Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Guadalajara

Escuela de Graduados en Ingeniería y Arquitectura (EGIA)

Maestría en Ciencias de la Computación

Technical Comparison of an IoT System and a Traditional Computing System

Victor Manuel Rodriguez Bahena

Advisor: Marcos Ruben de Alba Rosano

Guadalajara (Jal), 19 de abril de 2016

Acknowledgments

First and foremost, I have to thank my parents for their love and support throughout my life. My brother and loved ones deserve my wholehearted thanks as well. Thank you for your understanding, patience and encouragement in many difficult moments.

I would like to express my sincere gratitude to my advisor PhD. Marcos Ruben de Alba Rosano for the continuous support of my master degree study and related research, for his patience, motivation, and immense knowledge.

I would like to sincerely thank Intel company as well as the Monterrey Institute of Technology and Higher Education for his support with the scholarship and support to finish this achievement in my life.

Summary

The use of IoT devices is rising quickly, by 2022 is it expected to have 14 billions of IoT devices creating a rise of data processed, transmitted and stored. It would create so much traffic that would be similar to a security attack. Apart from the problems of transmitted data, power consumption of computing systems that store and process data generated by IoT devices is important to be studied. If current trends continue, a system to handle IoT data would require 100 megawatts. This research work proposes a method to improve the use of IoT platforms by reducing the amount of data transmitted, as well as, the amount of high performance servers needed for data processing. At the end a detailed case of study based on industrial benchmarks for distributed systems, power consumption analysis and optimal operating system selection, describes a methodology to maximize power efficiency.

Table of Contents

List of Tables

List of figures

Motivation

Every day we are in contact with computing and sensing devices everywhere. Our daily activities, for example exercising, could be tracked by multiple sensors that measure heart rate, distance, speed and consumed calories. These devices, which are provided with unique identifiers and the ability to transfer data over a network, are part of the Internet of Things (IoT) technology.

IoT systems are more than just computing and sensing devices. They include storage and information processing for sensors to display meaningful information on an easy to access user interface.

The use of IoT devices is rising quickly, as well as its problems. According to [1] by 2022 it is expected to have 14 billions of IoT devices running in the world. The rise of IoT will lead to an explosion in the volume of data transmitted, collected and processed. In addition, the power consumption of computing systems that store and process data generated by IoT devices is a very important design restriction to make these systems successful. If current trends continue, a system to handle IoT data would require 100 megawatts [2].

However, the computational power that the IoT systems have, might be enough to process all the data they generate. As theoretical physicist Michio Kaku mention in [3]: *"Today, your cell phone has more computer power than all of NASA back in 1969, when it placed two astronauts on the moon"*. This fact motivated this work to investigate whether if it is possible that a network of IoT systems could process the data they generate on their own at real time. In order to answer that question it is necessary to make a technical comparison between a network of IoT systems and a traditional computing system used for storage and processing of data coming from IoT devices.

This thesis focuses on the development of a methodology to determine if it is necessary to send sensors data to a centralized processing system, or if it can be managed by a network of IoT computing systems.

Justification

Nowadays, there are multiple examples of IoT systems with large processing capabilities, which could be exploited to avoid sending data to a centralized processing system. Despite of this characteristic, it is frequently seen that there is a large variety of IoT applications that continue sending data over the Internet to a centralized processing system, without using the processing capabilities of IoT devices.

There are several IoT examples in which the system performs remote control of the environment through the Internet. In all of them it can be seen that applying IoT technology it is possible to improve the operation and system flexibility, by using the wireless sensor network instead of the traditional wired sensor network. In several cases, it is observed that such systems dedicate a general-purpose microprocessor just to transmit data to the centralized control system; even though, the problem being solved in such type of centralized control systems is a simple conditional rule. However, there are other projects that use ultra-low power microprocessors. There are several industrial applications that use embedded systems for autonomous robots. In such scenarios embedded platforms are chosen due to their low power consumption and adequate processing capabilities. Projects where an embedded system is the central processing unit are not common. Due to the fact that traditional centralized processing systems are used to collect data, there is an excessive use of IoT devices for transmitting that data.

There are sources that demonstrate that the total amount of user data to be stored, or processed doubles every two years. The first problems due to this trend are currently being faced by industry; such is the case of the aircraft industry.

In 2013, the Virgin Atlantic® airline announced that the Boeing 787® aircraft was going to be one of the first highly connected planes. Each one of these airplanes was expected to create over half of a terabyte of data per transatlantic flight. Due to the costs of handling and transmitting such amount of data daily, the airline started to look for cloud-based solutions inside the airplanes. This is an expensive solution; because the cost

in space and energy required to handle a cloud-based solution inside an airplane is too high.

Current IoT architectures are not efficiently using the computing resources they have. In addition to this, the lack of standard methodologies to design distributed self-sustainable networks of IoT systems is causing problems in the industry. Such and other reasons, which will be discussed in the document, justify pursuing a quantitative comparison of IoT distributed low power computing systems to high-performance computing systems in terms of power efficiency.

CHAPTER 1

Introduction

This work presents a detailed study of how within a network of embedded systems, computing devices collaborate within each other to solve parallel problems. At the end a detailed case of study based on industrial benchmarks for distributed systems, power consumption analysis and optimal operating system selection, describes a methodology to maximize power efficiency.

1.1 Background

The evolution of computing technology has made an incredible progress since the first general-purpose electronic computer was created. Today, less than 300 dollars will purchase a personal computer that has more performance, more memory, and more disk storage than a computer bought in 1985 for 1 million dollars[4]. Incredible advances have been possible thanks to innovations in computer and software design.

Since the commercial use of computers (started in 1951 with the introduction of UNIVAC [5]) the development of computers has had multiple changes, not only from the architectural point of view but also from the application point of view. In [4] the authors present the evolution of the computers in term of performance over the years since 1980. The first important breakpoint in the history of computers came in the beginning 1970s whit the emergence of the microprocessor.

The 1980s saw the rise of the desktop computer based on microprocessors, in the

form of both personal computers and workstations. The 1990s saw the emergence of the Internet and the World Wide Web, the first successful laptop computing devices, and the emergence of high performance computing systems for business purposes (servers). During these years the performance of the microprocessors increase around 27% each year [4].

However, in the first years of the 2000 decade a dramatic change in the computer architecture happened. The idea of increasing computing performance in processors, based mainly on increasing the clock frequency, could not longer be held. This is due to the fact that the power density (amount of power per volume unit) is multiplied with increases in frequency.

The solution, to keep meet Moore's [6] law, was shifting to real parallelism by doubling the number of processors on the die. This was the birth of multi-core microprocessors area. The idea considered, increasing the computing performance by limiting the power density at the same time. This new computer architecture broke the paradigm of using small microprocessors for embedded platforms. Instead of this it was possible to have more computing power with less frequency. Thanks to this radical change there has been a rapid evolution of the computing and multimedia capabilities on embedded systems.

According to [7] *"An embedded system is a special-purpose system in which the computer is completely encapsulated by the device it controls"* Unlike a general-purpose computer, an embedded system performs pre-defined tasks, usually with very specific requirements. Examples of these are: microwave ovens, washing machines, printers, and GPS (Global Positioning System) systems. All those electronic based appliances that started to emerge around 35 years ago [5].

The variety of embedded applications requires a wide spread of processing power and cost. They include 8-bit and 16-bit processors that may cost few cents, 32-bit microprocessors that execute 100 million instructions per second and cost less than few dollars, and high-end processors for the newest video games or network switches that cost at least 100 dollars and can execute one billion of instructions per second [4].

Continuous design tradeoffs of factors like: cost, size, power density, performance and connectivity has caused the computing technology to evolve rapidly, today ubiquitous computing is a reality. According to Mark [8], ubiquitous computing is *"the method of*

enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user". This means that computing power could be made available anywhere and at any time.

According to [5], there is a transformation from ubiquitous computing to advanced ubiquitous computing. Advanced ubiquitous computing is an extension of ubiquitous environment that improves connectivity between devices. The main characteristics of such environment can be listed as follows: a large number of heterogeneous devices, new communication technology, and Internet of Things (IoT) among others.

One of the most accurate definitions of IoT given by [9] where it mentions that *"Internet of Things refers to physical and virtual objects that have unique identities and are connected to the Internet to facilitate intelligent applications"*. IoT enables interconnection, via the Internet, of computing devices embedded in everyday usage objects, enabling them to send and receive data. Differences respect to traditional embedded systems include Internet connectivity and smaller power consumption. IoT systems must always be connected to the Internet and require lower power consumption. An IoT solution has the following parts (figure 1.1).

- The Thing (computing devices): In the Internet of Things, a thing can be any natural or man-made object that can be assigned a unique identifier and provided with the ability to transfer data over a network.

- Network Connection: Network Connections provide connectivity between computing devices and the Internet, a network, or another computing device.

- Cloud Computing data centers for storage and big data analysis: The data by itself is not useful to the end user. After data is sent and stored into the cloud computing data centers is necessary to run big data solutions that present meaningful information to users.

- Presentation Devices: Dashboards have to be hosted on some kind of display, it could be a desktop computer running an application, a tablet, or a smart phone accessing to a web page. It could even be a purpose-built device like a retail kiosk,

an intelligent vending machine or a control panel. The goal is to present information coming from the big data analysis.

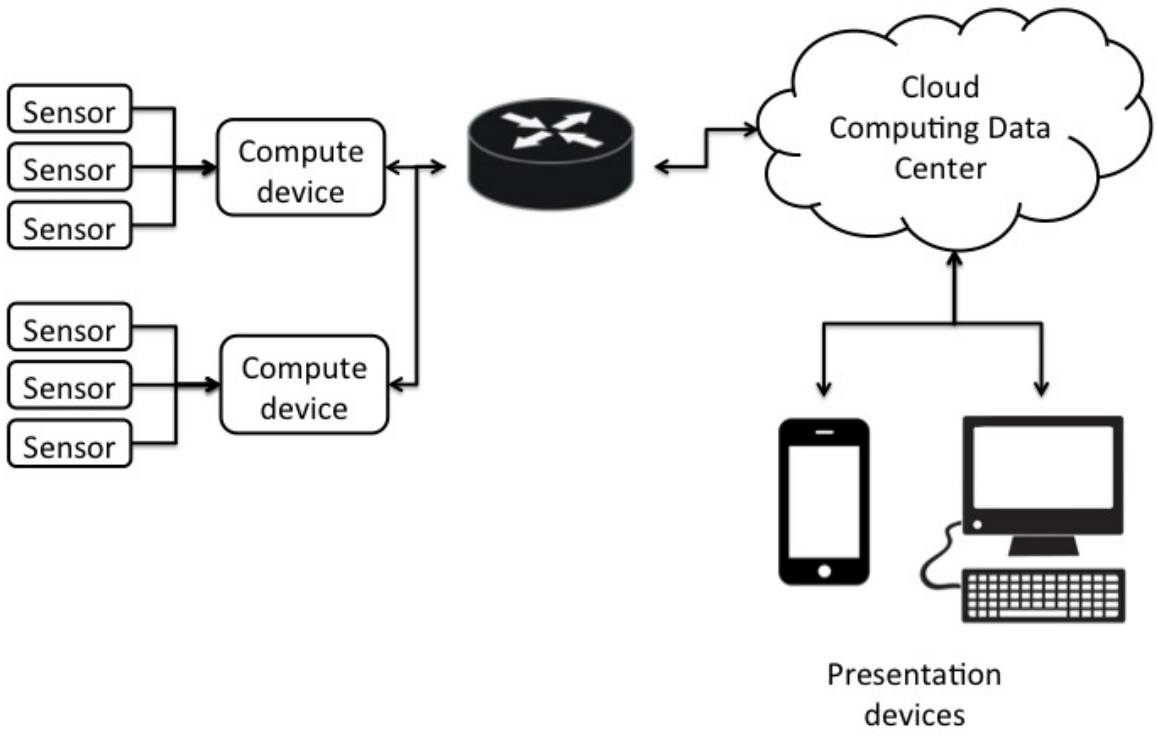


Figure 1.1: An IoT system diagram

The IoT revolution lacks of industry standards. There are thousands of computing devices and sensors from different vendors that appear on the market every day, each one with its unique way to send or store data in centralized cloud computing centers.

There are two main projects that are organizing information to establish standards for IoT communications:

- Open Connectivity Foundation (OCF)[10]: The OCF tries to create a set of open specifications and protocols to enable devices from a variety of manufacturers to securely and seamlessly interact with one another. Regardless of the manufacturer, operating system or chipset the devices that adhere to the OCF specifications should communicate together.

- AllSeen Alliance (before known as AllJoyn) [11]: Is an open source project for the development of a universal software framework aimed at the interoperability among heterogeneous devices, dynamic creation of proximal networks and execution of distributed applications. The framework provides a common interface towards smart devices.

Both standards try to solve a simple problem: To establish communications standards among the industry. This means that all IoT devices could communicate with each other in spite of the vendor type. Regardless of these, none has defined a common standard in terms of IoT technology.

On the other side we daily use computerized technology like: air conditioners, televisions and cars; many of them connected to the Internet, In spite of the existing efforts to develop standard network protocols for IoT systems there is no one to make the IoT systems analyze their data on their own , instead of sending all data to cloud data centers for analysis. This is a problem in the short term because the solution might require to add another server (which may have space and economic constrains).

1.2 Problem Definition

The use of IoT devices is rising quickly, according to [1], by 2022 is it expected to have 14 billions of IoT devices creating a rise of data processed, transmitted and stored. Considering a scenario where all those IoT devices would try to send 1 kilobyte of data to centralized servers at the same time; it would create so much traffic that would be similar to a security attack [1], that would collapse centralized data centers.

Apart from the problems of transmitted data, power consumption of computing systems that store and process data generated by IoT devices is important to be studied. If current trends continue, a system to handle IoT data would require 100 megawatts. [2].

Despite of the variety of IoT applications, like those mentioned in [13] [14], all of them are based on the same IoT principle of sending data over the Internet to a centralized

[1] In the DDoS attack, high amount of network traffic with maximum performance are generated and transmitted to the target systems[12])

processing system without using the processing capabilities of the IoT device. In many IoT designs [14] it is used a dedicated microprocessor just to transmit data to the centralized control system; regardless that the logic of the centralized control system is just a simple conditional rule. On the other hand, there are very few examples, like those listed in [15], where the computing power of the embedded platforms is exploited. If the IoT industry continues creating solutions without optimizing the use of the computing power of the embedded platform sending data over the Internet; then more critical examples as that of Boeing 787® [2] will be seen.

This research work proposes a method to improve the use of all IoT platforms by reducing the amount of data transmitted, as well as, the amount of high performance servers needed for data processing.

1.3 Main Objective

The main objective of this work is to show how an IoT network could be self sustainable. It means to make it solve their own computing problems without the need to send data to data centers (or to the cloud). The work will study when is necessary to send data to data centers or the cloud. It will estimate the maximum number of embedded platforms that provide the maximum level of performance with the minimal amount of power consumption. At the same time, a list of applications that are good candidates for that approach is presented.

This work intends to provide to the IoT industry a way to determine whether applications can take advantage of the communication between IoT devices to process their own data instead of sending information to data centers.

1.4 Hypothesis

On recent years it has been observed that unsustainable power consumption has driven

[2] The Boeing 787® aircraft ordered by Virgin Atlantic® for delivery dramatically increases the volume of data the airline will need to deal with (half terabyte in a transatlantic flight). Because they can't handle that much terabytes of data everyday coming from various airplanes they are looking for adding servers inside the airplanes [16]

the microprocessor industry to integrate multiple cores into a single die, or multicore, as an architectural solution in order to increase performance and reduce power density. The same approach might be followed for IoT systems by creating a self sustainable network of IoT systems, where processing could be solved with parallel computing technology and distributed clusters).

With the current computing power of ultra-low-voltage microprocessors platforms (which are core systems of the IoT devices) it is possible to create a cluster with the optimal number of embedded platforms. All interconnected in a network that provides maximum performance with small power consumption. This characteristic is determined by power efficiency of the network. The power efficiency is quantified by performance per watt [17].

The key result is to determine at what point it is better to send data to data centers. This consist on finding what is the maximum number of systems that this kind of network can handle with optimal energy efficiency.

The development of metrics to evaluate energy efficiency on the basis of performance and power models is described in [18]. According to [18], the formula for performance per watt (Perf/W), which represents the performance achievable at the same cooling capacity [3] (Watts/h) based on the average power (W) is shown bellow in 1.1:

$$\frac{Perf}{W} = \frac{1}{(1 + (n-1)k(1-f))} \tag{1.1}$$

According to Amdahl's law[18], the formula for computing the theoretical maximum speedup (or performance) achievable through parallelization is as follows:

$$Perf = \frac{1}{(1-f) + \frac{f}{n}} \tag{1.2}$$

Where n is the number of processors, f is the fraction of computation that can be parallelized (a value from 0 to 1) and k, to represent the fraction of power the processor

[3]Cooling Capacity refers to the amount of cooling that system can remove. It is measured as the number of BTUs per hour of heat that the unit can remove from the air. (A BTU or British Thermal Unit). It can also be measured as Watts per hour or tons (the amount of water at a given temperature that can be frozen in a given amount of time) [19]

consumes in idle state (a value from 0 to 1).

In [18] it is demonstrated that a symmetric many-core processor can easily lose its energy efficiency as the number of cores increases. To achieve the best possible energy efficiency, it is suggested a many-core alternative, featuring many small, energy-efficient cores integrated with a full-blown processor. It is shown that having knowledge of the amount of parallelism available in an application prior to execution, it is possible to find the optimal number of active cores to maximize performance for a certain given cooling capacity and energy in a system.

In this thesis work it is proposed the hypothesis that a cluster of ultra-low power IoT platforms can be as computing powerful and energy efficient as the system described in [18], but in this case n is the number of ultra-low power platforms instead of processor cores.

Results of a simulation to illustrate such hypothesis is described in Figure 1.2. At the beginning , the increment in the number of nodes in the studied network produces an increment on performance; however, it is expected to reach a maximum point at which power efficiency becomes stable and it will remain in such state up to a certain point at which it will start to decrease. Such behavior is expected, as the number of platforms increases it is expected to get some performance benefit, because the amount of work to be done is distributed among different platforms, but as more are added due to the power they consume the performance gain starts to minimize. When the ideal number of platforms is exceeded, the power efficiency decrease rapidly.

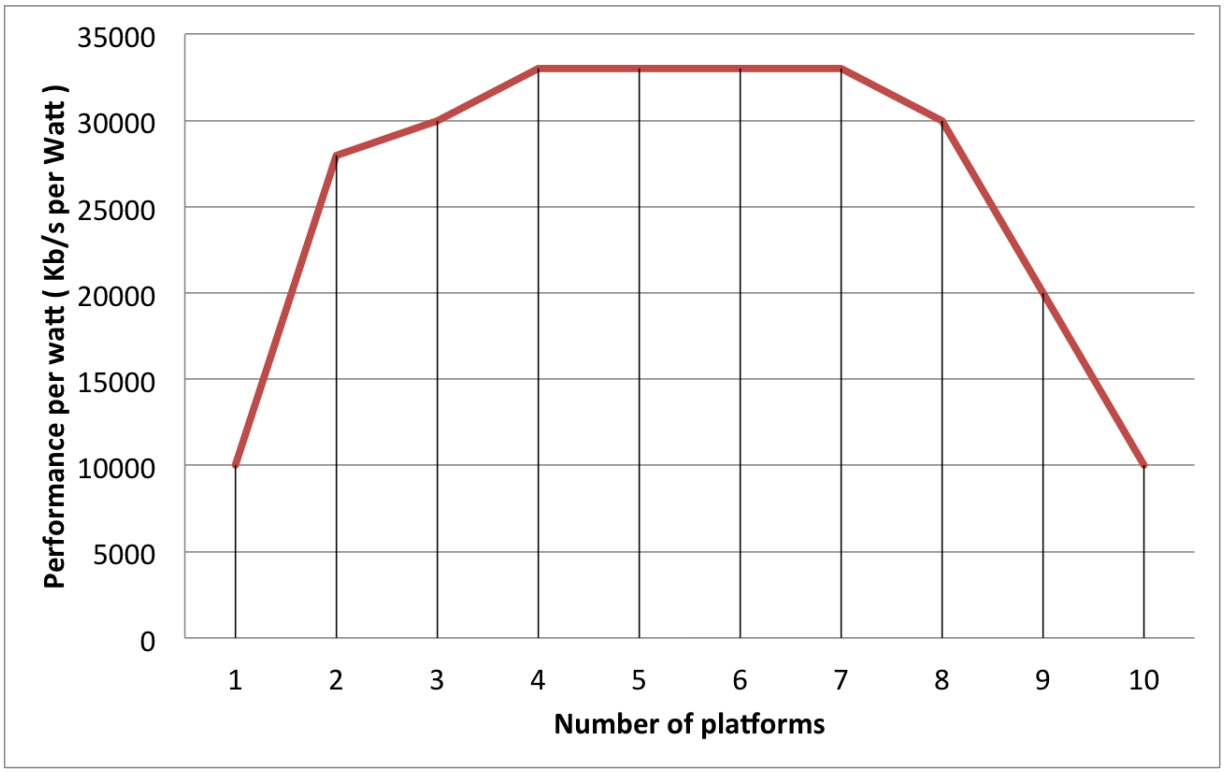


Figure 1.2: Hypothesis of energy efficiency behavior in embedded cluster

After finding these curves for command benchmarks it will be easy for the industry of IoT systems to determine if their applications can take advantage of communicate their IoT devices among each others instead of sending the information to their data centers.

1.5 Methodology

In this work it is investigated the number the number of platforms to obtain the best power efficiency for a given set of benchmarks. Recent research work [20] [21] [22] [23] demonstrate an increasing interest in the topic. Such work focuses on three main fields:

- Study of low-voltage platforms with large computing power.
- Study of Operating Systems for IoT platforms
- Workload balancing schemes for embedded platforms

The methodology proposed in this work includes the following steps:

- To select the correct embedded platform: There are dozens of embedded and IoT platforms, this is required to analyze and select the best platform.
- To select a set of benchmarks. Is important to identify a set of benchmarks that covers a variety of IoT applications.
- To select an Operating System for the system
- To create embedded clusters for energy efficiency measurements. This consist of the integration of selected hardware, operating system and computing paradigm.
- To find the optimal number of embedded systems in the cluster that provides the highest power efficiency.
- To release improvements and disagreements as open source.
- To implement a solution for a greenhouse. Implementation of an IoT system to monitor environmental parameters of a greenhouse.

CHAPTER 2

State of the Art

This chapter describes recent research work on parallel and distributed computing in embedded platforms and clusters of embedded systems.

2.1 Embedded Distributed Computing

With the proliferation of high performance workstations and the emergence of high speed networks in the early 1990s, the rise of interest in parallel and distributed computing grow rapidly . The first clusters of computers required communication protocols to share data and workload. In [24] one of the most popular paradigms for parallel and distributed computing is described: Message Passing Interface, MPI. The main objective of MPI was to provide a library for the development of parallel and distributed applications with message passing primitives, as well as to deploy a host network to share the workload among the workstations.

In [24], it is published MPICH, the first practical implementation of the MPI protocol. MPICH is a high performance and widely portable implementation of the MPI .[25]. MPI was designed for high performance computing on both, massively parallel machines and on workstation clusters.[1]. It provides libraries for C and Fortran programing languages. It allows the development of applications to share data and workload between multiple computing nodes. From the point of view of the programmer it is necessary to specify what section of the code needs to be run in parallel at the nodes. Although MPICH was

[1]More details about MPICH are covered in Chapter 3

one of the first MPI implementations, it is not the only one; nowadays, there are multiple that follow the same principles.

MPI implementations are designed for clusters of high performance computers. Clusters where it is expected to have a large amount of resources with operating systems that enable distributed computing; however they consume. Despite of the fact that many researchers use MPI implementations in the high performance computing community, there is a small group of people interested in the same approach of parallel and distributed computing applied to embedded systems. One of the first approaches was eMPICH [22]. eMPICH has two design paradigms for embedded MPI: top-down and bottom-up.

- The top-down approach focused on removing and simplifying features of full MPI implementations, and reorganizing the code in order to decrease the amount of unneeded code linked into an application. Some examples of changes include: to remove support for data heterogeneity, removing error checking and reducing multiple send modes. These changes reduce the complexity of MPI to send, receive and handle messages.

- The bottom-up approach implemented a full rewrite of the MPI protocol First it was a very small, six-function version of eMPICH. Latter it was created a sequence of eMPICH libraries by adding various functionality to the base implementation.

In [22] it is discussed that besides reducing the executable code size of the MPI layer, there are other issues that need to be addressed within the scope of embedded MPI. Many of them are related to the lack of proper operating system for embedded platforms. The operating systems is responsible to manage techniques for inter-processor communication. In order to create a parallel and distributed computing system for embedded platforms it is necessary to solve these problems first.

2.1.1 Thread-based approach

To achieve the implementation of MPI for embedded platforms it is required to have several system resources for example: volatile and non-volatile memory, as well as the use

of a microprocessor instead of a low-voltage microcontroler with comprehensive operating system support. Those resources were not available in an embedded platform at the end of 1990s. One of the projects that follows this approach is AzequiaMPI [26].

AzequiaMPI is an MPI implementation that uses threads instead of processes, making MPI applications lighter . The AzequiaMPI protocol runs on the Texas Instruments® TMS320C6000® family of digital signal processors (DSPs). [2]. Threads are used for small tasks, whereas processes are used for larger tasks. Another difference between a thread and a process is that threads that run within the same process share the same address space, whereas different processes do not.

According to [26], running an MPI node as a process imposes some disadvantages, process context switches and synchronization are expensive. Second, message passing between two MPI nodes in the same machine must go through a system buffer, affecting the communication efficiency.

The idea that [26] proposed was radical, using threads instead of processes. In the original idea presented by [24], the MPI node is a process [3], in contrast in [26] the MPI node is a thread in a thread-based implementation.

Before the publication of [26] the idea of thread was popular. In [27] it is proposed TMPI (threaded MPI), a technique that allows MPI nodes to be executed safety and efficiently as threads. According to [27], current MPI implementations on shared-memory machines map each MPI node to an OS process, which can suffer serious performance degradation in the presence of multiprogramming. To reduce the problem they propose a runtime support that includes an efficient communication protocol that uses lock-free data structures and takes advantage of address space sharing among threads. The experiments on [27] show that TMPI has significant performance advantages in comparison to a traditional MPI approach.

Another example of thread-based techniques is [28] . In such work TOMPI (thread only MPI) is presented. TOMPI is a threads only implementation of MPI. While the implementation is partial it supports the commonly used MPI features. The communication

[2]The DSP is a specialized microprocessor with its architecture optimized for digital signal processing tasks. DSPs are broadly used in high performance real time signal processing applications.

[3]Other MPI implementations that do the same include MPICH and OpenMPI

is designed to be efficient by using asynchronous communication by default. The idea was to extend TOMPI to efficiently support the entire MPI standard; however it was just an early proof of concept prototype that implements just a reduced set of MPI primitives.

Some tests (Figure 3 in [26]) compare the performance of AzequiaMPI, TOMPI and MPICH in a single Linux PC. The results show that the thread-based implementations double the performance of MPICH, on other hand, the performance of AzequiaMPI is quite similar to the TOMPI project.

The results in [26] conclude that better threads support would contribute to simplify the AzequiaMPI solution, eliminating the need of an operating system, improving the performance and reducing its memory footprint.

2.1.2 Process-based Approach

The work discussed in [29] and [30] demonstrates that MPI is the correct programming model in order to build a hybrid cluster of embedded platforms hiding hardware complexities from the programmer and promoting code portability. In [29] the authors describe a lightweight subset MPI standard implementation called TMD-MPI to execute parallel C programs across multiple FPGAs. A simple NoC [4] was developed to enable communications within and across FPGAs, on top of which TMD-MPI can send and receive messages.

The experiments in [29] show that, for short messages, communications between multiple processors can perform better than the network of Pentium 3 machines using a 100Mb/s Ethernet network. TMD-MPI, does not depend on the existence of an operating system for the functions implemented. TMD-MPI is small enough that can work with internal RAM in an FPGA. Currently, the size of the library is 8.7 KB, which makes it suitable for embedded systems.

One of the most advanced process MPI implementations for embedded systems is the Lightweight Message Passing Interface (LMPI)[21]. The idea of LMPI is separation of its server part (LMPI server) and the very thin client part (LMPI client). Both parts can reside on different hardware, or on the same hardware. Multiple clients can be associated

[4]Network on a chip (NoC) is a communication subsystem on an integrate in a system in a chip

with a server. LMPI servers support full capability of MPI and can be implemented using pre-existing MPI implementations.

Although LMPI is dedicated to embedded systems, to demonstrate the benefits of LMPI and to show some initial results, in [21] it was built an LMPI server using MPICH on a non-embedded system. LMPI client consumes far less computation and communication bandwidth than typical implementations of MPI, like MPICH. As a result, LMPI client is suitable for embedded systems with limited computation power and memory. That work demonstrated the low overhead of LMPI clients on Linux workstations, which is as low as 10% of MPICH for two benchmark applications. LMPI clients are highly portable because they don't rely on the operating system support. All they require from the embedded system is networking support to the LMPI server.

These light and limited versions of MPI either process-based or thread-based are relevant proofs of the raising role of MPI in the embedded world. However, they have challanges, one of them is portability. The portability of code among different platforms needs to be considered. For example MPI programs that run in a traditional Linux cluster (eg. a Bewolf Cluster) should also be able to run in a cluster of embedded platforms by code recompilation. In [26] it is shown that the support of an operating system is essential for portability to success.

2.2 Multicore Communication Application Program Interface (MCAPI)

The increment in the complexity of the new embedded applications started to require more specialized integrated circuits that could help the microprocessor with all these task in parallel. As we know modern embedded platforms use multi-core microprocessors to solve this problem.

The lack of a framework to share the workload among the multi core of these new embedded applications was an initial problem. One of the first Multicore Communication systems is MCPI. According to [31] the purpose of MCAPI, which is a message-passing API, is *To capture the basic elements of communication and synchronization that are*

required for closely distributed (multiple cores on a chip and/or chips on a board) embedded systems

There are already several programming models and tools for multiprocessor System-on-Chip (MPSoC) programming [32] [33]. These tools help in partitioning and mapping the problem to the specific platform. However, these solutions are too heavy-weight and doesn't support a wide range of embedded platforms.

On the other side MCAPI provides a limited number of calls with sufficient communication functionality. MCAPI is scalable and can support virtually any number of cores, each with a different processing architecture and each running the same or a different operating system.

Some inter-processor communications protocols [32] require a full TCP/IP stack to exchange data, creating a bloated memory footprint. MCAPI, on the other hand, does not require this and is much more lightweight. At the end, an application using MCAPI will see an standard set of function calls to send and receive data to and from any core in the system.

MCAPI is not a protocol specification. This is an implementation issue, due to the fact that with multiple vendors of embedded platforms a standard is becoming a necessity. Another disadvantage is that MCAPI just send an receive data within the cores of the embedded platform not among multiple embedded platforms.

As we have seen for that purpose exist MPI (either process or thread base). There are key differences between the two technologies. MPI can be used to create programs that adapt to the resources available in a dynamic network of devices (embedded platforms, servers, computers). If one system goes down the workload is re distributed the next time. Besides, MPI focuses on the share of workload (wither process base or thread base) and exchanging messages between them. It doesn't matter if the devices are side-by-side or half a world apart.

MPI is not a good fit for communication within the cores of a server, where different cores might run and avoid bottlenecks in bus and memory access. This is where MCAPI comes in, Allowing a light communication among the cores.

2.3 Clusters of Embedded Systems

Despite of the type of MPI implementation either process-based or, thread-based, it is proved that when multiple embedded processors are available, the workload can be distributed in such away that each processor runs at a very low-power level; meanwhile, the performance is compensated by parallelism. Based on this idea some efforts have been done in [23] and [34]. Those propose the idea of partitioning the computation onto a multi-processor architecture that consumes significantly less power than a single processor.

In [34] the authors examine the use of multiple constrained processors running at lowered voltage and frequency to perform a similar amount of work in a shorter time and lower power than a uniprocessor. The workload studied in that work is a video encoder. As a result it is proved that four processors sharing the workload at a lower frequency can save up to as 56% of energy compared to a uniprocessor running the workload. According to [34] a future work will involve developing an analytical model that estimates the minimum amount of parallelism that is needed for a multiprocessor to save energy.

Another example of a distributed embedded application is presented in [23]. In that work the author implemented an image processing algorithm with MPI on a low voltage computer. As a result the author shows that when multiple low-voltage processors are available, the workload can be distributed in such a way that each processor runs at a very low-power level; meanwhile the performance is compensated by parallelism. The author also confirms that distributed programming tools are available to leverage the design and exploration of distributed embedded applications. However he emphasizes that the communication mechanism of MPI needs further study.

As can be seen there is a need to make new experiments with the latest technology available. Nowadays, we have ultra-low-voltage microprocessor platforms, customized operating systems for these platforms and innovative technologies for achieving MPI implementations. Few works, like those described in [35] and [36] have address the problem of parallel and distributing computing with embedded platforms. These design parameters are considered for the development of this work.

CHAPTER 3

Theoretical Framework

This chapter will describe some basic topics to help understand the experiments described in chapters ahead and their justification.

3.1 The need of parallel computing

Based on current technologies we have today (for example smartphones, tablets, smart cars and more) the computer industry has achieved tremendous gain. Computer technology has been evolving like any other technology. However, the progress of computer architectures has been much less consistent. During the first 25 years of electronic computers [1] the improvement in performance increased about 25% per year [4].

It was at the beginning of the 1970s when the world saw the emergence of the microprocessor. Its appearance caused a major change in technology, which allowed industry to improve scalability of integrated circuits. After the introduction of the microprocessor, the improvement in performance per year on computer architectures reached a 35% [4].

However, advances in computer architecture were not the only responsible for this great increase in performance. In particular, two significant changes made the life of users easier. First, the invention of the C programing language, which caused a reduction in the development of assembly language programs, made programs easier to read and to use. The C programing language gave the user the power to handle memory and peripheral devices in a much more friendlier way. Second, the creation of standardized and free

[1] Since 1951 with the introduction of UNIVAC

operating systems, such as UNIX and Linux. These operating systems lowered the cost and risk of bringing out to the market new products.

In the decade of 1980 the idea of making the microprocessor architecture faster started to take form with the RISC (Reduced Instruction Set Computer) architecture [4]. The RISC microprocessor is designed to perform a smaller number of types of computer instructions so that it can operate at a higher speed. The RISC-based computers raised the performance bar. This architecture principle in conjunction with the transistor size reduction allowed to have much more computing power in less space it led to 16 years of sustained growth in performance at an annual rate of over 50%.

It was around the years 2003 to 2005 that a dramatic change seized the semiconductor industry and the manufactures of processors. The increasing rate of computing performance in processors, based simply on screwing up the clock frequency, was not longer sustainable. The problem with increasing the frequency in the microprocessor was that the heat in the chip also increased. In fact, in 2004 Intel® canceled its high-performance uniprocessor projects declaring that *"the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors"* [4].

The answer of the industry to meet Moore's law [2], was the shifting to real parallelism by doubling the number of processors per chip die. This was the birth of the multicore era.

With the multicore it emerged the need to change the paradigm in programing languages. The programs that had been designed before this change were mostly a sequence of instructions to calculate or control a system. The multicore architecture brought the widespread of parallel programing. Parallel programing is designed to provide faster execution of programs by executing independent code section in parallel.

However, not all the problems can be solved using parallel programing techniques. In order to use this approach the problem need to be represented as a collection of simultaneously executing tasks. This is especially the case in many areas of scientific, mathematical, and artificial intelligence programming. After the birth of parallel computing, these technology areas have evolved rapidly.

[2]The number of transistors in a dense integrated circuit doubles approximately every two years[6].

At the same time that parallel emerged, many other technologies were already established: the emergence of the Internet, the World Wide Web, the cellphones, and the laptops. According to [4], these technologies have led to three different computing markets: desktop computing, servers and embedded.

The problem studied in this work requires understanding of server and embedded technologies.

3.2 Servers Systems

The growth in the number of mobile personal computers coupled with the popularity of cellphones. Such growth changed the role of servers to provide scalable and reliable storage and computing services. The emergence of faster Internet connections accelerated the demand of web-based services which caused the transition of computing power from personal computers to servers.

However, the fact to provide storage and computing services to thousands of users simultaneously implied a large responsibility. A failure on a server system is by far more catastrophic than a failure of a single desktop computer because servers must operate seven days per week, 24 hours per day. For this reason, reliability is a key design parameter on server systems.

A second key feature on server systems is scalability. With the number of users changing every minute the ability to scale up the computing capacity in server is critical. A web sale site for sales must be able to respond every request, as well as during peak hours, for example during Christmas shoppings. A five minute failure, might represent multimillion loses.

The third key design feature is throughput. Servers are designed for efficient throughput [3] in terms of transactions per minute or Web pages served per second. From the user perspective point of view this is the speed of response.

Due to the previously described design parameters the server technology has evolved. The cloud era is dominating the computing and storage services. According to [38] *"Cloud*

[3]Throughput is a measure of how many units of information a system can process in a given amount of time [37]

computing is a set of resources and services offered through the Internet". Cloud services are delivered from data centers located around the world. Cloud computing provides virtual resources via the Internet. The best example of cloud computing is the streaming video of video services. Nowadays, users can stream online videos at any time, without the need to storage the movies at home. All resources and infrastructure are provided upon request. Scalability, reliability and throughput are guaranteed by computing service providers.

3.3 Embedded Systems

The birth of multicore architecture not only provides the servers with much more computing power, but also breaks the paradigm of using low computing power microprocessors for embedded platforms. Today it is possible to have more computing power with less frequency. This change has allowed a rapid evolution of computing and multimedia capabilities for embedded systems. The computers technology has evolved in such a way that today there is more computing power in a cellphone than all of NASA back in 1969 [3].

According to [7] *"An embedded system is a special-purpose system in which the computer is completely encapsulated by the device it controls"* Unlike a general-purpose computer, such as a personal computer, an embedded system performs pre-defined tasks, usually with very specific requirements. Examples of these are:microwaves, washing machines, printers, and GPS (Global Positioning System) systems. These electronic gadgets started to emerge more than 35 years ago [5].

The variety of embedded applications includes a wide spread of processing power and cost. They include 8-bit and 16-bit processors that may cost a few cents, 32-bit microprocessors that execute 100 million instructions per second and cost less than a few dollars, and high-end processors for the newest video games or network switches that cost at least 100 dollars and can execute one billion of instructions per second [4].

Since its origins, the RISC technology has been the default technology in modern embedded architectures. Due to the fact that RISC microprocessors are designed to execute

a smaller number of types of instructions, the power consumption is smaller.

The increment in the complexity of new embedded applications started to require more specialized integrated circuits that could support the microprocessor to process tasks in parallel . Wireless networking cards, Digital Signal Processors, I/O controls, peripherals (such as USB controllers) and analog interfaces (including ADCs and DACs) became part of the requirements of an embedded platform. Architecture designers realized that communication with these components decreased the performance and increased power consumption. For that reason, during the last decade ti has been seen the emergence of System on a Chip (SOC) embedded platforms.

The SoC is an integrated circuit with different processing components into a single chip. Putting these components in the same integrated circuit the communication latency and power consumption was reduced considerably. Since the beginning of SoC architectures the variety of gadgets using embedded platforms has increased. Every year some new goals are pursued: for example, increment in computing performance, cost, size and power density reduction.

3.4 Embedded Linux Systems

Computers-based devices are everywhere. In office desktops, in our kitchens, in living rooms, in microwave ovens, in regular ovens, in cellphones, in portable digital music players, etc.

A few years ago embedded systems were not very powerful, they executed special-purpose, proprietary operating systems that were very different from industry-standard ones. Today, embedded computers have more computing capacity.

Along with larger computing capacity it comes the capability to run a full-fledged operating system, such as Linux. Using Linux for an embedded product makes a lot of sense. The reasons are that it is a free operating system and it has an easy user experience, such that the number of Linux users has increased rapidly. Linux has been one of the most sustainable projects in the history of computing. The fact that a large community of developers finds novel ways to improve performance and fix critical failures every day

are the key to think that Linux could be the best solution in terms of sustainability for embedded platforms.

According to [7] there are multiple reasons why Linux is the best choice for modern embedded platforms.

1. Linux supports a huge variety of applications and networking protocols. Linux is scalable, from small consumer-oriented devices to large, heavy-iron, carrier-class switches and routers.

2. Linux can be deployed without royalties required by traditional proprietary embedded operating systems.

3. Linux has attracted a huge number of active developers, enabling rapid support of new hardware architectures, platforms, and devices.

4. An increasing number of hardware and software vendors, including virtually all of the top-tier manufacturers and ISVs.

Due to these and other reasons, it is seen an accelerated adoption rate of Linux in many common embedded platforms. With the birth of SoC systems the use of complete operating systems was needed due to the requirement of handling processes concurrency, memory management, and network connectivity.

The Linux operating system was chosen for the studied embedded platform. It has many configure options, there are no standard methodologies or templates to reuse the process to customize a Linux embedded environment which is a complex job for software engineers. Every new embedded computing company creates its own version according to its needs, without any standards, with very low maintainability and robustness. In 2010 there was a change in the industry of embedded systems, it was announced of a project to leverage environment configuration: The Yocto project.

The Yocto Project is an open source collaboration that provides templates, tools and methods to create customized Linux systems for embedded products regardless of the hardware architecture[39]. It started in 2010 as a collaboration between hardware manufac-

turers, open-source operating systems vendors, and electronics companies to standardize embedded Linux development [40].

The Yocto project is a complete embedded Linux development environment with tools, meta-data, and documentation. The free tools (eg. emulation environments, debuggers, application toolkit generators) are easy to use, powerful and allow projects development with optimization at the project prototype phase. The Yocto Project allows its users to focus on specific product features and development.

The Yocto Project software development environment in Figure 3.2) shows targeting the ARM, MIPS, Power-PC and x86 architectures for a variety of platforms, including x86-64 and emulated ones. It has a set of components to design, develop, build, debug, simulate, and test the complete software stack using Linux. The X window system, GNOME mobile-based application frameworks, and Qt frameworks are examples of pieces of software that can be developed with Yocto.

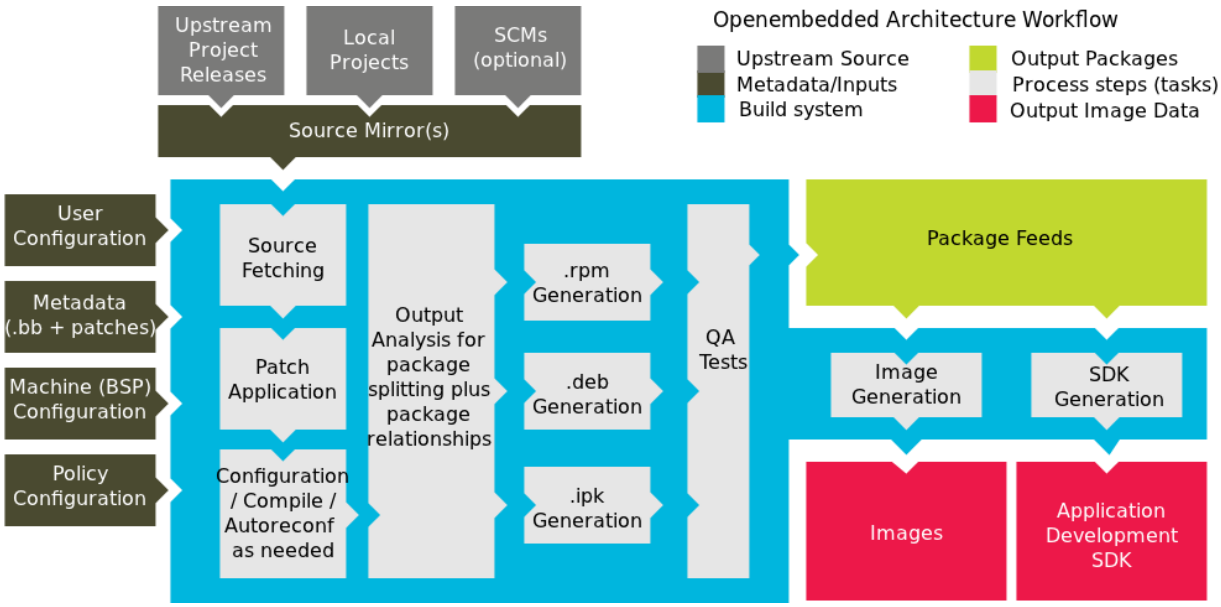


Figure 3.1: The Yocto project development environment

Yocto project plays an important role in the IoT. To develop a standard solution for multiple platforms Yocto is the best choice. Such solution enables deployement to multiple IoT devices.

3.5 Ubiquitous Computing and IoT

The optimization of design parameters like cost, size, and power density in combination with the increase in performance and connectivity has made computing technology to evolve into ubiquitous computing. According to Mark [8], ubiquitous computing is defined as *"the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user"*. This means that the computing power is available anywhere and at any time.

According to [5], nowadays ubiquitous computing is evolving into advanced ubiquitous computing. An advanced ubiquitous computing is an extension of an ubiquitous environment that improves connectivity between devices. The main characteristics of this environment can be stated as follows:

- A large number of heterogeneous devices.
- Availability of new communication technology.

Ubiquitous computing includes devices such as notebook computers, tablets, smartphones and wearable computers. Most of these devices operate under many different operating systems. New communication technology 4G, 5G and the introduction of IPv6 provides larger and faster data bandwidth than 3G.

One of the most accurate definitions of IoT is the one given by [9] where it mentions that "Internet of Things refers to physical and virtual objects that have unique identities and are connected to the internet to facilitate intelligent applications.". IoT enables the interconnection, via the Internet, of computing devices embedded in everyday use objects, enabling them to send and receive data. The main difference of IoT systems with traditional embedded systems is the Internet connectivity and less power consumption. IoT systems must always be connected to the internet which require a lower power consumption.

IoT computing is a new era of computing technology that explores in collaboration with cloud computing the capability to have smart applications in multiple scenarios. In the core of these technologies an invisible architecture design was established, transparent to

the user, but always there sustaining the availability, scalability and reliability of systems, it was the distributed systems architecture.

3.6 Distributed Systems

We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages [41]. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building, or in the same room. The definition of distributed systems (given by [41]) has the following significant characteristics:

1. **Concurrency:** In a network of computers, programs run concurrently. Programs share resources, such as web pages, or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers or memory) to the network.

2. **No global clock:** When programs need to cooperate, they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks there is no single global notion of the correct time. This is a direct consequence of the fact that the only communication is by sending messages through a network.

3. **Independent failures:** All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a crash), is

not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running.

Each one of these characteristics is also present in a modern IoT system. Figure 3.2 represents an IoT system showing its distributed systems characteristics.

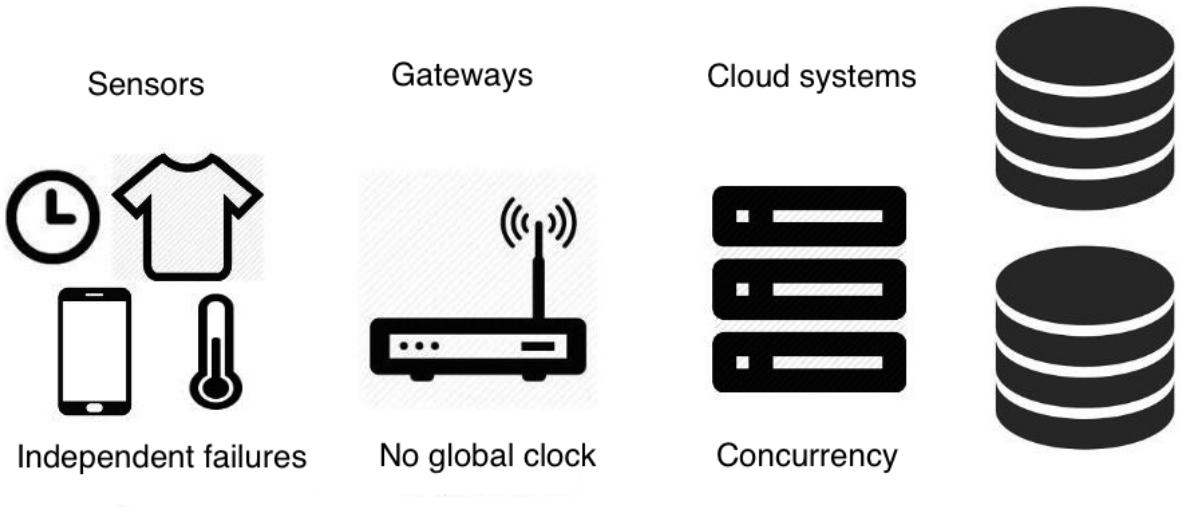


Figure 3.2: IoT system as a distributed system

1. **Concurrency:** In an IoT system there are multiple systems trying to use the same resource. For example, all IoT devices are trying to access the same data base or the same access point. All of them are competing for similar resources, a good IoT design needs to schedule the use of limited resources in an efficient way.

2. **No global clock:** None of the systems in an IoT network (either sensors or processing devices) have the same clock. They have to use a message-based mechanism to communicate with each other.

3. **Independent failures** In a regular IoT network multiple systems could fail (either sensors, processing devices, or one of the data centers). Despite of the failures others components should keep working without any problem.

   All these characteristics enforce the idea that the nature of an IoT system is being treated as a distributed system. With this idea it is simpler to adapt solutions of distributed systems into IoT distributed systems.

3.7 Message-Passing Interface Library

The Message-Passing Interface (MPI) is a message-passing library interface specification[42]. MPI addresses primarily the message passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

MPI is not a language, all its operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and FORTRAN, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers.

Despite of all the advantages that MPI has, it is not widely used in embedded systems due to the fact that abstraction layers require extensive system resources with comprehensive operating systems support, which may not be available to an embedded platform. This work uses a powerful ultra-low-voltage microprocessor platform [43]. It was a challenge to set up MPI for such system.

Recent research work [20] [26] [22] describes proofs-of-concepts MPI implementations for embedded systems. It has been an increase of interest in the topic. However, none of those has been implemented in current embedded Linux operating systems (like Yocto [39]) or Fedora[44]).

Review of current operating systems shows that the only one missing MPI is Yocto.

Operating System	**MPI library version 3.1**
Fedora	Yes
Clear Linux for Intel Architecture	Yes
OS generated with Yocto project	No

Table 3.1: MinnowBoard MAX Linux distributions with MPI

3.8 Performance and Power Efficiency

According to [4] the meaning of performance may be different according to the application. The user of a desktop computer may says that a computer is faster when a program

runs in less time; a Website administrator may says that a computer is faster when it completes more transactions per hour. The computer user is interested in reducing response time (the time between the start and the completion of an event) [4], also referred as execution time. The administrator of a large data processing center may be interested in increasing throughput (the total amount of work done in a given time) [4].

According to [4], *To compare design alternatives, the performance of two different computers, say, X and Y is related.* The phrase "X is faster than Y" is used to refer that the response, or execution time is lower on X than on Y for the given task. In particular, "X is n times faster than " is represented by equation 3.1 from [4].

$$n = \frac{Executiontimex}{Executiontimey} \tag{3.1}$$

Since execution time is the reciprocal of performance, the following relationship holds in formula 3.2 from [4].

$$n = \frac{Executiontimex}{Executiontimey} = \frac{\frac{1}{Performancex}}{\frac{1}{Performancey}} = \frac{Performnacex}{Performancey} \tag{3.2}$$

Another metric to consider is throughput. According to [4] *"the throughput of X is 1.3 times higher than Y signifies that the number of tasks completed per unit time on computer X is 1.3 times the number completed on Y"* . Unfortunately, time is not always the metric quoted in comparing the performance of computers. The only consistent and reliable measure of performance is the execution time of real programs.

The most straightforward definition of execution time is given by [4] *"it is called wall-clock time, response time, or elapsed time, which is the latency to complete a task, including disk accesses, memory accesses, input/output activities and operating system overhead everything"*. The response time considered by the user is the elapsed time of the program, not the CPU time.

In parallel programing the performance metric is measure in a different way. With current computing power of devices it is possible to create a cluster of computers. For example, a set of interconnected embedded systems in a network that provides a large amount of performance with the smaller power consumption than a higher performance

computing system. This characteristic is determined by the power efficiency of the network. The power efficiency is quantified by performance per watt [17].

The goal is to determine what is the optimal point at which it is better to send data to a server instead of doing local processing. It is important to answer the question : What is the maximum number of systems that such type of cluster can support to keep savings energy efficiency?

The development of metrics to evaluate energy efficiency on the basis of performance and power models is described in [18]. According to [18], the formula for computing the theoretical maximum speedup (or performance) achievable through parallelization is given by the following equation 3.3:

$$Perf = \frac{1}{(1-f) + \frac{f}{n}} \tag{3.3}$$

Where n is the number of processors, f is the fraction of computation that programmers can parallelize (from 0 to 1). To model the power consumption of a P many-core processor, [18] introduces a new variable, k, to represent the fraction of power the processor consumes in idle. It is expressed by equation 3.4 bellow:

$$\frac{Perf}{W} = \frac{1}{(1 + (n-1)k(1-f))} \tag{3.4}$$

In [18], it is demonstrated that a symmetric many-core processor can easily lose its energy efficiency as the number of cores increases. To achieve the best possible energy efficiency, their work suggests a many-core alternative, featuring many small, energy-efficient cores integrated with a full-blown processor. It shows that having knowledge on the amount of parallelism available in an application before its execution, it is possible to find the optimal number of active cores for maximizing performance for a given cooling capacity and energy in a system.

3.9 Benchmarks

In computer science a benchmark is a set of programs utilized to measure software or hardware or performance of applications using a standard methodology. The best choice of benchmarks to measure performance is to select end user applications. Due to the complexity of current applications software engineers are using small versions of them as benchmarks. According to [4], there are three kind of them:

- Kernels, which are small, key pieces of real applications.
- Toy programs, which are 100-line programs from simple programming assignments.
- Synthetic benchmarks, which are fake programs invented to try to match the profile and behavior of real applications.

One of the most successful attempts to create standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), which had its roots in the late 1980. SPEC's intention is to deliver better benchmarks for workstations. [4]

In terms of parallel and distributed computing, there are numerous MPI benchmark suites available, such as Mpptest [25], MP-Bench [45], and SKaMPI [46]. Many of them provide timing results for message passing routines. This is useful for performance modelling and analysis of parallel programs, as well as for understanding the performance of parallel machines. Based on results of [47] it was decided to use MPIbench as the default benchmark testing framework. MPIbench is a benchmark suite that allows performance assessment of MPI on clusters of workstations. MPIbench tests MPI calls.

3.9.1 MPIbench

The benchmark suite utilized on this work is MPIbench (or MPbench) version 4 [48]. It is a set of programs to measure the performance of dominant MPI functions. The

[4] All the SPEC benchmark suites and their results are found at www.spec.org.

runtime behavior of these functions dominates the total execution time of a distributed application.

MPIBench tests eight MPI calls. The following functions are measured:

- Bandwidth (BB/second)
- Gap Time (time to launch a message and continue) (μs)
- Roundtrip or 2 * Latency (transactions/second)
- Asynchronous bidirectional bandwidth (KB/second)
- Broadcast (KB/second)
- Sum reduction (KB/second)
- All-reduce (KB/second)
- AlltoAll (KB/second)

Each of these tests performs the following steps:

- Set up the test.
- Start the timer.
- Loop of MPI operations.
- Verify that those operations have completed.
- Stop the timer.

By default, MPIBench programs measures transmission and reception of messages of different sizes, from 4 to 216 bytes. It repeats such process for 100 iterations. As part of the set up , each test is run a single time, before testing, to allow caching and routing set up. The cache is flushed before each repetition and before each new message size is tested. The cache is not flushed however between iterations of the same message size, which are averaged.

Next, each one of the tests is briefly described in order to understand the experiments. A detailed description with illustrations and examples is presented in [49] and [50].

3.9.1.1. Bandwidth

MPIBench measures bandwidth with a doubly nested loop. The outer loop varies the message size, and the inner loop measures the *send* operation over the iteration count. After the iteration count is reached, the slave process acknowledges the data it has received by sending a four-byte message back to the master. The master's pseudo code for this test is as follows:

```
1 do over all message sizes
2     start timer
3     do over iteration count
4         send(message size)
5         recv (4)
6     stop timer
```

The slaves' pseudo code is as follows:

```
1 do over all message sizes
2     start timer
3     do over iteration count
4         recv(message size)
5         send(4)
6     stop timer
```

3.9.1.2. Bidirectional Bandwidth

MPIBench measures bidirectional bandwidth with a doubly nested loop [5] The outer loop varies the message size, the inner loop measures the send operation over the iteration count. Both processes execute a non-blocking receive, then a non-blocking send, and then a wait for each iteration. The next iteration is prevented from proceeding until the previous one is finished by the `MPLWaitall()` call, which will not allow execution to continue until both messages have been completed.

The code for this test is as follows:

[5] The bidirectional bandwidth test is similar to the bandwidth test, except that both the nodes involved send out a fixed number of back-to-back messages and wait for the reply. This test measures the maximum sustainable aggregate bandwidth by two nodes [48].

```
1  do over all message sizes
2     start timer
3     do over iteration count
4         immediate (nonblocking) receive(message size)
5         immediate (nonblocking) send(message size)
6         wait until messages on both ends
7             have been received (MPLWaitall())
8     stop timer
```

3.9.1.3. Roundtrip

Roundtrip times are measured in the same way as in the bandwidth test; except that, the slave process, after receiving the message, echoes back to the master. This benchmark is often referred to as pingpong. The unit of this metric is transactions per second, which is a common metric for database and server applications. No acknowledgment is needed with this test as it is implicit given its semantics.

The master's pseudo code for this test is as follows:

```
1   do over all message sizes
2     start timer
3     do over iteration count
4         send(message size)
5         recv(message size)
6         stop timer
```

The slaves' pseudo code is as follows:

```
1  do over all message sizes
2     start timer
3     do over iteration count
4         recv(message size)
5         send(message size)
6     stop timer
```

3.9.1.4. Application Latency

Application latency is something relatively unique to MPBench. This benchmark can properly be described as one that measures the time for an application to issue a send and continue computing. This benchmark is the same as bandwidth except that it does not acknowledge the data and report results in units of time.

The master's pseudo code for this test is as follows:

```
1 do over all message sizes
2     start timer
3     do over iteration count
4         send(message size)
5     stop timer
```

The slaves' pseudo code is as follows:

```
1    do over all message sizes
2         start timer
3         do over iteration count
4             recv(message size)
5         stop timer
```

3.9.1.5. Broadcast and Reduce

Both of these benchmarks return the number of megabytes per second computed from the iteration count and the length argument given to function call.

Here is the pseudo code for both the master and the slave:

```
1    do over all message sizes
2         start timer
3         do over iteration count
4             reduce or broadcast(message size)
5         stop timer
```

3.9.1.6. AllReduce

AllReduce is a derivative of an all-to-all communication, where every process has data for every other. While this operation could easily be implemented with a reduce followed by a broadcast, that would be highly inefficient for large message sizes.

Here is the pseudo code for both the master and the slave:

```
1    do over all message sizes
2        start timer
3        do over iteration count
4            allreduce(message size)
5        stop timer
```

3.9.1.7. All-to-all

MPBench measures a kind of round-robin communication among multiple processes. The outer loop varies the message size, and the inner loop measures the send operation over the iteration count. Each process sends a message of the size of the total message size divided by the number of processes to every other process.

The code for this test is as follows:

```
1    do over all message sizes
2        start timer
3        do over iteration count
4            all-to-all(message size)
5        stop timer
```

CHAPTER 4

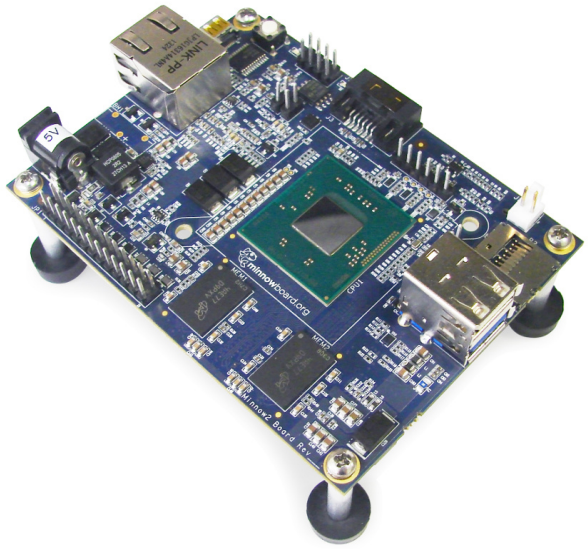
System Architecture

In this chapter it is described the architecture of the embedded system employed in this work. It is provided a hardware and software description. It also includes an explanation of the MPI implementation utilized use. At the end, it is presented the description of the implemented topology of the system.

4.1 Selected Embedded System

4.1.1 Development Board

The development board utilized to perform experiments is a MinnowBoard MAX [43] and it has a processor Intel Atom-TM Processor E3825 ® [51]. The main characteristics are described in Table 4.1:



.png

Figure 4.1: Minnow board max platform

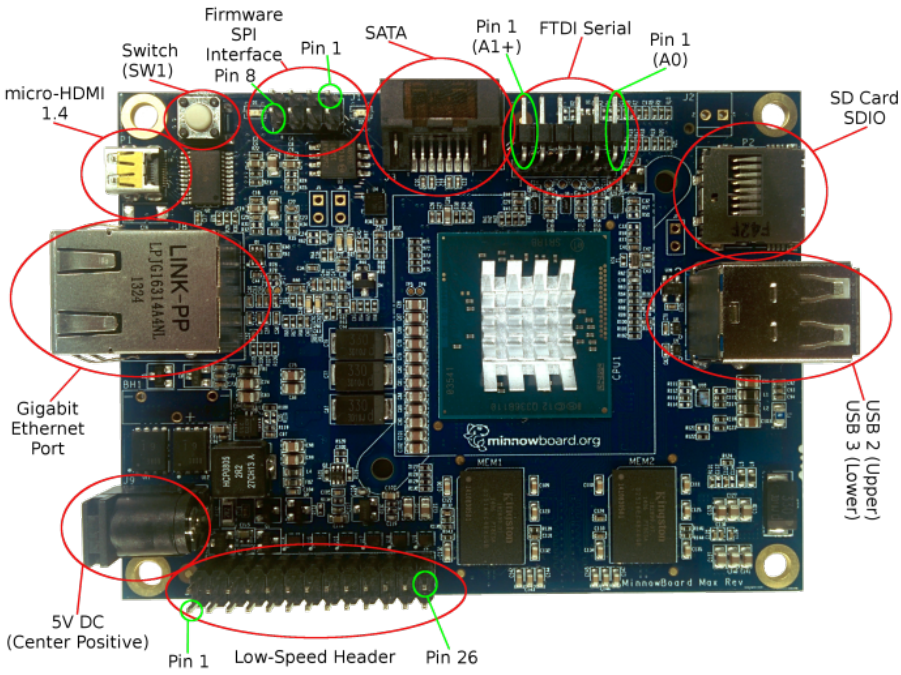


Figure 4.2: Connection ports of the MinnowBoard MAX

Processor Number	E3825
# of cores	2
# of Thread	2
Clock Speed	1.3 GHz
L2 Cache	1MB
Instruction Set	64 bit

Table 4.1: Intel® Atom Processor E3825 Specifications

Figure 4.1 and 4.2 shows pictures of the MinnowBoard MAX . It is a development platform for both professionals and entrepreneurs. The MinnowBoard MAX is an open hardware platform: anyone can have access to see the source design and documentation, can study it, modify it and share it.

4.1.2 Embedded Operating System

Typical embedded system does not contain an operating system (usually); but due to the complexity of current embedded applications is plenty necessary to have an operating system that manage all the processes that current embedded applications have.

To perform the experiments on the selected board, it was decided to include a full operating system running. It includes a boot manager, a full file system , kernel and user spaces. According to [43] the board supports multiple kind of operating systems. In this work three different, Linux-based, operating systems:

- Fedora [44]
- Clear Linux for Intel Architecture [52]
- Yocto [39]

The work performs a comparison between the three. There are multiple configurations for each one, such that the performance of implemented embedded cluster behaves differently.

4.1.3 Embedded Firmware

The MinnowBoard MAX board is a low-cost, commercially available, reference platform for hardware, software and firmware developers who wish to work with an open design development environment. Its design specifications and materials have been provided to the open community for the purpose of enabling experimentation and development of technical solutions based upon the MinnowBoard MAX board.

The download page presents firmware components for the reference board. For this work were downloaded the official BIOS Firmware Binary Images [53].

4.2 Selected Traditional Computing System

The selected traditional computing system we chose is the Intel NUC D54250WYK platform. As we can see in [54] and [55] this kind of platforms are used as media centers and cloud system for heterogeneous sensors. This part will describe hardware of the development board as well as the software we are going to use (operating system and firmware).

4.2.1 Development Board

The platform we will use for our experiment is the Intel® NUC D54250WYK Their main characteristics are described in Table 4.2 [56].

Processor Number	Intel CoreâĎć i5-4250U Processor
# of cores	2
# of Thread	4
Clock Speed	1.3 GHz
L2 Cache	3MB
Instruction Set	64 bit

Table 4.2: Intel® NUC D54250WYK Specifications

4.2.2 Operating System

The board support multiple kind of Operating Systems. In [57] there is a list of available operating systems to install. However, due to the good performance results [58] we are going to use the Clear Linux project for Intel architecture [52]. The Clear Linux Project for Intel® Architecture is a project that is building a Linux OS distribution for various cloud use cases. The goal of Clear Linux OS is to showcase the best of Intel Architecture technology, from low-level kernel features to more complex items that span across the entire operating system stack.

4.3 Studied Network Topologies

In computer networking, topology refers to the layout of connected devices. This section describes standard topologies of networking.

According to [59] *"In networking a topology is a virtual shape or structure. This shape does not necessarily correspond to the actual physical layout of the devices on the network"*. For example, the computers on a home network may be arranged in a circle in a family room, but it would be highly unlikely to find a ring topology there.

According to [59] network topologies are categorized into the following basic types:

- Bus
- Ring
- Star
- Tree
- Mesh

More complex networks can be built as hybrids of two or more of the above basic topologies.

For the experiments of this work it is utilized the star topology (see Figure 4.3). According to [60] *"In a star topology , each device on the network connects to a centralized*

device via a single cable. This arrangement creates a point to point network connection between the two devices and overall gives the appearance of a star".

The network does not necessarily have to have the appearance of a star to be classified as a star network, but all of the nodes on the network must be connected to one central device. Because each device must have its own cable, a star topology requires far more able than other topologies.

The star topology is considered the easiest topology to design and implement. One of the biggest advantages of the star topology is that computers can be connected to and disconnected without affecting the other systems. However, in the star topology, all devices on the network connect to central device, and this central device creates a single point of failure on the network [60].

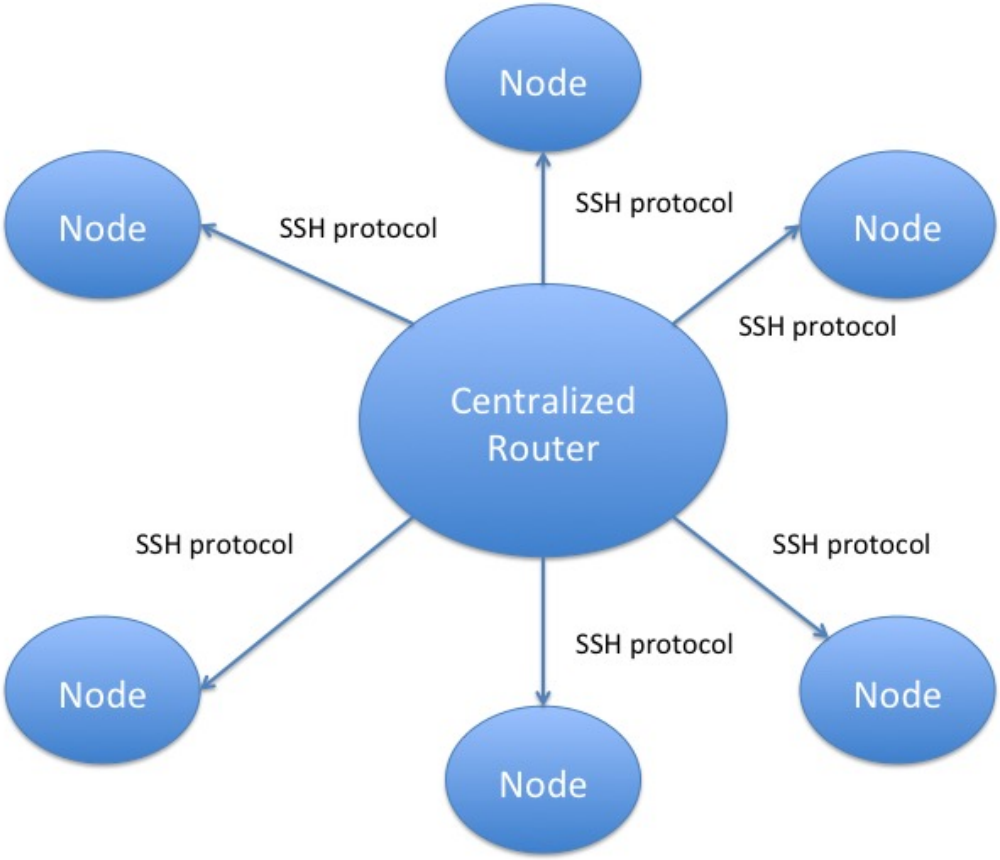


Figure 4.3: Star topology diagram

4.4 Architecture Diagram

Figure 4.4 shows the diagram of the architecture of the embedded cluster employed in this research work in an star topology.

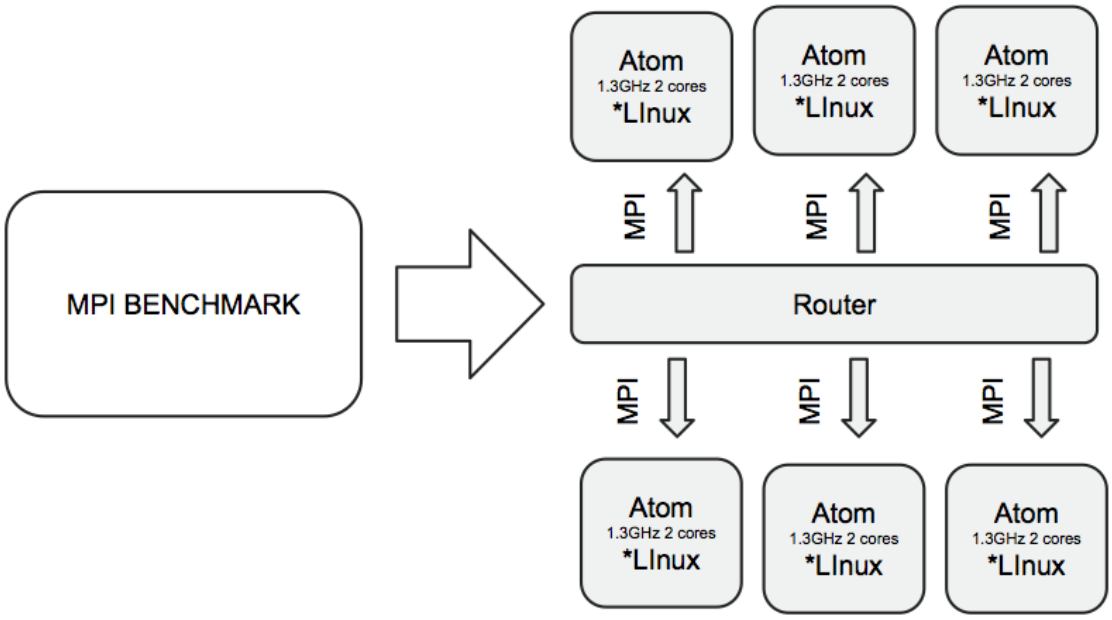


Figure 4.4: System architecture diagram

CHAPTER 5

Experiments and Results

This section describes the experiments and results while searching to conditions for the embedded distributed system. Each experiment is described in the same order it was tested and explained the reasons why that it was run. The steps that are involved are:

- To run a simple sanity test in the embedded platform.
- To run the MPI Bench test suite in the embedded platform.
- To evaluate operating systems for the embedded platform.
- To evaluate MPI library.
- To test MPI in a cluster of embedded systems.
- To compare the performance between a cluster of embedded systems to a traditional computing system, using MPI bench.
- To compare the power consumption of a cluster of embedded systems to a traditional computing system, using MPI bench.

5.1 MPI Sanity Testing in an embedded platform

The system under test (Hardware and Software) for this experiment is described in the Table 5.1

Platform under test	MinnowBoard MAX
Number of platforms	1
Operating System	Fedora

Table 5.1: Description of system under test (HW/SW) for MPI sanity testing experiment

To find the environment that gives better performance of MPIbench tests, first it is necessary to run the basic and simple MPI test. A simple code to prove that the MPI implementation is working as expected is shown in Figure 5.1

```
1  /*************************************************************************
2  * AUTHOR: Blaise Barney
3  * https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_hello.c
4  *************************************************************************/
5  #include "mpi.h"
6  #include <stdio.h>
7  #include <stdlib.h>
8  #define  MASTER     0
9
10 int main (int argc, char *argv[]){
11     int   numtasks, taskid, len;
12     char hostname[MPI_MAX_PROCESSOR_NAME];
13
14     // Initialize the MPI environment
15     MPI_Init(&argc, &argv);
16     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
17
18     // Get the rank of the process
19     MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
20
21     // Get the name of the processor
22     MPI_Get_processor_name(hostname, &len);
23
24     // Print off a hello world message
25     printf ("Hello␣from␣task␣%␣d␣on␣%␣s!\n", taskid, hostname);
26     if (taskid == MASTER)
27            printf("MASTER:␣Number␣of␣MPI␣tasks␣is:␣%␣d\n",numtasks);
28
29     // Finalize the MPI environment.
30     MPI_Finalize();
31 }
```

Figure 5.1: MPI code in C to print taskid, hostname and number of MPI tasks

The way to compile it is:

```
$ mpicc -o mpi_hello_world mpi_hello_world.c
```

The way to run it is:

```
$ mpirun -n 4 -f host_file ./mpi_hello_world
```

The program is compiled, it is ready to be executed. Running MPI programs on a cluster of nodes requires to setup a host file

The host file contains names of all of the compute nodes on which the MPI job will execute. For ease of execution, all of the nodes must have SSH access, and should have an *authorized keys* file to avoid password prompts for SSH. A simple host file looks like the one shown in Figure 5.2.

```
$ cat hostfile
  node1
  node2
  node3
```

Figure 5.2: MPI hostfile example

Each of the nodes is described in a ssh file as the one shown in Figure 5.3.

```
$ cat ~/.ssh/config
  Host node1
      HostName node1-ip-or-hostname
      User user-of-the-ssh-key
      Port port-if-nedded
```

Figure 5.3: ssh configuration file example

To run on a single system is not necessary to have a host file; however, for the experiments of this work it was needed more than one system.

The result of the basic MPI test is shown in Figure 5.4

```
victor@minnow-1 tmp $ mpirun -n 4  ./hello
Hello from task 0 on minnow-1
MASTER: Number of MPI tasks is: 4
Hello from task 1 on minnow-1
Hello from task 2 on minnow-1
Hello from task 3 on minnow-1
```

Figure 5.4: Results of MPI basic code in Figure 5.1

5.1.1 Results

After this experiment completed in the embedded platform with a regular GNU/Linux OS (Fedora), It was proved that it is possible to run MPI in an embedded system. However, there are different GNU/Linux Operating Systems (for example, Suse/Debian/RedHat). Next is needed to evaluate whether exists a Linux based operating system for embedded systems which would provide the best power efficiency.

5.2 Evaluation of MPI benchmarks testing in an embedded platform

The system under test (Hardware and Software) for this experiment is described in the Table 5.2

Platform under test	MinnowBoard MAX
Number of platforms	1
Operating System	Fedora

Table 5.2: Description of system under test (HW/SW) for MPI benchmarks experiment

The first approach is to find if it is possible to run a sanity MPI test (Figure 5.1) with a regular GNU Linux operating system on the embedded system (MinnowBoard MAX [43]), From the list of GNU/Linux based operating systems that support the embedded system [43] the one we selected was Fedora [44] project due to the fact that it has support for MPI libraries.

While MPIBench can be run from the command line, it is designed to be run from a makefile. Running it with a makefile automates the collection and presentation process. The steps to run the benchmark are shown bellow:

```
victor@minnow-1 $ llcbench $ make linux-mpich
victor@minnow-1 $ llcbench $ make mp-bench
victor@minnow-1 $ llcbench $ make mp-run
```

After last command the benchmark starts to run under the MinnowBoard MAX, the results are gathered in a tarball file.

5.2.1 Results

The experiment was successful, there fore it is possible to run the MPI benchmarks on the embedded platform with a regular Linux operating system.

5.3 Evaluating MPI Benchmark on Multiple Operating Systems

The system under test (Hardware and Software) for this experiment is described in the Table 5.3

Platform under test	Minnow Board Max
Number of platforms	1
Operating System	Fedora and Clear Linux

Table 5.3: Description of system under test (HW/SW) for MPI benchmark evaluation on multiple operating systems

The Clear Linux Project for Intel Architecture [52] is a distribution built for various cloud computing use cases. The project want to showcase the best of Intel Architecture technology, from low-level kernel features to complex applications that span across the entire OS stack.

5.3.1 Results

There were found interesting results that indicate that a customized OS could improve the performance numbers of MPI benchmarks. Results are presented in the next figures.

In Figure 5.5, the *all reduce* benchmark shows more stable performance running in a customized OS. This can be seen in the drop of speed (KB/sec) close to the test with a message size of 32 KB.

In Figure 5.6, the *all to all* benchmark shows the same performance running in a customized OS than in the Fedora OS.

The same results as presented in Figure 5.5 can bee seen in the *bandwidth* benchmark (Figure 5.7); however, in the *all to all* (Figure 5.6) there are no drops in speed at any size of the message under test. This is mainly because of the nature of the test (Described in Chapter 3). The *all reduce* and the *bandwidth* tests execute similar tasks. Meanwhile, *bandwidth* executes send and receives.

On the other hand MPI *all reduce* will reduce the values and distribute the results to all processes. MPI *all reduce* operates in each process and then broadcasts the result of the operation to all the individuals of the distributed system.

The Clear Linux OS has specific implementations in the kernel that improve the network performance, specially in the kernel network area.

On the other hand, the *all to all* sends data from all the nodes, this means that each node of the system needs to send and receive data to every other node. This increases the latency of the network.

The same results can be seen in the *bi directional bandwidth* (Figure 5.8) and the *broadcast* (Figure 5.9) test.

The MPBench measures bidirectional bandwidth with a nested loop. The outer loop varies the message size, and the inner loop measures the send operation over the iteration count. Both processes execute a non-blocking receive, then a non-blocking send. A non blocking send and receive means that each process will release the communication channel to other processes. If we check the number of processes running in Fedora , the number is 30% higher. This can bee seen in the Figure 5.10

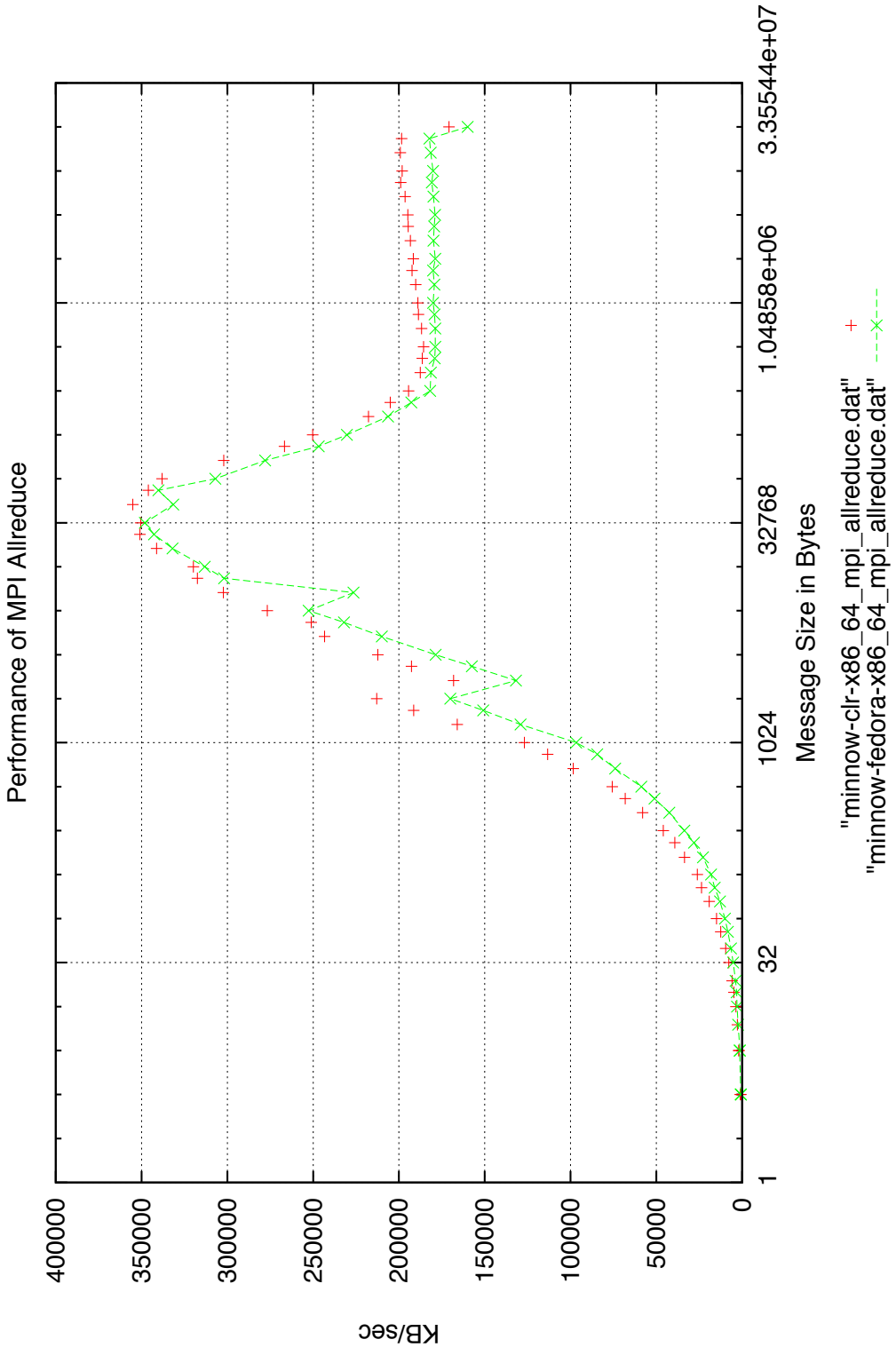


Figure 5.5: *MPI all reduce* benchmark running in MinnowBoard MAX with Clear Linux and Fedora.

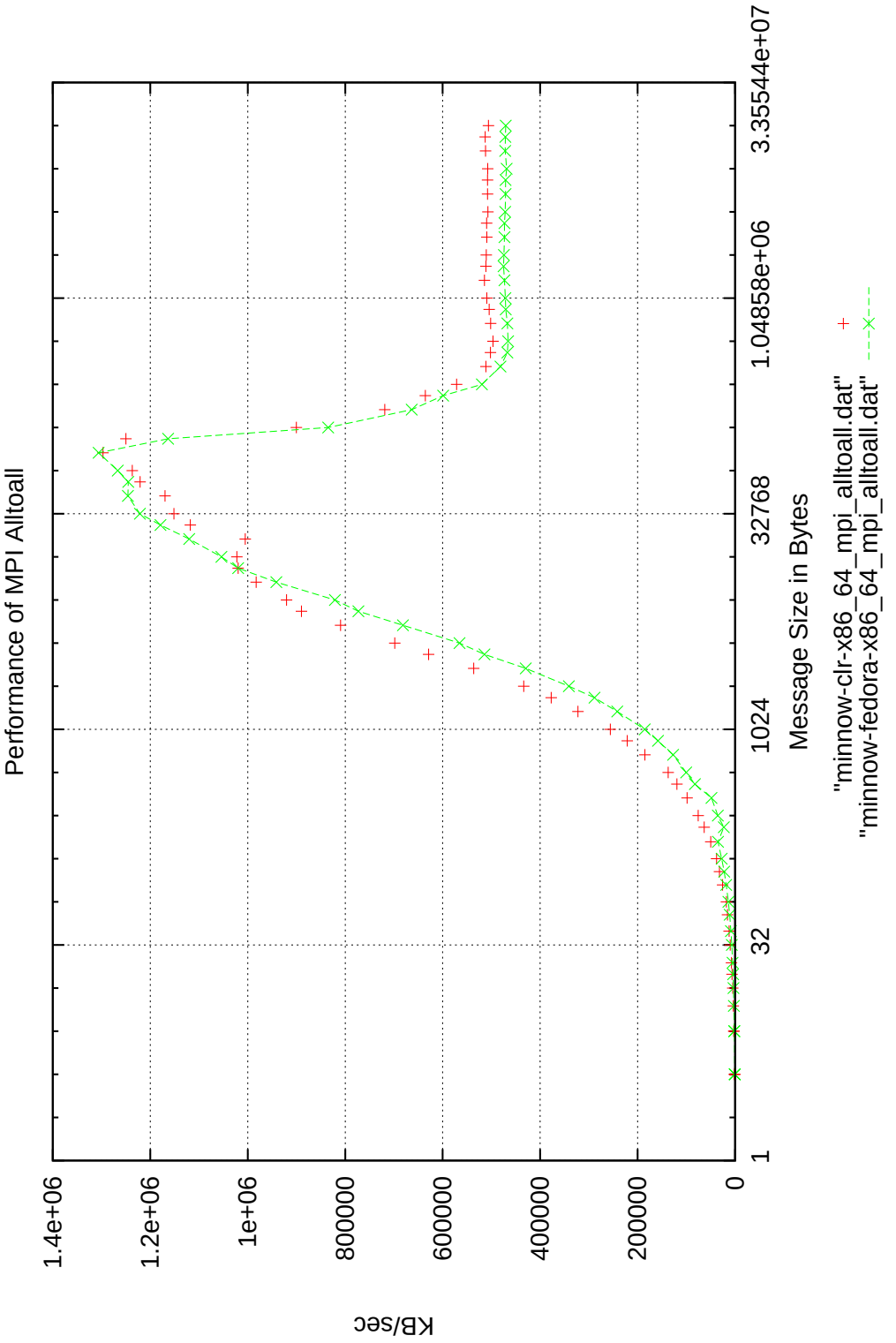


Figure 5.6: *MPI all to all* benchmark running in MinnowBoard MAX with Clear Linux and Fedora.

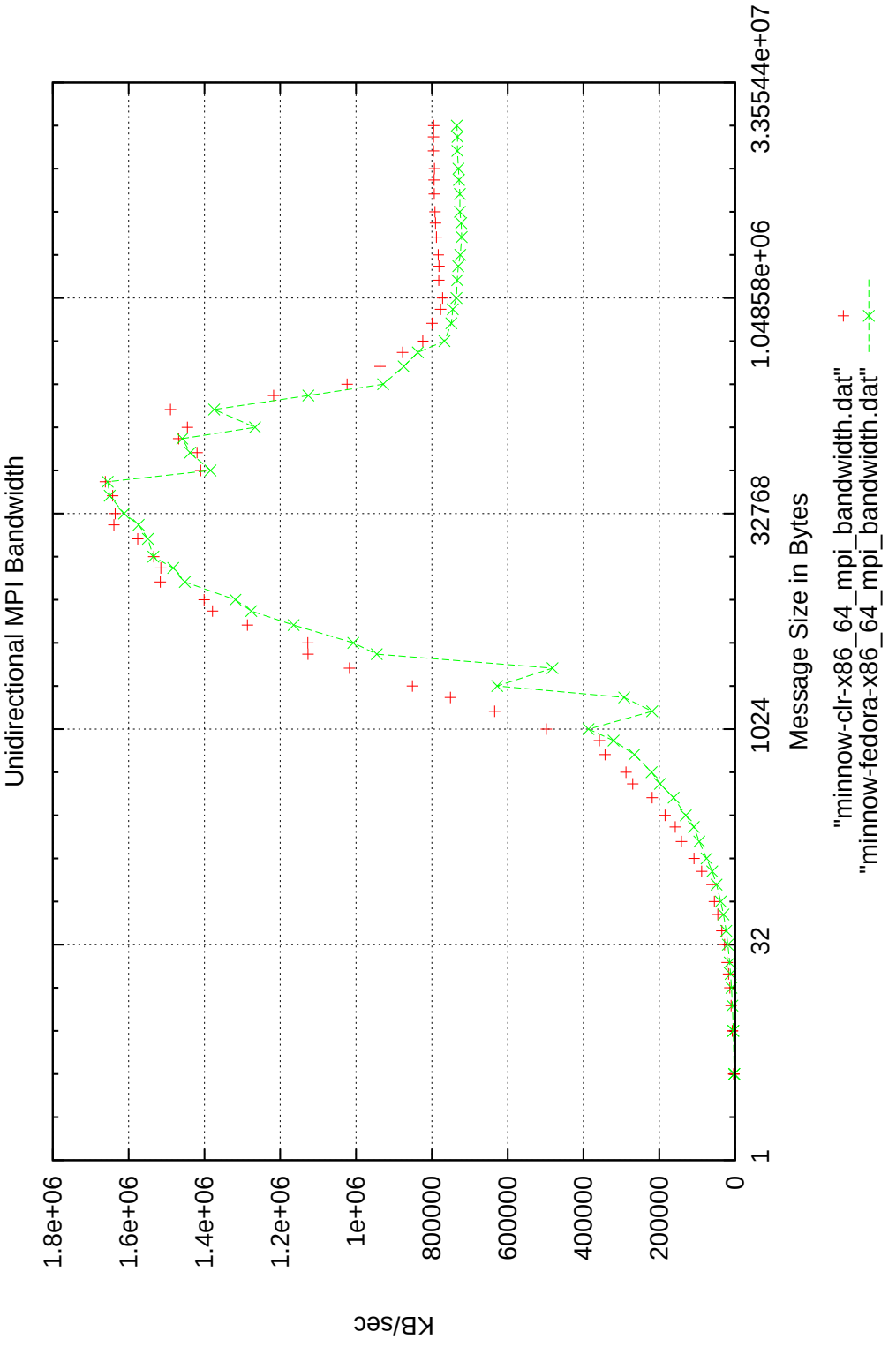


Figure 5.7: *MPI bandwidth* benchmark running in MinnowBoard MAX with Clear Linux and Fedora.

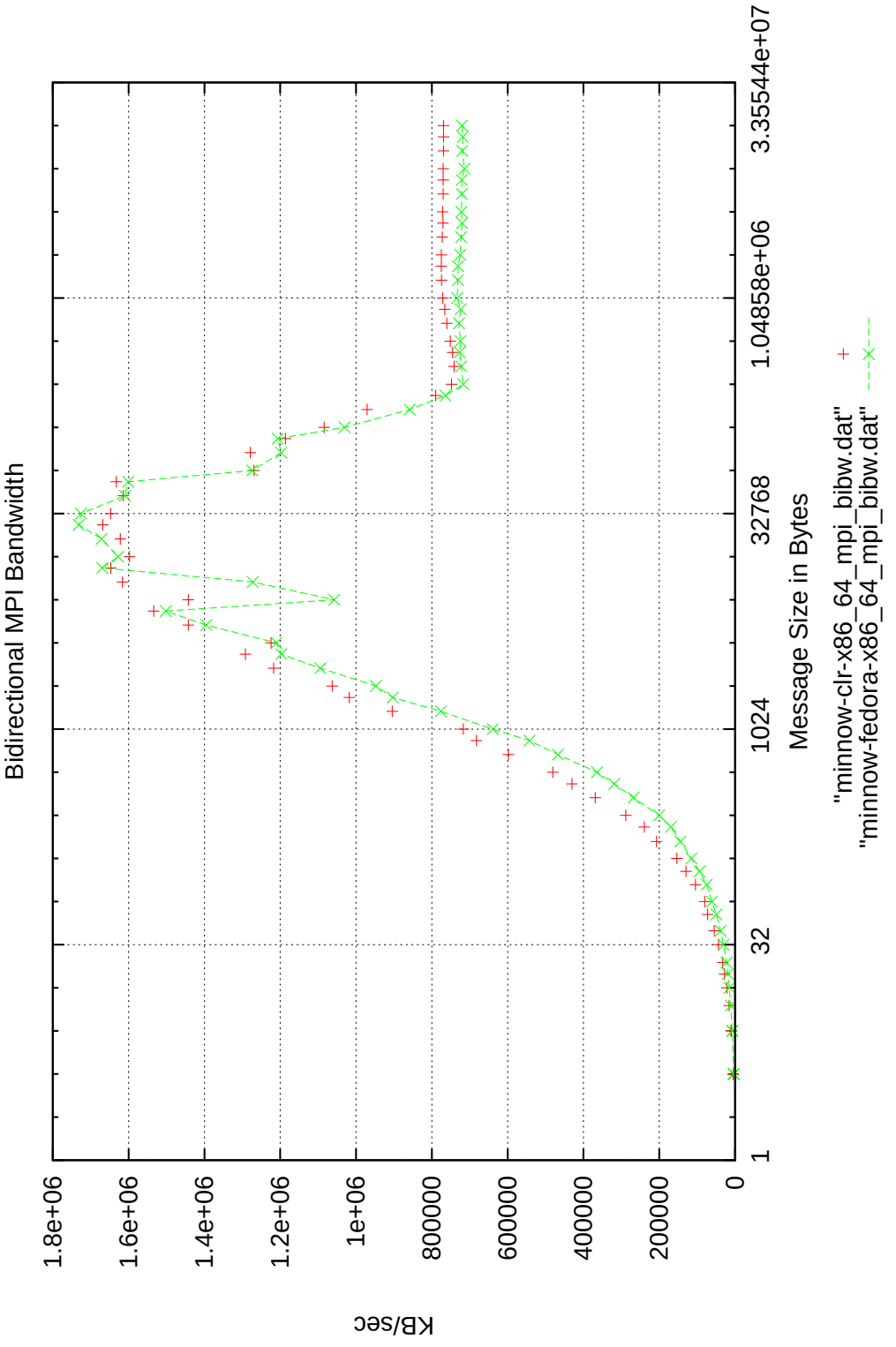


Figure 5.8: *MPI Bi directional* bandwidth running in MinnowBoard MAX with Clear Linux and Fedora.

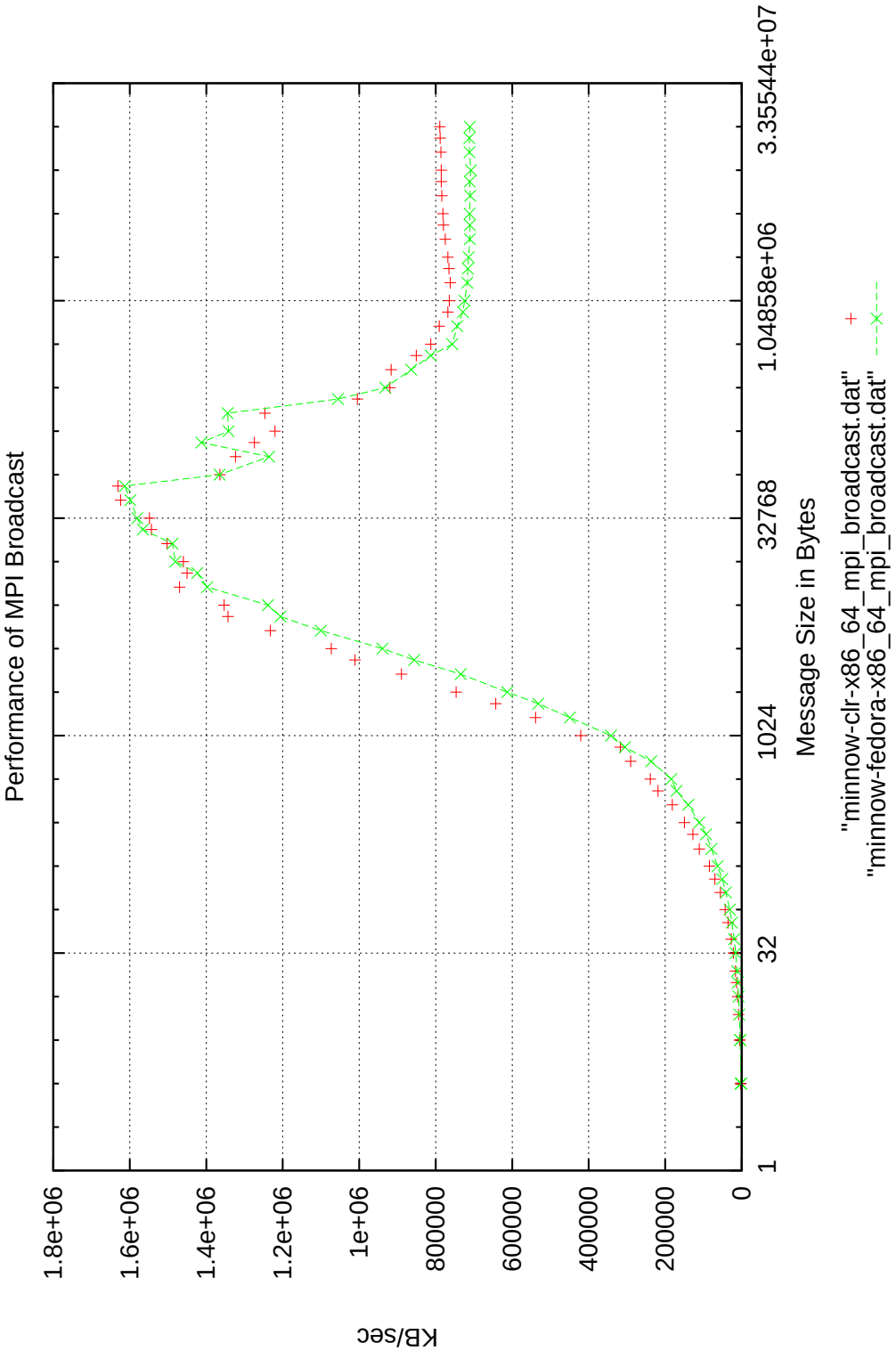


Figure 5.9: *MPI broadcast* benchmark running in MinnowBoard MAX with Clear Linux and Fedora (higher is better)

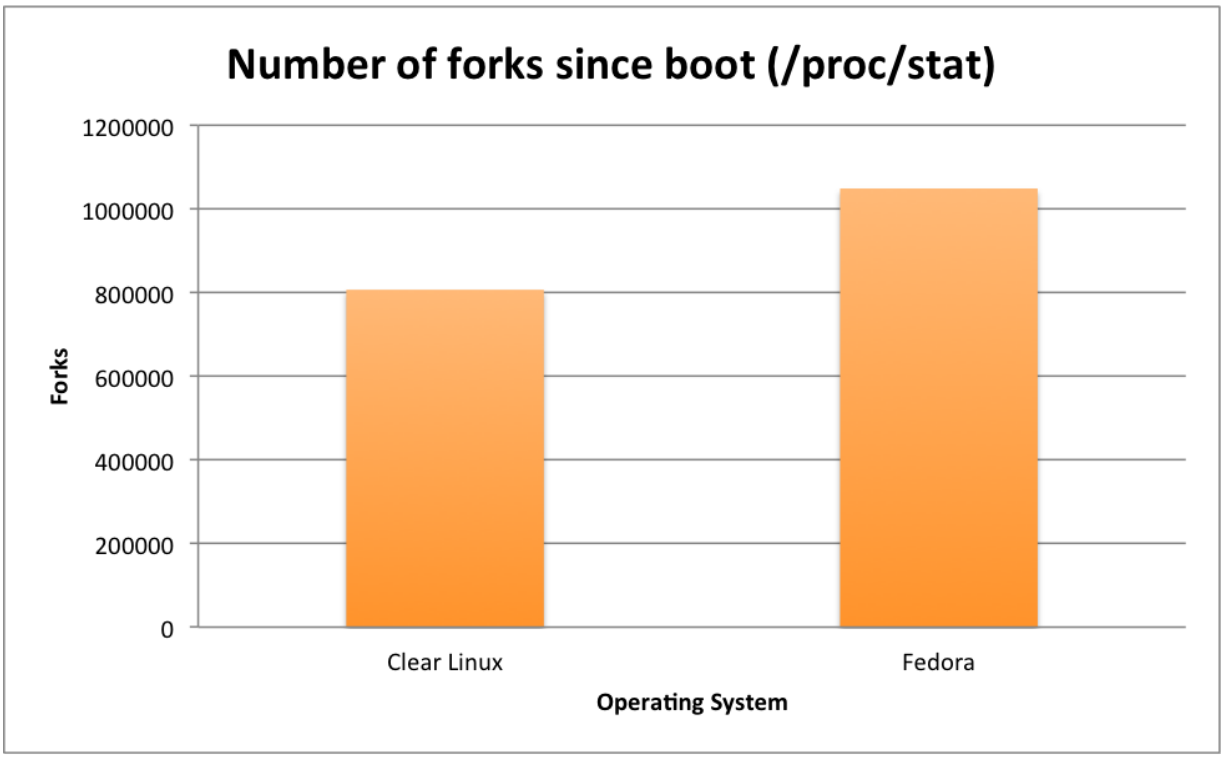


Figure 5.10: Number of forks since booting process reported in /proc/stat file

The *latency* benchmark can be described as one that measures the time for an application to issue a send and continue computing (More details in Chapter 3).

Figure 5.11 shows the latency in both operating systems is similar. The significant change is at the beginning of the test (before the package size reach be 32 Bytes). This is due to the fact that the test includes one system, therefore there is no network message. It does not have to travel through multiple points due to the simplicity of the cluster network (the send message is received immediately)

Roundtrip times are measured in much the same way as *bandwidth*, except that, the slave process, after receiving the message, echoes it back to the master. No acknowledgment is needed with this test as it is implicit given its semantics. The master's pseudo code for this test is described in Chapter 3.

Similar results to *roundtrip* test are shown in Figure 5.12

Based on the results from these experiments it can be observed that customized operating system provides benefits to a cluster of embedded platforms. Experiments with two operating systems, one commercial (Fedora) and one customized (Clear linux for Intel

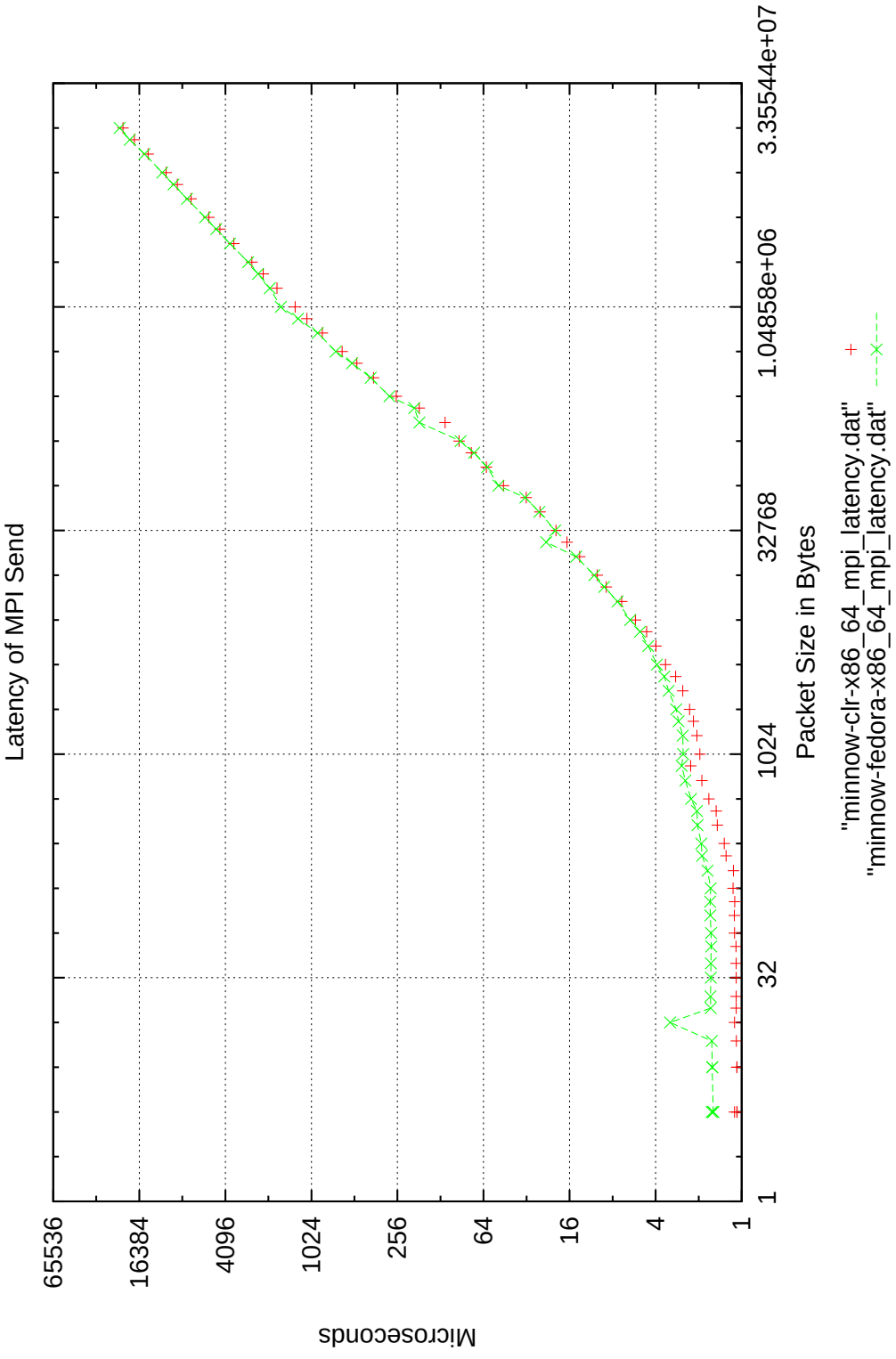


Figure 5.11: *MPI latency* benchmark running in MinnowBoard MAX with Clear Linux and Fedora

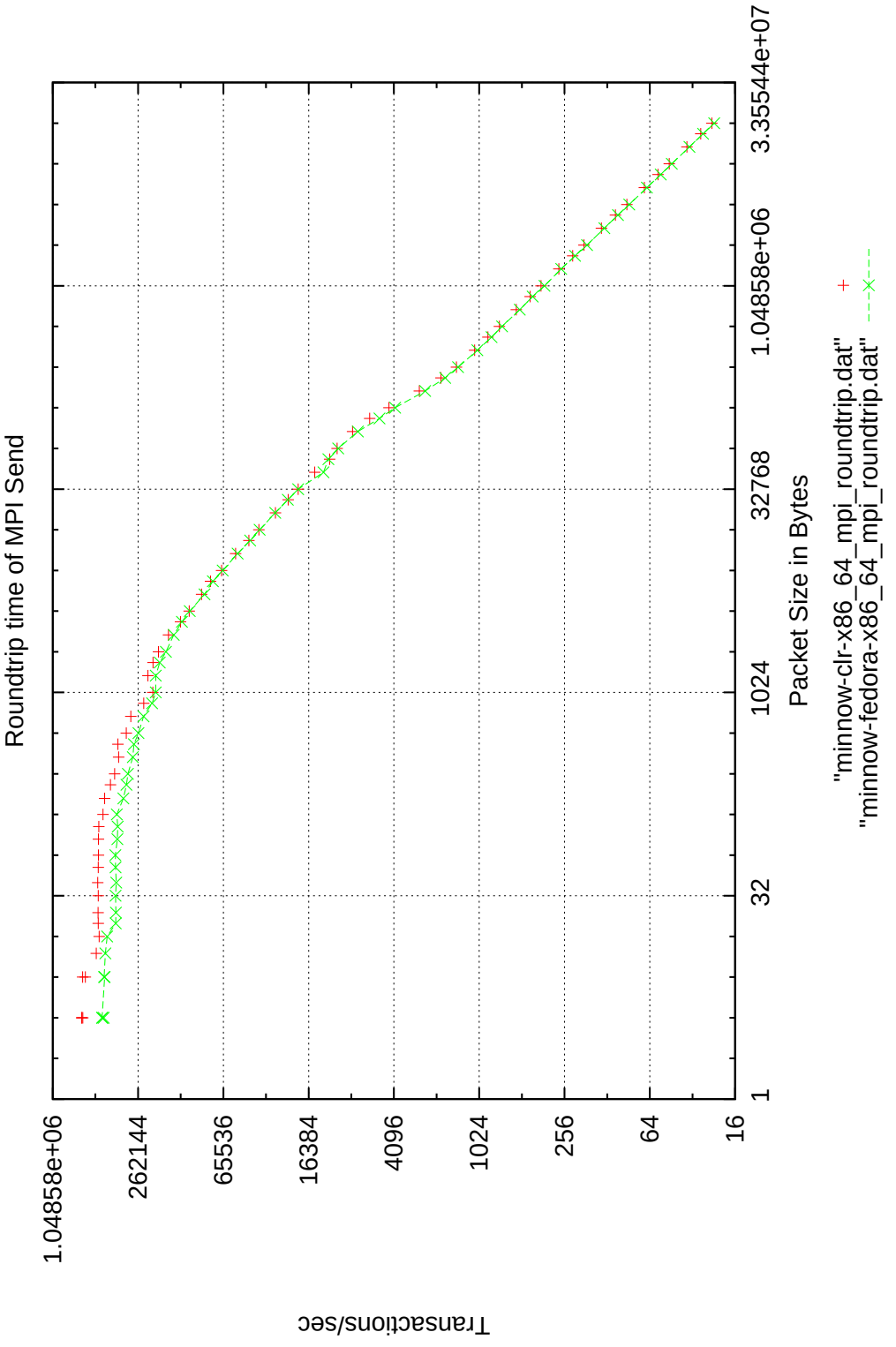


Figure 5.12: *MPI Roundtrip* benchmark running in MinnowBoard MAX with Clear Linux and Fedora.

Architecture). It was worth trying with an operating system designed for embedded and IoT platforms.

The standard of the embedded industry to generate a custom Linux OS is the Yocto project. In table 3.1, the operating systems generated with the Yocto project do not have an implementation of the MPI library.

To implement MPI experiments using Yocto is necessary to port the MPI implementation.

5.4 Porting MPI to Yocto

The system under test (Hardware and Software) for this experiment is described in the Table 5.4

Platform under test	Minnow Board Max
Number of platforms	1
Operating System	Fedora, Clear Linux and Yocto

Table 5.4: Description of system under test (HW/SW) for MPI benchmark evaluation on Yocto

The Yocto project [39] generates a custom Linux OS for embedded systems. The way to add a new capability to the system is by adding a new receipt. This is described in the Yocto documentation. In order to make the Yocto evaluation it was necessary to implement MPI in Yocto. The receipt implemented for MPI is shown in Figure 5.13

```
SUMMARY = "Message Passing Interface (MPI) implementation"
HOMEPAGE = "http://www.mpich.org/"
SECTION = "devel"

LICENSE = "BSD-2-Clause"
LIC_FILES_CHKSUM = "file://COPYRIGHT;md5=2106f0435056f3dd9349747a766e5816"

SRC_URI = " \
        http://www.mpich.org/static/downloads/${PV}/mpich-${PV}.tar.gz \
"
SRC_URI[md5sum] = "40dc408b1e03cc36d80209baaa2d32b7"
SRC_URI[sha256sum] = "455
   ccfaf4ec724d2cf5d8bff1f3d26a958ad196121e7ea26504fd3018757652d"
CACHED_CONFIGUREVARS += "BASH_SHELL=${base_bindir}/bash"

RDEPENDS_${PN} += "bash perl libxml2"
S = "${WORKDIR}/${BP}"

EXTRA_OECONF = "--enable-debuginfo --enable-fast \
                --enable-shared --with-pm=gforker  \
                        --disable-rpath --disable-f77 \
                --disable-fc --disable-fortran --disable-cxx"

inherit autotools-brokensep gettext

do_configure_prepend() {
    autoreconf --verbose --install --force -I . -I confdb/ -I maint/
    oe_runconf
    exit
}
```

Figure 5.13: Receipt to enable MPI libraries in Yocto operating systems

This receipt is part of the meta-openembedded layer, (`http://cgit.openembedded.org/cgit.cgi/meta-openembedded/tree/meta-oe/recipes-devtools/mpich/mpich_3.1.1.bb?h=master`) a layer for all the embedded tools that the Linux distributions might need, in this layer the user can add editors and other libraries that the embedded application might need.

The core of the developed MPI implementation for this work consist in the configuration part. The configuration is described in the "EXTRA_OECONF". The reasons for that are:

- *enable-debuginfo*: For debugging the system when needed
- *enable-fast*: Turns off error checking and collection of internal timing information
- *eenable-shared*; Enable shared library support in order to split the MPI capabilities in multiple binaries and shared object libraries . This makes the compiler and process manager applications smaller in size.
- *ewith-pm=gforker*: The gforker process manager is primarily intended as a debugging aid as it simplifies development and testing of MPI programs on a single node or processor.
- *edisable-rpath* : Do not link static libraries, if so the build applications generated by this MPI implementation will not run in other MPI systems with different paths to libraries.
- *edisable-f77*: Build the Fortran 77 bindings (enabled by default).
- *edisable-fc*: Build the Fortran 90 bindings (enabled by default), This work is not supporting Fortran, it was needed to disable it.
- *edisable-fortran*: Yocto project does not support Fortran.
- *edisable-cxx*: Build the C++ bindings (enabled by default). This work does not support C++ to generate a small MPI implementation.

5.4.1 Results

With this configuration it was created an MPI implementation for Yocto, not just the compiler, but also the MPI process manager. The results of doing this can be seen in the following graphs of the MPI benchmarks, now running with a Yocto base image.

The performance of *MPI reduce* test is similar despite the operating system is installed on the platform. This can be seen in the Figure 5.14.

The performance of *MPI all reduce* test is grater with the Yocto based operating system. This can be seen in the Figure 5.15.

The results with the Yocto based operating system have the same tendency that with the Clear Linux OS. Excepting the *all to all* experiment, where the gain in performance was higher with a message size larger than 32 KB. This is because of the test measures a round-robin communication between multiple processes [1].

MPI *all to all* is a collective operation in which all processes send the same amount of data to each other, and receive the same amount of data from each other. Each process has its own data, at the end, all the processes have the same amount of data.

Each of the process are blocking processes, which means that the number of processes does not affect the performance of the test. However, the configuration of the Yocto Base operating system has some modifications for the specific tested platforms. Mainly in the kernel section as it is shown in the Figure 5.17.

```
1  # enable cpu frequency scaling and stats for powertop
2  CONFIG_CPU_FREQ=y
3  CONFIG_CPU_FREQ_STAT=y
4  CONFIG_X86_ACPI_CPUFREQ=y
5  CONFIG_X86_INTEL_PSTATE=y
6  CONFIG_CPU_FREQ_GOV_ONDEMAND=y
7  CONFIG_CPU_FREQ_GOV_PERFORMANCE=y
8  CONFIG_CPU_FREQ_DEFAULT_GOV_ONDEMAND=y
```

Figure 5.17: Performance improvement in Kernel for Yocto operating systems.

[1] More information regarding to *all to all* test in Chapter 3

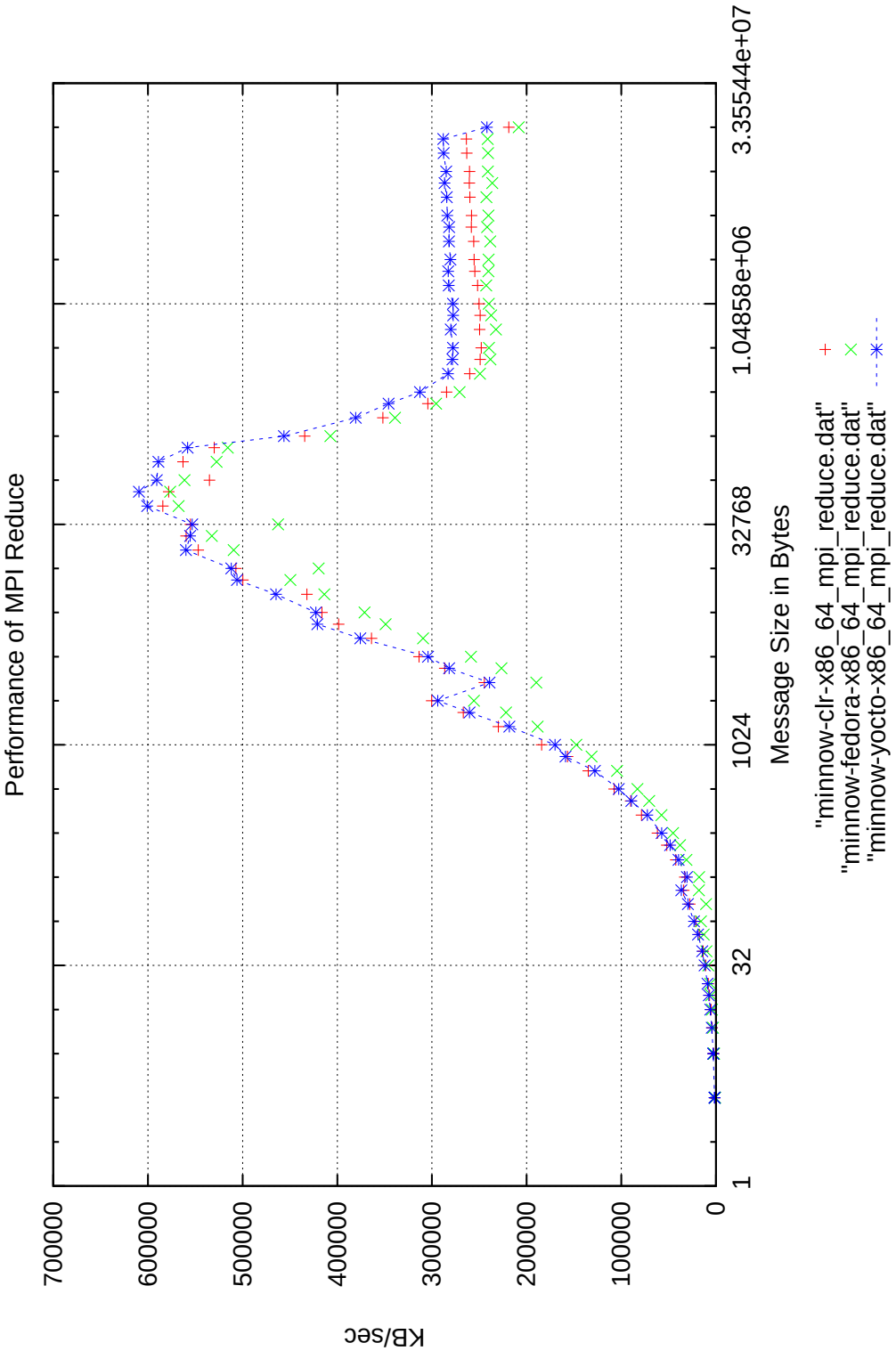


Figure 5.14: MPI Reduce benchmark running in MinnowBoard MAX with Clear Linux, Yocto and Fedora.

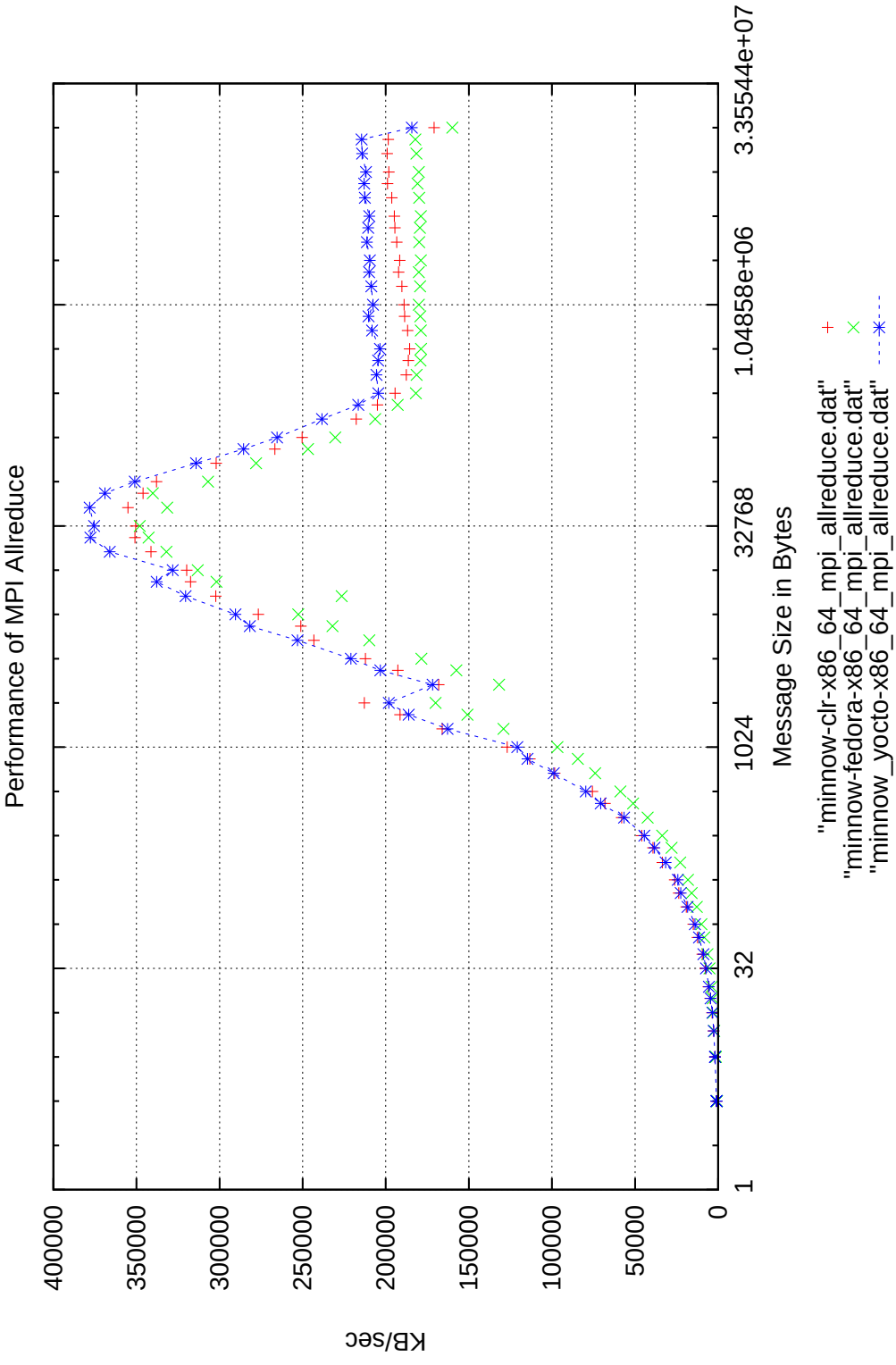


Figure 5.15: *MPI all reduce* benchmark running in MinnowBoard MAX with Clear Linux, Yocto and Fedora (higher is better)

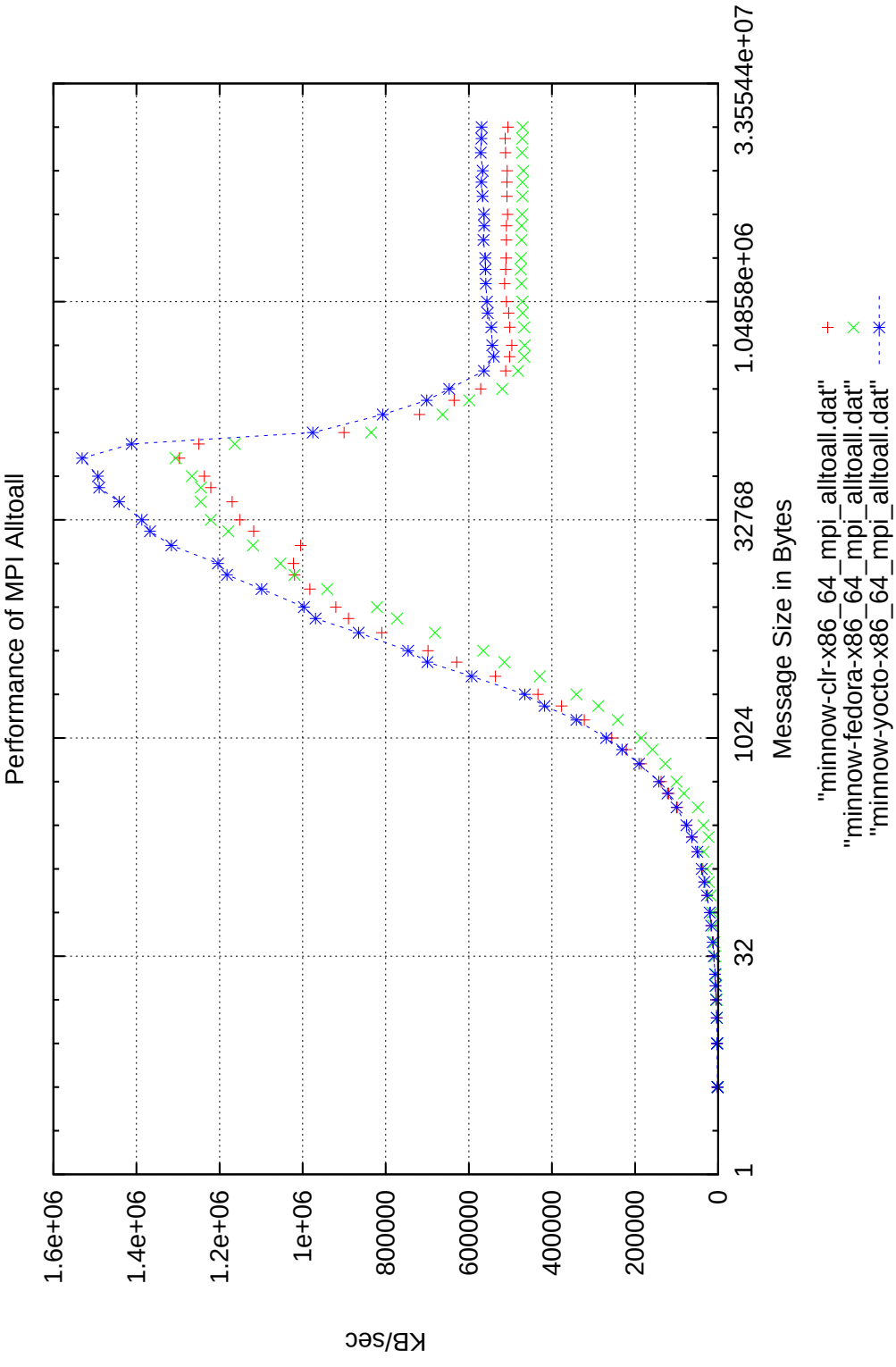


Figure 5.16: *MPI all to all* benchmark running in MinnowBoard MAX with Clear Linux, Yocto and Fedora (higher is better)

The Yocto kernel for the platform under test has the CPU clock frequency scaling enable. Clock scaling allows to change the clock speed of CPUs at runtime. This is a nice method to save battery during runtime, as well as, a nice method to boost the speed of the system when required. In this case the speed of the system is increased upon request when the send/receive method requires it.

The performance of *MPI bandwidth* test is higher with the Yocto based operating system. This can be seen in the Figure 5.18.

Same result can be seen in Figure 5.19 and 5.20. Specially after the message size is 32KB. The test performance of the test *MPI bandwidth* is is higher due to the use of a Yocto operating system.

Based on the results gotten in the previous tests of bi directional band width (figure 5.8) and the broadcast test (Figure 5.9), the main reason why Clear Linux ran with higher speed was because of the number of processes (forks) since fighting for the same resources (network / memory / CPU). For this experiment the number of booting processes in Yocto is 15% lower than CLR and 35% lower than Fedora. This can bee seen in the Figure 5.21.

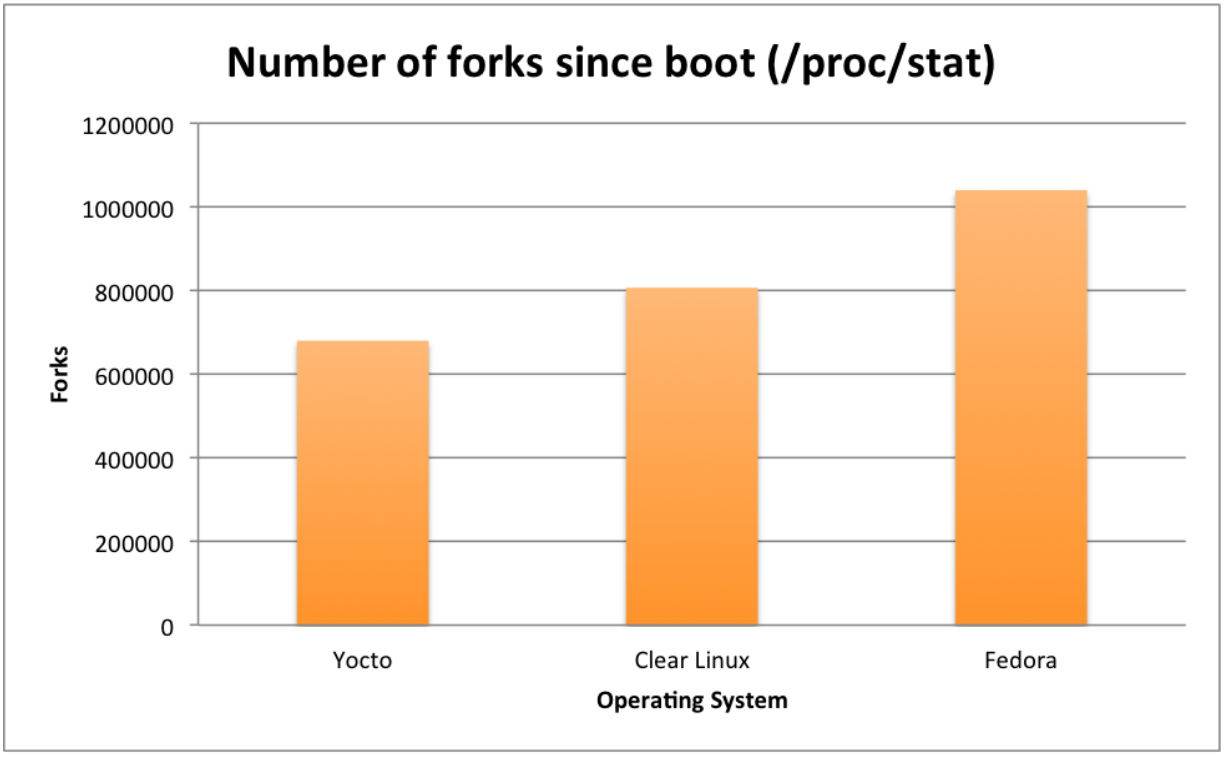


Figure 5.21: Number of forks since booting process reported in /proc/stat file

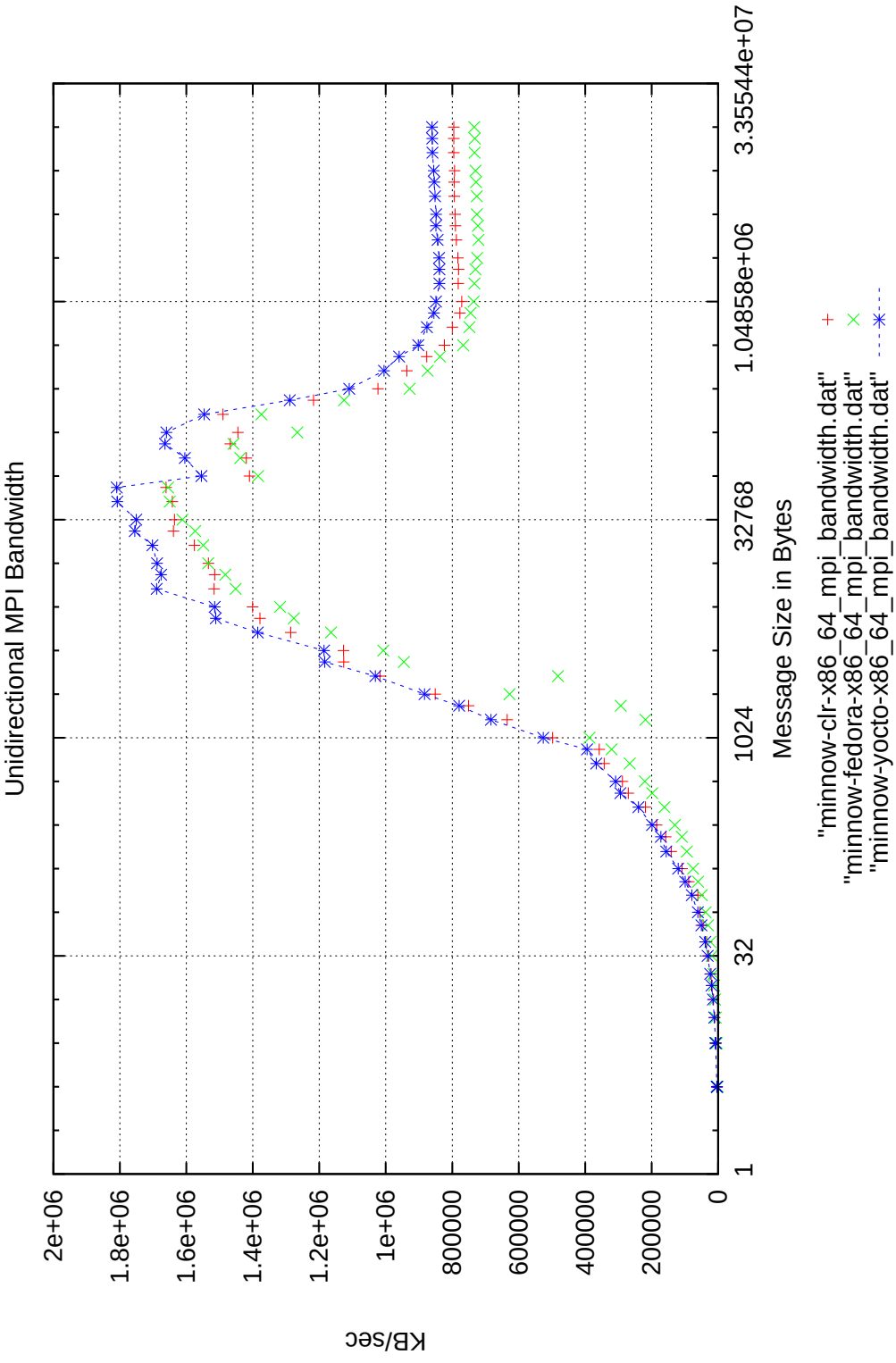


Figure 5.18: *MPI bandwidth* benchmark running in Minnowboard MAX with Clear Linux, Yocto and Fedora.

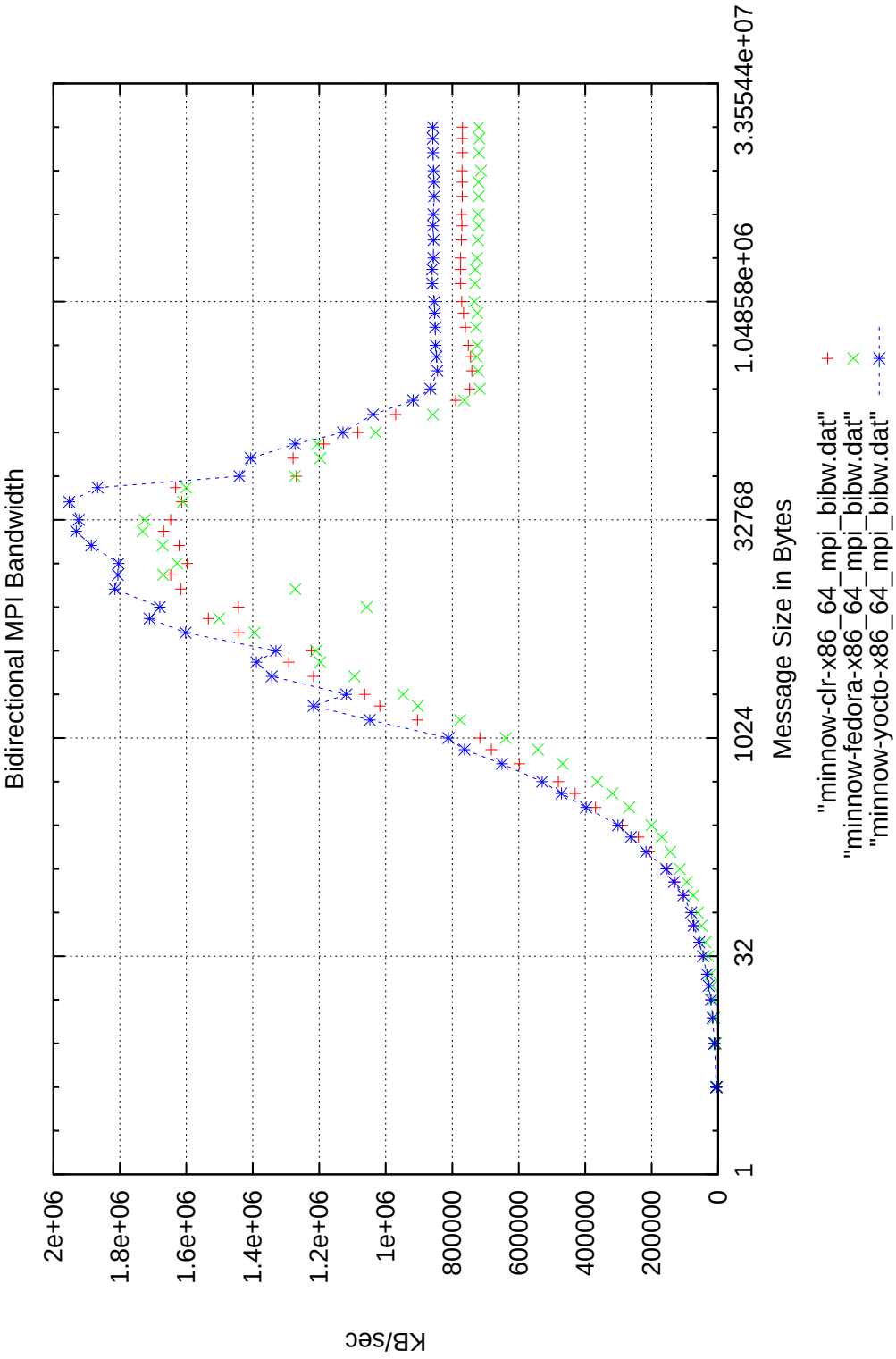


Figure 5.19: *MPI Bi directional* bandwidth running in MinnowBoard MAX with Clear Linux, Yocto and Fedora.

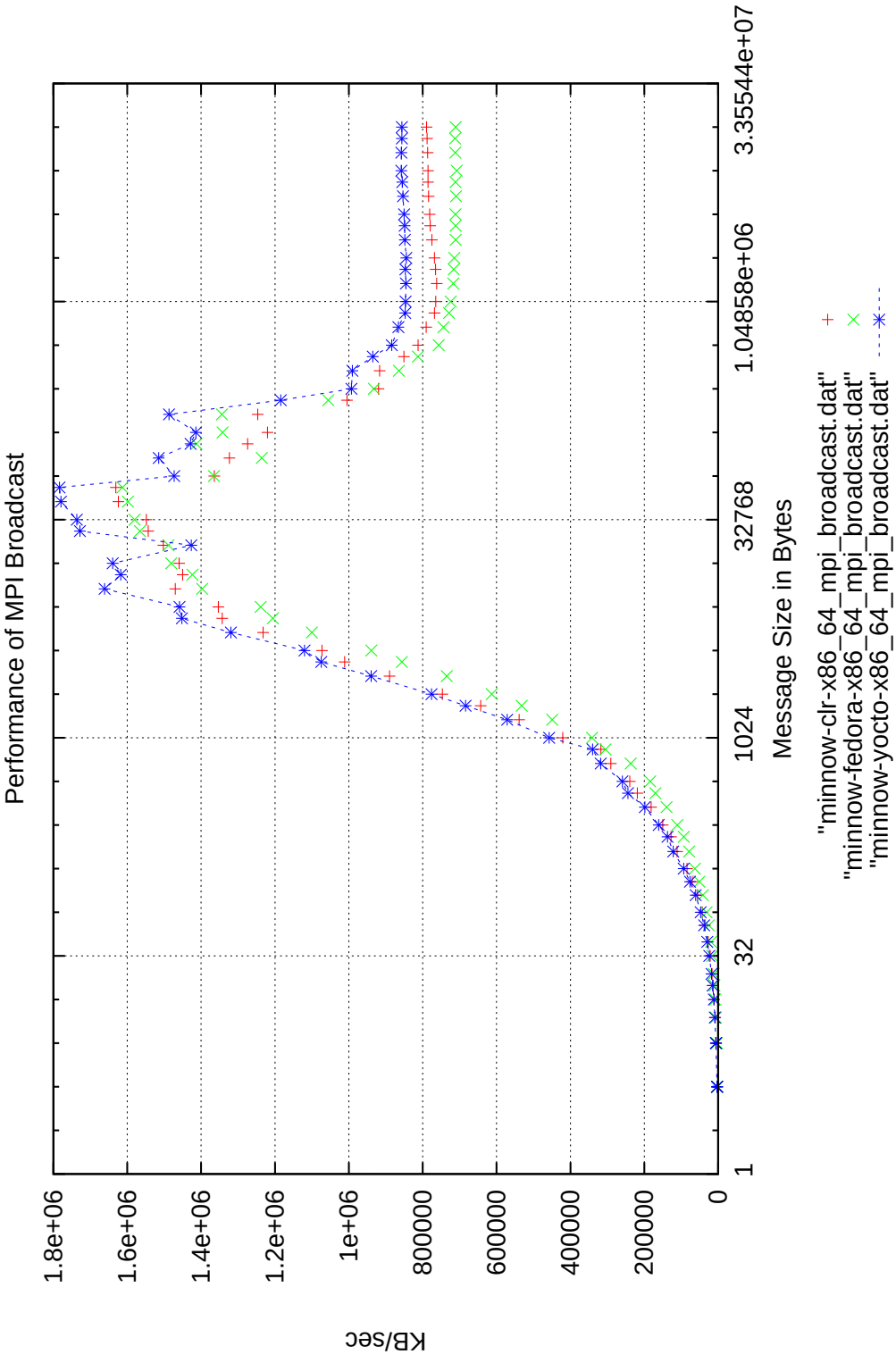


Figure 5.20: *MPI Broadcast* benchmark running in Minnowboard with Clear Linux, Yocto and Fedora

Due to the fact that the test are running in a single platform, latency test (Figure 5.22 has the same performance despite the change in operating system.

Figure 5.22 has similar results than Figure 5.23). Despite the operating system running on the platform the performance is the same, due to the fact that they were a single platform.

After these experiment is proved that the use of Yocto as the operating system for the embedded platforms improves the performance of the MPI benchmarks.

5.5 MPI Benchmark Performance Comparison Between Cluster of Embedded Systems and a Desktop Computing System

The system under test (Hardware and Software) for this experiment is described in the Table 5.5

Platform under test	MinnowBoard MAX and Intel NUC D54250WYK
Number of embedded platforms	6
Number of desktop platforms	1
Operating System on embedded platform	Yocto
Operating System on desktop platform	Clear Linux

Table 5.5: Description of system under test (HW/SW) for MPI benchmark performance comparison between cluster of embedded systems and a desktop computing system

After doing the experiments in a single Atom-based system it was decided to do the experiments in a cluster of them. The experiment is limited to the number of available systems we can gather. In this case, it was possible to get six MinnowBoard MAX platforms [43] platforms to do the experiments. The full diagram of the cluster is described below in Figure 4.4.

All these systems execute the MPIBenchmarks with the following ssh config file:

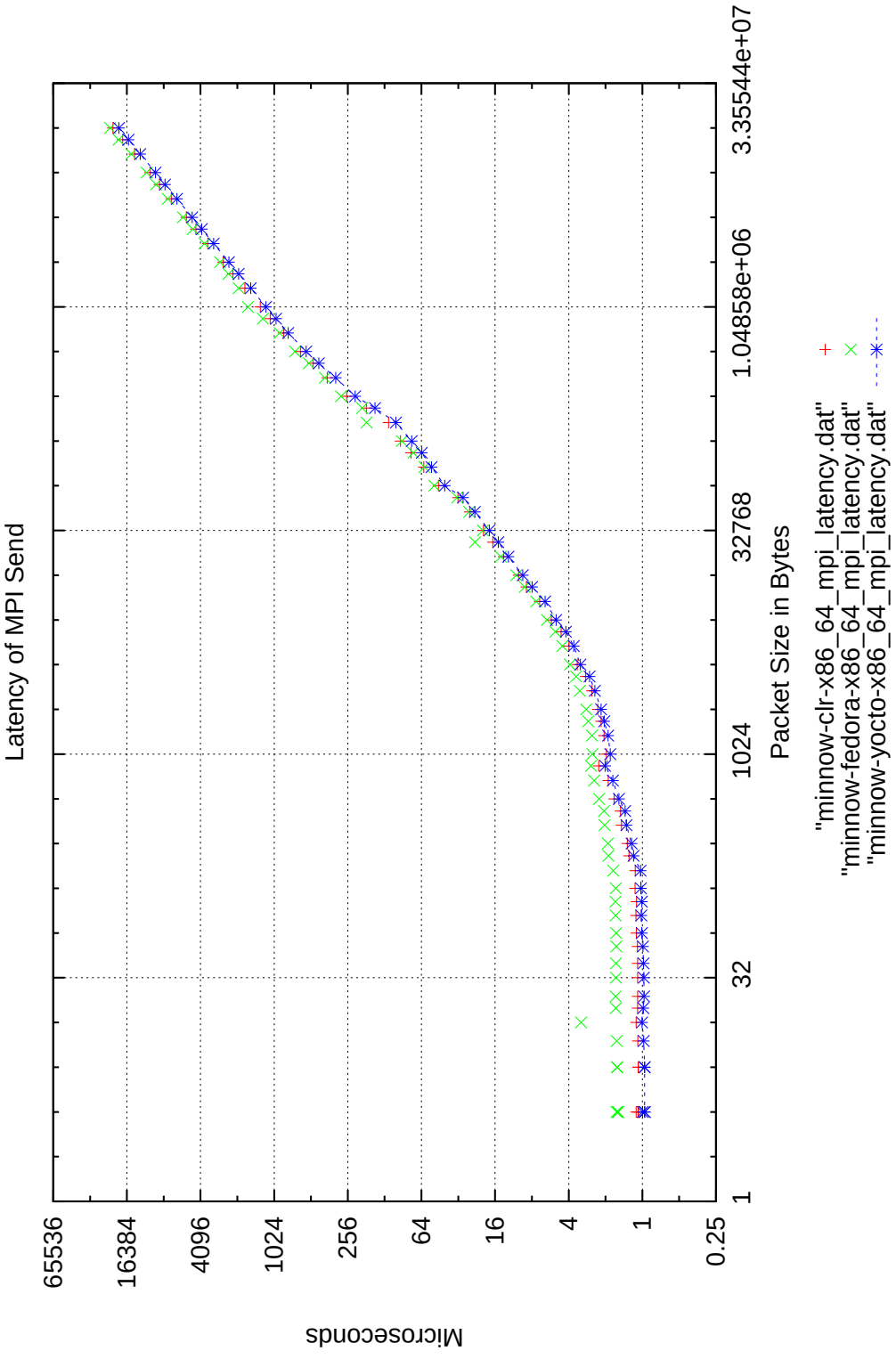


Figure 5.22: *MPI latency* running in MinnowBoard MAX with Clear Linux, Yocto and Fedora.

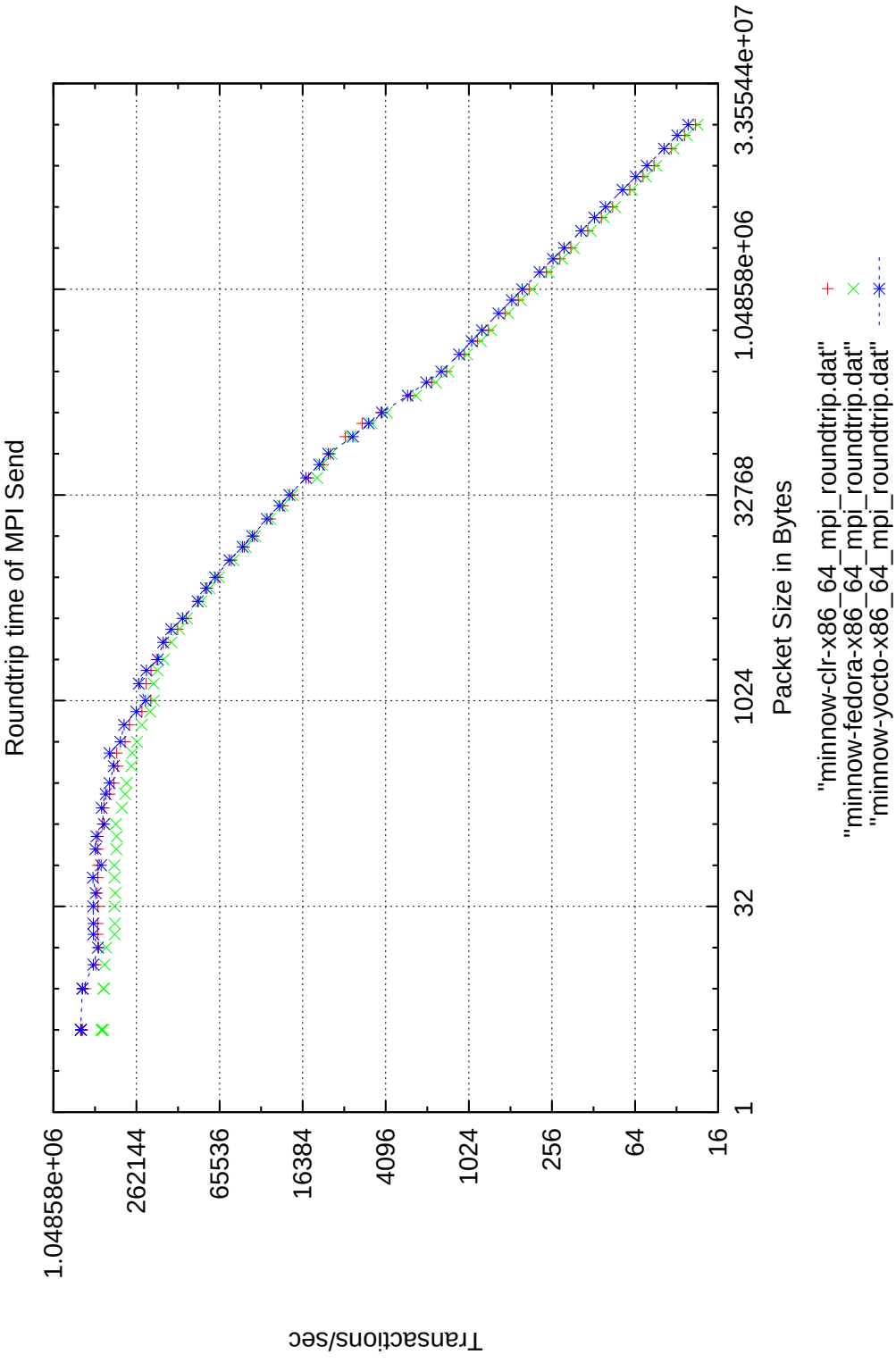


Figure 5.23: *MPI Bi directional* bandwidth running in MinnowBoard MAX with Clear Linux, Yocto and Fedora.

```
$ cat ~/.ssh/config
  Host node1
      HostName node1-ip-or-hostname
      User user-of-the-ssh-key
      Port port-if-nedded

  Host node2
      HostName node1-ip-or-hostname
      User user-of-the-ssh-key
      Port port-if-nedded

  Host node3
      HostName node1-ip-or-hostname
      User user-of-the-ssh-key
      Port port-if-nedded
```

And the following MPI config file:

```
$ cat hostfile
  node1
  node2
  node3
```

This is necessary to make the systems communicate each other through MPI , if we don't include this configuration there will not be any communications due to the fact that MPI requires an ssh connection.

The idea of this experiment is to compare the performance of the cluster of embedded systems to a traditional computing system (NUC D54250WYK [56]). In this case, the desktop system is based on the processor i5-4250U CPU. The main components of that system is described in Chapter 4 (Table 4.2).

5.5.1 Results

After executing the MPI benchmark in the cluster of embedded systems and in the NUC D54250WYK [56] (with Clear Linux OS [52]) their comparison results are shown in

following figures.

The *all reduce* benchmark starts to have a gain in performance with the addition of more embedded platforms into the cluster of embedded platforms. With two platforms the gain is not seen; however, with three or more the gain is substantial. When the system has five or four platforms it has similar performance. With six platforms the performance is similar (even in the drop point) that the desktop platform [56]

In the *reduce* benchmark the performance is constant across the increment of platforms in the cluster of embedded platforms. Even the similar performance that three and four platforms present.

In the *All to all* benchmark presented in Figure5.26 the performance at 218MB is similar despite the number of platforms the cluster has. In this test the performance of even six platforms is not close to the performance that a NUC [56] system can has.

In the *Bandwidth* performance test it can be seen an interesting behaivor with a cluster of 5 MinnowBoard MAX systems (Figure 5.27). At 218 MB the performance became unstable. The performance can be similar to the one presented by four or 6 platforms. Besides that the performance with other number of platforms is similar to the one presented in Figure 5.26. Even the fact that the performance at 218MB is similar despite the number of platforms the cluster has.

In the *Bi directional bandwidth* benchmark in Figure 5.28 a similar behavior than in Figure 5.27. At 218 MB the performance became unstable, with similar performance from four, five and six platforms. Besides that the increment in performance is constant. Even with three or four platforms the performance improvement is high. From these results it can be seen that for applications that requires speeds between 27MB and 218MB, a cluster of four, five or six embedded platforms can bring similar performance that a traditional desktop system, in terms of bi directional bandwidth.

The *Broadcast* benchmark in cluster of embedded platforms with Yocto OS presents similar performance with three, four, five or six platforms. In this case the fluctuation of performance among these configurations starts 27MB until 1,7 GB.

Latency is higher (Figure 5.30 and Figure 5.31)in the cluster of embedded platforms. This is a side effect of the distributed systems. Total latency is a combination of both

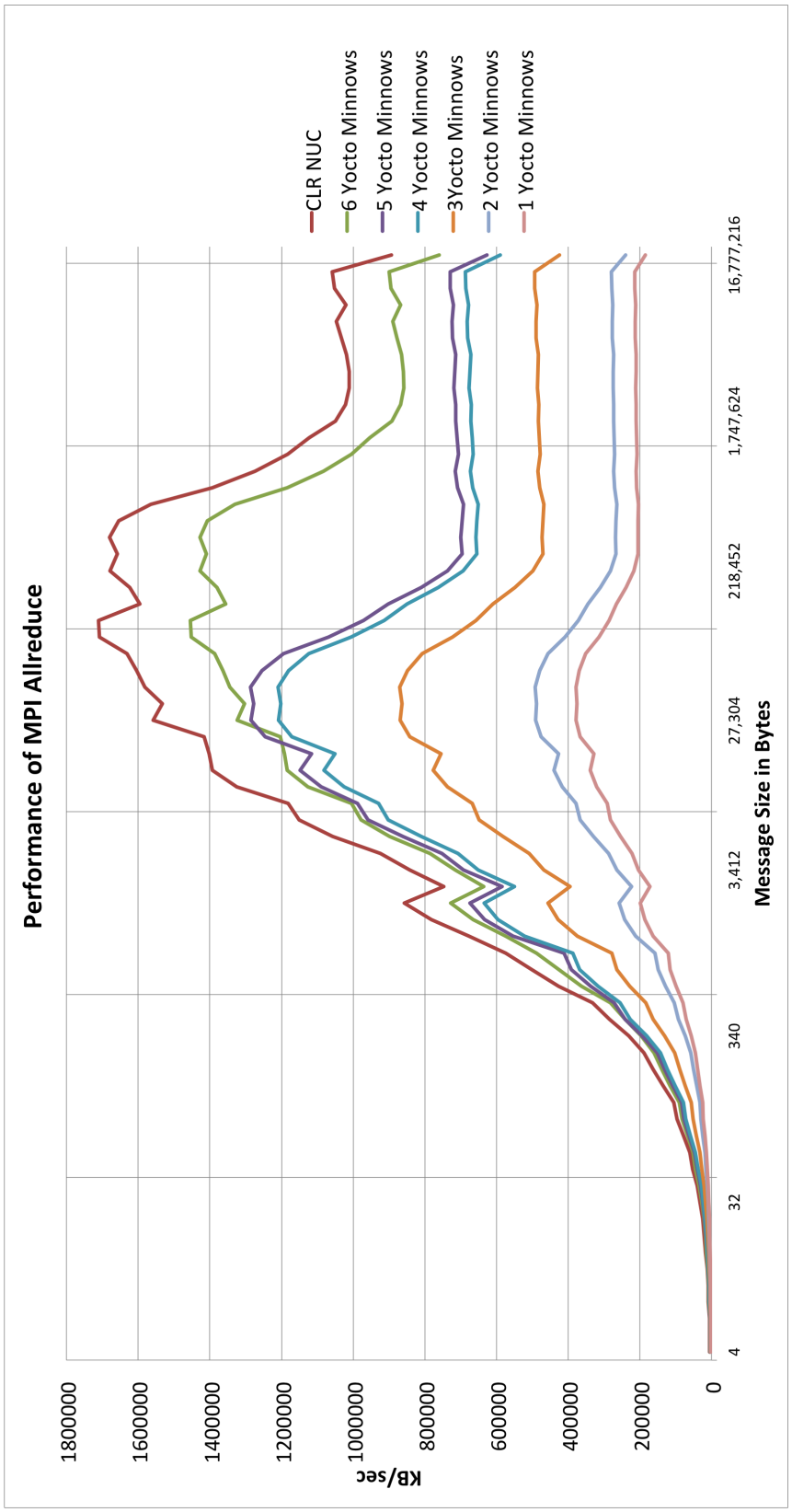


Figure 5.24: *Allreduce* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

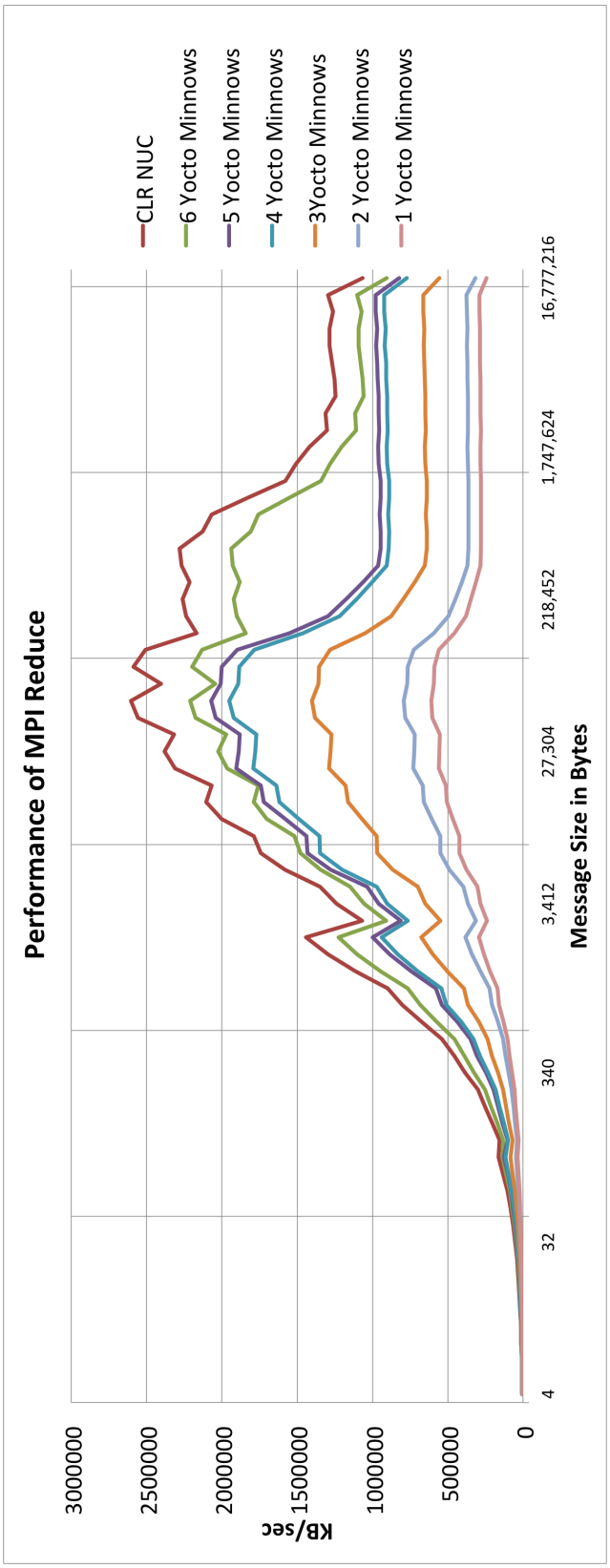


Figure 5.25: *Reduce* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

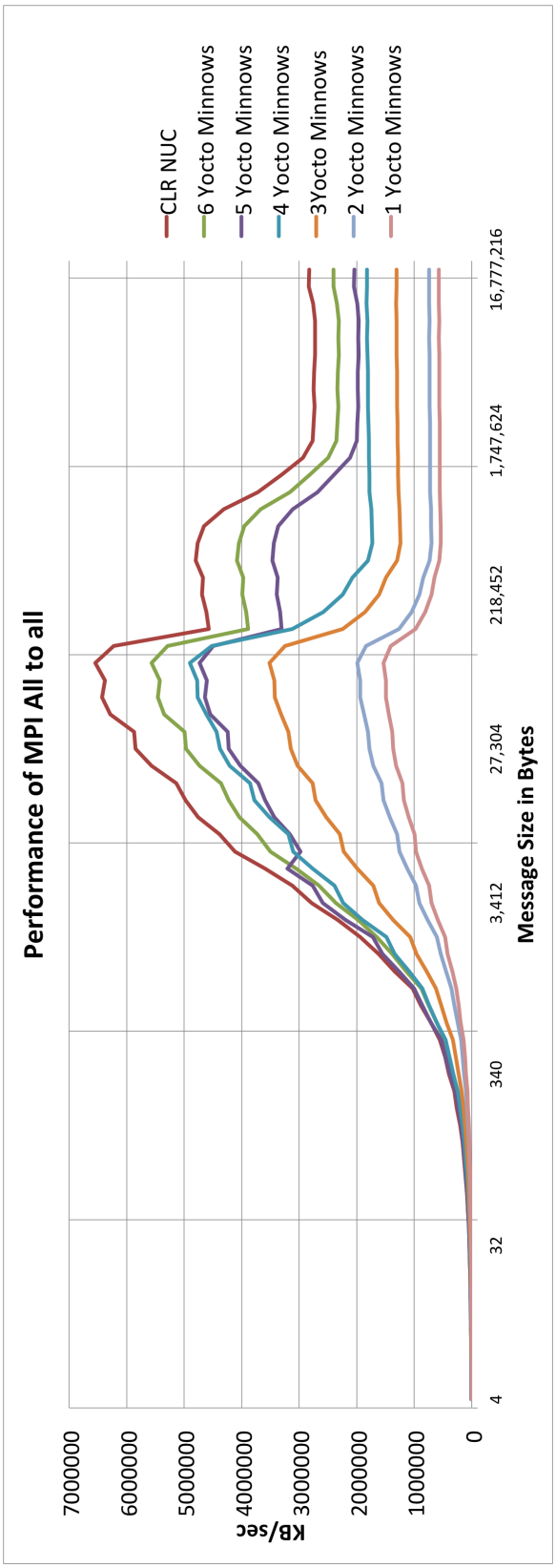


Figure 5.26: *All to All* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

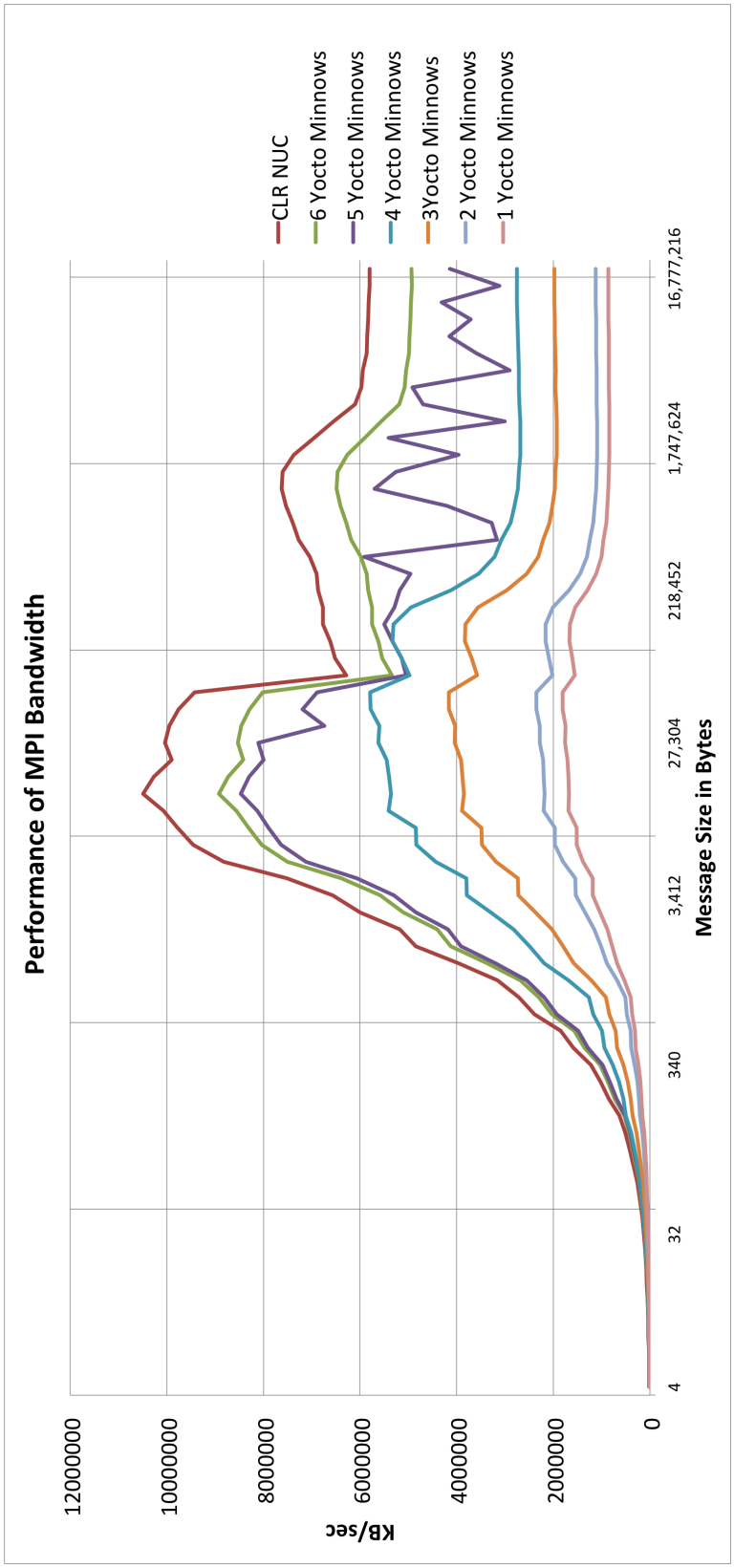


Figure 5.27: *Bandwidth* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

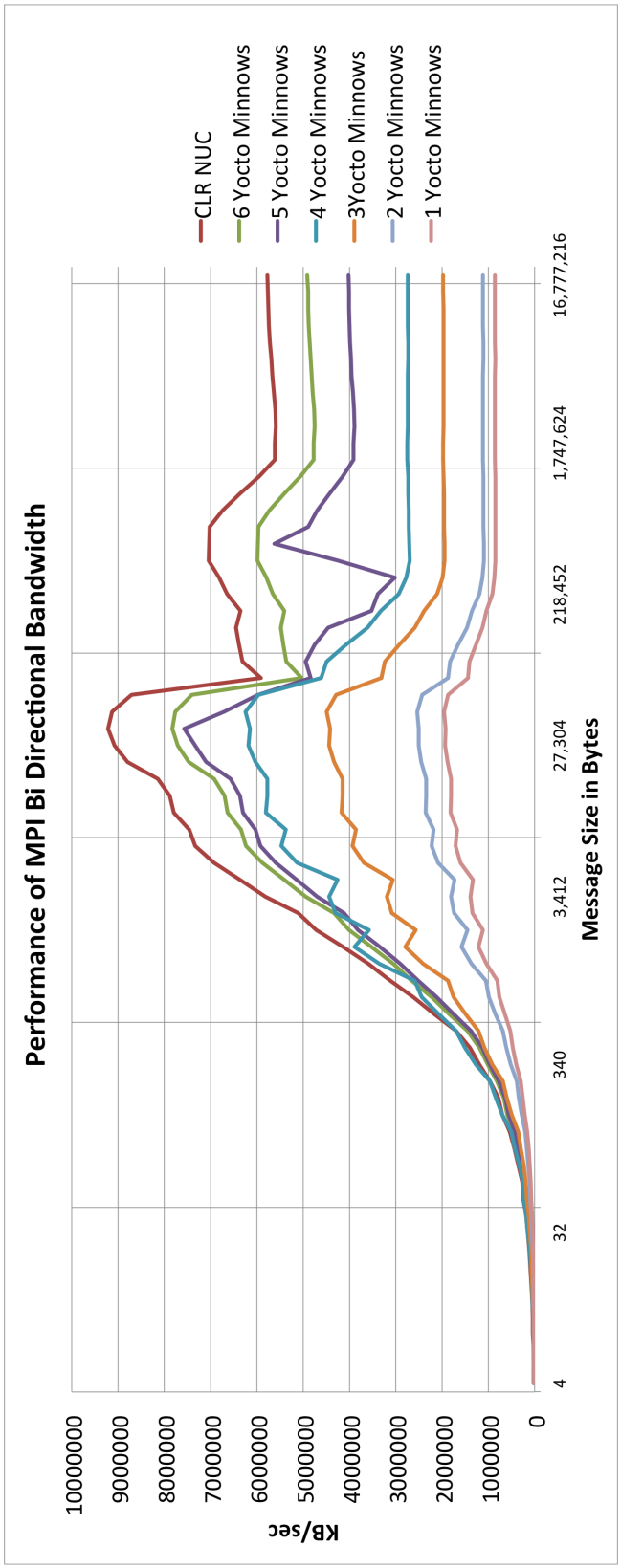


Figure 5.28: *Bi directional bandwidth* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

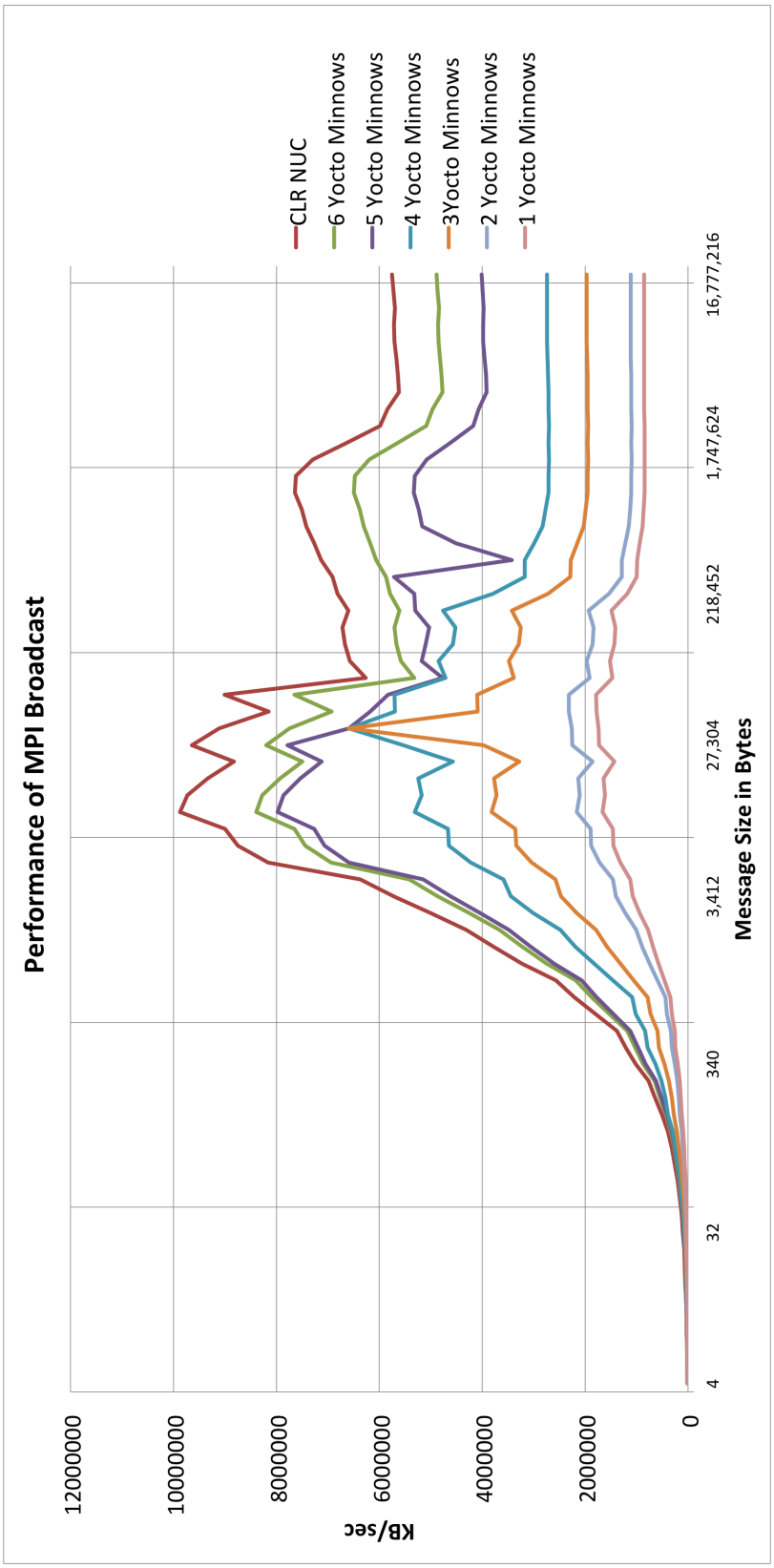


Figure 5.29: *Broadcast* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

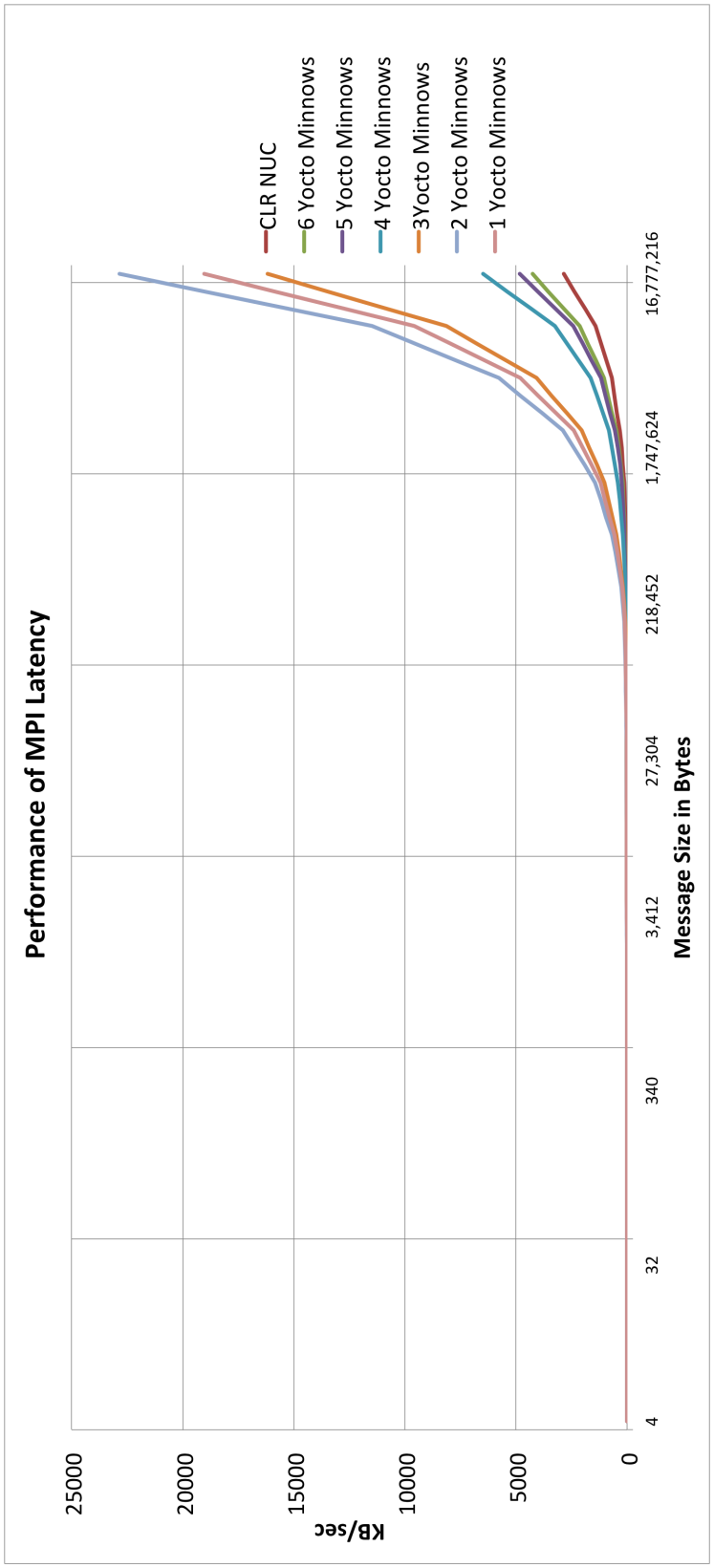


Figure 5.30: *Latency* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

hardware and software factors, with the software contribution generally being much greater than that of the hardware. Latency is really important when the program is dominated by communication, and the messages are small. This can happen when scaling to larger numbers of nodes. In the experiments we perform we can see that the latency of the cluster of embedded platforms is high in comparison to the centralized system [56]

5.6 Power Efficiency Comparison of Cluster of Embedded Systems to a Desktop Computing System

It has been described that for six systems of embedded platform it is not possible to get the same performance impact that with desktop computing systems.So far we have been able to see the performance comparison of a traditional computer system and a cluster of embedded systems, only up to 85% of the performance for some tests. The hypothesis of this work is to determine the optimal point at which it is better to send data to a server instead of performing local computations.

According to the hypothesis of this work, the expected behavior of power efficiency for the cluster of embedded systems is represented in Figure 1.2. In the beginning the increment on the number of nodes in the network will increment the performance (top part of on equation 3.4), but at the same time the amount of watts will increase making the energy efficiency flat at some point (if the lower part of the equation increases then energy efficiency tends to decrease).

The next natural step is to measure the power consumption of both systems : the cluster of embedded systems and the D54250WYK.

The instrument to measure the power consumption of the system is the WT300 Series Digital Power Meter (from Yokogawa) . Digital Power Analyzers are instruments for laboratory, manufacturing test, or embedded applications to accurately measure and analyze electrical power characteristics.

Table 5.6 describes the power consumption of a single MinnowBoard MAX.

The Table 5.7 describe the power consumption of a single NUC D54250WYK.

It is seen that the average power consumption of the NUC is close to 20 watts in the

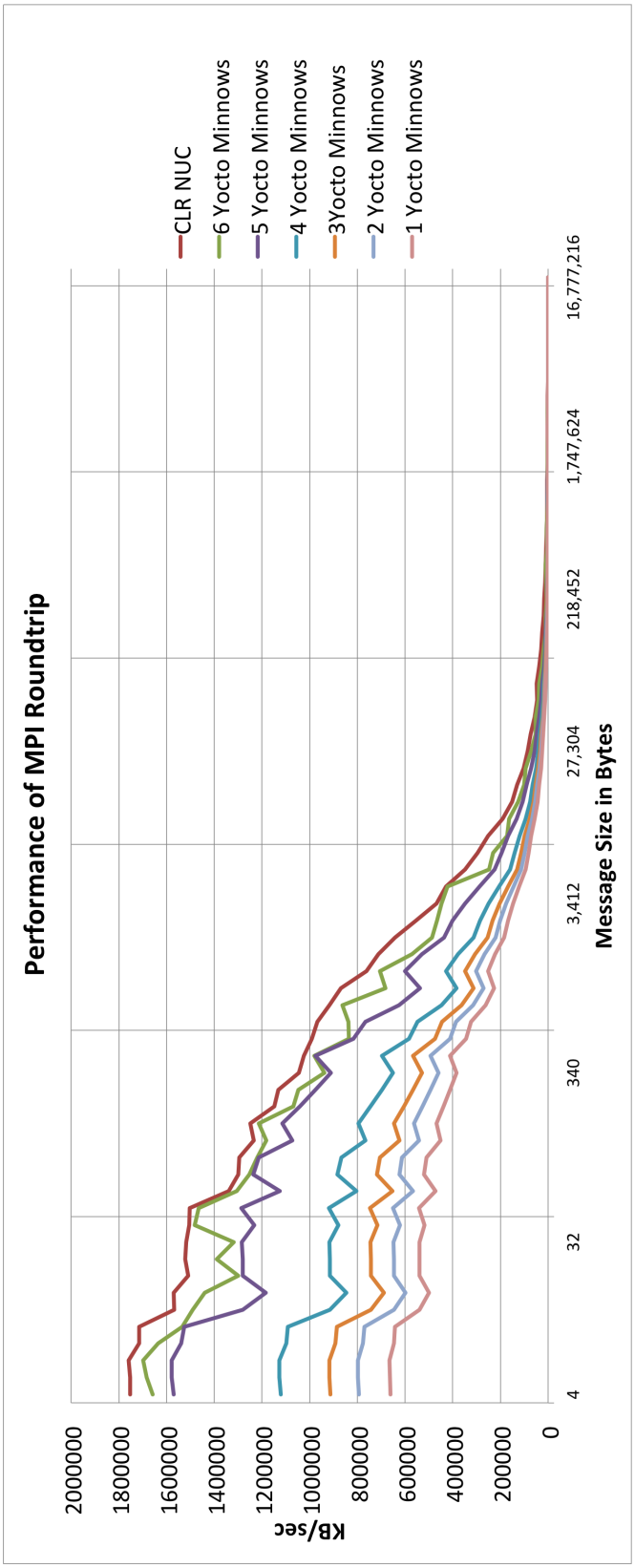


Figure 5.31: *Roundtrip* benchmark in cluster of embedded platforms with Yocto OS and NUC with Clear Linux OS.

	All reduce	Reduce	All to all	Bandwidht	Bi direct bandwidth	Boradcast	Latency	Roundtrip
Average Power Consumption (Watts)	4.234	2.231	2.876	4.9871	4.7865	4.8721	4.1235	4.8563
MAX Watts	5.6139	5.875	5.934	5.823	5.749	5.853	5.761	5.703
MIN Watts	2.8874	2.234	2.7532	2.8342	2.7431	2.6548	2.934	2.985
Average Power at idle (Watts)	2.5663	2.5663	2.5663	2.5663	2.5663	2.5663	2.5663	2.5663

Table 5.6: Power consumption of Minnow Board Max with MPI benchmarking

	All reduce	Reduce	All to all	Bandwidht	Bi directional bandwidth	Boradcast	Latency	Roundtrip
Average Power Consumption (Watts)	18.0742	19.2853	18.6589	18.2385	18.8423	19.4739	18.8664	19.6473
MAX Watts	20.3010	20.1011	20.4240	20.8761	19.9573	20.8436	20.3242	20.3452
MIN Watts	5.2046	4.4239	5.9853	6.5838	5.4086	4.5973	6.2199	6.8419
Average Power at idle (Watts)	4.4801	4.4801	4.4801	4.4801	4.4801	4.4801	4.4801	4.4801

Table 5.7: Power consumption of NUC D54250WYK with MPI benchmarking

meantime the average power consumption of the embedded system is close to 2 Watts. This is due to the design of the embedded system, just the CPU in the embedded system is designed to consume 1 watt in idle mode.

Next it was measured the power consumption of the embedded cluster, this can bee seen in Figure 5.32:

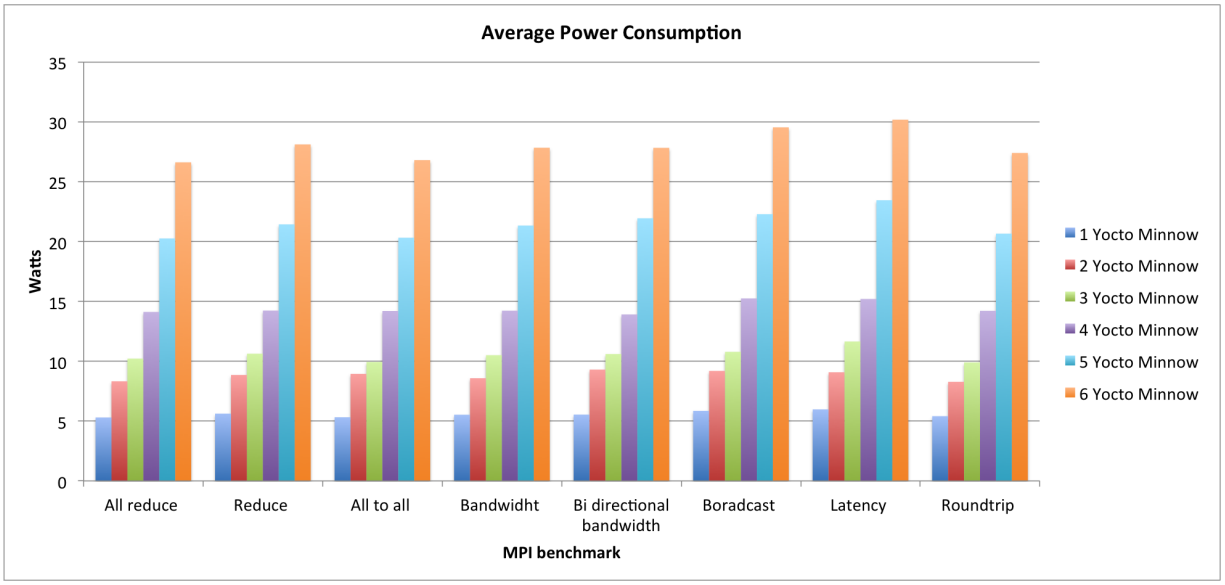


Figure 5.32: Average power consumption in cluster of MinnowBoard MAX platforms

The increment in power consumption can be seen as a linear function in all the MPI benchmarks presents in Figure 5.32. Starting from one MinnowBoard MAX platform with 5 watts until six MinnowBoard MAX platforms that consume up to 30 watts.

To finish the experiments is necesary to measure the power efficiency (equation 3.4) of cluster of embdded platforms. The energy efficiency characterization of the systems can

be described in the following figures.

For the *Allreduce* benchmark the power efficiency is similar to the one presented in the hypothesis in Chapter 1 (Figure 5.33). Despite the gain in performance with the addition of nodes to cluster of embdded platforms the increment in power consumption affects the power efficiency of the system. The power efficiency of one platform is 10,000 KB/s per watt less than the desktop system. If the system increase to two the power efficiency decreases even more, due to the fact that the power consumption is higher than the performance gain. Is until we have three and four platforms that we can see a sustainable gain in performance efficiency. This is correlated with the gain in performance we saw in Figure 5.24.

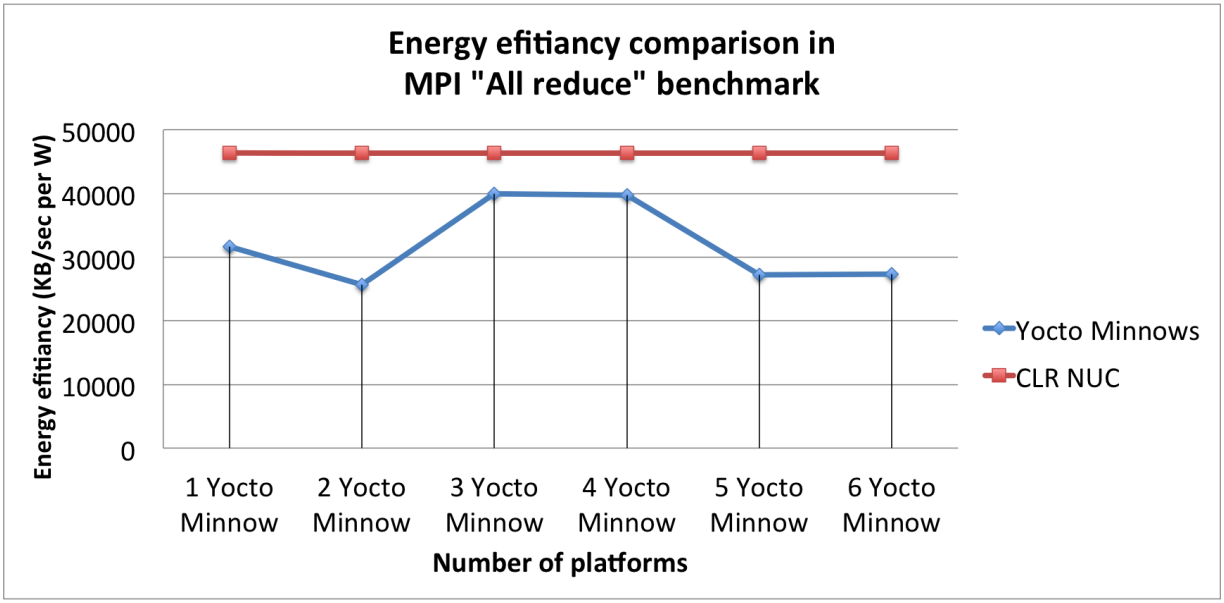


Figure 5.33: Energy efficiency comparison on MPI *All reduce* benchmark

The same behavior presented in Figure 5.33. Even the drop in power efficiency at five platforms. This is due to the fact that the power consumption with that amount of platforms is compare to the power consumed by the NUC system [56], 20 watts. In both cases is not possible to reach a higher power efficiency than the desktop system.

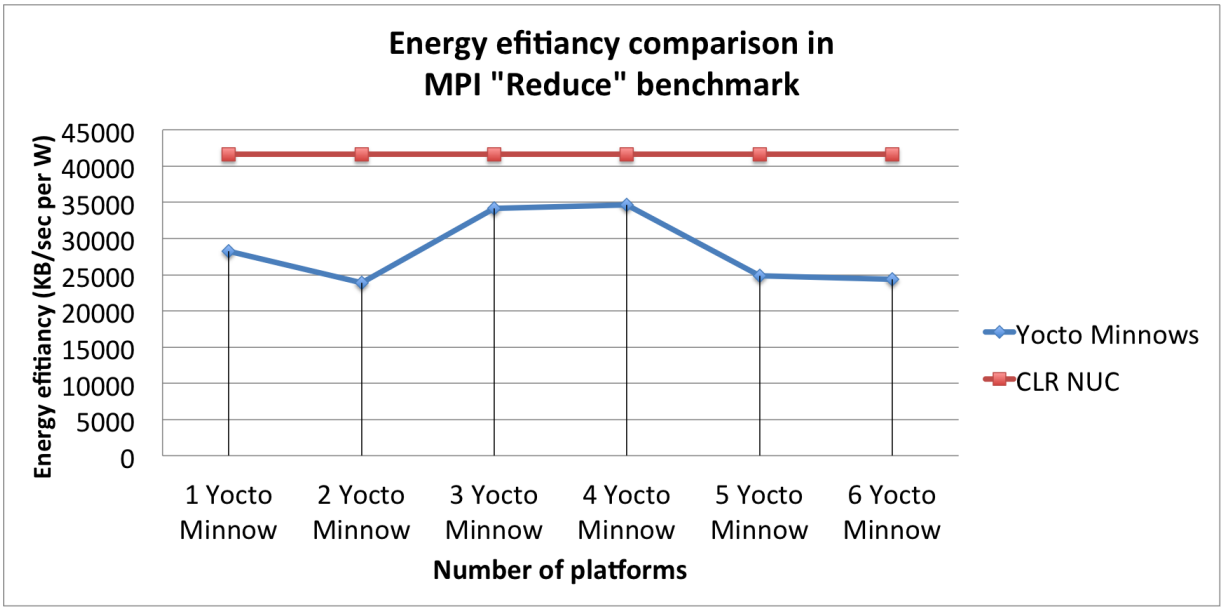


Figure 5.34: Energy efficiency comparison on MPI *Reduce* benchmark

However, in the Figure 5.35 with three or four platforms the cluster of embedded systems present a higher power efficiency than the desktop system [56]. This is possible due to the fact that performance of the *All to all* benchmark presented in the Figure 5.26is similar despite the number of platforms the cluster has. In this test the performance of even six platforms is not close to the performance that a NUC [56] system can has; however, the power consumtion is stable with this experiment.

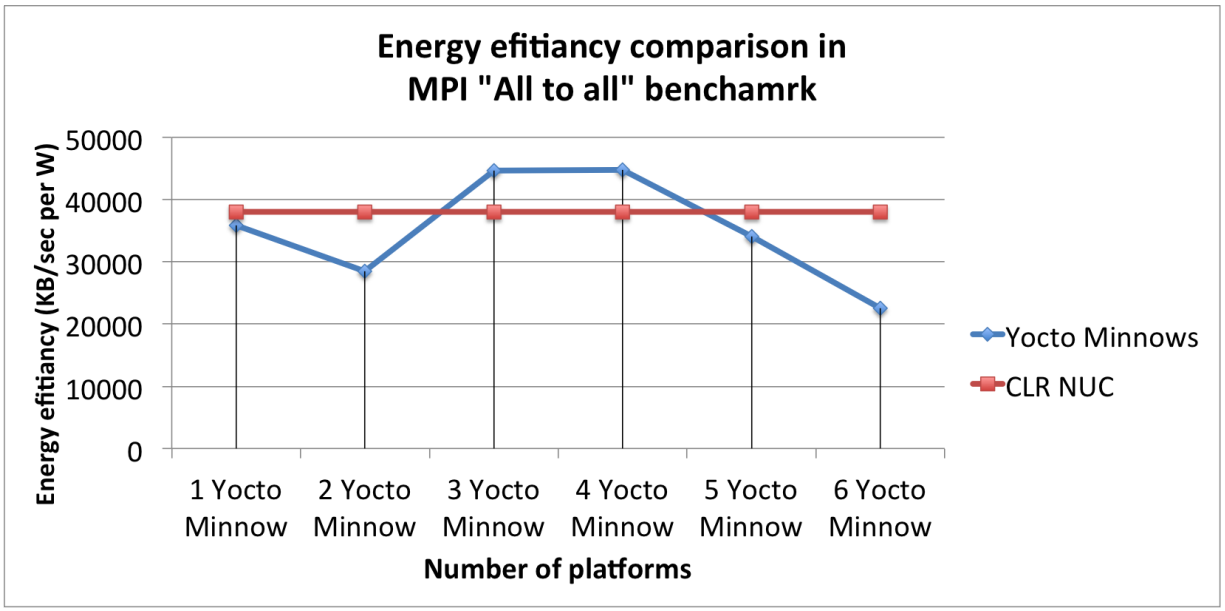


Figure 5.35: Energy efficiency comparison on MPI *All to all* benchmark

The Figure 5.36 also present the same behavior. With three or four platforms the cluster of embedded systems present a higher power efficiency than the desktop system [56]. This is possible due to the fact that performance of the *Bi directional Bandwidth* benchmark performance (Figure 5.28). At 218 MB the performance became unstable, with similar performance from four, five and six platforms. Besides that the increment in performance is constant. Even with three or four platforms the performance improvement is high. From these results it can be seen that for applications that requires speeds between 27MB and 218MB, a cluster of four, five or six embedded platforms can bring similar performance that a traditional desktop system, in terms of bi directional bandwidth.

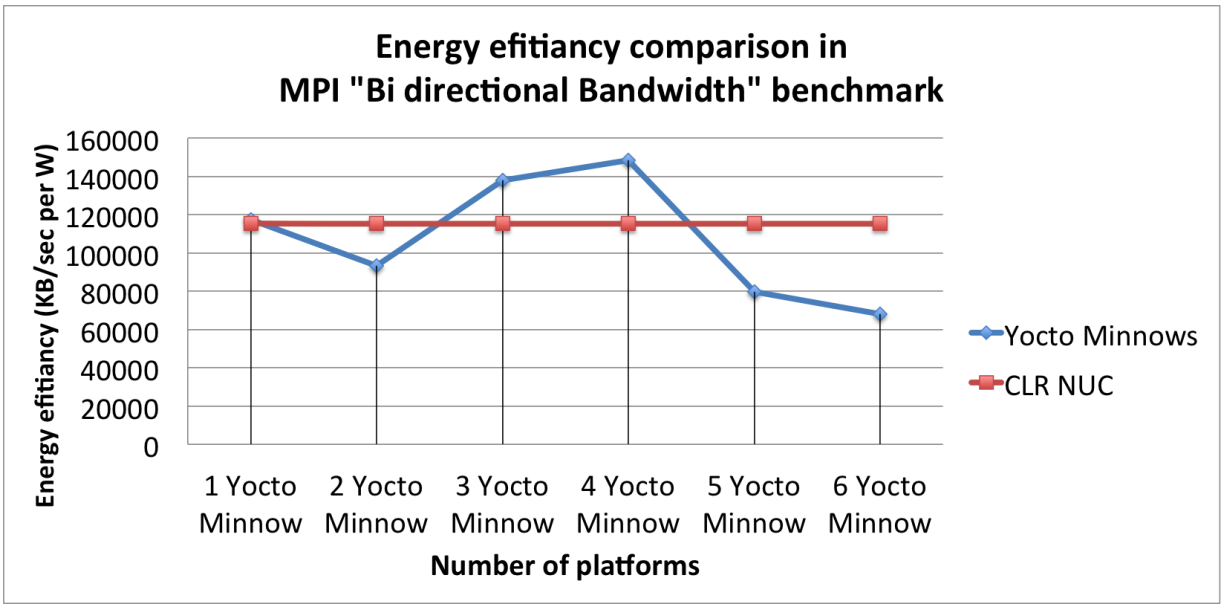


Figure 5.36: Energy efficiency comparison on MPI *Bi directional Bandwidth* benchmark

The *Bandwidth* and *Broadcast Bandwidth* benchmarks are the ones with worst power efficiency. This is due to the network latency we saw before in Figure 5.30 and Figure 5.31. Latency is really important when the program is dominated by communication, and the messages are small. This can happen when scaling to larger numbers of nodes. Despite the fact of an increment in three up to five platforms, the power efficiency of the cluster of embedded systems is low in comparison to the to the centralized system [56]

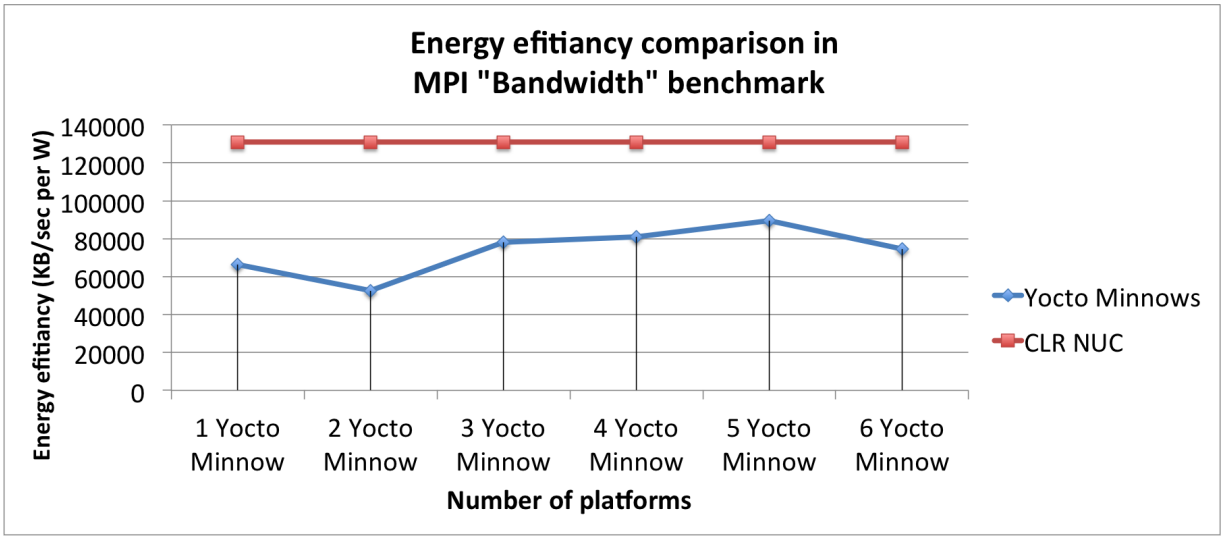


Figure 5.37: Energy efficiency comparison on MPI *Bandwidth* benchmark

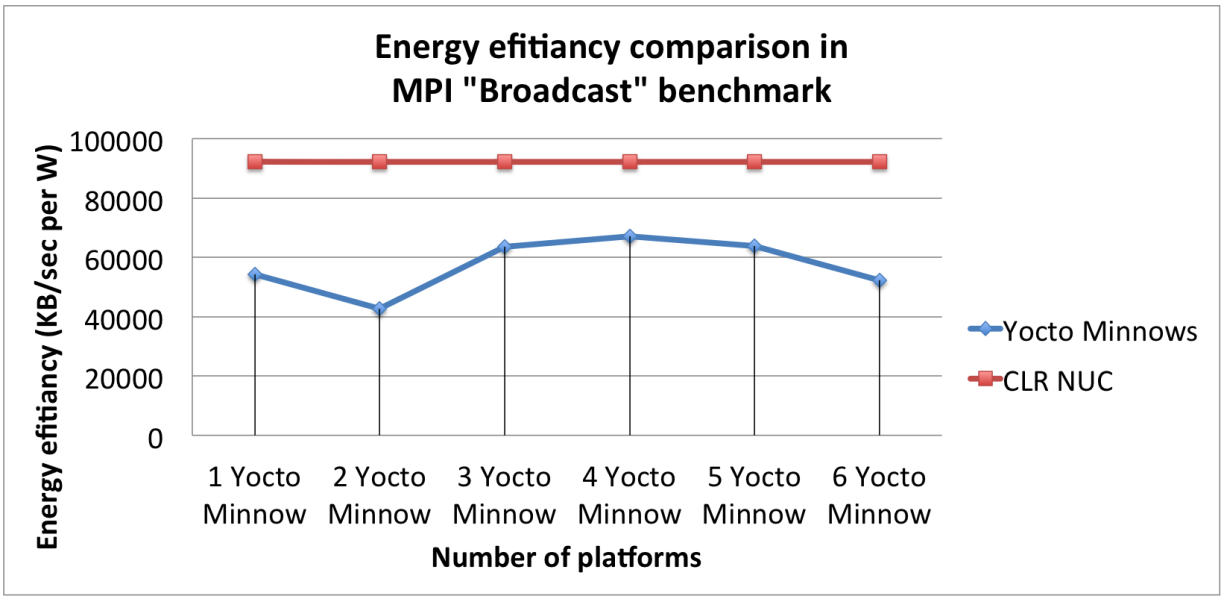


Figure 5.38: Energy efficiency comparison on MPI *Broadcast Bandwidth* benchmark

CHAPTER 6

Conclusions

6.1 Conclusions

In this thesis work it was proposed the hypothesis that a cluster of ultra-low power IoT platforms can be as computing powerful and energy efficient as a traditional computing system.

In chapter one we present a simulation to illustrate such hypothesis. It is described in Figure 1.2. At the beginning , the increment in the number of nodes in the studied network produces an increment on performance; however, it is expected to reach a maximum point at which power efficiency becomes stable and it will remain in such state up to a certain point at which it will start to decrease.

As we can see in the previous graphs this behavior was achieved as expected. At the number of platforms increases it is expected the performance benefit increase, because the amount of work to be done is distributed among different platforms, but as more are added due to the power they consume the performance gain starts to minimize. When the ideal number of platforms is exceeded, the power efficiency decrease rapidly.

In all the experiment we realize, the ideal number of platforms is always between three and four. This gave us the confidence to say that as a conclusion that the energy efficiency (using MPI Benchmarks as a reference workload) of an IoT distributed system is similar to a traditional computing system [56] with three or four nodes.

The side effect that is presented is the latency. The benchmarks that measure latency

present a lower power efficiency as a cluster of embedded platforms than the traditional desktop system.

The main objective of this work was to provide to the IoT industry a way to determine whether applications can take advantage of the communication between IoT devices to process their own data instead of sending information to data centers. Based on the results presented we can say that if the IoT system has at least four or five embedded platform with similar characteristics to the one presented here [43] is better to process their own data instead of sending it to data centers.

6.2 Future Work

As we know the IoT revolution lacks of industry standards. There are thousands of computing devices and sensors from different vendors that appear on the market every day. There are two main projects that are organizing information to establish standards for IoT communications (described in Chapter 1).

The AllSeen Alliance (before known as AllJoyn) is one of them . It is an open source project for the development of a universal software framework aimed at the interoperability among heterogeneous devices, dynamic creation of proximal networks and execution of distributed applications. The framework provides a common interface towards smart devices.

As future work will be worth to try to implement libraries for parallel and distributed computing as part of AllSeen Alliance standard. With this, the future IoT products will have the chance to determine whether applications can take advantage of the communication between IoT devices to process their own data instead of sending information to data centers.

Bibliography

[1] E. B. et al, "The internet of things: The eco-system for sustainable growth," *International Conference on Computer Systems and Applications (AICCSA)*, no. 11, 2014.

[2] X. F. et al, "Power and energy profiling of scientific applications on distributed systems," *Parallel and Distributed Processing Symposium*, no. 19, p. 34, 2005.

[3] M. Kaku, *Physics of the Future: How Science Will Shape Human Destiny and Our Daily Lives by the Year 2100.* Doubleday, 2011.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach.* Denise E. M. Penrose, 2007.

[5] N. H. A. et al, "Advanced ubiquitous computing to support smart city smart village applications," *Electrical Engineering and Informatics (ICEEI),*, pp. 720 – 725, 2015.

[6] C. A. Mack, "Keynote: Moore's law 3.0," *Microelectronics and Electron Devices (WMED), 2013 IEEE Workshop on*, pp. xiii–xiii, 2013.

[7] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach.* Prentice Hall, 2006.

[8] M. Weiser, "Some computer science issues in ubiquitous computing," pp. 75 – 84, 1993. [Online]. Available: http://www.ubiq.com/hypertext/weiser/UbiCACM.html

[9] A. Bahga and V. Madisetti, *Internet of things: a hands-on approach.* VPT, 2014.

[10] T. Myerson, "New open connectivity foundation will further innovation of the internet of things." [On-

line]. Available: https://blogs.windows.com/windowsexperience/2016/02/19/new-open-connectivity-foundation-will-further-innovation-of-the-internet-of-things/

[11] M. Villari, "Alljoyn lambda: An architecture for the management of smart environments in iot," *Smart Computing Workshops (SMARTCOMP Workshops)*, pp. 9 – 14, 2014.

[12] Y.-S. C. et al, "Integrated ddos attack defense infrastructure for effective attack prevention," *Information Technology Convergence and Services (ITCS),*, pp. 1 –8, 2010.

[13] L. D. et al, "Intelligent agriculture greenhouse environment monitoring system based on iot technology," *Intelligent Transportation, Big Data and Smart City (ICITBS)*, pp. 487 – 490, 2015.

[14] Z. D. et al, "Design and implementation of safety management system for oil depot based on internet of things," *Green Computing and Communications (GreenCom)*, pp. 449 – 252, 2012.

[15] W. C. L. et al, "Autonomous industrial tank floor inspection robot," *International Conference on Signal and Image Processing Applications (ICSIPA)*, pp. 473 – 475, 2015.

[16] M. Finnegan, "Boeing 787s to create half a terabyte of data per flight, says virgin atlantic." [Online]. Available: http://www.computerworlduk.com/news/data/boeing-787s-create-half-terabyte-of-data-per-flight-says-virgin-atlantic-3433595/

[17] J. M. et al, "An analytical framework for estimating scale-out and scale-up power efficiency of heterogeneous manycores," *Computers, IEEE Transactions on*, pp. 367 – 381, 2016.

[18] D. H. Woo and H.-H. S. Lee, "Extending amdahl's law for energy-efficient computing in the many-core era," *Computer*, pp. 24 –31, 2008.

[19] J. Siegenthaler, *Heating with Renewable Energy, 1st Edition.* Cengage Learning, 2014.

[20] M. Saldana, "Mpi as an abstraction for software-hardware interaction for hprcs," *High-Performance Reconfigurable Computing Technology and Applications*, pp. 1 – 10, 2008.

[21] A. A. et al, "Lmpi: Mpi for heterogeneous embedded distributed systems," *Parallel and Distributed Systems*, pp. 8 pp –, 2006.

[22] T. P. McMahon and A. Skjellum, "empi/empich: embedding mpi," *MPI Developer's Conference*, pp. 180 – 184, 1996.

[23] J. Liu, "Mpi for embedded systems: A case study." [Online]. Available: www.ece.uci.edu/~jinfengl/course/ECE238F02.pdf

[24] S. Y. P. et al, "A multithreaded message-passing system for high performance distributed computing applications," pp. 258–265, 1998.

[25] W. Gropp and E. Lusk, "Reproducible measurements of mpi performance characteristics," *Proceedings of the PVM/MPI Users' Group Meeting (LNCS 1697)*, pp. 11–18, 1999.

[26] J. A. R.-G. et al, "A pthreads-based mpi-1 implementation for mmu-less machines," *Reconfigurable Computing and FPGAs*, pp. 277 – 282, 2008.

[27] H. T. et al, "Program transformation and runtime support for threaded mpi execution on shared memory machines," *ACM Transactions on Programming Languages and Systems*, pp. 673–700, 2000.

[28] E. D. Demaine, "A threads-only mpi implementation for the development of parallel programs," *International Symposium on High Performance Computing Systems*, pp. 153–163, 1997.

[29] M. Saldana and P. Chow, "Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas," *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 329–334, 2006.

[30] J. W. et al, "A reconfigurable cluster-on-chip architecture with mpi communication layer," *Proceedings of the 14th Annual IEEE Symposium on 'Field-Programmable Custom Computing Machines (FCCM'06)*, pp. 1 –6, 2006.

[31] M. Levy and S. Brehmer, "Multicore communications api working group." [Online]. Available: http://www.multicore-association.org/workgroup/mcapi.php

[32] W. W. et al, "Multiprocessor system-on-chip (mpsoc) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[33] L. M. et al, "Multicore communications api (mcapi) implementation on an fpga multiprocessor," pp. 286–293, 2011.

[34] E. F. e. a. Weglarz, "Minimizing energy consumption for high-performance processing," *Design Automation Conference, 2002. Proceedings of ASP-DAC*, pp. 199 – 204, 2002.

[35] V. Rodriguez and M. de Alba, "Embedded distributed systems: A case of study." [Online]. Available: https://www.overleaf.com/articles/embedded-distributed-systems-a-case-of-study-with-clear-linux-projectfor-intel-r-architecture/wpcgjhykxdmj/viewer.pdf

[36] V. Rodriguez and M. R. de Alba, "Embedded distributed systems: A case of study with clear linux." [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/ELC_Embedded_Distributed.pptx_.pdf

[37] D. S. et al, "Effective throughput: a unified benchmark for pilot-aided ofdm/sdma wireless communication systems," vol. 3, pp. 1603–1613 vol.3, 2003.

[38] F. B. Shaikh and S. Haider, "Security threats in cloud computing," *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pp. 214–219, 2011.

[39] S. Rifenbark, "Yocto project development manual." [Online]. Available: http://www.yoctoproject.org/docs/1.7/dev-manual/dev-manual.pdf

[40] A. L. et al, "Framework for industrial embedded system product development and management," pp. 1–6, 2013.

[41] G. C. et al, *Distributed systems concepts and design*, 5th ed. Pearson Education Limited, 2013.

[42] "Mpi: A message-passing interface standard version 3.1." [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[43] "Minnowboard max." [Online]. Available: http://elinux.org/Minnowboard:MinnowMax

[44] "Fedora project." [Online]. Available: https://getfedora.org/

[45] A. C. et al, "New techniques for collective communications in clusters: a case study with mpi," pp. 185–192, 2001.

[46] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately measuring collective operations at massive scale," pp. 1–8, 2008.

[47] D. A. Grove and P. D. Coddington, "Communication benchmarking and performance modelling of mpi programs on cluster computers," pp. 249–, 2004.

[48] P. J. M. et al, "The mpbench report." [Online]. Available: cl.cs.utk.edu/~mucci/latest/pubs/mpbench.pdf

[49] W. Kendall, "Mpi tutorial." [Online]. Available: http://mpitutorial.com/tutorials/

[50] P. Pacheco, *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[51] "Intel atom processor e3825." [Online]. Available: http://ark.intel.com/products/78474/Intel-Atom-Processor-E3825-1M-Cache-1_33-GHz

[52] "Clear linux project for intel® architecture." [Online]. Available: https://clearlinux.org/

[53] "Minnowboard max firmwware." [Online]. Available: https://firmware.intel.com/projects/minnowboardmax

[54] Z. B. et al, "Dptsv: A dynamic priority task scheduling strategy for tss deadlock based on value evaluation," *China Communications*, vol. 13, no. 1, pp. 161–175, 2016.

[55] S. S. et al, "Multi-sensor fusion for robust autonomous flight in indoor and outdoor environments with a rotorcraft mav," pp. 4974–4981, 2014.

[56] "Intel® nuc d54250wyk." [Online]. Available: http://ark.intel.com/products/76977/Intel-NUC-Kit-D54250WYK

[57] "Intel® nuc d54250wyk operating systems." [Online]. Available: http://www.intel.com/content/www/us/en/support/boards-and-kits/000005471.html?wapkw=-falske

[58] M. Larabel, "How ubuntu 16.04 is performing compared to five other linux distributions," 2016. [Online]. Available: http://www.phoronix.com/scan.php?page=article&item=2016-feb-6way&num=1

[59] B. Mitchell, "Introduction to computer network topology," 2005. [Online]. Available: http://compnetworking.about.com/od/networkdesign/a/topologies.htm

[60] D. Bird and M. Hardwood, *Network+ Training Guide*. Paul Boger, 2003.