Embedded Distributed Systems: A Case of Study

Víctor Rodríguez Bahena
PhD Maria Guadalupe Sanchez Cervantes

November 28 2014

1 Abstract

Power consumption is a troublesome design constraint for HPC systems. If current trends continue, future petaflop systems will require 100 megawatts of power to maintain high-performance. To address this problem the power and energy characteristics of high performance systems must be characterised. The main idea of this project is to design a methodology for the optimal selection (minimal number of systems that maximise performance and minimise energy consumption) of a network topology for high performance applications using ultra-low-voltage microprocessors platforms (Intel® AtomTM Processor E3825 - Minnowboard).

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 a case study to explore the applicability to apply MPI as the communication engine for distributed embedded applications. We setup MPI on a distributed embedded platform and experiment with an the MPI benchmark algorithm

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) [2, 1], 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 eMPI/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, however, it requires an operating system that supports threads, which in embedded systems it is not always available.

However 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 realise 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 efficient realisation of applications with multi-core or many-core processors in an embedded system is a great challenge. With application-specific architectures it is possible to save energy,reduce latency or increase throughput according to the realised operations, in contrast to the usage of standard CPU's. Besides the optimisation of the processor architecture, also the integration of the cores in the embedded environment plays an important role. This means, the number of applied cores and their coupling to memories or bus systems has to be chosen carefully, in order to avoid bottlenecks in the processing chain. This is the main problem we are going to face during this research process.

The main objective will be to design a methodology for the optimal selection (minimal number of systems that maximise performance and minimise energy consumption) of a network topology for high performance applications using ultra-low-voltage microprocessors platforms (Intel® AtomTM Processor E3825)

In order to make the experiments we will create a distributed computer system based on multiple Intel® AtomTM Processor E3825 (the configuration change will not be automated). The figure 4 shows a basic sketch of the distributed system we will create. A distributed computer system consists of multiple software components that are on multiple computers, but run as a single system. The computers that are in a distributed system can be physically close together and connected by a local network, or they can be geographically distant and connected by a wide area network. Our experiments will be addressed in a lab network (same room). Part of our task to achieve our main goal will be to make such a network work as a single computer

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) in order to check the gap in the multiple systems

- Compare an Embedded system with custom SW (Linux from scratch system and MPI for embedded (LMPI)) against a regular development system (with the same OS and MPI tools) in order to check the gap in the multiple systems

- 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 first experiment.

The platform we will 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]

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

Table 1: Minnowboard CPU characteristics

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:

- 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 reduction MPI_Allreduce combines values from all processes and distributes the result back to all processes) graphs the dev boar can support 8X times the maximum speed that Minnowboard, then in both platforms the drop of speed is extremely fast until reach a minimal point of stability with packages grater than 1.04e+06 Bytes. In the first approach we could assume that the Minnowboard has a poor performance, however if we look the graph from a higher perspective the reality is that the Minnowboard can sustain a better quality of transaction. The dramatic drop after the increment of 1.04e+06 Bytes is not reflected on the Minnowboard. On the Minnowboard the speed is the same until the size of the packages reach the 3.3e+07 Bytes.

A similar behavior occurs on the Alltoall (Each process sends to every other process).

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, where each process performs 2n (n being the number of processes in communicator comm) independent point-to-point communications (including communication with itself).

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);
```

Each process breaks up its local sendbuf into n blocks - each containing sendcount elements of type sendtype - and divides its recvbuf similarly according to recvcount and recvtype. Process j sends the k-th block of its local sendbuf to process k, which places the data in the j-th block of its local recvbuf. The amount of data sent must be equal to the amount of data received, pairwise, between every pair of processes.

After knowing this we can see on the graphs an unusual behavior. On the Development board results there is a drop on the speed after the transportation of 32 Kb packages, meanwhile on the Minnowboard the speed will not pass the 1.4e+06 (8 times slower than the dev board) in 32KB packages the minnow do not present this drop.

Although this is not a rule for all the tests. In the case of Bidirectional Bandwidth , the code send and receive in a bidirectional way (really simple algorithm) :

Algorithm:

```
                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);
          }
```

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 (this includes the case where no handle in the list is active).

The problem is that in the case of Minnowboard there is an important drop (in the middle of 1Kb and 32Kb). This problem might be caused do to a hardware/network problem, it makes sense due to the fact that they are waiting to all the co-processors. There are more cores inside the CPU of the development board than on the Minnowboard, this could generate the drop on the speed.

The only benchmark that shows a better performance all the time on the dev board is the latency. In this benchmark 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);
}
```

In this specific benchmark we can see that the Dev Board is highly better than the Minnowboard. However this 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 system to maintain a better performance (even with high volume packages). After this case of study we demonstrate that an embedded system might be use full for HPC applications, however the latency is a major problem that require HW reconfiguration.

One of the future works will have to cover the last two points: (so far we have just cover the first one):

- 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) in order to check the gap in the multiple systems
- Compare an Embedded system with custom SW (Linux from scratch system and MPI for embedded (LMPI)) against a regular development system (with the same OS and MPI tools) in order to check the gap in the multiple systems
- 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

After this will will be able 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

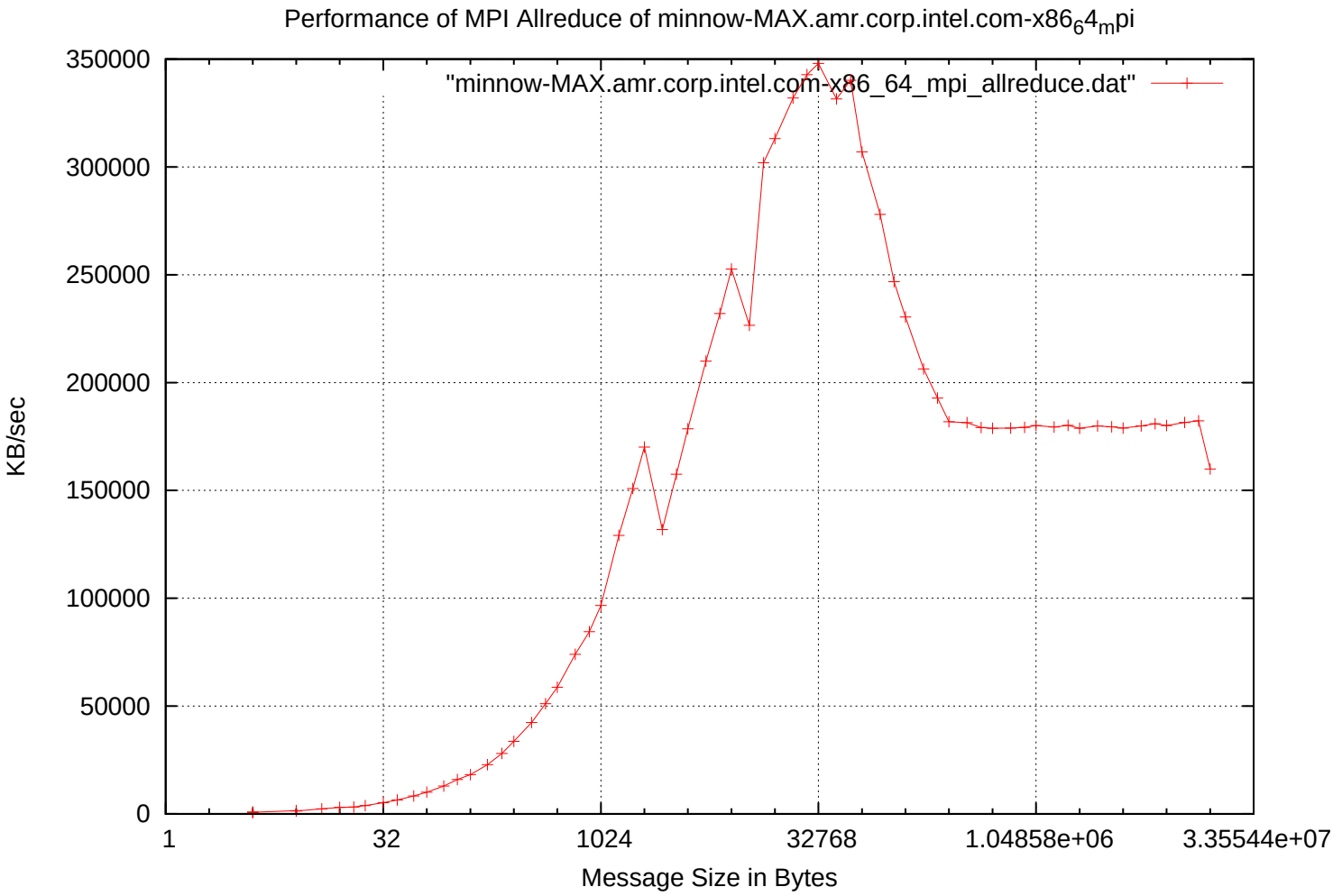

Figure 2: All reduce Dev Board

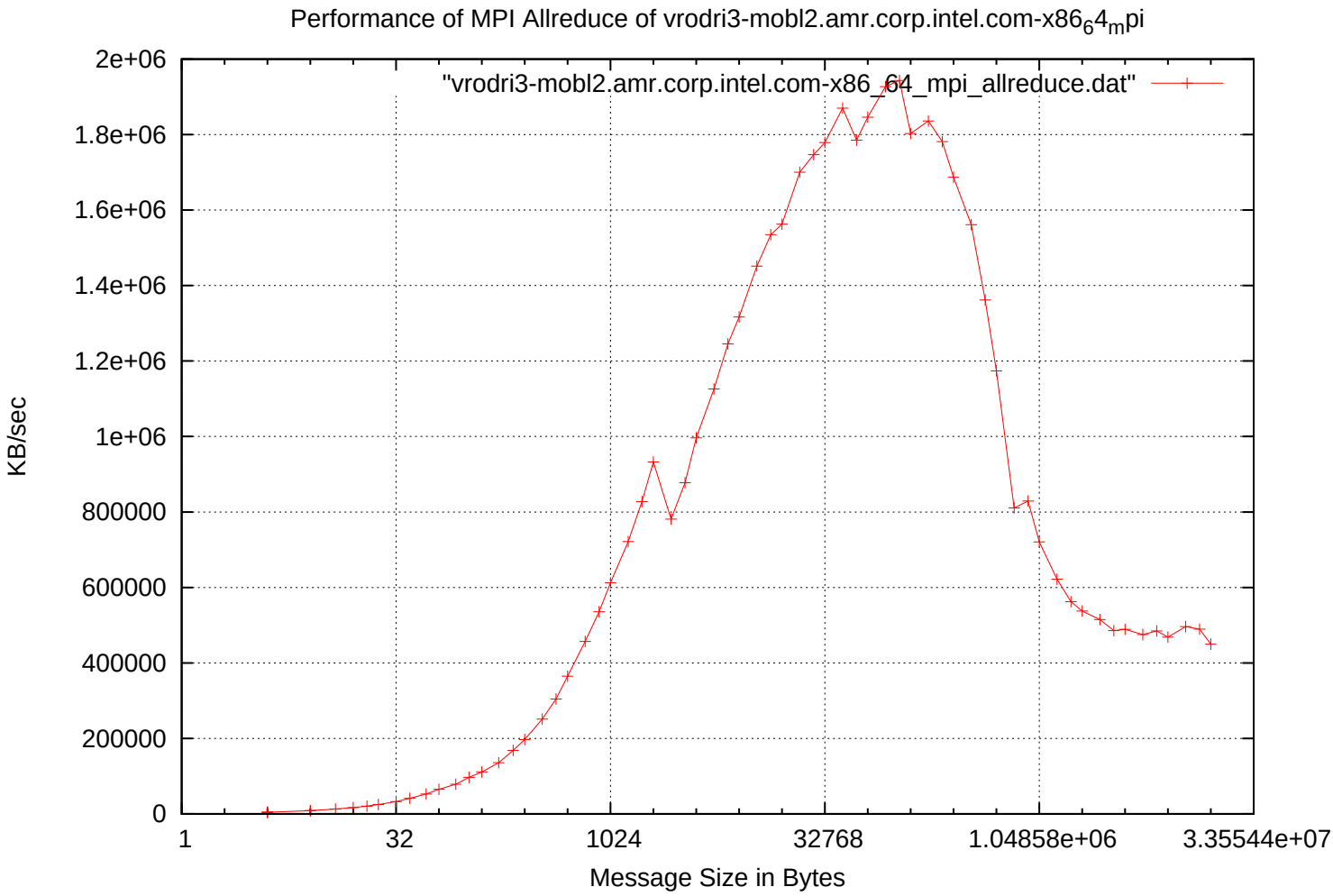

Figure 3: All to all minnowboard

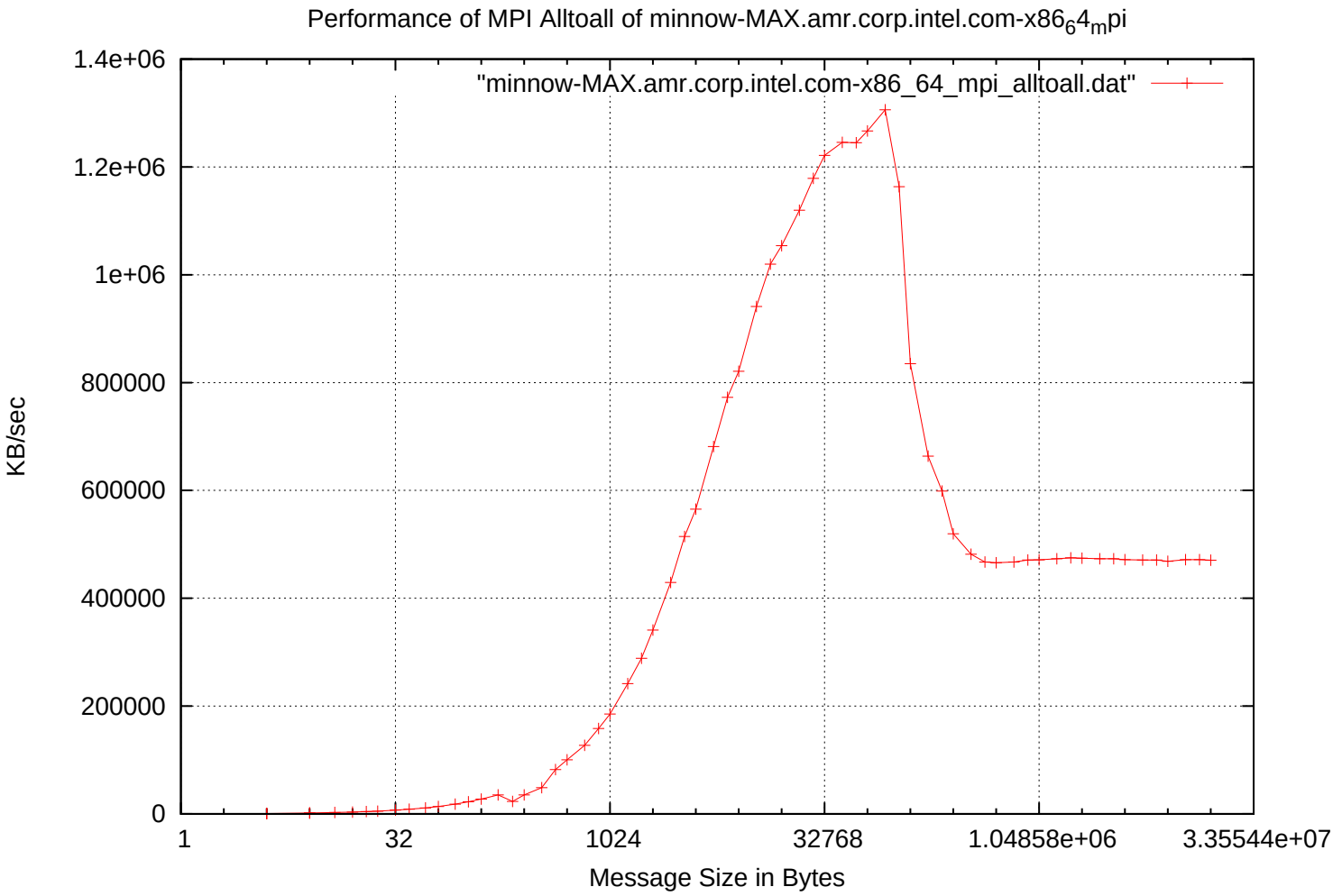

Figure 4: All to all Dev Board

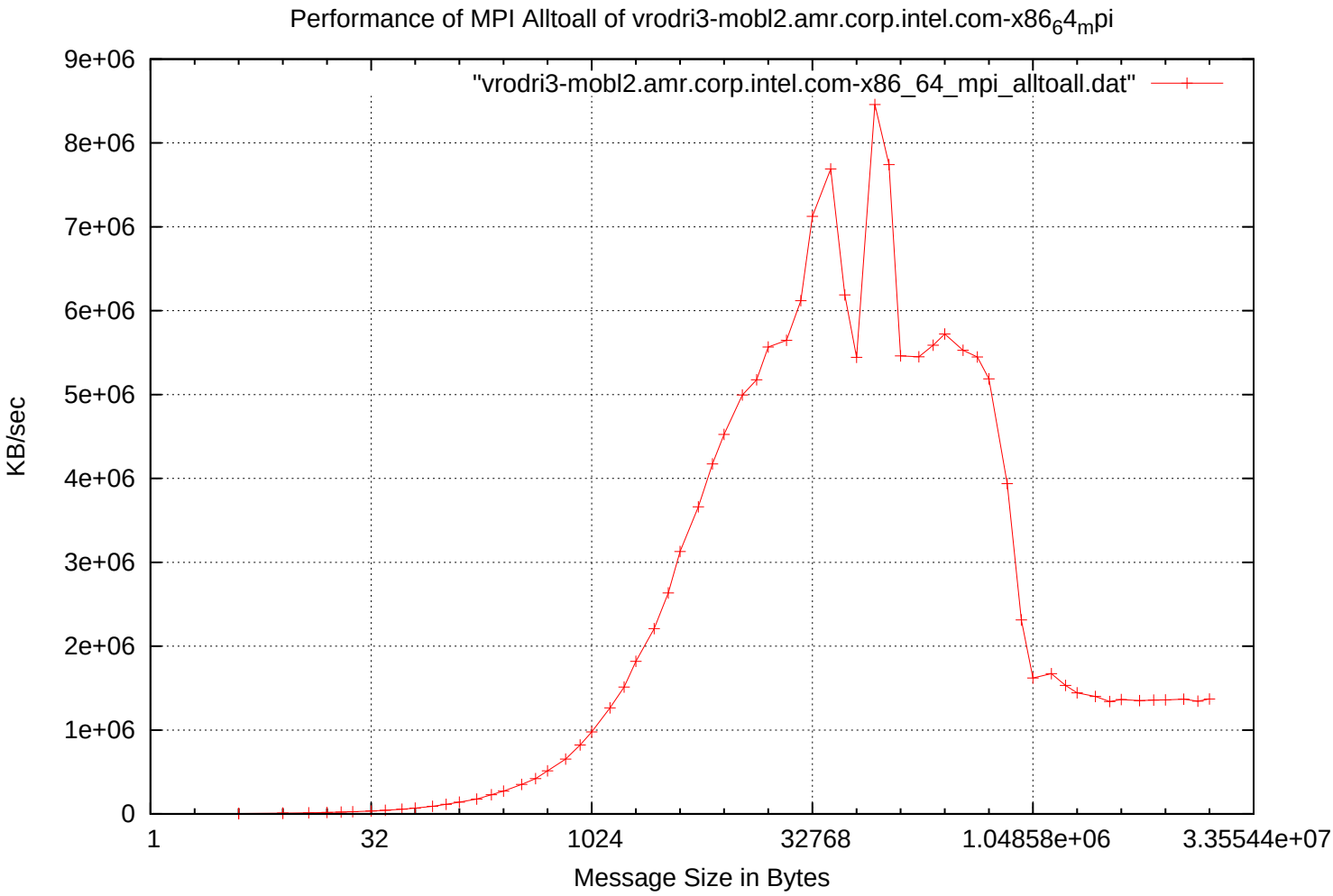

Figure 5: Bandwith on minnowboard

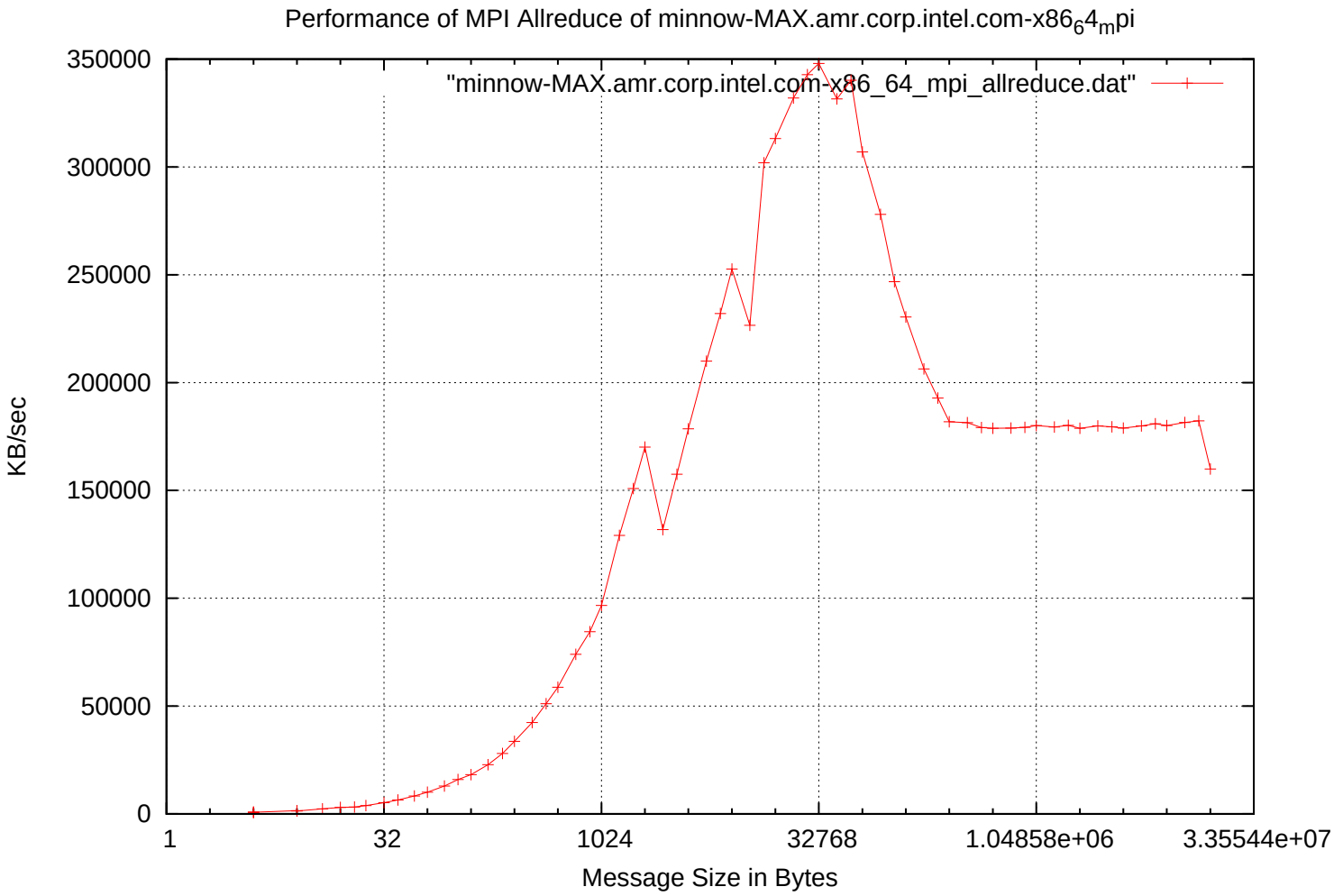

Figure 6: Bandwith on Dev Board

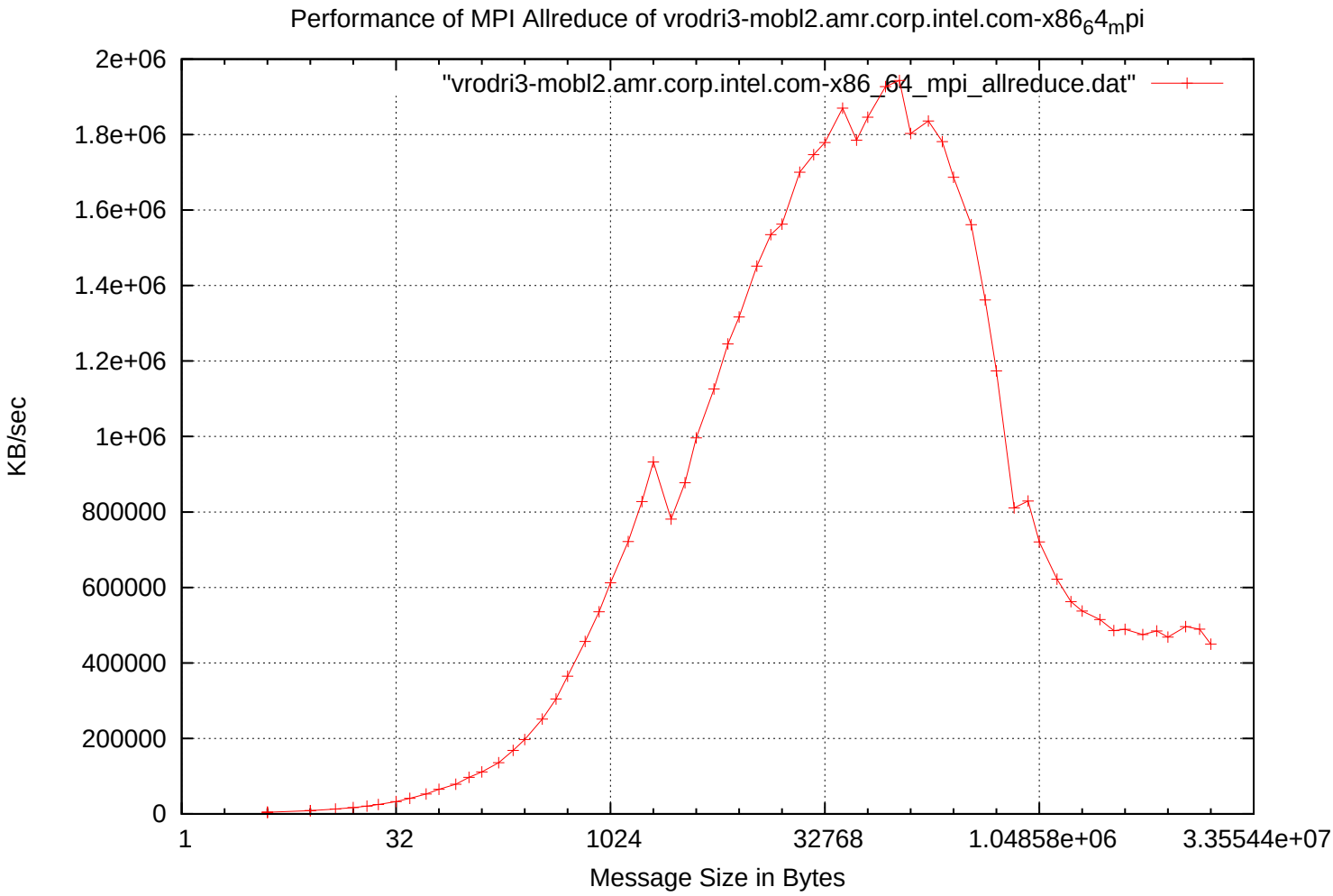

Figure 7: BidirectionalBandwidth minnowboard

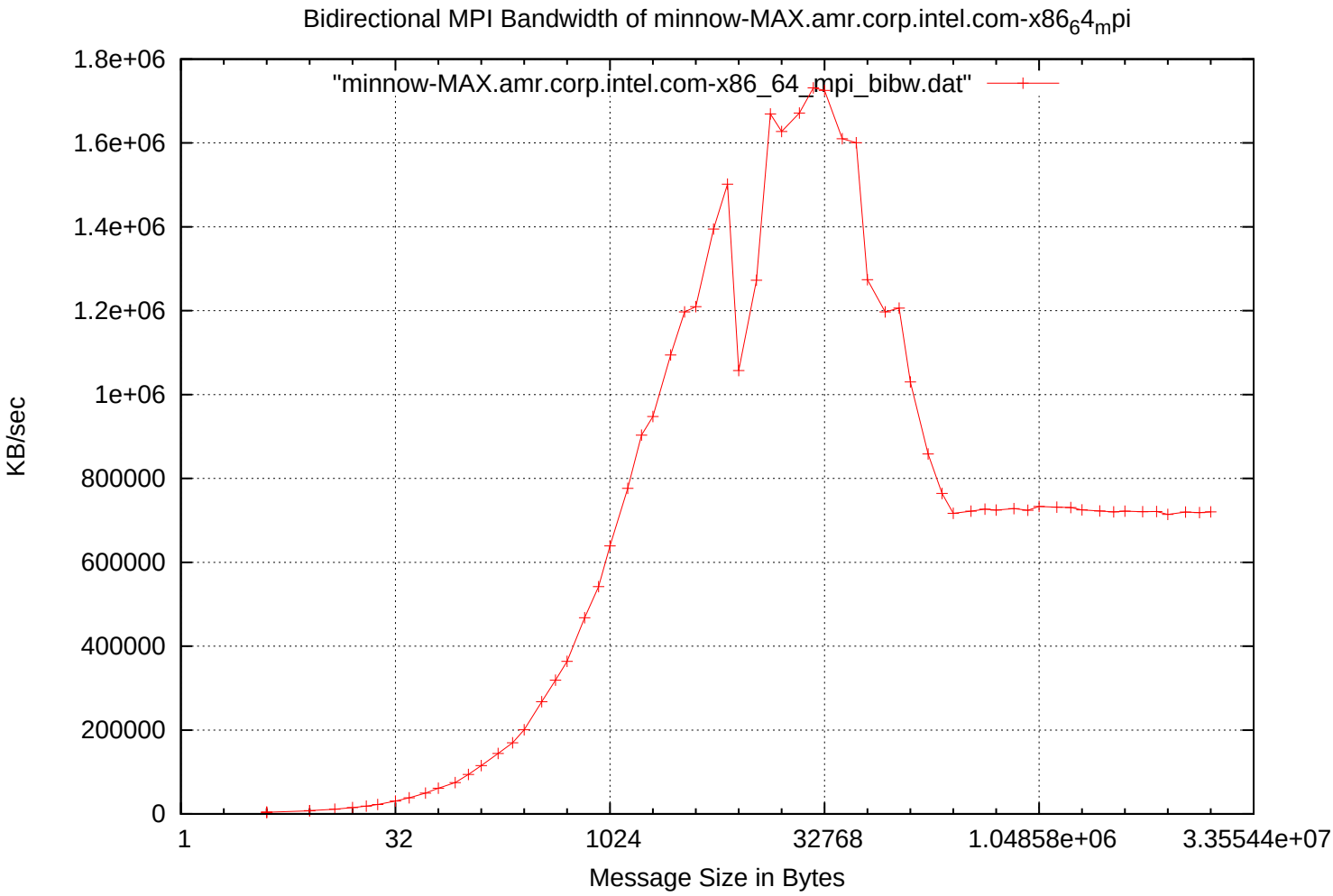

Figure 8: BidirectionalBandwidth Dev Board

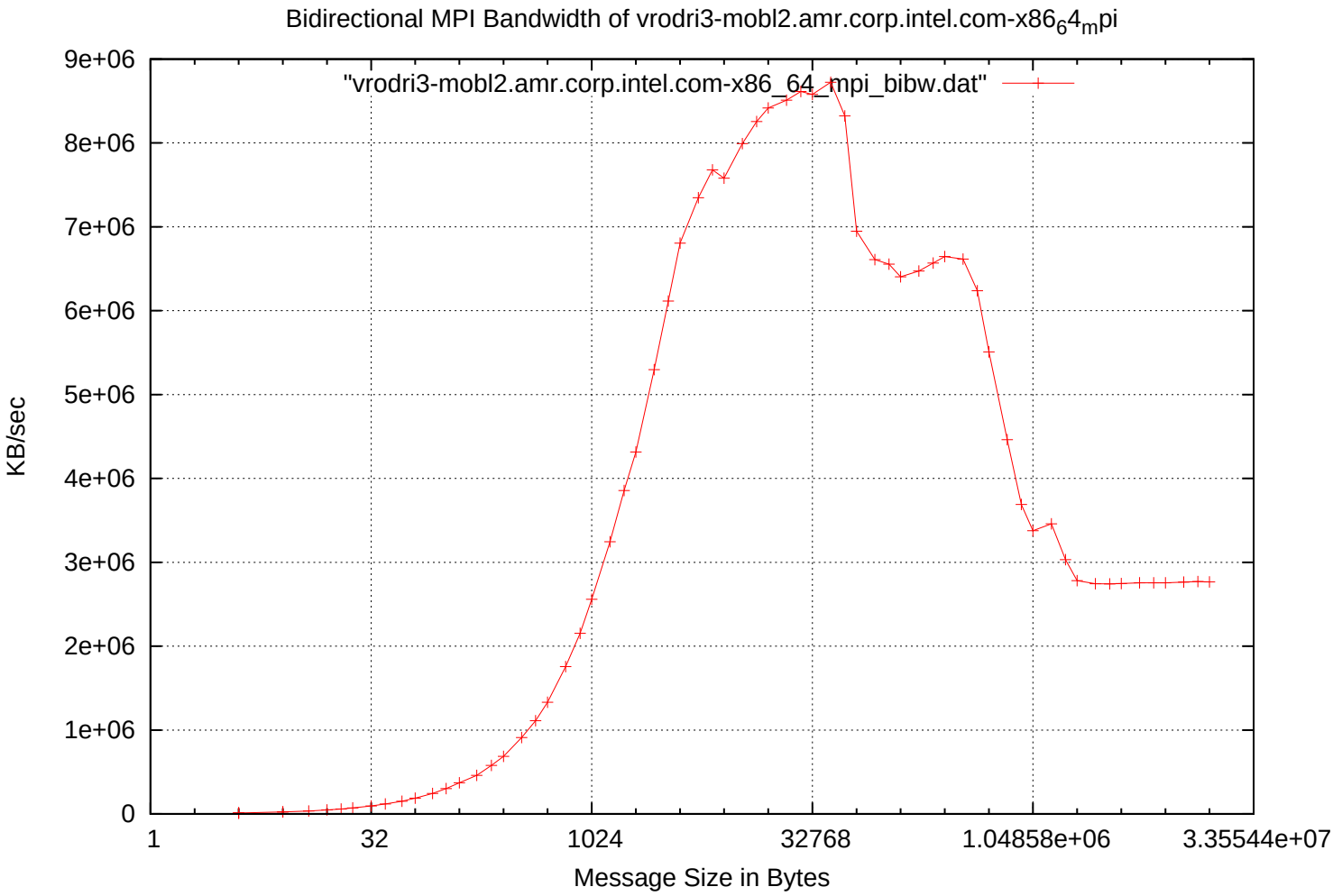

Figure 9: Broadcast minnowboard

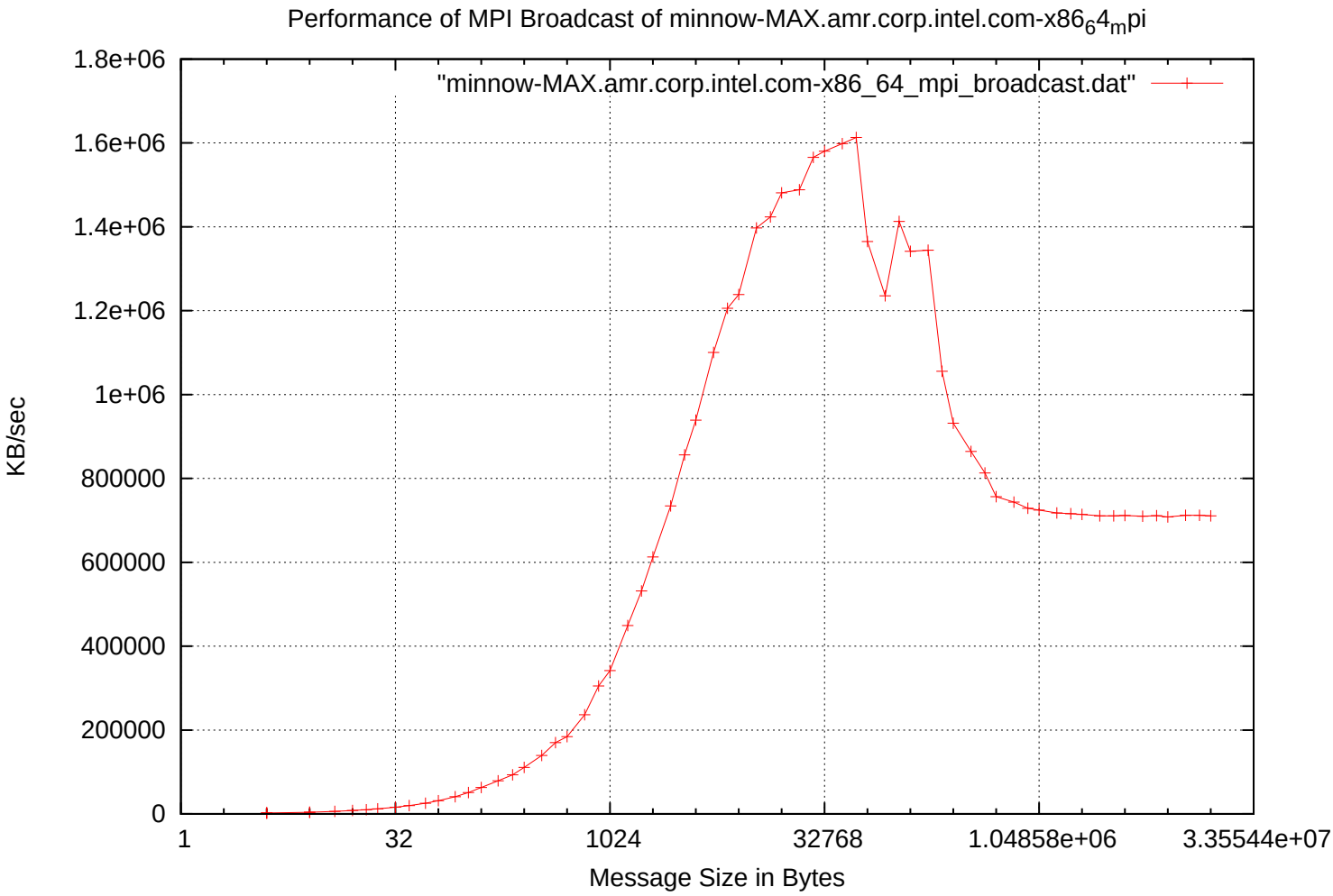

Figure 10: Broadcast Dev Board

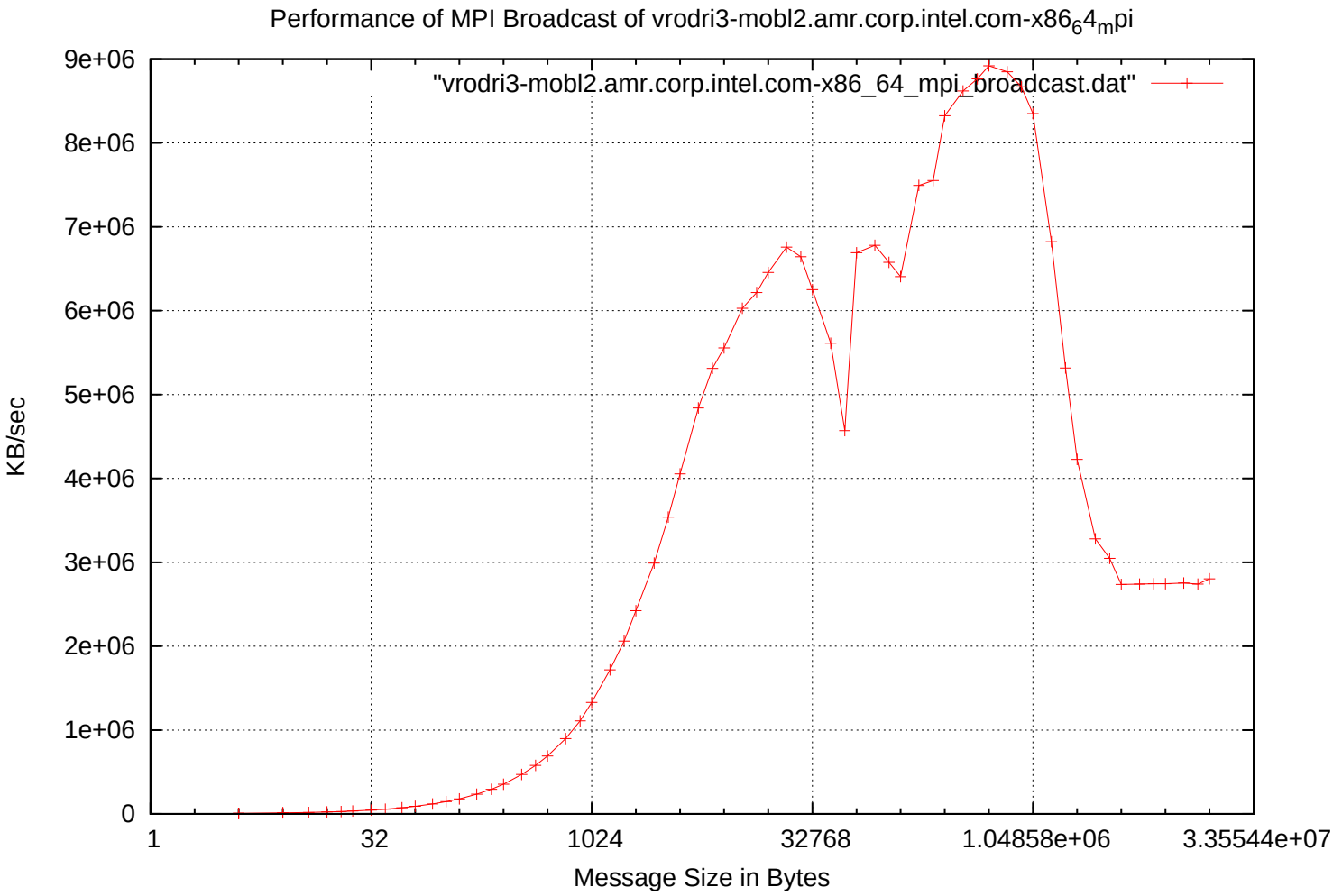

Figure 11: Application latency or Gap time on minnowboard

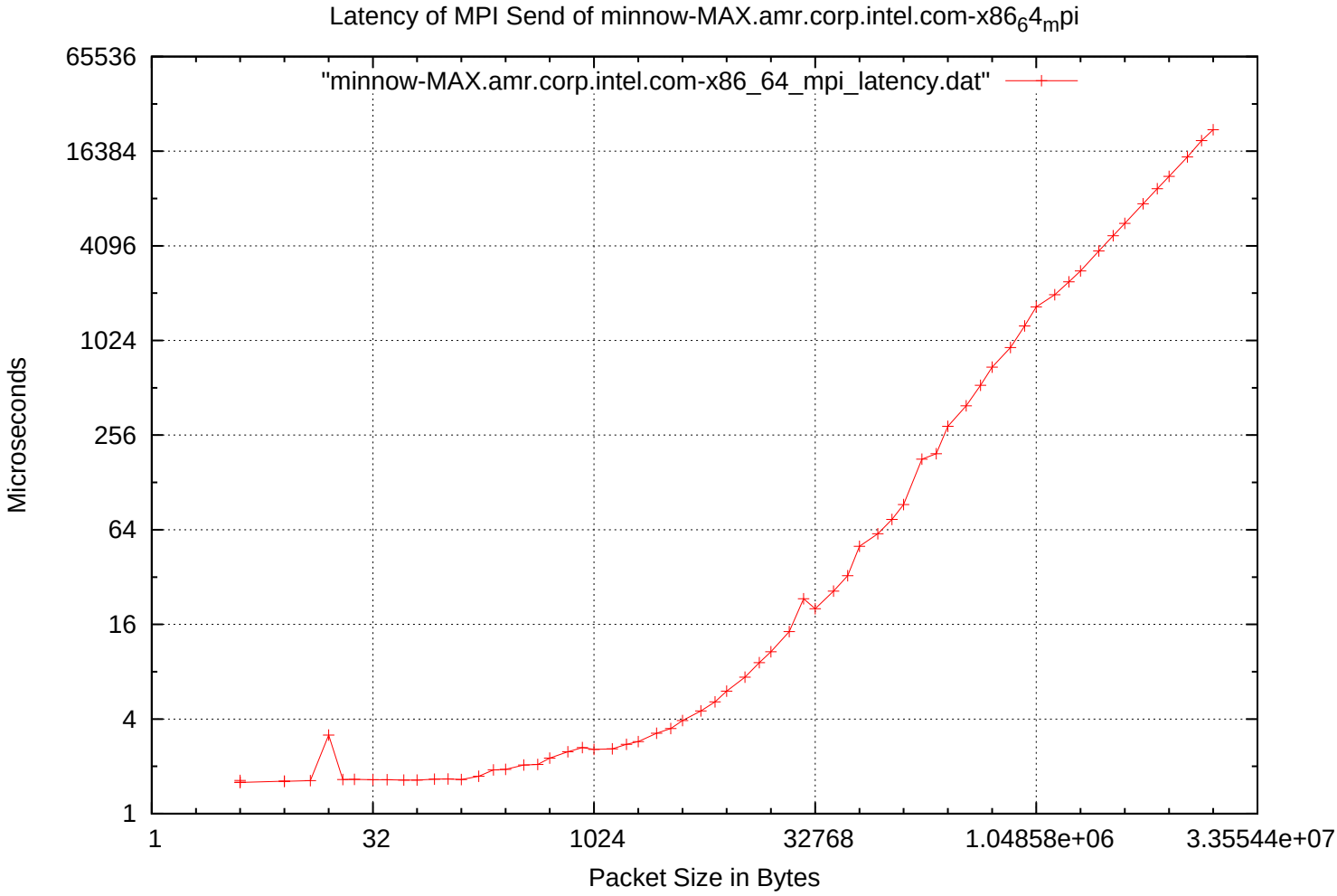

Figure 12: Application latency or Gap time on Dev Board

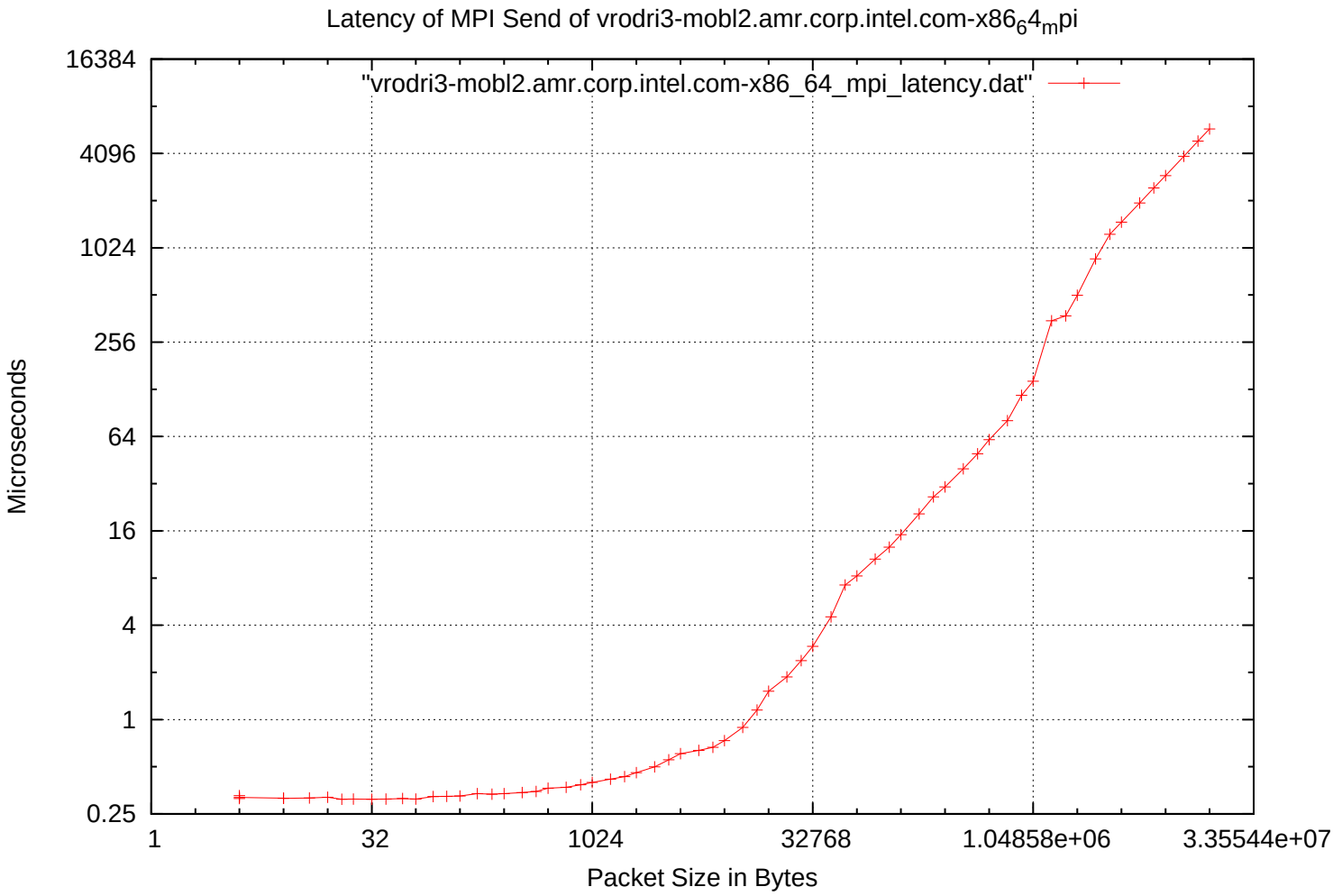

Figure 13: Roundtrip or 2 * Latency on minnowboard

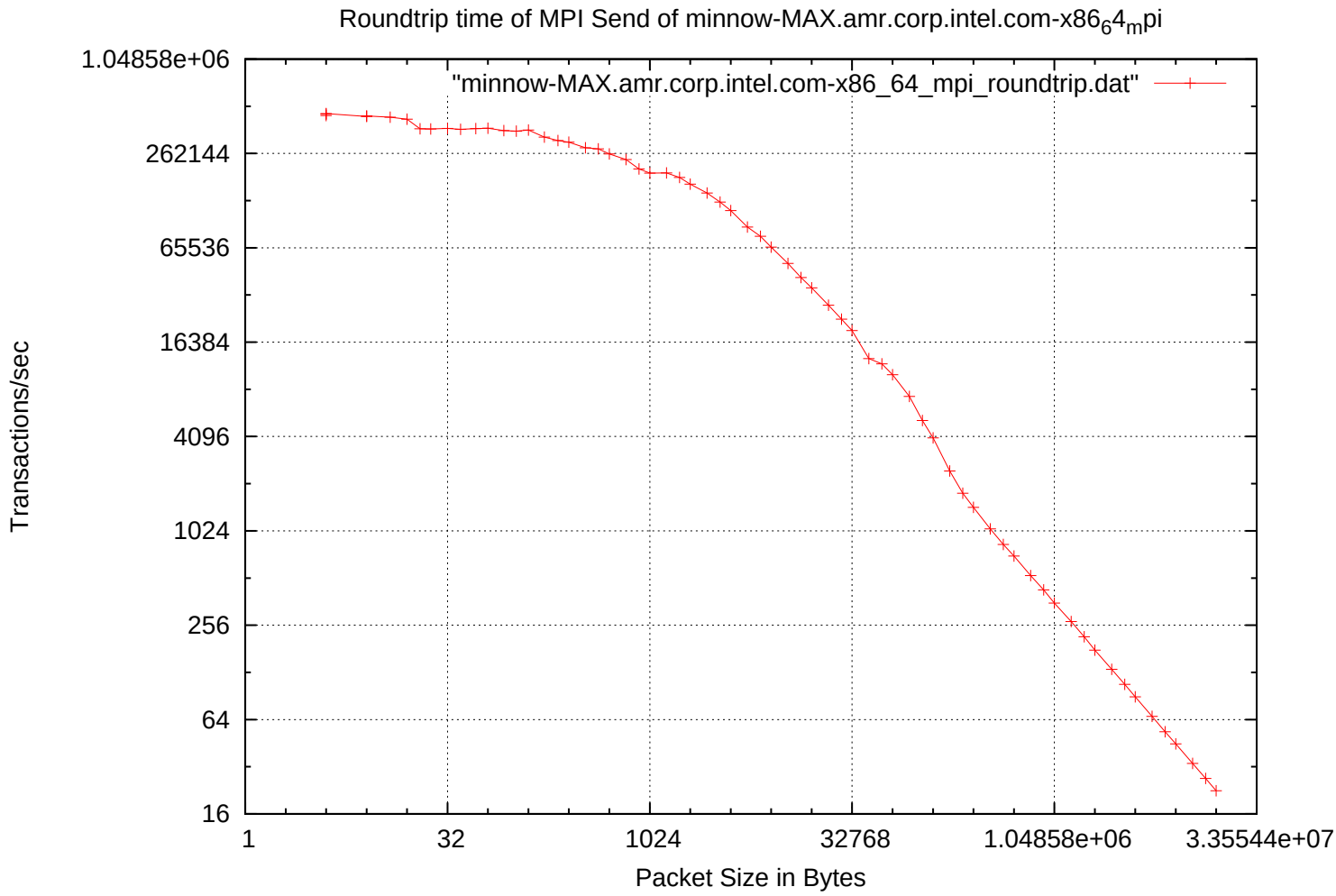

Figure 14: Roundtrip or 2 * Latency on Dev Board