Distributed Message Queue Based Linux Cluster

Rachee Singh
Emaad Ahmed Manzoor
Mohit Yadav
Murtuza Kutub

The Purpose

We want to construct a Linux cluster which is reasonably fault tolerant for executing simple parallel algorithms across multiple nodes, with a visible speedup. We intend put together a system that can serve as a useful pedagogical tool for a parallel computing course's lab sessions.

About Distributed Message Queues

We make use of queues which are distributed across nodes and can be accessed via TCP sockets. The parallel computation is broken down to a producer consumer model. Producers push jobs to the queue and consumers pull jobs from the queue. These "jobs" on the queue contain arbitrary data.

There are many distributed message queues available today, for instance: RabbitMQ and ZeroMQ. These are quite popular but happen to be very generic. Our task involves a special subset of what a queue can serve: job queuing. We need a mechanism to farm out jobs to workers, monitor their progress and do the necessary cleanup if a worker fails to perform.

Celery was another such specialized solution, but at first glance, seemed to be heavily tied in to Django applications. Gearman was another that seemed very convenient; it brought with it an entire framework to farm out jobs, persist queues, accommodate job server failure and the works, but we settled with something lighter until we realized the need of a more sophisticated solution.

Beanstalkd was a fitting solution. It is the lightest work-queue we could find that supported some form of queue persistence and worker failure recovery. Also, Beanstalkd is easy to setup on each node of the cluster.

Preparing the Ingredients

Here are some packages that are required on every node of the cluster:

- ***python-setuptools or python-distribute:*** These are group of utility scripts to install Python modules. Install this using your package manager.
- ***python-yaml or pyYAML:*** Python libraries to parse YAML matter. This package also can be installed via your package manager.
- ***Beanstalkd:*** The Beanstalk queue server. This package has to be installed from source by cloning the most recent version from the source at Github.
  To install, change into the source directory of Beanstalkd to compile the sources and run tests:

  ```
  $ make
  $ ./check.sh
  ```

  The installation would have created a "beanstalkd" binary in the installation directory. Run the Beanstalk server on port 11300 (this is arbitrary) on your local host using it:

  ```
  $ ./beanstalkd -l 127.0.0.1 -p 11300
  ```

- ***Beanstalkc:*** Python bindings for Beanstalkd; We need to install this from source too. You'll just need to switch to the Beanstalkc source directory and run the provided install script:

  ```
  $ sudo python setup.py install
  ```

Testing The Combination

With the beanstalkd server running, we can test this setup. On the python interpreter, type the following sequence of commands:

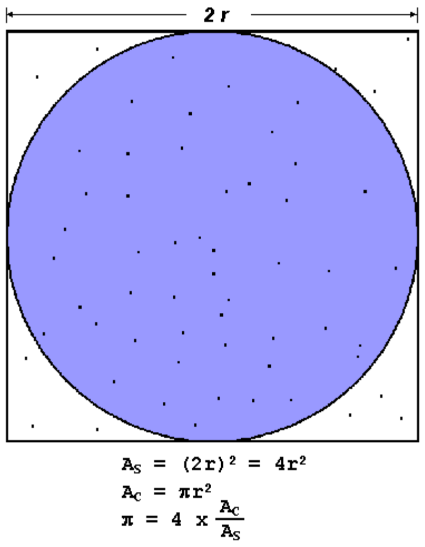
```
>> import beanstalkc
>> bean = beanstalkc.Connection(host='127.0.0.1', port=11300)
>> bean.put('Job1')
1
>> job = bean.reserve()
1
>> job.body
Job1
>> job.delete()
```

The listing above created a connection to the beanstalkd queue and added a new task; tasks in Beanstalkd are always strings, any non-string data you send across must be serialized. We then reserved a job from the queue. This is typically done by a worker that intends to work on that job. Once the job is reserved, you can retrieve its body and do some work on it. When the work is done, you need to notify the Beanstalkd server by deleting the job from the queue. If a worker fails to delete a job, the worker is forsaken and its job is placed back onto the tail of the queue after a specified time interval.

Application: Computing π

We picked a simple application: π computation, which can be effectively parallelized to test our model. In this section we will describe the algorithm briefly in the context of our distributed message queue setup.

The code for the algorithm was derived from that presented in Introduction to Parallel Computing. This algorithm can be easily parallelized using a master-worker setup:

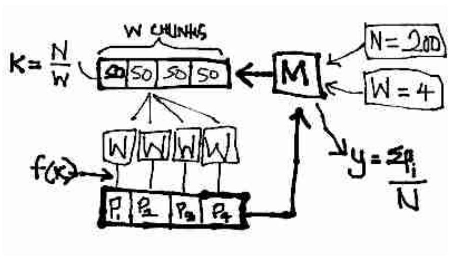


- The master allocates to each worker the number of points to generate within the square. The number of points is derived simply be splitting the total number of points we desire among the number of workers we expect to have. More points lead to more accurate estimation.
- The worker receives the number of points it must generate from the master, and then generates those points, or throws those darts into the square. It calculates the number of darts that land within the circle and returns that number to the master.

System Design

The system follows a simple producer-consumer model. There are two queues; the one that the master M pushes to and workers consume from, called message-for-worker, and the one that the workers push to and the master consumes from, called message-for-client.

M accepts as a parameter N (the number of points to be considered) and W (the number of workers it can expect to be active). It then splits the N points equally among W workers and pushes each worker's quota to message-for-worker queue.



Each worker consumes a single item k from message-for-worker and performs f(k): throwing k points into the square and then counting the number of points falling within the circle. Each worker pushes this count to message-for-client.

The master waits for all workers to complete, and then sums all the workers' responses in message-for-client, divides by the total number of darts N and multiplies the resultant by four to get the estimated value of π.

Implementation: Computing π

We implemented the above algorithm on our 4 node distributed message queue based cluster. The complete and most recent code can be retrieved from the source. In this section we will describe what the code does, in detail.

The Client Code

The Client or master pushes jobs to the msg-for-worker queue and expects the result of computations by workers in the msg-for-client queue. We now associated convenient names to these queues:

client.py

```python
import beanstalkc

QUEUE_TO_USE = "msg_for_worker"
QUEUE_TO_WATCH = "msg_for_client"
QUEUE_TO_IGNORE = "default"
```

“default” is a queue that is created by default by the Beanstalkd server, it would do us good to ignore it.

client.py
```python
N = 10 ** 6     # Number of darts to throw in total
K = 2           # Number of jobs to split the work into
TTR = 40        # Deadline for jobs to complete (seconds)

beanstalk = beanstalkc.Connection(host="127.0.0.1", port=11300)

beanstalk.use(QUEUE_TO_USE)
beanstalk.watch(QUEUE_TO_WATCH)
beanstalk.ignore(QUEUE_TO_IGNORE)
```

In the above code listing, N is the total number of points we are considering, and directly affects the accuracy of our estimate and the computing power required. K needs to be set equal to the number of worker nodes you expect to see active. Leaving these unchanged won’t break any functionality, but it will take longer duration to compute the counts if there are fewer workers.
It also tells the Beanstalkd server which queues to watch, or consume from, use, or push to, and ignore.

client.py
```python
use_queue_was_full = False
while True:  # Block until the queue we push to is empty
    ready_jobs = beanstalk.stats_tube(QUEUE_TO_USE)["current-jobs-ready"]
    reserved_jobs = beanstalk.stats_tube(QUEUE_TO_USE)["current-jobs-reserved"]
    if (ready_jobs + reserved_jobs ) == 0:
        break
    else:
        use_queue_was_full = True
```

The above segment of code enables server fault tolerance, which we discuss in a later sub-section.

cleint.py
```python
points_per_job = int(N / K)
if not use_queue_was_full:
    for i in range(K):  # Push all the jobs to the queue
        print "Put job", str(i), "to job queue"
        beanstalk.put(str(points_per_job), ttr=TTR)
    print

```

```
total_circle_count = 0
responses_received = 0
for i in range(K):
    job = beanstalk.reserve()
    responses_received += 1
    total_count = total_circle_count + int(job.body)
    job.delete()
    print "Received " + str(responses_received) + "/" + str(K) + " responses"
beanstalk.close()
```

The above segment pushes the worker quotas to the queue and waits for results. On receiving results, it accumulates them in a variable and then closes the connection to Beanstalkd.

client.py

```
pi = (4.0 * total_count) / (responses_received * points_per_job)

print
print "The value of pi is " + str(pi)
print "That took ", str(time.time() - start_time), " seconds!"
```

In the above listing, we use the accumulated sum of the circle counts returned by the workers to compute the estimated value of π , in accordance with the algorithm described earlier.

The Worker Code

The worker code simply implements the algorithm in the context of the system, both of which have been described above. Most of the boilerplate matches that in the client, the differences have been briefly summarized below:

worker.py

```
while True:                              # Indefinitely wait for jobs to do

    print "Connecting to job server..."
    beanstalk = beanstalkc.Connection(host='127.0.0.1', port=11300)

    beanstalk.watch(QUEUE_TO_WATCH)      # Reserve jobs from this queue
    beanstalk.ignore(QUEUE_TO_IGNORE)    # Ignore this queue for reservation

    print "Waiting for jobs..."
    job = beanstalk.reserve()            # Blocks until a job is available
```

This makes our worker a daemon of sorts; it sits around indefinitely, waiting for jobs to do, and on completion of a job, goes back to waiting for the next job to work on.

worker.py

```python
n = int(job.body)                          # n = number of points to throw in the square
start_time = time.time()

# Algorithm Reference: Monte-Carlo estimation of Pi
# Source: https://computing.llnl.gov/tutorials/parallel_comp/#ExamplesPI

square_edge = 2.0                           # Square edge length
center_x, center_y = 1.0, 1.0               # Circle center coordinates
radius = 1.0                                # Circle radius
circle_count = 0                            # Counts points within the circle

print
print "Generating", str(n), "points..."
try:
    for i in xrange(n):
        x = random.random() * square_edge         # 0 <= x < square_edge
        y = random.random() * square_edge         # 0 <= y < square_edge
        if (x - center_x) ** 2 + (y - center_y) ** 2 <= radius ** 2:
            circle_count = circle_count + 1
except Exception:
    print "Job failed.", n
    job.release()

beanstalk.use(QUEUE_TO_USE)                # Push the calculated number of points
beanstalk.put(str(circle_count))           # in the circle to the client queue

job.delete()                               # Tell the job server you're done
beanstalk.close()

print "Finished job in", str(time.time() - start_time), "seconds"
print
```

This simply implements the estimation algorithm discussed in an earlier section. The result of the algorithm is pushed to QUEUE-TO-USE by the worker.

Computing π: Running the Code

Getting the code running involves simply running the worker Python script on a number of terminals to spawn as many workers as you need, and then starting the client Python code to farm out jobs to the workers. If you're attempting this in a distributed setup, ensure you make the necessary modifications in the server address and port.

Computing π: Fault Tolerance

Worker Fault Tolerance

Worker fault tolerance is baked in to the Beanstalkd architecture. Workers work on jobs from the queue by reserving them. On completion, they are expected to delete the job from the queue, which is a way of notifying the server that it has completed its job. If, however, a worker fails to delete a job within a configurable time interval TTR, the job is re-added to the tail of the queue. Thus, in the worst case when all but one worker fail, the single worker will sequentially consume and work on all the items in its queue.

Server Fault Tolerance

To provide some form of persistence for the in-memory queue, Beanstalkd supports an option called binlog. Enabled with the b option to the Beanstalkd executable, this periodically stores the state of the queue to a flat file in the directory specified. On restarting from server failure with the -b switch, the in-memory queue is reconstructed from the files in this directory.

However, to take advantage of server restarts and the binlog, the client code will need to be augmented with some failure-resume intelligence. In particular, the client must not re-push data into its queue if there exist items remaining from before the server failed. This is enabled with this bit of code in the client:

client.py

```python
use_queue_was_full = False
while True:  # Block until the queue we push to is empty

    ready_jobs = beanstalk.stats_tube(QUEUE_TO_USE)["current-jobs-ready"]
    reserved_jobs = beanstalk.stats_tube(QUEUE_TO_USE)["current-jobs-reserved"]
    if (ready_jobs + reserved_jobs ) == 0:
        break
    else:
        use_queue_was_full = True
```

This uses Beanstalkd's statistics queries to gain information about the number of items in the queue it's pushing to. Until that queue is empty, it blocks and waits.

Application: Matrix Multiplication

We also implemented a simple (non-work optimal) parallel algorithm for matrix multiplication. In this case, the master (client) has the matrices A and B which have to be multiplied. The jobs that the client adds to the message-for-worker (similar to the previous application) consist of the i_{th} row of matrix A and j_{th} column of matrix B.

each worker picks one such job from the queue and compute the entry C_{ij}. Then the worker puts the computed value into a shared memory accessible by the client. For this purpose, we used memcached which is distributed in-memory key value store for small chunks of arbitrary data.

Along with storing the result of the multiplication in shared memory, the worker sends a confirmation to the client that it finished computation. This is to ensure that the client while extracting results knows that all workers have finished execution.

Implementation: Matrix Multiplication

The Client Code

client.py

```
port beanstalkc
port memcache

EUE_TO_WATCH = "msg_for_client"
EUE_TO_USE = "msg_for_worker"
EUE_TO_IGNORE = "default"

We want to multiply matrix 1 and matrix2
trix1 = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
trix2 = [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ]

int "Connecting to beanstalkd server.."
anstalk = beanstalkc.Connection( '10.4.12.63', 11300 )

anstalk.use( QUEUE_TO_USE )
anstalk.ignore( QUEUE_TO_IGNORE )
anstalk.watch( QUEUE_TO_WATCH )

= len( matrix1 )

 = memcache.Client(['10.4.12.63:11211'], debug=0)
trix3 = [ [] * n ]

int 'Sending jobs to jobs queue..'
r i in range(n):
  for j in range(n):
      print 'Sending a row and column to job queue'
      stringRow = ' '.join( map( str, matrix1[ i ] ) )
      stringColumn = ' '.join( map( str, [ row[j] for row in matrix2 ] ) )
      # Separating the row and column with a new line
      beanstalk.put( str(n*i) + '\n' + str(j) + '\n' + stringRow + '\n' + stringColumn )
```

```

int 'Sent all jobs to queue.'

Each of the jobs returned would contain confirmation if
Cij was completed
r i in range( n**2 ):
  job = beanstalk.reserve()
  print 'Confirmation: Worker finished computing C' + job.body

int 'All workers finished.'
int 'Extracting result from memory..'

int 'Got the result, Here it is: '
r i in range(n):
  for j in range(n):
      print ( mc.get( str( i*n + j ) ) ),
  print
```

The Worker Code

worker.py

```
port beanstalkc
port memcache
port time
port operator

EUE_TO_WATCH = "msg_for_worker"
EUE_TO_USE = "msg_for_client"
EUE_TO_IGNORE = "default"

ile True:                             # Indefinitely wait for jobs to do

  print "Connecting to job server..."
  beanstalk = beanstalkc.Connection(host='10.4.12.63', port=11300)

  beanstalk.watch(QUEUE_TO_WATCH)      # Reserve jobs from this queue
  beanstalk.ignore(QUEUE_TO_IGNORE)    # Ignore this queue for reservation

  print "Waiting for jobs..."

  job = beanstalk.reserve()            # Blocks until a job is available

  print "Reserved a job.."

```

```
[ strI, strJ, stringRow, stringColumn ] = job.body.split( '\n' )
I = int( strI )
J = int( strJ )
row = map( int, stringRow.split() )
column = map( int, stringColumn.split() )

n = len( row )

start_time = time.time()

print 'Computing Cij..'
try:
    result = reduce( operator.add, map( operator.mul, row, column ) )
except Exception, e:
    print e
    print "Job failed.", n
    job.release()

mc = memcache.Client(['10.4.12.63:11211'], debug=0)
print 'Wrote result to',  str(I+J)
mc.set( str( I + J ), result )
beanstalk.use(QUEUE_TO_USE)
beanstalk.put( strI + strJ )

job.delete()                          # Tell the job server you're done
beanstalk.close()

print "Finished job in", str(time.time() - start_time), "seconds"
print
```

Future Work

- **Other distributed job queues:** We would like to experiment with other distributed job queues, for instance Gearman. In case there happens to be a better alternative, we would like to implement the setup described in this document using it.
- **A library of common parallel computing related algorithms:** We would like to add more parallel algorithms to the small list algorithms we have implemented on this cluster. We plan to begin with the most common ones like list ranking, sorting, tree traversal etc.