Embedded Distributed Systems: A Case of Study with Clear Linux Project for Intel® Architecture

Author: Víctor Rodríguez Bahena / Co-author: Marcos de Alba

November 28 2014

1 Abstract

The rise of IoT interconnected objects will lead to an explosion in the volume of data that is collected, transmitted and processed. This explosion in volume will require novel methods for this transmission and processing.Power consumption and performance is one of the main design constraint for these systems. If current trends continue, future petaflop systems will require 100 megawatts of power. To address this problem the trend is towards the autonomous and responsible behavior of resources This demo shows how a network of ultra-low-voltage microprocessors platforms (Intel R AtomTM Processor E3815-Minnow-Max) can process their own data (running real HPC workloads) without the need of an external HPC system. This paper shows the impact of using a custom OS for x86 architecture in a embedded distributed system.

2 Introduction

Computer technology has made incredible progress in the roughly 60 years since the first general-purpose electronic computer was created. Today, less than 500 USD will purchase a personal computer that has more performance, more main memory, and more disk storage than a computer bought in 1985 for 1 million dollars. This rapid improvement has come both from advances in the technology used to build computers and from innovation in computer design. [1]

As we have seen it was around the years 2003 to 2005 that a dramatic change seized the semiconductor industry and the manufactures of processors. The increasing of computing performance in processors, based on simply screwing up the clock frequency, could not longer be hold ed. Scaling of the technology processes, leading to smaller channel lengths and shorter switching times in the devices, and measures like instruction-level-parallelism and out-of-order processing, leading to high fill rates in the processor pipelines, were the guarantors to meet Moore's law.[2]

The answer of the industry to that development, in order to still meet Moore's law, was the shifting to real parallelism by doubling the number of processors on one chip die. This was the birth of the multi-core area. The benefits of multi-core computing, to meet Moore's law and to limit the power density at the same time, at least at the moment this statement holds, are also the reason that parallel computing based on multi-core processors is underway to capture more and more also the world of embedded processing.[3]

Where do we find these task parallelism in embedded systems? A good example are automotive applications Multi-core technology in combination with a broadband efficient network system offers the possibility to save components, too, by migrating functionality that is now distributed among a quite large number of compute devices to fewer cores.

The purpose of this paper is to present a case study to explore the benefits of using a customized OS (against the community-supported OS: Fedora) on a distributed embedded system. We will use MPI as the communication engine for our benchmarks and experiments.

3 Theoretical Framework

In recent years several mature techniques for high level abstractions for inter-processor communication are available, such as Message Passing Interfaces (MPI), the problem is that these abstraction layers require extensive system resources with comprehensive operating systems support, which may not be available to an embedded platform.

Recent researches [4] [6] [5] describe proof-of-concept MPI implementations targeting embedded systems, showing an increasing interest in the topic. These implementations have a varying degree of functionality and requirements. These papers also discuss different ways to address the limitations found in typical embedded systems. For example, in the eMPl/eMPICH project [5] the main focus is to port MPICH to an embedded platform and reduce its memory footprint by removing some MPI functions. Azequia-MPI [6] is an MPI implementation that uses threads instead of processes making MPI applications more lightweight, Although, it requires an operating system that supports threads, which in embedded systems it is not always available.

In recent years there has been some studies in this field. One of the firsts is the adaption of the MPI protocol for embedded systems , LMPI [7] (Light Message Passing Interface). The noble 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 with a server. LMPI servers support full capability of MPI and can be implemented using pre-existing MPI implementation. Although LMPI is dedicated to embedded systems, to demonstrate the benefits of LMPI and show some initial results, they built LMPI server using MPICH on a non-embedded system. LMPI client consumes far less computation and communication bandwidth than typical implementations of MPI, such as MPICH. As a result, LMPI client is suitable for embedded systems with limited computation power and memory. They 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.

All these research always talk about the lack of an operating system for Distributed System, However there are some works related to this area[8]. Those are the distributed operating systems. The architecture and design of a distributed operating system must realize both individual node and global system goals. Architecture and design must be approached in a manner consistent with separating policy and mechanism. In doing so, a distributed operating system attempts to provide an efficient and reliable distributed computing framework allowing for an absolute minimal user awareness of the underlying command and control efforts

With these techniques, distributed programming can be made much more efficient. However, very few researchers have studied high level distributed programming in embedded systems

4 Objective

The main objective of this work will be to prove that a distributed embedded system (Intel® AtomTM Processor E3825) running real HPC workloads (MPI benchmarks) can be improved by the use of a customized operating system

5 Justification

The need of more complex and smart applications (they must adapt their performance as well as power) has risen the bar to create distributed systems based on parallel embedded platforms.

By definition: A distributed system consists of a collection of autonomous computers, connected through a network and distribution middle-ware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

Advantages:

1. **Partioning Workload**: By partitioning the workload onto multiple processors, each processor is now responsible for only a fraction of the workload. The processors can now afford to slow down by dynamic voltage scaling (DVS) to run at more power-efficient states, and the degraded performance on each processor can still contribute to an increased system-wide performance by the increased parallelism.

2. **Heterogeneous HW**: Another advantage with a distributed scheme is that heterogeneous hardware such as DSP and other accelerators can further improve power efficiency of various stages of the computation through specialisation.

Disadvantages:

1. **Network**: Despite the fact the distributed systems may have many attractive properties, they pay a higher price for message-passing communications. Each node now must handle not only communication with the external world, but also extra communication on the internal network. As a result, even if the actual data payload is not large on an absolute scale, the communication appears very expensive and does not scale to a few more nodes

2. **Lack of optimised OS**: A typical embedded system often does not contain an operating system. Crafting distributed programs on such a bare-bone platform is extremely difficult and error-prone. Although many higher-level abstractions such as Message Passing Interfaces (MPI) have been proposed to facilitate distributed programming, these abstraction layers require extensive system resources with comprehensive operating systems support, which may not be available to an embedded platform

However in recent years we have seen an emergence of a new class of full-fledged embedded systems (they are fully loaded with sufficient system resources as well as networking and other peripheral devices, and a complete version of the operating system with network support) In addition, they are typically designed with power-management technology in order to extend the battery life

With these gaps closed there might be a chance to merge the parallel and distributed paradigms on the embedded world. A merging point of technologies from different domains often inspires technology innovations in new domains.

6 Development

According to these in consideration there are multiple scenarios to test the capability of an embedded distributed system:

- Compare an Embedded system with generic SW (Linux base OS (Fedora/Ubuntu/Debian) and generic MPI protocol (MPICH)) against a regular development system (with the same OS and MPI toolS)
- Compare an Embedded system with a distributed operating system against the same embedded system with custom SW (Linux from scratch system)
- Compare an Embedded system with a distributed operating system against the same embedded system with custom SW (Linux from scratch system and MPI for embedded (LMPI)) in order to check the gap in the multiple systems

For this report we will execute the experiment of the second scenario, due to the fact that we have already done the study of the first scenario. In that case we realize that despite the fact that the minnow Max ran 8 times slower than the regular development system (NUC Haswell system) the Minnow Max was more stable and with less drops in performance.

The platform we use for our experiment is the Intel® AtomTM Processor E3825. Their main characteristics are described on 1. The main limitation will be the number of Cores that we have. This is me minimal number of cores we could have to run parallel applications. [9]

The operating system we will use is the Fedora 19 system, the description of the system is listed on the fedora project site home page (http://fedoraproject.org)

The benchmark we will use to measure the performance is MPIbench. This is a program to measure the performance of some critical MPI functions. By critical it means that the behavior of these functions can dominate the run time of a distributed application. MPBench has now been integrated into LLCbench (Low Level Characterisation Benchmarks)

The MPIfunctions that it stress are:

Procesor Number	E3825
#Cores	2
#Threads	2
Clock SPeed	1.33GHz
L2 Cache	1MB
Instruction Set	64 bits

Table 1: Minnowboard CPU characteristics

- MPI_Send/MPI_Recv Bandwidth (Kb/second vs. bytes)
- MPI_Send/MPI_Recv Application latency or Gap time (us vs. bytes)
- MPI_Send/MPI_Recv Roundtrip or 2 * Latency (trns/second vs. bytes)
- MPI_Send/MPI_Recv() BidirectionalBandwidth (Kb/second vs. bytes)
- MPI_Bcast broadcast (Kb/second vs. bytes)
- MPI_Reduce reduction (sum) (Kb/second vs. bytes)
- MPI_AllReduce reduction (sum) (Kb/second vs. bytes)
- MPI_Alltoall Each process sends to every other process (Kb/sec vs. bytes)

7 Results

The results after the execution of the benchmarks are described on the Appendix section (for minnow board and then for development board):

- MPI_Send/MPI_Recv Bandwidth (Kb/second vs. bytes)
- MPI_Send/MPI_Recv Application latency or Gap time (us vs. bytes)
- MPI_Send/MPI_Recv Roundtrip or 2 * Latency (trns/second vs. bytes)
- MPI_Send/MPI_Recv() BidirectionalBandwidth (Kb/second vs. bytes)
- MPI_Bcast broadcast (Kb/second vs. bytes)
- MPI_Reduce reduction (sum) (Kb/second vs. bytes)
- MPI_AllReduce reduction (sum) (Kb/second vs. bytes)
- MPI_Alltoall Each process sends to every other process (Kb/sec vs. bytes)

As seen on the results presented on the AllReduce (MPI_Allreduce combines values from all processes and distributes the result back to all processes) graphs in both OS's (either Clear LInux or Fedora) the drop of speed is extremely fast until reach a minimal point of stability with packages grater than 1.04e+06 Bytes. if we look the graph we can see that Clear Linux can sustain a better quality of transaction (much more stable and with less drops). The dramatic drop after the increment of 1.04e+06 Bytes is not reflected on the Clear Linux system. On the Clear Linux the speed is the same until the size of the packages reach the 3.3e+07 Bytes. A similar behavior occurs on the unidirectional/bidirectional and broadcast MPI bandwith.

Analyzing the Bidirectional Bandwidth algorithm (as an example of the root cause of this behaivor):

```
if (am_i_the_master()){
        TIMER_START;
        for (i=0; i<cnt; i++){
                mp_irecv(dest_rank, 2, destbuf, bytes, &requestarray[1]);
                mp_isend(dest_rank, 1, sendbuf, bytes, &requestarray[0]);
                MPI_Waitall(2, requestarray, statusarray);
        }

else if (am_i_the_slave()){
        for (i=0; i<cnt; i++) {
                mp_irecv(source_rank, 1, destbuf, bytes, &requestarray[0]);
                mp_isend(source_rank, 2, sendbuf, bytes, &requestarray[1]);
                MPI_Waitall(2, requestarray, statusarray);
        }
```

We can see that at the end they use MPI_Waitall. MPI_Waitall blocks until all communication operations associated with active handles in the list complete, and returns the status of all these operations. In the case of Clear Linux there is a reduced number of process running in background (No Xserver/Xorg/etc); besides there is an implementation of systemd. This helps in lack of process trying to gain control of the system and memory at the same time, which will be reflected every time the MPI_Waitall arrives.

In case of the All to all experiment (Each process sends to every other process). we don't see a huge gain on the performance. MPI_Alltoall 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. The operation of this routine can be represented as follows:

Algorithm:

```
  MPI_Comm_size(comm, &n);
  for (i = 0, i < n; i++)
      MPI_Send(sendbuf + i * sendcount * extent(sendtype),\
          sendcount, sendtype, i,..., comm);
  for (i = 0, i < n; i++)
      MPI_Recv(recvbuf + i * recvcount *extent(recvtype), \
          recvcount, recvtype, i, ..., comm);
```

As we can see here there is no chance to other process to compete for memory or CPU resources.

The latency and round trip benchmarks show a similar performance all the time despite the Operating System running . For MPI the definition of latency is the time to launch a message in the network's buffer:

Algorithm:

```
 if (am_i_the_master())
   {
    TIMER_START;
    for (i=0; i<cnt; i++)
    {
        if (flush & FLUSH_BETWEEN_ITERATIONS)
            flushall(1);
        mp_send(dest_rank, 1, sendbuf, bytes);
    }
    TIMER_STOP;
    mp_recv(dest_rank, 2, destbuf, 4);
    total = TIMER_ELAPSED;
    total -= calibrate_cache_flush(cnt);
    return(total/(double)cnt);
}
```

The low performance (either in Clear Linux or Fedora) is an expected behavior. Embedded systems have traditionally been much more sensitive to both the interrupt latency and Central Processing Unit (CPU) overhead involved in servicing interrupts as compared to conventional Personal Computers (PC).

8 Conclusion

This case of study demonstrate not only the capability of an embedded platform (Intel® AtomTM Processor E3825 - Minnowboard) to execute a heavy MPI workload , but the capability for the Clear Linux system to maintain a better performance (even with high volume packages) than a none customized OS. After this case of study we demonstrate that an embedded system with a customized OS might be useful for HPC applications, however the latency is a major problem that require HW reconfiguration.

Future work will will be to start the measurement of power consumption. This might be a key characteristic that make the embedded systems a possibility to establish parallel/distributed programming paradigms to facilitate the development of distributed embedded applications.

9 References

References

[1] Hennessy, J., & Patterson, D. (2007). Computer architecture a quantitative approach (4th ed.). Amsterdam: Elsevier/Morgan Kaufmann.

[2] Amdahl, G. (n.d.). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18-20), AFIPS Press, Reston, Va., 1967, pp. 483-485, when Dr. Amda. IEEE Solid-State Circuits Newsletter, 19-20.

[3] Mattson, T., & Sanders, B. (2005). Patterns for parallel programming. Boston: Addison-Wesley.

[4] M. Saldana, A. Patel, C. Madill, N. D., A. Wang, A. Putnam, R . Wittig, and P. Chow, “MPI as an abstraction for software-hardware interaction for HPRCs,” in International Workshop on High-Performance Reconfig-F urable Computing Technology and Applications , Nov. 2008, pp. 1–10.

[5] T. P. McMahon and A. Skjellum, “eMPI/eMPICH: Embedding MPI” in MPI Developers Conference , 1996, pp. 180–184.

[6] J. Rico-Gallego, J. Alvarez-Llorente, F. Perogil-Duqu e, P. Antunez-Gomez, and J. Diaz-Martin, “A Pthreads-Based MPI-1 Implementation for MMU-Less Machines,” in International Conference on Reconfigurable Computing and FPGAs , Dec. 2008, pp. 277–282.

[7] J. Liu, MPI for Embedded Systems: A Case Study. 2003

[8] Sinha, P. (1997). Distributed operating systems: Concepts and design. New York: IEEE Press.

[9] Intel® AtomTM Processor E3825 SPECIFICATIONS. (n.d.). Retrieved from ARK Intel:`http://ark.intel.com/products/78474/Intel-Atom-Processor-E3825-1M-Cache-1_33-GHz`

10 Appendix

Figure 1: All reduce minnowboard

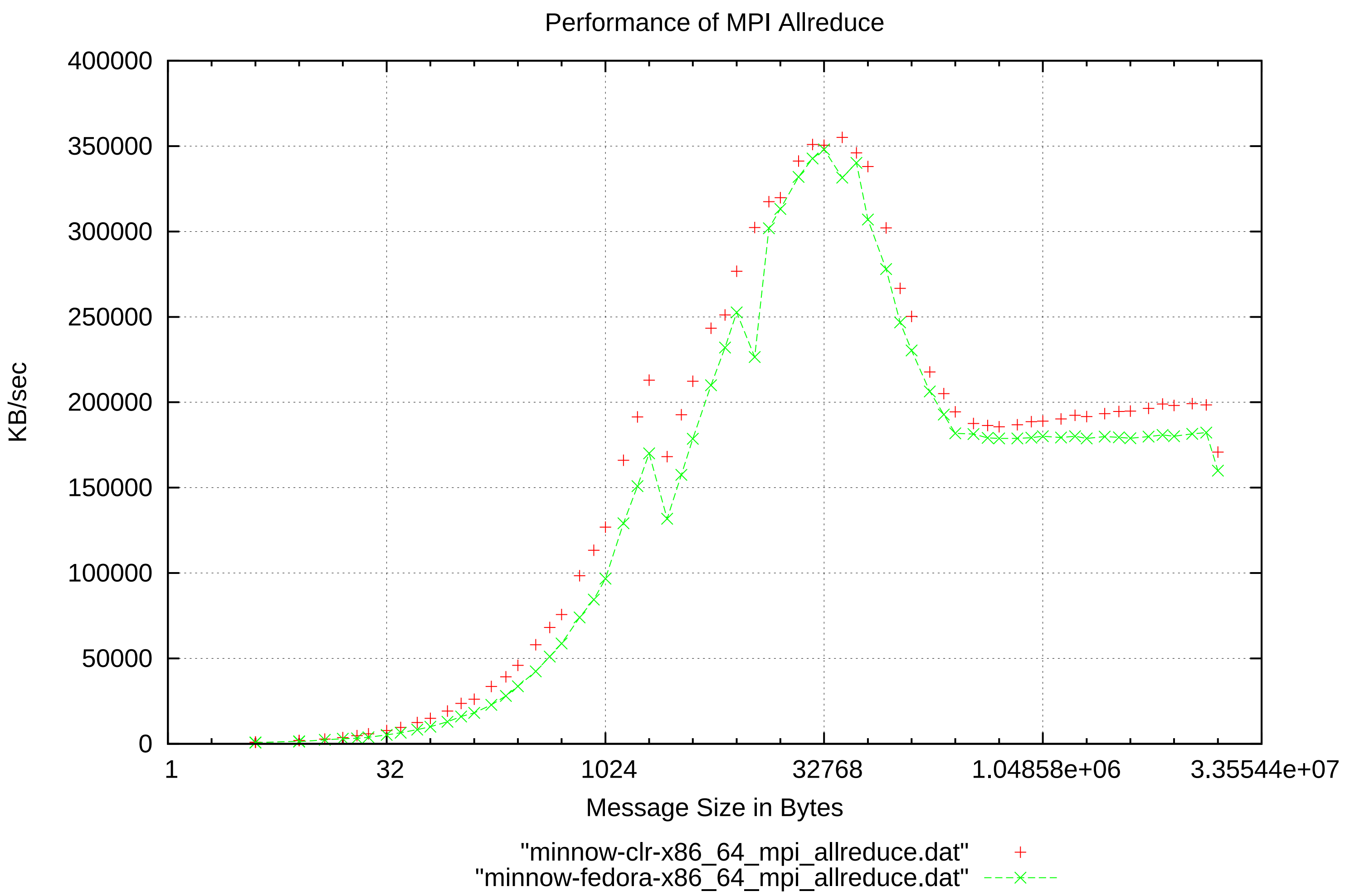


Figure 2: All to all minnowboard

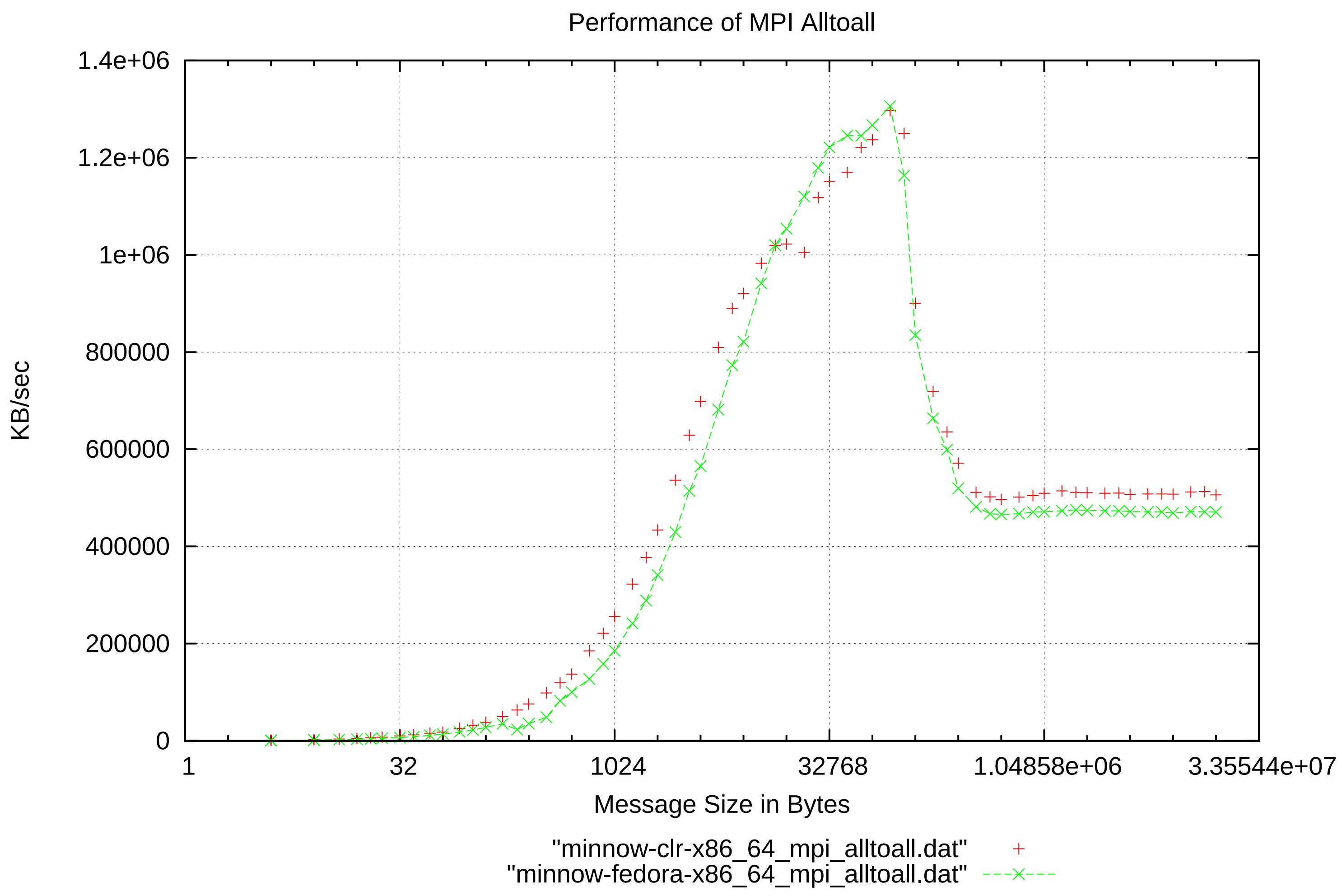


Figure 3: Bandwith on minnowboard

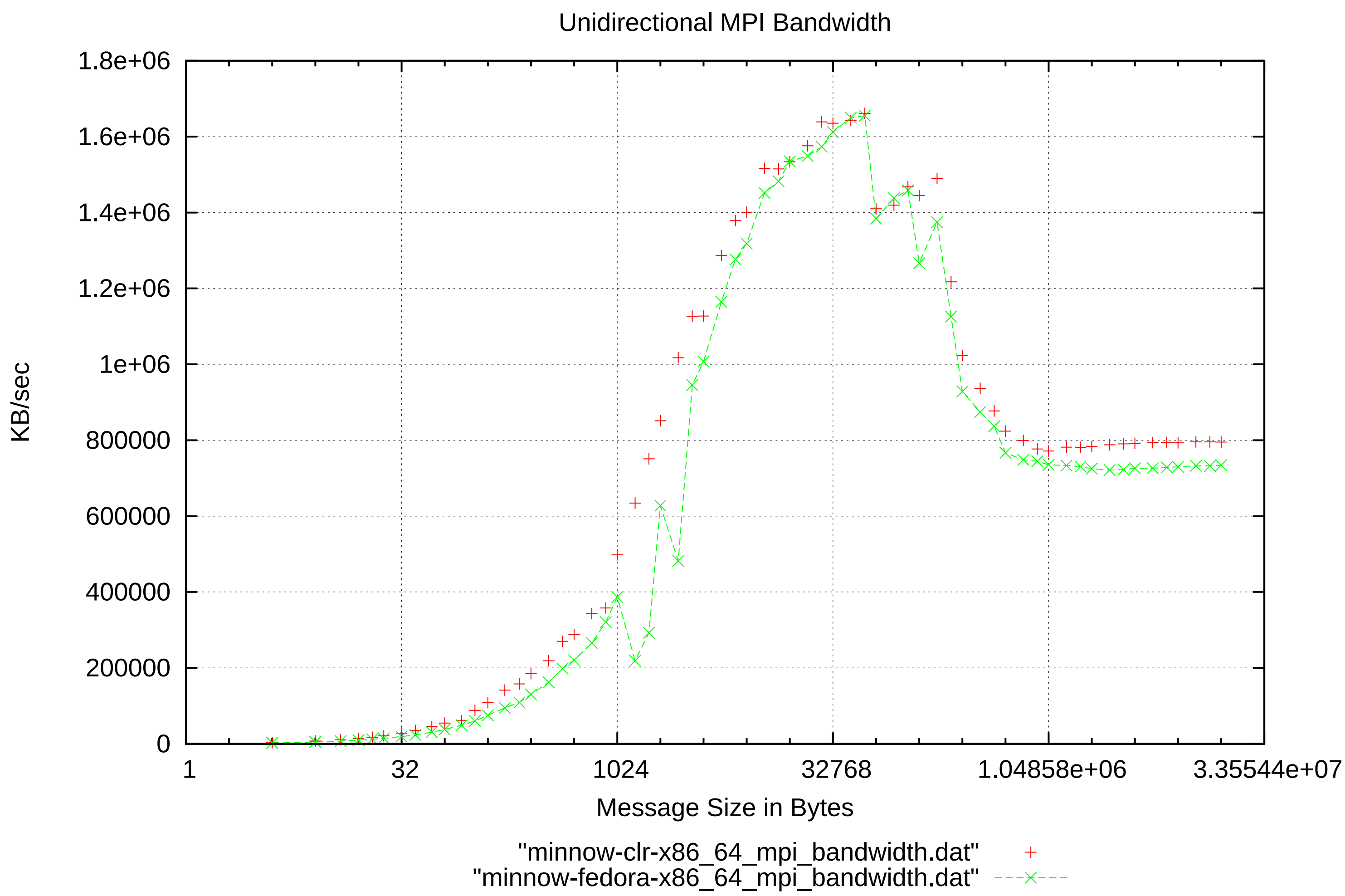


Figure 4: Bidirectional Bandwidth minnowboard

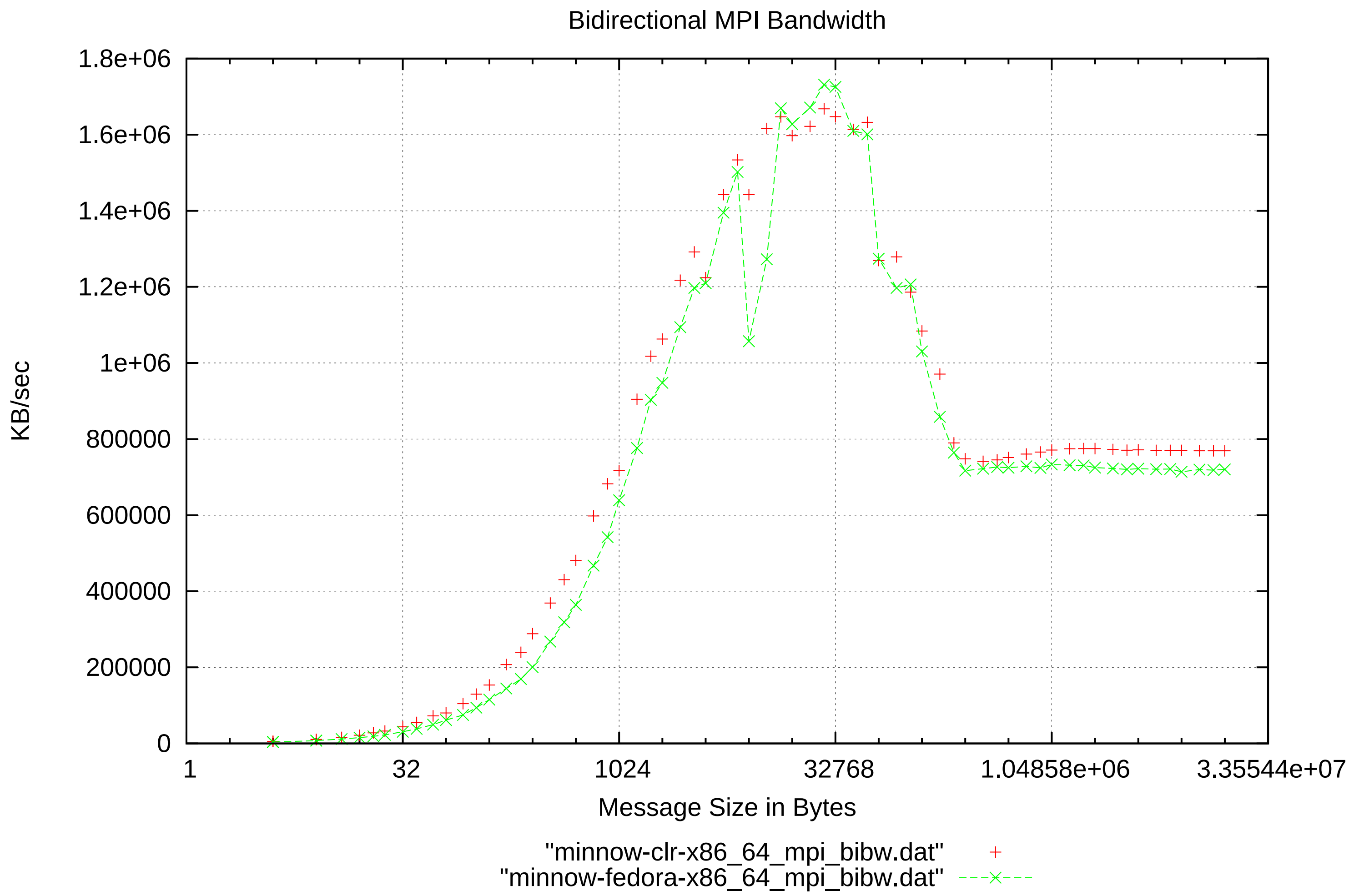


Figure 5: Broadcast minnowboard

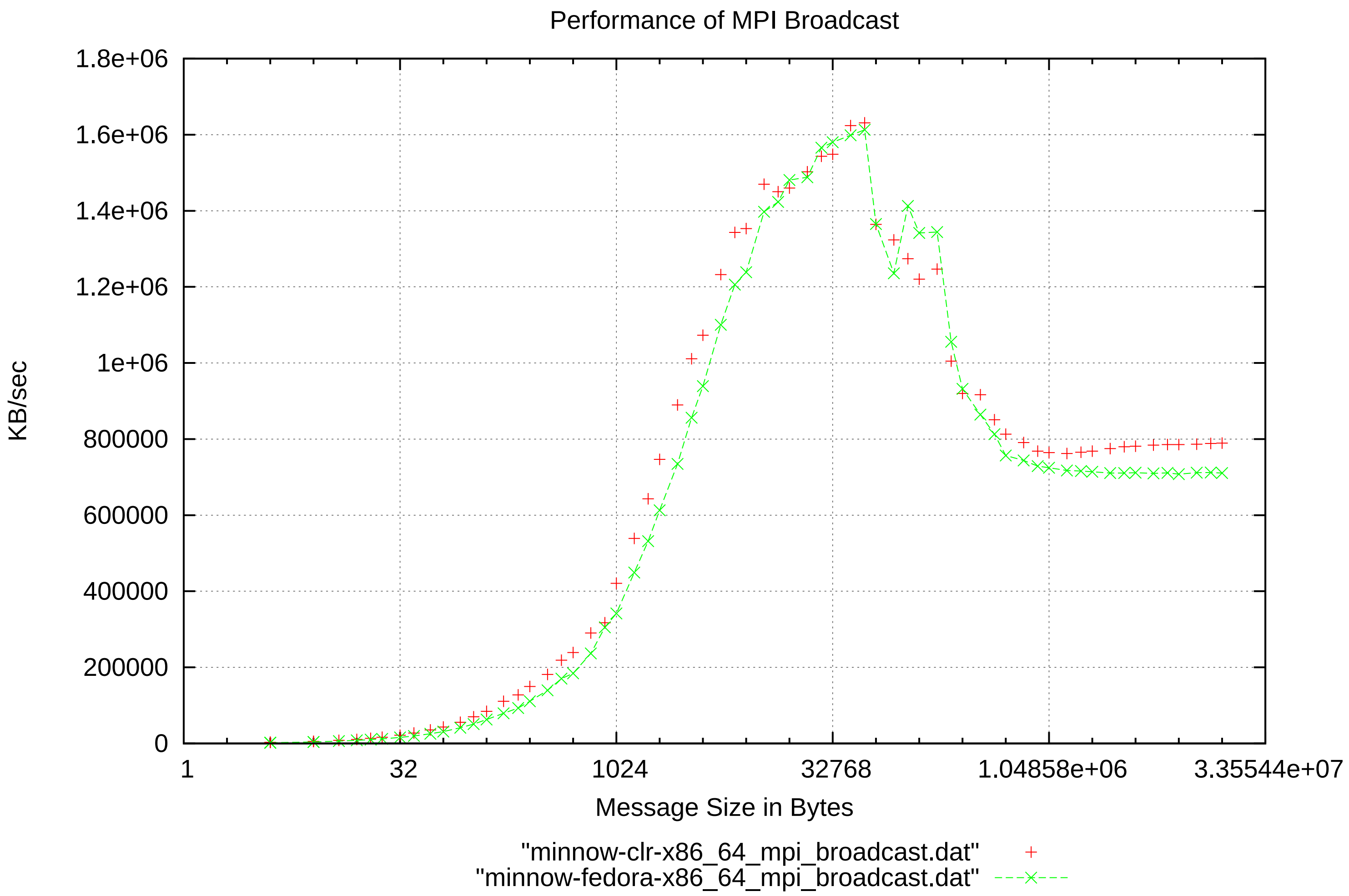


Figure 6: Application latency or Gap time on minnowboard

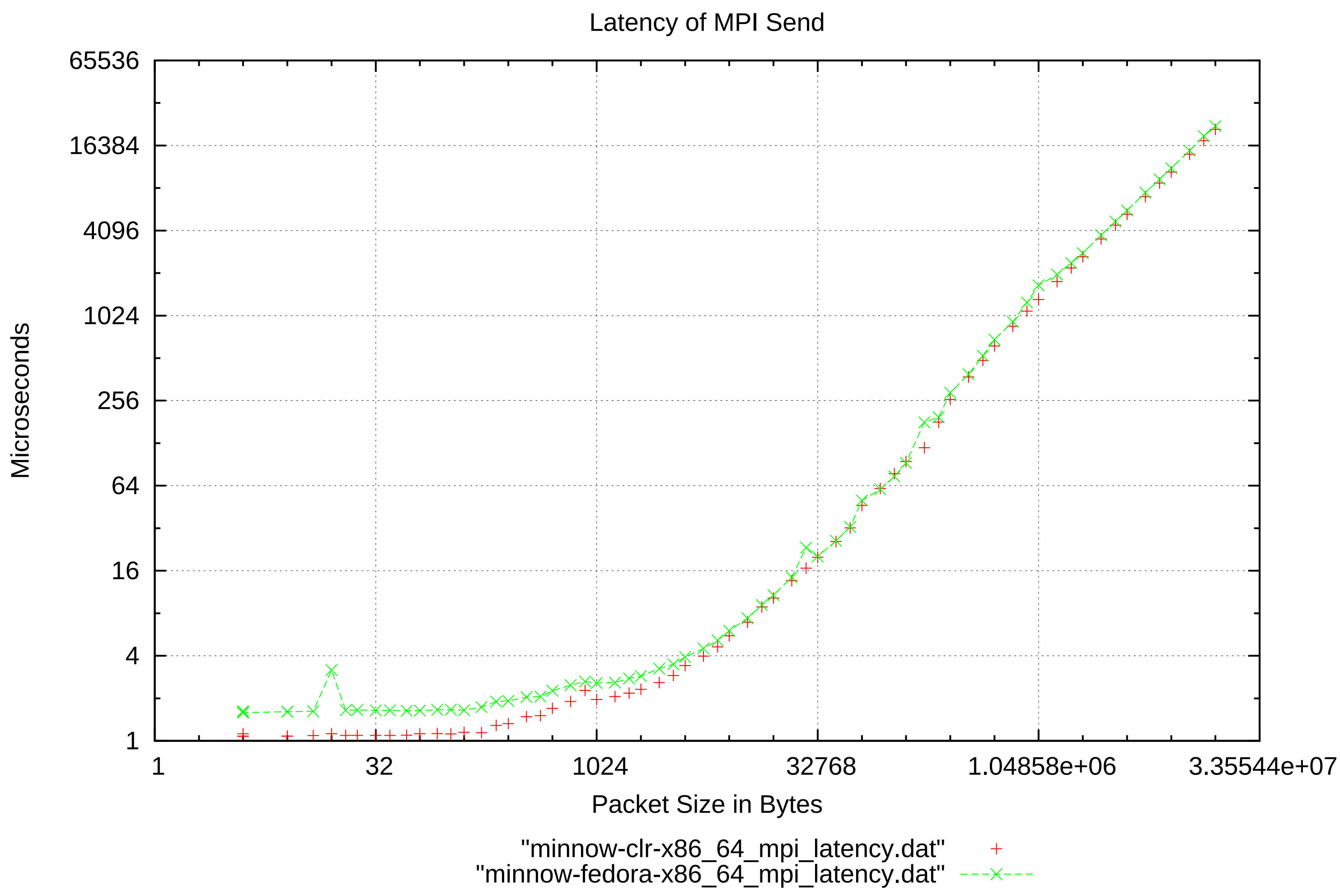
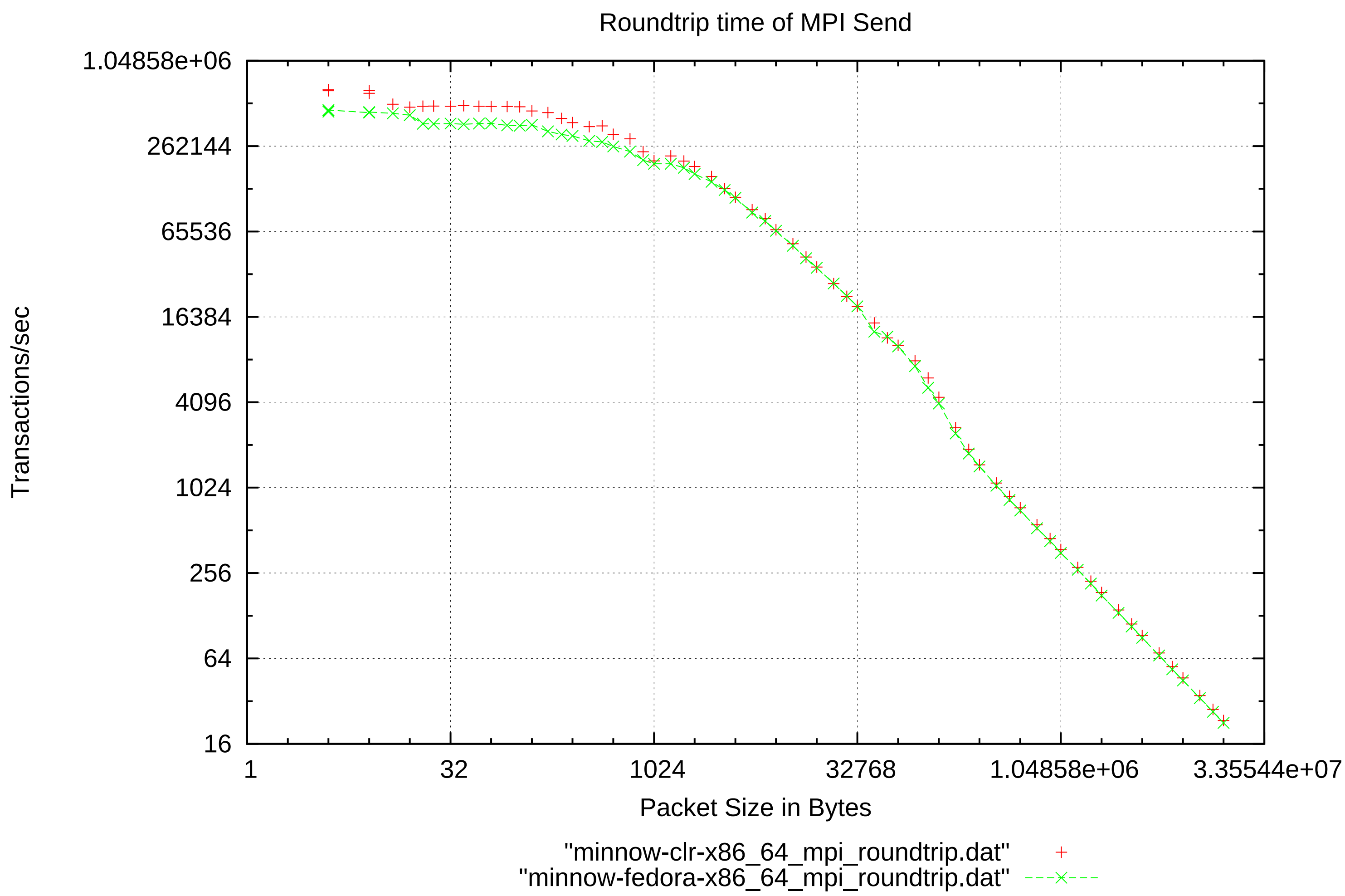


Figure 7: Roundtrip or 2 * Latency on minnowboard


Roundtrip time of MPI Send
Transactions/sec
1.04858e+06
1.04858e+06
3.35544e+07
Packet Size in Bytes
"minnow-clr-x86_64_mpi_roundtrip.dat"
"minnow-fedora-x86_64_mpi_roundtrip.dat"