Recursive Task Generation for Scalable SDF Graph Execution on Multicore Processors

Georgakarakos, Georgios; Lilius, Johan

Published in:
Proceedings 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing PDP 2020

DOI:
10.1109/PDP50117.2020.00037

Published: 01/03/2020

Recursive Task Generation for Scalable SDF Graph Execution on Multicore Processors

Georgios Georgakarakos, Johan Lilius
Faculty of Science and Engineering
Åbo Akademi University, Turku, Finland
firstname.lastname@abo.fi

***Abstract*—Dataflow modelling is a popular technique for describing parallel algorithms. Using dataflow, algorithm parallelism can be modelled and analysed efficiently at a high level of abstraction. However, challenges arise when translating dataflow semantics into executable code, mainly due to scheduling and synchronization overheads. Invoking task programming models in order to generate efficient code from dataflow representations has been proposed as a promising methodology to optimise the translation process.**

In this paper, we propose recursive task execution as an optimisation for the dataflow-based code generation process. Our approach is based on extracting synchronous dataflow graph information in order to reduce scheduling overheads and improve load balancing when executing task-based code on multicore processors. We use PREESM dataflow-based prototyping framework to implement and test our concept. Results show that our proposed optimisation enhances code scalability therefore enabling higher application throughput.

***Index Terms*—Dataflow, Task Programming Model, Multicore Scheduling**

I. Introduction

The immense development of highly parallel multicore architectures has empowered the realisation of very high performance applications. Modern multiprocessor chips integrate a high number of cores of variable type and elaborate interconnection schemes. This complexity in architecture level has promoted the usage of model-based design where algorithm description and analysis is decoupled from its implementation in a specific hardware platform.

Dataflow Models of Computation (MoC) [1] have gained popularity as an efficient methodology in order to describe algorithms that typically feature high level of parallelism and streaming flow of data. In dataflow, computation is partitioned in atomic units called *actors*, connected with edges or *buffers* that represent the flow of data across them. Additionally, no central control operation is required for the model to work. Those attributes enable flexible representation of e.g. signal processing algorithms. However, for frameworks that support algorithm implementation based on dataflow modelling [2], [3] , translation of dataflow semantics in executable code for multicore targets is not trivial due to scheduling overheads.

The correlation of dataflow and task programming models has been proposed in order to optimise the code generation process [4]. Dataflow-based code generation using tasks minimises job synchronisation overheads. However while actor-to-task assignment preserves the atomic attribute of an actor, it creates a bottleneck for the work stealing scheduler that constraints scalability. This is more evident in cases of applications based on algorithms with fine grain parallelism.

In this paper we propose an optimisation of the dataflow-based code generation process using recursive task generation. We utilise dataflow graph information so that the actor-to-task granularity is better suited for the task stealing scheduler. We use PREESM [5], a dataflow-based prototyping framework to implement and evaluate the task recursion effect in the generated code performance.

The rest of the paper is organised as follows: Section II presents the concepts used in our work and summarizes previous related work. Section III showcases the contribution of task recursion in code generation from dataflow graphs. In section IV, experimental evaluation of the generated code in multicore processors is presented and discussed. Section V concludes the paper.

II. Background-Previous Work

A. Synchronous Dataflow Model

Our work utilises a popular class of dataflow models, Synchronous Dataflow (SDF) [6]. In order for an application to be described using an SDF graph, it is divided into a number of nodes called *actors* which are connected with directed first-in first-out (FIFO) buffers. During its execution, an actor in the graph consumes and produces data samples called *tokens*.

Formally, an Synchronous Dataflow (SDF) graph $G = (\mathbb{V}, \mathbb{E})$ is a tuple of a finite set of actors $\mathbb{V}$ acting as vertices and a finite set of FIFO buffers $\mathbb{E}$ as directed edges between the actors. A FIFO buffer $b \in \mathbb{E}$ connects the output of some actor $A \in \mathbb{V}$ to actor $B \in \mathbb{V}$. All the FIFO buffers $b \in \mathbb{E}$ are connected to some actor in $\mathbb{V}$. A FIFO buffer, $b \in \mathbb{E}$, connecting $A, B \in \mathbb{V}$ has a production rate $\rho_b \in \mathbb{N}$, a consumption rate $\kappa_b \in \mathbb{N}$ and a delay $\delta_b \in \mathbb{N}$.

The important attribute of an SDF graph is that the production and consumption rate of tokens for an actor are fixed [7] thus actors can be scheduled at compile time. The quantity $\mathbf{q}(A)$ is called the repetition factor of $A \in \mathbb{V}$. For schedulable SDF graphs with no cyclic paths, the repetition factor expresses the data parallelism of the graph. Both task-level and data-level parallelism of the actors in the graph are identified by constructing a Directed Acyclic Graph (DAG) from the SDF graph. Formally, a DAG is a tuple, $DAG = (\mathbb{V}_{dag}, \mathbb{E}_{dag})$, where $\mathbb{V}_{dag}$ consists of $\mathbf{q}(A)$ instances of A, $\forall A \in \mathbb{V}$. $\mathbb{E}_{dag}$ is a set that consists of edges between all the instances of the actors in G.

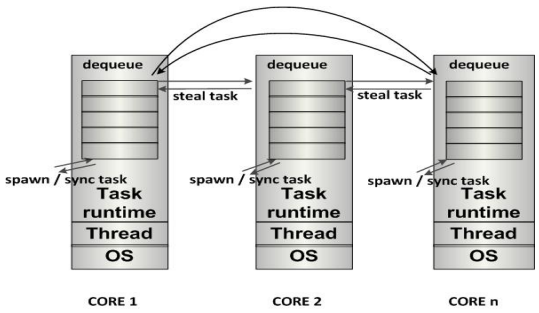

Fig. 1. Work stealing in task programming models.

In this work, we focus on particular SDF graphs that include uniform repetition count (URC) subgraphs. An SDF graph has a URC subgraph if there is a subset of actors $\mathbb{Z} \subseteq \mathbb{V}$ so that $\forall z1, z2 \in \mathbb{Z}$, $z1 \neq z2$, $\mathbf{q}(z1) = \mathbf{q}(z2)$. A consequence of URC is that $\forall z1, z2 \in \mathbb{Z}$, each instance $z1_i$ is connected to a unique instance $z2_j$ when $\delta_b \mod \rho_b = \delta_b \mod \kappa_b = 0$.

B. Task Programming Models

Task programming models [8], [9], [10], have evolved as an optimisation of traditional multithread-based programming. Their main novelty is the notion of atomic run-to-completion entities, *tasks*. Task semantics include a) *spawn*, the extraction of the task entity from the main function in order to execute asynchronously from it and b) *sync*, the return of the control to the calling function after a task has finished execution. Task scheduling and execution details are then decoupled from application development and delegated in the task runtime.

Task models often feature runtime schedulers like *work-stealing* [11] that perform dynamic load balancing among cores. In work stealing another task semantic, *steal*, is defined: a core pushes and pops tasks from the bottom of its dequeue while other cores can steal a task from the top of the same dequeue when they are idle, as shown in Figure 1.

Initially, the task programming model targeted data-parallel applications (nonetheless in [9] and [12] task-based runtime systems that support a more generic flow-based style of execution are presented). Therefore, a *divide-and-conquer* task management is possible to be used where multiple task instantiations of a function (or *kernel*) operate on a data chunk. This can be effectively implemented using recursive task execution. Task recursion combined with work stealing makes load balancing easily adaptive to the available processing resources.

C. Task-based Code Generation in PREESM

PREESM is an open source dataflow-based prototyping environment for analysis and integration of parallel signal processing applications in multicore embedded systems. Developers describe applications as parametrised SDF graphs as in Figure 3 and provide high level details about the target multicore architecture including a number of cores. Frameworks like PREESM check the SDF graph for consistency and schedulability and transform it into a DAG that exposes the application's parallelism. As a next step, the scheduling process executes after which the code generation process is launched. The default code generation in PREESM invokes one static thread per available core.

The work in [13] augments the generated code with OpenMP compiler directives in order to better fit data parallel applications to an array processor. Nevertheless, the granularity of the actor grouping and the processor details are considered at compile time. That makes the approach hard to scale with a variable number of processing cores. The work in [4] proposes a methodology to use task programming models for code generation from SDF graphs. The concept, shown in Figure 2, is to bypass the bottleneck of the static scheduler and shift scheduling in runtime using task libraries. In [14], the concept of *task wrapper* is introduced (Figure 4). In this case, tasks that can run to completion without intermediate synchronization are grouped together. Then runtime scheduling can be performed in wrapper level so that task synchronization overhead is reduced. Nevertheless, those works do not address the potential bottleneck caused by task stealing during runtime scheduling; this constrains the scalability and therefore the performance.

In our work we propose a scalability optimisation for the task code generation process that addresses the overheads of runtime scheduling by exploiting more efficiently the work stealing process.

III. Task Recursion

Traversing an SDF graph in order to handle actors as tasks can lead in situations where high runtime overheads appear with respect to task management i.e. spawn, steal and sync processes. Indeed, for graphs like in Figure 4, the spawning of tasks by following graph dependencies as in [4], places them into a single task dequeue, the one belonging to the host core i.e. the core that executes the main function. The rest of the available cores can perform successful stealing only from the host. This occurring contention for task stealing can become critical for performance in a growing multicore platform, especially in the case of graphs with fine grain actor parallelism. Workarounds for relaxing this overhead include

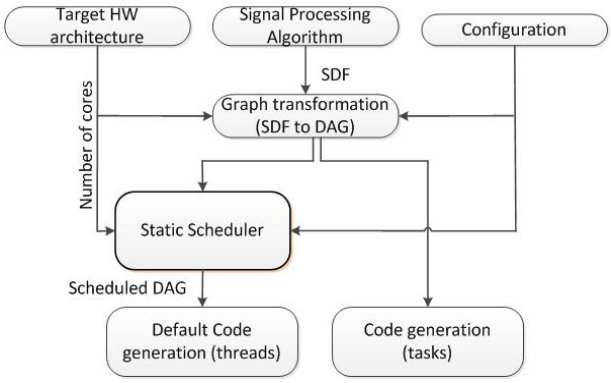

Fig. 2. PREESM code generation flow

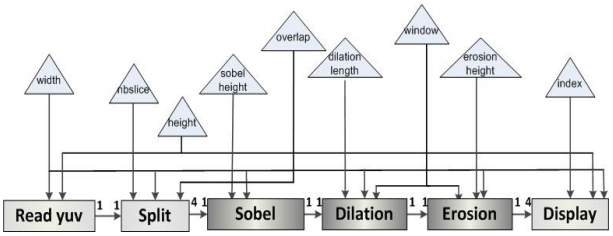

Fig. 3. Sobel morpho algorithm SDF (parameter nbSlice=4)

setting a limit for the number of stealable tasks in a dequeue [15], or splitting a task dequeue in private and public portions, where the host core operates on the private and the rest cores on the public portion [16]. However those options might reduce potential parallelism as non-dependent tasks are forced to be executed sequentially. As mentioned already, the typical way for task programming models to alleviate this problem is the handling of tasks in a divide-and-conquer fashion as part of the load balancing optimisation process. That makes work stealing able to operate on coarser task granularity. Also, stealable tasks can be distributed among task dequeues of all available cores. Actually, many task programming libraries feature macros that implement this kind of task spawning using recursion loops e.g. *cilk_for* in Cilk [17]. However this typically requires manual changes on the source code to ensure that no data dependencies exist among tasks.

Alternatively, our approach is to exploit the SDF description in order to generate the tasks needed for recursive execution. With respect to a DAG of a URC-SDF graph, we create a *root* task that includes all instances of an actor. Then we repeatedly divide the root task and group actor instances in smaller tasks in order to create all required tasks for recursion, down to the point that tasks containing a single actor instance are created. In this way, once a root task is spawned by a core, a task *binary tree* unfolds dynamically. With respect to Figure 5, the core executing the root task spawns 2 new *leaf* tasks one of which can be stolen and executed by another core. Eventually, the same happens for all binary tree nodes (leaf tasks) which are then distributed to available cores if they are idle. Therefore the generated code's scalability can be improved as the task distribution and load balancing can tolerate high granularity in the application parallelism.

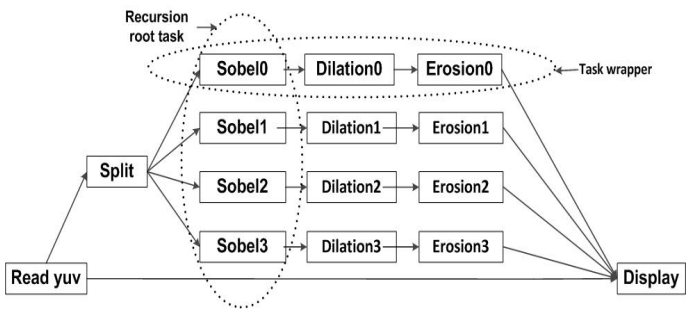

Fig. 4. Sobel morpho algorithm DAG

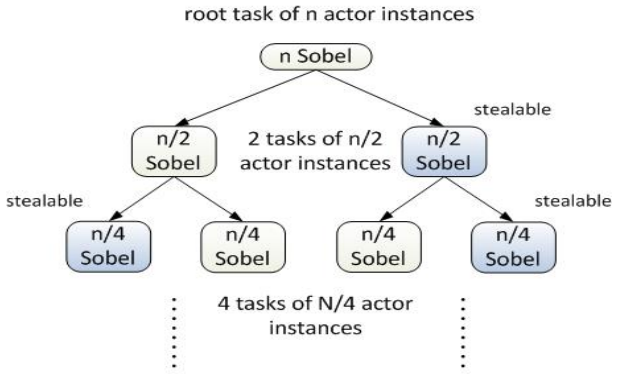

Fig. 5. Example of recursive task execution of Sobel actor instances in Sobel morpho DAG (with n=4, binary tree height=2, binary tree size=7)

A. Recursion in Task Code Generator

We implement recursive task execution and apply it in the context of the task-based code generator of PREESM [14] which utilizes Wool and its work stealing scheduler [10] as task library. Specifically, we add a *recursive_code_generation* procedure to the code generation process. This procedure defines tasks that group actor instances of the DAG in order to realize runtime execution according to the binary tree of Figure 5. For the DAG of Figure 4, this creates 3 recursive executions, one for each of the Sobel, Dilation and Erosion actors. For actor grouping, we use the task wrapper concept. Indeed, it is optimal to consider recursion applied to a task wrapper that groups 1 actor instance from each Sobel, Dilation and Erosion actor. Therefore, the binary tree of Figure 5 executes with a root task of n task wrappers.

The *recursive_code_generation* procedure is summarized in Algorithm 1: The start of the URC subgraph is traced by traversing the actor list of V_{dag} which is typically provided by the dataflow framework, in this case PREESM. Then a double loop is used to iterate on the binary tree height number and on the number of tasks wrappers to be defined per height. Currently our code generator supports URC subgraphs where actor repetition count (actor instantiations) is a power of 2, further extension to arbitrary count is scheduled for future work. In this case, the height of the task binary tree *bl* is given by:

$$bl = log_2(q(A)) \tag{1}$$

The number of task wrappers with the same binary tree height value *nw* is given by:

$$nw = 2^{li}, 0 \leqslant li \leqslant bl, li \in \mathbb{N}, \tag{2}$$

For each task wrapper, 2 indexes are calculated that reflect the actor instances assigned to it (line 10 of Algorithm 1).This is reflected with the VOID_TASK_1 definition in the code output. Actor instance numbering range is from 0 to $\mathbf{q}(A)$. Within each task wrapper, two children wrappers are spawned (line 12 and 13 of Algorithm 1) according to the binary tree of Figure 5. This is reflected in the code output with the SPAWN and respective SYNC keywords for each wrapper. Figure 6

```
INPUTS V_dag, q(A)
 rec_t = q(A)
 rec_i = bl = log2(q(A))
for actors ∈ V_dag do
    if first_URC_actor then
        for j ∈ [0, rec_i] do
            med = rec_t/2
            nw = 2^j
            for i ∈ [0, nw] do
                task(i * rec_t , ((i + 1) * rec_t) − 1)
                {
                 spawn_task_1(rec_t * i , rec_t * i +
                 med − 1)
                 spawn_task_2(rec_t * i + med, ((i +
                 1) * rec_t) − 1)
                 sync_tasks
                }
            end
             rec_t = med
        end
    end
end

```

Algorithm 1: Recursive task code generation implementation

```
// recursion wrappers definitions
//for n task wrappers
//sobel_wrap=sobel+dilation+erosion
  VOID_TASK_1(sobel_wrap_0n-1,int,level){
  SPAWN(sobel_wrap_0,(n/2-1),1);
  SPAWN(sobel_wrap_n/2,n-1,1);
  SYNC(sobel_wrap_n/2,n-1);
  SYNC(sobel_wrap_0,(n/2-1));
 }
 VOID_TASK_1(sobel_wrap_0,(n/2-1),int,level){..}
 .....
 VOID_TASK_1(sobel_wrap_n/2,n-1,int,level){..}
 .....
  //main function
  VOID_TASK_2(computation...{
   for(i=0;i<frames;++){
   readYUV();
    split();
    // recursion wrapper
   SPAWN(sobel_wrap_0n-1);
   SYNC(sobel_wrap_0n-1);
   merge();
   yuvDisplay();
   }
  }
```

Fig. 6. Pseudo code output for Sobel Mopho algorithm

illustrates the code generated for task wrappers as well as the main function for the algorithm of Figure 4, where the root task wrapper is called. SPAWN statements push respective tasks to the core's dequeue. Each SYNC statement matches the immediately previous active SPAWN statement. SYNCs are executed sequentially. This makes the task of the second SPAWN immediately executed in its host core, while the task of the first SPAWN is stealable by other cores.

IV. Experimental Results-Discussion

The generated code featuring task recursion is tested on an Intel Skylake-based platform (32 vCPUs, 120 GB memory, Ubuntu 16.04 LTS OS). Our test application is Sobel Morpho (Figure 3), a popular image filtering algorithm that represents well URC SDF graphs [18]. We use a HD resolution (1920x1088) 10-frame video sequence as input data and average frames-per-second (fps) value as our performance metric.

We compare the performances of the generated code with recursion against the generated code with simple task execution, on the same platform. Figure 7 shows the results of the generated code execution based on recursive tasks against execution based on simple tasks for Intel Skylake processor. The horizontal axis is the number of cores while the vertical axis is the fps value. A 'slice' is a partition of the input data (HD video frame) of the Sobel Morpho algorithm.

As can be seen from the results, both versions of the code generator output (with and without recursion) perform similarly for low slice count. In fact the proposed recursive execution is slightly less efficient in terms of throughput than the non-recursive one. This is attributed to the overhead from the higher number of tasks that need to be generated for recursion. In our implementation, recursive execution of N tasks corresponds to executing a number of tasks equal to the nodes of a binary tree (as in Figure 5) with N leaf nodes. Therefore the total tasks to be created, spawned and executed are given by

$$Nrecursive = 2 * N - 1, \tag{3}$$

However, for higher slice count the recursive execution scales better and maintains significant throughput increase as opposed to the simple task execution. Although the number of recursive tasks is still higher, the respective overhead is hidden from the efficient task stealing and load balancing of the recursively executed code. For the task library in use (Wool), the maximum number of stealable tasks in a queue is configurable. In fact this value is set by default to 3 + $\log_2 N$ where N is the number of worker threads (cores). According to [19], this value is selected as a trade-off between allowing enough parallelism and minimizing task stealing overhead. While this value fits well to recursive task execution which requires a low number of stealable tasks, it limits the scalability in the case of simple task execution where all tasks are placed in one task queue in one core and a high number of tasks have to be stolen.

This is evident in Figure 8, which shows the distribution of leaf node tasks to cores from Sobel Morpho execution (8 and 32 slices, 80 and 320 total tasks respectively) in Intel Skylake for both cases. It can be seen that for 8 slices, the simple task execution has similar task distribution to the recursive one as enough tasks are stealable. However for 32 slices, the amount of non-stealable tasks is too high thus they are spawned and executed by the same core. Consequently, in case of simple task execution, scalability cannot be maintained without considering the level of application parallelism and the

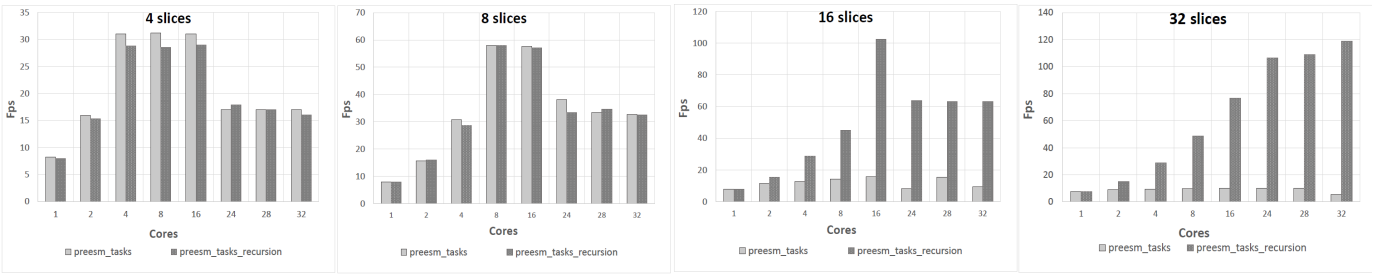

Fig. 7. Recursive against simple task execution for Sobel Morpho application on Intel Skylake

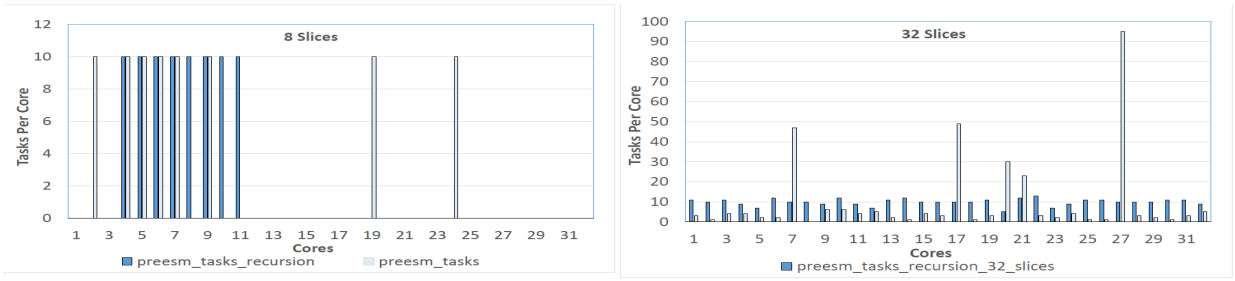

Fig. 8. Distribution of tasks to cores for Sobel Morpho application

number of available cores. On the contrary, adding recursion to our code generator produces output that achieves good performance by maintaining scalability independently from the granularity of the application parallelism and the number of available cores.

V. Conclusion

In this paper we propose task recursion as an optimisation of the usage of task models for executing SDF graphs in multicore processors. Our approach is based on extracting synchronous dataflow graph information in order to reduce scheduling overheads. We use PREESM framework to enhance the task-based code generation process with recursive task execution. We evaluate the performance of the code featuring recursive tasks in a 32-core Intel Skylake processor. Results show that our concept enhances scalability thus improves performance in terms of application throughput. Plans for future work include extension of the recursive code generator to support SDF graphs with arbitrary values of actor repetition counts. Furthermore, we plan to explore the adaptation of task recursion for SDF graph execution in heterogeneous platforms.

References

[1] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," *Parallel computing*, vol. 25, no. 13-14, 1999.

[2] J. Heulot, M. Pelcat, K. Desnos, J.-F. Nezan, and S. Aridhi, "Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*. IEEE, 2014, pp. 167–171.

[3] C.-C. Shen, L.-H. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S. S. Bhattacharyya, "The dspcad lightweight dataflow environment: Introduction to lide version 0.1," Tech. Rep., 2011.

[4] G. Georgakarakos, S. Kanur, J. Lilius, and K. Desnos, "Task-based execution of synchronous dataflow graphs for scalable multicore computing," in *Signal Processing Systems (SiPS), 2017 IEEE International Workshop on*. IEEE, 2017, pp. 1–6.

[5] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*. IEEE, 2014.

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

[7] E. A. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, Jan 1987.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

[9] T. Willhalm and N. Popovici, "Putting intel® threading building blocks to work," in *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, 2008, pp. 3–4.

[10] K.-F. Faxen, "Efficient work stealing for fine grained parallelism," in *Parallel Processing (ICPP), 2010 39th International Conference on*. IEEE, 2010, pp. 313–322.

[11] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5.

[12] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, 2011.

[13] J. Hascoët, K. Desnos, J.-F. Nezan, and B. D. de Dinechin, "Hierarchical dataflow model for efficient programming of clustered manycore processors," in *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference on*. IEEE, 2017.

[14] G. Georgakarakos and J. Lilius, "Efficient task-based code generation for sdf graph execution on multicore processors," in *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2018, pp. 112–117.

[15] T. van Dijk and J. C. van de Pol, "Lace: non-blocking split deque for work-stealing," in *European Conference on Parallel Processing*. Springer, 2014, pp. 206–217.

[16] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*.

[17] C. E. Leiserson, "The cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[18] N. Sengar and D. Kapoor, "Edge detection by combination of morphological operators with different edge detection operators," *International Journal of Information & Computation Technology*, vol. 4, no. 11.

[19] K.-F. Faxén, "Wool 0.1 users guide," 2009.