Spark on Ampere® Processors

Reference Architecture

June 22, 2023

Document Issue 1.00

AMPERE.

Contents

AMPERE.

Contents (continued)

Figures

Tables

1. Introduction

The influence of Arm technology has expanded beyond mobile and even desktop computing. Arm processors are now used in both on-premises and cloud servers for a variety of workloads. These processors offer improved performance per rack, reduced power consumption, and lower costs, leading to capital expenditure (CapEx) and operational expenditure (OpEx) optimization. Arm servers are widely deployed in datacenters as well as in cloud and edge computing environments. This paper contrasts Big Data Spark performance on Ampere® Altra® Max and Intel Ice Lake based servers.

2. Scope and Audience

The purpose of this document is to provide guidance on setting up, tuning, and evaluating the performance of Apache Spark on a test bed featuring Ampere Altra Max processors. This document covers the architectural concepts of Arm and Apache Spark, then provides steps on installing and tuning Spark on a multi-node cluster. However, it is important to note that the values and parameters provided are general guidelines and should not be considered as final and optimized values.

This document is intended for sales engineers, IT and cloud architects, IT and cloud managers, and customers seeking to leverage the performance and power efficiency benefits of Ampere Arm servers across their IT infrastructure. It aims to provide valuable insights and technical guidance to professionals who are interested in implementing and optimizing Arm-based Spark solutions.

3. Ampere Altra Max Processors

Ampere processors offer a comprehensive System-on-Chip (SoC) solution specifically designed for Cloud Native server applications. These processors are equipped with a wide range of powerful features to meet the demands of modern enterprise infrastructure and to maximize the performance per given rack of compute infrastructure. With support for up to 128 high-performance Arm 64-bit cores per socket, they deliver exceptional processing power and linear scalability. The processors also feature eight channels of DDR4 memory support, ensuring efficient data processing and storage capabilities. Furthermore, the inclusion of 128 channels of PCIe Gen4 interfaces enables fast and reliable data transfer between components and peripherals.

The Ampere Altra family excels in handling intensive tasks such as data analytics, artificial intelligence (AI), database storage, telecommunications, edge computing, and web hosting. Thus, datacenter operators can enhance the efficiency of their infrastructure and exceed the performance demands of today's cloud-based applications and services.

The processors are specifically engineered to deliver exceptional energy efficiency, resulting in industry-leading performance per watt (Perf/Watt) at the individual processor level. This means that each server equipped with Ampere Altra processors delivers high performance while consuming less power than a comparable legacy x86 server.

Optimizing performance per rack not only reduces operating costs for datacenters but also contributes to a significant reduction in carbon footprint. By adopting Ampere Altra Max processors, organizations can align their datacenter operations with sustainability goals and reduce their environmental impact. The combination of performance and energy efficiency offered by Ampere Altra processors provides a compelling solution for datacenters, leading to cost savings, improved sustainability, and a greener approach to computing.

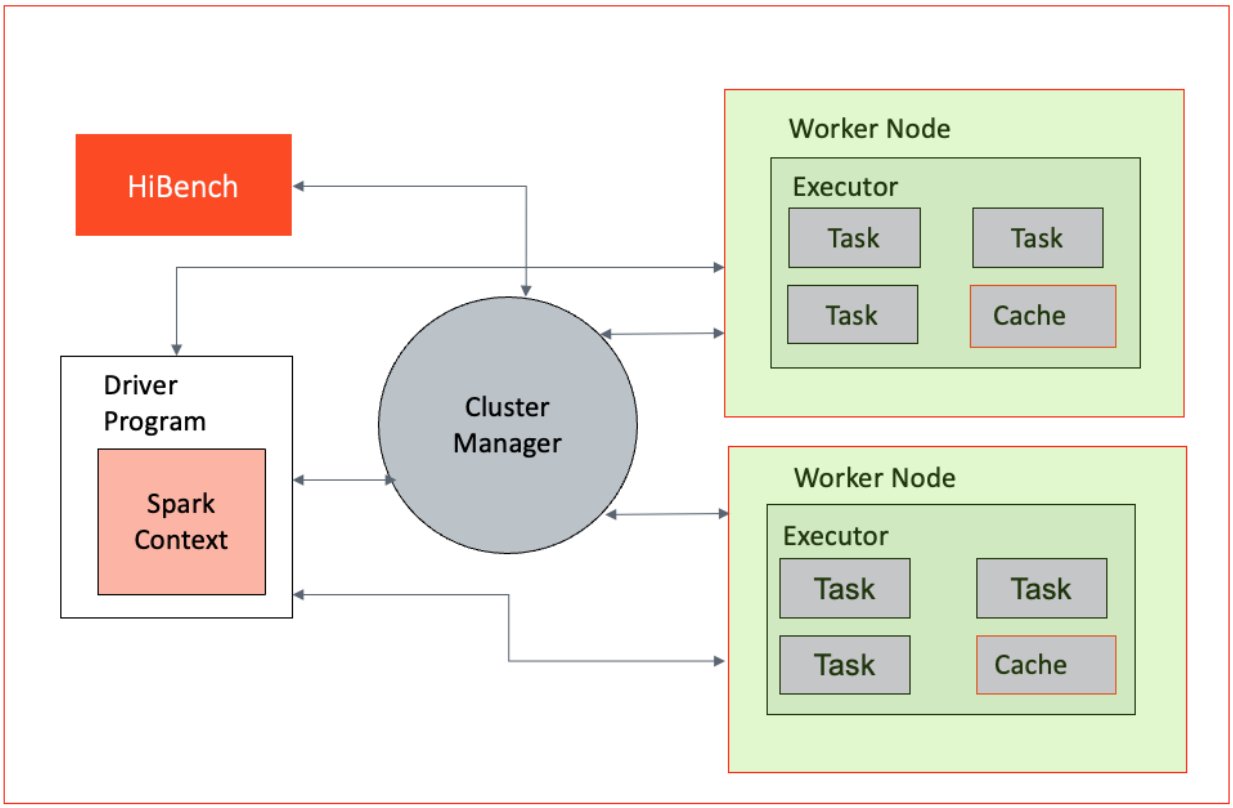
4. Apache Spark

Apache Spark is a versatile data processing framework used in data engineering, data science, and machine learning tasks. It is capable of operating on both a single-node machine and clusters, making it suitable for handling large-scale data processing tasks. By leveraging its distributed computing capabilities, Spark can efficiently distribute data processing tasks across multiple nodes, either independently or in conjunction with other distributed computing tools.

Spark utilizes in-memory caching, which allows for quick access to data and optimized query execution, enabling fast analytic queries on datasets of any size. It provides APIs in popular programming languages such as Java, Scala, and Python, making it accessible to a wide range of developers. Spark supports various workloads, including real-time analytics, batch processing, interactive queries, and machine learning, making it a comprehensive solution for diverse data processing requirements.

4.1 Spark Architecture and Components

Figure 1: Spark Architecture



4.1.1 Spark Driver

The Spark Driver serves as the central controller of the Spark execution engine and is responsible for managing the overall state of the Spark cluster. It interacts with the cluster manager to acquire the necessary resources such as virtual CPUs (vCPUs) and memory. Once the resources are obtained, the driver launches the executors, which are responsible for executing the actual tasks of the Spark application. Additionally, the Spark driver plays a crucial role in maintaining the state of the application running on the cluster. It keeps track of various important information, such as the execution plan, task scheduling, and the data transformations and actions to be performed. The driver coordinates the execution of tasks across the available executors, ensuring efficient data processing and computation.

The Spark driver acts as a control unit, orchestrating the execution of Spark applications on the cluster and maintaining the necessary states and communication with the cluster manager and executors.

4.1.2 Spark Executors

Spark Executors are responsible for executing the tasks assigned to them by the Spark driver. Once the driver distributes the tasks across the available executors, each executor independently processes its assigned tasks. The executors run these tasks in parallel, leveraging the resources allocated to them, such as CPU and memory. They perform the necessary computations, transformations, and actions specified in the Spark application code. This includes operations like data transformations, filtering, aggregations, and machine learning algorithms, depending on the nature of the tasks. During the execution of the tasks, the executors communicate with the driver, providing updates on their progress and reporting the results of each task.

4.1.3 Cluster Manager

The Cluster Manager is responsible for maintaining the cluster of machines on which the Spark applications run. It handles resource allocation, scheduling, and management of the Spark driver and executors, ensuring efficient execution of Spark applications on the available cluster resources.

When a Spark application is submitted, the driver communicates with the Custer Manager to request the necessary resources, such as CPU, memory, and storage, to run the application. It ensures that the resources are distributed effectively to meet the requirements of the Spark application. This includes tasks such as assigning containers or worker nodes to execute the Spark executors and ensuring that the required dependencies and configurations are in place.

There are various cluster managers that can be used with Spark, including the Standalone Cluster Manager, Apache Hadoop YARN (Yet Another Resource Negotiator), and Apache Mesos. By working with different cluster managers, Spark provides flexibility in deploying and running applications in various environments. This allows users to choose the cluster manager that best fits their infrastructure and requirements.

4.1.4 Spark Core

The Spark Core serves as the underlying execution engine for the Spark platform, forming the basis for all other Spark functionality. It offers powerful capabilities such as in-memory computing and the ability to reference datasets stored on external storage systems. One of the key components of the Spark Core is the Resilient Distributed Dataset (RDD), which serves as the primary programming abstraction in Spark. RDDs enable fault-tolerant and distributed data processing across a cluster.

The Spark Core provides a wide range of APIs for creating, manipulating, and transforming RDDs. These APIs are available in multiple programming languages including Java, Python, Scala, and R. This flexibility allows developers to work with the Spark Core using their preferred language and leverages the rich ecosystem of libraries and tools available in those languages.

4.1.5 Spark RDD

Spark uses a concept called Resilient Distributed Dataset (RDD), an abstraction that represents an immutable collection of objects that can be split across a cluster. RDDs can be created from various data sources, including SQL databases and NoSQL stores. The Spark Core, which is built upon the RDD model, provides essential functionalities such as mapping and reducing operations. It also offers built-in support for joining datasets, filtering, sampling, and aggregation, making it a powerful tool for data processing. When executing tasks, Spark splits them into smaller subtasks and distributes them across multiple executor processes running on the cluster. This enables the parallel execution of tasks across the available computational resources, resulting in improved performance and scalability.

4.1.6 Spark Scheduler

The Spark Scheduler is a vital component responsible for task scheduling and execution. It uses Directed Acyclic Graph (DAG) and employs a task-oriented approach for scheduling tasks. The scheduler analyzes the dependencies between different stages and tasks of a Spark application, represented by the DAG. It determines the optimal order in which tasks should be executed to achieve efficient computation and minimize data movement across the cluster.

By understanding the dependencies and requirements of each task, the scheduler assigns resources, such as CPU and memory, to the tasks. It considers factors like data locality, where possible, to reduce network overhead and improve performance. The task-oriented approach of the Spark Scheduler allows it to break down the application into smaller, manageable tasks and distribute them across the available resources. This enables parallel execution and efficient utilization of the cluster's computing power.

4.1.7 Spark SQL

Spark SQL is a widely used component of Apache Spark that facilitates the creation of applications for processing structured data. It adopts a data frame approach and allows efficient and flexible data manipulation. One of the key features of Spark SQL is its ability to interface with various data storage systems. It provides built-in support for reading and writing data from and to different datastores, including JSON, HDFS, JDBC, and Parquet. This makes it easy to work with structured data residing in different formats and storage systems.

Additionally, Spark SQL extends its connectivity beyond the built-in datastores. It offers connectors that enable integration with other popular datastores such as MongoDB, Cassandra, and HBase. These connectors allow users to seamlessly interact with and process data stored in these systems using Spark SQL's powerful querying and processing capabilities.

4.1.8 Spark MLlib

In addition to its core functionalities, Apache Spark includes bundled libraries for machine learning and graph analysis techniques. One such library is MLlib, which provides a comprehensive framework for developing machine learning pipelines.

MLlib simplifies the implementation of machine learning workflows by offering a wide range of tools and algorithms. Apache Spark's bundled library MLlib provides a comprehensive framework for machine learning pipelines. It simplifies the implementation of feature extraction and transformations on structured datasets and offers a wide range of machine learning algorithms. MLlib empowers developers to build scalable and efficient machine learning workflows, enabling them to leverage the power of Spark for advanced analytics and data-driven applications.

4.1.9 Distributed Storage

Spark does not provide its own distributed file system. However, it can effectively utilize existing distributed file systems to store and access large datasets across multiple servers.

One commonly used distributed file system with Spark is the Hadoop Distributed File System (HDFS). HDFS allows for the distribution of files across a cluster of machines, organizing data into consistent sets of blocks stored on each node. Spark can leverage HDFS to efficiently read and write data during its processing tasks. When Spark processes data, it typically copies the required data from the distributed file system into its memory. By doing so, Spark reduces the need for frequent interactions with the underlying file system, resulting in faster processing compared to traditional Hadoop MapReduce jobs. As the dataset size increases, additional servers with local disks can be added to the distributed file system, allowing for horizontal scalability and improved performance.

By reducing the need for frequent disk interactions and utilizing memory-based operations, Spark provides faster processing times compared to Hadoop MapReduce.

4.1.10 Spark Jobs, Stages, and Tasks

In a Spark application, the execution flow is organized into a hierarchical structure consisting of jobs, stages, and tasks.

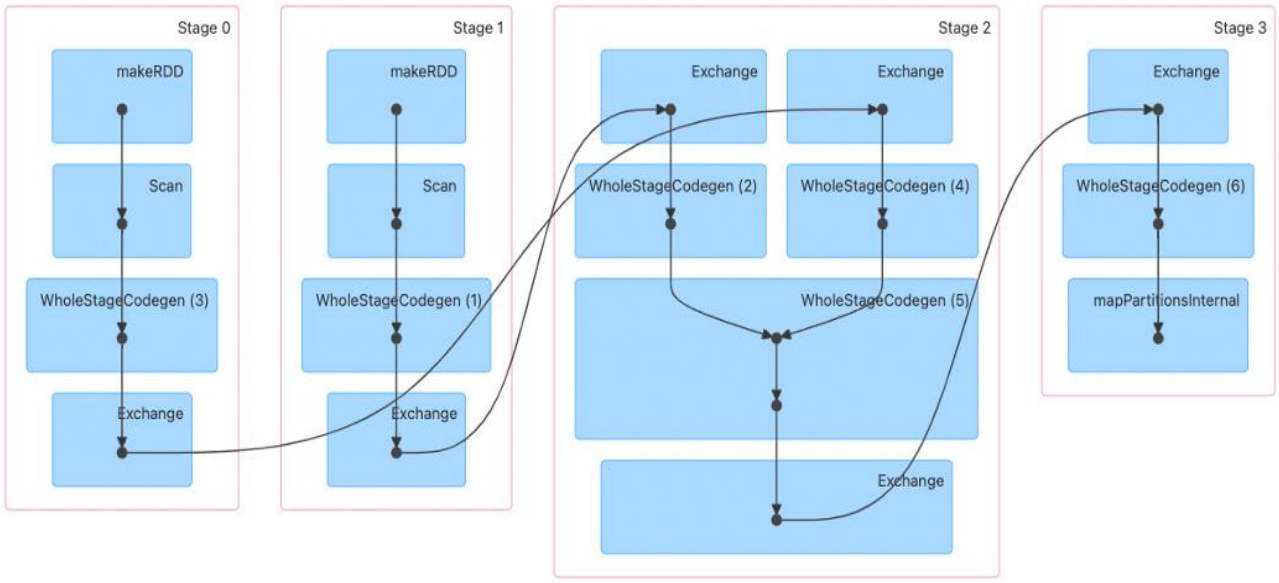
A job represents a high-level unit of work within a Spark application. It can be seen as a complete computation that needs to be performed, involving multiple stages and transformations on the input data.

A stage is a logical division of tasks that share the same shuffle dependencies, meaning they need to exchange data with each other during execution. Stages are created when there is a shuffle operation, such as a groupBy or a join, that requires data to be redistributed across the cluster.

Within each stage, there are multiple tasks. A task represents the smallest unit of work in Spark, representing a single operation that can be executed on a partition of the data. Tasks are typically executed in parallel across multiple nodes in the cluster, with each node responsible for processing a subset of the data.

Spark intelligently partitions the data and schedules tasks across the cluster to maximize parallelism and optimize performance. It automatically determines the optimal number of tasks and assigns them to available resources, considering factors such as data locality to minimize data shuffling between nodes. Spark handles the management and coordination of tasks within each stage, ensuring that they are executed efficiently and leveraging the parallel processing capabilities of the cluster.

Figure 2: Spark Jobs, Stages, and Tasks



Shuffle boundaries introduce a barrier where stages/tasks must wait for the previous stage to finish before they fetch map outputs. In *Figure 2*, Stage 0 and Stage 1 are executed in parallel, while Stage 2 and Stage 3 are executed sequentially. Hence, Stage 2 has to wait until both Stage 0 and Stage 1 are complete. This execution plan is evaluated by Spark.

5. Spark Test Bed

Two Spark clusters were set up for performance benchmarking. Both the clusters used HDFS as file storage. The first cluster was equipped with HPE RL300 servers powered by Ampere Altra Max processors, while the second cluster used Dell PowerEdge R650 servers powered by Intel Ice Lake processors.

6. Equipment Under Test

Table 1: Equipment Under Test

SPECIFICATIONS	CLUSTER 1	CLUSTER 2
Architecture	Ampere Altra Max, Arm64	Intel Ice Lake, x86-64
Make and Model	HPE RL300	Dell PowerEdge R650
Cluster Nodes	3	3
CPU	Ampere M128-30	Intel Xeon SP 6342
Sockets/Node	1	2
Cores/Socket	128	24
Threads/Socket	128	48
CPU Speed	3.0 GHz	2.8 GHz/3.50 GHz
Memory	512 GB DDR4	512 GB DDR4
Network Card	1x Mellanox CX-6 Dx	1x Mellanox CX-6 Dx
Storage	1x 960 GB Samsung for OS and 4x Micron 7450 Gen 4 NVME for data	1x 960 GB Samsung for OS and 4x ScaleFlux CSD 3010 Gen 4 NVME for data
Kernel Version	4.18.0-348.7.1	5.15.0-60-generic
Operating System	CentOS 8.5	Ubuntu 22.04 LTS
YARN Version	3.3.4	3.3.4
Spark Version	3.3.1	3.3.1

7. Spark Installation and Cluster Setup

We set up the cluster with the HDFS. Hence, we installed Spark as a Hadoop user and configured the disks for HDFS.

7.1 BIOS Setup

Before proceeding with the installation, see *Section A.1, BIOS Changes* for BIOS settings on Ampere servers. Boot the servers to BIOS and make the necessary changes if needed.

7.2 OS Install

Most modern open-source or enterprise-supported Linux distributions support AArch64. You can choose any OS supported on Arm. Map your CD/DVD in the KVM console and install the supported OS on the servers.

7.3 Networking Setup

Set up a public network on one of the available interfaces. This can be used to log into any of the servers and when client communication is needed. Set up a private network for communication between the cluster nodes.

7.4 Storage Setup

Choose a drive of your choice for the OS install, clear any old partitions, reformat, and choose the disk to install the OS. Samsung 960 GB drives were chosen for the OS install in this setup.

7.5 Create Hadoop User

Create a user named “hadoop” as part of the OS Install. This user was used for both Hadoop and Spark daemons on the test bed.

7.6 Post-Install Steps

Perform the following post-install steps on all the nodes on the OS after the install.

1. yum or apt update on the nodes.
2. Install packages like dstat, net-tools, lm-sensors, linux-tools-generic, Python, and sysstat for your monitoring needs.
3. Set up ssh trust between all the nodes.
4. Update the **/etc/sudoers** file for nopasswd for the hadoop user.
5. Update **/etc/security/limits.conf** according to *Section A.3, /etc/security/limits.conf*.
6. Update **/etc/sysctl.conf** according to *Section A.2, /etc/sysctl.conf*.
7. Update hugepages according to *Section A.4, Miscellaneous Kernel Changes* and hibench.conf according to *Section A.11, hibench.conf*.
8. If necessary, make changes to **/etc/rc.d** to keep the above changes permanent after every reboot.
9. Set up NVMe disks as an xfs file system for HDFS.
    a. Create a single partition on each of the NVMe disks with fdisk or parted.
    b. Create a file system on each of the created partitions using mkfs.xfs -f /dev/nvme[0-n]n1p1.
    c. Create directories for mounting using mkdir -p /root/nvme[0-n]1p1.
    d. Update **/etc/fstab** with entries and mount the file system. The UUID of each partition in fstab can be extracted from the `blkid` command.
    e. Change ownership of these directories to the “hadoop” user created earlier.

7.7 Spark Install

Download Hadoop 3.3.4 from the Apache website, Spark 3.3.1 from Apache Spark, and JDK11 and JDK17 for Arm/AArch64. We use JDK11 for Hadoop and JDK17 for Spark installs. Extract the tarballs under the hadoop user home directory.

Update the Spark and Hadoop configuration files in **~/hadoop/spark/conf** and **~/hadoop/etc/hadoop/** and the environment parameters in .bashrc according to *Section A5, .bashrc File* through *Section A10, spark-defaults.conf*. Depending on the hardware specifications on cores, memory, and disk capacities, these may have to be altered. Update the workers file to include the set of data nodes.

Run the following commands:

```
hdfs namdenode -format
scp -r ~/hadoop <datanodes>:~/hadoop
~/hadoop/sbin/start-all.sh
~/spark/sbin/start-all.sh
```

These commands should start the Spark master, worker, and other hadoop daemons.

8. Performance Tuning

Several components interact across multiple systems within the Spark framework. The performance of Spark is influenced by several factors, such as BIOS settings, operating system parameters, network and disk subsystems, and the configuration of the stack. To optimize the configuration settings, prior experience with Hadoop and Spark is helpful. It is important to note that performance tuning is an iterative process, and the parameters provided in *Appendix A, Configuration* are merely reference values obtained through a few iterations.

8.1 BIOS

When it comes to tuning systems for running Spark, a good starting point is the BIOS. It is important to note that the parameter names and options may differ slightly depending on the system manufacturer. In *Section A.1, BIOS Changes*, you can find a few BIOS tuning parameters that are helpful in optimizing the performance.

8.2 Linux

Occasionally, there can be conflicts between the subcomponents of a Linux system, such as the network and disk, which can impact overall performance. The objective is to optimize the system to achieve optimal disk and network throughput and identify and resolve any bottlenecks that may arise.

8.3 Network

To evaluate the network infrastructure, the iperf utility can be utilized to conduct stress tests. Adjusting the TX/RX ring buffers and the number of interrupt queues to align with the cores on the NUMA node where the Network Interface Card (NIC) is located can help optimize performance. However, if the BIOS setting is already configured as chipset-ANC in a monolithic manner, these modifications may not be necessary.

8.4 Disks

When working in a Hadoop or Spark environment, it is advisable to pay attention to certain aspects such as aligned partitions, queue depth, number of requests, and file system mount options. Aligned partitions can be created using the parted utility. Queue depth and nr_requests can be configured through the **/sys/block/<device>/device|queue** directory. For HDFS, one important file system option (fstab) is noatime. To test the disk subsystem, the fio tool can be employed.

8.5 Spark Configuration Parameters

There are several tunable parameters on Spark. Only a few of them are addressed here. Tune your parameters by observing the resource usage from **Error! Hyperlink reference not valid.**. For details, refer to https://spark.apache.org/docs/latest/web-ui.html.

8.5.1 Using Data Frames over RDD

It is preferred to use dataset or data frames over RDD that include several optimizations to improve the performance of spark workloads. Spark data frames can handle the data better by storing and managing it efficiently as it maintains the structure of the data and column types.

8.5.2 Using Serialized Data Formats

In Spark jobs, a common scenario involves writing data to a file, which is then read by another job and written to another file for subsequent Spark processing. To optimize this data flow, it is recommended to write the intermediate data into a serialized file format such as Parquet. Using Parquet as the intermediate file format can yield improved performance compared to formats like CSV or JSON. Parquet is a columnar file format designed to accelerate query processing. It organizes data in a columnar manner, allowing for more efficient compression and encoding techniques. This columnar storage format enables faster data access and processing, particularly for operations that involve selecting specific columns or performing aggregations.

By leveraging Parquet as the intermediate file format, Spark jobs can benefit from faster transformation operations. The columnar storage and optimized encoding techniques offered by Parquet, as well as its compatibility to processing frameworks like Hadoop, contribute to improved query performance and reduced data processing time.

8.5.3 Reducing Shuffle Operations

Shuffling is a crucial operation in Spark that involves redistributing data across different executors and nodes within a cluster. Operations such as joins, groupBy, and reduceBy trigger shuffling on RDDs and data frames. Shuffling entails disk I/O, data serialization, and network I/O, and while it cannot be entirely eliminated, minimizing shuffling can significantly improve performance.

The key parameter to consider for tuning shuffling in Spark is "spark.sql.shuffle.partitions," which can be adjusted in the spark-defaults.conf file to suit your specific workload.

The optimal configuration for shuffling depends on factors such as the dataset size, the number of cores available, and the available memory. Increasing the number of partitions results in a higher number of partitioned files with a lower number of records in each partition. Conversely, too few partitions with a large amount of data in each partition can lead to out-of-memory errors.

By fine-tuning the spark.sql.shuffle.partitions parameter and finding the right balance between the number of partitions and the size of each partition, you can optimize shuffling and improve the performance of your Spark jobs. This iterative process may involve running multiple experiments with different values to achieve the best possible configuration.

8.5.4 Spark Executor Cores

The number of cores allocated to each Spark Executor is an important consideration for optimal performance. In general, allocating around five cores per executor tends to be a reasonable allocation when using the HDFS.

When running Spark alongside Hadoop daemons, it is vital to reserve a portion of the available cores for these daemons. This ensures that the Hadoop infrastructure functions smoothly alongside Spark. The remaining cores can then be distributed among the Spark executors for executing data processing tasks.

By striking a balance between allocating cores to Hadoop daemons and Spark executors, you can ensure that both systems coexist effectively, enabling efficient and parallel processing of data. It is important to adjust the allocation based on the specific requirements of your cluster and workload to achieve optimal performance.

8.5.5 Spark Executor Instances

The number of Spark executor instances represent the total count of executor instances that can be spawned across all worker nodes for data processing. To calculate the total number of cores consumed by a Spark application, you can multiply the number of executors by the cores allocated per executor.

The Spark UI provides information on the actual utilization of cores during task execution, indicating the extent to which the available cores are being utilized. It is recommended to maximize this utilization based on the availability of system resources.

By efficiently utilizing the available cores, you can optimize the processing capacity of your Spark application and improve overall performance. It is important to consider the resources available in your cluster and adjust the number of executor instances and cores allocated per executor accordingly. This allows for effective utilization of resources and maximizes the computational power of your Spark application.

8.5.6 Executor and Driver Memory

The memory configuration for Spark's driver and executors plays a critical role in determining the available memory for these components. It is important to tune these values based on the memory requirements of your Spark application and the memory availability within your YARN scheduler and NodeManager resource allocation parameters.

The executor memory refers to the memory allocated for each executor, while the driver memory represents the memory allocated for the Spark driver. These values should be adjusted carefully to ensure optimal performance and avoid memory-related issues.

When tuning the memory configuration, it is essential to consider the overall memory availability in your environment and consider any memory constraints imposed by the YARN scheduler and NodeManager settings. By aligning the memory allocation with the available resources, you can optimize the memory utilization and prevent potential out-of-memory errors or performance degradation (swapping or disk spills).

It is recommended to monitor the memory usage with Spark UI and adjust the configuration iteratively to achieve the best performance for your Spark workload.

9. Benchmark Tools

We used both Intel HiBench and TPC-DS benchmarking tools to measure the performance of the clusters.

9.1 HiBench

To assess and compare the performance of the clusters in handling Big Data workloads, we utilized the widely used benchmarking suite called HiBench. HiBench is specifically designed for evaluating the performance of Big Data frameworks like Apache Hadoop and Apache Spark. It includes a comprehensive set of workload-specific benchmarks that simulate real-world Big Data processing scenarios. For more detailed information about HiBench, visit https://github.com/Intel-bigdata/HiBench. By executing HiBench on both clusters, you can analyze and compare their capabilities in terms of data processing speed, scalability, and resource utilization. The benchmark results provide valuable insights into the performance of each cluster under various Big Data workloads.

1. Update the hibench.conf file, like the scale, profile, parallelism parameters and the list of master and slave nodes.
2. Run **~HiBench/bin/workloads/micro/terasort/prepare/prepare.sh**.
3. Run **~HiBench/bin/workloads/micro/terasort/spark/run.sh**.

After executing these steps, a file named hibench.report is generated within the report directory. Additionally, a file named bench.log contains comprehensive information regarding the execution.

To measure the clusters' performance, we conducted the TeraSort benchmark on both clusters and recorded the throughput generated by each cluster.

9.2 TPC-DS

TPC-DS is an industry-standard decision support benchmark that models various aspects of a decision support system, encompassing data maintenance and queries. Its purpose is to assist organizations in making informed decisions regarding their technology choices for decision support systems. TPC benchmarks aim to provide objective performance data that is relevant to industry users. For more detailed information, visit https://www.tpc.org/tpcds/.

We ran TPC-DS benchmark on both the clusters, where the total time required to complete all 99 SQL queries was measured to assess performance (lower time denoting better performance).

Furthermore, TeraSort and HiBench benchmarks were conducted using a 3 TB dataset. To gain insights into system performance, we monitored various parameters such as total power consumption, CPU power, CPU utilization, as well as disk and network utilization. This monitoring was accomplished utilizing tools such as Grafana, IPMI, and Redfish.

10. Performance Tests on Three Node Clusters

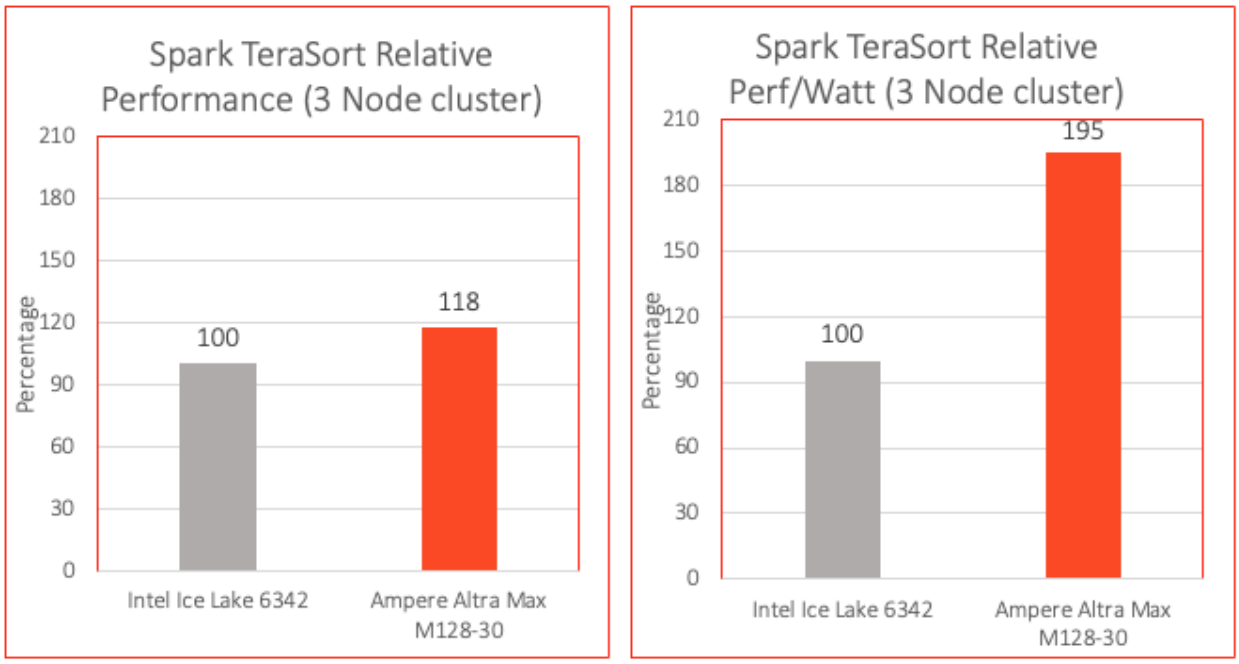
10.1 TeraSort Performance

To assess the performance comparison between the two clusters, the TeraSort benchmark was executed on both clusters using the HiBench benchmarking tool. The benchmarking results were analyzed to evaluate the performance and scalability of each cluster under the given workload.

During our testing, we observed that the TeraSort throughput on the Altra Max systems surpassed that of the Intel Ice Lake systems by approximately 18%, as shown in *Figure 3*.

In addition to the performance assessment, we also measured the energy efficiency of each cluster by calculating the Perf/Watt ratio. This metric, which divides the TeraSort cluster throughput (in MBPS) by the total power consumed by the cluster (in watts) during the benchmarking interval, is crucial for evaluating energy efficiency in datacenters. Notably, the Altra Max system revealed a superior Perf/Watt ratio, reaching around 195% while executing the TeraSort benchmark, as shown in *Figure 3*.

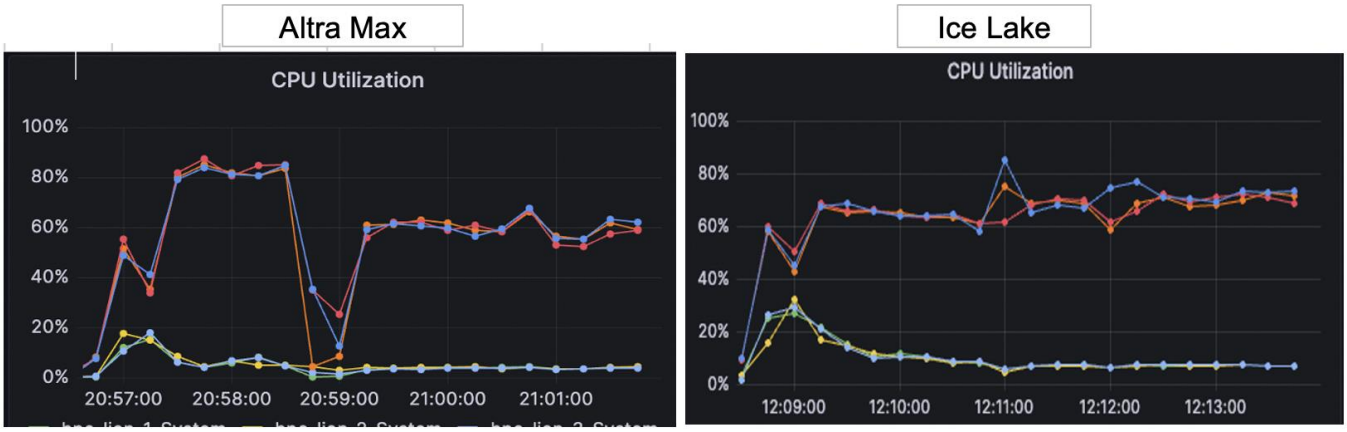
Figure 3: TeraSort Performance



10.1.1 CPU Utilization

The Grafana graphs in *Figure 4* indicate that both systems were operating at their maximum utilization levels while running TeraSort benchmark.

Figure 4: CPU Utilization



10.1.2 Disk and Network Utilization

Figure 5 and *Figure 6* indicate that both the HPE RL300 and Dell PowerEdge R650 servers achieved approximately 10 GB/s disk throughput and 10Gb/s network throughput.

Figure 5: Disk Utilization

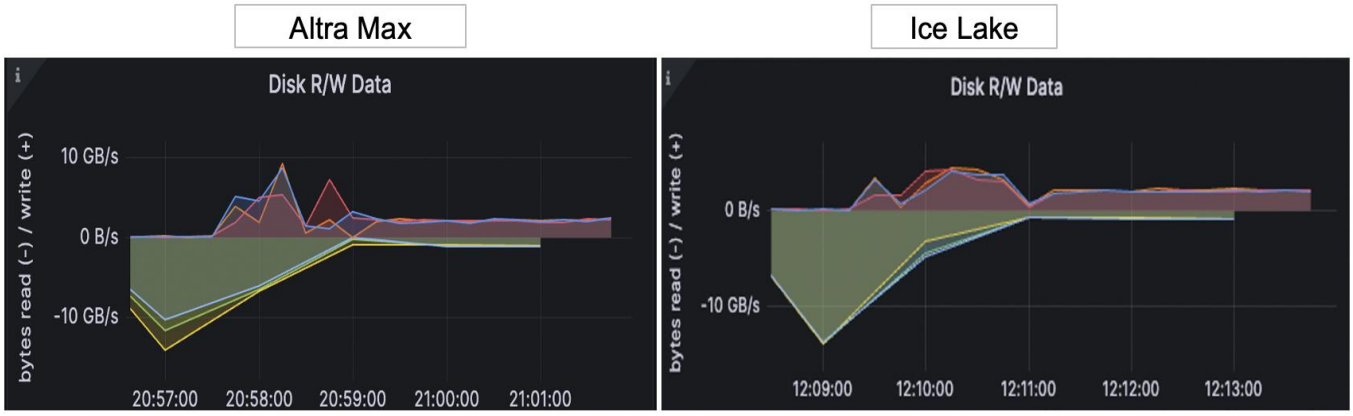
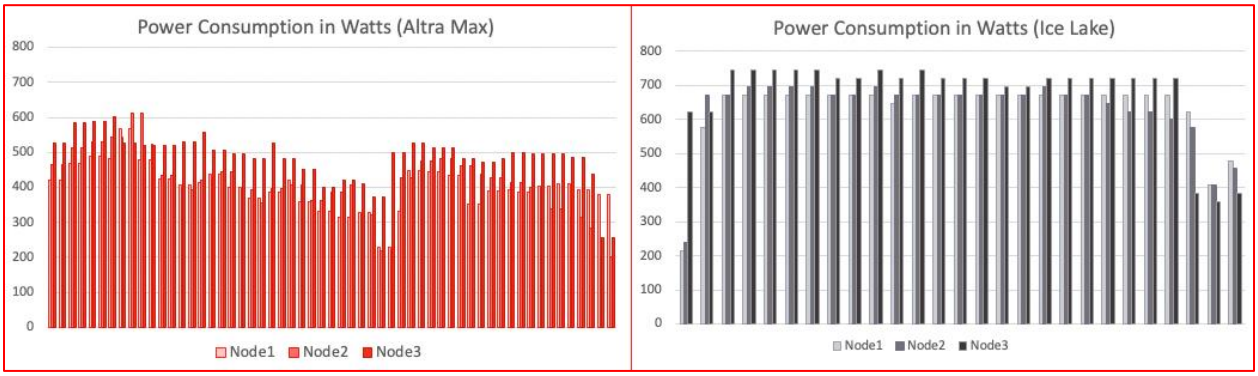


Figure 6: Network Utilization

10.1.3 Power Consumption

Figure 7 provides a visual representation of the power consumption of the clusters during the execution of the TeraSort benchmark. HPE RL300 servers, which are equipped with Ampere Altra Max processors, exhibited notably lower power consumption in comparison to the Dell PowerEdge servers with Intel Ice Lake processors.

Figure 7: Power Consumption



10.2 TPC-DS Performance

The TPC-DS benchmarking tool was used to execute the TPC-DS workload on the clusters. The performance evaluation was based on the total time required to execute all 99 SQL queries on the clusters. We observed that the TPC-DS queries completed approximately 21% faster on the Altra Max systems compared to the Intel Ice Lake systems, with both clusters operating on a dataset size of 3 TB, as shown in *Figure 8*.

To further assess the performance, we considered both the time taken and power consumption during the benchmarking process. By calculating the Perf/Watt ratio, which compares the TPC-DS workload throughput to the power consumed by the clusters, we gained insights into energy efficiency. *Figure 8* shows the Altra Max system demonstrated a superior Perf/Watt ratio of around 177% while executing TPC-DS workloads, outperforming the Intel Ice Lake systems.

Figure 8: TPC-DS Performance

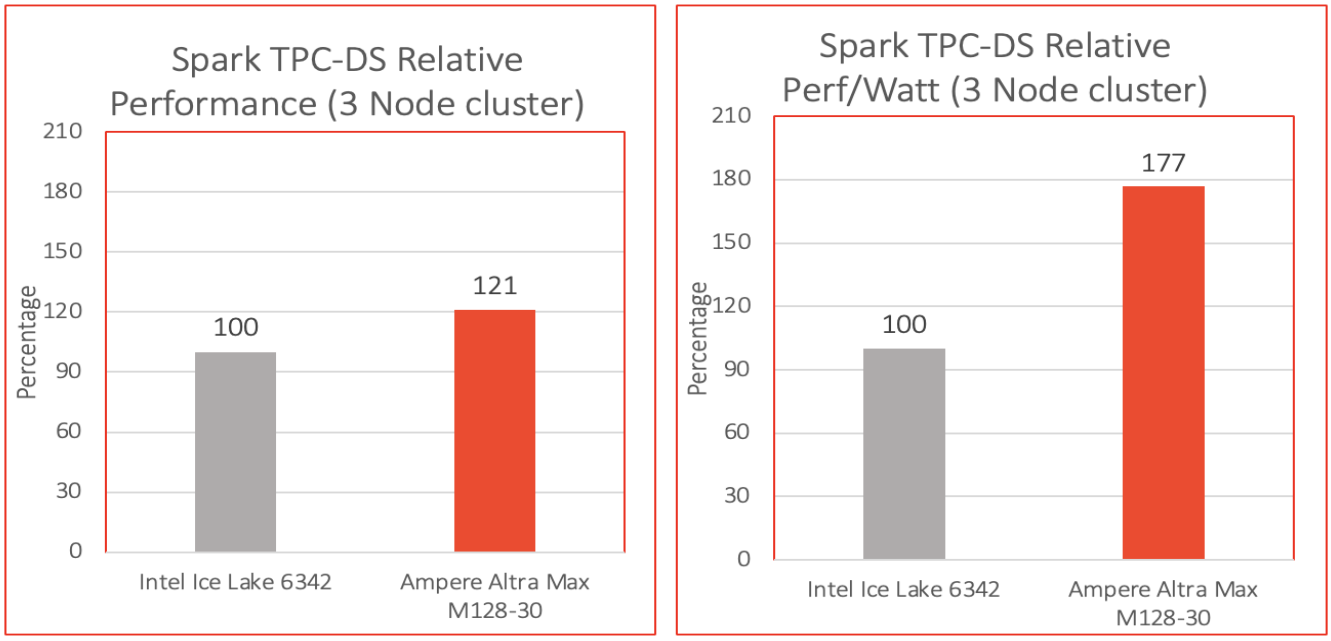
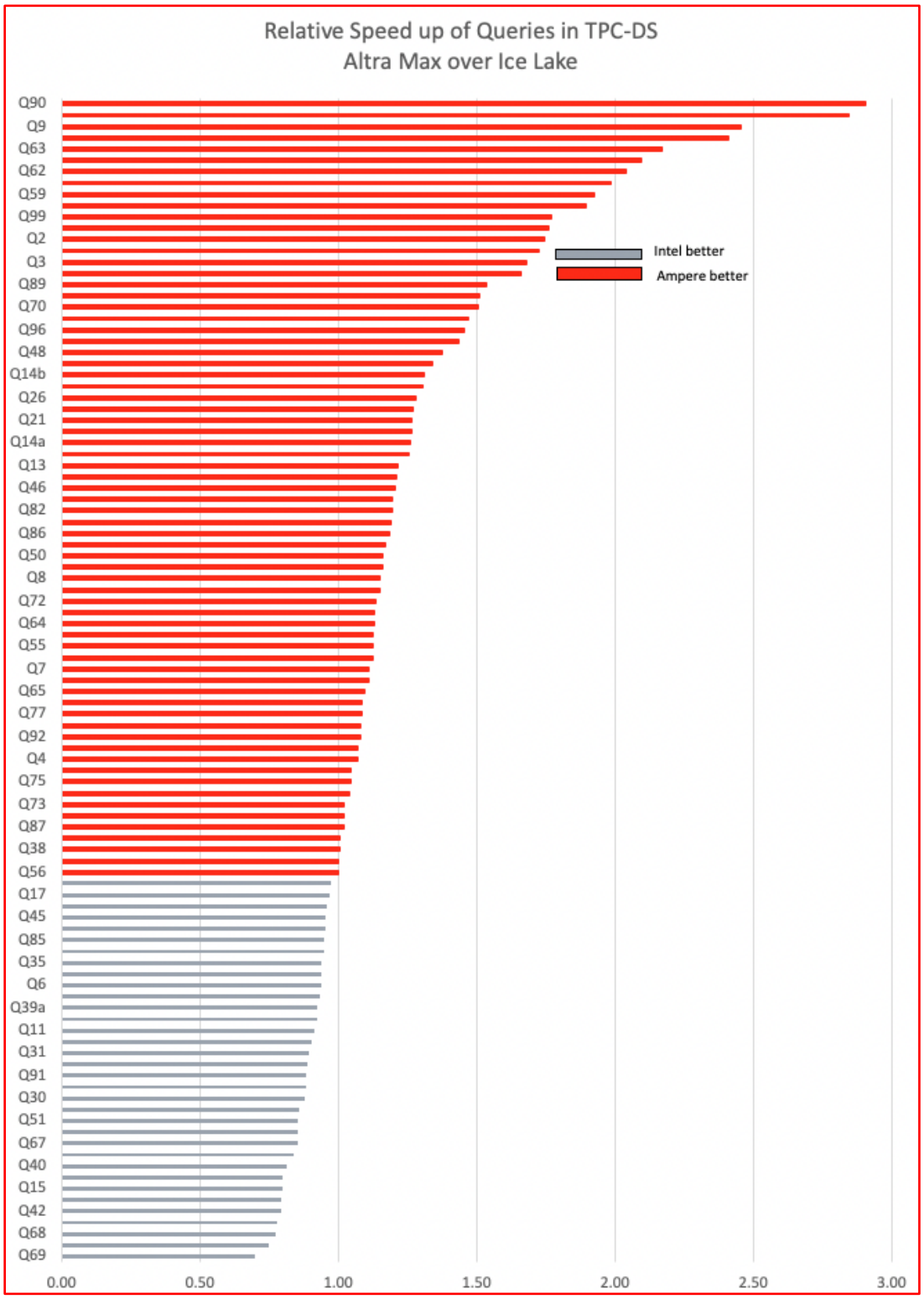


Figure 9 illustrates the speed-up of SQL queries (relative) during the execution of the TPC-DS benchmark on the HPE RL300 cluster equipped with Ampere Altra Max processors in comparison to the Dell PowerEdge R650 cluster featuring Intel Ice Lake processors. Although there were a few SQL queries that exhibited better performance on the Ice Lake systems, the majority of queries completed faster on the Altra Max systems.

Figure 9: Relative Speed Up of Queries in TPC-DS



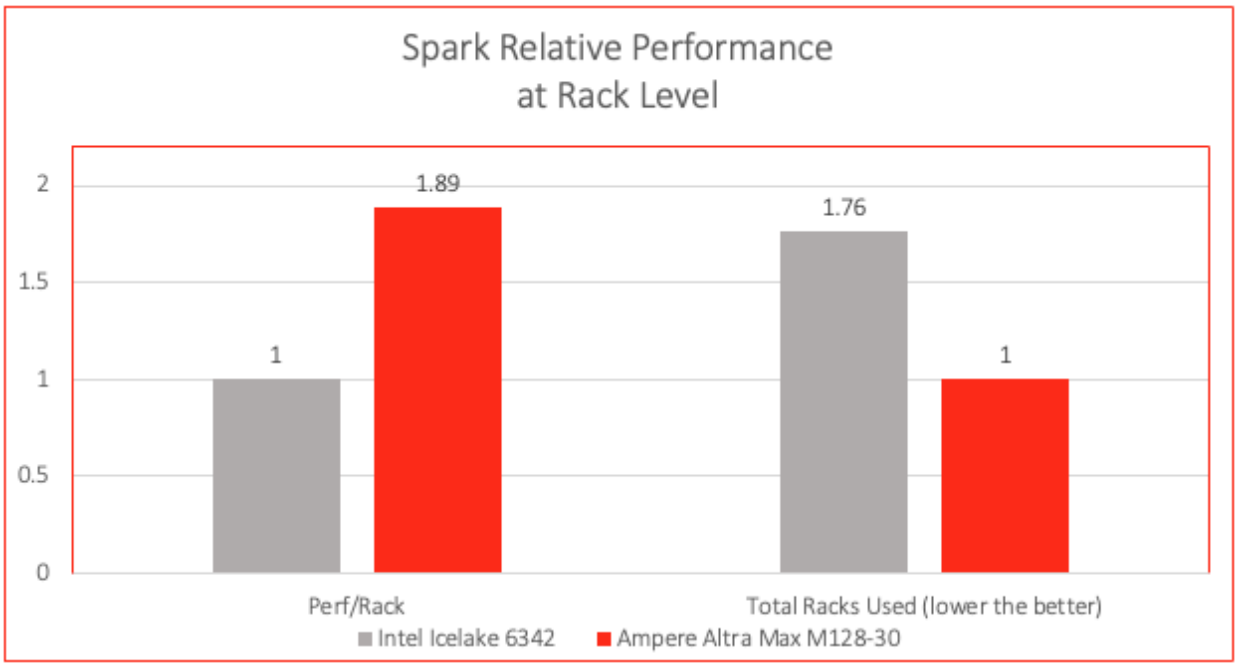
11. Rack and Datacenter Level Efficiency

When dealing with large-scale Big Data processing using frameworks like Spark, the number of servers required depends on the size of the dataset. Scalability and sustainability are key factors in designing the infrastructure.

To evaluate performance efficiency at the rack level, we extrapolated our findings from the 3-node clusters to the rack level. Our analysis considered a 42U rack with a 12 kW power budget, which accounts for network equipment and other components. The Ampere Altra Max CPUs not only delivered superior performance but also consumed less power. This resulted in a reduced number of Ampere racks needed to achieve an equivalent level of performance compared to x86 racks.

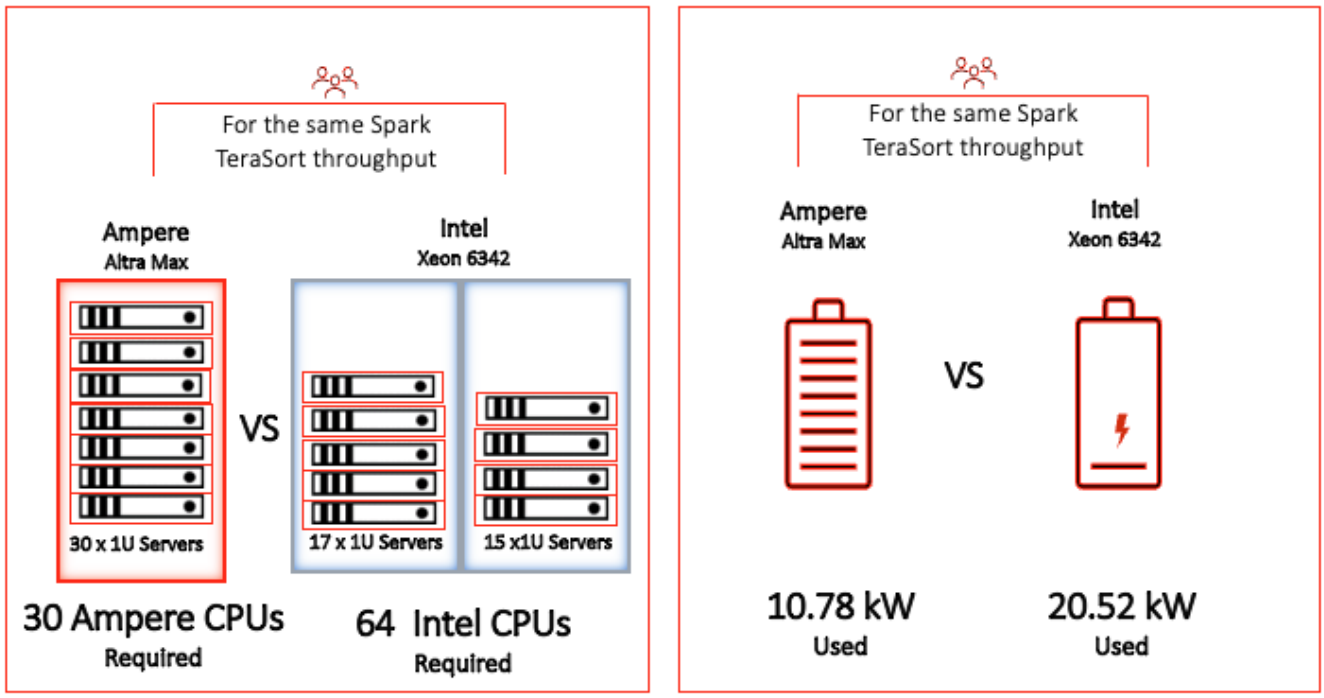
To measure performance at the rack level, we calculated the Perf/Rack metric by dividing the TeraSort throughput by the total power consumed. Based on our tests, the Altra Max servers outperformed the Intel Ice Lake servers by 89% in terms of the Perf/Rack metric, as shown in *Figure 10*.

Figure 10: Spark Relative Performance at the Rack Level



In terms of rack-level density, we observed that we could accommodate a greater number of Altra Max servers within the power limits of a rack. While a rack running a Spark workload could support a maximum of 17 Intel servers before reaching the power limit, we were able to fit 30 Altra Max servers. This represents a substantial 76% improvement in rack-level density when utilizing HPE RL300 servers equipped with Altra Max processors, as depicted in *Figure 11*.

Figure 11: CPU and Power Performance



By leveraging 30 Ampere Altra Max CPUs, it is feasible to achieve performance levels that are comparable to those achieved with 64 Intel Ice Lake CPUs. This performance equivalence leads to significant power savings of 90% when utilizing Altra Max servers for Spark workloads in comparison to Intel servers.

The architecture of Ampere processors offers sustainability advantages, delivering industry-leading performance per rack and the potential to reduce the overall resource footprint of Spark deployments by over 76%. This advantage is truly disruptive, positioning the Ampere Altra family as the most sustainable choice for on-premises and cloud deployments.

12. Conclusion

This paper presents a reference architecture for deploying Spark on a multi-node cluster with Ampere Altra Max processors. It compares this solution to a similar configuration based on Intel Ice Lake processors, with the goal of achieving maximum scalability at the rack level. The Ampere Altra Max processors offer exceptional power efficiency, linear scalability, and high performance.

Big Data solutions like Spark require significant computational power and persistent storage. By utilizing Altra Max processors for these applications, the advantages of both scale-up and scale-out architectures are leveraged. This approach allows for a densely packed core configuration per rack, resulting in reduced power consumption per rack while maintaining the same level of throughput.

Appendix A. Configuration

A.1 BIOS Changes

```
System Configuration -> BIOS Platform Configuration -> Processor Options -> ANC
Mode Monilithic

System Configuration -> Power and Performance -> Ampere Max Performance -> Enabled
```

A.2 /etc/sysctl.conf

```
kernel.pid_max = 4194303
fs.aio-max-nr = 1048576
net.ipv4.conf.default.rp_filter=1
net.ipv4.tcp_timestamps=0
net.ipv4.tcp_sack = 1
net.core.netdev_max_backlog = 25000
net.core.rmem_max = 2147483647
net.core.wmem_max = 2147483647
net.core.rmem_default = 33554431
net.core.wmem_default = 33554432
net.core.optmem_max = 40960
net.ipv4.tcp_rmem =8192 33554432 2147483647
net.ipv4.tcp_wmem =8192 33554432 2147483647
net.ipv4.tcp_low_latency=1
net.ipv4.tcp_adv_win_scale=1
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv4.conf.all.arp_filter=1
net.ipv4.tcp_retries2=5
net.ipv6.conf.lo.disable_ipv6 = 1
net.core.somaxconn = 65535
#memory cache settings
vm.swappiness=1
vm.overcommit_memory=0
vm.dirty_background_ratio=1
```

A.3 /etc/security/limits.conf

```
*           soft    nofile          65536
*           hard    nofile          65536
*           soft    nproc           65536
*           hard    nproc           65536
```

A.4 Miscellaneous Kernel Changes

```
#Disable Transparent Huge Page defrag
echo never> /sys/kernel/mm/transparent_hugepage/defrag
echo never > /sys/kernel/mm/transparent_hugepage/enabled

ifconfig enp1s0 mtu 9000
ifconfig enP1p1s0 mtu 9000
```

A.5 .bashrc File

```
export JAVA_HOME=/home/hadoop/jdk
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$classpath
export PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin

#HADOOP_HOME
export HADOOP_HOME=/home/hadoop/hadoop
export SPARK_HOME=/home/hadoop/spark
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH
```

A.6 core-site.xml

```
<configuration>

        <property>
                <name>fs.defaultFS</name>
                <value>hdfs://<server1>:9000</value>
        </property>

        <property>
                <name>hadoop.tmp.dir</name>
       <value>/data/data1/hadoop, /data/data2/hadoop, /data/data3/hadoop,
/data/data4/hadoop </value>
        </property>

        <property>
                <name>io.native.lib.available</name>
                <value>true</value>
        </property>

        <property>
                <name>io.compression.codecs</name>
<value>org.apache.hadoop.io.compress.GzipCodec,
org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.BZip2Codec,
com.hadoop.compression.lzo.LzoCodec, com.hadoop.compression.lzo.LzopCodec,
org.apache.hadoop.io.compress.SnappyCodec</value>
        </property>

        <property>
                <name>io.compression.codec.snappy.class</name>
                <value>org.apache.hadoop.io.compress.SnappyCodec</value>
        </property>

</configuration>
```

A.7 hdfs-site.xml

```
<configuration>
        <property>
                <name>dfs.replication</name>
                <value>3</value>
        </property>

        <property>
                <name>dfs.blocksize</name>
                <value>536870912</value>
        </property>

  <property>
                <name>dfs.namenode.name.dir</name>
                <value>file:/home/hadoop/hadoop_store/hdfs/namenode</value>
        </property>

        <property>
                <name>dfs.datanode.data.dir</name>
        <value>/data/data1/hadoop, /data/data2/hadoop, /data/data3/hadoop,
/data/data4/hadoop </value>
        </property>

</configuration>
```

A.8 yarn-site.xml

```
<configuration>
<!-- Site specific YARN configuration properties -->
        <property>
                <name>yarn.nodemanager.aux-services</name>
                <value>mapreduce_shuffle</value>
        </property>

        <property>
                <name>yarn.resourcemanager.hostname</name>
                <value><server1></value>
        </property>

        <property>
                <name>yarn.scheduler.minimum-allocation-mb</name>
                <value>1024</value>
        </property>

        <property>
                <name>yarn.scheduler.maximum-allocation-mb</name>
                <value>49152</value>
        </property>

        <property>
                <name>yarn.scheduler.minimum-allocation-vcores</name>
                <value>1</value>
        </property>

        <property>
                <name>yarn.scheduler.maximum-allocation-vcores</name>
                <value>112</value>
        </property>

        <property>
                <name>yarn.nodemanager.vmem-pmem-ratio</name>
                <value>4</value>
        </property>
```

```
        <property>
                <name>yarn.nodemanager.resource.memory-mb</name>
                <value>409600</value>
        </property>

        <property>
                <name>yarn.nodemanager.resource.cpu-vcores</name>
                <value>112</value>
        </property>

        <property>
                <name>yarn.log-aggregation-enable</name>
                <value>true</value>
        </property>
</configuration>
```

A.9 mapred-site.xml

```
<configuration>
        <property>
                <name>mapreduce.framework.name</name>
                <value>yarn</value>
        </property>

        <property>
                <name>yarn.app.mapreduce.am.env</name>
                <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
        </property>

        <property>
                <name>mapreduce.map.env</name>
                <value> HADOOP_MAPRED_HOME=$HADOOP_HOME,
LD_LIBRARY_PATH=$LD_LIBRARY_PATH </value>
        </property>

        <property>
                <name>mapreduce.reduce.env</name>
                <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
        </property>

             <property>
                <name>mapreduce.application.classpath</name>
                <value>$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/*,
$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/lib-examples/*,
$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/sources/*,
$HADOOP_MAPRED_HOME/share/hadoop/common/*,
$HADOOP_MAPRED_HOME/share/hadoop/common/lib/*,
$HADOOP_MAPRED_HOME/share/hadoop/yarn/*, $HADOOP_MAPRED_HOME/share/hadoop/yarn/lib/*,
$HADOOP_MAPRED_HOME/share/hadoop/hdfs/*,
$HADOOP_MAPRED_HOME/share/hadoop/hdfs/lib/*</value>
                </property>

        <property>
                <name>mapreduce.jobhistory.address</name>
                <value><server1>:10020</value>
        </property>

        <property>
                <name>mapreduce.jobhistory.webapp.address</name>
                <value><server1>:19888</value>
        </property>
```

```
<property>
        <name>mapreduce.map.memory.mb</name>
        <value>2048</value>
</property>

<property>
        <name>mapreduce.map.cpu.vcore</name>
        <value>1</value>
</property>

<property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>4096</value>
</property>

<property>
        <name>mapreduce.reduce.cpu.vcore</name>
        <value>1</value>
</property>

<property>
        <name>mapreduce.map.java.opts</name>
        <value>-Xmx1536m -XX:+UseParallelGC -XX:ParallelGCThreads=32</value>
</property>

<property>
        <name>mapreduce.reduce.java.opts</name>
        <value>-Xmx3072m -XX:+UseParallelGC -XX:ParallelGCThreads=32</value>
</property>

<property>
        <name>mapreduce.task.timeout</name>
        <value>6000000</value>
</property>

<property>
        <name>mapreduce.map.output.compress</name>
        <value>true</value>
</property>

<property>
        <name>mapreduce.map.output.compress.codec</name>
        <value>org.apache.hadoop.io.compress.SnappyCodec</value>
</property>

<property>
        <name>mapreduce.output.fileoutputformat.compress</name>
        <value>true</value>
</property>

<property>
        <name>mapreduce.output.fileoutputformat.compress.type</name>
        <value>BLOCK</value>
</property>

<property>
        <name>mapreduce.output.fileoutputformat.compress.codec</name>
        <value>org.apache.hadoop.io.compress.SnappyCodec</value>
</property>

<property>
        <name>mapreduce.reduce.shuffle.parallelcopies</name>
        <value>32</value>
</property>
```

```
        <property>
                <name>mapred.reduce.parallel.copies</name>
                <value>32</value>
        </property>

</configuration>
```

A.10 spark-defaults.conf

```
spark.driver.memory 14g # used driver memory as 64g for TPC-DS
spark.dynamicAllocation.enabled=false
spark.executor.cores 5
spark.executor.extraJavaOptions=-Djava.net.preferIPv4Stack=true -XX:+UseParallelGC -
XX:ParallelGCThreads=32
spark.executor.instances 70
spark.executor.memory 14g
spark.executorEnv.MKL_NUM_THREADS=1
spark.executorEnv.OPENBLAS_NUM_THREADS=1
spark.files.maxPartitionBytes 128m
spark.history.fs.logDirectory  hdfs://<Master Server>:9000/logs
spark.history.fs.update.interval 10s
spark.history.provider org.apache.spark.deploy.history.FsHistoryProvider
spark.history.ui.port 18080
spark.io.compression.codec=org.apache.spark.io.SnappyCompressionCodec
spark.io.compression.snappy.blockSize=512k
spark.kryoserializer.buffer 1024m
spark.master                     yarn
spark.master.ui.port 8080
spark.network.crypto.enabled=false
spark.shuffle.compress true
spark.shuffle.spill.compress true
spark.sql.shuffle.partitions 12000
spark.ui.port 8080
spark.worker.ui.port 8081
spark.yarn.archive hdfs://<Master Server>:9000/spark-libs.jar
spark.yarn.jars=/home/hadoop/spark/jars/*,/home/hadoop/spark/yarn/*
```

A.11 hibench.conf

```
hibench.default.map/shuffle.parallelism 12000
# 3 node cluster
hibench.scale.profile bigdata
# the bigdata size configured as hibench.terasort.bigdata.datasize
30000000000 in ~/HiBench/conf/workloads/micro/terasort.conf
```

Revision History

ISSUE	DATE	DESCRIPTION
1.00	June 22, 2023	Initial issue.

June 22, 2023

Ampere Computing reserves the right to change or discontinue this product without notice.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

The information contained in this document is subject to change or withdrawal at any time without notice and is being provided on an "AS IS" basis without warranty or indemnity of any kind, whether express or implied, including without limitation, the implied warranties of non-infringement, merchantability, or fitness for a particular purpose.

Any products, services, or programs discussed in this document are sold or licensed under Ampere Computing's standard terms and conditions, copies of which may be obtained from your local Ampere Computing representative. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Ampere Computing or third parties.

Without limiting the generality of the foregoing, any performance data contained in this document was determined in a specific or controlled environment and not submitted to any formal Ampere Computing test. Therefore, the results obtained in other operating environments may vary significantly. Under no circumstances will Ampere Computing be liable for any damages whatsoever arising out of or resulting from any use of the document or the information contained herein.

Ampere Computing
4655 Great America Parkway, Santa Clara, CA 95054
Phone: (669) 770-3700
https://www.amperecomputing.com

Ampere Computing reserves the right to make changes to its products, its datasheets, or related documentation, without notice and warrants its products solely pursuant to its terms and conditions of sale, only to substantially comply with the latest available datasheet.