Using the Message Passing Interface (MPI) on the HEC

Multi-node (distributed memory) parallel jobs are typically written using MPI, a C/C++ and Fortran based Application Programming Interface that implements **job parallelism** via passing (network) messages between co-ordinated processes.

Related pages

- Page:High End Computing (HEC) help
- Page:Requesting specific node types for jobs on the HEC
- Page:Submitting jobs on the HEC

More information

Compiling MPI codes

The HEC currently supports the OpenMPI implementation of MPI, available through one of several modules described below. The implementation offers compiler wrappers to assist with compiling MPI codes. The wrappers call an underlying compiler, and automatically take care of locating the correct version of MPI libraries and include files. The compiler wrappers are:

Wrapper name	Supported Language
mpicc	C
mpiCC	C++
mpif77	Fortran 77
mpif90	Fortran 90

These wrappers can be called once the relevant module has been added to your environment and are used in exactly the same way that a standard compiler can be called.

More complex MPI applications are typically built with a Makefile and/or a configure script. These applications can be built by specifying the compiler wrappers in place of the general compilers. It isn't necessary to specify the location or name of the MPI libraries or header files - the wrappers will handle these details automatically - so these fields can typically be left blank in a Makefile.

Each version of OpenMPI has been built in one of three flavours, one for each of the three supported compilers on the HEC: gcc, intel and pgi. For example, the version of OpenMPI 1.4.3 built for Intel compilers will have the module name: openmpi/1.4.3-intel. When submitting MPI jobs, care should be take to load the same module version and flavour as was used to originally build it.

Submitting OpenMPI Jobs

Here is a sample job to run the IMB benchmark across 2 nodes, and 16 cores per node:

```
#$ -S /bin/bash

#$ -q parallel
#$ -l node_type=10Geth*
#$ -l nodes=2

source /etc/profile

module add openmpi/1.8.1-intel

mpirun IMB-MPI1
```

Anatomy of an MPI job script

In the above job script example, many of the SGE job directives are explained under the basic batch job submission page. Below is a description of the additional entries required to launch MPI jobs:

Queue selection

```
#$ -q parallel
```

This signals to the job scheduler that this shoud be run on the parallel queue, which offers full support for parallel jobs.

Node type selection

```
#$ -l node_type=10Geth*
```

This specifies which type of compute node the parallel job will run on. Different nodes have different architectures and numbers of cores; to ensure optimum placement all nodes used in the same parallel job must be of the same type. The different types of node are described in full in Requesting specific node types for jobs on the HEC. The pattern **10Geth*** allows jobs to run on both **10Geth64G** and **10Geth128G** which are compute nodes with 10 Gbit ethernet network connectivity and either 64G or 128G memory respectively. It is recommended to use **10Geth*** for most parallel jobs, and specify **10Geth128G** only when more memory is required.

Specifying job size (number of cores used)

```
#$ -l nodes=2
```

This specifies that the job requires two whole nodes. The nodes will be used exclusively for the parallel job - no other serial or parallel job may co-exist on the selected compute nodes.

Currently, all compute nodes offer 16 cores per node, but this may change as we refresh the cluster’s compute nodes. The job will run using a number of cores equal to the number of nodes selected multiplied by that node_type's core count. The example above uses **node s=2** along with **node_type=10Geth*** which is an 16-core node type, meaning that the job will run on 2 nodes with 16 cores per node, ie a 32-core job. the node type selected been **n ode_type=10Geth64G** (a 16-core node type).

Advanced use of nodes= syntax

On rare occasions, it may be required to run fewer than the default number of processes per node. The nodes request can be supplemented by a processes-per-node value. For example, to run the above IMB benchmark using 2 nodes with 1 process per node (effectively measuring point-to-point performance between two nodes), you can use the following instead:

```
#$ -l nodes=2,ppn=1
```

As MPI jobs exclusively book whole compute nodes, specifying fewer processes per node than the number of available cores will result in the remaining cores going unused - they cannot be used by other users or other jobs. Only specify a ppn value when absolutely necessary.

Calling the MPI application:

The final line in the job script:

```
mpirun IMB-MPI1
```

is the call to the parallel application (in this case IMB-MPI1) wrapped in a call to the mpirun application which will handle the parallel startup of the user application. Note that **mpirun** d oes not need to be told the number of processes to run; OpenMPI automatically picks this value up from the job scheduler.

The name of the MPI application should typically be the last argument to mpirun. For MPI applications that require their own additional arguments, you should place them after the call to the application itself, as arguments before the application call are interpreted by the mpirun command.

Testing suggests OpenMPI supports basic input redirection on the assumption that standard input is read by rank zero of the application.

A note on memory resource requests for MPI jobs

As parallel jobs reserve whole nodes, memory reservation is automatically set to the selected compute node's full memory. Larger memory compute nodes are a sparse resource and in high demand. Parallel jobs requiring them may wait a very long time to schedule. It may be preferable to split your job over a larger number of standard-memory nodes.

Running small parallel jobs

As not all parallel jobs scale efficiently to the size of at least a single node, an alternative syntax can be used to request parallel jobs with a small core count. The **nodes=** syntax can be replaced by a request for the specific amount of cores requires using the **np=** syntax . For example:

```
#$ -S /bin/bash

#$ -q parallel
#$ -l np=4
#$ -l h_vmem=1G

source /etc/profile

module add openmpi/1.8.1-intel

mpirun --bind-to none IMB-MPI1
```

The above job runs the same application as the first example but requests only 4 cores, all on the same compute node. As this syntax doesn't create a node reservation, the remaining cores on the compute node will be available for other jobs.

Note that smaller MPI jobs must specify their memory requirements - as they don't reserve a whole node, they cannot assume all the node's memory will be available. The memory resource request uses the same syntax as for serial jobs (e.g. **#$ -l h_vmem=1G**). This memory request is **per core**, so the job in the above example is requesting 4G of memory (4 cores, 1G per core).

Note also that the **np=** syntax ensures that all job slots for the job are on the same node. As a result, the value should never be greater than the maximum number of cores on the largest compute node – this is currently 16.

Tips on parallel jobs

- Not all jobs scale well when parallelised - running on n cores will not result in your code running n times faster. Always test an application with different job sizes (including single-core) to find the 'sweet spot' which best uses the resources available.
- The system has been designed to support MPI jobs with moderate message passing on up to 64 processors. Applications with lighter message passing loads may scale higher than this.
- As parallel jobs require much more resource than regular single-core batch jobs, there is usually a much longer wait between job submission and job launch, particularly when the cluster is busy. Opt to submit jobs as serial rather than parallel unless the improvement in runtime is essential.
- It can be frustrating to wait a long time for a parallel job to launch, only for it to quickly fail due to bugs in the job script. You can test a new parallel job script by directing it to the test queue. Be aware that the test queue only offers a single, 16-core compute node of type **10Geth64G.**