Studying Dask - 50.012 Networks Project

Ho Jin Kind (1002846), Khong Jia Wei (1002700), Myo Oo (1002829),
See Wan Yi Faith (1002851), Timothy Liu Kaihui (1002653)

26 November 2019

Abstract

We study Dask (a Python distributed computation library) in terms of its network communication protocols and understand the design choices made. In addition, by studying impact of the network on Dask application performance, we aim to provide insights that will help to inform network requirements for Dask. This will in turn inform the impact of physical network design choices on Dask application performance.

1 Introduction

Dask (Rocklin 2015) is a widely-used library for distributed, parallel computing in Python. Dask is popular and easy to use as it natively uses existing Python libraries, APIs and data types, allowing Python users to easily adapt their code to scale up using Dask.

Dask accelerates a wide range of economically significant workloads that are developed using the Python programming language. This is possible as Dask can effectively parallelize Python libraries that people commonly use on their personal computers, like NumPy, Pandas and Scikit-Learn. While these packages were not designed to scale beyond a single machine (or even a single CPU core), Dask is capable of scaling Python computations up to large distributed computing clusters connected by high speed interconnect, following high performance computing (HPC) paradigms for aggregating the combined ability of individual computation nodes to perform large-scale computations (insideHPC 2019).

2 Related Work

There is a lack of comprehensive study about the impact of network performance on Dask and how it relates to the design of Dask network communication protocols. However, since Dask falls into the category of High Performance Computing (HPC), we can do a survey of the papers that examine the impact of networking on HPC.

2.1 Metrics describing Network Performance

There are two main metrics that describe network performance, latency and throughput, which will differ for different types of network transport layer protocols, such as TCP, UDP and MPI which are commonly used in HPC (Huang 2005). In particular, Dask uses TCP (Dask Development Team 2016). As such, we will focus on the measurement of latency and throughput of TCP connections while evaluating network performance. Latency is measured in the form of round-trip time, or RTT, as it is a better measure of network performance as in HPC, where application latency is closely characterised by the time it takes for packet to be transmitted from one process on one node, through the NIC, through the switch, to a NIC on a second node and to a process on the second node (HPC Advisory Council 2009).

2.2 Approaches to Analyse Network Performance

There are two main ways of analysing the network performance, passive and active. Passive approaches perform a packet capture to record historical network traffic and analyse the performance. Active approaches introduces specific workloads into the system to measure the performance characteristics of the network and the application. In a HPC setting, active approaches are preferred since we can easily launch new workloads and take direct measurements of the network performance (Huang 2005). Performance studies generally show a positive correlation between the network performance and application performance for scientific applications like NAMD (Phillips 2005) and MPQC (Peng 2018). This is a result which we hope to replicate for Dask.

2.3 Benchmarking Dask

It is not a straightforward task to design a workload to benchmark distributed parallel computing as we need to take into account the characteristics of the workload. The workload can be characterised by how well it is parallelizable and the degree of data locality and dependencies, which affects the amount and frequency of message passing between Worker nodes. In particular, element-wise operations in an array are extremely parallelizable, while an operation such as `sum` involves a tree reduction, which is less parallelizable with complex data dependencies (Rocklin 2017). Due to limited time and resources, we will need to design a single workload that attempts to be representative of a real-world workload, which will fall in the range between extremely and poorly parallelizable.

2.4 Dask Protocol and Communication Design

We did an extensive study on the well-written Dask documentation (Dask Development Team 2016). In particular, we made extensive use of the documentation about the Dask distributed library, that allowed us to glean much information regarding how the Dask Protocol and Communication layers are designed. These are explored in Section 6, where we look at how the heartbeat mechanism is implemented, as well as how messages are exchanged in general.

3 Problem Statement

We aim to understand and evaluate the following:

- Design of Dask network communication protocol
- Impact of network performance on Dask application performance

In particular, we will examine the impact of various network factors on Dask performance when performing a certain workload.

In particular, the network factors we will examine are:

- Network bandwidth class, such as 10 gigabit, 1 gigabit networks
- Network throughput
- Network delay and round-trip time (RTT)
- Packet loss

By understanding the impact of various factors that characterise network performance on Dask performance, we were able to derive insights to better inform real-world network requirements for Dask computing clusters using real performance data.

4 Approach Statement

4.1 Dask communication protocol design

We will examine the Dask documentation to understand the various aspects of Dask communication protocols, in particular the overall architecture of the Dask application and the communication layer.

We will create our own Dask cluster in a virtualised environment. This will allow us to record actual Dask network traffic (via packet capture) to see how it relates to what is described in the documentation.

4.2 Impact of network performance on Dask

We will vary the various factors that affect the overall network performance in a controlled setting and measure the impact of individual factors on Dask application performance via an active approach. We will launch a Dask computation and measure the average time taken for the computation to complete. We correlate the Dask application performance to the network performance and relate that back to the design of the Dask communication protocols and the network design of a Dask cluster.

5 Experiment Setup

We have created a virtualised environment using the Proxmox VE hypervisor software, running on Debian 10. Proxmox VE uses Linux Kernel Virtual Machine (KVM) as the underlying hypervisor (Proxmox Server Solutions 2019). We have a total of 5 Ubuntu virtual machines (VM) that are part of our experiment setup:

- 1 x Client (also used for packet capture)
- 1 x Dask scheduler
- 3 x Dask workers

Virtualisation Host
We used a single PC running the Proxmox VE hypervisor on top of Debian 10. The CPU is a 10-core Intel Xeon E5-2680v2 (2.80GHz) with hyper-threading and virtualisation extensions enabled. The system memory is provided by quad channel DDR3 (1333 MHz) RAM with ECC.

5.1 Dask Application Setup

The overview of our network and Dask setup are shown below:

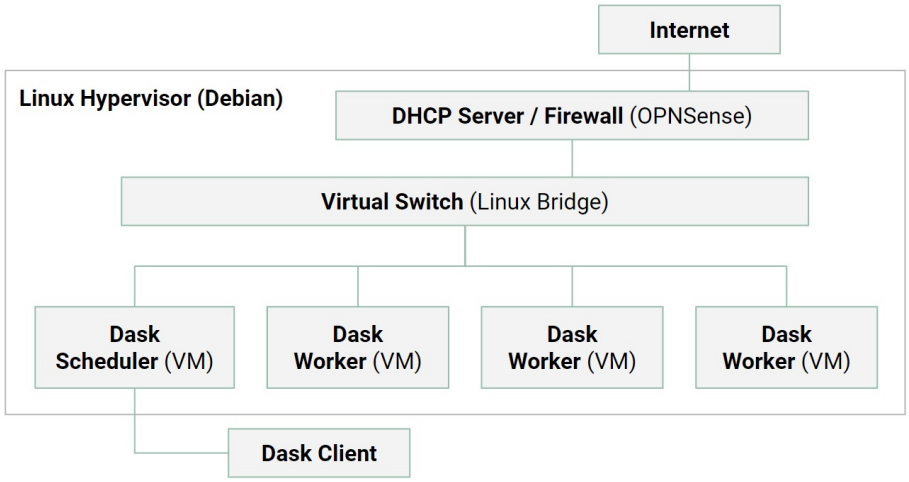


Figure 1: Overall Architecture of Setup

```
client = Client("192.168.1.116:8786")
client
```

Out[1]:

Client
Scheduler: tcp://192.168.1.116:8786
Dashboard: http://192.168.1.116:8787/status

Cluster
Workers: 3
Cores: 24
Memory: 18.74 GB

Figure 2: Dask Client output showing number of workers

5.2 Packet Capture During Experiments

We use the Client virtual machine (which is external to the Dask cluster) to do the packet capture with Wireshark. This is done to avoid the substantial CPU and RAM overhead of doing the packet capture on the Dask cluster, which will add an additional factor that affects the performance. In fact, the packet capture was so intensive that the our first attempt to do a packet capture resulted in the virtual machine running out of RAM. This is due to the extremely large amount of traffic during a Dask computation. We rely on the hypervisor to isolate system resources between the virtual machines.

However, to achieve a complete packet capture in the virtualised environment, we will need two additional configurations to be made:

Linux Bridge Packet Forwarding
On the hypervisor (physical host machine), we configured the Linux network bridge to have the property `bridge_ageing 0`. This is crucial for packet capture to work properly as it effectively configures the switch to not learn any IP-MAC address mapping in its internal forwarding table, and hence will forward every packet to every MAC address connected to the bridge (Nanni 2016). This behaviour is similar to that of a Layer 1 Hub, as opposed to a Layer 2 Switch which should only forward packets to the intended destination once the IP-MAC address mapping is learnt in the form of a MAC address lookup table.

Promiscuous Mode Network Interface
On a network interface not specifically set to promiscuous mode, packets received that are not intended for the current host will be ignored and dropped (Weadock 2009). However, on our Client VM, we want to perform a full packet capture. Hence, we set the network interface to promiscuous mode: `ip link set ens18 promisc on`.

We use this same setup for all the packet capturing for all the experiments.

5.3 Network Performance During Experiments

We made use of two methods to vary the network performance during our experiments:

1. KVM hypervisor features (different type of network interface card, bandwidth cap)
2. Linux Traffic Control Network Emulator (`tc-netem`)

Using these methods in the same virtualised environment, we are able to vary specific properties of the network in a controlled setting.

6 Design of Dask Communication Protocols

6.1 Dask Heartbeat Mechanism

A Dask cluster uses a heartbeat mechanism to ensure workers are in a healthy state ("liveliness"). In addition, this enables the Dask scheduler to maintain a "central clock" to accurately keep track of events in the cluster. This is required since the individual clocks of the nodes across the cluster may not be in perfect sync (Dask Development Team 2019c).

By performing a network capture while the cluster is idle, we are able to observe the mechanism for the heartbeat exchange between the Dask Scheduler and a Worker node. The overview of the packet exchange between the Central Task Scheduler and Worker Node is shown below:

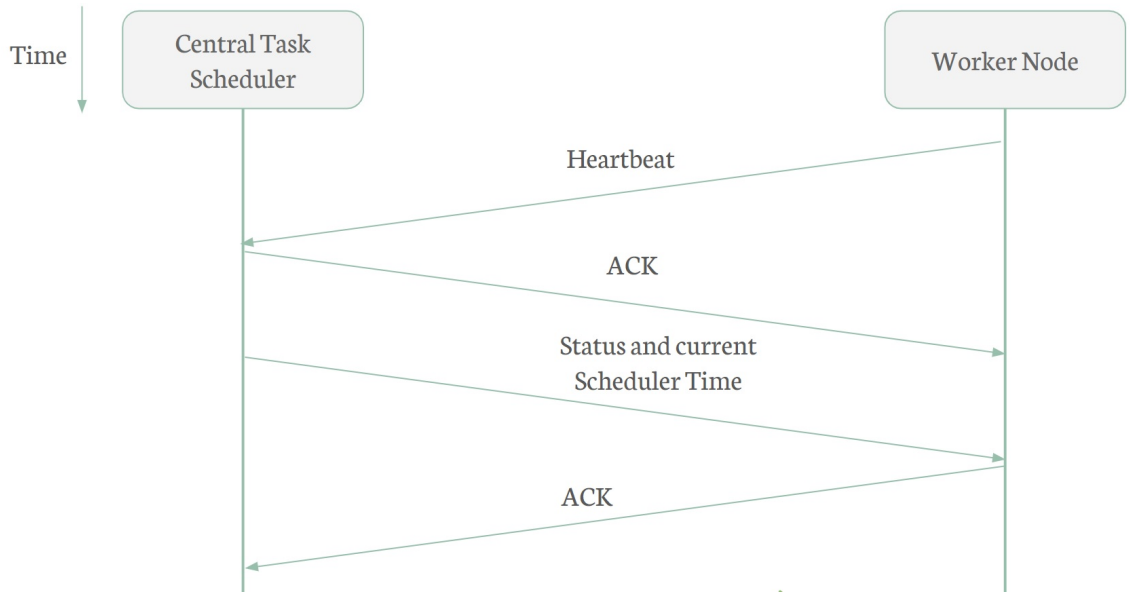


Figure 3: Scheduler-Worker Overview

6.1.1 Worker Heartbeat

The heartbeat from one particular Worker node is observed. The source IP of the packet is the IP of the Worker, `192.168.1.117`, and the destination IP is the IP of the Scheduler, `192.168.1.116`.

Source IP	192.168.1.117
Destination IP	192.168.1.116
Length of Data	279 bytes
Sequence	280
ACK	78

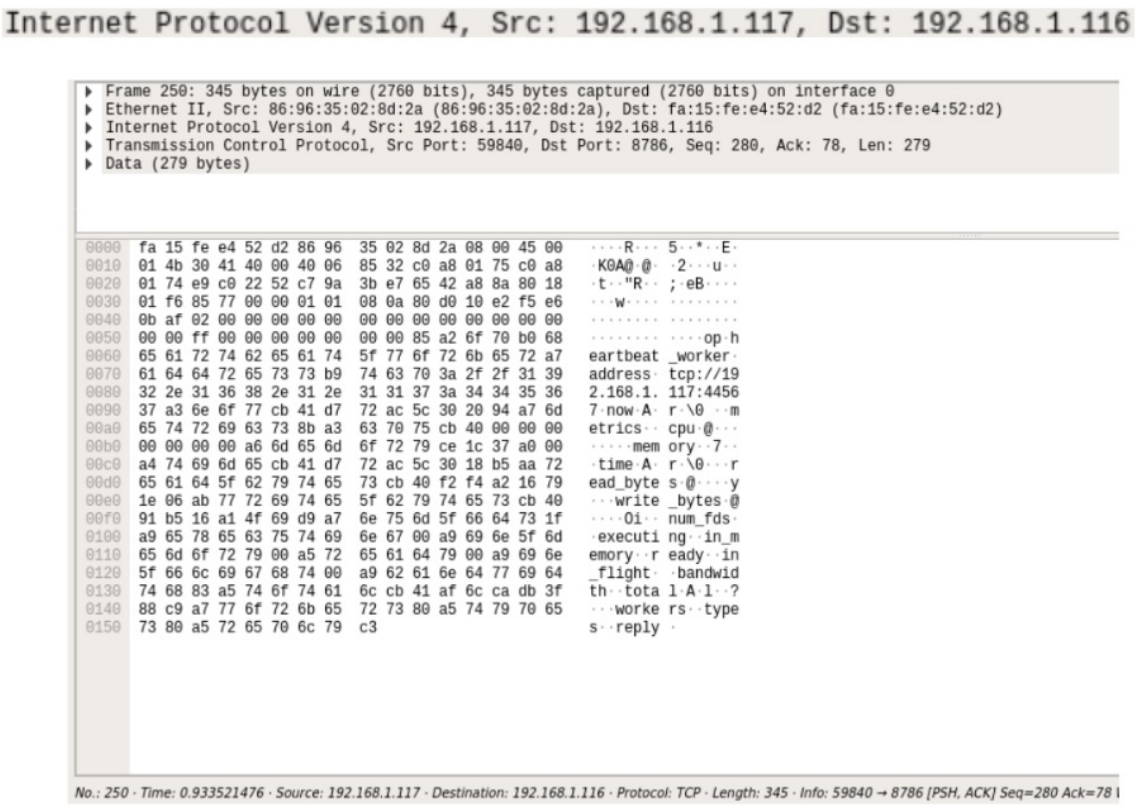


Figure 4: Worker Heartbeat

6.1.2 Scheduler ACK

The ACK sent from the Scheduler is then observed, with packet source IP being that of the Scheduler, `192.168.1.116`.

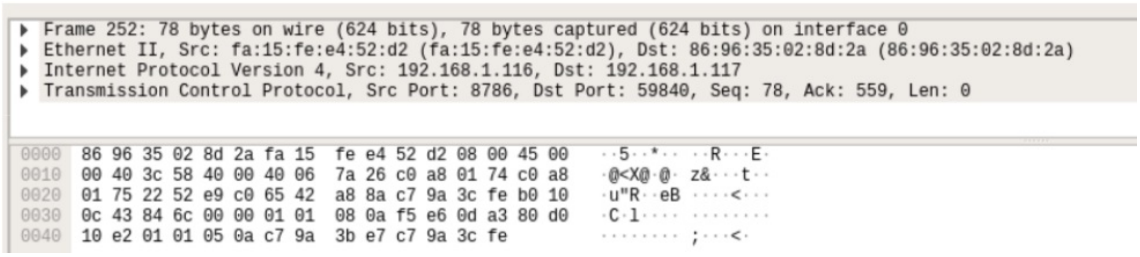


Figure 5: Scheduler ACK

Source IP	192.168.1.116
Destination IP	192.168.1.117
Length of Data	0 bytes
Sequence	78
ACK	559

6.1.3 Scheduler Status, Time, Heartbeat Interval

The scheduler with the IP of `192.168.1.116` sends the status, time and heartbeat interval.

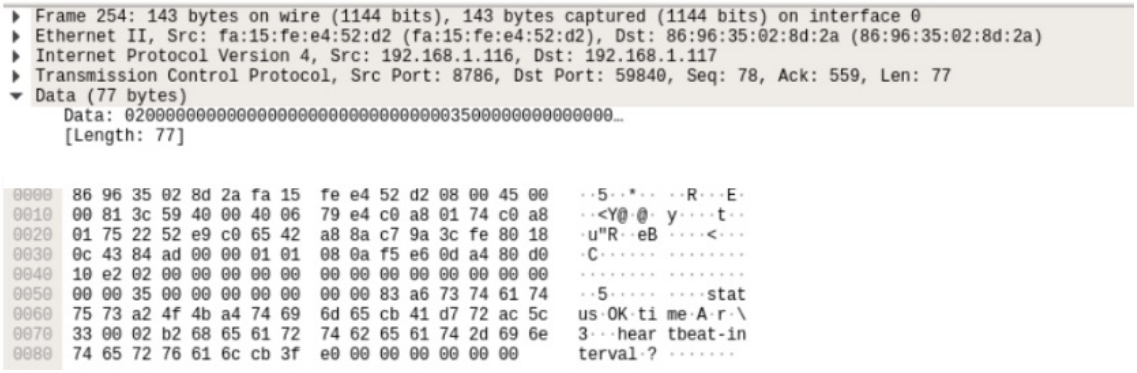


Figure 6: Scheduler Status, Time, Heartbeat Interval

Source IP	192.168.1.116
Destination IP	192.168.1.117
Length of Data	77 bytes
Sequence	78
ACK	559
Interval	1022ms

6.1.4 Worker ACK

The ACK sent from the worker is observed with the IP of the worker being `192.168.1.117`.

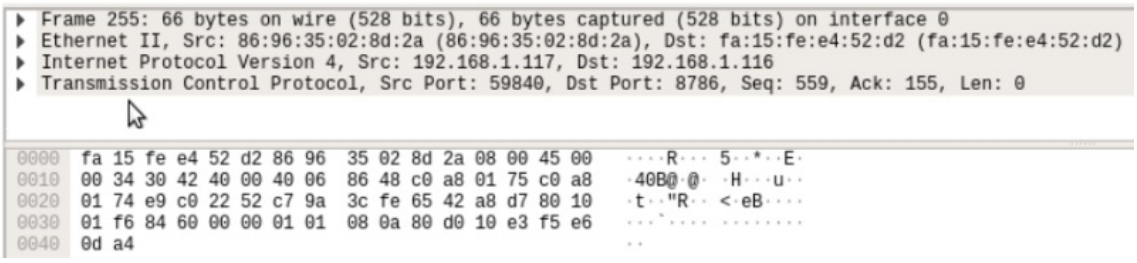


Figure 7: Worker ACK

Source IP	192.168.1.117
Destination IP	192.168.1.116
Length of Data	0 bytes
Sequence	559
ACK	155

6.2 Dask Communication Layer

The Dask communication layer handles appropriate encoding and shipping of messages between the distributed endpoints (Dask Development Team 2019a). The communication layer is able to select between different transport implementations, depending on user choice or (possibly) internal optimizations. In particular, minimizes data movement when possible, exploiting data locality (Dask Development Team 2019b). This also means that intermediate results are never transferred back to the scheduler, and only exchanged between Worker nodes when necessary.

6.2.1 Current Encoding Schemes

Most messages are encoded with MsgPack, an efficient, self-describing semi-structured serialization format. (MessagePack 2019). This provides Dask with a highly performant serialization format for the packets to be exchanged between nodes, which is crucial for performance.

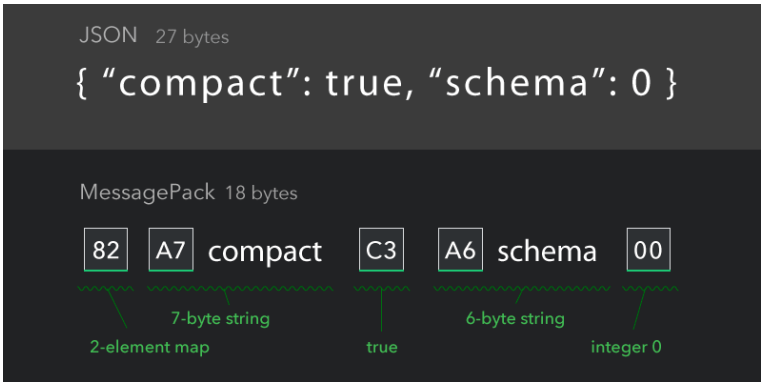


Figure 8: MsgPack encoding

TCP is the transport layer protocol used by Dask. Like any other TCP application, Dask uses TCP sockets and allows for IPv4 and IPv6 addresses. In addition, users can specify keys and certificates in order to configure Dask to use secure TLS over TCP.

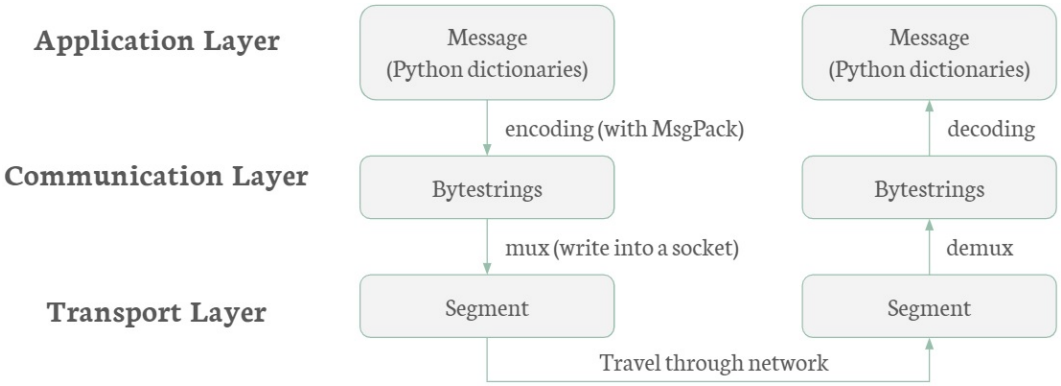


Figure 9: Layer Overview

7 Network Performance Analysis

We measured the performance of 5 different virtual network interfaces cards (NICs) by observing the average RTT, average throughput and consequently the average computation time.

7.1 Overview of Network Interface Cards

We provide a brief overview of the different NICs below, in order of performance level (from high to low).

VirtIO The VirtIO network interface is a paravirtualized network interface, promising the highest levels of performance by not requiring the use of any emulation, and using a virtualization-specific device driver in the guest OS. VirtIO allows packets to be switched directly to other hosts via an efficient data plane, bypassing both the hypervisor and the QEMU process which is typically used for emulating the network interface. This data plane is implemented via the Linux `vhost` interface, which allows it to be offloaded to a host kernel thread (Ariel 2019).

VirtIO with 10 Gbit cap The VirtIO paravirtualized network interface with a 10 gigabit throughput cap imposed by KVM, configured through Proxmox VE. This is intended to simulate the performance of a real-world 10 gigabit network (10GBASE-X).

VirtIO with 1 Gbit cap The VirtIO paravirtualized network interface with a 1 gigabit throughput cap imposed by KVM, configured through Proxmox VE. This is intended to simulate the performance of a real-world 1 gigabit network (1000BASE-X). An interesting comparison can be made when we contrast this with the emulated Intel E1000 network interface.

Intel E1000 The Intel E1000 emulated network interface attempts to replicate the feature-set as well as mimic the behaviour of Intel's 82545EM 1 gigabit (1000BASE-X) network interface card (Nobel 2014). Due to the emulated interface (implemented by QEMU), there is a context switch for every packet going between the kernel and the VM. This results in greater processing overhead, ultimately resulting in greater RTT and lower throughput compared to paravirtualised network interfaces like VirtIO.

Realtek 8139 The Realtek 8139 is another emulated network interface. It attempts to replicate the feature-set and mimic the behaviour of Realtek's 8139D network interface, which follows the Fast Ethernet (100BASE-TX) standard. It is the slowest network interface, and is included out of interest in comparing RTT and throughput, and is too slow to be of practical use in any Dask computations.

7.2 Network Interface Performance

In this section, we will evaluate the performance of the individual network interfaces and how they can affect Dask application performance.

7.2.1 Round-trip Time

We measured the average round-trip time (RTT) with `ping` across 30 samples. A lower average RTT is more ideal as it means less network latency between the various nodes while performing a computation task.

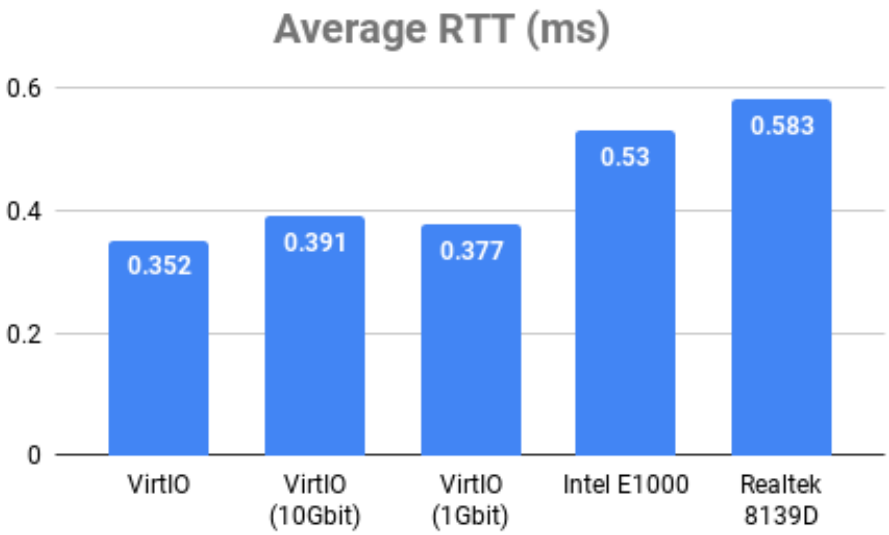


Figure 10: Average RTT

Due to the advantages provided by paravirtualization, the VirtIO network interfaces have much lower RTT compared to the emulated devices (Intel E1000 and Realtek 8139).

7.2.2 TCP Throughput

We measured the average TCP throughput achievable on each network interface with `iperf`, averaged across 5 samples. A higher average throughput is more ideal while performing a computation task.

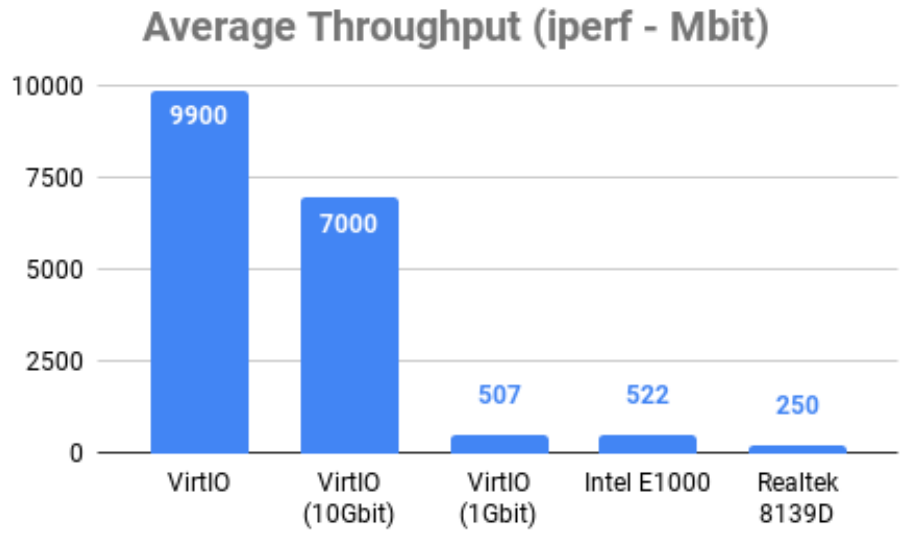


Figure 11: Average Throughput

Accuracy of E1000/RTL8139D Performance We see that the Intel E1000 exceeds the performance of VirtIO capped to 1 gigabit throughput, and Realtek 8139 exceeding *(significantly!)* the expected performance of a 100 megabit network interface. This is a result of the device emulation in QEMU being focused on driver compatibility through the emulation, rather than being accurate in terms of actual performance. The implication of this is that our results for Dask application performance might not be accurate for a non-virtualised replica of the current set-up. However, this does not invalidate our current subject of study, which aims to relate performance of the current network interface (whatever it might be) to Dask application performance.

7.3 Distributed Computation Experiment Setup

7.3.1 Example Computation

For all our experiments, we designed a single, non-trivial array operation to be performed with Dask. This workload was designed to have both well-parallelizable (large amounts of element-wise array operations) and some more challenging parts (sum, which involves a tree reduction). This represents a non-trivial array operation that can provide a loose approximation for a real-world workload.

```
w = da.random.random((10000, 10000), chunks=(1000, 1000))
x = da.random.random((10000, 10000), chunks=(1000, 1000))
y = da.random.random((10000, 10000), chunks=(1000, 1000))
z = da.random.random((10000, 10000), chunks=(1000, 1000))
numerator = ((w + w.T) * (x + x.T) * (y + y.T) * (z + z.T)).sum()
denominator = (w * x * y * z).sum()
result = numerator/denominator
```

In all cases, the Dask computation is designed with the above code and executed with `result.compute()`.

To demonstrate that this computation is indeed scalable using Dask, we perform a simple experiment.

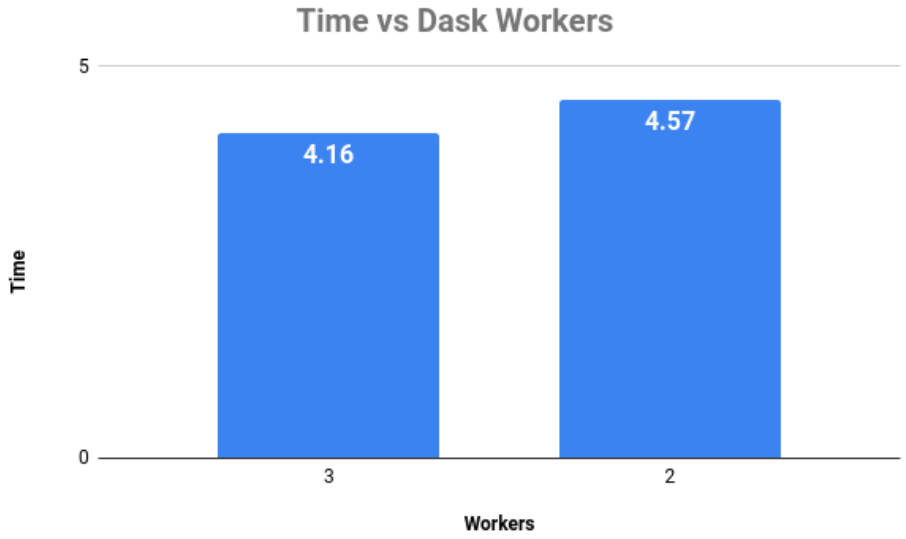


Figure 12: Comparison of computation time when worker count is decreased from 3 to 2

7.3.2 Average Computation Time

We measured the average computation time across 10 samples. A lower average computation time is more ideal.

Here, we can make some analysis on the correlation between computation time, RTT and throughput. We can see that with a lower average RTT and higher throughput, the computation time decreases, which is not surprising.

However, we can derive an interesting insight from the comparison between VirtIO (1 gigabit) and Intel E1000 (emulated 1 gigabit):

- Intel E1000 provides higher throughput
- VirtIO (1 gigabit) provides lower RTT
- VirtIO (1 gigabit) completes the computation faster

From this result, it would seem that a network interface with lower RTT results in better results despite also having a lower throughput.

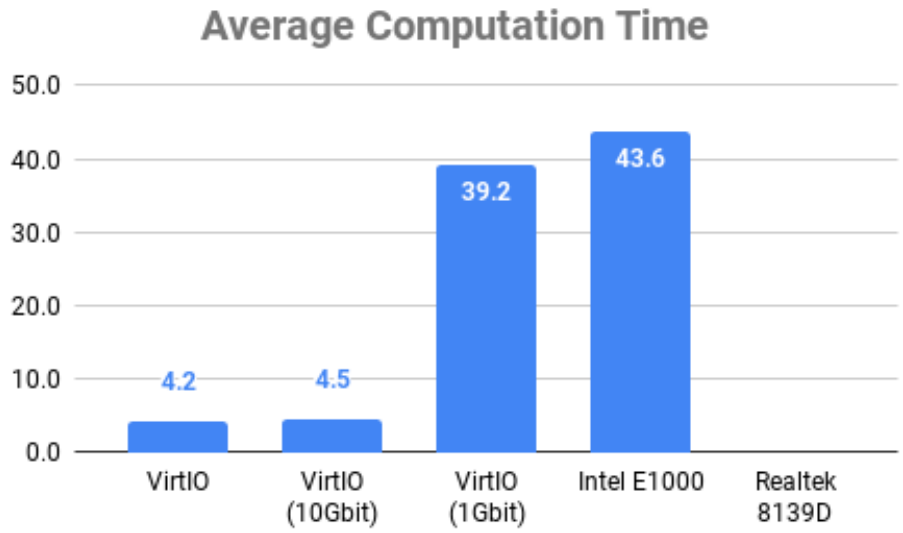


Figure 13: Average computation time when using different network interfaces with different performance levels.

7.4 Dask Performance Analysis

To better measure the impact of adjusting the individual factors that affect network performance, we will be using the highest performance VirtIO network interface for all our experiments below. These experiments will make use of the network emulator built into the Linux Traffic Control mechanism (`tc-netem`).

7.4.1 Impact of Delay on Scheduler and Worker

We incrementally increase the **egress** delay on the network interface of either the Scheduler or one of the Workers. Adding delay by a certain value has the impact of adding that amount of time to the RTT. We then record the average time taken to complete 10 rounds of the same computation. We can see that, as expected, the delay will cause an increase in the computation time. The graphs show the relationship between the added delay and the time taken for computation.

Adding Delay on Scheduler

We can see that the computation time increases proportionally as delay at the Scheduler increases. This is because it takes longer for the Workers to receive the packets from the Scheduler. Majority of these packets (about 48% of 1109 packets) contain a message that have the `op` type `compute_task`, which defines a computation task for the Worker.

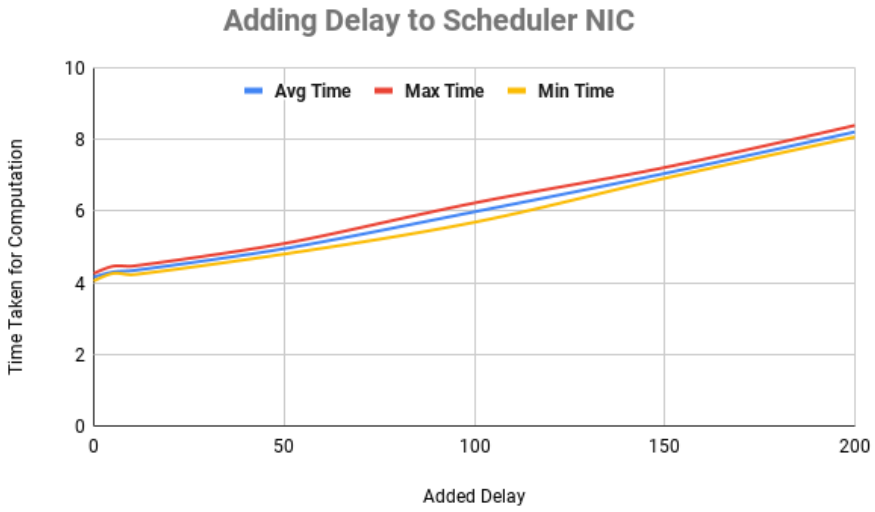


Figure 14: Impact of Delay on Scheduler

Adding Delay on 1 Worker

We can see that the computation time increases more than proportionally as delay at one Worker increases. This is primarily because it takes longer for the other Workers to receive the packets from that one Worker. Majority of these packets contain binary data (about 98% of 22398 packets) which are intermediate results that are exchanged between the Worker nodes. Also note that one Worker transmits about 20 times as many packets as the Scheduler, which explains the profoundly greater impact of increased transmission delay at a Worker.

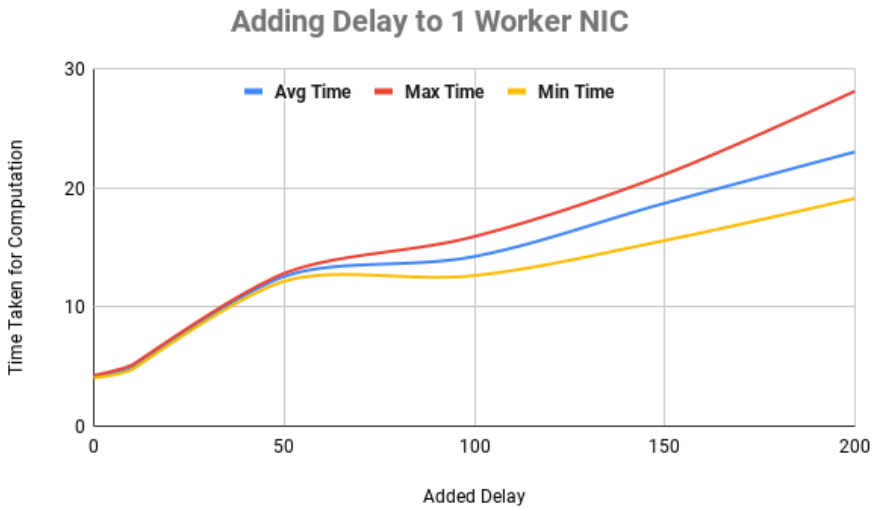


Figure 15: Impact of Delay on Worker

7.4.2 Impact of Packet Loss on Scheduler and Worker

We incrementally increase the **egress** packet loss on the network interface of either the Scheduler or one of the Workers, and record the average time taken to complete 10 rounds of the same computation. We can see that, as expected, the increasing packet loss will cause an increase in the computation time as the TCP protocol will be required to re-transmit the dropped packets until all the packets have been successfully transmitted.

Impact of Packet Loss on Scheduler

Dask can tolerate packet loss of communication from Scheduler to Worker very well. We can see that the average computation time increased only 0.8s (from 4.2s to 5.0s) when packet loss was incremented to 15%, which is unrealistically high, since in a normal network (no faulty hardware), packet loss is expected to be not more than 2% over a period of time (Pingman 2019).

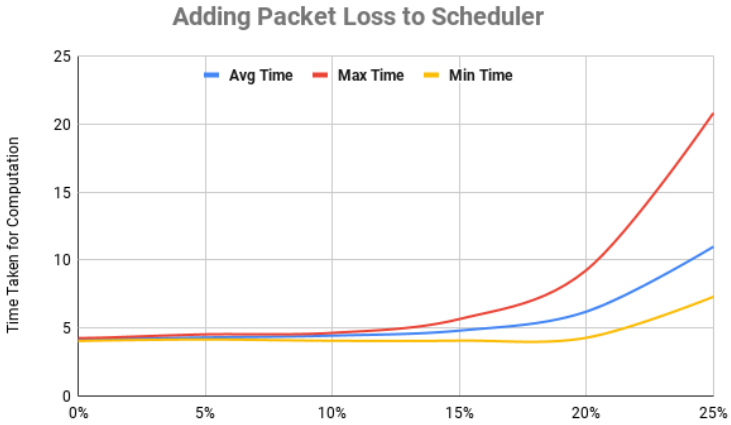


Figure 16: Impact of Packet Loss on Scheduler

Impact of Packet Loss on 1 Worker

As in the previous case (increasing delay), increasing packet loss on the Worker results in much greater impact on the Dask application performance.

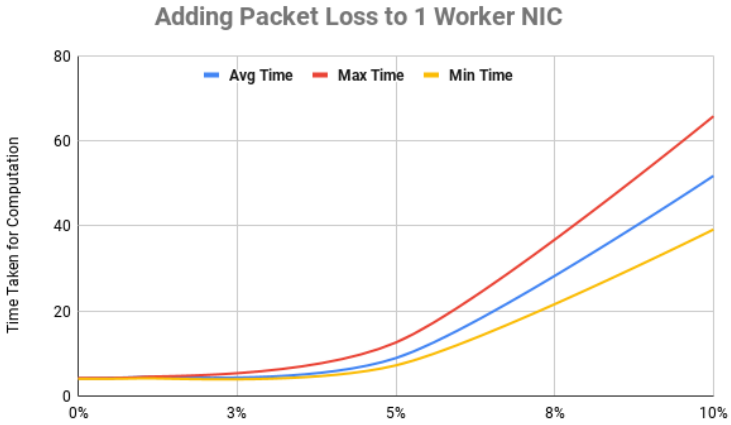


Figure 17: Impact of Packet Loss on Worker

8 Evaluation of Results

8.1 Explaining the Impact of Network Degradation

We can see that network degradation (increased delay or packet loss) affects the Dask application performance significantly more when it occurs on the Worker node, as opposed to the Scheduler node. The Dask documentation (Dask Development Team 2019b) already provided the main clues for this behaviour in stating that Dask Worker nodes exchange intermediate results without going through the Scheduler. We can see this in the stark difference in network activity of the Scheduler and Worker nodes.

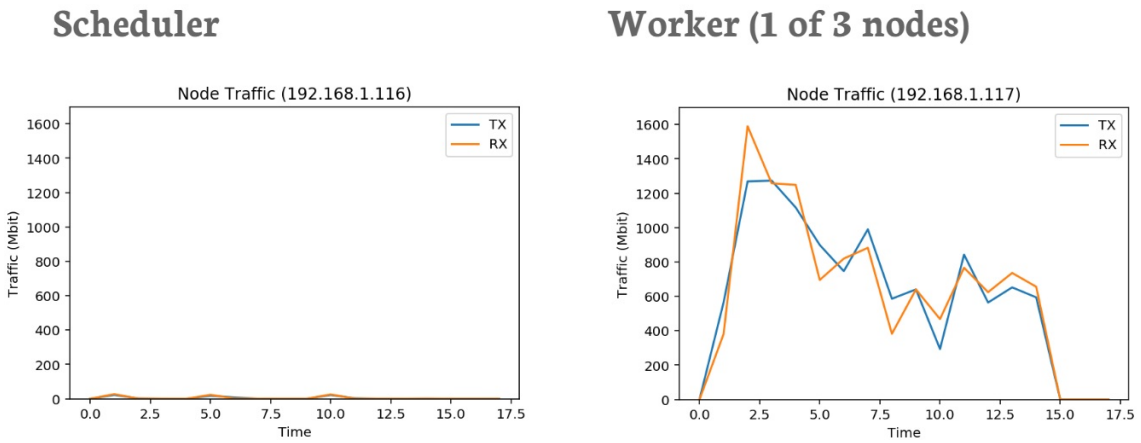


Figure 18: Node Traffic of Scheduler and 1 Worker

We can verify the network behaviour of the Scheduler and Worker nodes by doing a packet capture, followed by an inspection of the overall amount of data transferred, and the distribution of the different ops in the Dask messages that are being exchanged between Scheduler-Worker, and Worker-Worker pairs.

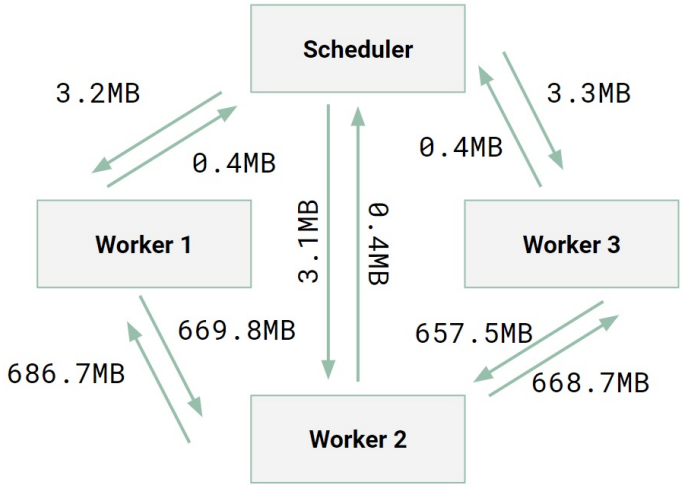


Figure 19: Size of Node Traffic of Scheduler and Workers

8.1.1 Investigating Messages between Nodes

In order to better understand this behaviour, we performed a packet capture to understand the amount and composition of messages that were sent from both the Scheduler and one Worker node. The results are shown below:

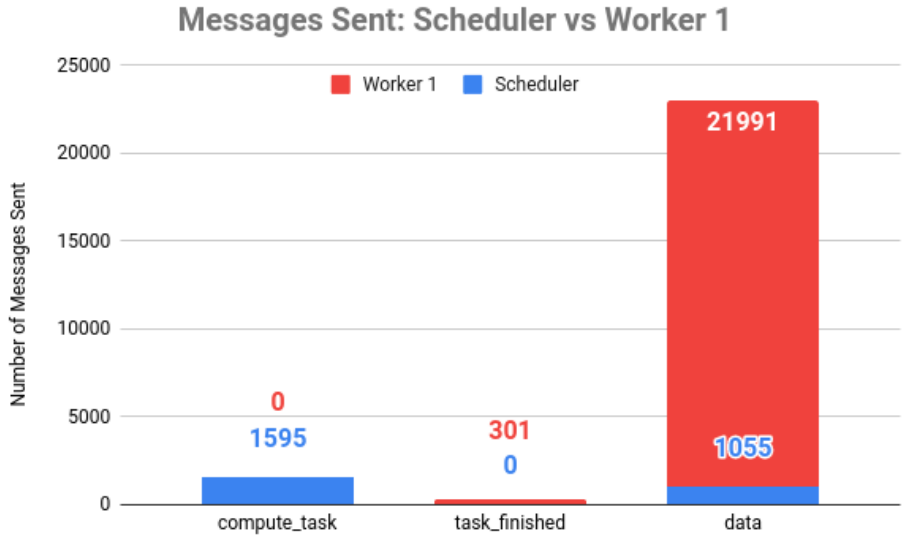


Figure 20: Messages sent by Scheduler vs Worker. Note the different composition and amount. Not all message types are shown for the sake of maintaining a clear diagram.

We can see that there is much more traffic exchanged (both in terms of number of packets and overall data transferred) between the Worker nodes, compared to between the Scheduler and Worker nodes. This is the result of the way the Dask is designed in order for more efficient computation performance. `compute-task` messages are sent from the Scheduler to the Worker nodes, which perform the task, exchanging intermediate results, and then return the result back to the Scheduler, indicating the results with a `key` value.

In order to generate numbers for the above analysis, we used Scapy (a Python library) to analyse the packet capture file. The type of Dask message can be identified via the presence of an `"op"` key in the packet contents. Some examples of packets that correspond to the diagram above are shown below (simplified representation):

```
# compute_task message
{"op": "compute-task",
 "function": "..."}

# task_complete message
{"op": "task-complete",
 "key": "y"}

# data message
{"y": b"..."}
```

8.2 Designing an Optimal Network for Dask

We can see that having a reliable network with high throughput and low RTT is very important, and that Dask performance scales with higher throughput and lower RTT. In addition, when packets between Worker nodes are dropped, impact on Dask performance is significant. This precludes using low-end network switches with low queue (buffer) size or low-throughput back-plane, as that will likely to result in packets being dropped.

8.2.1 Network Topology

When designing a larger network to be laid out in a real-world cluster, compromises might have to be made especially with a large number of nodes. For instances, the delay between individual nodes might not be the same due to different lengths of cable routing, or network topology requiring multiple hops between switches.

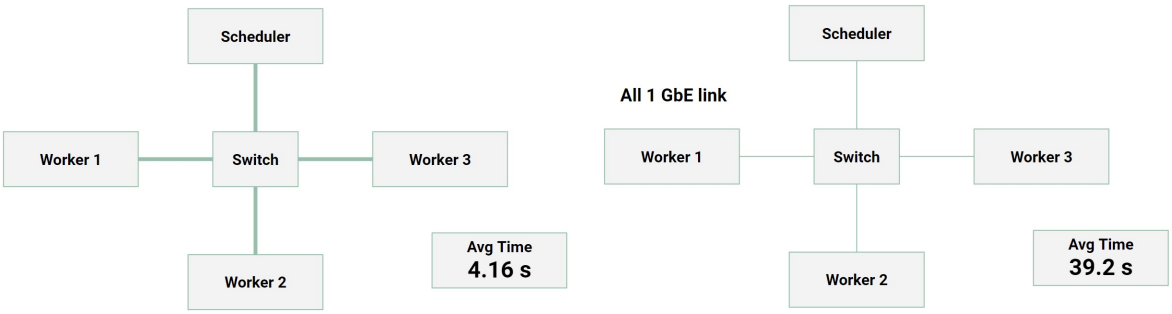
In this case, it is of greater importance to reduce RTT between Worker nodes (for example, to colocate them within the same switch) than it is to reduce RTT between the Scheduler and the Worker. We can demonstrate a similar effect in a limited fashion by showing the impact of reducing the network bandwidth class of either a Scheduler or a Worker from VirtIO to Intel E1000, with higher RTT and lower throughput.

(a) Changing only Scheduler Link to 1GbE (b) Changing one Worker Link to 1GbE

Figure 21: Effect of Links on Scheduler & Worker. Changing a link on a worker has a much greater negative impact on performance.

8.2.2 Network Bandwidth Class

Our experiments show that 10GbE is about 10 times better than 1GbE and performance continues to scale with network performance. When using 4x10GbE links between the switch and the scheduler and 3 workers, the average time taken is only 4.16s. On the other hand, using 4x1GbE links result in an average time of 39.2s. This implies that high-speed networking is a must to increase Dask application performance.



(a) Using 10GbE Links (b) Using 1GbE Links

Figure 22: Effect of Links on Average Time Taken. Reducing the network bandwidth class has a drastic impact on performance.

In addition, we also show that although the scheduler does not require the same degree of network I/O performance as the worker nodes, providing the scheduler with better network connectivity is measurably beneficial for Dask application performance.

Changing the link between the switch and the scheduler impacts the performance and increases the average time taken from 4.2s to 6.67s. On the other hand, changing the link between the switch and the worker greatly impacts the performance, increasing the average time taken to 23.7s. Hence, in a scenario where the higher-speed network ports are limited (e.g. on a particular switch with different speed ports), the higher-speed network ports should be allocated to the worker nodes.

The summary of our experiment results are documented in the table below.

Links	Average Time (s)
4 10GbE Links	4.16
1 1GbE Link (Scheduler)	6.67
1 1GbE Link (Worker)	23.7
4 1GbE Links	39.2

9 Conclusion

9.1 Code and Data Files

The code and data files (packet capture) that we used for our analysis can be found here: `https://github.com/tlkh/50.012-dask-network-project`.

9.2 Limitations

9.2.1 Limitations of Approach

Our approach is to do our experiments on a single host virtualised environment. As a result, it is possible that our experiments experience some level of CPU or memory (RAM) contention during the high-intensity part (the Dask computation) when the host CPU has to service the applications inside the virtual machines and also provide the virtualised network infrastructure. In particular, the virtualised network (Linux Bridge) is possibly CPU and memory intensive, and that is not accounted for in our experiments.

In practice, we monitor the host CPU utilization and observe that the peak CPU utilization usually does not exceed 70%, indicating that some CPU contention is possible but not significant. However, we do not have a direct way of observing system RAM utilization (how saturated the RAM bus is in terms of reads/writes), so that remains an unknown factor.

9.2.2 Limitations of Results

All our experiments are run with a single type of computation (consisting of various array transpose, addition, multiplication and reduction). However that are many other types of common workload that exhibit very different patterns in terms of amount of data movement and structure of data dependencies, both of which will be affected by the network. We did not explore wider kinds of workloads.

9.3 Future Work

In this study, we performed our experiments in a virtualised setting due to resource constraints. Hence, is worthwhile to explore the performance of Dask using the same methodology on a physical HPC cluster with high throughput networking (10GBe or even 100GBe) and evaluate how the results differ. Uwu.

References

Ariel, et al. (2019). *Introduction to virtio-networking and vhost-net*. URL: https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net.

Dask Development Team (2016). *Dask: Library for dynamic task scheduling*. URL: https://dask.org.

— (2019a). *Dask: Communications*. URL: https://distributed.dask.org/en/latest/communications.html.

— (2019b). *Dask: Data Locality*. URL: https://distributed.dask.org/en/latest/locality.html.

— (2019c). *Dask: Diagnosing Performance*. URL: https://distributed.dask.org/en/latest/diagnosing-performance.html.

HPC Advisory Council (2009). *Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing*. URL: https://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf.

Huang, et al (2005). "Network Performance in Distributed HPC Clusters". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*.

insideHPC (2019). *What is high performance computing?* URL: https://insidehpc.com/hpc-basic-training/what-is-hpc/.

MessagePack (2019). *MessagePack*. URL: https://msgpack.org/index.html.

Nanni, Dan (2016). *How to disable MAC learning in a Linux bridge*. URL: https://ask.xmodulo.com/disable-mac-learning-linux-bridge.html.

Nobel, Rickard (2014). *VMXNET3 vs E1000E and E1000 – part 1*. URL: http://rickardnobel.se/vmxnet3-vs-e1000e-and-e1000-part-1/.

Peng, et al (2018). *The Massively Parallel Quantum Chemistry Program (MPQC)*. URL: http://github.com/ValeevGroup/mpqc.

Phillips, et al (2005). "Scalable molecular dynamics with NAMD". In: *Proceedings of the Journal of Computational Chemistry*.

Pingman (2019). *What's 'normal' for latency and packet loss?* URL: https://www.pingman.com/kb/article/what-s-normal-for-latency-and-packet-loss-42.html.

Proxmox Server Solutions (2019). *Proxmox VE: Open-Source Virtualization Platform*. URL: https://www.proxmox.com/en/proxmox-ve.

Rocklin, Matthew (2015). "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: *Proceedings of the 14th Python in Science Conference*. Ed. by Kathryn Huff and James Bergstra, pp. 130–136.

— (2017). *Dask Benchmarks*. URL: https://matthewrocklin.com/blog/work/2017/07/03/scaling.

Weadock, Glenn (2009). *Wireshark and Promiscuous Mode*. URL: https://www.networkworld.com/article/2231903/wireshark-and-promiscuous-mode.html.