A Communication Library for Mapping Dataflow Applications on Manycore Architectures

Mingkun Yang, Süleyman Savas, Zain-ul-Abdin, and Tomas Nordström
Centre for Research on Embedded Systems (CERES),
Halmstad University, Sweden.
Emails: minyan09@student.hh.se, {Suleyman.Savas, Zain-ul-Abdin, Tomas.Nordstrom}@hh.se

***Abstract*—Dataflow programming is a promising paradigm for high performance embedded parallel computing. When mapping a dataflow program onto a manycore architecture a key component is the library to express the communication between the actors. In this paper we present a dataflow communication library supporting the CAL actor language. A first implementation of the communication library is created for Adapteva's manycore architecture Epiphany that contains an on-chip 2-D mesh network. Three different buffering methods, with and without direct memory access (DMA) transfer, have been implemented and evaluated. We have also made a preliminary study on the effect of mapping strategies of the actors onto the cores. The assessment of the library is based on a CAL implementation of a two dimensional inverse discrete cosine transform (2D-IDCT) and our own CAL-to-C compilation framework. As expected the results show that the most efficient actor-to-core mapping strategy is to keep the communication to the nearest-neighbor communication pattern as much as possible. Thus, the best way to place a pipelined sequence of computations like our 2D-IDCT is to place the actors into cores in a serpentine fashion. For this application we found that the simple receiver side buffer outperforms the more complicated buffering strategies that used DMA transfer.**

I. Introduction

The computational requirements of high-performance embedded applications, such as video processing in HDTV, baseband processing in telecommunication systems, and radar signal processing, all have reached a level where they cannot be met with traditional computing systems based on general-purpose digital signal processors. The conventional approach of using advanced architectural techniques in uni-processors such as branch prediction, out-of-order execution, and superscalar, in addition to the frequency scaling is reaching its limit due to the increased power dissipation and complexity. Manycore architectures consisting of tens or hundreds of processing cores offer the possibility of meeting the growing performance demand in an energy-efficient way by exploiting parallelism instead of scaling the clock frequency of a single powerful processor [1].

Unfortunately, the traditional programming languages, due to their sequential control flow, are struggling to utilize the emerging many-core architectures. The focus of these sequential languages is to provide abstractions for algorithm specification, but the abstractions, intentionally, do not say much about how they are mapped to underlying hardware. As a result, new languages based on parallel and concurrent programming paradigms have emerged, and the dataflow programming model seems to be a good candidate. However, applying the dataflow programming model on shared memory manycore architectures is a challenging task, as there are no explicitly defined mechanisms for communicating between the processing cores.

In this paper, we present a dataflow communication library that facilitates the mapping of applications developed in the dataflow programming model onto the manycore architectures. We use the CAL dataflow language [2] to program an emerging manycore architecture, namely Epiphany[3]. We describe three slightly different variants of the implementation of communication mechanisms adopted by the library and evaluate the results in the case study of two-dimensional inverse discrete cosine transform (2D-IDCT).

II. Background

In this section, we provide the background information about the manycore architecture that is used in this work, together with the background information about dataflow programming and the CAL language.

A. Epiphany

Adapteva's manycore architecture Epiphany [4] is a two-dimensional array of processing cores connected by a mesh network-on-chip. Each core has a floating-point RISC CPU, a direct memory access (DMA) engine, memory banks and a network interface for communication between processing cores. An overview of the Epiphany architecture can be seen in Figure 1. In the Epiphany

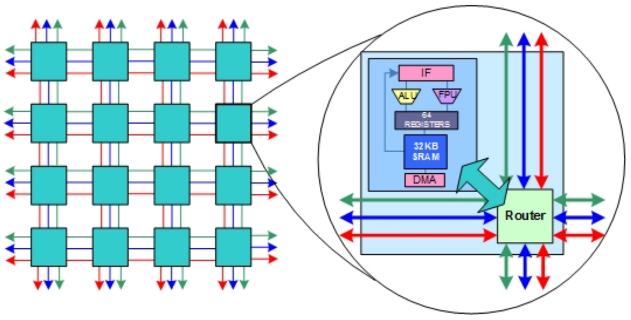

Fig. 1: Epiphany architecture overview.

architecture each core is a superscalar, floating-point, RISC CPU that can execute two floating point operations and a 64-bit memory load operation on every clock cycle. The cores are organized in a 2D mesh topology with only nearest-neighbor connections. Each core contains a network interface, a multi-channel DMA engine, a multicore address decoder, and a network-monitor. The on-chip node-to-node communication latencies are 1.5 clock cycles per routing hop, with zero startup overhead. The network consists of three parallel networks which are used individually for writing on-chip, writing off-chip, and all reading requests, respectively. Due to the differences between the networks, writes are approximately 16 times faster than reads for on-chip transactions. The transactions are done by using dimension-order routing (X-Y routing), which means that the data first travels along the row and then along the column. The DMA engine is able to generate a double-word transaction on every clock cycle and has its own dedicated 64-bit port to the local memory. The Epiphany architecture uses a shared memory model with a single, flat address space. Each core has its own aliased, local memory range which has a size of 32 kB. The local memory of each core is accessible globally from any other core by using the globally addressable IDs. However, even though all the internal memory of each core is mapped to the global address space, the cost (latency) of accessing them is not uniform as it will depend on the number of hops and contention in the mesh network.

B. *Dataflow Programming*

Dataflow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations or actors that operate on the dataflow. Dataflow programming emphasizes the movement of data and models programs as a series of connected actors. Each actor only communicates through explicitly defined input and output connectors and functions like a black box. As soon as all of its inputs become valid, an actor runs asynchronous from all other actors. Thus, dataflow languages are inherently parallel and can work well in large, decentralized systems. Dataflow programming is especially suited for streaming applications, where data is processed as a continuous stream, as, e.g., video, radar, or base station signal processing.

A number of dataflow languages and techniques exist [5], [6], and one interesting modern dataflow language is CAL (Cal Actor Language) [2], [7]. Which has recently been used in the standardization of MPEG Reconfigurable Video Coding (RVC) [8].

A CAL dataflow program consists of stateful operators, called actors, that transform input streams of data objects (tokens) into output streams. The actors are connected by FIFO channels and they consist of code blocks called actions. These actions transform the input data into output data, usually with the state of the actor changed.

C. *CAL Compilers*

CAL compilers have already been targeting a variety of platforms, including single-core processors, multicore processors, and programmable hardware. The Cal2C compiler [9] targets the single-core processors by generating sequential C-code. The Open-RVC CAL Compiler (ORCC) [10] generates multi-threaded C-code that can execute on a multicore processor using dedicated run-time system libraries. Similarly the d2c [11] compiler produces C-code that makes use of POSIX threads to execute on multicore processors.

In this paper, we will target manycore architectures and will base our work on an in-house CAL compiler that generates separate C-code for each actor instance so that it could be executed on individual processing cores. Therefore we do not require any run-time system support for concurrent execution of actors, in contrast to ORCC and d2c. The contribution of this paper lies in describing and evaluating a communication library that supports the communication between actors that are allocated to different cores in a manycore architecture.

III. The dataflow communication library

All the communication that is done between the actors is done through FIFO buffers, thus making this functionality a key component for the compilation of the applications developed in CAL onto a manycore architecture. We suggest to implement this functionality as a dataflow communication library, with only five basic functions. In addition to the traditional functions of `write` and `read` from the FIFO buffer we have added functions such as `connect` which logically connects two actors, `disconnect` which logically disconnects the actors, and finally a function `end_of_transmission` that flushes the buffer and indicates that there are no further tokens to be sent.

When implementing these buffers on the Epiphany architecture, two special features of this architecture need to be considered. First one is the speed difference between read and write transactions (as mentioned earlier, writes are faster). The second one is the potential use of DMA to speed up memory transfer and allowing the processor to do processing in parallel with the memory transfer.

We have investigated three ways to implement the FIFO buffering. The first implementation is a ‘one-end-buffer’ which places the buffer inside the input port in the destination core. (Putting the buffer on the source core would result in reads instead of writes and thus a tenfold slowdown.) The communication overhead resides completely in the sender.

If we want to use DMA, we need to have a buffer to both the sender and receiver side. In the second implementation (‘two-end-buffer’) each core performs read and write transactions on its local memory and then uses DMA to transfer the data. This transfer is performed when both sides are ready, which requires that the sender’s buffer is full and the receiver’s buffer is empty. Even though we are using DMA for data transfer, the processor will be busy waiting for the DMA to finish. This is obviously not very efficient, and this method should be seen as a transition to our third method.

To allow the DMA to work in parallel with the processing core, we have implemented a ‘double-two-end-buffer’ method, which introduces two “ping-pong” buffers on each side of the communication channel. This allows the cores to work on one local buffer while the data from the other buffer is transferred to the other core by means of DMA.

If the token production rate on the sending actor is equivalent to the token consumption rate on the receiving actor, it is expected that the ‘double-two-end-buffer’ method should be the most efficient. On the other hand, if there is a big imbalance in the production/consumption rate, all three buffering methods will suffer from blocking, after the buffer gets full.

We have implemented the broadcast capability in all the three implementations of the communication API. In all implementations, the synchronization between sender and receiver is achieved by polling the observed buffer directly. Because of this remote access a lot of traffic is generated during busy waiting. One possible optimization to reduce the traffic is to create local mirrors for the variables we are polling. In other words, the polling core should use the local mirrored version instead of the original variables in the remote core. Obviously, the observed core needs to update the mirrors in addition to the original variables, which introduces noticeable overhead. The possible benefit of this optimization will be analyzed in our investigation below.

The `write` and `read` function calls are designed to be *asynchronous* (non-blocking calls) by default, and they will be blocking only on buffer full and buffer empty respectively. The functions `end_of_transmission`, `connect`, and `disconnect` calls will always be blocking.

A. *Implementation Details*

In order to implement the `write` and `read` function calls with the use of DMA, we have implemented four internal support functions. The functions ‘`try_flush`’ and ‘`do_flush`’ are the two basic functions that are intend for asynchronous and synchronous function calls, respectively. For details see [12].

Figure 2 shows three consecutive states of the sender side buffer in the ‘two-end-buffer’ implementation, when using the ‘`try_flush`’ asynchronous function call. The red arrows indicate the next free slot,that will be occupied in the next write operation. The scenario begins with the buffer having only one free slot, as shown in Figure 2a. When one new token is generated and put into this buffer by calling the write function, the state of buffer will change to that shown in Figure 2b and the asynchronous function is called before updating the index pointer. Since this is an asynchronous function, the caller exits without blocking and the resulting buffer state is shown in Figure 2c. This function call will start a DMA transaction if the following three requirements are met: the source buffer is full, the destination buffer is empty, and there is an idle DMA channel. Otherwise it will exit immediately. The '`do_flush`' synchronous function is used to ensure that we do not overwrite the data that has not been transferred to the destination core yet. This function is called when the buffer is full and the core tries to send one token. Since it is possible that DMA has not yet started because any of the three requirements are not met, or alternatively DMA has been started but has not finished yet, this synchronous version has to take care of both of these cases so that it does not perform the same task more than once.

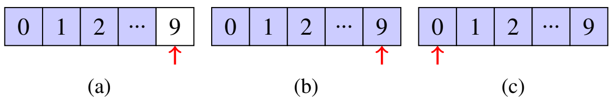

Fig. 2: The consecutive state of one buffer, when there's only one empty slot initially. One write function call will fill the empty slot with the token, and asynchronous function is called, at (2b), right before updating the write pointer.

Similar to the '`try_flush`' and '`do_flush`' functions, there are the '`try_distribute`' and '`do_distribute`' functions, which are implemented to transfer data in a broadcast manner from a single output port to multiple destination input ports. The current implementation for iterating the multiple destination ports is done in a sequence starting from the first till the last. The implementation of the two functions, used for distributing the tokens, varies in the case of 'two-end-buffer' and 'double-two-end-buffer' methods. The '`try_distribute`' is called asynchronously when one of the buffers is full, as shown in Figure 3a and on each subsequent write operation to the alternate buffer, as shown in Figure 3b and Figure 3c.

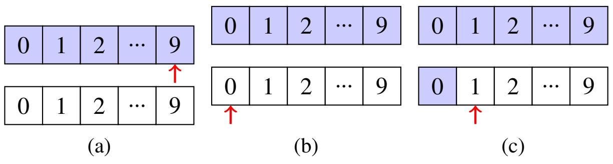

Fig. 3: The state of the buffers when the asynchronous function call is used and there are still available slots in the buffers.

IV. Results and Discussion

As our case study we chose the two-dimensional inverse discrete cosine transform (2D-IDCT), which is one component of MPEG standard video decoders. We use a 64,000 input data sample for one execution of complete application. Additionally, we also evaluate the performance with a minimal test case consisting of only the first 64 samples. This minimal test case allows us to remove certain buffering effects. The CAL implementation we are using consists of 15 actor instances mapped to 15 out of the 16 cores in the Epiphany chip (E16G301), which is executing at 400 MHz. This implementation of 2D-IDCT uses two one-dimensional inverse discrete cosine transforms in sequence, with all actor instances connected in a pipeline fashion.

Layout	Implementation's average execution time (ms)					
	v1	v2	v3	v1o	v2o	v3o
row-order	614	622	624	615	615	619
serpentine	612	619	620	612	613	618

TABLE I: Total execution time of 2D-IDCT for 64,000 samples using different layout and different implementations.

The total execution time for 64,000 samples using various implementations of our library is shown in Table I. Each value in the table is the average of 10 consecutive runs. The columns 'v1', 'v2', and 'v3' denote the three implementations: 'one-end-buffer', 'two-end-buffer', and 'double-two-end-buffer', respectively. The next three columns 'v1o', 'v2o', and 'v3o' reflect the optimization when polling local copies of synchronizing variables. The memory footprint of the three buffering methods is kept equal, thus the buffer size is 100, 50, 25 words for 'v1', 'v2', 'v3', respectively. We have also evaluated this application using other buffer sizes (ranging from 10 to 200). However, for this application the buffer size was not found to significantly affect the execution time.[1]

The two rows in Table I represent two different mapping layouts. In the 'row-order' layout actors are assigned to cores sequentially, *i.e.*, the actor sequence 0 to 15 is mapped to core id 0 to 15, thereby ignoring the fact that, *e.g.*, core 3 is not the nearest neighbor to core 4. On the other hand, the 'serpentine' layout takes into account the physical layout of the cores on the chip and the X-Y routing used, so that consecutive actors are mapped into neighboring cores. In this case, the actor sequence 0 to 15 is mapped to core ids {0-3, 7-4, 8-11, 15-12}, taking into consideration the fact that cores 3 and 7, as well as cores 11 and 15, are nearest neighbors to each other. Our results indicate that the 'serpentine' layout slightly outperforms the 'row-order' layout in all implementations. However, this result might not be universally applicable as our 2D-IDCT algorithm has a very specific communication pattern. Interestingly we only see improvement for the optimized methods, which poll local copies of synchronizing variables, for the DMA based buffering methods ('v2' and 'v3'). It seems that when using the 'one-end-buffer (v1o)', the overhead of maintaining local copies outweighs the benefit it introduces.

To further analyze the behavior of our library, Figures 4 - 6 present for each core the number of clock cycles spent in the communication library (API) in relation to the total number of clock cycles. The clock cycles spent on reading and writing to the external memory are not included in the number of cycles spent in the communication library.

Figure 4 shows the number of clock cycles corresponding to the execution times given in Table I. In this figure we see that a significant amount of the total time is spent in library calls. This unexpected behavior can be explained by the bottleneck in the last node, which is writing to the (slow) external memory. Thus, the clock cycles are not spent on reading from or writing to buffers, but instead waiting due to a full buffer. The full buffer at the last node leads to backward pressure until all nodes are affected.

To remove the effect of this "backpressure" in our further analysis, we evaluated our implementation using only 64 samples, and the results are shown in Figure 5 and 6, with or without extern memory access, respectively. The largest difference between the results in these two figures can be found at the first (core 0) and last (core 13) actor as they are interacting with external memory. Due to the better resolution of the total execution time in the last two figures,

[1]The raw data can be found at https://github.com/albertnetymk/com_lib_data

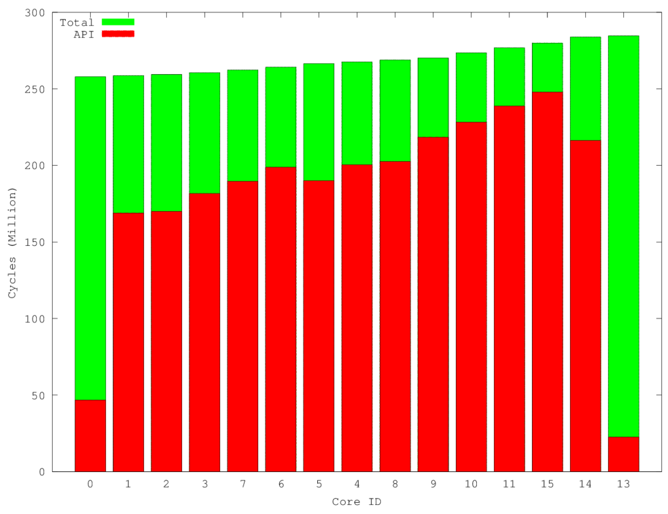

Fig. 4: Number of clock cycles spent in the communication library (API) in relation to the total number of clock cycles for each core (one-end-buffer optimized (v1o) implementation with buffer size 100 using 64000 input data tokens read from external memory).

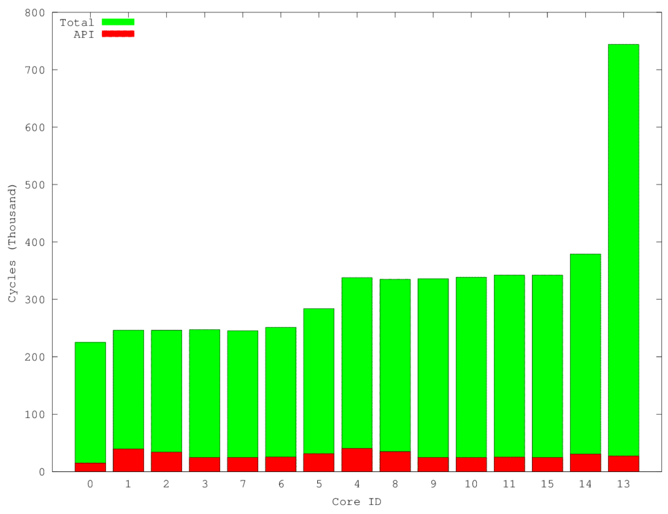

Fig. 5: Number of clock cycles spent in the communication library (API) in relation to the total number of clock cycles for each core (one-end-buffer optimized (v1o) implementation with buffer size 100 using 64 input data tokens read from external memory).

V. Future Work

In the future it will be necessary to run and evaluate our library on additional applications, especially those with more complex communication patterns. We would also like to explore methods to optimally map actors onto cores based on the application communication graph and a communication architecture description. To the dataflow communication library we intend to add some synchronization functions, supporting both ”core to core” synchronization and ”host with chip” synchronization. In the current library the method to distribute tokens to multiple receivers is done in a strict sequence and a more optimized order needs to be investigated.

VI. Conclusion

In this paper, we have introduced a new dataflow communication library supporting a message passing interface for dataflow languages.

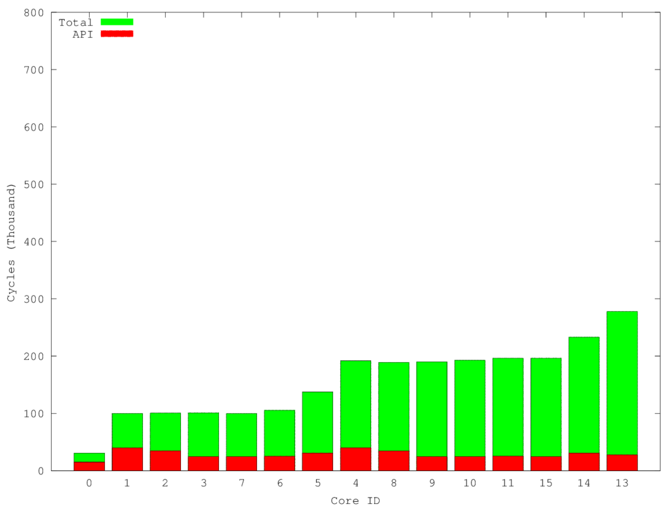

Fig. 6: Number of clock cycles spent in the communication library (API) in relation to the total number of clock cycles for each core (one-end-buffer optimized (v1o) implementation with buffer size 100 using 64 input data tokens read from local memory).

We have implemented and evaluated various buffering approaches using the 2D-IDCT application on Adapteva's Epiphany manycore architecture. The overall best buffering method was found to be the 'one-end-buffer' method with a single receiver buffer. The more advanced buffering methods trying to utilize DMA transfer did not perform as well for the selected application. However, it cannot be excluded that applications with more complex communication patterns could benefit from the 'double-two-end-buffer' method. In addition we have shown the benefit of carefully mapping the actors onto cores so that nearest-neighbor communication can be utilized.

References

[1] S. Borkar, “Thousand core chips - a technology perspective,” in *DAC 2007, San Diego, California, USA*, 2007, pp. 746–749.

[2] J. Eker and J. W. Janneck, “CAL language report specification of the CAL actor language,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M03/48, 2003.

[3] L. Gwennap, “Adapteva: More flops, less watts,” Microprocessor Report, 6/13/11-02, Tech. Rep., 2011.

[4] Epiphany, “Epiphany architecture reference G3, rev 3.12.12.18,” Adapteva, Tech. Rep., 2013.

[5] E. A. Lee and D. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[6] B. Bhattacharya and S. S. Bhattacaryya, “Parameterized dataflow modeling for DSP systems,” *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.

[7] J. Eker and J. W. Janneck, “Dataflow programming in CAL – balancing expressiveness, analyzability, and implementability,” in *Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), 2012*. IEEE, 2012, pp. 1120–1124.

[8] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, “Overview of the MPEG reconfigurable video coding framework,” *Journal of Signal Processing Systems, Springer*, 2009.

[9] G. Roquier, M. Wipliez, M. Raulet, J.-F. Nezan, and O. Déforges, “Software synthesis of CAL actors for the MPEG reconfigurable video coding framework,” in *15th IEEE International Conference on Image Processing, 2008. ICIP 2008*. IEEE, 2008, pp. 1408–1411.

[10] ORCC, “Open RVC-CAL compiler,” http://orcc.sourceforge.net/, Accessed: 3 Aug 2013, 2013.

[11] D2C, “CAL ARM compiler,” http://sourceforge.net/projects/opendf/, Accessed: 3 Aug 2013, 2013.

[12] M. Yang, “CAL code generator for epiphany architecture,” Master Thesis, Halmstad University, 2013, in preparation.