Enhancing MPI performance using atomic pipelined message broadcast in a distributed memory MPSoC

Jongsu Park¹, Heejun Yun¹, and Sangook Moon²a)
¹ School of Electrical and Electronic Engineering, Yonsei University, 50 Yonsei-ro, Seodaemun-gu, Seoul 120–749, Korea
² Department of Electronic Engineering, Monwon University, 88 Doanbuk-ro, Seo-gu, Daejeon 302–729, Korea
a) smoon@mokwon.ac.kr

Abstract: We propose a scheme for enhancing the MPI (Message Passing Interface) broadcast function performance with supporting hardware logic to reduce a number of synchronization processes which can be avoided on a distributed memory multiprocessor system on a chip (MPSoC). We accomplish this using the concept of atomic execution which facilitates full pipeline utilization. To validate our approach, we implemented a bus functional model with systemC and evaluated the results against various message data sizes and number of nodes. Evaluation results showed that performance improvement can be achieved by up to 230% over the precedent pipelined message broadcast method. Synthesis results with 4 processing nodes show that the extra hardware cost for the proposed atomic pipelined broadcast logic occupies only 2.4% of the entire area.

Keywords: distributed memory system, message broadcast, MPI, multiprocessor SoC

Classification: Integrated circuits

References
[1] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt and E. Tune: HPCA-2013 (2013) 188.
[2] MPI Forum: MPI: A Message-Passing Interface Standard Version 3.0 (2012) http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.
[3] R. Garibotti, L. Ost, R. Busseuil, M. Kourouma, C. Jones, G. Sassatelli and M. Robert: DAC (2013). DOI:10.1145/2463209.2488836
[4] B. Tafesse and V. Muthukumar: VLSI Des. 2013 (2013) 936181. DOI:10.1155/2013/936181
[5] J. Zhang, J. Zhai, W. Chen and W. Zheng: Euro-PAR (2009) 81. DOI:10.1007/978-3-642-03869-3_11
[6] J. Traff, A. Ripke, C. Siebert, P. Balaji, R. Thakur and W. Gropp: EuroPVM/MPI (2008) 84. DOI:10.1007/978-3-540-87475-1_16
[7] T. Chiba and T. Endo: CCGRID (2007) 487. DOI:10.1109/CCGRID.2007.59
[8] P. Patarasuk, X. Yuan and A. Faraj: J. Parallel Distributed Comput. 68
1 Introduction

MPI (Message Passing Interface) standard [1, 2] offers API (Application Programming Interface) support for message passing libraries, and the MPI standard is being used as the reference standard in effect. The MPI standard includes various kinds of point-to-point and collective communications. The collective communication functions defined in the MPI library convert into a set of point-to-point communication functions by the MPI library cell so as to provide the ease of programming. As the collective communication functions account for up to 80% of the data transmission latency, it is very important to improve the process of handling these functions [3, 4, 5].

MPI_Bcast (broadcast) function is one of the most frequently used collective communication functions. The function routine copies the memory data from the root node (core) to all the other nodes (cores) which belong to the same MPI communicator.

There have been a few diversified works of improving MPI broadcast algorithms [6, 7, 8] which adopt the pipeline architecture to boost MPI_Bcast on cluster systems. In the pipelined broadcast algorithms, data are packetized into $k$ partitions to make $k$-pipeline stages so as to activate most communication ports efficiently. However, we discovered that $k-1$ number of unnecessary synchronization processes could be avoided in a specific environment such as an MPSoC.

The object of this paper is to improve the performance of the MPI_Bcast function by enhancing the software procedure and by supplementing trivial hardware logic. Thus, we propose an atomic-style of pipelined broadcast method and hardware support for MPI unit using SystemC and Verilog HDL.

2 Atomic pipelined message broadcast scheme

2.1 Avoiding unnecessary synchronization processes

In the existing pipelined broadcast algorithms, each processing node plays one role out of three; head, body, and tail. For example, in Fig. 1, PN0 plays a head node, PN1 and PN2 body nodes, and PN3 a tail node. In the case of the head node, PN0 sends data toward PN1, where MPI_Bcast is converted into $k$ MPI_Send commands. In the case of PN3 which is a tail node, MPI_Bcast is converted into $k$ MPI_Recv. On the other hand, the body nodes such as PN1 and PN2 receive data from one node and send the data to another at the same time, where MPI_Bcast converts into $k$ MPI_Send as well as $k$ MPI_Recv.
commands. At this point, $k$ times as many as iterative synchronization processes occur, which we want to avoid if there is no need to synchronize in between the initiation and the completion of the broadcast procedure.

One way to avoid these iterative synchronization processes is by having solid network connections between all the processing nodes, which can be intrinsically found in MPSoCs’ multi-processor bus structure such as AMBA AXI. Using this bus structure, we can preempt all the communication channels by establishing specific protocols and use the pipeline seamlessly as shown in Fig. 2.

In order to realize the seamless atomic transfer, compiler should not convert the $MPI_{Bcast}$ routine into multiple point-to-point communications. Instead, a new method is required that is capable of storing the packetized data words from one node into buffers without synchronization and simultaneously forward them to another. Thus, we propose a new supplementary command, which we call $MPI_{Fwd}$, so that the $MPI_{Bcast}$ routine can be converted into $MPI_{Send}$, $MPI_{Recv}$, and $MPI_{Fwd}$ for the head, tail, and body nodes respectively.

![Fig. 1. Pipelined broadcast architecture with iterative synchronization processes.](image1)

![Fig. 2. Fully utilized pipelined broadcast architecture with only one initial synchronization process.](image2)

![Fig. 3. Atomic procedure of $MPI_{Fwd}$ command.](image3)
2.2 MPI_Fwd function

MPI_Fwd operation should follow the order described in Fig. 3, which applies to all body nodes. If the order is not followed, the process should enter into a deadlock. We treat the MPI_Fwd as a non-interruptible atomic command to ensure both full channel utilization and reliability. The procedure order is as follows:

1. Put request message to $i+1$ node.
2. Get request message from $i-1$ node.
3. Get ready message from $i+1$ node.
4. Put ready message to $i-1$ node.
5. Get and put data messages between adjacent nodes.
6. Get and put complete messages between adjacent nodes.

3 Proposed hardware

3.1 MPI hardware unit

Fig. 4 represents the top view block diagram of the proposed MPI hardware unit with atomic pipelined broadcast (MPI_Fwd) support in a single processing node related to the MPI bus. The processor takes MPI instructions from memory and converts them to the appropriate MPI commands. The memory controller interacts with the data memory by reading and writing the data required for MPI communications.

![Fig. 4. MPI unit hardware block diagram.](image)

3.2 Message processing engine

MPE (Message Processing Engine) is responsible for handling signals from the processor through PW (Processor Wrapper), as well as signals related to
the MPI bus (send_data, receive_data, request, ready, and complete). In MPE, we put a request buffer to temporarily store request messages coming from another processing node while waiting for the current MPI command to complete. For the MPI_Fwd command, we implement separate Sender and Receiver blocks, which facilitate simultaneous send and receive actions required in MPI_Fwd operations.

### 3.3 MPI bus

MPI bus conforms to a cross-bar structure, through which all processing nodes can send and receive data. Each processing node sends and receives data via MW (Master Wrapper) and SW (Slave Wrapper). MW and SW serve as interfaces between the MPI bus and the MPI unit. We implemented the MPI bus by modifying the AMBA AXI bus architecture optimized with the MPI communication.

### 3.4 Distributor

The Distributor’s role is to distribute the signals from the MPI bus to the MPE. Signals such as ready and complete are passed to the Message Control block, request is stored in the request buffer, and receive_data are sent to Receiver.

### 4 Evaluation results

We implemented the bus functional models (BFMs) of the MPSoC architecture using the proposed atomic pipelined broadcast method and compared them with the non-pipelined sequential-tree broadcast algorithm [9, 10], and the pipelined broadcast algorithm [11]. The BFMs were modeled to explore the MPI architectures in SystemC, considering the latencies of the functional blocks. We evaluated the performance by simulating communication traffic and measured the trace with variable sized message data from 4 to 4096 bytes, while increasing the number of processing nodes from 4 to 32.

Observing Fig. 5, we can see the performance improvement. In this implementation, we used 4 processing nodes to evaluate the proposed method. The Y axis represents the factor of enhancement for the method compared with the sequential tree algorithm. Note that the marks of the pipeline and pipeline-chain indicate the precedent pipelined broadcast algorithm and the proposed atomic pipelined broadcast algorithm respectively. The proposed method shows an enhancement of approximately 3 folds greater than [9], and steadily outperforms the pipelined broadcast algorithm [11] as well.

Fig. 6 shows the comparison of the proposed method with reference to the pipelined broadcast algorithm [11]. In every case the proposed method showed to have better performance, with the performance enhancement increasing proportional to the number of nodes. With regards to the data message size, however, the enhancement factor peaked at 3.3 times (230% gain) when the data size was around 256 bytes.
Due to the area limitation of the die size (3 mm×3 mm) for a multi-project-wafer, only 4 processing nodes (p0-p3) and the MPI bus were integrated. As a result, each processing node was equipped with one processor (ympu; 15.2%), a MPI unit, 4 K-bit instruction memory (i_mem; 4.4%), and 4 K-bit data memory (d_mem; 4.4%). The resulting gate count for the MPI hardware accounted for only 2.4% (0.6% occupancy in 4 nodes each) of the entire MPSoC.

5 Conclusions

An atomic pipelined broadcast method implemented and fabricated in an MPSoC with 4 processing nodes, showed up to 230% performance improvement compared with a state-of-the-art algorithm of the pipelined broadcast architecture used in MPI functions. Flood of applications and the accompanying expansion of computation complexity demand many more processing cores in MPSoCs, where the task balancing and the bottle-neck effect due to communication overheads should be resolved. In this paper, in order to reduce

![Fig. 5. Speed-up ratio of the pipelined broadcast and the proposed method compared with [9] (# of nodes : 4).](image)

![Fig. 6. Comparison of the proposed method with reference to the pipelined broadcast algorithm.](image)
the transmission time of one of the most frequently used MPI collective communication functions, \textit{MPI} \textit{Bcast}, we invent a novel method which facilitates full utilization of the bandwidth of the MPI bus implemented with trivial reinforcement in hardware design and implementation. For the method to work with hardware, we propose to add \textit{MPI} \textit{Fwd} instruction to the message processing engine in the MPI unit to achieve high data throughput. As a result, a number of unnecessary synchronization steps have been avoided, as there is no need for mid-process synchronization if the destination node to transfer data is already known. We believe this idea can not only be implemented in distributed memory MPSoCs but also be applied to clustered systems as well.