In Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia,
Seoul, Korea, October 2006

A Communication Interface for Multiprocessor Signal Processing Systems

Sankalita Saha†, Shuvra S. Bhattacharyya† and Wayne Wolf‡
†Dept. of ECE, University of Maryland, College Park, MD, 20742, USA
‡ Dept. of Electrical Engineering, Princeton University, NJ, 08544, USA
(ssaha,ssb)@umd.edu, wolf@princeton.edu

Abstract

Parallelization of embedded software is often desirable for power/performance-related considerations for computation-intensive applications that frequently occur in the signal-processing domain. Although hardware support for parallel computation is increasingly available in embedded processing platforms, there is a distinct lack of effective software support. One of the most widely known efforts in support of parallel software is the message passing interface (MPI). However, MPI suffers from several drawbacks with regards to customization to specialized parallel processing contexts, and performance degradation for communication-intensive applications. In this paper, we propose a new interface, the signal passing interface (SPI), that is targeted toward signal processing applications and addresses the limitations of MPI for this important domain of embedded software by integrating relevant properties of MPI and coarse-grain dataflow modeling. SPI is much easier and more intuitive to use, and due to its careful specialization, more performance-efficient for the targeted application domain. We present our preliminary version of SPI, along with experiments using SPI on a practical face detection system that demonstrate the capabilities of SPI.

1. Introduction

The use of parallel architectures for applications with high computational intensity is popular as it leads to better performance. Though a variety of work has explored the hardware aspects of parallel architectures, there is a significant lack of adequate software support for these platforms. One of the most important issues with regards to such systems is communication between processors. The most widely-known endeavor in this regard is the message passing interface (MPI) protocol, which is accepted by many as the standard for multiprocessor communication using message passing.

The main advantages of MPI are that it is portable — MPI has been implemented for almost all distributed memory architectures — and each implementation is optimized for the hardware on which it runs. However, MPI is designed for general-purpose multiprocessor applications. Thus, although MPI provides for various features and various forms of flexibility, MPI-based programs cannot leverage optimizations obtained by exploiting characteristics specific to certain application domains.

In this work, we propose a new multiprocessor communication protocol, which we call the signal passing interface (SPI). SPI is specialized for signal processing, which is an increasingly important application domain for embedded multiprocessor software. It attempts to overcome the overheads incurred by the use of MPI for signal processing applications by carefully integrating with MPI concepts from coarse-grain dataflow modeling and useful properties of interprocessor communication (IPC) that arise in this dataflow context. The resulting integration is a paradigm for multiprocessor communication that exchanges the general targetability of MPI for increased efficiency in signal processing systems.

In this paper, we present the preliminary version of our proposed SPI communication interface and implementation and testing of key features of the interface. We restrict ourselves to a special class of dataflow graphs called the synchronous dataflow (SDF) [9] model, which is popular in both commercial and research-oriented DSP design tools. Also, we use the self-timed execution model as the underlying execution model for SDF graphs on multiprocessor systems. This causes streamlining of communication patterns across processors and leads to the following advantages compared to MPI-based implementations.

- Elimination of ambiguities related to multiple messages: In an MPI program, the program developer has to ensure that all message sends are matched by the correct receive operations, otherwise the program may lead to race conditions. The deterministic properties of SDF schedules ensure this ordering inherently, and thus, the developer does not have to explicitly take care of such conditions.

- Separation of communication and computation: In SPI, the communication procedure is implemented as a call to a specialized communication actor (i.e., a specialized vertex in the dataflow graph). All details of the communication procedure are encapsulated within this actor and do not spill over into the main code. This actor is implemented in a library in the same fashion as other dataflow actors are. How the communication takes place can vary based on the implementation. Thus, the implementation can use MPI to perform the communication, or it can use OpenMP if the target is a shared-memory system, and so on. Also, a dataflow-based communication library can be used which would provide higher efficiency. The development of such a library is a useful direction for future work on SPI.

- Better buffer optimization: MPI does not guarantee to buffer arbitrarily long messages. This leads to complex buffering issues. In SPI, this is resolved by the use of buffer synchronization protocols which provide guarantees on buffer sizes and opportunities for buffer size minimization in conjunction with the assumed SDF modeling format.

Although our development of SPI is motivated by properties of signal processing applications, any application that can be represented as an SDF graph and is amenable to self-timed scheduling can be implemented on a multiprocessor system using SPI.

2. Related Work

With an increasing shift towards heterogeneous hardware/software platforms, the need for improved parallel software has been increasing. To this end, the most popular programming paradigm supported by most multiprocessor machines has been the message passing interface (MPI) [6]. And though various APIs for such platforms are becoming available (even commercially) such as OpenMAX which provides abstraction for specialized routines used in computer graphics and related applications, the focus on effective multiprocessor communication interfaces is lacking. Due to various drawbacks in MPI alternative protocols have been proposed such as message passing using the parallel vector machine (PVM) [13], extensions to the C language in unified parallel C (UPC) [3], and in recent years, OpenMP [4] for shared memory architectures. However, all of these software techniques target general-purpose applications and are not tuned towards the signal processing domain. Various attempts to adapt MPI to object-oriented environments such as Java have also emerged (e.g., Jace [1]).

In the domain of signal processing, two well-known modeling paradigms are dataflow graphs and Kahn Process Networks (KPN)[7] (relationships among dataflow modeling, KPNs, and signal processing design tools are discussed in [10]). However, since our work exploits properties of SDF that are not present in general KPN specifications, our preliminary version of SPI is not applicable to general KPN representations. Some tools that employ KPNs actually use restricted forms of KPNs that are more amenable to formal analysis [5] and a promising direction for further study is exploring the integration of SPI with such tools. Multiprocessor implementation of dataflow graphs has been explored extensively [12]. Most of these efforts have concentrated on efficient scheduling of tasks under different models and constraints, including many approaches that take into account costs associated with IPC and synchronization. There is a distinct lack, however, of approaches to the streamlining and standardizing of communication protocols in this domain, and of systematically relating the advances in scheduling-related optimizations to lower-level programming support that can fully exploit these optimizations. This paper addresses this practical gap in the application of dataflow graph theory to multiprocessor signal processing systems, with focus on distributed memory systems.

3. Dataflow-based Modeling of Signal Processing Systems

The format of coarse-grain dataflow graphs is one of the most natural and intuitive modeling paradigms for signal processing systems [9, 12]. In dataflow, a program is represented as a directed graph in which the nodes (called *actors*) correspond to computational modules, and data (called *tokens*) is passed between actors through FIFO queues that correspond to the edges of the graph. Actors can *fire* (perform their respective computations) when sufficient data is available on their input edges. When dataflow is applied to represent signal processing applications, actors can have arbitrary complexity. Typical signal processing actors are finite and infinite impulse response filters, fast Fourier transform computations and decimators.

When dataflow is used to model signal processing applications for parallel computation, four important classes of multiprocessor implementation can be realized: fully static, self-timed, static assignment and fully dynamic methods [8]. Among these classes, the self-timed method is often the most attractive option for embedded multiprocessors due to its ability to exploit the relatively high degree of compile-time predictability in signal processing applications, while also being robust with regards to actor execution times that are not exactly known or that may exhibit occasional variations [12]. For this reason, we target our initial version of the SPI methodology to the self-timed scheduling model, although adaptations of the methodology to other scheduling models is feasible, and is an interesting

topic for further investigation.

In self-timed scheduling for multiprocessor signal processing systems, each actor is assigned at compile-time to execute on a unique processor, and all such actor-to-processor assignments are fixed throughout the execution of the system. Furthermore, for each processor, the order of execution across all actors that are assigned to that processor is fixed at compile-time. However, unlike fully-static scheduling, self-timed scheduling does not fix the times at which actors will execute; the exact time at which an invocation of an actor will begin is determined, in general, by both the execution order of the corresponding processor, and the simple synchronization operations that must be carried out across processors.

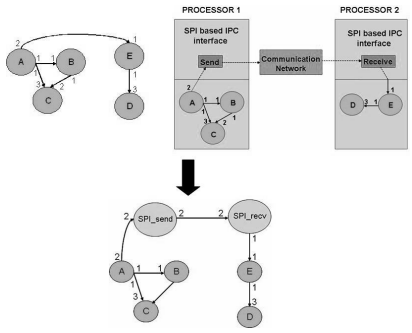

Figure 1. Model for implementing the SDF graph on left on a 2-processor system and the new SDF graph.

4. The Signal Passing Interface

4.1. Overview of the Interface

The FIFO buffers, associated with edges in a dataflow graph, queue *tokens* that are passed from the output of one actor to the input of another. When an actor is executed, it consumes a certain number of tokens from each of its inputs, and produces a certain number of tokens on its outputs. The SDF model is a restricted form of dataflow in which the number of tokens consumed/produced by an actor on each input/output port is constant. For a given dataflow graph, SPI inserts a pair of special actors (called SPI actors) for sending and receiving associated IPC data whenever an edge exists between actors that are assigned to two different processors. We call an edge whose incident actors are mapped to the same processor an *intraprocessor* edge, and an edge whose incident actors are mapped to different processors an *interprocessor* edge. Thus, a multiprocessor implementation of a dataflow graph through SPI involves introducing special SPI actors into the graph after partitioning and allocating them to different processors. To realize this practically, two forms of edge implementations are defined for every edge; one is the *intraprocessor* edge implementation that is implemented as a function call that writes to the (local) edge, while the other is the *interprocessor* edge implementation, which involves a call to the special SPI actor that takes care of the necessary IPC functionality.

In the serial code, the *interprocessor* edges are masked for convenient schedule-based selection. For a multiprocessor implementation, once the partitioning and scheduling of actors on the different processors is complete, the *interprocessor* edges are unmasked and the *intraprocessor* edges are masked accordingly. This enables the actors from the original graph to function independently of the number of processors, schedule, and memory architecture (e.g., whether it is based on shared or distributed memory). It is also a convenient method to explore different multiprocessor schedules – lack of information of both types of edges may lead to exclusion of optimization choices when partitioning and scheduling is done. It can also be used conveniently by both automatic code generation tools as well as parallel code that is written by hand with the aid of dataflow graph modeling. The experiments in this paper were carried out in the latter implementation context (human-derived parallel programs).

The static properties of SDF together with self-timed scheduling provide important opportunities for streamlining of interprocessor communication. This leads to the following differences between SPI and MPI implementations:

- In MPI, a send message contains the destination actor. In SPI, instead, the message contains the ID of the corresponding edge of the graph.
- In SPI, there is no need to specify the size of the buffer as in MPI. Once a valid schedule is generated for the graph, all the edge buffer sizes are known, and hence the edge ID can be used to determine the buffer size. Although buffer overflow conditions may occur if one of the processors produces data at a faster rate than the consumer processor consumes it, these conditions can systematically be eliminated in SPI by buffer synchronization protocols discussed below
- SPI allows only non-blocking calls. Any buffer overwriting or race conditions are avoided by the self-timed execution model that enforces 1) any actor can begin execution only after all its input edges have sufficient data, and 2) each actor is implemented on one processor only and not distributed across multiple processors.

Figure 1 shows an example of the implementation of a sample SDF graph on two processors, and the resulting new dataflow graph. The edge buffers for the actors of the interprocessor edges are shared with the respective SPI actors within a processor. After an actor A finishes execution, it

writes its output data to each output edge FIFO buffer e if the actor C that consumes data from e is implemented on the same processor. If actor C is implemented on a different processor, actor A writes to the edge shared between it and the corresponding SPI actor. The SPI actor is then invoked which initiates a *send* call to actor C. Completion of a *send* implies acknowledgement that the message has reached the receiving processor. If an acknowledgment is not received within a fixed time period, a *re-send* is performed.

We define two protocols to manage buffers for SPI based on several related factors. These protocols build on the existing BBS (bounded buffer synchronization) and UBS (unbounded buffer synchronization) protocols for synchronization in statically-scheduled, shared memory multiprocessor systems [2]. For certain edges, called *feedback edges*, it can be ensured from analysis of the self-timed schedule and an associated intermediate representation called the *synchronization graph* that the buffer size cannot exceed a fixed value, and using a buffer of this size will always prevent overflow [2]. Intuitively, a *feedback edge* is an IPC edge that is contained in one or more directed cycles of the synchronization graph. In such cases the efficient BBS protocol is applicable, while if an edge is not a feedback edge, or if it is a feedback edge but a buffer of the required size cannot be allocated, then the less efficient UBS protocol is employed.

To these considerations, we add the communication reliability factor and define a new set of protocols and associated software interfaces. Therefore, in our case, buffer overflow on an IPC buffer can occur due to any or a combination of the following reasons: (a) a producer actor consistently fills up the buffer (e.g., because it is on a faster or less-loaded processor) faster than the corresponding consumer actor can consume the data; (b) the varying communication time of the channel; and (c) multiple sends due to an unreliable communication channel. When we say that buffer overflow does not happen, it implies that none of the above conditions will occur. In both protocols we assume that two copies of the edge FIFO buffer are maintained — one in the sender SPI actor's memory and another in the receiver SPI actor's memory with a statically determined upper limit. Let S_a and R_a be actors in the original dataflow graph, while S and R be the corresponding special interprocessor communication actors that are inserted by SPI based on the given self-timed schedule. A read (rd) and write (wr) pointer are maintained by both the sender and the receiver.

4.2. SPI-BBS and SPI-UBS Protocols

If it can be guaranteed that a buffer will not exceed a predetermined size, then the SPI-BBS protocol is used. Let the maximum buffer size on both S and R be B_{bb}. Then S_a writes a token to the buffer, and increments the write pointer as $wr = (wr + 1) mod B_{bb}$. Once the number of tokens written into the write buffer is equal to the production rate of the corresponding actor, S sends a message (*non-blocking send*) to R containing the new data tokens. When S receives an acknowledgment for the send message, it modifies the read pointer. Upon receiving the message, R modifies its buffer, sets the write pointer accordingly, and sends an acknowledgement to S. R_a reads a token when $rd \neq wr$. After reading a token, R_a modifies the read pointer as $rd = (rd + 1) mod B_{bb}$.

The SPI-UBS Protocol is used when it cannot be guaranteed statically that an IPC buffer will not overflow through any admissible sequence of send/receive operations on the buffer. In this protocol, an additional counter of unread tokens (ur) is maintained. Let the buffer size on both S and R be B_{ub}. Before S_a writes to the buffer it checks if $ur \neq B_{ub}$. If yes, it writes to the buffer and increments the write pointer as in SPI-BBS, and also increments the count ur. Then S sends a message (*non-blocking send*) to R from offset rd. When S receives an acknowledgment for the *send* message, it modifies the read pointer and decrements the count of ur. Upon receiving the message, R checks if $ur \neq B_{ub}$. If yes, it modifies its buffer, sets the write pointer accordingly and increments the value of ur. However, R does not send an acknowledgement to S immediately. R_a reads a token when $ur \neq 0$. It reads the token from offset rd and modifies the read pointer as $rd = (rd + 1) mod B_{ub}$. The value of the count ur is decremented and acknowledgement for the *send* from S is sent.

In case of the receiver buffer being full, the receiver does not *send* an acknowledgement which causes the sender to *re-send* again. For the extreme case when the unreliability of the communication medium causes loss of acknowledgement from the receiver, special care has to be taken. For such cases, a tag may be used that flips between 1 and 0 for alternate messages. Then R checks the tag to ensure receipt of the correct message: if the tag is the same as the earlier message then the current message is due to a *re-send* and is not written into the buffer, instead an acknowledgment for the message is sent.

5. Implementation and Results

In this section, we describe an initial implementation of the SPI framework, and present experimental results observed from this implementation with a face detection application as a test bench. The underlying communication layer used is MPI. The current SPI library implements three functions for interprocessor communication — SPI_init, SPI_send and $SPI\_receive$ which are sufficient to provide communication and synchronization functionality for any multiprocessor implementation:

- $SPI\_init(schedule)$: Here, $schedule$ is the multiprocessor schedule for the SDF graph. In this function

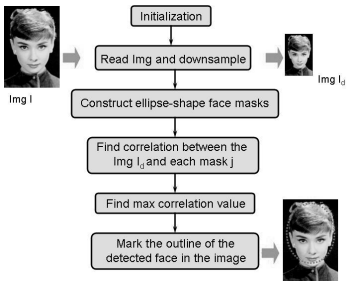

Figure 2. The flow of the face detection algorithm used in our experiments.

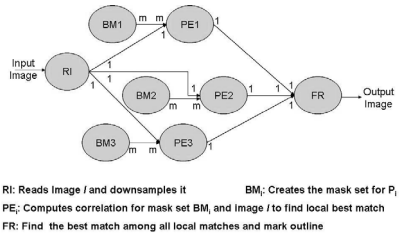

Figure 3. SDF graph of the face detection system for 3-processor parallelization.

initialization of parameters and buffer sizes for SPI is done. Also, assignment of buffer sizes for each interprocessor edge is performed and stored. The BBS or UBS protocol setting for the edges are done as well.

- $SPI\_send(data, datatype, edge\_id)$: Here, $data$ denotes a buffer that stores the data to be sent, $datatype$ denotes the type of data, and $edge\_id$ denotes the ID of the *interprocessor* edge. This function for the current implementation contains a call to a non-blocking *send* of MPI.
- $SPI\_receive(data, datatype, edge\_id)$: Here, $data$ denotes a buffer that stores the data to be received. The rest of the arguments are same as in SPI_send. In this function, instead of a non-blocking *send*, a non-blocking MPI *receive* function call is done.

5.1. Implementation Details

Image processing applications such as face detection are good candidates for parallelization as they are computation-intensive, and at the same time, inherently parallelizable. In this work, a shape-based approach proposed by Moon et al. [11] is used. Figure 2 shows the complete flow of the employed face detection algorithm. At the implementation level, this reduces to finding out correlations between a set of ellipse-shaped masks with the image in which a face is to be detected. Profiling results show that this operation (mask correlation) is computationally the most expensive operation. However, this operation is parallelizable. The mask set can be divided into subsets and each subset can be handled by a single processor independent from the rest. A coarse-grain dataflow model of the resulting face detection system is shown in Figure 3 for the case of 3 processors for mask correlation. Besides using multiple processors for the correlation operation, we use a separate processor to handle the required I/O operations.

Once the actor assignment is completed, SPI-based parallel code can be derived by unmasking the function call to the SPI actor for every edge, if the edge is an interprocessor edge. We use the SPI-UBS protocol, since fixed buffer sizes cannot be guaranteed for any of the edges of the given dataflow graph. We assume reliable communication, since the actual communication in our experiments is performed using MPI, which assumes reliable communication. Experimentation with the unreliable communication aspects of SPI is a topic for further work.

5.2. Results

Three versions of multiprocessor implementation of the face detection system were developed — Version 1 uses basic MPI ; this version kept the communication code simple but did not use non-blocking calls everywhere, Version 2 uses non-blocking calls only but with more lengthy communication code being nested within the computation code, and Version 3 is the SPI implementation. The targeted multiprocessor platform consisted of a combination of IBM Netfinity and Dell Poweredge hardware nodes: each node was a dual-processor PIII-550 (2xPIII-550Mhz) with 1GB memory and 2x18GB of disk capacity. Each node had a 1Gigabit link to the other nodes. Two test benches involving 126 and 114 masks, each of size 121x191 and 127x183 pixels, and image of size 128x192 pixels were used.

Comparison of performance (of individual processors) for the three implementations are illustrated in Table 1. As can be observed from this comparison, the SPI version performs significantly better than the first MPI version for all of the cases. Furthermore, the SPI version performs as well as the second MPI version, but produces smaller and simpler communication code compared to MPI, as illustrated in Figure 4. Comparing the MPI Version 1 (blocking calls only) and SPI code segments in this figure, we observe that the SPI implementation leads to better performance in terms of smaller communication header size. Also, the MPI Version 2 (non-blocking calls only) and SPI code segments in the same figure highlight that the SPI implementation uses fewer communication library calls than the MPI version in order to achieve comparable performance.

Table 1. Execution times for MPI and SPI implementations for 6 processors.

processor ID	Testbench 1(secs)			Testbench 2(secs)		
	MPI(1)	MPI(2)	SPI	MPI(1)	MPI(2)	SPI
0	1.55	1.55	1.54	1.419	1.379	1.38
1	1.46	1.44	1.43	1.337	1.295	1.296
2	1.41	1.36	1.36	1.346	1.297	1.296
3	1.48	1.44	1.43	1.346	1.293	1.29
4	1.48	1.44	1.43	1.356	1.296	1.296
5	1.42	1.37	1.37	1.351	1.311	1.31

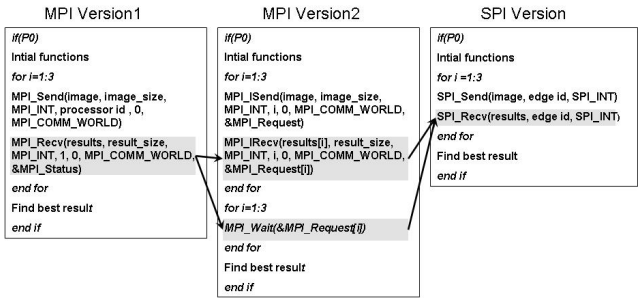

Figure 4. Abridged code showing communication code for 3 implementations of the face detection system

Thus, SPI provides an interface that leads to simple communication code with performance that is comparable to optimized MPI code. It resolves the issues of multiple *send*s and buffer management as well, thereby overcoming some of the drawbacks of MPI without affecting its advantages. In addition, SPI is retargetable: the actual communication layer can be implemented using other communication interfaces, such as OpenMP. An SPI communication library that implements the SPI communication primitives independently would provide for standalone operation, further efficiency improvement, and performance optimization, and is an important direction for future work.

6. Conclusions

In this paper, we presented the preliminary version of the signal passing interface (SPI), a new paradigm for implementing signal processing software on embedded multiprocessors. SPI achieves significant streamlining and associated advantages by integrating with the MPI framework properties of dataflow graph modeling, synchronous dataflow analysis, and self-timed scheduling. In our proposed SPI framework, protocols for interprocessor communication and efficient buffer management have been defined. Experiments with a face detection application have been conducted to demonstrate the capabilities of SPI. Our experiments showed that this application can be parallelized using the SPI framework in a simple and efficient manner. Future work includes implementation of an independent SPI library that supports the SPI communication primitives, and exploration of other optimization possibilities in the context of SPI.

References

[1] J. Bahi, S. Domas, and K. Mazouzi. Jace: A java environment for distributed asynchronous iterative computations. In *Proc. of 12th Euromico Conf. on Parallel, Distributed and Network-Based Processing*, pages 350–357, 2004.

[2] S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Minimizing synchronization overhead in statically scheduled multiprocessor systems. In *Proc. of Int. Conf. on Applicn. Specific Array Processors*, pages 298–309, Jul 1995.

[3] W. W. Carlson, J. M. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA/CCS.

[4] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, pages 46–55, 1998.

[5] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen. Affine nested loop programs and their binary cyclo-static dataflow counterparts. In *Proc. of the Intl. Conf. on Application Specific Systems, Architectures, and Processors*, Steamboat Springs, Colorado, September 2006.

[6] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, pages 84–90, 1996.

[7] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress*, 1974.

[8] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *IEEE Global Telecommunications Conf. and Exhibition*, pages 1279–1283, Nov 1989.

[9] E. A. Lee and D. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Trans. on Computers*, Feb 1987.

[10] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proc. of the IEEE*, page 773, May 1995.

[11] H. Moon, R. Chellappa, and A. Rosenfeld. Optimal edge-based shape detection. *IEEE Trans. on Image Processing*, 11:1209, 2002.

[12] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[13] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, pages 531–545, 1994.