Intel MPI

Intel's MPI implementation allows to build an MPI application once and run it on various interconnects. Good performance can be achieved also over proprietary interconnects if the vendor provides a DAPL or Libfabric implementation which Intel MPI can make use of.

- Setting up for the use of Intel MPI
  - CoolMUC-2 and CoolMUC-3 Clusters
  - SuperMUC (phase 2)
  - SuperMUC-NG
- Compiling and linking
  - Underlying compiler
  - Executing Intel MPI programs
    - Execution on SuperMUC (LoadLeveler)
    - Execution on the Linux Cluster or SuperMUC-NG (SLURM)
  - Executing hybrid-parallel programs
    - Hybrid program execution on SuperMUC phase 2 (LoadLeveler)
    - Hybrid program execution on the Linux Cluster or SuperMUC-NG (SLURM)
    - Handling environment variables
- Environment variables controlling the execution
- Hints
  - Generating Core dumps for debugging
  - Non-Blocking MPI Calls
  - Problems with start-up for very high task counts
  - MPI-3 support
- Documentation
  - General Information on MPI
  - Intel MPI documentation

Setting up for the use of Intel MPI

Intel MPI is available on all HPC systems at LRZ that support parallel processing in their batch queuing setup. The environment module `mpi.intel` makes available all tools needed to compile and execute MPI programs as described in the main MPI document. Since Intel MPI may not be binary compatible to other MPI flavours, you should completely re-compile and re-link your application under the `mpi.intel` environment.

CooLMUC-2 and CooLMUC-3 Clusters

The `mpi.intel` environment module is provided as a default setting on these systems.

SuperMUC (phase 2)

You need to unload the default MPI environment before loading the Intel MPI module:

```
module unload mpi.ibm
module load mpi.intel
```

SuperMUC-NG

The `mpi.intel` environment module is provided as a default setting on these systems.

Compiling and linking

The following table lists a number of options which can be used with the compiler wrappers in addition to the usual switches for optimization etc. The compiler wrappers’ names follow the usual mpicc, mpi90, mpiCC pattern.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>-mt_mpi</td>
<td>link against thread-safe MPI</td>
<td>Thread-safeness up to MPI_THREAD_MULTIPLE is provided. Note that this option is implied if you build with the -openmp switch.</td>
</tr>
<tr>
<td>-check_mpi</td>
<td>link with Intel Trace Collector MPI checking library.</td>
<td>Prior to invocation of the compiler and/or running the program, you need to load the special tracing module for this to work. Please see the page on ITAC for details.</td>
</tr>
<tr>
<td>-static_mpi</td>
<td>use static instead of dynamic MPI libraries</td>
<td>By default, dynamic linkage is performed.</td>
</tr>
<tr>
<td>-t [-log]</td>
<td>compile with MPI tracing, using Intel Trace Collector</td>
<td>Prior to invocation of the compiler and/or running the program, you need to load the special tracing module for this to work. Please see the page on ITAC for details.</td>
</tr>
</tbody>
</table>
Underlying compiler

The compiler used by Intel MPI default module is the Intel Fortran/C/C++ suite; the version of the compiler used depends on the presently loaded fortran /intel and ccomp/intel environment module. However it is possible to use other compilers with Intel MPI as well. The following table illustrates availability of such alternative compilers.

<table>
<thead>
<tr>
<th>Modules</th>
<th>Compiler</th>
<th>Supported Versions / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpi.intel/5. 1_pgi</td>
<td>PGI compilers</td>
<td>Usually, the default fortran/pgi must be loaded before this module.</td>
</tr>
<tr>
<td>mpi.intel/2018_gcc</td>
<td>GCC</td>
<td>The system GCC as well as at least a subset of LRZ-provided gcc modules are supported. Any supported gcc module must be loaded prior to the Intel MPI one.</td>
</tr>
</tbody>
</table>

Executing Intel MPI programs

The Hydra process management infrastructure, which is aware of the batch queuing system, is always used for starting up Intel MPI programs. This also applies if the mpiexec command is used.

Execution on SuperMUC (LoadLeveler)

Before executing the binary, the mpi.intel module must again be loaded in the executing shell (otherwise, the Intel MPI shared libraries will be replaced by IBM PE libraries); The mpiexec command should then be used inside your LoadLeveler script to start up the MPI program:

```
mpiexec [-n 12] ./myprog.exe
```

Since Intel MPI is integrated with LoadLeveler, it is usually not necessary to explicitly specify task numbers on the command line. Please check out the LoadLeveler example scripts for Intel MPI for specific scheduler-related settings.

Execution on the Linux Cluster or SuperMUC-NG (SLURM)

You can use either the SLURM srun command or the mpiexec command to start up your program inside a SLURM script or interactive salloc environment. For example,

```
mpiexec -n 32 ./myprog.exe
```

will start up 32 MPI tasks, using as many cores of the system. The same is done if you issue

```
srun -n 32 ./myprog.exe
```

Sometimes MPI tasks need more memory per task than is available per core. Then, you need to reserve more resources in your job and leave cores idling:

For example,

```
srun --cpus-per-task=2 -n 32 ./myprog.exe
```

or (on the MPP cluster with 16 cores per node)

```
mpiexec --perhost=8 -n 32 ./myprog.exe
```

would require 64 cores and allow each task to use a factor of 2 more memory.

Executing hybrid-parallel programs

This section deals with programs that use both MPI and OpenMP for parallelization. In this case, the number of cores used by each MPI task is usually equal to the number of OpenMP threads to be used by that task, and the latter is set via the environment variable OMP_NUM_THREADS. For example,

```
export OMP_NUM_THREADS=4
```
executed prior to the startup of the MPI program would cause each MPI task to use 4 threads; the job setup should therefore usually provide 4 cores to each MPI task. In order to perform appropriate pinning of the OpenMP threads, please use the compiler-specific pinning mechanism; for Intel compilers, the `KMP_AFFINITY` environment variable serves this purpose; however this will usually only work well on systems with Intel processors. Please consult the Intel MPI Reference Manual (see below) for information on how to perform pinning in more general setups.

**Hybrid program execution on SuperMUC phase 2 (LoadLeveler)**

Since Intel MPI is integrated with LoadLeveler, appropriate task distribution and pinning is automatically performed if you set up your LoadLeveler script appropriately. The `mpi.intel` module also provides a reasonable default setting for the `KMP_AFFINITY` variable.

**Hybrid program execution on the Linux Cluster or SuperMUC-NG (SLURM)**

The command sequence

```
export OMP_NUM_THREADS=4
srun --cpus-per-task=4 -n 12 ./myprog.exe
```

will start 12 MPI tasks with 4 threads each. However, the placement of tasks and threads is not optimal. A better way is to say

```
export OMP_NUM_THREADS=4
mpiexec --perhost=4 -n 12 ./myprog.exe
```

Note that the perhost argument must be the number of cores in a node, divided by the number of cores per task.

**Handling environment variables**

The `mpiexec` command takes a number of options to control how environment variables are transmitted to the started MPI tasks. A typical command line might look like

```
mpiexec -genv MY_VAR_1 value1 -genv MY_VAR_2 value2 -n 12 ./myprog.exe
```

Please consult the documentation linked below for further details and options.

**Environment variables controlling the execution**

Please consult the documentation for Intel MPI for the very large set of `I_MPI_*` variables which allow to extensively configure and optimize at compile as well as run time.

**Hints**

**Generating Core dumps for debugging**

This is by default deactivated, because it can cause significant disruption. Before starting up your program, please issue the following commands to activate generation of core dumps:

```
ulimit -c unlimited
export I_MPI_DEBUG_COREDUMP=1
```

Please do not use this on large-scale programs.

**Non-Blocking MPI Calls**

`MPI_Isend` and `MPI_Irecv` are non-blocking calls. However, this does not make the memory transfer asynchronous. The Intel® MPI Library does not spawn a separate thread for communication, so this will have to happen in the main program thread. When using shared memory, the CPU will need cycles in order to transfer the data. Those cycles typically occur during `Waitall`. If you are using RDMA, then the transfer can happen asynchronously, so there is a slight improvement. For more asynchronous behavior, you will want to use threading, and have one thread perform the `Waitall` call while other threads perform calculations.

**Problems with start-up for very high task counts**

Sometimes you may get error messages at start-up that obviously are triggered by the DAPL layer (to be deduced from the error message). Or startup may hang. In this case, please try inserting the command

```
module load dapl
```

before starting up your program with `mpiexec`. Note that these problems only are expected to arise for task counts > 4000.

**MPI-3 support**
Current versions support the MPI-3 interface. In particular, the new `mpi_f08` Fortran interface can be used in conjunction with the Intel Fortran compiler.

Documentation

**General Information on MPI**

Please refer to the [MPI page at LRZ](#) for the API documentation and information about MPI in general.

**Intel MPI documentation**

After the `mpi.intel` module is loaded, the `$MPI_DOC` environment variable points at a directory containing PDF format reference manuals and other documents.

For the most up-to-date release, the documentation can also be found on Intel's web site.