropriate documentation. The environment variable `MPIDL_DIR` must point to the location of mpiDL. For example, if you are using bash and have installed mpidl in `/usr/local/rsi/idl/products/mpidl`, you can add the following to your configuration:

```
export MPIDL_DIR="/usr/local/rsi/idl/products/mpidl"
```

Using csh, a similar line would be

```
setenv MPIDL_DIR /usr/local/rsi/idl/products/mpidl
```

For other shells, please consult the appropriate documentation.

The location of mpiDL IDL files should also be set in the IDL environment variables `IDL_PATH` and `IDL_DLM_PATH`. To do this, one could add a line similar to

```
export IDL_PATH="+$MPIDL_DIR:<IDL_DEFAULT>"
export IDL_DLM_PATH="+$MPIDL_DIR:<IDL_DEFAULT>"
```

in bash. Using csh, this would change to

```
setenv IDL_PATH "+$MPIDL_DIR:<IDL_DEFAULT>"
setenv IDL_DLM_PATH "+$MPIDL_DIR:<IDL_DEFAULT>"
```

The '+' in the IDL variables tells IDL to search the path recursively, ensuring that the directory specified and all directories under it become part of the search path. The `IDL_PATH` and `IDL_DLM_PATH` may already be set in your configuration. If so, you should add to the paths searched by these variables — paths are separated by ':'.

Additionally, it is useful to modify your `PATH` to include the `bin/` directory of the mpiDL installation and the location of `mpirun` and other MPI utilities. The nodes of the cluster will require a full path to find `runmpidl`, and having its location in your `PATH` makes this easier.

mpiDL requires the MPICH 1.2, MPICH2 1.0, or OpenMPI 1.2 implementation of MPI to be installed. MPICH is available at http://www-unix.mcs.anl.gov/mpi/mpich/.

A full IDL license is required on the head node. Runtime or interactive IDL licenses are required on all other nodes. Users should also be careful to ensure that all needed programs, libraries, etc. can be found by the workers. Also, mpiDL requires that the nodes be able to communicate via `rsh` or `ssh` — this should already be the case if the MPI implementation being used is functional.

4 mpiDL — Parallelizing IDL Scripts with MPI

4.1 Running mpiDL

mpiDL is invoked with the following:

```
runmpidl -np n [-machinefile ./nodes] myfile.sav
```

where `n` is the number of processes to run in parallel and `myfile.sav` is the IDL runtime save file to be restored and run on the cluster.

The `runmpidl` script will attempt to invoke an MPI run correctly, using either `mpiexec` or `mpirun` — these must be in your `PATH` environment variable. For MPI implementations using an `mpd` daemon, the `runmpidl` script will start `mpd` if it finds it in the same directory as `mpiexec`. It will also expand the path to the save file, ensuring that the nodes will also be able to access it correctly. Note, however, that if you specify the optional `nodes` file for your MPI run, it will need to have either a full or relative path — at least a './' is required. Finally, any other flags passed to `runmpidl` are in turn passed on to the appropiate MPI execution script.

4.2 Runtime IDL

Understanding how runtime IDL works will help in understanding how the cluster nodes will run. Upon startup, the specified IDL runtime save file is loaded on each process. IDL first looks for a procedure in the `.sav` file named "MAIN" and executes that procedure if it exists. If "MAIN" is not found, IDL looks for a procedure in the `.sav` with the same name as the file ("myfile" in the above example) and attempts to execute that procedure. If neither procedure exists, an error occurs. Save files must be used, as `.pro` files cannot be compiled in runtime mode.

`libidlmpi.so` and `libidlmpi.dlm` are are both in the `lib/` directory of the mpiDL installation. This location must be specified in the `IDL_DLM_PATH` environment variable (see Section 3 to see how to set this). Calling any of the routines in the library will automatically load the mpiDL `dlm` in IDL. Note that routines to be run and data cannot be stored in the same IDL save file.

4.3 Example Programs

For detailed examples of how to use the wrapped MPI calls in IDL, please see the included example programs. This code is provided for both testing the installation and to provide full, working examples to make learning how to use mpiDL easier. Each example demonstrates the use of different MPI routines.

The following examples are provided in the `examples/` directory of the mpiDL installation:

- `allgather_example.pro`
- `allgatherv_example.pro`
- `allreduce_example.pro`
- `alltoall_example.pro`
- `alltoallv_example.pro`
- `bcast_example.pro`
- `gather_example.pro`
- `group_example.pro`
- `isendrecv_example.pro`

- parallel_pi.pro
- reduce_example.pro
- scatter_example.pro
- sendrecv_example.pro
- gravity_example.pro

Each example includes an IDL .pro file and a .sav file. To run an example (such as bcast_example), call runmpidl as described above with the .sav file of the example you wish to run. For instance, to run bcast_example, you may do something similar to the following:

```
runmpidl -np 4 bcast_example.sav
```

Examining the IDL code and comments in the .pro files will show how to use the mpiDL functions and allow you to quickly see how mpiDL procedures are constructed and how to use MPI function calls.

4.3.1 Creating a New Runtime Save File

To create an IDL .sav file from your own mypro.pro procedures, execute the following in IDL:

1. Run IDL.
2. At the IDL> prompt, type
   (a) IDL> .RESET_SESSION
   (b) IDL> .COMPILE mypro
   (c) IDL> RESOLVE_ALL
   (d) IDL> SAVE,filename='mypro.sav',/ROUTINES
   (e) IDL> EXIT

This will compile your procedure, resolve any dependencies, and create a .sav file that can be used by mpiDL. Another method for creating .sav files is to run mpiDL with a single processor and save the result. For example, the following commands run on with a full IDL license will create a .sav file:

1. runmpidl -np 1 /path/to/mpidlstart -i
   At the resulting IDL prompt, execute the following:
2. IDL> mypro
3. IDL> SAVE,filename='mypro.sav',/ROUTINES
4. IDL> EXIT

However, this method of creating a .sav will save only functions and procedures that have been run by the mypro procedure. If this method is used to create a save file, make sure that all functionality you wish to include is exercised.

4.4 MPI Functions Included in mpiDL

mpiDL is an implementation of a subset of MPI functions which can be called natively from IDL. mpiDL provides access to MPI routines through wrappers. These routines are written in IDL and provide an IDL-like syntax and error checking for underlying MPI calls. These function names are prefixed with "`MPIDL_`" and correspond to the matching MPI function prefixed with "`MPI_`". See the function reference in Section 5 for the proper syntax of the `MPIDL_` functions.

The wrapped `MPIDL_` functions are:

- MPI Command and control
  - `MPIDL_INIT`
  - `MPIDL_INITIALIZED`
  - `MPIDL_FINALIZE`
  - `MPIDL_COMM_RANK`
  - `MPIDL_COMM_SIZE`
  - `MPIDL_COMM_CREATE`
  - `MPIDL_COMM_GROUP`
  - `MPIDL_GROUP_RANK`
  - `MPIDL_GROUP_SIZE`
  - `MPIDL_GROUP_INCL`
- Point-to-point communication
  - `MPIDL_SEND`
  - `MPIDL_RECV`
  - `MPIDL_ISEND`
  - `MPIDL_IRECV`
  - `MPIDL_PROBE`
  - `MPIDL_IPROBE`
  - `MPIDL_GET_COUNT`
- 1-to-many, many-to-1 communication
  - `MPIDL_BARRIER`
  - `MPIDL_BCAST`
  - `MPIDL_SCATTER`
  - `MPIDL_GATHER`
  - `MPIDL_ALLGATHER`
  - `MPIDL_ALLGATHERV`
  - `MPIDL_ALLTOALL`
  - `MPIDL_ALLTOALLV`
  - `MPIDL_REDUCE`
  - `MPIDL_ALLREDUCE`
  - `MPIDL_WAIT`

4.5 mpiDL Benchmarks

The goal of parallel programming is to provide improved performance in complex, computationally intense calculations. The following example shows how mpiDL can be used to achieve this goal.

In this example (`gravity_example.pro` — source code is available with the mpiDL distribution), a simulation of some large number of masses are distributed in space is performed. The motion of each particle under the force of gravity can be calculated on a distributed cluster. Each process keeps track of the position of every particle at each time step, updating the positions using the collective communication routine `MPI_ALLGATHER`. However, each process calculates the force from all masses only on a subset of particles. The time to do this calculation increases as the total number of particles squared, so speedup is seen as the number of working processes increases. Figure 1 shows the mean computational speedup as a function of the number of processes. Of course, not all problems are suited to parallelization, and how a parallel algorithm is implemented can have a dramatic effect on the resulting improvements in performance.

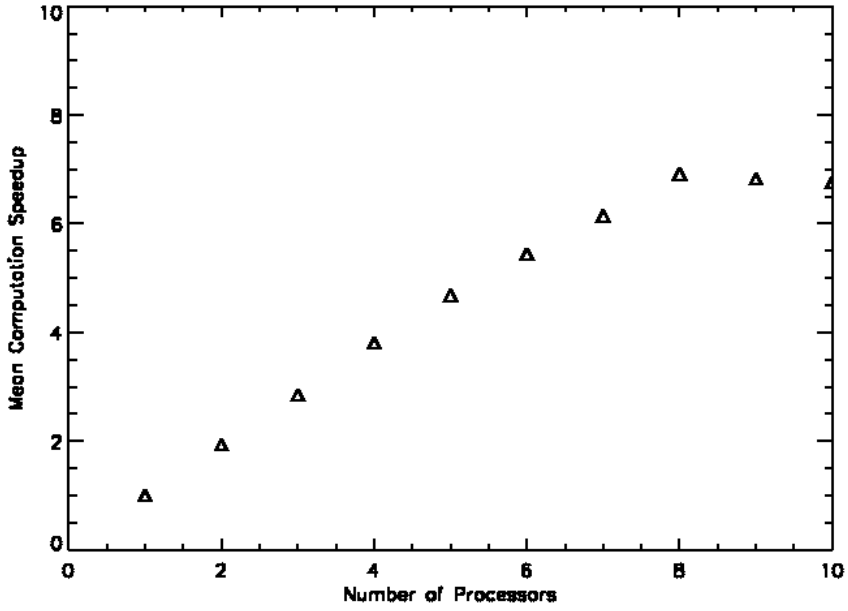

Figure 1: Benchmark of gravity example

5 mpiDL Function Reference

mpiDL provides a high-level interface, written in IDL, to the MPI library. The routines in the high-level interface are prefixed with '`MPIDL_`', and correspond with the matching MPI function. For more information on the MPI functions themselves, consult the MPI documentation. One set of good documents is located at http://www-unix.mcs.anl.gov/mpi/. In this section, we discuss the high-level interface, with syntax and interface appropriate for inclusion in standard IDL programs.

5.1 MPIDL_ALLGATHER

This function provides a method of collecting a distributed array from among all processes and then re-distributing the resulting complete array back to the nodes. Functionally, this is the same as using `MPIDL_GATHER` and then calling `MPIDL_BCAST`; however, the implementation of this function is generally more efficient than doing the steps independently.

The block of data sent by the jth process is placed in the jth block of the returned array on every process.

Syntax

Result = MPIDL_ALLGATHER(Send_Data [, SENDCOUNT=sendcount]
[, RECVCOUNT = recvcount] [,COMM=comm] [, ERROR=error])

Return Value

Returns an array composed of blocks of data sent from every process.

Arguments

Send_Data

The data being sent to all processes in the communicator.

Keywords

SENDCOUNT

Optional. The number of elements of the data being sent. The default value is N_ELEMENTS(Send_Data).

RECVCOUNT

Optional. The number of elements of the returned array. The default is the value of SENDCOUNT.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Allgather` call.

5.2 MPIDL_ALLGATHERV

MPIDL_ALLGATHERV gathers data from all processes and delivers it to all processes. This function is similar to MPIDL_ALLGATHER, but allows different processes to send different amounts of data. The block of data sent from the jth process is received by every process and placed in the jth block of the returned array.

Syntax

Result = MPIDL_ALLGATHERV(Send_Data, SENDCOUNT=sendcount,
RECVCOUNTS = recvcounts, DISPLS=displs [,COMM=comm] [, ERROR=error])

Return Value

Returns an array composed of blocks of data sent from every process.

Arguments

Send_Data

The data being sent to all processes in the communicator.

Keywords

SENDCOUNT

The number of elements being sent from each individual process.

RECVCOUNTS

An array of length equal to the number of processes in the communicator containing the number of elements being received from each process.

DISPLS

An array of length equal to the number of processes in the communicator containing the offsets relative to the beginning of the received array at which the received data is to be placed. The jth entry is the displacement of data received from the jth process.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Allgatherv call.

5.3 MPIDL_ALLREDUCE

MPIDL_ALLREDUCE reduces values contained on all processes to a single value. This function is similar to MPIDL_REDUCE but all processes receive the reduced array. The operation is applied to each element of the arrays across all processes. Thus the zeroth element of the Result array is the specified operation as applied to the zeroth elements of all of the Send_Data arrays which are distributed on each process.

Note: the MPI operations MPI_MAXLOC and MPI_MINLOC are not supported.

Syntax

Result = MPIDL_ALLREDUCE(Send_Data [, COUNT=count] [,/MPI_MAX, /MPI_MIN, /MPI_SUM, /MPI_PROD, /MPI_LAND, /MPI_BAND, /MPI_LOR, /MPI_BOR, /MPI_LXOR, /MPI_BXOR] [, COMM=comm] [, ERROR=error])

Return Value

Returns an array of length COUNT containing the reduced values.

Arguments

Send_Data

An array on each process which contains values to be reduced. Significant on every process.

Keywords

COUNT

Optional. The number of elements in the reduced array. Default value is equal to the number of elements in Send_Data.

MPI_MAX

Keyword argument specifying that the reduction operation is to compute the maximum value of the data.

MPI_MAX

Keyword argument specifying that the reduction operation is to compute the maximum value of the data.

MPI_MIN

Keyword argument specifying that the reduction operation is to compute the minimum value of the data.

MPI_SUM

Keyword argument specifying that the reduction operation is to compute the sum of the data.

MPI_PROD

Keyword argument specifying that the reduction operation is to compute the product of the data.

MPI_LAND

Keyword argument specifying that the reduction operation is to compute the logical AND of the data.

MPI_BAND

Keyword argument specifying that the reduction operation is to compute the bitwise AND of the data.

MPI_LOR

Keyword argument specifying that the reduction operation is to compute the logical OR of the data.

MPI_BOR

Keyword argument specifying that the reduction operation is to compute the bitwise OR of the data.

MPI_LXOR

Keyword argument specifying that the reduction operation is to compute the logical XOR (exclusive OR) of the data.

MPI_BXOR

Keyword argument specifying that the reduction operation is to compute the bitwise XOR of the data.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Allreduce` call.

5.4 MPIDL_ALLTOALL

MPIDL_ALLTOALL sends data from all processes to all processes.

Syntax

Result = MPIDL_ALLTOALL(Send_Data, COUNT=count[,COMM=comm] [, ERROR=error])

Return Value

Returns an array of length RECVCOUNT * (Number of processes in the communicator) containing blocks of data sent from each process. The jth block of the array corresponds to data sent from the jth process.

Arguments

Send_Data

The array to be distributed to all processes.

Keywords

COUNT

Optional. The number of elements to be sent and received from each process. Default value is the number of elements of Send_Data divided by the number of processes.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Alltoall call.

5.5 MPIDL_ALLTOALLV

MPIDL_ALLTOALLV sends data from all processes to all processes. This function is like MPIDL_ALLTOALL except that different amounts of data may be sent from different processes.

Syntax

Result = MPIDL_ALLTOALL(Send_Data, SENDCNTS=sendcnts,
RECVCNTS=recvcnts, SDISPLS=sdispls, RDISPLS=rdispls[, COMM=comm] [, ERROR=error])

Return Value

Returns an array containing blocks of data from each process. The blocks may vary in size and the size of the returned array may vary on each receiving process.

Arguments

Send_Data

The array on a given process to be distributed.

Keywords

SENDCNTS

An array of length equal to the number of processes in the communicator containing the number of elements to be sent to all processes. The jth entry is the number of elements to send to the jth process.

SDISPLS

An array of length equal to the number of processes in the communicator containing the displacement relative to the beginning of Send_Data from which to take a block of data to send. The jth element is the displacement of data to send to the jth process.

RECVCNTS

An array of length equal to the number of processes in the communicator containing the maximum number of elements which can be received from each process. The jth entry is the number of elements to receive from the jth process.

RDISPLS

An array of length equal to the number of processes in the communicator containing the displacement relative to the beginning of received array in which to place data which is received. The jth element is the displacement of data to received from the jth process.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Alltoallv call.

5.6 MPIDL_BARRIER

This function stops execution of a procedure until all processes in the communicator have entered the function. This is useful in synchronizing procedures and ensuring that execution on different processes does not become unbalanced.

Syntax

MPIDL_BARRIER([COMM=comm][, ERROR=error])

Return Value

None.

Arguments

None.

Keywords

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Barrier` call.

5.7 MPIDL_BCAST

This function is a 1-to-many communication that broadcasts a message to all processes in the communication group, including the broadcasting process. This function should be called by *all* processes with the same COUNT and ROOT keywords. At the end of the call, the data variable at all processes will contain the value from the root. The receivers must have the array containing the data defined with the correct dimension, or an error will occur.

Note that all processes must declare the named variable Data before calling `MPIDL_BCAST`. Failing to declare this variable will also result in an error.

Syntax

MPIDL_BCAST, Data, COUNT=count, ROOT=root [,COMM=comm] [, ERROR=error]

Return Value

None.

Arguments

Data

The array to be broadcast to all processes.

Keywords

COUNT

The number of elements to be broadcast.

ROOT

The rank of the process that will broadcast the message. This is the process that holds the value to be given to the rest of the group.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Bcast` call.

5.8 MPIDL_COMM_CREATE

This function returns a long integer that is the identifier for a new communicator.

Syntax

Result = MPIDL_COMM_CREATE(GROUP=group [,COMM=comm] [, ERROR=error])

Return Value

Returns a non-negative long integer representing the identifier for a new communicator.

Arguments

None.

Keywords

GROUP

A long representing the group from which to create the new communicator.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Comm_Create` call.

5.9 MPIDL_COMM_GROUP

This function returns a long integer that is the group underlying a given communicator.

Syntax

Result = MPIDL_COMM_GROUP([COMM=comm] [, ERROR=error])

Return Value

Returns a non-negative long integer representing the group underlying the communicator.

Arguments

None.

Keywords

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Comm_Group` call.

5.10 MPIDL_COMM_RANK

This function returns a long integer that is the rank of the process in its communicator. This rank is a unique identifier for the process.

Syntax

Result = MPIDL_COMM_RANK([COMM=comm] [, ERROR=error])

Return Value

Returns a non-negative long integer representing the rank of the process.

Arguments

None.

Keywords

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Comm_Rank call.

5.11 MPIDL_COMM_SIZE

This function returns a non-negative long integer representing the total number of processes in the communicator.

Syntax

Result = MPIDL_COMM_SIZE([COMM=comm] [, ERROR=error])

Return Value

A long integer representing the total number of processes in the communicator.

Arguments

None.

Keywords

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Comm_Size` call.

5.12 MPI_CONSTS

This function creates a named structure that contains values for all of the MPI_-defined constants, such as can be found in mpi.h. The values of the MPI constants are installation dependent, and will vary from system to system.

Syntax

MPI_CONSTS

Return Value

None. Creates a named structure called MPI that contains all MPI defined constants. The structure members are named to correspond with the MPI defined constants. For example, the IDL value MPI.ERR_RANK corresponds to the MPI constant MPI_ERR_RANK.

The structure also contains an array, MPI.IDL_TYPEMAP, that maps IDL types to MPI types. For instance, IDL variables of type long have type 3. The value of MPI.IDL_TYPEMAP[3] is 6, which is the value of the MPI constant MPI_INT (and the IDL value MPI.INT).

The MPI structure also contains the definition of MPI.STATUS, a long array of length MPI.STATUS_SIZE. Status objects are used in functions such as MPIDL_RECV and MPIDL_GET_COUNT, and contain information about MPI messages.

Arguments

None.

Keywords

None.

5.13 MPIDL_FINALIZE

This function performs the necessary shutdown tasks needed for MPI. mpiDL calls `MPIDL_INIT` and `MPIDL_FINALIZE` automatically when starting and shutting down the remote IDL processes. This function should only be called by the user when mpiDL is running interactively on the root process. If the master node is running an interactive session and `MPIDL_FINALIZE` is not called (or exits abnormally) mpiDL may leave orphaned or zombie processes on the cluster. The user should kill these processes manually if this happens.

Syntax

MPIDL_FINALIZE [, ERROR=error]

Return Value

None.

Arguments

None.

Keywords

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Finalize` call.

5.14 MPIDL_GATHER

MPIDL_GATHER gathers together data from all processes onto a single process.

Syntax

Result = MPIDL_GATHER(Send_Data, SENDCOUNT=sendcount, RECVCOUNT=recvcount, ROOT=root [, COMM=comm] [, ERROR=error])

Return Value

Returns an array of length RECVCOUNT * (Number of processes in the communicator) containing blocks of data sent from each process. The jth block of the array corresponds to data sent from the jth process.

Arguments

Send_Data

An array of length SENDCOUNT. Significant on every process.

Keywords

SENDCOUNT

Optional. The number of elements being sent from each individual process. Default value is the number of elements of Send_Data.

RECVCOUNT

Optional. The number of elements of the data received from any individual process. Significant only at ROOT. Default value is SENDCOUNT.

ROOT

The rank of the process which is to receive all of the gathered data. Significant on all processes.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Gather call.

5.15 MPIDL_GET_COUNT

This function takes the status of a MPI call (such as that returned from MPIDL_RECV or MPIDL_PROBE) and returns the number of elements in that message. This is especially useful when processes may be sending an array of data of unknown length.

Syntax

Result =MPIDL_GET_COUNT(Status[,/BYTE, /LONG, /INT, /FLOAT, /DOUBLE, /STRING] [,COMM=comm] [, ERROR=error])

Return Value

Returns the number of elements that will be received by the operation designated by the argument STATUS.

Arguments

STATUS

The status of an MPI call to be checked. This data is returned from MPIDL_PROBE and MPIDL_IPROBE.

Keywords

BYTE

Optional. Set this keyword if the data being received is of type BYTE.

LONG

Optional. Set this keyword if the data being received is of type LONG.

INT

Optional. Set this keyword if the data being received is of type INT.

FLOAT

Optional. Set this keyword if the data being received is of type FLOAT.

DOUBLE

Optional. Set this keyword if the data being received is of type DOUBLE. This is the default.

STRING

Optional. Set this keyword if the data being received is of type STRING.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Get_Count call.

5.16 MPIDL_GROUP_INCL

This function produces a new group by reordering an existing group and taking only listed members.

Syntax

Result = MPIDL_GROUP_INCL(RANKS=ranks [,GROUP=group] [, ERROR=error])

Return Value

Returns a non-negative long integer identifying the new group.

Arguments

None.

Keywords

RANKS

An array of longs representing the ranks of the processes in the existing group from which to create the new group.

GROUP

Optional. A long representing the existing group. If this keyword is not present, the default used is the group identified as the return value of `MPIDL_COMM_GROUP()`.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Group_Incl` call.

5.17 MPIDL_GROUP_RANK

This function returns a long integer that is the rank of the calling process in the current group. This rank is a unique identifier within the group.

Syntax

Result = MPIDL_GROUP_RANK([GROUP=group] [, ERROR=error])

Return Value

Returns a non-negative long integer representing the rank of the process within the group.

Arguments

None.

Keywords

GROUP

Optional. A long representing the group. If this keyword is not present, the default used is the group identified as the return value of `MPIDL_COMM_GROUP()`.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Group_Rank` call.

5.18 MPIDL_GROUP_SIZE

This function returns a long integer representing the total number of processes in a group.

Syntax

Result = MPIDL_GROUP_SIZE([GROUP=group] [, ERROR=error])

Return Value

A non-negative long integer representing the total number of processes in a group.

Arguments

None.

Keywords

GROUP

Optional. A long representing the group. If this keyword is not present, the default used is the group identified as the return value of `MPIDL_COMM_GROUP()`.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Group_Size` call.

5.19 MPIDL_INIT

This procedure initialized the MPI communicator, and establishes communication between slave and master nodes. This procedure is automatically called when mpiDL is run and in general should not be called from within IDL programs. The only exception is if the master node is being launched in interactive mode. In this case the master IDL process *must* call `MPIDL_INIT` to initialize MPI.

Syntax

MPIDL_INIT [, ERROR=error]

Return Value

None.

Arguments

None.

Keywords

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Init` call.

5.20 MPIDL_INITIALIZED

This function returns 1 if the MPI routine `MPIDL_INIT` has been executed successfully and returns 0 otherwise. This is one way to determine from within IDL that all MPI initialization happened properly at start-up.

Syntax

Result = MPIDL_INITIALIZED([ERROR=error])

Return Value

Returns one (of type long) if `MPIDL_INIT` has been run, zero otherwise.

Arguments

None.

Keywords

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Initialized` call.

5.21 MPIDL_IPROBE

MPIDL_IPROBE functions similarly to MPIDL_PROBE, except that MPIDL_IPROBE does not block — code execution will not wait for this function to return.

Syntax

Result = MPIDL_PROBE([SOURCE=source] [, TAG=tag][, COMM=comm] [, ERROR=error])

Return Value

Returns a status array (4 element long) that describes the data to be received. This array contains (in order) the count, id of processor sending the message, tag, and error code.

Keywords

SOURCE

Optional. The rank of the process to check for incoming messages. If not specified, returns the status of a message from any sending source.

TAG

Optional. A specified tag for which to check. If not specified, any tag is accepted.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Iprobe call.

5.22 MPIDL_IRECV

This function receives data sent by another process. It is similar to MPI_RECV, but MPI_IRECV is non-blocking — program execution may continue even if the message has not yet been sent.

Syntax

Result =MPIDL_IRECV(COUNT=count, SOURCE=source, REQUEST=request [, /BYTE, /LONG, /INT, /FLOAT, /DOUBLE, /STRING] [,TAG=tag] [,COMM=comm] [, ERROR=error])

Return Value

Returns a one-dimensional array of size count and the specified type. The returned array will initially contain zeroes until the receive is completed by calling MPIDL_WAIT. This function also sets the value of request, which is a required keyword.

MPIDL_IRECV does not properly receive scalar strings or string arrays, but scalar strings sent with MPIDL_ISEND can be received with a corresponding MPIDL_RECV call.

Arguments

None.

Keywords

COUNT

The number of elements in the data. This can be obtained from MPIDL_GET_COUNT.

SOURCE

Optional. The rank of the process from which to receive the data. If not specified, any source may be the sending process.

REQUEST

A name LONG variable containing the request ID of the buffered message. This is used as an input for finishing the communication, typically as an argument to MPIDL_WAIT.

BYTE

Optional. Set this keyword if the data being received is of type BYTE.

LONG

Optional. Set this keyword if the data being received is of type LONG.

INT

Optional. Set this keyword if the data being received is of type INT.

FLOAT

Optional. Set this keyword if the data being received is of type FLOAT.

DOUBLE

Optional. Set this keyword if the data being received is of type DOUBLE. This is the default.

STRING

Optional. Set this keyword if the data being received is of type STRING.

TAG

Optional. The MPI tag for this data.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Irecv` call.

5.23 MPIDL_ISEND

This function sends data from one process to another, similar to MPI_SEND, but MPI_ISEND is non-blocking — program execution may continue even if the message has not yet been received.

Syntax

MPIDL_ISEND, Data, DEST=destination, REQUEST=request[, TAG=tag] [, COMM=comm] [, ERROR=error]

Return Value

None.

Arguments

Data

Data to be sent to the destination. Data may be a scalar or array of type BYTE, INT, LONG, FLOAT, or DOUBLE. Multi-dimensional arrays may be sent, but will be received as 1-dimensional arrays; therefore, they will need to be reshaped on the receiving end. Scalar strings may also be sent, but string arrays are not supported.

Keywords

DEST

The rank of the destination process.

REQUEST

A named LONG variable containing the request ID of the buffered message. This is used as an input for finishing the communication, typically as an argument to MPIDL_WAIT.

TAG

Optional. The MPI tag for this data (see MPI documentation).

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Isend call.

5.24 MPIDL_PROBE

MPIDL_PROBE determines if a send message has been posted to be received by the calling process. The function returns information about the message, such as the source process that sent it and the size of the message. MPIDL_PROBE is a blocking function; the function will not return until a message that matches the keyword criteria is posted to be received.

Syntax

Result =MPIDL_PROBE([SOURCE=source] [, TAG=tag] [, COMM=comm] [, ERROR=error])

Return Value

Returns a status array (4 element, of type long) that describes the data to be received. This array contains (in order) the count, id of processor sending the message, tag, and error code.

Keywords

SOURCE

Optional. The rank of the process to check for incoming messages. If not specified, returns the status of a message from any sending source.

TAG

Optional. A specified tag for which to check. If not specified, any tag is accepted.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Probe call.

5.25 MPIDL_RECV

This function receives data from a process, acting as the counterpart to MPIDL_SEND. Using MPIDL_RECV requires knowing which process is sending the message. This function is blocking, meaning execution will halt on this statement until the required message is sent.

MPIDL_RECV will receive multi-dimensional arrays as a flattened one dimensional array — the user must reconstruct the shape after receiving the data. For example, if you receive an array sent as a two dimensional array of shape 3 by 4 with MPIDL_RECV, the result will be a one dimensional array containing twelve elements. The original array can be reconstructed from this using the IDL function REFORM.

Syntax

Result = MPIDL_RECV(COUNT=count, SOURCE=source [, /BYTE] [, /LONG] [, /INT] [, /FLOAT] [, /DOUBLE] [, /STRING] [,TAG=tag] [,COMM=comm] [, ERROR=error])

Return Value

Returns the data sent from SOURCE (via the MPIDL_SEND or MPIDL_ISEND function.)

Arguments

None

Keywords

COUNT

The number of elements in the data. This can be obtained from MPIDL_GET_COUNT if it is not known.

SOURCE

The rank of the process from which to receive the data.

BYTE

Optional. Set this keyword if the data being received is of type BYTE.

LONG

Optional. Set this keyword if the data being received is of type LONG.

INT

Optional. Set this keyword if the data being received is of type INT.

FLOAT

Optional. Set this keyword if the data being received is of type FLOAT.

DOUBLE

Optional. Set this keyword if the data being received is of type DOUBLE. This is the default.

STRING

Optional. Set this keyword if the data being received is of type STRING.

TAG

Optional. The MPI tag for this data.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Recv` call.

5.26 MPIDL_REDUCE

MPIDL_REDUCE reduces values contained on all processes to a single value on the ROOT process. The operation is applied to each element of the arrays across all processes. Thus the zeroth element of the Result array is the specified operation as applied to the zeroth elements of all of the Send_Data arrays which are distributed on each process.

Note that the operations MPI_MAXLOC and MPI_MINLOC are not supported by mpiDL.

Syntax

Result = MPIDL_Reduce(Send_Data, ROOT=root [, COUNT=count] [,/MPI_MAX, /MPI_MIN, /MPI_SUM, /MPI_PROD, /MPI_LAND, /MPI_BAND, /MPI_LOR, /MPI_BOR, /MPI_LXOR, /MPI_BXOR], [, COMM=comm] [, ERROR=error])

Return Value

Returns an array of length COUNT containing the reduced values.

Arguments

Send_Data

An array on each process which contains values to be reduced. Significant on every process.

Keywords

ROOT

The rank of the process which is to receive all of the reduced data. Significant on all processes.

COUNT

Optional. The number of elements in the reduced array. Default value is equal to the number of elements in Send_Data.

MPI_MAX

Keyword argument specifying that the reduction operation is to compute the maximum value of the data.

MPI_MAX

Keyword argument specifying that the reduction operation is to compute the maximum value of the data.

MPI_MIN

Keyword argument specifying that the reduction operation is to compute the minimum value of the data.

MPI_SUM

Keyword argument specifying that the reduction operation is to compute the sum of the data.

MPI_PROD

Keyword argument specifying that the reduction operation is to compute the product of the data.

MPI_LAND

Keyword argument specifying that the reduction operation is to compute the logical AND of the data.

MPI_BAND

Keyword argument specifying that the reduction operation is to compute the bitwise AND of the data.

MPI_LOR

Keyword argument specifying that the reduction operation is to compute the logical OR of the data.

MPI_BOR

Keyword argument specifying that the reduction operation is to compute the bitwise OR of the data.

MPI_LXOR

Keyword argument specifying that the reduction operation is to compute the logical XOR (exclusive OR) of the data.

MPI_BXOR

Keyword argument specifying that the reduction operation is to compute the bitwise XOR of the data.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Reduce` call.

5.27 MPIDL_SCATTER

This function is used to send data from one process to all other processes in the communication group. All processes in the communication group should call this function in order for it to complete.

Syntax

Result = MPIDL_SCATTER(Send_Data, ROOT=root [, SENDCOUNT=sendcount] [, RECVCOUNT=recvcount] [, COMM=comm] [, ERROR=error])

Return Value

Returns an array that will contain the fraction of the data given to the process.

Arguments

Send_Data

Array containing the data to be "scattered". Note that `Send_Data` must be defined on each process or IDL will generate an error.

Keywords

ROOT

The rank of the process scattering the data. This is the process that holds the array to be scattered to the rest of the group.

SENDCOUNT

Optional. The number of elements to be send to each process. This is significant only on the root process. The default value is N_ELEMENTS divided by the number of processors in the communicator group.

RECVCOUNT

Optional. The number of elements being received by each process. The default is the value of SENDCOUNT.

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the `MPI_COMM_WORLD` communicator.

ERROR

Optional. A named variable to receive the long integer error code from the `MPI_Scatter` call.

5.28 MPIDL_SEND

This function sends data from one process to another. This is a blocking communication, meaning that a call to MPIDL_SEND will not return until the message is received on the other end.

Syntax

MPIDL_SEND, Data, DEST=destination[, TAG=tag] [, COMM=comm] [, ERROR=error]

Return Value

None.

Arguments

Data

Data to be sent to the destination. Data may be a scalar or array of type BYTE, INT, LONG, FLOAT, or DOUBLE. Multi-dimensional arrays may be sent, but will be received as 1-dimensional arrays; therefore, they will need to be reshaped on the receiving end. Scalar strings may also be sent, but string arrays are not supported.

Keywords

DEST

The rank of the destination process.

TAG

Optional. The MPI tag for this data (see MPI documentation).

COMM

Optional. A long representing the communicator. If this keyword is not present, the default used is the MPI_COMM_WORLD communicator.

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Send call.

5.29 MPIDL_WAIT

MPIDL_WAIT completes a non-blocking send/receive. It returns a status object (4 element long array) given a valid request (long int) as input. Both sender and receiver must call MPIDL_WAIT with the same request but with *different status*. If both specify the same variable name for status, an error will occur (failing on a corrupted array descriptor).

Syntax

Result = MPIDL_WAIT(Request [, ERROR=error])

Return Value

Returns a status object (4 element long array) given a valid request (long int) as input.

Arguments

Request

The request ID of the message. This ID is created from a call to MPIDL_ISEND or MPIDL_IRECV.

Keywords

ERROR

Optional. A named variable to receive the long integer error code from the MPI_Wait call.

6 Troubleshooting Guide

6.1 Permission denied

"mpiDL seems to start, but I get a 'Permission Denied' error and MPI crashes."

This is probably caused by an error in your configuration of `rsh`. By default, MPI uses `rsh` to communicate between processes. The first step is to check that you can `rsh` from the master host to a slave host. If `rsh` is not enabled on your system, your MPI installation may be mis-configured. Contact your system administrator.

Even if you can `rsh` to a remote host, you may not be able to remotely execute commands, which is the source of the Permission Denied error. The solution is to either create a `.rhosts` file which specifies the hosts to allow, or to have your systems administrator modify the system `/etc/hosts.equiv` file to allow remote program execution using rsh. To enable rsh on a per-user basis, first create a file called `.rhosts` in your home directory. Next change the permissions of the `.rhosts` file by typing:

```
chmod og-rwx .rhosts
```

Finally, add one line for each host to enable, in the form

```
hostname username
```

See the MPICH FAQ at http://www-unix.mcs.anl.gov/mpi/mpich2/faq.htm for more information.

6.2 mpiDL hangs, does not finish

"The program just hangs instead of finishing."

This may be caused either by errors in the IDL code or by prematurely exiting IDL on a remote process. Using the EXIT routine in IDL will cause IDL to exit without finalizing MPI, which will hang all the other processes. Code errors executed on remote processes may be difficult to debug. Sometimes the typical stderr messages which indicate the error are suppressed. One helpful debugging technique is to assign the jobs that the master would normally do to a slave. So change loops which are normally done by the rank 0 process to be done by the rank 1 process, and vice versa.

6.3 Zombie processes

"I am leaving many zombie IDL processes on remote hosts."

If mpiDL exits abnormally (for instance by pressing CTRL-C on the master node) the remote processes may not exit cleanly, leaving zombie IDL processes. These must be manually killed or they may persist.

6.4 Unable to connect to X Windows display

"I get the error 'Unable to connect to X Windows display:' "

This may be caused by a number of problems. Depending on your cluster configuration, it may not be possible to open plot windows from remote processes. Plot commands on slaves are likely to fail giving this error and possibly leaving zombie processes. IDL system errors, such as license manager errors, may also try to open a message window, which can cause this error. Check your IDL license manager (through the program `lmstat` included in your IDL distribution) to ensure that each cluster node is properly licensed. Finally, codes which use the MESSAGE functionality of IDL to issue messages may also attempt to open windows, causing this error.

A simple solution to X Windows display issues may be to use ssh tunneling of X with the command

```
ssh -X yourhost
```

6.5 RSH errors

"My installation complains about not having the correct version of rsh, or that it needs Kerberos."

The MPICH libraries can use `rsh` for communication if configured to do so. The default installation of MPICH uses the system version of `rsh`, located in `/usr/bin/rsh` on most systems. However, some installations of MPICH use the version supplied by the Kerberos packages. If you encounter this problem, you may need to ensure that mpiDL can find the needed (non-Kerberos) rsh in `/usr/bin/rsh`.

6.6 "Connection refused" — RSH server not running

If mpiDL gives error messages about "connection refused" you may need to ensure that the rsh server is installed and running.

6.7 File not found errors

A problem that may be encountered if your environment variables are not correctly set. If this happens, IDL may give errors relating to not finding .so files or DLM files. To correct this, make sure your environment variables (such as `MPIDL_DIR` are pointing to the correct installation of mpiDL for your cluster — for example, if your cluster is running on 64 bit processors, make sure you are not attempting to use the 32 bit version of mpiDL. See the installation section for detailed instructions on setting these varaibles correctly.

6.8 "I've set my environment variables, but it's not working!"

If you are running bash as your shell, you may need to modify how your shell configuration is done. Depending on your shell's setup, you may need to source your `.bashrc` from your `.bash_profile`.