WHITE PAPER

HADOOP PERFORMANCE TUNING

Abstract

This paper explains tuning of Hadoop configuration parameters which directly affects Map-Reduce job performance under various conditions, to achieve maximum performance. The paper is intended for the users who are already familiar with Hadoop- HDFS and Map-Reduce.

Impetus Technologies Inc.
www.impetus.com

October 2009

HADOOP PERFORMANCE TESTING

Document Flow

The document contains Map-Reduce job workflow. It describes different phases of Map-Reduce operations and usage of configuration parameters at different stages in the Map-Reduce job. It explains the configuration parameters, their default values, pros, cons, and suggested values in different conditions. It also explains other performance aspects such as temporary space and Reducer lazy initialization. Further the document provides a detailed case study with statistics.

Definitions

The document highlights the following Hadoop configuration parameters with respect to performance tuning-

- **`dfs.block.size`** : Specifies the size of data blocks in which the input data set is split
- **`mapred.compress.map.output`** : Specifies whether to compress output of maps.
- **`mapred.map/reduce.tasks.speculative.execution`**: When a task (map/reduce) runs very slowly (due to hardware degradation or software mis-configuration) than expected. The Job Tracker runs another equivalent task as a backup on another node. This is known as speculative execution. The output of the task which finishes first is taken and the other task is killed.
- **`mapred.tasktracker.map/reduce.tasks.maximum`** : The maximum number of map/reduce tasks that will be run simultaneously by a task tracker.
- **`io.sort.mb`** : The size of in-memory buffer (in MBs) used by map task for sorting its output.
- **`io.sort.factor`** : The maximum number of streams to merge at once when sorting files. This property is also used in reduce phase. It's fairly common to increase this to 100.
- **`mapred.job.reuse.jvm.num.tasks`** : The maximum number of tasks to run for a given job for each JVM on a tasktracker. A value of –1 indicates no limit: the same JVM may be used for all tasks for a job.
- **`mapred.reduce.parallel.copies`** : The number of threads used to copy map outputs to the Reducer.

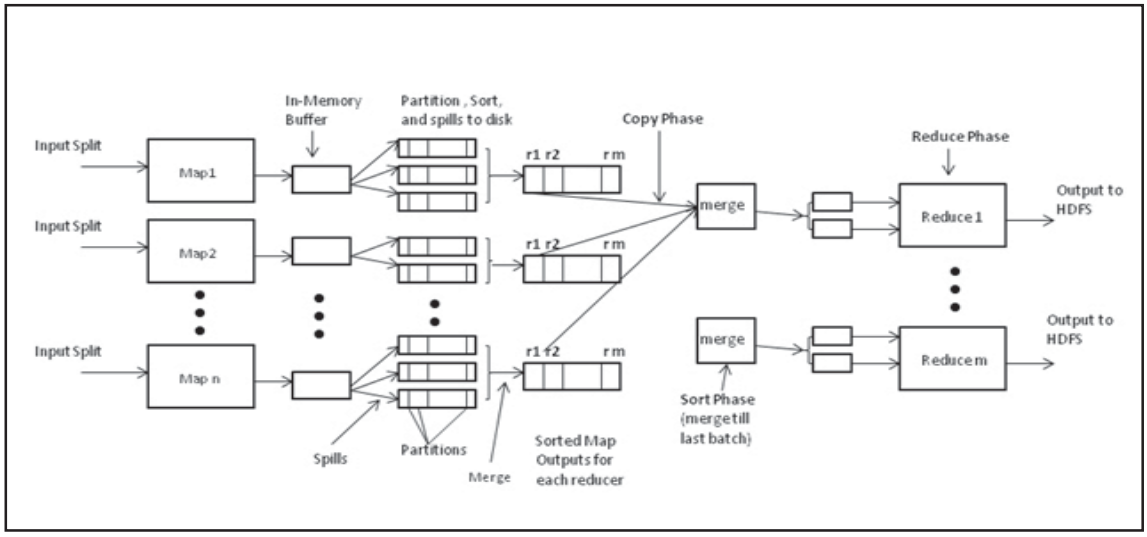
Map Reduce Work Flow

The diagram below explains various phases in Map-Reduce job and the data flow across the job.

Map Operations: Map task involves the following actions

- **Map Processing:** HDFS splits the large input data set into smaller data blocks (64 MB by default) controlled by the property **`dfs.block.size`**. Data blocks are provided as an input to map tasks. The number of blocks to each map depends on `mapred.min.split.size` and **`mapred.max.split.size`**. If **`mapred.min.split.size`** is less

HADOOP PERFORMANCE TESTING



than block size and **mapred.max.split.size** is greater than block size then 1 block is sent to each map task. The block data is split into key value pairs based on the Input Format. The map function is invoked for every key value pair in the input. Output generated by map function is written in a circular memory buffer, associated with each map. The buffer is 100 MB by default and can be controlled by the property **io.sort.mb**.

- **Spill:** When the buffer size reaches a threshold size controlled by **io.sort.spill.percent** (default 0.80 or 80%), a background thread starts to spill the contents to disk. While the spill takes place map continues to write data to the buffer unless it is full. Spills are written in round-robin fashion to the directories specified by the **mapred.local.dir** property, in a job-specific subdirectory. A new spill file is created each time the memory buffer reaches to spill threshold.
- **Partitioning** Before writing to the disk the background thread divides the data into partitions (based on the partitioner used) corresponding to the Reducer where they will be sent.
- **Sorting:** In-memory sort is performed on key (based on **compareTo** method of key class). The sorted output is provided to the combiner function if any.
- **Merging:** Before the map task is finished, the spill files are merged into a single partitioned and sorted output file. The configuration property **io.sort.factor** controls the maximum number of streams to merge at once; the default is 10.
- **Compression:** The map output can be compressed before writing to the disk for faster disk writing, lesser disk space, and to reduce the amount of data to transfer to the Reducer. By default the output is not compressed, but it is easy to enable by setting **mapred.compress.map.output** to true. The compression library to use is specified by **mapred.map.output.compression.codec**.

HADOOP PERFORMANCE TESTING

Output file partitions are made available to the Reducers over HTTP. The number of worker threads used to serve the file partitions is controlled by the task `tracker.http.threads` property—this setting is per tasktracker, not per map task slot. The default of 40 may need increasing for large clusters running large jobs.

Reduce Operations: The Reducer has three phases

1. **Copy:** Each map task's output for the corresponding Reducer is copied as soon as map task completes. The reduce task has a small number of copier threads so that it can fetch map outputs in parallel. The default is 5 threads, but can be changed by setting the `mapred.reduce.parallel.copies` property. The map output is copied to the reduce tasktracker's memory buffer which is controlled by `mapred.job.shuffle.input.buffer.percent` (specifies the proportion of the heap to use for this purpose). When the in-memory buffer reaches a threshold size (controlled by `mapred.job.shuffle.merge.percent`), or reaches a threshold number of map outputs (`mapred.inmem.merge.threshold`), it is merged and spilled to disk. As the copies accumulate on disk, a background thread merges them into larger, sorted files. This saves some time in subsequent merging.
2. **Sort:** This phase should actually be called the *Merge phase* as the sorting is done at the map side. This phase starts when all the maps have been completed and their output has been copied. Map outputs are merged maintaining their sorting order. This is done in rounds. For example if there were 40 map outputs and the merge factor was 10 (the default, controlled by the `io.sort.factor` property, just like in the map's merge) then there would be 4 rounds. In first round 4 files will be merged and in remaining 3 rounds 10 files are merged. The last batch of files is not merged and directly given to the reduce phase.
3. **Reduce:** During reduce phase the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output filesystem, typically HDFS.

Parameters affecting Performance

`dfs.block.size:` File system block size

- **Default**: 67108864 (bytes)
- **Suggestions**:
  - **Small cluster and large data set:** default block size will create a large number of map tasks.

    e.g. Input data size = 160 GB and `dfs.block.size` = 64 MB then the minimum no. of maps= (160*1024)/64 = 2560 maps.

    If `dfs.block.size` = 128 MB minimum no. of maps= (160*1024)/128 = 1280 maps.

    If `dfs.block.size` = 256 MB minimum no. of maps= (160*1024)/256 = 640 maps.

    In a small cluster (6-7 nodes) the map task creation overhead is considerable. So `dfs.block.size` should be large in this case but small

HADOOP PERFORMANCE TESTING

enough to utilize all the cluster resources.

- The block size should be set according to size of the cluster, map task complexity, map task capacity of cluster and average size of input files. If the map contains the computation such that one data block is taking much more time than the other block, then the dfs block size should be lesser.

`mapred.compress.map.output:` Map Output Compression

- **Default:** False
- **Pros:** Faster disk write, saves disk space, less time in data transfer (from Mappers to Reducers).
- **Cons:** Overhead in compression at Mappers and decompression at Reducers.
- **Suggestions:** For large cluster and large jobs this property should be set true. The compression codec can also be set through the property **`mapred.map.output.compression.codec`** (Default is **`org.apache.hadoop.io.compress.DefaultCodec`**).

`mapred.map/reduce.tasks.speculative.execution:` Enable/Disable task (map/reduce) speculative execution

- **Default:** True
- **Pros:** Reduces the job time if the task progress is slow due to memory unavailability, hardware degradation.
- **Cons:** Increases the job time if the task progress is slow due to complex and large calculations. On a busy cluster speculative execution can reduce overall throughput, since redundant tasks are being executed in an attempt to bring down the execution time for a single job.
- **Suggestions:** In large jobs where average task completion time is significant (> 1 hr) due to complex and large calculations and high throughput is required the speculative execution should be set to false.

`mapred.tasktracker.map/reduce.tasks.maximum:` Maximum tasks (map/reduce) for a tasktracker

- **Default:** 2
- **Suggestions:**
  - This value should be set according to the hardware specification of cluster nodes and resource requirements of tasks (map/reduce).

    e.g. a node has 8GB main memory + 8 core CPU + swap space

    Maximum memory required by a task ~ 500MB

    Memory required by **`tasktracker, Datanode`** and other processes ~ (1 + 1 +1) = 3GB

    Maximum tasks that can be run = (8-3) GB/500MB = 10

    Number of map or reduce task (out of the maximum tasks) can be decided on the basis of memory usage and computation complexities of the tasks.
  - The memory available to each task (JVM) is controlled by

HADOOP PERFORMANCE TESTING

`mapred.child.java.opts` property. The default is –Xmx200m (200 MB). Other JVM options can also be provided in this property.

`io.sort.mb`: Buffer size (MBs) for sorting

- **Default:** 100
- **Suggestions:**
  - For large jobs (the jobs in which map output is very large), this value should be increased keeping in mind that it will increase the memory required by each map task. So the increment in this value should be according to the available memory at the node.
  - Greater the value of **`io.sort.mb`**, lesser will be the spills to the disk, saving write to the disk.

io.sort.factor: Stream merge factor

- **Default:** 10
- **Suggestions:**
  - For large jobs (the jobs in which map output is very large and number of maps are also large) which have large number of spills to disk, value of this property should be increased.
  - Increment in **`io.sort.factor`**, benefits in merging at Reducers since the last batch of streams (equal to **`io.sort.factor`**) are sent to the reduce function without merging, thus saving time in merging.

`mapred.job.reuse.jvm.num.tasks`:

Reuse single JVM

- **Default:** 1
- **Suggestions:** The overhead of JVM creation for each task is around 1 second. So for the tasks which live for seconds or a few minutes and have lengthy initialization, this value can be increased to gain performance.

`mapred.reduce.parallel.copies`:

Threads for parallel copy at Reducer

- **Default:** 5
- **Description:** The number of threads used to copy map outputs to the Reducer.
- **Suggestions:** For large jobs (the jobs in which map output is very large), value of this property can be increased keeping in mind that it will increase the total CPU usage.

Other Performance Aspects

Temporary space: Jobs which generate large intermediate data (map output) should have enough temporary space controlled by property **`mapred.local.dir`**. This property specifies list directories where the Map Reduce stores intermediate data for jobs. The data is cleaned-up after the job completes.

By default, replication factor for file storage on HDFS is 3, which means that every file has three replicas. As a thumb rule, at least 25% of the total hard disk should for

HADOOP PERFORMANCE TESTING

intermediate temporary output. So effectively, only ¼ hard disk space is available for business use.

The default value for **`mapred.local.dir`** is ${hadoop.tmp.dir}/mapred/local. So if **`mapred.local.dir`** is not set, **`hadoop.tmp.dir`** must have enough space to hold job's intermediate data. If the node doesn't have enough temporary space the task attempt will fail and starts a new attempt, thus reducing the performance.

JVM tuning: Besides normal java code optimizations, JVM settings for each child task also affect the processing time. On slave node end, the Tasktracker and data node use 1 GM RAM each. Effective use of the remaining RAM as well as choosing the right GC mechanism for each Map or Reduce task is very important for maximum utilization of hardware resources. The default max RAM for child tasks is 200MB which might be insufficient for many production grade jobs. The JVM settings for child tasks are governed by **`mapred.child.java.opts`** property.

Reducer Lazy Initialization: In M/R job Reducers are initialized with Mappers at the job initialization, but the reduce method is called in reduce phase when all the maps had been finished. So in large jobs where Reducer loads data (>100 MB for business logic) in-memory on initialization, the performance can be increased by lazily initializing Reducers i.e. loading data in reduce method controlled by an initialize flag variable which assures that it is loaded only once.

By lazily initializing Reducers which require memory (for business logic) on initialization, number of maps can be increased (controlled by **`mapred.tasktracker.map.tasks.maximum ;`**).

e.g. Total memory per node = 8 GB
Maximum memory required by each map task = 400 MB
If Reducer loads 400 MB of data (metadata for business logic) on initialization
Maximum memory required by each reduce task = 600 MB
No. of reduce tasks to run = 4
Maximum memory required by **`Tasktracker + Datanode`** = 2GB

Before Lazy initialization of Reducers

Maximum memory required by all Reducers throughout the job = 600 * 4 = 2400 MB

HADOOP PERFORMANCE TESTING

Number of map tasks can be run = (8-2-2.4) GB /400 MB = 9

After Lazy initialization of Reducers

Maximum memory required by all Reducers during copy and sort phase = (600-400)*4 = 800 MB

Number of map tasks can be run = (8-2-0.8) GB /400 MB = 13

So by lazily initializing Reducers, 4 more map tasks can be run, thereby increasing the performance.

Case Study 1

Organizations that work with huge amount of data logs offer perfect examples of Hadoop requirements. This case study is for an organization dealing in analyzing and aggregating information collected from various data logs of size more than 160GB per month.

Problem Statement: Processing more than 160 GB of data logs based on some business logic, using around 1.2 GB of metadata (complex hierarchical look up/ master data). Processing involves complex business logic, grouping based on different fields and sorting. It also involves processing metadata from raw metadata files.

Challenges:

1. Metadata is not actually metadata since it is updated while processing based on some field.
2. Processing generates huge intermediate data (~ 15 TB).
3. Large number of records (> 170 million) in a few aggregation groups.
4. Metadata (~1.2 GB) should remain in-memory while processing.
5. Providing large metadata to different parallel jobs and tasks.
6. Solution to be run on small cluster of 7 nodes.
7. Multiple outputs were required from single input files.

Cluster Specifications:

`Datanodes+tasktrackers` = 6 machines

`Namenode+jobtracker` = 1 machine

Each machine has 8GB main memory, 8 core CPU, 5 TB Hard drive

Cloudera Hadoop 0.18.3

Approaches: The problem was divided into 5 Map Reduce jobs- Metadata job, Input processing job for parsing the input files and generating 2 outputs, intermediate output 1 processing job, intermediate

HADOOP PERFORMANCE TESTING

output 2 processing job and an additional downstream aggregation job.

Metadata Processing: The first step was to process metadata from raw metadata files using some business logic. The metadata was in the form of in memory hash maps. Here the *challenge (5)* was how to provide this in memory data to different jobs and tasks in parallel environment since Hadoop doesn't have global (shared) storage among different tasks. The solution arrived at was to serialize all the hash maps and store into the files (~ 390 MB) then add these files to **distributed cache** (A cache provided by Hadoop, which copies all the resource added into it, to the local file system of each node) and de-serialize it in all the tasks.

- **Approach 1:** Due to the *challenge 1*, it was decided that whichever job updates metadata should be processed sequentially i.e. only 1 map and 1 reduce task should run. The main processing job had the business logic as part of Maps.

  Feasibility: This approach worked only for small input data set (< 500 MB), since the memory requirements were high due to the reason that aggregation records (key,value) ran into 170 million+ records in some cases. Overall this approach was unfeasible for the target input data set

- **Approach 2:** As a solution for *challenge 1*, all the processing was moved to the Reducers and the map output was partitioned on the fields on the basis of which, the metadata was updated. Hadoop default hash partitioner was used and the map output key's hash code method was written accordingly. Now all the jobs could run in parallel.

  Feasibilty: This approach could process more data then the previous approach but was still not feasible to process the target input data set.

- **Approach 3:** Two jobs (input processing and intermediate output 1) were merged together (since one job had all the processing on Mapper and other on Reducer) to save time to extra read and write (>320 GB) to HDFS. Now this combined job was generating huge intermediate data ~ 15 TB (*challenge 2*) which cause the job to fail with error

  ```
  Exception in thread "main"
  org.apache.hadoop.fs.FSError:
  java.io.IOException:  No
  space left on device at
  org.apache.hadoop.fs.
  LocalFileSystem$
  LocalFSFileOutputStream.write(
  LocalFileSystem.java:150)
  ```

  The solution to the above error was to increase the space provided to the **mapred.local.dir** directories so that 15 TB data can be stored. In addition to the above solution the property **mapred.compress.map.output** was set to true.

  Now came the *challenge 3*, the grouping on 1 field causes ~170 million records in a few groups and the business logic was such that the

HADOOP PERFORMANCE TESTING

intermediate data generated by processing each group should remain in memory unless the new group comes, causing memory shortage and ultimately the job failure. The solution to this was tweaking the business logic such that the grouping can be done on more than one fields thereby reducing the number of records in a group.

Another optimization to overcome the effect of *challenge 6* was to increase **`dfs.block.size`** to 256 MB since its default value (64MB) was creating ~5000 maps. So in a small cluster of 7 nodes the map creation overhead was significant. After the block size change, 631 maps were being created for input data.

Feasibility: This approach was feasible but the estimated time to process target input data set was more than 200 hrs.

- **Approach 4:** Another solution to the *challenge 4* was to use memcached for metadata, instead of keeping it in memory, so that number of maps can be increased. First the spy memcached client was used but it had some data leakages and didn't work. Finally Danga memcached client was used and it worked well. Throughput using memcached was around 10000 transactions per second.

  Here are some statistics of the job running with memcached

  Stats: Attempt 1

  Feasibility: Time taken = ~ **136 hrs**

  No. of **`memcached`** servers = 1

  HDFS data processing rate ~16 MB per minute

  `Memcached` server memory ~1.2 GB

  No. of Mappers per node = 10

  Total Number of **`Mappers`** = 60

  Total Number of Reducers =12

  HDFS chunk size = 256 MB

  `Io.sort.mb` = 300

  `Io.sort.factor` = 75

  Memory usage per node

  Swap space = 32GB

  Per **`Mapper`**/Reducer memory ~ 0.5GB

  Total memory usage ~ 0.5 *12 + 1 +1 (**`tasktracker`** and **`datanode`**) ~ 8GB

  Stats: Attempt 2

  Small metadata maps –in memory - Big maps –**`memcached`**, Lazy Reducer Initialization

  Feasibility: Time taken ~ **71 hrs**

  No. of **`memcached`** servers = 15 (2*6+3 = 15)

  HDFS data processing rate ~30 MB per minute

  Average loaded data on **`memcache`** servers = 105M on 14 servers, 210M on 1 **`memcached`** server (with double weight)

  No. of **`Mappers`** per node = 8

  Number of **`Mappers`** = 8*6=48

  Number of Reducers =12

  HDFS chunk size = 256 MB

  `Io.sort.mb` = 300

  `Io.sort.factor` = 75

HADOOP PERFORMANCE TESTING

Memory usage per node

Swap space = 32GB

Per **`Mapper`** memory ~ 0.8GB

Per Reducer memory ~ 50 MB (during copy) + 1.5 (during sort and reduce)

Total memory usage ~ 0.8 *8+ 100 MB + 1 +1 (**`tasktracker`** and **`datanode`**) ~ 8.5GB

Conclusion - **Memcached** servers were showing a **thruput** of around 10000 fetches per second (150000 hits per second across 15 nodes). However, this was not sufficient to meet the requirements for faster processing or match up with in memory hash map lookups.

- **Approach 5:** This approach uses the fact that Reducer would start (processing data) after copy and reduce by which time; **Mappers** would have shut down and released the memory (Reducer Lazy Initialization).

  Feasibility: Time Taken ~**42 hrs**

  HDFS data processing rate ~8 GB per hour

  Maps finished = 19 hrs 37 mins

  No. of **`Mappers`** per node = 5

  No. of **`Mappers`**= 5*6 = 30

  No. of **`Reducers`**= 2*6 = 12

  HDFS chunk size = 256 MB

  `Io.sort.mb` = 300

  `Io.sort.factor` = 75

  Memory usage per node

  Per node: RAM- ~8 GB, SWAP- 4-6 GB

  Per **`Mapper`** memory ~ 1.1-1.8GB

  Per **`Reducer`** memory ~ 50 MB (during copy) + 1.1-2 GB (during sort and reduce)

  Total memory usage in **`Mapper`** processing ~ 1.5 GB *5 + 50 MB *2 + 1 +1 (**`tasktracker`** and **`datanode`**) ~ 9.6GB

- **Approach 6:** Approach 5 was run again with the following configurations

  No. of **`Mappers`** per node = 4

  No. of **`Reducers`** per node = 5

  No. of **`Mappers`** = 4*6 = 24

  No. of **`Reducers`** = 5*6 = 30

  HDFS chunk size = 256 MB

  `Io.sort.mb` = 600

  `Io.sort.factor` = 120

  Memory usage per node

  Per node: RAM- ~8 GB, SWAP- 4-6 GB

  Per **`Mapper`** memory ~ 1.5-2.1GB

  Per **`Reducer`** memory ~ 50 MB (during copy) + 1.3-2 GB (during sort and reduce)

  Total memory usage in **`Mapper`** processing ~ 1.8 GB *4 + 50 MB *5 + 1 +1 (**`tasktracker`** and **`datanode`**) ~ 9.5GB

  Feasibility: Time taken ~ **39hrs**

  Approach 6 gave maximum performance and took minimum time in the given cluster specifications. Only 4 Map jobs per node could be run as use of any other additional Map was causing excessive swap memory which was a major bottleneck for faster processing.

HADOOP PERFORMANCE TESTING

- **Approach 7:** Approach 6 was run again but with Tokyo cabinet (a Berkley DB like file storage system) for metadata handling to overcome excessive RAM usage to store metadata with the following configurations

  No. of Mappers per node = 7

  No. of Reducers per node = 7

  No. of Mappers= 7*6 = 42

  No. of Reducers= 7*6 = 42

  HDFS chunk size = 256 MB

  Io.sort.mb = 400

  Io.sort.factor = 120

  Memory usage per node

  Per node: RAM- ~8 GB,

  Per Mapper memory ~ 800MB

  Per Reducer memory ~ 70 MB (during copy) + 700MB (during sort and reduce)

  Total memory usage in Mapper processing ~ 800 MB *7 + 70 MB *7 + 1 +1 (**tasktracker** and **datanode**) ~ 8GB

  Feasibility: Time taken ~ **35hrs**

Conclusion: Approach 7 gave maximum performance and took minimum time in the given cluster specifications.

Conclusion

Performance of Hadoop Map-Reduce job can be increased without increasing the hardware cost, by just tuning some parameters according to the cluster specifications, input data size and processing complexities.

References

1. *http://hadoop.apache.org/hdfs/ http://hadoop.apache.org/common/docs/current/hdfs_design.html*
2. *http://hadoop.apache.org/common/docs/current/mapred_tutorial.html*
3. *O' Reilly, Hadoop- The Definitive Guide by Tom White*

HADOOP PERFORMANCE TESTING

About Impetus

Impetus is a pure play product engineering company providing outsourced software product design, R&D and related services to leading global software and technology enabled companies. Headquartered in San Jose (CA), and with multiple development centers across India, Impetus has been involved in hundreds of product launches over its 15 year association with clients ranging from enterprise class companies to innovative technology start-ups.

Impetus builds cutting-edge products utilizing the latest and emerging technologies for a diverse client base across domains including Digital Media, Telecommunications, Healthcare, Mobile Applications, Web2.0, IPTV, and Internet Advertising. The company creates leading edge products that underscore future trends while also offering premium design consulting and product architecture services.

Impetus brings forth a perfect synergy of expert talent and competent engineering skills, coupled with technology frameworks and product development maturity to help you build sustainable, scalable and efficient next-generation products, and bring them faster to the market.

Corporate Headquarters

UNITED STATES

Impetus Technologies, Inc.

5300 Stevens Creek Boulevard, Suite 450

San Jose, CA 95129, USA.

Phone: 408.252.7111

Regional Development Centers

INDIA

- New Delhi
- Indore
- Hyderabad

To know more

Visit: http://www.impetus.com

Email: inquiry@impetus.com