The following answer covers the 3 main aspects mentioned in title - number of executors, executor memory and number of cores. There may be other parameters like driver memory and others which I did not address as of this answer, but would like to add in near future.

**Case 1 Hardware - 6 Nodes, and Each node 16 cores, 64 GB RAM**

Each executor is a JVM instance. So we can have multiple executors in a single Node

First 1 core and 1 GB is needed for OS and Hadoop Daemons, so available are 15 cores, 63 GB RAM for each node

*Start with how to choose number of cores*:

```
Number of cores = Concurrent tasks as executor can run

So we might think, more concurrent tasks for each executor will give better performance. But r
any application with more than 5 concurrent tasks, would lead to bad show. So stick this to 5.

This number came from the ability of executor and not from how many cores a system has. So the
even if you have double(32) cores in the CPU.
```

*Number of executors:*

```
Coming back to next step, with 5 as cores per executor, and 15 as total available cores in one
3 executors per node.

So with 6 nodes, and 3 executors per node - we get 18 executors. Out of 18 we need 1 executor

This 17 is the number we give to spark using --num-executors while running from spark-submit s
```

*Memory for each executor:*

```
From above step, we have 3 executors  per node. And available RAM is 63 GB

So memory for each executor is 63/3 = 21GB.

However small overhead memory is also needed to determine the full memory request to YARN for
Formula for that over head is max(384, .07 * spark.executor.memory)

Calculating that overhead - .07 * 21 (Here 21 is calculated as above 63/3)
                          = 1.47

Since 1.47 GB > 384 MB, the over head is 1.47.
Take the above from each 21 above => 21 - 1.47 ~ 19 GB

So executor memory - 19 GB
```

Final numbers - Executors - 17, Cores 5, Executor Memory - 19 GB

**Case 2 Hardware : Same 6 Node, 32 Cores, 64 GB**

5 is same for good concurrency

Number of executors for each node = 32/5 ~ 6

So total executors = 6 * 6 Nodes = 36. Then final number is 36 - 1 for AM = 35

Executor memory is : 6 executors for each node. 63/6 ~ 10 . Over head is .07 * 10 = 700 MB. So rounding to 1GB as over head, we get 10-1 = 9 GB

Final numbers - Executors - 35, Cores 5, Executor Memory - 9 GB

**Case 3**

The above scenarios start with accepting number of cores as fixed and moving to # of executors and memory.

Now for first case, if we think we dont need 19 GB, and just 10 GB is sufficient, then following are the numbers:

cores 5 # of executors for each node = 3

At this stage, this would lead to 21, and then 19 as per our first calculation. But since we thought 10 is ok (assume little overhead), then we cant switch # of executors per node to 6 (like 63/10). Becase with 6 executors per node and 5 cores it comes down to 30 cores per node, when we only have 16 cores. So we also need to change number of cores for each executor.

So calculating again,

The magic number 5 comes to 3 (any number less than or equal to 5). So with 3 cores, and 15 available cores - we get 5 executors per node. So (5*6 -1) = 29 executors

So memory is 63/5 ~ 12. Over head is 12*.07=.84 So executor memory is 12 - 1 GB = 11 GB

Final Numbers are 29 executors, 3 cores, executor memory is 11 GB

***Dynamic Allocation:***

*Note : Upper bound for the number of executors if dynamic allocation is enabled. So this says that spark application can eat away all the resources if needed. So in a cluster where you have other applications are running and they also need cores to run the tasks, please make sure you do it at cluster level. I mean you can allocate specific number of cores for YARN based on user access. So you can create spark_user may be and then give cores (min/max) for that user. These limits are for sharing between spark and other applications which run on YARN.*

spark.dynamicAllocation.enabled - When this is set to true - We need not mention executors. The reason is below:

The static params number we give at spark-submit is for the entire job duration. However if dynamic allocation comes into picture, there would be different stages like

**What to start with :**

Initial number of executors (*spark.dynamicAllocation.initialExecutors*) to start with

**How many :**

Then based on load (tasks pending) how many to request. This would eventually be the numbers what we give at spark-submit in static way. So once the initial executor numbers are set, we go to min (*spark.dynamicAllocation.minExecutors*) and max (*spark.dynamicAllocation.maxExecutors*) numbers.

**When to ask or give:**

When do we request new executors (*spark.dynamicAllocation.schedulerBacklogTimeout*) - There have been pending tasks for this much duration. so request. number of executors requested in each round increases exponentially from the previous round. For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds. At a specific point, the above max comes into picture

when do we give away an executor (*spark.dynamicAllocation.executorIdleTimeout*) -

Please correct me if I missed anything. The above is my understanding based on the blog i shared in question and some online resources. Thank you.

**References:**


- http://site.clairvoyantsoft.com/understanding-resource-allocation-configurations-spark-application/
- http://spark.apache.org/docs/latest/configuration.html#dynamic-allocation
- http://spark.apache.org/docs/latest/job-scheduling.html#resource-allocation-policy