Parallel Jobs

1. Introduction

- A parallel job is a job that uses more than one CPU.
- Since the cluster is a shared resource, it is the GE that allocates CPUs, known as slots in GE jargon, to jobs, hence a parallel job must request a set of slots when it is submitted.
- A parallel job must request a parallel environment and a number of slots using the `-pe <pe-name> N` specification to `qsub`, where
    - `<pe-name>` is either `mpich`, `orte` or `mthread` (see below);
    - the value of `N` is the number of requested slots and can be specified a `N-M`, where `N` is the minimum and `M` the maximum number of slots requested;
    - that option can be an embedded directive.
    - The job script accesses the number of allocated slots via an environment variable (`NSLOTS`) and,
    - for MPI jobs, gets the list of computed nodes via a so-called machines file.
- There are several types of parallel jobs:

    a. MPI or distributed jobs: the CPUs can be distributed over multiple compute nodes.

PROs	There is conceptually no limit on how many CPUs can be used, the cumulative amount of CPUs and memory a job can use can get quite large. The GE can find (a lot of) unused CPUs on a busy machine by finding them on different nodes
CONs	Each CPU is assumed to be on a separate compute node and thus each process must communicate with the other CPUs to exchange information (aka message passing). Programming can get more complicated and the inter-process communication can become a bottleneck.

    b. Multi-threaded jobs: all the CPUs *must be* on the same compute node.

PROs	All CPUs can share a common memory space, inter-process communication can be very efficient (being local) and programming can be simpler;
CONs	Can only use as many CPUs as there are on the largest compute node, and can get them only if they are not in use by someone else.

    c. Hybrid jobs: the CPUs are distributed, but with the same number of CPUs on each compute node.

PROs	The CPUs on the same node can share a common memory space, while not all CPUs ate on the same compute node, hence the total number of CPUs is not limited to the number of CPUs on the largest compute node;
CONs	Coding must mix inter-process communication (like MPI) with shared memory and multi-threading (like OpenMP). This can be tricky, but some problems can greatly benefit from this model.

- How do I know which type of parallel job to submit to?
  The author of the software will in most cases specify if the application can be parallelized and how
    - Some analyses are parallelized by submitting a slew of independent serial jobs,
        - in which case using a job array may be the best approach;
    - some analyses use explicit message passing (`MPI` or `OpenMPI`); while
    - some analyses use a programming model that can use multiple threads (or `OpenMP`); while
    - some mix both: message passing and mutli-threading.

2. MPI, or Distributed Parallel Jobs with Explicit Message Passing

- An MPI job runs code that uses an explicit message passing programming scheme known as MPI.
- There are two distinct implementations of the MPI protocol:

    - MPICH and
    - ORTE;
- OpenMPI is an ORTE implementation of MPI;
- MVAPICH is a MPICH implementation, using explicitly the InfiniBand as transport fabric (faster message passing).

⚠️ OpenMPI is not OpenMP
- OpenMPI is the ORTE implementation of MPI;
- OpenMP is an API for multi-platform shared-memory parallel programming.

The following grid of modules, corresponding to a combination of compiler & implementation, is available on Hydra:

	ORTE	MPICH	MVAPICH
GNU	gcc/openmpi	gcc/mpich	gcc/mvapich
GNU gcc 4.4.7	gcc/4.4/openmpi	gcc/4.4/mpich	gcc/4.4/mvapich
GNU gcc 4.9.1	gcc/4.9/openmpi	gcc/4.9/mpich	gcc/4.9/mvapich
GNU gcc 4.9.2	gcc/4.9/openmpi-4.9.2	n/a	n/a
Intel	intel/mpi	n/a	n/a
Intel v15.x	intel/15/mpi	n/a	n/a
Intel v16.x	intel/16/mpi	n/a	n/a
PGI	pgi/openmpi	pgi/mpich	pgi/mvapich
PGI v14.x	pgi/14/openmpi	pgi/14/mpich	pgi/14/mvapich
PGI v15.x	pgi/15/openmpi	pgi/15/mpich	pgi/15/mvapich
PGI v15.9	pgi/15/openmpi-15.9	pgi/15/mpich-15.9	pgi/15/mvapich-15.9

In fact, there are more version specific modules available, check with

```
% ( module -t avail ) | & grep pi
```

for a complete list,. You can also use

```
% module whatis <module-name>
```

or

```
% module help <module-name>
```

where <module-name> is one of the listed module, to get more specific information.

2.a ORTE or OpenMPI

The following example shows how to write an ORTE/OpenMPI job script:

Example of a ORTE/OpenMPI job script, using Bourne shell syntax

```
# /bin/sh
#
#$ -S /bin/sh
#$ -cwd -j y -N hello -o hello.log
#$ -pe orte 72
#
echo + `date` job $JOB_NAME started in $QUEUE with jobID=$JOB_ID on $HOSTNAME
echo + NSLOTS = $NSLOTS distributed over:
cat $PE_HOSTFILE
#
# load gcc's compiler and openmpi
module load gcc/4.9/openmpi-4.9.2
#
# run the program hello
mpirun -np $NSLOTS ./hello
#
echo = `date` job $JOB_NAME done
```

This example will

- show the content of the machine file (i.e. the distribution of compute nodes)
- load the OpenMPI module for gcc version 4.9.2, and
- run the program `hello`,
- requesting 72 slots (CPUs).Hybrid Jobs

It assumes that the program `hello` was built using `gcc v4.9.2`.

2.b MPICH or MVAPICH

The following example shows how to write a MVAPICH job script:

Example of a MVAPICH job script, using C-shell syntax

```
# /bin/csh
#
#$ -cwd -j y - N hello -o hello.log
#$ -pe mpich 72
#
echo + `date` job $JOB_NAME started in $QUEUE with jobID=$JOB_ID on $HOSTNAME
echo using $NSLOTS slots on:
sort $TMPDIR/machines | uniq -c
#
# load PGI's mvapich
module load pgi/mvapich
#
# run the program hello
mpirun -np $NSLOTS -machinefile $TMPDIR/machines ./hello
#
echo = `date` job $JOB_NAME done
```

This example will

- show the content of the machine file (i.e. the distribution of compute nodes),
    - using `sort` & `uniq` to produce a compact list, in a "`hostname no_of_CPUs`" format.
- load the `MVAPICH` module for the PGI compiler, and
- run the program `hello`,
- requesting 72 slots (CPUs).

It assumes that the program `hello` was build using the PGI compiler and the `MVAPICH` library/module to enable the IB as the transport fabric.

You could replace `MVAPICH` by `MVPICH` if you do not want to use the IB.

2.c Notes

- The command `mpirun` is *aliased* by the module file to use the full path of the correct version for each case.

- Do not use a full path specification to a version of `mpirun,` using a wrong version of `mpirun` will result in unpredictable results.
  You can check which version corresponds to a module with either
  ```
  % module show <module-name>
  ```
  or, if you use the C-shell,
  ```
  % alias mpirun
  ```
  of, is your use the Bourne shell
  ```
  % declare -f mpirun
  ```

- The following snippet is a simple trick to get the MPI code full path and use it on the `mpirun` line:

Bourne shell syntax	C-Shell syntax

`# load the module`
`module load tools/mpidemo`
`# get the full path`
`bin=`which demo_mpi``
`# show what will be executed`
`declare -f mpirun`
`echo mpirun -np $NSLOTS $bin`
`# run it`
`mpirun -np $NSLOTS $bin`
	`# load the module`
	`module load tools/mpidemo`
	`# get the full path`
	`set bin = `which demo_mpi``
	`# show what will be executed`
	`alias mpirun`
	`echo mpirun -np $NSLOTS $bin`
	`# run it`
	`mpirun -np $NSLOTS $bin`

  This example assumes that the module `tools/mpidemo` set things up to run the MPI code `demo_mpi`, and

  shows how to prevent having to use a full path anywhere. While `mpirun` search your `PATH` to find the executable,

  the `mpirun` started on other compute nodes may not have the same `PATH` since the associated module is likely to not have been loaded on that other node.

- The error message:

  `[proxy:0:0@compute-N-M.local] HYDU_create_process`

  `(./utils/launch/launch.c:75): execvp error on file <code> (No such file or directory)`

  means the `mpirun` could not find the executable `<code>`.

- In most cases you should use `mvapich` and *not* `mpich`, and use the IB fabric.

- One can query the technical implementation details of MPI for each module,

  since each MPI-enabling module implements a slightly different version of MPI.

  - You query the precise details of each implementation as follows:

Command	to
`% module show <module-file>`	Show how the module changes your Un*x environment. All the modules set the same env variables: `MPILIB` `MPIINC` `MPIBIN` plus either `OPENMPI`, `MPICH`, or `MVAPICH`, and set the alias `mpirun` to use the corresponding full path.
`% module help <module-file>`	Show details on the module, and how to retrieve the details of the specific build.
	Depending on the MPI implementation (`ORTE` or `MPICH`)
`% module load <module-file>` `% ompi_info [-all]`	Show precise details of an `ORTE` implementation (as shown by `module help <module-file>`)
or	

`% module load <module-file>`
 	
`% mpirun -info`
	Show precise details of an `MPICH/MVAPICH` implementation
	
	(as shown by `module help <module-file>`)

3. Multi-threaded, or OpenMP, Parallel Jobs

A multi-threaded job is a job that will make use of more than one CPU but needs all the CPUs to be on the same compute node.

3.a Multi-threaded jobs

The following example shows how to write a multi-threaded script:

Example of a Multi-threaded job script, using Bourne shell syntax

```
# /bin/sh
#
#$ -S /bin/sh
#$ -cwd -j y -N demo -o demo.log
#$ -pe mthread 32
#
echo + `date` job $JOB_NAME started in $QUEUE with jobID=$JOB_ID on $HOSTNAME
echo + NSLOTS = $NSLOTS
#
# load the demo (fictional) module
module load tools/demo
#
# convert the generic parameter file, gen-params, to a specific file
# where the number of thread are inserted where the string MTHREADS is found
sed "s/MTHREADS/$NSLOTS/" gen-params > all-params
#
# run the demo program specifying all-params as the parameter file
demo -p all-params
#
echo = `date` job $JOB_NAME done
```

This example will run the tool `demo` using 32 CPUs (slots). The script

- loads the `tools/demo` module,
- parses the file `gen-params` and replaces every occurrence of the string `MTHREAD` by the allocated number of slots (via `$NSLOTS`),
    - using the stream editor `sed` (man sed),
- saves the result to a file called `all-params,`
- runs the tool `demo` and with the parameter file `all-params`.

3.b OpenMP jobs

The following example shows how to write an OpenMP job script:

Example of a OpenMP job script, using C-shell syntax

```
# /bin/csh
#
#$ -cwd -j y -N hellomp -o hellomp.log
#$ -pe mthread 32
#
echo + `date` job $JOB_NAME started in $QUEUE with jobID=$JOB_ID on $HOSTNAME
echo + NSLOTS = $NSLOTS
#
# load the pgi module
module load pgi
#
# set the variable OMP_NUM_THREADS to the content of NSLOTS
# this tell OpenMP applications how many threads/slots/CPUs to use
setenv OMP_NUM_THREADS $NSLOTS
#
# run the hellomp OpenMP program, build w/ PGI
./hellomp
#
echo = `date` job $JOB_NAME done
```

This example will run the program `hellomp`, that was compiled with the PGI compiler, using 32 threads (CPUs/slots). The script

- loads the pgi module,
- sets `OMP_NUM_THREADS` to the content of `NSLOTS` to specify the number of threads

- runs the program `hellomp`.

4. Hybrid Jobs

- A hybrid job is a job that will use `N` slots/CPUs/threads distributed as `M` CPUs/threads on `K` compute nodes, where `N = K x M`.
- The software must be written as to make use of this configuration, usually by combining `MPI/OpenMPI` and `OpenMP/Multi-threading` coding.

How to Run a Hybrid Job

To run a job as a hybrid job you need to

1. request one of the hybrid PEs (parallel environments)
2. source a configuration file that is created at run-time (`$TMPDIR/set-hybrid-config`)
3. run a code written for a hybrid PE.

Examples

Examples of hybrid jobs are under `/home/hpc/examples/hybrid`.

⚠️ The key line(s) in the job file examples are ⚠️:

csh syntax	sh syntax

`if (-e $TMPDIR/set-hybrid-config) then`
`  source $TMPDIR/set-hybrid-config`
`endif`
	`if [ -e $TMPDIR/set-hybrid-config ]`
	`then`
	`  source $TMPDIR/set-hybrid-config`
	`fi`

- The if statement (optional) allows the job file to be run in non-hybrid mode (if applicable).
- You will see in the output/log file the following:

-pe h8 32

```
--------------- hybrid_start 1.0/1 ---------------
hybrid_start: remember to 'source $TMPDIR/set-hybrid-config' to properly setup your env
--------------------------------------------------
....
PE=h8 NSLOTS=4 OMP_NUM_THREADS=8
```

- The last line shown above will be produced only of you source the configuration file.
- Use `$NSLOTS` and `$OMP_NUM_TREADS` whenever needed.

Available PEs

You can use one of the following hybrid PEs:

Name	No of CPUs/node	Example	Means
`h2`	2	`-pe h2 64`	request 64 slots distributed as 2 CPUs/node on 32 different nodes
`h4`	4	`-pe h4 64`	request 64 slots as 4 CPUs/node on 16 different nodes
`h8`	8	`-pe h8 64`	request 64 slots as 8 CPUs/node on 8 different nodes
`h12`	12	`-pe h12 48`	request 48 slots as 12 CPUs/node on 4 different nodes
`h16`	16	`-pe h16 64`	request 64 slots as 16 CPUs/node on 4 different nodes
etc... up to `h32` by increment of 4: `h2, h4, h8, h12, h16, h20, h24, h28` and `h32`			

Note

⚠️ You specify `M` and `N`, where `N = K x M`:

- `-pe h8 34` stands for `M=8 N=32 K=4`
- It will run on `K` different nodes, each node can use `M` threads/CPUs,
- The job will be rejected if `N` is not a multiple of `M`,
- The hybrid PEs are available in all the hi-CPUs queues (`?ThC.q`).

More Details/Explanations

Simple job, using C-shell syntax, to run an `OpenMPI/OpenMP` hybrid code, called `hybrid`.

hybrid.job

```
# /bin/csh
#
#$ -q mThC.q -pe h8 64
#$ -N hybrid -o hybrid.log -cwd -j y
#
echo $JOB_NAME started `date` on $HOSTNAME in $QUEUE jobID=$JOB_ID
#
if (-e $TMPDIR/set-hybrid-config) then
  source $TMPDIR/set-hybrid-config
endif
#
module load gcc/4.4/openmpi
mpirun -np $NSLOTS -hostfile $HOSTFILE ./hybrid
#
echo `date` $JOB_NAME done.
```

This job will produce as output:

hybrid.log

```
--------------- hybrid_start 1.0/1 ---------------
hybrid_start: remember to 'source $TMPDIR/set-hybrid-config' to properly setup your env
------------------------------------------------
Warning: no access to tty (Bad file descriptor).
Thus no job control in this shell.
hybrid started Thu Aug 18 13:32:05 EDT 2016 on compute-1-4.local in mThC.q jobID=6374206
PE=h8 NSLOTS=8 OMP_NUM_THREADS=8
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  1, iSize=  8, hostname=compute-1-4.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  5, iSize=  8, hostname=compute-3-11.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  2, iSize=  8, hostname=compute-1-5.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  3, iSize=  8, hostname=compute-1-6.local
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
```

```
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
hello from iRank=  4, iSize=  8, hostname=compute-2-15.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  6, iSize=  8, hostname=compute-3-3.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  7, iSize=  8, hostname=compute-3-4.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
hello from iRank=  0, iSize=  8, hostname=compute-1-3.local
Thu Aug 18 13:32:10 EDT 2016 hybrid done.
```

The program `hybrid` corresponds to the following trivial F90 code:

hybrid.f90

```
program hello
  !
  include 'mpif.h'
  !
  integer iErr, iRank, iSize
  integer mpiComm, msgTag
  !
  character*40 hostname
  call HOSTNM(hostname)
  !
  mpiComm = MPI_COMM_WORLD
  msgTag  = 0
  !
  call MPI_INIT(iErr)
  call MPI_COMM_RANK(mpiComm, iRank, iErr)
  call MPI_COMM_SIZE(mpiComm, iSize, iErr)
  !
  !$omp parallel
  print 9000, 'hello from iRank=',iRank, &
       ', iSize=', iSize, &
       ', hostname=', trim(hostname)
  !$omp end parallel
  !
  call MPI_FINALIZE(iErr)
9000 format(a,i3,a,i3,a,a)
!
end program hello
```

The configuration file (`$TMPDIR/set-hybrid-config`) does the following:

- resets the values of `NSLOTS,` and sets `OMP_NUM_THREADS,` and show their values, and
- rewrites the machine file (`$MACHINEFILE`, for `MPI`), and the host file (`$HOSTFILE, for OpenMPI`),
- shows the resulting values of `PE, NSLOTS` and `OMP_NUM_THREADS.`

You can find more examples in `~hpc/examples/hybrid`, where this example is build and run for differents compilers (`gnu, Intel, PGI`) , using either `MPI` or `OpenMPI` and using the `sh` or `csh` syntax.

Last Updated 08 May 2019 SGK.