# DF-Threads: A Scalable and Efficient Execution Paradigm for Edge Computing and HPC

Roberto Giorgi[1]

*Department of Information Engineering and Mathematics, University of Siena, Italy*

**Abstract**
Scalable and distributed computing systems are widely deployed but hide a large toll in terms of energy consumption. A wider adoption of dataflow concepts at any level of the software/hardware stack of HPC system can lead to a reduction of the intrinsic inefficiency of current systems. By leveraging structured parallel programming based on FastFlow, we are exploring the effectiveness of DataFlow Threads (DF-Threads) in tandem with such programming model for Edge Computing and HPC.

## 1. Introduction

Dataflow methodologies have been explored at multiple levels of granularity. At the instruction level, superscalar processors have effectively implemented this by enabling instructions to execute out-of-order as soon as their operands are ready [1, 2].

Programming paradigms such as OmpSs2 and OpenMP manage data flow among tasks and orchestrate the scheduling of potentially large dataflow/asynchronous tasks across available computational resources, including CPU cores, GPU cores, and accelerators [3, 4, 5], while new programming workflows are deemed more appropriate for the Compute Continuum [6, 7].

Despite these advancements, the hardware-software interface still faces challenges: i) a streamlined and efficient mechanism for managing thread-level parallelization, and ii) a widely accepted memory consistency model.

These challenges stem from the requirements for synchronization, consistency, and coherency—a persistent issue exacerbated by the proliferation of cost-effective, massively parallel systems and domain-specific accelerators. The TERAFLUX [8] and AXIOM projects [9, 3] have investigated DataFlow-Threads (DF-Threads) as a potential solution for improving performance scalability while providing a straightforward interface for future massively parallel systems [10, 11, 4] DF-Threads can be integrated into the architecture with the addition of a few new instructions, thereby enhancing existing processors to offer more efficient and effective parallelism. Our experiments indicate nearly perfect scalability for systems with over 1000 general-purpose x86_64 (extended) cores operating on off-the-shelf Linux-based operating systems [12, 13].

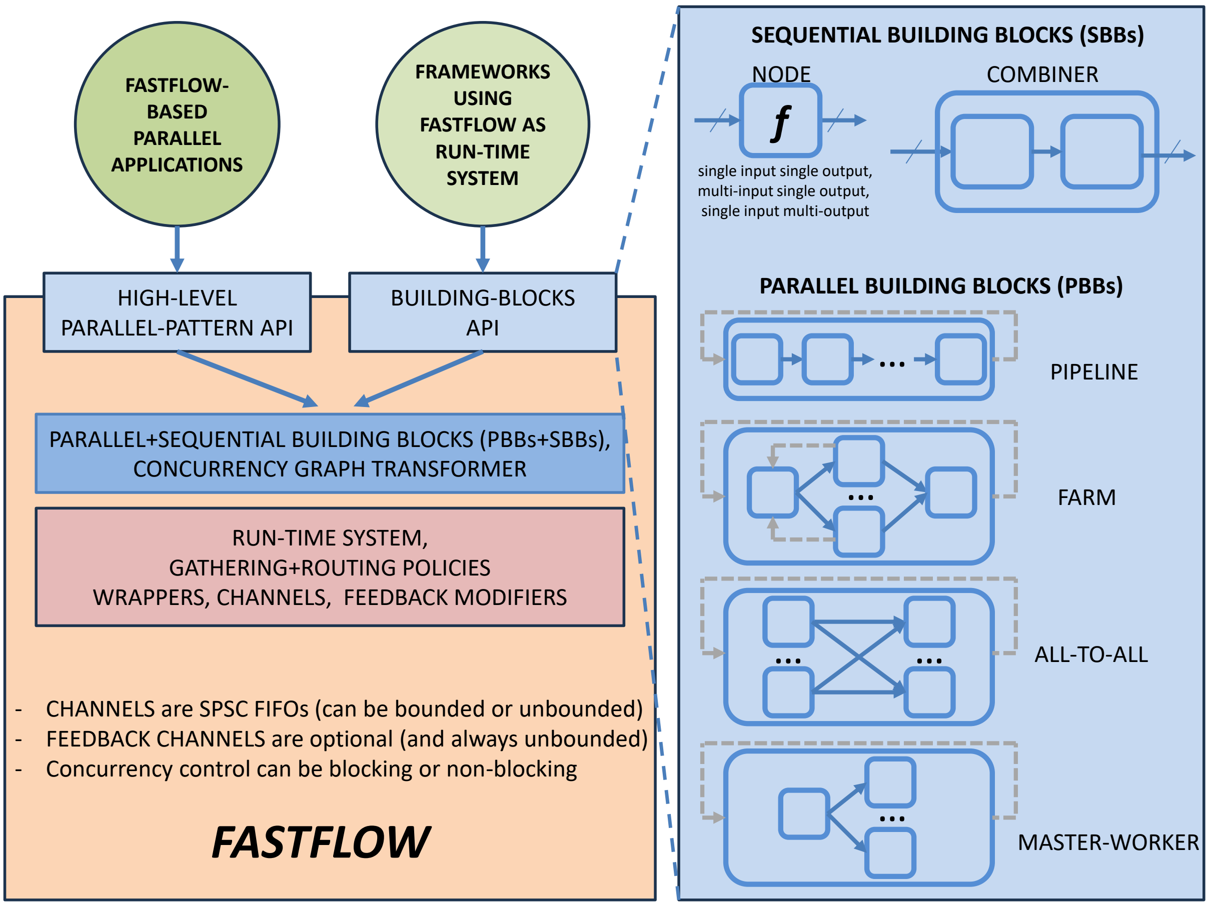
In this work, for the first time, a dataflow-based structured parallel programming model like FastFlow [14, 15] is connected to a lower-level dataflow execution model like the DF-Threads. The main objective of this work is to improve the overall energy efficiency and programmability of HPC applications.

## 2. A Brief introduction to DF-Threads and FastFlow

The FastFlow framework [14] consists of a header-only C++ template library that allows programmers to create parallel applications structured as dataflow graphs. Adhering to the thread-based parallelism model, in each FastFlow node embodies a sequential computational unit executed by a dedicated thread. Communication between nodes relies on non-blocking synchronization for fast data processing, especially in high-frequency streaming environments.

*BigHPC2024: Special Track on Big Data and High-Performance Computing, co-located with the 3rd Italian Conference on Big Data and Data Science, ITADATA2024, September 17 – 19, 2024, Pisa, Italy.*

giorgi@unisi.it (R. Giorgi)



FASTFLOW-BASED PARALLEL APPLICATIONS
FRAMEWORKS USING FASTFLOW AS RUN-TIME SYSTEM
HIGH-LEVEL PARALLEL-PATTERN API
BUILDING-BLOCKS API
PARALLEL+SEQUENTIAL BUILDING BLOCKS (PBBs+SBBs), CONCURRENCY GRAPH TRANSFORMER
RUN-TIME SYSTEM, GATHERING+ROUTING POLICIES WRAPPERS, CHANNELS, FEEDBACK MODIFIERS
- CHANNELS are SPSC FIFOs (can be bounded or unbounded)
- FEEDBACK CHANNELS are optional (and always unbounded)
- Concurrency control can be blocking or non-blocking
FASTFLOW
SEQUENTIAL BUILDING BLOCKS (SBBs)
NODE
COMBINER
f
single input single output, multi-input single output, single input multi-output
PARALLEL BUILDING BLOCKS (PBBs)
PIPELINE
FARM
ALL-TO-ALL
MASTER-WORKER

**Figure 1:** (Left) FastFlow software stack. (Right) Graphical notation of the FastFlow's Building Blocks (from [17]).

Within a single node, FastFlow channels manage references to data allocated on the heap rather than to plain data, with ownership of these references being transferred from the sender (producer) to the receiver (consumer). The FastFlow programming paradigm has served as a foundational technology in large projects such as ParaPhrase, REPARA, RePhrase, TextaROSSA [16] and in Flagship-3 of ICSC Spoke-1 (FutureHPC). FastFlow *Building Blocks* define a reduced set of structured parallel components to build and orchestrate skeleton, parallel patterns and more complex parallel structures [17] (Fig. 1).

At a lower level in the software stack is the execution paradigm called "Dataflow Threads" (DF-Threads) [11] which has its roots in the dataflow execution models implemented in machines such as the IBM BlueGene-C (Cyclops64) [18] and the Scheduled Data-Flow (SDF) architecture [19]. The DF-Threads paradigm can be used to provide performance scalability, extensibility, fault tolerance (repeating computations in time and/or space whose inputs are preserved), and isolation. DF-Threads is a hybrid dataflow/controlflow representation of computation that allows for a reduction in excessive synchronization. DF-Threads can also be mapped onto many forms of parallelization provided by other programming models. DF-Threads were introduced in the TERAFLUX project for the x86_64 architecture [11, 4]. In the AXIOM project the DF-Thread execution model was further developed with specific hardware to support scheduling [20].

Given the above premises, it seems natural to connect the high-level programming model provided by FastFlow with the lower-level dataflow support provided by the DF-Threads (Fig. 2). The combination of this two framework can provide a "win-win" solution as it can enhance the support for distributed execution of FASTFLOW, while it can provide a power high-level programming model for the DF-threads.

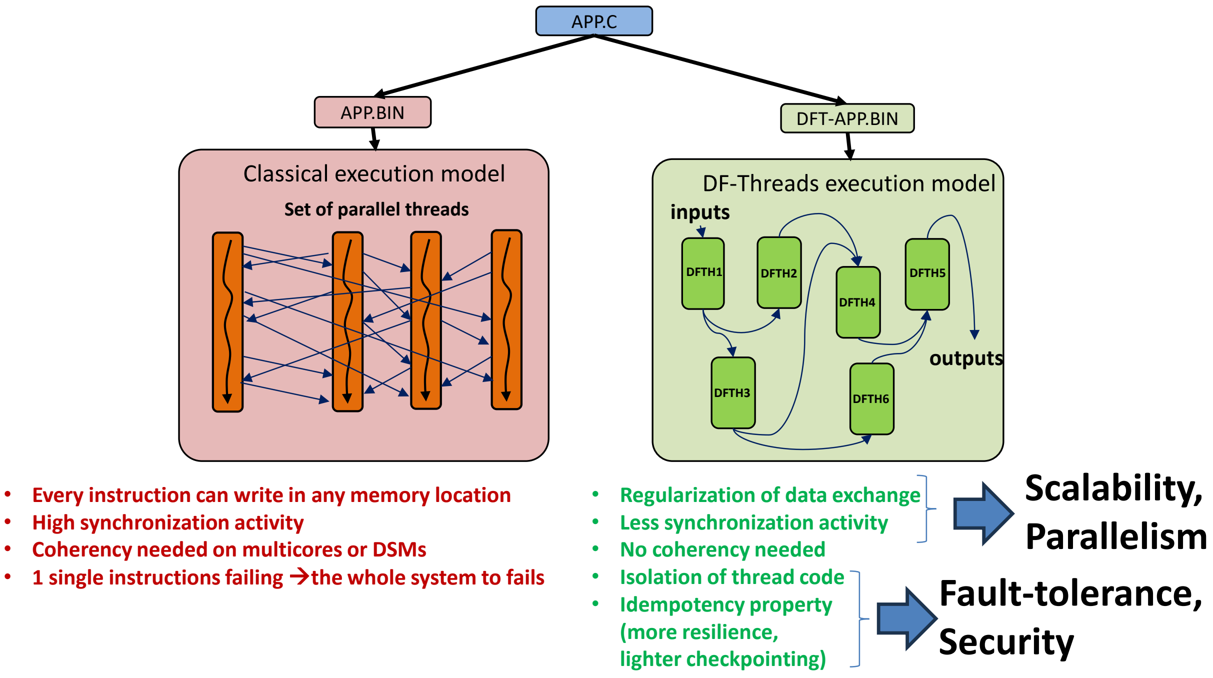

APP.C
APP.BIN
DFT-APP.BIN
Classical execution model
Set of parallel threads
DF-Threads execution model
inputs
DFTH1
DFTH2
DFTH3
DFTH4
DFTH5
DFTH6
outputs
Every instruction can write in any memory location
High synchronization activity
Coherency needed on multicores or DSMs
1 single instructions failing → the whole system to fails
Regularization of data exchange
Less synchronization activity
No coherency needed
Isolation of thread code
Idempotency property (more resilience, lighter checkpointing)
Scalability, Parallelism
Fault-tolerance, Security

**Figure 2:** How DF-Threads work (simplified overview).

## 3. Preliminary Evaluation

### 3.1. Methodology

To show the potential of DF-Threads execution we compare here the execution time in case of a simple benchmark with OpenMPI [21]. The evaluation is based on the HP-Labs COTSon[1] simulator [22], an X86_64 full-system simulator, which also accounts for OS effects. Key parameters for the modeled cores are detailed in Table 1. Furthermore, the simulator has been extended to support DF-Threads [11], including modeling a Distributed Thread Scheduler [23].

Table 1: Multicore architectural parameters.

| Parameter | Description |
|---|---|
| SoC | single-core connected by a shared-bus, IO-hub, MC, high-speed transceivers |
| Core | 3GHz, in-order superscalar |
| Branch Predictor | two-level (history length=14bits, pattern-history table=16kB, 8-cycle missprediction penalty) |
| L1 Cache | Private I-cache 32 KB, 4 ways, 2-cycle latency - private D-cache 32 KB, 2 ways, 3-cycle latency |
| L2 Cache | Private 256 KB, 4 ways, 5-cycle latency |
| L3 Cache | Shared 4GB, 4 ways, 20-cycle latency |
| Coherence protocol | MOESI |
| Main Memory | 4 GB, 100 cycles latency |
| I-L1-TLB, D-L1-TLB | 64 entries, full-associative, 1-cycle latency |
| L2-TLB | 512 entries, direct access, 1-cycle latency |
| Write/Read queues | 200 Bytes each, 1-cycle latency |

### 3.2. Matrix Multiplication Benchmark

For the initial tests, we selected the Matrix Multiplication kernel. The Matrix Multiplication benchmark has the following characteristics: blocked matrix multiplication using the classical 3 nested loops algorithm; matrices [2] of size $n \times n$, where $n = 216, 432, 864$; block size $b = 8$.

[1]Please note that the COTSon simulator is open-source: https://cotson.sf.net and other related open-source software is also available at https://download.axiom-project.eu

[2]Please note that even if this matrix sizes may seem small, they cause a quite large simulation time (in terms of hours or days in case of power estimation). Therefore, our focus is to derive the main properties of the framework while suggesting future

The input matrices' sizes are chosen to avoid particular multiples of powers of 2 to prevent excessive cache conflicts that might skew the evaluation of the system's basic behavior. The threads in each test case are generated so that each thread performs the dot-products of each block, thus the expected number of threads is $n/b$.

In the benchmark, we focus on the computation region of interest, excluding the data preparation and result verification from the evaluation. Additionally, any I/O messages (e.g., 'printf') are removed from the computational part. Consequently, any OS-related activity pertains only to managing the data needed for computation or moved across nodes.

For DF-Threads, the kernel activity is mostly under 10% (except in the case of 16 nodes, where the number of threads $n/b$ is too low, resulting in an imbalance).

### 3.3. DF-Threads versus OpenMPI

In OpenMPI, the workload is distributed among the available worker threads, even in a single-node configuration, which incurs a greater overhead compared to DF-Threads. A direct comparison between OpenMPI and DF-Threads is provided in Fig. 3, where it is evident that:

- OpenMPI slightly benefits from local cores.
- DF-Threads scales well with both the number of cores and the number of nodes, achieving an advantage of about 28x compared to the execution OpenMPI on the same number of nodes for n=216. This is due to the good performance and scaling of DF-Threads as well as the relatively poor performance of OpenMPI for this input size.

For an input size of 864, DF-Threads still shows a significant advantage, about 3.5x over OpenMPI on the same number of nodes, confirming DF-Threads' competitive edge. When comparing a system with 8 nodes and 4 cores per node, DF-Threads achieve a speedup of 14x over OpenMPI (not shown in the figure). The diminishing execution time advantage of DF-Threads vs. OpenMPI highlights a large advantage for DF-Threads in case of smaller granularity. However, our preliminary results confirm that when combined with the power consumption evaluation, the DF-Threads offer a greater advantage also as the number of nodes increases.

### 3.4. FastFlow versus OpenMPI

While FastFlow was originally designed for support shared-memory application running on multicores, the distributed version of FastFlow currently optionally uses the OpenMPI backend. Furthermore, the shared-memory version defaults to using non-blocking concurrency control mode, while the distributed version employs blocking mode for its runtime system.

FastFlow supports various parallel programming paradigms, and matrix multiplication can be implemented using different patterns such as pipelines or task farms. While the framework primarily focuses on shared-memory parallelism, it also supports distributed memory systems using extensions like *FastFlow-DM*.

For distributed systems, leveraging OpenMPI alongside FastFlow can provide an efficient solution. FastFlow's high-level parallel constructs simplify the development process while maintaining performance. Previous experiments on FastFlow targeting distributed systems show that there is a substantial overhead in using MPI based primitives [24]. Therefore, DF-Threads can offer a good opportunity for enhancing the performance of a parallel application, while relying and skeleton, parallel patterns and building blocks offered by FastFlow.

engineering work.

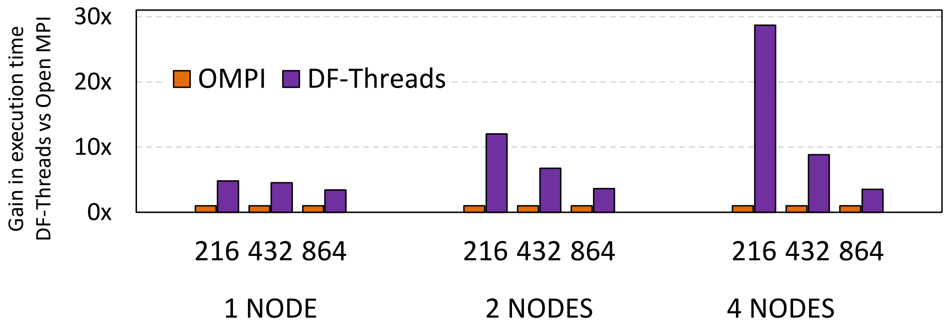


**Figure 3:** (Data adapted from [4]) Gain in execution time of DF-Threads compared to Open MPI (normalized to 1). The benchmark is Blocked Matrix Multiplication for different sizes of the square matrices (216, 432, 864 – the block size is 8 elements) and 1, 2, 4 nodes. FULL-SYSTEM simulation.

### 3.5. Power Savings

The power consumption estimation is also based on the COTSon framework via the MCPAT utility. In the preliminary experiments, with the matrix size $n = 216$ we have measured a consistent power saving when using DF-Threads (Tab.2) of a factor of about 2.7x.

Table 2: Power Consumption of DF-Threads versus OpenMPI for the matrix size $n = 216$, 1 node, 1 core.

| OpenMPI | DF-Threads |
|---|---|
| 0.5115 W | 0.1867W |

## 4. Conclusions

In this paper we analyzed the potentials offered by using a tandem approach based on FastFlow for the high-level programming and on DF-Threads for the underlying execution model. Based on OpenMPI as a common reference for evaluating this potential of improvement, we expect a considerable benefit for both extending FastFlow and DF-Threads. The future work will include a prototype system for demonstrating the combined performance on a set of parallel applications such as P$^3$ARSEC suite [25].

## Acknowledgement

We thank the anonymous reviewers for their comments that helped improve this work. This work is partly funded by the European Union - NextGenerationEU - via the PNRR M4C2-Inv1.4 Italian Research Center on High-Performance Computing, Big-Data and Quantum Computing, cascade funding project EDGE-ME, MUR-ID: CN0000013.

## References

[1] W. Hwu, P. Y. N., Hpsm, a high performance restricted data flow architecture having minimal functionality, ACM SIGARCH Computer Architecture News 14 (1986). URL: http://dx.doi.org/10.1145/17356.17391. doi:10.1145/17356.17391.

[2] D. K. Wang, N. S. Kim, Diag: a dataflow-inspired architecture for general-purpose processors, Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (2021). URL: http://dx.doi.org/10.1145/3445814.3446703. doi:10.1145/3445814.3446703.

[3] R. Giorgi, S. Mazumdar, S. Viola, P. Gai, S. Garzarella, B. Morelli, D. Pnevmatikatos, D. Theodoropoulos, C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, Modeling

multi-board communication in the axiom cyber-physical system, Ada User Journal 37 (2016) 228–235.

[4] R. Giorgi, Scalable embedded computing through reconfigurable hardware: comparing df-threads, cilk, OpenMPI and jump, ELSEVIER Microprocessors and Microsystems 63 (2018) 66–74. doi:10.1016/j.micpro.2018.08.005.

[5] R. Giorgi, F. Khalili, M. Procaccini, AXIOM: A Scalable, Efficient and Reconfigurable Embedded Platform, in: IEEE Proc.DATEi, IEEE, Florence, Italy, 2019, pp. 1–6.

[6] I. Colonnelli, M. Aldinucci, B. Cantalupo, L. Padovani, S. Rabellino, C. Spampinato, R. Morelli, D. C. Rosario, N. Magini, C. Cavazzoni, Distributed workflows with jupyter, Future Generation Computer Systems 128 (2022) 282–298. URL: http://dx.doi.org/10.1016/j.future.2021.10.007. doi:10.1016/j.future.2021.10.007.

[7] M. Danelutto, P. Dazzi, M. Torquati, Structuring the continuum, in: L. Barolli (Ed.), Advanced Information Networking and Applications, Springer Nature Switzerland, Cham, 2024, pp. 212–223.

[8] R. Giorgi, Exploring future many-core architectures: The TERAFLUX evaluation framework, in: Advances in Computers, Advances in Computers, Elsevier, 2017, pp. 33 – 72. URL: http://www.sciencedirect.com/science/article/pii/S0065245816300584. doi:10.1016/bs.adcom.2016.09.002.

[9] D. Theodoropoulos, D. Pnevmatikatos, C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, C. Segura, C. Fernandez, D. Oro, J. R. Saeta, P. Gai, A. Rizzo, R. Giorgi, The axiom project (agile, extensible, fast i/o module), 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) (2015). URL: http://dx.doi.org/10.1109/samos.2015.7363684. doi:10.1109/samos.2015.7363684.

[10] R. Giorgi, et al., TERAFLUX: Harnessing dataflow in next generation teradevices, ELSEVIER Microprocessors and Microsystems 38 (2014) 976–990.

[11] R. Giorgi, P. Faraboschi, An introduction to DF-Threads and their execution model, in: IEEE MPP, Paris, France, 2014, pp. 60–65.

[12] N. Ho, A. Portero, M. Solinas, A. Scionti, A. Mondelli, P. Faraboschi, R. Giorgi, Simulating a multi-core x86-64 architecture with hardware isa extension supporting a data-flow execution model, in: IEEE Proceedings of the AIMS-2014, Madrid, Spain, 2014, pp. 264–269. doi:10.1109/AIMS.2014.41.

[13] L. Verdoscia, R. Giorgi, A data-flow soft-core processor for accelerating scientific calculation on FPGAs, Mathematical Problems in Engineering 2016 (2016) 1–21. Article ID 3190234.

[14] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: High-level and efficient streaming on multicore, 2017. URL: http://dx.doi.org/10.1002/9781119332015.ch13. doi:10.1002/9781119332015.ch13.

[15] M. Danelutto, G. Mencagli, A. Ottimo, F. Iannone, P. Palazzari, Fastflow targeting fpgas, in: 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), IEEE, 2023. URL: http://dx.doi.org/10.1109/PDP59025.2023.00023. doi:10.1109/pdp59025.2023.00023.

[16] W. Fornaciari, F. Terraneo, G. Agosta, Z. Giuseppe, L. Saraceno, G. Lancione, D. Gregori, M. Celino, The TEXTAROSSA Approach to Thermal Control of Future HPC Systems, Springer International Publishing, 2022, p. 420–433. URL: http://dx.doi.org/10.1007/978-3-031-15074-6_27. doi:10.1007/978-3-031-15074-6_27.

[17] N. Tonci, M. Torquati, G. Mencagli, M. Danelutto, Distributed-memory fastflow building blocks, International Journal of Parallel Programming 51 (2022) 1–21. URL: http://dx.doi.org/10.1007/s10766-022-00750-5. doi:10.1007/s10766-022-00750-5.

[18] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, G. Gao, A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture, Proceedings 20th IEEE International Parallel Distributed Processing Symposium (2006). URL: http://dx.doi.org/10.1109/ipdps.2006.1639301. doi:10.1109/ipdps.2006.1639301.

[19] K. M. Kavi, R. Giorgi, J. Arul, Scheduled dataflow: Execution paradigm, architecture, and performance evaluation, IEEE Trans. Computers 50 (2001) 834–846.

[20] A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, D. Theodoropoulos, D. Pnevmatikatos, P. Gai, S. Garzarella, D. Oro, J. Hernando, N. Bettin, A. Pomella, M. Procaccini, R. Giorgi, The axiom project: Iot on heterogeneous embedded platforms, IEEE Design and Test 38 (2021) 74–81. doi:10.1109/MDAT.2019.2952335.

[21] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proc., 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.

[22] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, D. Ortega, COTSon: infrastructure for full system simulation, SIGOPS Oper. Syst. Rev. 43 (2009) 52–61.

[23] R. Giorgi, A. Scionti, A scalable thread scheduling co-processor based on data-flow principles, ELSEVIER Future Generation Computer Systems 53 (2015) 100–108.

[24] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, M. Torquati, Targeting distributed systems in fastflow, 2013, pp. 47–56. doi:10.1007/978-3-642-36949-0_7.

[25] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, M. Danelutto, Bringing parallel patterns out of the corner: The p 3 arsec benchmark suite, ACM Transactions on Architecture and Code Optimization 14 (2017) 1–26. URL: http://dx.doi.org/10.1145/3132710. doi:10.1145/3132710.