Getting started in HPC

Contents:

1. Before applying for resources

- Note that CHPC systems use Linux - there is no support for Windows or OSX applications.
- Jobs are managed by a scheduler (PBS Pro). Queue times are a function of the resources requested (number of compute nodes and estimated compute time) as well as how busy the cluster is; this could vary anywhere between 0 seconds and several weeks.
- Our processor chips are generally not much faster than that on a typical laptop, so your project will only really benefit if your code can take advantage of multi-cpu implementations (i.e. parallel programming with e.g. MPI or OpenMP).
- (Try to) assess whether your proposed project/code:
  1) is therefore actually suitable for deployment in an HPC environment** (if not immediately, then try to think of ways in which it may be adapted as such),
  2) and / requires a license of any kind.

  If you're unsure about these, visit https://users.chpc.ac.za/helpdesk/tickets/submit/ or send us an email at helpdesk@chpc.ac.za, telling us in more detail about the code(s) you are wanting to run, noting aspects such as CPU, storage and memory requirements.

*Contact info:
Dr. Sean February, Senior Research Scientist (Astrophysics/Cosmology)
email: sfebruary@csir.co.za
ph.: +27 21 658 2740

**Yes to either of the following is a good indication:
- It is memory/data intensive (i.e. more than ~90% of what your workstation can currently handle)?
- Will it run quicker the more cpus you throw at it, and by how much (e.g. thrice as fast with triple the cores, or only half as fast)?

2. Hands-on exercises

After getting an account at CHPC, it could be useful to try a few exercises to get yourself going. [Note that in the mean time, these may also be run on your linux desktop, assuming you have openMPI[1], Anaconda python[2] and mpi4py[3] installed.]

2.1 Basic resource checks

Let's start by creating working directories for this tutorial and navigating to one of them:

```
[user@login ~]$ pwd
[user@login ~]$ /home/user
[user@login ~]$ mkdir CHPC_getting_started
[user@login ~]$ cd CHPC_getting_started
[user@login CHPC_getting_started]$ mkdir resource_check
[user@login CHPC_getting_started]$ cd resource_check
[user@login resource_check]$ pwd
[user@login resource_check]$ /home/user/CHPC_getting_started/resource_check
```

Next, we'll create a PBS bash script, called `pbs.resource_check`, that contains a few key commands to get us started. [While learning vim is highly recommended, nano works well too for basic editing.] Enter the following inside `pbs.resource_check`:

```
#!/bin/sh
#PBS -N ResourceCheck
#PBS -q normal
#PBS -P PROJECT_SHORTNAME
#PBS -l select=2:ncpus=24:mpiprocs=4:nodetype=haswell_reg
#PBS -l place=free
#PBS -l walltime=00:01:00
#PBS -o /home/user/CHPC_getting_started/resource_check/stdout
#PBS -e /home/user/CHPC_getting_started/resource_check/stderr
#PBS -m abe -M your_email@address

cd $PBS_O_WORKDIR
LOGFILE=log.resource_check

echo "My job starts here" > $LOGFILE
date >> $LOGFILE

echo "   " >> $LOGFILE
echo `cat $PBS_NODEFILE` >> $LOGFILE
echo "   " >> $LOGFILE

echo "My job ends here" >> $LOGFILE
date >> $LOGFILE
```

Once you're happy, submit the latter to the scheduler as follows:

```
[user@login resource_check]$ qsub pbs.resource_check
```

and check that the output written to `log.resource_check` makes sense. Change the numbers of nodes (e.g. `select=3`) or `mpiprocs` (always keep `ncpus=24`) and note any changes. The status of jobs can be checked with the `qstat` command. For more info, `man qstat`.

---

[1] This requires some level of experience with installing of software. To be followed up via private communication.

[2] Download the relevant package here: https://www.continuum.io/downloads taking note whether your system is 32/64-bit.

[3] Once your openMPI is setup, running "pip install mpi4py" should do the trick.

2.2 Estimating π with Monte Carlo simulations (in Python)

In this exercise we'll run some Python code that estimates π via a Monte Carlo simulation (see Fig. 1 below). For a unit circle, the approximation amounts to:

$$\pi = 4 * \frac{A_{circle}}{A_{square}}$$

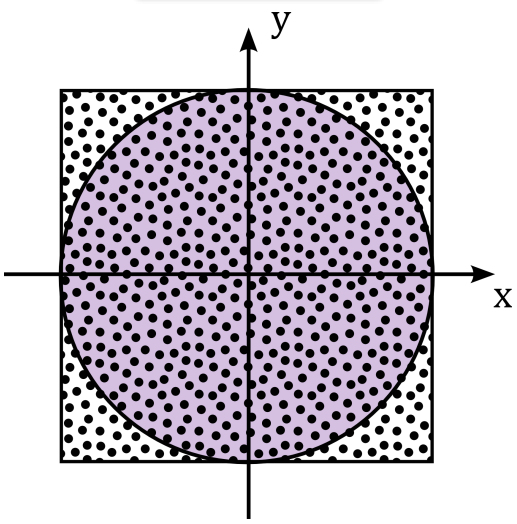


Figure 1. The basic geometrical setup of the Monte Carlo simulation to estimate π. The circle here has a radius of 1.

We'll first run a couple of scripts in serial and then move onto a parallel implementation. For these exercises we will `module add chpc/python chpc/openmpi/2.0.0/gcc-6.1.0_java-1.8.0_73` to source the appropriate python and Open MPI.

2.2.1 Serial case: loop vs. vectorisation

```
[user@login resource_check] $ cd ..
[user@login CHPC_getting_started] $ mkdir -p MonteCarloPi/serial
[user@login CHPC_getting_started] $ cd MonteCarloPi/serial
[user@login serial] $ pwd
[user@login serial] $ /home/user/CHPC_getting_started/MonteCarloPi/serial
```

Create the following two files: `main_std.py` (a `for` loop implementation)

```
import numpy as np
import time
# ^^^^^^^^^^^^^^^^^^ useful functions ^^^^^^^^^^^^^^^^^^ #
# Main routine
def pi_simulation_std(x,y):
  pointsIncircle = 0
  for i in range(len(x)):
    if is_in_circle_std(x[i],y[i]):
      pointsIncircle += 1
  return np.float(pointsIncircle) / np.float(len(x)) * 4.0

# Check whether a given (x,y) point is inside the
# unit circle (returns True) or not (returns False)
def is_in_circle_std(x,y):
  if x**2.0 + y**2.0 <= 1.0:
    return True
  else:
```

```
    return False
# Generate two arrays of random numbers, i.e. our (x,y) points.
def generate_random_points(NumberOfTrials,seed):
    np.random.seed(seed)
    x = np.random.rand(NumberOfTrials, 1)*2.0 - 1.0
    y = np.random.rand(NumberOfTrials, 1)*2.0 - 1.0
    return x,y
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ #

# Set number of trials
N = 5000000
# Seed for random number generator (for reproducibility)
seed = 999
# Start timing the program
st = time.time()
print("No. of trials, seed: ", N, seed)
x,y = generate_random_points(N,seed)
# Call main program, assign the result to a variable
pi_est = pi_simulation_std(x,y)
# Print out the estimate of pi rounded to 8 decimal places
print("Pi = ", np.round(pi_est,8))
print("Elapsed time (seconds):", time.time() - st)
```

and `main_vec.py` (using `numpy` routines)

```
import numpy as np
import time
# ^^^^^^^^^^^^^^^^^^^^ useful functions ^^^^^^^^^^^^^^^^^^^^ #
# Main routine
def pi_simulation_vec(x,y):
  pointsIncircle = is_in_circle_vec(x,y)
  return np.float(pointsIncircle) / np.float(len(x)) * 4.0

# Check whether a given (x,y) point is inside the
# unit circle (returns True) or not (returns False)
def is_in_circle_vec(x,y):
  return len(np.where((x**2.0 + y**2.0 <= 1.0)==True)[0])

# Generate random pairs of (x,y) points
def generate_random_points(NumberOfTrials,seed):
    np.random.seed(seed)
    x = np.random.rand(NumberOfTrials, 1)*2.0 - 1.0
    y = np.random.rand(NumberOfTrials, 1)*2.0 - 1.0
    return x,y
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ #

# Set number of trials
N = 5000000
# Seed for random number generator (for reproducibility)
seed = 999
# Start timing the program
st = time.time()
print("No. of trials, seed: ", N, seed)
x,y = generate_random_points(N,seed)
# Call main program, assign the result to a variable
pi_est = pi_simulation_vec(x,y)
# Print out the estimate of pi rounded to 8 decimal places
print("Pi = ", np.round(pi_est,8))
print("Elapsed time (seconds):", time.time() - st)
```

Keeping things simple for now, let's assume we've accessed an interactive session on the cluster, or equivalently are using a laptop/desktop. Running the above scripts should produce (more or less) the following:

```
[user@login serial] $ python main_std.py
No. of trials, seed:  5000000 999
Pi =  3.1411504
Elapsed time (seconds): 15.0865240
```

and

```
[user@login serial] $ python main_vec.py
No. of trials, seed:  5000000 999
Pi =  3.1411504
Elapsed time (seconds): 0.31577706
```

Note the much faster numpy-vectorised version, which also contains fewer lines of code than the looped version.

2.2.1 Parallel case: loop only

```
[user@login serial] $ cd ..
[user@login MonteCarloPi] $ mkdir parallel && cd parallel
[user@login parallel] $ pwd
[user@login parallel] $ /home/user/CHPC_getting_started/MonteCarloPi/
parallel
```

Create the following file (`main_std_mpi.py`):

```
# MPI-parallelised version of our Monte Carlo integrator
# to estimate pi. Run by doing:
# mpirun -np # python main_std_mpi.py
# at the command line, where # is the desired number of processors.
import numpy as np
from mpi4py import MPI

# Initialise the MPI communicator, and number of processors
comm = MPI.COMM_WORLD
# Extract the rank IDs (rank=0: processor 1, rank=1: processor 2 etc.)
rank = comm.rank
# Get the number of processors used to run the script
num_procs = comm.size
# Initialise an array for the pi estimates we will store from each rank,
# i.e. rank 0 will have pi_est=[a,0,0,0..], rank 1 pi_est=[0,b,0,0...] etc.
# as we as an array for the combined estimates, that will look like [a,b,c,..].
pi_est = np.zeros(num_procs)
pi_est_combined = np.zeros(num_procs)

# ^^^^^^^^^^^^^^^^^^ useful functions ^^^^^^^^^^^^^^^^^^ #
# Main routine
def pi_simulation_std(x,y):
  pointsIncircle = 0
  for i in range(len(x)):
    if is_in_circle_std(x[i],y[i]):
      pointsIncircle += 1
  return np.float(pointsIncircle) / np.float(len(x)) * 4.0

# Check whether a given (x,y) point is inside the
# unit circle (returns True) or not (returns False)
def is_in_circle_std(x,y):
  if x**2.0 + y**2.0 <= 1.0:
    return True
  else:
    return False
```

```
# Generate two arrays of random numbers, i.e. our (x,y) points.
def generate_random_points(NumberOfTrials,seed):
  np.random.seed(seed)
  x = np.random.rand(NumberOfTrials, 1)*2.0 - 1.0
  y = np.random.rand(NumberOfTrials, 1)*2.0 - 1.0
  return x,y

# Routine to send consecutive sets of points generated by rank 0 to all
# other ranks, i.e. if rank 0 generates x=[a,b,c,d,e,f,g], and we use 3
# ranks, then rank 1 receives x_split=[d,e], rank 2 receives x_split=[f,g] where
# in rank 0 we have x_split=[a,b,c]
def pi_split_sendrecv(N,seed,chunk,rem,num_procs,randx,randy):
  if rank==0:
    if num_procs > 1:
      for i in range(num_procs-1):
        # start at rank 1
        j = i+1
        # Index at which the data array to be sent, begins
        start = (chunk+rem)*j
        # Index at which the data array to be sent, ends
        end = start + chunk
        # Send the relavant chunks of data from rank 0 to all other ranks
        comm.Send([randx[start:end], chunk, MPI.DOUBLE], dest=j, tag=j)
        comm.Send([randy[start:end], MPI.DOUBLE], dest=j, tag=j*2)

      # rank 0 keeps the first chunk+rem elements
      randx_split = randx[0:chunk+rem]
      randy_split = randy[0:chunk+rem]
      # rank 0 contribution to pi_est
      pi_est = pi_simulation_std(randx_split,randy_split)
      return pi_est
    else:
      # rank 0 is the only rank, so use all points to estimate pi
      pi_est = pi_simulation_std(randx,randy)
      return pi_est
  else:
    # Initialise the arrays to be received by each rank from rank 0
    randx_split = np.zeros(chunk)
    randy_split = np.zeros(chunk)
    comm.Recv([randx_split, chunk, MPI.DOUBLE], source=0, tag=rank)
    comm.Recv([randy_split, chunk, MPI.DOUBLE], source=0, tag=rank*2)
    # Estimate of pi from each rank other than rank 0
    pi_est = pi_simulation_std(randx_split,randy_split)
    return pi_est

# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ #
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ #
# Set number of trials
N = 5000000
# Seed for random number generator (for reproducibility)
seed = 999

# Syncronise all ranks (i.e. processors)
comm.Barrier()
# Start timing the program
wt = MPI.Wtime()

# Determine the length of the array that the various ranks will
# get for the split data, i.e. rank 0 gets data with length chunk+rem,
```

```
# all the other ranks get data with length chunk.
chunk = int(np.floor(float(N)/float(num_procs)))
rem = N - chunk*num_procs

# Generate the points via rank 0 only
comm.Barrier()
if rank == 0:
    print("No. of trials, seed: ", np.float(N), seed)
    print("chunk size, rem: ", chunk, rem)
    randx,randy = generate_random_points(N,seed)
else:
    randx,randy = 0,0
comm.Barrier()

# Perform the estimates of pi from each rank, which involves the sending (from rank 0)
# and receiving (by every other rank) of data if num_procs > 1
pi_est[rank] = pi_split_sendrecv(N,seed,chunk,rem,num_procs,randx,randy)

# Use MPI reduce sum to fill up array of pi estimates from each rank
comm.Barrier()
if num_procs > 1:
   comm.Reduce([pi_est, MPI.DOUBLE], [pi_est_combined, MPI.DOUBLE], op=MPI.SUM, root=0)
comm.Barrier()

# Finally, print out the result, which is the sum of the pi estimates from each rank,
# divided by the number of processors
if rank == 0:
    print("Pi = ", np.round(np.sum(pi_est_combined)/num_procs,8))
    print("Time taken to compute pi (seconds)", MPI.Wtime()-wt)
else:
    pass
comm.Barrier()
```

Running the above script should produce (more or less) the following for 2 processors:

```
[user@login serial] $ mpirun -np 2 python main_std_mpi.py
No. of trials, seed:  5000000 999
Pi =  3.1411504
Elapsed time (seconds): 8.569018125
```

and

```
[user@login serial] $ mpirun -np 3 python main_std_mpi.py
No. of trials, seed:  5000000 999
Pi =  3.1411504
Elapsed time (seconds): 6.18495607
```

for 3 processors.

You should run a few of these with different numbers of processors (if possible), and notice the time taken will not continue to decrease as the number of processors is increased due to the additional time taken for communication between processors (i.e. increased overhead). Also feel free to change (increase) the number of random points (for a more precise estimate of π) and again, notice the speedup.

3. Useful links

- Website: http://www.chpc.ac.za
- Wiki: http://wiki.chpc.ac.za/
- Apply for resources: http://users.chpc.ac.za

- Helpdesk: https://users.chpc.ac.za/helpdesk/tickets/submit/
  (or email helpdesk@chpc.ac.za)

4. Appendix

4.1 Job submission with PBS Pro (see also `man qsub`)

4.1.1 Via script (i.e. qsub `job.sh`)

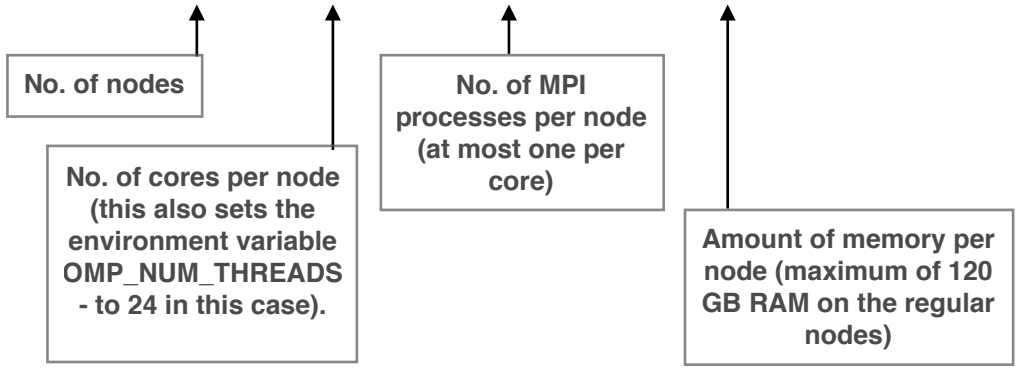
where `job.sh` contains the following PBS Pro directives:

```
#PBS -N MCpiParallel          % Give your particular job a name
                              % (no spaces allowed) that helps you
                              % identify it when checking qstat.

#PBS -q normal                % Name of the queue to be put
                              % on. “normal” is a standard.

#PBS -P SHORTNAME             % Specify the shortname for the
                              % relevant project registered on the
                              % system.

#PBS -l select=1:ncpus=24:mpiprocs=2:mem=120gb
```




```
#PBS -l walltime=00:05:00     % Set as close as possible to the
                              % expected time it will take for
                              % your job to complete (format:
                              % HH:MM:SS).

#PBS -o /path/to/stdout
#PBS -e /path/to/stderr       } Set paths for writing of standard output & error.

#PBS -m abe -M your_email@address % To receive an email when your
                                  % job begins (‘b’), aborts (‘a’),
                                  % ends (‘e’).
```

NB! Your job should not write out any data to your home directory (/home/yourusername), but rather to the lustre file system (/mnt/lustre/users/yourusername/).

4.1.2 Interactively

Instead of submitting a job via a script to the scheduler, it is often useful to be able to get access to a compute node for either compiling the necessary code (NB! this is the recommended way to go about any compilations) or for testing out small jobs first before scaling up to multiple nodes. The way to achieve this is to simply write out, in one line, the contents of the script as in the previous example (4.1.1) but including the `-I` flag, i.e. at the login node:

```
qsub -I -q smp -P SHORTNAME -l select=1:ncpus=24:mpiprocs=12 -
l walltime=01:00:00
```

where the `smp` queue is chosen to allow access to a single compute node only.

Note that for jobs that are parallelised with `openmp`, the OMP_NUM_THREADS variable is set equal to `ncpus` (and in this case one should probably set `mpiprocs=1` or equivalently leave the latter out).

Finally, note that there is a very useful option to keen an interactive session running in the background should it be necessary to have one open for a while but also to be able to logout of the cluster and return to it later. One program for this is `screen`: by running it and then launching your interactive job, you may detach that particular screen (`ctrl a d`) and exit the cluster, and reattach said screen via `screen -r`. See `man screen` for more details.