Hierarchical Dataflow Model for Efficient Programming of Clustered Manycore Processors

Julien Hascoët, Karol Desnos, Jean-François Nezan, Benoît Dupont de Dinechin

Hierarchical Dataflow Model for Efficient Programming of Clustered Manycore Processors

Julien Hascoët[1,2], Karol Desnos[2], Jean-François Nezan[2], Benoît Dupont de Dinechin[1]
[1] Kalray, Montbonnot-Saint-Martin, France
[2] IETR, INSA Rennes, CNRS UMR 6164, UBL, Rennes, France
email: {jhascoet, benoit.dinechin}@kalray.eu, {kdesnos, jnezan}@insa-rennes.fr

***Abstract*—Programming Multiprocessor Systems-on-Chips (MPSoCs) with hundreds of heterogeneous Processing Elements (PEs), complex memory architectures, and Networks-on-Chips (NoCs) remains a challenge for embedded system designers. Dataflow Models of Computation (MoCs) are increasingly used for developing parallel applications as their high-level of abstraction eases the automation of mapping, task scheduling and memory allocation onto MPSoCs. This paper introduces a technique for deploying hierarchical dataflow graphs efficiently onto MPSoC. The proposed technique exploits different granularity of dataflow parallelism to generate both NoC-based communications and nested OpenMP loops. Deployment of an image processing application on a many-core MPSoC results in speedups of up to 58.7 compared to the sequential execution.**

I. Introduction

After decades of exponential growth, the processing capability of individual PEs have leveled-off, due to complexity and power consumption considerations. In order to cope with the rising complexity of modern embedded applications, such as wireless communications or computer vision algorithms, MPSoCs have now increased their processing capabilities by integrating more and more PEs, a wide variety of memory architectures, and complex on-chip interconnects. For example, KeyStone is a shared memory architecture, whereas Epiphany adopts a distributed memory architecture. Clustered architectures [1] are also a promising step towards high performance on-chip computing. A clustered architecture consists of a set of clusters communicating through a NoC, where a cluster designates a set of PEs sharing a local memory.

High-level Application Programming Interfaces (APIs) OpenMP and OpenCL are widely used to program multi-core architectures. Compared to manual thread programming, these APIs ease the specification of parallel computation, notably by relieving the developer from explicitly specifying thread synchronizations. Despite their high-level abstraction of parallelism, portability of OpenCL and OpenMP code is not optimal. Indeed, the implementation efficiency of a given OpenMP or OpenCL code will be different depending on the targeted architecture and runtime; and the developer will often have to adapt its code depending on the targeted architecture.

Dataflow MoCs are widely used for the specification of data-driven algorithms in many application areas. A dataflow graph is composed of communication edges representing First-In First-Out queues (FIFOs), that connect vertices (actors) responsible for performing computations. Dataflow MoCs are architecture agnostic, which makes them highly valuable for the specification of applications that can be deployed on a wide variety of embedded systems. The process responsible for generating efficient multi-core code from a portable dataflow description, for a specified target architecture, is called software synthesis [2].

The Synchronous Dataflow (SDF) MoC introduced in [3] is a specialization of the dataflow MoC that specifies for each FIFO the fixed number of data tokens produced and consumed at each execution (firing) of connected actors. SDF is probably the most studied dataflow MoC. Its popularity is largely due to its **analyzability**, its predictability, and its natural description of concurrency, which make it suitable for efficient execution on MPSoCs. The analyzability of the SDF MoC is a particularly strong asset when synthesizing software from a dataflow description [2].

In this paper, we show how the properties of the Interface-Based Synchronous Dataflow (IBSDF) MoC, a hierarchical extension of the SDF MoC, can be exploited to synthesize efficient software for modern MPSoCs. The hierarchy feature of the IBSDF MoC is used for the **mapping** of computation on PEs, for **code generation**, and for efficient management of **off- and on-chip communications**. A new code generation scheme for IBSDF graphs, which automatically constructs loop nests and inserts OpenMP directives, is detailed in this paper. The portability of our approach, which makes it possible to target both shared memory architectures (Intel x86 and TI Keystone) and clustered architectures (Kalray MPPA®), is illustrated with a state-of-the-art computer vision application. On the clustered architecture, inter-cluster parallelism relies on automatically generated one-sided asynchronous NoC communications, while intra-cluster parallelism is based on OpenMP3 multi-threading directives.

The paper is organized as follows: Section II presents the IBSDF MoC targeted in this work and related works from the literature. Our contribution is then described in Section III. Section IV presents a thorough evaluation of our technique for the deployment of a computer vision application onto various architectures. Section V concludes this paper.

II. Context and Related Work

A. Programming MPSoCs

Programming multi-/manycore architectures efficiently is a huge challenge. Although many programming APIs adopting

978-1-5090-4825-0/17/$31.00

various MoCs can be found in literature, no universal parallel programming model fitting all architectures and all applications exist. Pthread and OpenMP3 are multi-thread programming models for shared memory architectures where all the PEs access a common memory address space. The memory consistency between PEs is ensured at synchronization points in multi-threaded programs.

OpenMP4, OpenCL and CUDA are acceleration programming models. The purpose of acceleration programming models is to offload an application's heavy computations onto external computing resources, such as external CPUs, GPUs, FPGAs or hardware specific accelerators. The memory coherence is maintain by the execution runtime of the model.

The aforementioned programming models often propose dedicated features, like dedicated preprocessing directives or specific instructions via intrinsics, to target specific architectures. In order to be usable, these programming models and the target-specific features must be supported by the tool chain of the hardware provider. The developer uses the available features depending on the application, the targeted architecture and identified bottlenecks. The programming thus requires a deep understanding of the application, the hardware and (available) runtime libraries. It often takes months to port an application on a new architecture.

On the other hand, tools such as ORCC, PREESM, SCADE / Lustre, or SigmaC are alternatives providing developers with a higher level of abstraction based on dataflow MoC. SigmaC is a language based on the Cyclo-Static Dataflow (CSDF) model which is an extension of the SDF model. SigmaC [4] was supported in the Kalray MPPA® toolchain and was well-suited for time-critical applications with a static behavior, i.e. computations are the same for all data and cannot change dynamically.

ORCC is a compilation framework based on a language called RVC-CAL. This language is based on the Dataflow Process Network (DPN) MoC which is a dynamic and non-deterministic dataflow model [5]. Unfortunately this language does not provide static analyzability for programming time-critical embedded systems.

SCADE/Lustre is a language for reactive system programming as it provides a logical time notion. The main differences with the other approaches is that SCADE allows sampling (sensors typically) in the firing of nodes. PREESM proposes two SDF extensions IBSDF and Parameterized Interface Synchronous Data-Flow (PiSDF), providing features that ease the application description while retaining the predictability of the SDF model. The work presented in this paper is based on the IBSDF dataflow model. This predictability is used in the proposed automatic mapping of the application onto the targeted architecture.

B. IBSDF Dataflow Model Background

The Interface-Based Synchronous Dataflow (IBSDF) [6] MoC is a hierarchical extension of the SDF model. In addition to the SDF semantics, IBSDF adds the possibility to specify the internal behavior of an actor with a dataflow subgraph instead of specifying it with code.

The repetition vector RV of an SDF graph G is a vector containing an integer value $RV(a)$ for each actor a of G. An SDF graph completes a graph iteration when each actor is executed as many times as specified by the RV, thus bringing back the graph to its initial state in terms of number of data tokens stored in each FIFO. RV is computed at compile time using static data rates of actors [3]. The IBSDF compositional feature enables independent computation of the RV of each hierarchical (sub)graph [6].

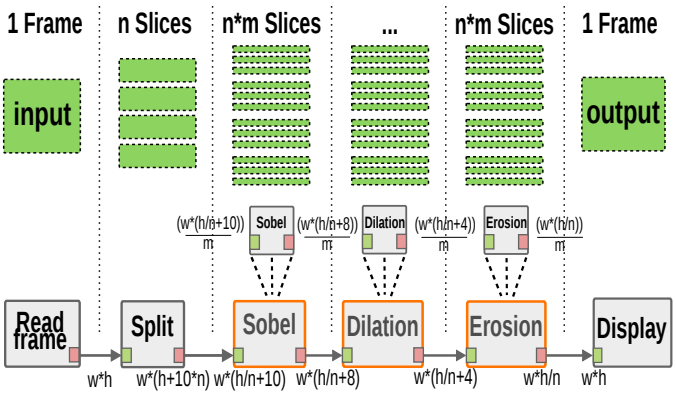


Fig. 1: IBSDF Graph of the Application Use-Case

The IBSDF graph of an image processing use-case is presented in figure 1. The IBSDF graph is composed of six actors at the top level of the hierarchy. Three actors of the top-level graph (with orange borders) are hierarchical actors. Each of the three hierarchical subgraphs includes a single actor whose production and consumption rates, and thus its number of executions, are controlled by a parameter m. Indivisible data tokens exchanged in this graph are pixel lines of width w. The $ReadFrame$ actor produces an image that is then divided into overlapping slices by the $Split$ actor. Then, the $Sobel$, $Dilation$, and $Erosion$ actors perform standard image processing filtering and morphological operations.

The topology of this application graph illustrates the opportunity to map successive hierarchical actors on a single compute cluster, thus reducing data movements and increasing the arithmetic intensity which is the amount of processing done for each byte of data transferred to a compute cluster. Maximizing the arithmetic intensity is essential for achieving decent performance on applications running on a clustered architecture or any other multi-core *CPU*s.

Another hierarchical generalization of the SDF MoC called Deterministic SDF with Shared FIFOs (DSSF) is proposed in [7]. The main difference between the DSSF and the IBSDF MoCs is that DSSF compositionality results from a graph analysis, whereas IBSDF graphs are inherently compositional. In DSSF, a bottom-up analysis is used to expose compositionality of the of hierarchical graph, when possible. Based on this analysis, hierarchical actor can be translated into equivalent modular code with variable consumption and production rates.

In the IBSDF MoC, the compositionality is enforced by the model semantics and execution rules, which make it possible to translate each hierarchical actor into an equivalent code with fixed production and consumption rates.

C. IBSDF-based Design Flow and Limitations

The development flow presented in figure 2 shows typical design and compilation steps from the IBSDF graph specification by the developer, using a graphical user interface, to the software synthesis.

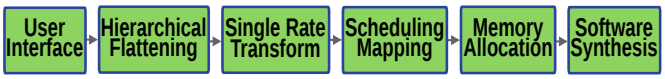


Fig. 2: Rapid Prototyping Flow

In the last version of PREESM, hierarchical IBSDF actors are systematically flattened: hierarchical actors are replaced by all actors and FIFOs contained in their subgraphs. The flattening decreases the graph granularity as actors of lower levels of hierarchy are all gathered in a single graph.

The *Single-Rate* transformation is applied to the flattened graph in order to reveal all application parallelism. The single-rate transformation consists of converting the SDF graph into an equivalent single-rate SDF graph where each actor is duplicated as many times as specified by the RV. The resulting *Single-Rate* SDF graph is used for subsequent mapping, scheduling, memory allocation, and code generation operations.

Flattening all the hierarchy is problematic when targeting large dataflow graphs and architectures with hundreds of PEs. First because the mapping, the scheduling, and the memory allocation are NP-hard problems. Second because fine-grained synchronizations can strongly degrade system performances. Let's consider the use-case in the figure 1 with the flattening operation: the number of automatically generated actors after the *Single-Rate* transformation becomes $3 * n * m$.

III. CONTRIBUTION

A. Hierarchical Approach

The hierarchical approach consists of exploiting the graph hierarchy in the different steps of the development flow presented in figure 2 instead of systematically flattening it. In our method, we propose specifying whether or not the hierarchical actors shall be flattened. The method exploits several granularities of parallelism captured by nested, non-flattened, hierarchical graphs. We define a *clustered actor* as a non-flattened hierarchical actor. Clustered actors are large in terms of memory footprint and computation. They can be mapped to a PE as a single actor, and are thus simpler to map than the equivalent set of actors resulting from a flattening of the graph. We use the heterogeneous memory static allocator described in [8] to allocate buffers in the system, not only for the distributed memory architecture, but also for shared memory architectures. The memory allocation is simpler and more efficient on clustered actors.

In the targeted architectures, two levels of parallelism are exploited: both coarse-grained and fine-grained parallelism. Coarse-grained parallelism is found at the top-level of the hierarchy, where the graph contains clustered actors. Fine-grained parallelism is retrieved in subgraphs of non-flattened hierarchical actors. In our case, the fine-grain parallelism is extracted from RVs during the software synthesis of hierarchical actors.

This approach is not only adapted for clustered many-core processors but also for off-the-shelf processors and multi-core Digital Signal Processing (DSP). Indeed, the software synthesis for hierarchical actors produces *For-Loops* that are exploited by compiler optimizations to extract the instruction level parallelism. Therefore, our proposal takes advantage of both low-level (Section III-C) parallelization at core level and high-level (Section III-B) parallelism for the cluster (but also for the core) using the hierarchical mapping strategy.

Finally, our approach provides the possibility of choosing the best hierarchy level. The PREESM user decides in his own interest and even depending on the architecture what must remain hierarchical and what should be flattened. The scheduling is executed after the single-rate graph transformation is applied. These tuning capabilities are performed by the PREESM user for rapid-prototyping on MPSoC.

B. High-Level Hierarchy (Inter-Cluster)

The high-level mapping of hierarchical actors is more efficient as it is adapted for coarse-grained granularity. The high-level hierarchy is applied to inter-cluster parallelism. Coarse-grained mapping is provided by the hierarchy feature and provides several advantages.

First, the hierarchical actor software synthesis makes it possible to generate automatically memory access coalescing for actors of their subgraph, whereas flattening the hierarchy generates many smaller data transfers (one for each firing of the actor in the flattened subgraph). Memory coalescing is a key to increasing performance on parallel code running on a manycore processor as it reduces the number of Remote Direct Memory Access (RDMA) transactions.

Secondly, the hierarchical actor software synthesis reduces drastically the mapping complexity when the number of cores and actors increases. The compute clusters of the MPPA® processor are seen as a single multi-core CPU for coarse-grained application mapping. On a clustered architecture, hierarchical actors will be mapped on a compute cluster, whereas actors of subgraphs will be mapped at core level.

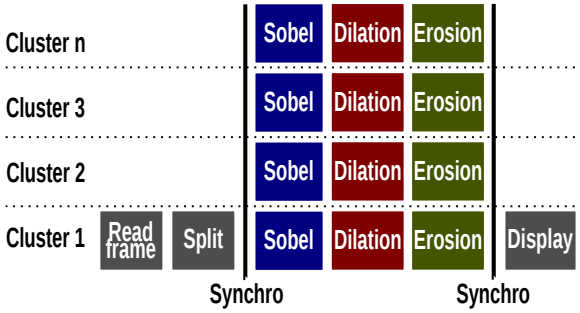


Fig. 3: Gantt Chart of the Hierarchic Scheduling

Figure 3 shows the inter-cluster parallelism at the top-level of hierarchy. The compute clusters run concurrently in parallel with the mapped hierarchical actors (*Sobel*, *Dilation* and *Erosion* hierarchical actors) and performs inter-cluster synchronizations. The *Split* (zero copy) actor has a fork role and unlocks all other compute clusters (from 1 to n) when the data are ready to be computed. Once the parallel region of the compute cluster is ended all contributors merge all results to the *Display* actor (zero copy).

C. Low-Level Hierarchy (Intra-Cluster)

The new support of hierarchical actors mapping and code generation allows both code factorization and more computation efficiency on fine-grained code regions. The low-level hierarchy exploits intra-cluster parallelism.

Fine-grained parallelism implies several concurrent computations where the synchronization and memory consistency need to be managed efficiently. The generated code sections of hierarchical actors are automatically parallelized using OpenMP (if available). The intra-cluster parallelism is automatically extracted from RVs in sub-graphs that contains actors that have a potential source of parallelism. For instance, if a hierarchical actor A consumes $N * M$ tokens and actor B in hierarchical actor A consumes N, an RV of M (see l.8, 12, 16 in Fig. 4) will automatically be extracted and generated by the parallel code generator. This new feature allows for the automatic extraction of the RVs in sub-graphs that contains actors that have a potential source of parallelism. Repetition vectors are generated in C language using a static finite *For-Loop*. Naively, a *For-Loop* is sequential, but our new software synthesis adds an *"omp parallel for"* to get parallelism for any architecture supporting OpenMP3 (multi-threading model). The parallel section is automatically generated contrary to [5] where it is manually inserted. In this case, OpenMP3 is very efficient as the number of *For-Loop* iterations is fixed and thus it is solved at compile time. Thus thread-level parallelism is reached in the clustered actors that are mapped onto the compute clusters (or cores). Synchronization points are done via fork and join regions of the OpenMP runtime (see l.7, 11, 15 in Fig. 4). As such, the memory consistency points and the synchronization point placements are known at compile time. Therefore, it provides predictable execution time, not only for sequential execution, but also for parallel execution when using OpenMP3 on finite *For-Loops*.

D. Automatic Explicit Communications

In order to exploit the performance of complex manycore processors, the application needs to be broken into pieces (ie: actors communicating with FIFO) and it implies effort. On such architecture, the efficiency of communications is crucial. One contribution in this paper is the use of automatically generated **RDMA explicit memory accesses** onto the targeted massively parallel processor, which outperforms the shared memory approach provided by data caches through *Load/Store*. Thenceforth, the introduced code generation is similar

```
1  /* Inter-Cluster Synchronization */
2  synchro(...);
3  /* Global to Local Memory Coalescing */
4  get(..., tag); /* reads buffer */
5  wait(tag); /* wait end of transfer */
6  /* Parallel Hierarchical Sobel */
7  #pragma omp parallel for /* intra-cluster */
8  for(int i=0;i<M;i++)
9    sobel(...); /* Sobel Kernel */
10 /* Parallel Hierarchical Dilation */
11 #pragma omp parallel for /* intra-cluster */
12 for(int i=0;i<M;i++)
13   dilation(...); /* Dilation Kernel */
14 /* Parallel Hierarchical Erosion */
15 #pragma omp parallel for /* intra-cluster */
16 for(int i=0;i<M;i++)
17   erosion(...); /* Erosion Kernel */
18 /* Local to Global Memory Coalescing */
19 put(..., tag); /* send buffer */
20 wait(tag); /* wait end of transfer */
21 /* Inter-Cluster Synchronization */
22 synchro(...);
```

Fig. 4: Parallel Codegen for Compute Cluster of the MPPA®

to [9] based on OpenMP mixed with MPI-3 and using one-sided communications. The main difference is that our parallel code is generated automatically from an IBSDF dataflow graph and not handwritten like in [9]. The basic concept of one-sided communications is that each compute cluster is a master of one or several remote memories, which are, in our case, the global DDR memories and other compute cluster local memories.

Regarding inter-cluster parallelism, the synchronizations use explicit transfers directly across compute clusters (l.2, 22 in Fig. 4). In our current software synthesis, inter-cluster data transfers go through the main global memory (DDRs).

The compute clusters perform explicit memory data transfers that are based on highly efficient RDMA *put-get* memory accesses because of the local memories. Figure 4 shows where RDMA accesses are done with *put* and *get* primitives (l.19, 4 in Fig. 4 respectively). The RDMA transaction completion is ensured by the *wait* primitive (l.5, 20 in Fig. 4).

Such code generation has several advantages for an architecture like the MPPA® but also for other DSPs or general purpose processors. On the MPPA® we automatically catch coalesced memory accesses at code generation as shown in figure 4. Memory coalescing means that multiple data transfers are merged in one. It allows both the reduction of global memory data requests (requests traffic) and optimizes the usage of the scratch-pad memory (local memory). When chaining kernels locally without any communications other than intra-cluster communications and synchronizations (shared memory), the execution is very efficient. The automatic optimization provided with code generation allows for the limitation of data movement that are both very time and power consuming. The code generation in figure 4 illustrates what is done on dataflow applications where both spatial and temporal data locality are exploited.

Multi-core CPUs	TI C6678 EVM 1 GHz		Core i7-3820 3.6 GHz	
Nb Cores	FPS	Speedup	FPS	Speedup
1	8.9	1.0	49.3	1.0
2	17.6	1.9	91.6	1.8
4	33.8	3.8	155.6	3.1
8	64.4	7.2	211.5	4.2

TABLE I: Parallel For-Loop onto TI DSP and Intel Processor

IV. Experimental Evaluation

The targeted use case is an image filtering application consisting of basic image processing application building blocks, namely the sobel, erosion, and dilation kernels. Benchmarks have been run with a VGA resolution ($640 * 480$) for all targeted architectures. The main purpose of this experimental evaluation is to show that the proposed hierarchical code generation has benefits for both mapping/scheduling as well as for the memory allocation. All benchmarks have been compiled using the *GCC GNU Compiler* using *O3* optimization. No assembly nor intrinsic optimization are used.

Kalray MPPA® Bostan Second Generation: Regarding the benchmark environment, the MPPA2-256 is plugged into the motherboard of an Intel host processor where MPPA®'s IOs perform PCI-Express communications in real-time. Two modes of execution are used. The first one uses the software emulated L2 cache where global memory accesses are done by *Load-Store*. The second uses RDMA to perform explicit memory accesses, as illustrated in the code in figure 4. We focus our analysis on explicit memory accesses over RDMA as the software emulated L2 cache provides lower performances because of irregular memory access patterns. Kalray's OpenMP implementation is based on *GCC libgomp* which is ported over Kalray's proprietary OS providing pthread primitives enabling OpenMP3. When the L2 cache is not used, the buffer allocation done by [8] should never exceed the 2 Mega bytes of local memory for each cluster.

Texas Instruments C66X: TI C66X runs 8 DSP cores at 1 GHz. This MPSoC has a hardware L2 data cache enabling accesses to the global memory. I/Os are managed before and after running the application. Paper [10] presents the efficient bare metal implementation of OpenMP3 for the TI C66x.

A. Results and Discussions

This section presents the results for several multi-core architectures, but the main focus will be given to the Kalray's manycore processor. Table I presents the measured performances using the hierarchical actor software synthesis presented in III-C. Compared to the single-core execution, a fair speedup is achieved on the TI C66X, with a maximum speedup of 7.2 on 8 cores. The Intel Sandybridge off-the-shelf processor also presents fair speedup, up to 4.2, which is fair for an architecture with 4 hyper-threaded cores.

Table II shows mono-cluster (CPU of 16 VLIW cores) results using explicit communications and the distributed shared memory which emulates a software L2 data cache for off-chip memory accesses. As described in figure 4, the software synthesis that uses explicit memory accesses using RDMA outperforms the naive shared memory approach over data cache by **22%**. This table shows speedups of 13.4 when using explicit communications and 11.2 when data accesses are performed by L2 data cache, thus the scalability is honorable in both cases.

The application mapping is performed at cluster level. Clusters are considered as multi-core CPUs in order to map clustered actors and we exploit subgraph parallelism inside clusters when possible to obtain thread-level parallelism.

Figure 5b plots the application performance in frames per second (FPS), observed when using a variable number of cores per clusters, and variable number of clusters on the MPPA®. Using one core in each of the 16 clusters provides lower performance than using 16 cores of a single cluster because of the intensive usage of the local on-chip memory and NoC communications are reduced compared to the multi-cluster approach. However, in our case, this runtime overhead remains low as the parallelism is known statically. It can be noticed that performances in 5b for one cluster are lower than the ones shown in the mono-cluster configuration of table II. This is due to the used Operating System (OS) running on the compute cluster which is a bare-like-system providing better results (however not usable for OpenMP3).

A total speedup of **58.7** is reached when using all 16 cores of all 16 compute clusters. Although we have a good scalability but we hit the memory bandwidth wall (Section IV-B2) of many-core processors when the 256 cores are competing for the global main memory. Thus we focus on local memory usage at code generation to save main memory bandwidth.

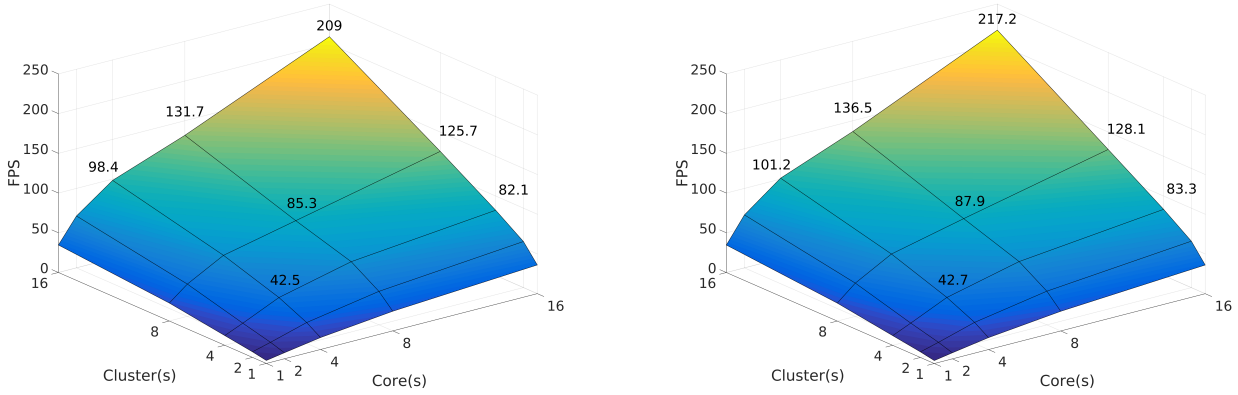
B. Comparisons with Flat IBSDF Mapping

1) Performances Analysis: For shared memory architectures Intel and TI C6678 EVM, the flat IBSDF gives same performances as in table I. Figure 5a performances are lower than 5b by **4%** when using all processing elements of the manycore. This difference is mainly due to RDMA memory accesses coalescing, which are provided by the hierarchical mapping approach. The flat state-of-the-art IBSDF mapping makes each core perform RDMA transactions, which increase the ratio communication vs compute by **7.8%** compared to our new hierarchical approach.

2) Memory-Wall of Manycore: In this application, NoC communications are less than 8% of the whole processing time

Mono-cluster MPPA®	MPPA® 400 MHz L2 Cache		MPPA® 400 MHz RDMA	
Nb Cores	FPS	Speedup	FPS	Speedup
1	3.6	1.0	3.7	1.0
2	6.9	1.9	7.4	2.0
4	13.3	3.7	14.5	3.9
8	24.4	6.8	27.4	7.4
16	40.5	11.2	49.4	13.4

TABLE II: Parallel For-Loop onto an MPPA® Cluster



(a) Flat Mapping

(b) Hierarchical Mapping

Fig. 5: MPPA® Matrix Result in FPS

when using $1/4$ of the processor's capabilities (for instance 8 clusters with 8 cores in each cluster). We hit the memory wall when using more than half of the chip's capability. The ratio between computation and NoC communications is more than 30% when all PEs are computing. Indeed, there are a lot of processing elements competing for global memory accesses.

3) Mapping: The mapping problem is well-known to be NP-complete. Its complexity increases exponentially with the number of cores and actors. On a manycore with 256 processing elements it becomes very complex for both theoretical mapping algorithms and their implementation efficiency in real life. In our case use, once the application parallelism is revealed by applying the flattening and single-rate transformation to all hierarchical actors, the resulting graph contains more than 1,000 actors and 800 edges to be mapped on 256 cores. Thus, the flat IBSDF graph takes **26 minutes** to be scheduled and mapped on the processing elements. In the hierarchical mapping approach, it takes less than **one second**. On more complex applications, for instance use cases with more than 10,000 actors, the hierarchical approach is a must-have feature as the mapping time explodes.

V. Conclusion & Future Works

This paper proposes a new technique to exploit both coarse-grained and fine-grained parallelism based on a hierarchical dataflow MoC. The main advantage of this special actor is that it provides the transformation workflow with scheduling and code generation simplifications. It also helps retrieving data locality, which is crucial for high performances and power consumption. Indeed data movement costs a lot for embedded MPSoC. The fine-grained parallelism is retrieved by applying *omp parallel for* onto RVs automatically extracted in a hierarchical actor. We show that this approach not only matches manycore processors with a distributed memory architecture but also multi-core architectures with shared memory. In the future, System-on-Chip (SoC) will embeds more and more heterogeneous cores, therefore, the mapping of such architectures will become more and more complex. In our use-case we targeted a low-level image processing application which shows significant speedup when the number of cores increases. The mapping of an application is not a simple problem and it is becoming more and more complex with increases to architectural complexity (number of cores, core heterogeneity within the same SoC, memory hierarchy, hardware accelerators). Future work will consider software synthesis for more complex hierarchical actors.

References

[1] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *DATE*, 2012, pp. 983–987.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software synthesis from dataflow graphs*. Springer Science & Business Media, 2012, vol. 360.

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

[4] B. Dupont de Dinechin, R. Ayrignac, P.E. Beaucamps, P. Couvert, B. Ganne, P. Guironnet de Massas, F. Jacquet, S. Jones, N. Morey Chaisemartin, F. Riss, T. Strudel, "A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, 2013.

[5] M. Chavarras, F. Pescador, M. J. Garrido, E. Juarez, and C. Sanz, "A multicore DSP HEVC decoder using an actorbased dataflow model and OpenMP," in *IEEE Transactions on Consumer Electronics*, vol. 61, no. 2, pp. 236–244.

[6] J. Piat, S. Bhattacharyya, and M. Raulet, "Interface-based hierarchy for synchronous data-flow graphs," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*. IEEE, 2009, pp. 145–150.

[7] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, p. 83, 2013.

[8] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi, "Distributed memory allocation technique for synchronous dataflow graphs," in *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*. IEEE, 2016, pp. 45–50.

[9] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. W. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Leveraging mpis one-sided communication interface for shared-memory programming," in *European MPI Users' Group Meeting*. Springer, 2012, pp. 132–141.

[10] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault, "Openmp on the low-power ti keystone ii arm/dsp system-on-chip," in *International Workshop on OpenMP*. Springer, 2013, pp. 114–127.