

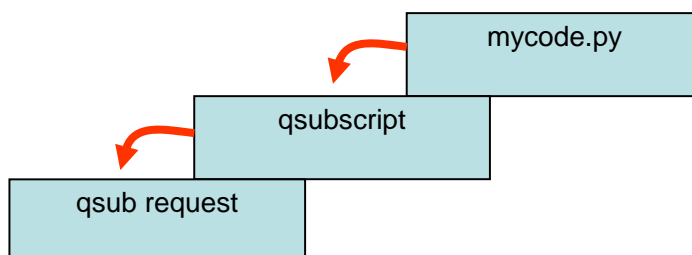
## Using Star-P with Python in Batch Environments

ISC Star-P software enables parallel computing in Python, on SMP machines or distributed x86/64 clusters, under work load managers such as PBS Pro. This note provides an illustration of the simple steps needed to make it possible. The note will focus on clusters as the operation on a multi-core SMP machine is even simpler.

First, of course, Star-P server software must be installed on the cluster nodes. The installation of Star-P includes installing a pre-tested version of Python and supported Python modules, such as NumPy. As part of the installation, configuration checks are performed to ensure that a compatible shared file system is present and that a keyless ssh connectivity is enabled and operational between the cluster nodes in all combinations, including each node to itself.

The basic process of using Star-P with Python for parallel batch computing consists of three steps:

1. Creating the “mycode.py” file that contains the Star-P/Python code written for parallel computing
2. Creating a “qsubscript” script file that contains the setup instructions and invokes mycode.py to execute the intended computation
3. Submitting the “qsubscript” job to PBS for queuing and execution, using “qsub” request command.



Once the “mycode.py” and “qsubscript” files are ready, the sequence of events is as follows:

1. Submit the “qsub” request to PBS,
2. PBS schedules the job
3. PBS starts the job, on schedule, executes the “qsubscript” script.
4. The “qsubscript” sets up the configuration and executes the “mycode.py” code.
5. Done.

Let’s now look into the details.

The command submitting the qsub job request to PBS looks like that:

```
qsub <myarga> -z qsubscript
```

where <myarga> is an optional list of argument, and `-z qsubscript` is a switch invoking execution of the qsubscript script.

The “qsubscript” script file consist of three parts:

Part 1 (obtaining information about available cluster nodes, from PBS, for Star-P):

```
# copy PBS nodefile to Star-P machine file, this is accomplished by:
cp $PBS_NODEFILE/shared/starp/server/path/config/machine_file.system_default
```

Part 2 (choosing the head node):

```
# pick the first node in the list as head-node, for Star-P to connect to
HEADNODE="$(head -1 /shared/starp/server/path/config/machine_file.system_default)"
```

Part 3 (executing the “mycode.py” code):

```
# execute “mycode.py” place results in “myoutput.out”
/shared/server/path/bin/python /shared/home/dir/mycode.py $HEADNODE 8 >
/shared/home/dir/myoutput.out
```

The “mycode.py” code file consist of two parts:

Part 1 (importing modules and configuring Star-P server for computation):

```
# import starp and Python sys module, and time module for benchmarking
import starp, time, sys

# Get the name of node to connect to, and number of processes
node = sys.argv[1]
np = int(sys.argv[2])

# Open a connection
starp.defaultConnect(node, '/shared/starp/server/path/', numProcs=np)
```

Part 2 (the actual computation code, see example below)

```
# start of Star-P/Python code
... ..
# end of Star-P/Python code
```

## ***Mycode.py example***

```
# import starp and Python sys module, and time module for benchmarking
import starp, time, sys

# Get the name of node to connect to, and number of processes
node = sys.argv[1]
np = int(sys.argv[2])

# Open a connection
starp.defaultConnect(node, '/shared/starp/server/path/', numProcs=np)

# Perform the computation: Invert 32 matrices of 1000x1000 size
start_time = time.time()
x = starp.random.rand(1000,1000,32)
y = starp.ppeval('numpy.linalg.inv',x)
end_time = time.time()

# Print the timing results
print 'Total time: %e s, number of cores = %d\n' % (end_time-start_time, starp.np())
```

### ***Performance chart – matrix inversion in task-parallel***

The above example exhibits linear scaling as more processors and nodes are added.

