National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Computing Systems Laboratory

Performance Analysis and Modeling of Parallel Applications in Distributed Memory Architectures

Dimploma Thesis

Fotios Branikas

Supervisor: Georgios Goumas
Associate Professor NTUA

Athens, April 2024

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Ανάλυση και πρόβλεψη παράλληλων εφαρμογών σε αρχιτεκτονικές κατανεμημένης μνήμης

Διπλωματική Εργασία

Μπρανίκας Φώτιος

Επιβλέπων: Γεώργιος Γκούμας
Αναπληρωτής
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Απριλίου 2024.

.....	.....	.....
Γεώργιος Γκούμας	Νεκτάριος Κοζύρης	Διονύσιος Πνευματικάτος
Αναπληρωτής Καθηγητής, Ε.Μ.Π.	Καθηγητής, Ε.Μ.Π.	Καθηγητής, Ε.Μ.Π.

Αθήνα, Απρίλιος 2024

.....

Μπρανίκας Φώτιος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Table of Contents

List of Figures

Listings

List of Tables

Περίληψη

Η ανάπτυξη υπολογιστών υψηλής επίδοσης (High Performance Computing, HPC) είναι ένας τομέας της επιστήμης των υπολογιστών, που παρέχει λύσεις σε πολλά προβλήματα που αντιμετωπίζουν οι σύγχρονοι επιστήμονες και μηχανικοί. Ο χρόνος σε συστήματα HPC είναι συχνά ένας ακριβός πόρος. Για αυτόν τον λόγο, για να μεγιστοποιηθεί η χρήση τέτοιων συστημάτων, οι μηχανικοί και οι προγραμματιστές παράλληλου εφαρμογών, αναλύουν και αναζητούν βελτιστοποιήσεις στις αρχιτεκτονικές και τα παράλληλα προγράμματα. Για τον ίδιο λόγο, η σύνταξη μοντέλων επίδοσης είναι επίσης ωφέλιμη. Τα μοντέλα αυτά, μπορούν να παρέχουν πληροφορίες για τη λήψη διαφόρων αποφάσεων, χωρίς το κόστος που προκύπτει από την εκτέλεση ενός προγράμματος.

Η παρούσα διπλωματική εργασία παρουσιάζει μια εις βάθος ανάλυση της επίδοσης μιας οικογένειας παράλληλων εφαρμογών για μια αρχιτεκτονική κατανεμημένης μνήμης, καθώς και μια προσπάθεια σύνταξης ενός μοντέλου επίδοσης. Η τελευταία είναι μια αρκετά περίπλοκη διαδικασία που απαιτεί βαθιά κατανόηση των φαινομένων που μπορεί να συμβούν κατά την εκτέλεση ενός παράλληλου προγράμματος, γι' αυτό και συνοδεύτηκε από την προαναφερθείσα ανάλυση. Η σύνταξη ενός εξαιρετικά ακριβούς μοντέλου είναι εξαιρετικά χρήσιμο επίτευγμα, ωστόσο για έναν μηχανικό HPC, το ταξίδι που απαιτείται για αυτόν τον στόχο είναι από μόνο του μεγάλης σημασίας και εξίσου ωφέλιμο.

Λέξεις Κλειδιά: συστήματα παράλληλης επεξεργασίας, προγραμματιστικό μοντέλο ανταλλαγής μηνυμάτων, MPI, αρχιτεκτονικές κατανεμημένης μνήμης, computer cluster, ημι-εμπιρικά μοντέλα, μοντέλα δένδρων αποφάσεων, μέθοδοι ensemble.

Abstract

High performance computing (HPC) is an area of computer science, that provides solutions to a lot of problems present in contemporary sciences and engineering. Time on HPC systems is often a costly resource. For this reason, to maximize the usage of such systems, engineers and parallel software developers, analyze and seek optimizations in architectures and parallel programs. For the same reason, the compilation of predictive performance models is also of great benefit. Such models can provide insights that are useful for making various choices, without the costs that come with actually executing a program.

This thesis presents an in-depth performance analysis of a family of parallel applications for a distributed memory architecture, as well as an attempt at the compilation of a predictive performance model. The latter is quite a complex procedure that requires a deep understanding of the phenomena that may occur during the execution of a parallel program, which is why it was accompanied by the formerly mentioned analysis. The compilation of a highly accurate model can be a great and highly useful achievement, however for an HPC engineer, the journey required for this goal is itself of great importance.

Keywords: parallel processing systems, message passing programming model, MPI, distributed memory architectures, computer cluster, semi-empirical models, decision tree models, ensemble method models.

Introduction

The development of High Performance Computing (HPC) has had a significant impact in the resolution of various complex problems of modern sciences. It finds usages in a plethora of sectors including machine learning and artificial intelligence, intricate multivariable physics simulations, climate change and genomics. Particularly as we near the physical boundaries of Moore's Law, gaining performance just by increasing the number of transistors is a nonviable strategy. This has led to the development of elaborate multicore and accelerator architectures that are used in both HPC systems and consumer electronics.

A fundamental classification for multicore HPC systems, has to do with the way the memory is organised. In **Shared Memory Systems**, multiple processing cores are attached to a common memory bank. For these types of systems, a deeper categorization can be made. Namely, if all processing cores are evenly and exclusively connected to the shared memory, then the system has Uniform Memory Access (UMA). On the other hand, if parts of the memory system are closer to some processing cores, then the system has Non-Uniform Memory Access (NUMA). In many cases, shared memory systems are used with a global address space for all processes.

In **Distributed Memory Systems** multiple processing units with their own private memory hierarchy, are connected using an interconnection network. In most use cases, each processing unit uses a private address space. This absence of shared memory, leads to data sharing between processing units through Message Passing on the interconnection network. Another widely used architecture is a hybrid of the two mentioned above. In this case, multiple shared memory systems are connected using an interconnection network, thus forming a distributed memory system, commonly known as a **Computer Cluster**.

An essential limitation in all these kinds of systems is the bottleneck created by the disparity in speed between data transfers and computations. In most modern systems, the processing units can perform computations on data with a much greater speed than the rate at which data can reach them through the memory bus. This problem is also present in distributed memory systems, where message passing can cause congestion in both the interconnection network and in the (often shared) memory systems of the nodes of a cluster, as it is a memory intensive task.

Apart from the development of the hardware architecture, the study and analysis of parallel algorithms and their performance, is also of great importance. It can

provide an engineer with great insights into how performance varies across different architectures, and lead them to code optimizations. One can also move towards compiling performance prediction models. They can help in making the choice of the type of HPC system for a parallel program, without the cost of having to execute it. These models can be analytical, or rely on statistical regression and other machine learning techniques. For the latter, data collection for each system is needed, in order to train the model.

In this thesis, our attention is centered on the performance of parallel algorithms within distributed memory systems, specifically analyzing communication time and the factors influencing it in an HPC system. After reviewing the existing literature on the modeling (both analytical and empirical) and analysis of communication time, we experimented with semi-empirical and empirical models on ARIS, a Fat Tree cluster. The techniques employed for data collection and modeling, along with the insights derived from these processes and the performance of the resulting models, are all detailed in the subsequent chapters.

1. Related Work and Goals of Present Study

1.1 Related Work

Communication performance analysis and modeling have been a topic of interest ever since network communication became a necessity in distributed memory HPC systems. [Culler et al., 1993] proposed the analytical LogP model, with parameters that depend on the message size, the network bandwidth, processor overhead and the gap between messages. Although this model addresses some major issues that have to do with communication like limited bandwidth, it has limitations regarding the message size, global network topology and effects, as well as how local computation may affect communication (e.g. through cache effects or communication/computation overlap). Still, LogP's approach to adapting to different machine parameters, paved the way for a family of models which improved upon some of its deficiencies. For example the LogGP model [Alexandrov et al., 1995] introduced an additional parameter in the model, to account for longer messages, while LogGPS [Ino et al., 2001] added yet another parameter, for synchronization. Throughout the years, there have been several extensions to LogGP. However, contemporary network hardware and architecture, as well as the variety of communication patterns found in applications, make it more difficult to express communication performance analytically. This is a general problem with analytical models, as there is a clear tradeoff between model complexity and accuracy [Hoefler et al., 2011]. That being said, Hoefler et al. also demonstrated that they can be used for various optimizations throughout the lifecycle of an HPC system and/or application.

Another approach that has gained popularity in more recent years is empirical modeling. These models leverage data gathered from benchmarks or tailored "data generator" programs for training. As a result, they are able to capture more intricate conditions that can occur in an execution environment that may not be apparent in large-scale systems. [Papadopoulou et al., 2017] proposed a methodology for highly accurate predictive communication time modeling. This methodology involves sweeping a selected benchmark over a space of features. The features had to do with the application communication profile, the execution environment, and other machine-specific parameters. The resulting dataset was fed into a model building process that aimed to find the appropriate tree-based ensemble model for optimal performance while simultaneously avoiding overfiting. The previously mentioned tradeoff is also present in this methodology, where high-prediction accuracy comes at the cost of

model "transparency" (especially for those not very familiar with machine learning). Nevertheless, the great performance offered by such models makes them an appealing option for more complex systems.

Finally, it should also be noted that [Karapanagiotis, 2023], where the main focus was performance prediction in shared memory architectures, served as a starting point and an inspiration for the present study.

1.2 Goals of Present Study and Outline

Models are mechanisms that provide an estimation of a phenomenon. It is undeniable that designing a capable model requires a deep understanding of the selected phenomenon. Delivering a model that produces accurate results is commendable, but the journey that such a task requires is of equal importance. This thesis describes such a journey, for the phenomenon of communication time in a cluster computer system.

The first chapters are an examination of the execution environment and the measurement methods which include the compilation of a custom data generator application. Subsequently, some important case studies of the execution of this application are presented, in order to better understand the behaviour of a parallel application in a cluster environment, as well as to how this application may be deployed for data collection in a machine learning model. In the second half of this thesis, the focus shifts to modeling, where a simple, benchmark-based semi-empirical model is examined, before deploying a more advanced regression model. The performance of this model is analyzed, from a statistical perspective, as well as from a more practical standpoint, where we test the predictability for the BT pseudo application of the NAS parallel benchmarks.

2. Target Applications and Feature Space

2.1 Stencil Applications

This thesis concentrates on a family of applications known as stencil computations. In stencils, data access is regular. Parallel implementations usually involve partitioning an N-dimensional data grid. Each available process is assigned a subdivision of the data grid. A lot of these applications, feature an outer time loop. On each time iteration, processes perform computations and communications with other neighbouring processes. Communication is generally required for the exchange of data located at the boundaries of each process's working set.

An example of such an application is the Jacobi method for solving a strictly diagonally dominant system of linear equations. Listing 2.1 presents the pseudocode for an MPI implementation of the Jacobi kernel. The iterable *Neighbours* contains a list of the neighbouring processes, which for a 2D data grid, would be something among the lines of [north, west, south, east]. The *compute* function consists of the computational part of each time iteration and is performed for each element of the data grid. It depends on the neighbouring elements and their previous values. *MPI_ Waitall*, forces the process to wait for its message requests to be completed, before it can begin computation.

```
for time:
    for iNeighbour in Neighbours:
        MPI_Irecv(iNeighbourBorderData, iNeighbour)
        MPI_Isend(myBorderData, iNeighbour)

    MPI_Waitall(MessagesToSend,MessagesToRecv)

    for i in rows:
        for j in columns:
            compute(i, j, u_previous, u_current)
```

Listing 2.1 Pseudocode for an MPI implementation of the Jacobi kernel

Note about Listings

The Listings presented in this thesis, are simplified snippets of pseudocode. The actual programs were written in C using MPI.

The two distinct phases for communication and computation in the aforementioned implementation make it relatively easy to make code adjustments, so that the computation and the communication loads are configurable. For communication, the configurable parameters are the message size and the number of messages. The effects of changing these parameters are self-explanatory. For computation, one configurable parameter can be the number of operations performed on each array element. If this number is one (operation per element), then the task of computation is generally memory bound. This is because more time is spent on fetching each element than the time it takes to perform one operation on it. As the number of operations grows, the task tends to become more and more compute bound, since more time is spent on operations than on fetching. A program with these configurable parameters could act as a data generator that provides data for analysis and modeling.

2.2 Data Generator Application

Listing 2.2 shows the pseudocode for the data generator application, based on the Jacobi kernel. In the communication phase, each process sends messages to its neighbours. If the chosen number of messages is greater than the number of neighbours, then it re-iterates the *Neighbours* array, until all messages have been sent. For computation, an additional nested loop has been added, that repeats for a chosen number of extra operations on each element, as described previously.

```
for time:
    iNeighbourIndex = 0
    while MessagesSent < NumberOfMessages:
        if iNeighbourIndex > NumberOfNeighbours-1:
           iNeighbourIndex = 0
        iNeighbour = Neighbours[iNeighbourIndex]
        MPI_Irecv(MessageSize, iNeighbour)
        MPI_Isend(MessageSize, iNeighbour)
        MessagesSent++
        iNeighbourIndex++

    MPI_Waitall(MessagesToSend,MessagesToRecv)

    for i in rows:
        for j in columns:
            for NumberOfExtraOperations:
                compute(i, j, u_previous, u_current)
```

Listing 2.2 Pseudocode for the configurable data generator

For the experiments that were conducted, a 2D Cartesian MPI communicator was used. In order for the communication phase to be as homogenous as possible, processes on the borders of the communicator, replace their missing neighbours with processes on the opposite border. This communication pattern is shown in Figure 2.1 for 16 processes. Note that as shown in Listing 2.2 when the number of messages exceeds 4, the communication pattern repeats from Message 1.

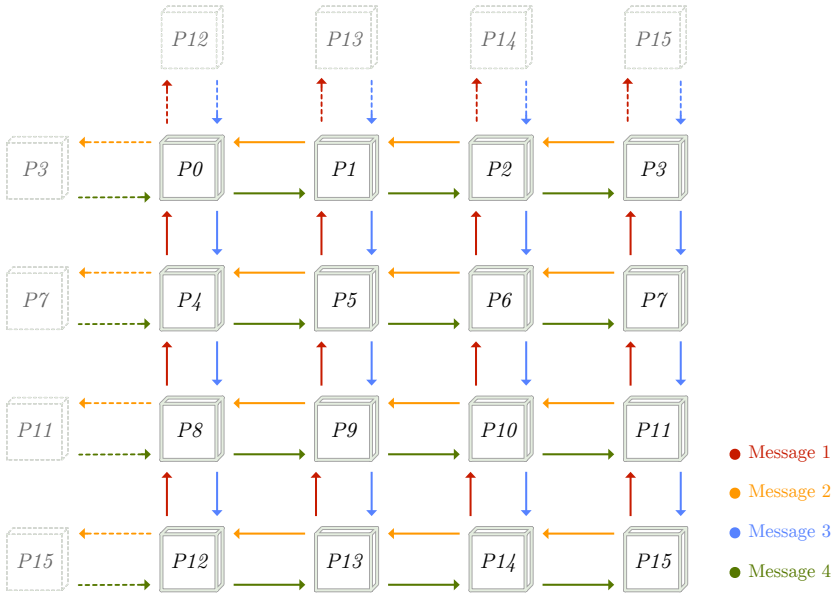

Figure 2.1 Data generator communication pattern

For the sake of simplicity, the data generator application can be summarized into the version presented in Listing 2.3. This representation summarizes the configurable parameters for the phases of computation and communication. It also makes it clear, that *code-wise*, there is no overlap between communication and computation. In an ideal scenario of perfectly balanced resources, an unforced synchronization would occur between processes, and this lack of overlap would also translate into the execution of the parallel program. Some experimentation showed that this is not the case. This subject will be expanded upon later.

```
for time:
    communication(NumberOfMessages, MessageSize)

    MPI_Waitall(MessagesToSend,MessagesToRecv)

    computation(WorkingSetSize, NumberOfExtraOperations)
```

Listing 2.3 Simplified pseudocode for the data generator

2.3 Choices for Data Sizes

Following a distinction that has already been made, there are two choices for data sizes to be made; one for communication and one for computation. For computation, choosing to follow **Weak Scaling** is the logical choice for executing in a cluster system, to leverage the full potential of the available resources. This means that the working set size of each process can remain constant while more processes are added, so that the total problem size grows with the available resources. For communication, the size of the messages was chosen to be a function of the working set size, namely some multiple of its square root. This was done to imitate a lot of problems which use stencil computations, where the data exchanged between processes is a row, a column or in the case of three dimensions a surface of a data grid.

2.4 Other Parameters and Feature Space

Apart from the parameters of communication and computation, another category of parameters has to do with the execution environment. Since the execution environment is a cluster with multicore nodes, the parameters we focus on are the number of computing nodes and the number of processes per node. In summary, the features that can be configured in our setup are the following:

- Working Set Size (per process)
- Number of Extra Computing Operations
- Message Size (depends on the working set size in the context of our experiments)
- Number of Messages
- Number of Computing Nodes
- Processes per Node

3. Execution Environment and Measurements

3.1 Execution Environment

As mentioned before, the experimental part of this thesis was conducted on up to 64 nodes of ARIS Thin Nodes island. A node of this system consists of two 10-core processors and hyper-threading was not used. This way, each node can facilitate up to twenty processes. Each processor has a 25 MB L3 Cache shared between ten cores, and each node has 64 GB of RAM shared among two processors. MPI's map-by node option was used to organize the processes onto the available cores. With this mapping, MPI ranks are shuffled alternately between processors and nodes. Figure 3.1 summarizes the above using two nodes as an example, where the squares represent a processor core and the numbers the MPI rank of the corresponding process.

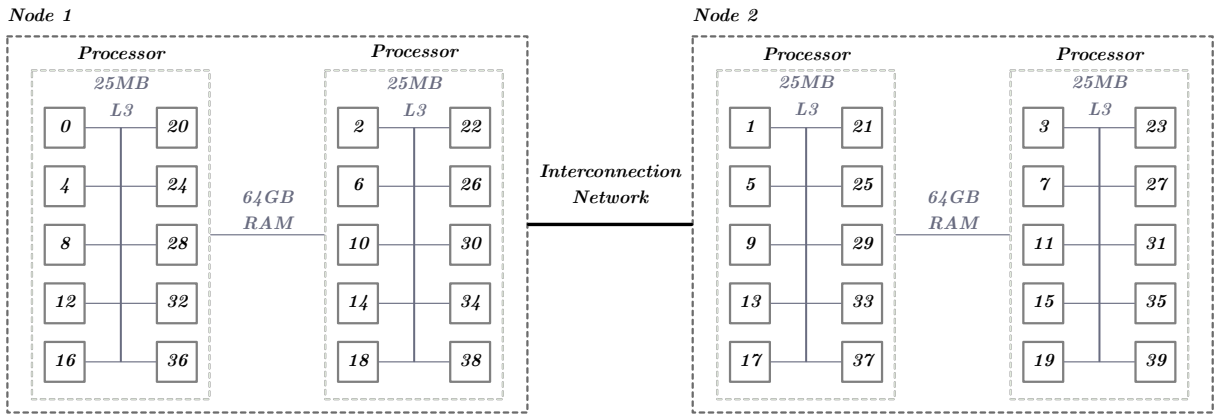

Figure 3.1 ARIS nodes populated using MPI's map-by node option

To follow Weak Scaling in a balanced manner, a base problem size was set for the minimum number of processes that the experiments ran. This base problem size was chosen, so that the base working set size per process is 1MiB. From there, the problem size was multiplied by the same factor that the total processes were. Because a 2D Cartesian Communicator was used, there had to be a match between the dimension of the communicator that was increased, and the dimension of the data grid. For experiments with different working set sizes per process, the base of 1MiB was multiplied appropriately on both dimensions of the data grid. Table 3.1 shows an

example of how data sizes are matched with different numbers of processes. In this example, four nodes are used, and the problem size is the number of doubles (eight bytes each) per dimension.

Table 3.1 Example for the matching of data size and number of processes

Number Of Processes	8	16	32	64	80
X Rank Dimension	4	4	8	8	8
Y Rank Dimension	2	4	4	8	10
Problem Size X	2048	2048	4096	4096	4096
Problem Size Y	1024	2048	2048	4096	5120

For the numbers of processes that were less than the maximum possible for each number of nodes, it was made sure that the experiments were isolated and no other processes were using the extra cores. Finally, it should be noted that openmpi version 4.0.5 and gnu version 8 were used for all the experiments.

3.2 Measurement Method

The cost metric that was chosen is time. Specifically, there were three timers that were used. One for the total running time of the kernel of the data generator application, one for computation time and one for communication time. Listing 3.1 shows the pseudocode including timers.

```
gettimeofday(totalTimeStart)
for time:
    gettimeofday(communicationTimeStart)
    communication(NumberOfMessages, MessageSize)
    MPI_Waitall(MessagesToSend,MessagesToRecv)
    gettimeofday(communicationTimeStop)
    communicationTime += communicationTimeStop - communicationTimeStart

    gettimeofday(totalTimeStart)
    computation(WorkingSetSize, NumberOfExtraOperations)
    gettimeofday(totalTimeStart)
    computationTime += computationTimeStop - computationTimeStart

gettimeofday(totalTimeStop)
totalTime = totalTimeStop - totalTimeStart
```

Listing 3.1 Simplified pseudocode for the data generator with timers

In all the experiments, the number of time iterations is 32. All processes keep their own timers and write their result in a shared file after completing all time loops. As

a way to ensure that the above timers give an accurate result, Figure 3.2 shows the measured communication time versus a derived communication time which originated from subtracting the computation time from the total time. The points represent different runs for different values of the available parameters. In this context, the values are not relevant and will be explored in another analysis, further bellow. Each point is an average of all the times reported by all ranks for a certain experiment. It is apparent, that the derived and the measured communication times are almost completely identical.

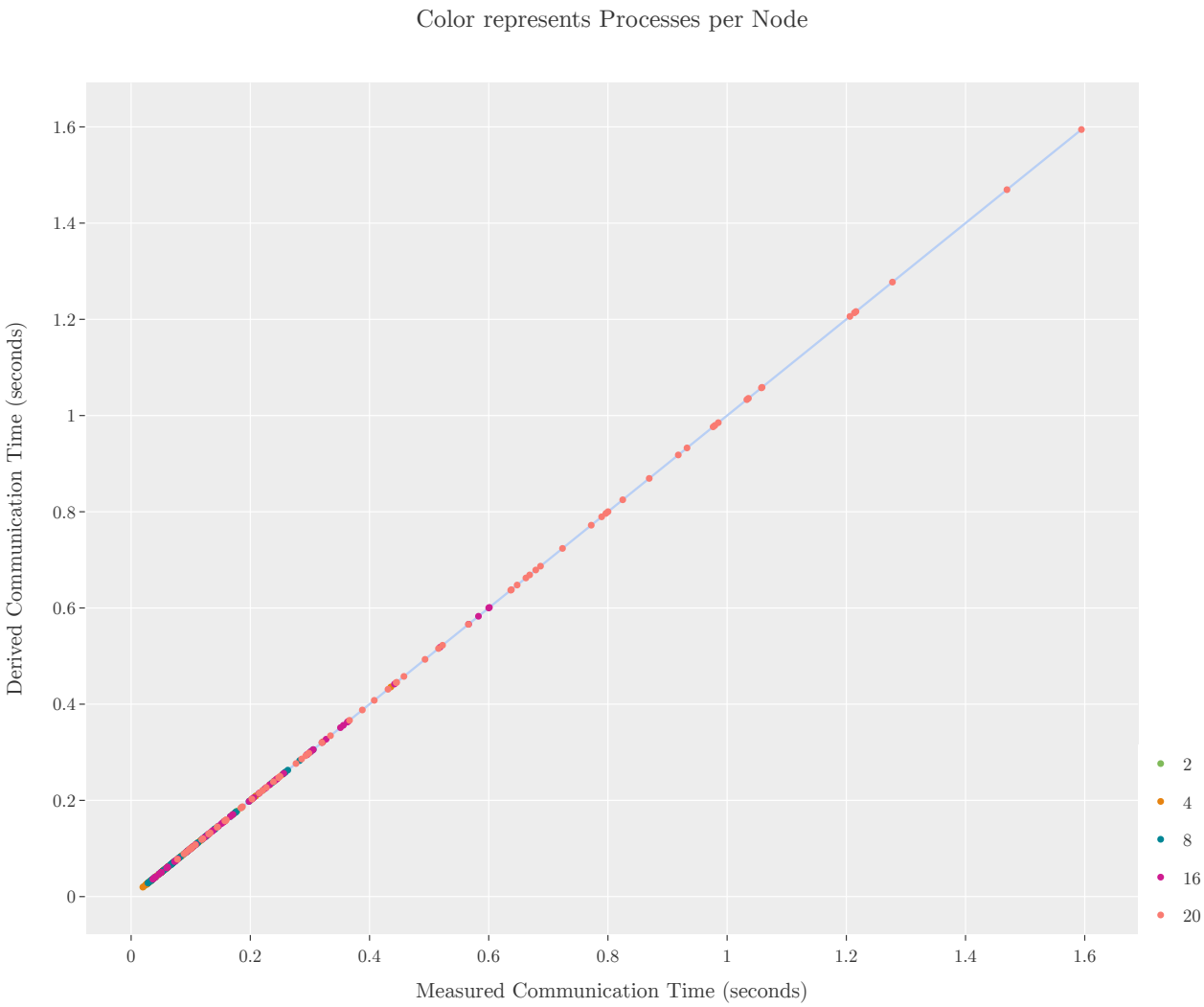

Figure 3.2 Derived vs. Measured communication time

The above phrase *"an average of all the times reported by all ranks"* is not a light statement. In fact, averaging a cost metric without mentioning variance or a confidence interval is a common fallacy [Hoefler, Belli, 2015]. For this reason, it was deemed important to include a deeper analysis of some execution data, before moving to data collection for modeling. This analysis is included in the next chapter.

3.3 Different Parts of Communication Time

Finally, there is an important clarification that should be made about the parts of the communication phase in the data generator application. Specifically, as Listing 3.1 shows, the timers for the communication phase, include the call to *MPI_ Waitall*. This inclusion is necessary, since non-blocking communication is used. Communication does not only consist of data travelling through the network or the memory, but also of the procedure processes have to follow in order to send and receive messages. The former part is what the call to *MPI_ Waitall* is: a process waiting for its communication requests to be fulfilled. This concept will also be important in the next chapter.

4. Preliminary Experiments

4.1 Statistical Analysis

Figure 4.1 shows Box Plots for experiments with different values of the working set size per process, for 64 and 4 nodes. The values for the other parameters are 20 processors per node (fully populated), 1 computing operation (so that the computation phase is memory bound), 8 messages per time iteration and a message size equal to $\sqrt{\text{Working Set Size}}$.

On the x-axis are the different values of the working set size. Each of the points represents a communication time reported from a different MPI process. This way, for 4 nodes each box plot represents 80 points and for 64 nodes, 1280. The dashed rhombus inside each box plot expresses the standard deviation, with the dashed line on its middle being the mean value of each population.

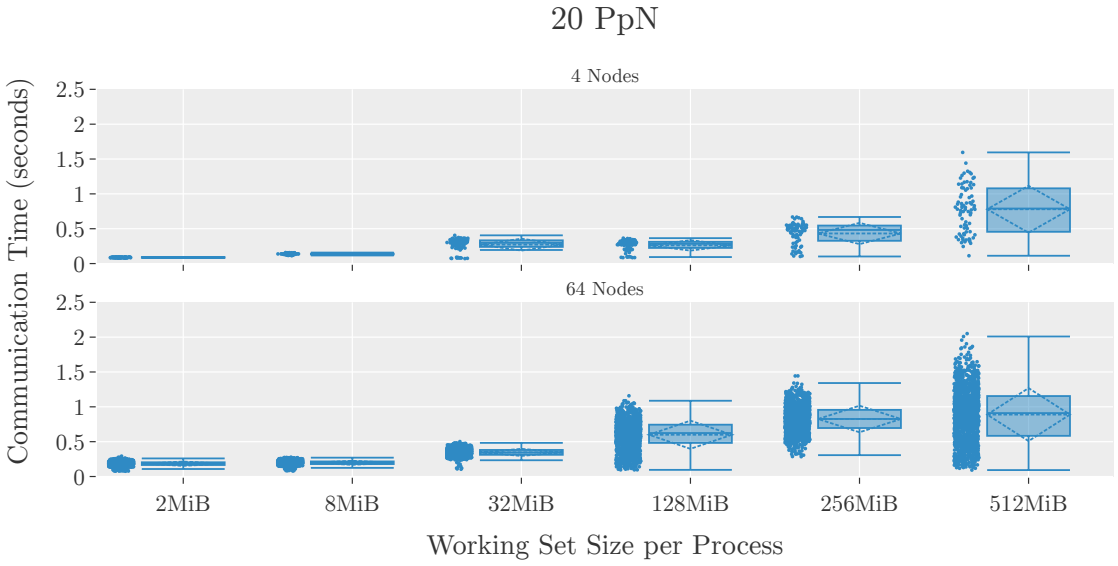

Figure 4.1 Communication time box plots for various working set sizes

One observation from the above plots is that communication time seems to generally increase with the working set size. This change will be seen across all the examples

in this chapter. The obvious explanation for this behavior lies in the fact that the message size grows with the working set size. However, to check whether there are other factors at play, some experiments with constant message sizes were made and are included in Section 4.2, "Communicational Parameters".

On another note, the reported communication times for both numbers of nodes are in the same order of magnitude, despite the total processes being 80 and 1280. This is a first sign to an approach of using a small partition of a cluster to model behaviour on a larger scale and is explored in the next chapters that focus on modeling.

Another observation for both 4 and 64 Nodes is that the spread of the reported times seems to grow with the Working Set Size (and consequently the Message Size). To examine the distribution of the reported communication times for 64 Nodes, Figure 4.2 shows the corresponding probability density histograms for each working set size, by seeing the reported time by each process, as a random variable. Additionally, each of the continuous lines represents a normal distribution $N(\mu, \sigma^2)$ with a μ and σ equal to the matching values from the data of each working set size.

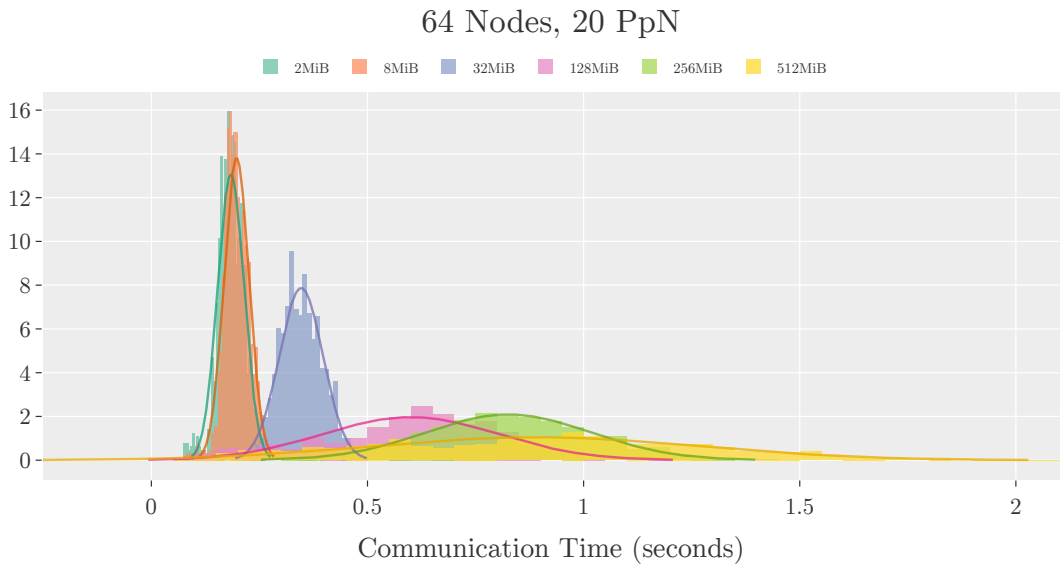

Figure 4.2 Communication time probability density histogram for 64 nodes

It is apparent that the times reported by all the ranks, mimic normal distributions with an increasing mean and standard deviation. This behaviour was observed for several different configurations of the parameters. During our observations, it was deducted that configurations with a higher number of total processes, and therefore a greater number of samples, imitate the normal distribution much more closely than the ones with a lower number of samples.

Generally, this likeness between the distribution of the communications reported by all processes in an experiment, and the normal distribution, highlights some important aspects regarding the experimental data. Firstly, it shows that the data from all processes from a single execution have a central tendency around their mean value.

This is positive and makes the mean a relatively good representative value for each experiment, especially if it is accompanied by the standard deviation. Another positive observation is that the normal distribution is a symmetric distribution, this may indicate that the experimental setup is also symmetrical and unbiased, making it better for a machine learning dataset.

To put this behaviour in the context of our specific experiments, it indicates a communication cost imbalance which is intensified by greater data sizes, that can be attributed to a combination of several different factors. An examination of some of them follows in the next sections. First we look into the parameters that affect communication directly, and then a scenario of communication-computation interference is explored.

An important concept to take into consideration for the next sections, is the combination of rank mapping (Figure 3.1) and the communication pattern (Figure 2.1). During the communication phase of each time iteration, a process exchanges messages with neighbours which are both on the same node and on other nodes. This means that a part of the communication can happen on the interconnection network, while another on the shared memory in a node. The ratio between these different types of communication can vary from process to process because of the rank mapping. This heterogeneity, plays a crucial part in the communication performance of some of the following cases.

4.2 Communicational Parameters

In this section, the behaviour of the reported communication times is examined for direct changes in the communication parameters. These parameters are the message size and the number of messages. Figure 4.3 shows plots for several configurations in 64 fully populated nodes. Each point is a mean value of reported time by all processes taking part in each experiment, with the continuous colored overlay bands showing the standard deviation. The values of the different communication parameters are the following:

- number of messages = [2, 4, 8]
- message size = [1, 5, 10, 50, 100] * $\sqrt{\text{Working Set Size}}$

After an examination of these graphs, it is obvious that there is a distinct difference between larger and smaller message sizes. Specifically, the intuitively expected behaviour of more messages that have a greater size, having a great impact on communication time, does not seem to occur in a regular manner for the smaller sizes. Another great difference between the two message size scales is the standard deviation. Larger sizes have a smaller (relative) standard deviation than the smaller sizes. These differences can be attributed to both memory and network usage.

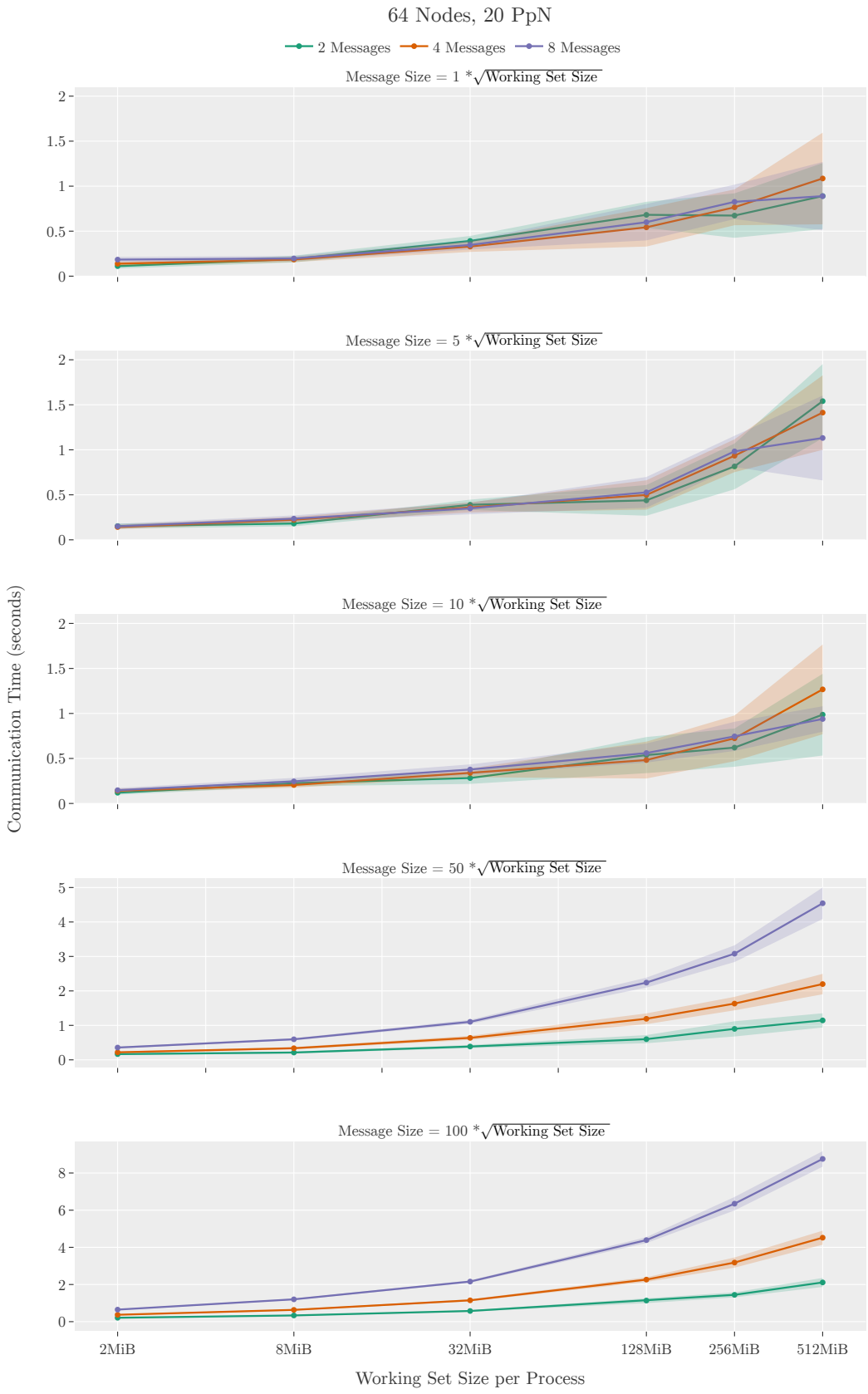

Figure 4.3 Communication Time Plots for various Message Sizes and Number of Messages (64 Nodes)

When it comes to memory usage, smaller messages may fit in the different cache memories and benefit from the high speed that they offer. However, even in the case where there are no beneficial cache effects, the smaller sizes do not stress the memory to its limits, resulting in better performance. As long as the message sizes remain small, the benefits from these effects remain relatively unchanged for different sizes. This would explain why there are no large differences in communication performance for the different smaller message sizes. Finally, the smaller standard deviation observed for the large message sizes, could be caused by the fact that for greater sizes, memory effects are less random. Because of the general disparity in communication performance, some examples from both message scales will be examined separately.

Large Messages

The first example presented in Figure 4.4, investigates the effect that varying the number of processes in a node has on performance. In these plots, the number of the total processes remains constant and equal to 128, while the number of processes per node increases. The message size is equal to 50 * $\sqrt{\text{Working Set Size}}$. This setup is useful because the problem size remains constant throughout all the variations. Generally, the performance seems to worsen in more tightly populated nodes. The effect is more prominent for 8 messages.

There are two things of interest happening as the number of nodes is higher and the number of processes per node is lower. Firstly, the available memory on each node is shared among fewer processes, and secondly, there is more communication facilitated on the interconnection network instead of within the node using the shared memory. Both of these conditions contribute to better performance.

Having more communication on the interconnection network (*2 PpN* in the plots) results in a similarity between the different numbers of messages. This probably has to do with the fact that the bandwidth of the network can easily handle the communicational load, and the relatively small changes in the number of messages, are not enough to stress it to its limits, or make a great difference in performance. As the number of processes per node is increased and communication uses more and more memory, the difference in the number of messages is more acute.

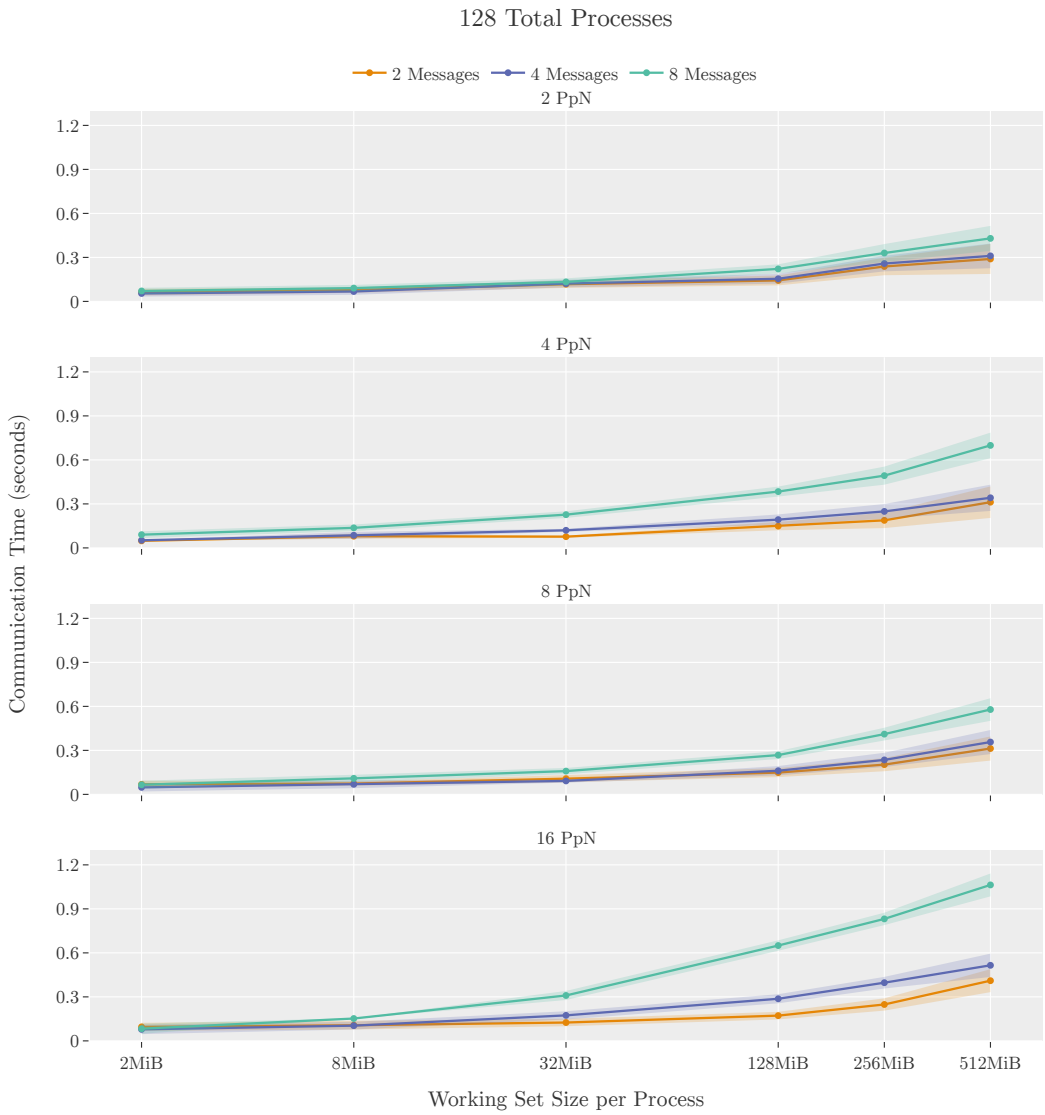

Figure 4.4 Communication Time Plots for a fixed Number of Processes (128 Processes, Large Messages)

Generally, even though the observations from the previous example stand, the changes in performance are not of great magnitude, in contrast to the changes that can be observed when changing other parameters (like the message size in the last two subplots of Figure 4.3). A stark change in performance, for a fixed message size can be seen in Figure 4.5, where the varying parameter is the number of processes per node (for a fixed number of nodes equal to 64). In this case, the number of total processes and the problem size, changes linearly. For large working set sizes (and thus message sizes, since they are directly connected), this linearity seems to translate to the communication time, as with each change in the PpN parameter; a similar change seems to occur in the mean communication time. This behaviour is beneficial, especially for predictability and modeling, since large changes in performance originate from an interpretable change in the parameters.

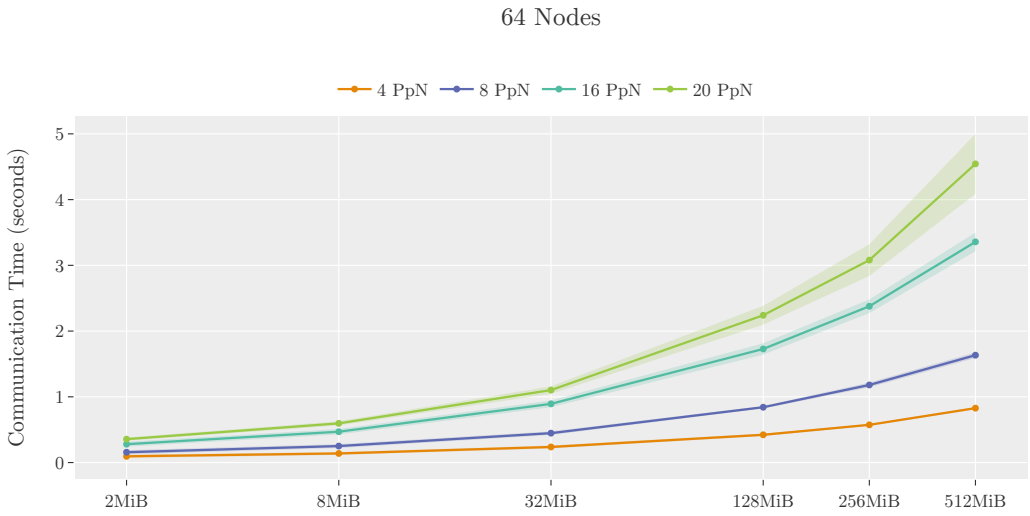

Figure 4.5 Communication Time Plots for various Processes per Node (64 Nodes, Large Messages)

Small Messages

In the case of small messages, the number of messages does not play a significant role. For this reason the following examples, retain a constant number of messages equal to 8, and a message size is equal to 5 * $\sqrt{\text{Working Set Size}}$. This was done to focus on other effects that may play a more important role for small messages.

As mentioned previously, the performance for smaller message sizes, relies on the memory usage. This way, memory contention between processes during communication may have a significant impact. To explore this scenario, Figure 4.6 shows box plots for 4 different setups with 128 total processes, similarly to the first example for the larger messages sizes.

A general trend that can be seen in these plots is that with more sparsely populated nodes, communication time tends to decrease, especially for larger data sizes. This is noteworthy, since the communication load (number of messages and message size) and the number of total processes remain the same.

The effects of changing the node density that were mentioned for larger messages, still stand. With sparser nodes, more communication is happening on the interconnection network, and each process has more memory at its disposal. On the one hand, just having more memory per process is enough to lower the communication time. This is because communication (whether its happening locally or using the network) is a memory intensive task that uses memory for send/receive buffers among other things. On the other hand, more communication facilitated on the interconnection network instead of within the node, may be a reason for the lower communication time, even for the larger data sizes in more sparse configurations.

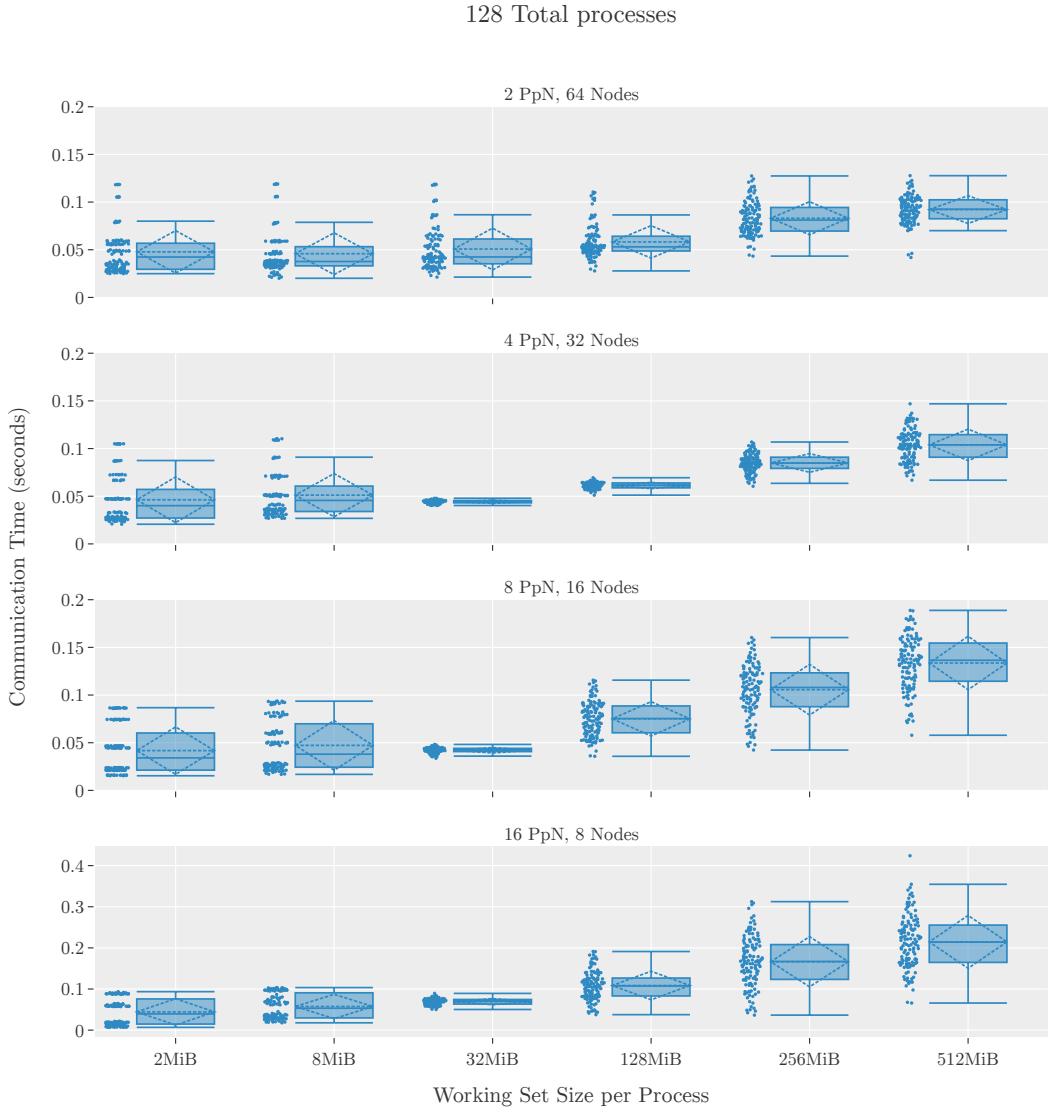

Figure 4.6 Communication Time Box Plots for a fixed Number of Processes (128 Processes, Small Messages)

One interesting change that can be observed in the above cases is the disparity within the reported communication times on each separate working set size. Specifically, in some plots, groups of reported times can be seen for relatively low data sizes. A plausible scenario for these groups may be that each one of them represents processes that have the same ratio of communication happening on shared memory to communication happening on the network, or generally have a similar communication cost. As the working set and message sizes grow, these groups become less discrete because the data sizes become relatively larger and memory contention becomes more intense, adding randomness and making the reported times sparser. To get a better image of the distributions of the reported times, Figure 4.6 shows the probabilty density histograms.

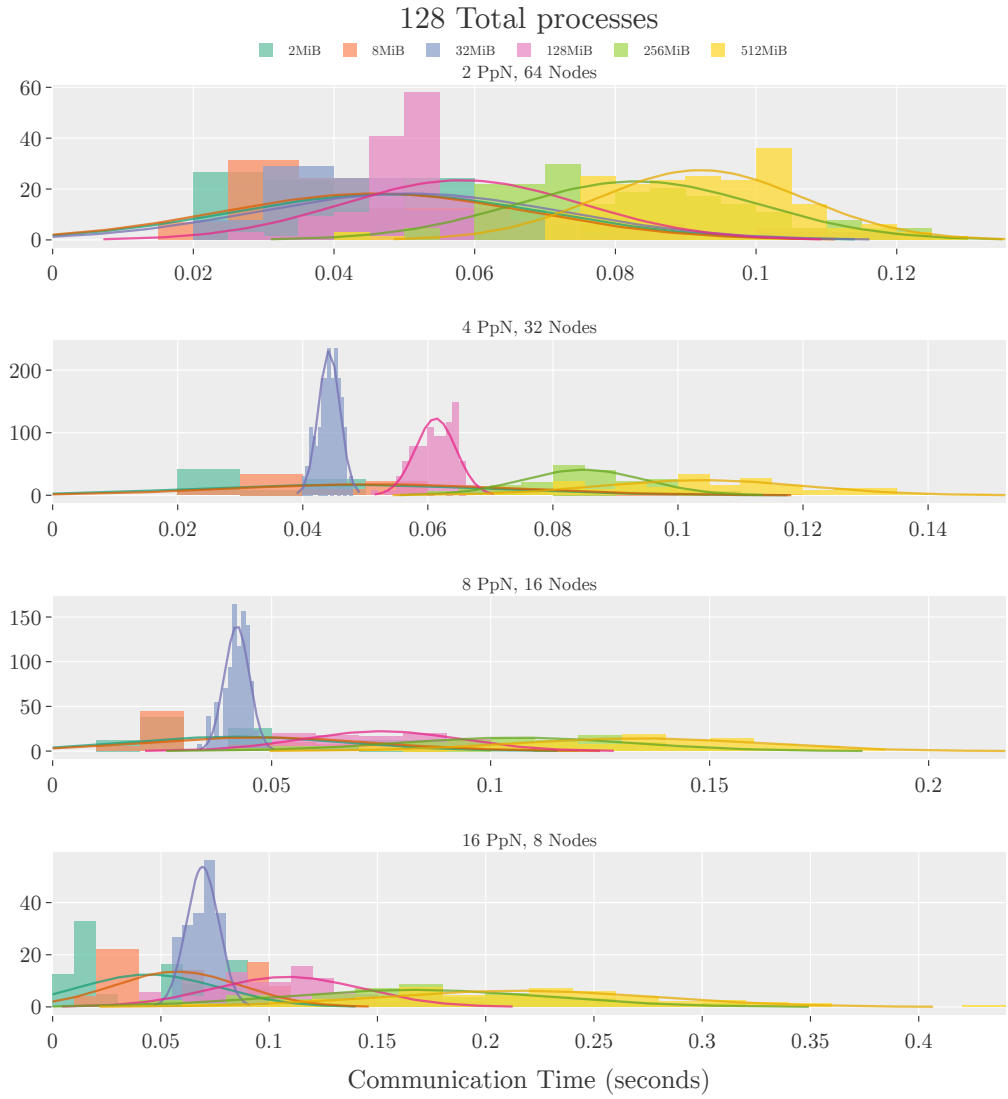

Figure 4.7 Communication Time Probability Density Histograms for a fixed Number of Processes (128 Processes, Small Messages)

It is obvious that on sparser nodes, and for smaller data sizes across all the numbers of nodes, the distributions of the reported times, are closer to a uniform distribution than a normal distribution. As memory effects become greater, either because of memory being shared among more processes, or because of larger data sizes, the distributions start to have the bell shape of the normal distribution. This may be happening because, as mentioned, memory effects add a degree of randomness. One peculiar observation from the above distribution plots is the similarity for *32MiB*, for *4, 8, and 16 Processes per Node*. The low standard deviation of the reported times may be caused by some beneficial memory effect (e.g. the message size is compatible with the sizes of the per-core caches). The fact that something similar is not observed for 2 PpN, where communication is largely happening on the network, also supports this scenario of a beneficial memory effect.

Lastly, in similar fashion to the larger messages, Figure 4.8. shows the communication times for a fixed number of nodes and various numbers of processes per node.

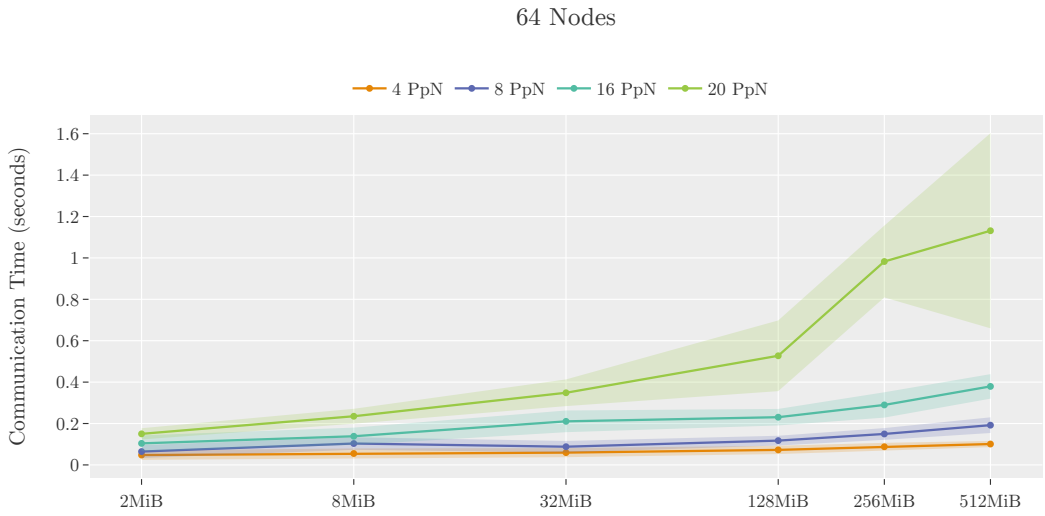

Figure 4.8 Communication Time Plots for various Processes per Node (64 Nodes, Small Messages)

The behaviour of communication time loosely following the linearity of the change in the PpN parameter, is present, as it was for larger messages. However, in this case, there is a substantial increase in the standard deviation, especially when the nodes become fully populated. This may be caused by various memory effects, as discussed in the previous example.

Constant Message Size

Finally, to examine possible factors that cause the increase of communication time with the working set size for both small and large messages, a set of experiments with constant message size was conducted. The plots in Figure 4.9 show communication times for constant message sizes for both size scales. It is apparent that for larger messages, communication time remains relatively constant with a low standard deviation. However, for smaller (and fewer) messages, the increase of communication time with the working set size that has been observed in the previous examples can be seen. Considering that no communicational parameters change this behaviour may indicate that in the case of smaller communicational and computational loads, the system can utilize the available resources in a way that benefits communication time. This may happen by parallelizing the different parts of the communication phase (Section 3.3, “Different Parts of Communication Time”). In any case, the fact that communication time changes without any changes in communicational parameters, motivates a deeper investigation in this matter.

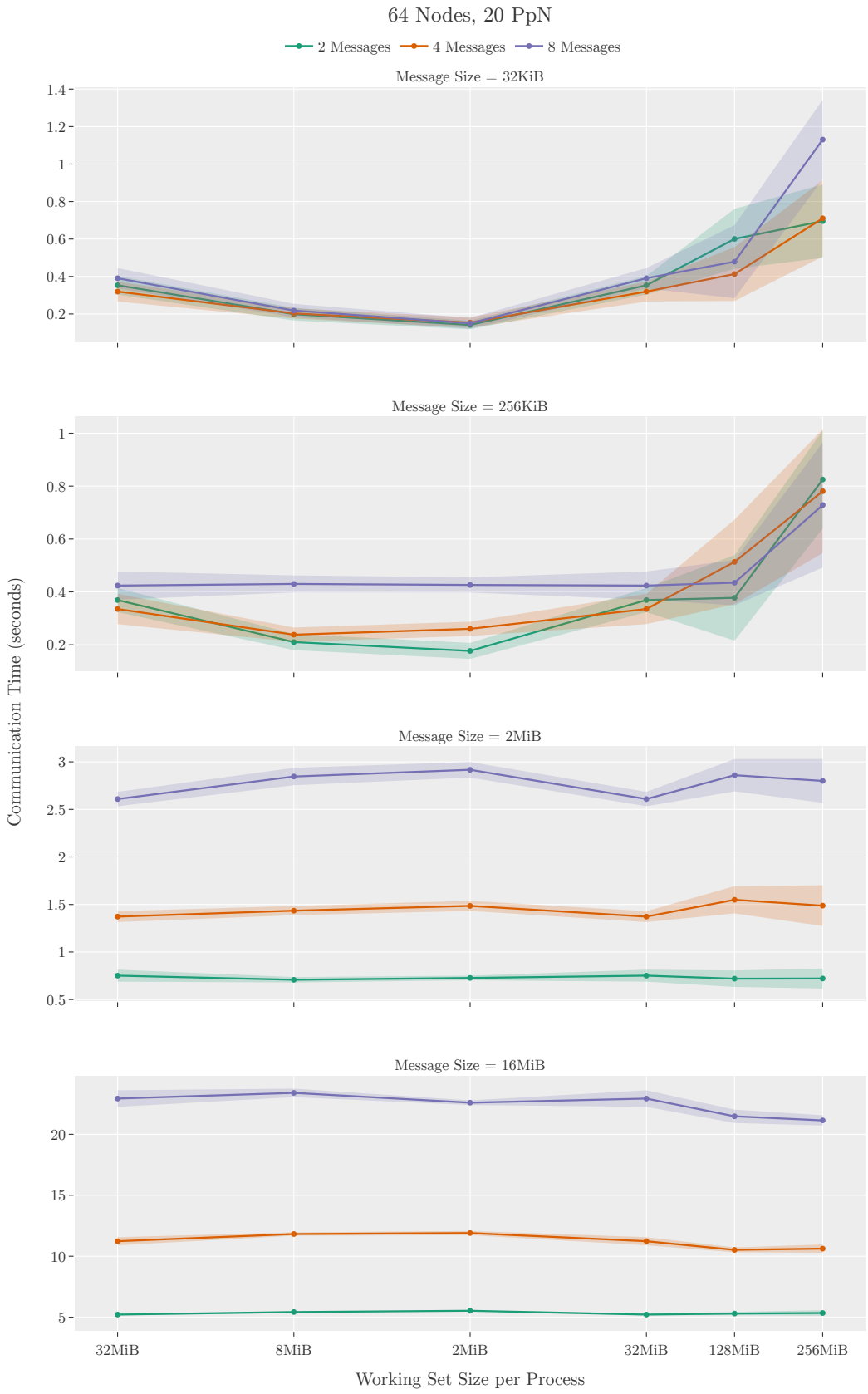

Figure 4.9 Communication Time Plots for Constant Message Sizes

4.3 Computation-Communication Interference

Up until this point the sole focus has been communication performance without taking computation into consideration. Instead, the changes in communication performance that have been observed have been mostly attributed to memory, network and mapping effects. While experimenting with an early version of the Data Generator Application, on which, the message sizes remained constant and did not change with the working set size, a change in communication time with the change of the working set size occurred, as it does with the examples that have been shown thus far. At the time, a logical possible explanation was that since nothing changed in terms of communication parameters, this increase of communication time could be due to computation-communication interference between processes on the same node.

In an attempt to determine if there is a computation-communication interference, a small adjustment to the code of the Data Generator Application was made; forced synchronization between the computation and the communication phases using a barrier was added, as shown in Listing 4.1.

```
gettimeofday(totalTimeStart)
for time:
    gettimeofday(communicationTimeStart)
    communication(NumberOfMessages, MessageSize)
    MPI_Waitall(MessagesToSend,MessagesToRecv)
    gettimeofday(communicationTimeStop)
    communicationTime += communicationTimeStop - communicationTimeStart

    MPI_Barrier(custom_communicator); // synchronize a set of processes

    gettimeofday(totalTimeStart)
    computation(WorkingSetSize, NumberOfExtraOperations)
    gettimeofday(totalTimeStart)
    computationTime += computationTimeStop - computationTimeStart

gettimeofday(totalTimeStop)
totalTime = totalTimeStop - totalTimeStart
```

Listing 4.1 Simplified pseudocode for the Data Generator with Forced Synchronization

The *custom_ communicator* on which the barrier is imposed, can be any MPI communicator, meaning any subset of processes. The ones that were chosen are a global barrier for all the processes in the system and a socket barrier for processes belonging to the same processor socket. The first one was chosen because it is a common barrier and its effect is relatively straightforward. The second one was chosen to examine the effects how processes on the same node may affect each other during the two different phases of execution (communication and computation). It was implemented using the *OMPI_ COMM_ TYPE_ SOCKET* split type with *MPI_ Comm_ split_ type*.

Configurations that include these two barriers, as well as the initial no barrier version, were executed for the three different computational load types. However, before moving on to the results from these executions, it should be noted that for these cases, derived communication time is used instead of measured communication time. This way, the time that the system spent on barriers is included. That being said, the measured communication time is also useful, in order to extract the time spent on barriers. The above can be summarized by the following expressions:

$$\textit{derived communication time} = \textit{total time} - \textit{computation time}$$

$$\textit{barrier time} = \textit{total time} - \textit{computation time} - \textit{measured communication time}$$

Since computation becomes relevant in this analysis, it should be noted that from this point forward, configurations of the Data Generator Application which have only one computation operation per time iteration, are going to be called *Memory Bound*, whereas configurations with an X number of operations per iteration are going to be called *Compute Bound X*.

Memory Bound

Figure 4.10 shows plots for the *derived* communication time as a function of working set size for the three different cases of barriers, for a Memory Bound computational phase and various processes per node. In this case, a small message size and a large message size have been included. It is apparent that for the large message size, the barriers have no significant impact. The reason for this will be examined further bellow, but first we are going to focus on the small message sizes.

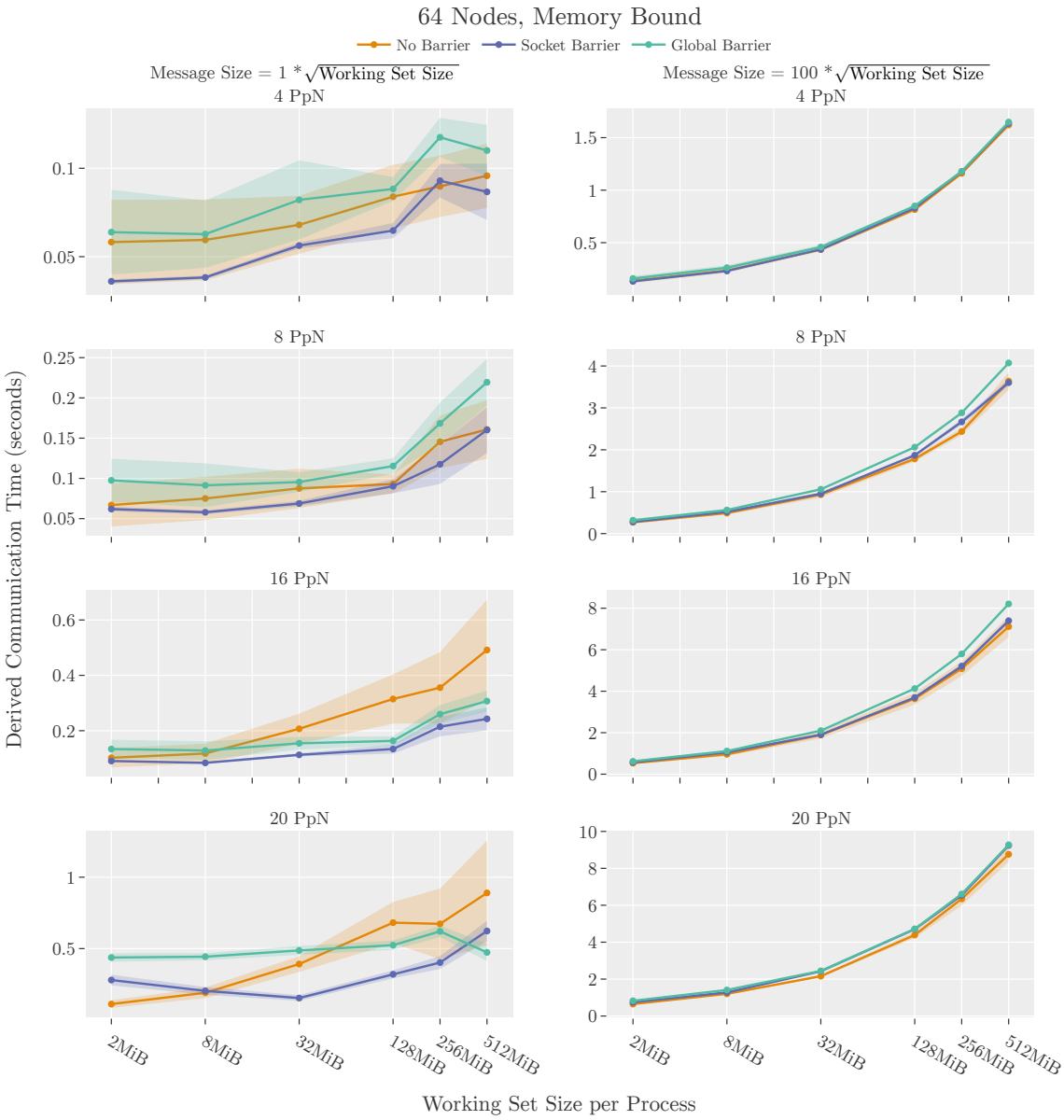

Figure 4.10 Communication Time Plots for various Barrier Types (Memory Bound)

For a small message size, the above plots show that the different versions of barriers between the phases of execution, result to different communication performance for the memory bound computational load. Specifically, imposing a socket barrier generally yields a better communication performance compared to the other two versions across all the different values of the processes per node parameter. On the other hand, imposing a global barrier seems to have a negative effect for sparsely populated nodes and lower data sizes. Another general observation is that as the number of processes per node is increased, the standard deviation of the reported communication times seems to decrease for the versions with the barriers. On the contrary, the no barrier version shows a relatively high standard deviation, an effect which becomes more prominent for higher data sizes, especially for a higher number of processes per node.

Both of the barriers add a forced synchronization between the two phases of execution. This comes with certain advantages and disadvantages. The main advantage is that it gives processes time to finish one phase before moving to the other one, while the main disadvantage is that it introduces an idle waiting time. This way, there is a trade-off between the idle waiting time and the positive impact it has on the resources that are shared among processes that reside on the same node.

A plausible scenario is that without synchronization, some processes on a node finish the communication (or computation) phase slightly earlier than others and begin computation (or communication). When the rest of the processes on the same node are finishing their own communication phase and require using the shared memory resources, there is interference between their communication and the other processes' computation. Because communication time is a smaller fraction of the total time than computation time (this will be expanded upon later), it is sensitive to this interference.

The fact that the socket barrier generally outperforms the other two versions, supports the previous scenario, as the socket barrier is essentially a focused synchronization between processes sharing resources and gets rid of the aforementioned interference. While the global barrier does offer the same synchronization, the fact that it is less focused, introduces an additional idle time across all nodes. As the previous plots show, this additional cost, often overshadows the advantages that the socket level synchronization offers.

On another note, the increased standard deviation of the no barrier version that is seen for more processes per node and greater data sizes may be caused by the fact that memory effects introduce uncertainty to the performance, as discussed from another point of view in the previous section. The decreased standard deviation of the versions with the barriers may be another indicator of how their forced synchronization reduces the communication-computation memory interference.

Finally, when it comes to the lack of impact of the barriers on the case with the large message size, it may have to do with the fact that the larger absolute communication time that is caused by the greater sizes. Namely, this absolute delta in communication time makes it more likely that processes on the same node finish the communication phase without any time skew and that's why there is no interference between communication and computation. This is a pattern that may be repeated in a similar fashion by the computation phase, as discussed in the next paragraphs.

Compute Bound 16 and 32

The following plots shows communication time for various numbers of processes per node, for the computational load of Compute Bound 16. In contrast to the Memory Bound load, in this case, imposing barriers is not particularly beneficial for any node density. The performance for the socket barrier and the no barrier version is similar in a lot of cases, while the global barrier has the worst performance across all cases.

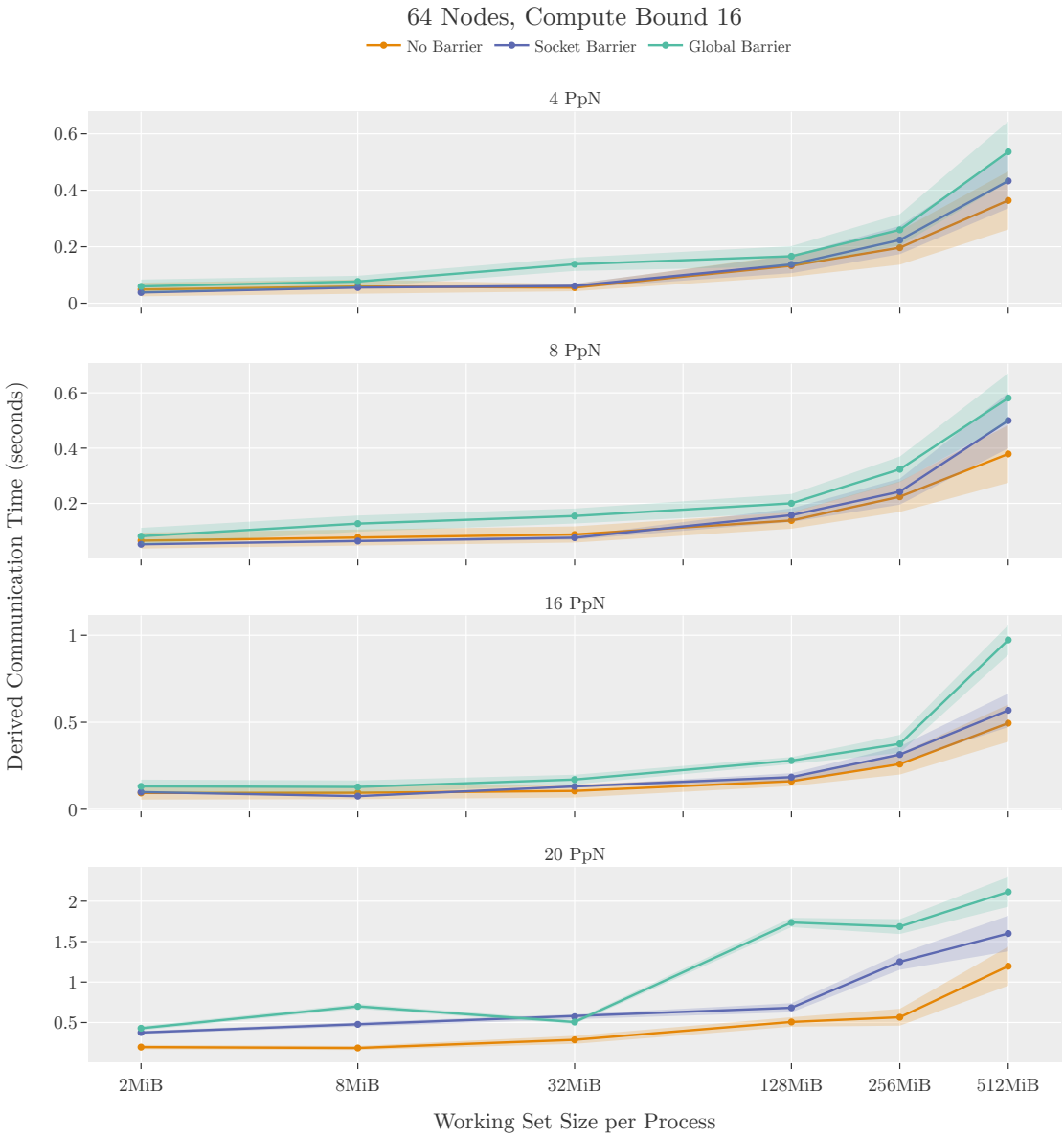

Figure 4.11 Communication Time Plots for various Barrier Types (Compute Bound 16)

The system acts in a similar manner for Compute bound 32, as shown in an example in Figure 4.12.

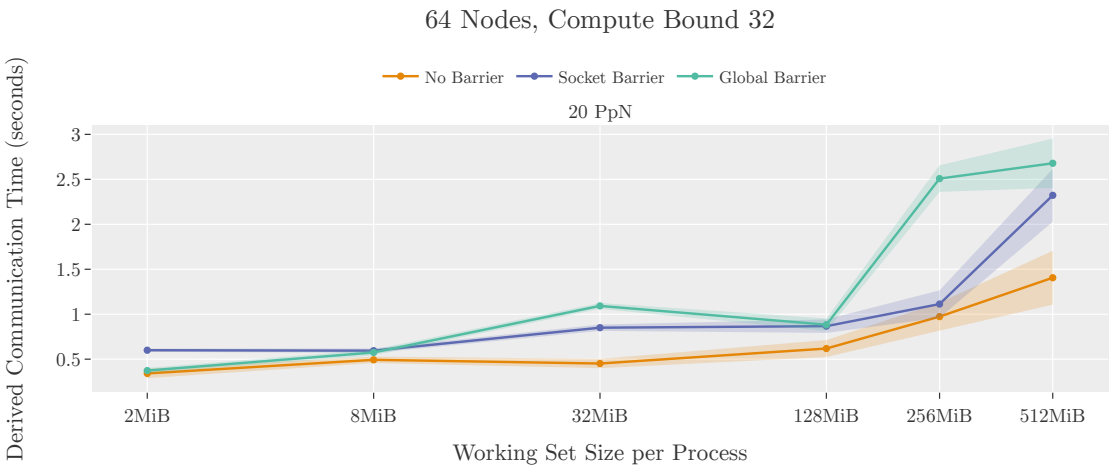

Figure 4.12 Communication Time Plots for various Barrier Types (Compute Bound 32)

This behaviour shows that in the cases of compute bound loads, there is no clear contention between processes in the two different phases of execution. A reason for this may be that because the computation phase for compute bound loads takes significantly more time than the on one in a memory bound load, processes are more likely to finish the computation phase at around the same time. This is essentially a paraphrase of the previously proposed scenario for the behaviour observed for the memory bound load; processes that are on the same node and on different phases of execution contend for the resources because they are not strictly synchronized and applying fine-grained synchronization is beneficial.

4.4 Insights Gained

In summary, through the experiments of the above sections, it was shown that there can be resource contention in the context of both communication and computation, that translates into a significant change of performance. This interference, is more prone to happen when communication has a relatively smaller size (small messages) or computation is relatively smaller and memory intensive. The other main takeaway, is that as the working set size grows, so does communication time. One factor for this is the fact that the message size is a direct function of the wokring set size, but even if the message size was constant, this change would still occur at some degree. This is because, having unsynchronized processes with two phases of execution causes a time drift that affects the waiting time for each process' communication requests. This phenomenon is something that a relatively simple model, as the one in the following chapter, may not capture.

5. Models and their Evaluation

This chapter delves into the realm of modeling, examining various modeling approaches alongside the metrics used for their evaluation.

5.1 Types of Models

Analytical Models

Analytical models are the types of models that are grounded in mathematics and theory. They have closed form solutions and rely on formal methodologies to represent the behaviour of parallel applications in an HPC system. Their strength lies in their ability to provide insights into the fundamental aspects of application behaviour and architecture design without requiring extensive empirical testing. However, as systems and applications become more complex, the task of prediction becomes more difficult for these models, prompting a shift towards more sophisticated tools. Examples of this approach were mentioned in Chapter 1, *Related Work and Goals of Present Study*. In the context of this thesis, we will not examine analytical modeling to a deeper level; nevertheless it provides a great example of the model complexity/performance tradeoff that will be mentioned later in this chapter.

Semi-empirical Models

These models are the meeting point of theory and observation. They combine data from the execution of a benchmark/data generator program with a theoretical framework. During the construction of such a model, this framework is refined using empirical data to adjust how the different parameters affect predictions. The empirical data may also be used to provide base-cases for making predictions. While the combination of theory and practical data sounds like a great middle ground, these models may suffer from the same weaknesses that analytical models do; as the applications and systems get more complicated, capturing more complex phenomena is more challenging and the number of independent variables may rise and make it more difficult to form an expression. A relatively simple example of this approach, as well as an examination of its predictive power, is presented in the next chapter.

Empirical Models - Machine Learning

ornckprc rpoekk

In the case of this approach data is prioritized over theory. These models use a dataset constructed from the execution of a data generator program, for a range of values of selected features/independent variables. Machine learning algorithms are leveraged to identify patterns, correlations, and predictive factors within the data. The fact that such algorithms are agnostic to the theoretical background of the phenomenon that is to be modelled, makes them a double-edged sword. On the one hand, they can easily adapt to accurately predict performance, even in the presence of more complex correlations between underlying phenomena (e.g. computation/ communication interference). On the other hand, the lack of a need for a theoretical framework may make these models over-trained to the data that was used during training. For this reason, the application used for data generation should be well-designed and deeply understood in order to be able to identify the pitfalls that may occur. Additionally, machine learning models may lack the interpretability of more theory-driven models, making it difficult to extract insights into the underlying mechanisms of application behaviour.

Model Complexity/Performance/Range Tradeoff

Before moving on to model evaluation metrics, an important tradeoff in modeling should be mentioned. Specifically, when designing and reviewing models, one can observe a pattern when it comes to how complex a model is, how well it performs and how many different cases it may cover. More complex models will probably perform better than simpler ones, at the possible cost of being more targeted in specific cases. This is a concept that is present in the design of any tool. For example, let's take a screwdriver. The tool designer has to choose between what screw head they want to cover. By choosing a flat-head screwdriver, we design a tool that can be used in a different range of screws but not always efficiently. A philips or even a torx screw can be undone by a flat driver with some extra effort, but the result is a damaged tool and screw which is undesired. An argument can be made that it is better to design specialized and more complex tools that have a well-defined purpose and range of applications, even if the latter is relatively small. In the context of this thesis this means that a machine-learning-based model for a specific family of applications and computer architectures may be a better choice for an accurate model, while keeping in mind the limits of where such a model can be used.

5.2 Model Performance Metrics

In this section, some common metrics for model evaluation that will be used in the following chapters are laid out. For the mathematical expressions that follow, n is the number of samples in a set of measurements, y are the measured values (that comprise

a population), $\bar{y}$ is the mean value of a population, and $\hat{y}$ are the corresponding predicted values.

Root Mean Square Error (RMSE)

This metric is a measure of prediction error that has the same unit of the predicted variable. It is a standard way to measure the error of a model. Mathematically it is expressed as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}$$

Percentage Error and Mean Percentage Error (MPE)

The percentage error is a metric that expresses the relative difference between a prediction an actual value. It can be both negative and positive. A negative value denotes that a model over-predicts, while a positive one that it under-predicts. It is given by the following expression:

$$Percentage\ Error = 100\%\ \frac{y_i - \hat{y}_i}{y_i}$$

This metric can be used in the form of multiple values (one for each measurement) and as a single mean value, where we can get the mean percentage error:

$$MPE = \frac{100\%}{N}\sum_{i=1}^{N}\frac{y_i - \hat{y}_i}{y_i}$$

In the case of the MPE, positive and negative errors may offset each other. For this reason, the value and the sign of this cumulative metric can be considered a bias of over-prediction or under-prediction.

Lastly, one (maybe obvious but) important clarification regarding these metrics, is that a negative value hides a greater absolute difference than the corresponding opposite positive value does. We can see this in an example of $\pm 60\%$ errors. In the positive percentage, by performing the proper operations we get that $\hat{y} = 0.4\ y$, whereas for the case of the negative percentage we get that $\hat{y} = 1.6\ y$. This is something that is a weakness of the following metric.

Mean Absolute Percentage Error (MAPE)

The mean absolute percentage error is very similar to the previous MPE metric, with the difference that absolute values are used in the expression:

$$MAPE = \frac{100\%}{N}\sum_{i=1}^{N}\frac{|y_i - \hat{y}_i|}{|y_i|}$$

This metric can be used instead of the MPE when we do not care about the over/ under-prediction bias that was mentioned and just want an absolute size of the error in percentage terms.

Coefficient of Determination (R2)

The coefficient of determination is a metric commonly used in modeling and is given by the following expression:

$$R2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

The numerator of the above ratio is the sum of squares of residuals and can be perceived as a measure of the absolute error of the predictions. The denominator is the total sum of squares and is proportional to the variance of the population of the measurements. This mathematical expression, pits the 'variance' of the predicted values against the variance of the population of the actual values. In other words, this metric provides a measure of the fraction of variance that our predictions cover. In the best case, which all the predictions match all the actual values the second term is zeroed, and R2 = 1. In the naive case that all the predictions are equal to the mean value of the population of measurements, the second term is equal to one, no variance is explained by the predictions, and R2 = 0. If the predictions perform worse than the previous case of the mean in terms of unexplained variance, the coefficient of determination can be negative.

6. Semi-Empirical Model

In this chapter, we explore the approach of a semi-empirical model. As mentioned, for a for such a model, we need an analytical expression, based on theory and general observations, and some data, collected via an experimental analysis for the different parameters that may exist in the analytical expression. For the latter, data from the execution of the *Exchange* class of the Intel MPI Benchmarks was used. As for the analytical expression, it will be deducted after an examination of the benchmark data.

6.1 Exchange MPI Benchmark

"Exchange" is part of the Intel MPI Benchmarks, and is a communication pattern that is somewhat similar to the one in our custom data generator application; it can be seen in Figure 6.1.

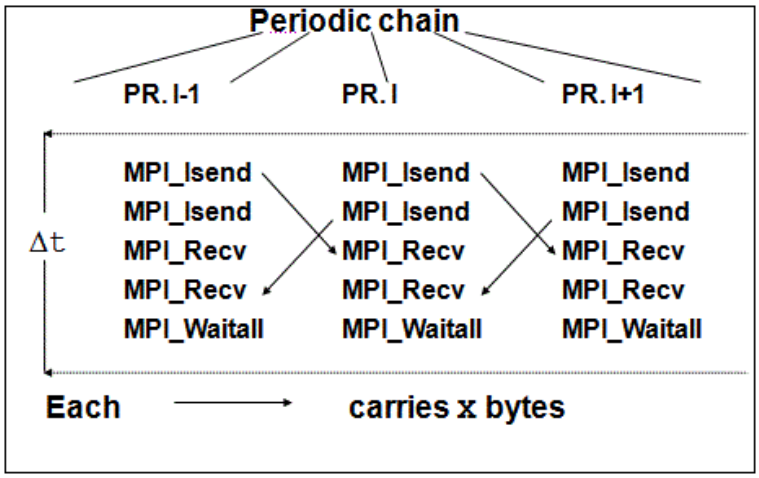

Figure 6.1 Exchange Intel MPI Benchmarks
(source: Intel MPI Benchmarks User Guide)

It is apparent that this communication pattern is similar to the communication phase of one time iteration of the data generator application. Each process exchanges two messages with two neighbours. It should also be mentioned that for the execution of this benchmark, the same map-by node option was used.

The measurements from the execution of this benchmark on 64 nodes of Aris can be seen in Figure 6.2. The measured communication time is an average of a number of repetitions that the benchmark performs, and is the Δt shown in Figure 6.1. The different lines, represent a different number of processes per node that the benchmark was run. As expected, the reported time grows with the message size and with the number of processes per node.

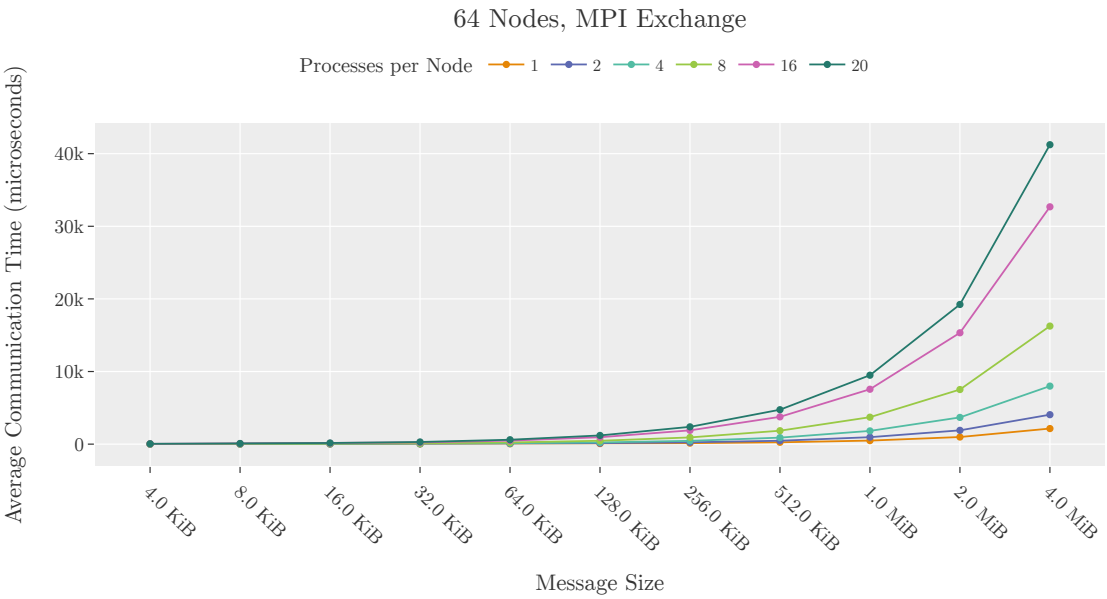

Figure 6.2 Exchange Benchmark Results on 64 Nodes

6.2 Building an Analytical Expression

To build an analytical expression for the semi-empirical model, the first step is to identify the independent variables of the model. Since the Exchange benchmark was chosen as a database of measurements, the parameters of this program have to be taken into consideration. With that being said, the features mentioned in Section 2.4, "Other Parameters and Feature Space" can be a starting point. From those, we can immediately exclude the features that have to do with computation, since there is no such phase in the benchmark. Additionally, the fact that communication is static in terms of the number of messages, eliminates that parameter. Finally, since we are examining the case of 64 nodes, and since it has been shown previously chapter (Section 4.1, "Statistical Analysis") that communication time stays on the same range of values when the only change is the number of nodes, we can also rule out the number of nodes as a parameter. All of the above, leave the following parameters as the independent variables of the semi-empirical models:

- Message Size
- Processes per node

Processes per node

Since the smallest number of processes per node is 1, an intuitive direction, is to use the values measured for the various message sizes and 1 PpN as a base case. This way, the number of processes per node, can be used as a multiplier. To check whether this relationship stands, Table 6.1 shows some examples of the average measured time for a variable number of processes per node.

Table 6.1 Exchange benchmark data for various configurations

Message Size	Processes per Node	Average Communication Time (μsec)
32 KiB	1 (64 processes)	27.49
	2 (128 processes)	43.77
	4 (256 processes)	66.73
	8 (512 processes)	116.48
	16 (1024 processes)	235.55
256 KiB	1	136.91
	2	250.50
	4	453.08
	8	923.11
	16	1897.49
2 MiB	1	980.38
	2	1900.20
	4	3679.86
	8	7520.41
	16	15333.66

The above values show that by doubling the number of processes per node, the time reported, also changes by a factor close to 2. The presence of uncertainty in the experimental measurements makes us more receptive to the differences that emerge.

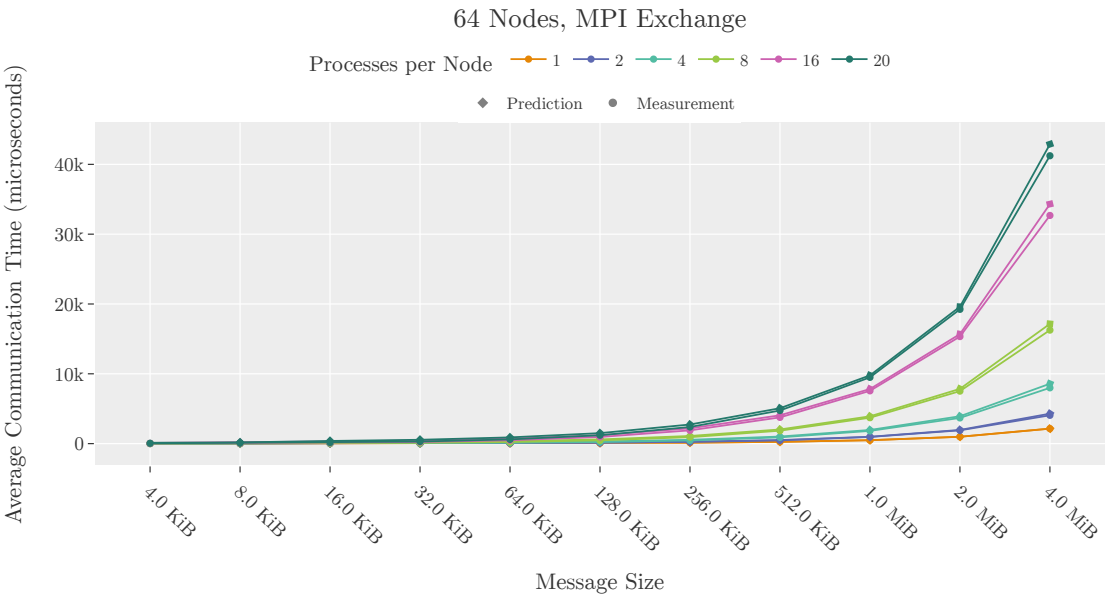

Figure 6.3 Exchange Benchmark predictions using PpN as a factor

Figure 6.3 shows predictions for the average communication time, made by using the values of one process per node and all the different message sizes as base cases, in an expression that uses PpN as a factor:

$$Communication\,Time(PpN, Message\,Size) = PpN * Base\,Case(Message\,Size).$$

The general image that is given is that the predictions are fairly close to the measured values. The 'empirical' parts of this simple model are the base-cases for the different messages sizes. In the next section, the base case will be reduced to only one measurement, by adding the Message Size as a parameter.

Message Size

To examine the effect of the message size to the communication time, isolated from the effect of the PpN parameter the following ratio can be considered:

$$Communication\ Ratio = \frac{Communication\ Time}{Base\ Time * PpN},$$

where the base time is a single value of a measured communication time for 1 PpN and a certain message size, chosen to be equal to 4 KiB. With this ratio, we can make observations on how communication time changes with changes to the message size without having the changes that PpN has (in the context of the proposed model). This is because in the previous section this parameter was chosen as a multiplier and in the above communication ratio, it was used as a divisor. Another useful ratio that will be used is the ratio of a message size to the message size of the base case (4 KiB):

$$Message\ Size\ Ratio = \frac{Message\ Size}{Base\ Case\ Message\ Size}.$$

Table 6.2 shows some examples of the above ratios that helped in making the choice for how the effect of the message size can be expressed analytically.

Table 6.2 Exchange benchmark data with aiding ratios

Processes per Node	Message Size	Message Size Ratio	Communication Ratio
1 (64 processes)	256 KiB	64.0	22.44
	512 KiB	128.0	41.53
	1 MiB	256.0	80.11
	2 MiB	512.0	160.71
	4 MiB	1024.0	351.55
2 (128 processes)	256 KiB	64.0	20.53
	512 KiB	128.0	38.29
	1 MiB	256.0	78.47
	2 MiB	512.0	155.75
	4 MiB	1024.0	332.44
4 (256 processes)	256 KiB	64.0	18.56
	512 KiB	128.0	36.71
	1 MiB	256.0	74.66
	2 MiB	512.0	150.81
	4 MiB	1024.0	327.50
8 (512 processes)	256 KiB	64.0	18.91
	512 KiB	128.0	37.90
	1 MiB	256.0	76.05
	2 MiB	512.0	154.10
	4 MiB	1024.0	333.18
16 (1024 processes)	256 KiB	64.0	19.44
	512 KiB	128.0	38.49
	1 MiB	256.0	77.46
	2 MiB	512.0	157.10
	4 MiB	1024.0	334.86

When observing the change of the 2 ratios for the different values of the number of processes per node, a pattern can be observed. Namely, the message size ratio seems to consistently be close to about three times the communication ratio (the observed values range from about 2.8 to 3.2 times for the different sizes). This leads

to the following expression, which combines the processes per node parameter and the message size:

$$Communication\,Time(PpN, Message\,Size) = PpN * \frac{1}{3}\,\frac{Message\,Size}{Base\,Message\,Size} * Base\,Case\,,$$

where the *base case* is the communication time measured for 1 PpN and 4 KiB.

Figure 6.4 shows the prediction this semi-empirical model makes versus the actual times that were measured. Before looking into any performance metric of the model, when comparing the general pictures that Figure 6.4 and Figure 6.3 show, the latter seems more accurate. This is expected on one degree, since it uses more experimental data, which capture the behaviour of the system on a deeper level when compared to the analytical expression.

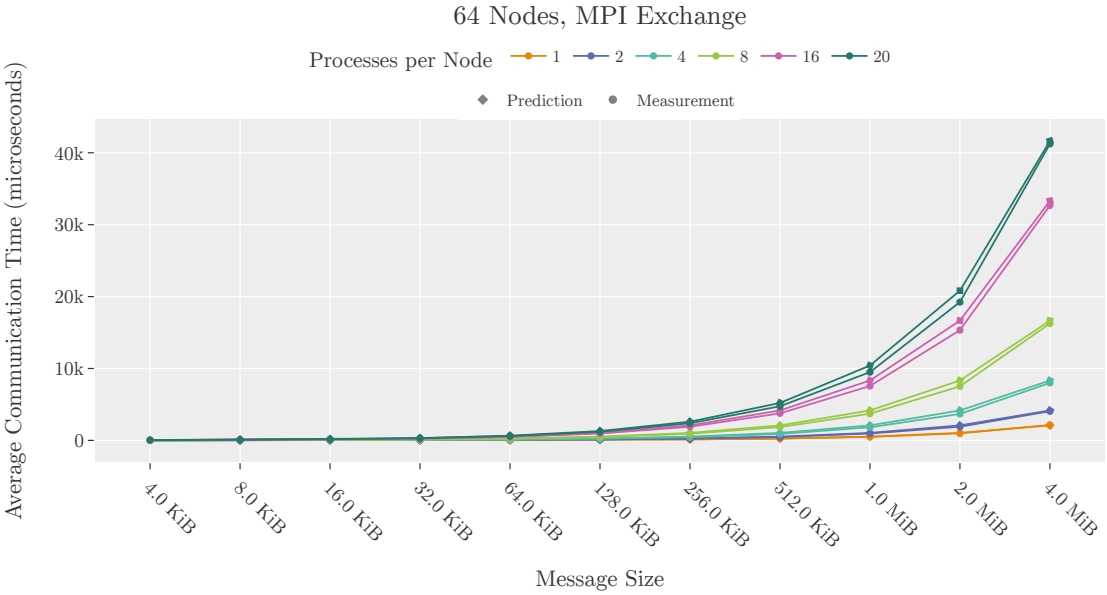

Figure 6.4 Exchange Benchmark semi-empirical model predictions

Model Performance

Moving on to examining the performance of the semi-empirical model on the data gathered from the Exchange benchmark, Figure 6.5 shows the percentage error for the predictions made for all the different values of the parameters.

There are two general behaviours observed from this plot, that may have a common cause. The first one is that as the number of processes per node grows, the percentage error is reduced. Secondly, as the message size is increased, the percentage error decreases up to a certain size after which it increases slightly. This means that for a small number of processes per node and a small message size, the model under-predicts, while for greater values of these parameters, the model slightly over-predicts. One reason for these observations may be that for the smaller values of both of the parameters, the value of time that is to be predicted is extremely low. This can be seen in the previous plots (e.g. Figure 6.3), where the time vales for larger message

sizes and higher numbers of processes per node grow significantly. This means that if the absolute difference between the model predictions and the actual values, remains in a low order of magnitude (which is what seems to be happening), the percentage error will be larger when the actual value is in the range of the absolute difference.

In practice, this can be seen in the values of the percentage error, which starts off at 50%-70% for small message sizes and few processes per node, and reaches values in the range of ±10% as the parameters grow.

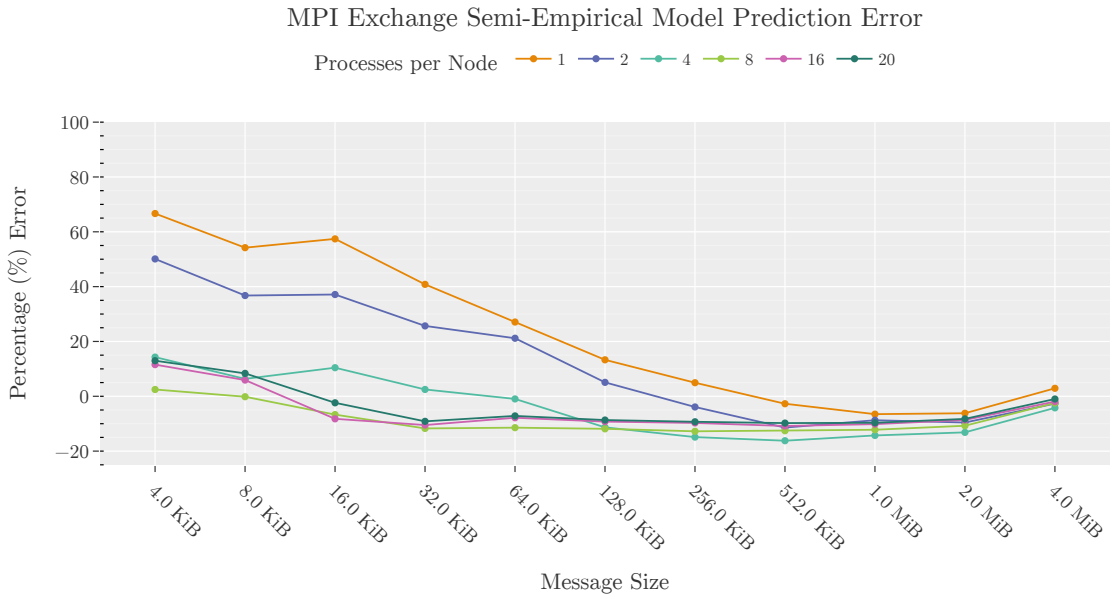

Figure 6.5 Exchange Benchmark semi-empirical model percentage error

6.3 Differences with the Data Generator Application

All in all, the proposed semi-empirical model, seems to have an acceptable predictive performance when it comes to the Exchange benchmark. However, this is not necessarily indicative of a good predictive performance in general. In this section, we see that the model performs very differently when trying to predict the communication performance of the data generator application, and we explore the reasons behind these differences.

When trying to predict the performance of the data generator application, an adaptation needs to be made to the analytical expression of the model. Namely, the whole expression has to be multiplied by the number of time iterations (equal to 32), as the exchange benchmark (which is the basis of this model), only reports one iteration of a communication phase. With this multiplication, we assume that all time iterations have a similar performance. This is not an illogical assumption to make, especially considering that the time reported by the exchange benchmark is a mean value of several repetitions. After this adaptation, a prediction was made for several different configurations of processes per node and message size. The model consistently under-predicts the communication time of the data generator application, and the

percentage error is in the range of 90%-100%. Some examples for a working set size of 32MiB, 64 nodes and three different message sizes are shown in Table 6.3:

Table 6.3 Semi-empirical model predictions of the data generator application

Message Size	Processes per Node	Communication Time (seconds)	Prediction (seconds)	Percentage Error (%)
	2	0.065	0.00013	99.8
	4	0.056	0.00026	99.5
32 KiB	8	0.085	0.00052	99.3
	16	0.16	0.00104	99.3
	20	0.35	0.0013	99.6
	2	0.074	0.00104	98.6
	4	0.077	0.00208	97.3
256 KiB	8	0.086	0.00416	95.1
	16	0.17	0.0083	95.2
	20	0.36	0.0104	97.1
	2	0.12	0.0083	93.3
	4	0.18	0.016	90.8
2 MiB	8	0.31	0.033	89.5
	16	0.6	0.066	88.9
	20	0.75	0.083	88.9

The predictions are constantly one or two orders of magnitude lower that the actual values, in the above table. The contrast between this and the relatively good performance of the model on the data from the exchange benchmark leads to the direction of contemplating the differences between the data generator application and the exchange benchmark.

The first step is to ensure that this change persists beyond the predictions of the semi-empirical model. For this reason Table 6.4 compares the communication times reported by both the exchange benchmark and the data generator application (32MiB working set size, 64 nodes). For the smaller message sizes of these examples, the same order of magnitude discrepancy can be seen, while for the larger message size, the exchange benchmark still reports a smaller communication time, but the difference is not as large. This difference between large and small messages probably has to do with details discussed previously on Section 4.2, “Communicational Parameters”.

Table 6.4 Data Generator Application vs. Exchange Benchmark

Message Size	Processes per Node	Data Gen. App. Communication Time (seconds)	32 * Exchange Benchmark Time (seconds)
32 KiB	2	0.065	0.00140
	4	0.056	0.00214
	8	0.085	0.00373
	16	0.16	0.00754
	20	0.35	0.00954
256 KiB	2	0.074	0.00802
	4	0.077	0.01450
	8	0.086	0.02954
	16	0.17	0.06072
	20	0.36	0.07620
2 MiB	2	0.12	0.0083
	4	0.18	0.06081
	8	0.31	0.24065
	16	0.6	0.49068
	20	0.75	0.61552

A brief comparison between the two applications can shed light into the discrepancies observed in the previous example. With an inspection of Figure 2.1 for two messages, and Figure 6.1, a similarity arises between the two communication patterns. Nevertheless, there is a significant difference. The exchange benchmark is only composed of a communication phase and no computation is involved. As observed, this absence of computation significantly influences the performance of the two applications, despite their seemingly similar communication patterns on paper. Chapter 4, *Preliminary Experiments* inclined towards this difference, especially on the sections that examined how the two phases of execution may interfere.

6.4 Conclusion

In this chapter, we saw a live example of the trade-off between simplicity and accuracy in a performance prediction model. The semi-empirical model that was implemented was fairly simple, both in terms of data collection and in the construction of the analytical expression. While the model has an acceptable performance on predicting cases of the benchmark used for its development, it performs poorly in predicting more realistic execution scenarios. For this reason, in the next two chapters, we see how we can deploy machine learning methods to capture more complex scenarios with more independent variables for both computation and communication.

7. Machine Learning Models

In this chapter, we approach the problem of modeling from a machine learning perspective. First, a theoretical background for the techniques that were used is provided. After that, the selected type of model is presented. The data pre-processing and some other needed details and strategies are also mentioned.

7.1 Theoretical Background

The task we want to achieve, is to create a model that predicts the communication time, given the values of some parameters (features). For this, any supervised learning regression model can be used. The way these models work in general is by iterating through a given dataset and choosing/tuning a function that best fits the data, under a loss function. A loss function is a function that quantifies an event or a change of the independent variables during the training of the model. This way, regression algorithms find the proper relationships between the chosen features (independent variables) and the label (target variable).

Different algorithms use different methods to find the aforementioned relationships. One coarse example of such a method is finding the optimized coefficients (by training with a dataset) of a closed form function of the features (e.g. linear, polynomial regression). To make predictions, these models plug in the values of the features to the fitted function, and provide a prediction. Another, somewhat different approach is making subsets of the provided data by splitting it based on the values of the features in a way that minimizes a loss function. To make a prediction, the model considers the given values of the features, matches it to the proper subset and provides the subset's mean value as a prediction. This methodology is used by Decision Trees.

For our models, we chose the second approach due to the number of features, their (sometimes) non-linear relationship with time, and how they may interact with each other. Some details about decision tree algorithms and ensemble methods follow.

Decision Trees for Regression

Decision tree learning is a supervised learning technique that can be used both in classification and regression. As mentioned, they are based on splitting the dataset in subsets. During training, the dataset splits into branches based on feature values, creating a tree-like structure of decisions. Each split is chosen to minimize the variance

within each branch, aiming to have leaves (end nodes) with homogenous and/or similar values. This process of splitting continues until a predefined stopping criterion is met, such as a minimum number of samples in a leaf or a maximum tree depth. A decision tree using features from the data generator application can be seen in Figure 7.1. Each node in this schematic example contains the condition of the next split and the value of communication time provided by its subset. The green lines represent cases where a condition of a node is met, and red lines the cases were it is not.

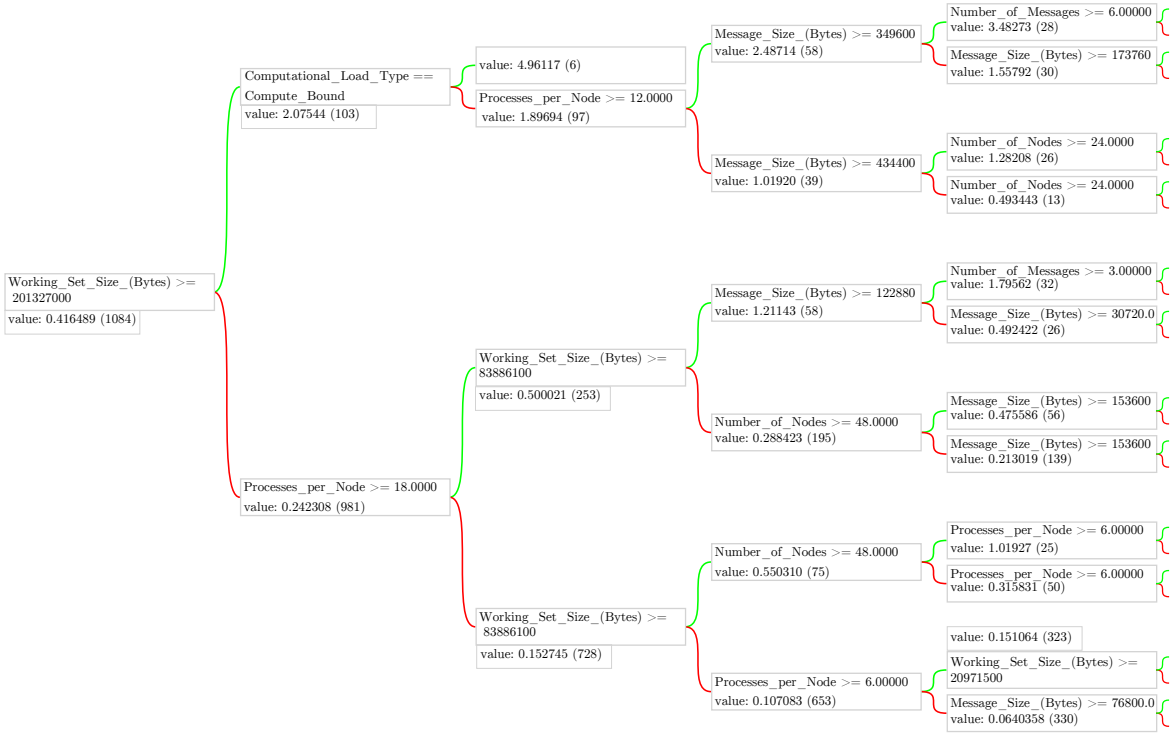

Figure 7.1 Example of a Decision Tree

Behind the above shape, hides one of the major advantages of decision trees; they are relatively easy to interpret as a series of conditions. However, these models are prone to overfitting to the training data, especially when they are left to grow in depth. This happens because at large depths, the nodes start representing smaller and smaller subsets of data. One widely used method to avoid this is to use multiple decision trees. The different methods that combine multiple trees are called Ensemble Methods.

Ensemble Methods based on Decision Trees

Ensemble learning in general is the combination of multiple predictive models into one. As noted above, the specific use of this method with decision trees is of interest for our modeling efforts.

One way to combine multiple decision trees is by making several different trees, to be trained in parallel. In this case, each tree can have a different configuration (e.g. a random subset of features, or a random subset of the training data). This way,

different trees can cover a vast range of the cases present in the training dataset. To make predictions, the average of the predictions made by all trees is taken. The model that was just described is known as a Random Forest. Figure 7.2 shows a general schematic explanation of this method.

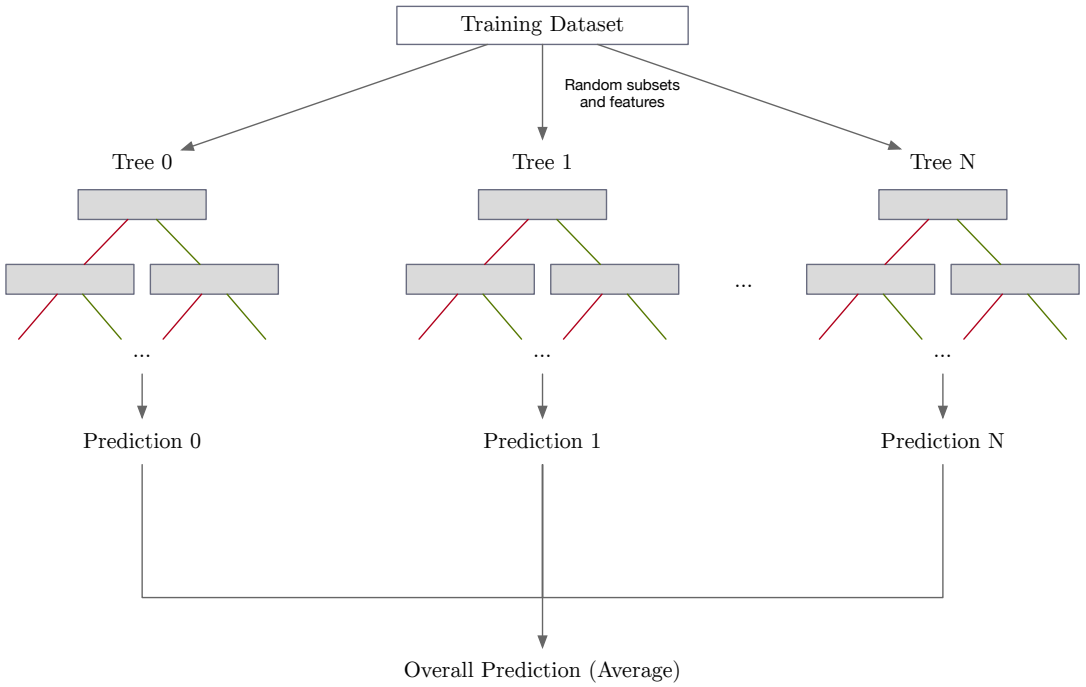

Figure 7.2 Random Forest Schematic Explanation

A different approach, is to arrange trees sequentially in a method called Boosting. Specifically, during training, the model first makes a constant prediction on the training dataset. The residuals (difference between predicted and actual value) for these predictions are calculated for every element of the dataset. These residuals are then used to train a new decision tree based on the values of the features. The newly predicted residuals from this tree are added to the corresponding predictions made in the previous step. This sum is then used in a one-dimensional optimization problem that uses a loss function and aims to find the proper weight for this set of residuals, depending on how close it moved the previous predicted values to the actual ones. The resulting weighted residuals are added to the predictions made in the previous step. Then, the new residuals are calculated and the process is repeated for a predetermined number of trees. The above is a brief explanation of a Gradient Boosting model and can also be seen inFigure 7.3. These simplified explanations for random forests and gradient boosting were made using the proposals made by [Breiman, 2001] and [Friedman, 2000] respectively.

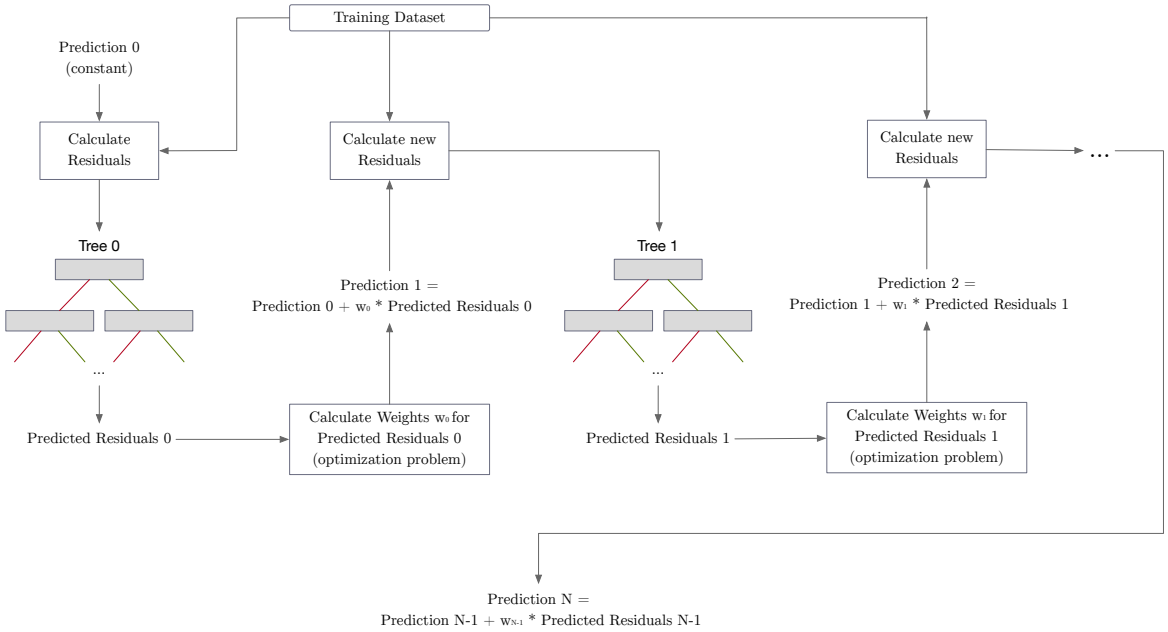

Figure 7.3 Gradient Boosting Schematic Explanation

In general, ensemble models that use boosting outperform the ones that use bagging (the general method used by the random forest). The main advantage boosting provides is a kind of "memory", meaning that in every step, the model adapts based on the predictions of the previous step. However, this does not come for free. Training times are usually longer for boosting models, than they are for the ones that use bagging.

The implementations of the two ensemble learning models that were used in this thesis are the ones provided in the TensorFlow Decision Forests (TF-DF) library which uses the YDF Decision Forests library. We experimented on our dataset using both random forests and gradient boosting, for several different configurations of each model's hyperparameters. Ultimately, the gradient boosting method constantly proved to provide superior predictive performance, leading to it being the choice for the final models. However, random forests were a great introduction to the area of ensemble methods because of their simplicity compared to gradient boosting. This is another example of the repeatedly mentioned trade-off between model simplicity and performance.

Hyperparameter Tuning

Another machine learning concept that was used is hyperparameter tuning. It is a process that aims at optimizing the parameters of a model that are not learned from the data during training. In the context of decision tree ensemble methods, some examples of these hyperparameters are the number of total trees and the depth of each tree. This process involves searching through a space of hyperparameter values to find the combination that yields the best performance, evaluated using cross-validation techniques to avoid overfitting. The final models, have their hyperparameters tuned, using a random search over a predefined hyperparameter space.

In the remaining sections of the present chapter, we look at some details of the dataset and some strategies that were followed for its best usage.

Feature Importance

Feature importances are numerical scores used in machine learning models that express how important the features were during training. There are several different techniques for all the different model types that produce different scores for each feature. For our models, we used a Decision Forest specific method. Namely, each feature's importance is calculated as the inverse of the average minimum depth of its first occurrence across all the tree paths (INV_MEAN_MIN_DEPTH in the YDF documentation).

7.2 Data Collection and Filtering

The features that were used are the ones mentioned inSection 2.4, "Other Parameters and Feature Space", with the only change being that the number of extra computing operations is replaced by the variable *Computational Load Type* which can be either memory or compute bound (with 1 or 16 computing operations respectively). The values that were swept for all features are the following:

- Working Set Size (per process) = [2MiB, 8MiB, 32MiB, 128MiB, 256MiB, 512MiB]
- Computational Load Type = ['Memory Bound', 'Compute Bound']
- Message Size = [1, 5, 10, 50, 100] * $\sqrt{\text{Working Set Size}}$
- Number of Messages = [2, 4, 8]
- Number of Computing Nodes = [4, 8, 16, 32, 64]
- Processes per Node = [2, 4, 8, 16, 20]

In order for a model of communication time to make practical sense and to clear up the dataset, it was deemed necessary to filter out corner cases that occurred during data collection. We consider corner cases, the ones where communication time is either a very high or a very low fraction of the total time. In the case of very low communication time, trying to predict a small proportion of the total time does not make any sense (e.g. if the total execution time is 10 seconds and communication time is in the range of 0.01 seconds). On the contrary, a very large communication time that is accompanied by a very small computation time is not a scenario that occurs in real stencil application problems. The criterion for the filtering of the data set is the following:

$$0.1 \leq \frac{Communication\ Time}{Total\ Time} \leq 0.8$$

After it is applied, we are left with about half of the initial dataset. Figure 7.4 shows histograms for each feature on the filtered data. It is apparent that there are no

significant anomalies or irregularities for the features, except for large working set sizes. This is because in these cases, computation time increases extremely. This peculiarity is also translated in the message size, as it is dependent on the working set size.

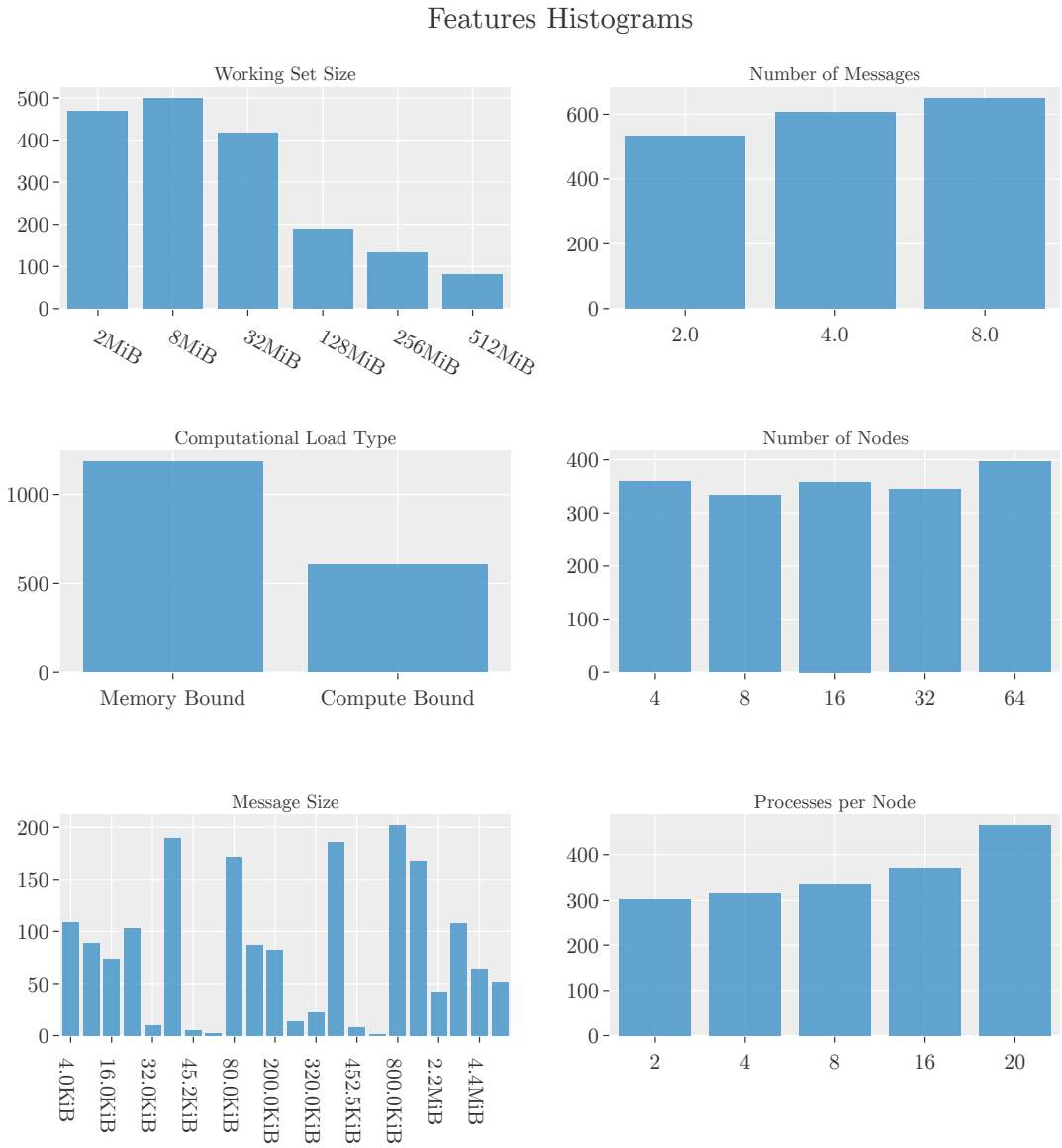

Figure 7.4 Filtered Dataset Feature Histograms

7.3 Model Scenarios

In this section we explore three different scenarios of how the collected dataset can be used. The resulting performance for each of these cases is included in the next chapter. It should be noted that a 60/40% training/testing split of the data (sub)sets was used in all the following models.

Message Scale Scenario

Inspired by the evolution of the LogP model family and by the vast difference of performance observed in Section 4.2, "Communicational Parameters" for the two message size scales, the first scenario is one where the dataset is split into two subsets. The first one includes experiments with larger messages while the other includes configurations with smaller messages. Each of these subsets is then used to train a different hyperparameter-tuned gradient boosting model. For the small scale, we consider the instances where

$$\textit{Message Size} = [1,\ 5,\ 10] * \sqrt{\text{Working Set Size}}$$

and for the large scale

$$\textit{Message Size} = [50,\ 100] * \sqrt{\text{Working Set Size}} \quad .$$

Train Small/Test Big Model

In this scenario, the model is trained using data from [4, 8, 16] nodes, while configurations with [32, 64] nodes are used to check the model performance. This scenario is of great practical interest, as such a model would be able to provide predictions without spending a lot of system resources.

Main Model

For this scenario, the whole dataset was used in a single model.

8. Model Results and Performance

This chapter presents the resulting models of the aforementioned scenarios. Specifically, it includes performance metrics (discussed in Section 5.2, “Model Performance Metrics”), a "Measured vs. Predicted Time" plot and the feature importances along with some comments, for all cases.

8.1 Message Size Scale Models

Table 8.1 Message Size Scale Models Metrics (Testing Set)

Message Size Scale	**R2**	**RMSE**	**MAPE**
Small Messages	0.785	0.162	0.239
Large Messages	0.876	0.475	0.201

Comparing the performance metric values for the two models, the fact that the large message model has a greater value of the R2 metric, means that it does a better job at explaining unseen data variance. This may have to do with a behavior that was mentioned in Section 4.2, “Communicational Parameters”, where for smaller messages, performance differences for smaller message sizes were not as clear as they were for larger messages. On the other hand, the smaller RMSE value of the small message model is explained by the fact that communication times for smaller messages have smaller absolute values.

Table 8.2 Message Size Scale Models Feature Importances

Feature	**Small Message Model**	**Large Message Model**
Message Size	0.52	0.65
Number of Messages	0.27	0.26
Number of Nodes	0.3	0.23
Processes per Node	0.24	0.19
Working Set Size	0.2	0.16
Computational Load Type	0.19	0.16

In both message size scales, the message size is the most important feature. However, in the case of smaller messages, some of the other features (especially the ones that have to do with the execution environment), have slightly greater importance when compared to the large message model. This was a behaviour also observed in the experiments presented in Section 4.2, "Communicational Parameters".

The following plots, show the predicted communicated times pitted against their actual values. Each point represents a different experimental configuration. The points that lie above the blue line are over-predictions, and the ones that lie beneath the line are under-predictions. In both of the following plots, where the color represents the message size, it is apparent that the points that are the furthest from the blue line are for configurations for relatively larger message sizes. When examining these kinds of plots for all the different feature color groupings, a general trend was seen (not only for these message scale models, but across all the scenarios). The greatest miss-predictions occur for the heavier communicational and computational loads and on the configurations with more nodes that are tightly populated. These cases are the ones that stress the system the most and thus may give way to relatively unpredictable behaviours.

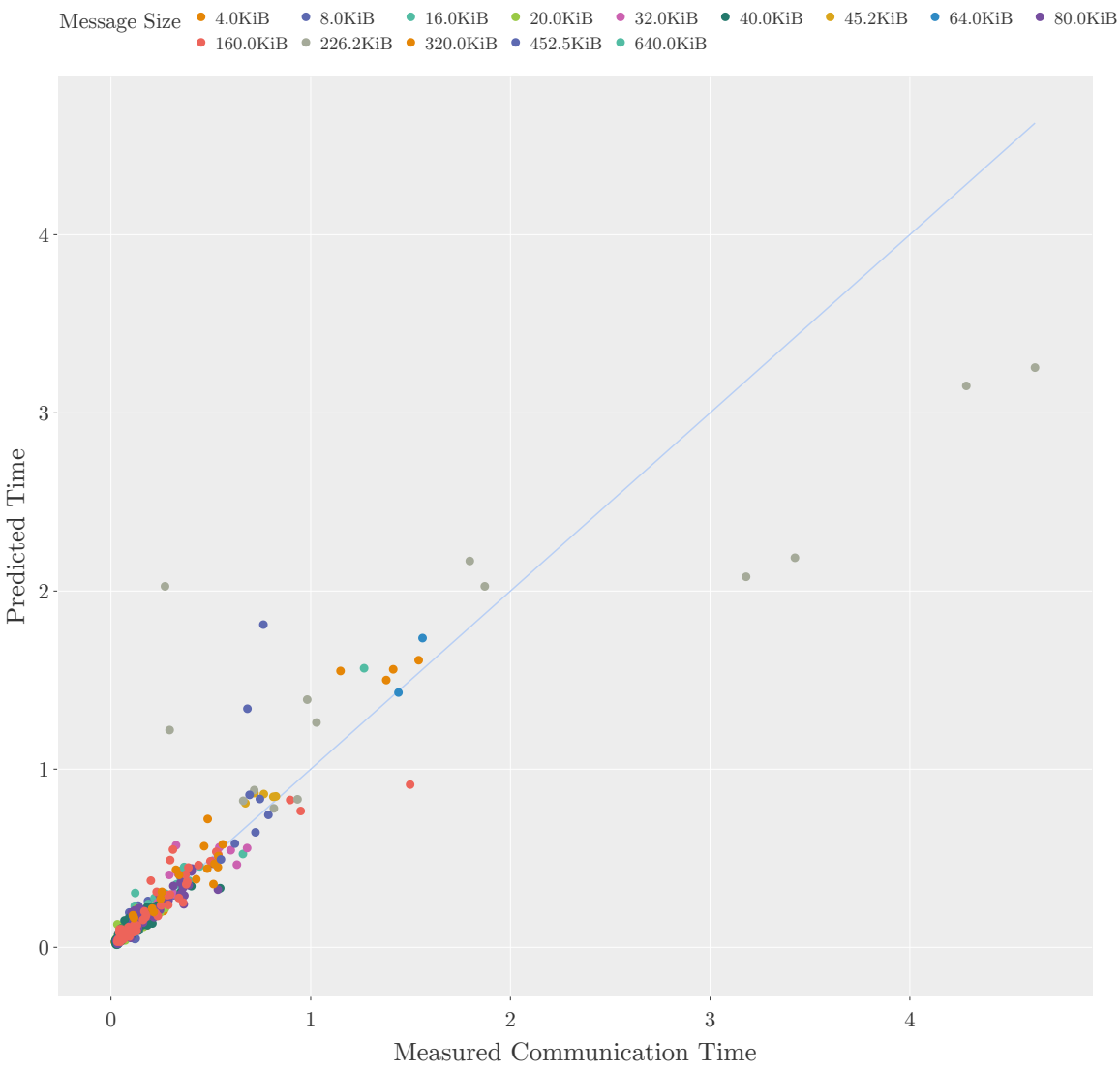

Figure 8.1 Small Message Model (Prediction vs. Actual)

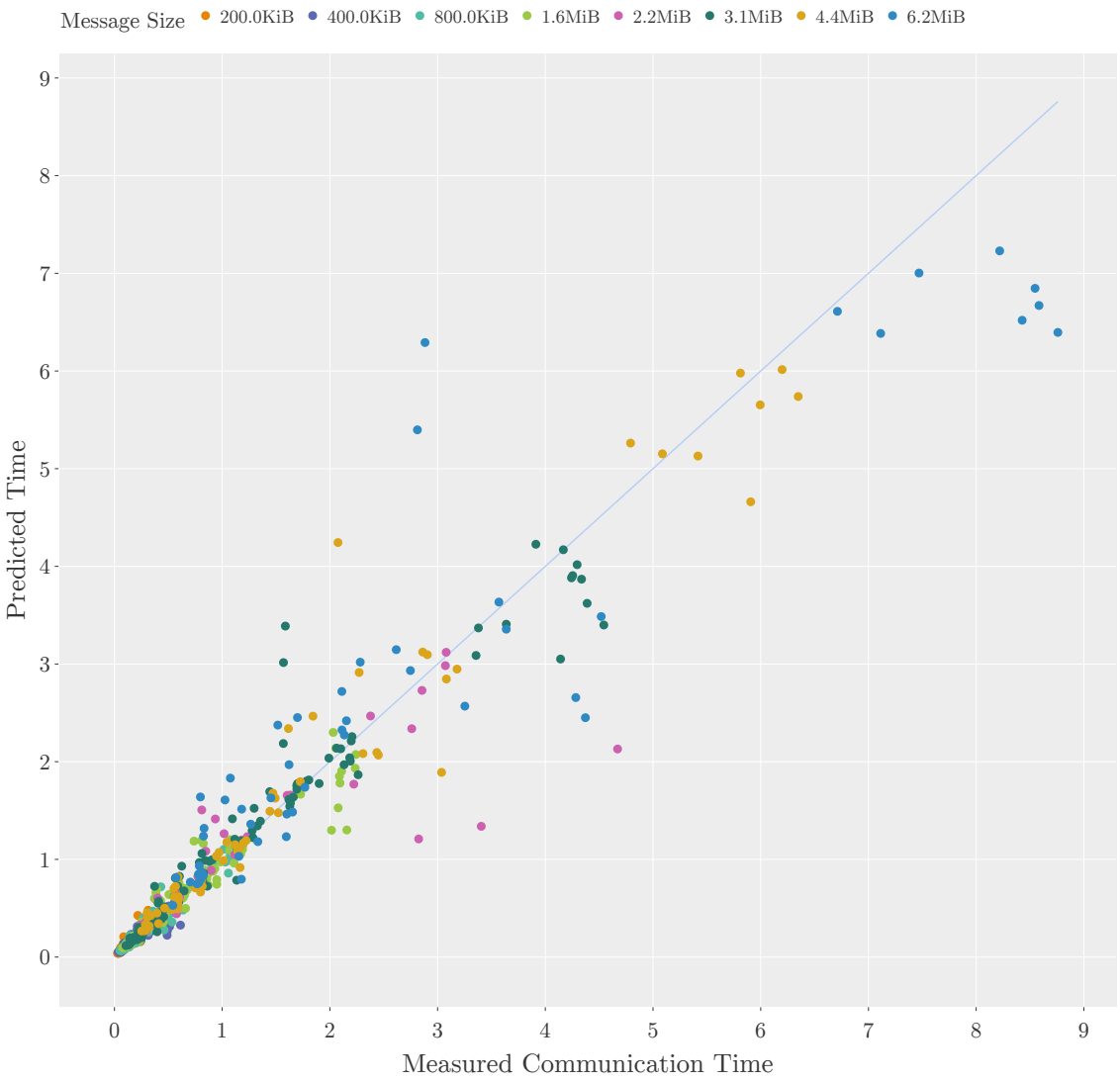

Figure 8.2 Large Message Model (Prediction vs. Actual)

8.2 Train Small/Test Big Model

Table 8.3 Train Small/Test Big Model (Testing Set)

Number of Nodes Subset	R2	RMSE	MAPE
4, 8, 16	0.854	0.326	0.241
32, 64	0.466	0.671	3.986

The above values, show that the model performs acceptably on the testing set of the [4, 8, 16] nodes subset, but there is a drop in general model performance for [32, 64] nodes. This is expected, since the model has seen no samples with the later number of nodes parameter values during training. However, considering the context of this scenario and after examining the results in the following plots, this approach may be beneficial since it would use fewer system resources for building a dataset.

Table 8.4 Train Small/Test Big Model Feature Importances

Feature	Importance
Message Size	0.7
Number of Messages	0.28
Number of Nodes	0.23
Processes per Node	0.21
Working Set Size	0.19
Computational Load Type	0.19

The above importances, show an expected behavior for a communication time model, where the most important features are relevant to communication, then follow the features that have to do with the execution environment and the computation phase parameters are last.

The following plots, once again, show this model's predictions against their actual values. For this scenario, the number of nodes color grouping was also included. This was done to show that most of the discrepancies between predictions and actual values occur for the unseen testing configurations of [32, 64] nodes. At the same time, the size of messages color grouping shows that configurations with larger messages are once again, more likely to be miss-predicted. The possible reason of maximum stress on the system that was mentioned in the previous scenario still stands.

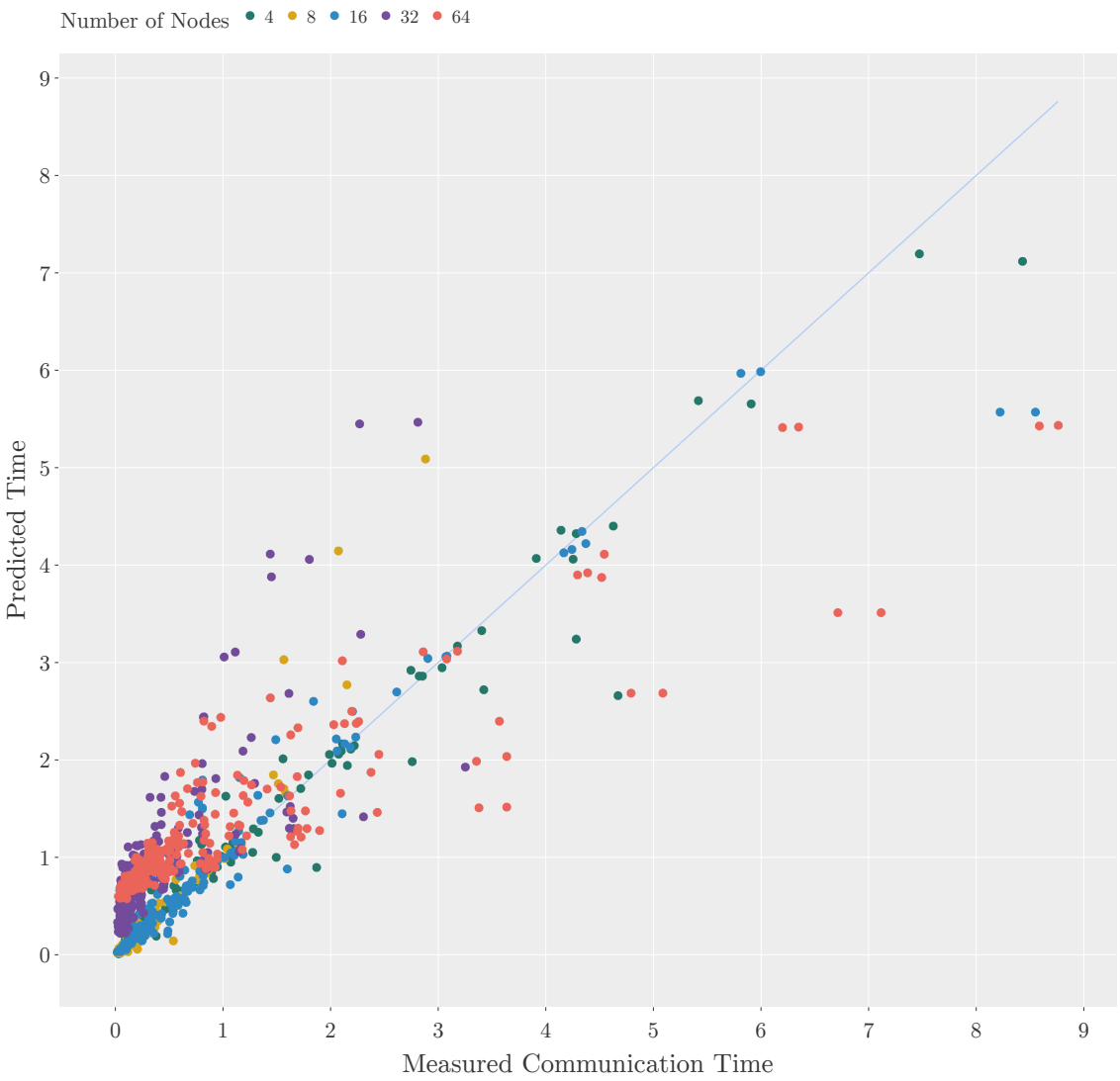

Figure 8.3 Train Small/Test Big Model (Prediction vs. Actual, grouped by Number of Nodes)

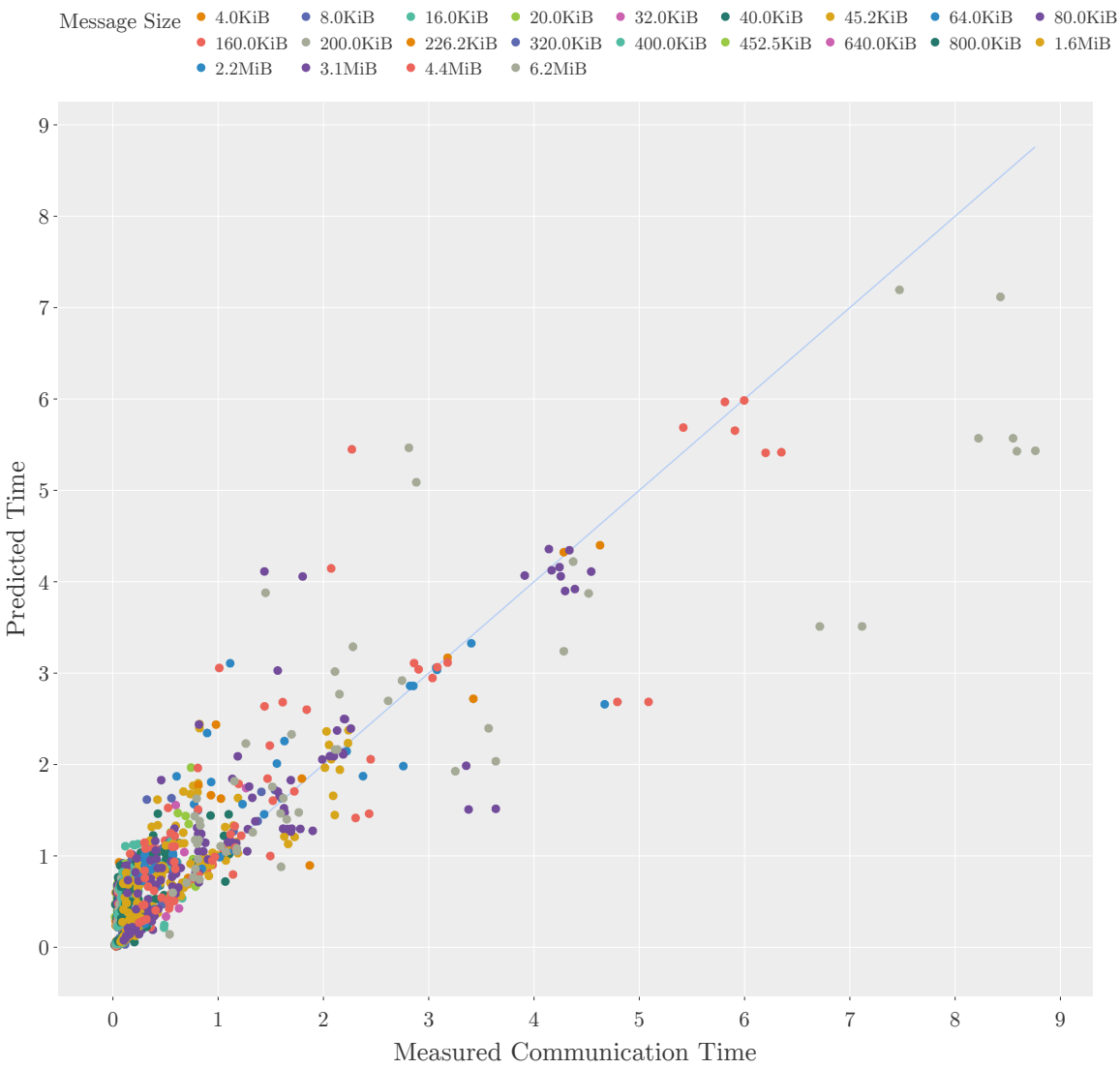

Figure 8.4 Train Small/Test Big Model (Prediction vs. Actual, grouped by Message Size)

8.3 Main Model

Table 8.5 Main Model Metrics (Testing Set)

R2	RMSE	MAPE
0.858	0.359	0.262

Table 8.6 Main Model Feature Importances

Feature	Importance
Message Size	0.66
Number of Messages	0.36
Number of Nodes	0.35
Processes per Node	0.29
Working Set Size	0.28
Computational Load Type	0.28

The performance metric values indicate that the main model has an acceptable performance for both unexplained data variance (high R2) and absolute predictive power (relatively low absolute and percentage errors). However, these metrics evaluate the performance of the model for data that originates from the data generator application. As mentioned, in the next chapter the model's performance is tested against the NAS BT pseudo-application.

When it comes to the feature importances, they seem to be somewhat more balanced compared to the previous cases. This may be a result of the larger size of the dataset that the main model is trained on. The greater dataset size may also have to do with the better overall performance, also seen in the following plot.

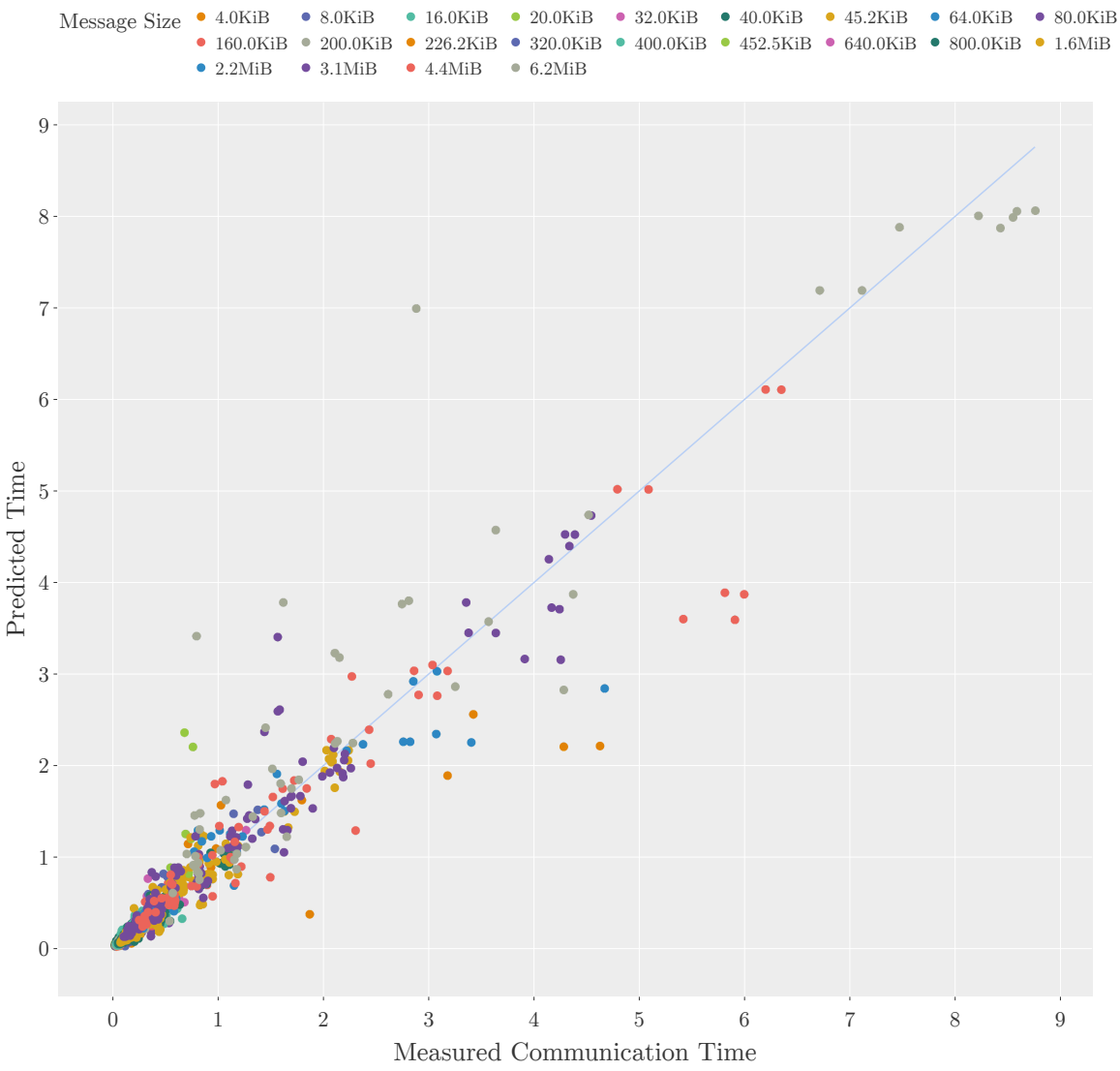

Figure 8.5 Main Model (Prediction vs. Actual)

9. Predicting the NAS BT Pseudo-application

In this chapter, the predictive power of the main model is tested against the NAS BT pseudo-application. To predict the performance of BT, several different configurations of the application were executed on different execution configurations. Attempting to predict the performance of this application, requires an understanding of its code and of the times that it reports in order to properly adapt each configuration to our model's features.

9.1 Analysis of the NAS BT Kernel

One of the reason this pseudo-application was chosen, is that its computational kernel bears some similarities to the data generator application kernel. By examining the application's code and with the help of the analysis provided in [Van der Wijngaart et al., 2012], a better understanding of BT's execution was gained. Specifically, there are four distinct phases of execution that repeat over a time for-loop and occur in a 3-dimensional datagrid as shown in Listing 9.1. The first phase is a stencil computation on all data points, while the other three, x/y/z-solve have two sub-phases, Forward (Gaussian) Elimination (FE) and Back-Substitution (BS) where both communication and computation occur in a regular pattern on the three different grid dimensions. For each of these phases (and sub-phases) communication and computation times are measured separately.

```
for time:
    rhs

    xSolve

    ySolve

    zSolve
```

Listing 9.1 Overview of NAS BT's kernel

To properly adapt the measurements from the different phases of NAS BT to be compatible with the main model that uses data from the data generator application, a search for the pattern of the data generator application kernel (simplified in Listing 2.3) was conducted in all the above phases. It was concluded that the two sub-phases of FE and BS for each dimension show this pattern but with different data sizes (working set and message size) between them. Listing 9.2 shows the simplified pseudocode for the FE phase of xsolve. As mentioned the BS phase is similar but with different data sizes and both of these phases are repeated for all three dimensions. Figure 9.1 shows a simplified schematic comparison of BT and the data generator application.

```
for xDimension:
    xSolveCellFE // performs computations on the other two dimensions

    xExchangeSolveFEInfo // exchange of 2D faces

    MPI_Waitall(MessagesToSend,MessagesToRecv)
```

Listing 9.2 NAS BT xsolve FE phase

A similarity between Listing 9.2 and Listing 2.3 can be observed. Namely, the outer time for-loop in the data generator application is replaced by an outer dimension loop in the FE phase, and the computations and data exchanges occur in 2D data faces.

With those observations in mind, the way that data from BT was adapted, is by taking the FE and BS phases for one dimension as two different data generator configurations. For each of those, we consider the outer time and dimension loops as one larger time for-loop and make the appropriate division to the measurements to match the data generator application outer-loop. A simplified, schematic comparison of the two kernels can be seen in Figure 9.1.

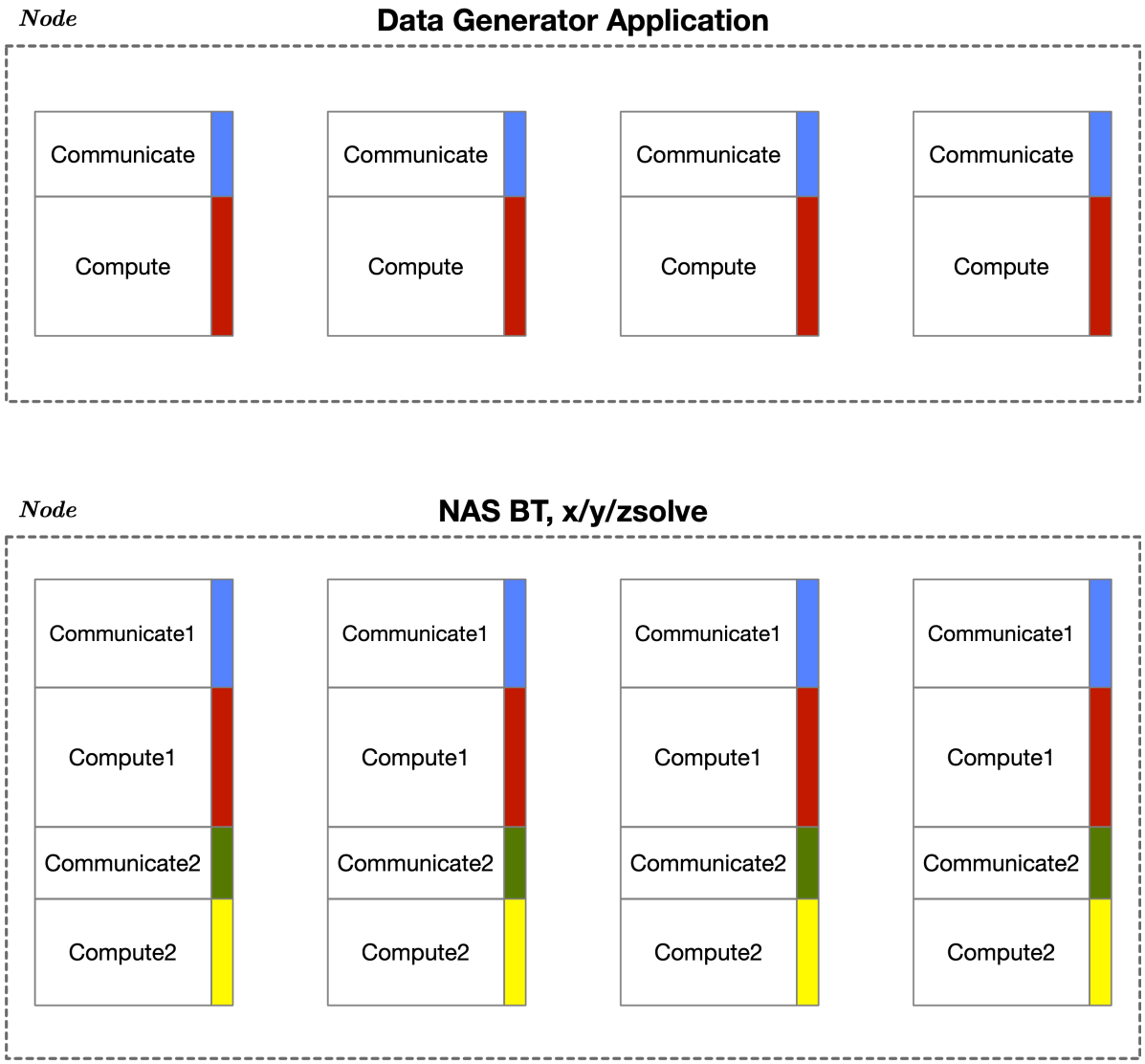

Figure 9.1 BT vs. Data Generator Application

9.2 Results

The following plots, show the predictions made on data adapted from several different configurations of NAS BT. Figure 9.2 groups the different configurations by the number of nodes used, while Figure 9.3 highlights the absolute error between the actual values and the predicted ones.

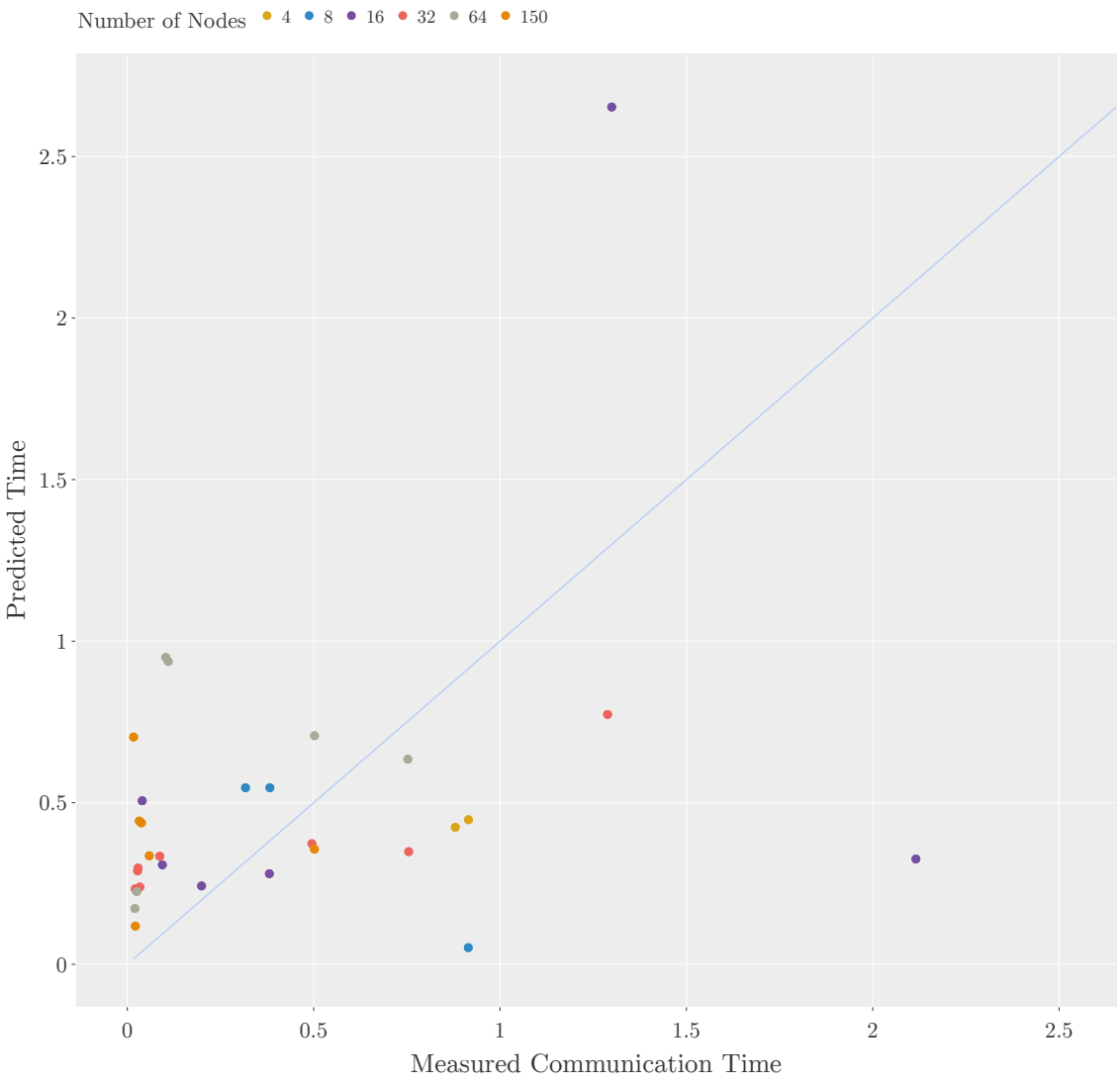

Figure 9.2 NAS BT Main Model Prediction vs. Actual (Grouped by Number of Nodes)

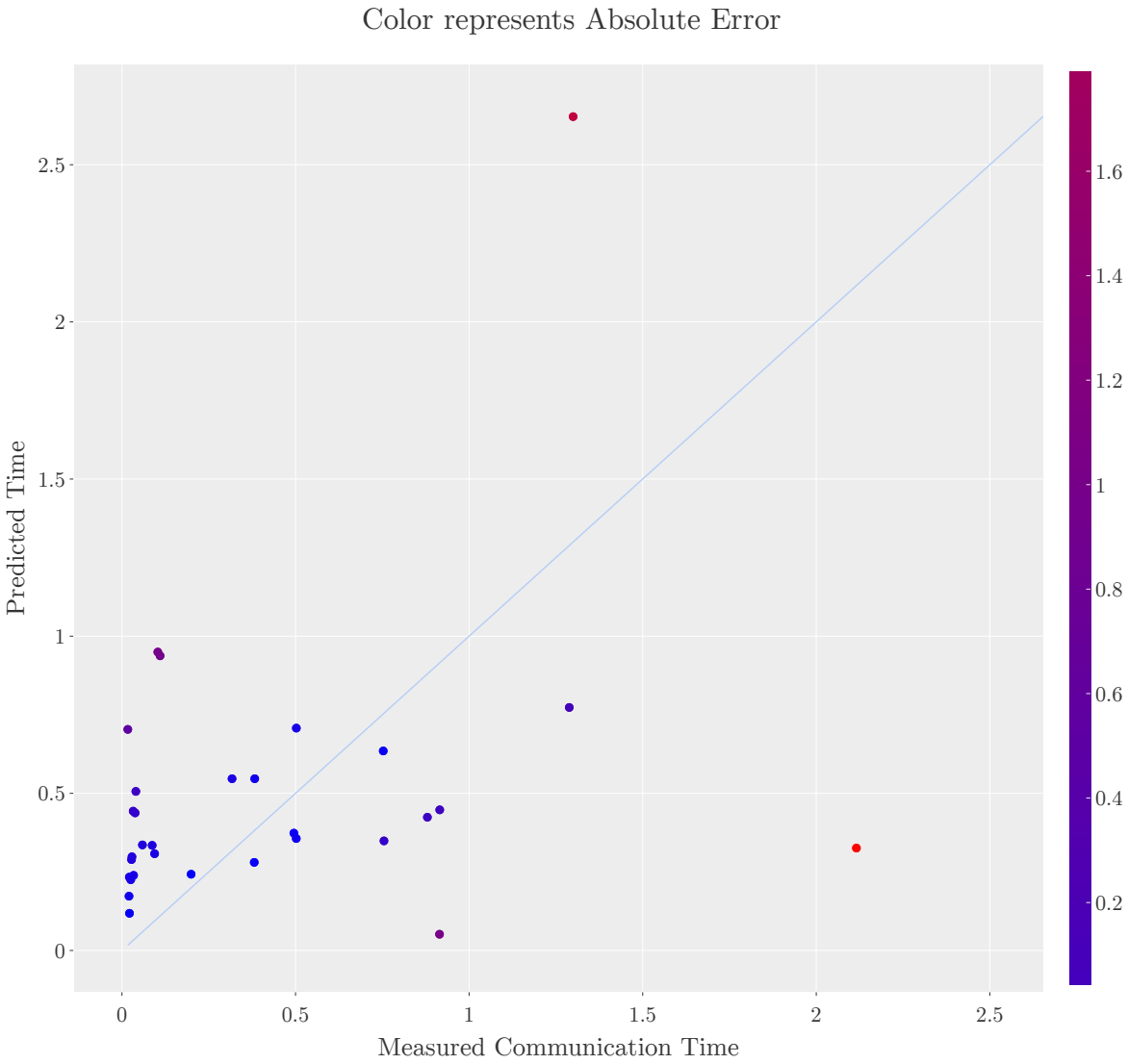

Figure 9.3 NAS BT Main Model Prediction vs. Actual (Colored Absolute Error)

While the values of the absolute error are seemingly low, considering the actual values of several points, the model mispredicts a large number of the different configurations. This can be attributed to several factors, including the 'naivety' of the data adaptation that was described in the previous section. Another factor may hide in an observation made when inspecting the reported data by BT. Specifically, in a lot of cases there was a large discrepancy (order of magnitude) between the maximum, the minimum and the average times reported by all processes. For the context of this thesis, it was chosen to not dive deeper into the peculiarities of this pseudo-application. Instead, this attempt to predict a whole different application served as an opportunity to understand how kernels with similar parts may behave differently when combined in different ways and configurations. It was also another proof of the model complexity/performance tradeoff, where the simple code of the data generator application and the simple data adaptation, attributed to the model under-performing.

References

The Intel® MPI Benchmarks

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken *LogP: Towards a realistic model of parallel computation* volume 28 ACM 1993

Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman *LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation.* In Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, pages 95-105 ACM 1995

Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara *LogGPS: a parallel computational model for synchronization analysis.* ACM SIGPLAN Notices, volume 36 pages 133-142 ACM 2001

Torsten Hoefler, William Gropp, William Kramer, and Marc Snir *Performance modeling for systematic performance tuning.* State of the Practice Reports, page 6 ACM 2011

Nikela Papadopoulou, Georgios Goumas, and Nectarios Koziris *Predictive communication modeling for HPC applications.* Cluster Computing, volume 2, number 3, pages 2725-2747 2017

Efstratios Karapanagiotis and Georgios Goumas *Parallel application performance prediction in shared memory architectures.* 2023

Michael McCool James Reinders Arch Robison *Structured Parallel Programming: Patterns for Efficient Computation (1st. ed.)* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 2012

Torsten Hoefler and Roberto Belli *Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results.* In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15). Association for Computing Machinery, New York, NY, USA Article 73, 1–12. 2015

Friedman Jerome *Greedy Function Approximation: A Gradient Boosting Machine.* The Annals of Statistics, volume 29.

Breiman Leo *Random Forests.* Machine Learning, volume 45, number 1, pages 5-32 2001

R. F. V. der Wijngaart, S. Sridharan, and V. W. Lee *Extending the bt nas parallel benchmark to exascale computing* pages 1-9 2012