OpenNebula - Bug #1024
unefficient use of threads under heavy load

12/09/2011 05:05 PM - Maxence Dunnewind

Status:	Closed	**Start date:**	12/09/2011
Priority:	Normal	**Due date:**	
Assignee:	Tino Vázquez	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
Resolution:	worksforme	**Pull request:**	
Affected Versions:	OpenNebula 3.0		

Description

I'm working with ONE (3.0) to deploy lot of VMs in a short time. One of the things I saw looking at the log file of the deployed VMs was that the last vms of the submitted "pool" were waiting for many dozens of seconds before starting the vmm_exec code.

Talking with tino, he said that it could be due to the limited number of threads (15 by default) used for the vmm script.

Looked at ActionManager.rb and VirtualMachineDriver.rb, put some debugs, and saw logs like :

> [Thu Dec 08 09:56:07 +0100 2011] [vmm_exec] Action is {:threaded=>true, :method=>#<Method: ExecDriver#deploy>, :args=>["10785", "mynode1", "/srv/cloud/one/var/10785/images/deployment.0", "-"], :id=>10785}, action_queue has 43 items and 4 threads are running

We saw that even if there are 40+ actions waiting in the queue, only 4 threads of 15 were running.

Another interesting thing is that I got lot of line like :

> [Fri Dec 09 11:16:02 +0100 2011] [vmm_exec] Action is nil, action_queue has 14 items and 2 threads are running

Meaning that no "correct" action has been found in the queue.

Start looking at the code, interesting things I saw :
- in ActionManager::run_action :

```
def run_action
  action = get_runable_action

  if action
    [...snip...]
  end
```

It means that only one action (if anything but nil is returned by get_runable_action) can be started when run_action is called.
But this method is called from a synchronized code. The issue is that when you deploy a lot of VMs, there is a lot of contention on this mutex (at least 2 request per vm, one for the deploy, one for the prob).
So when the script runs one action, it then frees the mutex and try to re-acquire it. But because of contention, lot of requests can pass before the code in start_listener gets the mutex again.
So for one action triggered, many more than one item can be put in the queue, and because of contention and time between 2 calls of

run_action, not enough threads are started.

The (dirty) patch I did is to put all the code of run_action in a loop:

```
def run_action
  [@concurrency - @num_running, @action_queue.size].min.times do
    action = get_runable_action

    if action
      @num_running += 1
      [...snip...]
    end
  end
```

That way, we will always run as many threads as we can.

Logs with this patch looks like :

```
[Fri Dec 09 11:22:10 +0100 2011] [vmm_exec] Action is {:threaded=>true, :method=>#<Method: ExecDriver#deploy>,
:args=>["12474", "paradent-9.rennes.grid5000.fr", "/srv/cloud/one/var/12474/images/deployment.0", "-"], :id=>12474},
action_queue has 20 items and 14 threads are running
```

We saw that this time, as many thread as we can are running.

Second point: I saw that only 1 vm per node seems to be deployed at a time, even if I set -h to more than 1

Looking at the code, it seems that the contention point is into VirtualMachineManager::get_first_runnable :

```
action_index=nil
@action_queue.each_with_index do |action, index|
  if action[:args][HOST_ARG]
    if !@hosts.include?(action[:args][HOST_ARG])
      action_index=index
      break
    end
  else
    action_index=index
    break
  end
end
```

If I understand this code correctly, it limits the number of action in parrallel on a node to ... one.
Since my tm code is supposed to be concurrency-ready (modified version of tm_lvm), I updated this code to allow many deployment at same time.
I did not just removed it because I don't know if it can be usefull somewhere else.

With those changes, I have been able to deploy 320 VM on 10 nodes in less than 9 minutes.

Maxence

History

#1 - 12/09/2011 11:46 PM - Ruben S. Montero

Hi,

First thanks for your comments, and for the time digging into this ;)

Anyway:

1.- The code (run_action) is call within a critical section to avoid multiple threads updating shared variables at the same time. There should be no contention as the thread enters there, creates a new one an leave the section.

2.- As far as I see the loop included in your patch has the same goal as the one in the main thread (start_listener)

```
while ((@concurrency - @num_running)==0) || empty_queue
```

However, your point about the number of concurrent action on the same host is true and there not to overload some hypervisors (e.g. KVM+libvirt does not behave well if starting > 2 or 3 VMs at the same time). As you point out that will limit the number of VMs deployed at the same time. But note that this limitation only applies to the virtualization drivers and not the transfer manager ones...

I'll try to look the reason for "43 items and 4 threads", if you start the sample driver in VirtualMachineDriver.rb and feed it through stdin I see the 15 threads running...

Thanks again for your feedback

#2 - 12/10/2011 11:28 AM - Maxence Dunnewind

Hey,

First about 2. This is not exactly the same goal. With the loop you're talking about, the behaviour is (if I understand it correctly) :

```
while ((@concurrency - @num_running)==0) || empty_queue
  @threads_cond.wait(@threads_mutex)
```

in "plain text" :

```
while true:
    while there is no thread available or no job to schedule :
        wait for the mutex
    try to schedule at more one job
```

My loop has a different behaviour which is :

```
when called (ie. when I have the mutex):
  while there is at least one thread available and one item in the queue:
    try to schedule one job
```

this loop is done without releasing the mutex, so without need to wait to get it again.

About 1. I should test a little more if the loop in start_listener really waits on the Condition for a "long" time, or if the limitation is due to something else (concurrency limit per host).

About the "only applies to TM",saw that yes (directly inherits OpenNebulaDriver).

Anyway, 9 minutes for 320 vm seems to be a good result (9 minutes is the total time, including request to occi, scheduling,deployment, startup until they acan be reached by ping). thanks ONE :)

Another extra question, why not use a Queue object to avoid reimplementing the synchronization by hand ? Guess it's because of the need to manage cancel/delete ?

#3 - 12/10/2011 04:00 PM - Ruben S. Montero

Hi,

The logic is as follows

```
start_listener
  unless no finalize
    if queue is empty or reached max number of threads
      sleep
    end

    run an action
  end
```

The if above is implemented with a while because when the thread wakes up needs to reevaluate the condition (posix threads although don't know if its needed in the Ruby implementation)

So there is no release nor wait for the mutex if there are pending actions or slots available. Whenever an action thread ends or a new action arrives the listener thread is signaled...

This is more or less what queue gives you but we also need the signaling between threads to trigger events.

I am really interested in finding out what helped you in your setup see if we can apply that upstream.

Re-thinking about your logs the action nil is normal if a new action can not be scheduled (hosts already running an action). Also we may have some monitoring vs deployment issue (i.e. while monitoring a host the action slot for the host is not free and deploys are delayed)... If that's the case we could test less conservative options (e.g. let monitor run parallel with other actions, configure the number of actions per host...)

Thanks

Ruben

#4 - 12/10/2011 05:13 PM - Maxence Dunnewind

Hi,

> *So there is no release nor wait for the mutex if there are pending actions or slots available. Whenever an action thread ends or a new action arrives the listener thread is signaled...*

I don't understand the code like that. For what I understand, It would be the case if the the "while true" was inside the synchronize block.
But in our case, let's say we have 2 items in the queue and the mutex is free, the process will be :

```
while true
  take the mutex
    we have available thread and queue is not empty, so don't go in the loop
    run_action (ie, run at most one task)
  free the mutex
```

Let's say during the time the process has the mutex, some other processes ask for it (let's say calling trigger_action).
After having started one task, the process frees the mutex, which is took by one of the waiting call, then freed, then took by some other process, then ...
The process which calls run_action will eventually take the mutex again for sure, but it will never be able to run more than one task each time it gets the mutex. So with lot of vm deployed "at same time", lot of processes will be waiting for the mutex inside trigger_action(at least for deploy and poll calls), and so maybe a "long" time will be needed between two call of "run_action".

Am I understanding something incorrectly ?

> *This is more or less what queue gives you but we also need the signaling between threads to trigger events.*

Not sure what you call "signaling" here.

> *I am really interested in finding out what helped you in your setup see if we can apply that upstream.*

Will be really happy to help you. The most efficient change seems to be the add of "[@concurrency - @num_running, @action_queue.size].min.times" code.

> *Re-thinking about your logs the action nil is normal if a new action can not be scheduled (hosts already running an action). Also we may have some monitoring vs deployment issue (i.e. while monitoring a host the action slot for the host is not free and deploys are delayed)... If that's the case we could test less conservative options (e.g. let monitor run parallel with other actions, configure the number of actions per host...)*

Yes I saw that indeed. I think the monitoring vs deployment issue can be real. The queue quickly contains a lot of "poll", so when deploying lot of vms quickly on a few hosts, the first vms will be deployed quickly, but the latest will need to wait a long time before being submitted, because lot of poll will be in queue. One solution would be to indeed allow poll even if some other actions are running on the host.
I will try this quickly (monday). I already patched get_first_runnable in VirtualMachineDriver.rb to allow exec of deploy even if some action if already running on host, but did not see real speedup.

Maxence

#5 - 02/02/2012 10:01 AM - Maxence Dunnewind

Well, made a few more checks.
The real bottleneck which causes unefficient use of threads is that all events are put in the same queue (well ... that's an array) and the condition is "don't run 2 events at same time on a same host".
When you submit many (dozens of) vms in a short period of time, the latest 'deploy' order will be delayed because of all the "poll" items that are put in queue.
I added a simple condition like "if the order is a poll, run it, even if another order is already running at same time on the host", which as immediate effect to really use all the threads.

Maxence

#6 - 05/08/2013 09:12 PM - Ruben S. Montero

- Status changed from New to Closed
- Resolution set to worksforme

We cannot reproduce it.