Intel® MPI Library Developer Guide for Linux* OS

Contents

Introducing Intel MPI Library

Documentation for older versions of the Intel® MPI Library are available for download only. For a list of available Intel® Parallel Studio XE documentation by product version, see Download Documentation for Intel Parallel Studio XE. For previous versions of Intel MPI Library documentation, see the Legacy Documentation page.

Find useful information about using the Intel® MPI Library in the following topics:

Introducing Intel MPI Library gives a general introduction for the Intel MPI Library and its components.

Compiling and Linking

Compiling and Linking provides instructions on compiling and linking MPI applications with the Intel MPI Library.

Running MPI Programs

Running Applications describes how to run MPI and hybrid OpenMP*/MPI programs in the command line and through job schedulers.

Analysis and Tuning

Analysis and Tuning discusses the methods for MPI analysis using the built-in Intel MPI Library features and other analyzing tools.

Troubleshooting

Troubleshooting provides the troubleshooting steps for some common issues with the Intel MPI Library.

Examples

Examples provides code examples for Asynchronous Progress Control and Multiple Endpoints Support features.

Introduction

The *Intel® MPI Library Developer Guide* explains how to use the Intel MPI Library in some common usage scenarios. It provides information regarding compiling, running, debugging, tuning, and analyzing MPI applications, as well as troubleshooting information.

This *Developer Guide* helps a user familiar with the message passing interface to start using the Intel MPI Library. For full information, see the *Intel® MPI Library Developer Reference*.

Introducing Intel® MPI Library

The Intel® MPI Library is a multi-fabric message-passing library that implements the Message Passing Interface, version 3.1 (MPI-3.1) specification. It provides a standard library across Intel® platforms that:

- Delivers best in class performance for enterprise, divisional, departmental and workgroup high performance computing. The Intel® MPI Library focuses on improving application performance on Intel® architecture based clusters.
- Enables you to adopt MPI-3.1 functions as your needs dictate.

Conventions and Symbols

The following conventions are used in this document:

This type style	Document names
`This type style`	Commands, arguments, options, file names
THIS_TYPE_STYLE	Environment variables
<this type style>	Variables, or placeholders for actual values
[items]	Optional items
{ item | item }	Selectable items separated by vertical bar(s)

Related Information

To get more information about the Intel® MPI Library, explore the following resources:

- *Intel® MPI Library Release Notes* for updated information on requirements, technical support, and known limitations.
- *Intel® MPI Library Developer Reference* for in-depth knowledge of the product features, commands, options, and environment variables.

For additional resources, see:

- Intel® MPI Library Product Web Site
- Intel® Software Documentation Library
- Intel® Software Products Support

1 Installation and Prerequisites

This section describes the installation process and prerequisite steps.

Installation

New Installations

The Intel® MPI Library is part of the Intel® oneAPI HPC Toolkit (HPC Kit), which includes a variety of tools to help you build, analyze, and deploy HPC applications. You can download the HPC Kit here, or visit the HPC Kit product page for information on downloading a stand-alone or runtime version of the Intel MPI Library.

Upgrading

If you have a previous version of the Intel® MPI Library for Linux* OS installed, you do not need to uninstall it before installing a newer version.

Extract the `l_mpi[-rt]_p_<version>.<package-num>.tar.gz` package by using following command:

```
$ tar -xvzf l_mpi[-rt]_p_<version>.<package-num>.tar.gz
```

This command creates the subdirectory `l_mpi[-rt]_p_<version>.<package-num>`.

To start installation, run `install.sh`. The default installation path for the Intel MPI Library is `/opt/intel/compilers_and_libraries_<version>.<update>.<package-num>/linux/mpi`.

There are two different installations:

- RPM-based installation - this installation requires root password. The product can be installed either on a shared file system or on each node of your cluster.
- Non-RPM installation - this installation does not require root access and it installs all scripts, libraries, and files in the desired directory (usually `$HOME` for the user).

Scripts, include files and libraries for different architectures, are located in different directories. By default, you can find binary files and all needed scripts under the `<install-dir>` directory.

Prerequisite Steps

Before you start using any of the Intel® MPI Library functionality, make sure to establish the proper environment settings:

1. Set up the Intel MPI Library environment by sourcing the setvars.sh script, which is found in your installation directory (by default, `/opt/intel/oneapi/mpi/<version>`).

   NOTE You must run `setvars` at the start of each session. See the Intel oneAPI HPC Toolkit Get Started Guide for more details, including options for automatic setup of environment settings.

2. To run an MPI application on a cluster, the Intel MPI Library needs to know names of all its nodes. Create a text file listing the cluster node names. The format of the file is one name per line, and the lines starting with `#` are ignored. To get the name of a node, use the `hostname` utility.

A sample host file may look as follows:

```
$ cat ./hosts
# This line is ignored
clusternode1
clusternode2
clusternode3
clusternode4
```

3. For communication between cluster nodes, in most cases the Intel MPI Library uses the SSH protocol. You need to establish a passwordless SSH connection to ensure proper communication of MPI processes.

After completing these steps, you are ready to use the Intel MPI Library.

2 Compiling and Linking

This section gives instructions on how to compile and link MPI applications , and provides details on support of different compilers:

- Compiling an MPI program
- Compilers support
- ILP64 Support

Compiling an MPI Program

This topic describes the basic steps required to compile and link an MPI program, using the Intel® MPI Library SDK.

To simplify linking with MPI library files, Intel MPI Library provides a set of compiler wrapper scripts with the `mpi` prefix for all supported compilers. To compile and link an MPI program, do the following:

1. Make sure you have a compiler in your `PATH` environment variable. For example, to check if you have the Intel® C Compiler, enter the command:

```
$ which icc
```

   If the command is not found, add the full path to your compiler into the `PATH`. For Intel® compilers, you can source the script `setvars.[c]sh` to set the required environment variables.
2. Source the `setvars.[c]sh` script in the installation directory to set the proper environment variables for the Intel MPI Library (the default installation directory location is `/opt/intel/oneapi/mpi/<version>`
3. Compile your MPI program using the appropriate compiler wrapper script. For example, to compile a C program with the Intel® C Compiler, use the mpiicc script as follows:

```
$ mpiicc myprog.c -o myprog
```

You will get an executable file `myprog` in the current directory, which you can start immediately. For instructions of how to launch MPI applications, see Running an MPI Program.

> **NOTE** By default, the resulting executable file is linked with the multi-threaded optimized library. If you need to use another library configuration, see Selecting Library Configuration.

For details on the available compiler wrapper scripts, see the Developer Reference.

Compiling an MPI/OpenMP* Program

To compile a hybrid MPI/OpenMP* program using the Intel® compiler, use the `-qopenmp` option. For example:

```
$ mpiicc -qopenmp test.c -o testc
```

This enables the underlying compiler to generate multi-threaded code based on the OpenMP* pragmas in the source. For details on running such programs, refer to Running an MPI/OpenMP* Program.

Adding Debug Information

If you need to debug your application, add the -g option to the compilation command line. For example:

```
$ mpiicc -g test.c -o testc
```

This adds debug information to the resulting binary, enabling you to debug your application. Debug information is also used by analysis tools like Intel® Trace Analyzer and Collector to map the resulting trace file to the source code.

Test MPI Programs

The Intel® MPI Library comes with a set of source files for simple MPI programs that enable you to test your installation. Test program sources are available for all supported programming languages and are located in the `test` directory in your installation directory.

See Also

Intel® MPI Library Developer Reference, section Command Reference > Compiler Commands

Compilers Support

Intel® MPI Library supports the GCC* and Intel® compilers out of the box. It uses binding libraries to provide support for different `glibc` versions and different compilers. These libraries provide C++, Fortran 77, Fortran 90, and Fortran 2008 interfaces.

The following binding libraries are used for GCC* and Intel® compilers:

- `libmpicxx.{a|so}` — for `g++` version 3.4 or higher
- `libmpifort.{a|so}` — for `g77`/`gfortran` interface for GCC and Intel® compilers

Your application gets linked against the correct GCC* and Intel® compilers binding libraries, if you use one of the following compiler wrappers: `mpicc`, `mpicxx`, `mpifc`, `mpif77`, `mpif90`, `mpigcc`, `mpigxx`, `mpiicc`, `mpiicpc`, or `mpiifort`.

For other compilers, PGI* and Absoft* in particular, there is a binding kit that allows you to add support for a certain compiler to the Intel® MPI Library. This binding kit provides all the necessary source files, convenience scripts, and instructions you need, and is located in the `<install_dir>/binding` directory.

To add support for the PGI* C, PGI* Fortran 77, Absoft* Fortran 77 compilers, you need to manually create the appropriate wrapper script (see instructions in the binding kit *Readme*). When using these compilers, keep in mind the following limitations:

- Your PGI* compiled source files must not transfer `long double` entities
- Your Absoft* based build procedure must use the `-g77`, `-B108` compiler options

To add support for the PGI* C++, PGI* Fortran 95, Absoft* Fortran 95, and GNU* Fortran 95 (4.0 and newer) compilers, you need to build extra binding libraries. Refer to the binding kit *Readme* for detailed instructions.

ILP64 Support

The term ILP64 denotes that integer, long, and pointer data entities all occupy 8 bytes. This differs from the more conventional LP64 model, in which only long and pointer data entities occupy 8 bytes while integer entities occupy 4 bytes. More information on the historical background and the programming model philosophy can be found, for example, in `http://www.unix.org/version2/whatsnew/lp64_wp.html`

Intel® MPI Library provides support for the ILP64 model for Fortran applications. To enable the ILP64 mode, do the following:

Use the Fortran compiler wrapper option `-i8` for separate compilation and the `-ilp64` option for separate linking. For example:

```
$ mpiifort -i8 -c test.f
$ mpiifort -ilp64 -o test test.o
```

For simple programs, use the Fortran compiler wrapper option `-i8` for compilation and linkage. Specifying `-i8` will automatically assume the ILP64 library. For example:

```
$ mpiifort -i8 test.f
```

When running the application, use the `-ilp64` option to preload the ILP64 interface. For example:

```
$ mpirun -ilp64 -n 2 ./myprog
```

The following limitations are present in the Intel MPI Library in regard to this functionality:

- Data type counts and other arguments with values larger than 2^{31} - 1 are not supported.
- Special MPI types `MPI_FLOAT_INT`, `MPI_DOUBLE_INT`, `MPI_LONG_INT`, `MPI_SHORT_INT`, `MPI_2INT`, `MPI_LONG_DOUBLE_INT`, `MPI_2INTEGER` are not changed and still use a 4-byte integer field.
- Predefined communicator attributes `MPI_APPNUM`, `MPI_HOST`, `MPI_IO`, `MPI_LASTUSEDCODE`, `MPI_TAG_UB`, `MPI_UNIVERSE_SIZE`, and `MPI_WTIME_IS_GLOBAL` are returned by the functions `MPI_GET_ATTR` and `MPI_COMM_GET_ATTR` as 4-byte integers. The same holds for the predefined attributes that may be attached to the window and file objects.
- Do not use the `-i8` option to compile MPI callback functions, such as error handling functions, or user-defined reduction operations.
- Do not use the `-i8` option with the deprecated functions that store or retrieve the 4-byte integer attribute (for example, `MPI_ATTR_GET`, `MPI_ATTR_PUT`, etc.). Use their recommended alternatives instead (`MPI_COMM_GET_ATTR`, `MPI_COMM_SET_ATTR`, etc).
- If you want to use the Intel® Trace Collector with the Intel MPI Library ILP64 executable files, you must use a special Intel Trace Collector library. If necessary, the `mpiifort` compiler wrapper will select the correct Intel Trace Collector library automatically.
- There is currently no support for C and C++ applications.

3 Running Applications

After you have compiled and linked your MPI application, you are ready to run it. This topic provides instructions on how to run various MPI applications in various modes:

Running Intel® MPI Library in Containers

A container is a self-contained execution environment platform that enable flexibility and portability of your MPI appliction. It lets you package an application and its dependencies in a virtual container that can run on an operating system, such as Linux*.

This guide describes the use of the Intel® MPI Library with the Singularity* container type.

Singularity Containers

Singularity* is a lightweight container model aligned with the needs of High Performance Computing (HPC). Singularity has a built-in support of MPI and allows you to leverage the resources of the host you are on, including HPC interconnects, resource managers, and accelerators.

This chapter provides information on running Intel® MPI Library in a Singularity container built from a recipe file. To run Intel® MPI Library in a Singularity environment, do the following:

1. Make sure you have the following components installed on each machine of a cluster:
   a. Singularity (version not lower than 3.0).
   b. A container including your application.
   c. Intel MPI Library.
2. Create a Singularity recipe file and use it to build a container.
3. Run your MPI application from the Singularity container.

Build a Singularity* Container for an MPI Application

There are several ways to build Singularity* containers described in the Singularity official documentation.

This section demonstrates how to build a container for an MPI application from scratch using recipes. Singularity recipes are files that include software requirements, environment variables, metadata, and other useful details for designing a custom container.

Recipe File Structure

A recipe file consists of the header and sections. The header part defines the core operating system and core packages to be installed. In particular:

- **Bootstrap** - specifies the bootstrap module.
- **OSVersion** - specifies the OS version. Required if only you have specified the %{OSVERSION} variable in **MirrorURL.**

- **MirrorURL** - specifies the URL to use as a mirror to download the OS.
- **Include** - specifies additional packages to be installed into the core OS (optional).

The content of a recipe file is divided into sections that execute commands at different times during the build process. The build process stops if a command fails. The main sections of a recipe are:

- **%help** - provides help information.
- **%setup** - executes commands on the host system outside of the container after the base OS is installed.
- **%post** - executes commands within the container after the base OS has been installed at build time.
- **%environment** - adds environment variables sourced at runtime. If you need environment variables sourced during build time, define them in the %post section.

Build a Container

After the recipe file is created, use it to create a Singularity container. The example below shows how to build a container with default parameters:

```
$ singularity build mpi.img ./Singularity_recipe_mpi
```

Run the Application with a Container

You can choose from three usage models for running your application using a Singularity* container:

1. Everything packed into a single container
2. The Intel MPI Library installed both inside and outside the container
3. The Intel MPI Library outside the container

Usage model 1: Everything packed into a single container

This approach presumes that the Intel® MPI library, target application, and all its dependencies are packed into a container.

Recipe file

```
BootStrap: yum
OSVersion: 8
MirrorURL: http://linux-ftp.jf.intel.com/pub/mirrors/centos/8/BaseOS/$basearch/os/
Include: yum
%environment
source /opt/intel/oneapi/mpi/latest/env/vars.sh
%post
export http_proxy=http://***
yum repolist
yum install -y yum-utils
tee > /tmp/oneAPI.repo << EOF
[oneAPI]
name=Intel(R) oneAPI repository
baseurl=https://yum.repos.intel.com/oneapi
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://yum.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB
EOF
mv /tmp/oneAPI.repo /etc/yum.repos.d
yum install -y intel-oneapi-mpi*
yum install -y sudo wget vi which numactl bzip2 tar gcc hostname util-linux redhat-lsb openssh-
server openssh-clients
```

Launch

When recipe is created, execute the following command:

```
$ singularity exec <container-name> mpirun -n <number-of-processes> -ppn <processes-per-node> -
hostlist <hosts> <application>
```

Usage model 2: The Intel MPI Library installed both inside and outside the container

In this approach, additional dependency on hosts (for example, external mpirun) is required. Each rank is a separate Singularity container instance execution.

Recipe file

```
BootStrap: yum
OSVersion: 8
MirrorURL: http://linux-ftp.jf.intel.com/pub/mirrors/centos/8/BaseOS/$basearch/os/
Include: yum
%environment
source /opt/intel/oneapi/mpi/latest/env/vars.sh
%post
export http_proxy=http://***
yum repolist
yum install -y yum-utils
tee > /tmp/oneAPI.repo << EOF
[oneAPI]
name=Intel(R) oneAPI repository
baseurl=https://yum.repos.intel.com/oneapi
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://yum.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB
EOF
mv /tmp/oneAPI.repo /etc/yum.repos.d
yum install -y intel-oneapi-mpi*
yum install -y sudo wget vi which numactl bzip2 tar gcc hostname util-linux redhat-lsb
```

Launch

When recipe is created, execute the following command:

```
$ mpirun  -n <number-of-processes> -ppn <processes-per-node> -hostlist <hosts> singularity exec
<container-name> <application>
```

Usage model 3: The Intel MPI Library outside the container

In this approach, additional dependency on hosts (for example, external mpirun) is required. Each host has a single Singularity container instance executed for all ranks.

Recipe file

```
BootStrap: yum
OSVersion: 8
MirrorURL: http://linux-ftp.jf.intel.com/pub/mirrors/centos/8/BaseOS/$basearch/os/
Include: yum
%environment
source /opt/intel/oneapi/mpi/latest/env/vars.sh
%post
export http_proxy=http://***
yum repolist
yum install -y yum-utils
tee > /tmp/oneAPI.repo << EOF
[oneAPI]
name=Intel(R) oneAPI repository
```

```
baseurl=https://yum.repos.intel.com/oneapi
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://yum.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB
EOF
mv /tmp/oneAPI.repo /etc/yum.repos.d
yum install -y intel-oneapi-mpi*
yum install -y sudo wget vi which numactl bzip2 tar gcc hostname util-linux redhat-lsb openssh-
server openssh-clients
```

Launch

When recipe is created, execute the following command:

```
$ singularity shell --bind  <path-to-mpi-installation-on-hosts:/mnt> mpirun -n <number-of-
processes> -ppn <processes-per-node> -hostlist <hosts> <application>
```

See Also

Singularity Official Documentation

Selecting a Library Configuration

You can specify a particular configuration of the Intel® MPI Library to be used, depending on your purposes. This can be a library optimized for multi-threading debug or release version with the global or per-object lock.

To specify the configuration, source the `vars.[c]sh` script with the `-i_mpi_library_kind` environment variable and the release, or `debug`, `release_mt`, or `debug_mt` argument. For example:

```
$ . <install-dir>/env/vars.sh -i_mpi_library_kind=release
```

`-i_mpi_library_kind` environment variable sets the library configuration. See the Intel® MPI Library Developer Reference for details.

You can use the following arguments:

Argument	Definition
release	Set this argument to use multi-threaded optimized library (with the global lock). This is the default value
debug	Set this argument to use multi-threaded debug library (with the global lock)
release_mt	Set this argument to use multi-threaded optimized library (with per-object lock for the thread-split model)
debug_mt	Set this argument to use multi-threaded debug library (with per-object lock for the thread-split model)

NOTE You do not need to recompile the application to change the configuration. Source the `vars.[c]sh` script with appropriate arguments before an application launch.

If you want to enable or disable usage of libfabric* from the Intel MPI Library, set the `-i_mpi_ofi_internal` environment variable. See the Intel® MPI Library Developer Reference for details.

Running an MPI Program

Before running an MPI program, place it to a shared location and make sure it is accessible from all cluster nodes. Alternatively, you can have a local copy of your program on all the nodes. In this case, make sure the paths to the program match.

Run the MPI program using the `mpirun` command. The command line syntax is as follows:

```
$ mpirun -n <number-of-processes> -ppn <processes-per-node> -f <hostfile>./myprog
```

For example:

```
$ mpirun -n 4 -ppn 2 -f hosts ./myprog
```

In the command line above:

- `-n` sets the number of MPI processes to launch; if the option is not specified, the process manager pulls the host list from a job scheduler, or uses the number of cores on the machine.
- `-ppn` sets the number of processes to launch on each node; if the option is not specified, processes are assigned to the physical cores on the first node; if the number of cores is exceeded, the next node is used.
- `-f` specifies the path to the host file listing the cluster nodes; alternatively, you can use the `-hosts` option to specify a comma-separated list of nodes; if hosts are not specified, the local node is used.
- `myprog` is the name of your MPI program.

The `mpirun` command is a wrapper around the `mpiexec.hydra` command, which invokes the Hydra process manager. Consequently, you can use all `mpiexec.hydra` options with the `mpirun` command.

For the list of all available options, run `mpirun` with the `-help` option, or see the *Intel® MPI Library Developer Reference*, section *Command Reference > Hydra Process Manager Command*.

> **NOTE** The commands `mpirun` and `mpiexec.hydra` are interchangeable. However, you are recommended to use the `mpirun` command for the following reasons:
>
> - You can specify all `mpiexec.hydra` options with the `mpirun` command.
> - The `mpirun` command detects if the MPI job is submitted from within a session allocated using a job scheduler like PBS Pro* or LSF*. Thus, you are recommended to use `mpirun` when an MPI program is running under a batch scheduler or job manager.

See Also

Controlling Process Placement

Job Schedulers Support

Running an MPI/OpenMP* Program

To run a hybrid MPI/OpenMP* program, follow these steps:

1. Make sure the thread-safe (debug or release, as desired) Intel® MPI Library configuration is enabled (release is the default version). To switch to such a configuration, source`vars.[c]sh` with the appropriate argument. See Selecting Library Configuration for details. For example:

```
$ source vars.sh release
```

2. Set the `I_MPI_PIN_DOMAIN` environment variable to specify the desired process pinning scheme. The recommended value is `omp`:

```
$ export I_MPI_PIN_DOMAIN=omp
```

   This sets the process pinning domain size to be equal to `OMP_NUM_THREADS`. Therefore, if for example `OMP_NUM_THREADS` is equal to `4`, each MPI process can create up to four threads within the corresponding domain (set of logical processors). If `OMP_NUM_THREADS` is not set, each node is treated as a separate domain, which allows as many threads per MPI process as there are cores.

> **NOTE** For pinning OpenMP* threads within the domain, use the Intel® compiler `KMP_AFFINITY` environment variable. See the Intel compiler documentation for more details.

3. Run your hybrid program as a regular MPI program. You can set the `OMP_NUM_THREADS` and `I_MPI_PIN_DOMAIN` variables directly in the launch command. For example:

```
$ mpirun -n 4 -genv OMP_NUM_THREADS=4 -genv I_MPI_PIN_DOMAIN=omp ./myprog
```

See Also

Intel® MPI Library Developer Reference, section *Tuning Reference* > *Process Pinning* > > *Interoperability with OpenMP**.

MPMD Launch Mode

Intel® MPI Library supports the multiple programs, multiple data (MPMD) launch mode. There are two ways to do this.

The easiest way is to create a configuration file and pass it to the `-configfile` option. A configuration file should contain a set of arguments for `mpirun`, one group per line. For example:

```
$ cat mpmd_config-n 1 -host node1 ./io <io_args>
-n 4 -host node2 ./compute <compute_args_1>
-n 4 -host node3 ./compute <compute_args_2$ mpirun -configfile mpmd_config
```

Alternatively, you can pass a set of options to the command line by separating each group with a colon:

```
$ mpirun -n 1 -host node1 ./io <io_args> :\
-n 4 -host node2 ./compute <compute_args_1> :\
-n 4 -host node3 ./compute <compute_args_2>
```

The examples above are equivalent. The `io` program is launched as one process on `node1`, and the `compute` program is launched on `node2` and `node3` as four processes on each.

When an MPI job is launched, the working directory is set to the working directory of the machine where the job is launched. To change this, use the `-wdir <path>`.

Use `-env <var> <value>` to set an environment variable for only one argument set. Using `-genv` instead applies the environment variable to all argument sets. By default, all environment variables are propagated from the environment during the launch.

Fabrics Control

The Intel® MPI Library switched from the Open Fabrics Alliance* (OFA) framework to the Open Fabrics Interfaces* (OFI) framework and currently supports libfabric*.

> **NOTE** The supported fabric environment has changed since Intel® MPI Library 2017 Update 1. The `dapl`, `tcp`, `tmi`, and `ofa` fabrics are now deprecated.

OFI is a framework focused on exporting communication services to applications. OFI is specifically designed to meet the performance and scalability requirements of high-performance computing (HPC) applications running in a tightly coupled network environment. The key components of OFI are application interfaces, provider libraries, kernel services, daemons, and test applications.

Libfabric is a library that defines and exports the user-space API of OFI, and is typically the only software that applications deal with directly. The libfabric's API does not depend on the underlying networking protocols, as well as on the implementation of the particular networking devices, over which it may be implemented. OFI is based on the notion of application centric I/O, meaning that the libfabric library is designed to align fabric services with application needs, providing a tight semantic fit between applications and the underlying fabric hardware. This reduces overall software overhead and improves application efficiency when transmitting or receiving data over a fabric.

For more information, refer to the following topics:

- Selecting Fabrics

Selecting Fabrics

Intel® MPI Library enables you to select a communication fabric at runtime without having to recompile your application. By default, it automatically selects the most appropriate fabric based on your software and hardware configuration. This means that in most cases you do not have to bother about manually selecting a fabric.

However, in certain situations specifying a particular communication fabric can boost performance of your application. The following fabrics are available:

Fabric	Network hardware and software used
shm	Shared memory (for intra-node communication only).
ofi	OpenFabrics Interfaces* (OFI)-capable network fabrics, such as Intel® True Scale Fabric, Intel® Omni-Path Architecture, InfiniBand* and Ethernet (through OFI API).

Use the `I_MPI_FABRICS` environment variable to specify a fabric. The description is available in the *Developer Reference*, section *Tuning Reference* > *Fabrics Control*.

Libfabric* Support

The Intel® MPI Library switched from the Open Fabrics Alliance* (OFA) framework to the Open Fabrics Interfaces* (OFI) framework and currently supports libfabric*.

Enabling Libfabric Support

By default, the script that sets the environmental variables (`vars.[c]sh`) sets the environment to libfabric shipped with the Intel MPI Library. To disable this, use the `I_MPI_OFI_LIBRARY_INTERNAL` environment variable or the `-ofi_internal` option passed to the script:

```
# Do not set the environment to libfabric from the Intel MPI Library.
$ source <install-dir>/env/vars.sh -ofi_internal=0

# Set the environment to libfabric from the Intel MPI Library.
$ source <install-dir>/env/vars.sh -ofi_internal=1

# A short form of -ofi-internal=1
$ source <install-dir>/env/vars.sh
```

NOTE Set the `I_MPI_DEBUG` environment variable to 1 before running an MPI application to see the libfabric version and provider.

Example

```
$ export I_MPI_DEBUG=1
$ mpiexec -n 1 IMB-MPI1 -help
[0] MPI startup(): libfabric version: 1.5.0
[0] MPI startup(): libfabric provider: psm2


...
```

Supported Providers

- libmlx-fi.so

- libpsmx2-fi.so
- librxm-fi.so
- libsockets-fi.so
- libtcp-fi.so
- libverbs-fi.so

See Also

- Intel® MPI LIbrary 2019 over libfabric
- "OFI-Capable Network Fabrics Control" in the Intel MPI Library Developer Reference

OFI* Providers Support

Intel® MPI Library supports mlx, tcp, psm2,psm3, sockets, verbs, and RxM OFI* providers. Each OFI provider is built as a separate dynamic library to ensure that a single libfabric* library can be run on top of different network adapters.

Additionally, Intel MPI Library supports the efa provider, which is not a part of the Intel® MPI Library package and supplied by AWS EFA installer. Please see the efa section below for more details.

> **NOTE** Use the environment variable `FI_PROVIDER` to select a provider. Set the `FI_PROVIDER_PATH` environment variable to specify the path to provider libraries.

To get a full list of environment variables available for configuring OFI, run the following command:

```
$ fi_info  -e
```

mlx

The MLX provider runs over the UCX that is currently available for the Mellanox InfiniBand* hardware.

For more information on using MLX with InfiniBand, see Improve Performance and Stability with Intel MPI Library on InfiniBand.

The following runtime parameters can be used:

Name	Description
`FI_MLX_INJECT_LIMIT`	Sets the control for maximal tinject/inject message sizes.
`FI_MLX_ENABLE_SPAWN`	Enables dynamic processes support.
`FI_MLX_TLS`	Specifies the transports available for the MLX provider.

tcp

The TCP provider is a general purpose provider for the Intel MPI Library that can be used on any system that supports TCP sockets to implement the libfabric API. The provider lets you run the Intel MPI Library application over regular Ethernet, in a cloud environment that has no specific fast interconnect (e.g., GCP, Ethernet empowered Azure*, and AWS* instances) or using IPoIB.

The following runtime parameters can be used:

Name	Description
`FI_TCP_IFACE`	Specifies a particular network interface.
`FI_TCP_PORT_LOW_RANGE` `FI_TCP_PORT_HIGH_RANGE`	Sets the range of ports to be used by the TCP provider for its passive endpoint creation. This is useful when only a range of ports are allowed by the firewall for TCP connections.

psm2

The PSM2 provider runs over the PSM 2.x interface supported by the Intel® Omni-Path Fabric. PSM 2.x has all the PSM 1.x features, plus a set of new functions with enhanced capabilities. Since PSM 1.x and PSM 2.x are not application binary interface (ABI) compatible, the PSM2 provider works with PSM 2.x only and does not support Intel® True Scale Fabric.

The following runtime parameters can be used:

Name	Description
`FI_PSM2_INJECT_SIZE`	Define the maximum message size allowed for `fi_inject` and `fi_tinject` calls. The default value is 64.
`FI_PSM2_LAZY_CONN`	Control the connection mode established between PSM2 endpoints that OFI endpoints are built on top of. When set to 0 (eager connection mode), connections are established when addresses are inserted into the address vector. When set to 1 (lazy connection mode), connections are established when addresses are used the first time in communication. **NOTE** Lazy connection mode may reduce the start-up time on large systems at the expense of higher data path overhead.

psm3

The Intel® Performance Scaled Messaging 3 (Intel® PSM3) provider is a high-performance protocol that provides a low-level communication interface for the Intel® Ethernet Fabric Suite family of products. PSM3 enables mechanisms that are necessary for implementing higher level communication interfaces in parallel environments such as MPI and AI training frameworks.

The Intel® PSM3 interface differs from the Intel® Omni-Path PSM2 interface in the following ways:

- PSM3 includes new features and optimizations for Intel® Ethernet Fabric hardware and processors.
- PSM3 supports only the Open Fabrics Interface (OFI, aka Libfabric). The PSM API is no longer exposed.
- PSM3 includes performance improvements specific to the Intel® Ethernet Fabric Suite.
- PSM3 supports standard Ethernet networks and leverages standard RoCEv2 protocols as implemented by the Intel® Ethernet Fabric Suite NICs.

The following runtime parameters can be used:

Name	Description
`PSM3_NIC`	Specifies the Device Unit number or the RDMA device name (as shown in `ibv_devices`). The specified unit number is relative to the alphabetical sort of the RDMA device names. Unit 0 is the first name. Default: `PSM3_NIC=any`.
`PSM3_RDMA`	Controls the use of RC QPs and RDMA. Options: • 0 - Use only UD QPs. • 1 - Use Rendezvous module for node-to-node level RC QPs for Rendezvous. • 2 - Use user space RC QPs for Rendezvous. • 3 - Use user space RC QPs for eager and Rendezvous. Default: 0
`PSM3_ALLOW_ROUTERS`	Indicates whether endpoints with different IP subnets should be considered accessible.

Name	Description
	• 0 - Consider endpoints with different IPv4 subnets inaccessible. • 1 - Consider all endpoints accessible, even if they have different IPv4 subnets. Default: 0
PSM3_IDENTIFY	Enables verbose output of the PSM3 software version identification, including library location, build date, Rendezvous module API version (if the Rendezvous module is used), process rank IDs, total ranks per node, total ranks in the job, and NIC selected. Options: • 0 - disabled. No output. • 1 - enabled on all processes. • • 1: - enabled only on rank 0 (abbreviation for PSM3_IDENTIFY=1:*:rank0). • 1:pattern - enabled only on processes whose label matches the extended glob pattern. Default: 0

For the full list of controls and details, refer to the Intel® Ethernet Fabric Suite Host Software User Guide.

For more details about Intel® Ethernet Fabric Suite and PSM3 provider, see Intel® Ethernet Fabric Suite documentation.

sockets

The sockets provider is a general purpose provider that can be used on any system that supports TCP sockets. The provider is not intended to provide performance improvements over regular TCP sockets, but rather to allow developers to write, test, and debug application code even on platforms that do not have high-performance fabric hardware. The sockets provider supports all libfabric provider requirements and interfaces.

The following runtime parameter can be used:

Name	Description
FI_SOCKETS_IFACE	Define the prefix or the name of the network interface. By default, it uses any.

verbs

The verbs provider enables applications using OFI to be run over any verbs hardware (InfiniBand*, iWarp*, and so on). It uses the Linux Verbs API for network transport and provides a translation of OFI calls to appropriate verbs API calls. It uses librdmacm for communication management and libibverbs for other control and data transfer operations.

The verbs provider uses RxM utility provider to emulate FI_EP_RDM endpoint over verbs FI_EP_MSG endpoint by default. The verbs provider with FI_EP_RDM endpoint can be used instead of RxM by setting the FI_PROVIDER=^ofi_rxm runtime parameter.

The following runtime parameters can be used:

Name	Description
FI_VERBS_INLINE_SIZE	Define the maximum message size allowed for fi_inject and fi_tinject calls. The default value is 64.
FI_VERBS_IFACE	Define the prefix or the full name of the network interface associated with the verbs device. The default value is ib.

Name	Description
FI_VERBS_MR_CACHE_ENABLE	Enable Memory Registration caching. The default value is 0. Set this environment variable to enable the memory registration cache. **NOTE** Cache usage substantially improves performance, but may lead to correctness issues.

Dependencies

The verbs provider requires libibverbs (v1.1.8 or newer) and librdmacm (v1.0.16 or newer). If you are compiling libfabric from source and want to enable verbs support, it is essential to have the matching header files for the above two libraries. If the libraries and header files are not in default paths, specify them in the CFLAGS, LDFLAGS, and LD_LIBRARY_PATH environment variables.

RxM

The RxM (RDM over MSG) provider (ofi_rxm) is a utility provider that supports FI_EP_RDM endpoint emulated over FI_EP_MSG endpoint of the core provider.

The RxM provider requires the core provider to support the following features:

- MSG endpoints (FI_EP_MSG)
- FI_MSG transport (to support data transfers)
- FI_RMA transport (to support rendezvous protocol for large messages and RMA transfers)
- FI_OPT_CM_DATA_SIZE of at least 24 bytes

The following runtime parameters can be used:

Name	Description
FI_OFI_RXM_BUFFER_SIZE	Define the transmit buffer size/inject size. Messages of smaller size are transmitted via an eager protocol and those above would be transmitted via a rendezvous protocol. Transmitted data is copied up to the specified size. By default, the size is 16k.
FI_OFI_RXM_SAR_LIMIT	Control the RxM SAR (Segmentation and Reassembly) protocol. Messages of greater size are transmitted via rendezvous protocol.
FI_OFI_RXM_USE_SRX	Control the RxM receive path. If the variable is set to 1, the RxM uses Shared Receive Context of the core provider. The default value is 0. **NOTE** Setting this variable to 1 improves memory consumption, but may increase small message latency as a side-effect.

efa

The efa provider enables applications to be run over AWS EFA hardware (Elastic Fabric Adapter).

Please refer to Amazon EC2 User Guide for OFI and Intel® MPI installation on EFA-enabled instances.

Job Schedulers Support

The Intel® MPI Library supports the majority of commonly used job schedulers in the HPC field.

The following job schedulers are supported on Linux* OS:

- Altair* PBS Pro*

- Torque*
- OpenPBS*
- IBM* Platform LSF*
- Parallelnavi* NQS*
- SLURM*
- Univa* Grid Engine*

The Hydra Process manager detects Job Schedulers automatically by checking specific environment variables. These variables are used to determine how many nodes were allocated, which nodes, and the number of processes per tasks.

Altair PBS Pro*, TORQUE*, and OpenPBS*

If you use one of these job schedulers, and `$PBS_ENVIRONMENT` exists with the value `PBS_BATCH` or `PBS_INTERACTIVE`, `mpirun` uses `$PBS_NODEFILE` as a machine file for `mpirun`. You do not need to specify the `-machinefile` option explicitly.

The following is an example of a batch job script:

```
#PBS -l nodes=4:ppn=4
#PBS -q queue_name
cd $PBS_O_WORKDIR
mpirun -n 16 ./myprog
```

IBM Platform LSF*

The IBM Platform LSF* job scheduler is detected automatically if the `$LSB_MCPU_HOSTS` and `$LSF_BINDIR` environment variables are set.

The Hydra process manager uses these variables to determine how many nodes were allocated, which nodes, and the number of processes per tasks. To run processes on the remote nodes, the Hydra process manager uses the `blaunch` utility by default. This utility is provided by the IBM Platform LSF.

The number of processes, the number of processes per node, and node names may be overridden by the usual Hydra options (`-n`, `-ppn`, `-hosts`).

Examples:

```
bsub -n 16 mpirun ./myprog
bsub -n 16 mpirun -n 2 -ppn 1 ./myprog
```

Parallelnavi NQS*

If you use the Parallelnavi NQS job scheduler and the `$ENVIRONMENT`, `$QSUB_REQID`, `$QSUB_NODEINF` options are set, the `$QSUB_NODEINF` file is used as a machine file for `mpirun`. Also, `/usr/bin/plesh` is used as remote shell by the process manager during startup.

Slurm*

The Slurm job scheduler can be detected automatically by `mpirun` and `mpiexec`. Job scheduler detection is enabled in `mpirun` by default and enabled in `mpiexec` if hostnames are not specified. The only prerequisite is setting `I_MPI_PIN_RESPECT_CPUSET=0`.

For autodetection, the Hydra process manger uses these environment variables:

- `SLURM_JOBID`
- `SLURM_NODELIST`
- `SLURM_NNODES`
- `SLURM_NTASKS_PER_NODE` or `SLURM_NTASKS`
- `SLURM_CPUS_PER_TASK`

Using these variables, Hydra can determine which nodes are available, how many nodes were allocated, the number of MPI processes per node, and the domain size per MPI process. `SLURM_NTASKS_PER_NODE` is used for the implicit specification of `I_MPI_PERHOST`, or alternatively `SLURM_NTASKS/SLURM_NNODES`. The value of `SLURM_CPUS_PER_TASK` defines implicitly `I_MPI_PIN_DOMAIN` and overwrites the "auto" default. If some of the the Slurm variables are not defined the corresponding Intel MPI defaults are used. Based on the environment detection it is sufficient to execute the following simple command line under Slurm:

```
export I_MPI_PIN_RESPECT_CPUSET=0; mpirun ./myprog
```

The approach works in standard situations with simple Slurm pinning (for example, only using the Slurm flag `--cpus-per-task`). If a Slurm job requires a more complicated pinning setup (using the Slurm flag `--cpu-bind`) then the process pinning may be incorrect. In this case, full pinning control is gained by launching the MPI run with `srun` or enable Intel MPI Library pinning by setting the `I_MPI_PIN_RESPECT_CPUSET=0` environment variable (see the Developer Reference, "Process Pinning" and "Environmental Variables for Process Pinning"). When using `mpirun`, the required pinning has to be explicitly replicated using `I_MPI_PIN_DOMAIN`.

If the Slurm job scheduler was not detected automatically, you can set the `I_MPI_HYDRA_RMK=slurm` or `I_MPI_HYDRA_BOOTSTRAP=slurm` variables (see the Developer Reference, "Hydra Environment Variables").

To run processes on the remote nodes, Hydra uses the `srun` utility. These environment variables control which utility is used in this case (see the Developer Reference, "Hydra Environment Variables"):

- `I_MPI_HYDRA_BOOTSTRAP`
- `I_MPI_HYDRA_BOOTSTRAP_EXEC`
- `I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS`

You can also launch applications with the `srun` utility without Hydra by setting the `I_MPI_PMI_LIBRARY` environment variable (see the Developer Reference, "Other Environment Variables").

PMI versions currently supported are PMI-1 and PMI-2.

By default, the Intel MPI Library uses per-host process placement provided by the scheduler. This means that the `-ppn` option has no effect. To change this behavior and control process placement through `-ppn` (and related options and variables), set `I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=off`

By default, the Intel MPI Library uses the process pinning provided by Slurm. If the job was launched using `mpirun` or `mpiexec` and some Slurm options for pinning were set, then process pinning may be incorrect. In this case, launch your job with `srun` or enable Intel MPI Library pinning by setting the `I_MPI_PIN_RESPECT_CPUSET=0` environment variable (see the Developer Reference, "Process Pinning" and "Environmental Variables for Process Pinning").

Examples:

```
# Allocate nodes.
salloc --nodes=<number-of-nodes> --partition=<partition> --ntasks-per-node=<number-of-processes-per-node>

# Run your application using Hydra.
mpiexec ./myprog
#or
mpirun ./myprog

# Run your application using srun with the PMI-1 interface.
I_MPI_PMI_LIBRARY=<path-to-libpmi.so>/libpmi.so srun ./myprog

# Run your application using srun with the PMI-2 interface.
I_MPI_PMI_LIBRARY=<path-to-libpmi2.so>/libpmi2.so srun --mpi=pmi2 ./myprog

# Change per-host process placement.
```

```
I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=off mpiexec -n 2 -ppn 1 ./myprog

# Change per-host process placement and hostnames and use srun utility for remote launch.
I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=off mpiexec -n 2 -ppn 1 -hosts host3,host1 -
bootstrap=slurm ./myprog

# Use Intel MPI Library pinning.
I_MPI_PIN_RESPECT_CPUSET=off mpiexec ./myprog

# Use the --cpus-per-task Slurm option in Intel MPI Library pinning.
salloc --cpus-per-task=<cpus-per-task> --nodes=<number-of-nodes> --partition=<partition> --
ntasks-per-node=<number-of-processes-per-node> I_MPI_PIN_RESPECT_CPUSET=off mpiexec ./myprog
#or
I_MPI_PIN_DOMAIN=${SLURM_CPUS_PER_TASK} I_MPI_PIN_RESPECT_CPUSET=off mpiexec ./myprog
```

Univa Grid Engine*

If you use the Univa Grid Engine job scheduler and the `$PE_HOSTFILE` is set, then two files will be generated: `/tmp/sge_hostfile_${username}_$$` and `/tmp/sge_machifile_${username}_$$`. The latter is used as the machine file for `mpirun`. These files are removed when the job is completed.

SIGINT, SIGTERM Signals Intercepting

If resources allocated to a job exceed the limit, most job schedulers terminate the job by sending a signal to all processes.

For example, Torque* sends `SIGTERM` three times to a job and if this job is still alive, `SIGKILL` will be sent to terminate it.

For Univa Grid Engine, the default signal to terminate a job is `SIGKILL`. The Intel MPI Library is unable to process or catch that signal causing `mpirun` to kill the entire job. You can change the value of the termination signal through the following queue configuration:

1. Use the following command to see available queues:

```
$ qconf -sql
```

2. Execute the following command to modify the queue settings:

```
$ qconf -mq <queue_name>
```

3. Find `terminate_method` and change signal to `SIGTERM`.
4. Save queue configuration.

Controlling Per-Host Process Placement

When using a job scheduler, by default the Intel MPI Library uses per-host process placement provided by the scheduler. This means that the `-ppn` option has no effect. To change this behavior and control process placement through `-ppn` (and related options and variables), use the `I_MPI_JOB_RESPECT_PROCESS_PLACEMENT` environment variable:

```
$ export I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=off
```

Controlling Process Placement

Placement of MPI processes over the cluster nodes plays a significant role in application performance. Intel® MPI Library provides several options to control process placement.

By default, when you run an MPI program, the process manager launches all MPI processes specified with `-n` on the current node. If you use a job scheduler, processes are assigned according to the information received from the scheduler.

Specifying Hosts

You can explicitly specify the nodes on which you want to run the application using the `-hosts` option. This option takes a comma-separated list of node names as an argument. Use the `-ppn` option to specify the number of processes per node. For example:

```
$ mpirun  -n 4 -ppn 2 -hosts node1,node2 ./testc
Hello world: rank 0 of 4 running on node1
Hello world: rank 1 of 4 running on node1
Hello world: rank 2 of 4 running on node2
Hello world: rank 3 of 4 running on node2
```

To get the name of a node, use the `hostname` utility.

An alternative to using the `-hosts` option is creation of a host file that lists the cluster nodes. The format of the file is one name per line, and the lines starting with `#` are ignored. Use the `-f` option to pass the file to `mpirun`. For example:

```
$ cat ./hosts
#nodes
node1
node2
$ mpirun  -n 4 -ppn 2 -f hosts ./testc
```

This program launch produces the same output as the previous example.

If the `-ppn` option is not specified, the process manager assigns as many processes to the first node as there are physical cores on it. Then the next node is used. That is, assuming there are four cores on `node1` and you launch six processes overall, four processes are launched on `node1`, and the remaining two processes are launched on `node2`. For example:

```
$ mpirun -n 6 -hosts node1,node2 ./testc
Hello world: rank 0 of 6 running on node1
Hello world: rank 1 of 6 running on node1
Hello world: rank 2 of 6 running on node1
Hello world: rank 3 of 6 running on node1
Hello world: rank 4 of 6 running on node2
Hello world: rank 5 of 6 running on node2
```

NOTE If you use a job scheduler, specifying hosts is unnecessary. The processes manager uses the host list provided by the scheduler.

Using a Machine File

A machine file is similar to a host file with the only difference that you can assign a specific number of processes to particular nodes directly in the file. Contents of a sample machine file may look as follows:

```
$ cat ./machines
node1:2
node2:2
```

Specify the file with the `-machine` option. Running a simple test program produces the following output:

```
$ mpirun  -machine machines ./testc
Hello world: rank 0 of 4 running on node1
Hello world: rank 1 of 4 running on node1
Hello world: rank 2 of 4 running on node2
Hello world: rank 3 of 4 running on node2
```

Using Argument Sets

Argument sets are unique groups of arguments specific to a particular node. Combined together, the argument sets make up a single MPI job. You can provide argument sets on the command line, or in a configuration file. To specify a node, use the `-host` option.

On the command line, argument sets should be separated by a colon ':'. Global options (applied to all argument sets) should appear first, and local options (applied only to the current argument set) should be specified within an argument set. For example:

```
$ mpirun  -genv I_MPI_DEBUG=2 -host node1 -n 2 ./testc  : -host node2 -n 2 ./testc
```

In the configuration file, each argument set should appear on a new line. Global options should appear on the first line of the file. For example:

```
$ cat ./config
-genv I_MPI_DEBUG=2-host node1 -n 2 ./testc
-host node2 -n 2 ./testc
```

Specify the configuration file with the `-configfile` option:

```
$ mpirun -configfile config
Hello world: rank 0 of 4 running on node1
Hello world: rank 1 of 4 running on node1
Hello world: rank 2 of 4 running on node2
Hello world: rank 3 of 4 running on node2
```

See Also

Controlling Process Placement with the Intel® MPI Library (online article)

Job Schedulers Support

Java* MPI Applications Support

The Intel® MPI Library provides an experimental feature to enable support for Java MPI applications. Java bindings are available for a subset of MPI-2 routines. For a full list of supported routines, refer to the Developer Reference, section *Miscellaneous* > *Java Bindings for MPI-2 Routines*.

Running Java MPI applications

Follow these steps to set up the environment and run your Java MPI application:

1. Source `mpivars.sh` from the Intel® MPI Library package to set up all required environment variables, including `LIBRARY_PATH` and `CLASSPATH`.
2. Build your Java MPI application as usual.
3. Update `CLASSPATH` with the path to the `jar` application or pass it explicitly with the `-cp` option of the `java` command.
4. Run your Java MPI application using the following command:

```
$ mpirun <options> java <app>
```

where:

- *<options>* is a list of `mpirun` options
- *<app>* is the main class of your Java application

For example:

```
$ mpirun -n 8 -ppn 1 -f ./hostfile java mpi.samples.Allreduce
```

Development Recommendations

You can use the following tips when developing Java* MPI applications:

- To reduce memory footprint, you can use Java direct buffers as buffer parameters of collective operations in addition to using Java arrays. This approach allows you to allocate the memory out of the JVM heap and avoid additional memory copying when passing the pointer to the buffer from JVM to the native layer.
- When you create Java MPI entities such as `Group`, `Comm`, `Datatype`, and similar, memory is allocated on the native layer and is not tracked by the garbage collector. Therefore, this memory must be released explicitly. Pointers to the allocated memory are stored in a special pool and can be deallocated using one of the following methods:
  - `entity.free()`: frees the memory backing the `entity` Java object, which can be an instance of `Comm`, `Group`, etc.
  - `AllocablePool.remove(entity)`: frees the memory backing the `entity` Java object, which can be an instance of `Comm`, `Group`, etc.
  - `AllocablePool.cleanUp()`: explicitly deallocates the memory backing all Java MPI objects created by that moment.
  - `MPI.Finalize()`: implicitly deallocates the memory backing all Java MPI objects and that has not been explicitly deallocated by that moment.

4 Debugging Applications

This section explains how to debug MPI applications using the debugger tools:

Debugging

The Intel® MPI Library supports the GDB* and Allinea* DDT debuggers for debugging MPI applications. Before using a debugger, make sure you have the application debug symbols available. To generate debug symbols, compile your application with the `-g` option.

GDB*: The GNU Project Debugger

Use the following command to launch the GDB debugger with Intel® MPI Library:

```
$ mpirun -gdb -n 4 ./testc
```

You can work with the GDB debugger as you usually do with a single-process application. For details on how to work with parallel programs, see the GDB documentation at http://www.gnu.org/software/gdb/.

You can also attach to a running job with:

```
$ mpirun -n 4 -gdba <pid>
```

Where `<pid>` is the process ID for the running MPI process.

DDT* Debugger

You can debug MPI applications using the Allinea DDT* debugger. Intel does not provide support for this debugger, you should obtain the support from Allinea. According to the DDT documentation, DDT supports the Express Launch feature for the Intel MPI Library. You can debug your application as follows:

```
$ ddt mpirun -n <number-of-processes> [<other-mpirun-arguments>] <executable>
```

If you have issues with the DDT debugger, refer to the DDT documentation for help.

See Also

Using -gtool for Debugging

Using -gtool for Debugging

The `-gtool` runtime option can help you with debugging, when attaching to several processes at once. Instead of attaching to each process individually, you can specify all the processes in a single command line. For example:

```
$ mpirun -n 16 -gtool "gdb:3,5,7-9=attach" ./myprog
```

The command line above attaches the GNU* Debugger (GDB*) to processes 3, 5, 7, 8 and 9.

See Also

Intel® MPI Library Developer Reference, section *Command Reference* > *Hydra Process Manager Command* > *Global Options* > *gtool Options*

5 Analysis and Tuning

Intel® MPI Library provides a variety of options for analyzing MPI applications. Some of these options are available within the Intel MPI Library, while some require additional analysis tools. For such tools, Intel MPI Library provides compilation and runtime options and environment variables for easier interoperability.

Displaying MPI Debug Information

The `I_MPI_DEBUG` environment variable provides a convenient way to get detailed information about an MPI application at runtime. You can set the variable value from 0 (the default value) to 1000. The higher the value, the more debug information you get. For example:

```
$ mpirun  -genv I_MPI_DEBUG=2 -n 2 ./testc
-genv I_MPI_DEBUG=2 -n 2 testc[1] MPI startup(): Internal info: pinning initialization was
done[0] MPI startup(): Internal info: pinning initialization was done...
```

> **NOTE** High values of `I_MPI_DEBUG` can output a lot of information and significantly reduce performance of your application. A value of `I_MPI_DEBUG=5` is generally a good starting point, which provides sufficient information to find common errors.

By default, each printed line contains the MPI rank number and the message. You can also print additional information in front of each message, like process ID, time, host name and other information, or exclude some information printed by default. You can do this in two ways:

- Add the '+' sign in front of the debug level number. In this case, each line is prefixed by the string *<rank>#<pid>@<hostname>*. For example:

```
$ mpirun -genv I_MPI_DEBUG=+2 -n 2 ./testc
[0#3520@clusternode1] MPI startup(): Multi-threaded optimized library
...
```

  To exclude any information printed in front of the message, add the '–' sign in a similar manner.
- Add the appropriate flag after the debug level number to include or exclude some information. For example, to include time but exclude the rank number:

```
$ mpirun -genv I_MPI_DEBUG=2,time,norank -n 2 ./testc
11:59:59 MPI startup(): Multi-threaded optimized library
...
```

  For the list of all available flags, see the description of `I_MPI_DEBUG` in the *Developer Reference*.

To redirect the debug information output from `stdout` to `stderr` or a text file, use the `I_MPI_DEBUG_OUTPUT` environment variable:

```
$ mpirun -genv I_MPI_DEBUG=2 -genv I_MPI_DEBUG_OUTPUT=/tmp/debug_output.txt -n 2 ./testc
```

Note that the output file name should not be longer than 256 symbols.

See Also

Intel® MPI Library Developer Reference, section *Miscellaneous* > *Other Environment Variables* > *I_MPI_DEBUG*

Tracing Applications

The Intel® MPI Library provides a variety of options for analyzing MPI applications. Some of these options are available within the Intel MPI Library, while some require additional analysis tools. For these tools, the Intel MPI Library provides compilation and runtime options and environment variables for easier interoperability.

The Intel MPI Library is tightly integrated with the Intel® Trace Analyzer and Collector, which enables you to analyze and debug MPI applications. The Intel MPI Library has several compile- and runtime options to simplify the application analysis. Apart from the Intel Trace Analyzer and Collector, there is also a tool called Application Performance Snapshot intended for a higher level MPI analysis.

Intel Trace Analyzer and Collector is available as standalone software and as part of the Intel® oneAPI HPC Toolkit. Before proceeding to the next steps, make sure you have the product installed.

High-Level Performance Analysis

For a high-level application analysis, Intel provides a lightweight analysis tool Application Performance Snapshot (APS), which can analyze MPI and non-MPI applications. The tool provides general information about the application, such as MPI and OpenMP* utilization time and load balance, MPI operations usage, memory and disk usage, and other information. This information enables you to get a general idea about the application performance and identify spots for a more thorough analysis.

Follow these steps to analyze an application with the APS:

1. Set up the environment for the compiler, Intel MPI Library, and APS.

```
$ source <install-dir>/setvars.sh
```

```
$ source<install-dir>/vtune/<version>/env/vars.sh
```

2. Run your application with the `-aps` option of `mpirun`:

```
$ mpirun -n 4 -aps ./myprog
```

   APS will generate a directory with the statistics files `aps_result_<date>-<time>`.

3. Launch the `aps-report` tool and pass the generated statistics to the tool:

```
$ aps-report ./aps_result_<date>-<time>
```

   You will see the analysis results printed in the console window. Also, APS will generate an HTML report `aps_report_<date>_<time>.html` containing the same information.

For more details, refer to the *Application Performance Snapshot User Guide*.

Trace an Application

To analyze an application with the Intel Trace Analyzer and Collector, first you need generate a trace file of your application, and then open this file in Intel® Trace Analyzer to analyze communication patterns, time utilization, etc. Tracing is performed by preloading the Intel Trace Collector profiling library at runtime, which intercepts all MPI calls and generates a trace file. Intel MPI Library provides the `-trace` (`-t`) option to simplify this process.

Complete the following steps:

1. Set up the environment for the Intel MPI Library, and Intel Trace Analyzer and Collector.

```
$ source <mpi-install-dir>/env/vars.sh
```

```
$ source <itac-install-dir>/env/vars.sh
```

2. Trace your application with the Intel Trace Collector:

```
$ mpirun -trace -n 4 ./myprog
```

   As a result, a trace file `.stf` is generated. For the example above, it is `myprog.stf`.

3. Analyze the application with the Intel Trace Analyzer:

```
$ traceanalyzer ./myprog.stf &
```

The workflow above is the most common scenario of tracing with the Intel Trace Collector. For other tracing scenarios, see the Intel Trace Collector documentation.

See Also

Application Performance Snapshot User Guide

Intel Trace Collector User and Reference Guide

Interoperability with Other Tools Through -gtool

To simplify interoperability with other analysis tools, Intel® MPI Library provides the `-gtool` option (also available as the `I_MPI_GTOOL` environment variable). Using the `-gtool` option, you can analyze specific MPI processes with VTune™ Profiler, Intel® Advisor, Valgrind,* and other tools through the `mpiexec.hydra` or `mpirun` commands.

The Intel Advisor CLI uses `-gtool` to analyze MPI applications. For sample command lines and example scenarios, including the Weather Research and Forecasting (WRF) Model, see Analyze Vectorization and Memory Aspects of an MPI Application.

If you are not using the `-gtool` option, to analyze an MPI process with, for example, VTune Profiler, you must specify the relevant command in the corresponding argument set:

```
$ mpirun -n 3 ./myprog : -n 1 vtune -c advanced-hotspots -r ah -- ./myprog
```

The `-gtool` option allows you to specify a single analysis command for all argument sets (separated by colons ':') at once. Even though it is allowed to use `-gtool` within a single argument set, it is not recommended to use it in several sets at once and combine the two analysis methods (with `-gtool` and argument sets).

For example, to analyze processes 3, 5, 6, and 7 with the VTune Profiler, you can use the following command line:

```
$ mpirun -n 8 -gtool "vtune -collect hotspots -r result:3,5-7" ./myprog
```

The `-gtool` option also provides several methods for finer process selection. For example, you can easily analyze only one process on each host, using the `exclusive` launch mode:

```
$ mpirun -n 8 -ppn 4 -hosts node1,node2 -gtool "vtune -collect hotspots -r
result:all=exclusive" ./myprog
```

You can also use the `-gtoolfile` option to specify `-gtool` parameters in a configuration file. All the same rules apply. Additionally, you can separate different command lines with section breaks.

For example, if `gtool_config_file` contains the following settings:

```
env VARIABLE1=value1 VARIABLE2=value2:3,5,7-9; env VARIABLE3=value3:0,11
env VARIABLE4=value4:1,12
```

The following command sets `VARIABLE1` and `VARIABLE2` for processes 3, 5, 7, 8, and 9 and sets `VARIABLE3` for processes 0 and 11, while `VARIABLE4` is set for processes 1 and 12:

```
$ mpirun -n 16 -gtoolfile gtool_config_file a.out
```

Using `-gtool` for Debugging

The `-gtool` runtime option can help you with debugging, when attaching to several processes at once. Instead of attaching to each process individually, you can specify all the processes in a single command line. For example:

```
$ mpirun -n 16 -gtool "gdb:3,5,7-9=attach" ./myprog
```

The command line above attaches the GNU* Debugger (GDB*) to processes 3, 5, 7, 8 and 9.

> **NOTE** Do not use the `-gdb` and `-gtool` options together. Use one option at a time.

See Also

- Analyze Vectorization and Memory Aspects of an MPI Application
- Intel® MPI Library Developer Reference, gtool Options

MPI Tuning

The Intel® MPI Library includes the `mpitune` tuning utility, which allows you to automatically adjust Intel MPI Library parameters, such as collective operation algorithms, to your cluster configuration or application. The tuner iteratively launches a benchmarking application with different configurations to measure performance and stores the results of each launch. Based on these results, the tuner generates optimal values for the parameters that are being tuned.

> **NOTE** The `mpitune` usage model changed in the 2018 release. Tuning parameters should be specified in configuration files rather than as command-line options.

Configuration File Format

All tuner parameters should be specified in two configuration files, passed to the tuner with the `--config-file` option. A typical configuration file consists of the main section, specifying generic options, and search space sections for specific library parameters (for example, for specific collective operations). Configuration files differ in mode and dump-file fields only. To comment a line, use the hash symbol #.

You can also specify MPI options to simplify `mpitune` usage. MPI options are useful for Intel® MPI Benchmarks that have special templates for `mpitune` located at `<install-dir>/etc/tune_cfg`. The templates require no changes in configuration files to be made.

For example, to tune the `Bcast` collective algorithm, use the following option:

```
$ mpitune -np 2 -ppn 2 -hosts HOST1 -m analyze -c <path-to-Bcast.cfg>
```

Experienced users can change configuration files to use this option for other applications.

Output Format

The tuner presents results in a JSON tree view (since the 2019 release), where the `comm_id=-1` layer is added automatically for each tree:

```
{
  "coll=Reduce": {
    "ppn=2": {
      "comm_size=2": {
        "comm_id=-1": {
```

```
          "msg_size=243": {
            "REDUCE=8": {}
          },
          "msg_size=319": {
            "REDUCE=11": {}
          },
          "msg_size=8192": {
            "REDUCE=8": {}
          },
          "msg_size=28383": {
            "REDUCE=9": {}
          },
          "msg_size=-1": {
            "REDUCE=1": {}
          }
        }
      }
    }
  }
}
```

To add the resulting JSON tree to the library, use the `I_MPI_TUNING` environment variable.

Old Output Format

The old output format is only valid for Intel MPI Library 2018 and prior versions:

```
I_MPI_ADJUST_BCAST=2:0-0;1:1-64;2:65-509;1:510-8832;3:8833-0
```

Use the resulting variable value with the application launch to achieve performance gain.

See Also

For details on `mpitune` configuration options, refer to the Developer Reference section *Command Reference > mpitune*.

6 Troubleshooting

This section provides the troubleshooting information on typical MPI failures with corresponding output messages and behavior when a failure occurs.

If you encounter errors or failures when using the Intel® MPI Library, take the following general troubleshooting steps first:

1. Check the *System Requirements* section and the *Known Issues* section in the *Intel® MPI Library Release Notes*.
2. Check accessibility of the hosts. Run a simple non-MPI application (for example, the `hostname` utility) on the problem hosts using `mpirun`. For example:

```
$ mpirun -ppn 1 -n 2 -hosts node01,node02 hostname
node01
node02
```

   This may help reveal an environmental problem (such as the MPI remote access mechanism is not configured properly), or a connectivity problem (such as unreachable hosts).
3. Run the MPI application with debug information enabled: set the environment variables `I_MPI_DEBUG=6` and/or `I_MPI_HYDRA_DEBUG=on`. Increase the integer value of debug level to get more information. This action helps narrow down to the problematic component.
4. If you have the availability, download and install the latest version of Intel MPI Library from the official product page and check if your problem persists.
5. If the problem still persists, you can submit a ticket via the Support page, or ask experts on the community forum.

Error Message: Bad Termination

NOTE: The values in the tables below may not reflect the exact node or MPI process where a failure can occur.

Case 1

Error Message

```
===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   RANK 1 PID 27494 RUNNING AT node1
=   KILLED BY SIGNAL: 11 (Segmentation fault)
===================================================================================
```

or:

```
===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   RANK 1 PID 27494 RUNNING AT node1
=   KILLED BY SIGNAL: 8 (Floating point exception)
===================================================================================
```

Cause

One of MPI processes is terminated by a signal (for example, `Segmentation fault` or `Floating point exception`) on the `node01`.

Solution

Find the reason of the MPI process termination. It can be the out-of-memory issue in case of `Segmentation fault` or division by zero in case of `Floating point exception`.

Case 2

Error Message

```
===================================================================================
= BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
= RANK 1 PID 20066 RUNNING AT node01
= KILLED BY SIGNAL: 9 (Killed)
===================================================================================
```

Cause

One of MPI processes is terminated by a signal (for example, `SIGTERM` or `SIGKILL`) on the `node01` due to:

- the host reboot;
- an unexpected signal received;
- out-of-memory manager (OOM) errors;
- killing by the process manager (if another process was terminated before the current process);
- job termination by the Job Scheduler (PBS Pro*, SLURM*) in case of resources limitation (for example, walltime or cputime limitation).

Solution

1. Check the system log files.
2. Try to find the reason of the MPI process termination and fix the issue.

Error Message: No such file or Directory

Error Message

```
[proxy:0:0@node1] HYD_spawn
(../../../../../src/pm/i_hydra/libhydra/spawn/hydra_spawn.c:113): execvp error on file {path to
binary file}/{binary file} (No such file or directory)
```

Cause

Wrong path to the binary file or the binary file does not exist on the `node01`. The name of the binary file is misprinted or the shared space cannot be reached.

Solution

Check the name of the binary file and check if the shared path is available across all the nodes.

Error Message: Permission Denied

Case 1

Error Message

```
[proxy:0:0@node1] HYD_spawn
(../../../../../src/pm/i_hydra/libhydra/spawn/hydra_spawn.c:113): execvp error on file {path to
binary file}/{binary file} (Permission denied)
```

Cause

You do not have permissions to execute the binary file.

Solution

Check your execute permissions for {binary file} and for folders in {path to binary file}.

Case 2

Error Message

```
[proxy:0:0@node1] HYD_spawn
(../../../../../src/pm/i_hydra/libhydra/spawn/hydra_spawn.c:113): execvp error on file {path to
binary file}/{binary file} (Permission denied)
```

Cause

You exceeded the limitation of 16 groups on Linux* OS.

Solution

Try reducing the number of groups.

Error Message: Fatal Error

Case 1

Error Message

```
Abort(1094543) on node 0 (rank 0 in comm 0): Fatal error in PMPI_Init: Other MPI error, error
stack:
MPIR_Init_thread(653).....:
MPID_Init(860)............:
MPIDI_NM_mpi_init_hook(698): OFI addrinfo() failed
(ofi_init.h:698:MPIDI_NM_mpi_init_hook:No data available)
```

Cause

The current provider cannot be run on these nodes. The MPI application is run over the psm2 provider on the non-Intel® Omni-Path card or over the verbs provider on the non-InfiniBand*, non-iWARP, or non-RoCE card.

Solution

1. Change the provider or run MPI application on the right nodes. Use fi_info to get information about the current provider.
2. Check if services are running on nodes (opafm for Intel® Omni-Path and opensmd for InfiniBand).

Case 2

Error Message

```
Abort(6337423) on node 0 (rank 0 in comm 0): Fatal error in PMPI_Init_thread:
Other MPI error, error stack:
…
MPIDI_OFI_send_handler(704).............: OFI tagged inject failed
(ofi_impl.h:704:MPIDI_OFI_send_handler:Transport endpoint is not connected)
```

Cause

OFI transport uses IP interface without access to remote ranks.

Solution

Set FI_SOCKET_IFACE If the socket provider is used or FI_TCP_IFACE and FI_VERBS_IFACE in case of TCP and verbs providers, respectively. To retrieve the list of configured and active IP interfaces, use the ifconfig utility.

Case 3

Error Message

```
Abort(6337423) on node 0 (rank 0 in comm 0): Fatal error in PMPI_Init_thread:
Other MPI error, error stack:
…
MPIDI_OFI_send_handler(704).............: OFI tagged inject failed
(ofi_impl.h:704:MPIDI_OFI_send_handler:Transport endpoint is not connected)
```

Cause

Ethernet is used as an interconnection network.

Solution

Run `FI_PROVIDER = sockets mpirun …` to overcome this problem.

Error Message: Bad File Descriptor

Error Message

```
[mpiexec@node00] HYD_sock_write (../../../../../src/pm/i_hydra/libhydra/sock/hydra_sock_intel.c:
353): write error (Bad file descriptor)
[mpiexec@node00] cmd_bcast_root (../../../../../src/pm/i_hydra/mpiexec/mpiexec.c:147): error
sending cwd cmd to proxy
[mpiexec@node00] stdin_cb (../../../../../src/pm/i_hydra/mpiexec/mpiexec.c:324): unable to send
response downstream
[mpiexec@node00] HYDI_dmx_poll_wait_for_event (../../../../../src/pm/i_hydra/libhydra/demux/
hydra_demux_poll.c:79): callback returned error status
[mpiexec@node00] main (../../../../../src/pm/i_hydra/mpiexec/mpiexec.c:2064): error waiting for
event
```

or:

```
[mpiexec@host1] wait_proxies_to_terminate (../../../../../src/pm/i_hydra/mpiexec/intel/
i_mpiexec.c:389): downstream from host host2 exited with status 255
```

Cause

The remote `hydra_pmi_proxy` process is unavailable due to:

- the host reboot;
- an unexpected signal received;
- out-of-memory manager (OOM) errors;
- job termination by the Job Scheduler (PBS Pro*, SLURM*) in case of resources limitation (for example, walltime or cputime limitation).

Solution

1. Check the system log files.
2. Try to find the reason of the `hydra_pmi_proxy` process termination and fix the issue.

Error Message: Too Many Open Files

Error Message

```
[proxy:0:0@host1] HYD_spawn (../../../../../src/pm/i_hydra/libhydra/spawn/intel/hydra_spawn.c:
57): pipe error (Too many open files)
[proxy:0:0@host1] launch_processes (../../../../../src/pm/i_hydra/proxy/proxy.c:509): error
creating process
[proxy:0:0@host1] main (../../../../../src/pm/i_hydra/proxy/proxy.c:860): error
launching_processes
```

Cause

Too many processes per node are launched on Linux* OS.

Solution

Specify fewer processes per node by the `-ppn` option or the `I_MPI_PERHOST` environment variable.

Problem: High Memory Consumption Readings

Problem

The Intel® MPI Library's virtual memory consumption appears to be unreasonably high, when using basic profiling tools.

Cause

The Intel MPI Library uses a shared virtual memory region within a node, which is mapped into each process's virtual address space. As a result, when virtual memory consumption is queried for any of the ranks, the size of the entire memory region is reported. All ranks on a node report the size of the same shared region. Most of the virtual memory measuring tools are unaware of this shared virtual memory region, and as a result present incorrect data in this context.

Solution

Use `/proc/<PID>/smaps` files, which provide detailed information about a process's memory consumption. For more information, see Evaluating Virtual Memory Consumption in Intel® MPI Library.

Problem: MPI Application Hangs

Problem

MPI application hangs without any output.

Case 1

Cause

Application does not use MPI in a correct way.

Solution

Run your MPI application with the `-check_mpi` option to perform correctness checking. The correctness checker is specifically designed to find MPI errors, and provides tight integration with the Intel® MPI Library. In case of a deadlock, the checker will set up a one-minute timeout and show the state of each rank.

For more information, refer to this page.

Case 2

Cause

The remote service (for example, SSH) is not running on all nodes or it is not configured properly.

Solution

Check the state of the remote service on the nodes and connection to all nodes.

Case 3

Cause

The Intel® MPI Library runtime scripts are not available, so the shared space cannot be reached.

Solution

Check if the shared path is available across all the nodes.

Case 4

Cause

Different CPU architectures are used in a single MPI run.

Solution

Set `export I_MPI_PLATFORM=<arch>` , where `<arch>` is the oldest platform you have, for example `skx`. Note that usage of different CPU architectures in a single MPI job negatively affects application performance, so it is recommended not to mix different CPU architecture in a single MPI job.

Problem: Password Required

Problem

Password required.

Cause

The Intel® MPI Library uses SSH mechanism to access remote nodes. SSH requires password and this may cause the MPI application hang.

Solution

1. Check the SSH settings.
2. Make sure that the passwordless authorization by public keys is enabled and configured.

Problem: Cannot Execute Binary File

Problem

Cannot execute a binary file.

Cause

Wrong format or architecture of the binary executable file.

Solution

Check the accuracy of the binary file and command line options.

Problem: MPI limitation for Docker*

Problem

The command fails with the following message:

```
[root@n1 /]# I_MPI_DEBUG=12 mpirun -n 2 -ppn 1 -env I_MPI_PIN_DOMAIN socket IMB-MPI1 bcast
impi_shm_heap_init(): mbind failed (p=0x7f3078b0e000, size=536870912)
impi_shm_heap_init(): mbind failed (p=0x7f9b808bc000, size=536870912)
```

Cause

MPI has a limitation on the `dev/shm` area. It should be not less than 4GB for a node with 2 sockets. By default, the Docker* container set 64MB, which is not enough.

Solution

1. Make sure the problem is the small size of `shm` area:

```
$df -h /dev/shm
root@n1 /]# df -h /dev/shm
Filesystem      Size Used Avail Use% Mounted on
shm             4.0G   0  4.0G   0%  /dev/shm
```

2. If it is true, restart Docker using the following command:

```
docker run --shm-size=4gb …
```

7 Additional Supported Features

Intel® MPI Library supports the following features:

- Asynchronous Progress Control
- Multiple Endpoints Support

Asynchronous Progress Control

Intel® MPI Library supports asynchronous progress threads that allow you to manage communication in parallel with application computation and, as a result, achieve better communication/computation overlapping. This feature is supported for the `release_mt` and `debug_mt` versions only.

> **NOTE** Asynchronous progress has a full support for MPI point-to-point operations, blocking collectives, and a partial support for non-blocking collectives (`MPI_Ibcast`, `MPI_Ireduce`, and `MPI_Iallreduce`).

To enable asynchronous progress, pass 1 to the `I_MPI_ASYNC_PROGRESS` environment variable. You can define the number of asynchronous progress threads by setting the `I_MPI_ASYNC_PROGRESS_THREADS` environment variable. The `I_MPI_ASYNC_PROGRESS_ID_KEY` variable sets the MPI info object key that is used to define the progress `thread_id` for a communicator.

Setting the `I_MPI_ASYNC_PROGRESS_PIN` environment variable allows you to control the pinning of the asynchronous progress threads. In case of N progress threads per process, the first N logical processors from the list will be assigned to the threads of the first local process, while the next N logical processors - to the second local process and so on.

Example

For example, If the thread affinity is `0,1,2,3` with 2 progress threads per process and 2 processes per node, then the progress threads of the first local process are pinned to logical processors `0` and `1`, while the progress threads of the second local process are pinned to processors `2` and `3`.

The code example is available below or in the `async_progress_sample.c` file in the `doc/examples` subdirectory of the package.

See Also

async_progress_sample.c

For more information on environment variables, refer to the Intel® MPI Library Developer Reference, section *Environment Variable Reference > Environment Variable Reference for Asynchronous Progress Control.*

Multiple Endpoints Support

The traditional MPI/OpenMP* threading model has certain performance issues. Thread safe access to some MPI objects, such as requests or communicators, requires an internal synchronization between threads; the performance of the typical hybrid application, which uses MPI calls from several threads per rank, is often lower than expected.

The PSM2 Multiple Endpoints (Multi-EP) support in the Intel® MPI Library makes it possible to eliminate most of the cross-thread synchronization points in the MPI workflow, at the cost of some limitations on what is allowed by the standard `MPI_THREAD_MULTIPLE` thread support level. The Multi-EP support, implemented

with MPI_THREAD_SPLIT (thread-split) programming model, implies several requirements to the application program code to meet, and introduces a few runtime switches. These requirements, limitations, and usage rules are discussed in the sections below.

- MPI_THREAD_SPLIT Programming Model
- Threading Runtimes Support
- Program Examples
- Code Change Guide

MPI_THREAD_SPLIT Programming Model

This feature is supported for the release_mt and debug_mt versions only.

The communication patterns that comply with the thread-split model must not allow cross-thread access to MPI objects to avoid thread synchronization and must disambiguate message matching, so that threads could be separately addressed and not more than one thread could match the message at the same time. Provided that, the user must notify the Intel MPI Library that the program complies with thread-split model, that is, it is safe to apply the high performance optimization.

Each MPI_THREAD_SPLIT-compliant program can be executed correctly with a thread-compliant MPI implementation under MPI_THREAD_MULTIPLE, but not every MPI_THREAD_MULTIPLE-compliant program follows the MPI_THREAD_SPLIT model.

This model allows MPI to apply optimizations that would not be possible otherwise, such as binding specific hardware resources to concurrently communicating threads and providing lockless access to MPI objects.

Since MPI_THREAD_SPLIT is a non-standard programming model, it is disabled by default and can be enabled by setting the environment variable I_MPI_THREAD_SPLIT. If enabled, the threading runtime control must also be enabled to enable the programming model optimizations (see Threading Runtimes Support).

Setting the I_MPI_THREAD_SPLIT variable does not affect behavior at other threading levels such as SINGLE and FUNNELED. To make this extension effective, request the MPI_THREAD_MULTIPLE level of support at MPI_Init_thread().

NOTE: Thread-split model has support for MPI point-to-point operations and blocking collectives.

MPI_THREAD_SPLIT Model Description

As mentioned above, an MPI_THREAD_SPLIT-compliant program must be at least a thread-compliant MPI program (supporting the MPI_THREAD_MULTIPLE threading level). In addition to that, the following rules apply:

1. Different threads of a process must not use the same communicator concurrently.
2. Any request created in a thread must not be accessed by other threads, that is, any non-blocking operation must be completed, checked for completion, or probed in the same thread.
3. Communication completion calls that imply operation progress such as MPI_Wait(), MPI_Test() being called from a thread don't guarantee progress in other threads.

The model implies that each process thread has a distinct logical thread number thread_id. thread_id must be set to a number in the range *0* to NT-1, where NT is the number of threads that can be run concurrently. thread_id can be set implicitly, or your application can assign it to a thread. Depending on the assignment method, there are two usage submodels:

1. Implicit model: both you and the MPI implementation know the logical thread number in advance via a deterministic thread number query routine of the threading runtime. The implicit model is only supported for OpenMP* runtimes via omp_get_thread_num().
2. Explicit model: you pass thread_id as an integer value converted to a string to MPI by setting an MPI Info object (referred to as info key in this document) to a communicator. The key thread_id must be used. This model fits task-based parallelism, where a task can be scheduled on any process thread.

The `I_MPI_THREAD_ID_KEY` variable sets the MPI info object key that is used to explicitly define the `thread_id` for a thread (`thread_id` by default).

Within the model, only threads with the same `thread_id` can communicate. To illustrate it, the following communication pattern complies to the `MPI_THREAD_SPLIT` model: Suppose Comm A and Comm B are two distinct communicators, aggregating the same ranks. The system of these two communicators will fit the `MPI_THREAD_SPLIT` model only if all threads with `thread_id #0` use Comm A, while all threads with `thread_id #1` use Comm B.

Threading Runtimes Support

The MPI thread-split programming model has special support of the OpenMP* runtime, but you may use any other threading runtime. The support differs in the way you communicate with the MPI runtime to set up `thread_id` for a thread and the way you set up the number of threads to be run concurrently. If you choose the OpenMP runtime support, make sure you have the OpenMP runtime library, which comes with the recent Intel® compilers and GNU* gcc compilers, and link the application against it.

The support is controlled with the `I_MPI_THREAD_RUNTIME` environment variable. Since the threading runtime support is a non-standard functionality, you must enable it explicitly using the `generic` or `openmp` argument for the OpenMP* runtime support.

You can set the maximum number of threads to be used in each process concurrently with the `I_MPI_THREAD_MAX` environment variable. This helps the MPI implementation allocate the hardware resources efficiently. By default, the maximum number of threads per rank is 1.

OpenMP* Threading Runtime

The OpenMP runtime supports the both implicit and explicit submodels. By default, the Intel MPI Library assumes that `thread_id` is set with the `omp_get_thread_num()` function call defined in the OpenMP standard. This scenario corresponds to the implicit submodel. You can use the explicit submodel by setting the `thread_id` info key for a communicator, which is particularly useful for OpenMP tasks.

By default, the maximum number of threads is set with the `omp_get_max_threads()` function. To override this function, set either `I_MPI_THREAD_MAX` or `OMP_NUM_THREADS` environment variable.

See Also

thread_split_omp_for.c

thread_split_omp_task.c

thread_split_pthreads.c

Program Examples

pthreads - Explicit Submodel

```
#include <mpi.h>
#include <pthread.h>
#define n 2
int thread_id[n];
MPI_Comm split_comm[n];
pthread_t thread[n];
void *worker(void *arg) {
    int i = *((int*)arg), j = i;
    MPI_Comm comm = split_comm[i];
    MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, comm);
```

```
    printf("Thread %d: allreduce returned %d\n", i, j);
}
int main() {
    MPI_Info info;
    int i, provided;
    char s[16];
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    MPI_Info_create(&info);
    for (i = 0; i < n; i++) {
        MPI_Comm_dup(MPI_COMM_WORLD, &split_comm[i]);
        sprintf(s, "%d", i);
        MPI_Info_set(info, "thread_id", s);
        MPI_Comm_set_info(split_comm[i], info);
        thread_id[i] = i;
        pthread_create(&thread[i], NULL, worker, (void*) &thread_id[i]);
    }
    for (i = 0; i < n; i++) {
        pthread_join(thread[i], NULL);
    }
    MPI_Info_free(&info);
    MPI_Finalize();
}
```

OpenMP Runtime - Implicit Submodel

```
#include <mpi.h>
#include <omp.h>
#define n 2
MPI_Comm split_comm[n];
int main() {
    int i, provided;
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    for (i = 0; i < n; i++)
        MPI_Comm_dup(MPI_COMM_WORLD, &split_comm[i]);

#pragma omp parallel for num_threads(n)
    for (i = 0; i < n; i++) {
        int j = i;
        MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, split_comm[i]);
        printf("Thread %d: allreduce returned %d\n", i, j);
    }
    MPI_Finalize();
}
```

OpenMP Tasks - Explicit Submodel

```
#include <mpi.h>
#include <omp.h>
#define n 2
MPI_Comm split_comm[n];
int main() {
    MPI_Info info;
    int i, provided;
    char s[16];
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    MPI_Info_create(&info);
    for (i = 0; i < n; i++) {
        MPI_Comm_dup(MPI_COMM_WORLD, &split_comm[i]);
```

```
        sprintf(s, "%d", i);
        MPI_Info_set(info, "thread_id", s);
        MPI_Comm_set_info(split_comm[i], info);
    }
#pragma omp parallel num_threads(n)
    {
#pragma omp task
        {
            int j = 1;
            MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, split_comm[1]);
            printf("OMP thread %d, logical thread %d: allreduce returned %d\n",
                    omp_get_thread_num(), 1, j);
        }
#pragma omp task
        {
            int j = 0;
            MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, split_comm[0]);
            printf("OMP thread %d, logical thread %d: allreduce returned %d\n",
                    omp_get_thread_num(), 0, j);
        }
    }
    MPI_Info_free(&info);
    MPI_Finalize();
}
```

Code Change Guide

The example in this section shows you one of the ways to change a legacy program to effectively use the advantages of the `MPI_THREAD_SPLIT` threading model.

In the original code (thread_split.cpp), the functions `work_portion_1()`, `work_portion_2()`, and `work_portion_3()` represent a CPU load that modifies the content of the memory pointed to by the `in` and `out` pointers. In this particular example, these functions perform correctness checking of the `MPI_Allreduce()` function.

Changes Required to Use the OpenMP* Threading Model

1. To run MPI functions in a multithreaded environment, `MPI_Init_thread()` with the argument equal to `MPI_THREAD_MULTIPLE` must be called instead of `MPI_Init()`.
2. According to the `MPI_THREAD_SPLIT` model, in each thread you must execute MPI operations over the communicator specific to this thread only. So, in this example, the `MPI_COMM_WORLD` communicator must be duplicated several times so that each thread has its own copy of `MPI_COMM_WORLD`.

   NOTE: The limitation is that communicators must be used in such a way that the thread with `thread_id` n on one node communicates only with the thread with `thread_id` m on the other. Communications between different threads (`thread_id` n on one node, `thread_id` m on the other) are not supported.
3. The data to transfer must be split so that each thread handles its own portion of the input and output data.
4. The barrier becomes a two-stage one: the barriers on the MPI level and the OpenMP level must be combined.
5. Check that the runtime sets up a reasonable affinity for OpenMP threads. Typically, the OpenMP runtime does this out of the box, but sometimes, setting up the `OMP_PLACES=cores` environment variable might be necessary for optimal multi-threaded MPI performance.

Changes Required to Use the POSIX Threading Model

1. To run MPI functions in a multithreaded environment, `MPI_Init_thread()` with the argument equal to `MPI_THREAD_MULTIPLE` must be called instead of `MPI_Init()`.
2. You must execute MPI collective operation over a specific communicator in each thread. So the duplication of `MPI_COMM_WORLD` should be made, creating a specific communicator for each thread.
3. The info key `thread_id` must be properly set for each of the duplicated communicators.

   NOTE: The limitation is that communicators must be used in such a way that the thread with `thread_id`*n* on one node communicates only with the thread with `thread_id`*m* on the other. Communications between different threads (`thread_id`*n* on one node, `thread_id`*m* on the other) are not supported.
4. The data to transfer must be split so that each thread handles its own portion of the input and output data.
5. The barrier becomes a two-stage one: the barriers on the MPI level and the POSIX level must be combined.
6. The affinity of POSIX threads can be set up explicitly to reach optimal multithreaded MPI performance.

See Also

thread_split.cpp

8

Examples

This section contains examples that are also available in the `doc/examples` subdirectory of the package:

- async_progress_sample.c
- thread_split.cpp
- thread_split_omp_for.c
- thread_split_omp_task.c
- thread_split_pthreads.c

async_progress_sample.c

```
#define PROGRESS_THREAD_COUNT 4
MPI_Comm comms[PROGRESS_THREAD_COUNT];
MPI_Request requests[PROGRESS_THREAD_COUNT];
MPI_Info info;
int idx;
/* create “per-thread” communicators and assign thread id for each communicator */
for (idx = 0; idx < PROGRESS_THREAD_COUNT; idx++)
{
        MPI_Comm_dup(MPI_COMM_WORLD, &comms[idx]);
        char thread_id_str[256] = { 0 };
        sprintf(thread_id_str, "%d", idx);
        MPI_Info_create(&info);
        MPI_Info_set(info, "thread_id", thread_id_str);
        MPI_Comm_set_info(comms[idx], info);
        MPI_Info_free(&info);
    }
/* distribute MPI operations between communicators – i.e. between progress threads */
for (idx = 0; idx < PROGRESS_THREAD_COUNT; idx++)
{
    MPI_Iallreduce(…, comms[idx], &requests[idx]);
}
MPI_Waitall(PROGRESS_THREAD_COUNT, requests, …)
```

See Also

Asynchronous Progress Control

thread_split.cpp

```
*/
#include <mpi.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <vector>
#include <string>
#include <utility>
#include <assert.h>
#include <sys/time.h>
// Choose threading model:
```

```
enum { THR_OPENMP = 1, THR_POSIX = 2, THR_NONE = 0 } threading = THR_POSIX;
template <typename T> MPI_Datatype get_mpi_type();
template <> MPI_Datatype get_mpi_type<char>() { return MPI_CHAR; }
template <> MPI_Datatype get_mpi_type<int>() { return MPI_INT; }
template <> MPI_Datatype get_mpi_type<float>() { return MPI_FLOAT; }
template <> MPI_Datatype get_mpi_type<double>() { return MPI_DOUBLE; }
int main_threaded(int argc, char **argv);
template <typename T>
bool work_portion_2(T *in, size_t count, size_t niter, int rank, int nranks)
{
    memset(in, 0, sizeof(T) * count);
}
template <typename T>
bool work_portion_1(T *in, size_t count, size_t niter, int rank, int nranks);
template <> bool work_portion_1<char>(char *in, size_t count, size_t niter, int rank, int
nranks) { return true; }
template <> bool work_portion_1<int>(int *in, size_t count, size_t niter, int rank, int nranks)
{
    for (size_t i = 0; i < count; i++) {
        in[i] = (int)(niter * (rank+1) * i);
    }
    return true;
}
template <typename T>
bool work_portion_3(T *in, size_t count, size_t niter, int rank, int nranks);
template <> bool work_portion_3<char>(char *out, size_t count, size_t niter, int rank, int
nranks) { return true; }
template <> bool work_portion_3<int>(int *out, size_t count, size_t niter, int rank, int nranks)
{
    bool result = true;
    for (size_t i = 0; i < count; i++) {
        result = (result && (out[i] == (int)(niter * nranks*(nranks+1)*i/2)));
    }
    return result;
}
int main_threaded_openmp(int argc, char **argv);
int main_threaded_posix(int argc, char **argv);
int main(int argc, char **argv)
{
    if (argc > 1) {
        if (!strcasecmp(argv[1], "openmp")) threading = THR_OPENMP;
        if (!strcasecmp(argv[1], "posix")) threading = THR_POSIX;
        if (!strcasecmp(argv[1], "none")) threading = THR_NONE;
    }
    if (threading == THR_OPENMP) {
        main_threaded_openmp(argc, argv);
        return 0;
    } else if (threading == THR_POSIX) {
        main_threaded_posix(argc, argv);
        return 0;
    }
    printf("No threading\n");
    int rank, nranks;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
    typedef int type;
    size_t count = 1024*1024;
```

```
    int niter = 100;
    type *in = (type *)malloc(count * sizeof(type));
    type *out = (type *)malloc(count * sizeof(type));
    for (int j = 1; j < niter+1; j++) {
        work_portion_1<type>(in, count, j, rank, nranks);
        work_portion_2<type>(out, count, j, rank, nranks);
        MPI_Allreduce(in, out, count, get_mpi_type<type>(), MPI_SUM, MPI_COMM_WORLD);
        assert(work_portion_3<type>(out, count, j, rank, nranks));
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
#include <omp.h>
void omp_aware_barrier(MPI_Comm &comm, int thread)
{
    assert(thread != 0 || comm != MPI_COMM_NULL);
#pragma omp barrier
    if (thread == 0)
        MPI_Barrier(comm);
#pragma omp barrier
}
struct offset_and_count { size_t offset; size_t count; };
int main_threaded_openmp(int argc, char **argv)
{
    printf("OpenMP\n");
    int rank, nranks, provided = 0;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    assert(provided == MPI_THREAD_MULTIPLE);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
    typedef int type;
    size_t count = 1024 * 1024;
    int niter = 100;
    type *in = (type *)malloc(count * sizeof(type));
    type *out = (type *)malloc(count * sizeof(type));
    // Divide workload for multiple threads.
    // Save (offset, count) pair for each piece
    size_t nthreads = 8;
    if (argc > 2) {
        nthreads = atoi(argv[2]);
    }
    size_t nparts = (count > nthreads) ? nthreads : count;
    // Use nparts, it might be less than nthreads
    size_t base = count / nparts;
    size_t rest = count % nparts;
    size_t base_off = 0;
    std::vector<offset_and_count> offs_and_counts(nparts);
    for (size_t i = 0; i < nparts; i++) {
        offs_and_counts[i].offset = base_off; // off
        base_off += (offs_and_counts[i].count = base + (i<rest?1:0)); // size
    }
    // Duplicate a communicator for each thread
    std::vector<MPI_Comm> comms(nparts, MPI_COMM_NULL);
    for (size_t i = 0; i < nparts; i++) {
        MPI_Comm &new_comm = comms[i];
        MPI_Comm_dup(MPI_COMM_WORLD, &new_comm);
    }
```

```
    // Go into parallel region, use precalculated (offset, count) pairs to separate workload
    // use separated communicators from comms[]
    // use omp_aware_barrier instead of normal MPI_COMM_WORLD barrier
#pragma omp parallel num_threads(nparts)
    {
        int thread = omp_get_thread_num();
        offset_and_count &offs = offs_and_counts[thread];
        MPI_Comm &comm = comms[thread];
        for (int j = 1; j < niter+1; j++) {
            if (!offs.count) { omp_aware_barrier(comm, thread); continue; }
            work_portion_1<type>(in + offs.offset, offs.count, j, rank, nranks);
            work_portion_2<type>(out + offs.offset, offs.count, j, rank, nranks);
            MPI_Allreduce(in + offs.offset, out + offs.offset, offs.count, get_mpi_type<type>(),
MPI_SUM, comm);
            assert(work_portion_3<type>(out + offs.offset, offs.count, j, rank, nranks));
            omp_aware_barrier(comm, thread);
        }
    }
    MPI_Finalize();
    return 0;
}
#include <pthread.h>
#include <sys/time.h>
#include <sched.h>
void pthreads_aware_barrier(MPI_Comm &comm, pthread_barrier_t &barrier, int thread)
{
    assert(thread != 0 || comm != MPI_COMM_NULL);
    pthread_barrier_wait(&barrier);
    if (thread == 0)
        MPI_Barrier(comm);
    pthread_barrier_wait(&barrier);
}
struct global_data {
    typedef int type;
    type *in, *out;
    int niter;
    size_t count;
    int rank, nranks;
    pthread_barrier_t barrier;
};
struct thread_local_data {
    size_t offset;
    size_t count;
    int thread_id;
    MPI_Comm *comm;
    global_data *global;
};
void *worker(void *arg_ptr)
{
    thread_local_data &thr_local = *((thread_local_data *)arg_ptr);
    global_data &global = *(thr_local.global);
    global_data::type *in = global.in;
    global_data::type *out = global.out;
    int &niter = global.niter;
    int &rank = global.rank;
    int &nranks = global.nranks;
    pthread_barrier_t &barrier = global.barrier;
    size_t &offset = thr_local.offset;
```

```
    size_t &count = thr_local.count;
    int &thread = thr_local.thread_id;
    MPI_Comm &comm = *(thr_local.comm);
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(thread, &mask);
    int res = sched_setaffinity(0, sizeof(mask), &mask);
    if (res == -1)
        printf("failed set_thread_affinity()\n");
    for (int j = 1; j < global.niter+1; j++) {
        if (!thr_local.count) { pthreads_aware_barrier(comm, barrier, thread); continue; }
        work_portion_1<global_data::type>(in + offset, count, j, rank, nranks);
        work_portion_2<global_data::type>(out + offset, count, j, rank, nranks);
        MPI_Allreduce(in + offset, out + offset, count, get_mpi_type<global_data::type>(),
MPI_SUM, comm);
        assert(work_portion_3<global_data::type>(out + offset, count, j, rank, nranks));
        pthreads_aware_barrier(comm, barrier, thread);
    }
}
int main_threaded_posix(int argc, char **argv)
{
    printf("POSIX\n");
    int provided = 0;
    global_data global;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    assert(provided == MPI_THREAD_MULTIPLE);
    MPI_Comm_rank(MPI_COMM_WORLD, &global.rank);
    MPI_Comm_size(MPI_COMM_WORLD, &global.nranks);
    global.count = 1024 * 1024;
    global.niter = 100;
    global.in = (global_data::type *)malloc(global.count * sizeof(global_data::type));
    global.out = (global_data::type *)malloc(global.count * sizeof(global_data::type));
    // Divide workload for multiple threads.
    // Save (offset, count) pair for each piece
    size_t nthreads = 8;
    if (argc > 2) {
        nthreads = atoi(argv[2]);
    }
    size_t nparts = ((global.count > nthreads) ? nthreads : global.count);
    pthread_barrier_init(&global.barrier, NULL, nparts);
    // Use nparts, it might be less than nthreads
    size_t base = global.count / nparts;
    size_t rest = global.count % nparts;
    size_t base_off = 0;
    std::vector<thread_local_data> thr_local(nparts);
    for (size_t i = 0; i < nparts; i++) {
        thr_local[i].offset = base_off; // off
        base_off += (thr_local[i].count = base + (i<rest?1:0)); // size
        thr_local[i].thread_id = i;
    }
    // Duplicate a communicator for each thread
    std::vector<MPI_Comm> comms(nparts);
    MPI_Info info;
    MPI_Info_create(&info);
    char s[16];
    for (size_t i = 0; i < nparts; i++) {
        MPI_Comm &new_comm = comms[i];
        MPI_Comm_dup(MPI_COMM_WORLD, &new_comm);
```

```
        snprintf(s, sizeof s, "%d", i);
        MPI_Info_set(info, "thread_id", s);
        MPI_Comm_set_info(new_comm, info);
        thr_local[i].comm = &new_comm;
        thr_local[i].global = &global;
    }
    // Start parallel POSIX threads
    std::vector<pthread_t> pids(nparts);
    for (size_t i = 0; i < nparts; i++) {
        pthread_create(&pids[i], NULL, worker, (void *)&thr_local[i]);
    }
    // Wait for all POSIX threads to complete
    for (size_t i = 0; i < nparts; i++) {
        pthread_join(pids[i], NULL);
    }
    MPI_Info_free(&info);
    MPI_Finalize();
    return 0;
}
```

See Also

Code Change Guide

thread_split_omp_for.c

```
#include <mpi.h>
#include <omp.h>
#define n 2
MPI_Comm split_comm[n];
int main()
{
    int i, provided;
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    for (i = 0; i < n; i++)
        MPI_Comm_dup(MPI_COMM_WORLD, &split_comm[i]);
#pragma omp parallel for num_threads(n)
    for (i = 0; i < n; i++) {
        int j = i;
        MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, split_comm[i]);
        printf("Thread %d: allreduce returned %d\n", i, j);
    }
    MPI_Finalize();
}
```

See Also

thread_split_omp_task.c

thread_split_pthreads.c

Threading Runtimes Support

thread_split_omp_task.c

```
#include <mpi.h>
#include <omp.h>
#define n 2
MPI_Comm split_comm[n];
int main()
{
    MPI_Info info;
    int i, provided;
    char s[16];
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    MPI_Info_create(&info);
    for (i = 0; i < n; i++) {
        MPI_Comm_dup(MPI_COMM_WORLD, &split_comm[i]);
        sprintf(s, "%d", i);
        MPI_Info_set(info, "thread_id", s);
        MPI_Comm_set_info(split_comm[i], info);
    }
#pragma omp parallel num_threads(n)
    {
#pragma omp task
        {
            int j = 1;
            MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, split_comm[1]);
            printf("OMP thread %d, logical thread %d: allreduce returned %d\n",
                   omp_get_thread_num(), 1, j);
        }
#pragma omp task
        {
            int j = 0;
            MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, split_comm[0]);
            printf("OMP thread %d, logical thread %d: allreduce returned %d\n",
                   omp_get_thread_num(), 0, j);
        }
    }
    MPI_Info_free(&info);
    MPI_Finalize();
}
```

See Also

thread_split_omp_for.c

thread_split_pthreads.c

Threading Runtimes Support

thread_split_pthreads.c

```
#include <mpi.h>
#include <pthread.h>
#define n 2
int thread_id[n];
MPI_Comm split_comm[n];
pthread_t thread[n];
```

```
void *worker(void *arg)
{
    int i = *((int *) arg), j = i;
    MPI_Comm comm = split_comm[i];
    MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, comm);
    printf("Thread %d: allreduce returned %d\n", i, j);
}
int main()
{
    MPI_Info info;
    int i, provided;
    char s[16];
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    MPI_Info_create(&info);
    for (i = 0; i < n; i++) {
        MPI_Comm_dup(MPI_COMM_WORLD, &split_comm[i]);
        sprintf(s, "%d", i);
        MPI_Info_set(info, "thread_id", s);
        MPI_Comm_set_info(split_comm[i], info);
        thread_id[i] = i;
        pthread_create(&thread[i], NULL, worker, (void *) &thread_id[i]);
    }
    for (i = 0; i < n; i++) {
        pthread_join(thread[i], NULL);
    }
    MPI_Info_free(&info);
    MPI_Finalize();
}
```

See Also

thread_split_omp_for.c

thread_split_omp_task.c

Threading Runtimes Support

9 Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.