

University of Tennessee at Chattanooga

UTC Scholar

Honors Theses

Student Research, Creative Works, and
Publications

5-2021

Simplifying the creation of virtual topologies using MPI Sessions

Tom Herschberg

University of Tennessee at Chattanooga, tom-herschberg@utc.edu

Follow this and additional works at: <https://scholar.utc.edu/honors-theses>



Part of the [Software Engineering Commons](#)

Recommended Citation

Herschberg, Tom, "Simplifying the creation of virtual topologies using MPI Sessions" (2021). *Honors Theses*.

This Theses is brought to you for free and open access by the Student Research, Creative Works, and Publications at UTC Scholar. It has been accepted for inclusion in Honors Theses by an authorized administrator of UTC Scholar. For more information, please contact scholar@utc.edu.

Simplifying the Creation of Virtual Topologies

Using MPI Sessions

by

Tom Herschberg

Departmental Honors Thesis

The University of Tennessee at Chattanooga

Department of Computer Science

Examination Date: April 8th, 2021

Anthony Skjellum
Professor of Computer Science
(Chair)

Eleni Panagiotou
Assistant Professor of Mathematics
(Committee Member)

ABSTRACT

As supercomputers have approached exascale performance, several scalability issues have emerged within MPI. These issues arise because MPI includes all processes in the World model, which consumes unacceptable amounts of time and resources at large scale. The Sessions model was developed to combat these issues by removing the requirement of `MPI_COMM_WORLD`, which provides a more scalable method of creating communication groups in large jobs. Additionally, the Sessions model enables the creation of virtual topologies directly from sets of processes allocated to the execution of a parallel application rather than building virtual topologies from an existing communication group such as `MPI_COMM_WORLD`.

For this project, I implemented the Sessions model in ExaMPI, an MPI implementation designed for modularity, extensibility, and understandability. I also created topological variations of several common communication algorithms and topological connection building to further take advantage of the benefits of the Sessions model. I found that using the Sessions model reduces the time and resources used when a large parallel application begins executing. Additionally, I found that using topological connection building and topological communication algorithms is faster than traditional all-to-all connection building in certain situations.

ACKNOWLEDGMENTS

First, I thank my thesis director Dr. Tony Skjellum for his continued mentorship and expertise throughout my time as a researcher at UTC. I also thank Dr. Eleni Panagiotou for providing many of the research skills I needed to complete this thesis. I owe many thanks to Derek Schafer, whose knowledge of C++ and ExaMPI made this thesis possible. Finally, I owe gratitude to Dr. Howard Pritchard for his advice on implementation details and testing.

I acknowledge funding received from the Research Experience for Undergraduates (REU) program through supplements to NSF funding at the University of Tennessee at Chattanooga, under grants # 1822191, 1821926, 1821431, and DMS-1913180.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGMENTS	iii
1. INTRODUCTION	1
1.1 Outline	2
2. BACKGROUND AND MOTIVATION	3
2.1 Background	3
2.1.1 Initialization	3
2.1.2 Communicator and Group Creation	4
2.2 Motivation	5
3. IMPLEMENTATION	6
3.1 Implementing the MPI Sessions API	6
3.2 Changes to the Runtime System	7
3.3 Dynamic Initialization	8
3.4 Topological Connection Building	8
3.5 Topological Communication Algorithms	9
4. EVALUATION	11
4.1 Experimental Setup	11
4.2 Results	11
4.2.1 Initialization Time	12
4.2.2 MPI Function Execution Time	14
5. CONCLUSION	19
5.1 Discussion of Results	19
5.2 Limitations	20
5.3 Future Work	21

REFERENCES	23
APPENDICES	
A. GRAPHS	25
B. BENCHMARKS	63

CHAPTER 1

INTRODUCTION

The Message Passing Interface (MPI) is currently the *de facto* standard for communication between peer groups of processes in a parallel program [7]. MPI's governing body, the MPI Forum, works diligently to ensure that every new version of MPI is backwards compatible to prevent older applications from breaking after an update. This backwards compatibility requirement, however, has contributed to numerous scalability problems as the high performance computing industry has evolved. As supercomputers continue to approach exascale (one quintillion calculations per second), it is becoming increasingly important to resolve the scalability issues facing MPI. Specifically, one of the largest problems currently facing MPI is handling the massive process spaces of the largest supercomputers. It is not uncommon for some applications to use millions of processes during execution. This becomes difficult for MPI to handle because, when MPI is initialized, every single process must be initialized to a single communicator called `MPI_COMM_WORLD`, which enables communication between each node. Because the process space is so large, creating `MPI_COMM_WORLD` is often unacceptably time- and resource-intensive. What is more, communication has substantial structure, so that all processes do not communicate with each other, but rather only in sparse, topological subsets/subgraphs that are application, algorithm, and collective-communication dependent.

One recently proposed solution to this problem is the concept of MPI Sessions [6]. MPI Sessions removes the requirement of initializing all processes to `MPI_COMM_WORLD` by allowing the creation of communicators using different groups of processes provided to the application by the runtime system. This enables the creation of multiple MPI environments, each of which can be customized and optimized to a much finer degree than previously possible. Because

it relaxes the requirement for global initialization, MPI Sessions can enable the use of sparser connection building. Building connections in the shape of a sparse virtual topology, such as a ring, uses far less resources than building connections between all processes in a job and is much more scalable than the current method of building connections. By removing the requirement for a global communicator and connections between all processes, MPI Sessions makes the process of creating efficient, scalable communicators much more straightforward. The purpose of this work is to show the benefits of leveraging the relaxed global requirements of MPI Sessions to create scalable, sparse communication patterns within MPI applications.

1.1 Outline

The remainder of this thesis is structured as follows: Chapter 2 provides background on the Sessions model, as well as the motivation for this project. Chapter 3 describes the modifications made to ExaMPI in order to support the Sessions model and topological communication patterns. Chapter 4 presents performance results for the Sessions model compared to the traditional MPI.COMM.WORLD model. Finally, Chapter 5 concludes the project and outlines future work.

CHAPTER 2

BACKGROUND AND MOTIVATION

2.1 Background

The MPI Sessions model provides several API additions that address some of the emergent problems in the current MPI Standard. To aid in understanding the concepts present later in this thesis, the key API additions are outlined in the following subsections. For simplicity, the current state of the MPI API will be referred to as the *World model*, while the additions to the API will be referred to as the *Sessions model*.¹

2.1.1 Initialization

One of the largest differences between the World model and the Sessions model is how they are initialized. In the World model, each process must initialize the MPI library by calling the function *MPI_Init()* and finalize the MPI library by calling the function *MPI_Finalize()* exactly once. Additionally, all other MPI functions must be called after *MPI_Init()* and before *MPI_Finalize()*. The World model does not provide a mechanism for reinitializing MPI once *MPI_Finalize()* has been called. When *MPI_Init()* is called, every process is added to a communication object—a communicator—called *MPI_COMM_WORLD*, which can require massive amounts of memory if there is a large number of processes (a world group). The Sessions model addresses this issue by eliminating the creation of global state upon initialization. Instead, it uses a local handle to the MPI library called an *MPI_Session*. Once a process creates an *MPI_Session* using the *MPI_Session_init()* function, it uses that *MPI_Session* as

¹This nomenclature has become common parlance in the MPI Forum.

an independent handle to the MPI library in order to call other MPI functions. Unlike the World model, communicator creation is not done upon initialization, which reduces startup overhead. Once a process is finished with a given `MPI_Session`, it will destroy the session using `MPI_Session_finalize()`. Because each `MPI_Session` is an independent local handle to the MPI library, a process can have several `MPI_Sessions` active at the same time, which is impossible in the World model. The Session model also allows for `MPI_Session_init()` to be called at any point in an application's execution, allowing for `MPI_Session` objects to be created even after all other `MPI_Session` objects have been finalized and destroyed.

2.1.2 *Communicator and Group Creation*

The MPI Standard outlines two objects that are designed to group processes together. An `MPI_Group` is an ordered set of processes that each have a unique rank. This association is purely local and the creation of `MPI_Groups` does not require any communication between different processes. An `MPI_Comm`, on the other hand, is an object created collectively between all members of a given `MPI_Group` that facilitates communication between the members of that `MPI_Group`. Any communication between processes in an MPI application must be done using an `MPI_Comm`, a communicator. This is the reason for the creation of `MPI_COMM_WORLD`, which is an `MPI_Comm`, upon initialization in the World model. Because `MPI_COMM_WORLD` contains all processes, any process can easily communicate with any other process and build smaller `MPI_Comms` if desired. However, because creating an `MPI_Comm` requires a collective operation (non-local and synchronizing) on all processes in that communicator, they are relatively expensive to create and store at large scale. The Sessions model addresses this by changing the process of creating `MPI_Comms`. Rather than creating `MPI_COMM_WORLD` first and then building down like in the World model, the Sessions model builds up from an `MPI_Group` to an `MPI_Comm`. To do this, the Sessions model introduces the concept of process sets, which are ordered sets of processes similar to `MPI_Groups` but are discovered by querying the underlying runtime system. Each process can query the runtime system for the number of available process sets using the `MPI_Session_get_num_psets()` function, then get the name of the n th process set by calling the

MPI_Session_get_nth_pset() function. Process set names follow the Uniform Resource Identifier (URI) format. There are two process sets that must always be available: `mpi://WORLD`, which is an ordered set of all processes, and `mpi://SELF`. Once the name of a process set has been obtained, the *MPI_Group_from_session_pset()* function can be used to create an `MPI_Group` that matches the process set that was passed into the function. From there, the *MPI_Comm_create_from_group()* function can be used to create an `MPI_Comm` for the given `MPI_Group`. By building up from process sets rather than building down from `MPI_COMM_WORLD`, the Sessions model is able to avoid the overhead associated with creating a very large `MPI_Comm` when such an `MPI_Comm` is not required for the application. If `MPI_COMM_WORLD` is required, the Sessions model can still create it using the `mpi://WORLD` process set, making it compatible with the World model.

2.2 Motivation

While the benefits of MPI Sessions have been theorized, there is currently little opportunity to experiment and gather results using the Sessions model. No MPI middleware has implemented the MPI Sessions API, and only one, Open MPI, has a functioning prototype [5]. However, Open MPI is a production-grade middleware product, and has been optimized to such a degree that it is difficult to introduce new concepts without requiring significant changes to the code base. For this reason, MPI Sessions has not yet been proven to provide any performance benefits. In order to demonstrate the value of MPI Sessions, it needs to be implemented in a readable and extendable way. This would allow for the functionalities of MPI Sessions to be taken advantage of by other concepts designed to increase the performance of MPI applications. For example, removing the requirement of a global communicator enables the use of sparser virtual topologies from the beginning of an application, which reduces startup overhead. Such performance benefits are impossible to measure until MPI Sessions can be leveraged by other performance-saving concepts. Therefore, to observe the value of MPI Sessions, it needed to be implemented in a way that facilitated experimentation. The purpose of this work is to provide such an implementation of the MPI Sessions API, and to augment it with topological connection building and communication algorithms to improve the performance of MPI applications.

CHAPTER 3

IMPLEMENTATION

When doing MPI research, it is important to select the right MPI implementation to modify since it takes time to get familiar with the architecture. MPI implementations are notoriously dense and difficult to read, so choosing the correct implementation to work with at the beginning of a research project is an important step in finishing that project in time. For this reason, I did not choose an production-grade MPI implementation such as Open MPI [3] or MPICH [4]. These implementations focus on high performance (plus simultaneous implementation portability), which means they have gone through many iterations and have many interacting components that have been fine-tuned to work a specific way. Because the changes required by the Sessions model are large, fitting them into such an optimized implementation would take too much time. Instead, I chose to modify ExaMPI, which is an MPI implementation designed to be research friendly [9]. This allowed me to rapidly experiment with new ideas without getting bogged down in the implementation details. The following sections address the changes I made to various parts of ExaMPI's architecture.

3.1 Implementing the MPI Sessions API

The most important change to ExaMPI accomplished in this project was the addition of the complete set of functions required of any MPI Sessions implementation by the MPI Standard [8]. Further, all functions adhere to the requirements put forth by the MPI Standard to ensure that they function correctly. For example, the *MPI_Session_init()* method was written to be a local function rather than a global function such as *MPI_Init()*. Two process sets, `mpi://WORLD` and `mpi://SELF`, are mandated by the MPI Standard as well, so additional code was written to

create these process sets at runtime and provide them to each process within an MPI application. The *MPI_Session_get_num_psets()* and *MPI_Session_get_nth_pset()* methods were added to enable the selection of process sets within an MPI application, and the *MPI_Group_from_session_pset()* method was added to enable the creation of MPI_Groups from process sets. Finally, the *MPI_Comm_create_from_group()* method was added to enable the creation of an MPI_Comm from an MPI_Group without a parent communicator. The internal structures of MPI_Group and MPI_Comm objects in ExaMPI were not modified.

3.2 Changes to the Runtime System

ExaMPI uses the `mpiexec` command to launch MPI applications. These commands save information about the runtime environment into environment variables and then spawn a subprocess for the actual application to execute from. For several reasons, this approach needed to be modified in order to support the Sessions model. The `mpiexec` command needed to be modified to accept process set names as command line arguments. Additionally, because the Sessions model allows for multiple sessions to run through one call to `mpiexec`, logic had to be added to enable the execution of multiple applications with varying amounts of process counts from the same job submission. Each session must be independent, so information regarding which process belongs to which session needed to be stored in environment variables for later use by ExaMPI. This information is used to ensure that processes in different sessions are unable to build connections to each other. Another change needed was to make `mpiexec` spawn a separate subprocess for each application associated with a given job submission rather than spawning a single subprocess for the entire job. By isolating each subprocess, any unwanted dependencies between sessions is prevented. No changes to ExaMPI's runtime daemons were necessary. Finally, a flag was added to `mpiexec` to enable the user to choose which internal communication topology to use at runtime. Users can include the `--use_ring` flag in their `mpiexec` call to tell ExaMPI to build connections in a ring pattern rather than an all-to-all pattern. This is discussed further in Section 3.4.

3.3 Dynamic Initialization

The Sessions model allows for multiple `MPI_Sessions` to be created and finalized at any point during the execution of an MPI application. To enable this capability, ExaMPI had to be modified to support dynamic initialization. This was done by creating a global variable to keep track of the number of active `MPI_Sessions` within a given application. When `MPI_Init()` or `MPI_Session_init()` is called, this variable is incremented. When `MPI_Finalize()` or `MPI_Session_finalize()` is called, the variable is decremented, and if the value is 0 after this decrement, ExaMPI performs its normal finalization and teardown functions and the application stops executing. Without this change, ExaMPI would finalize and cease execution as soon as one `MPI_Session` called `MPI_Session_finalize()`, causing all other active `MPI_Sessions` to be destroyed erroneously. Importantly, this approach maintains compatibility with the World model method of initialization and finalization, `MPI_Init()` and `MPI_Finalize()`, so as to ensure backwards compatibility with MPI applications written for the World model.

3.4 Topological Connection Building

The transport layer in ExaMPI was designed to enable the abstraction of network APIs. This is done through a Transport class, which is responsible for handling any memory associated with the network. The most common transport used by ExaMPI is the TCP transport, but the way it was implemented posed some problems for this project. The first problem is that the TCP transport was designed with `MPI_COMM_WORLD` in mind, which means each process is assumed to be able to connect to any other process. However, in the Sessions model, processes in different `MPI_Sessions` are not permitted to communicate with each other because each `MPI_Session` is an independent handle to the MPI library. Another issue with the TCP transport is that connections between all processes are established when MPI is initialized. This means that, even when using the Sessions model, the TCP transport builds all of the connections needed for `MPI_COMM_WORLD`, which makes it impossible to reduce the startup overhead when running large MPI applications. For these reasons, a new Transport class was created and designed with the Sessions model

in mind. This new transport, called TCPSessions, uses information from the runtime system to build connections only between processes in the same `MPI_Session`. Additionally, in order to take advantage of the performance benefits of using topological communication patterns, the TCPSessions transport only builds connections to the ranks immediately above and below a given process. This significantly reduces the cost of initializing the transport, since each process only has to build two connections rather than connecting to all other processes in the job. This approach did present its own challenges, though. TCP does not have a way to forward a message to another host, so it would be impossible for a process to send a message to a process with rank that is not immediately above or below it. Therefore, a message forwarding protocol was added to the TCPSessions transport.

3.5 Topological Communication Algorithms

Having fewer connections when using the TCPSessions transport caused other problems within ExaMPI as well. Specifically, all of the communication algorithms used to send data between processes were designed under the World model. For example, the *MPI_Reduce()* function takes input elements from each process and returns output elements to a root process. In ExaMPI, this is done by having the root process receive from all other processes¹, and having all other processes send to the root process. When using the TCP transport, this works as intended because all processes have connection information for all other processes. When using the TCPSessions transport, though, this implementation fails above a certain number of processes. If a job with four processes tries to do an *MPI_Reduce()* with root=0 using the TCPSessions transport, process 2 will not be able to send its information to process 0 because it only built connections to process 1 and process 3, the ranks immediately above and below it. Therefore, new communication algorithms had to be designed to be compatible with topological connection building. In the topological version of *MPI_Reduce()*, the root process only receives from the process below it, the process above the root process only sends to the process above it, and all other processes receive from the

¹Tree-based reduction algorithms are being made standard in a forthcoming release of ExaMPI.

process below it and send to the process above it. Because each process only sends to and receives from its immediate neighbors, the connections built by the TCPSessions transport are sufficient.

For this project, I created topological versions of the *MPI_Reduce()*, *MPI_Gather()*, *MPI_Scatter()*, and *MPI_Bcast()* functions. I chose to implement these functions because most other communication algorithms, such as *MPI_Allreduce()* and *MPI_Allgather()*, can be done as a combination of topological communication algorithms that have already been implemented.

CHAPTER 4

EVALUATION

In order to evaluate the changes and additions made to ExaMPI during this project, micro-benchmarks were developed for several different MPI functions. These tests provided timing results for the original TCP transport as well as the new TCPSessions transport so that the two transports could be compared in terms of efficiency.

4.1 Experimental Setup

The results in this section were gathered using ExaMPI’s *develop* branch at SHA af5b88f for the World model and ExaMPI’s *feat-sessions* branch at SHA a341497 for the Sessions model. Data was collected on a single node running CentOS Linux 7 with two AMD EPYC 7662 64-Core processors and 512 GB of memory. The data was gathered during normal operating hours, so the node was addressing other workloads alongside but isolated from the data collecting runs for this project.

4.2 Results

The following subsections compare the TCP transport, which uses all-to-all connection building, to the TCPSessions transport, which uses ring connection building where each process connects to the process immediately above and below it in the rank order. Note that the plots for the TCP transport only provide data for up to 32 processes, while the plots for the TCPSessions transport provides data for up to 128 processes. By building fewer connections at startup, ExaMPI is able to handle many more processes during initialization when using the TCPSessions transport.

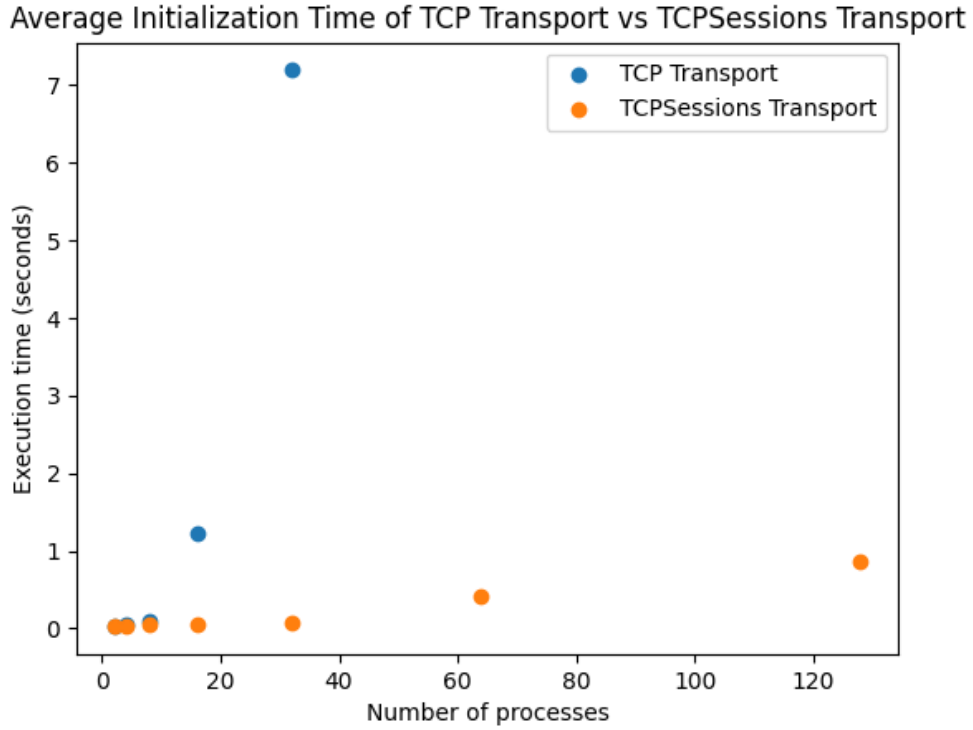


Figure 4.1 Average initialization time of TCP transport vs TCPSessions transport

4.2.1 Initialization Time

By far, the largest benefit of using the TCPSessions transport over the TCP transport is initialization time. This is because a large portion of initialization time in ExaMPI is spent building connections. Since the TCPSessions transport builds far fewer connections than the TCP transport, it is able to complete its setup much faster. MPI initialization times were collected by measuring execution time of *MPI_Init()* for both transports using a small initialization benchmark program written for this project. This benchmark was run 100 times per transport at increasing numbers of processes, with the results being averaged and plotted as depicted in Figure 4.1. The source code for this timing benchmark can be found in Appendix B.

Not only can the TCPSessions transport handle far more concurrent processes than the TCP transport, it can also initialize those processes much faster. The large difference in initialization time occurs due to the way connection building is done in ExaMPI. ExaMPI uses a static triangular

connection building pattern, which means each process must perform three socket calls for each other process in the allocation. A description of the connection building algorithm in ExaMPI is shown in Algorithm 1.

Algorithm 1 Static Triangular Connection Building

```

1: for  $rank = 0, \dots, my\_rank - 1$  do
2:   Attempt to connect with  $rank$ 
3:   Send  $my\_rank$  to  $rank$ 
4: end for
5: for  $rank = my\_rank + 1, \dots, num\_ranks - 1$  do
6:   Accept connection from  $rank$ 
7:   Receive  $their\_rank$  from  $rank$ 
8: end for

```

Because socket calls are relatively expensive operations, connection building can cause initialization time to become unacceptably slow at higher process counts. The sending and receiving of ranks in Algorithm 1 is required because the order in which connection requests are sent and received is not guaranteed. Therefore, each process needs to know the rank of the process with which it just connected. Connection building in the TCPSessions transport addresses several of the shortcomings in the static triangular connection building method. First, because each process only connects with its nearest neighbors in the rank order, there are far fewer necessary socket calls. Additionally, since each process always has only two neighbors, the number of connections required for each process remains constant rather than growing linearly. Therefore, the total number of connections required for the application grows linearly in relation to the number of processes in the TCPSessions transport, whereas the total number of connections required grows quadratically in the TCP transport. More specifically, the total number of connections C_N required by the TCP transport is shown in Equation 4.1.

$$\begin{aligned}
C_N &= (N - 1) + (N - 1) + \dots + (N - 1), \\
C_N &= N(N - 1).
\end{aligned}
\tag{4.1}$$

The total number of connections C_N required by the TCPSessions transport is shown in Equation 4.2.

$$C_N = (2) + (2) + \dots + (2) \quad (4.2)$$

$$C_N = 2N$$

For example, an application with 128 processes would require 128×127 , or 16,256 connections using the TCP transport but would only require 2×128 , or 256 connections using the TCPSessions transport [2].

Finally, because the ranks of a process's neighbors are always known, the additional send and receive needed to obtain the connecting process's rank is not required, meaning each process must only perform two socket calls. These improvements are what contribute the most to the significant drop in initialization overhead when using the TCPSessions transport.

4.2.2 *MPI Function Execution Time*

Additional benchmark programs were written to measure the execution time of the *MPI_Bcast()* and *MPI_Gather()* functions using the TCP and TCPSessions transport. The source code for these benchmarks can be found in Appendix B. To minimize external contributions to the timing results, the persistent variants of these communication algorithms were used. These functions were run with buffer sizes of 1, 100, and 1000 integers to get a better picture of the two transports' performance with different message sizes. In addition to the varying buffer sizes, several different internal communication algorithms were tested. Specifically, three different algorithms were used with each transport in order to compare the performance of the transports when using different internal communication topologies: linear, ring, and binomial. Figure 4.2 illustrates the three communication algorithms performing a simple gather collective operation with eight processes.

To obtain the results, both transports were tested with all three types of communication algorithm at all three buffer sizes. Each of the possible combinations were run 100 times at increasing numbers of processes, with the first five runs being discarded to allow the communication paths to "warm up." The average and maximum execution time were then

aggregated and plotted. A demonstrative example is shown in Figure 4.3; the rest of the plots can be found in Appendix A.

Figure 4.3 shows the typical differences between the TCP and TCPSessions transports. The TCPSessions transport has more variance resulting from the ring structure of the internal connections. If a process tries to send a message to another process that is not one of its neighbors in the rank order, that message must be forwarded along the connection ring until it reaches its original destination. Therefore, if any process is experiencing slowdown, it is likely going to affect the entire execution and cause variations in the final execution time. Since every process has connection information for every other process when using the TCP transport, the TCP transport is not subject to forwarding slowdowns and thus has less variance. This difference, along with the natural variation associated with running benchmarks without exclusive access to the system, account for an average execution time that is about 0.0001 seconds slower when using the TCPSessions transport.

For the sake of brevity, each combination of transport, buffer size, and communication algorithm will not be discussed at length. Instead, Table 4.1 contains a brief comparison between the TCP and TCPSessions transports for each combination. All comparisons are done at 32 processes because that is the maximum number of processes that the TCP transport can consistently handle¹. The table is structured as follows: Algorithm Type describes the internal communication topology used by ExaMPI, MPI Function describes the operation being performed, Buffer Size is the number of elements being passed to the MPI function, Faster Avg lists which transport had the faster average execution time, the first % Diff is the percent difference between the average execution times of the two transports, Higher Max lists which transport had the highest maximum execution time, and the second % Diff represents the percent difference between the maximum execution times of the two transports.

¹This limitation is being removed in the near future.

Algorithm Type	MPI Function	Buffer Size	Faster Avg	% Diff	Higher Max	% Diff
Linear	Bcast	1	TCP	49.4	TCP	64.0
Linear	Bcast	100	TCP	59.4	TCP	71.8
Linear	Bcast	1000	TCP	59.0	TCP	79.7
Linear	Gather	1	TCPSessions	4.3	TCP	0.4
Linear	Gather	100	TCP	24.0	TCPSessions	9.1
Linear	Gather	1000	TCP	50.6	TCPSessions	145.4
Ring	Bcast	1	TCP	21.7	TCPSessions	70.5
Ring	Bcast	100	TCP	11.2	TCP	114.3
Ring	Bcast	1000	TCP	2.4	TCP	98.5
Ring	Gather	1	TCP	21.7	TCPSessions	50.2
Ring	Gather	100	TCP	34.6	TCP	2.4
Ring	Gather	1000	TCP	140.4	TCPSessions	107.7
Binomial	Bcast	1	TCP	87.7	TCP	96.4
Binomial	Bcast	100	TCP	115.0	TCPSessions	59.0
Binomial	Bcast	1000	TCP	102.8	TCPSessions	96.8
Binomial	Gather	1	TCP	8.0	TCP	155.4
Binomial	Gather	100	TCP	3.6	TCP	134.2
Binomial	Gather	1000	TCP	42.2	TCPSessions	57.3

Table 4.1 Results from every combination of algorithm type, MPI function, and buffer size

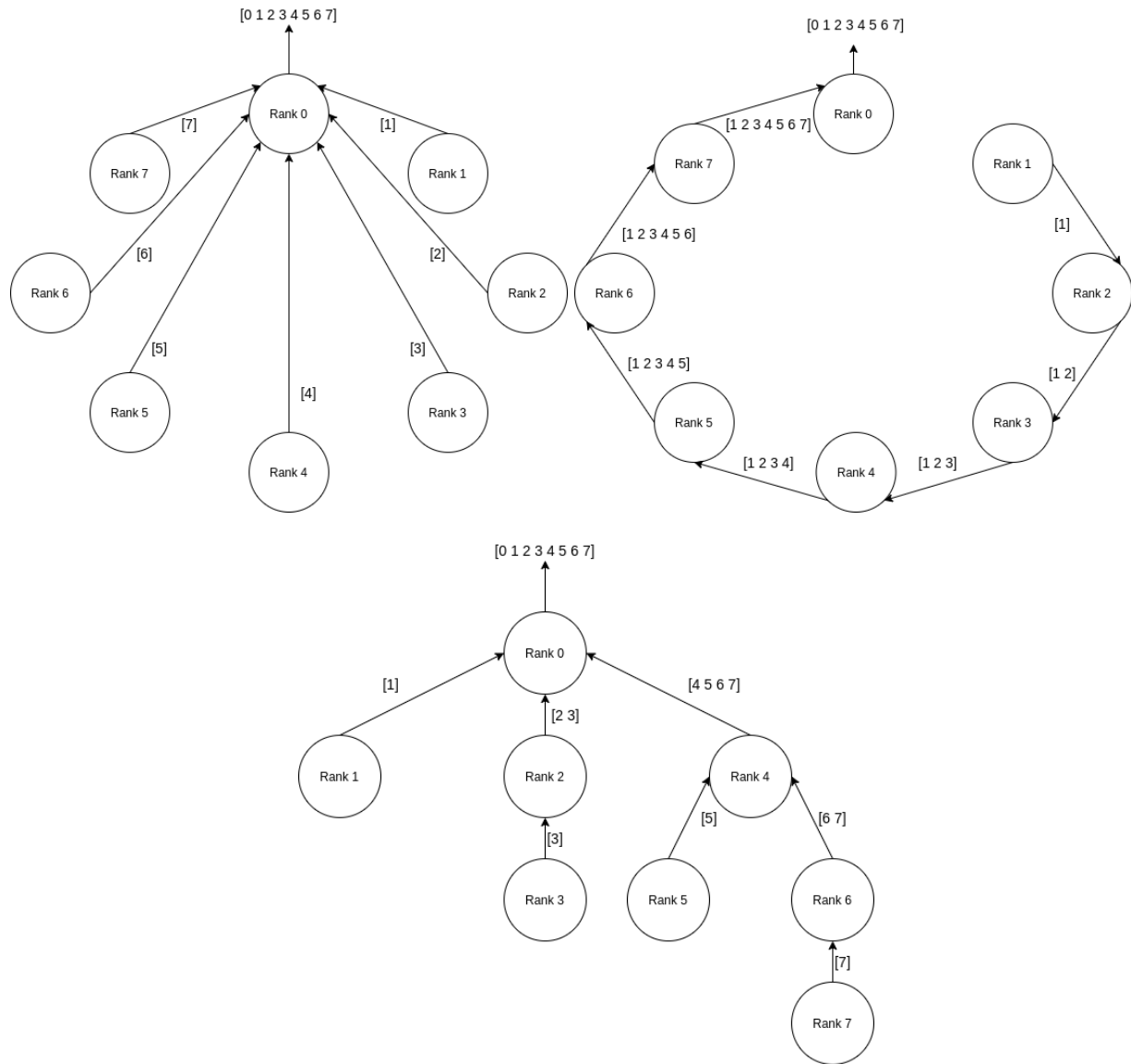


Figure 4.2 Linear Gather (top left), Ring Gather (top right), and Binomial Gather (bottom center)

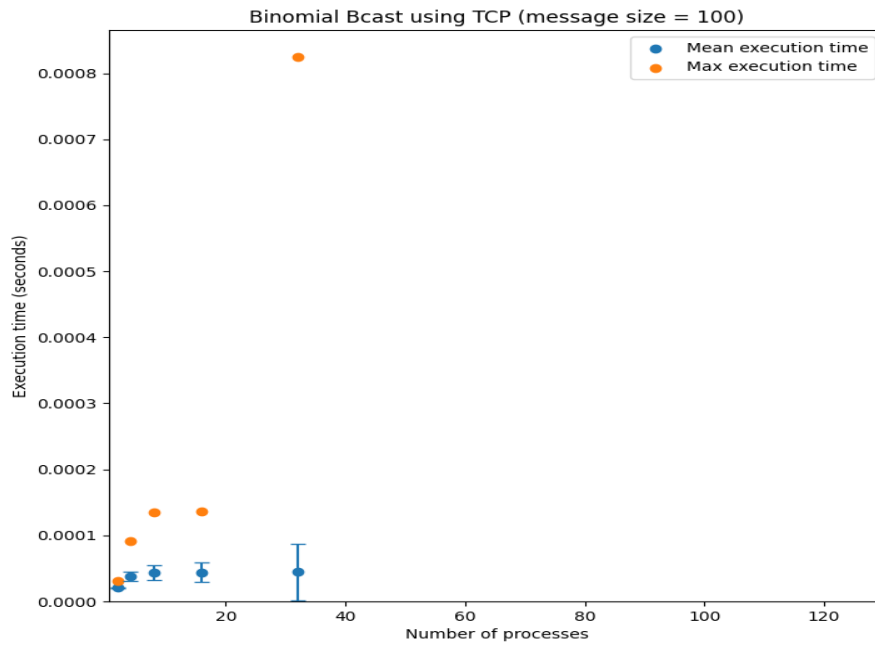
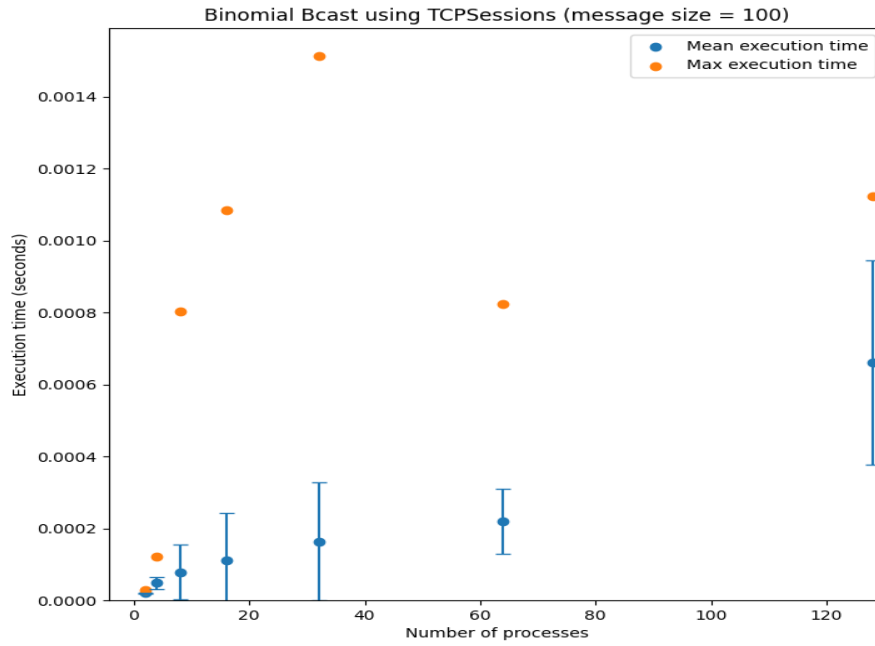


Figure 4.3 Binomial broadcast with buffer size of 100 on TCPSessions transport (top) vs. Binomial broadcast with buffer size of 100 on TCP transport (bottom)

CHAPTER 5

CONCLUSION

This chapter concludes the thesis by discussing the results from Chapter 4, the limits of the work, and future work that could build upon the work accomplished in this project.

5.1 Discussion of Results

The TCP transport outperforms the TCPSessions transport in all but one test, where TCPSessions was slightly faster. This is expected for the reasons stated in Chapter 4: the TCP transport requires less indirect communication since it builds connections between all processes upon initialization. The effects of this store-and-forward mechanism are often more pronounced at larger buffer sizes, because message forwarding involves copying the application buffer multiple times. If the application buffer is large, it will take longer to copy, making each forward slower. In general, the numbers generated by these benchmark applications were quite small. This means that the results include natural variance introduced by the inherent volatility of a system frequently used for computational research.

While the TCP transport tends to outperform the TCPSessions transport during MPI function execution, the opposite is true for initialization. Importantly, the time saved from initialization using the TCPSessions transport far outweighs the time saved from function execution using the TCP transport. At 32 processes, the TCP transport took over seven seconds to initialize while the TCPSessions transport took less than 0.1 seconds. Further, using the TCPSessions transport enabled the use of far more processors during one job by lowering the number of connections required upon initialization. Therefore, the TCPSessions transport is better for applications that require a large number of processors.

Another result of this project was the implementation of a functioning MPI Sessions API in ExaMPI. Because ExaMPI is designed with research and readability in mind, having MPI Sessions added to the code base will allow for faster and easier experimentation with the various functionality that MPI Sessions provides. Additionally, once MPI Sessions is officially added to the MPI Standard, the work of this project will serve as a useful resource for developers of other MPI middleware attempting to implement MPI Sessions themselves.

The final result of this project to be discussed is the creation of topological communication algorithms. These communication algorithms were designed as a ring, so that each process only communicates with its immediate neighbors in the process rank order. This was done in order to take advantage of the small number of connections made by the TCPSessions transport. However, once a forwarding protocol was added to the TCPSessions transport, we found that the linear and binomial communication algorithms outperformed the ring communication algorithm despite the overhead associated with forwarding messages. Topological communication algorithms, such as ring algorithms, would be more relevant in situations where message forwarding is particularly expensive or impossible, but the ability to experiment in such situations was not present during the course of this project.

5.2 Limitations

The most significant limitation of this project was the amount of available system resources. Because ExaMPI is currently only compatible with the job scheduler Slurm, the number of systems that could be used to generate results was severely limited. There is currently only one cluster at UTC that has Slurm installed and configured, but it is a small system that is almost always running at maximum capacity. Testing the changes made to ExaMPI during this project on multiple nodes of a cluster would require exclusive use of several nodes, which was unrealistic given the popularity of the only compatible cluster. Therefore, all of the results in this thesis were gathered on one node with 128 available cores. It is likely that the TCPSessions transport would perform worse when used between nodes in a cluster as the socket calls that occur during message forwarding are more expensive over a network than they are within a node. This hypothesis cannot currently be

tested, though, because of a lack of a compatible and available system. Additionally, though the ring communication algorithms performed worse than the linear and binomial algorithms in these conditions, it is possible that they would perform at similar or even faster speeds when message forwarding is more expensive.

5.3 Future Work

There are several natural continuations of the work done for this thesis. First, enabling more interaction between the runtime system, job scheduler, and MPI application would allow for the creation of better process sets for use by MPI Sessions. For example, a user might want to ask the job scheduler to allocate a set of processes that are contained within a physical hardware topology such as a ring. By passing this set of processes from the job scheduler to the runtime system to the MPI application, it would be possible to create a virtual communication topology that matches the hardware topology passed from the job scheduler. This would enable better data locality, which has been shown to increase application and network performance [1].

Similarly, enabling the runtime system of ExaMPI to modify process sets during the MPI application would yield interesting results. This concept, called dynamic process sets, is one way that MPI could be made more elastic. Currently, MPI application rely on the number of processes to be constant throughout its execution, which makes it difficult to run such applications on the cloud. Dynamic process sets could be the key to removing the restriction on changing the number of processes, which would allow for MPI applications to be run on far more systems than currently possible.

Another opportunity is to combine sessions, topologies, and direct-to-persistent collective operation constructors. In such a model, general-purpose communicators would be completely avoided in favor of a series of topological collective operations that derived directly from psets. This is a logical extension to the Sessions model and collective and topology chapters of MPI in a future edition of the standard, such as MPI-5.

Finally, a necessary addition to ExaMPI in order for the changes from this project to be fully utilized is a method of automatically selecting topological transports and algorithms at runtime or

compile time. Currently, a user must specify whether they want to use them, which means that most users will likely ignore the option in favor of what they know. By analyzing the amount of collective communication, point-to-point communication, processes, and necessary connections, ExaMPI may be able to automatically speed up an MPI application's execution time by selecting topological transports and algorithms that better suit the nature of the application.

REFERENCES

- [1] Agung, M., Amrizal, M. A., Egawa, R., and Takizawa, H. (2019). An automatic MPI process mapping method considering locality and memory congestion on NUMA systems. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, pages 17–24. IEEE. 21
- [2] Dumas, J. (2016). *Computer Architecture: Fundamentals and Principles of Computer Design*. CRC Press. 14
- [3] Graham, R. L., Woodall, T. S., and Squyres, J. M. (2006). Open MPI: A flexible high performance MPI. In Wyrzykowski, R., Dongarra, J., Meyer, N., and Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg. Springer Berlin Heidelberg. 6
- [4] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI Message Passing Interface standard. *Parallel computing*, 22(6):789–828. 6
- [5] Hjelm, N., Pritchard, H., Gutiérrez, S. K., Holmes, D. J., Castain, R., and Skjellum, A. (2019). MPI Sessions: Evaluation of an implementation in Open MPI. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE. 5
- [6] Holmes, D., Mohror, K., Grant, R. E., Skjellum, A., Schulz, M., Bland, W., and Squyres, J. M. (2016). MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. In *Proceedings of the 23rd European MPI Users’ Group Meeting, EuroMPI 2016*, page 121–129, New York, NY, USA. Association for Computing Machinery. 1
- [7] Message Passing Interface Forum (2015). MPI: A Message Passing Interface Standard. <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf> ; Last downloaded April 27, 2021. 1

[8] Message Passing Interface Forum (2020). MPI: A Message Passing Interface Standard. <https://www.mpi-forum.org/docs/drafts/mpi-2020-draft-report.pdf> ; Last downloaded April 27, 2021.

6

[9] Skjellum, A., Rüfenacht, M., Sultana, N., Schafer, D., Laguna, I., and Mohror, K. (2020). ExaMPI: A modern design and implementation to accelerate Message Passing Interface innovation. In *High Performance Computing*, pages 153–169. Springer International Publishing. 6

APPENDIX A

GRAPHS

Average Initialization Time of TCP Transport vs TCPSessions Transport

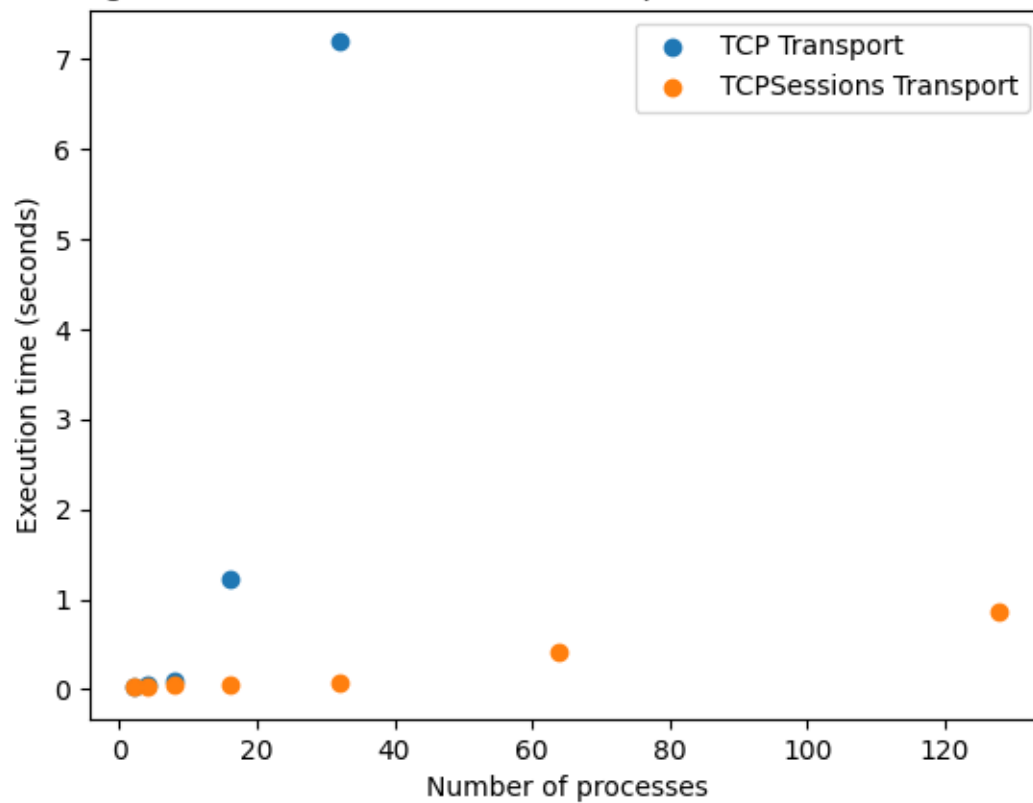


Figure A.1 Average initialization time of TCP transport vs TCPSessions transport

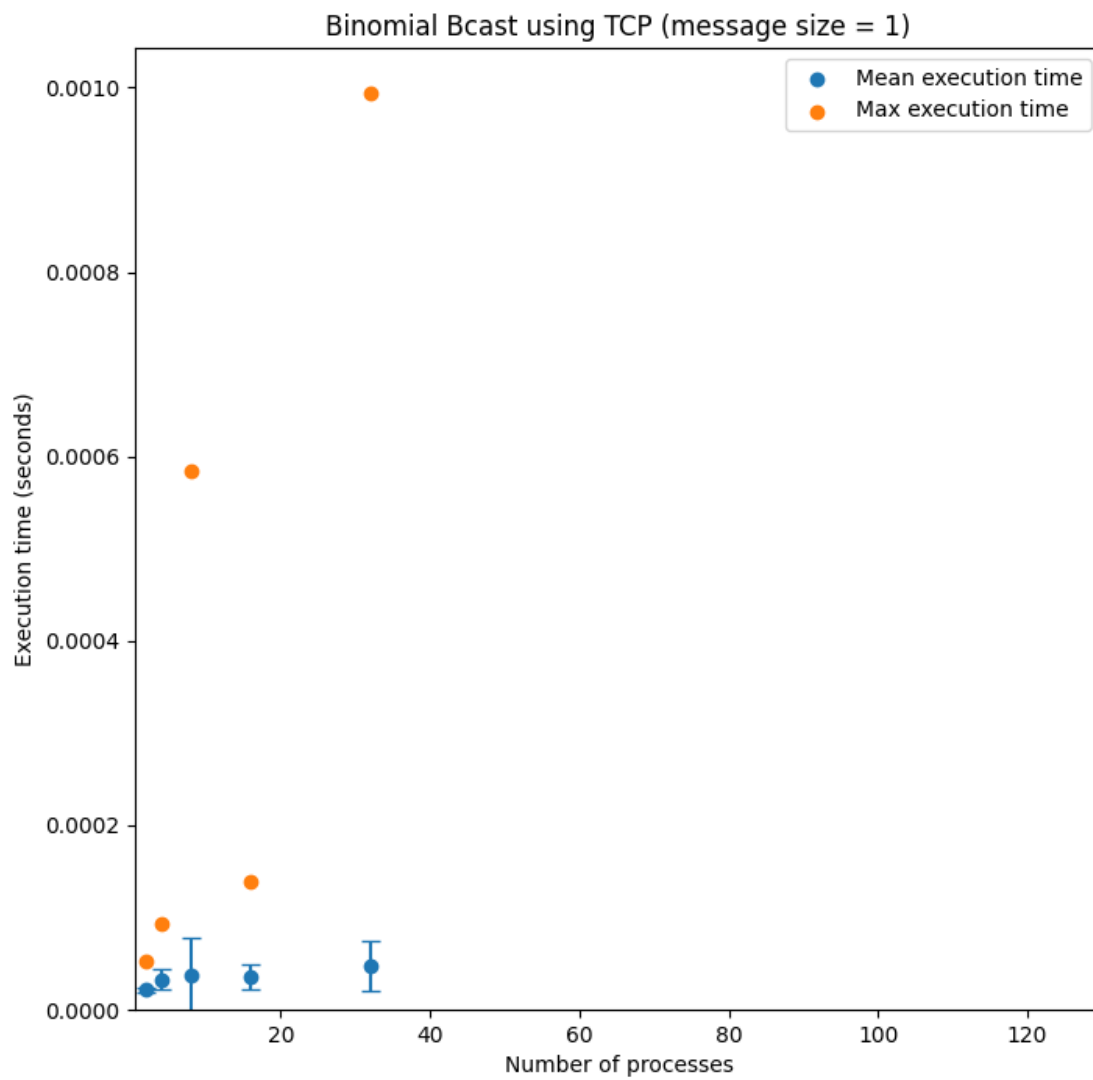


Figure A.2 Binomial bcast with buffer size 1 on TCP transport

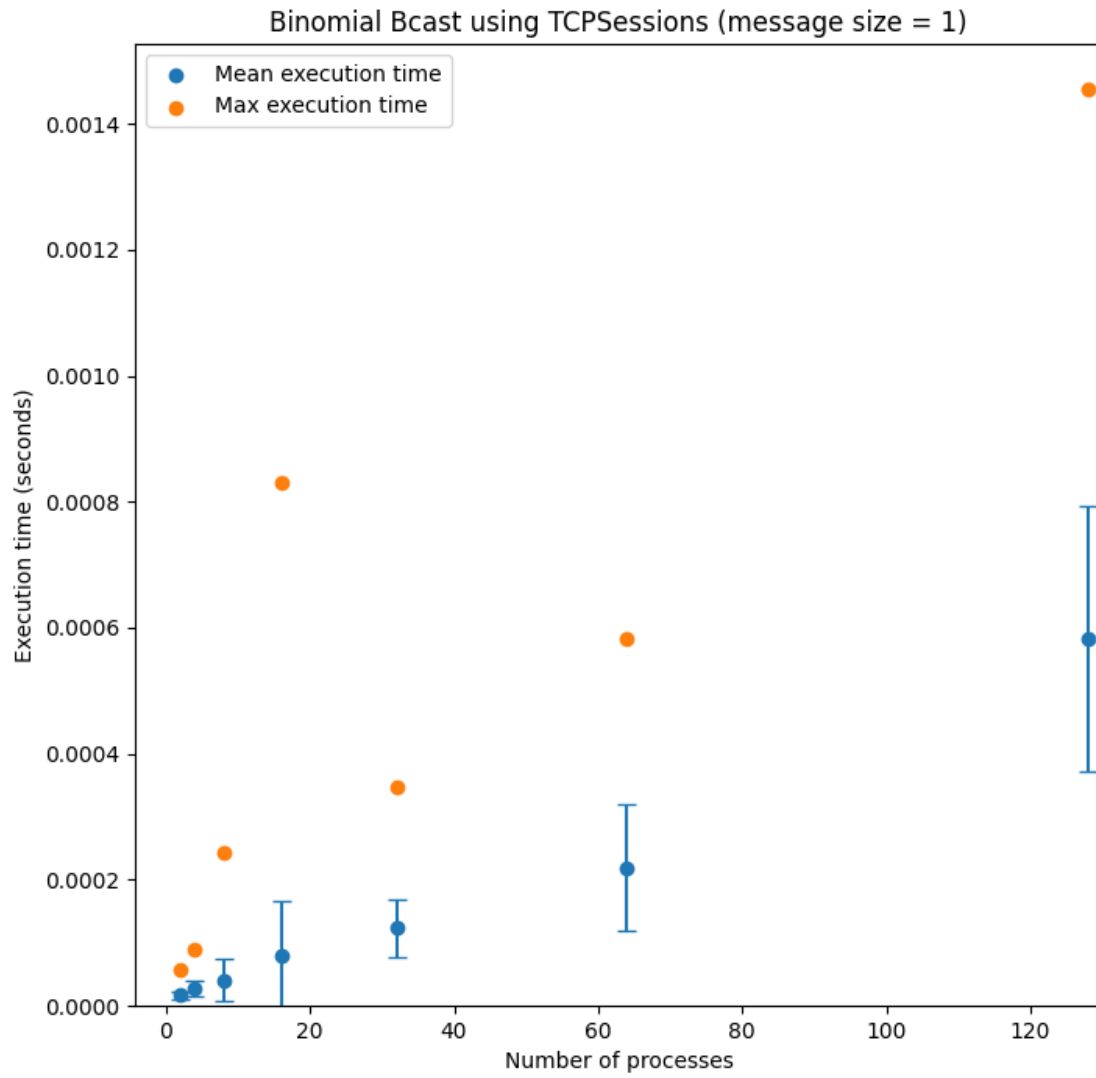


Figure A.3 Binomial bcast with buffer size 1 on TCPSessions transport

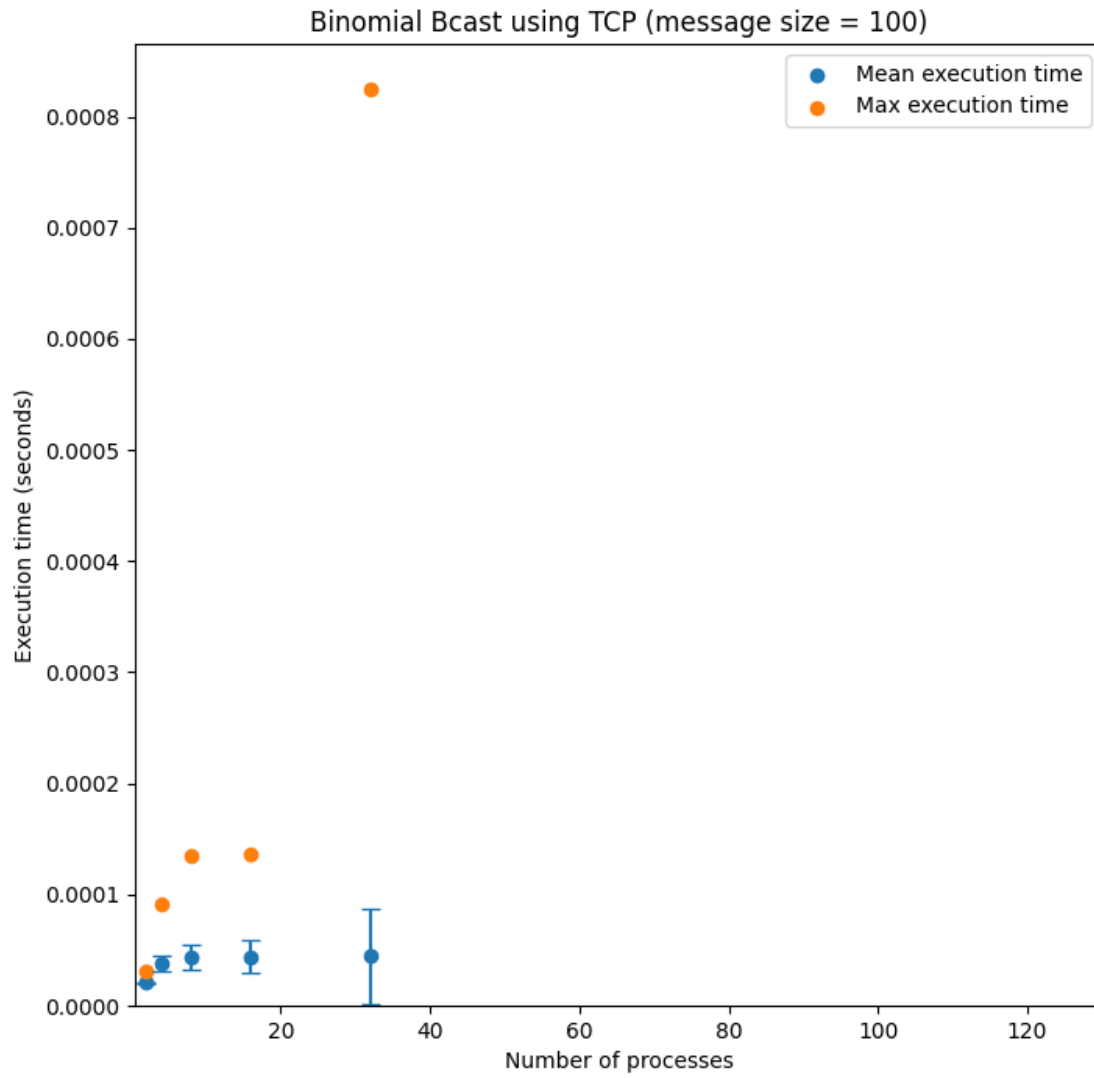


Figure A.4 Binomial bcast with buffer size 100 on TCP transport

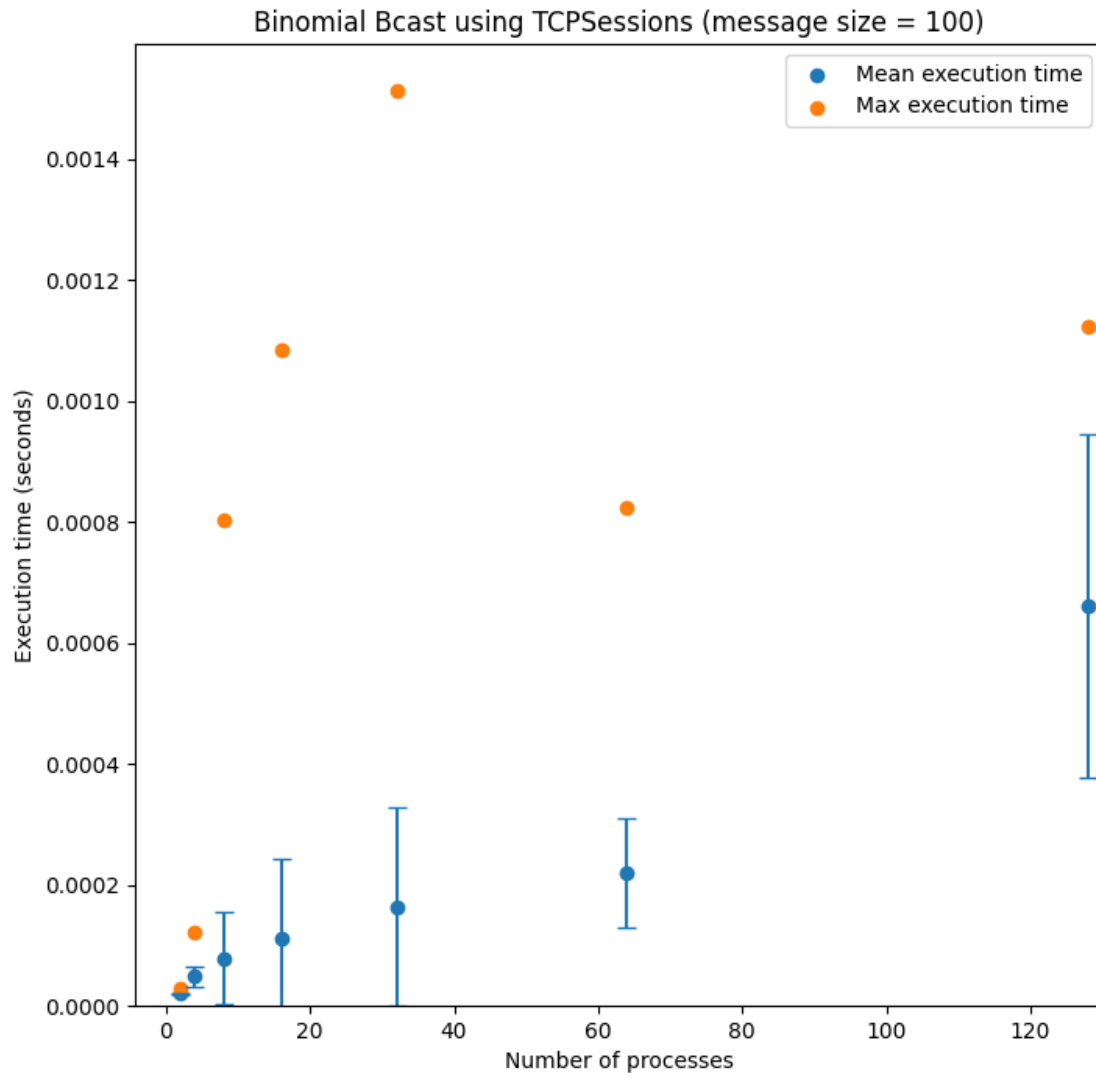


Figure A.5 Binomial bcast with buffer size 100 on TCPSessions transport

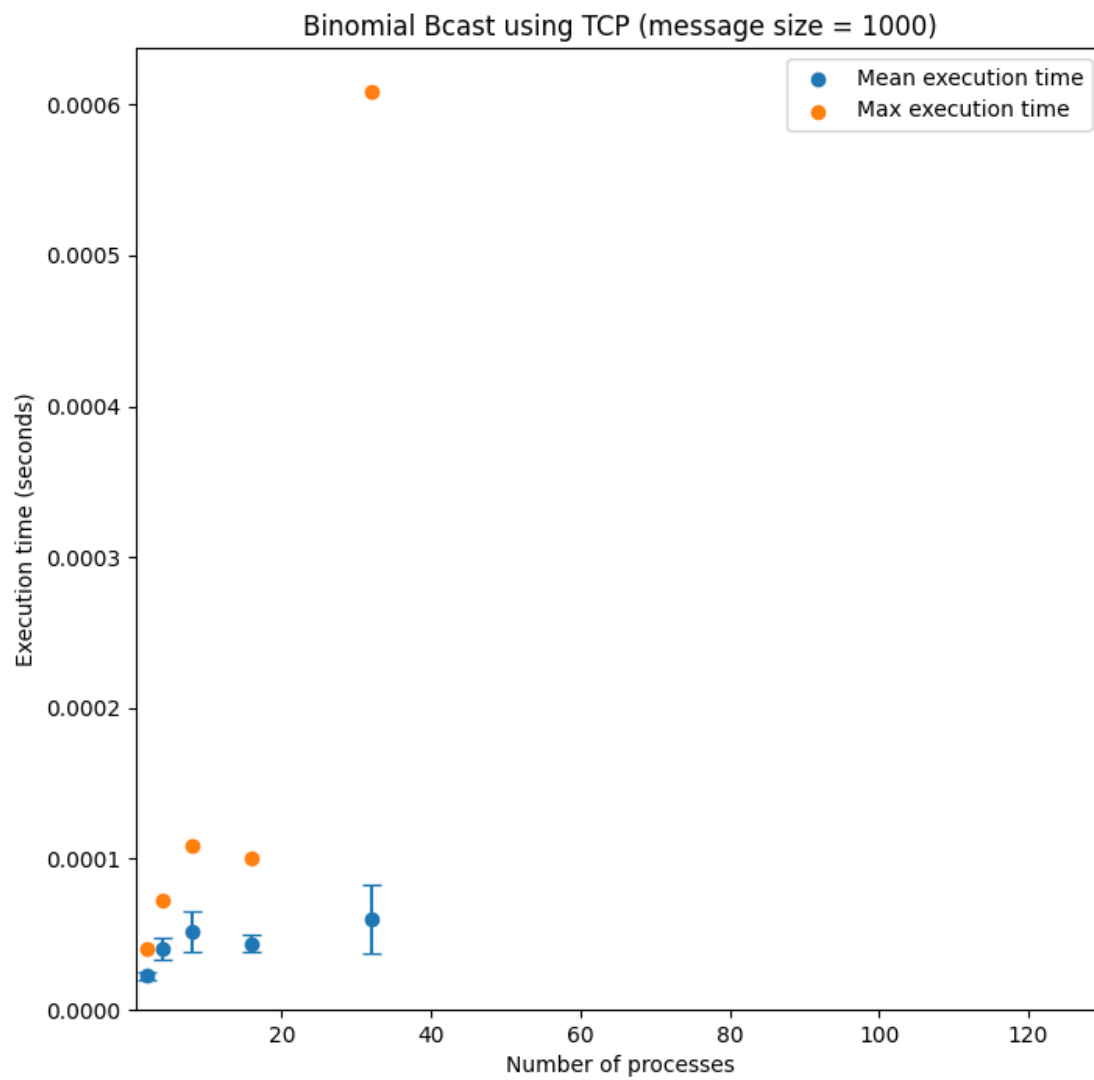


Figure A.6 Binomial bcast with buffer size 1000 on TCP transport

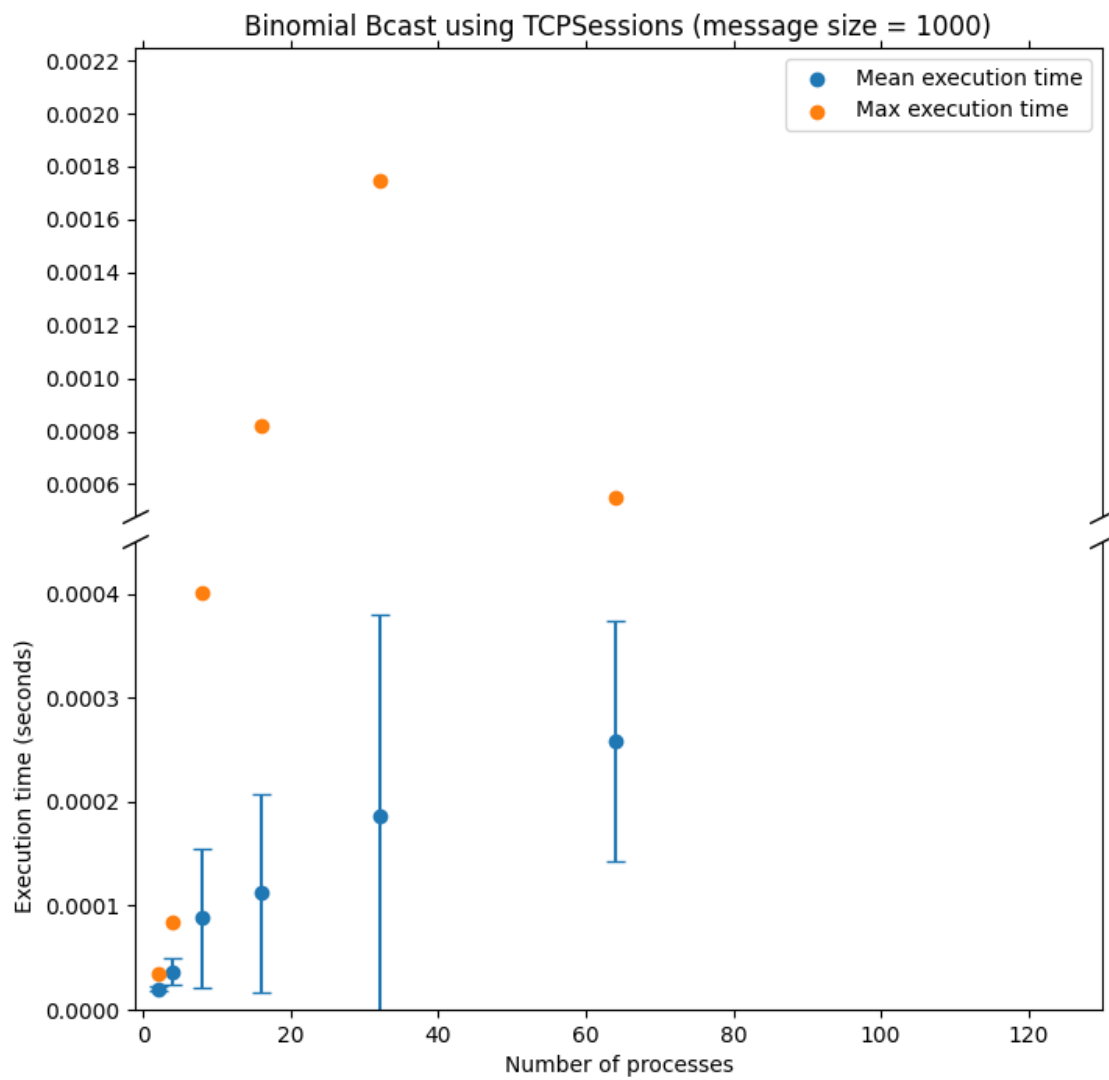


Figure A.7 Binomial bcast with buffer size 1000 on TCPSessions transport

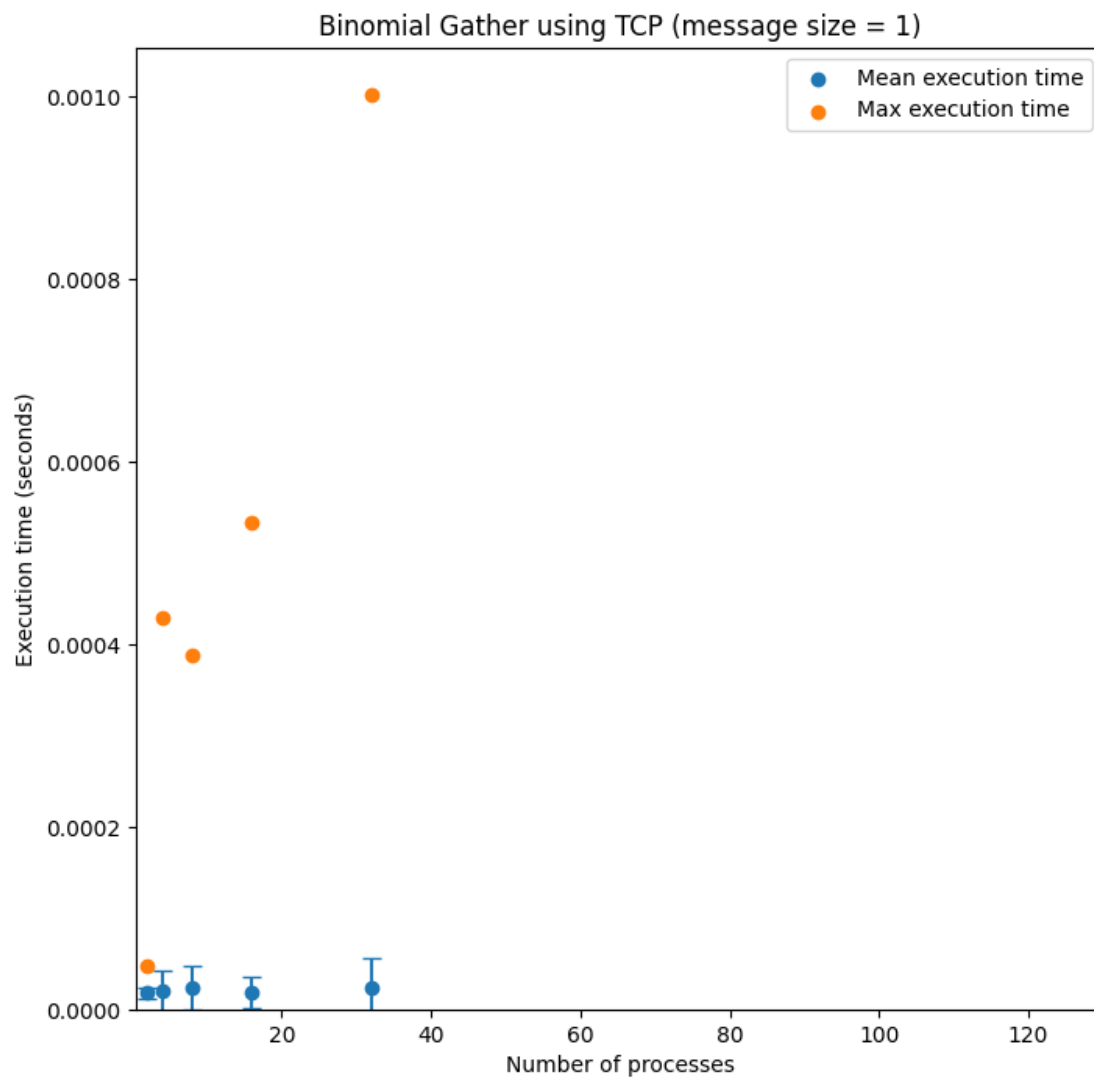


Figure A.8 Binomial gather with buffer size 1 on TCP transport

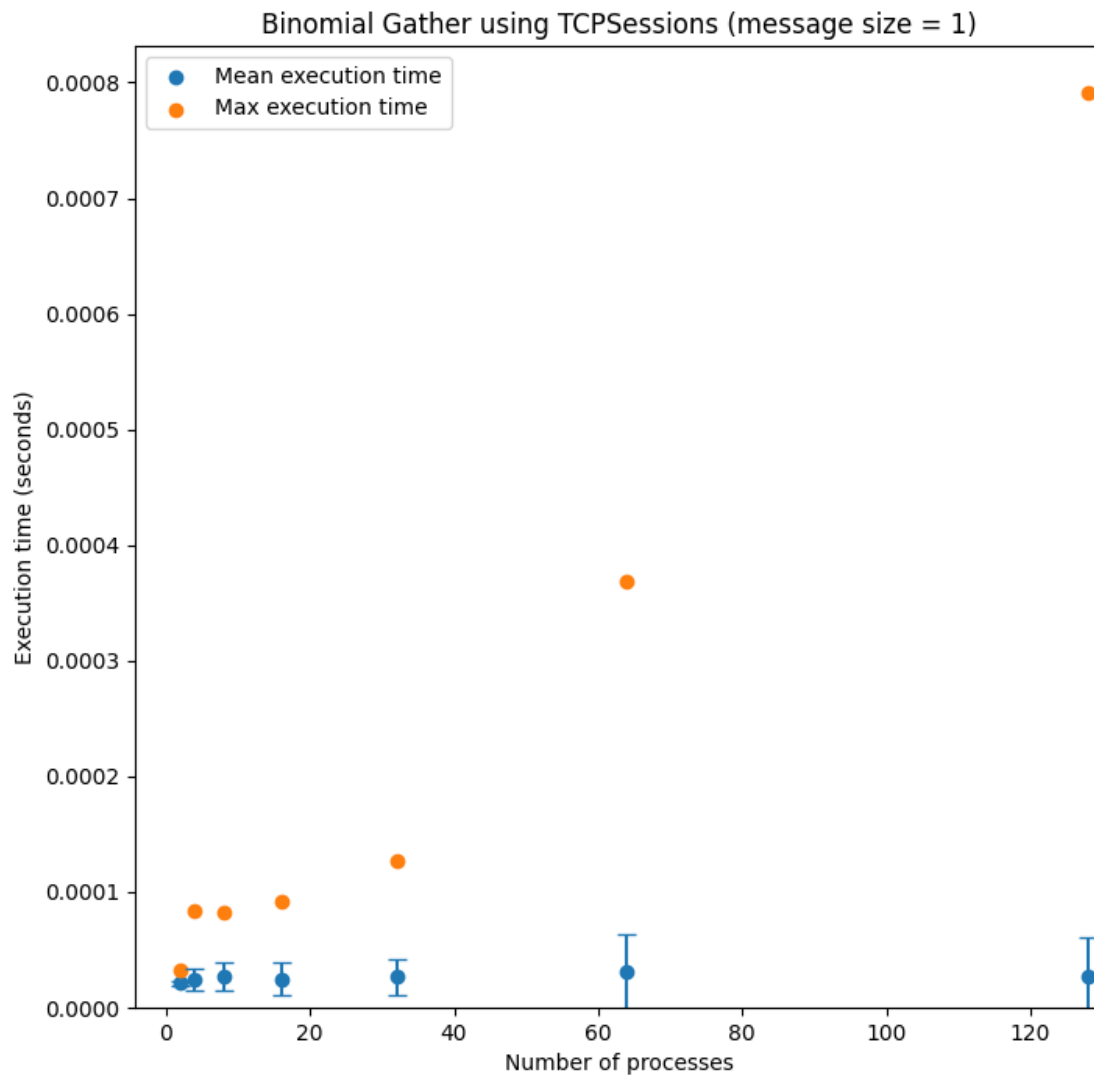


Figure A.9 Binomial gather with buffer size 1 on TCPSessions transport

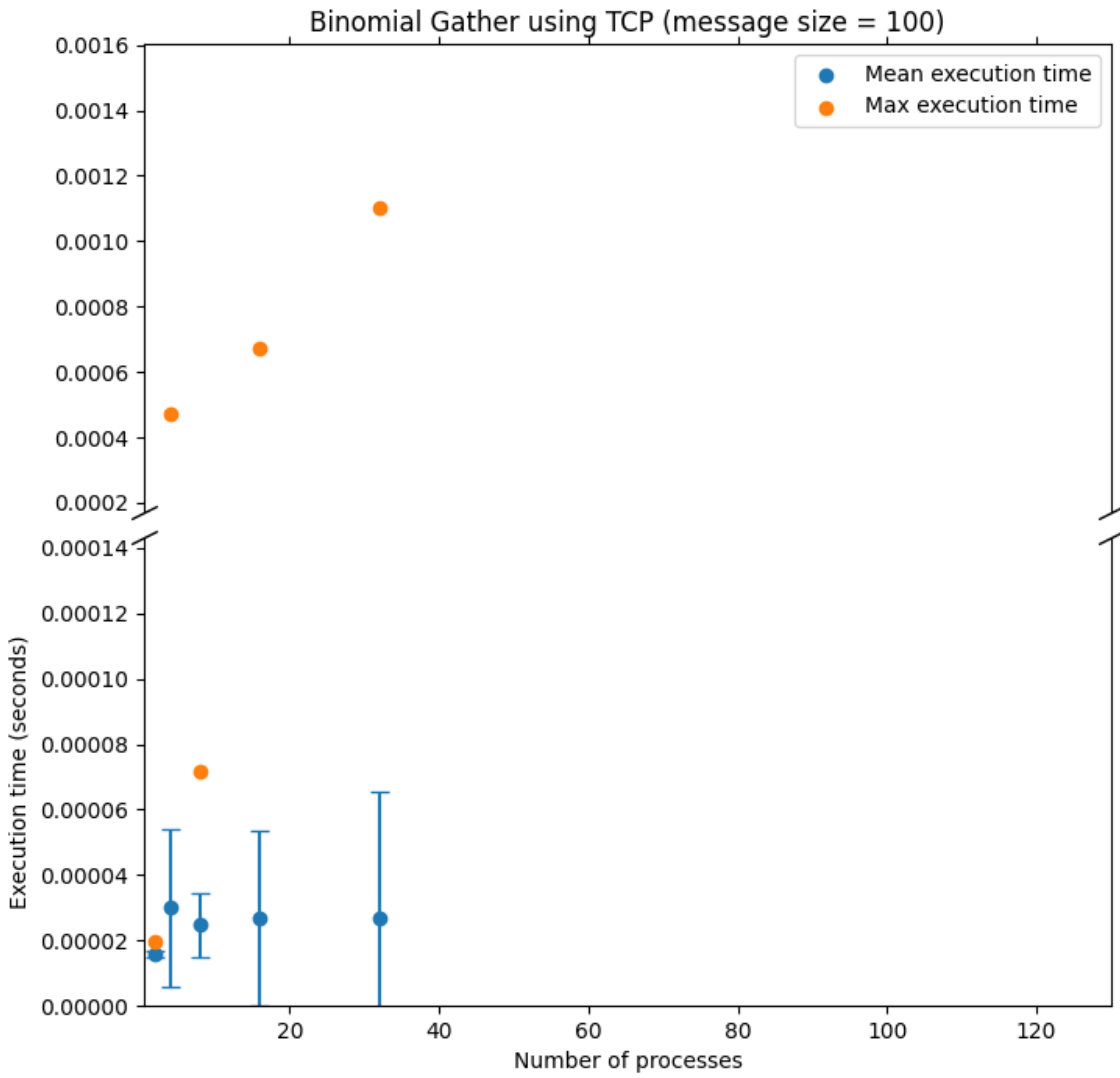


Figure A.10 Binomial gather with buffer size 100 on TCP transport

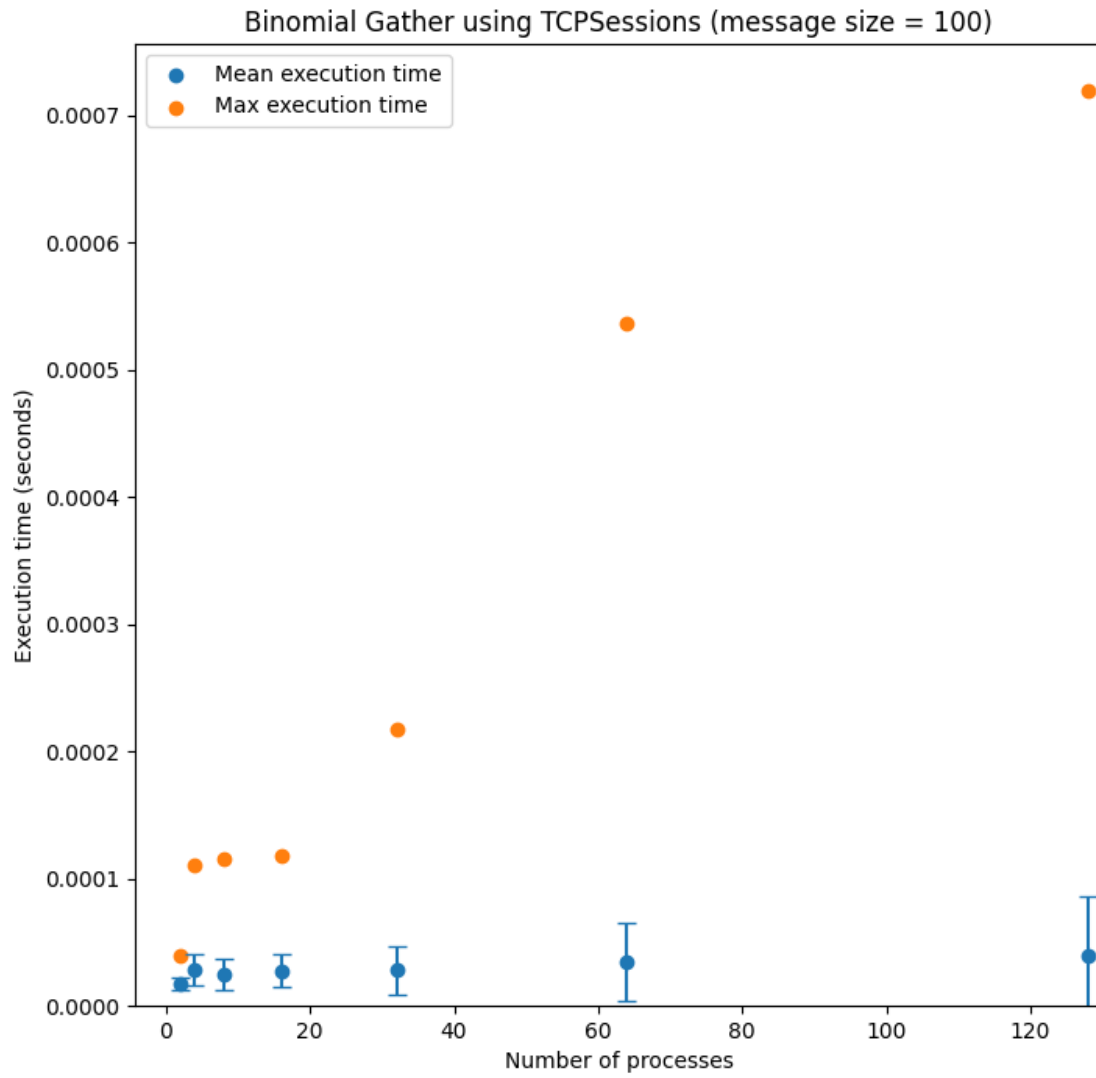


Figure A.11 Binomial gather with buffer size 100 on TCPSessions transport

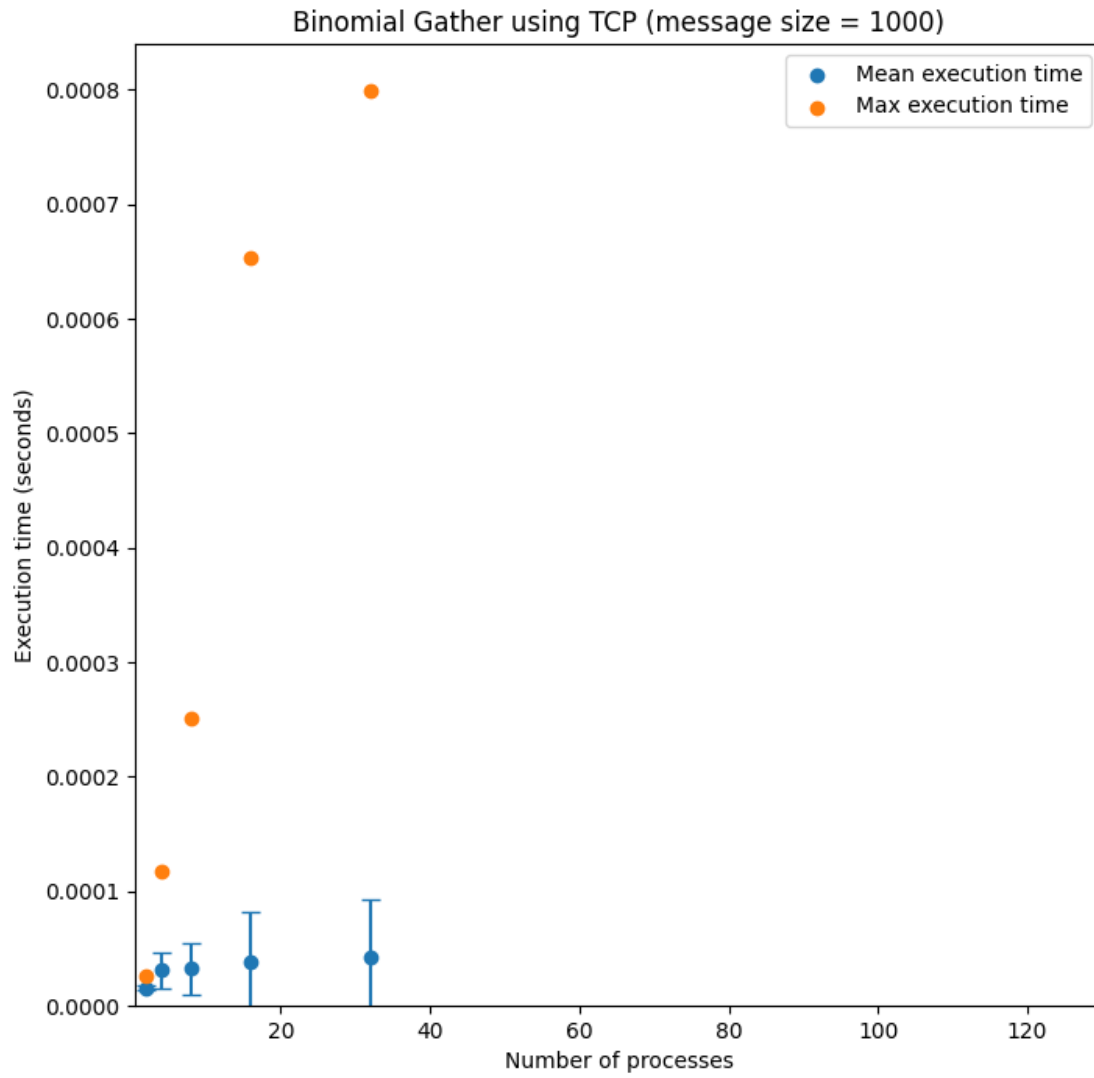


Figure A.12 Binomial gather with buffer size 1000 on TCP transport

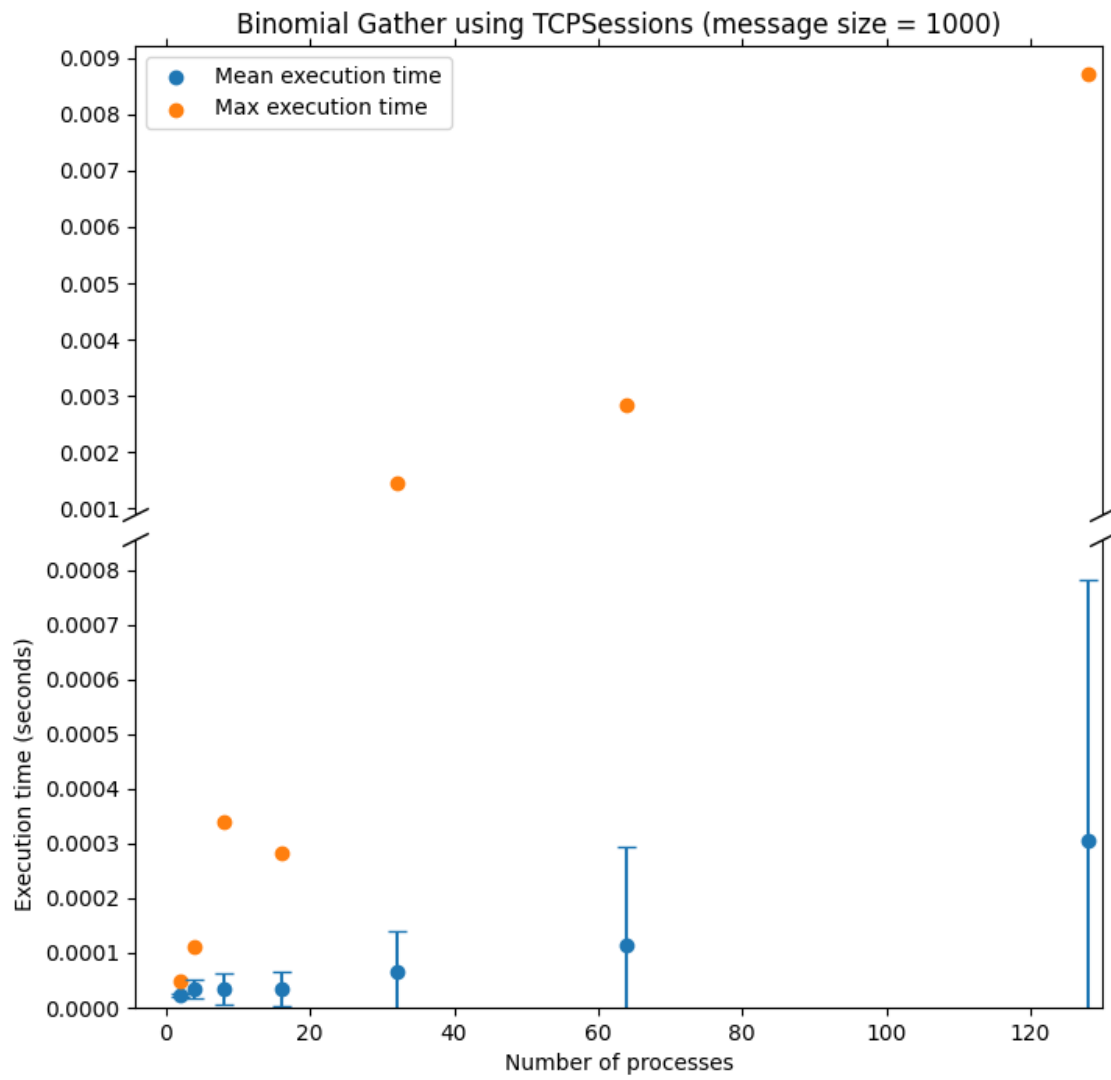


Figure A.13 Binomial gather with buffer size 1000 on TCPSessions transport

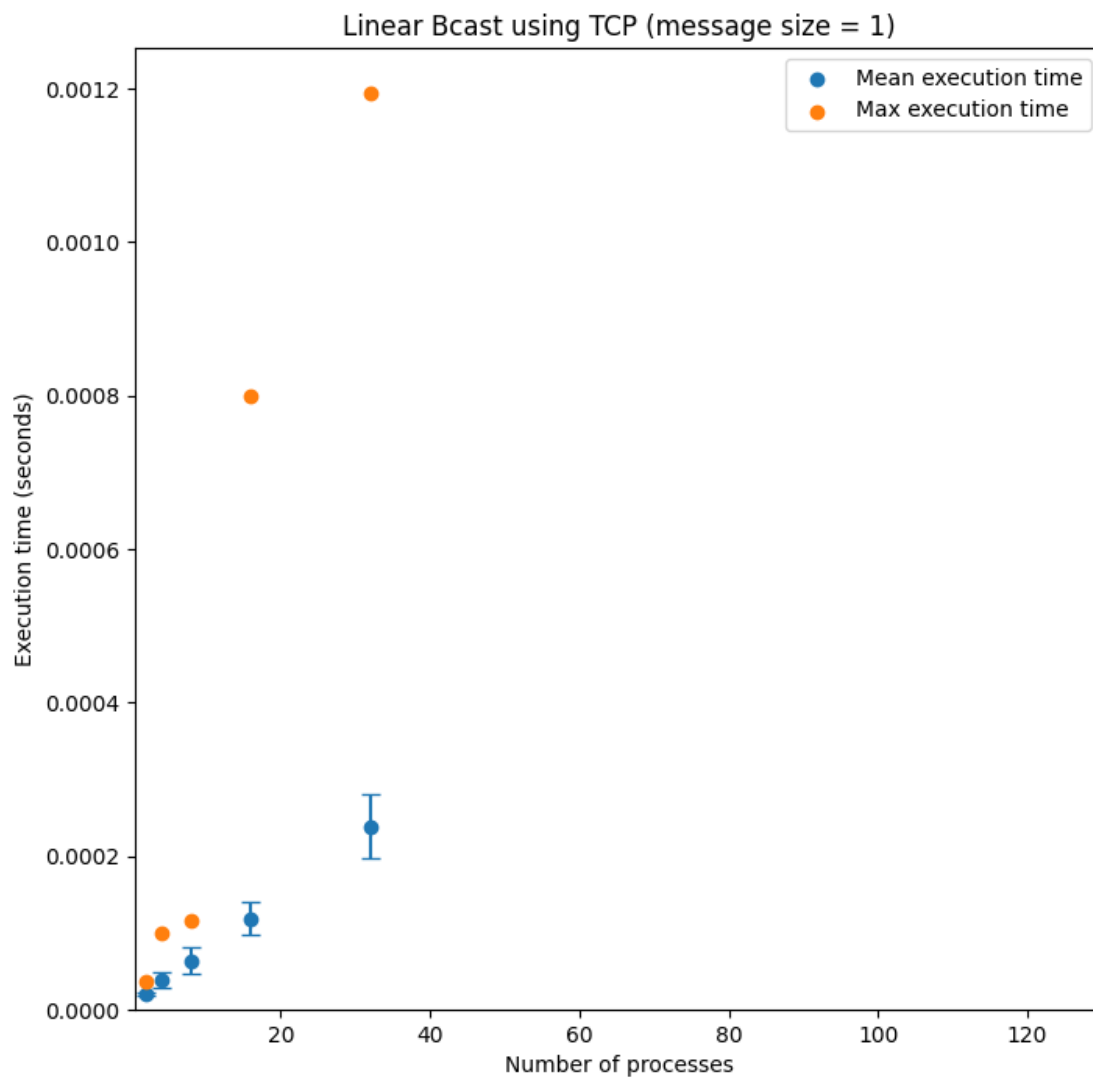


Figure A.14 Linear bcast with buffer size 1 on TCP transport

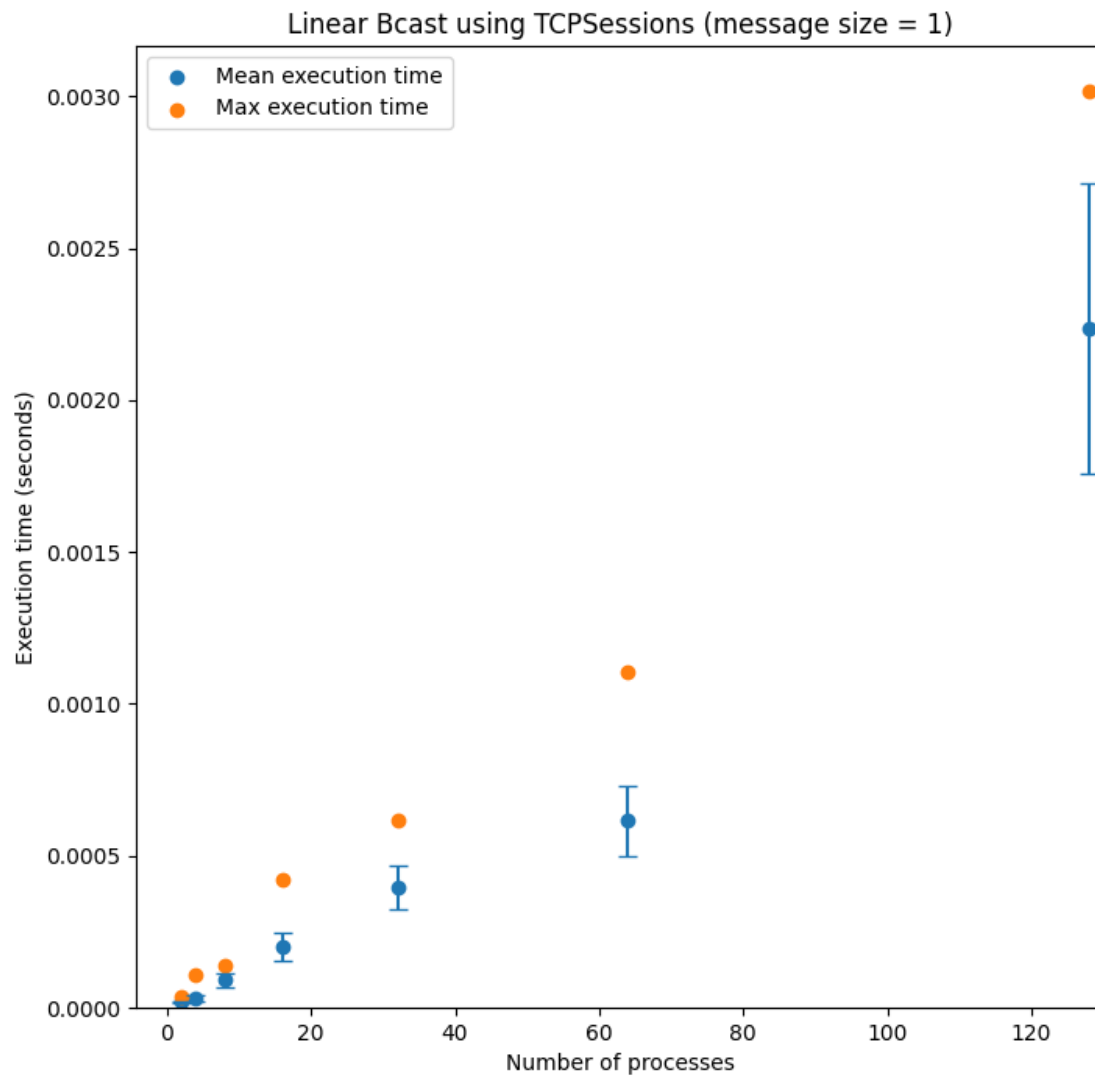


Figure A.15 Linear bcast with buffer size 1 on TCPSessions transport

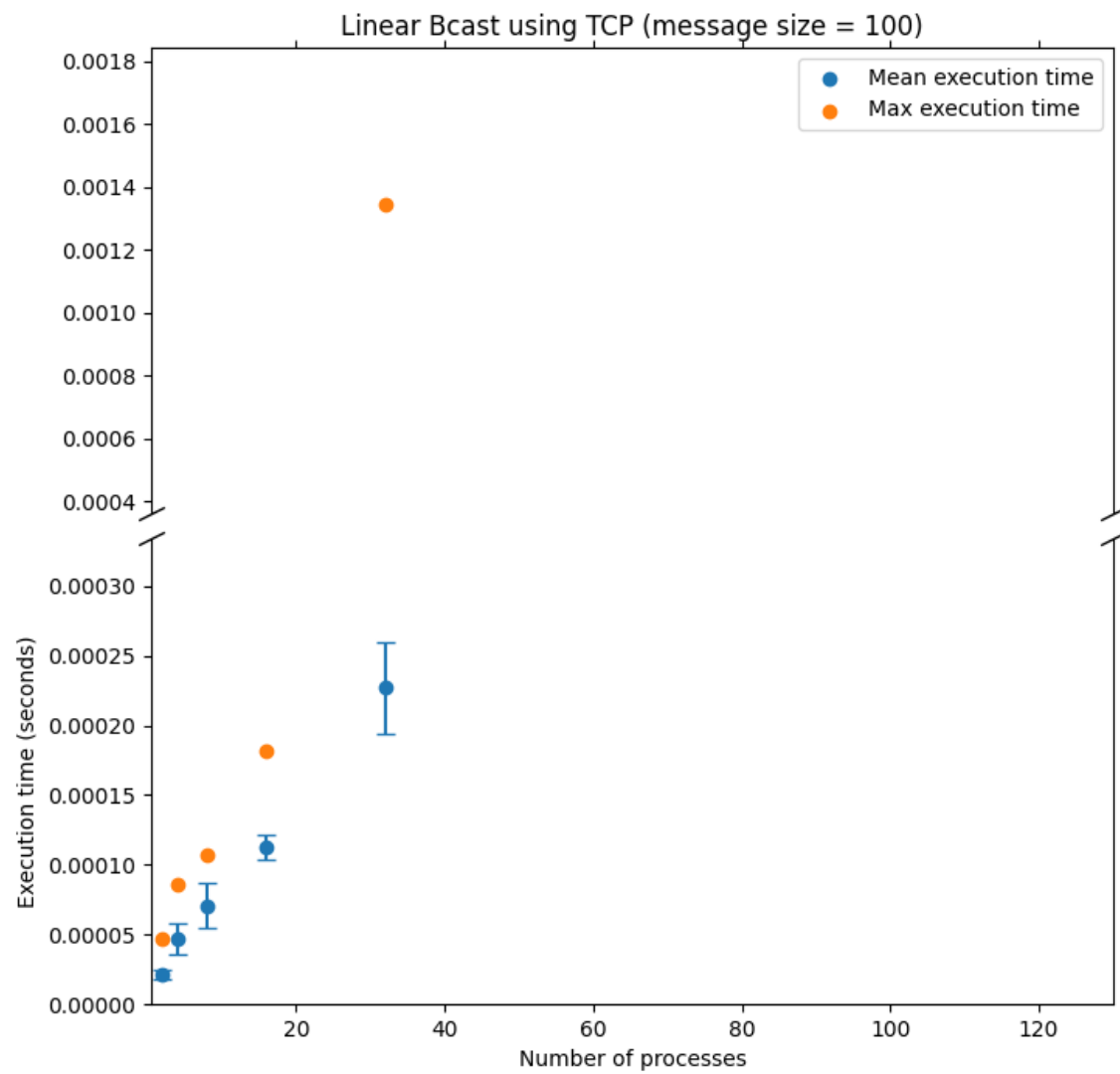


Figure A.16 Linear bcast with buffer size 100 on TCP transport

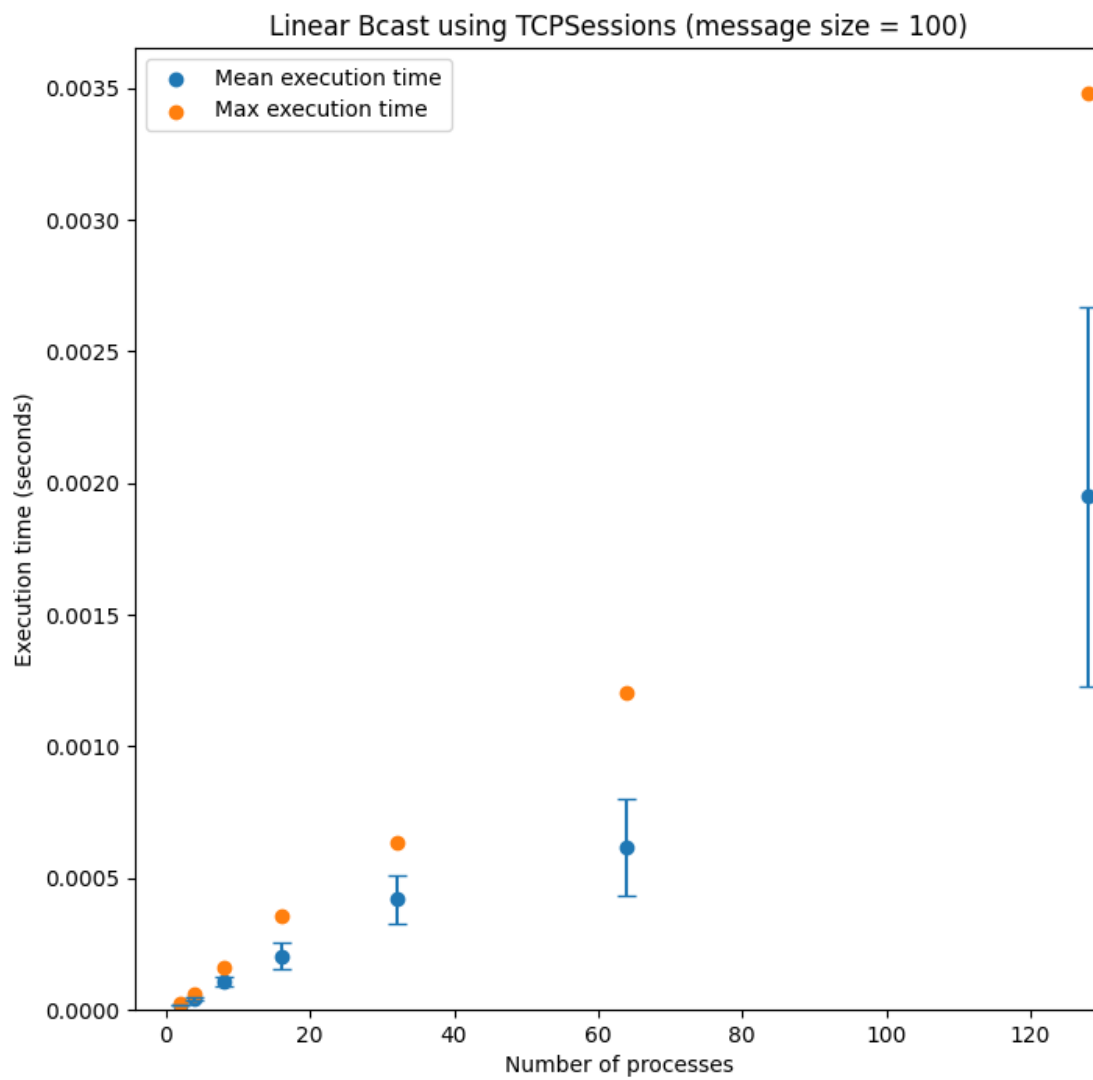


Figure A.17 Linear bcast with buffer size 100 on TCPSessions transport

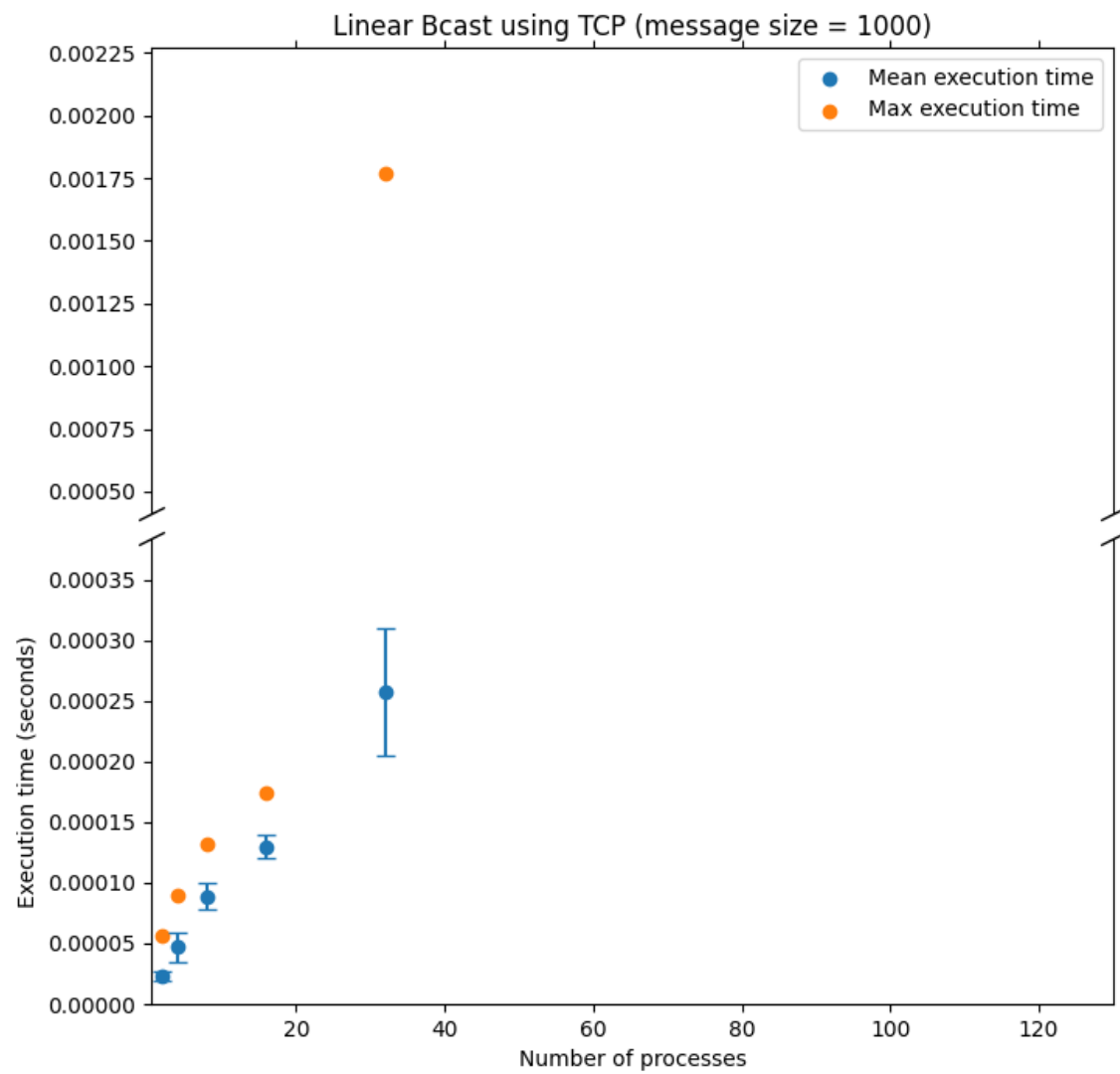


Figure A.18 Linear bcast with buffer size 1000 on TCP transport

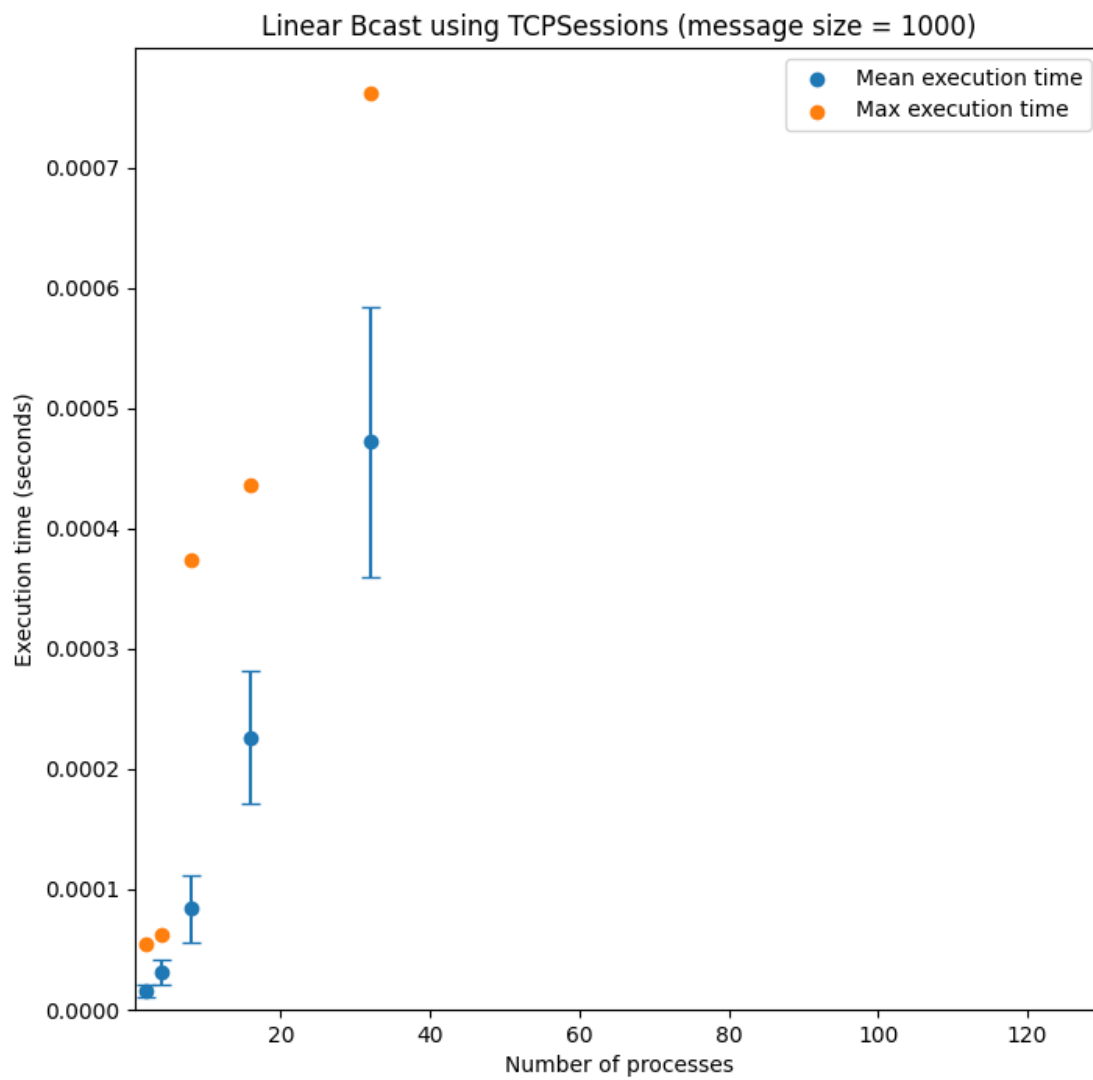


Figure A.19 Linear bcast with buffer size 1000 on TCPSessions transport

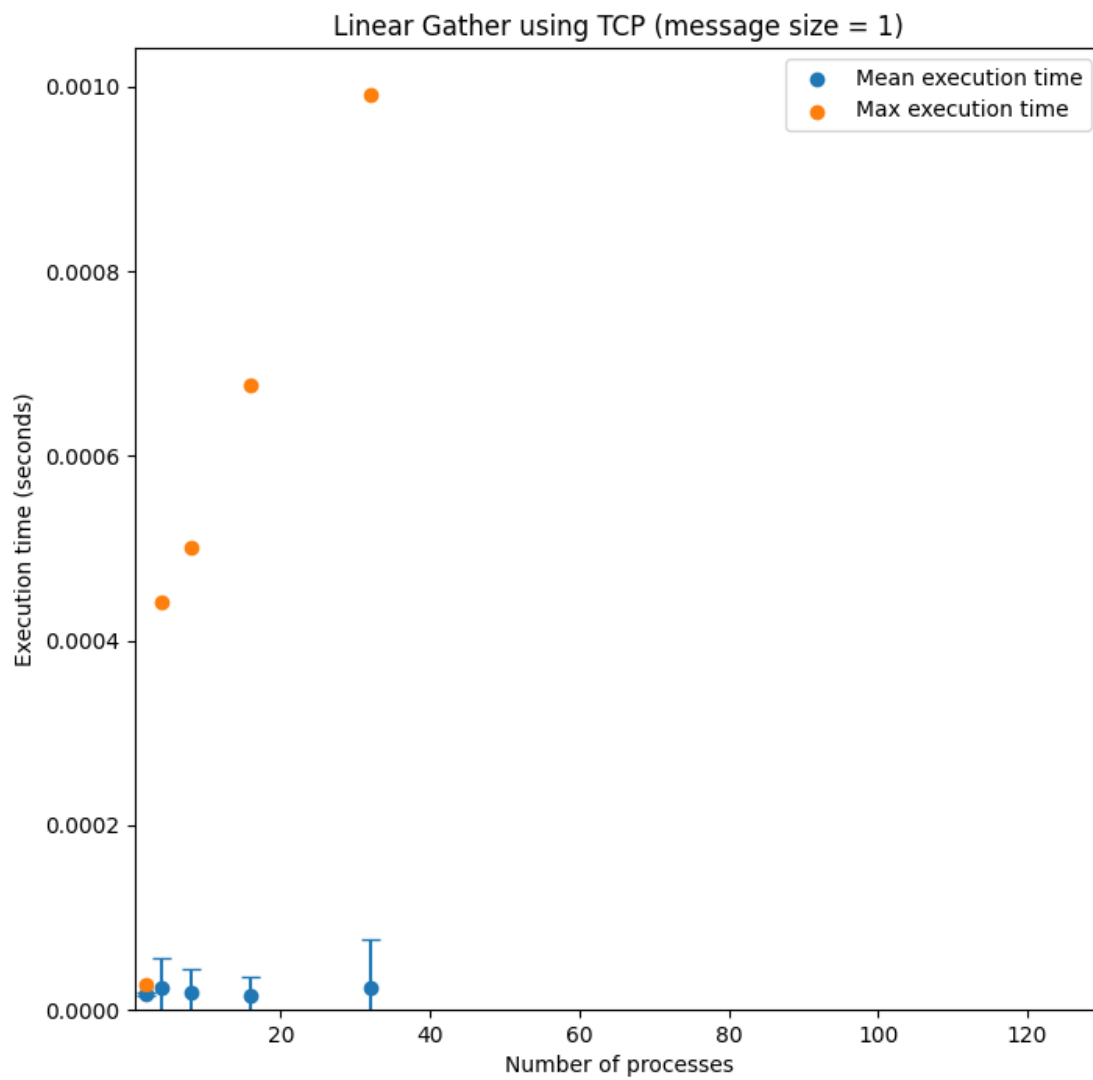


Figure A.20 Linear gather with buffer size 1 on TCP transport

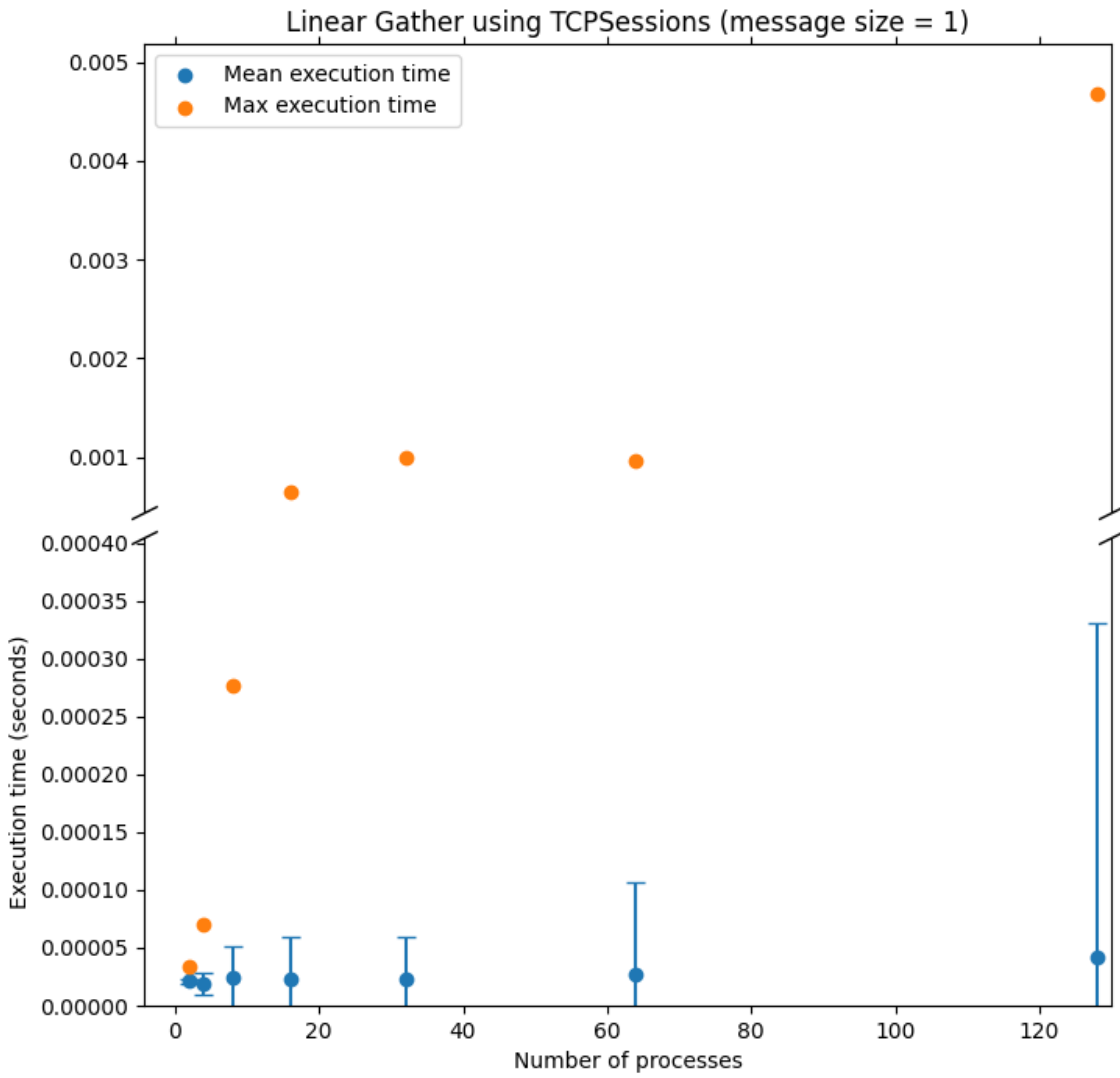


Figure A.21 Linear gather with buffer size 1 on TCPSessions transport

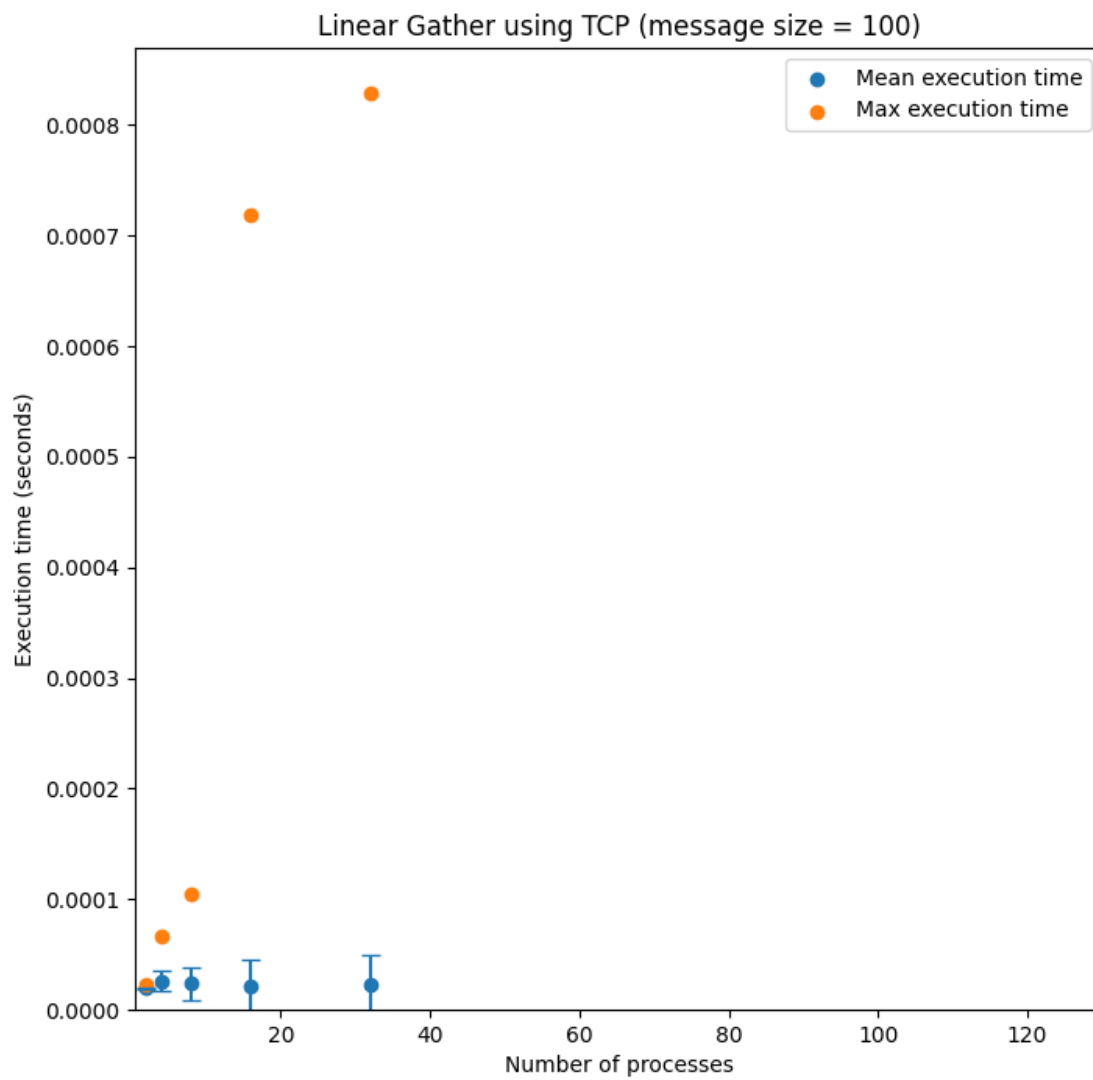


Figure A.22 Linear gather with buffer size 100 on TCP transport

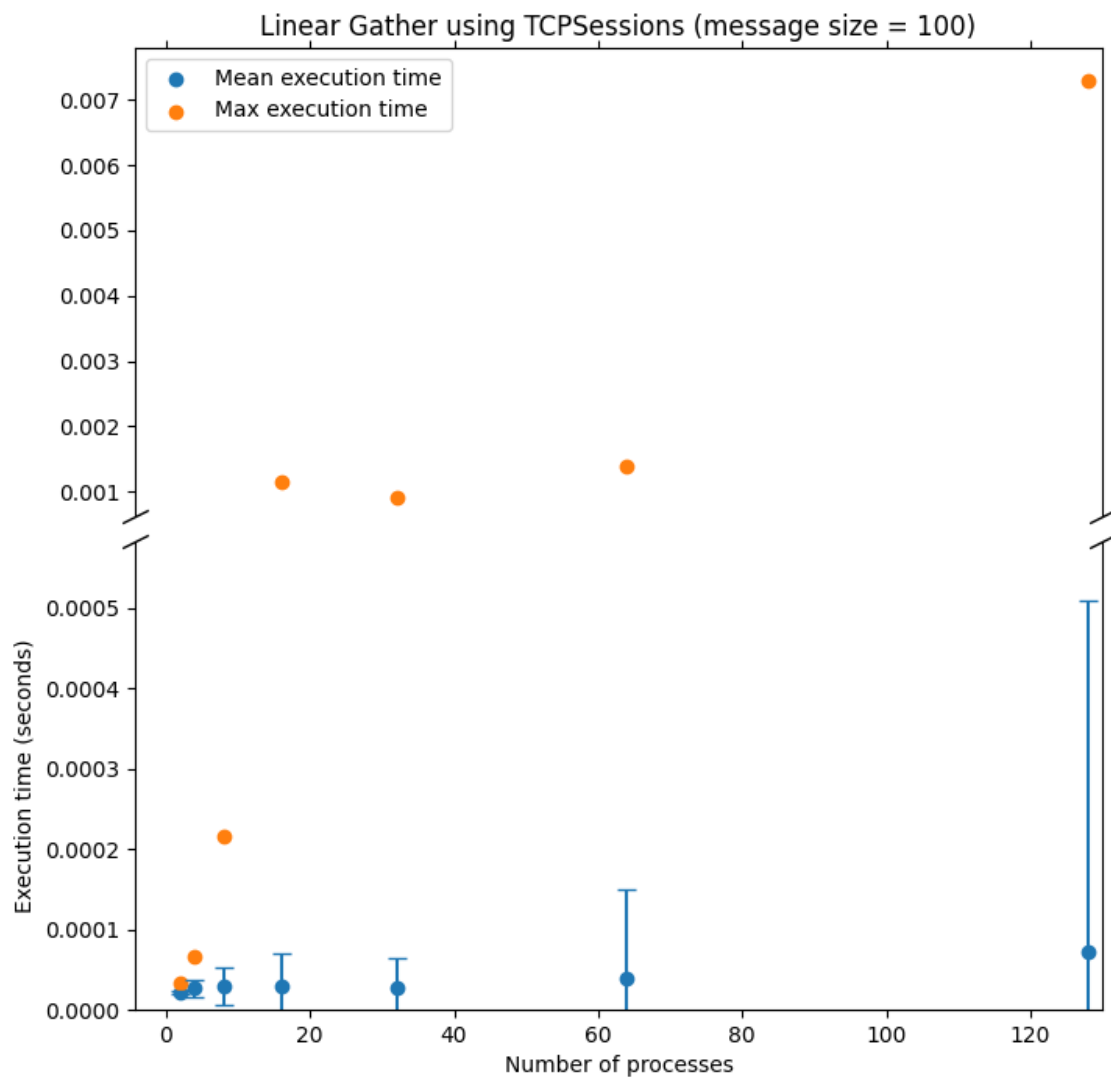


Figure A.23 Linear gather with buffer size 100 on TCPSessions transport

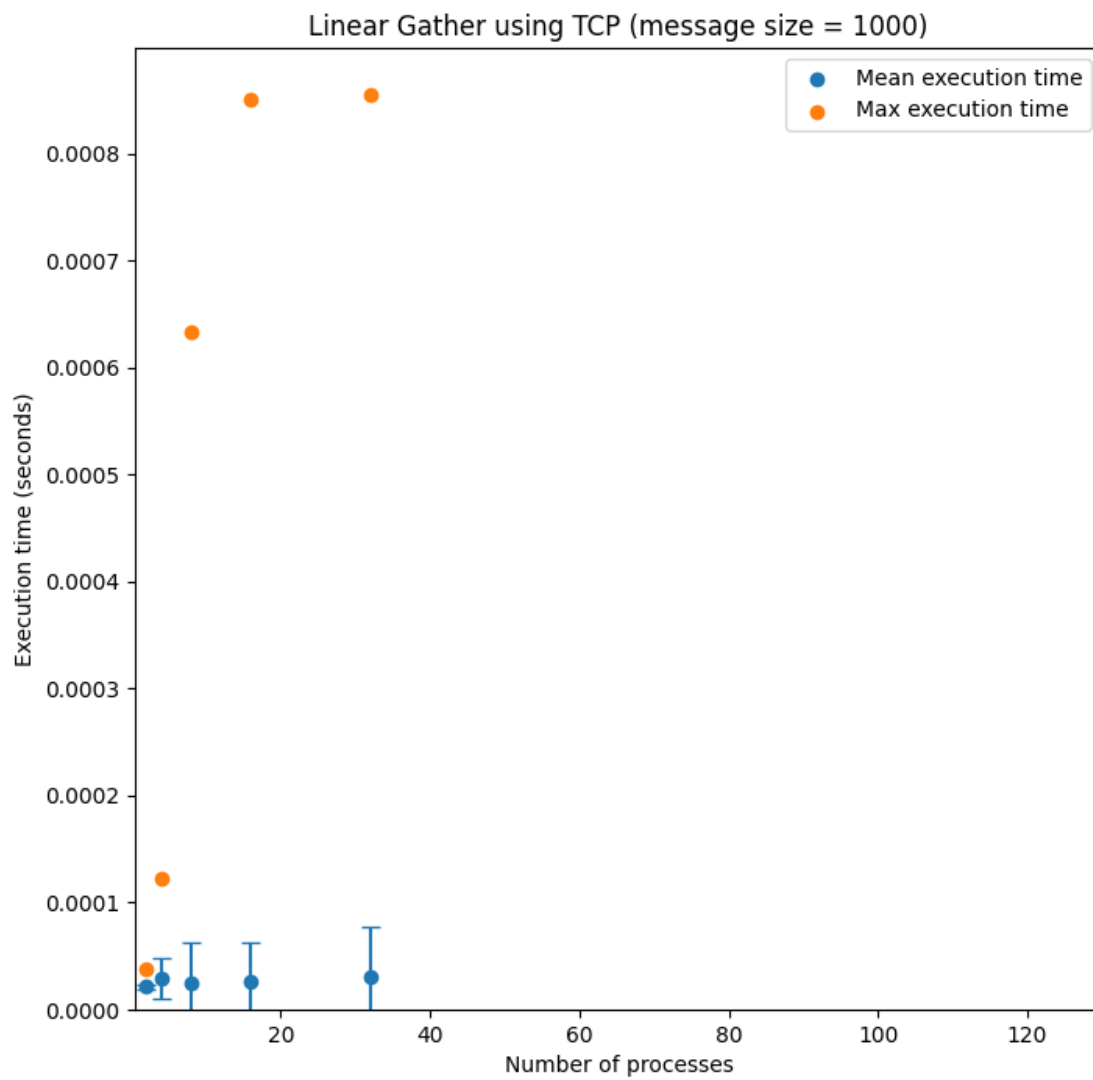


Figure A.24 Linear gather with buffer size 1000 on TCP transport

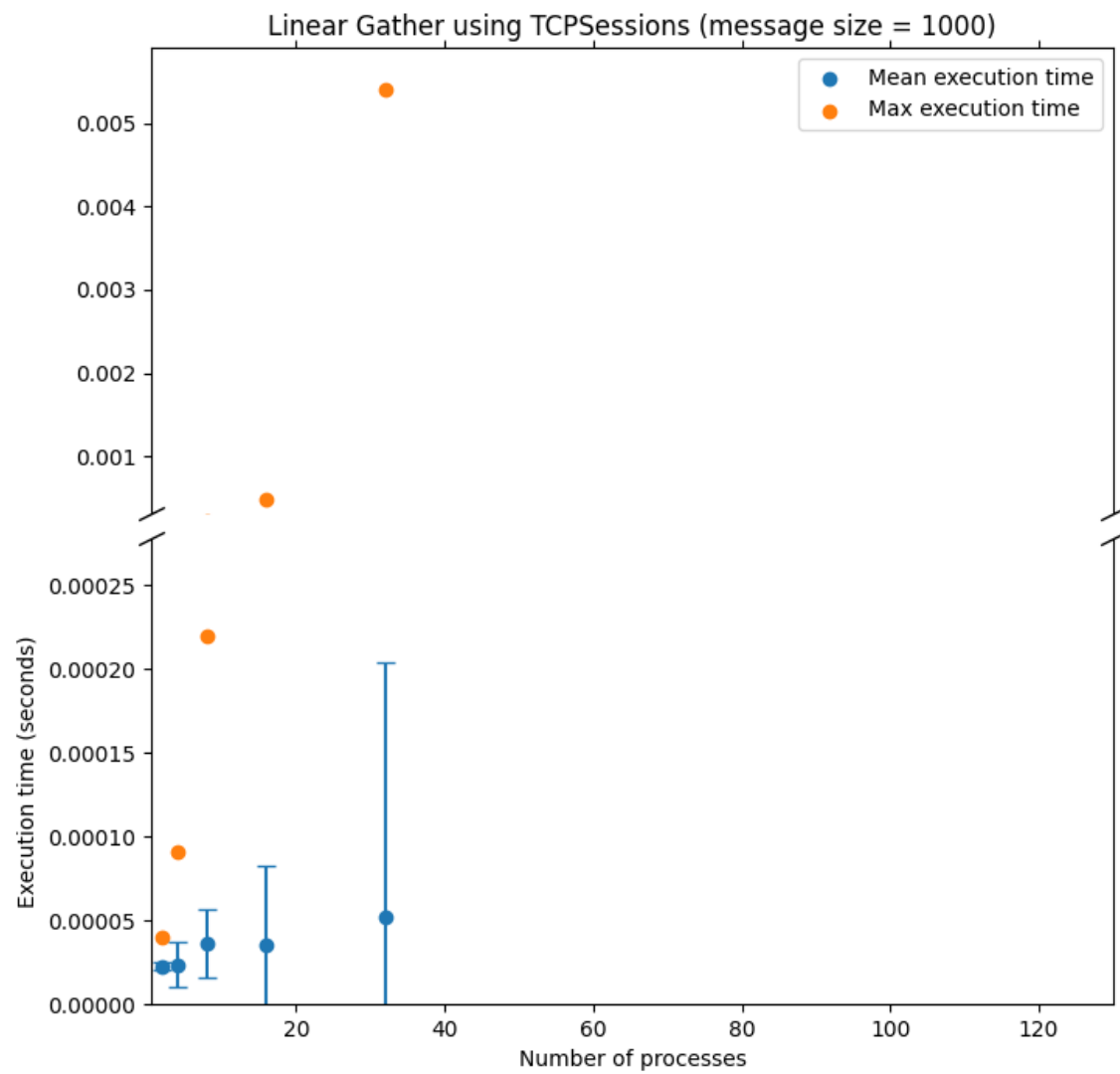


Figure A.25 Linear gather with buffer size 1000 on TCPSessions transport

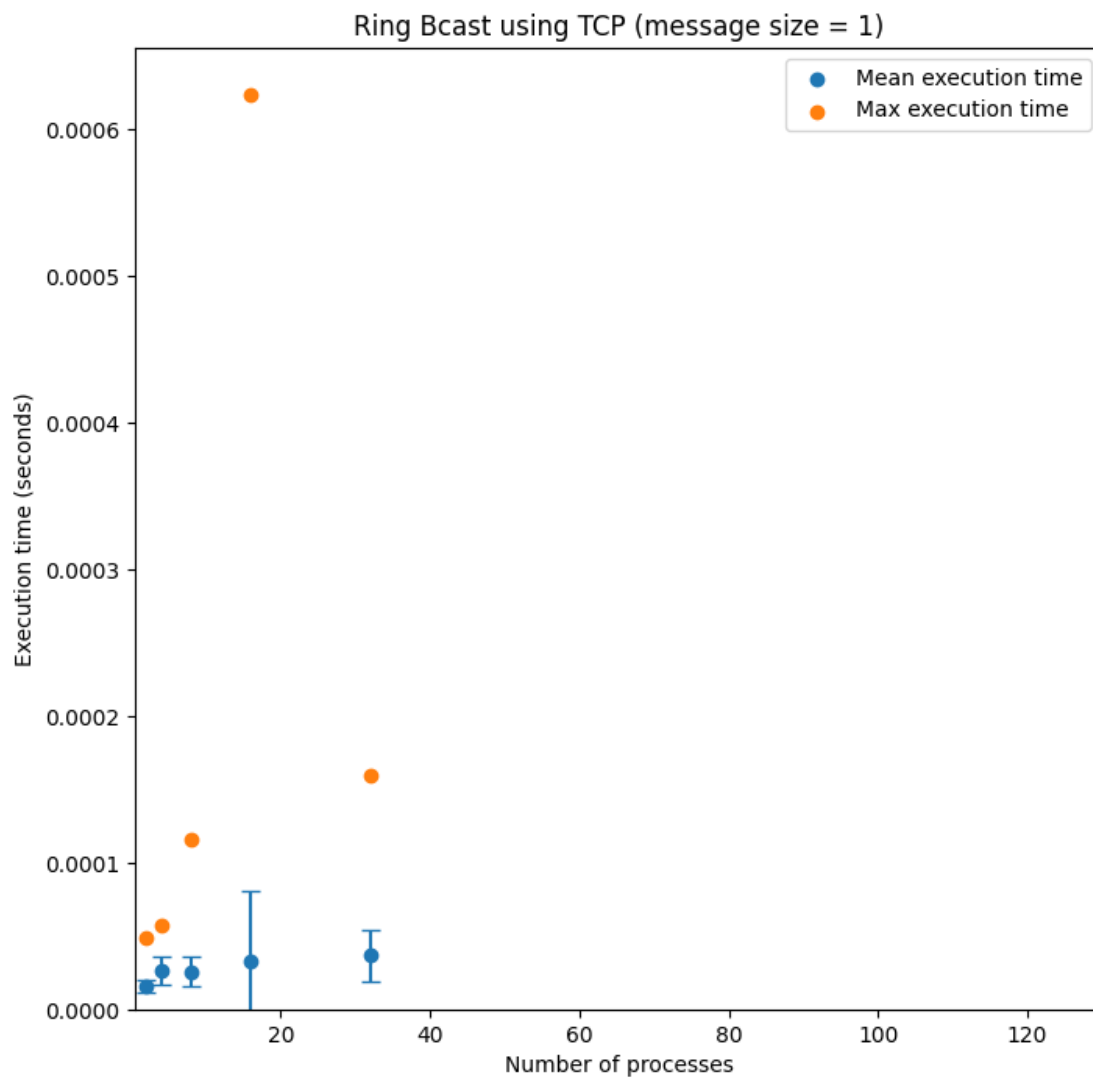


Figure A.26 Ring bcast with buffer size 1 on TCP transport

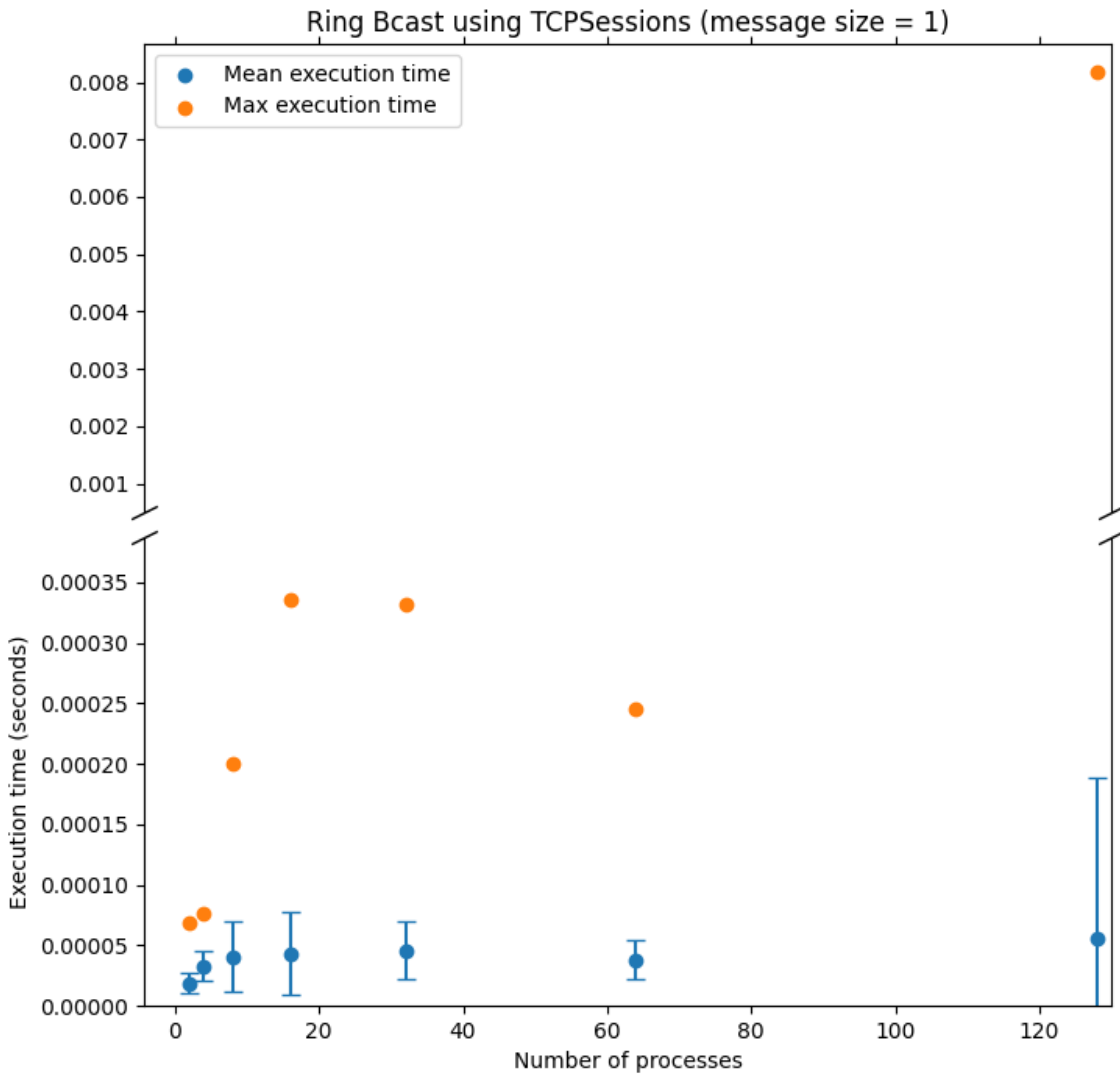


Figure A.27 Ring bcast with buffer size 1 on TCPSessions transport

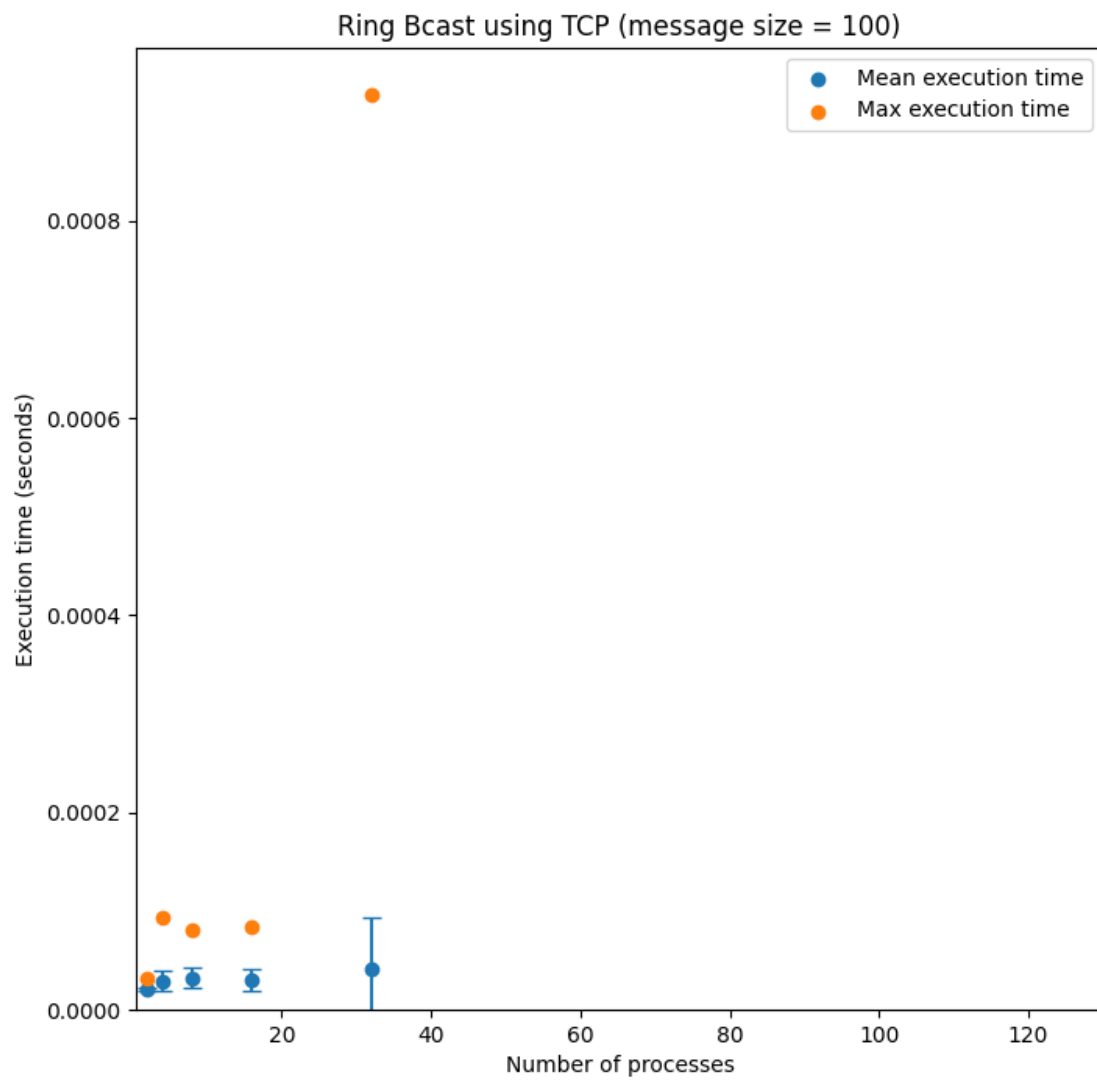


Figure A.28 Ring bcast with buffer size 100 on TCP transport

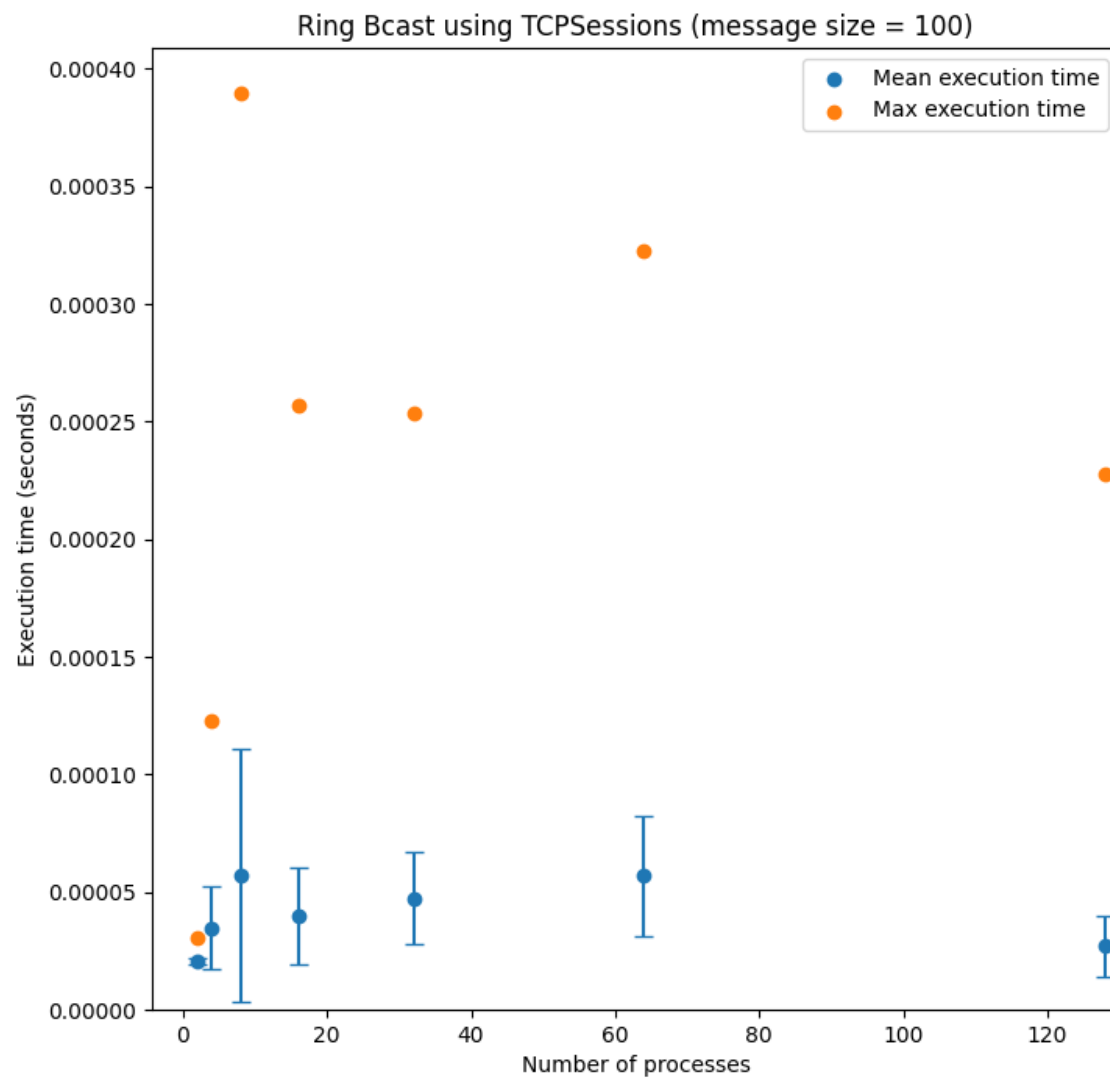


Figure A.29 Ring bcast with buffer size 100 on TCPSessions transport

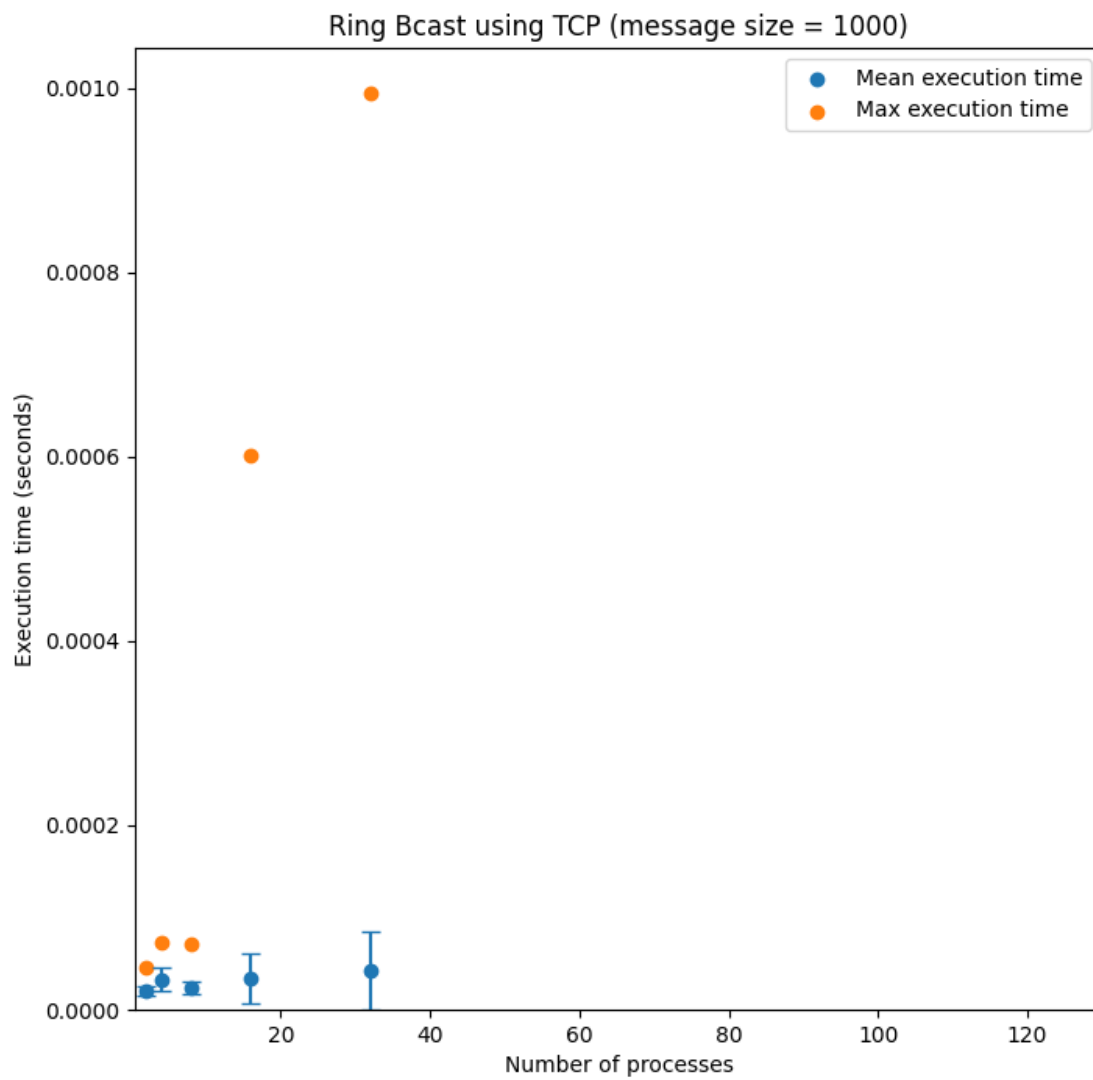


Figure A.30 Ring bcast with buffer size 1000 on TCP transport

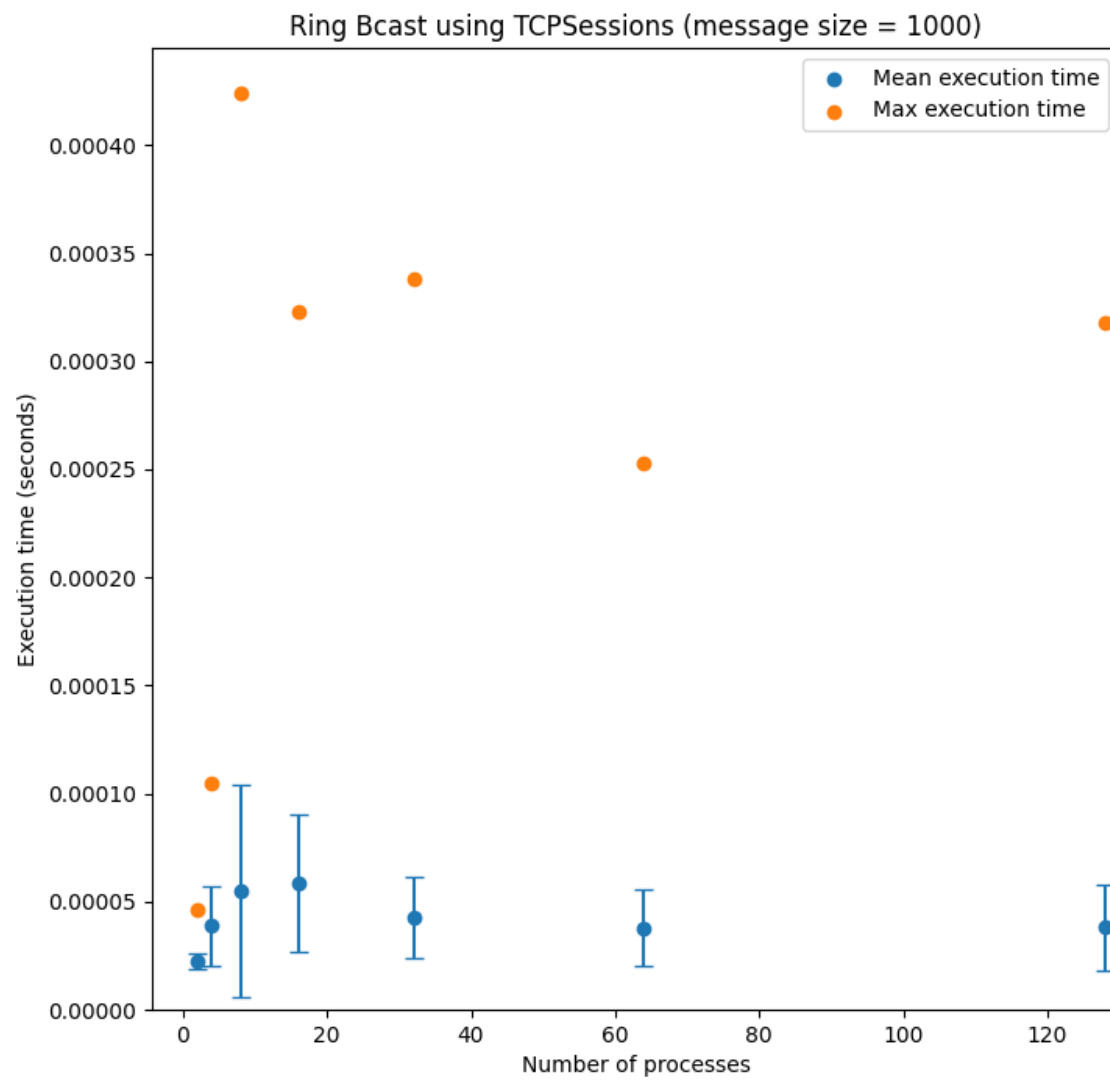


Figure A.31 Ring bcast with buffer size 1000 on TCPSessions transport

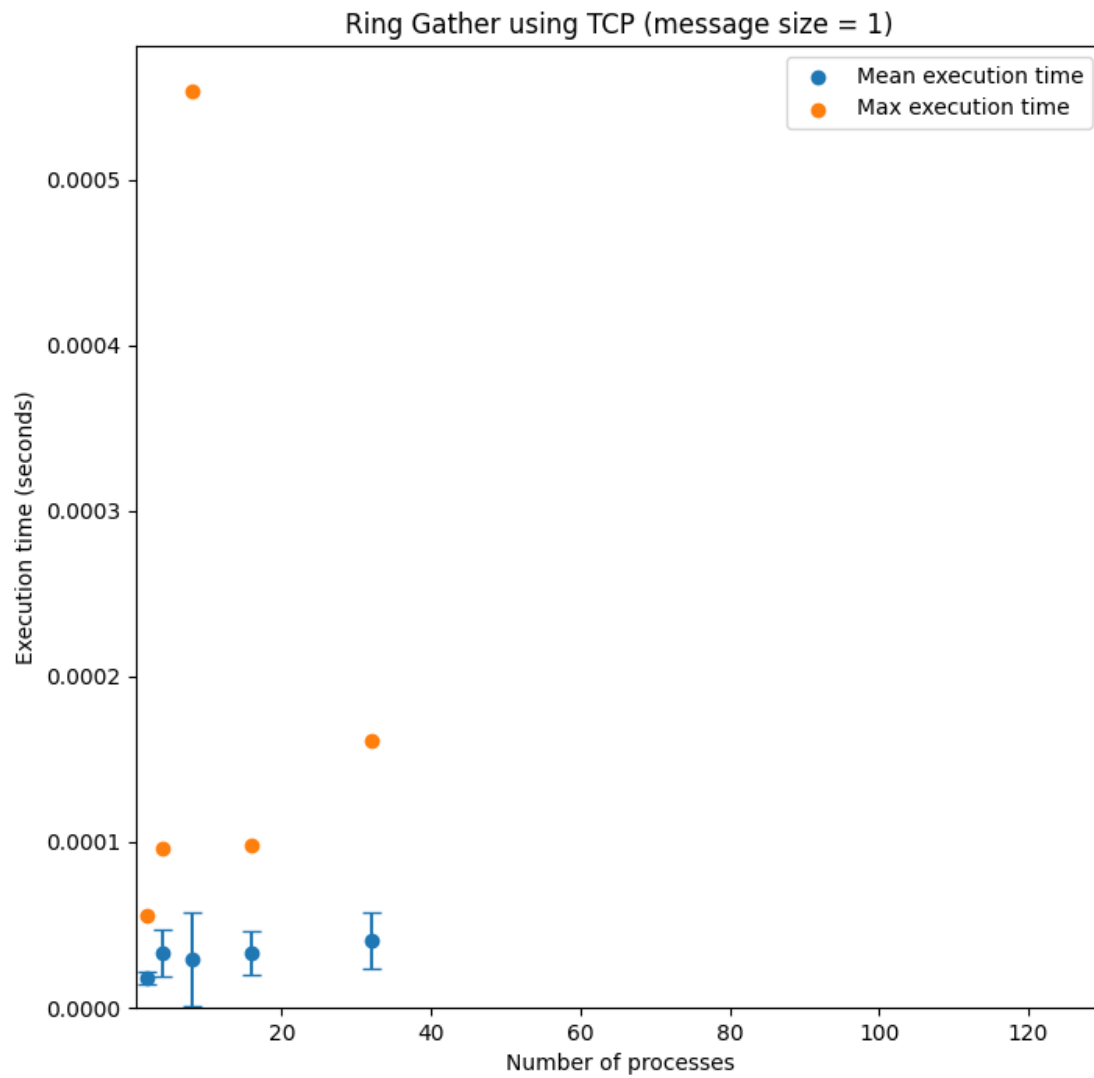


Figure A.32 Ring gather with buffer size 1 on TCP transport

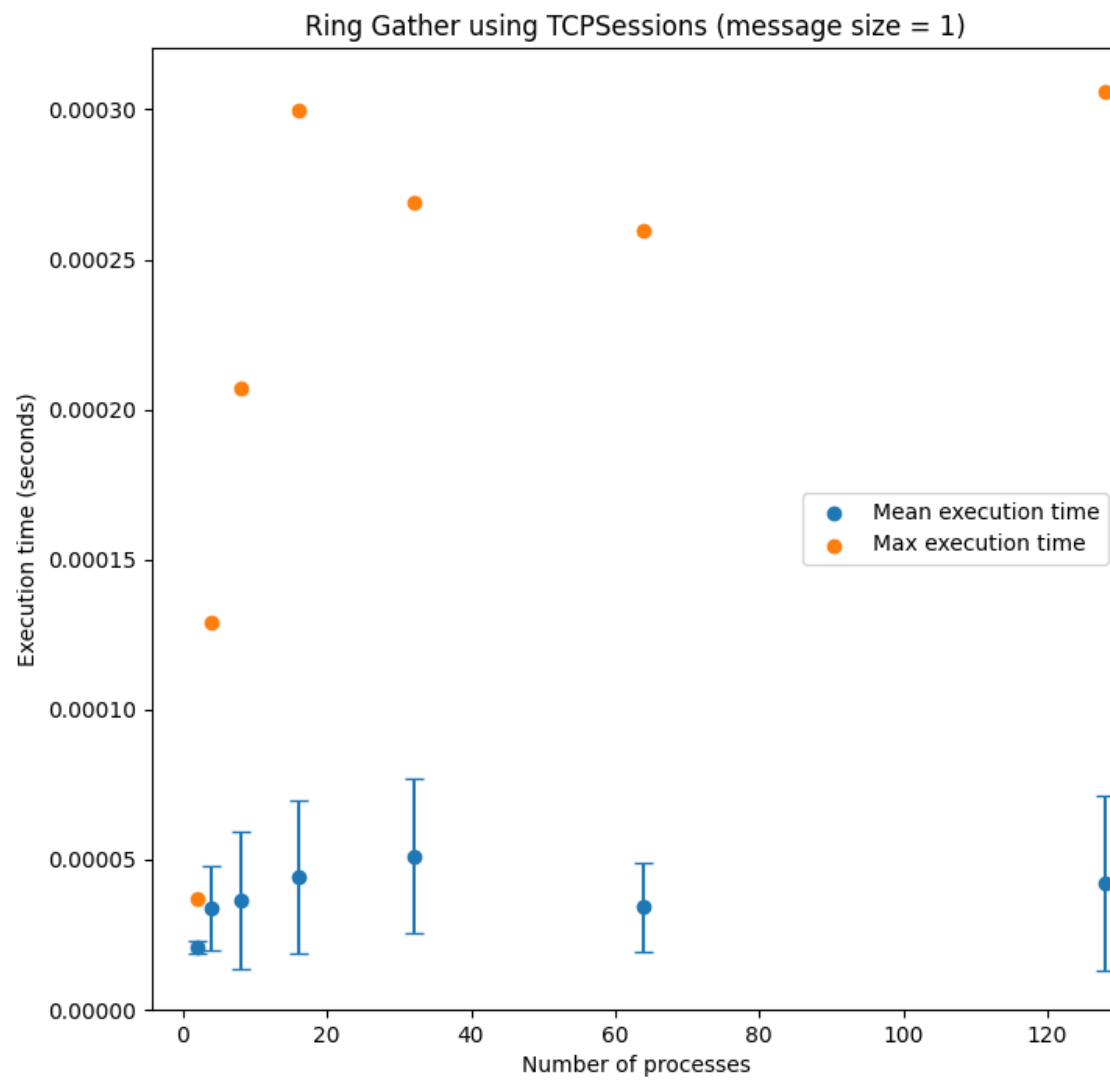


Figure A.33 Ring gather with buffer size 1 on TCPSessions transport

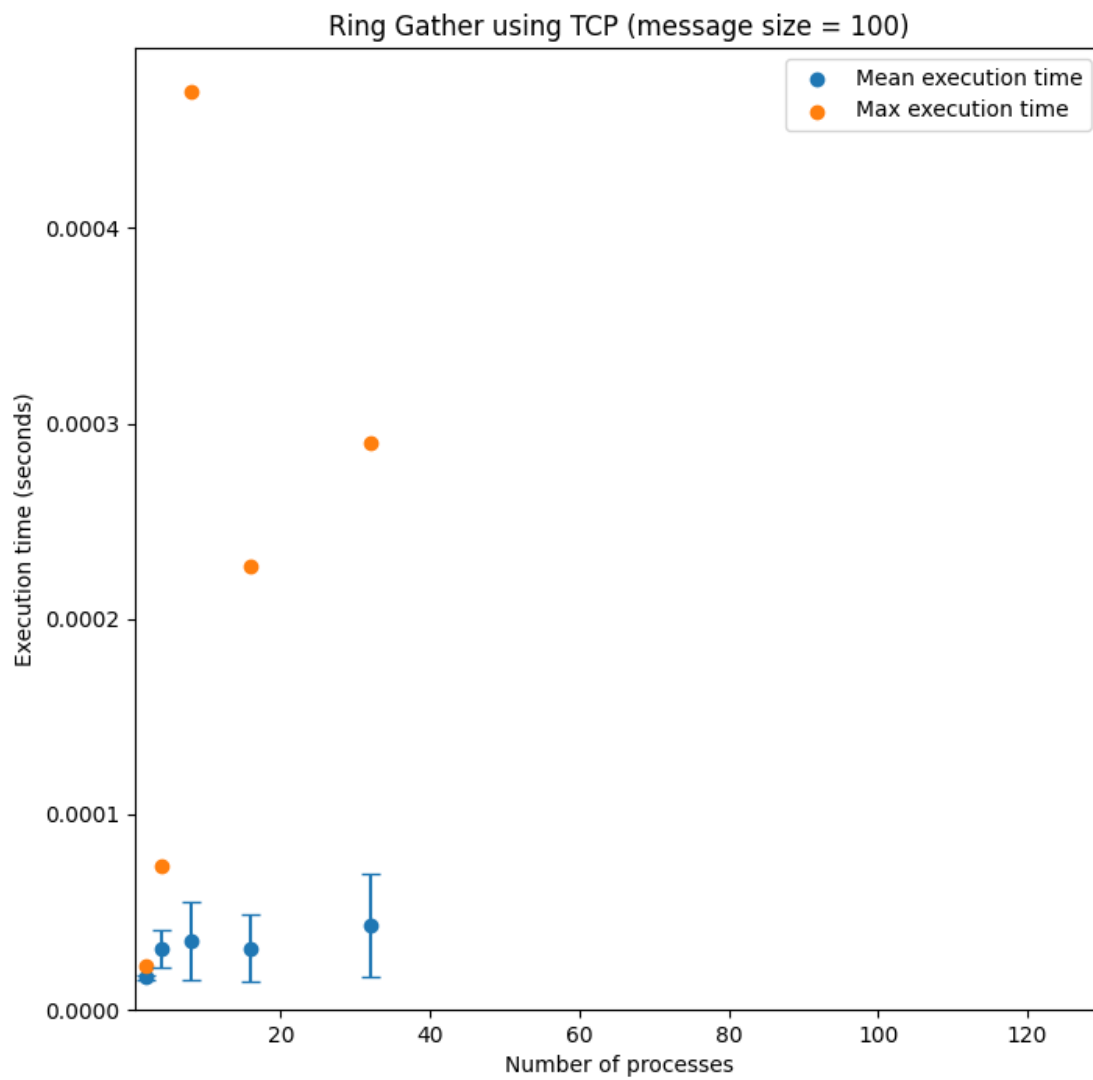


Figure A.34 Ring gather with buffer size 100 on TCP transport

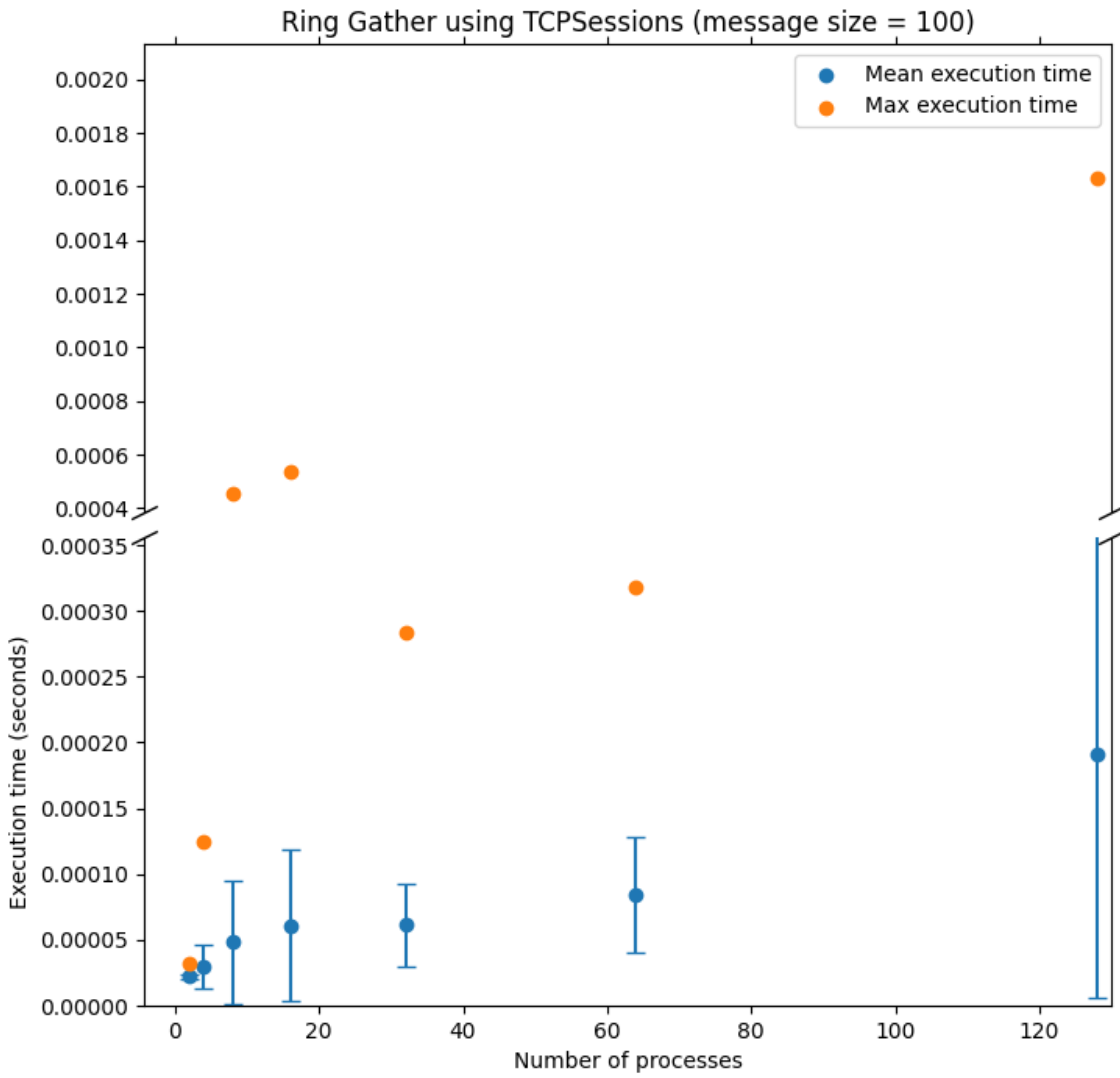


Figure A.35 Ring gather with buffer size 100 on TCPSessions transport

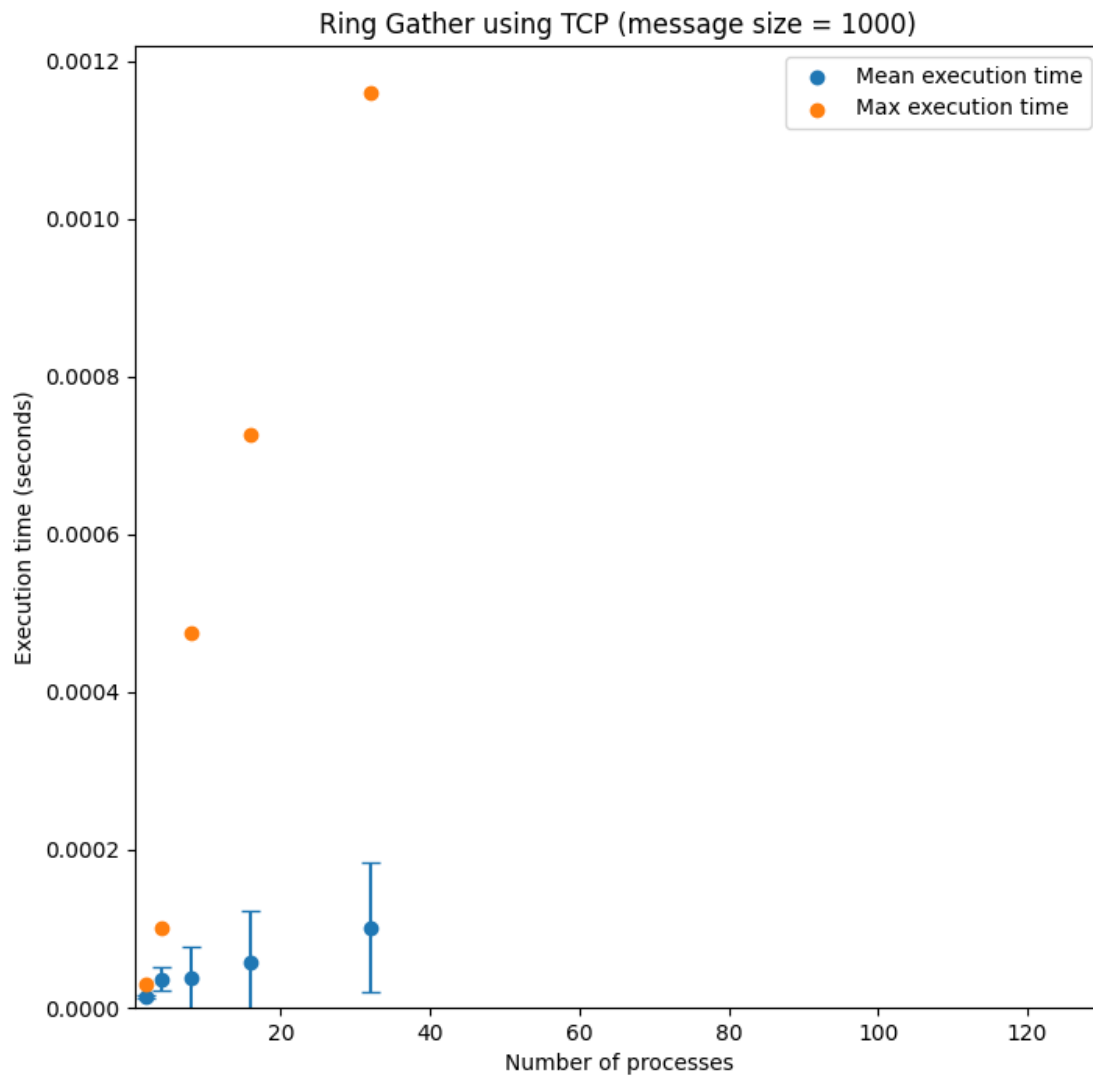


Figure A.36 Ring gather with buffer size 1000 on TCP transport

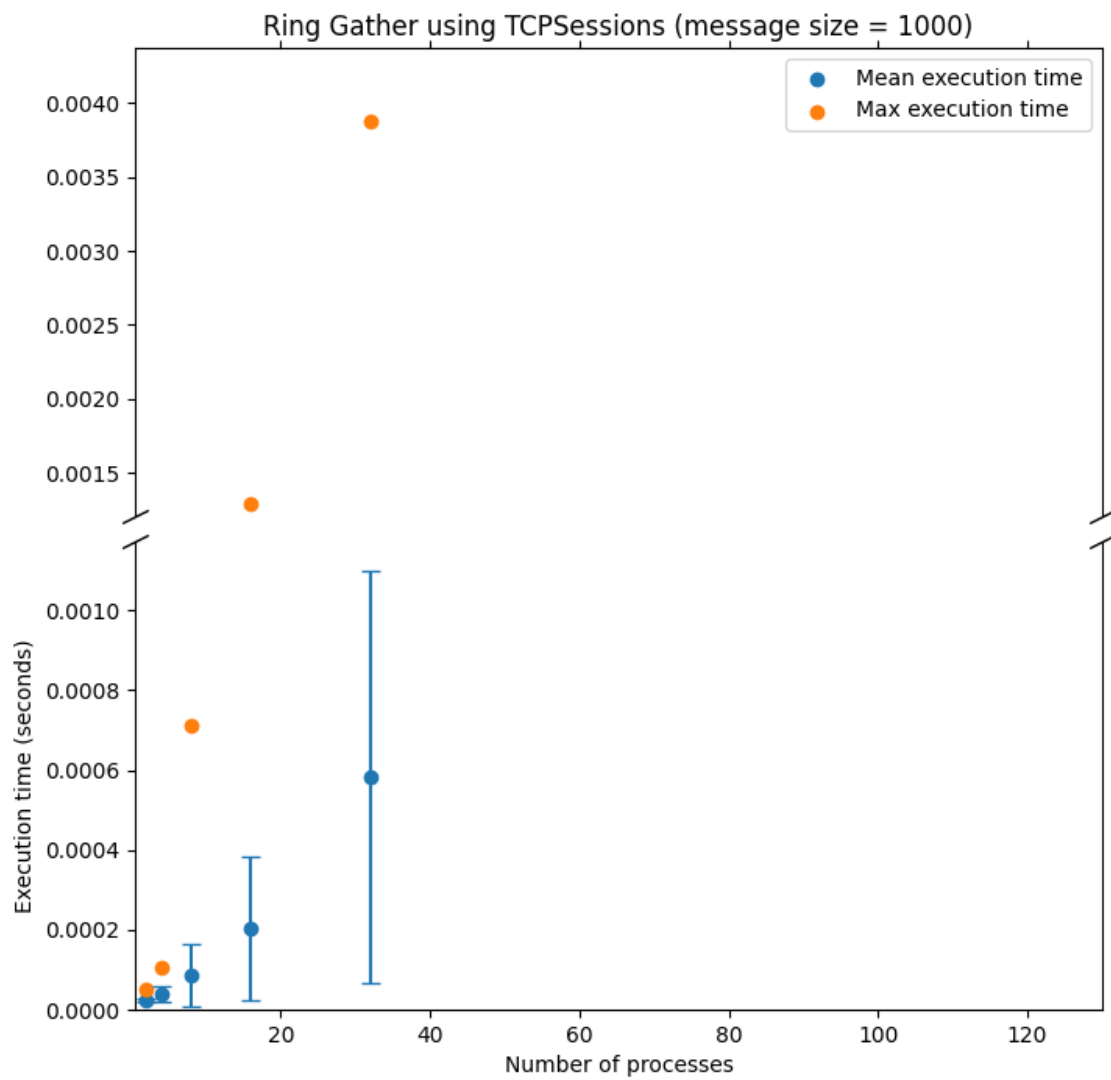


Figure A.37 Ring gather with buffer size 1000 on TCPSessions transport

APPENDIX B
BENCHMARKS

```
int main(int argc, char **argv)
{
    double start, end;
    start = MPI_Wtime();
    MPI_Init(&argc, &argv);
    end = MPI_Wtime();
    std::cout.precision(8);
    std::cout << std::fixed;
    std::cout << end - start << "\n";
    MPI_Finalize();
}
```

Figure B.1 Timing benchmark for initialization

```

int main(int argc, char **argv)
{
    double bcaststart, bcastend;
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int root = 0;
    int num_runs = 100;
    int count = 1;
    int my_num[count];
    for(int i = 1; i <= count; i++)
    {
        my_num[i-1] = rank*i;
    }
    MPI_Request req;
    MPI_Bcast_init(&my_num, count, MPI_INT, root, MPI_COMM_WORLD, &req);

    std::cout.precision(8);
    std::cout << std::fixed;
    for(int i = 0; i < num_runs; i++)
    {
        bcaststart = MPI_Wtime();
        MPI_Start(&req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        bcastend = MPI_Wtime();
        std::cout << bcastend - bcaststart << "\n";
    }

    MPI_Request_free(&req);
    MPI_Finalize();
}

```

Figure B.2 Timing benchmark for bcast operation with buffer size of 1

```

int main(int argc, char **argv)
{
    double bcaststart, bcastend;
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int root = 0;
    int num_runs = 100;
    int count = 100;
    int my_num[count];
    for(int i = 1; i <= count; i++)
    {
        my_num[i-1] = rank*i;
    }
    MPI_Request req;
    MPI_Bcast_init(&my_num, count, MPI_INT, root, MPI_COMM_WORLD, &req);

    std::cout.precision(8);
    std::cout << std::fixed;
    for(int i = 0; i < num_runs; i++)
    {
        bcaststart = MPI_Wtime();
        MPI_Start(&req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        bcastend = MPI_Wtime();
        std::cout << bcastend - bcaststart << "\n";
    }

    MPI_Request_free(&req);
    MPI_Finalize();
}

```

Figure B.3 Timing benchmark for bcast operation with buffer size of 100


```

int main(int argc, char **argv)
{
    double bcaststart, bcastend;
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int root = 0;
    int num_runs = 100;
    int count = 1000;
    int my_num[count];
    for(int i = 1; i <= count; i++)
    {
        my_num[i-1] = rank*i;
    }
    MPI_Request req;
    MPI_Bcast_init(&my_num, count, MPI_INT, root, MPI_COMM_WORLD, &req);

    std::cout.precision(8);
    std::cout << std::fixed;
    for(int i = 0; i < num_runs; i++)
    {
        bcaststart = MPI_Wtime();
        MPI_Start(&req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        bcastend = MPI_Wtime();
        std::cout << bcastend - bcaststart << "\n";
    }

    MPI_Request_free(&req);
    MPI_Finalize();
}

```

Figure B.4 Timing benchmark for bcast operation with buffer size of 1000

```

int main(int argc, char **argv)
{
    double gatherstart, gatherend;
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int root = 0;
    int num_runs = 100;
    int count = 1;
    int my_num[count];
    for(int i = 1; i <= count; i++)
    {
        my_num[i-1] = rank*i;
    }
    MPI_Request req;
    MPI_Gather_init(&my_num, count, MPI_INT, root, MPI_COMM_WORLD, &req);

    std::cout.precision(8);
    std::cout << std::fixed;
    for(int i = 0; i < num_runs; i++)
    {
        gatherstart = MPI_Wtime();
        MPI_Start(&req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        gatherend = MPI_Wtime();
        std::cout << gatherend - gatherstart << "\n";
    }

    MPI_Request_free(&req);
    MPI_Finalize();
}

```

Figure B.5 Timing benchmark for gather operation with buffer size of 1

```

int main(int argc, char **argv)
{
    double gatherstart, gatherend;
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int root = 0;
    int num_runs = 100;
    int count = 100;
    int my_num[count];
    for(int i = 1; i <= count; i++)
    {
        my_num[i-1] = rank*i;
    }
    MPI_Request req;
    MPI_Gather_init(&my_num, count, MPI_INT, root, MPI_COMM_WORLD, &req);

    std::cout.precision(8);
    std::cout << std::fixed;
    for(int i = 0; i < num_runs; i++)
    {
        gatherstart = MPI_Wtime();
        MPI_Start(&req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        gatherend = MPI_Wtime();
        std::cout << gatherend - gatherstart << "\n";
    }

    MPI_Request_free(&req);
    MPI_Finalize();
}

```

Figure B.6 Timing benchmark for gather operation with buffer size of 100

```

int main(int argc, char **argv)
{
    double gatherstart, gatherend;
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int root = 0;
    int num_runs = 100;
    int count = 1000;
    int my_num[count];
    for(int i = 1; i <= count; i++)
    {
        my_num[i-1] = rank*i;
    }
    MPI_Request req;
    MPI_Gather_init(&my_num, count, MPI_INT, root, MPI_COMM_WORLD, &req);

    std::cout.precision(8);
    std::cout << std::fixed;
    for(int i = 0; i < num_runs; i++)
    {
        gatherstart = MPI_Wtime();
        MPI_Start(&req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        gatherend = MPI_Wtime();
        std::cout << gatherend - gatherstart << "\n";
    }

    MPI_Request_free(&req);
    MPI_Finalize();
}

```

Figure B.7 Timing benchmark for gather operation with buffer size of 1000