

# MPI Jobs

Last review date	Reviewer
2009-09-15	Marco Bencivenni Enrico Fattibene

## Table of Contents

### [MPI Jobs](#)

- [MPI and its implementations](#)
- [Wrapper script for `mpi- start`](#)
- [Hooks for `mpi- start`](#)
- [Defining the job and executable](#)
- [Running the MPI job](#)

# MPI Jobs

How to compile and run a simple Message Passing Interface job on the grid.

## MPI and its implementations

The *Message Passing Interface (MPI)* is commonly used to handle the communications between tasks in parallel applications. There are two versions of MPI, MPI-1 and MPI-2. Two implementations of MPI-1 (LAM and MPICH) and two implementations of MPI-2 (OpenMPI and MPICH2) are supported. Individual sites may choose to support only a subset of these implementations, or none at all.

In the past, running MPI applications on the EGEE infrastructure required significant hand- tuning for each site. This was needed to compensate for sites that did or did not have a shared file system, location of the default scratch space, etc. The current configuration allows jobs to be more portable and allows the user more flexibility.

The increased portability and flexibility is achieved by working around hard- coded constraints from the RB and by off- loading much of the initialisation work to the `mpi- start` scripts. The `mpi- start` scripts are developed by the [int.eu.grid](#) project and based on the work of the [MPI working group](#) that contains members from both int.eu.grid and EGEE.

Using the `mpi- start` system requires the user to define a *wrapper script* and a set of *hooks*. The `mpi- start` system then handles most of the low- level details of running the MPI job on a particular site.

## Wrapper script for `mpi- start`

Users typically use a script that sets up paths and other internal settings to initiate the `mpi- start` processing. The following script (named "mpi- start- wrapper.sh") is generic and should not need to have significant modifications made to it.

```
#!/bin/bash

# Pull in the arguments.
MY_EXECUTABLE=`pwd`/$1
MPI_FLAVOR=$2

# Convert flavor to lowercase for passing to mpi-start.
MPI_FLAVOR_LOWER=`echo $MPI_FLAVOR | tr '[:upper:]' '[:lower:]'`

# Pull out the correct paths for the requested flavor.
eval MPI_PATH=`printenv MPI_${MPI_FLAVOR}_PATH`

# Ensure the prefix is correctly set. Don't rely on the defaults.
```

```

eval I2G_${MPI_FLAVOR}_PREFIX=${MPI_PATH}
export I2G_${MPI_FLAVOR}_PREFIX

# Touch the executable. It exist must for the shared file system
check.
# If it does not, then mpi-start may try to distribute the executable
# when it shouldn't.
touch $MY_EXECUTABLE

# Setup for mpi-start.
export I2G_MPI_APPLICATION=${MY_EXECUTABLE}
export I2G_MPI_APPLICATION_ARGS=
export I2G_MPI_TYPE=${MPI_FLAVOR_LOWER}
export I2G_MPI_PRE_RUN_HOOK=mpi-hooks.sh
export I2G_MPI_POST_RUN_HOOK=mpi-hooks.sh

# If these are set then you will get more debugging information.
export I2G_MPI_START_VERBOSE=1
#export I2G_MPI_START_DEBUG=1

# Invoke mpi-start.
$I2G_MPI_START

```

The script first sets up the environment for the chosen flavor of MPI using environment variables supplied by the system administrator. It then defines the executable, arguments, MPI flavor, and location of the hook scripts for `mpi-start`. The user may optionally ask for more logging information with the verbose and debug environment variables. Lastly, the wrapper invokes `mpi-start` itself.

## Hooks for `mpi-start`

The user may write a script that is called before and after the MPI executable is run. The *pre-hook* can be used, for example, to compile the executable itself or download data. The *post-hook* can be used to analyze results or to save the results on the grid.

The following example (named "`mpi-hooks.sh`") compiles the executable before running it; the *post-hook* only writes a message to the standard output. A real-world job would likely save the results of the job somewhere on the grid for user retrieval.

```

#!/bin/sh

#
# This function will be called before the MPI executable is started.
# You can, for example, compile the executable itself.
#
pre_run_hook () {

    # Compile the program.
    echo "Compiling ${I2G_MPI_APPLICATION}"

    # Actually compile the program.
    cmd="mpicc ${MPI_MPICC_OPTS} -o ${I2G_MPI_APPLICATION} ${I2G_MPI_APPLICATION}.c"
    echo $cmd
    $cmd
    if [ ! $? -eq 0 ]; then
        echo "Error compiling program. Exiting..."
        exit 1
    fi

    # Everything's OK.
    echo "Successfully compiled ${I2G_MPI_APPLICATION}"
}

```

```

    return 0
}

#
# This function will be called before the MPI executable is finished.
# A typical case for this is to upload the results to a storage
# element.
#
post_run_hook () {

    echo "Executing post hook."
    echo "Finished the post hook."

    return 0
}

```

The pre- and post- hooks may be defined in separate files, but the names of the functions must be named exactly "pre\_run\_hook" and "post\_run\_hook".

## Defining the job and executable

Running the MPI job itself is not significantly different from running a standard grid job. The user must define a JDL file describing the requirements for the job. An example is:

```

#
# mpi-test.jdl
#
JobType          = "Normal";
CpuNumber        = 16;
Executable       = "mpi-start-wrapper.sh";
Arguments        = "mpi-test OPENMPI";
StdOutput        = "mpi-test.out";
StdError         = "mpi-test.err";
InputSandbox     = {"mpi-start-wrapper.sh", "mpi-hooks.sh", "mpi-test.c"};
OutputSandbox    = {"mpi-test.err", "mpi-test.out"};
Requirements =
  Member("MPI-START", other.GlueHostApplicationSoftwareRunTimeEnvironment)
  && Member("OPENMPI", other.GlueHostApplicationSoftwareRunTimeEnvironment)
  # && RegExp("grid.*lal.in2p3.fr.*sdj$", other.GlueCEUniqueID)
  ;
#
# - the end
#

```

The `JobType` must be "Normal" and the attribute `CpuNumber` must be defined (16 in this example). Despite the name of the attribute, this attribute defines the number of CPUs required by the job. It is not possible to request more complicated topologies based on nodes and CPUs.

This example uses the OpenMPI implementation of the MPI-2 standard. The other supported implementations can be selected by changing `OPENMPI` (in two places) to the name of the desired implementation. The other names are "LAM", "MPICH", and "MPICH2". The `JobType` attribute must be "Normal" in all cases; it selects for an MPI job in general and not the specific implementation.

All of the files for the above example JDL file have been defined except for the actual MPI program. This is a simple Hello World example written in C. The code is:

```

/*  hello.c
 *
 *  Simple "Hello World" program in MPI.
 *
 */

#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {

    int numprocs; /* Number of processors */
    int procnum; /* Processor number */

    /* Initialize MPI */
    MPI_Init(&argc, &argv);

    /* Find this processor number */
    MPI_Comm_rank(MPI_COMM_WORLD, &procnum);

    /* Find the number of processors */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    printf ("Hello world! from processor %d out of %d\n", procnum, numprocs);

    /* Shut down MPI */
    MPI_Finalize();
    return 0;
}

```

It is highly recommended to compile the MPI locally. Many compilation options are specific to the software installed or hardware installed on a site. Sending a binary file risks sub-optimal performance at best and crashes at worst.

## Running the MPI job

Running the MPI job is no different from any other grid job. Use the commands `glite-wms-job-submit`, `glite-wms-job-status`, and `glite-wms-job-output` to submit, check the status, and recover the output of a job.

If the job ran correctly, then the standard output should contain something like the following:

```

Hello world! from processor 15 out of 16
Hello world! from processor 0 out of 16
Hello world! from processor 1 out of 16
Hello world! from processor 7 out of 16
Hello world! from processor 2 out of 16
Hello world! from processor 3 out of 16
Hello world! from processor 4 out of 16
Hello world! from processor 6 out of 16
Hello world! from processor 8 out of 16
Hello world! from processor 9 out of 16
Hello world! from processor 12 out of 16
Hello world! from processor 5 out of 16
Hello world! from processor 10 out of 16
Hello world! from processor 14 out of 16
Hello world! from processor 11 out of 16
Hello world! from processor 13 out of 16

```

If there are problems running the job and the standard output and error do not contain enough information, setting the `mpi-start` debug flag in the wrapper script may help.