Efficient Handling of Arrays in Dataflow Process Networks

Daniel Baudisch, Jens Brandt and Klaus Schneider
Embedded Systems Group
Department of Computer Science
University of Kaiserslautern
http://es.cs.uni-kl.de

Abstract—Dataflow process networks (DPN) have been proposed as a programming model for distributed parallel systems that have communication paths with unpredictable latencies. The purely data-driven execution of DPNs does not require a global coordination and therefore allows one to easily map DPNs to many parallel hardware and software architectures with distributed memories. Problems due to shared memory communication like data races do not exist since the communication is done via point-to-point FIFO buffers between the process nodes. On the other hand, this distributed programming model leads to high communication costs if large data structures like arrays have to be communicated between nodes. This paper addresses the reduction of the communication costs due to arrays that are processed by – and therefore sent between – DPN nodes. The presented methods are given at a high level of abstraction and do not impose specific constraints on the used target architectures. Our experimental results show clearly the advantage of our approach compared to the standard message passing between the nodes.

I. INTRODUCTION

A. Model-based Design of Embedded Systems

Compared to the design of traditional software systems, the design of embedded systems is much more difficult since in addition to the functional correctness, further non-functional constraints like the consideration of energy consumption and computation within certain real-time bounds have to be fulfilled. Moreover, one has to deal with heterogeneous and application-specific processors. For this reason, model-based design of embedded systems become more and more popular, where code is synthesized from models at high levels of abstraction. Simulation and verification can be done at the high-level models, while code generation is done automatically and often guided by additional means to assure correctness-by-construction, like code certification of verified code generation.

For example, our research group has developed the Averest1 system where the model-based design starts with the synchronous programming language Quartz [34]. The execution of a synchronous program is divided into macro steps that correspond with single interactions of the system with its environment. Each macro step consists of micro steps that are atomic actions, and the order of the micro steps within a macro step is purely determined by their data dependencies. Since each variable that is written in a macro step must be read by micro steps only after it has been written, the programming model gives the programmer the view as if the execution of micro steps is done in a parallel zero-time execution.

1http://www.averest.org

Synchronous languages have many advantages [8]. For example, they lend themselves well for formal verification since (1) most model checking procedures have been defined for synchronous transition systems and since (2) synchronous languages have formally defined semantics. Simulation and debugging benefits from the determinacy of synchronous programs which allows one to replicate once observed behaviors. Moreover, checking the worst case execution time of macro steps is relatively simple since macro steps do not contain loops and the finite number of micro steps is known at compile time. Finally, one can easily generate synchronous digital hardware circuits as well as single-threaded software from synchronous programs.

B. From Synchronous Systems to Distributed Systems

Synthesis of multithreaded software that should run efficiently on distributed hardware or software architectures with distributed memories is not straightforward from synchronous languages. Implementing the synchronous model in a direct way only makes sense if macro steps are large enough to amortize the required synchronization overhead due to the threads. For typical synchronous programs, this is however not the case, so that a naive synthesis of multithreaded software will lead to poor performance results.

For this reason, we developed translations from synchronous models to distributed models of computation, in particular, to dataflow process networks [17, 24, 28, 29] which are a well-understood distributed model of computation. These translations must not only deal with the different model of computation, i.e., the change from cycle-based to data-driven computation, but also with the change from a shared memory model to a message-passing model: A dataflow process network (DPN) thereby consists of a set of process nodes that communicate with other process nodes via point-to-point FIFO buffers. Each process node operates independently of the other nodes so that there is no need for a global coordination. Instead, the execution is purely data-driven, i.e., a node can perform an execution step if sufficient input data is available for that step. If a node performs an execution step, it consumes the required input values from its input buffers and produces output values in its output buffers. In the pure DPN model, it is assumed that the process nodes do not have local memories and implement therefore state-less functions from input to output values.

DPNs have many advantages for the code generation of heterogeneous multiprocessors with distributed memories. A particular advantage is that for the implementation of a
process node’s function, there are neither restrictions on the programming language nor on the hardware to implement process nodes. Another advantage of DPNs as a basis for code generation for distributed systems is that all parallelism that can be statically exploited from an application is explicitly represented by a DPN. Furthermore, the FIFO-buffers in a DPN (1) precisely describe the data dependencies between nodes, and thereby, they (2) are flexible enough to be implemented by different communication protocols (e.g., latency insensitive protocol (LIP) [15], synchronous elastic flow (SELF) protocol [16], or simple FIFO buffers in hard- or software).

Although DPNs look pretty simple, their correct and efficient implementation has to consider several difficult problems. The first problem is to ensure boundedness of memory: The size of the buffers is typically unbounded, so one has to avoid that there is a FIFO buffer where more data values are produced than consumed in the long run. To this end, additional means like static scheduling (if possible) [10, 11, 19, 26, 27] or the introduction of acknowledgements to introduce back-pressure are required to assure bounded memory requirements.

Another problem is to ensure the determinacy of the DPN: Even though each process node implements a function that maps input values to corresponding output values, it is not the case that the entire DPN must also be deterministic, i.e. a function from input streams to output streams, as well. Similar to data races on shared memory systems, the behavior of a node may depend on the timing when tokens on buffers arrive, e.g. caused by varying timing of schedules of producer nodes. For this reason, one has to ensure that each node implements a continuous function which can be done by simple additional requirements as in Kahn’s non-blocking reads [24]. Our automatically synthesized DPNs provably satisfy these additional requirements.

For this reason, we are interested in the automatic translation of DPNs from synchronous programs. In our previous work, we have already shown how synchronous programs can be compiled to synchronous guarded actions [34, 35]. These guarded actions are partitioned into classes of actions writing to the same variable. Obviously, this makes sense since each output buffer of a node corresponds to a variable of the original program, and its writing node must be uniquely determined. To achieve coarser grained nodes, one can collapse nodes into larger nodes, and we already experimented with partitions to generate pipelines [5] and partitions to define multiple threads with as few inter-thread communication as possible [4, 6, 7].

C. Efficient Use of Arrays in DPNs

While these synthesis methods work well for scalar data, the DPN programming model becomes expensive when compound data types have to be dealt with. For example, compound data types are trees, lists, dictionaries, or what is more important for embedded applications: arrays. Recall that process nodes implement mathematical functions; thus, they map input values to output values. Consequently, if a DPN deals with arrays, then the nodes writing values to and reading values from the array have to send the entire array from one node to the other, which clearly leads to high communication costs. Note that it is no solution to augment the DPN with a shared memory, since this will lead in most cases to a non-deterministic behavior due to the process nodes’ independent executions (reading from and writing to shared data values will then probably suffer from data races).

In this paper, we therefore consider the problem to reduce the communication costs in DPNs that deal with arrays. The main idea is thereby that we define for each array of the given program one process node (called the writer of the array) in the DPN that is responsible for all write updates of the array. Each node reading the array will maintain a local copy of the array and will receive from the writer of the array the corresponding updates. While the array may be large, the updates typically contain only a few assignments to array elements instead of the entire array. Depending on the application, the presented approach automatically reduces the amount of data that has to be sent through a DPN’s communication channels. It is not difficult to see that our proposed algorithm is correct, so that we are able to automatically translate synchronous programs to deterministic DPNs having this optimized use of arrays.

Typical examples of applications that benefit from this approach are all kinds of embedded systems that deal with arrays. In particular, this covers multimedia applications or, more general, applications that deal with digital signal processing. It is also not hard to see that the approach can be applied to other non-scalar data types as well, but our experimental results are currently limited to arrays, since these are the most important compound data types in embedded applications.

The paper is organized as follows: Section II lists some related work, Section III lists some preliminary notations, and Section IV explains our method to reduce the communication costs of arrays between nodes. The benchmarks and the evaluation of the results are described in Section V and are followed by our conclusions in Section VI.

II. RELATED WORK

The reduction of communication costs in DPNs has been already addressed by many papers: For example, [21, 39] mainly consider the elimination of redundant messages. Similar to our approach, they are working on dataflow graphs, but their source is a structured program consisting of loops and if-statements. Their optimization is based on the introduction of so-called ‘Section Communication Descriptors’ and ‘Availability Section Descriptors’ which are obtained from the analysis of loops in the source program. Restricted to the consideration of loops, their approach does not target the communication optimization in general DPNs. In addition, Tims et. al. [36] consider reduction of communication in clusters using the distribution interrelationship flowgraph (DIF), which is an extension of the control flow graph of the source program. Their approach fully concentrates on the dynamic distribution of arrays in a structured SPMD programs consisting of a sequence of several loops working on arrays that are parallelized to nodes in a cluster. In contrast, our approach is based on generic DPNs, where the nodes contain different code and are not the result of loop parallelization.

A demand-driven optimization has been proposed in [30, 31] by Arvind. The basic idea is to generate and send data only, when it is requested from an output (similar to lazy evaluation in functional programming languages). Clearly, this approach does not try to reduce communication costs, but
rather to avoid unnecessary computations at all, while keeping the communications as in the original DPN.

The problem to deal with compound data types in DPNs was heavily discussed in the construction data flow computers like the Monsoon computer and their related programming languages like ID or VAL [1–3, 37]. To this end, Arvind presents in [1, 2] special memories that were called I-structures to ensure the single-assignment rule to each variable. These memory cells were endowed with a special protocol to avoid reads before writes and to avoid overwriting a once generated value. I-structures were refined to M-structures in [3] to allow re-use of memory cells, which was not possible with I-structures.

The problem also exists in functional programming languages whose virtual machines also suffer from unnecessary copy operations. For example, adding a single element to a list causes duplication of all elements in the source list. For this reason, some functional programming languages like F# introduced mutable data types which is — in the context of several threads — nothing else than shared memory. Hence, it has to be dealt with care due to potential data races.

Software creation from DPN descriptions, particularly CAL [18], has already been considered in [38]. The language CAL supports the usage of arrays only as local/state variables of nodes. Interaction between nodes by sending arrays is not supported. Hence, communication of arrays and optimization must be manually implemented without any compiler support.

III. PRELIMINARIES

As already explained in the introduction, we propose a model-based design of embedded systems that starts with a high-level system description given in the synchronous programming language Quartz [34]. The synchronous programs are translated to an intermediate representation based on synchronous guarded actions that are then translated to dataflow process networks. In this section, we explain some details of this design flow as far as needed to explain the core of this paper, which is the reduction of communication costs of DPN nodes that work with array types.

```plaintext
module AudioDelay(\( N \)} \{ \text{i: input memorized nat \ a: local memorized \( [N] \}\) nat \( a[j] \) \; // store the last \( N \) values of \( i \) \( \text{nat}(j) \) \; // array indices \}
\text{loop} \{ 
\text{p: pause; \quad if(c): a[j] = i; \quad next(j) = (j+1) \% N; \quad j0 = (j-delay) \% N; \quad o = a[j0]; \}
\}
Figure 1. Example Module AudioDelay as Quartz Program.

A. Synchronous Systems

Synchronous programming languages [8, 22] like Esterel [9], Lustre [22] or Quartz [33–35] became popular for the design of reactive systems. Their computation is divided into macro steps that correspond with single interactions of the system with its environment. In each macro step, the synchronous system reads all inputs, and determines values for all outputs as well as a next internal state to resume the computation of the next macro step. For example, Figure 1 shows a Quartz module with inputs \( i, \text{delay}, \text{and c} \) and a single output \( o \).

In Quartz programs as e.g. the one shown in Figure 1, macro steps are separated from each other by means of the \text{pause} statement that also declares a control flow location which is called \( p \) in Figure 1. Thus, module AudioDelay executes the code below its \text{pause} statement in each macro step.

By the synchronous programming model, it is assumed that the code between two \text{pause} statements is executed in zero-time. For code generation, this means that the atomic actions in such a macro step have to be ordered such that all writes to a variable are performed before that variable is read. Thus, the order of the micro steps within a macro step is purely determined by their data dependencies. The compiler therefore has to check whether such an execution is always possible and has to generate the code accordingly.

```plaintext
system AudioDelay:
\text{interface:}
\begin{align*}
\text{i: input memorized nat} \quad & \text{delay: input memorized nat} \\
\text{c: input memorized bool} & \text{o: inout memorized nat} \\
\text{locals:}
\begin{align*}
\text{p: label bool} & \quad \text{a: local memorized \( [N]\) nat} \\
\text{j: local memorized nat(0)} & \quad \text{j0: local memorized nat(0)} \\
\end{align*}
\text{init:}
\begin{align*}
\text{control flow:}
\text{True} & \Rightarrow \text{next}(p) = \text{True} \\
\text{data flow:}
\begin{align*}
p & \Rightarrow \text{next}(p) = \text{True} \\
p & \Rightarrow \text{data flow:} \\
\text{ckp} & \Rightarrow a[j] = i \\
p & \Rightarrow \text{next}(j) = (j+1)N \\
p & \Rightarrow j0 = (j-delay)N \\
p & \Rightarrow o = a[j0]
\end{align*}
\end{align*}
\}
Figure 2. Guarded Actions of Module AudioDelay.

In our Averest system, we therefore compile the synchronous programs into synchronous guarded actions (SGAs) which are the atomic micro steps of a macro step. Each guarded action is of the form \( \gamma \Rightarrow \alpha \) with a boolean condition \( \gamma \) called the guard and an atomic action \( \alpha \). In each macro step, all guards of the guarded actions are evaluated, and since we have synchronous guarded actions, all actions whose guards are evaluated to true have to be executed.

In the following, we will assume that all guarded actions are either immediate assignments \( \gamma \Rightarrow x = \tau \) or delayed assignments \( \gamma \Rightarrow \text{next}(x) = \tau \). In both cases, the right hand side expression \( \tau \) is evaluated in the current variable environment, and the value is assigned to the left hand side \( x \). In immediate assignments, this transfer to the left hand side \( x \) is done in the same macro step, while in delayed assignments this is done only at the beginning of the next macro step.

We have already explained in the introduction that synchronous languages have many advantages for a model-based design of reactive embedded systems like their effective use in formal verification, deterministic simulation, and synthesis with correctness-by-construction guarantees for single-threaded software and synchronous digital hardware circuits (see [8, 12, 22, 23, 25]). Current research focuses on the synthesis of multi-threaded software [5, 6, 32] and so-called globally asynchronous locally synchronous (GALS) systems that consist of synchronous cores that work asynchronously together. For both research directions, we envision the use of translations to data flow process networks that we describe in the next section in more detail.
B. Dataflow Process Networks

Data-flow process networks (DPN) have been proposed as a parallel programming model for distributed parallel systems [17, 24, 28]. DPNs consist of a set of process nodes that are connected with each other by unbounded FIFO buffers. Each process node of the DPN consumes data values from its incoming buffers and produces values that are put into its output buffers. The behavior of each node should therefore be a mathematical function mapping input values to output values.

Process nodes can be invoked whenever there are sufficiently many data values in their input buffers. This way, they act independent of each other, and just have to consider the input buffers to decide whether an execution step can be performed. Different possible computations steps are often described in tables by so-called firing rules. They consist of an input pattern and a corresponding action that consumes values from the input buffers and produces values for the output buffers.

DPNs are therefore a perfect programming model for distributed systems while synchronous systems typically work on shared memory. Since we are interested in the synthesis of multithreaded software as well as of distributed embedded systems in general, we therefore considered translations of synchronous guarded actions to DPNs [4–7]. To this end, all guarded actions writing to the same variable are grouped into a DPN node. This node produces the values that a variable will have during the computation of the synchronous system. A more coarse grained partitioning can be achieved by merging nodes as described in [4].

Considering our running example of Figures 1 and 2, we obtain the DPN shown in Figure 3 (we have omitted trivial guards that are true, which also pertains for label p in the example in hand). Note that the lines connecting the process nodes represent FIFO buffers. The orientation of the edges depicts the direction of the data flow in these buffers, i.e., they are pointing from the node that produces tokens to the node that consumes tokens.

Recall also that in DPNs there is always exactly one producer and one consumer per FIFO buffer. The reader may have noticed that node A reads and writes variable j and that also nodes B and C read j. A copy of the initial value of j is in each buffer from A. Each time a value from the buffer from A to itself is consumed, it is incremented and three copies are produced in the buffers connecting A with B, A, and C.

C. Inefficiency due to Compound Data Types like Arrays

As already mentioned in the introduction, efficient communication between DPN nodes is one of the main issues in mapping DPNs to concrete target architectures. If the data values that are send between the process nodes are only of small scalar types like integers, etc., it is no problem to send and receive them via FIFO buffers. However, if the data types are large as in case of arrays, this becomes a major problem for the efficient implementation of DPNs. Recall that it is no solution to augment the DPN with a global shared memory because the independent execution of DPN nodes will suffer from data races that lead to nondeterminism. Another solution would be to automatically synthesize locks to shared memories which is also currently a research topic for weak memory models (see e.g. [13, 14, 20]), but very different to the approach considered here.

Consider again our example in Figure 3: Each time, node C is fired, one element in a is written, so that array a is modified. For this reason, the complete array a must be sent to node D by writing array a in the FIFO buffer that is represented by the edge between nodes C and D, such that D can read the correct element a[j] that must be sent to the environment.

Even this very simple example makes clear that this implementation is obviously unsatisfactory. In this example, we could simple merge nodes C and D to a single node to avoid the communication of the array. This is clearly a reasonable solution for this example, but in general this would impose hard restrictions for the partition of guarded actions to DPN nodes. We will discuss in the following section, which is the core of this paper, an algorithm that transforms the given DPN into another one that avoids the communication of the entire array, but does not modify the topology of the DPN.

IV. REMOVING ARRAYS FROM COMMUNICATION

Having explained the problem, we consider in this section an algorithm that can be used to reduce communication costs of arrays in DPNs that are generated from synchronous guarded actions. We first describe the main idea with the help of our running example, and then present the algorithm that automatically generates the optimized DPN. To this end, we have to introduce some notation to make the algorithm and its explanation more readable.

A. Main Idea

The overall idea presented in this paper is that arrays are no longer sent through the FIFO buffers. Instead, we distinguish between the (unique) writer of an array, and the readers of the array. Both the writer and the readers maintain local copies of the array, but only the writer holds the code for updating the array as defined in the guarded actions. Clearly, it will update
its local copy by executing the code as before, but instead of sending the entire updated array to all readers, it will only send the required information to update the local copies of the readers.

For example, an accordingly optimized version of the DPN of Figure 3 is shown in Figure 4. Nodes C and D have been modified as follows: whenever a array must be updated, a signal valid_a is used. Node C then generates values valid_a, idx_a for the index of an element of a that must be overwritten by value rhs_a. As before, node C performs this update a[idx_a] = rhs_a on its local copy a. Note that the computation of idx_a and rhs_a are typically more complex than in the considered example.

Instead of sending the updated value of a to node D, node C will now only send the values idx_a and rhs_a, so that node D will first update its local copy a[idx_a] of a by executing a_a[idx_a] = rhs_a. The previous code to produce output o is changed accordingly, i.e., the value is now taken from the local array a_a instead of the now unknown array a.

### B. Formal Foundation of the Transformation

After the informal description of the idea of our transformation, we will next define an algorithm to optimize the array communication of DPNs that have been obtained from synchronous guarded actions (SGAs): We consider only local variables for optimization since only local variables are communicated between nodes.

In the following, let A be the set of the DPN nodes and let L be the set of local variables of the DPN, i.e. all buffers except for buffers connecting the DPN with its environment. The guarded actions of a DPN’s node n are denoted as behavior(n). Conversely, owner(A) is the name of a node that contains an action A (the node is uniquely determined since we partition the original set of guarded actions as DPN nodes).

We distinguish between an array a and its elements a[0],...a[N−1] that we also call (memory) cells of the array. The variable that is addressed by an expressions e is obtained by function varOfCell, i.e. varOfCell(a[0]) = varOfCell(a[i+1]) = a. Finally, ID(e) denotes a set containing the index expression of an expression e unless this is a constant, e.g. ID(a[i]) = {i}, ID(a[0]) = {}. ID(i) = {}.

Next, we define functions to get read and written variables of synchronous guarded actions in Definition 1. Note that the definition of rdVars(A) must consider the left-hand side of assignment A, which may contain variables that are read, i.e. indices in an array access.

**Definition 1 (Read and Write Variables).** Let \( \text{Vars}(\tau) \) denote the variables occurring in the expression \( \tau \). Then, the variables read and written by a SGA are defined as follows:

- \( \text{rdVars}(\gamma \Rightarrow x = \tau) := \text{Vars}(\gamma) \cup (\bigcup_{\delta \in ID(\delta)} \text{Vars}(\delta)) \cup \text{Vars}(\tau) \)
- \( \text{wrVars}(\gamma \Rightarrow x = \tau) := \text{varOfCell}(x) \)

Based on \( \text{rdVars}() \) and \( \text{wrVars}() \), we can define the set of actions and nodes that read from, respectively write to a variable in Definition 2. Note that, according to the definition of DPNs, a variable can only be written by at most one node, hence \( |\text{writer}(v)| = 1 \) holds.

**Definition 2 (Reading and Writing SGAs and DPN nodes).** The set of SGAs/DPN nodes reading and writing to a variable v is determined as follows:

- \( \text{rdActs}(v) := \{A | v \in \text{rdVars}(A)\} \)
- \( \text{wrActs}(v) := \{A | v \in \text{wrVars}(A)\} \)
- \( \text{reader}(v) := \{n \mid \text{behavior}(n) \cap \text{rdActs}(v) \neq \emptyset\} \)
- \( \text{writer}(v) := \{n \mid \text{behavior}(n) \cap \text{wrActs}(v) \neq \emptyset\} \)

Thus, \( \text{reader}(v) \) is the set of nodes reading variable v, while \( \text{writer}(v) \) is the singleton set of nodes that writes values to variable v.

**Definition 3 (Writing SGAs in DPN node).** The set of SGAs in a node n writing to variable v is determined as follows:

- \( \text{W}_{A}(v, n) := \text{wrActs}(v) \cap \text{behavior}(n) \)

Moreover, for any guarded action \( A = (\gamma \Rightarrow x = \tau) \), the expression \( \gamma(A) \) denotes its guard, \( x(A) \) denotes its left-hand side expression x, and \( \tau(A) \) denotes its right-hand side expression \( \tau \). Finally, let \( \text{ReplaceVar}(A, v, v') \) be the function that renames variable v to \( v' \) in the given actions A, i.e. all occurrences of variable v are replaced by the expression \( v' \).

### C. Array Communication Reduction Algorithm

The reduction of communication of arrays between nodes is done by function RMArrayCom shown in Figure 5. Function RMArrayCom thereby determines first the set of array variables \( L' \) that have to be communicated from their writer to a reader (different to the writer). An improved algorithm may consider also whether a single variable is worth the
transformation depends on the size of the array and the computations made for its updates. We then call function RMArrayComOfVar(v) to remove the array communication of array variable v. This call generates further variables that are collected and finally returned by RMArrayCom.

Each call RMArrayComOfVar(v) starts with the creation of a local copy v\textsuperscript{\text{local}} of the array v in each reader node n (line 3–7). Then, each occurrence of v is replaced by the corresponding access to the local copy v\textsuperscript{\text{local}} of the node n. Since neither v nor one of its copies is communicated between nodes anymore, the algorithm has to change the set of actions that describe the behavior of the concerned nodes. In general, it is possible that a node reads its written variables. Hence, we must ensure that a copy of v is only added for, respectively renamed in reading nodes that do not write to v, i.e. n must be excluded from reader(v) (line 4, 6).

The remaining part of RMArrayComOfVar adds the transportation of changes by adding update tuples and corresponding actions. To ensure correctness, each change will be sent using a separate tuple containing the change. We use the synchronous MoC as an example, where all changes to the copy v\textsuperscript{\text{local}} of v may be evaluated at the same time. The parallel evaluation involves a potentially parallel access to the data structure that is used to transport changes. Hence, to avoid races, each change is stored in a separate update tuple. Depending on the current MoC of the nodes, this might be improved further, e.g. in a sequential MoC, all changes could be sent using a single buffer. This might reduce the required number of buffers, and therefore, the memory requirements. However, the reduction of used buffers will barely effect the speed, since the required bandwidth stays unchanged.

Each update tuple consists of a valid flag, an index and the value. The valid flag valid indicates whether an update has to be made. If the valid flag is set, the index ID identifies the element that has to be changed, and \tau contains the value that must be assigned to the copy of v.

In addition, our implementation omits the communication of array writes if the written cell already contains the value to be assigned. In particular, for each assignment to the copied variable, a tuple for the change is created (line 10). The first action (line 12) stores the state, whether the assignment to v\textsuperscript{\text{local}} has to be executed - and therefore, to be communicated to reading nodes. The following two actions (line 13, 14) copy the right-hand side expression and the index of the element to the corresponding members of the tuple. The remaining part of RMArrayComOfVar updates contains the value. The valid flag indicates whether an update has to be made. If the valid flag is set, the index ID identifies the element that has to be changed, and \tau contains the value that must be assigned to the copy of v.

In addition, our implementation omits the communication of array writes if the written cell already contains the value to be assigned. In particular, for each assignment to the copied variable, a tuple for the change is created (line 10). The first action (line 12) stores the state, whether the assignment to v\textsuperscript{\text{local}} has to be executed - and therefore, to be communicated to reading nodes. The following two actions (line 13, 14) copy the right-hand side expression and the index of the element to the corresponding members of the tuple. The following two actions (line 13, 14) copy the right-hand side expression and the index of the element to the corresponding members of the tuple.

Accordingly, each reader obtains a similar action to commit the change to its local copy of v (line 18, 19). The action has to be inserted using the insertCC(behavior(n), A) function. The concrete implementation of insertCC depends on the MoC of n. The easiest implementation is the one for the synchronous MoC, where the ordering of the actions does not matter because all actions have to be evaluated at the same time. Hence, insertCC for the synchronous MoC is defined as behavior(n) = behavior(n) ∪ A. The implementation for sequential languages, must consider the ordering of instructions. In particular, the actions A must be executed before any other action that might read the targeted cell is executed. Thus, A must be inserted at the beginning of behavior(n). An ordering of updates does not have to be considered because we assumed that a memory cell is written at most once in each iteration of a node.

V. Benchmarks

The algorithm described in the previous section has been implemented on top of our Averest framework. The following benchmarks were written in the synchronous programming language Quartz and compiled to DPNs as described in [4]. In addition, the communication reduction as presented in this paper has been applied to measure its effect. Finally, the DPN was translated to C code for running on SMP systems. To this end, the compiler translates each node to a separate thread of the Posix Pthreads library. Communication is done using two different implementations of FIFO buffers: the first one is taken from Intel’s Thread Building Blocks library. The second one is a custom queue implementation, that is tweaked for our DPN purposes, particularly each buffer has a unique writer and a unique reader thread.

To evaluate our approach, we considered the following benchmarks: Delay is the already presented running example of this paper. The implementation of an audio delay represents a typical implementation of ring buffers which can be also used as bounded FIFO queues. The buffer size of the Delay benchmark is given in milliseconds (ms) and based on 44.1kHz sampling frequency. MatrixMul is a sequential matrix multiplication that has been resource optimized, i.e. it requires at most one multiplication and one adder per iteration. It represents a typical hardware-optimized program that is translated to software for testing and validation purposes. Pitchshift is an optimized DFT-IDFT transformation to change the pitch of an audio stream without changing the playback speed. Finally, the Landscape Renderer is an example for a graphics application: it renders a 3D image of a height field (see Figure 8).
The speed-up of the execution times are shown in Figure 6. In addition, Figure 7 shows a plot of the execution times of the Delay benchmark on the Dual Xeon machine. The times of the unoptimized version increase with the size of the ring buffer. This was expected because the complete buffer must be sent between nodes. In contrast, the execution time of the optimized version of this benchmark remains constant since it is independent of the buffer size. The number of changes per step are constant, and since only changes are communicated, the execution time is no longer influenced by the buffer size. As a result, the speedup of this benchmark increases with growing size of the buffer. The times of this benchmark on the i5 behave similar to that of the Xeon and were omitted to keep the plot readable.

MatrixMult shows the weakest speedup but reaches a quadratic mean of 1.63. Due to the synchronous model of computation, the program has to read two matrices and to write one matrix in each iteration. The optimization is only applied to the local communication, which is only a small part. Hence, the speedup is heavily influenced by the non-optimized environment communication.

Similar to Delay, the Pitchshift uses also a ring buffer and has several operations working on it, i.e. changes must be communicated to several nodes. Hence, the optimization has a more distinct influence on the speedup in this application.

While the Landscape Renderer can render all rows in a picture in parallel, each row requires some sequential drawing operations. In particular, several steps are required to render the picture which changes at most one pixel per line in each step. The results obtained for this benchmark clearly confirm the expected speed-ups: While the size of the rendered picture increases proportional to the amount of data that has to be sent in the unoptimized version, the amount of data in the optimized version scales with the number of actions working on the array. As a consequence, the speed-up scales proportionally. The results made on the i5 seem to be irregular compared to the results of the Xeon. In this case, the results of the different queue implementations drift away, e.g. the speed-ups in row “Landscape Rend. (H=800)” are expected to be about twice as much as in row “Landscape Rend. (H=400)”, which is only fulfilled by the Intel TBB concurrent bounded queue. We think that this irregularity comes from the queue implementation whose data alignment significantly influences the cache behavior of this processor in this example.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Speedup on i5-750</th>
<th>Speedup on 2xXeon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>custom queue</td>
<td>Intel TBB queue</td>
</tr>
<tr>
<td>Delay (max. 50ms)</td>
<td>53.29</td>
<td>19.02</td>
</tr>
<tr>
<td>Delay (max. 100ms)</td>
<td>98.19</td>
<td>39.90</td>
</tr>
<tr>
<td>Delay (max. 200ms)</td>
<td>189.43</td>
<td>66.30</td>
</tr>
<tr>
<td>Delay (max. 500ms)</td>
<td>343.85</td>
<td>208.37</td>
</tr>
<tr>
<td>MatrixMult (16x16)</td>
<td>1.14</td>
<td>1.35</td>
</tr>
<tr>
<td>MatrixMult (32x32)</td>
<td>1.88</td>
<td>1.47</td>
</tr>
<tr>
<td>MatrixMult (48x48)</td>
<td>1.56</td>
<td>1.53</td>
</tr>
<tr>
<td>Pitchshift</td>
<td>809.00</td>
<td>289.00</td>
</tr>
<tr>
<td>Landscape Rend. (H=400)</td>
<td>8.35</td>
<td>3.96</td>
</tr>
<tr>
<td>Landscape Rend. (H=800)</td>
<td>34.73</td>
<td>4.66</td>
</tr>
<tr>
<td>Landscape Rend. (H=1600)</td>
<td>36.41</td>
<td>20.26</td>
</tr>
</tbody>
</table>

Figure 6. Experimental results for four benchmarks (Delay, MatrixMult, Pitchshift, and Landscape Rendering). Each column shows the speed-up of the version obtained by removing array communication compared to its non-optimized version.

Figure 8. Output of the landscape benchmark, which renders a 3D image of a height field.

VI. Conclusions

In this paper, we have shown that we can efficiently reduce communication costs in data flow process networks that deal with arrays. In contrast to the pure DPN paradigm, we do not send arrays between nodes. Instead we maintain local copies in nodes that either read or write the array. One of the nodes called the writer node of the array is responsible for computing all the updates of the array that are then sent to the reader nodes instead of the entire array. The reader nodes will then also update their local copies of the array before performing their original actions. Our transformation can be applied automatically to DPNs generated from synchronous guarded actions and require no manual interaction. The benchmarks confirmed the expected increase of performance of the DPNs by the proposed transformation.

References


