Whitepaper 2016-001

iRODS Networking Performance

Authors:

Terrell G. Russell
Jason M. Coposky
Benjamin Keller

Renaissance Computing Institute (RENCI)

University of North Carolina at Chapel Hill

Contents

1 Summary

iRODS 4.1.9 presents a significant improvement over 4.1.8 (and all prior versions).

With proper tuning of the Linux TCP kernel settings, iRODS 4.1.9, released July 28, 2016, represents up to a 100x speedup over iRODS 4.1.8 at high latency (100ms RTT).

This whitepaper demonstrates the recent gains in configurability and throughput as well as defines best practice for administrators and organizations using iRODS.

2 Introduction

iRODS networking has historically been managed through compile-time `#define` settings. Beginning with iRODS 4.1.0, these compile-time settings have been extracted and are now controlled through configuration variables in `server_config.json`.

As recently as iRODS 4.1.8, the TCP send and receive window sizes were also set within the compiled iRODS server code. This defining of the window sizes had the unfortunate effect of overriding the Linux kernel's TCP auto tuning and severely reducing overall throughput over higher latency connections.

This whitepaper is the first comprehensive look at how iRODS behaves in both low- and high-latency, high-bandwidth network scenarios.

iRODS 4.1.8 and 4.1.9 networking is compared across five variables: transfer command, file size, network RTT, TCP maximum buffer size, iRODS buffer size, and parallel threads.

The code to generate this dataset is available and we encourage others to add new datasets to this initial baseline.

3 Test Setup

This set of tests were performed on two bare metal computers running CentOS 7.2. Each machine had 32 processors, 256GiB of RAM, and rotational disks capable of writing 165MiB/s. They were installed in the same rack and held a 10Gbps point-to-point connection.

The values used for the $2 * 6 * 3 * 2 * 3 * 6 = 1296$ combinations initially tested are included in Table 1. Each combination was run 3 times. Median transfer times are reported.

Variable	Value
Transfer Command	iput, iget
File Size	35MiB, 100MiB, 500MiB, 1GiB, 5GiB, 10GiB
Network RTT (delay)	~0ms, 50ms, 100ms
TCP Buffers	default, tuned
iRODS Buffers	4MiB, 50MiB, 100MiB
Parallel Threads	Streaming, 2, 3, 4, 8, 16

Table 1: Independent Variables

The network latency was naturally 0.2 milliseconds. For the purposes of this analysis, 0.2ms was coded as 0ms. Additional artificial network delays of 50ms and 100ms were introduced and managed via the traffic control binary (`tc`).

```
Add 100ms delay
$ sudo tc qdisc add dev eth1 root netem delay 100ms

Delete the delay
$ sudo tc qdisc del dev eth1 root netem
```

The max TCP buffers on each machine were either the default, or they were increased to 100MiB.

```
Default TCP Buffers
$ sudo sysctl net.ipv4.tcp_rmem='4096        87380   6291456'
$ sudo sysctl net.ipv4.tcp_wmem='4096        16384   4194304'
$ sudo sysctl net.core.rmem_max=212992
$ sudo sysctl net.core.wmem_max=212992

Tuned TCP Buffers
$ sudo sysctl net.ipv4.tcp_rmem='4096        87380   104857600'
$ sudo sysctl net.ipv4.tcp_wmem='4096        87380   104857600'
$ sudo sysctl net.core.rmem_max=104857600
$ sudo sysctl net.core.wmem_max=104857600
```

The iRODS Buffers were changed on each machine by manipulating the `irods_transfer_buffer_size_for_parallel_transfer_in_megabytes` variable.

Each of the five variables were handled by the python test harness. The python test harness manipulated the settings on both ends of the connection, transferred a file (`iput` or `iget`) of various sizes (35MiB, 100MiB, 500MiB, 1GiB, 5GiB, and 10GiB) with a varying number of threads (Streaming, 2, 3, 4, 8, 16), and then removed the file.

```
$ iput -N3 5Gfile
```

The elapsed time for each transfer was recorded into a comma separated values file (csv). The graphs were generated with R. All of the test harness code and graph generation code used is available at `https://github.com/irods/contrib`.

3.1 iperf3 Baseline

`iperf3` was used to generate a baseline throughput rate at ~0ms, 50ms, and 100ms.

`iperf3` was configured for 15 seconds of memory-to-memory testing, omitting the first 15 seconds of possible TCP slowstart, and running in parallel to 1-5 ports listening on the other machine. The other parameters forced verbose output in megabytes. The values reported in Table 2 and Figure 1 are the median of 5 samples.

```
$ for i in $(seq 3); do iperf3 -p 2010$i -c 10g1 -t15 -O30 -V -fM & done
```

	~0ms delay MBytes/sec	50ms delay MBytes/sec	100ms delay MBytes/sec
5 Threads	1121.0	982.0	800.5
4 Threads	1122.0	978.0	742.9
3 Threads	1122.0	1014.0	798.0
2 Threads	1122.0	1092.0	696.0
1 Thread	1123.0	997.0	498.0

Table 2: Median iperf3 Maximum Sustained Throughput, n=5

The sustained throughput at ~0ms RTT was very consistent and near capacity for a 10Gbps connection. There was a ~10% reduction in throughput at 50ms RTT and another 20-30% reduction at 100ms RTT. A single thread only supported ~500 MBytes/sec at 100ms RTT and adding more threads increased the throughput up to ~800 MBytes/sec.

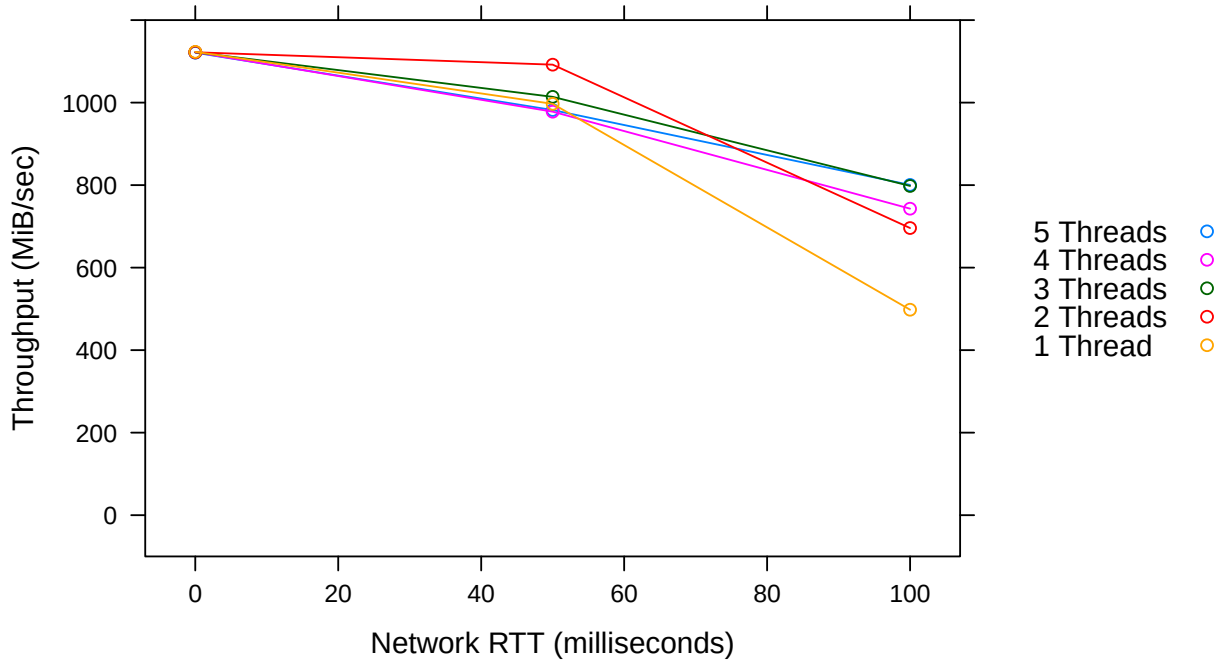

Figure 1: iperf3 @ 10Gbps point-to-point, 1-5 Parallel TCP Threads

4 iRODS 4.1.8

iRODS 4.1.8 has a very predictable, but unbalanced, network performance profile.

At low-latency (~0ms), all six panels in Figures 2-13 behave relatively identically and move the file at a significant portion of the available connection.

But once a delay is introduced, and the TCP buffers remain at their default settings, the TCP buffers are constantly maxed out and the network is starved while each end waits on the other server before sending the next buffer. The transfer time only decreases when multiple threads are used to push more data onto the network at a time.

Increasing the iRODS Buffer Size (moving from the left panels to the right panels) has no effect since the Default TCP Buffers (bottom row) are much too small to take any advantage of a larger iRODS Buffer, and the Tuned TCP Buffers (top row) take as much advantage as they can, even when the iRODS Buffers are set to the default 4MiB.

The streaming scenario is the worst case for all of these panels (with delay) due to the framing done by the iRODS Protocol on every buffer and the fact that the iRODS Protocol requires a couple round trips for every buffer to be acknowledged.

Across the file sizes (35MiB, 100MiB, 500MiB, 1GiB, 5GiB, 10GiB) for both `iput` and `iget` (Figures 2-13), the six panels remain consistent and show a strictly linear relationship between file size and transfer time.

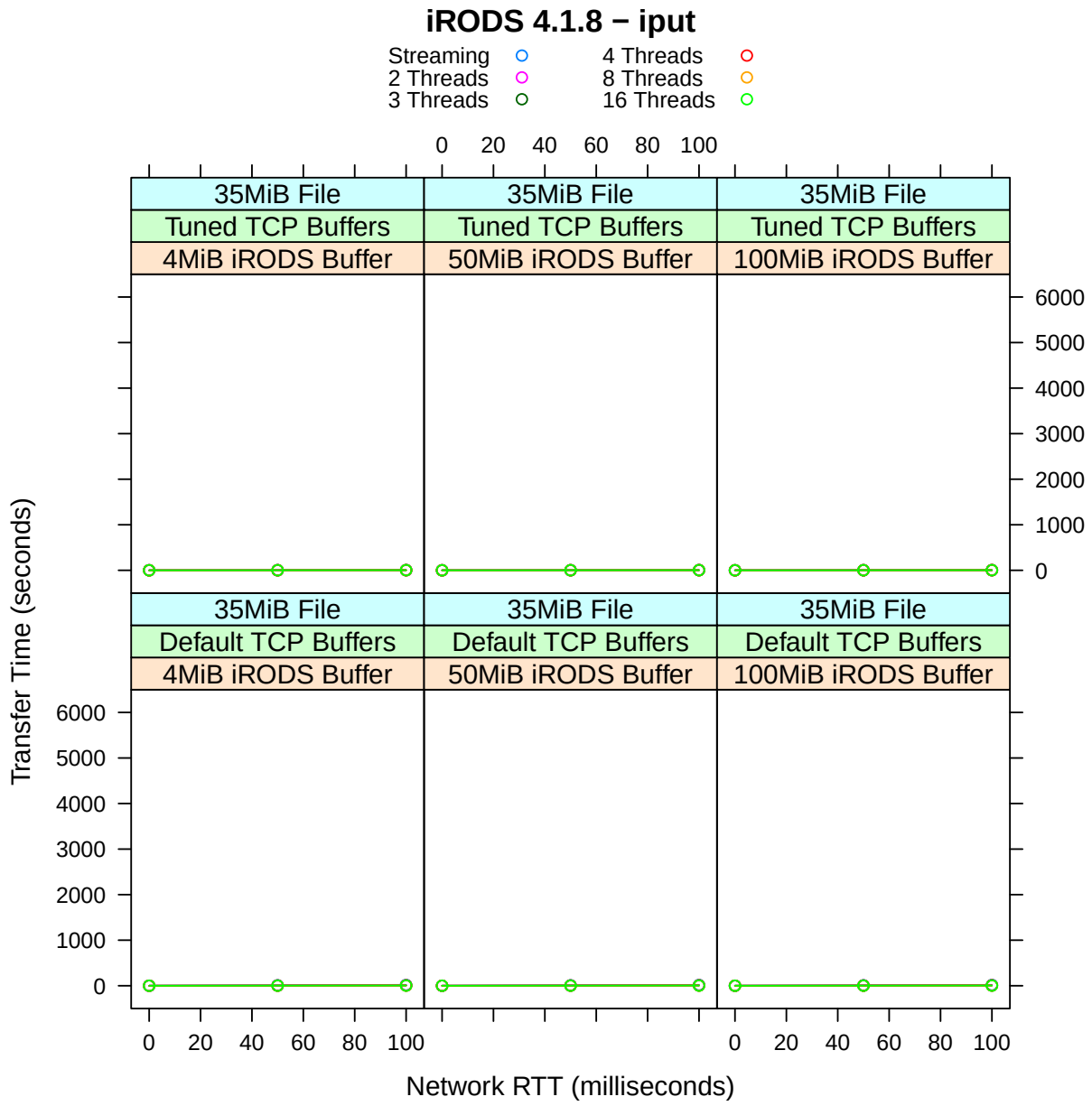

Figure 2: 4.1.8 iput, n=3

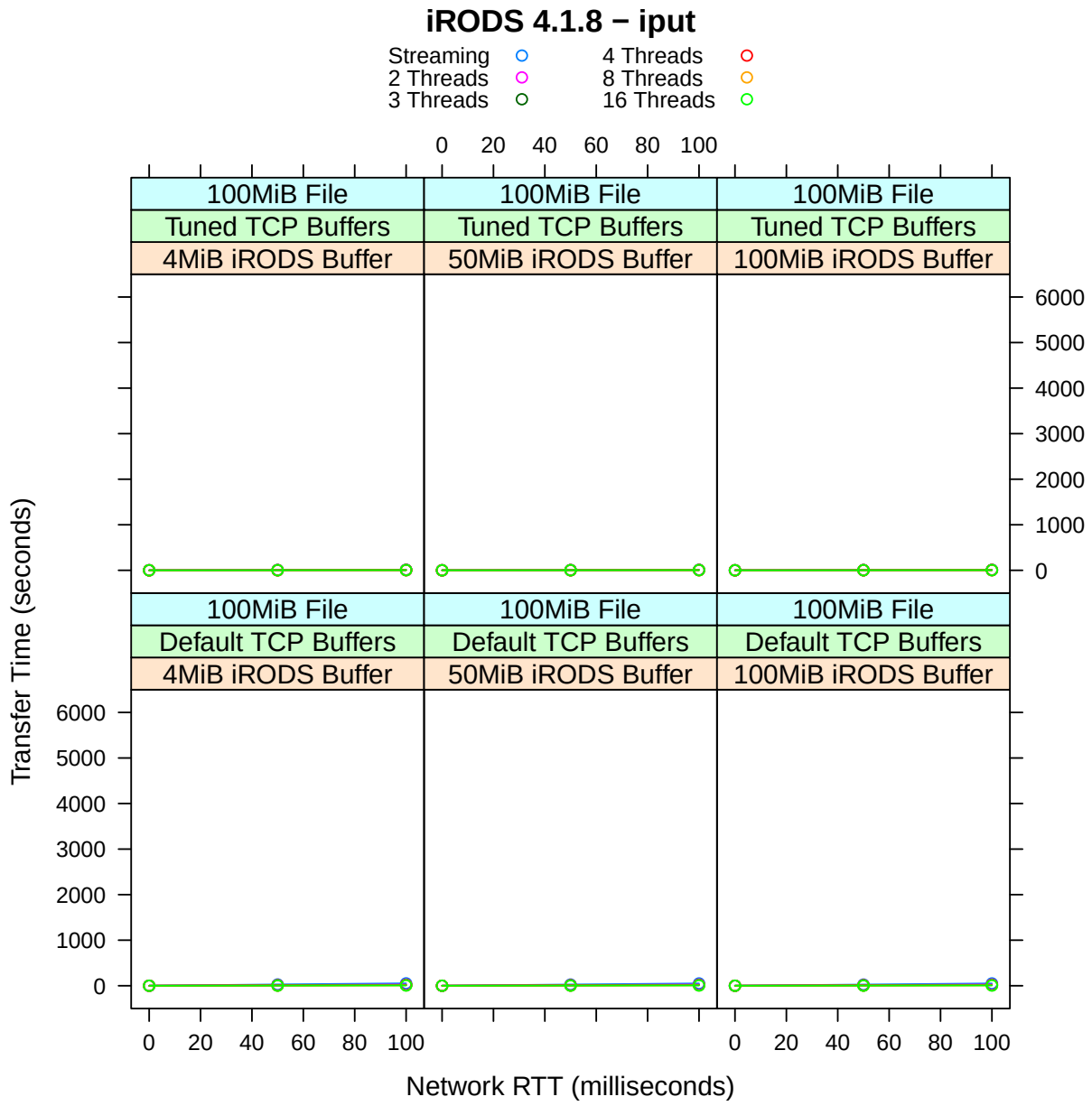

Figure 3: 4.1.8 iput, n=3

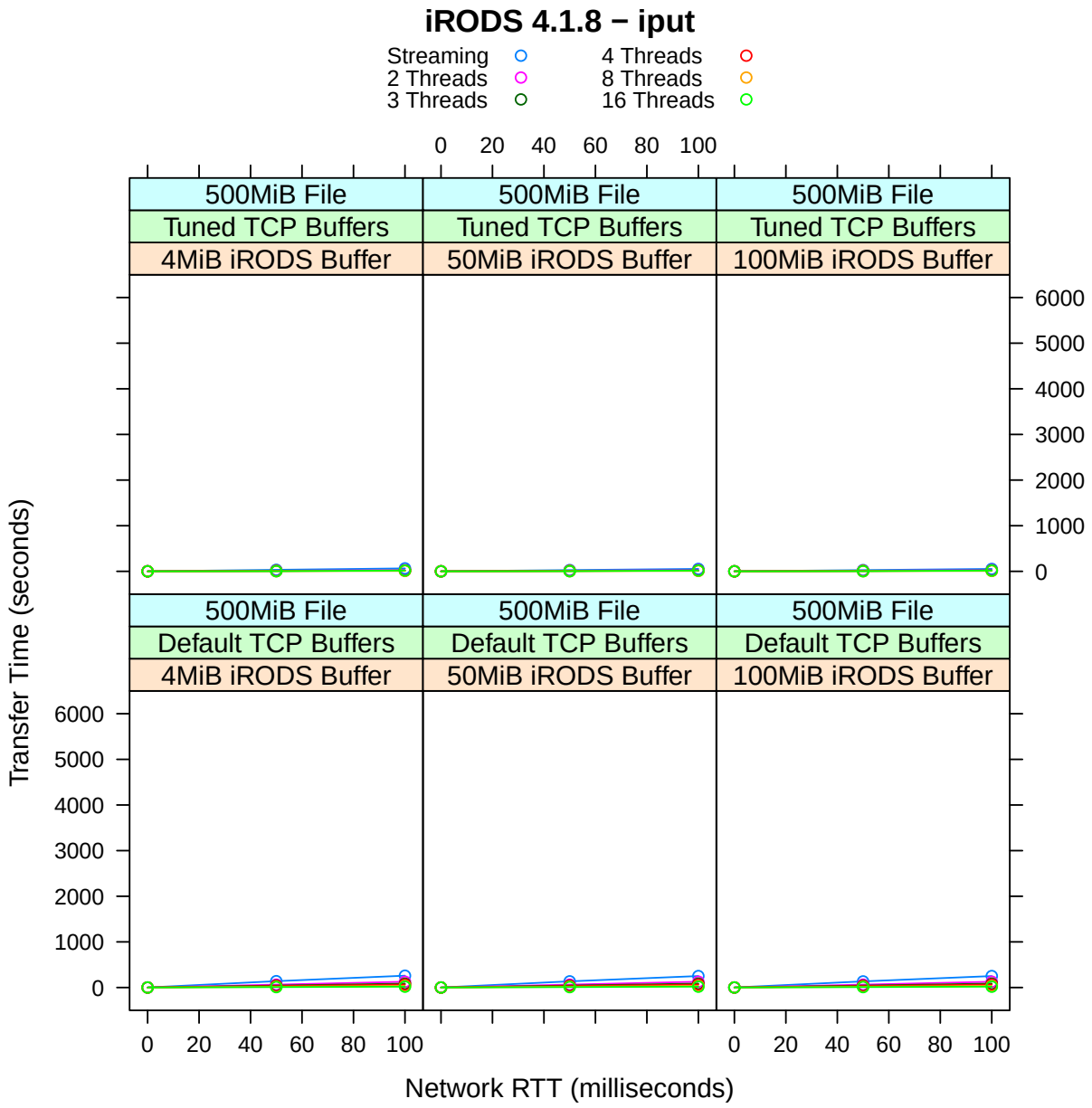

Figure 4: 4.1.8 iput, n=3

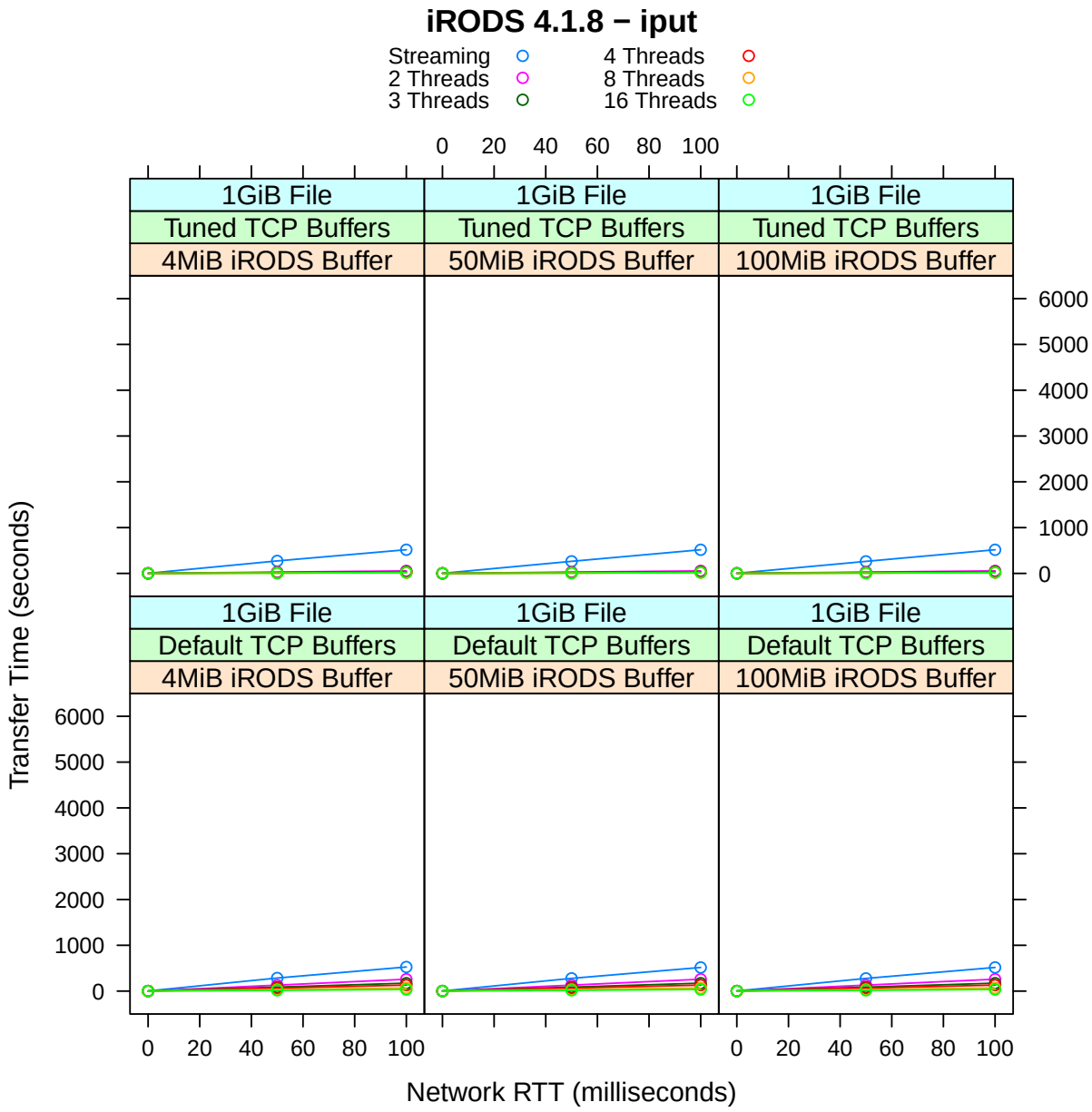

Figure 5: 4.1.8 iput, n=3

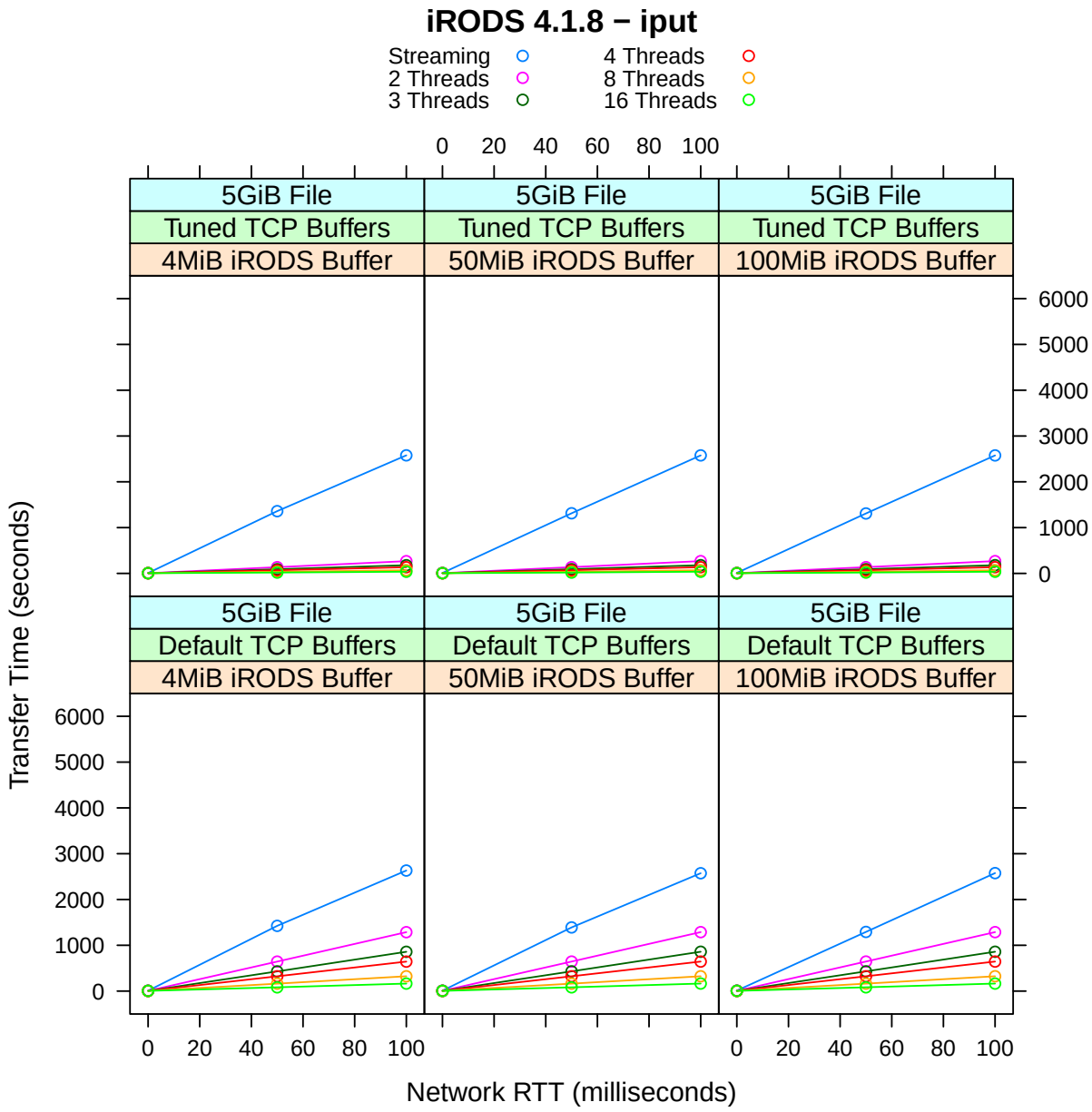

Figure 6: 4.1.8 iput, n=3

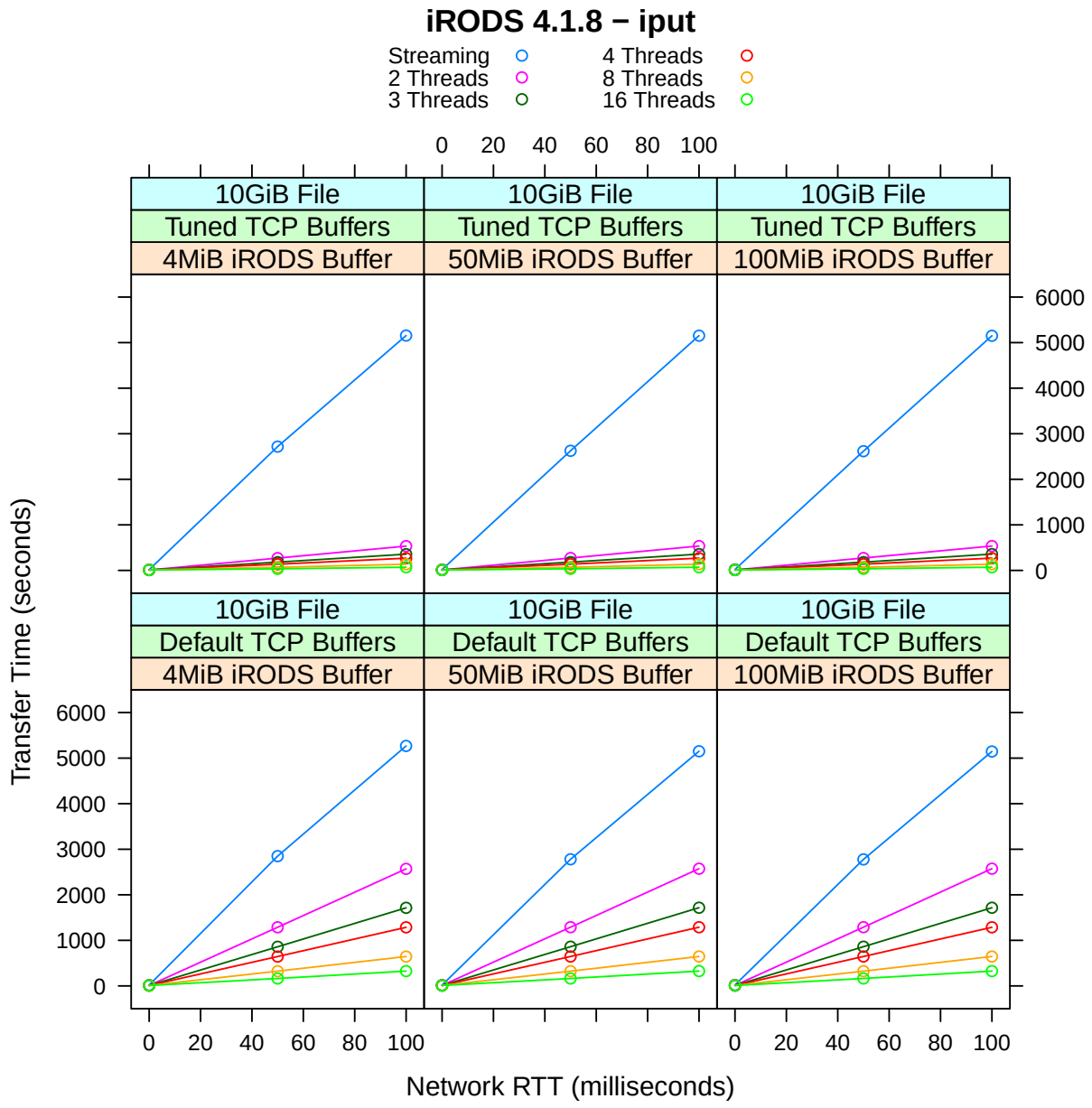

Figure 7: 4.1.8 iput, n=3

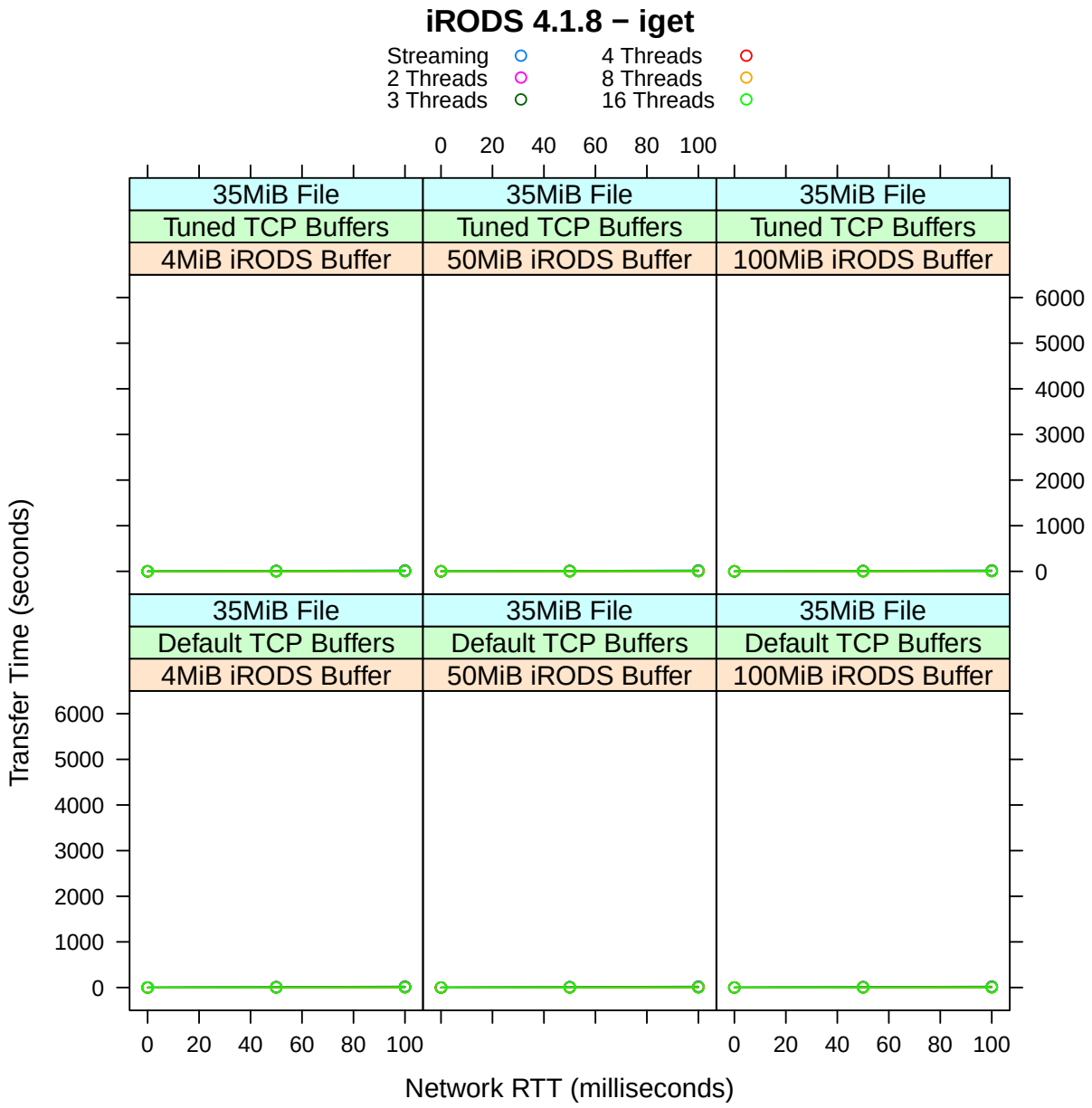

Figure 8: 4.1.8 iget, n=3

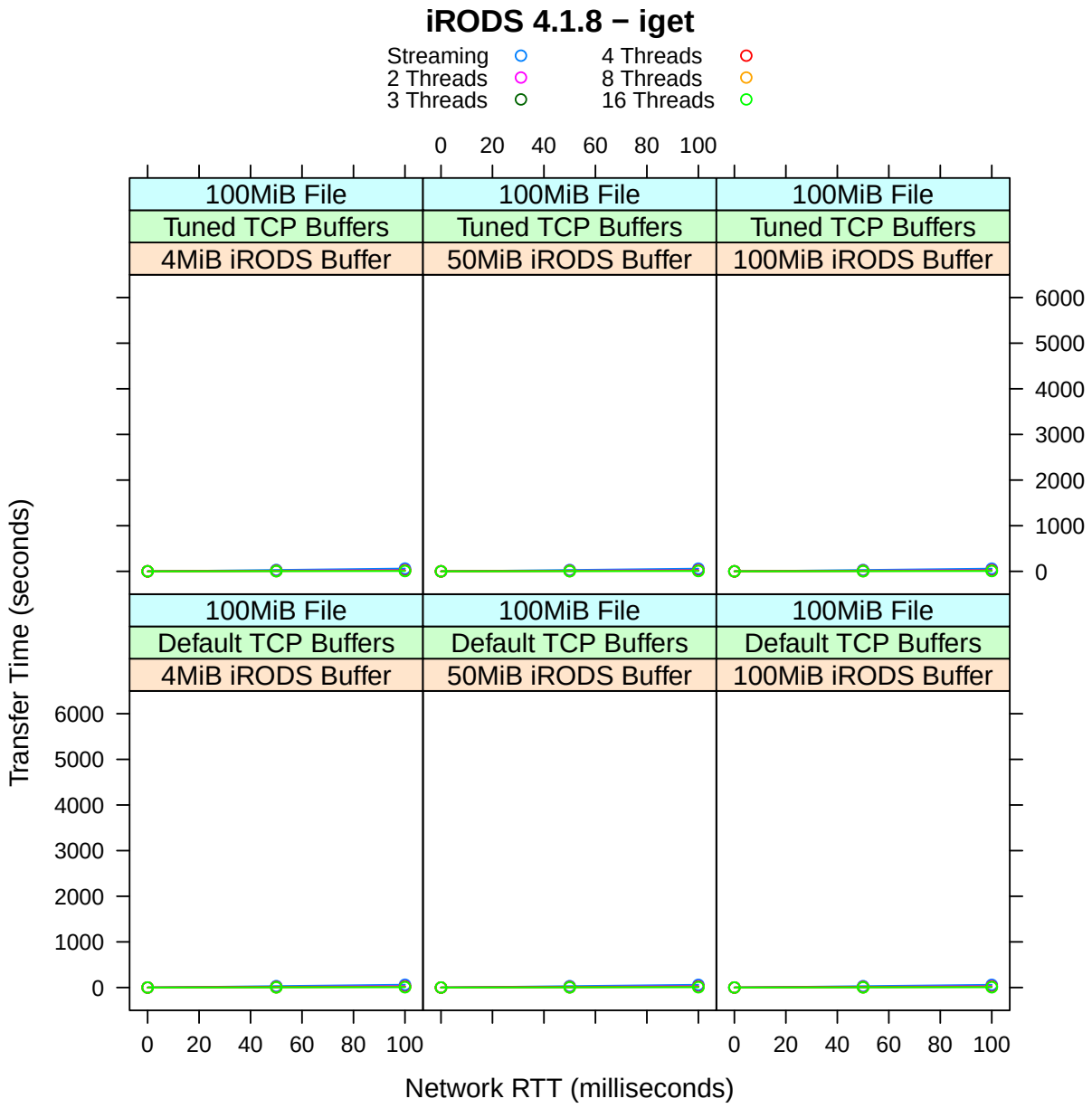

Figure 9: 4.1.8 iget, n=3

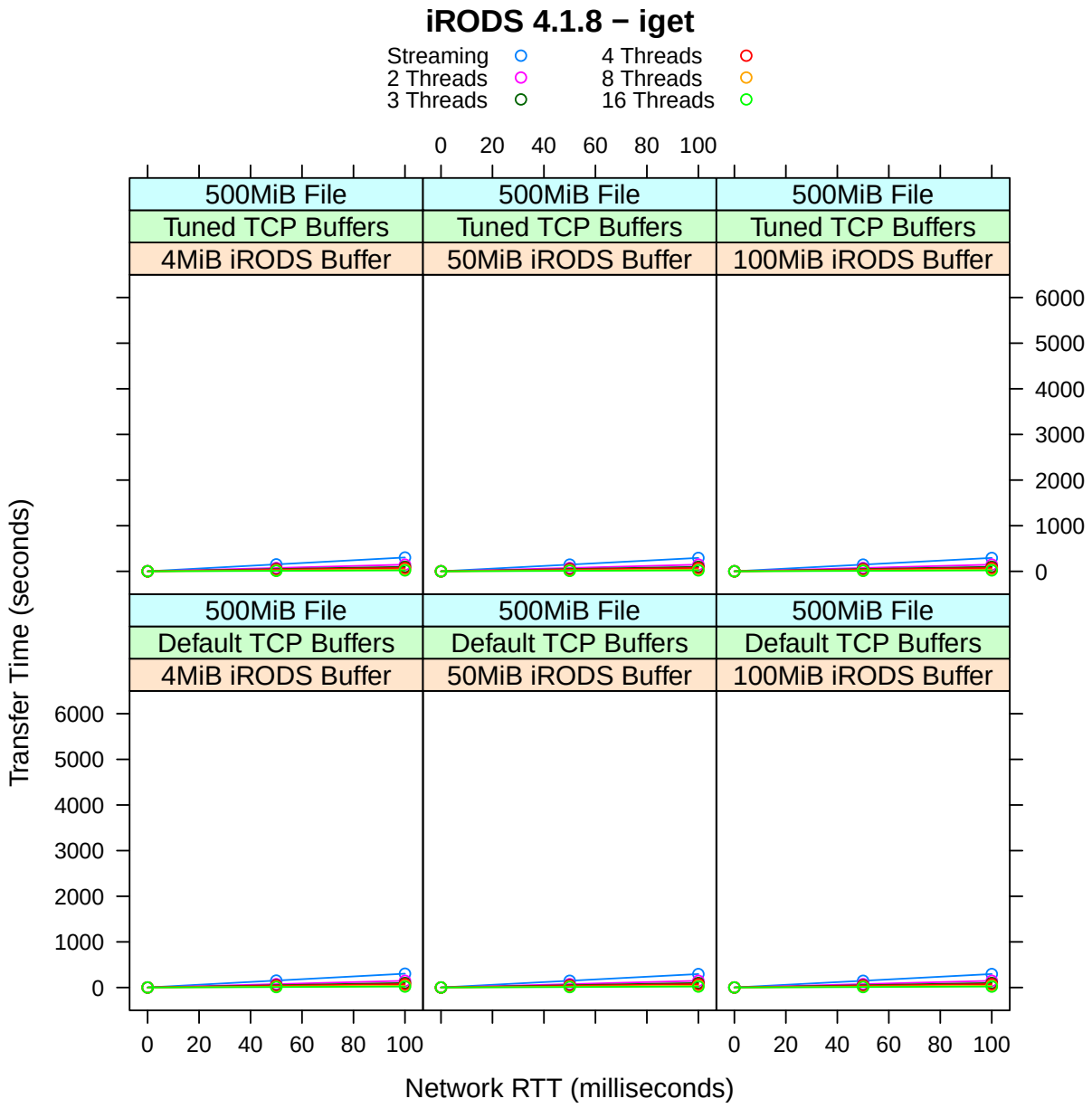

Figure 10: 4.1.8 iget, n=3

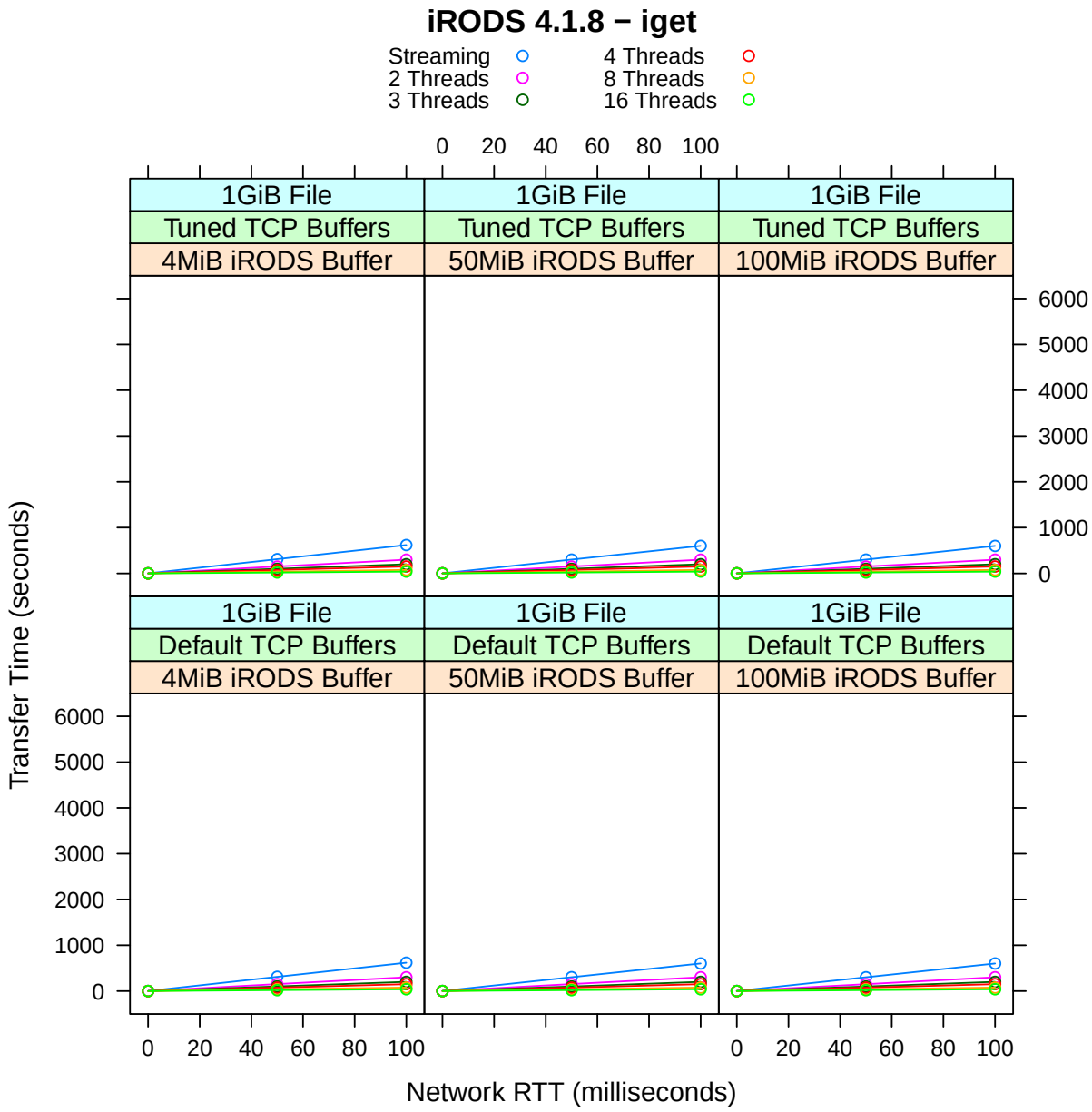

Figure 11: 4.1.8 iget, n=3

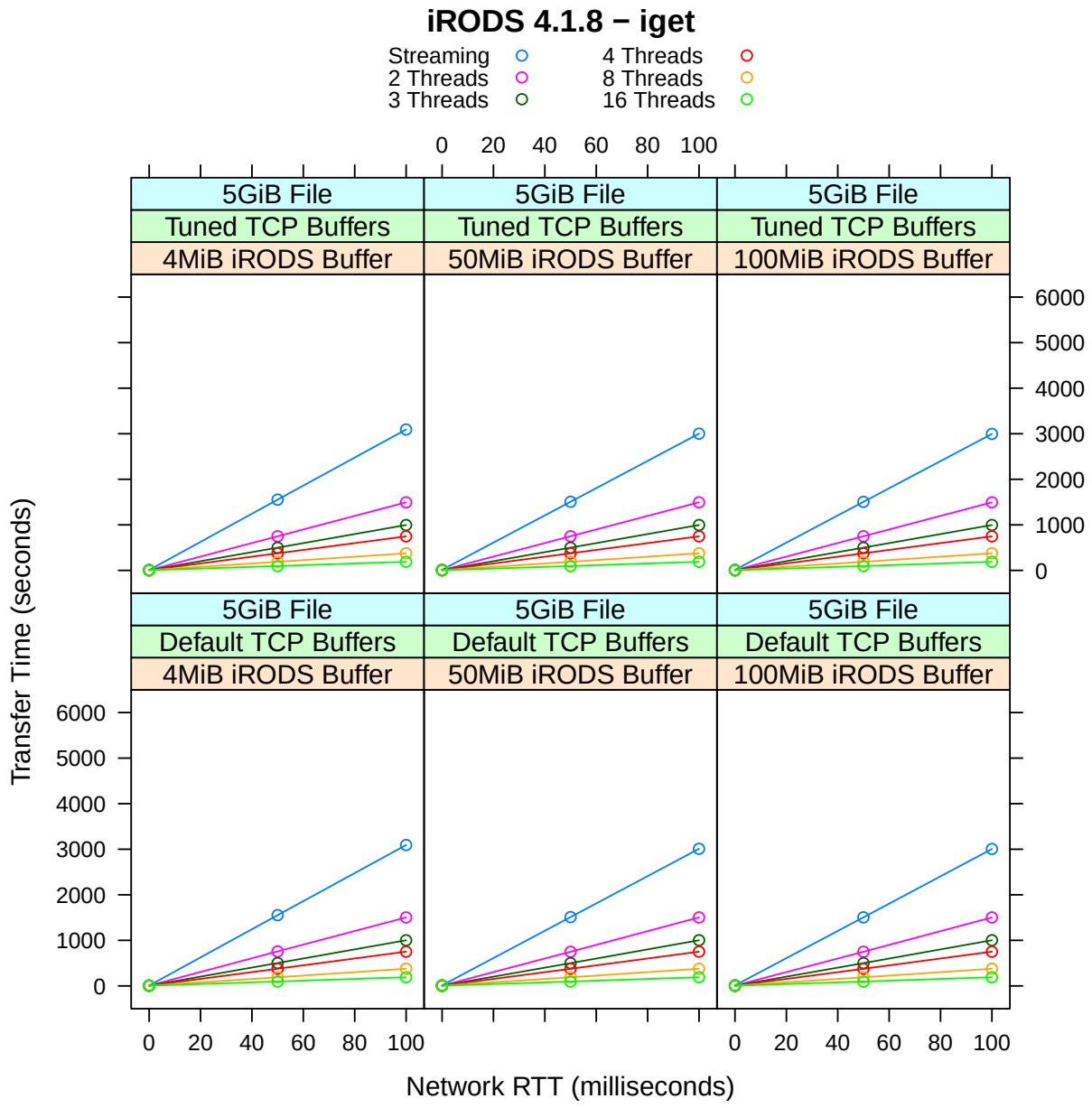

Figure 12: 4.1.8 iget, n=3

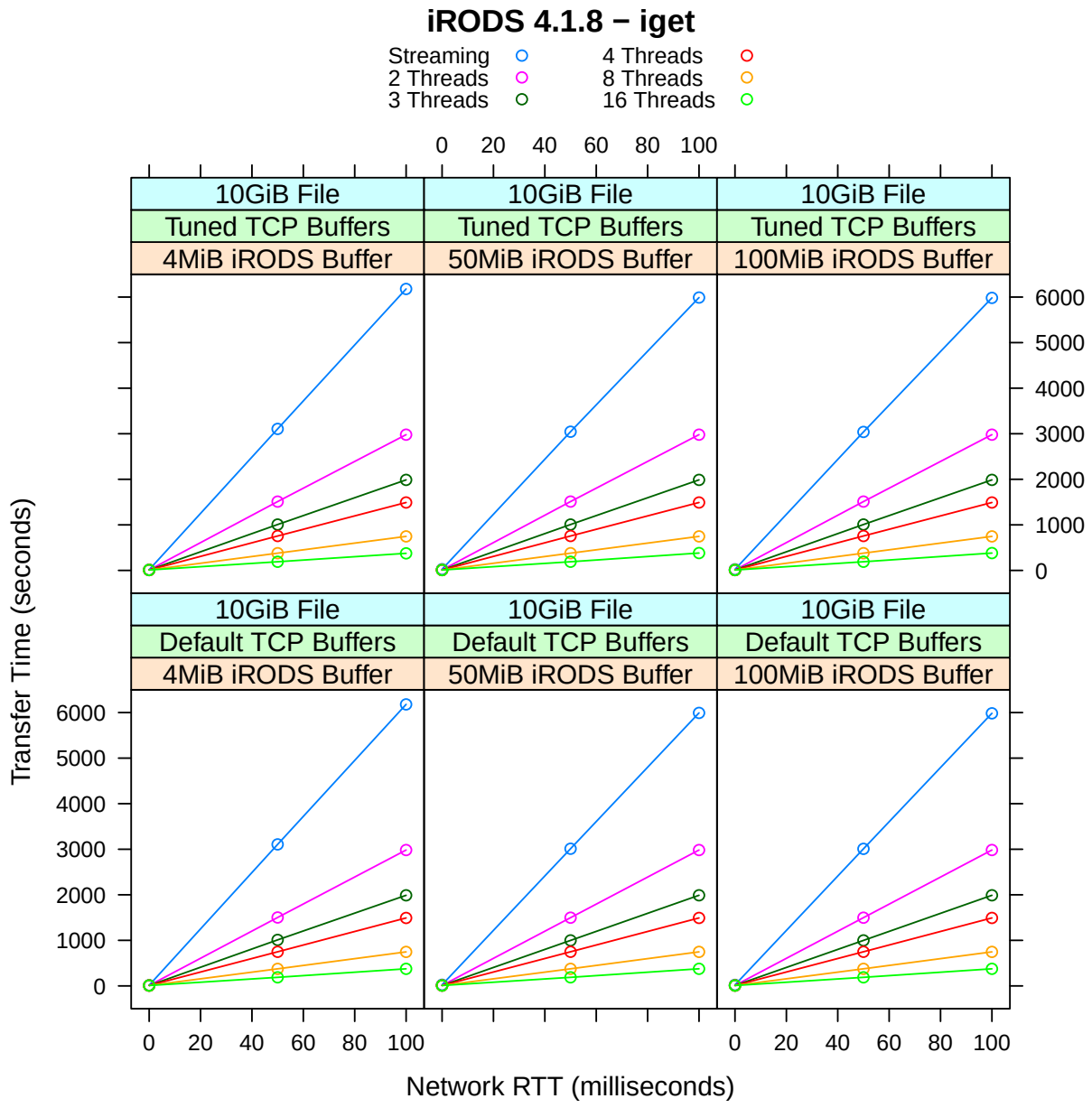

Figure 13: 4.1.8 iget, n=3

5 iRODS 4.1.9

iRODS 4.1.9 presents a significant improvement across the board for higher latency connections.

When the latency is low (˜0ms), iRODS 4.1.9 moves files as quickly as the network will allow.

Similar to 4.1.8, once a latency is added to the connection and the TCP Buffers are at their default settings (bottom row), iRODS can most effectively increase throughput by increasing the number of threads. This shows the well documented knowledge that the default linux kernel TCP settings are not designed for high-bandwidth, high-latency connections. Increasing the iRODS Buffers from 4MiB to 50MiB does reduce the transfer time by up to 30% for the streaming use case as the larger iRODS Buffer fills the available default TCP Buffers and is optimized by the dynamic kernel auto tuning.

After the TCP Buffers are increased (top row), the throughput is increased dramatically. The Tuned TCP Buffers and 4MiB iRODS Buffer panel (top left) illustrates what happens when the iRODS Buffer size is small enough to starve the network and waste the available bandwidth.

For the 500MB and larger file transfers (Figure 16-19 and Figures 22-25), the tuned TCP Buffers panels (top row) present some interesting inversion. When the iRODS Buffer is increased to 50MiB and 100MiB, since the network is fast enough to support the transfer fairly quickly, the additional overhead of managing more threads is greater than just sending the file with fewer threads. When the iRODS Buffer is held to 4MiB (left panel), it remains the limiting factor for the streaming use case and takes the longest time since every buffer is framed by the iRODS protocol and must wait for the TCP back and forth.

The Tuned TCP Buffers and 100MiB iRODS Buffer panel (top right of Figures 14-25) demonstrates the best case scenario and is investigated more thoroughly in Figure 26.

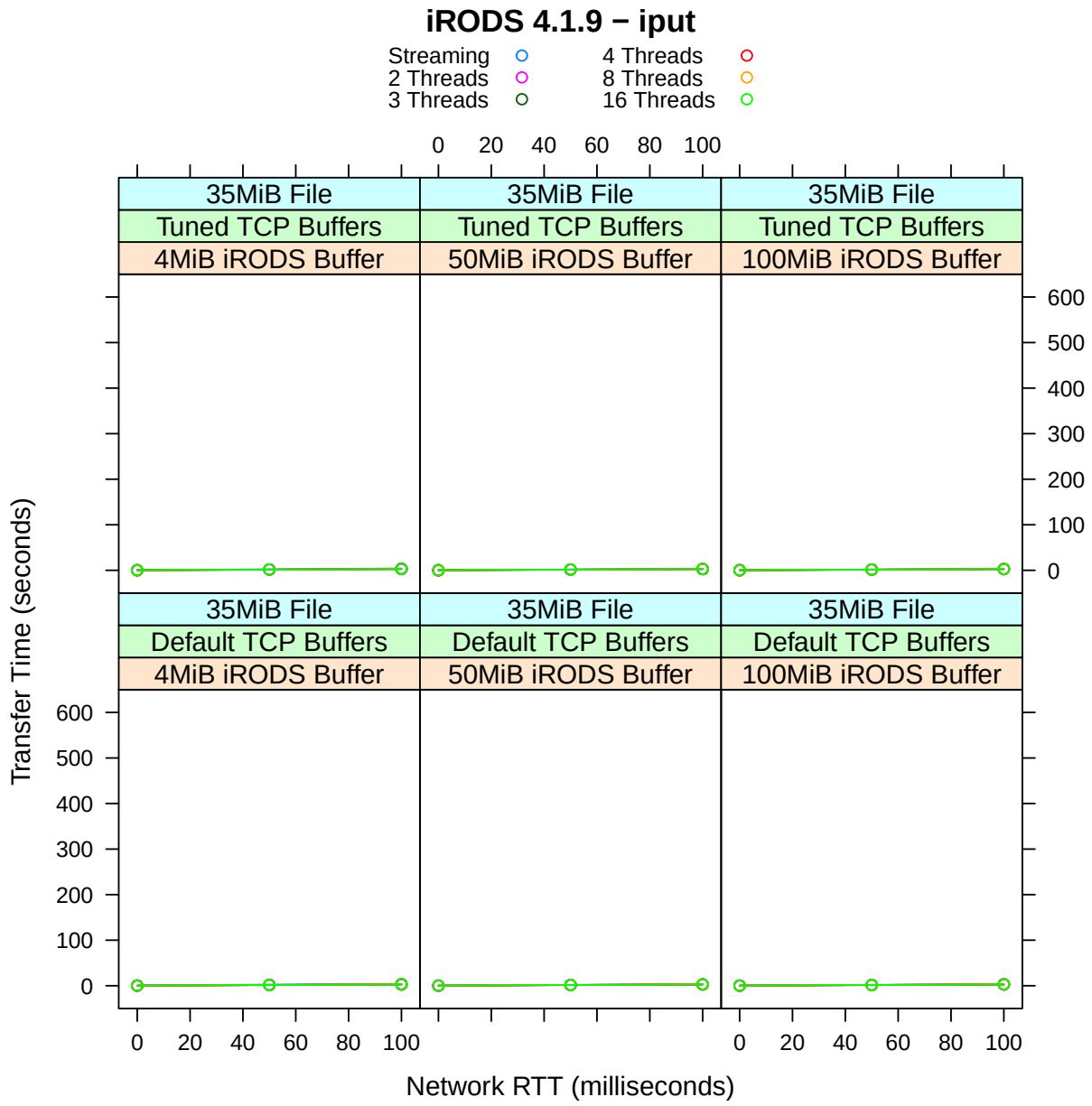

Figure 14: 4.1.9 iput, n=3

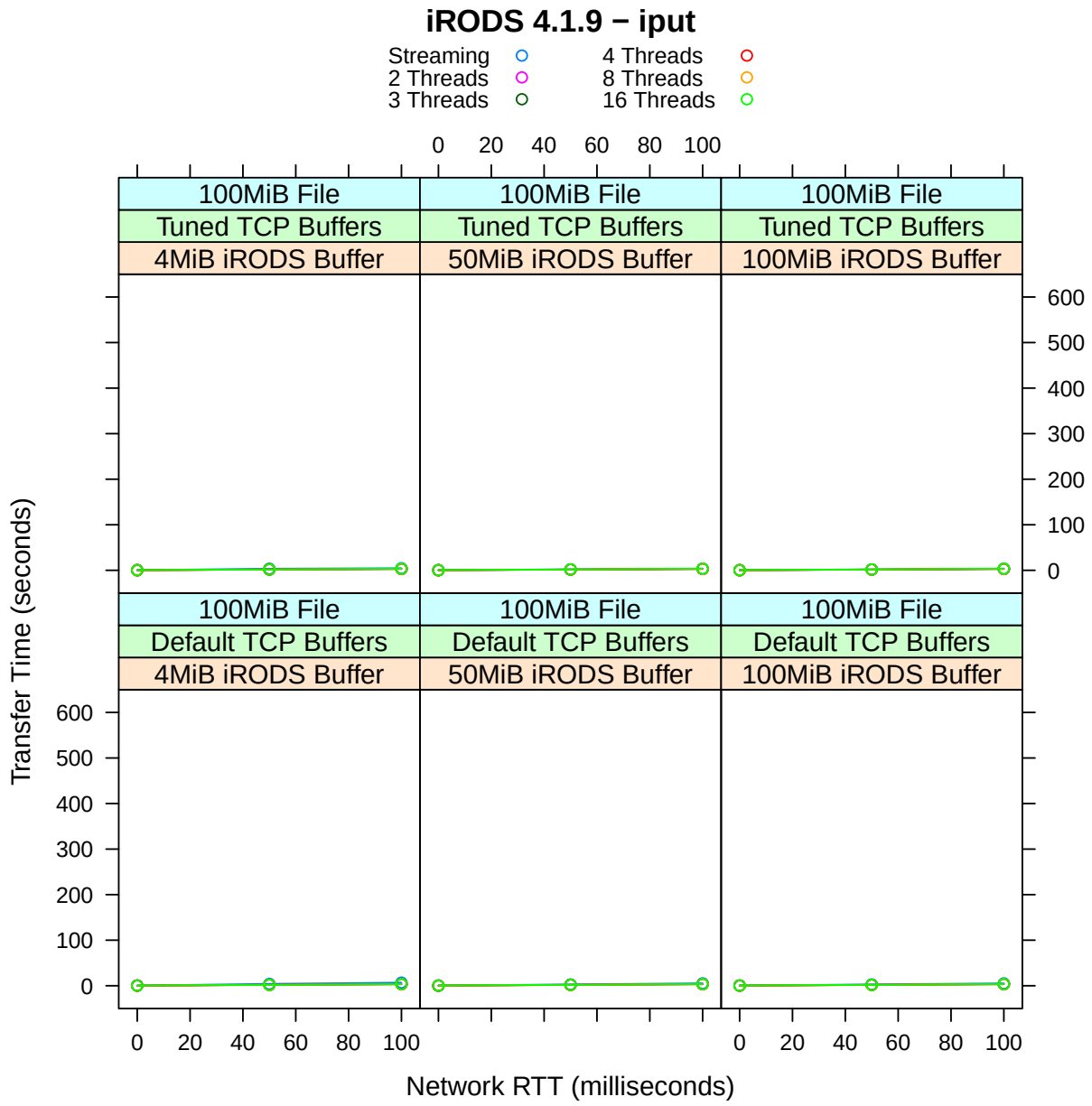

Figure 15: 4.1.9 iput, n=3

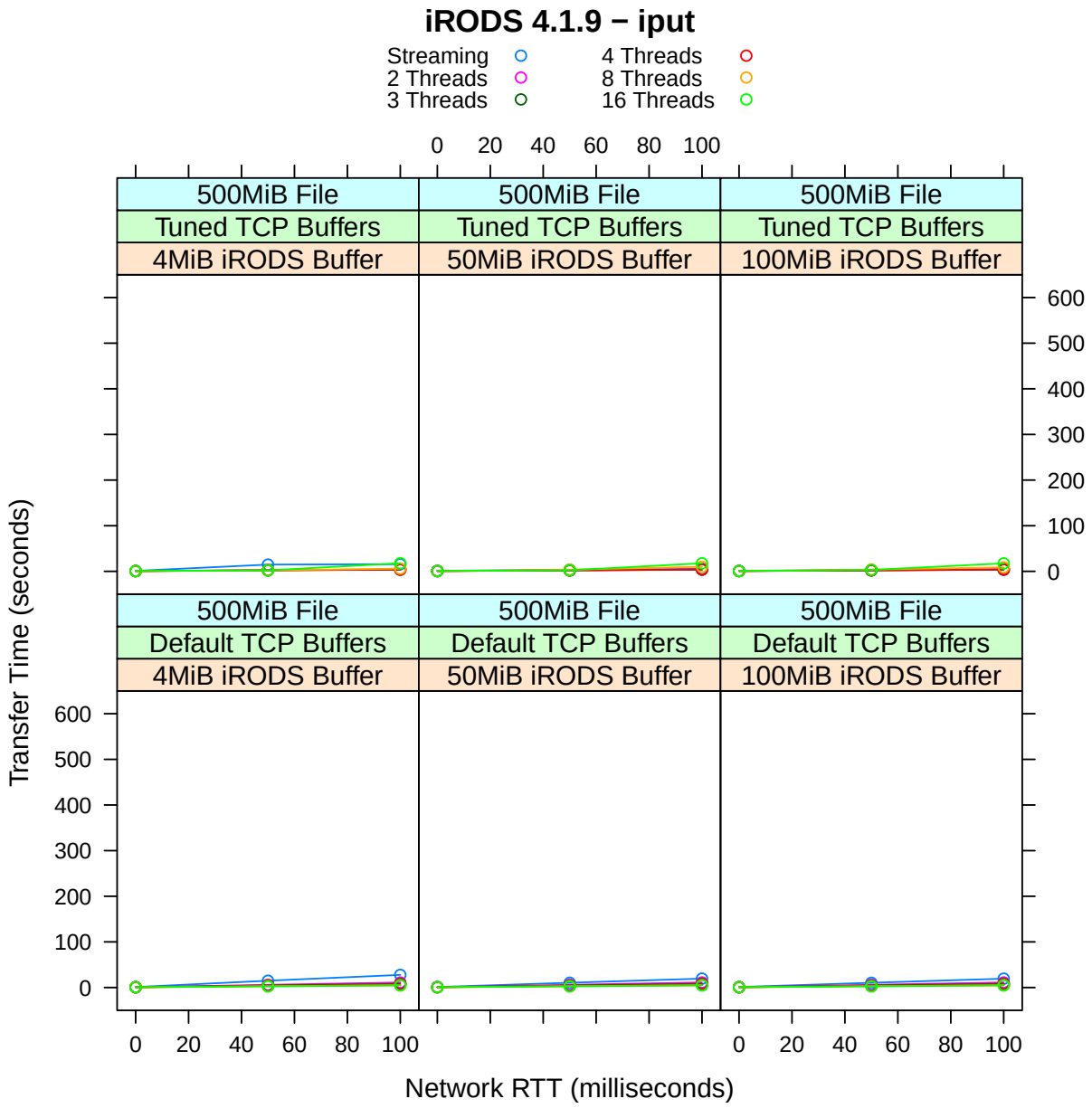

Figure 16: 4.1.9 iput, n=3

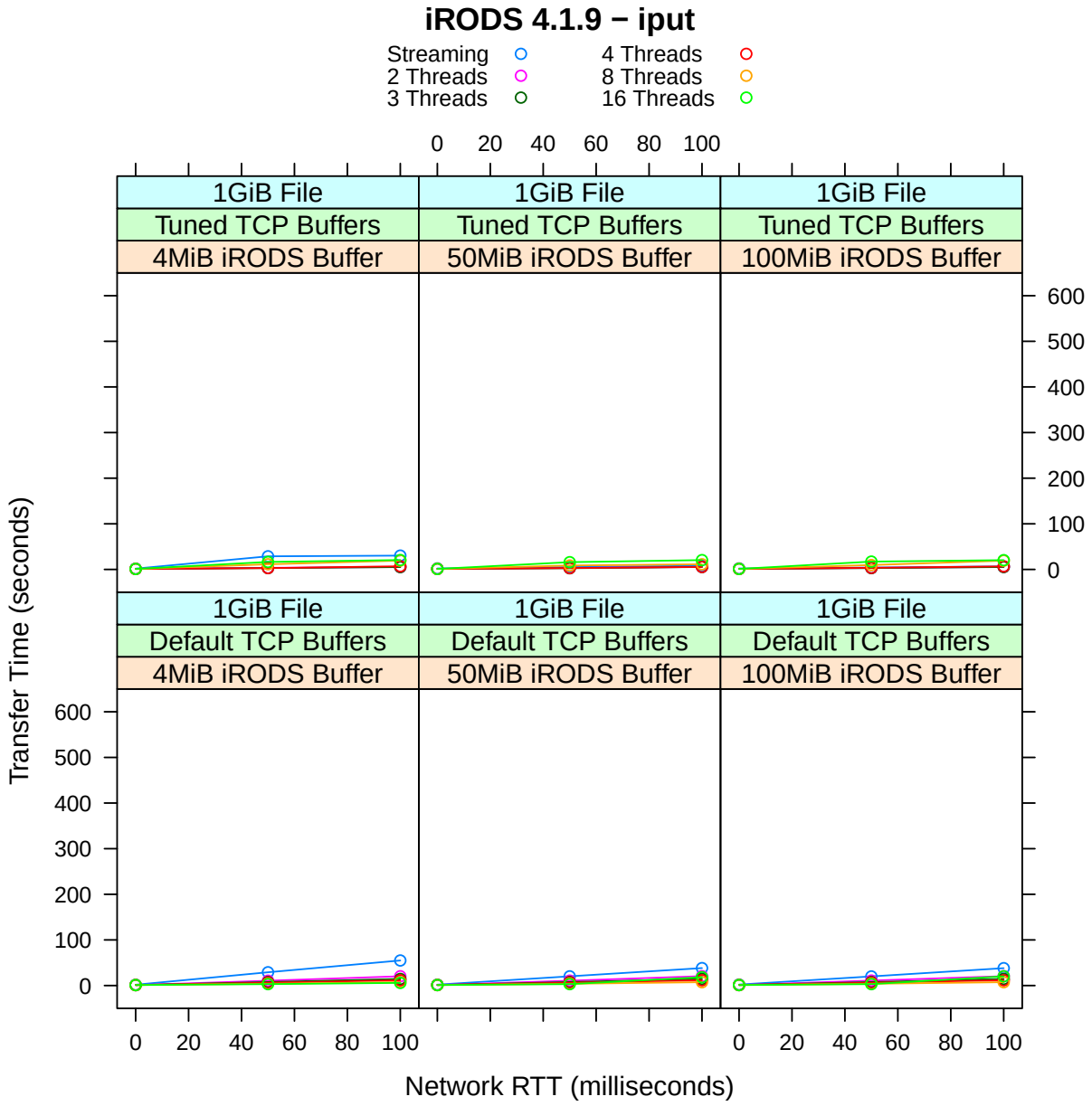

Figure 17: 4.1.9 iput, n=3

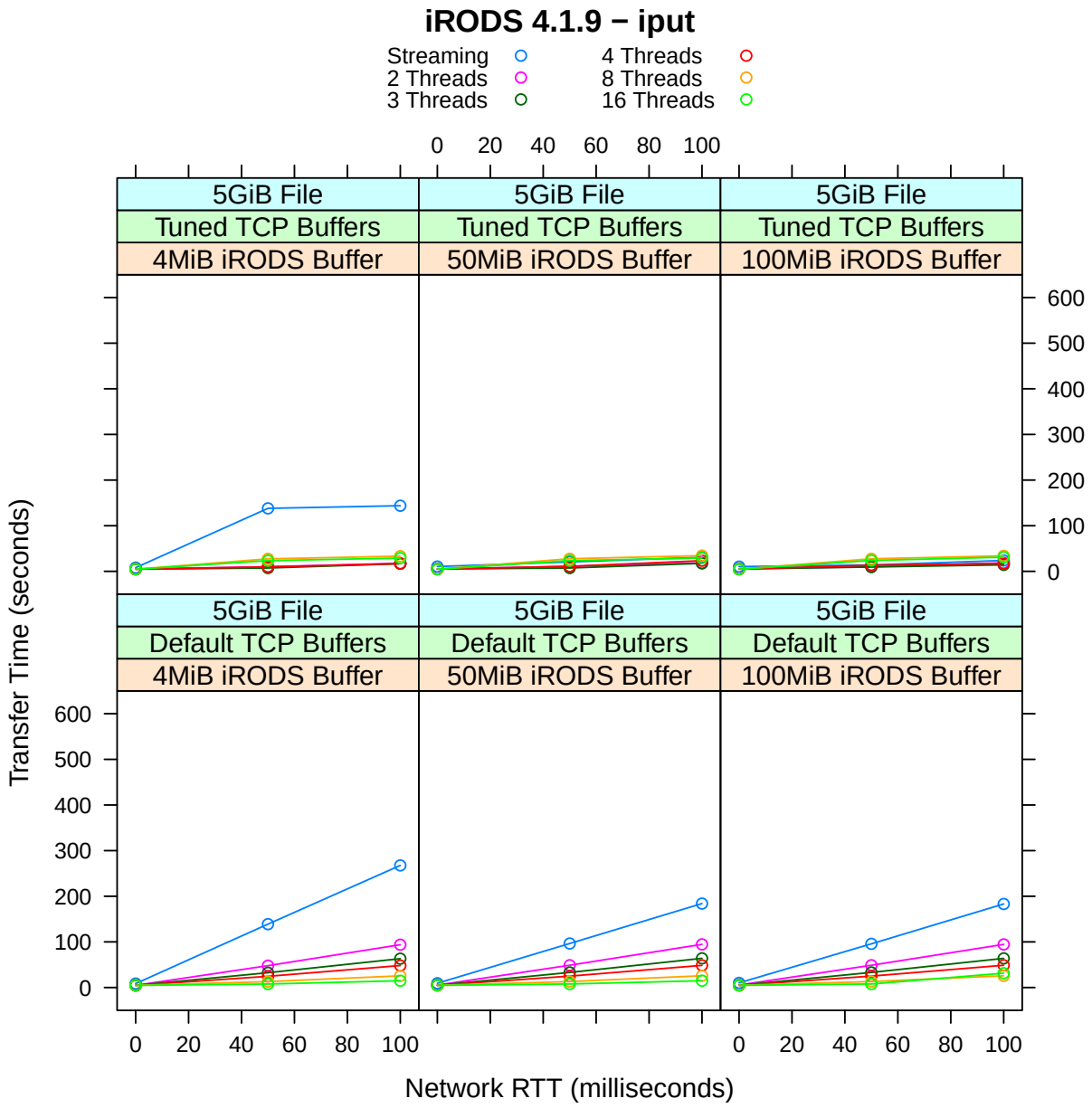

Figure 18: 4.1.9 iput, n=3

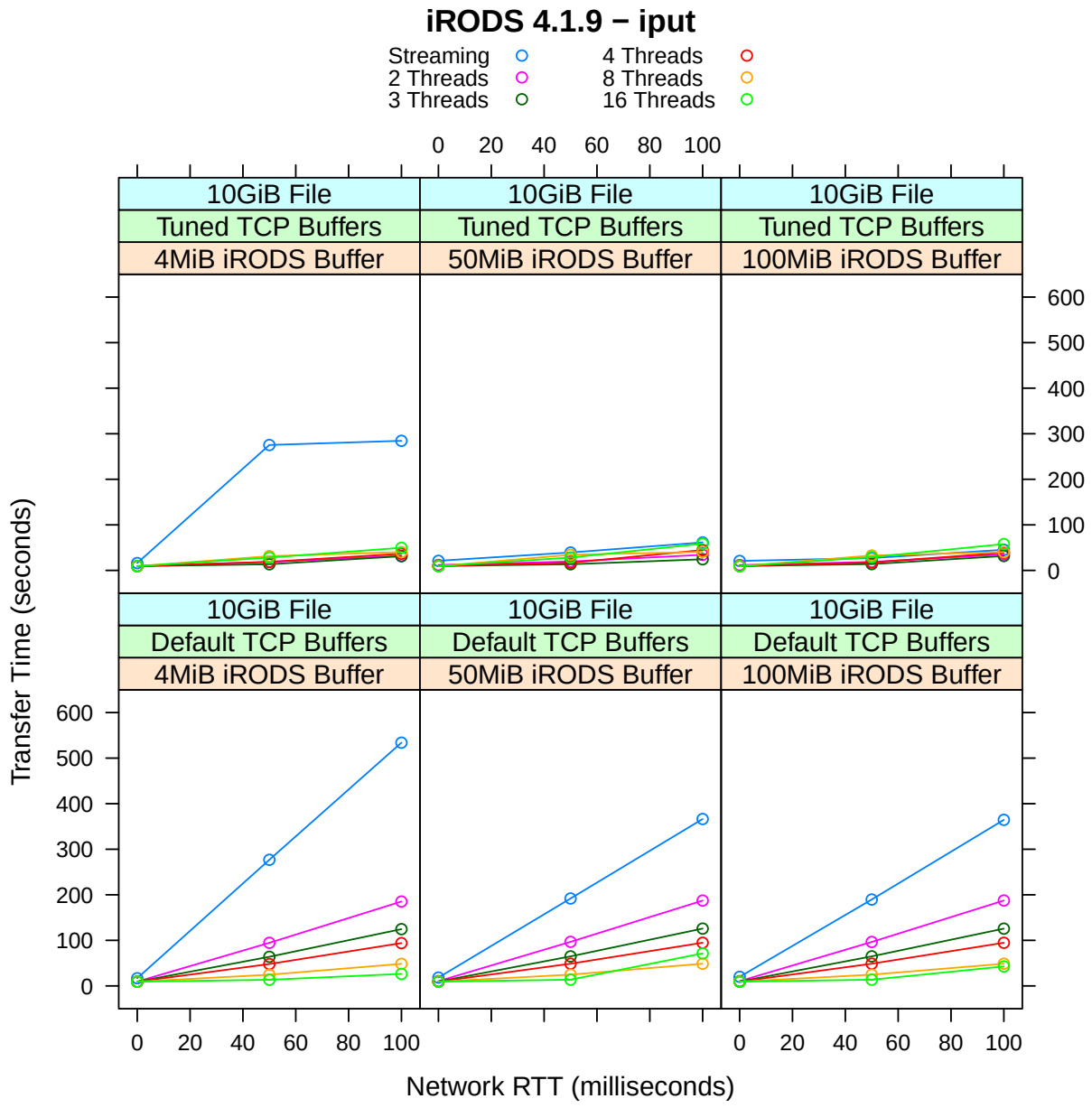

Figure 19: 4.1.9 iput, n=3

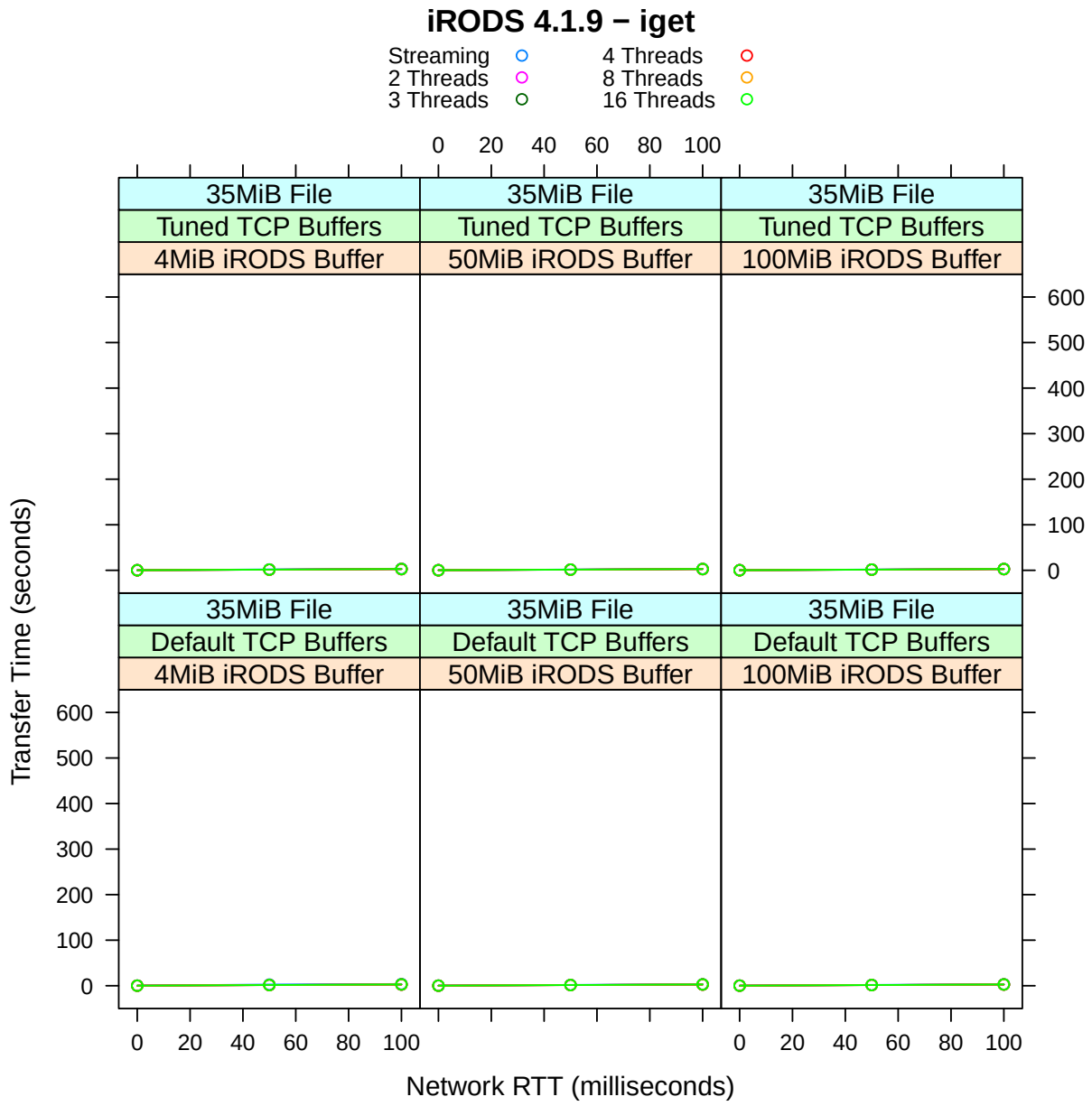

Figure 20: 4.1.9 iget, n=3

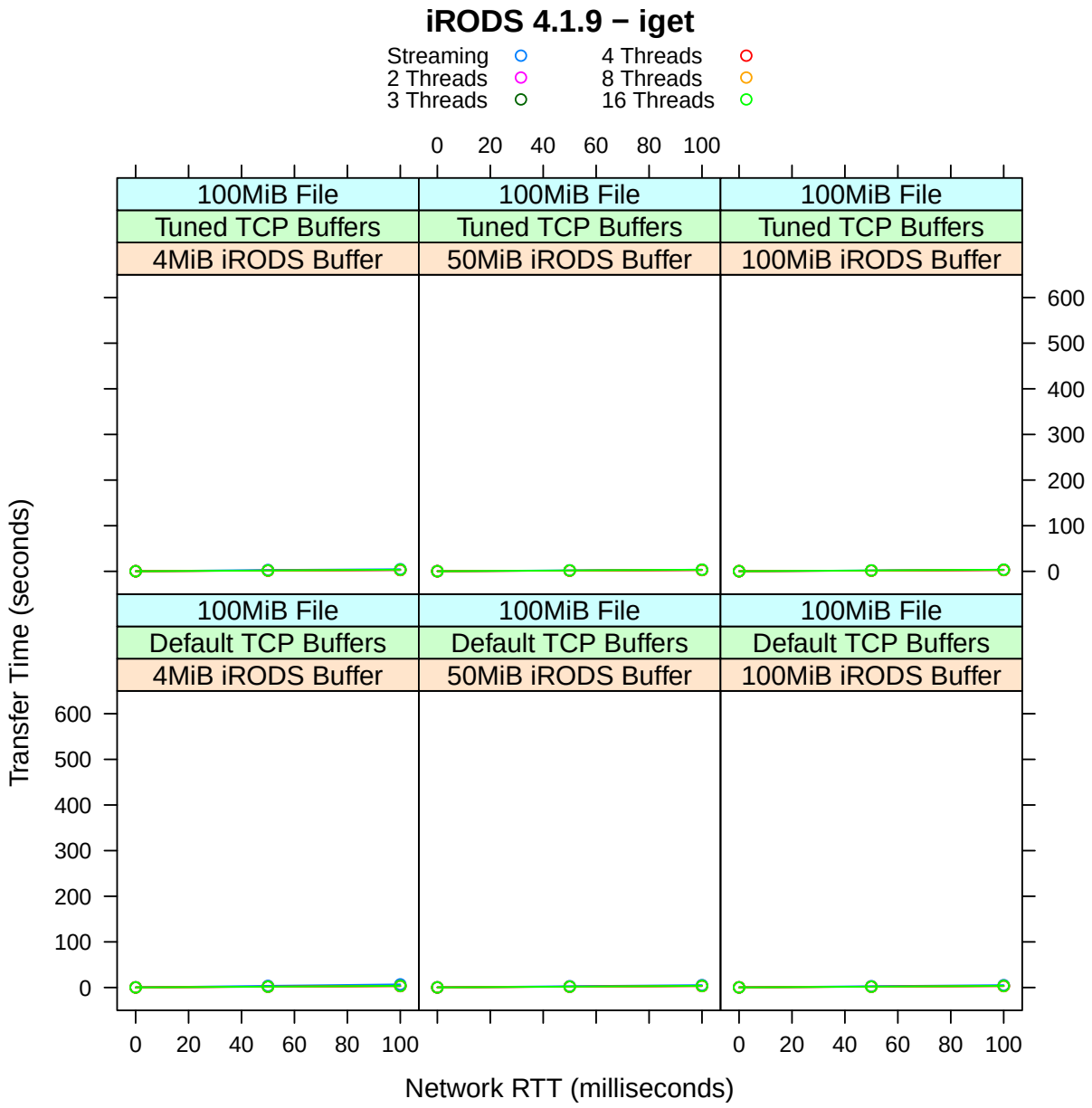

Figure 21: 4.1.9 iget, n=3

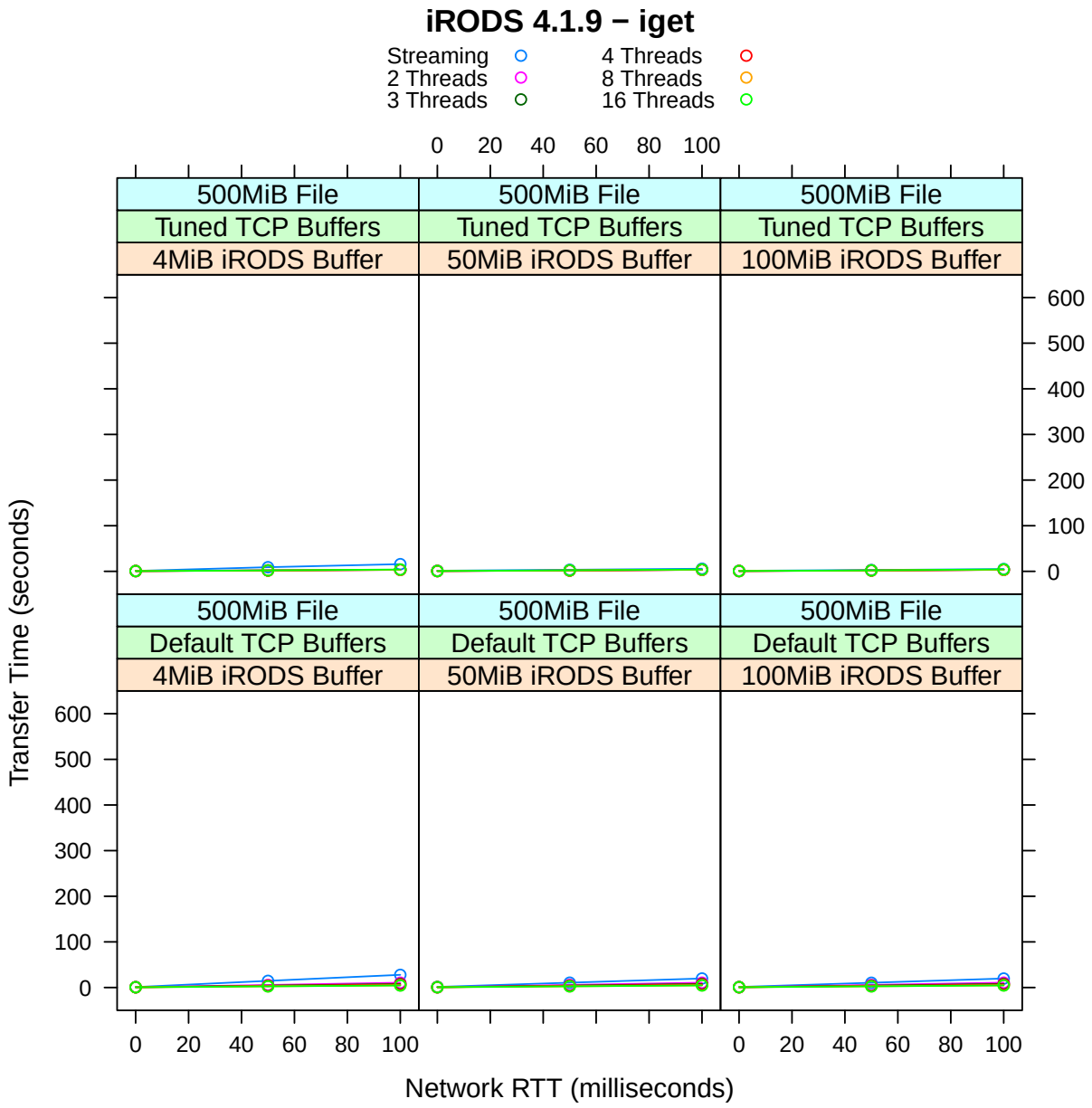

Figure 22: 4.1.9 iget, n=3

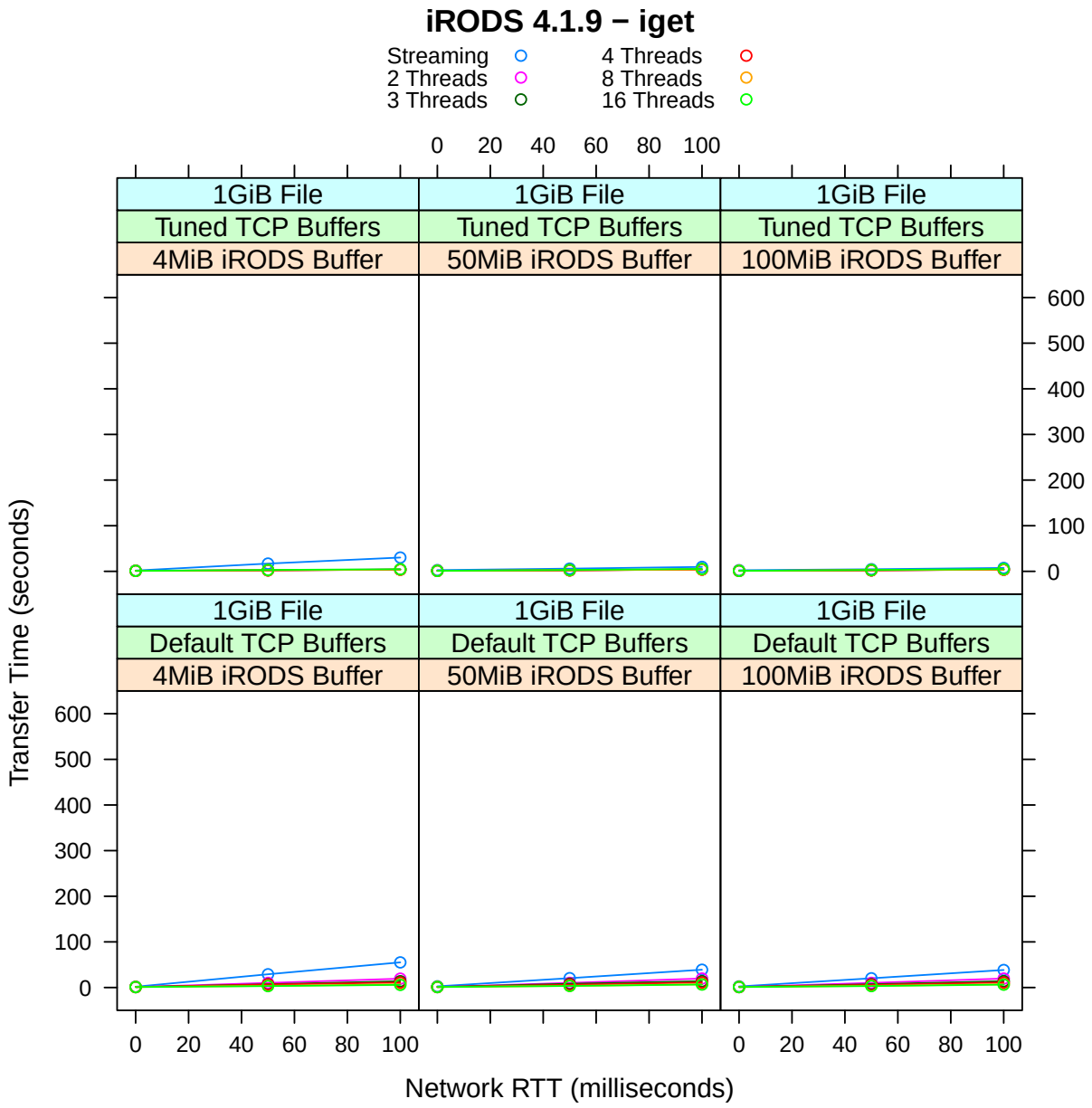

Figure 23: 4.1.9 iget, n=3

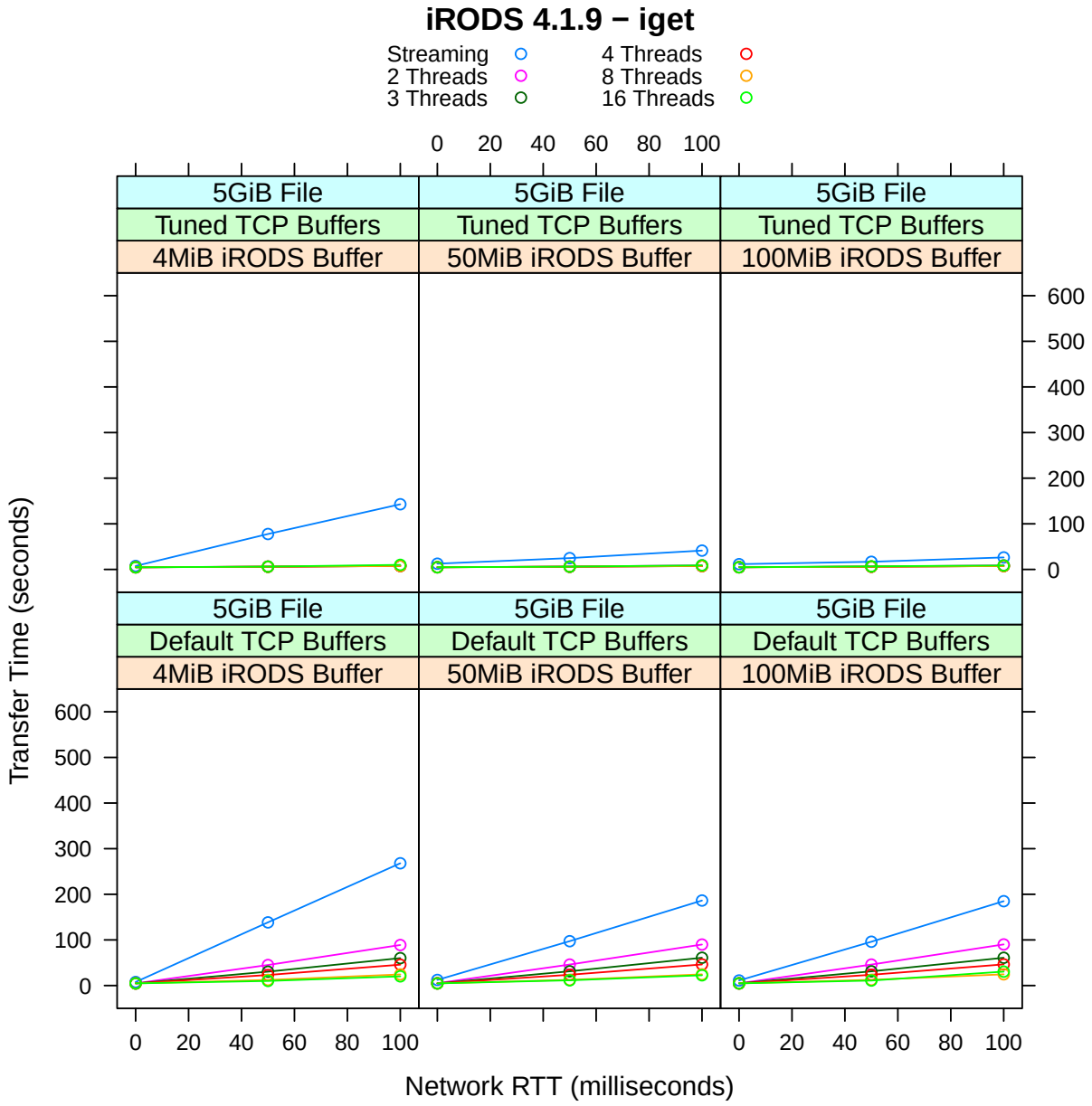

Figure 24: 4.1.9 iget, n=3

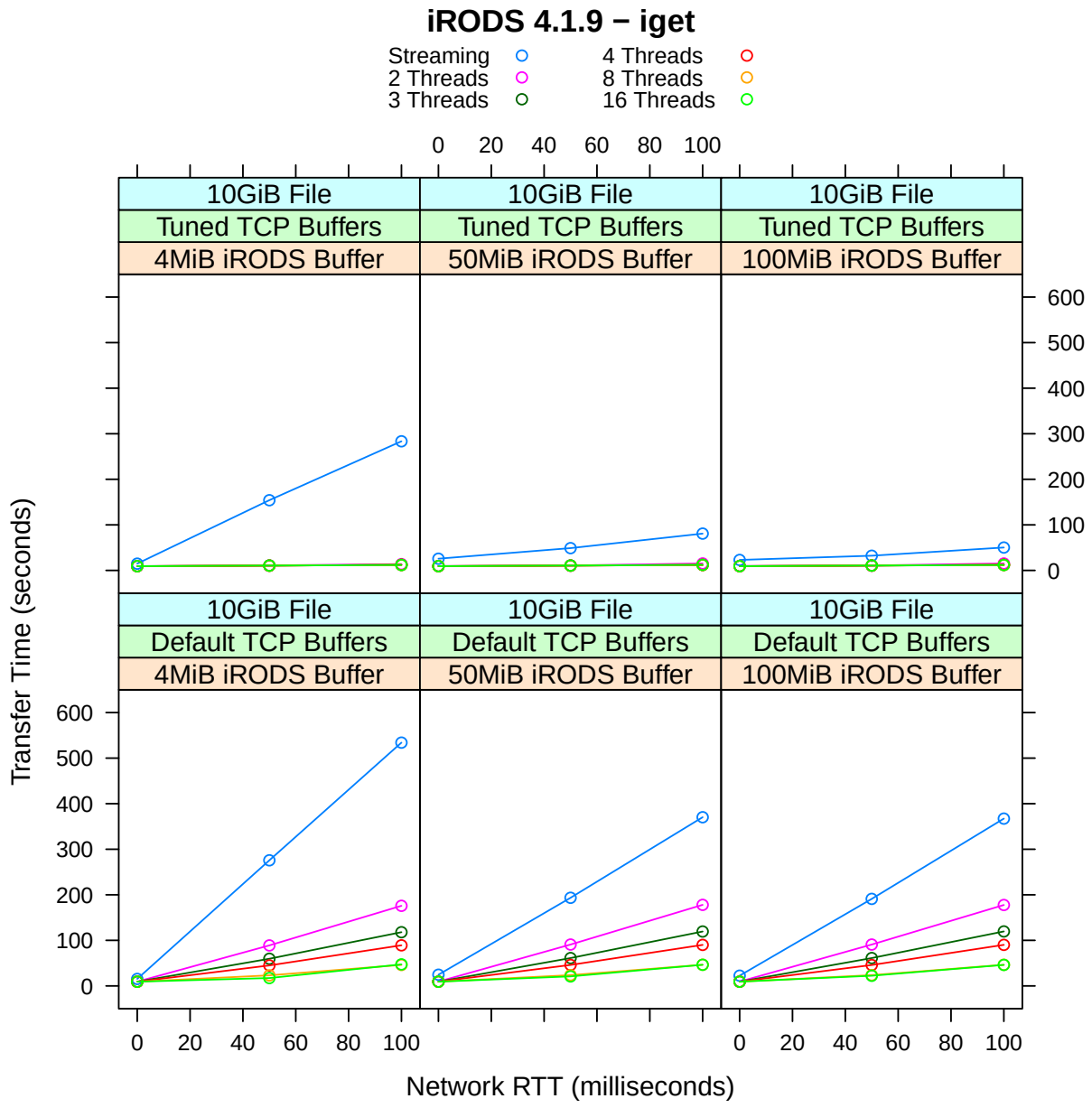

Figure 25: 4.1.9 iget, n=3

6 Best Practice

The most consistently fast way to `iput` and `iget` files via iRODS 4.1.9 is with 3 Threads, the iRODS Buffer set to 100MiB, and Tuned TCP Buffers.

6.1 Threads

Figure 26 investigates the best number of threads to maximize throughput for `iput` and `iget`. The fastest transfer times were most consistently achieved with 3 Threads.

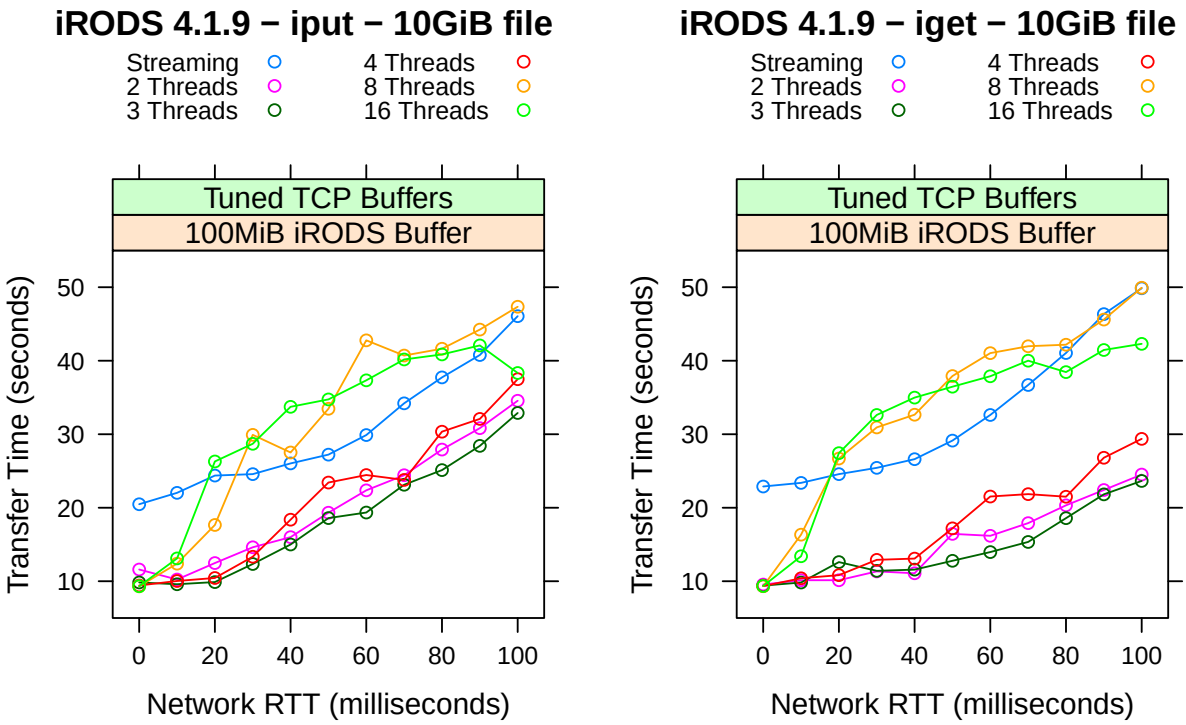

Figure 26: 4.1.9 w/ Tuned TCP Buffers, 100MiB iRODS Buffer, n=10

The partial listing of timings in Table 3 are conservative since they include some small overhead introduced by the python test harness. However, they are applied consistently, so relative assessments remain valid.

	iput						**iget**					
	~0ms delay		50ms delay		100ms delay		~0ms delay		50ms delay		100ms delay	
	seconds	MiB/s	seconds	MiB/s	seconds	MiB/s	seconds	MiB/s	seconds	MiB/s	seconds	MiB/s
Streaming	20.5	500	27.2	376	46.1	222	22.9	447	29.1	351	49.9	205
2 Threads	11.6	883	19.3	530	34.6	296	9.6	1072	16.4	623	24.5	418
3 Threads	9.8	1041	18.6	551	32.9	311	9.4	1089	12.8	801	23.6	433
4 Threads	9.4	1087	23.4	437	37.5	273	9.3	1096	17.2	595	29.4	349
8 Threads	9.3	1099	33.5	306	47.3	216	9.3	1098	37.9	270	49.9	205
16 Threads	9.3	1098	34.7	295	38.3	267	9.4	1095	36.5	281	42.3	242

Table 3: Median 4.1.9 Transfer Times and Maximum Sustained Throughput

6.2 iRODS Buffer Size

Figure 27 investigates the best iRODS Buffer size while using 3 threads (`-N3`).

The iRODS Buffer had little effect on the transfer time (even at 100ms RTT, the variance was under 2 seconds). This agrees with the visual analysis of Figures 14-25 where the top rows were relatively flat and only showed improvement when the iRODS Buffer was increased for Streaming.

Picking 100MiB for the iRODS Buffer size gives strong parallel transfer performance and allows Streaming to improve as much as possible.

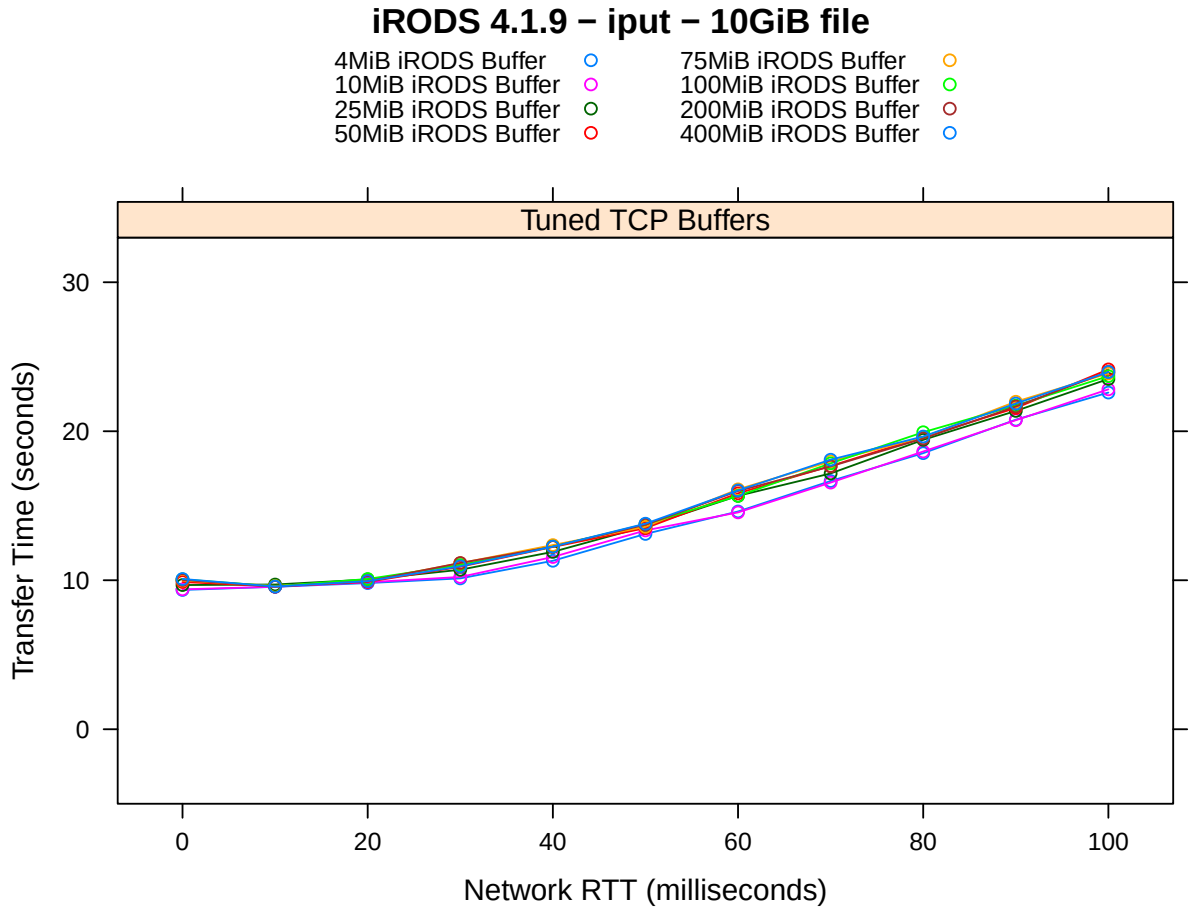

Figure 27: 4.1.9 iput w/ Tuned TCP Buffers and 3 threads, n=10

6.3 TCP Buffer Size

Figures 2-25 show that a larger TCP Buffer Size increases the throughput on high-latency, high-bandwidth connections. There were no cases where a smaller maximum TCP Buffer Size outperformed the larger TCP Buffer Size. With auto tuning in the kernel, the operating system will take advantage of the larger buffers when it can.

7 Comparison

iRODS 4.1.9 presents a significant improvement over 4.1.8.

Both `iput` and `iget` improved and responded similarly under varying conditions.

Tables 4 and 5 compare the best case scenarios tested, Tuned TCP Buffers and 100MiB iRODS Buffer, between 4.1.8 and 4.1.9 (upper right panels of Figures 7 and 19 for `iput` and Figures 13 and 25 for `iget`) over a 10Gbps network connection with a 100ms RTT.

The observed speedup is primarily due to iRODS 4.1.9 no longer setting the TCP send and receive window sizes, allowing TCP auto tuning to handle the window sizes dynamically.

iput	4.1.8		4.1.9		
	seconds	MiB/s	seconds	MiB/s	Speedup
Streaming	5150.7	2	46.1	222	112x
2 Threads	534.9	19	34.6	296	15x
3 Threads	357.6	29	32.9	311	11x
4 Threads	268.8	38	37.5	273	7x
8 Threads	135.8	75	47.3	216	3x
16 Threads	70.3	146	38.3	267	2x

Table 4: 10GiB iput w/ Tuned TCP Buffers, 100MiB iRODS Buffer, 100ms RTT

iget	4.1.8		4.1.9		
	seconds	MiB/s	seconds	MiB/s	Speedup
Streaming	5982.6	2	49.9	205	120x
2 Threads	2979.8	3	24.5	418	122x
3 Threads	1987.2	5	23.6	433	84x
4 Threads	1490.8	7	29.4	349	51x
8 Threads	746.6	14	49.9	205	15x
16 Threads	381.0	27	42.3	242	9x

Table 5: 10GiB iget w/ Tuned TCP Buffers, 100MiB iRODS Buffer, 100ms RTT

8 Conclusion

iRODS 4.1.9 transfers files up to two orders of magnitude (100x) faster than iRODS 4.1.8.

It is recommended for high-bandwidth, high-latency connections:

- to use 3 Threads for maximum throughput during parallel transfer.
- to increase the iRODS Buffer to 100MiB.
- to increase the maximum TCP Buffer Size.

9 Acknowledgements

The authors would like to thank the entire iRODS Consortium Membership as well as the following individuals for ideas, edits, and suggestions:

- Matthew Astley, Wellcome Trust Sanger Institute
- Dale Carder, University of Wisconsin
- John Constable, Wellcome Trust Sanger Institute
- Michael Conway, DICE at UNC-Chapel Hill
- Tony Edgin, CyVerse
- Nirav Merchant, CyVerse
- Reagan Moore, DICE at UNC-Chapel Hill
- Chris Rutledge, RENCI at UNC-Chapel Hill
- Marcin Sliwowski, RENCI at UNC-Chapel Hill
- Ton Smeele, Utrecht University
- Stephen Worth, EMC
- Hao Xu, DICE at UNC-Chapel Hill