Bridging a Data-Flow Execution Model to a Lightweight Programming Model

Roberto Giorgi and Marco Procaccini
Department of Information Engineering and Mathematics
University of Siena
Siena, Italy
{giorgi,procaccini}@diism.unisi.it

***Abstract*—Starting from a Data-Flow execution model called "DF-Threads", we defined a minimalistic API to enable an efficient implementation in the hardware of the distribution of the threads across the cores of a single multi-core system and across the remote cores of a cluster. We aim at proposing this API as a simple programming model in C language that can potentially permit an easy interface between DF-Threads and generic programming models. Clusters are typically programmed with MPI, therefore we evaluated our approach against OpenMPI. If we consider the delivered GFLOPS per core, DF-Threads are also competitive in respect to CUDA. In the basic examples, that we used in this initial investigation, DF-Threads achieve better performance-per-core compared to OpenMPI and CUDA. In particular, OpenMPI has a large portion of OS-kernel activity, which is slowing down its performance.**

***Index Terms*—Performance evaluation, Computer architecture, Computer simulation, Matrix Multiply, Distributed computing, High performance computing, Data-Flow computing**

I. Introduction

The original motivation for research in Data-Flow computing was the possibility of exploiting of its massive parallelism [1], [2]. The performance scaling of the processors has basically followed the path of designing deeper pipelines, increasing clock rates and number of cores in a chip. However, when the single chip performance is not any more sufficient, a distributed architecture becomes an interesting solution. In such case, the full parallelism has not yet been completely exploited due to both execution model and programming model limitations [3], thus creating need for more research.

The Data-Flow execution model is capable of taking advantage of the full parallelism offered by a multi-core and multi-node systems [4]–[13] by introducing a new paradigm, which internally represents applications as a direct graph named program Data-Flow graph. Applications are represented as a set of nodes, where each node may represent an instruction or anything else (according to Data-Flow principles [2]). Directed arcs between nodes represent the data dependencies between the nodes. Whenever all inputs are available, the node is ready for firing. This stands in contrast to the Von Neumann execution model, in which an instruction is only executed when the program counter reaches it, without considering if it can be executed earlier or not. The key advantage is that, in Data-Flow, more than one instruction can be executed at once (in fact superscalar processors exploit internally this Data-Flow principle through the dynamic scheduling of instructions). Thus, if several nodes become ready at the same time, they can potentially be executed in parallel. This simple principle provides an opportunity for massive parallel execution. Past attempts to design an entire machine based on that principle where not successful (e.g., Manchester Data-flow Machine, Explicit Token Store architecture [14], [15]) mainly due to the too fine grain of the approach, i.e., at instruction level. In the context of the AXIOM project [16]–[22], we explored the feasibility of a novel Data-Flow execution model (Data-Flow Threads or DF-Threads [23]) in a heterogeneous and distributed environment, prototyped on our own FPGA-based boards [21]. DF-Threads allows us to offload portions of code to a hardware engine in order to achieve a better scalability than a software scheduler [24]–[27].

In a first instance, we explored the design space by using the HPLabs COTSon simulator [28] to find the most efficient implementation of the execution model. After that, we tested the DF-Threads on a cluster of AXIOM-Boards (Figure 1) [29]. The AXIOM-Board has several low power cores and an FPGA. Moreover, we can build a complete distributed system, in which AXIOM-Boards can be connected through inexpensive high-speed custom USB-C cables, reaching up to 18 Gb/s per channel (and having four available channels).

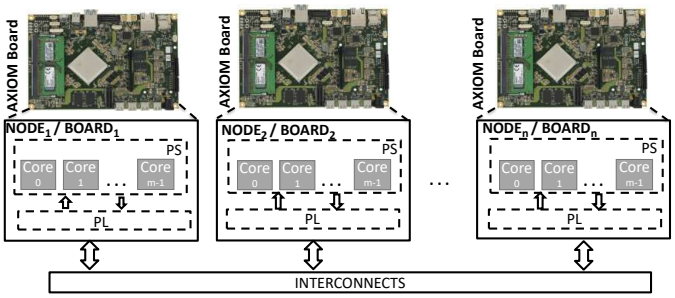

Fig. 1: Architecture of the distributed system based on the AXIOM boards. The Distributed System consists of N Nodes based on an FPGA SoC, which includes a Processing System (PS) and Programmable Logic (PL). The nodes of the system are connected through USB-C cables without the need of an external switch.

This paper makes the following contributions:

- It gives an example of how to translate a recursive generic program into Data-Flow programming style.
- It provides an initial quantitative comparison of DF-Threads, OpenMPI and CUDA.

In the Section II, we briefly describe the lightweight Data-Flow programming model; in Section III we illustrate the methodology used for the experiments; in Section IV, we evaluate our Data-Flow execution model, and finally, we conclude the paper.

II. A Lightweight Data-Flow Programming Model

The DF-Threads *execution model* relies on the program Data-Flow graph, in which each node of the graph represents a fine-grain thread named DF-Thread. The execution of the DF-Threads follows the producer-consumer paradigm, in which a DF-Thread (consumer) can execute only when all its inputs have been produced by other DF-Threads (producers). The lifetime of a DF-Thread is defined by 4 API calls (potentially instructions) [30], which are briefly recalled in Table I. With additional instructions, it is also possible to subscribe portions of shared memory for collective operations.

For the purpose of easier mapping generic program code, we elevated this API to a lightweight programming model, in which DF-Threads are simple C functions without the need of using the stack for passing parameters and with the addition of Data-Flow semantics. A DF-Threads can therefore be implemented as illustrated in Figure 2, with explicit management of the input frame (where input data is stored) in coordination with the linker. In Table I, we give the semantics of the *df_ldframe* and *df_destroy* typically placed respectively at the beginning and end of the DF-Thread:

```
void a_df_thread (void) {
   df_ldframe()
   <df_thread_body>
   df_destroy()
}
```

Fig. 2: A DF-Thread in C language.

As an example, we show here how we can translate the Recursive Fibonacci code into a DF-Thread (Figure 3). In this case, we map the original code (left) into two DF-Threads named *fibo* and *adder* (right). The *df_schedule* defines how many inputs the next instances will receive. The *df_write* fills up the input frames of next instances. As soon as all inputs of the target thread have been written, the target thread is executable. At the end, the DF-Thread notifies that its metadata can be removed (*df_destroy*). This approach allows us to use a standard compiler (i.e., GCC) for producing the binary for the target architecture (e.g., x86_64, aarch64).

III. Methodology

In this section, we describe the experimental work flow and the software tools that we used during the design and test of the DF-Threads execution model.

```
int fibo (int n)
{
  if (n <= 1) return n;
  return fibo(n-1)+fibo(n-2);
}
```

```
void fibo(void) { // DF-Thread Fibonacci
  uint64_t* myfp =(uint64_t*) df_ldframe();
  int n = myfp[1];
  if (n <= 1) {
   df_write(myfp[0],n);
  }else {
   uint64_t* tfib1 = df_schedule(&fibo,2);
   uint64_t* tfib2 = df_schedule(&fibo,2);
   df_write(tfib1[1], n-1);
   df_write(tfib2[1], n-2);
   uint64_t* tadd = df_schedule(&adder,3);
   df_write(tadd[0],  &myfp[0]);
   df_write(tfib1[0], tadd+1));
   df_write(tfib2[0], tadd+2);
   }
   df_destroy();
}
void adder(void) {
  uint64_t* myfp =(uint64_t*)df_ldframe();
  uint64_t f1 = myfp[1];
  uint64_t f2 = myfp[2];
  df_write(myfp[0],f1+f2);
  df_destroy();
}
```

Fig. 3: Example of translation of the Recursive Fibonacci program from C code into DF-Threads API. The two main operations (fibo and adder) are translated into the DF-Threads API.

TABLE I: DF-Threads API

void* df_ldframe();
Loads the data from the (self) input frame
void* df_schedule(void* ip, uint64_t sc);
This function allocates the resources: a data frame of size 'sc' words and returns its Frame Pointer (fp); 'ip' specifies the Instruction Pointer to the first instruction of the code. The allocated DF-Thread is not executed until its 'sc' reaches 0.
void df_write(void* fp, uint64_t val);
The data 'val' is stored into the frame pointed by 'fp'. Note: we assume that writes are snooped by the architecture (in particular by the hardware scheduler) so that, for every word that is written, the 'sc' of the DF-Thread - to which the fp belongs – is decremented.
void df_destroy();
The thread that invokes df_destroy finishes and its frame is freed.

In a preliminary phase, we defined the DF-Threads execution model into a customized version of the HP Labs COTSon Simulator [31]–[34], which permits us to decouple the functional execution from the timing behavior for an easier modeling. Thanks to COTSon simulator, we can model a complete distributed system with many-cores and multi-nodes, in which it is possible to run an off-the-shelf Linux Distribution for a full system simulation [35]. We compared our implementation with: i) OpenMPI, a typically used programming model for clusters and ii) CUDA as it is another widely used solution for performance scaling.

We also needed to design supporting tools to reduce the experimentation time from days/weeks to hours/minutes [36].

In order to have a realistic base for the timing, we also validated our simulations against the timing obtained on the AXIOM-boards, where we gradually migrated the designed Intellectual Property (IP) blocks (Figure 1). The AXIOM platform includes four 64-bit ARM Cortex-A53 cores (at 1.5 GHz); 32 KiB L1 Cache and 1MiB of L2 Cache, programmable logic and fast transceivers .

The Processing Systems (PS) of the AXIOM-board, runs a full Linux Ubuntu 16.04 and it starts the program, while the accelerated portions are offloaded, via the programming

model illustrated in Section II, to the Programmable Logic (PL). The PL also includes a Network Interface (NI) that allows the fast communication among the AXIOM-boards. At the hardware level, the soft-IPs are also responsible to distribute the workload among the nodes of the Distributed System and manage the metadata of the DF-Threads. For the CUDA experiments, we used the Tesla-C1060 board, with 240 CUDA core, 610 MHz GPU clock and 4 GiB of RAM.

IV. Experimental Results

For the sake of this initial exploration, we consider two simple benchmarks for the evaluation of the DF-Threads: Recursive Fibonacci and Matrix Multiplication.

- **Recursive Fibonacci** (RFIB) benchmark has been chosen to stress the thread management and quickly evaluate the performance, while there is a need of scheduling many threads. This benchmark takes as input the n of Fibonacci and a threshold which stops the generation of the parallel recursive calls.
- **Matrix Multiplication** (MM) benchmark involves more memory operations than RFIB and also is a widely used kernel in machine learning. We used a blocked Matrix Multiplication implementation, where a matrix is partitioned in multiple sub-matrices, or blocks, according to the block size that is set. We use square matrices with 448 as size (to avoid a multiple of a power of two, which may cause multiple cache conflicts on a few cache lines) and with 8 as block size. We focus our measurements only on the computational Region of Interest (ROI) of the benchmark as it is the usual practice in comparisons. The Matrix Multiply algorithm used to evaluate OpenMPI and CUDA is the standard available version for such programming models. Results of the benchmarks are checked for correctness at the end of the run. Multiple runs (at least 5) have been also repeated to reduce possible statistical oscillations.

A. Recursive Fibonacci

We evaluate the Recursive Fibonacci benchmark with an input size of 35 and 13 as threshold, by varying the number of nodes of the distributed system up to 16. As can be seen in the Figure 4, DF-Threads show a good degree of scalability. The results confirm that the scheduler of the DF-Threads can handle and distribute properly many fine-grain threads among multiple nodes. CUDA and OpenMPI has been not evaluated with the Fibonacci benchmark due to their poor effectiveness with recursive execution on such platforms.

B. Block Matrix Multiply

We use the GFLOPS/core metric to compare the performance of the DF-Threads with OpenMPI and CUDA, due to the currently lower number of cores of our Distributed System in respect of CUDA. The matrix size is 448 with 8 as block size. As we can see in the Figure 5, the GFLOPS/core of the DF-Threads outperforms both CUDA and OpenMPI.

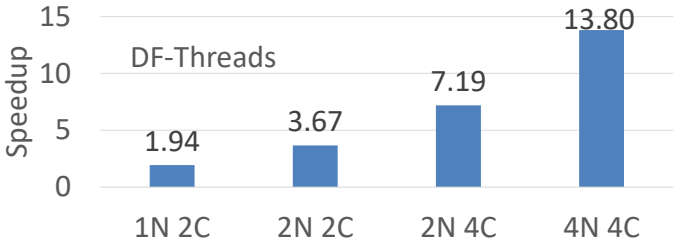

Fig. 4: Speedup of the Recursive Fibonacci benchmark with input size 35 and threshold 13, by using the DF-Threads and by varying the number of nodes (N) and cores (C) of the distributed system.

OpenMPI is outperformed by a large factor of about 5.5x in the case of 1 node (1N) or about 140x in the case of 16 nodes (16N). This is due to several factors: first of all the OpenMPI runtime library represents a wide middleware layer; secondly, there is the need of invoking system calls that in turn may need a time consuming operating system activity to move content buffer and manage the send and receive operations on the physical media. In the DF-Threads such overheads are reduced, thanks to the simpler interface and the hardware management of the data frames and thread metadata. As depicted in Figure 6, the kernel activity of the DF-Threads is quite limited in comparison with OpenMPI.

CUDA is outperformed by almost a factor of 1.7x among all configurations of the distributed system, due to the efficiency of our scheduling mechanisms. The CUDA platform that we used is one of the first Tesla boards available on the market, but our implementation of DF-Threads is also at the first version and it is not yet optimized. Therefore, the DF-Threads is capable to exploit better the resources of the distributed system, also thanks to the parallelism exposed by the Data-Flow mechanism and the DF-Thread API.

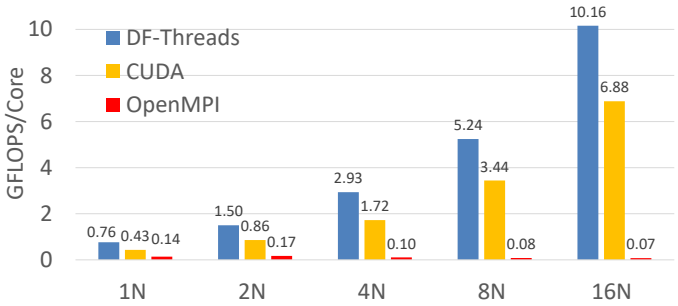

Fig. 5: GFLOPS/core comparison between DF-Threads, OpenMPI and CUDA by using the block Matrix Multiply benchmark with 448x448 as matrix size and 8 as block size. The number of nodes (N) varies from 1 to 16.

Moreover, we think that there is still much space for further optimization of the DF-Threads. For example, the data locality could be improved and we are investigating a pre-fetching policy, which could load the data into the cache before starting the execution of the DF-Threads.

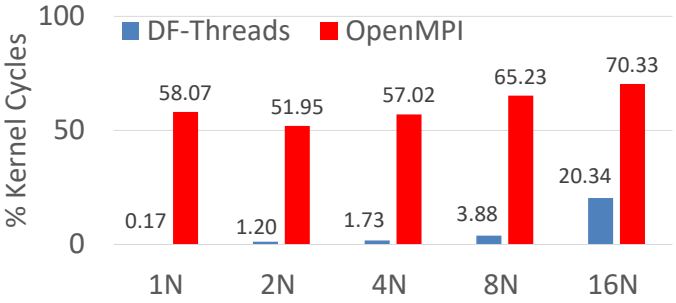

Fig. 6: DF-Threads and OpenMPI kernel cycles (left) for the block Matrix Multiplication benchmark with matrix size 448x448 and block size 8. The number of nodes (N) varies from 1 to 16.

V. Conclusions

The Data-Flow execution model is a viable paradigm to be explored today to achieve high degree of parallelism in the modern many-cores multi-nodes architectures. This paper presented how the DF-Threads execution model can bridge the Data-Flow execution to a simple C-based programming model through the DF-Threads API. Our experiments show the capability of the DF-Thread execution model to distribute and manage many fine-grain threads among multiple nodes. We compared the DF-Thread with OpenMPI and CUDA by using the block Matrix Multiplication benchmark and we found that DF-Thread outperform both OpenMPI and CUDA in terms of GFLOPS/core. Future work will expand the capability of automatic translation and demonstrate a larger set of benchmarks.

Acknowledgment

The work of this paper is partly funded by the European Commission on AXIOM H2020 (id. 645496), TERAFLUX (id. 249013), HiPEAC (id. 779656). The authors would like to thank the anonymous reviewers for their helpful comments.

References

[1] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, pp. 126–132, ACM, 1975.

[2] J. B. Dennis, "Data flow supercomputers," *Computer*, pp. 48–56, 1980.

[3] W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM computing surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[4] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Trans. on Computers*, pp. 940–948, 1986.

[5] A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero, and R. Giorgi, "Dataflow support in x86-64 multicore architectures through small hardware extensions," in *IEEE Proc. DSD*, pp. 526–529, August 2015.

[6] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," *Parallel Comput.*, vol. 25, Dec. 1999.

[7] L. A. Marzulo, T. A. Alves, F. M. França, and V. S. Costa, "Couillard: Parallel programming via coarse-grained data-flow compilation," *Parallel Computing*, vol. 40, no. 10, pp. 661–680, 2014.

[8] C. Meenderinck and B. Juurlink, "Nexus: Hardware support for task-based programming," in *2011 14th Euromicro Conference on Digital System Design*, pp. 442–445, IEEE, 2011.

[9] J. Castrillon, R. Leupers, and G. Ascheid, "Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, 2013.

[10] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the 26th International Conf on Compiler Construction*, pp. 76–86, ACM, 2017.

[11] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: a functional data-parallel ir for high-performance gpu code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 74–85, IEEE, 2017.

[12] S. Ertel, J. Adam, and J. Castrillon, "Supporting fine-grained dataflow parallelism in big data systems," in *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 41–50, ACM, 2018.

[13] K. Huang, I. Bacivarov, J. Liu, and W. Haid, "A modular fast simulation framework for stream-oriented mpsoc," in *2009 IEEE International Symposium on Industrial Embedded Systems*, pp. 74–81, IEEE, 2009.

[14] J. R. Gurd, "The manchester dataflow machine," *Computer Physics Communications*, vol. 37, no. 1-3, pp. 49–62, 1985.

[15] D. E. Culler and G. M. Papadopoulos, "The explicit token store," *Journal of Parallel and Distributed Comp.*, vol. 10, no. 4, pp. 289–308, 1990.

[16] Theodoropoulos *et al.*, "The axiom platform for next-generation cyber physical systems," *ELSEVIER Microprocessors and Microsystems*, pp. 540–555, 2017.

[17] R. Giorgi, "Scalable embedded systems: Towards the convergence of high-performance and embedded computing," in *EUC 2015*.

[18] C. Alvarez *et al.*, "The AXIOM software layers," in *IEEE Proc. 18th EUROMICRO-DSD*, pp. 117–124, Aug. 2015.

[19] C. Alvarez *et al.*, "The AXIOM software layers," *ELSEVIER Microprocessors and Microsystems*, vol. 47, Part B, pp. 262–277, 2016.

[20] R. Giorgi, N. Bettin, P. Gai, X. Martorell, and A. Rizzo, *AXIOM: A Flexible Platform for the Smart Home*, pp. 57–74. Springer, 2016.

[21] R. Giorgi, "AXIOM: A 64-bit reconfigurable hardware/software platform for scalable embedded computing," in *IEEE 6th Mediterranean Conf. on Embedded Computing (MECO)*, pp. 113–116, June 2017.

[22] D. Theodoropoulos *et al.*, "The AXIOM project (agile, extensible, fast i/o module)," in *Proc. 15th Int.l Conf. on Embedded Computer Systems: Architecture, MOdeling and Simulation*, pp. 262–269, July 2015.

[23] R. Giorgi and P. Faraboschi, "An introduction to DF-Threads and their execution model," in *IEEE MPP*, (Paris, France), pp. 60–65, Oct. 2014.

[24] X. Tan, J. Bosch, D. Jiménez-González, C. Álvarez-Martínez, E. Ayguadé, and M. Valero, "Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 225–234, IEEE, 2016.

[25] K. Stavrou *et al.*, "Programming abstractions and toolchain for dataflow multithreading architectures," in *Proc. 8th Int.l Symp. on Parallel and Distributed Computing (ISPDC 2009)*, pp. 107–114, IEEE, July 2009.

[26] L. Verdoscia and R. Giorgi, "A data-flow soft-core processor for accelerating scientific calculation on FPGAs," *Mathematical Problems in Engineering*, vol. 2016, pp. 1–21, Apr. 2016. article ID 3190234.

[27] R. Giorgi and A. Scionti, "A scalable thread scheduling co-processor based on data-flow principles," *ELSEVIER Future Generation Computer Systems*, vol. 53, pp. 100–108, Dec. 2015.

[28] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52–61, 2009.

[29] R. Giorgi, F. Khalili, and M. Procaccini, "Axiom: A scalable, efficient and reconfigurable embedded platform," in *IEEE Proc. Design, Automation and Test in Europe (DATE)*, (Florence, Italy), pp. 1–6, Mar. 2019.

[30] R. Giorgi, "Teraflux: Exploiting dataflow parallelism in teradevices," in *ACM Computing Frontiers*, (Cagliari, Italy), pp. 303–304, May 2012.

[31] N. Ho *et al.*, "Simulating a multi-core x86-64 architecture with hardware isa extension supporting a data-flow execution model," in *IEEE Proc. AIMS-2014*, (Madrid, Spain), pp. 264–269, Nov. 2014.

[32] N. Ho *et al.*, "Enhancing an x86_64 multi-core architecture with data-flow execution support," in *ACM Computing Frontiers*, pp. 1–2, 2015.

[33] A. Portero, Z. Yu, and R. Giorgi, "Teraflux: Exploiting tera-device computing challenges," *ELSEVIER*, vol. 7, pp. 146–147, 2011.

[34] A. Portero *et al.*, "Simulating the future kilo-x86-64 core processors and their infrastructure," in *45th Annual Simulation Symp. (ANSS12)*, (Orlando, FL), pp. 62–67, Mar 2012.

[35] R. Giorgi, F. Khalili, and M. Procaccini, "Analyzing the impact of operating system activity of different linux distributions in a distributed environment," in *IEEE Euromicro Int.l Conf. on Parallel, Distributed, and Network-Based Processing*, (Pavia, Italy), pp. 422–429, Feb. 2019.

[36] R. Giorgi, F. Khalili, and M. Procaccini, "A design space exploration tool set for future 1k-core high-performance computers," in *ACM RAPIDO Workshop*, pp. 1–6, 2019.