An Implementation of MPI on FPGA for Distributed Memory Multiprocessing

Francisco Pires
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa, Portugal
francisco.pires@tecnico.ulisboa.pt

Mário Véstias
INESC-ID, ISEL
Instituto Politécnico de Lisboa
mvestias@deetc.isel.pt

Horácio Neto
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa, Portugal
hcn@inesc-id.pt

***Abstract*—In the context of distributed memory processing systems, this work presents an implementation of the Message Passing Interface (MPI) using FPGA soft-processors. This implementation consists not only in a C library but also in the configured FPGA hardware to support the communication between all the processors. Considering the limitations of the target devices, the low resource utilization is emphasized as well as the hardware scalability and the software reliability. Experimental results with several functions, including matrix-vector multiplication and backward substitution, on a 8-processor architecture validate the developed work and show that algorithms may be accelerated with good performance efficiencies.**

I. Introduction

Embedded computing applications have become very demanding over the years. In fact, a single core general-purpose microprocessor may not achieve the desired performance when running some specific algorithms, especially the ones with real-time constraints. Since the development of single core processors seems stagnated (the processor frequency has reached a limit due to power consumption and thermal reasons) but the number of transistors per chip increases every year, the multiprocessor approach is actually considered a viable solution to improve the performance of the most demanding embedded applications.

Multiprocessor Systems-on-Chip (MPSoCs) on Field Programmable Gate Arrays (FPGAs) exploit the parallelism of multi-processors in order to achieve better algorithmic performances for embedded systems.

Several studies and applications that solve these algorithmic problems using FPGA soft cores - or even heterogeneous systems with both soft and hard cores - have been frequently discussed. In these works, the proposed goals are frequently achieved but the application development cycle is never separated from the hardware implementation. This means that a software developer for these systems must also know the features of a hardware implementation, spending a considerable amount of his developing time writing low-level code to the application he wants to develop. Therefore, a lightweight version of the Message Passing Interface (MPI) [1] standard programming model that abstracts the communication between multiple soft processors on FPGA devices is proposed in this work.

The MPI is usually used for data exchange in a paradigm of distributed-memory high-processing computers. In fact, the MPI has been considered by diverse authors the de facto standard in this context for twenty years [2], [3]. The global levels of adoption of this programming interface are an obvious advantage since the embedded software developer does not need to learn a new library specification. Furthermore, a large amount of MPI applications, originally intended for clusters of workstations or supercomputers, may be ported to embedded systems using the implementation suggested here.

Since embedded systems are different from the usual cluster and supercomputer systems that MPI aims for, the implementation in this work approaches new questions regarding the small use of resources and system portability. Also, the implementation library must take into account the directives and the prototypes determined by the MPI-Forum in order to not confuse a software developer. Therefore, the main objectives of the work presented in this document consisted in studying and developing solutions that implement a low cost MPI interface in distributed memory soft-processors, considering scalability and resource utilization important constraints to be considered in this proposal.

Section II provides a description of the Message Passing Interface. The related work is described in section III. Section IV describes the proposed massage passing interface hardware and software implementation in FPGA. Section V describes and analyzes the results of the proposed MPI architecture in FPGA. Finally, section VI concludes the paper.

II. Message Passing Interface

The Message Passing Interface (MPI) was introduced in 1994 [1] to define a general library standard for parallel communication systems. The MPI protocol considers more than 100 functions though the great majority of the programs only use a small set of point-to-point and collective communication functions. In fact, a basic set of MPI functions provides the essential tools for the resolution of almost every parallelizable problem, namely:

- *MPI_Init* - the function where the entire MPI environment is started and important attributes, like the number of processors and the ranking of each processor, are set;
- *MPI_Comm_size* - return the number of processors to the user;
- *MPI_Comm_rank* - return the rank to the user;
- *MPI_Finalize* - shuts down the communication environment;

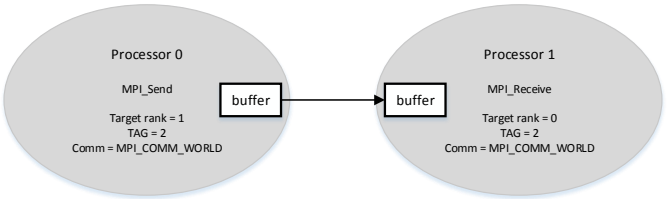

Fig. 1. Example of matched MPI message triples.

- *MPI_Send* and *MPI_Recv* - the core functions for the data transferring between multiple processors.

The implementation of the send and receive functions could adopt one of four different modes: the synchronous mode where both processors (the sender and the receiver) handshake and wait for each other to start the data transfer; the buffered mode, where the sender processor writes the data on the buffer and does not wait for the receiver to start the transfer; the standard mode, where it is up to the MPI implementation to determine whether the messages are buffered or not; and the ready mode where the sending operation only works if a receive request has been already posted.

Each MPI point-to-point function message is identified by a (target processor rank, tag, communicator) triple. The maximum value allowed for the rank is obviously related to the number of processors executing the application while the maximum tag value is defined by the implementation.

The communicator is a specific feature of the MPI standard that sets the communication context within or between groups of processors (intracommunicators and intercommunicators, respectively). The MPI message triple defines whether an *MPI_Send* message request matches or not an *MPI_Recv* request. Since the packet arrive order on some type of network is not deterministic, some message triples may arrive in a different order than originally expected, causing eventual matching problems. To solve this issue, the MPI standard suggests the use of queue buffers for the unexpected messages and pending receives (see figure 1).

Besides *MPI_Send* and *MPI_Recv* functions, collective functions are also defined by the MPIForum. The collective functions may be defined as facilities where multiple processors interact with each other using just a single function call. The main advantage brought by the collective functions is the less effort for the application developer to code certain problems. For example, with the basic set, if one processor wants to receive a data piece from every other processor and accumulate all the values received in a local variable, the application developer must call the *MPI_Send* function on the sender processors and implement a loop of *MPI_Recvs* on the root (receiver processor) where in each iteration the value received is accumulated in a local variable. With collective functions support, the application developer may simply call the *MPI_Reduce* function on every processor and all the reduction work is done internally (see figure 2).

Anther important function is the *MPI_Barrier*, used to synchronize all processors belonging to the same context of a communicator. Using just point-to-point MPI functions, the implementation of synchronization would require a significant

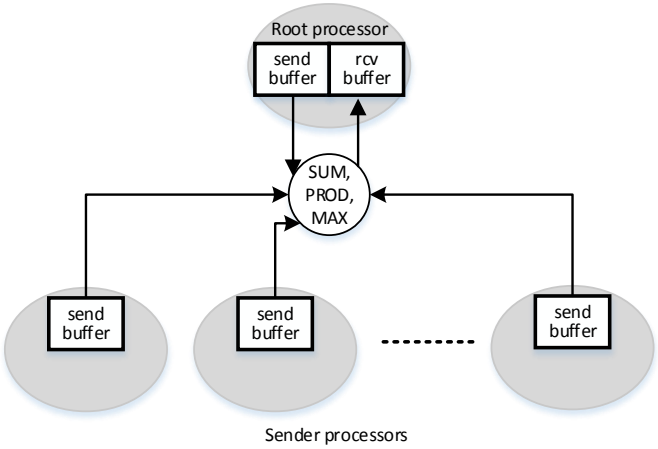

Fig. 2. *MPI_Reduce* function.

effort from the MPI user. With a single call to the *MPI_Barrier*, the complexity of synchronization is abstracted from the user.

Depending on the MPI implementation, some collective functions may have its internal code independent from the point-to-point functions, dealing directly with a lower layer of the software architecture. When these cases occur, the performance of those collective calls is usually optimized. Hence, the MPI user shall call, whenever possible, the collective functions instead of working over the point to point functions.

III. Related Work

Since the first MPI-1.0 reference document describing the MPI interface, several implementations and extensions have been proposed. While the majority of the implementations (like the FT-MPI [4], LAM/MPI, MPICH [5] or the OpenMPI [6]) aimed the state-of-the workstations and supercomputer systems running conventional operating systems, some work on MPI implementations for embedded systems has also been made, like the SCMP Multiprocessor [7].

Specifically on the FPGA field, some standard implementations for either a basic set of MPI functions or specific collective communication functions may be found. Initially, some implementations that intended to directly port the MPI libraries from the standard workstation versions to the FPGA systems (like the eMPI [8]) were presented. Their ports were considered too heavy, especially when in comparison with implementations designed specifically for the FPGA systems.

Other approaches did not port directly the MPI versions but relied on operating system facilities, which is a big disadvantage since many simple FPGA soft processors are not intended to run with operating systems. The works presented by Gao et al.[9], proposing specific hardware cores to accelerate the MPI Barrier function, and by Brightwell et al.[10], suggesting a new data structure to store and search more efficiently the MPI communication requests, are important developments on this topic.

Comparing with the implementation described in this paper, the works presented by Williams et al. [11] and Saldaña et al. [12] are the ones with the most similar objectives. In Williams, a reduced set of the MPI standard is implemented

for multiple Microblazes (Xilinx FPGA soft processors) connected to each other through Fast Simplex Links (FSLs). The number of FSL interfaces in a Microblaze processor is limited and, therefore, the system scalability is restricted to the maximum number of eight Microblaze FSL interfaces. Actually, the Microblaze soft-processors are also compatible with the AXI-Stream interfaces. Tailoring the system proposed by Williams to AXI-Stream would extend the maximum number of processors to 16. However, two memory elements (usually organized in a FIFO fashion) are required for each pair of linked Microblazes. Defining N_P as the number of Microblaze processors, the number of required memory elements for the MPI communication structure, M, is given by $N_P \times (N_P - 1)$. A design with 16 processors would, therefore, require 240 memory elements. This means that this architecture may be very memory demanding.

In [12] the problem of a limited number of processors is solved by using a custom network-on-chip (NoC). The interconnects (designated NetInterfaces by the authors) may receive an instruction from the receiver processor to select which sending processor is preferred in situations of contention. The eventual packets that arrive from other processors at the same time of the dealt transfer are retained in FIFO memories until the NetInferface is free. One memory element is needed per interface of each NetInterface. Two memory elements are also needed per Microblaze due to the FSL bidirectional interfaces used. Since there is one NetInterface per processor the total number of memory elements is $N_P \times (N_P - 1) + 2 \times N_P$, requiring more memory elements than in the [11].

Saldaña also proposes a version where a hardware accelerator engine is attached to each Microblaze in order to accelerate the tasks of receiving, analyzing and storing communication requests. This version requires three more memory elements per accelerator: two FIFOs to exchange data with the NetInterfaces and a queue memory to store unexpected MPI requests. The new value of M is therefore $N_P \times (N_P - 1) + 5 \times N_P$.

Looking at these equations and analyzing the architectures, it is clear that Saldaña's architecture is more scalable but it occupies much more memory and LUTs when compared to the architecture suggested by Williams. To benchmark his work, Saldaña implemented the Jacobi algorithm to solve the heat equation. Using a group of both Microblaze and PowerPC405 processors, maximum efficiency is achieved up to 10 processors while the maximum speedup obtained is around 27 using 40 processors. The authors justify the loss of performance with the limitations of resources of the FPGA device they used (Xilinx XC2VP100).

Over the years, Saldaña's work focused on heterogeneous systems where the FPGA soft-processors and specific hardware engines are able to communicate with x86 hard processors through MPI requests and data sent to a shared memory [13]. Dedicated hardware implementations for the collective *MPI_Bcast* and *MPI_Reduce* functions were also upgrades verified [14]. All these improvements made this MPI proposal in a complete and efficient solution, however, all the intellectual property (IP) cores implemented and all hardware used is too heavy for medium and low-cost FPGA systems.

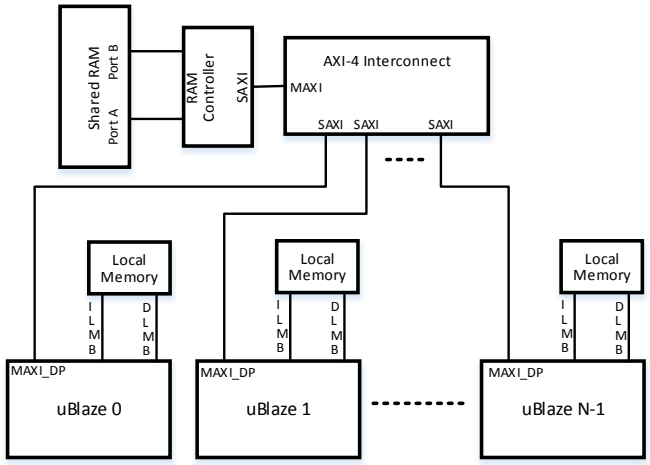

Fig. 3. Generic hardware architecture used to implement the MPI software.

IV. Hardware and Software Design of the Distributed Memory Multi-Processing System

In the following sections, we describe the proposed multi-processing architecture and the design of the software MPI routines over this architecture.

A. Hardware Design of the Multi-Processing Architecture

Since the target is a Xilinx FPGA, the Microblaze soft-processor was chosen as the processing core of the architecture. Microblaze is compatible with the standard C language libraries, do not take an excessive LUT area (allowing implementations on very low-cost FPGA devices) and provide interfaces compatible with the required protocols (AXI-4 and AXI-Stream) for efficient external communication.

The architecture defined to link all the processors consists in using an AXI-4 Interconnect that allows all the processors to access a shared RAM memory. It's important to note that this AXI RAM memory is only used for the MPI functions, each soft-processor remains with their local BRAM memories to store the application code, heap and stack (see figure 3).

Instead of using a shared memory, a system where all the processors were linked through AXI-Streams was also considered. In this case, the number of memory resources needed was considered too high, as can be seen in the related work section. The high resource requirements are even more evident when a custom NoC is implemented and an interconnect is needed for each processor.

Following with the shared RAM architecture, some hardware cores with the objective to accelerate the asynchronous point-to-point MPI communication functions were considered (the processors could execute some portion of computational work while these engines would send and receive data). After some experiments, it was concluded that these cores had an excessively complex computing work accessing and processing the BRAM memory. In addition, the resources to implement this structure would limit the scalability of the system (it would take a considerable LUT space and would require at least two additional FIFO memories per processor).

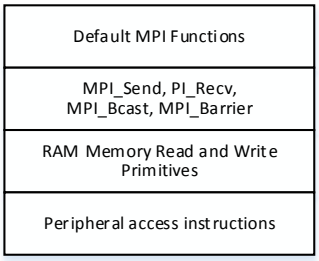

Fig. 4. Software stack of developed library.

The AXI-4 Interconnect, optionally, provides FIFO buffers to accelerate burst transfers. Since the burst mode is not compatible with the AXI DP interface used by the Microblazes, this shared memory architecture only needs one memory element (M = 1) to implement the communication structure of a fully working MPI library. An AXI Interconnect has a limit of 16 slave interfaces. Thus, when more than 16 processors are inserted, a second AXI-Interconnect level must be inserted. This fact does not have a big impact in the resource utilization since an AXI-Interconnect takes much fewer LUTs than a Microblaze processor.

B. Software Design of MPI Functions

Since we are targeting embedded system devices that may have very scarce resources, the software design had to take into account the space that the own code occupies focusing only on the essential MPI features. The final size of the developed MPI code was, therefore, around 20 kB (actual workstation implementations occupy about 100 MB). The developed library for the C programming language is organized according to a layered approach (see figure 4).

While the point-to-point *MPI_Send* and *MPI_Recv* functions and the optimized collective *MPI_Bcast* and *MPI_Barrier* functions were built directly over the BRAM memory access macros, the other MPI functions were developed on a higher level abstraction. Therefore, these functions call the lower level MPI functions instead of directly calling the BRAM memory read and write macros. This layer structure eases the system portability and future software improvements.

Another important point of this software implementation is the way how the data stored in the shared BRAM is organized and how that memory organization can be related to the MPI communication modes defined by the MPI-Forum. Considering that N_P processors are being used, the software logically divides the memory in $N_P \times (N_P - 1)$ blocks, each one consisting in the communication zone for a combination of two processors: a sender and a receiver (see figure 5).

The logical division by blocks eases the search process since multiple processors are trying to read and write in the same memory. The logical combinations depend on the role of the processors, e.g. processor 0 being a sender and processor 1 a receiver is a different combination of processor 1 being a sender and processor 0 a receiver. With this approach, the shared memory is both a link and a synchronous communication buffer. With the exception of the logical block receiver0-sender1 (the first in the memory), each block contains an area

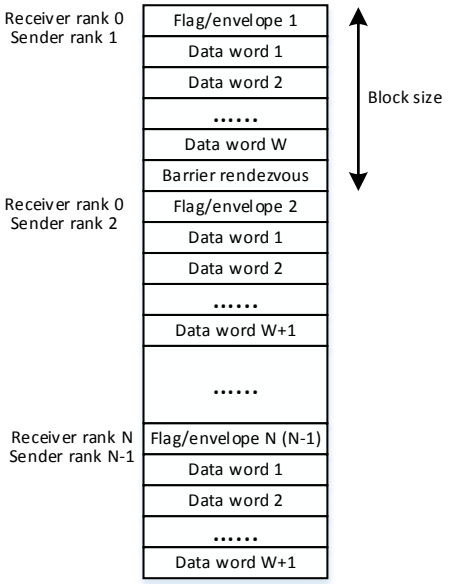

Fig. 5. Logical organization of the shared memory.

of one integer (4 bytes, 1 memory position), where the TAG envelope of the message is set by the sender, and an area (the remaining block size) reserved for data to be transferred. The first block (receiver0-sender1) has the same structure but an additional position in order to do a semaphore for the *MPI_Barrier* function.

The data transfer area not only allows all MPI datatypes that take 4 bytes but also the 8 bytes datatypes (*MPI_LONG_LONG* and *MPI_DOUBLE*). Each 8-byte value is halved in 2 memory data words. The communication mode may be considered the standard since, depending on the situation, a sender processor may or not send the data without waiting for the receiver processor. The conditions that hold back the sender are when the receiver did not read yet the latest data transmission and when the data to send is bigger than the block size.

The implementation of each MPI function developed is now described:

MPI_Init - In this function, every processor computes the size for each communication block of the memory, sets different environment variables (like the group size or the process rank) and finally synchronizes with the other processors by calling the *MPI_Barrier* function.

This initialization function relies on the Programmable Logic reset of the first launched processor to initialize the shared memory.

MPI_Comm_size - This function just returns the value of the already set variable of the number of processors;

MPI_Comm_rank - Returns the value of the processor rank. This value is set by the Xilinx tools when the hardware is implemented;

MPI_Send and **MPI_Recv** - Since the memory is logically divided into blocks, when a processor wants to send a set of values, goes to the attributed block and checks in the

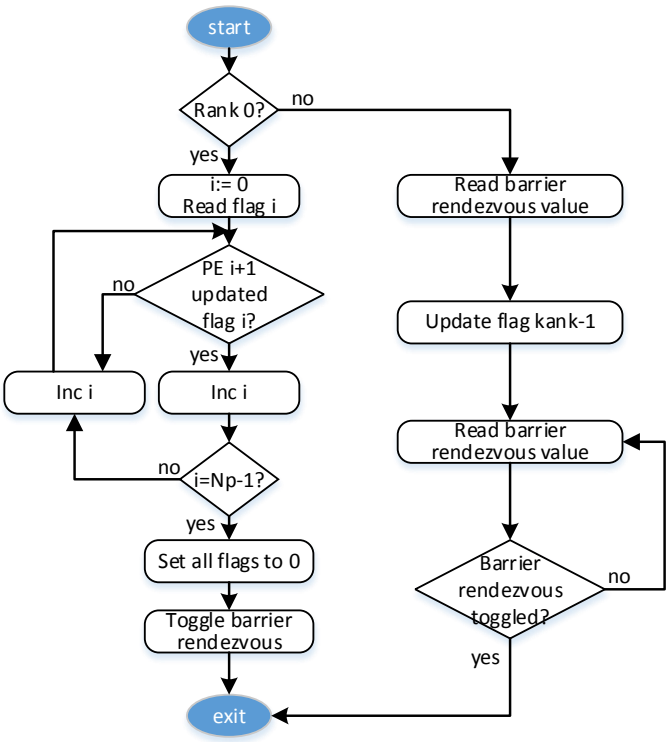

Fig. 6. Flowchart of the developed MPI Barrier function.

envelope position if the latest transmission was completed. If so, the *Send* function writes the data and changes the value of envelope position to the message tag. On the receiver side, the processor is reading in the appropriate block in the envelope position while it does not contain the tag value. When the value appears, the receiver reads the data and sets that envelope as free;

MPI_Barrier - The Barrier is implemented with the help of the envelope positions used for the point-to-point communications and a specific semaphore/rendezvous position (see figure 6). All the processors, except the master processor 0, set to a new value the envelope position of the block where these processors act as senders and the processor 0 act as receiver. The processor 0 reads those positions and, after noticing that all those positions were changed, it toggles the value of the dedicated barrier rendezvous position meaning that all the other processors are clear to leave the barrier;

MPI_Bcast - Though a broadcast function is easily implemented over the already developed point-to-point functions, a different and more efficient approach was taken in order to exploit the advantages of the hardware configuration of the system. In this implementation, all the processors synchronize and the root processor writes once in a block. The receiving processors store the data from that block and inform the root processor that the data has been read;

MPI_Reduce, **MPI_Gather**, **MPI_Scatter** - These collective communication functions were built over the point-to-point communication routines. Since it is not expected a significant performance improvement from implementations independent from those point-to-point functions (taking into account the defined architecture), this approach saves an important amount of code space, which is vital for an embedded implementation where the memory resources are very limited. The Reduce operations supported are the **MPI_SUM** and the **MPI_PROD**.

MPI_Finalize - This function sets the defined MPI global variables to the original state and calls the MPI Barrier where each process waits for the others to finish the MPI session.

TABLE I. FOOTPRINTS OF THE IMPLEMENTED MPI FUNCTION SETS IN THE MICROBLAZE.

Function set	Functions Implemented	Footprint
Basic set	MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv, MPI_Finalize, MPI_Barrier	10.3 KB
Collective set	Basic set functions, MPI_Bcast, MPI_Reduce, MPI_Gather, MPI_Scatter,	25.4 KB

Table I shows the footprint of each set of functions.

There is a significant increase of the footprint when the set that also implements the collective functions is used. The main reason for this increase is the presence of the MPI Reduce function which is responsible for 60 % of this increase. In fact, allowing both the sum and product as reduce operations for 4-bytes and 8-bytes floating-point and non floating-point datatypes required a significant amount of assembly code. In this table, the MPI Barrier primitive was considered a function belonging to the Basic Set because the fundamental functions that start and finish the MPI session use the barrier to synchronize all the processors.

V. IMPLEMENTATION AND RESULTS

The system was implemeted and tested in a Xilinx Zynq-7020 All Programmable SoC FPGA. The Zynq-7020 provides a heterogeneous environment where FPGA programmable logic (Xilinx Artix-7 technology) is connected to an ARM Cortex-A9 based processing system. The software tools used for system prototyping and testing were Xilinx Vivado 2014.4 and Xilinx Software Development Kit (SDK).

The hardware testbed has eight non-cached Microblazes, each one with 32 kB of internal BRAM memory. All the internal data and instructions were stored on these memories. The shared memory was set with 8kB of BRAM memory and a clock frequency of 100 MHz was used for all the programmable logic. This configuration with eight Microblazes utilizes 66 BRAMs (47%) and 10541 LUTs (20%) available on the Zedboard.

To test the implementation of the library, several algorithms with different features were implemented and tested. Fo space reasons, we only present here two of them (1) matrix-vector multiplication and (2) backward substitution.

A. Matrix-Vector Multiplication

The matrix-vector multiplication algorithm was developed taking into account the scarce resources of the FPGA device used. This mathematical operation is simply formalized as Ax = y, where A is the input matrix, x is an input vector and y is the computed output vector.

Due to the limitations of the FPGA in terms of BRAMs, the Microblazes usually cannot store the complete matrix A. To overcome this problem, the processors only store a single row of A and the x vector to compute the corresponding y

TABLE II. EXECUTION TIMES FOR MATRIX-VECTOR MULTIPLICATION.

Multiple uBlaze execution times (ms)					
	Number of uBlazes				
Size of matrix	1	2	4	6	8
50 × 50	22	12	9	7	6
100 × 100	88	47	30	25	23
150 × 150	200	106	65	53	50
200 × 200	357	189	118	97	87
250 × 250	559	296	180	148	137
300 × 300	806	427	264	214	196

TABLE III. SPEEDUPS FOR MATRIX-VECTOR MULTIPLICATION.

Multiple uBlaze speedups				
	Number of uBlazes			
Size of matrix	2	4	6	8
50 × 50	1.8	3.1	4.0	4.0
100 × 100	1.9	3.3	3.9	4.4
150 × 150	1.9	3.3	3.9	4.4
200 × 200	1.9	3.1	4.0	4.5
250 × 250	1.9	3.1	4.0	4.4
300 × 300	1.9	3.2	4.0	4.5

element. At each iteration, the processors receive a new row of A and discard the last one. After all the processors compute their corresponding *y* elements, the entire *y* data is gathered to the master processor.

The execution times (see table II) and speedups (see table III) for single precision data were obtained for different sizes of the matrix A and number of processors.

We have also determined the performance efficiency (obtained performance/peak performance) of the architecture (see table IV)

With two processors the efficiency is above 90 % (the efficiency is only limited by the time spent in communication functions). With more than two processors, the efficiency starts decreasing. The cause of this new effect is the contention verified on the AXI-Interconnect (which uses a round-robin policy in these cases).

B. Backward Substitution

Considering a system of equations defined in the matrix form Ax = b, the backward substitution is the process of solving that system of equations when the matrix A is a triangular superior matrix. The algorithm to solve a problem of this kind is generically described by the C code in Listing 1.

Listing 1. Backward substitution example

```
for (i = n-1; i >= 0; i++)
  b[i] = b[i]/A[i][i];
  for (j = 0; j < i; j++)
    b[j] = b[j] - b[i] * a[j][i];
```

The parallelization of the back substitution is less efficient because of a poor scalability. This poor scalability is owing to the fact that only the internal loop of the algorithm is parallelizable. Therefore, every processor must run all the iterations of the external loop and compute/receive the b[i] value (pivot) of the external loop. The broadcast of this value is a critical point in the parallel algorithm and, because of this fact, the optimized MPI Bcast function had an ideal environment for testing in this algorithm. To complete the parallel algorithm, there is the usual data gathering of the vector *b* to the master processor after the computation. Tables V and VI shows the results obtained with this benchmark.

TABLE IV. SYSTEM EFFICIENCY FOR MATRIX-VECTOR MULTIPLICATION.

Number of uBlazes	Average Eff.	Maximum Eff.
2	0.94	0.94
4	0.73	0.78
6	0.60	0.63
8	0.50	0.51

TABLE V. EXECUTING TIMES FOR BACK SUBSTITUTION.

Multiple uBlaze execution times (ms)					
	Number of uBlazes				
Size of matrix	1	2	4	6	8
100 × 100	46	30	20	18	20
200 × 200	183	116	74	63	70
300 × 300	414	257	161	136	151
400 × 400	738	454	282	263	261
500 × 500	1155	707	437	364	403
600 × 600	1669	1018	627	520	575

TABLE VI. SPEEDUPS FOR FOR BACK SUBSTITUTION.

Multiple uBlaze speedups				
	Number of uBlazes			
Size of matrix	2	4	6	8
100 × 100	1.51	2.30	2.56	2.30
200 × 200	1.58	2.47	2.90	2.61
300 × 300	1.61	2.57	3.04	2.74
400 × 400	1.63	2.64	3.17	2.87
500 × 500	1.71	2.84	3.24	2.89
600 × 600	1.64	2.66	3.21	2.90

We have also determined the performance efficiency (obtained performance/peak performance) of the architecture (see table VII)

The results show that the best performance and maximum speedup was achieved with 6 processors followed by the 8 processors architecture. The high communication/ computation ratio, and consequent contention in the architecture interconnect, and a hard parallelization pattern limit the system scalability for backward substitution.

VI. CONCLUSIONS

The main objectives of the work presented in this document consisted in studying and developing solutions that implement the MPI interface in distributed memory soft-processors on FPGA.

In order to minimize the internal memory size of each processor, the implemented MPI library was intended to be as simple as possible (but compatible with every relevant datatype). The developed footprint of the MPI basic set is approximately 10 kB and the remaining implemented collective functions do not increase significantly that footprint value.

Good performance and efficiencies were achieved for two tested algorithms with distinct communication requirements.

TABLE VII. SYSTEM EFFICIENCY FOR BACKWARD-SUBSTITUTION.

Number of uBlazes	Average Eff.	Maximum Eff.
2	0.80	0.82
4	0.64	0.67
6	0.50	0.53
8	0.34	0.36

Future iterations of this work shall provide hardware improvements in order to minimize the effect of the interconnect contention, like using clusters of processors each with its shared RAM. Extend the compatible MPI functions, like *MPI_Allgather*, *MPI_ISend* and *MPI_IRecv*, to improve portability. Consider the implementation of some functions in hardware, like *MPI_Reduce*.

ACKNOWLEDGMENT

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references PTDC/EEA-ELC/122098/2010 and UID/CEC/50021/2013.

REFERENCES

[1] MPI Forum, http://icl.cs.utk.edu/ftmpi/ [Accessed in September, 2015]

[2] S. Lakshminarayana, S. Gosh, and N.Balakrishnan, "Implementation of MPI over HTTP", in 7th International Conference on High-Performance Computing and Networking, 1999, pp.1299-1302.

[3] E. Marques, F. Martins, V. Vasconcelos, N. Ng and N. Martins, "Towards deductive verification of MPI programs against session types", in Programming Language Approaches to Concurrency- and Communication-cEntric Software, 2013, pp.103-113.

[4] FT-MPI,http://icl.cs.utk.edu/ftmpi/ [Accessed in September, 2015]

[5] MPICH, https://www.mpich.org/ [Accessed in September, 2015]

[6] OpenMPI, https://www.open-mpi.org/ [Accessed in September, 2015]

[7] J. Poole, "Implementation of a Hardware-Optimized MPI Library for the SCMP Multiprocessor", MSc Thesis, 2004.

[8] T. P. McMahon and A. Skjellum, "eMPI/eMPICH: Embedding MPI," in MPI Developers Conference, 1996, pp.180-184.

[9] S. Gao, A. Schmidt, and R. Sass, "Hardware implementation of MPI Barrier on an FPGA cluster", in International Conference on Field Programmable Logic and Applications, 2009, pp.12-17.

[10] R. Brightwell, K. Hemmert, R. Murphy, A. Rodrigues and K. Underwood "A Hardware Acceleration Unit for MPI Queue Processing", in Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, 2005 pp.96-109.

[11] J. A. Williams, I. Syed, J. Wu, and N. W. Bergmann "A Reconfigurable Cluster-on- Chip Architecture with MPI Communication Layer", in Proceedings of 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2006, pp.350-352.

[12] M. Saldana and P. Chow, "TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs", in Proceedings of the 16th International Conference on Field-Programmable Logic and Applications, 2006, pp.1-6.

[13] M. Saldana, A. Patel, C. Madill, D. Nunes, D.Wang, H. Styles, A. Putnam, R. Wittig and P. Cho, "MPI as an Abstraction for Software-Hardware Interaction for HPRCs", in Second International Workshop on High-Performance Recofigurable Computing Technology and Applications, 2008, pp.1-10.

[14] Y. Peng, M. Saldana and P. Chow, "Hardware Support for Broadcast and Reduce in MPSoC", in International Conference on Field Programmable Logic and Applications, 2011, pp.144-150.