UNIVERSITÀ
DI SIENA 1240

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Ph.D. Program in Smart Computing

Dipartimento di Ingegneria dell'Informazione (DINFO)
Dipartimento di Ingegneria dell'informazione e Scienze Matematiche (DIISM)

Reconfigurable Architectures for Accelerating Distributed Applications

A Graph Processing Application Case Study

Amin Sahebi

Dissertation presented in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in Smart Computing

Ph.D. Program in Smart Computing
University of Florence, University of Pisa, University of Siena

Reconfigurable Architectures for Accelerating Distributed Applications

A Graph Processing Application Case Study

Amin Sahebi

Advisor:

____

Prof. Roberto Giorgi

Head of the Ph.D. Program:

____

Prof. Stefano Berretti

Evaluation Committee:
Prof. Tullio Vardanega, *Università degli Studi di Padova, UNIPD*
Prof. Alessandro Lonardo, *Instituto Nazionale di Fisica Nucleare (INFN)*

XXXIV ciclo — January 2022

Copyright © 2022 by Amin Sahebi.

Acknowledgments

I want to first thank my supervisor Prof. Roberto Giorgi, and sincerely appreciate how he has been continuously encouraging and guiding me in these three years, and also how he has always been available, supportive of all of my efforts and struggles.

I want to thank my fiancé, Parisa, my life support and courage, this work could not be done without her all the motivation, support and love sent to me. I'm also grateful to my family, my parents, my lovely sister, and my brothers for their consistent support.

Then, I would like to thank my Supervisory Committee for their help, feedback, and suggestions within three phases of my annual evaluations. Prof. Antonio Prete from the University of Pisa and Prof. Sandro Bartolini from the University of Siena, plus the reviewers and evaluation committee members, Prof. Tullio Vardanega from Università degli Studi di Padova (UNIPD), and Prof. Allesandro Lonardo from Instituto Nazionale di Fisica Nucleare (INFN) for their insightful comments and suggestions to extremely strengthen the quality of this thesis.

In the last year of my Ph.D. I had this opportunity to spend 6 months as visiting researcher in a foreign institution. I took this opportunity and collaborated with the "Custom Computing Research Center" at Imperial College London, UK. Indeed, it was a great and fruitful collaboration. I want to thank Prof. Wayne Luk for his kind advise and admitted me to join his group and Prof. Georgi Gaidajiev for his suggestions and guidance throughout the collaboration. I am also thankful to Dear Marco Barbone for his kind suggestions, collaboration, and availability. As I have been admitted in as Smart Computing Program as a Ph.D. candidate with the "Regione Toscana scholarship", I also would like to thank "Regione Toscana" for the Pegaso grant. Moreover, this thesis is partly funded by the European Commission on AXIOM H2020 (id. 645496), TERAFLUX (id. 249013), and HiPEAC (id. 871174).

In addition, I would like also appreciate "Xilinx University Program" for their kind hardware donation to the University of Siena and giving me the access to the XACC Xilinx Adaptive Computing Cluster in the ETH Zürich center.

Siena, Italy
January 2022
Amin

Contents

Contents v

List of Figures vii

List of Tables xi

1 Introduction 1
1.1 Motivation and contribution 2
1.2 Thesis Structure 2

2 Related works 5
2.1 Dataflow architecture 6
2.2 Taxonomy of dataflow execution models 6
2.3 Fine-grained Parallelism Approaches in Dataflow Architecture 11
2.4 Dataflow Runtime Libraries 14
2.5 Dataflow Execution Models using Hardware Accelerator 16

3 Dataflow Execution Model Baseline Study 21
3.1 Methodology 22
3.2 Reference code and Hardware 24
3.3 Environment, Compilers, and Setups 24
3.4 Literature Study 27
3.5 Recursive Fibonacci 28
3.6 Blocked Matrix Multiplication 38
3.7 Blocked Cholesky Factorization 44
3.8 Histogram 48
3.9 Conclusion 52

4 Dataflow Runtime 53
4.1 DRT: A Lightweight Dataflow Runtime To Debug and Develop Dataflow Programs 53
4.1.1 Background 53
4.1.2 Writing dataflow codes with the DF-Threads API 55
4.1.3 Introducing DRT 57
4.1.4 Evaluation 59

5 Proposal for a Distributed Large Scale Graph Processing on multi-FPGA Platform 63
5.1 Introduction 63
5.2 Background and Motivation 66
5.3 Related Work 66
5.4 Hardware Implementation 71
5.4.1 Host Program 71
5.4.2 FPGA Kernels 72
5.5 Evaluation and Performance Model 78
5.6 Conclusion 80

6 Dataflow Case Studies 83
6.1 A Dataflow Methodology for Accelerating FFT 83
6.1.1 Introduction and Theoretical background 83
6.1.2 Reducing the needed twiddle factors 85
6.1.3 The 8-point FFT Data-Flow Graph 86
6.1.4 Experimental validation 88
6.1.5 Related work 89
6.1.6 Conclusion and future works 90

7 A Custom Board To Perform Distributed Computing 91
7.1 Gluon: The High-Speed Interconnect Solution 91
7.2 Conclusion 93

8 Conclusion and Future Studies 95

Bibliography 97

A Appendix A - Publications 111
A.1 Peer-reviewed Conference Papers 111
A.2 Workshops and Posters 111

A Appendix B - Parallel Programming Models Notes 113
A.1 MPI profiling with mpiP 113
A.2 Pure OpenMPI on Hyper threaded Hardware 117
A.3 Scripts 118

List of Figures

2.1 The most influential dataflow inspired and non-dataflow architectures and their timeline. 7

2.2 A codelet Graph (CDG) shows the codelets and their dependencies connected through a multiple threaded blocks linked together [141]. 13

2.3 A token-arc representation of data and dependency in the static and dynamic dataflow. 13

2.4 The Dataflow Thread execution model high-level architecture versus well-known modern dataflow approaches. 17

2.5 The abstract model of the processing node using dynamic dataflow along with synchronization unit proposed in [130] and [129]. 17

2.6 The dataflow RISC multiprocessor organization proposed in [109]. In this work the dataflow semantics and its instruction set based on I-structure method is mapped to the RISC processor. 18

2.7 RISC-V Dataflow Extension instruction illustration in (a) and (b) presented in [35]. (c) shows the dataflow token. Here the tag is used to match operands that are part of the same instruction; for example, if an instruction includes two operands, both of them will have the same tag. 19

2.8 An overview of the Picos and Rocket chip system architecture [108]. 19

3.1 The methodology to develop the desired baseline. 23

3.2 The separated functions development in all algorithms and programming models to ensure Region of Interest measures the same part of the algorithm in all benchmarks. 23

3.3 The speedup of the DF-Thread in the multi-node experiment. 33

3.4 The speedup of the DF-Thread in the multi-core experiment. 34

3.5 The RFIB multi-node experiment within DF-Thread and MPI. The baseline is MPI and we show how DF-Thread is comparable within the well-known programming model for variety of RFIB indexes. 36

3.6 The Execution time comparison of DF-Threads against Cilk with the RFIB experiment on 8 cores experiment. 37

3.7 The scalability of DF-Threads against Cilk with the RFIB experiment on 8 cores experiment. 37

3.8 The RFIB multi-node experiment within DF-Thread and MPICH. Here we show how much is the Speedup of DF-Thread compared to MPI execution model. 38

3.9 The Blocked Matrix Multiplication algorithm sketch that is used in our benchmark and its execution model. 38
3.10 A sample structure of the hardware and how MPI processes are distributed throughout the hardware resources. 39
3.11 Scheme of the Blocked Cholesky and how kernels are distributed throughout the hardware resources. 44

4.1 Simplified representation of the DF-Threads execution model. On the left, we represent the irregular read and write of generic threads. On the right, the exchange of data among threads happens in a more regular fashion [88]. 54
4.2 Illustrating the operations of the basic DRT API functions with a simple Recursive Fibonacci (RFIB) example. On the left, there is the representation of the RFIB function and its coding in DF-Thread style. On the right, we detail the specific dynamic behavior. Example rearranged from [104]. 56
4.3 The role of DRT in developing applications based on the DataFlow Threads (DFT) execution model. In the top part, we show the current setup of DRT. In the bottom part, we show the production framework that we envision. The idea is that DRT could help develop a future DRT backend of a standard compiler. 56
4.4 DRT sample output. DRT_DEBUG is an environment variable for specifying the debug level. The DF-Threads functions are mapped to internal operations where TS stands for thread scheduling, TE stands for thread-end, TD stands for thread drop, TW stands for thread write, *ip* stands for instruction pointer, and *fp* stands for frame pointer. Other debugging information is *fi* for frame index, *sc* stands for synchronization count, *ipnew/fpnew* are the *ip/fp* just freed. 58
4.5 An example of a modeled function in the DRT implementation, where METADATA extracts the metadata pointer from the frame, MDSC is the offset of the synchronization count, and MDQSTATUS is the offset of the status bits that indicate whether the frame is in ready or waiting status. 58
4.6 RFIB execution time speedup comparison between DRT, DARTS and OCR runtime. Here OCR is the baseline. DRT reaches better performance due to a simplified management of the dataflow execution. 60
4.7 Blocked Matrix Multiplication execution time speedup comparison between DRT and DARTS and OCR, with the DARTS as baseline. While for larger Matrix sizes the execution time tends to be the same for three tools, it is important to note that during the development-cycle, we typically use smaller inputs. So, the shorter execution time of DRT during tests helps focus on the development. 61
4.8 Simulation time speedup comparison between DRT and the COTSon simulator by using the RFIB example. DRT significantly decreases the development-cycle time to develop a dataflow program. 62

5.1 Graph partitioning scheme used in our system preprocessing method. Figure (a) shows a sample graph, Figure (b) and Figure (c) show the partitioned edge into 2 chunks which inside each chunk there are 2 blocks therefore in total we have 4 blocks of edge which is shown in each block. 69

5.2 The Hadoop framework for distributed graph processing high-level overview. 71
5.3 The Alveo card harwdare accelerator hardware and its compilation framework. 72
5.4 The Host software is responsible to drive kernel and dispatch data between CPU and Hardware accelerator. 73
5.5 The hardware implementation high-level overview. 77
5.6 The speedup gained while using multiple kernel instances against running the application using sequential kernel. 78

6.1 Twiddle factor position for a length-2 DFT. 84
6.2 Relationship among twiddle factors: $\widetilde{W}$ means a rotation of $-\pi/2$, whilst $\overline{W}$ means the conjugate. 85
6.3 Eliminating Complex Multiplication based on our proposed method. 86
6.4 8-point butterfly FFT. 87
6.5 Data-Flow Program Graph (DPG) for the first two steps. 87
6.6 DPG after the third step. 89

7.1 The cluster of FPGAs using Gluon boards. The network is capable of creating different network topologies such as Mesh, Star, 2D-Torus etc. 92
7.2 Gluon board block diagram and throughput to test the functionality of the board. 92

A.1 Results of profiling BMM with MPI shows an example of the MPI overhead over 15 cores, it can be seen from the results that most of the workers have unbalanced MPI overhead across available resources, and in this case not having a ideal speedup. (Experiment has been done on TFX2) 116
A.2 Results obtained from the BMM with OpenMPI with different Matrix sizes. It turns out while the second thread of the core is calling, the performance decreases due to the shared-memory resources that Pure OpenMPI implementation can not manage it and moreover kernel scheduler is not optimized for this execution model, However on logical cores the trend is almost linear. . . . 119

List of Tables

2.1 Most influential and well-known dataflow architectures proposed in last four decades. 8

3.1 The status of the benchmarks that have been developed for the baseline study, software simulation and hardware implementation, N stands for Node and C stands for Cores. 24

3.2 The hardware specification of platforms used in this experiment. 24

3.3 The configurations of the experiment that has been done on **Lab146**. 26

3.4 The configurations of the experiment that has been done on **TFX2** a Single Multi-core Machine. 26

3.5 The configurations of the experiment that has been done on **TFX3** a Single Multi-core Machine. 26

3.6 The sequential version of Recursive Fibonacci on three platforms TFX2, Lab146 and COTSon simulator. 30

3.7 The Recursive Fibonacci on platform TFX2 with Cilk programming model. . . 31

3.8 The Recursive Fibonacci on platform TFX2 with MPICH programming model. 31

3.9 The Multi-core experiment execution time for some selected indexes of RFIB on TFX2 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms). 35

3.10 The Multi-node experiment execution time for some selected indexes of RFIB on LAB146 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms). Each node in this experiment has 1 Core. 35

3.11 The sequential version of Blocked Matrix Multiplication on three platforms TFX2, Lab146 and COTSon simulator. 39

3.12 The Blocked Matrix Multiplication experiment on platform TFX2 with Cilk programming model, here we consider BLOCK size equal to 8 to set the granularity. 40

3.13 The Multi-core experiment execution time for some selected indexes of BMM on TFX2 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms) and the BLOCKSZ is equal to 8. 43

3.14 The Multi-node experiment execution time for some selected indexes of BMM on TFX2 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms). Each node in this experiment has 1 Core and the BLOCKSZ is equal to 8. 43
3.15 The sequential version of Blocked Cholesky on three platforms TFX2, Lab146 and COTSon simulator. 45
3.16 The Blocked Cholesky experiment on platform TFX2 with Cilk programming model, here we consider BLOCK size equal to 4 to set the granularity. 45
3.17 The Blocked Cholesky experiment on platform TFX2 with MPI programming model, here we consider tile size 4 to set the granularity. 48
3.18 The sequential version of Histogram on three platforms TFX2, Lab146 and COTSon simulator. We consider "bin" size equal to 4 for this measurement. . . 49
3.19 The Histogram experiment on platform TFX2 with Cilk programming model, here we consider "bin"" size equal to 4. 49
3.20 The Histogram experiment on platform TFX2 with MPI programming model, here we consider "bin"" size equal to 4. 49

4.1 DF-Threads function definitions [65] 55
4.2 The function name and its corresponding frame pointer address that are shown in Fig.4.4 (same as in objdump tool). 59
4.3 Comparing lightweight DRT with other tools for developing dataflow codes and the related architectures. As we can see DRT, is using only 300 lines of C code. 61

5.1 The cloud service cost based on Amazon cloud cost calculator for FPGA f1 instance cloud servers. 65
5.2 The cloud service cost based on Amazon cloud cost calculator for CPU instance cloud servers. 65
5.3 Brief overview of the most related recent studies on FPGA accelerators and their features compared to this work. 68
5.4 Most recent and well-known graph partitioning suitable for FPGA implementation. 70
5.5 The XACC xilinx server used to evaluate the real implementation. 78
5.6 The alveo U250 resource utilization in this experiment. 78
5.7 The performance model report, which has been calculated based on the bottlenecks like PCIe Rate, Computation time, Communication time, etc. Here the number of SLR regions used is equal to 1. 79
5.8 The datasets for evaluating our proposed study. We choose them based on the size and the structure of the datasets to be comparable with other works. 79
5.9 The evaluation of the hardware implementation of the GridGraph algorithm on CPU and FPGA platform. 80

6.1 Summarized number of different operations in DPG 88
6.2 Experiment Result for our proposed FFT Algorithm 89

A.1 The configurations of the experiment that has been done on **TFX3** a Single Multi-core Machine. 118

List of Listings

3.1 Recursive Fibonacci function code . . . 29
3.2 Recursive Fibonacci function code using Cilk programming model. . . . 30
3.3 The worker function of the Recursive Fibonacci function code using MPI programming model. . . . 32
3.4 The root function of the Recursive Fibonacci function code using MPI programming model. . . . 32
3.5 BMM using cilk programming model function code . . . 40
3.6 BMM using MPI programming model function code . . . 42
3.7 Blocked Cholesky sequential version function code . . . 46
3.8 Blocked Cholesky using Cilk programming model function code . . . 47
3.9 Histogram sequential version function code . . . 48
3.10 Histogram with Cilk programming model function code . . . 50
3.11 Histogram with MPI programming model function code . . . 51

5.1 The function declaration of creating aligned vectors. . . . 73
5.2 The OpenCL commands used to create buffers between host and kernel. . . . 74
5.3 The OpenCL commands to run the kernel using appropirate arguments and pointing to created buffers in Listing 5.2. . . . 75
5.4 The kernel function declaration. . . . 77
5.5 The multi kernel configuration that enables running multiple instances of the kernel in parallel from the host program. . . . 77

A.1 How to compile and set environment variables for mpiP profiling tool . . . 113
A.2 The command specifications to specify the region of interest to profile with mpiP 114
A.3 The output of profiling the OpenMPI benchmark with mpiP from LLNL repository. The results show the MPI overhead and UsrTime compared to the overall execution time. . . . 114
A.4 The aggregate collective report from mpiP profiling, matrix size 2000 and number of workers 15 . . . 115
A.5 The mpiP output log and how to calculate manually the runtime, this runtime time is not a part of the mpiP output log. . . . 117
A.6 The script to iterate the experiment of a loop and collect the numbers, producing a suitable csv file that can be used for reports . . . 118
A.7 The script to fetch numbers from a generated csv file and plot it using GNUPLOT 120

A.8 The GNUPLOT script uses to create graphs using the csv file output. The user should take care of the name of the csv file and other decorations such as title, fonts, etc. 121

List of Abbreviations

AI	Artificial Intelligence.
API	Application Programming Interface.
ASIC	Application Specific Integrated Circuit.
AXI	Advanced eXtendable Interface.
AXIOM	Agile eXtensible Input Output Module.
BMM	Blocked Matrix Multiplication.
CPU	Central Processing Unit.
DSP	Digital Signal Processor.
DSM	Distributed Shared Memory.
DSE	Design Space Exploration.
DFG	Dataflow Graph.
DF-Threads	Dataflow-Threads.
DLP	Data Level Parallelsim.
HBM	High Bandwidth Memory.
HLS	High Level Synthesis.
ILP	Instruction Level Parallelism.
ISA	Instruction Set Architecture.
IP	Intellectual Property.
GPU	Graphic Processing Unit.
GCC	Gnu Compiler Collection.
FPGA	Field Programmable Gate Array.
FP	Frame Pointer.
FIFO	First-In, First-Out.
FFT	Fast Fourier Transform.

LUT	Look-up Table.
NIC	Network Interface Card.
OS	Operating System.
PL	Programmable Logic.
PS	Processing System.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computer.
RDMA	Remote Direct Memory Access.
RFIB	Recursive Fibonacci.
ROI	Region of Interest.
SC	Synchronization Count.
SSSP	Single Source Shortest Path.
SoC	System on Chip.
SpMV	Sparse Matrix-Vector Multiplication.
SPI	Serial Peripheral Interface.
TLP	Thread Level Parallelism.
WCC	Weakly Connected Components.

Abstract

This thesis mainly focuses on state-of-the-art challenges of distributed execution models and research the system support for artificial intelligence and high performance computing applications. In this context, we focus on investigating in detail about co-designing the Dataflow-Threads execution model [61]. Moreover, to facilitate support, development, and debug the Dataflow-Threads execution model, we introduced DRT; a lightweight Dataflow runtime [68]. DRT has been written in portable C code (tested with the GNU C compiler), and it is open-source. It can be used on real machines based on architectures like x86, AArch, RISC-V ISA.

Furthermore, we consider major problematic applications in the domain of the Artificial Intelligence (AI) and High Performance Computing (HPC) and address the main challenges and bottlenecks to extend our dataflow runtime. To do this, we used widely known benchmarks to stress the capabilities of the DF-Threads execution model and its evaluation against other parallel programming models. We choose Blocked Matrix Multiplication and Recursive Fibonacci. Matrix multiplication is one of the main kernels of AI and HPC Applications. Plus, Recursive Fibonacci is a simple benchmark which creates enormous number of threads and processes and stress the entire execution model.

In this thesis, we are mainly interested in heterogeneous platforms. A heterogeneous platform is a hardware device that contains a range of computing components, such as multicore CPUs, GPU, or FPGAs. Their capabilities have provided many features for researchers to use this kind of structure in their state-of-the-art works. Heterogeneous systems are flexible, cost-efficient, and well-supported by communities. Our work focuses mainly on CPU+FPGA Heterogeneous systems, mostly a general-purpose CPU (x86 or ARM) within a Unix-based operating system besides an FPGA accelerator. Subsequently, because of a need in our hardware platform structure, we design and fabricate the Gluon board, which uses serial transceivers in Xilinx Ultrascale+ Heterogeneous accelerator and facilitates GTH transceivers in high rate data transfer applications. Gluon boards are modular and can carry up to 18 Gbps on each lane with specific data types and payload sizes. The end-user cost to manufacture the Gluon board is less than 400 euros with enormous capabilities.

Moreover, a real application demonstrates a distributed graph processing application to express the distributed computing execution model and further extend our execution model to cover the real-world application like Graph Processing in large scale. In the first step, we provided a comprehensive baseline, designed and proposed a large scale distributed graph processing

application and evaluated it within the PageRank algorithm using well-known datasets. We show how graph partitioning combined with a multi-FPGA architecture leads to higher performance without limitation on the size of the graph, even when the graph has trillions of vertices. Our performance analysis, in the case of PageRank, forecasts performance improvement of up to 20 times and a cost-normalized improvement of up to 12 times when comparing the proposed approach on one Xilinx Alveo U250 FPGA accelerator against a state-of-the-art baseline graph processing software implementation on a Intel Xeon server CPU with a 40-core processor at 2.50 GHz.

1 Introduction

> *What is essential is invisible to the eye.*
>
> — **Antoine de Saint-Exupery**
> (Writer, 1900-1944)

Reconfigurable computing is an emerging field that has rapidly grown during recent years, and numerous research areas increasingly deal with reconfigurable hardware and their architecture to implement computations. From wearable gadgets to datacenters, reconfigurable hardware is employed on a large scale and the trend of using reconfigurable hardware grows everyday in all aspects. Different technology of reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), and Graphics Processing Units (GPUs) could successfully mitigate the bottlenecks of the ASIC hardware while they had the majority of the market before. Basically, ASIC hardware accelerators were effective, however, they are fraught with challenges. The major problem with ASIC is that as algorithms change rapidly, hardware must be re-designed and re-verified, which is costly in development time and time-to-market. Consequently, algorithm innovation becomes more difficult without access to a flexible hardware. Furthermore, fixed accelerators cannot be shared across applications, making them more costly in the fabrication process. Ideally, a hardware that is capable of executing compute-intensive algorithms at high performance with much lower power than programmable architectures is required while remaining broadly applicable and adaptable. This work, in collaboration with my research group, mostly focus on FPGA as a suitable hardware to adapt and re-configure effectively our new concepts and build the new prototype. FPGA has several advantages to DSP and GPU for instance FPGA has more power efficiency than GPU [161]. In addition to power consumption, FPGA can cover a broad range of applications while GPU is mostly used for vector processing and even though the development time with GPU is less than FPGA. GPU is more power hungry and likely is avoided in datacenters and compute intensive applications when power and money has the crucial role, whereas, FPGA offer some distinct advantages like low latency when it comes to the networking applications. Finally, The most important advantage of FPGA regards irregular algorithms like Graph Processing is that FPGA has sequential data access and can mitigate the irregular access to the memory whilst GPU and CPU have conventional random memory access architecture. We will discuss this more in details later in the next chapters.

1.1 Motivation and contribution

Our group Ph.D. research line expresses the state-of-the-art challenges of distributed execution models and research the system support for high performance computing applications. In this context, our research group focus is on Dataflow-Threads and its execution model [67].

The contributions of this thesis are as follows:

1) Providing comprehensive literature review to understand how Dataflow-Threads can be distinguished from other recent studies.

2) Providing a solid baseline and benchmark suite to compare our Dataflow-Threads execution model with other parallel programming models, such as OpenMPI and Cilk. A unique algorithm for all mentioned programming models is developed and unique methodology is performed to collect the results. Moreover, this work investigates in detail to show which bottleneck has been faced within each experiment.

3) Providing a case study of the Distributed Graph processing on the Multi-FPGA platform. This is a collaboration between two research groups to exploit the potential of distributed reconfigurable platforms within the real-world application. This work mainly studied PageRank and compared the performance model within actual implementation and baseline studies.

1.2 Thesis Structure

The structure of this thesis is as follows:

Chapter 1

This chapter is the introductory discussion about the Program Execution Models (PXM), and more specifically, I discuss Dataflow Execution Models and the recent works in the literature. I briefly point to the most important research and their contributions by their category. This chapter introduces the different categories of Dataflow Abstraction and their Execution Models. Then, a review of the most challenging and significant works to mitigate any failure or bottleneck within this Abstraction Model will be addresses. This study shortly introduce previous works in our laboratory, which have been done previously and present their contributions briefly.

The aim of the chapter is to answer the following question:

Q. ***There have been many studies and investigations in Dataflow architectures since early 70's, The question is what is the status of the Dataflow Execution Models nowadays? How much are they successful to surpassing conventional methods?***

Chapter 2

This chapter reviews the most recent literature studies based on Dataflow Execution Models. Moreover, I concisely point to some recent studies more similar to our work. In this chapter, I differentiate our work from other studies, and I demonstrate the variety of the contributions and their impact on the Dataflow Computing subject, and finally show a landscape of state-of-art implementations focusing on Dataflow Execution Models.

The aim of the chapter is to answer the following question:

Q. ***This thesis focuses on Hybrid Dataflow Execution Model, The question is based on the literature, what are challenges, bottlenecks and approaches recent studies have addressed? and finally, what is the main features of a fine-grained dataflow execution model?***

In particular, I discuss the selected studies implementation and their issues and I compare them with ours and other related ones.

Chapter 3

This chapter discusses the need for a strong baseline for our research line. Many studies in the literature discuss the bottlenecks and propose new methods in parallel programming. However, a strong baseline in which all the aspects that have been studied carefully is missing. In particular, parallel computing is a challenging concept, and many methodologies are proposed to program, calculate and measure the metrics for the specific computation. This work investigates several essential benchmarks in parallel programming and shows essential aspects of a baseline and measurements. This work compares well-known parallel programming models like OpenMPI, OpenMP and Cilk and discuss the lessons learned during the baseline experiments.

The aim of the chapter is to answer the following question:

Q. ***The question is based on the proposed methodology, how we can have a strong baseline to evaluate our execution model and in particular our Dataflow Thread execution model?***

Chapter 4

This chapter, introduces the Dataflow Thread Runtime (DRT). DRT is a lightweight runtime to develop and debug dataflow based examples to be targeted by a future compiler for the dataflow programs. The key point of DRT is that most of the dataflow runtimes do not provide test and debug feature for developers to follow the execution model of the dataflow program. DRT enables such feature for developers and has potential to be a backend of the compiler in our workflow.

The aim of the chapter is to answer the following question:

Q. ***The question is as we investigated in the literature, how we can have a lightweight runtime to test and debug Dataflow Thread programs? DRT is the solution to answer this need.***

Chapter 5

In this chapter with the collaboration of my reseach group express a *Big Scale Distributed Graph Processing* as a case study to fullfill the ”borse Pegaso ciclo 34” obligatory activity abroad the country. The materials in this chapter are a collaboration between the University of Siena (Smart Computing Program) and ”Custom Computing Research Group at Imperial College London”. In this period, I defined a project based on the a Reconfigurable Graph Processing Model that addresses the current challenges and needs to cover both groups Ph.D. goals. This study can bring possibilities to the user and have many advantages despite a few current ones in the literature and I am progressing with the implementation. The aim of the chapter is to answer the following question:

Q. ***The question is can the distributed graph processing on multi-FPGA platform be a good solution for large scake graph processing and can be effectively compared with CPU based graph processing solutions?***

Chapter 6

At the beginning of my Ph.D. studies, I had this chance to collaborate with Dr. Lorenzo Verdoscia at the Institute for High Performance Computing and Networking (CNR). Our collaborating concluded with an Dataflow Approach to accelerating FFT application and we got the **Best Paper Award** of the 8th Mediterranean Conference on Embedded Computing (MECO) in Budva, Montenegro, 2019.

Q. ***The question is can we provide a fast and reliable methodology based on Dataflow concepts to process FFT as a widely used kernel in many applications?***

Chapter 7

This chapter presents *Individual Research Activity* that have been carried out during the Ph.D. program. The studies presented in this chapter are collaboration between myself with other colleagues. I present the Gluon board in this chapter. GLUON board is the modified and enhanced version of the TEBT0808 board from Trenz Company [69], which with the new design is able to power up the FPGA module with unix-based operating system. GLUON enables serial transceivers in Xilinx Ultra-scale+ structure and facilitates using GTH transceivers in high rate data transfer applications.

Chapter 8

This chapter concludes the thesis. In this chapter we summarize the most important findings from this work, the main achievements and what has been undergoing will be pointed in three years of Ph.D. program. Of course the time limitation does not allow to dig into all aspect of the theoretical and implementation of the study. I discuss in this chapter what is left to be completed and what is the overseen studies to addresses in the future works.

2 Related works

> *The more you know, the more you realize you know nothing.*
>
> **— Socrates, Philosopher, 400 BC**

This chapter contains two major parts. In the first part, I introduce Dataflow Architecture and its Execution model. I shortly introduce our concept of Dataflow-Threads and its potential to be deployed on reconfigurable architecture. To this end, we demonstrate our concept, the challenges and bottlenecks, and the achievements so far. The achievements and improvements of our research line in the recent years and the similar state-of-the-art studies will be addressed.

The dataflow model represents a revolutionary alternative to the control flow (also known as the von Neumann) model since the execution is driven only by the availability of operands. A pure dataflow execution model has no program counter (PC) and global memory, the two major elements of the von Neumann model that become bottlenecks of its performance [138]. In dataflow computing, only limitation of parallelism is the data dependences between instructions in the application program [152]. However, it is well investigated that data driven feature of dataflow computers have the potential for exploiting all the parallelism available in a program [44, 43, 10, 12, 11].

In conventional von Neumann machines, due to the dependence of the commands from the PC, which also fetches load and write commands, there may be delays in the entire program's execution, whereas, in a dataflow machine, the dependencies and instructions are inside the processor already. The procedure to see all these dependencies is called *Dataflow Graph* and dataflow machines use dataflow graphs as their machine language. A dataflow graph (DFG) comprises arcs and nodes, with the nodes representing locations where variables are assigned or used and the arcs representing the link between the places where a variable is allocated and the places where the assigned value is used later. Based on the graph, the independent instructions can now be executed first, followed by instructions whose operands are available later. Unlike conventional machine languages, dataflow graphs specify only a partial order for the instruction execution and thus provide opportunities for parallel and pipelined execution at the Instruction level.

In this chapter, I discuss the related studies in terms of expressing most influential works and their approaches to mitigate the deficiency and enhance the strength of dataflow execution models. There are several category field of studies that I can discuss separately towards them in this chapter. The categories are described as follows:

1) Dataflow Architecture Discussion

2) Fine-grained Parallelism Approaches in Dataflow Architecture
3) Dataflow Runtime Libraries
4) Dataflow Execution Models using Hardware Accelerator

2.1 Dataflow architecture

So far we had a brief understanding of what is a dataflow architecture. There have been several efforts to build a dataflow hardware. In Fig. 2.1, the dataflow inspired models that have been studied or even built, and used for several years has been depicted. These works that are mentioned here are the most well-known studies, some works have been ommitted for the sake of readability of the picture. Our work is based on Teraflux [67] and DF-Threads execution model on Heterogeneous platform like AXIOM [63] that can be seen in the Fig. 2.1.

Beside these mainline researches, there have been many studies about dataflow architecture with different research subjects. Some earlier studies try to introduce and build the dataflow hardware [117, 76], some other ones try to introduce the dataflow languages and their compiler [168], and some other research are based on developing dataflow execution models and their runtime libraries [157, 61, 5, 60, 102, 50]. An increasing number of studies have introduced their application based study based on dataflow computing. These works cover a wide range of domains such as power efficiency [174, 70, 22], high performance computing [52, 55, 133], computing scientific algorithms [159, 158], artificial intelligence [169, 101] and accelerators for general purpose applications [163].

2.2 Taxonomy of dataflow execution models

What was mentioned already was a brief discussion about the dataflow architecture and the studies since 1970. This section presents a taxonomy of dataflow architecture, programming model, and execution models. There are several efforts in the literature to provide a comprehensive study about dataflow [173, 152, 155, 21, 97, 138]. However, several items are missing in these works, which are as follows. **First**, they did not cover the recent studies and most covered works have been done between 70's and 90's. **Second**, it is not obvious where and how dataflow is going forward and how dataflow can be used in the next generation of computers.

Table 2.1 shows a survey of dataflow architecture based on the state-of-the-art. In this Table, we show the most influential works and thier taxonomies and importants feature to point out. In this table, the level of parallelism categorizes in three levels, **ILP** (*Instruction Level Parallelism*), **DLP** (*Data Level Parallelism*), and **TLP** (*Task level Parallelism*).

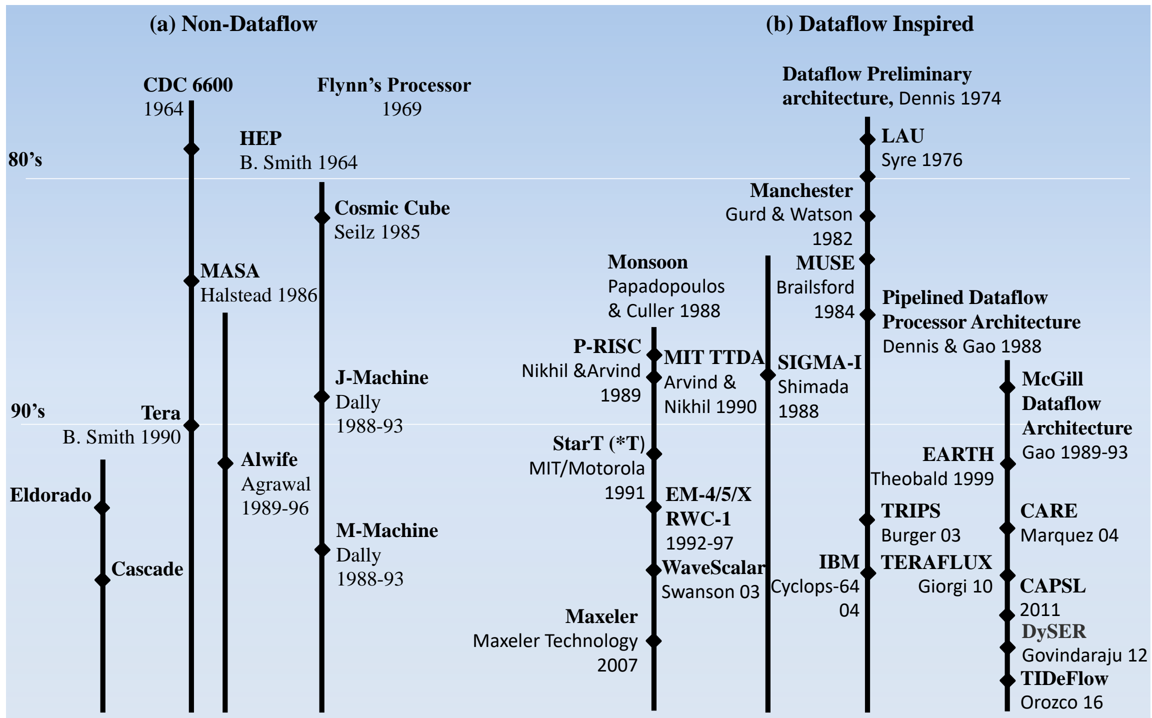


Fig. 2.1.: The most influential dataflow inspired and non-dataflow architectures and their timeline.

I aim to find the most bold features and the roadmap of these works. Moreover, I would like to show what is the possible future of the dataflow architecture. More specifically, what is the potential and where does the dataflow architecture go in the next years.

Static Dataflow

Dennis and Misunas [44] proposed the Static Dataflow model of computation to design, evaluate, and implement computations that work on endless streams of data. A static dataflow model is a directed graph of computational actors with FIFO channels connecting them. The amount of tokens consumed and created by an actor must be fixed and pre-specified in the static dataflow semantics. This ensures that crucial model features such as deadlock-free and memory-bounded infinite computation, throughput, latency, and execution schedule can be determined. There are other several efforts based on static dataflow model such as HDFM [154], NEC [146], DDM1 [41]. Although static dataflow was a basis of many efforts on 70's till 90's, however, there were several fundamental issues with this model. We briefly discuss some issues regarding the static dataflow model,

1) There was a mismatch between the model and the implementation, The model requires **unbounded FIFO** token queues per arc (which represent data dependences among instructions) but the architecture provides **storage** for one token per arc.

2) The architecture does not ensure FIFO order in the reuse of an operand slot.

3) The static model does not support the **function calls** and **loops**

4) No **Data Structures** is supported in this model. So it was hard to program and exploit the data locality.

Tab. 2.1.: Most influential and well-known dataflow architectures proposed in last four decades.

Dataflow Architecture	Key Concept	Key Features	Level of Parallelism	Studies References	Presenting Year
Static Dataflow	In static dataflow, a program is described as a set of operator nodes, called actors, interconnected by a set of data-carrying arcs, called links. Data is passed through this graph in one and only one packet called tokens.	1) Too much parallelism. 2) Acknowlegment is needed for each token and makes the tokens traffic double 3) Asynchronous execution. 4) Order of instruction execution is unpredictable, thus the execution in uncertain. 5) Loops as a major bottleneck of many programs can only be parallelized in sequential manner.	ILP	[44, 154, 146, 41, 10, 80, 19]	1974-1987
Dynamic Dataflow	In contrast with statis dataflow model, In dynamic dataflow concept, the number of tokens that can be carry data in dataflow graph is unlimited.	1) Loops can be parallelized, each iteration as independent subgraph in the whole dataflow graph model. 2) Suitable for stream computing models. 3) Introduce the data structure to dataflow architecture. 4) Not efficient in terms of memory, since there is a huge need to store abundant number of tokens.	TLP	[11, 138, 156, 124]	1987-1993
Hybrid Dataflow	Introduced in the late 80's and significant studies have been done quickly for almost two decades. There are two types of Hybrid model. Dataflow/Control flow and Control flow/Dataflow. The former got most attention and has the benefit of scheduling based on dataflow semantic, whilst each basic block inside executes sequential control flow computing.	1) Combine the dataflow and von-Neumann models of computation can mitigate the deficiency of both models. 2) Control the granularity while being substantially power efficient. 3) More program developing friendly than others.	TLP, ILP	[149, 126, 19, 164, 96, 67]	1982-2004
Threaded Dataflow	This model is also known as "Data-Driven Multithreading". The key factor of this model is to introduce the Task Synchronization Unit (TSU) to the dataflow semantics.	1) A "thread of instructions" is issued consecutively by the matching unit without matching further tokens except for the first instruction of the thread. 2) Data passed between instructions of the same thread is stored in registers instead of written back to memory. 3) The cycle-by-cycle consecutive instruction interleaving of threads is same as von-Neumann control flow execution model.	TLP, ILP	[72, 151, 117, 135, 140, 36, 129, 79]	1988-1996
Spatial Dataflow	In a producer/consumer fashion, spatial dataflow maps directly a Dataflow Graph (DFG) on resources available on a hardware (PEs). Therefore, provide the parallelism to compute program on a core in space not in time.	1) There is no need for instructions because the hardware itself represents the computation. 2) There is also no need for memory decode logic, branch prediction or out of order scheduling. 3) Ease of programming development. 4) Suitable for many cases specifically streaming application.	TLP	[105, 29, 28]	2006-2015
Stream Dataflow	The concept is to set the operations (kernel functions) that applied to each element in a stream given a sequence of data (a stream). Stream dataflow take advantage of integrating stream programming language with dataflow execution model concepts.	1) Is based on streaming memory access and reuse patterns 2) High concurrancy beside low power consumption and low overhead 3) less available hardware built so far.	ILP	[110, 165, 120, 89]	2015-2019

5) There are **too much parallelism** generated by the architecture but there is no way to schedule them effecively to leverage this level of parallelism.

These limitations manke researchers to investigate more about the fundamental problems regard static dataflow. In TTDA [10] and HPS [80] in the 80's, researchers tried to mitigate the static dataflow issues by adding more synchronization and resource managing ability to the tokens, while preserving sequential ISA semantics. These efforts brought the idea of starting another branch of the dataflow architecture which represented as *Dynamic Dataflow*.

Dynamic Dataflow

Once loop iterations and subprogram invocations could run in parallel, a dataflow machine's speed improves dramatically. To achieve this, each loop iteration or subprogram invocation should be executed as a separate instance of a re-entrant subgraph. This replication, on the other hand, is simply theoretical. Only one copy of any dataflow graph is actually kept in memory in a real implementation. Each token has a tag consisting of the address of the instruction for which the particular data value is destined and other information defining the computational context in which that data is to be used. This concept was led to build the hardwares based on Dynamic Dataflow concepts such as MIT Tagged-Token Dataflow Architecture [11] and Monsoon [117].

Spatial Dataflow

Spatial architecture is a type of accelerator that uses direct connection between a number of very basic processing engines (PEs) to utilise significant computational parallelism . Different algorithms can be built or coded into them, which are then mapped onto the PEs utilising specific dataflows. Spatial architectures, as opposed to SIMD/SIMT designs, are better suited to applications with producer-consumer interactions or that may benefit from efficient data exchange throughout an area of PEs. One example of such design is Maxeler Dataflow Engines [105]. Data is streamed from memory onto a chip on a dataflow engine (DFE), where it goes directly from one functional unit to another without being written to off-chip memory until the entire operation is completed. A control flow core executes operations at different points in time on the same functional unit "computing in time", whereas a dataflow core does computation on a chip spatially "computing in space". Instructions are not required in a DFE because the DFE itself represents the computation. As a result, no memory decode logic, branch prediction, or out of order scheduling is required, allowing the chip to devote all of its resources to computation [118].

Another work that leverage spatial dataflow is Eyeriss [31]. Eyeriss is made up of a collection of processing elements (PEs), each of which contains logic to compute multiply-and-accumulate (MAC) and local scratch pad (SPad) memory to take advantage of data reuse, as well as global buffers (GLBs), which provide an extra level of memory hierarchy between the PEs and the off-chip DRAM. There are two version of Eyeriss project. One major difference between Eyeriss v1 and Eyeriss v2 is that the latter uses two-level memory hierarchy [29].

Hybrid Dataflow

There have been two basic models in computer architectures: (**1**) the ***von Neumann sequential*** control model; and (**2**) the ***dataflow data-driven*** computing model. The parallel architectures based on the von Neumann model aim to exploit coarse-grain parallelism, while the traditional dataflow architecture model was conceptualized to handle fine-grain parallelism. For the past years, researchers have debated which model is a "more efficient" basis for future large-scale parallel computer systems [58].

One of the very first hardwares based on hybrid dataflow was MDFA [57]. In MDFA work clearly the concept of integrating von Neumann model of computing with dataflow computing model is implemented and discussed. At that moment, there were other proposed methods called "macro-dataflow", which was different from the concept of nowadays so called "hybrid dataflow".

There is not a clear discussion that when and how the integration of "von Neumann sequential" execution model plus "dataflow semantics" named hybrid dataflow but there are many works that discussed and studied this concept and produced many great works in this context such as [67, 88, 102, 117, 111]. In these mentioned work, explicitly the potential of integrating control flow computing model plus dataflow execution model is discussed and hardware prototypes have been created moslty using reconfigurable devices such as FPGA. There are imbiguity of using integration of control flow model with dataflow in hybrid dataflow computing, we must point out that, in particular we are focusing on **Dataflow-> Control flow** execution model. This concept employs basic blocks of containing control flow program, scheduled based on the dataflow execution paradigm. The basic block is a set of sequential instructions, where data is passed between instructions using register or memory as a usual conventional method to execute control flow programs.

Threaded Dataflow

In some previous studies, Hybrid Dataflow and Threaded Dataflow are categorized in one seed. However, they are slightly different. Threaded Dataflow is a dataflow modification approach in which instructions from specific *instruction streams* are executed in sequential machine cycles. Whereas in Hybrid Dataflow this means *threaded basic blocks* are being executed in a dataflow way). EM-4 [135] and its updated version EM-X [90], and Monsoon [117] are well-known successful projects based on Threaded Dataflow. The implementation of an effective *synchronisation mechanism* is the key design problem in all threaded dataflow machines. Direct matching is a synchronisation mechanism that does not require the use of associative mechanisms [153]. In [145] authors introduced an implementation of threaded dataflow model on FPGA. This work is the development of a Thread Synchronization Unit (TSU) on FPGA, a hardware unit that enables thread execution on a chip multiprocessor utilising dataflow rules. Threads are executed depending on data availability, which means that a thread is launched only if its input data is available. This execution model is known as the non-blocking Data-Driven Multithreading model. Other efforts, such as [175], have exploited this approach to build an recursively organized data-driven machine, RWC-1 is capable of automatically and dynamically allocating concurrent tasks to the available hardware units.

Stream Dataflow

By controlling the number of parallel computations that may be done, the stream processing paradigm simplifies parallel software and hardware. A set of operations (kernel functions) are applied to each element in a stream given a sequence of data (a stream). Kernel functions are often pipelined, and best local on-chip memory reuse aims to minimize the bandwidth loss caused by external memory access. Stream Dataflow is a concept of combining the stream computing programming language and dataflow semantics first proposed in [89] later enhanced and extended in [110]. Although in the work[110], authors provide novel insights in Dataflow context, however, still the evaluation is based on Softbrain RISC-V based simulator and there is not yet a real hardware implementation. Another high influential work in this context is [78].

The Merits of Dataflow Architectures Dataflow architectures same as other architectures have some advantages and some deficiencies. We talked about the possible deficiencies of dataflow architecture separately in the dataflow introduction. Here we exhaustively mention the merits of dataflow architecture and its execution model.

1) The dataflow paradigm only enforces actual data dependencies, hence it exposes the greatest degree of parallelism in a program.

2) The dataflow enables asynchronous data-driven execution of finer-grained tasks, which has the ability to make better use of the underlying hardware. Additionally, finer-grained elements have a lower memory usage, reducing the amount of memory required.

3) The dataflow can effectively endure memory and synchronization latencies.

4) Since there are only actual data dependencies, the dataflow does not require power-hungry modules like out-of-order execution and may use non-coherent memory structures.

5) The dataflow concept is well-suited to application-specific streaming hardware.

6) Integrating Dataflow/Control execution model pardigms can open many possibilities to develop the variety of applications. It has the advantages of both and mitigates deficiency of them.

2.3 Fine-grained Parallelism Approaches in Dataflow Architecture

In Fine-grained execution model, a program is split down into a large number of small elements, to take advantage of better exploiting the underlying hardware to be more faster and more power efficient. Processors resources are allocated to these elements independently. A parallel work has a little amount of resources linked with it, and it is equally split

across the processors. As a result, fine-grained parallelism makes load balancing easier. The number of processors resources required to accomplish the processing is high since each element processes less data. As a result, the overhead of communication and synchronization increases. To accomplish this characteristics, there have been several studies and proposed methods in the literature with different approaches [184, 138, 12, 67].

Dataflow Threads exploits these characteristics of fine-grained threads to spread them efficiently across nodes/core in the multi-node/multi-core platform. Design mechanism of a co-processor design including a load balancer, the design properties of synchronization unit, Memory model and other implementation notes and feature are covered in [106].

To extend the role of DF-Thread and its execution model, I elaborate in investigating in the literature by distinguishing the **DF-Thread** versus **task**, **token**, **codelet** and **framelet**.

DF-Thread

DF-Threads [61] introduces a low-level API, which enables a high-level code into a hybrid dataflow model that can benefit from the high parallelism while parallel computations have the potential to distribute over nodes and cores.

codelet

A codelet is a collection of machine instructions which are scheduled "atomically" as a non-preemptive, single unit of computation. In the codelet PXM, Codelets are the principal scheduling quantum. Codelets are expected (not required) to behave functionally, consuming inputs, working locally, and producing output leaving (ideally) no state behind. A codelet will only fire when all of the resources it requires are available. The recent works which are Codelet-Based Implementations are: DARTS [183], SWARM [95], FreshBreeze [42], and OCR [113]. Fig. 2.2 shows a Codelet Graph (CDG) and its dependencies. Codelets are linked together to form a codelet graph. In a CDG, each codelet acts as a producer and/or consumer. An initial codelet may fire, producing a result which multiple codelets can consume, giving way for more codelets to execute. Since codelets are linked together based on data dependencies, a CDG may benefit from the same properties as a dataflow graph. As implemented in DARTS [183], Threaded Procedure scheduler (TP scheduler) is responsible for load balancing TP between clusters, instantiating codelets, and distributing codelets within a cluster [141].

framelet

First introduced in [129] and the definition is that, for each instance of a thread, a small fixed size storage area (called framelet) is allocated in the Frame Store (see Fig. 2.5), to hold the incoming inputs to that thread instance. A framelet is large enough to hold inputs for most threads. When a particular thread's requirement exceeds the size of a single framelet, one or more additional overflow framelets are allocated as needed. The idea of framelet was proposed to increase the locality of threads and cache optimization.

token

Based on static dataflow model definition in [56], a program is represented by a directed

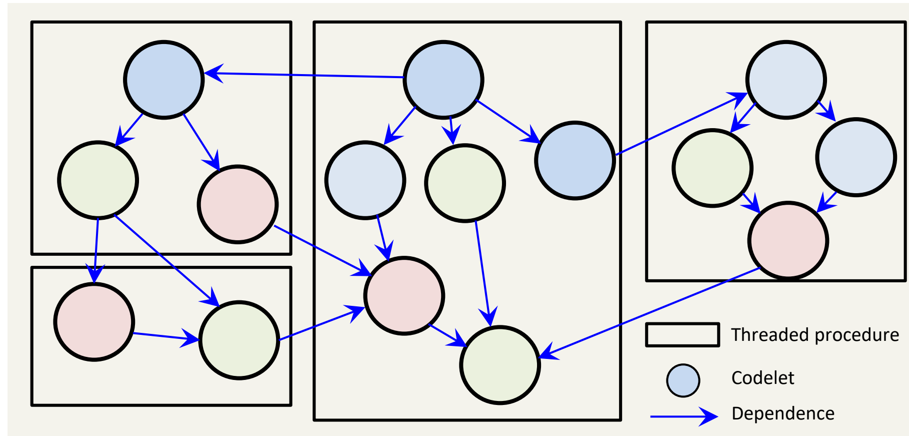


Fig. 2.2.: A codelet Graph (CDG) shows the codelets and their dependencies connected through a multiple threaded blocks linked together [141].

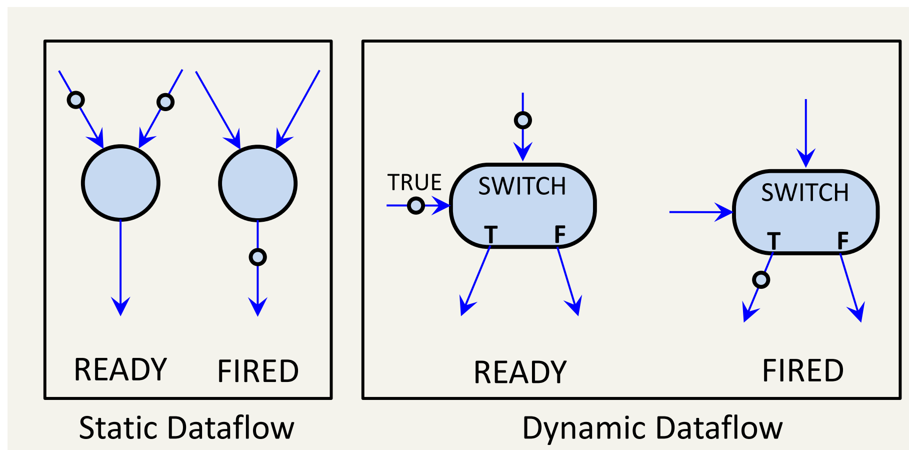


Fig. 2.3.: A token-arc representation of data and dependency in the static and dynamic dataflow.

graph. During the execution of the program, data propagate along the arcs in data packets, called tokens. Nodes in the graph are also called actors and arcs that transmit tokens which carry the values to be processed by these actors. Actors become activated for execution when tokens carrying operand values are at their input arcs [20]. A basic elements of dataflow graph, nodes, arc and token based on static dataflow models is illustrated in Fig. 2.3. Note that static dataflow allows only one token on one arc. This limitation mitigated later by proposing dynamic dataflow in which there is no limitation on number of token on each arc.

task

The dataflow task transfers data between sources and destinations while allowing the user to alter, clean, and edit data in the process. When a dataflow task is added to a package control flow, the package can extract, manipulate, and load data.

2.4 Dataflow Runtime Libraries

In recent years, there have been some works regarding Dataflow Runtime Libraries and their execution models that, in this section, I summarized and discussed them. In the following, we highlight some works that are related to our Dataflow Runtime (DRT), and I point out the differences. The discussion in details about Dataflow Threads Runtime (DRT) is appeared in Section 4.1.

BMDFM [119] Binary Modular DataFlow Machine, is a *Threaded Dataflow*, runtime environment for Shared Memory Processors (SMP) that provides a dataflow execution model with its extended instruction set. The contribution of this work is to define and design of the speculative tagging dynamic scheduling algorithm that is used to tag "ready instructions" for execution in the runtime dataflow engine. Moreover, this work provides the design of multi-threaded marshaled clustering (which have been prepared statically during the compilation stage) of the data loaded from the control virtual machine into the dataflow runtime. By the "virtual machine" it means an interface which provides the transparent dataflow semantics for conventional programming. BMDFM has been implemented on conventional multi-core platforms to show a complete parallelization environment. This work is categorized as "Threaded Dataflow" and is based on "tagged-token dataflow model".

FREDDO [103] uses the distribution of Data-Driven Multi-threads (DDM) over conventional multi-core processors. FREDDO is written in C++ and used the Object-Oriented programing features. This work is also categorized in "Threaded Dataflow" and based on a "tagged-token dataflow model" with a slight difference. The authors recommended using a "Data-Driven Multi-threading Thread (DThread)" and a Thread Scheduling Unit (TSU), which is a specific module that is responsible for scheduling of DThreads. In practice, a program in FREDDO is made up of DThreads code, Thread Templates, and the Dependency Graph. A DThread's information is stored in a Thread Template. The Dependency Graph depicts the DThreads' consumer-producer relationships. Before a DThread is scheduled for execution, the TSU unit in FREDOO verifies that the data required by it is available. In this work, the evaluation has been done via a complete benchmark suite using 10 application.

Sucuri [139] is a minimalistic Python dataflow library to execute Dataflow Graphs (DFGs) over a multi-core distributed platform. Sucuri is based on a centralized and local scheduler in each node that can execute the ready tasks in their local queues. The compiler partitions the DFG, then, during the runtime, each related DFG part will be distributed among the associated node.

Swift/T [168] is a new implementation of the swift language [167] that provides high-level programmability for implicit dataflow programming. It addresses some optimization for the Swift parallel scripting language, along with Turbine compiler, which C/C++/Fortran programmers can develop their software based on this platform.

SWARM (SWift Adaptive Runtime Machine) [95] is a software runtime with a codelet-based [184] execution mechanism. SWARM breaks down a program into tasks with runtime

dependencies and limitations that may be performed after all of the dependencies and constraints have been fulfilled. Therefore, the task execution is scheduled by the runtime depending on resource availability. SWARM also employs a work-stealing strategy for on-demand load balancing. In particular, SWARM is a platform and runtime system that uses a novel execution paradigm in a layer between the operating system and the application. It manages of available hardware and software resources (such as threads, memory, accelerators, and networking) and dynamically assigns work and data to them as they become available.

Trebuchet [6] presents the implementation of dynamic dataflow architecture. Trebuchet presents the execution of code blocks based on a multi-thread dataflow model. Trebuchet emphasize on parallelizing the instructions (ILP) using dataflow semantics. To use Trebuchet to parallelize sequential programs, they must first be compiled into TALM's dataflow assembly language and then executed on Trebuchet. Trebuchet is implemented as a virtual machine, with each PE corresponding to a thread on the host computer. Instructions are assigned to the PEs and fired according to the dataflow model when a program is performed on Trebuchet.

In XKaapi [60], the authors show a dataflow task acceleration on distributed heterogeneous target devices such as Multi-CPUs, and Multi-GPUs. XKaapi has been written in C++ language, and a work-stealing method has been presented for scheduling ready tasks via a runtime system. In this work, the authors proposed a locality-aware work stealing algorithm based on heuristics to manage data locality and tackle the cache unfriendly problem of classic work stealing. The work stealing approach is inspired from Cilk [116] programming model. The procedure is that an idle thread, called a thief, initiates a steal request to a random selected victim. On reply, the thief receives a copy of one ready task, leaving the original task marked as stolen.

These works use dataflow approaches to improve the execution time. In contrast, Dataflow thread -which is focus of this thesis- ambition is to provide a tool for ***testing and debugging dataflow benchmarks***, while the performance is obtained by deploying one DF-Thread implementation [61, 66, 157]. In particular, DRT represents a key element to develop a toolchain to support a dataflow execution model, which could be targeted by a compiler. While there are many similarities between DRT and the above works, I choose a more detailed comparison with the Codelet program execution model [184, 24] and Open Community Runtime (OCR) [104, 113].

In the Codelet execution model concept [184], Codelet is a fine grain event-driven unit of computation, smaller than a thread, aims to exploit the parallelism of Exascale platforms. The runtime environment DARTS[183] has been presented in such a way that a high-level program will turn into Codelet Graph with the API interface, and the runtime executes the Codelets based programs to exploit the maximum parallelism and power efficiency of the underlying hardware. DARTS uses a double level hierarchy to structure programs: threaded procedures (TP) and Codelets; TP includes several Codelets. In contrast, DF-Threads leaves more freedom to the programmer by using a flat hierarchy of threads.

The Open Community Runtime [104] is based on event driven tasks. OCR is a runtime that is influenced by the Codelet execution model and is inspired by the Asynchronous Many Task (AMT) models. A high-level program written on OCR runtime is organized with Directed Acyclic Graph, which is structured with relocatable data-blocks, events, or tasks. These elements are called nodes connected to each other by edges, which represent the dependencies between nodes. DARTS and OCR trigger threads by using both data and events. In DF-Threads, we do not need this distinction: events can be treated as data.

DF-Threads [61] introduces a low-level API, which enables a high-level code into a hybrid dataflow model that can benefit from the high parallelism while parallel computations are the potential to distribute over nodes and cores.

2.5 Dataflow Execution Models using Hardware Accelerator

In this section I add the related studies and their implementation comparison of Dataflow Architecture mainly on works that provide Dataflow Execution using Hardware Accelerator. The main reason is that we compare other related studies with our co-processor implementation of DF-Threads. The detail discussion about the proposed model is covered in [106, 122].

As can be seen in Fig. 2.4, the Dataflow Threads high-level overview and other state-of-the-art dataflow architectures are depicted. The main advantage of DF-Threads versus Spatial Dataflow and Stream Dataflow concept is that in DF-Thread we schedule the dataflow elements in the granularity equal to a thread. A thread is the smallest element of instructions that can be managed independently by the dataflow scheduler. As illustrated in Fig. 2.4, Stream Dataflow architecture is mainly based on Coarse Grain Reconfigurable Architecture, and in the Spatial Dataflow architecture, it is based on the parallelized PEs and available hardware on the device. The DF-Threads execution paradigm, in particular, is based on multi-threading, with dynamic dataflow principles being used to create a Dataflow graph (DFG) across threads and to efficiently leverage control flow execution within a thread. Furthermore, obtaining the ready-to-execute instructions is deterministic (when all of the thread's inputs are accessible), which makes it close to optimum since the DF-Threads management plans each thread's life-time and knows which thread will be run at any given time. In contrast, in spatial dataflow model, a program first will be translated into dataflow graph (DFG), then the DFG will be mapped on available PEs on the hardware.

Fig. 2.5 shows the abstract model of a node in a multi-node platform, representing a Dynamic dataflow multi-node execution model first proposed in [129]. In this model the local memory of each node consists of an Instruction Memory which is read by the Execution Unit and a Data Memory (or Frame Store) which is accessed by the Synchronization Unit. The Ready Queue contains the continuations representing those threads that are ready to execute. The Structure Memory stores data structures and is distributed among the nodes. The Mem Unit handles the structure memory requests. The execution is based on

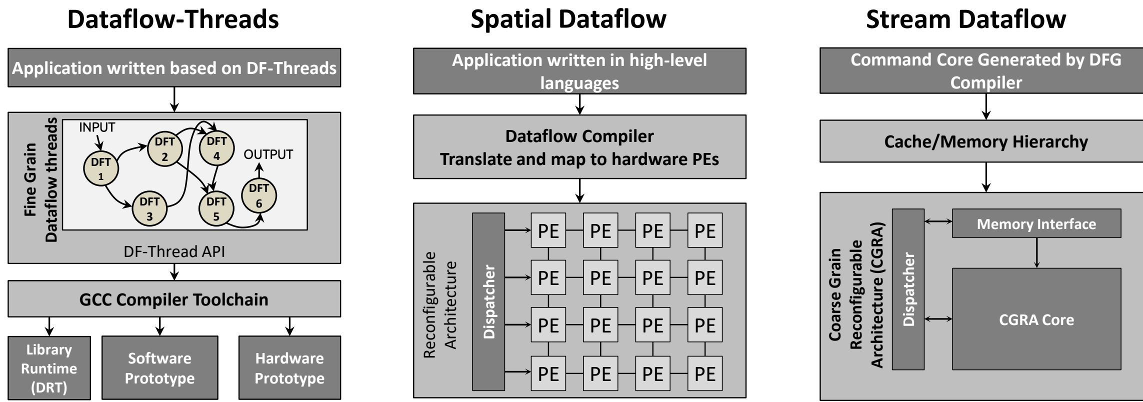


Fig. 2.4.: The Dataflow Thread execution model high-level architecture versus well-known modern dataflow approaches.

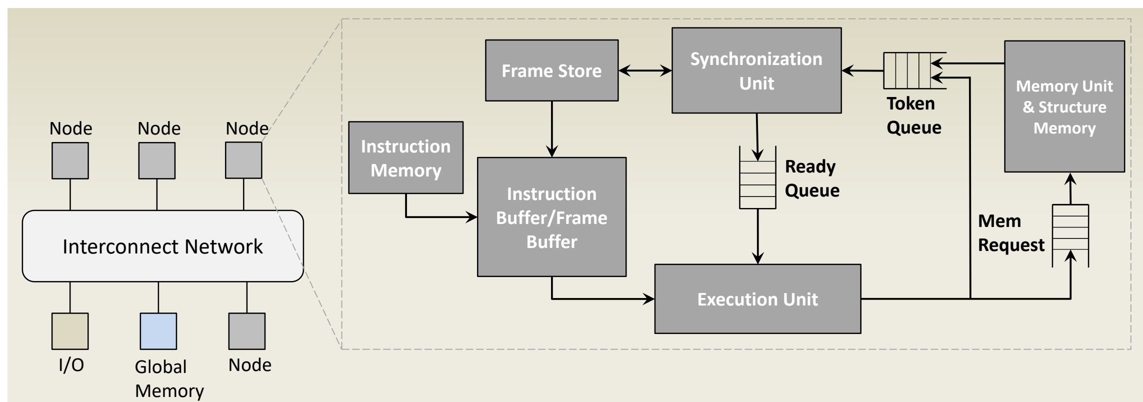


Fig. 2.5.: The abstract model of the processing node using dynamic dataflow along with synchronization unit proposed in [130] and [129].

dynamic dataflow scheduling where each actor, or a node in the dataflow graphs, represents a thread [129]. Whereas in Dataflow Threads, as discussed, a thread is the smallest element of instructions that can be managed independently by the dataflow scheduler. Dataflow-Threads provides much finer grained granularity of computation.

RISC-V Approach

The ultimate goal of the dataflow machine, is where the dataflow semantics mapped and deployed on a underlying machine, which could exploit underlying hardware to achieve an ideal performance. Eliminating the von Neumann control flow deficiencies and bottlenecks is the most ambition of dataflow machines.

To achieve this, there were difficulties which hindered to build a such machine [138]. By increasing the attitude to use reconfigurable hardware such as FPGA, researches become motivated to implement dataflow architecture on FPGA and the suitable architecture to implement the dataflow idea was RISC core. The idea first proposed in [109] and later by [35]. Fig. 2.6 shows a modest dataflow architecture to deploy on RISC-V hardware. The authors show that this approach not only can exploit both conventional and dataflow compiling technology but, more so than its predecessors, it can be viewed as a dataflow machine that can achieve complete software compatibility with conventional von Neumann

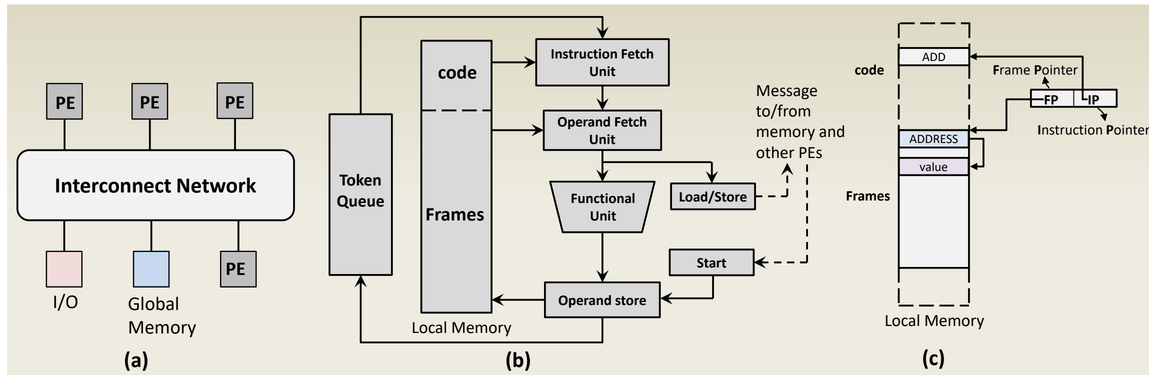


Fig. 2.6.: The dataflow RISC multiprocessor organization proposed in [109]. In this work the dataflow semantics and its instruction set based on I-structure method is mapped to the RISC processor.

machines. The authors proposed three instructions for their RISC-V dataflow machine pipeline, `START`, `READ` and `WRITE`. Which are responsible for **`SCHEDULE`**, **`FETCH`** and **`WRITE`** the frame token from/to different PEs.

There are other works which inspired from the concept of transporting dataflow architecture in RISC processor using reconfigurable devices. In [35], the authors proposed an ISA extension for a RICS-V dataflow implementation. This work which is based on "tagged token dataflow", proposes a dataflow (DF) extension to the RISC-V Instruction Set Architecture (ISA). To test the performance of the extension, the authors created a dataflow CPU model and integrated it with the Gem5 simulator. The authors designed a heterogeneous dataflow/Von Neumann architecture and defined the memory instruction set and RISC-V instruction bit-field illustrated in Fig. 2.7. Similar to conventional RISC-V instructions, the instructions are encoded with opcode and function bits to indicate the operation. Instead of specifying the source operands with the remaining bits, the dataflow instructions contain pointers to dependent instructions: Destination0, Destination1, and Destination2– which represent the arcs in a DF graph. To differentiate between left and right operands, the remaining bits D2, D1, and D0 are utilized.

Task Scheduling on RISC-V

Other studies which their principals are based on Data-Driven Multi-threaded Dataflow architecture are [108, 92]. We try to describe most important notes in these works. In [108], the authors developed a system prototype in which Task Scheduling functionality is directly mapped by logic embedded in the processor (RISC-V core) and made available to applications utilizing specific instructions (an interface to RISC-V core). The overview of the design is depicted in Fig. 2.8.

This design diminishes Task Scheduling overhead by eliminating FPGA-CPU connection latencies, allowing activities to be scheduled to cores at much greater rates. Designing this new architecture involved using the Chisel language [15], to integrate Picos [144], a mature Task Scheduling accelerator, to Rocket Chip [13], an open-source, silicon-proven, multi-core implementation of RISC-V. To evaluate Task Parallel application performance of

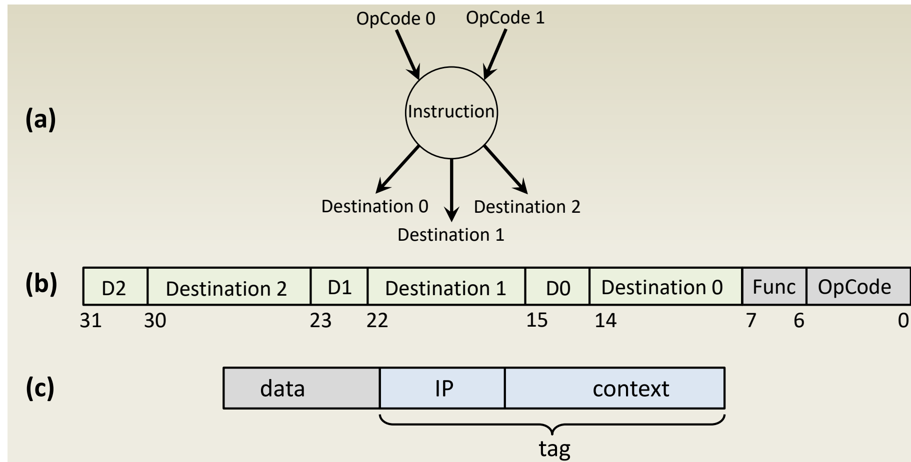


Fig. 2.7.: RISC-V Dataflow Extension instruction illustration in (a) and (b) presented in [35]. (c) shows the dataflow token. Here the tag is used to match operands that are part of the same instruction; for example, if an instruction includes two operands, both of them will have the same tag.

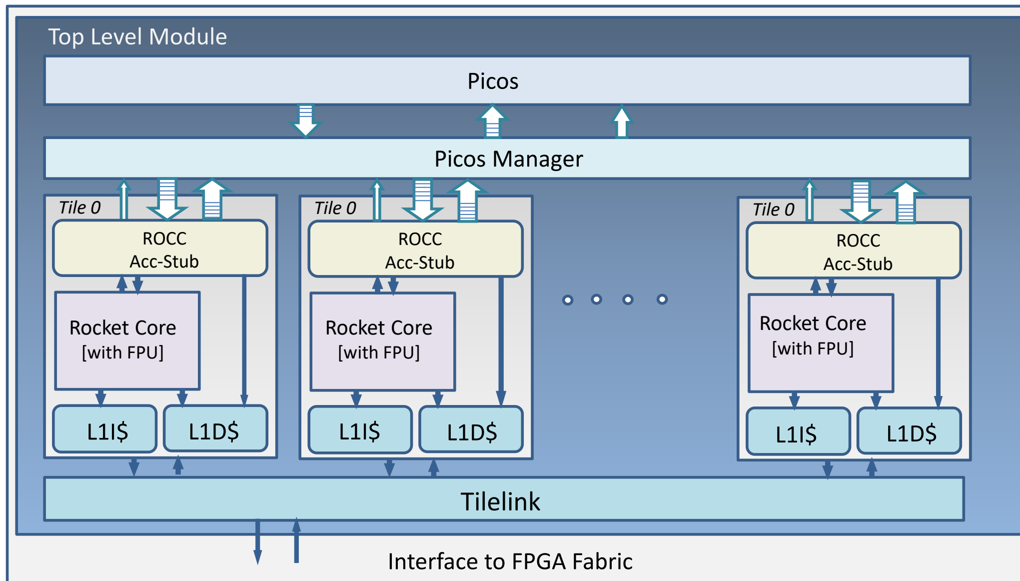


Fig. 2.8.: An overview of the Picos and Rocket chip system architecture [108].

this platform, the authors adapted Nanos [16] (a software-only Task Scheduling runtime, targeting the OmpSs programming model [51]) to the system. To develop a user interface, the authors also developed a lightweight, Task Scheduling runtime called Phentos. This work is explicitly using task-based dataflow architecture semantics [91, 143], however, this work is the one of the few ones which implement successfully the concept of Task-based fine-grained parallelism on hardware accelerator. In a nutshell, task parallelism is a technique for automatically translating sequential imperative programs into dataflow models, in which essential operations are executed simultaneously and asynchronously as soon as their input data becomes available. We found this approach similar to the Dataflow Threads approach that we discussed in this thesis and our previous works [67, 66].

3 Dataflow Execution Model Baseline Study

> *A picture is worth a thousand words. An interface is worth a thousand pictures.*
>
> — **Ben Shneiderman**
> (Professor for Computer Science)

The purpose of this chapter is to establish a baseline for the dataflow execution model by comparing different benchmarks with a detailed comparison to existing parallel execution models such as cilk and OpenMPI. This chapter demonstrates the evaluation results of the same algorithm for all benchmarks and the benefits and drawbacks of the dataflow execution paradigm using these experiments. This chapter focus on four benchmarks: Matrix Multiplication, Fibonacci, Blocked Cholesky, and Histogram. The reason to select these benchmarks is to evaluate the execution model with special stress on different aspects as described below:

- **Recursive Fibonacci**: Recursive Fibonacci (RFIB) benchmark is used to generate a high number of threads and widely used in order to stress the execution model to handle a big number of individual threads.
- **Blocked Matrix Multiplication**: Blocked Matrix Multiplication (BMM) is a very widely used kernel in many applications (especially in Artificial Intelligence, Deep Neural Networks, etc.), and it stresses memory hierarchy of the system.
- **Cholesky Factorization**: Based on linear algebra, the definition is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is useful for efficient numerical solutions like solving least square problems. It is useful to evaluate the computation and communication overhead of the execution model.
- **Histogram**: Histogram evaluates the memory model in case of data race condition. It may have multiple threads accessing the same bin of the histogram for updating the bin count.

Apart from the above benchmarks, the capability of our execution model with more sophisticated benchmarks that will stress the computation on different aspects will be demonstrated. Such as creating multiple threads, Moving data around as memory operations, and memory policies that will prevent concurrent read/write on a specific memory location. For these reasons, I choose graph computing applications like **PageRank**, which includes all these characteristics. To demonstrate the potential of the Dataflow-Threads execution model, I

evaluate the metrics on the most well-known execution models. The Sequential execution (which is the basic implementation of the mentioned benchmarks) then I evaluate the benchmarks **with the same algorithm** running on **MPI** [115, 77] and **Cilk** [33] programming models to compare how the execution will be working.

Along with the result comparisons and providing a strong baseline, I consider investigating a bit in the literature to compare the baseline results with other related works to validate the baseline results. To evaluate high-performance parallel computing methods and algorithms, we need to measure the efficiency of the evaluated method. we use the strong scaling method in which the experiment will be executed in a loop. The fixed-size problem will execute with a specific number of processors/workers/threads in each iteration. In the next iteration, the same problem size will be executed by more processors based on how many resources are available in the platform.

Finally, I show the execution time and calculate the speedup based on the Eq. 3.1.

$$\text{Speedup} = \frac{T_{sequential}}{T_n} \tag{3.1}$$

Where $T_{Sequential}$ is the computational time for running the software running sequentially on a platform, and T_n is the computational time running the same software with n processors. Here by processor we mean hardware unit in the platform like Core/Thread.

Ideally, I would like the program to achieve a linear speedup while the number of processors (speedup = n) is increased. It means that every processor would contribute all of its computational resources. Unfortunately, achieving this aim in real-world applications is quite difficult. In this section, I quickly describe the most critical bottlenecks, limitations, and procedures that I used to establish the baseline across our platforms, and at the end of each section I point out the lessons learned from each baseline.

For more explanation about the scalability experiment source codes and the scripts, please refer to Section Appendix A at the end of the thesis.

3.1 Methodology

The methodology to establish such baseline incorporated the following items:

1) Finding the most relevant benchmarks each of which can stress the execution model in such a way to show the potential of the execution model.

2) Implementing the appropriate algorithm with the different execution model including our proposed execution model.

3) Executing the same benchmarks with different execution models on the same platform (multi node or multi core).

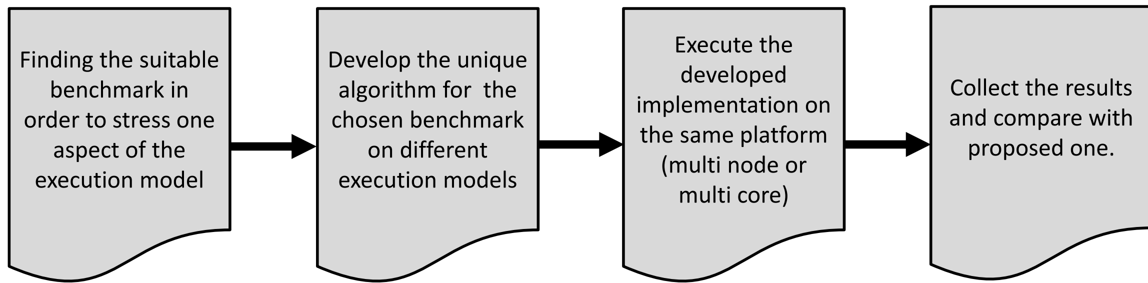


Fig. 3.1.: The methodology to develop the desired baseline.

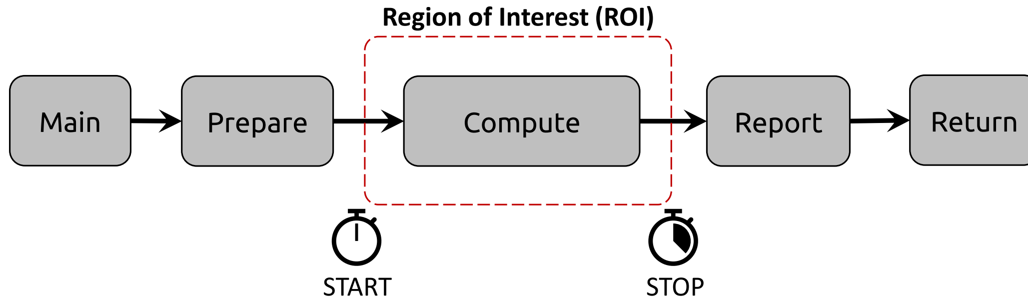


Fig. 3.2.: The separated functions development in all algorithms and programming models to ensure Region of Interest measures the same part of the algorithm in all benchmarks.

4) Comparing the results achieved from the previous item and show the comparison. This results can express the execution time, speedup, kernel activity, etc.

This methodology that is used in this thesis is depicted in Fig. 3.1. Execution time is the main metric to be investigated in these experiments, measured by using `gettimeofday(&tv, NULL)` function from `time.h` Linux library. Based on our coding style, we develop the program in separate function to eliminate the impact of declarations and memory initialization from the main goal of the study. To do this, as can be seen in Fig. 3.2, we develop the declarations and initialization in the `prepare()` function, then we call the `compute()` and start the measurement here, once this function is returned we stop the measurement. `Cilk_sync` is used in programming models like "Cilk" or in OpenMPI the `MPI_Barrier` to be sure that all the parallel instances are computed and reached to this point before stopping the measurement. Finally, we check and validate the result calculated within the `report()` function and compare the achieved result with the sequential version of implementation to inform the user that the execution was successful or not.

In some cases, to investigate more about the scaling and efficiency of the algorithm weak-scaling method is used. The appropriate report will be discussed later when we talk more about the efficiency of the benchmark execution. In the strong scaling we increase the number of cores/workers/processors with the fixed problem size, while in the weak scaling both the problem size (in one dimension) and the number of cores/workers/processors is increased.

To go through the baseline study, I adopt and develop the selective benchmarks to express the potential of our execution model. The list of the benchmarks and the status of their implementation is listed on Table 3.1.

Tab. 3.1.: The status of the benchmarks that have been developed for the baseline study, software simulation and hardware implementation, N stands for Node and C stands for Cores.

	Implementation	RFIB	BMM	Cholesky	Histogram	PageRank
Baseline	Sequential	✓	✓	✓	✓	✓
	MPI	✓	✓	✓	✓	✓
	Cilk	✓	✓	✓	✓	✓
Software Simulation	COTSon	✓	✓	✓	✓	-
Hardware Implementation	FPGA RISC-V	1N-1C	-	-	-	-
	FPGA ZYNQ	1N-1C 2N-1C	-	-	-	-

Tab. 3.2.: The hardware specification of platforms used in this experiment.

Hosts	OS	RAM	Processor	Sockets	Core per Socket	Thread per Core	L1d	L1i	L2	L3	MAX Freq
Udoo-x [1]	Ubuntu 16.04	8 GB	Core i5 9400F	4	1	1	24K	32K	1024K	-	1.6 GHz
RV0	Ubuntu 20.04	16 GB	SiFive RISC-V	4	1	1	32K	32K	2048K	-	1.0 GHz
Lab146 [1]	Ubuntu 18.04	8 GB	Core i5 9400F	1	6	1	32K	32K	256K	9216K	3.9 GHz
TFX2	Ubuntu 18.04	256 GB	AMD Opteron 6168	4	12	1	64K	64K	512K	5118K	1.9 GHz
TFX3	Ubuntu 18.04	1 TB	AMD Opteron 6272	4	8	2 [2]	16K	64K	2048K	6144K	1.9 GHz
COTSon [3]	ubuntu 18.04	4 GB	-	1	1	1	32K	32K	1024K	4M	1.0 GHz

[1] Udoo-x and Lab146 are multi-node.
[2] There is a dispute for TFX3, which seems with hwloc library [121], which OpeMPI uses this library to recognize the underlying hardware, the CPU topology is not hyper threaded.
[3] I use COTSon simulator to evaluate our Dataflow Thread programs.

3.2 Reference code and Hardware

All the experiments in this document are based on the C codes written and maintained in the our local repository. The source code of the all the experiments are available and will be sent upon the request.

I used different machines to evaluate the benchmarks, these machines have different configurations, the specifications of each machine such as Processor, Memory, Cache Hierarchy and other useful information are listed in Table 3.2.

In this configuration, we use machine named **TFX2** for Multi-core execution and machine named **Lab146** for running benchmarks on Multi-node execution.

3.3 Environment, Compilers, and Setups

In this section I show the performance analysis of the given algorithm with different environments, parallel programming models, and their setup with different hardware. The Compilers, their flags and the necessary environment variables are described below, see Table 3.3, Table 3.4 , and Table 3.5 for the configurations that have been used for each environment.

Furthermore, to do the correct time measurement one should consider the following items and take them into account to measure properly:

- Use the clock with suitable precision, not too much not too less, and make the Region of Interest (ROI) probes to the region which evolves the computational section of the program, not the initialization or report part.
- It is inappropriate to refer to scaling numbers with more than 1 CPU as the baseline.
- Measure multiple independent runs per problem size in a loop and measure the average value. In our study the default loop iteration to measure a stable timing is 10 iteration, in case it is easy to change the repetition number to higher value (for smaller problem sizes I did with 30 iterations).
- Various factors must be taken into account when more than one node (multi-node) is used:

  (a) Interconnect speed and latency

  (b) Max memory per node

  (c) Processors per node

  (d) Max processors

  (e) System variables and restrictions (e.g. stack size, CPU aggressive monitoring tool, etc.)

  Note that the adjustment of MPI parameters may also substantially enhance the performance of MPI-based applications. MPI programs also need a specific amount of memory for each MPI process.

- Additionally but not necessarily if possible measure using different systems (as I do in this document with several hardware). Most importantly ones that have significantly different processor / network balances (ie. CPU speed vs. interconnect speed).
- Take into account that the CPU is hyper thread enabled or not. This feature may cause some not expected results from parallel processing, unless you exactly know what you are dealing with hyper-threading.
- Last but not least, always check the workload of the machine you are using. By using `w` command in ubuntu, you simply realize the workload in last 1, 5, 15 minutes (in the load average section). If your machine is under another load, your results are most likely invalid.

Tab. 3.3.: The configurations of the experiment that has been done on **Lab146**.

Title	Description
Operating system	`Ubuntu 18.04.3 LTS`
OpenMPI	`version 2.1.1`
OpenMPI options	`mpirun -n $j -hostfile ~/hosts ./executable $i`
GCC Version	`version 7.4.0`
GCC Flags	`-O3`
Kernel	`4.15.0-65-generic`
CPU	`see Table ??`
Network	`10M/100M/1G multi-port switch`

Tab. 3.4.: The configurations of the experiment that has been done on **TFX2** a Single Multi-core Machine.

Title	Description
Operating system	`Ubuntu 18.04.3 LTS`
OpenMPI	`version 2.1.1`
OpenMPI options	`mpirun -n $j -hostfile ~/hosts ./executable $i`
GCC Version	`version 7.5.0`
GCC Flags	`-O3`
Kernel	`4.15.0-142-generic`
CPU	`see Table ??`
Network	`Single Multi-core Machine`

Tab. 3.5.: The configurations of the experiment that has been done on **TFX3** a Single Multi-core Machine.

Title	Description
Operating system	`Ubuntu 18.04.3 LTS`
OpenMPI	`version 2.1.1`
OpenMPI options	`mpirun -n $j -use-hwthread-cpus -hostfile ~/hosts ./executable $i`
GCC Version	`version 7.5.0`
GCC Flags	`-O3`
Kernel	`4.15.0-154-generic`
CPU	`see Table ??`
Network	`Single Multi-core Machine`

3.4 Literature Study

This document focuses on two parallel programming languages that are more closely related to our execution model. OpenMPI [115] and Cilk [33] are two well-known parallel programming languages in this scope, for which I attempt to provide a baseline based on the benchmarks mentioned and evaluate how these parallel programming paradigms work. OpenMPI has been introduced as an open-source implementation of MPI industry standard, developed and maintained by academia, the individual contributes, and some industry-based research centers and its first release was in 2010. Gradually it became popular, and many research centers and universities tried to contribute, even in industry section, some companies like Intel, IBM, and CUDA developed their branch and released their optimized and specialized version of OpenMPI.

Cilk first evolved at MIT in 1994 as a parallel programming extension to the C programming language. In 2006, Cilk became a startup named "Cilk Arts", which produced the open-supply platform Cilk++ as an extension to C++ compiler. Intel Corporation acquired Cilk Arts in 2009, brought vectorization directives, and renamed Cilk++ to the Intel Cilk Plus platform, making it industrial in its compiler. Nowadays a open-supply releases for the GCC and LLVM compilers are available.

Recently, Cilk development continued by a research group in MIT, and they changed the name to Opencilk [116]. OpenCilk is an open-source implementation of the Cilk concurrency platform developed and maintained by MIT researchers. Cilk applications are compiled using the Tapir/LLVM compiler, based on Clang and LLVM, and build parallel programs more efficiently than current compilers for parallel programming languages.

These two programming paradigms are still under development, and many publications choose them as a baseline to propose their new work. Here I use benchmarks that have been widely implemented using these two programming models. The difficulty here is that the related literature is still not mature enough because these programming models are still under development, and plenty of updates have been deployed so far in many different aspects. In this context, OpenMPI is well-known among High Performance Computing research groups, and there are many works, and active projects that use OpenMPI as a key programming model.

In [87], the authors use a matrix multiplication baseline that has been implemented with a two-sided method (`MPI_send() and MPI_recv()`) on Cray XC40 CPU node. This node has 32x Intel Xeon Processor E5-2690 v3 Haswell chips. The MPI compiler used is IntelMPI version 5.0.3, and the biggest matrix size that has been evaluated is **5120** (like many research in the literature, the authors didn't explain why and how they evaluated such numbers for the matrix size). The measurement method in this work uses MPI wall clock `MPI_Wtime` and does the measurement in a loop with 15 iterations. In another work, [134] there is a similar approach to build a baseline with different parallel programming models like OpenMP [148] and OpenMPI. In this work, the authors evaluated the matrix multiplication

experiment on an old machine with a Pentium D (3 GHz) processor, MPICH Compiler, Linux Fedora 14, and 1 GB of RAM. In [2], a comparison between five parallel programming models has been evaluated. In this article, the processor is Intel CoreTM2 Duo Processor T6500 with 4GB of RAM and a clock frequency of 2.10 GHz with Ubuntu 10 Operating System installed. Another work [103], authors present a Data-Driven Multi-threading approach and the evaluation is based on benchmarks such as: Recursive Fibonacci, Blocked Matrix Multiplication and Blocked Cholesky. They accordingly report the execution time for sequential version of mentioned benchmarks as a reference. In this work, the evaluation has done with HP server machine with 2 AMD Opteron 6276 processors running at 1.4 GHz that supports 32 threads. Each processor is an 8-core 64-bit Clustered Multi-Threaded (CMT) architecture with the capacity of running 16 threads simultaneously. Each core has a 16KB 4-way set associative L1 data cache, a 64K 2-way set associative L1 instruction cache and a 2MB 16-way set associative L2 cache. Also, each processor utilizes a 6MB 64-way set associative L3 cache. The server is equipped with a shared 48GB DDR3 RAM clocked at 1333MHz and the server runs the Ubuntu 14.04 OS.

In [99], the authors show the dynamic behavior of MPI programs using Recursive Fibonacci benchmark. In this work the evaluation and baseline have been done within 30 nodes from the French Grid'5000 testbed with two Dual Core processors (four cores per node) and Gigabit Ethernet network. The MPI-2 implementation used was MPICH2 version 1.0.8p1 that has support for dynamic process creation and multi-threaded MPI calls.

In [75] and [85], the authors show an approach for specific optimization on parallel programming load balancing and context-aware nested recursive algorithms. In these works the authors choose Cilk as evaluation baseline and Recursive Fibonacci as the benchmark. The evaluation of [99] have been done on Lenovo Thinkpad p51 20HH000TUS with an Intel i7 processor with four cores, eight virtual-cores/threads is used. It runs Ubuntu 18.04.4 LTS. Flag `-O3` is used for compilation with GCC compiler and and In [85] the evaluation have been done using Intel Xeon E5-4650 processors, each offering 8 cores clocked at a nominal frequency of 2.7 GHz (up to 3.3 GHz with Turbo Boost).

In [176], the authors provided a wide range of experiment on Intel® Transactional Synchronization extension for high performance computing. They proposed and evaluate a set of algorithm for Transactional Memory concept and to do their evaluation they used Histogram as a benchmark.

3.5 Recursive Fibonacci

As mentioned before, Recursive Fibonacci(RFIB) is a kernel that can easily create a huge number of threads. Therefore it is a suitable benchmark to evaluate how the execution model manages many threads during the execution. To explain the RFIB algorithm, it 'unwinds' the number you give it until it can get a value (0 or 1) and then adds that to the total. The "unwinding" occurs each time that the value of `n-2` and the value of the `n-1` is

```
int serialfib(int n)
{
    if(n < 1) return 1;
    return n < 2 ? n : serialfib(n-1) + serialfib(n-2);
}
int RFIB(int n, int th) {
    if(n<2)
        return n;
    if( n < th)
    {
        return serialfib(n);
    } else
    {
        return RFIB(n-2,th) + RFIB(n-1,th)
    }
}
```

Listing 3.1: Recursive Fibonacci function code

given to the RFIB method (n is the number of the RFIB index in each recursion) when the last line of the Listing 4.6 is reached.

With each recursion where the method variable number is NOT smaller than 2, the state or instance of the RFIB method is stored in memory, and the method is called again. Each time the RFIB method is called, the value passed in is less than the value passed in during the previous recursive call (by either 1 or 2). This procedure continues until the value returned is smaller than 2 (either 0 or 1).

The required RFIB number is then calculated by adding these values together. Each time a 0 or 1 is returned from one instance of the RFIB method to the previous instance of the RFIB method, and so on, this summing operation occurs.

Here we use a threshold or cut-off to control the granularity of the number of produced threads by RFIB. By this means, when the recursion reaches the number of the cut-off or threshold, the execution goes with a sequential version and won't go forward. In this way, the granularity will be adjustable by the user.

RFIB Sequential Execution

For the Recursive Fibonacci, the sequential implementation is evaluated by the reference code as mentioned in Section 3.2. The execution time for different index numbers are shown in Table 3.6. We keep the sequential results of the Recursive Fibonacci as a references to compare with other works, since it is the sequential version and other factors like algorithm and programming model overhead is not involved in the sequential version.

RFIB with Cilk

In order to evaluate the RFIB with cilk programming model, we installed the cilk version 5 using Ubuntu 18.04 official packages, and the implementation of the RFIB is the reference code as mentioned in the Section 3.2. The Listing 3.7 shows the algorithm implemented with cilk primitives to get insight into how the implementation is done. Cilk schedules processes using the work-stealing concept rather than the work-sharing approach. When a

Tab. 3.6.: The sequential version of Recursive Fibonacci on three platforms TFX2, Lab146 and COTSon simulator.

Fibonacci index	Execution time (ms)		
	TFX2	Lab146	COTSon
5+1	0.002	0.001	0.002
10+1	0.005	0.001	0.0063
15+1	0.047	0.003	0.046
20+1	0.5	0.028	0.528
20+10	0.35	0.023	0.35
25+12	3.4	0.215	3.63
30+20	23.2	2.1	38.24
35+22	199.9	23.67	432.856
40+30	2105.9	269.85	4955.83
45+32	21606.2	2964.8	55143.33

```
int serialfib(int n){ //sequential function
    if(n < 1) return 1;
    return n < 2 ? n : serialfib(n-1) + serialfib(n-2);
}
long int RFIB( int n,int th){ //the parallel recursive function
        if( n < th){
                return serialfib(n);
        }else{
                long int x = cilk_spawn RFIB(n-1,th);
                long int y = RFIB(n-2,th);

                cilk_sync;
                return x+y; }
}
```

Listing 3.2: Recursive Fibonacci function code using Cilk programming model.

thread is scheduled to execute in parallel whenever the runtime performs an asynchronous function call, this is known as work-sharing.

The execution time for different RFIB index numbers are summarized in Table 3.7.

RFIB with MPI

RFIB with MPI has been done within an inspiration from the study of dynamic behavior of MPI in [25]. In this work we evaluate the RFIB within dynamic process scheduling with MPI. To calculate the RFIB recursion, we use `MPI_Comm_spawn()` for each instance of RFIB. Every spawn requests results in the child-root connecting to the parent process to exchange information. An application that spawns tasks frequently will incur the overhead of this connection establishment and communication for every individual spawn. This overhead increases significantly while we use `MPI_send()` and `MPI_Recv()` to send and receive childs over the ranks within multi-core or multi-node platform. The reason is that for each spawning, the MPI Communicator creates a file descriptor for shared memory system and a SSH connection for multi-node platform. This factors together make the execution of RFIB within Dynamic MPI becomes much slower. Note that the scheduling

Tab. 3.7.: The Recursive Fibonacci on platform TFX2 with Cilk programming model.

Fibonacci index	Execution time (ms)			
	1N1C	1N2C	1N4C	1N8C
5+1	0.359	0.562	0.927	1.531
10+1	0.396	0.654	1.028	1.764
15+1	1.137	1.117	1.445	2.455
20+1	8.754	5.287	4.009	3.664
20+10	0.758	0.937	1.392	2.466
25+12	4.627	2.935	2.644	3.033
30+20	23.736	17.138	10.666	7.498
35+22	205.63	107.444	56.232	34.455
40+30	2217.565	1233.658	579.294	324.787
45+32	24549.696	12285.544	6621.103	3246.632

Tab. 3.8.: The Recursive Fibonacci on platform TFX2 with MPICH programming model.

Fibonacci index	Execution time (ms)			
	1N1C	1N2C	1N4C	1N8C
5+1	2300	1307	925	802
10+1	29296	15050	11650	10320
15+1	403203	248272	67059	179660
20+1	crash	crash	crash	crash
20+10	84374	45215	36218	35890
25+12	414870	228250	170620	161800
30+20	82270	49620	37800	35520
35+22	409030	234560	168510	164120
40+30	84470	47850	35700	35400
45+32	401770	253690	215370	172870

within dynamically processing of MPI was not successful on OpenMPI scheduler, therefore we switched to MPICH [77] and with Hydra scheduler we could successfully evaluate RFIB over different platforms both multi-core and multi-node.

The snippet in the Listing 3.3 and 3.4 shows the slave part of the RFIB development using MPI programming model first proposed in [25]. The execution time for different RFIB index numbers are summarized in Table 3.8.

RFIB with DF-Thread

To implement different benchmarks using DF-thread semantics we used the API functions that we covered in Section 4. To evaluate out DF-Thread code we have provide the same hardware platform similar to our real machines on COTson simulator [9, 122]. We configure our COTSon simulator to run the benchmarks based on multi-core or multi-node configurations requested. For the COTson architecture configuration that used in the evaluation of the benchmarks see Table **??**. **Lessons learned with RFIB**

We briefly show some numerical comparisons between small and large sizes of the RFIB indexed on some platforms and discuss about them.

```c
void worker_fib(char **argv)
{
    MPI_Comm children_comm[2];
    uint64_t x,y,n_worker;
    sprintf(command,"%s",argv[0]);
    argv += 1;
    n_worker  = atol (argv[0]);
    if (n_worker < thd || n_worker < 2 )
    {
          unit64_t res = serialfib(n_worker);
          MPI_Isend (&res, 1, MPI_DATATYPE, 0, 1, parent,&send);
          MPI_Wait(&send,&status);
    }else{
        sprintf (argv[0], "%ld", (n_worker - 1));
        MPI_Comm_spawn (command, argv, 1, local_info, myrank,
                MPI_COMM_SELF, &children_comm[0], MPI_ERRCODES_IGNORE);
        sprintf (argv[0], "%ld", (n_worker - 2));

         MPI_Comm_spawn (command, argv, 1, local_info, myrank,
                MPI_COMM_SELF, &children_comm[1], MPI_ERRCODES_IGNORE);
         MPI_Recv (&x, 1, MPI_DATATYPE, MPI_ANY_SOURCE, 1,
                children_comm[0], MPI_STATUS_IGNORE);

         MPI_Recv (&y, 1, MPI_DATATYPE, MPI_ANY_SOURCE, 1,
                children_comm[1],MPI_STATUS_IGNORE);

        fibn = x + y;              // computation
        MPI_Isend (&fibn, 1, MPI_DATATYPE, 0, 1, parent,&send);
        MPI_Wait(&send,&status);
    }
    MPI_Comm_disconnect(&parent);
}
```

Listing 3.3: The worker function of the Recursive Fibonacci function code using MPI programming model.

```c
void  root_fib(char **argv)
{
    MPI_Comm root_comm;
    int n_root=atoi(argv[1]);
    sprintf(command,"%s",argv[0]);
    argv+=1;
    if (n_root < thd){
        fibn = serialfib(n_root);
    }else{
      sprintf(argv[0],"%ld",n_root);
      MPI_Comm_spawn (command, argv, 1, local_info, myrank, MPI_COMM_SELF,
            &root_comm, MPI_ERRCODES_IGNORE);
      MPI_Recv (&fibn, 1, MPI_LONG, MPI_ANY_SOURCE, 1, root_comm,
                MPI_STATUS_IGNORE);
     MPI_Comm_disconnect(&root_comm);
    }
}
```

Listing 3.4: The root function of the Recursive Fibonacci function code using MPI programming model.

In Table 3.9, we summarize the multi-core experiment for RFIB. The multi-core experiment includes DF-Thread on COTSon versus shared memory models like Cilk and MPI. To this end, we use TFX2 platform for multi-core experiment and COTson simulator for DT-Thread evaluation.

Whereas in Table 3.9 we summarize the multi-node experiment on Lab146 platform, which is suitable for multi-node experiments. Same as multi-core platform, we evaluate DF-Threads on COTSon simulator with multi-node configuration. In Fig. 3.3 and Fig. 3.4, we show the scalability of the DF-Threads execution within RFIB benchmark over multi-core and multi-node platform. In multi-core apart from the execution time comparison with Cilk, we show the speedup plot with the MPI as the baseline that has been done on TFX2 platform. In multi-node we just evaluated RFIB with DF-Threads and MPI, in this case MPICH. Finally, we show the execution time evaluation and Speedup plot.

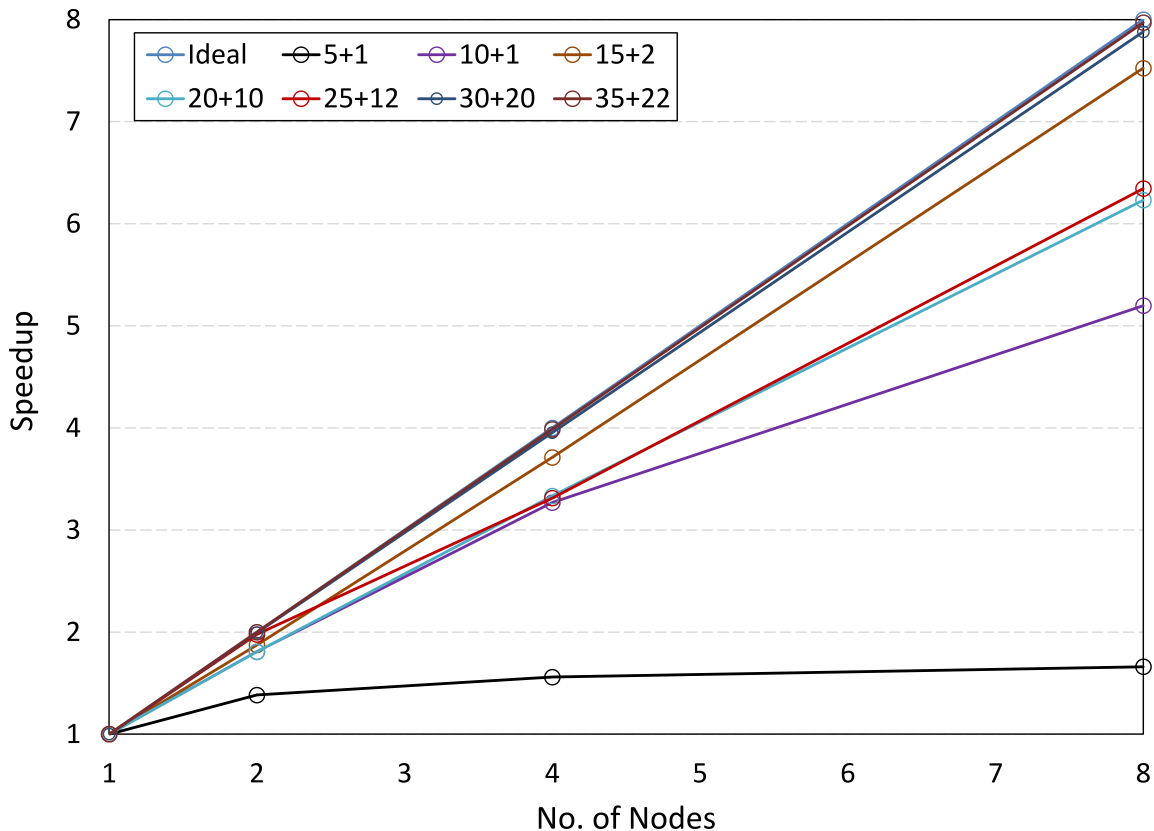


Fig. 3.3.: The speedup of the DF-Thread in the multi-node experiment.

The lessons we learn from these experiments on multi-core and multi-node platforms are as follows:

1) Recursive Fibonacci is a challenging benchmark for parallel programming model and it is hard to manage many fine-grained recursions among available hardware resources on multi-core and more on multi-node. DF-Thread has the capability to manage the recursion on both multi-node and multi-core platform with significantly shorter execution time compared to Cilk and MPI.

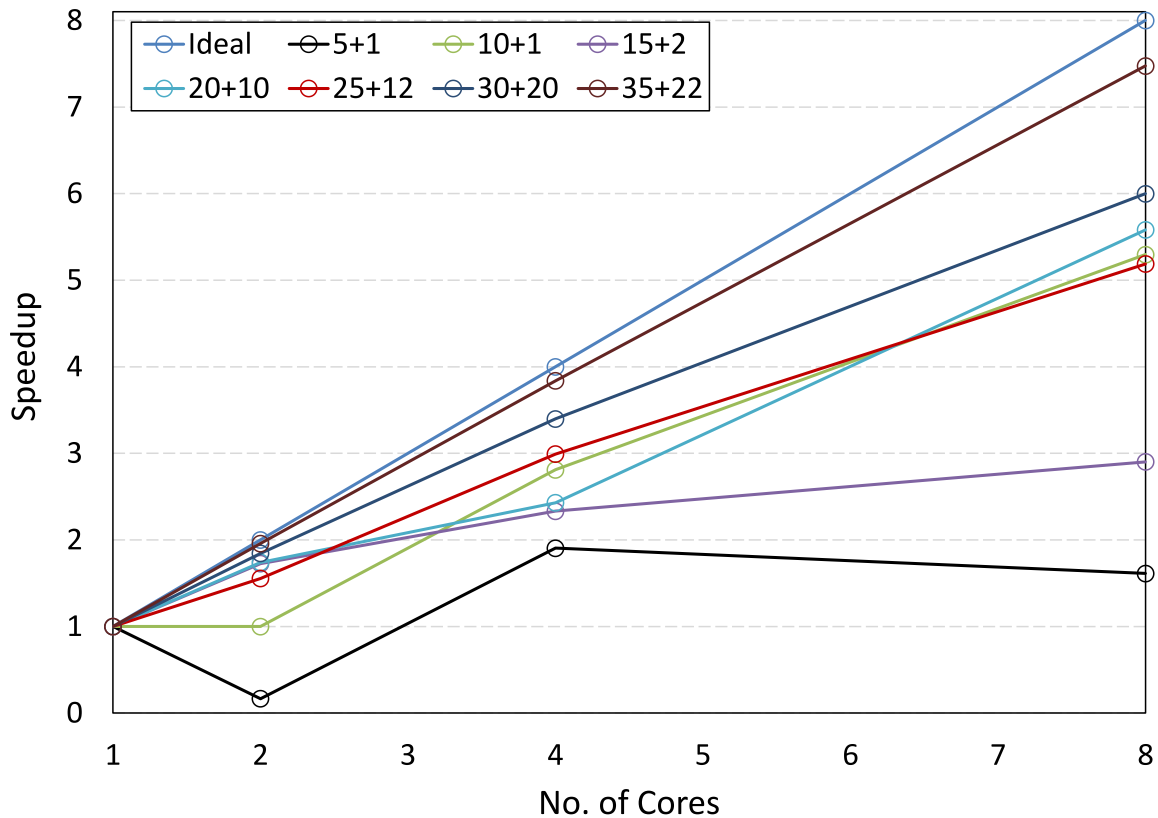


Fig. 3.4.: The speedup of the DF-Thread in the multi-core experiment.

2) DF-Threads shows an important scalability on multi-core experiment against Cilk. We show the 8 cores comparison of RFIB executed on COTSon simulator in Fig. 3.6 and Fig. 3.7.

3) Dynamic behavior of the recursion in MPI programming model is much slower than other programming models since in MPI, by spawning the child, a file descriptor and in multi-node experiment an SSH connection between childs will be created. This behavior makes the overall execution time much bigger than DF-Thread and Cilk.

4) On multi-core evaluation, there is a close competition between Cilk and DF-Threads. This reveals the fact that DF-Thread can be a good candidate for such execution model since it can easily distribute among cores and even nodes.

5) The scalability of the DF-Thread shows a close to ideal speedup in Fig. 3.3 and Fig. 3.4 for bigger workloads when The computation and network capacity is saturated.

Tab. 3.9.: The Multi-core experiment execution time for some selected indexes of RFIB on TFX2 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms).

Number of Processors	RFIB(10+1)			RFIB(15+1)			RFIB(20+10)			RFIB(25+12)			RFIB(35+22)		
	DF-Thread	MPICH	Cilk	DF-Thread	MPICH	Cilk	DF-Thread	MPICH	Cilk	DF-Thread	MPICH	Cilk	DF-thread	MPICH	Cilk
1	0.104	29296	0.396	0.88	403203	1.137	0.69	84374	0.78	6.13	414870	4.627	312	409030	205.63
2	0.16	15050	0.654	0.69	248272	1.117	0.4	45215	0.93	3.98	228250	2.935	37	234560	107.444
4	0.14	11650	1.028	0.45	67059	1.445	0.19	36218	1.39	2.1	170620	2.644	18.9	168510	56.232
8	0.01	10320	1.764	0.18	179660	2.455	0.17	35897	2.46	1.13	161800	3.033	9.7	164120	34.455

Tab. 3.10.: The Multi-node experiment execution time for some selected indexes of RFIB on LAB146 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms). Each node in this experiment has 1 Core.

Number of nodes	RFIB(10+1)		RFIB(15+1)		RFIB(20+10)		RFIB(25+12)		RFIB(35+22)	
	DF-Thread	MPICH	DF-Thread	MPICH	DF-Thread	MPICH	DF-Thread	MPICH	DF-thread	MPICH
1	0.163	7980	1.765	1075360	1.38	52048	12.2797	1171200	1449.21	1130460
2	0.09	9772	0.942	287439	0.76	24800	6.21021	320795	724.723	328180
4	0.049	7244	0.475	141450	0.41	19750	3.7078	197250	363.428	160750
8	0.031	5954	0.234	104730	0.22	17580	1.93543	106540	181.837	106560

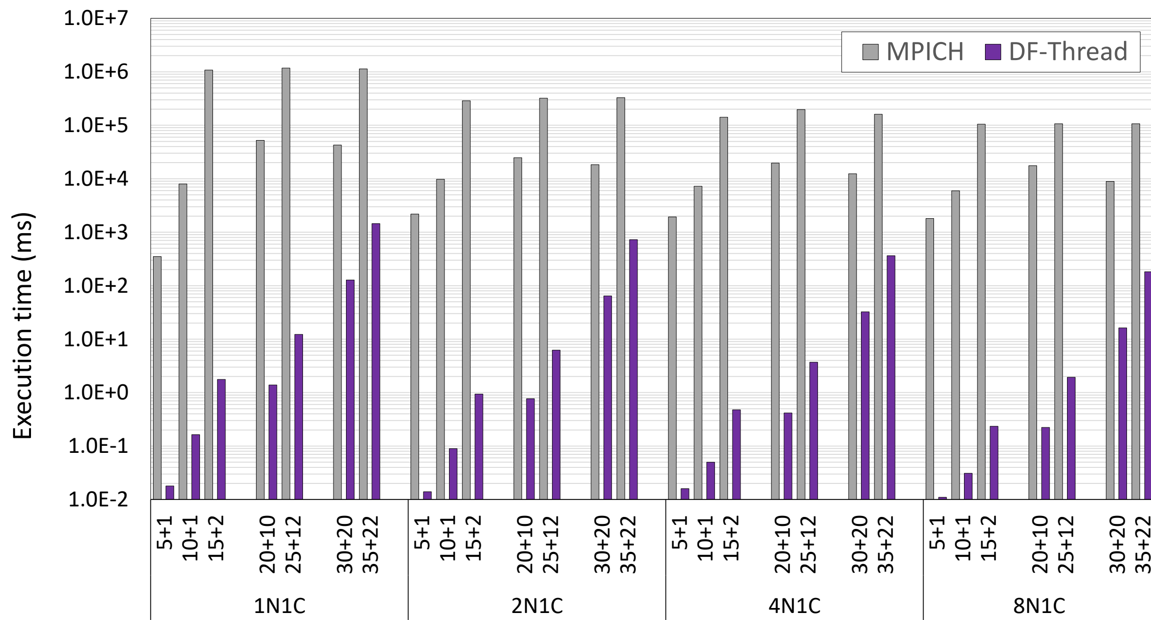


Fig. 3.5.: The RFIB multi-node experiment within DF-Thread and MPI. The baseline is MPI and we show how DF-Thread is comparable within the well-known programming model for variety of RFIB indexes.

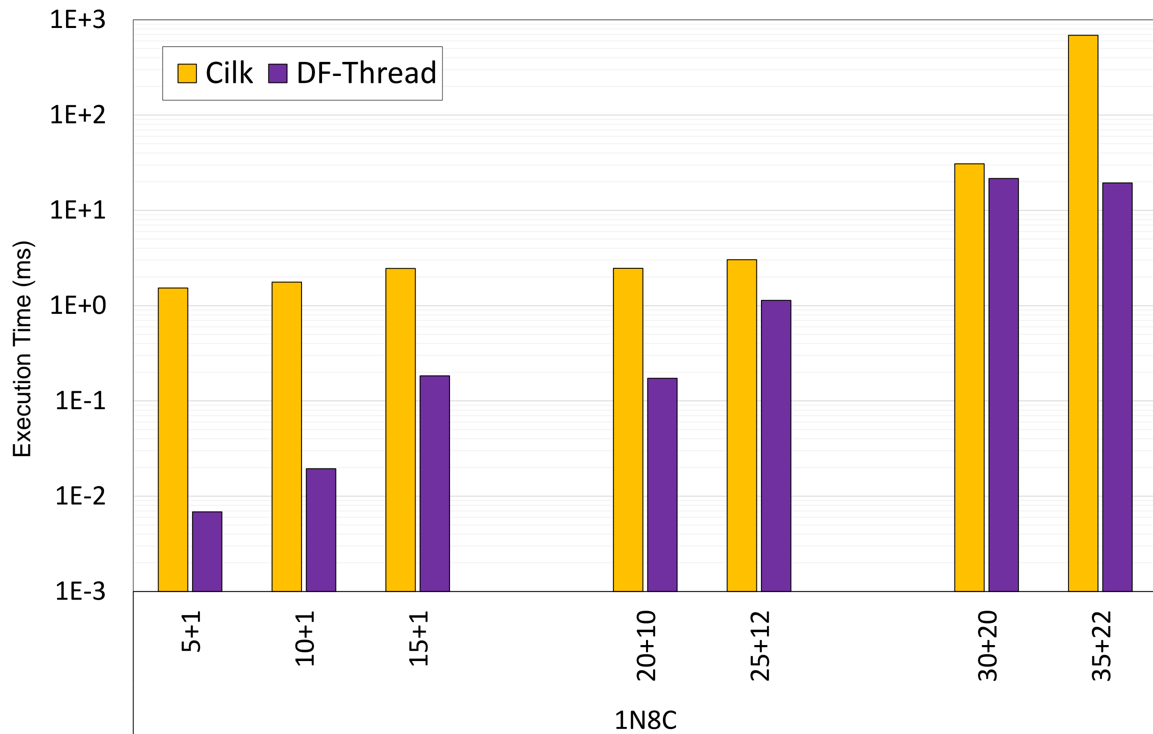


Fig. 3.6.: The Execution time comparison of DF-Threads against Cilk with the RFIB experiment on 8 cores experiment.

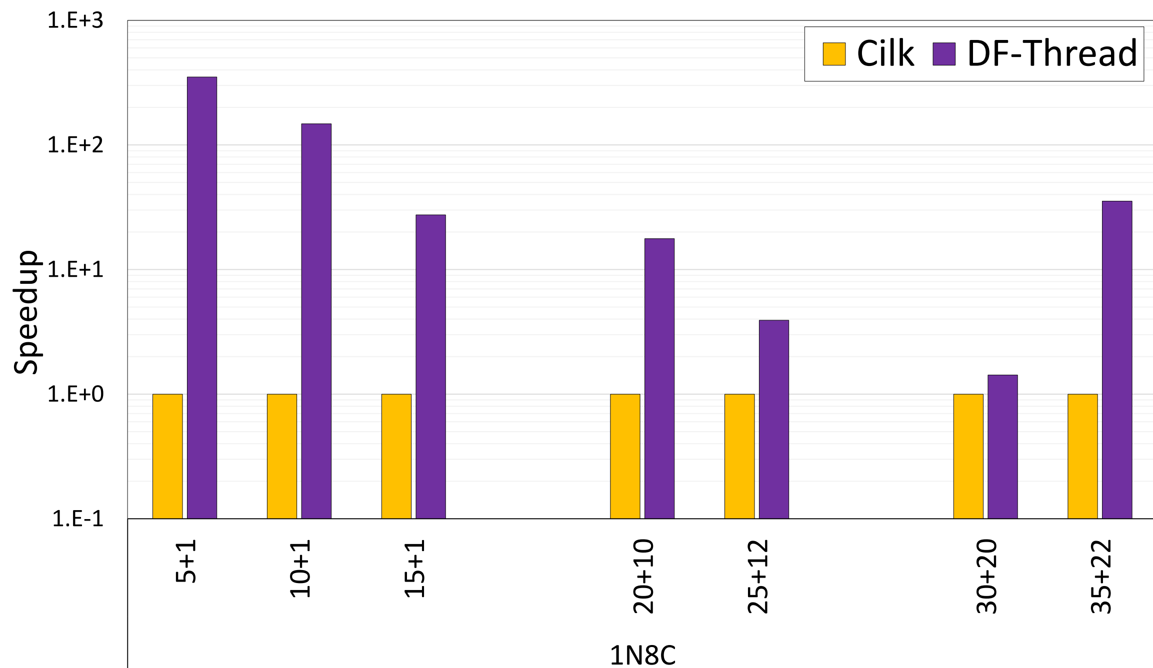


Fig. 3.7.: The scalability of DF-Threads against Cilk with the RFIB experiment on 8 cores experiment.

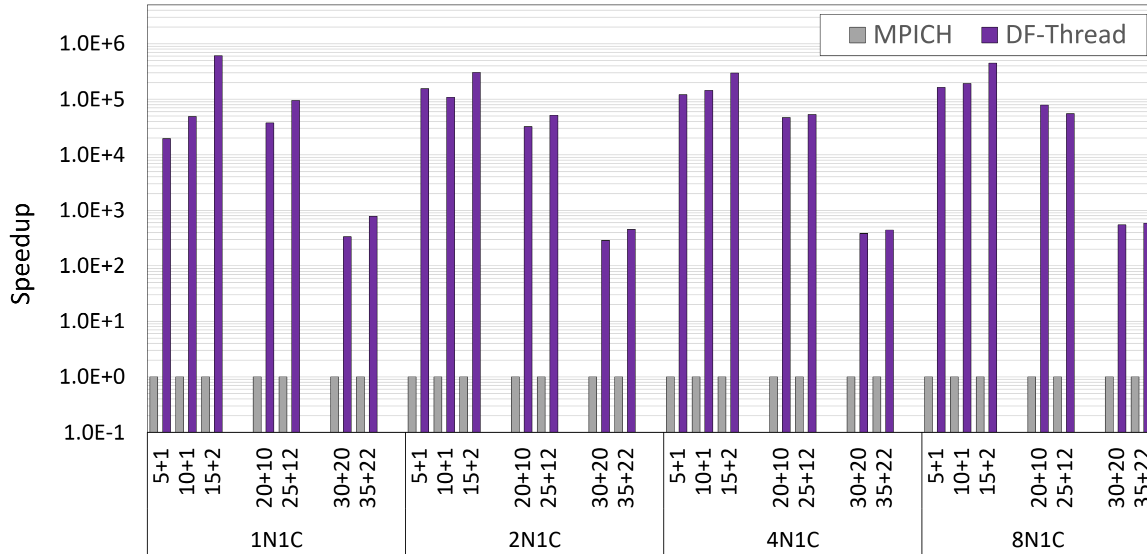


Fig. 3.8.: The RFIB multi-node experiment within DF-Thread and MPICH. Here we show how much is the Speedup of DF-Thread compared to MPI execution model.

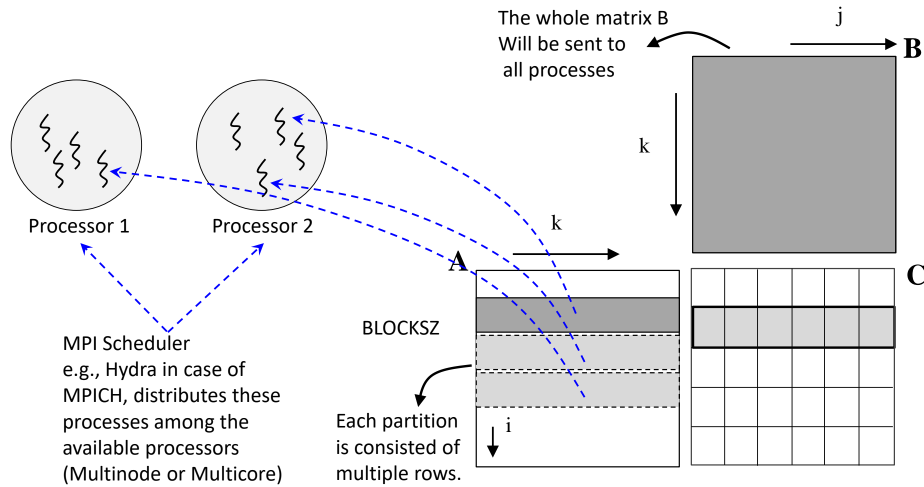


Fig. 3.9.: The Blocked Matrix Multiplication algorithm sketch that is used in our benchmark and its execution model.

3.6 Blocked Matrix Multiplication

Since the early 80s, Blocked Matrix Multiplication became a widely used kernel in many mathematics and computer science problems, and much research has been done to make this computation faster and more efficient. There are many solutions in the field of Mathematics and Computer Science to solve the Blocked Matrix Multiplication. One is partitioning the matrix into sub-matrices, vectors, or blocks. Here in our benchmark suite, we divide the matrix rows based on the size of the block (specify how many rows you want to partition) here we call is `BLOCKSZ`, and then the computation kernel will multiply this `BLOCKSZ×N` matrix to `N×N` matrices. Fig. 3.9 shows the sketch of the BMM algorithm.

BMM Sequential Execution

In this experiment we increase the problem size and execute the benchmark sequentially

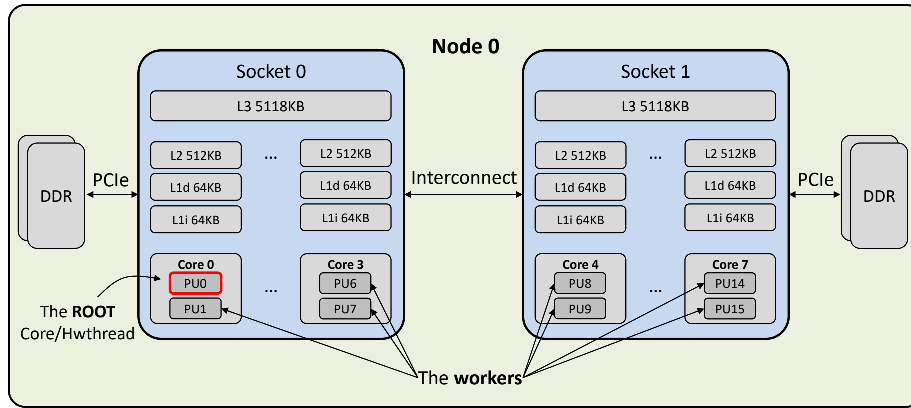


Fig. 3.10.: A sample structure of the hardware and how MPI processes are distributed throughout the hardware resources.

Tab. 3.11.: The sequential version of Blocked Matrix Multiplication on three platforms TFX2, Lab146 and COTSon simulator.

Matrix size	Execution time (ms)		
	TFX2	Lab146	COTSon
216+8	24	8	942.25
320+8	193	35	3167.07
400+8	226	58	6130.08
504+8	429	106	12160.56
640+8	3343	366	32020.2
808+8	7582	459	63436
1016+8	18896	3295	125933
1280+8	61867	4475	278347.66

on one core/hwthread. As expected, the execution time will be increased by increasing the problem size. The point is this execution time will be up to some hours if we just going to bigger numbers such as 8192. Even though for 1024 size it took some milliseconds to execute Matrix Multiplication, if we increase the problem size 2x times, the execution time will increase in order of $\mathcal{O}(n^3)$ times which in this case would be 8x times more time. The execution of BMM sequential of different hardware is shown in Table. 3.11.

BMM with Cilk

The first programming model that we demonstrate here is Cilk. The function code to explain the BMM using cilk programming model is shown in Listing 3.5. In this code, we use `cilk_spawn` to parallelize the matrix multiplication kernel among hardware resources. The Cilk compiler knows that a function call followed by the spawn keyword can be done asynchronously in a concurrent thread. The sync keyword makes the current thread wait for asynchronous function calls from the current context to finish before continuing. The Cilk keywords create a number of quirks to the C syntax. A Cilk function cannot be called using standard C calling conventions; instead, it must be called using spawn and then waited for with sync. Only a Cilk function can be used with the spawn keyword. In the context of a C function, the spawn keyword isn't allowed.

```
int matrixMultiply(DATA *a, DATA *b, DATA *c, int n, int n_local) {
    int i, j, k;
    DATA t;
    for(i=0; i<n_local; i++) {
        for (j=0; j<n; j++) {
            t = (DATA)0;
            for (k=0; k<n; k++) {
                t += a[i*n + k] * b[k*n + j];
            }
            c[i*n + j] = t;
        }
    }

    return 0;
}
int main(int argc, char argv[])
{
    //Matrix allocation, initialization, etc.
    for(int index=0; index<n; index+=BLOCKSZ)
    {
        int local_block=BLOCKSZ;
        cilk_spawn matrixMultiply(A, B, C, n, local_block);
    }
    cilk_sync;

}
```

Listing 3.5: BMM using cilk programming model function code

Tab. 3.12.: The Blocked Matrix Multiplication experiment on platform TFX2 with Cilk programming model, here we consider BLOCK size equal to 8 to set the granularity.

Matrix size	Execution time (ms)			
	1N1C	1N2C	1N4C	1N8C
216+8	162	86	52	35
320+8	539	278	144	83
400+8	1026	523	276	150
504+8	2035	1036	529	277
640+8	4993	2571	1267	653
808+8	9959	5096	2714	1395
1016+8	23159	12442	6469	3356
1280+8	51781	27525	14373	7567

BMM with MPI

One of the challenging benchmarks discussed here is the Blocked-Matrix Multiplication using MPI. MPI is the standard de-facto for the multi-node execution and there are many variants of the MPI compiler by different companies such as Intel MPI [83] and IBM Spectrum [81]. Note that MPI and its different versions and variants has dependency to the environment of the experiment. Tuning the parameters and environment are the most important factor to achieve a good result within MPI. In the following we discuss these right parameters and observed effects of bottlenecks or limitation with the MPI experiment.

Listing 3.6 shows the implementation of the BMM with MPI. In this implementation we used two-side communication method of MPI, which used `MPI_send()` and `MPI_Recv()` primitives. Note that in our algorithm we send the matrix B to all processes, this method has been proposed as a baseline implementation in [74]. Basically we partition the Matrix "A" into Size/BlockSZ number of rows, then we assign each of these to just one processes. For instance, if the Matrix A=64, and BlockSZ is 4, we have 16 partitions each of which has 4 rows. These 16 partitions will be assign to one process, therefore we create 16 MPI processes. Finally we execute these 16 processes on 1-2-4-8 cores or nodes, with the combinations of 1-N and N-1 (core-node).

To execute this experiment we assign each MPI process to one Core or Hwthread in Hypethreaded hardware as shown in Fig. 3.10.

Lessons learned with BMM

Blocked Matrix Multiplication is one of the most important benchmarks in the computer science literature. Matrix multiplication is one of the main kernels of AI and HPC Applications. BMM stress the memory operation capability of the execution model. Moreover, BMM is widely used to evaluate the memory and cache hierarchy performance of the system. We evaluate BMM over multi-core and multi-node platform using DF-Thread, Cilk and MPI programming models.

Table 3.13 shows the BMM evaluation on the TFX2 machine for multi-core experiment and comparison between DF-Thread, Cilk and MPI, and Table 3.14 shows the BMM evaluation on distributed machines in the Lab146 platform as multi-node experiment and comparison between DF-Thread and MPI.

```c
void compute()
{
  /* Send A by splitting it in row-wise parts */
  if (mpiRank == NODE_MASTER) {
      for (i=1; i<mpiSize; i++) {
          sizeToBeSent = n * getRowCount(n, i, mpiSize);
          MPI_Send(A + sizeSent, sizeToBeSent, MPI_DATATYPE, i, TAG_INIT,
                   MPI_COMM_WORLD);
          sizeSent += sizeToBeSent;
      }
  }
  else { /* Receive parts of A */
      MPI_Recv(A, n_ubound, MPI_DATATYPE, 0, TAG_INIT, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
  }
  MPI_Bcast(B, n*n, MPI_DATATYPE, 0, MPI_COMM_WORLD);
  matrixMultiply(A, B, C, n, n_local);
  /* Receive partial results from each WORKER */
  if (!mpiRank) {
      sizeSent = n_ubound;
      for (i=1; i<mpiSize; i++) {
          sizeToBeSent = n * getRowCount(n, i, mpiSize);
          MPI_Recv(C + sizeSent, sizeToBeSent, MPI_DATATYPE, i, TAG_RESULT,
                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          sizeSent += sizeToBeSent;
      }
  }
  else { /* Send partial results to ROOT */
      MPI_Send(C, n_ubound, MPI_DATATYPE, 0, TAG_RESULT, MPI_COMM_WORLD);
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
int matrixMultiply(uint64_t *a, uint64_t *b, uint64_t *c, int n, int
    n_local) {
    int i, j, k;
    for (i=0; i<n_local; i++) {
        for (j=0; j<n; j++) {
            for (k=0; k<n; k++) {
                c[i*n + j] += a[i*n + k] * b[k*n + j];
            }
        }
    }
    return 0;
}
int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {
    return (rowsTotal / mpiSize) + ((rowsTotal % mpiSize > mpiRank)?1:0);
}
```

Listing 3.6: BMM using MPI programming model function code

Tab. 3.13.: The Multi-core experiment execution time for some selected indexes of BMM on TFX2 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms) and the BLOCKSZ is equal to 8.

Number of processors	(216+8)			(320+8)			(400+8)			(504+8)			(640+8)		
	DF-Thread	MPICH	Cilk	DF-Thread	MPICH	Cilk	DF-Thread	MPICH	Cilk	DF-Thread	MPICH	Cilk	DF-thread	MPICH	Cilk
1	363.5	2371	162	1503.9	4634	539	2950.5	6807	1026	5591.3	12046	2035	30543.5	21962	4993
2	231.4	1058	86	890.8	2779	278	1751.1	5135	523	3630.7	7749	1036	14982.3	12072	2571
4	133.5	654	52	558.6	1256	144	1278.1	1953	276	2775.3	3457	529	8765.6	5446	1267
8	73.9	478	35	344.2	1036	83	882.3	1407	150	2058.8	1807	277	5445.8	4378	653

Tab. 3.14.: The Multi-node experiment execution time for some selected indexes of BMM on TFX2 machine is presented. In this experiment, the DF-Thread, MPICH and cilk are listed. Note that the execution time is normalized based on the clock frequency and reported value is the average value of 10 repetitions in the loop and is in milliseconds (ms). Each node in this experiment has 1 Core and the BLOCKSZ is equal to 8.

Number of nodes	(216+8)		(320+8)		(400+8)		(504+8)		(640+8)	
	DF-Thread	MPICH	DF-Thread	MPICH	DF-Thread	MPICH	DF-Thread	MPICH	DF-thread	MPICH
1	363.5	2371	1503.9	4634	2950.5	6807	5591.3	12046	30543.5	21962
2	231.4	1277	890.8	2893	1751.1	4715	3630.7	8667	14982.3	16891
4	133.5	533	558.6	1302	1278.1	2259	2775.3	4206	8765.6	8080
8	73.9	228	344.2	1279	882.3	2120	2058.8	3432	5445.8	7595

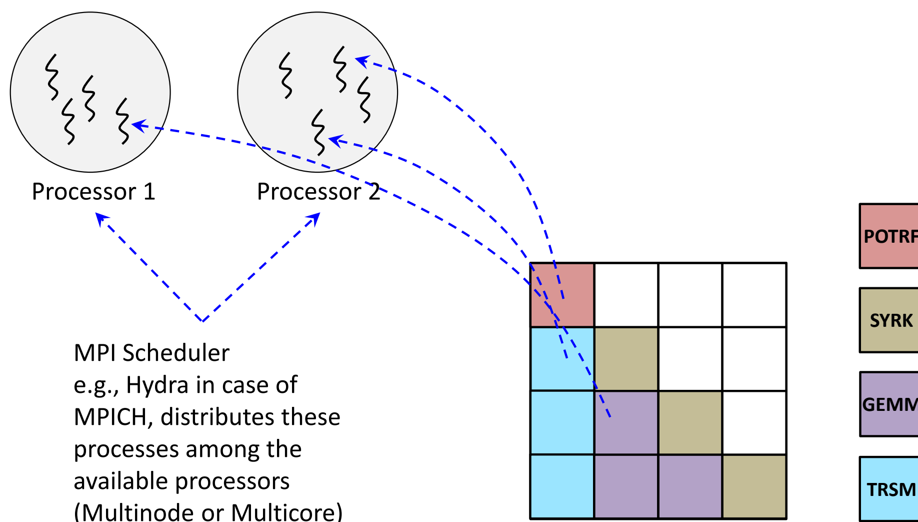


Fig. 3.11.: Scheme of the Blocked Cholesky and how kernels are distributed throughout the hardware resources.

3.7 Blocked Cholesky Factorization

In tile-based dense linear cholesky factorization, simply, Blocked Cholesky Factorization, a matrix A with dimension of N × N is split into NT × NT tiles, where each tile is of size A/NT = BlockSZ . This essentially divides one matrix factorization into many smaller matrix factorization, some of which have to be executed sequentially and some can be executed concurrently. Here, each of the factorizations performed on a tile is considered a kernel or task. By varying the tile size we can easily tune the parallelism and the overhead to execute a task. A classic, widely used and easy to understand example of a tile-based algorithm is the Cholesky Factorization [23]. Cholesky is a dense linear algebra algorithm which calculates the lower triangular matrix L of a symmetric positive definite matrix A , such that $A = LL^T$. This factorization has four different types of kernel/tasks: `syrk`, `gemm`, `potf2`, and `trsm`, which are successively applied on the trailing sub-matrix at each step of the algorithm. This procedure is depicted in the Fig. 3.11. The brief explanation of the description of each kernel is as follows:

1) `syrk`: is a symmetric rank-k update, which updates to the diagonal tile of the input matrix.

2) `gemm`: is a matrix-matrix multiplication, used to update tiles in trailing matrix.

3) `potrf`: performs an untiled version of Cholesky factorization of a diagonal tile of the input matrix and overrides it with the final elements of the output matrix.

4) `trsm`: is a triangular system solve, which applies transformation computed by POTRF to an off-diagonal tile below the diagonal tile operated by the last POTRF of the same column.

Blocked Cholesky Sequential Execution

Tab. 3.15.: The sequential version of Blocked Cholesky on three platforms TFX2, Lab146 and COTSon simulator.

Matrix size	Execution time (ms)		
	TFX2	Lab146	COTSon
1088+4	24	8	942.25
2112+4	193	35	3167.07
3008+4	226	58	6130.08
4032+4	429	106	12160.56

Tab. 3.16.: The Blocked Cholesky experiment on platform TFX2 with Cilk programming model, here we consider BLOCK size equal to 4 to set the granularity.

Matrix size	Execution time (ms)			
	1N1C	1N2C	1N4C	1N8C
1088+4	1252	764	495	267
2112+4	10098	6032	3376	1821
3008+4	31921	17537	9742	5244
4032+4	80023	41856	23516	12391

In this experiment we increase the problem size and execute the benchmark sequentially on one core/hwthread. We keep the sequential version of experiment as the reference to compared with other literature studies. To select an appropriate input size for blocked Cholesky we consider several facts. First, since we just compute the lower triangular part of the matrix, therefore the computation workload will be smaller than BMM. We have to ensure there is sufficient workload for the benchmark evaluation, therefore we choose bigger number for matrix sizes. Second, for the parallel programming models implementations, since the matrix will be divided by tiles, and then each tile will be assigned to a processor, the chosen size for matrix must be divisible to tile size and then number of processors.

The implemented blocked cholesky code is shown in Listing 3.7 and the sequential execution evaluation summarizes in Table 3.15.

Blocked Cholesky with Cilk

Cilk schedules processes using the work-stealing concept rather than the work-sharing approach. When a thread is scheduled to execute in parallel whenever the runtime performs an asynchronous function call, this is known as work-sharing.

The Blocked Cholesky written with Cilk programming model is listed in Listing 3.8 and the evaluation of the experiment is summarized in Table 3.17.

Blocked Cholesky with MPI

The execution model of Blocked Cholesky using MPI produces many processes based on the tile size given to the program, and perform mathematical operation based on this partitioned tiles over the whole matrix. The point is this mathematical kernel are dependant to each other and communicate during the execution. Therefore, one important item to

```
void potrf(float * const A, long ts, long ld)
{
    static int INFO;
    static const char L = 'L';
    spotrf_(&L, &ts, A, &ld, &INFO);
}
void trsm(float *A, float *B, long ts, long ld)
{
    static char LO = 'L', TR = 'T', NU = 'N', RI = 'R';
    static float DONE = 1.0;
    strsm_(&RI, &LO, &TR, &NU, &ts, &ts, &DONE, A, &ld, B, &ld );
}
void syrk(float *A, float *B, long ts, long ld)
{
    static char LO = 'L', NT = 'N';
    static float DONE = 1.0, DMONE = -1.0;
    ssyrk_(&LO, &NT, &ts, &ts, &DMONE, A, &ld, &DONE, B, &ld );
}
void gemm(float *A, float *B, float *C, long ts, long ld)
{
    static const char TR = 'T', NT = 'N';
    static float DONE = 1.0, DMONE = -1.0;
    sgemm_(&NT, &TR, &ts, &ts, &ts, &DMONE, A, &ld, B, &ld, &DONE, C, &ld);
}

void cholesky_blocked(const int ts, const int nt, float* Ah[nt][nt])
{
    int i, j, k;
    for (k = 0; k < nt; k++) {
        // Diagonal Block factorization
        potrf (Ah[k][k], ts, ts);
        // Triangular systems
        for (i = k + 1; i < nt; i++) {
            trsm (Ah[k][k], Ah[k][i], ts, ts);}
        // Update trailing matrix
        for (i = k + 1; i < nt; i++) {
          for (j = k + 1; j < i; j++) {
              gemm (Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);}
            syrk (Ah[k][i], Ah[i][i], ts, ts);
        }
    }
}
```

Listing 3.7: Blocked Cholesky sequential version function code

```
void potrf(float * A, long ts, long ld)
{
    static int INFO;
    static const char L = 'L';
    spotrf_(&L, &ts, A, &ld, &INFO);
}
void trsm(float *A, float *B, long ts, long ld)
{
    static char LO = 'L', TR = 'T', NU = 'N', RI = 'R';
    static float DONE = 1.0;
    strsm_(&RI, &LO, &TR, &NU, &ts, &ts, &DONE, A, &ld, B, &ld );
}
void syrk(float *A, float *B, long ts, long ld)
{
    static char LO = 'L', NT = 'N';
    static float DONE = 1.0, DMONE = -1.0;
    ssyrk_(&LO, &NT, &ts, &ts, &DMONE, A, &ld, &DONE, B, &ld );
}
void gemm(float *A, float *B, float *C, long ts, long ld)
{
    static const char TR = 'T', NT = 'N';
    static float DONE = 1.0, DMONE = -1.0;
    sgemm_(&NT, &TR, &ts, &ts, &ts, &DMONE, A, &ld, B, &ld, &DONE, C, &ld);
}
void compute(int ts, int nt, float* Ah[nt][nt]){
        for (int k = 0; k < nt; k++) {
                // Diagonal Block factorization
            potrf (Ah[k][k], ts, ts);
            cilk_for (int i = k + 1; i < nt; i++) {
            trsm (Ah[k][k], Ah[k][i], ts, ts);
            }
            cilk_for (int i = k + 1; i < nt; i++) {
                    cilk_for (int j = k + 1; j < i; j++) {
                    gemm (Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);
                    }
                    syrk (Ah[k][i], Ah[i][i], ts, ts);
                }
        }
}
```

Listing 3.8: Blocked Cholesky using Cilk programming model function code

Tab. 3.17.: The Blocked Cholesky experiment on platform TFX2 with MPI programming model, here we consider tile size 4 to set the granularity.

Matrix size	Execution time (ms)			
	1N1C	1N2C	1N4C	1N8C
1088+4	1707	1205	838	515
2112+4	13216	9128	6287	3725
3008+4	38862	28384	18807	11099
4032+4	95457	69404	45604	27517

```
unit64_t *histo, *colors;

void histo_seq(unit64_t *_histogram, unit64_t *_colors,  uint64_t size)
{
        if (size == 1) {
                _histogram[*_colors]++;
        }
        else {
                histo_seq(_histogram, _colors, size/2);
                histo_seq(_histogram, _colors + size/2, size - size/2);
        }
}
void compute()
{
        histo_seq(histo,colors, CSIZE);
}
```

Listing 3.9: Histogram sequential version function code

stress the execution model is evaluated by using Blocked Cholesky apllication. The Blocked Cholesky evaluation of the experiment using MPI programming model is summarized in Table 3.17.

3.8 Histogram

The first step in creating a histogram is to "bin" (or "bucket") the range of values, which means dividing the entire range into a series of intervals and counting how many values fall into each interval. Bins are often defined as non-overlapping, sequential periods of a variable. The bins (intervals) must be next to each other and are typically (but not always) of the same size. Even though the time complexity of the histogram algorithm is not heavy, however, the data conflict to match data in the appropriate bin or bucket is massive. This lightweight benchmark is suitable to stress the memory policies and widely used in data race conditions in transactional memory literature studies.

Histogram Sequential Execution

The sequential implementation of the histogram benchmark is shown in Listing 3.9. The sequential execution evaluation summarizes in Table. 3.18.

Tab. 3.18.: The sequential version of Histogram on three platforms TFX2, Lab146 and COTSon simulator. We consider "bin" size equal to 4 for this measurement.

Input size	Execution time (ms)		
	TFX2	Lab146	COTSon
2^{16}+4	0.77	0.14	8.69
2^{18}+4	2.69	0.559	36.05
2^{20}+4	8.82	2.3	143.9
2^{22}+4	34.7	9.02	567.1

Tab. 3.19.: The Histogram experiment on platform TFX2 with Cilk programming model, here we consider "bin"" size equal to 4.

Input size	Execution time (ms)			
	1N1C	1N2C	1N4C	1N8C
2^{16}+4	10	14	10	10
2^{18}+4	43	42	34	28
2^{20}+4	169	162	126	100
2^{22}+4	676	780	511	404

Histogram with Cilk

The single most prominent reason that parallel computing is not widely deployed today is because of race conditions. Identifying and debugging race conditions in parallel code is hard. Once a race condition has been found, no methodology currents exists to write a regression test to ensure that the bug is not reintroduced during future development. We developed the cilk implementation of the Histogram benchmark shown in the Listing 3.10. The evaluation of the experiment is summarized in Table 3.10.

Histogram with MPI

The implementation of the Histogram benchmark using MPI programming model is shows in the Listing 3.11. The evaluation of the experiment is summarized in Table 3.11.

Lessons learned with Histogram

There are several insights regarding the histogram evaluation. It is an interesting benchmark through our evaluation since it produces a high level of data conflict. Within smaller size of

Tab. 3.20.: The Histogram experiment on platform TFX2 with MPI programming model, here we consider "bin"" size equal to 4.

Input size	Execution time (ms)			
	1N1C	1N2C	1N4C	1N8C
2^{16}+4	0.7	0.6	1.19	0.9
2^{18}+4	1.7	2.1	2.17	2.1
2^{20}+4	8.1	6.2	5.69	6.6
2^{22}+4	29.16	23.49	31.2	22.46

```
uint64_t *histogram, *color;
typedef unsigned int uint;
typedef unsigned char uchar;
#define Cilk_lockvar pthread_mutex_t
#define Cilk_lock pthread_mutex_lock
#define Cilk_unlock pthread_mutex_unlock
void prepare(){
        color = malloc(CSIZE * sizeof(uint64_t));
        histogram = malloc(BSIZE * sizeof(uint64_t));
        lock  = (pthread_mutex_t *) calloc(BSIZE,sizeof(pthread_mutex_t));
        for (int i = 0; i < CSIZE; ++i)
        {
                color[i] = (unit64_t)rand()%BSIZE;
        }
}
void histo_cilk(unit64_t *histogram, unit64_t *color, uint64_t size)
{
        if (size == 1) {
                Cilk_lock(&lock[*color]);
                histogram[*color]++;
                Cilk_unlock(&lock[*color]);
        }
        else {
                cilk_spawn histo_cilk(histogram, color, size/2);
                histo_cilk(histogram, color + size/2, size - size/2);
                cilk_sync;
        }
}
void compute(){
        cilk_spawn histo_cilk(histogram, color, CSIZE);
}
```

Listing 3.10: Histogram with Cilk programming model function code

```
void hist_mpi(){
    uint64_t ssize = (CSIZE - 1) /nw + 1;
    uint64_t  histogram_private[BSIZE];
    for(int i=0; i<BSIZE; i++){
            histogram_private[i] = 0;
    }
    slice = malloc(ssize * sizeof(DATA));
    MPI_Status Stat;
      if (wid == MASTER) { // Distribute the work
          for (int i= 0; i<ssize; ++i) slice[i] = color[i]; // Assign 1st
    slice to master
          for(int w=1; w<nw; ++w) // Send the other slices to the slaves
              MPI_Send(color + w*ssize, ssize, MPI_INT, w, TAG_GENERAL,
    MPI_COMM_WORLD);
      } else { // slave
        int dataWaitingFlag; // Wait until a message is there to be
    received
        do MPI_Iprobe(MASTER, TAG_GENERAL, MPI_COMM_WORLD, &dataWaitingFlag
    , MPI_STATUS_IGNORE);
        while (!dataWaitingFlag);
               MPI_Recv(slice, ssize, MPI_INT, MASTER, TAG_GENERAL,
    MPI_COMM_WORLD, &Stat);
        }
        // Processing data
      for (int i= 0; i<ssize; ++i) histogram_private[slice[i]]++; free(
    slice);
      if (wid == MASTER) { // Process the partial results
          int w, done = 0;
          // Accumulate the result
          for (int i= 0; i<BSIZE; ++i) histogram[i] += histogram_private[i
    ];
          do {// Get partial histograms from slaves
                for (w=1; w<nw; ++w) { // round robin check
                    int dataWaitingFlag;
                    MPI_Iprobe(w, TAG_GENERAL, MPI_COMM_WORLD, &
    dataWaitingFlag, MPI_STATUS_IGNORE);
                    if (dataWaitingFlag) { // Get the message
                          MPI_Recv(histogram_private, BSIZE, MPI_INT, w,
    TAG_GENERAL, MPI_COMM_WORLD, &Stat);
                          ++done;
                          // Accumulate the result
                          for (int i= 0; i<BSIZE; ++i) histogram[i] +=
    histogram_private[i];
                          }
                    }
            } while (done < nw - 1);
        } else // slave: send back the partial result
              MPI_Send(histogram_private, BSIZE, MPI_INT, MASTER,
    TAG_GENERAL, MPI_COMM_WORLD);
}
```

Listing 3.11: Histogram with MPI programming model function code

the bin, the biggest data conflict will be occurred. Data races can decrease the performance significantly since the memory policy on parallel programming models locks the portion of memory that needs to be write/read with each thread or process. For instance, Cilk programming model shows significant performance degradation [32]. As shown in the implementation in Listing 3.10, `pthread_mutex_lock` and `pthread_mutex_lock` are POSIX definitions to protect data races in the memory. These policies will be a speed-bump of the performance. However, in MPI the situation is better since each MPI process has its own memory and Therefore there is no need to put memory lock to control memory races. Finally in DF-Thread execution model, we control the memory races by guards, these guards are defined in such a way to make a protection without degrading the performance in DF-Thread memory model.

3.9 Conclusion

We present a quantitative approach to build a strong baseline for our execution model. In this chapter, we develop and 4 more important benchmarks in the literature. We evaluate our experimental baseline with Recursive Fibonacci (RFIB), Blocked Matrix Multiplication (BMM), Blocked Cholesky and Histogram. Each of mentioned benchmarks has important feature to stress the execution model. We ensure using the same algorithm implemented for all the benchmarks to follow an apple-apple comparison. Then we choose different hardware platform to experiment the multi-core and multi-node evaluation. To do this, we choose TFX2 for multi-core evaluation and Lab146 for Multi-node. The results evidence that DF-Thread has an important potential compared to Cilk and MPI programming model to run over hardware resources. The evaluation of the benchmarks express the benefit of using DF-Thread is promising to be more faster and exploiting underlying hardware better than aforementioned ones, where the Cilk and MPI programming models are highly optimized and they have compiler support.

4 Dataflow Runtime

> *The greatest glory in living lies not in never falling,*
> *but in rising every time we fall.*
>
> — **Nelson Mandela**
> Former President of South Africa

4.1 DRT: A Lightweight Dataflow Runtime To Debug and Develop Dataflow Programs

Future computers may take advantage of a dataflow program execution model (PXM) for both performance and energy advantages. One key element to provide a compilation toolchain for such machines is a framework for developing initial benchmarks. DRT (Dataflow Runtime) is a tool that enables the fast prototyping of those benchmarks for the Dataflow Threads (DF-Threads) PXM. In this work, we show how to use DRT to develop dataflow based examples to be targeted by a future compiler for the dataflow PXM.

DRT has been written in portable C code (tested with the GNU C compiler), and it is open-source, therefore, it can be used on real machines based on architectures like x86, AArch, RISC-V ISA.

Here, we discuss some didactic examples, and we show how to study and debug the data exchange, which is flowing through frames that are detached from the data stack. We compare DRT against similar dataflow runtime libraries such as DARTS and OCR. Even though our environment is not yet optimized, we found that DRT outperforms the above runtime frameworks in terms of execution time. We also give an evaluation of the time and complexity to develop DF-Threads examples in DRT compared to the approach of using a full system simulator and FPGAs for more accurate modeling. Even though dataflow models have shown many great features, conventional programming languages do not support them very well [173]. This limitation, together with the possible high performance gains, motivated us to introduce a tool, which could reduce the gap between conventional languages and dataflow execution models.

4.1.1 Background

In this work, we present the Dataflow Runtime tool (DRT) to quickly develop and test the execution of dataflow codes based on DF-Threads API. Our contributions in this work are:

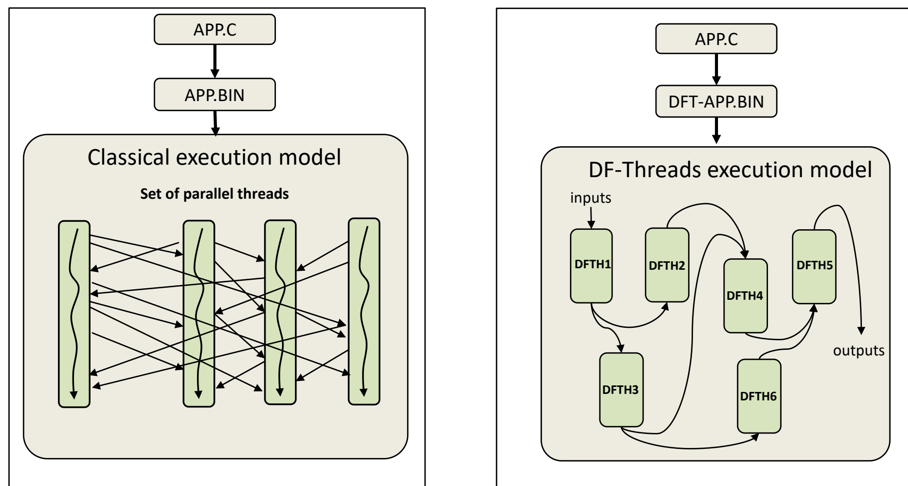


Fig. 4.1.: Simplified representation of the DF-Threads execution model. On the left, we represent the irregular read and write of generic threads. On the right, the exchange of data among threads happens in a more regular fashion [88].

- Introducing a dataflow runtime (DRT), which is presented first in this work.
- Illustrating how the DRT tool can be used for debugging and studying the movement of data frames (a feature that is not available in standard debuggers).
- Comparing the execution time speedup of DRT against similar dataflow runtime.

In 4.1, we show a simplified high-level overview of DF-Threads execution (right) and a classical (von Neumann style) execution (left). In the classical execution, the parallel threads can read/write from/to any location of the memory. Therefore, a high synchronization and coherency overhead may be generated. As mentioned in detail in [61], each of these DF-Threads has a different behavior according to the memory access pattern. Consequently, it may need different execution and hardware support. It is worth recalling that using standard libraries like *Pthreads* is not required. Here, we briefly recall the specification of the DF-Threads API:

- DF-Threads follow the dataflow semantics: a thread is ready when its input is fully available; it starts executing when the scheduler decides to assign it to a physical resource (e.g., a core).
- The management of a DF-Thread lifetime happens through the following functions, which are described in Table 4.1: ***df_schedule***, ***df_ldframe***, ***df_write***, ***df_destroy***.
- DF-Threads are isolated in terms of memory accesses, and their execution can be repeated in the case of faults since their inputs are retained [166].

Tab. 4.1.: DF-Threads function definitions [65]

DF-Threads API function	Description
uint64_t df_schedule(void* ip, uint64_t sc)	Create the DF-Thread and its associated frame; *sc* is the synchronization count, which represents the number of inputs that the DF-Thread will receive.
uint64_t df_ldframe()	Retrieve the frame pointer associated with the current DF-Thread.
uint64_t df_write(void* fp, uint64_t val)	The value *val* will be stored in a location pointed by *fp*, and for each write, the *sc* (which is specified by scheduler before) will be decremented.
uint64_t df_destroy()	Terminate the current DF-Thread and deallocate its input frame.

4.1.2 Writing dataflow codes with the DF-Threads API

This Section shows the workflow to map the desired application into a dataflow code (here DF-Threads). While this translation could be done by a compiler, we do not have such a compiler at the moment (the compiler could be future work).

We use fine-grain algorithms to show the potentiality of our tool in mapping several DF-Threads on real architectures. We choose the Recursive Fibonacci (RFIB) as a “simple yet complex enough” example to illustrate the development methodology for DF-Threads programs. The RFIB algorithm is a well-known example used to create many threads and stress the runtime and the scheduling management.

In 4.2, we describe the original C code and its mapping into DF-Threads, together with the dynamic behavior of the dataflow code. In this case, two DF-Threads are created: RFIB and “adder".

The key operation is the **df_schedule**, which creates a DF-Thread, whose code is specified by the parameter *ip* (the instruction pointer or the name of the corresponding function). With the same operation, a portion of memory (*frame*) is allocated and associated with the same DF-Thread. The size of the frame is determined by the number of inputs of the DF-Thread that is specified by the *sc* value of the **df_schedule**. The **df_schedule** returns the address (frame pointer) to the allocated memory space (the *frame*). The next step is to write the DF-Thread input and the output locations. This can be done by using the **df_write**. Once the frame pointer (*fp*) has been retrieved by the **df_ldframe**, the **df_write** will store the data (here `n-1`, `n-2`) in the location of *fp*[1] and *fp*[2], respectively. Please note that *fp*[0] has been reserved as the output location, into which the DF-Thread will write the result. For each write into the frame, the *sc* value will be decremented by 1 (this is implied by **df_write** and it is part of the implementation of the **df_write** itself).

In the end, **df_destroy** will terminate the current DF-Thread [66].

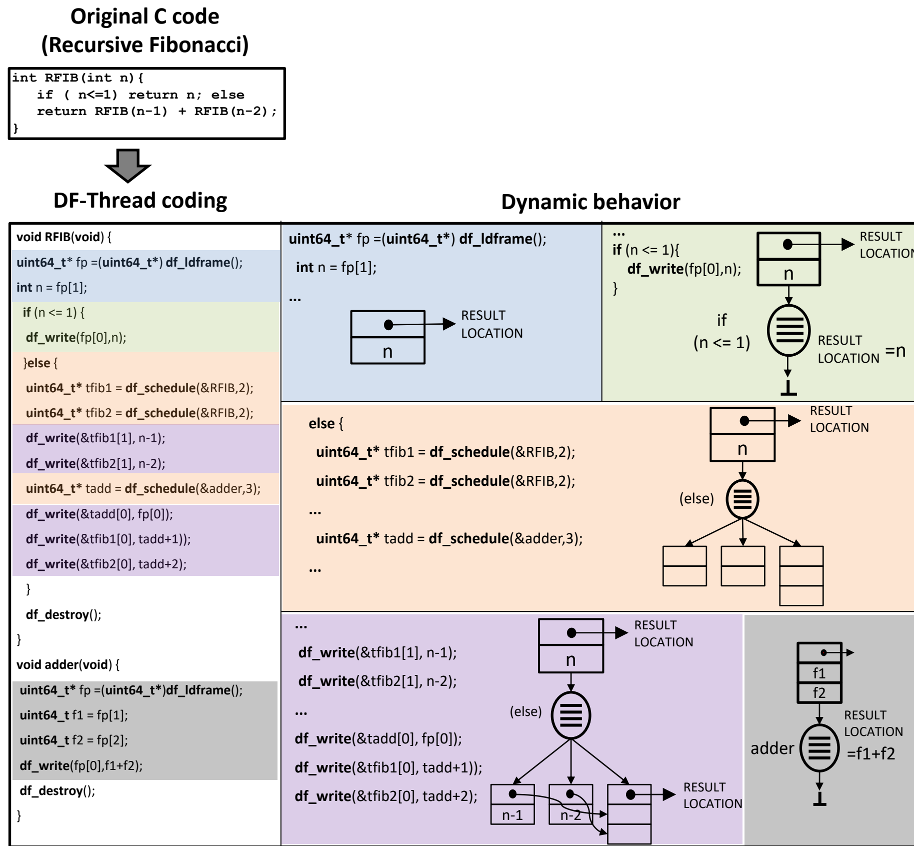


Fig. 4.2.: Illustrating the operations of the basic DRT API functions with a simple Recursive Fibonacci (RFIB) example. On the left, there is the representation of the RFIB function and its coding in DF-Thread style. On the right, we detail the specific dynamic behavior. Example rearranged from [104].

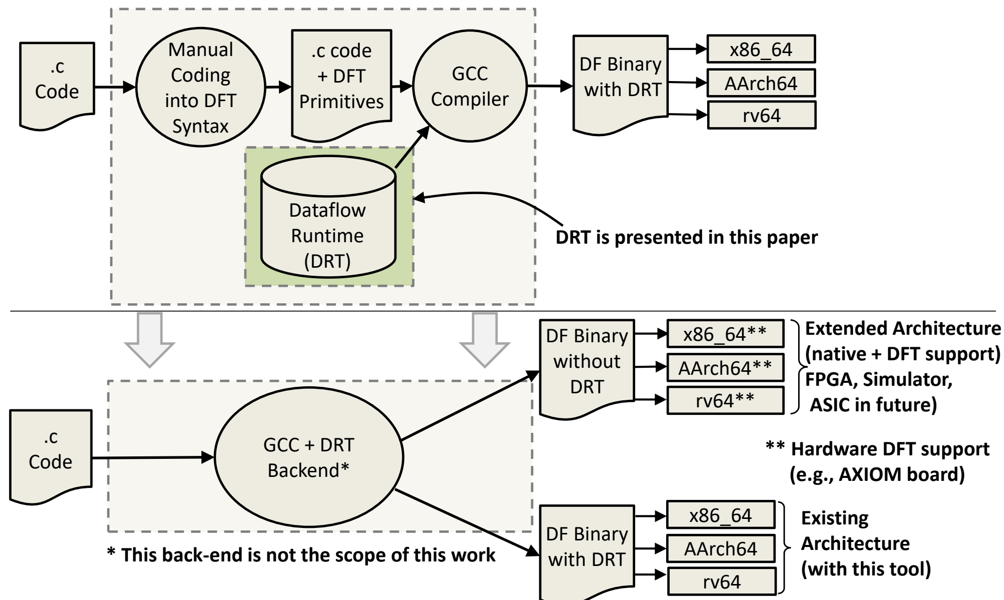


Fig. 4.3.: The role of DRT in developing applications based on the DataFlow Threads (DFT) execution model. In the top part, we show the current setup of DRT. In the bottom part, we show the production framework that we envision. The idea is that DRT could help develop a future DRT backend of a standard compiler.

4.1.3 Introducing DRT

Developing a novel architecture may require considerable time when using an architectural simulator [9, 62]. To reduce this development-cycle time, in the case of the dataflow execution model, we designed a tool that we call "Dataflow Run-Time" (DRT). The aim of the DRT is to make it easier for the software community to use a dataflow program execution model (here DF-Threads): by studying the simple examples that we propose, or building new examples, the compiler experts could derive an appropriate compilation path, which could target the DF-Threads PXM.

This tool is compatible with real machines like x86, AArch, RISC-V. DRT only requires the installation of the GCC compiler for compiling and running DF-Threads programs.

DRT enables the fast development and debugging of the DF-Threads' API and its data exchange mechanism, which is based on *frames* (see 4.2).

According to an initial test done in DRT, we can reduce the development-cycle time from minutes/hours to seconds (see Section 4.3).

As shown in Fig.4.3, we currently need to map manually ('manual coding into DFT syntax') high level programs ('.c code') to the DF-Threads API. Then, the DRT enables a standard compiler (GCC in our case) to generate a binary that can run on standard architectures. The availability of DRT provides a basis for direct writing dataflow codes but also enables compiler experts to further build on this workflow and integrate it in a compiler (lower part of Fig.4.3, which is not in the scope of this work).

Similar efforts exist like the Delaware Adaptive Runtime System (DARTS) [183] and the Open Community Runtime (OCR) [104], so we compare them with DRT in Section 4.3. DRT is available as open-source at `http://drt.sourceforge.net` [1].

In Fig.4.4, as an illustrative example for analyzing the benchmark behavior, we show the output of DRT when the debug level is set to three for the RFIB benchmark and its input is n=4. The first line describes the command line for executing a dataflow code with DRT. In the third line, the DRT initializes the environment and allocates the memory space for storing the frames based on the application requirements. Lines 4 and 5 show the creation of the scheduled function (the RFIB function, see Table 4.2) and the `report` function to collect the results. In lines 6 and 7, the *df_write* writes the value (*val*) in the output frame and decrements the associated synchronization count (*sc*).

Lines 10 to 19 describes the recursive calls of the RFIB functions. Finally, the current DF-Thread will be terminated, and its input frame will be deallocated (line 20).

The list of *ip* and *fp* addresses that are shown in Fig.4.4 correspond to the same addresses that can be retrieved through standard disassembler tools (e.g., objdump). However, the usage of such tools gives us only a static view, while DRT enables a dynamic analysis

[1] Checkout the DRT repository by this command: svn co https://svn.code.sf.net/p/drt/code/

```
1  ~/drt-code $ DRT_DEBUG=3 ./RFIB 4
2  computing Recursive Fibonacci(4)
3  -DRT: FRAME-MEM allocation+initialization done.
4  TS: fi=0  ip=0x403a46  fp=0x609f60  sc=1/1
5  TS: fi=1  ip=0x401795  fp=0x609fc0  sc=2/2
6  TW: fi=1  ip=0x401795  fp=0x609fc0 val=0x609f6000  sc=1/2
7  TW: fi=1  ip=0x401795  fp=0x609fc0 val=0x4  sc=0/2
8  ++main
9  -DRT: Starting Dataflow launcher.
10 TE: fi=1 ipnew=0x401795  fpnew=0x609fc0
11 TS: fi=2  ip=0x401795  fp=0x60a020  sc=2/2
12 TS: fi=3  ip=0x401795  fp=0x60a080  sc=2/2
13 TW: fi=2  ip=0x401795  fp=0x60a020 val=0x3  sc=1/2
14 TW: fi=3  ip=0x401795  fp=0x60a080 val=0x2  sc=1/2
15 TS: fi=4  ip=0x400d81  fp=0x60a0e0  sc=3/3
16 TW: fi=4  ip=0x400d81  fp=0x60a0e0 val=0x609f6000  sc=2/3
17 TW: fi=2  ip=0x401795  fp=0x60a020 val=0x60a0e001  sc=0/2
18 TW: fi=3  ip=0x401795  fp=0x60a080 val=0x60a0e002  sc=0/2
19 TD: fi=1  ip=0x401795  fp=0x609fc0  sc=2
20 TE: fi=2 ipnew=0x401795  fpnew=0x60a020
21 ++report
22 DF-Thread RFIB = 3
23 *** SUCCESS ***
```

Fig. 4.4.: DRT sample output. DRT_DEBUG is an environment variable for specifying the debug level. The DF-Threads functions are mapped to internal operations where TS stands for thread scheduling, TE stands for thread-end, TD stands for thread drop, TW stands for thread write, *ip* stands for instruction pointer, and *fp* stands for frame pointer. Other debugging information is *fi* for frame index, *sc* stands for synchronization count, *ipnew/fpnew* are the *ip/fp* just freed.

```
void df_write(uint64_t *fp, uint64_t val)
{
    *fp=val;           //write the value
    uint64_t *md=METADATA(fp); //retrieve metadata
    md[MDSC]--;        //decrement synchronization count
    if (md[MDSC] == 0) //move the frame to READY QUEUE
       TSETREADY(md[MDQSTATUS]);
}
```

Fig. 4.5.: An example of a modeled function in the DRT implementation, where METADATA extracts the metadata pointer from the frame, MDSC is the offset of the synchronization count, and MDQSTATUS is the offset of the status bits that indicate whether the frame is in ready or waiting status.

Tab. 4.2.: The function name and its corresponding frame pointer address that are shown in Fig.4.4 (same as in objdump tool).

Frame pointer address	Corresponding function
0x401795	RFIB
0x400d81	adder
0x403a46	report

showing the entire sequence of executed instructions with additional information about the DF-Threads, memory, and queue status.

For example, the *ip*=0x401795 corresponds to the address of the code of the RFIB function (see Table 4.2). All the corresponding functions and their *fp* addresses generated in the function RFIB are shown in Table 4.2.

In order to show the effectiveness of the internal modeling of the DRT function, we consider the implementation of the *df_write* function (see Fig.4.5). The *df_write* needs two arguments, the pointer to the output *frame* (*fp*) and the value to write in such *frame*. Internally, the *df_write* extracts the metadata pointer from the given *frame* and, based on the *sc* information, *df_write* decides whether the DF-Thread is in ready or waiting status. Other useful debugging information, not shown in this simple example for the sake of simplicity, are the status of queues, the total number of allocated frames, the total number of writes, total number of frames that are in ready or waiting status.

4.1.4 Evaluation

In this Section, we compare the performance of DRT against other similar environments, namely OCR [104] and DARTS [183]. OCR and DARTS use a dataflow model to manage threads, similarly to DRT: the common main idea is to decouple the higher layers of the software stack from the underlying hardware by using a possibly universal interface. For details about OCR, DARTS, and other related environments.

In this work we wrote some initial benchmarks manually due to the lack of a compiler. Therefore, at this stage, we cannot afford to make more extensive tests with large benchmarks.

To demonstrate the capabilities of the DRT, we selected two simple benchmarks:

- Recursive Fibonacci (RFIB) in order to generate a high number of threads easily.
- Blocked Matrix Multiplication (BMM) as it is a very commonly used kernel in many applications (especially in Artificial Intelligence, Deep Neural Networks, etc.), and it moves much data around.

The two benchmarks are using the same exact algorithm for all three frameworks. The output of the benchmarks is validated against the output produced with other independent tools executing the same benchmarks.

For the sake of simplicity, we analyze the sensitivity with the input set by using n=10, 15, 20, 25 for RFIB and s=128, 256, 512, 1024 for BMM, where n is the index of the corresponding Fibonacci number and s is the size of the square matrices that are multiplied. For the block size of the matrices, we used b=8, where b is the number of the elements inside a block.

The purpose of DRT is to explore the correctness of the dataflow execution, not to scale the performance across cores. Nevertheless, to make a fair comparison against other environments, we restricted our evaluation to a single core execution.

For each of the three runtime frameworks (DRT, DARTS, and OCR), we measure the time spent in the Region Of Interest (ROI) of each benchmark, and we repeat at least ten times the experiments to obtain statistically valid measurements. We report the execution time speedup by using DARTS as the baseline. As we can observe from Fig.4.6 and Fig.4.7, DRT can outperform by one order of magnitude DARTS for smaller inputs. DRT outperforms OCR by a factor of about 13x for n=25.

While the OCR and DARTS are well optimized, DRT can still be improved. However, as stated before the main goal of DRT is just to provide a tool for developing DF-Thread benchmarks and a future compiler; more performance could be achieved by using DF-Thread native support as shown in Fig.4.3.

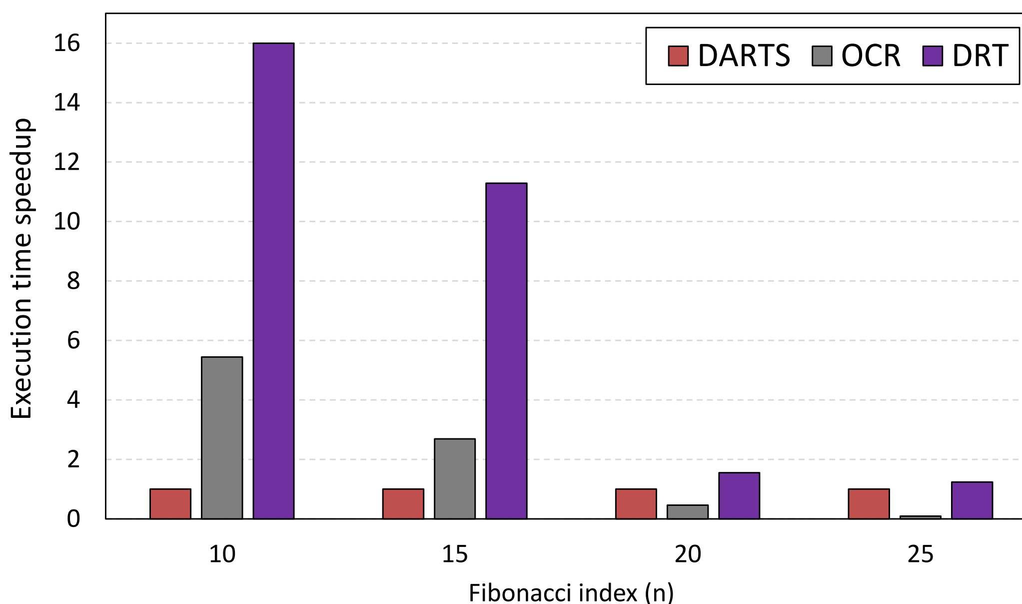


Fig. 4.6.: RFIB execution time speedup comparison between DRT, DARTS and OCR runtime. Here OCR is the baseline. DRT reaches better performance due to a simplified management of the dataflow execution.

While it is possible to develop DF-Threads codes on a simulator or on an FPGA prototype, we found that it is more productive to use a tool such as the DRT, a minimalistic API written in around 300 lines of C code, through which it is possible to test and debug the

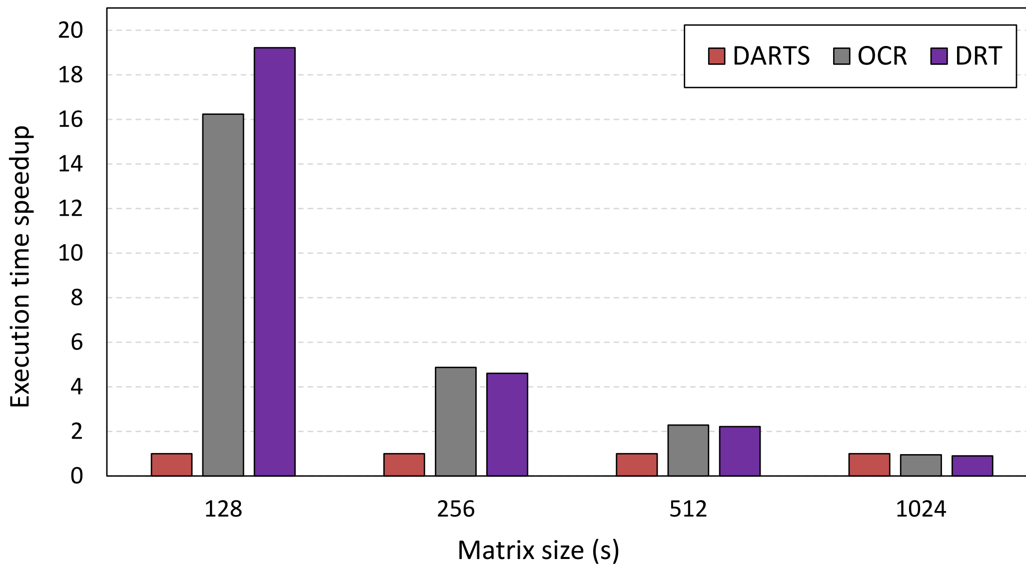


Fig. 4.7.: Blocked Matrix Multiplication execution time speedup comparison between DRT and DARTS and OCR, with the DARTS as baseline. While for larger Matrix sizes the execution time tends to be the same for three tools, it is important to note that during the development-cycle, we typically use smaller inputs. So, the shorter execution time of DRT during tests helps focus on the development.

Tab. 4.3.: Comparing lightweight DRT with other tools for developing dataflow codes and the related architectures. As we can see DRT, is using only 300 lines of C code.

	DRT	Simulator[94]	FPGA[123]
SLOC of the framework	∼ 300	∼ 112,000	∼ 1,000,000
Openness the development framework	High (open-source)	Medium (partly open-source)	Limited (proprietary tools)
Complexity of the development-cycle	Low (seconds)	High (minutes)	Very high (hours)

implementation of a specific feature in seconds, while doing that on an FPGA may require days [64] (see Table 4.3). In Fig.4.8, we show the simulation time of the COTSon simulator compared to the DRT. As we can see, we can obtain up to four orders of magnitude speedup while executing a benchmark RFIB. The speedup in simulation time of a simulator is lower compared to an FPGA, but the development-cycle time can be much higher; this is discussed below.

In terms of evaluating the DRT in relation to other approaches for developing the initial codes that use the dataflow execution model, we compare other tools for modeling new architectures like the simulator and the FPGA prototype in Table 4.3. The usage of these tools is necessary when exploring hardware support for the dataflow execution [62, 64].

We considered the following metrics:

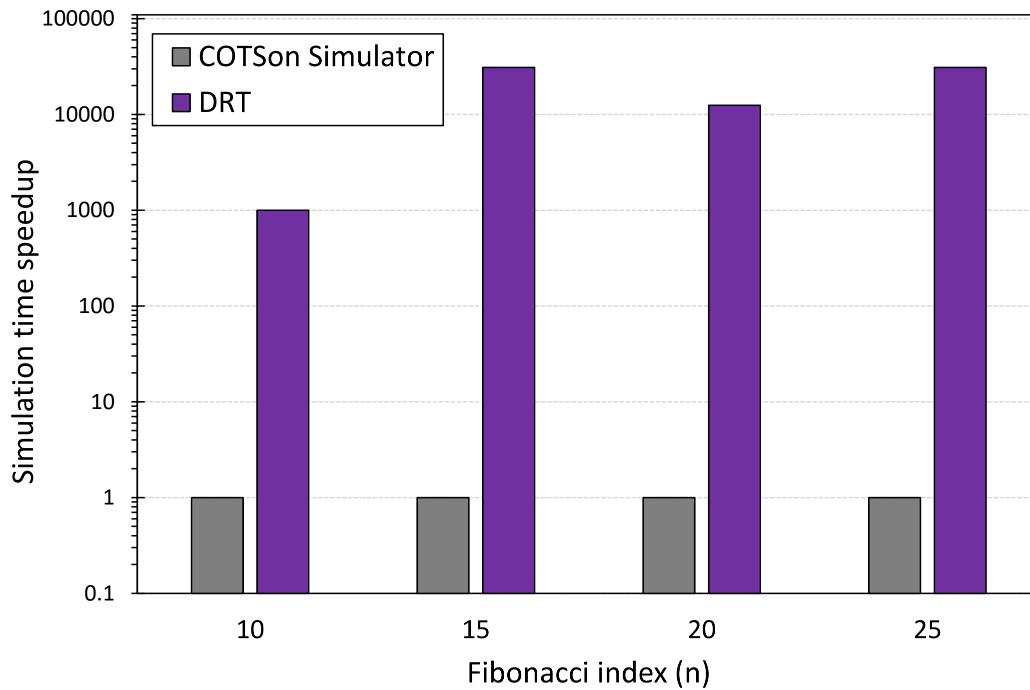


Fig. 4.8.: Simulation time speedup comparison between DRT and the COTSon simulator by using the RFIB example. DRT significantly decreases the development-cycle time to develop a dataflow program.

- SLOC[2]: these are the source lines of code of the corresponding framework; these numbers are all publicly available; for the simulator, we referred to the COTSon simulator [94], and for the FPGA, we referred to the software stack of the AXIOM board [123].
- Openness of the development framework: while DRT can be downloaded and installed in seconds, COTSon requires at least some hours to complete the setup and some days to become familiar with the modeling of the components; moreover, some parts of the code (AMD SimNow) are not open-source; regarding the FPGA-board, the software stack is open-source, but the tools are typically proprietary and may require licensing and complex setup procedures.
- The complexity of the development-cycle: while it is rather simple to make modifications, test, and debug a program through the DRT tool, it may require minutes to complete a full simulation in the COTSon simulator, and it may require hours to modify and re-generate a full design in the FPGA framework [62].

[2]Source lines of code

5 Proposal for a Distributed Large Scale Graph Processing on multi-FPGA Platform

> *A man who dares to waste one hour of time has not discovered the value of life.*
>
> — **Charles Darwin**
> 1802-1889

In the last year of the Smart Computing Ph.D. program, based on the ”borse Pegaso ciclo 34” rules, I had this opportunity to spend 6 months abroad Italy to study and collaborate with a foreign university according to my Ph.D. goals. Therefore, I took this opportunity to collaborate with the ”Custom Computing Research Group at Imperial College London” and collaborate with Prof. Wayne Luk and Prof. Gaydadjiev as a visiting researcher. In this period, we defined a project based on the Reconfigurable Graph Processing Model that addresses the current challenges and is suitable to cover the Ph.D. goals. This study can bring possibilities of large scale graph processing to the user and have many advantages despite a few current ones in the literature. We are progressing with the implementation in our laboratory.

5.1 Introduction

Processing large-scale graphs is a challenging concept due to the nature of the computation that causes irregular memory accesses. Managing such irregular accesses may cause significant performance degradation on both CPUs and GPUs. Thus, recent research trends propose graph processing acceleration with FPGAs. Moreover, in the case of large-scale graph processing, one major problem is that the graph does not fit into the limited amount of on-chip memory resources available on a modern FPGA. Due to the limited capacity of device memory, data would be transferred to and from the FPGA multiple times during the computation, and this transfer between on-chip and off-chip memory would be greater than the computation time. To maximize performance, it is necessary to overlap, hide and customize the data transfers to the highest degree so that the FPGA accelerator is always fully loaded. A possible way to overcome the limited resources on one FPGA accelerator is to develop a distributed architecture on a multi-FPGA platform using an efficient partitioning scheme. An efficient partitioning scheme aims to increase data locality and minimize communication between the partitions. This work uses an offline partitioning method to support the distributed large-scale graph processing concept. Our architecture

uses Hadoop at the higher level to map a graph to the underlying hardware. The higher layer of computation is responsible for gathering the blocks of data pre-processed and stored on the host's file system and distributing them to a lower layer of computation made of FPGAs. In this work, we show how graph partitioning combined with a multi-FPGA architecture will lead to high performance without limitation on the size of the graph, even when the graph has trillions of vertices. Our performance analysis, in the case of PageRank, forecasts performance improvement of up to 20 times and a cost-normalized improvement of up to 12 times when comparing the proposed approach on one Xilinx Alveo U250 FPGA accelerator against a state-of-the-art baseline graph processing software implementation on a Intel Xeon server CPU with a 40-core processor at 2.50 GHz.

Traditionally Graphs have been studied within a variety of subjects. The structure of the graphs made graph processing one of the most challenging applications among computer scientists [171, 136, 114, 17, 112, 49]. In this context, computing a very large scale graph is more challenging first and foremost because the whole graph size can not fit into the most modern accelerator's memory. Plus, the structure of the graphs is irregular and needs a high amount of random and irregular access to the memory. These factors contribute to a lack of data locality and the inability to achieve a high degree of parallelism. The literature study to elaborate on these challenges using modern architecture is bulk, and many researchers have used hardware accelerators to mitigate one of the following problems including modern FPGA accelerators.

Graph Size: In contrast to CPUs and GPUs, which use a cache hierarchy memory model, FPGAs have on-chip Block RAMs (BRAM). One of the most valuable features of BRAMs is their ability to provide high-throughput random access to memory [53, 178]. However, there are still some difficulties. The major problem is that the size of BRAMs in a single FPGA is far smaller than the graph's size or less than the size of the last level of cache on a powerful CPU. One alternative is to use multi-FPGA boards to increase resources, but this has the drawback of limiting connectivity between boards. However, the dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations [150].

Data Locality: The composition of most graphs is irregular, which means that a small portion of the entire graph will have access to the most significant portion and have a considerable impact on the total. As a result, in the vast majority of cases, data locality is a significant problem. Moreover, graphs are usually created based on a natural phenomenon, from Social Networks to Biological structures. Subsequently, the distribution of data in these graphs follows the Power Law distributions, complicating the locality of data during the computation [100, 37].

Irregular Data Access Pattern: In unstructured graphs, the data access pattern is entirely irregular. As a result, each data access to a irregular position would dramatically reduce the efficiency of overall computations [180, 177, 39].

Data Conflict: It's not uncommon for vertices from different locations to read/write to the same vertex simultaneously. As a result, a significant amount of data conflict must

Tab. 5.1.: The cloud service cost based on Amazon cloud cost calculator for FPGA f1 instance cloud servers.

Instance Name	FPGAs	vCPUs	Instance Memory (GiB)	SSD Storage (GB)	On-Demand Price/hr
f1.2xlarge	1	8	122	470	$1.65
f1.4xlarge	2	16	244	940	$3.30
f1.16xlarge	8	64	976	4 x 940	$13.20

Tab. 5.2.: The cloud service cost based on Amazon cloud cost calculator for CPU instance cloud servers.

Instance Name	vCPU	Memory	On-Demand Price/hr
m6gd.xlarge	4	16 GiB	$0.1808
m6gd.2xlarge	8	32 GiB	$0.3616
m6gd.4xlarge	16	64 GiB	$0.7232
m6gd.8xlarge	32	128 GiB	$1.4464
m6gd.12xlarge	48	192 GiB	$2.1696

be managed using memory model policies such as memory locks and atomic memory operations [179].

After discussing about the most significant features of Graphs and the challenges, now the question is, **why we choose "Distributed FPGA" as a hardware structure for this purpose?**

To elaborate answering this question we show an example summarizing cost of using CPU, FPGA and GPU resources. As an example, we target amazon AWS platform in this case. As can be seen in Table 5.1 and Table 5.2, the cost of different instances of Amazon FPGA and CPU cloud servers are listed. Assuming, FPGA is **10x** faster than CPU to compute a specific target. Subsequently, given an example of the execution time to compute a specific algorithm on 10 CPU instances is about 600 hours, on 100 CPUs would be 60 hours, and on 10 FPGA instances will be 6 hours.

Therefore, in this case based on the cost listed in the Amazon AWS cloud servers choosing the best CPU servers (m6gd.12xlarge with 192 GiB Memory) costs $216, whereas FPGA (f1.16xlarge with 4 x 940 GiB Memory) will cost $79 for this case. This brief example shows at the end FPGA cloud server will be almost 3 times cheaper than CPU instances, if the implemented algorithm has a good level of speedup against CPU, which is important factor to be considered as a big motivation and courage developing FPGA designs. This level of speedup against CPU is trivial for FPGAs and this increasingly is improving since FPGA is getting popularity among scientists.

5.2 Background and Motivation

The scale of recently proposed graph processing methods with a Multi-FPGA environment is limited. For instance, in very recent studies on FPGA such as ThunderGP [30, 39, 137], the size of the evaluated graph dataset is not very large datasets, and mostly are in the mid-range graph processing scale. This scale does not satisfy the need to use accelerators and the considerable development time on FPGAs rather than CPU and state-of-the-art algorithms to compute such graphs sizes. So, the **early motivation** of our work is to increase the scale of the graph, which has been eagerly looked for in very recent published papers [14, 39, 137].

The **second motivation** is the integration of a High-level platform such as Hadoop to deploy the distributed platform on top of the underlying hardware. The reason to choose Hadoop is that the graph construction is a data-parallel problem. Hadoop (MapReduce) is well-suited for this task. Moreover, Hadoop is a highly scalable storage platform since it can store and distribute enormous data sets across hundreds of inexpensive servers that operate in parallel.

As we know, Graph size is increasing rapidly and will be in order of Peta Bytes in the near future. Consequently, this amount of data will not be fitted into the memory, and at this point, the challenges of Graph Processing begins to find the method to partition the graph before feeding it into the computing system and memory hierarchy. Finding a partitioning algorithm suitable for distributed computing on FPGAs is the point.

The **third motivation** of this work is to emphasize this hardware as an underlying infrastructure for a cloud computing basis. As discussed in Section 5.1, one of our motivations is to provide a cost-effective infrastructure for cloud computing applications. The proposed method using FPGA must be fast enough rather than state-of-the-art CPU algorithms to aim this. There were some bottlenecks to achieving this goal, such as longer development cycle time and more difficulty developing FPGA programs. On the other hand, high-level language developments on FPGA using HLS is growing quickly and efficiently. This ease the development cycle time on FGPA to be much less and much flexible than before. In this work, we use Xilinx Vitis HLS [150]. The Xilinx Vitis HLS (previously Xilinx Vivado HLS) tool is a free High-Level Synthesis (HLS) tool created by Xilinx. Vitis HLS makes it simple to write complicated FPGA-based algorithms with C/C++ code. It can handle a wide range of data types (floating points, fixed points, etc.) as well as arithmetic functions. It also supports AXI4-Stream, allowing data to be readily exchanged with other IPs.

5.3 Related Work

To handle medium graphs with a systematic framework, the studies propose a system that leverages the edge-centric processing model for graph algorithms and the GAS (Global Address Space) paradigm to address the FPGA chip's memory space limitation. It employs a

portion of the on-board DRAM as an update buffer, where it temporarily stores intermediate processing results. The DRAM-based update buffer, on the other hand, creates a substantial I/O overheads, lowering graph processing performance.

ForeGraph [40] provides a system that makes advantage of the on-board DRAM grid representation of graphs and distributes the FPGA logic resources into many pipelines. Each pipeline has two BRAM-based vertex buffers (the L1 cache) that preserve the source and destination vertex intervals. The vertex intervals of contiguous edge blocks are first loaded into the vertex buffers attached to the pipelines during processing, and then the edges of these edge blocks are fed into the FPGA chip to be processed in parallel. The pipelines interface directly with the on-board DRAM to exchange vertex data in this technique, resulting in small pipeline delays and improving graph processing performance.

FabGraph [137] offers a second level (L2) of vertex cache that periodically stores vertex intervals to enhance pipeline efficiency and graph processing speed. However, when working with extremely sparse real-world graphs, the processing of streaming edges cannot keep up with the communication between the two cache levels in FabGraphs.

FPGP [38] uses the grid format to describe a huge input graph and stores the graph's vertex and edge data in on-board and host DRAM, respectively. The edges are sent into the FPGA through the host bus and processed during computation. The works [182] propose a method for processing huge graphs with an FPGA-based accelerator, in which graph data is immediately sent to the FPGA chip processing unit. According to the technique of directly exchanging data between the host DRAM and the FPGA chip, the bandwidth of the host bus determines the performance of graph processing, with low bandwidth resulting in poor graph processing and low utilisation rates of the FPGA chip's resources. Only two pipelines are constructed in FPGP when the host PCIe Interface has a bandwidth of 0.8 GB/s.

Other research, apart from the above systems that use FPGA-based accelerators to process small, medium, and big graphs, cover certain intriguing aspects. GraphOps [114], for example, presents a modular way to building graph accelerators in FPGA written in MAXJ on Maxeler platform. GraphGen [112] turns the input graph into an instruction stream that may be processed by pipelines designed using an FPGA board's logic resources.

GravF-M [48] provides a redesigned architecture from their previous work [49], that minimises communication across the inter-FPGA network considerably. *Although network bandwidth is the limiting factor for multi-FPGA performance on most systems, this can lead to a potential increase in overall system performance*. A three-stage programming technique that allows for this optimization while still giving the user freedom and making superstep synchronisation easier. Low-overhead partitioning techniques improve load balancing among PEs and FPGAs. The programming model for GravF-M is Migen, a Python-based tool to export Verilog codes to be synthesize with conventional tools such as Vivado.

In [14] the authors introduced a large-scale graph processing on single FPGA. They implemented the proposed work in Chisel [15] and synthesized using Vivado design suite. The

evaluation were conducted on Amazon AWS f1 instances, which include a Virtex UltraScale+ FPGA linked to the host PC through PCI express and four 16 GB DDR4 channels. The contribution of the work is to eliminate cache misses and exploit the multi-die feature of signle FPGA.

ThunderGP [30] provides an automated graph processing user interface. It is applicable for the user to automate the process of the processing with the desired application. ThunderGP uses efficient methods to compute the appropriate number of kernels while staying within the platform's memory bandwidth and fitting the kernels into SLRs. ThunderGP, in particular, bundles Scatter PEs, a shuffle, and Gather PEs into a kernel group known as a "scatter-gather" kernel group since they are all in the same pipeline. It, on the other hand, places Apply PEs in a separate kernel group called an apply kernel group. Each memory channel on multi-SLR platforms with multiple memory channels has one scatter-gather kernel group that buffers and processes the same set of destination vertices independently.

We briefly discussed about the most similar attempt to our methods, however, Table 5.3 shows a brief taxonomy of the selected studies, which are more close to our work.

Tab. 5.3.: Brief overview of the most related recent studies on FPGA accelerators and their features compared to this work.

Work	Distributed[1]	Language[2]	Implementation[3]	Evaluation Size[4]	Public[5]	FPGA Platform	Year
ForeGraph [40]	✓	HDL	Simulation	Medium	✗	Xilinx VCU110	2017
FabGraph [137]	✗	HLS	Simulation	Medium	✗	Xilinx VCU110 and VCU118	2019
HitGraph [179]	✗	HDL	Hardware	Small	✓	Xilinx Virtex US+	2019
ThunderGP [30]	✗	HLS	Hardware	Medium	✓	Alveo Family	2021
GraVF-M [48]	✓	Python [5]	Hardware	Medium	✓	Micron Pico se-6 platform	2019
GridGas [182]	✓	HDL	Hardware	Medium	✗	Xilinx Kintex	2018
FPGP [37]	✗	HDL	Hardware	Medium	✗	Xilinx Virtex-7	2016
Ref. [14]	✗	Chisel	Hardware	Large	✓	Xilinx Virtex US+ (AWS Platform)	2021
GraphOps [114]	✗	MAXJ	Hardware	Small	✗	Maxeler Boards	2016
This Work	✓	HLS/C++	Hardware	Very Large	✓	Alveo Family	2022

[1] Weather the algorithm is distributed or not.
[2] Which programming language is used.
[3] Weather the implementation is based on Software Simulation or on a Real Hardware.
[4] What is the scale of the evaluation graph dataset presented in the work.
[5] Weather the work is open-source and available for public.
[6] Migen, a Python-based tool to export Verilog codes to be synthesize with conventional tools such as Vivado.

Graph Partitioning

We need Edge partitioning, which provides chunk-based partition of edges for a certain vertex and resides the graph data on the host filesystem. Chunks (or blocks) should not relate to each other by any means, and the size of each block should be fit into the size of BRAM of the target FPGA. Blocks will be read sequentially from memory by the Kernel, and updated values will be written back to the memory. Graph partitioning is a well-known problem in graph computing literature. Many works addressed novel techniques, and algorithms for graph partitioning [181, 125, 162, 93]. We concluded with the Grid partitioning [181]. This approach can give us a high data locality, avoid data conflict, and can be mapped on the BRAM resources on the FPGA. The Scalability of Grid partitioning is also high since the data can be mapped exactly on different resources.

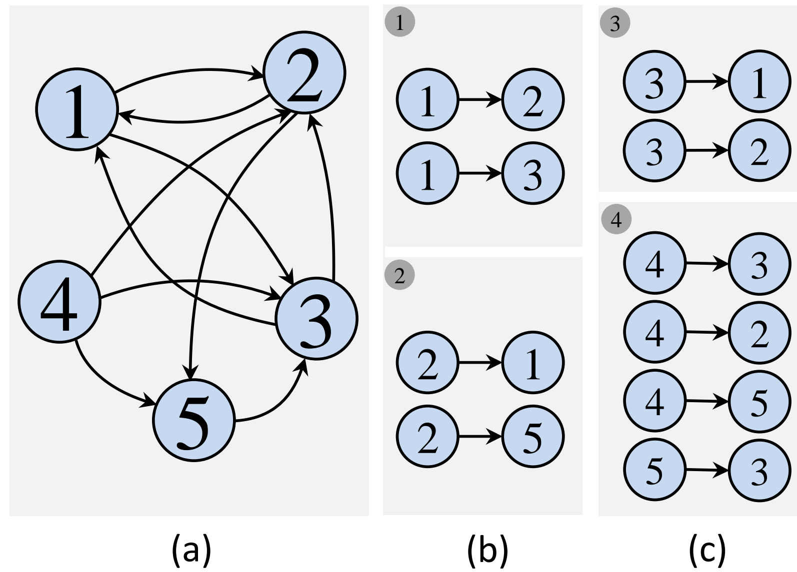


Fig. 5.1.: Graph partitioning scheme used in our system preprocessing method. Figure (a) shows a sample graph, Figure (b) and Figure (c) show the partitioned edge into 2 chunks which inside each chunk there are 2 blocks therefore in total we have 4 blocks of edge which is shown in each block.

Table 5.4 summarizes the most useful graph partitioning we discovered in the literature. Fig.5.1 gives an example graph, whose vertex set is partitioned into four equal-length subsets, and its 2x2 grid representations. we can observe that a given $G = (V, E)$, will be partitioned into P^2 blocks (P is the numbr of partitions the user asks for) according to the source and destination vertices. Each edge is placed into a block using the following rule: the source vertex determines the row of the block, and the destination vertex determines the column of the block.

Distributed Graph Processing

As graph structures grow in size and complexity, they have already exceeded the computing and memory capabilities of the most recent single processors. Distributed Parallel processing seems to be required to overcome the resource restrictions of single processors in graph calculations, given the success of parallel computing in many fields of scientific computing. However, parallel graph computing is complicated because of many items we mentioned earlier in Section 5.1. In the recent decade, developers had to use establish distributed systems or create their own systems before introducing cloud computing and Hadoop, which demanded extra work to offer fault tolerance and handle other parallel processing issues. Nowadays is even more affordable to elaborate on distributed computing resources with the growth of cloud computing servers and their tools. At this moment, researchers already have a reliable tool to process massive data sets thanks to the MapReduce idea and Hadoop (its open-source implementation) on CPUs and then less on GPU. However, on FPGA resources and cloud computing, this is still an open space to research and investigate the potential of a multi-FPGA platform to get the advantage of FPGA features compared to CPU and GPU instances.

Hadoop Framework for Graph Processing

Tab. 5.4.: Most recent and well-known graph partitioning suitable for FPGA implementation.

Graph Partitioning Algorithm	**Methodology**	**Programming Language**	**Graph partitioning**	**Source code**	**Platform**	**Year**	**Adaptability to Hadoop**
GridGraph [181]	Grid Partition of Edges	C++	Store edge partition blocks on disk	Public	CPU	2015	No
Lumos [162]	Grid Partition of Edges Plus cross-iteration propagation values support bulk synchronous processing	C++	Store edge partitions as blocks on disk	Public	CPU	2020	No
FabGraph [136]	Grid Partition of Edges plus Hash partitioning to support power Law graphs	C++	Store partition blocks on disk	not Public	Multi-FPGA	2019	No
PowerGraph [71]	Vertex-cut partitioning	C++, Java, Scala	Partitioning during Runtime	Public	CPU	2013	Yes
ThunderGP [30]	Vertex-cut partitioning	HLS-C/C++	Partitioning during Runtime	Public	Single-FPGA	2021	No
Foregraph [40]	Shard-Interval	HDL	Partitioning during Runtime	not Public	Multi-FPGA	2017	No

Hadoop framework is a popular open-source, distributed platform and programming model for demanding big data analytics computations [8]. Hadoop map-reduce is a distributed system of computation elements that uses a parallel programming model to handle massive data. Data processing is decomposed into two primitives in this model: **1)** a map function that processes incoming data in key/value format in parallel and generates intermediate data pairs, and **2)** a reduce function that merges these pairs into meaningful outputs. In a map-reduce architecture, a user application launches a Master controller and a sequence of mappers and reducers distributed across several compute nodes. The root node coordinates the generation of mappers and reducers and keeps track of their progress. A `map()` function and a `reduce()` function are both included in a typical map-reduce application. Apache Hadoop is a Java-based software framework for cluster-based distributed storage and processing. In this work, the proposed architecture involves three stages: loading the input graph, Mapping the Compute resources/workers, and Reducing the worker's results. There is an iteration phase to reach a good level of error. These steps must iterate over specific iterations. The first stage loads the graph (e.g., Subgraph data 1, Subgraph data 2, etc.), which various workers will process, e.g., worker 0 on CPU core 0, worker 1, CPU core 1, etc. The second stage performs computation based on graph format and given condition and performs various iterations on subgraph, e.g., Subgraph data part 1 on FPGA 1, Subgraph data part 2 on FPGA 2. The high-level overview of the proposed Hadoop framework is depicted in Fig. 5.2.

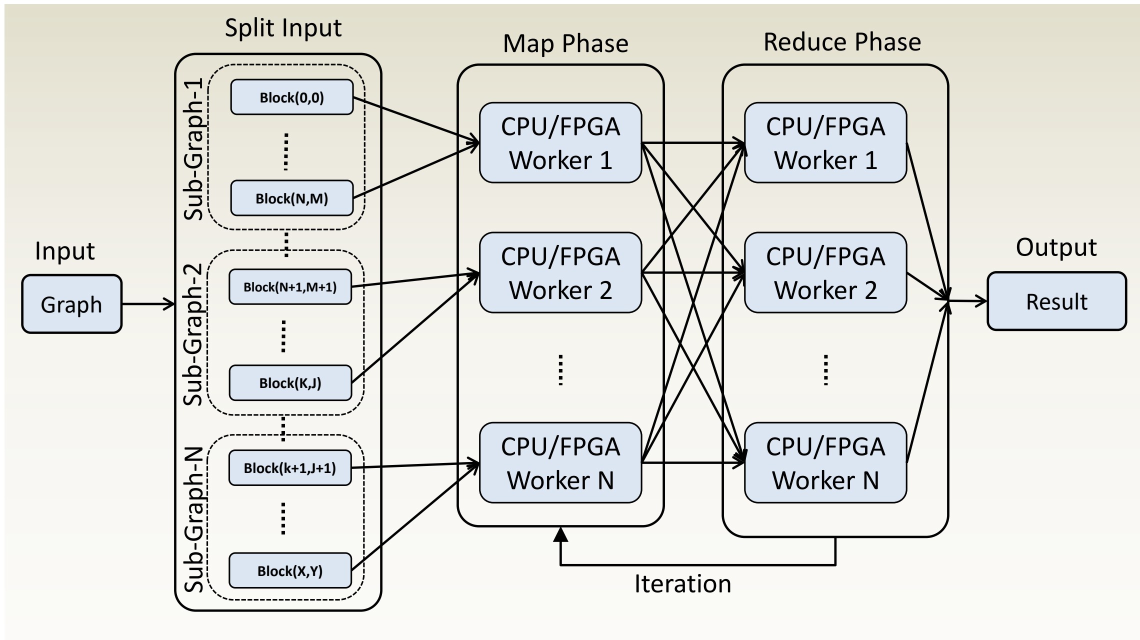


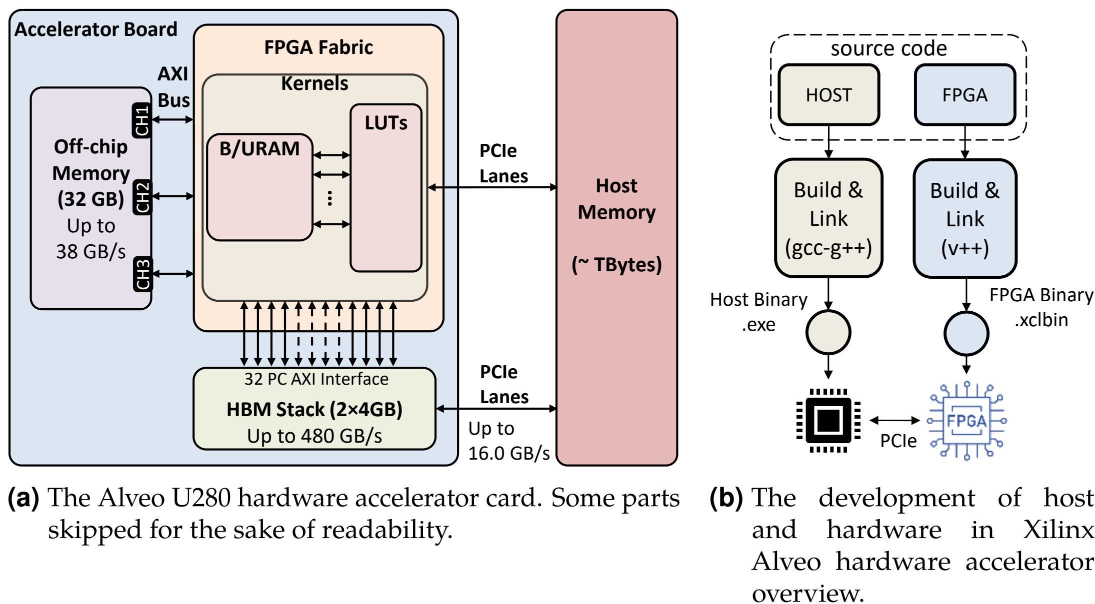
Fig. 5.2.: The Hadoop framework for distributed graph processing high-level overview.

5.4 Hardware Implementation

We use vivado HLS language to implement our proposed method. Our target design is Alveo boards from Xilinx . Fig.5.3a shows the Alveo U280 block diagram. We use vivado design suite (Vitis version 2021.1) program to develop and implement the design architecture. The implementation consists of two-phase, **Kernel** implementation and **Host** program. As shown in Fig.5.3b, the source code includes host and kernel C/C++ codes. The host code is responsible for implementing a software-based part of the design and will be executed on the CPU. Moreover, the host code is responsible for loading and driving the kernel code and executing it. On the other hand, kernel implementation is the hardware kernels that use accelerator resources and is written in Vivado HLS. Host code will be compiled with G++ compiler and Kernel source codes will be compiled with `V++` Vitis compiler with desired flags.

5.4.1 Host Program

As shown in Fig. 5.4 we divided the host program into a different section. The first section is to receive the user's graph and prepare the pre-process information and data. These include the graph blocks path, the number of vertices, the number of edges, and the number of partitions in a metadata file beside edges partitioned into blocks. Once the graph information is received, the host program starts to further process data and fetch blocks from the disk to the host RAM. This procedure is out of our measurement zone. Other studies in this field also consider data already existing in the host RAM and already prepared [137, 30]. The next step is to create **aligned** buffers to keep all data in the memory



(a) The Alveo U280 hardware accelerator card. Some parts skipped for the sake of readability.

(b) The development of host and hardware in Xilinx Alveo hardware accelerator overview.

Fig. 5.3.: The Alveo card harwdare accelerator hardware and its compilation framework.

aligned. Otherwise, Xilinx Runtime (XRT) will complain about the memory misalignment, and this will add extra **memcpy** during the runtime. To create aligned vectors, we use the function shown in Listing 5.1.

After creating an aligned vector and fetching all necessary information and data in Host RAM, we need to create the second stage, the bridge between host and kernels. This bridge consists of buffers. In Listing 5.2, we show how to Create the OpenCL buffers between host and kernel. In this snippet, `vector<cl::Buffer>` is an OpenCL namespace declaration of the type `Buffer` and `OCL_CHECK` is a macro definition to check the OpenCL functions declared return a successful value and does not have any error.

The last step is to instantiate the kernel(s) and set the kernel argument to be called within the host code (see Listing 5.3).

5.4.2 FPGA Kernels

This design decided to directly exchange data between Host Memory and Kernel local memory (array) using OpenCL functions. In this way, we have created small, efficient OpenCL buffers located between Host and Kernels. Graph data will be injected into these buffers and queued. The efficiency of the FPGA implementation is the main issue with the local arrays. To implement local arrays, we need memory on FPGA Fabric. FPGA can provide this memory as LUTs, BRAM blocks and Registers. However, these resources are not enough to cover one or a few big local arrays. It might necessitate a more extensive and more expensive FPGA chip. Using the DATAFLOW optimization and streaming the data through small, fast FIFOs helps minimize the usage of block RAM, but this requires the data to be consumed in a streaming sequential way and other complex optimizations.

```
template <typename T>
struct aligned_allocator
{
        using value_type = T;
        T* allocate(std::size_t num)
        {
                void* ptr = nullptr;
                if (posix_memalign(&ptr,4096,num*sizeof(T)))
                        throw std::bad_alloc();
                return reinterpret_cast<T*>(ptr);
        }
        void deallocate(T* p, std::size_t num)
        {
                free(p);
        }
};

int main() {
  ... //rest of the main function
  std::vector<uint32_t, aligned_allocator<uint32_t> > outdegree(vertices);
  std::vector<EdgeId, aligned_allocator<EdgeId> > src(graph.edges);
  std::vector<EdgeId, aligned_allocator<EdgeId> > dst(graph.edges);
  std::vector<EdgeId, aligned_allocator<EdgeId> > buffer_out(vertices);
  ... //rest of the main function
return 0;
}
```

Listing 5.1: The function declaration of creating aligned vectors.

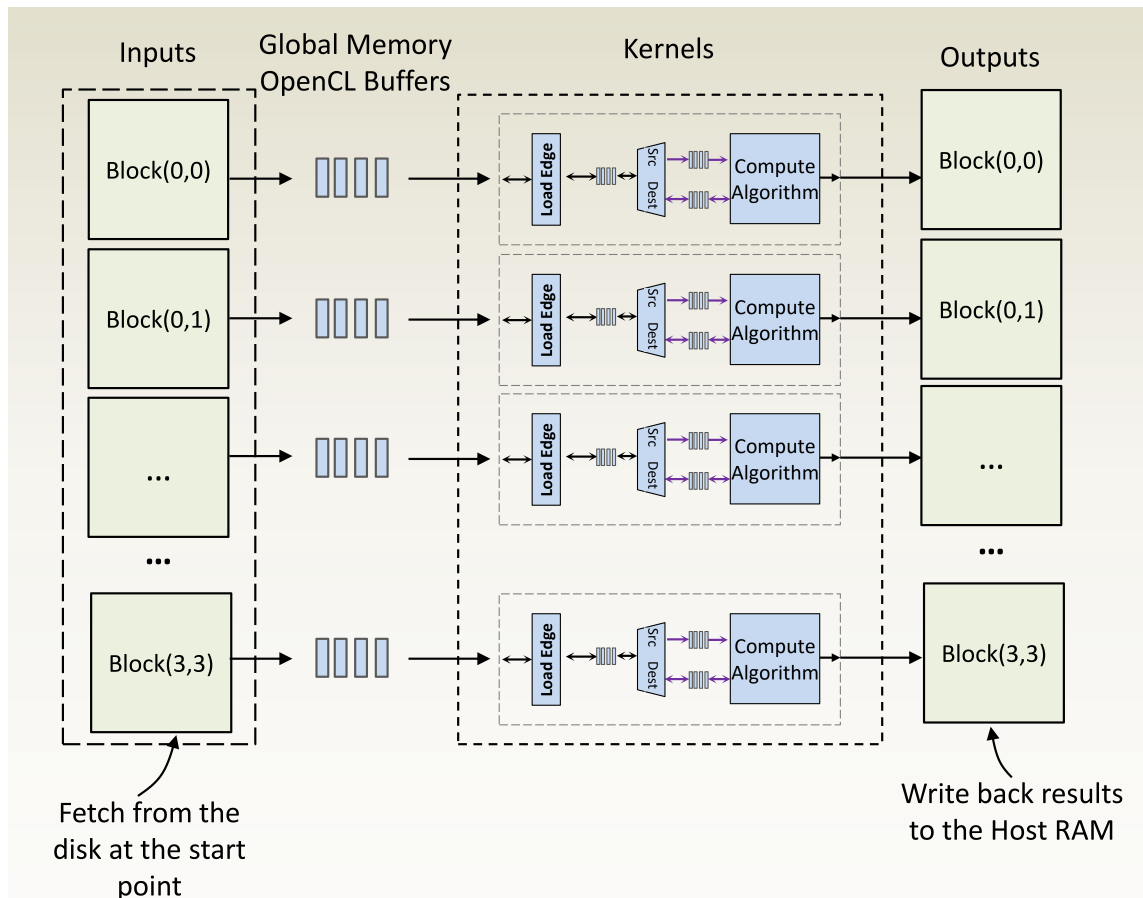


Fig. 5.4.: The Host software is responsible to drive kernel and dispatch data between CPU and Hardware accelerator.

```
std::vector<cl::Buffer> outDegree(num_cu); //outdegree buffer
std::vector<cl::Buffer> edgeSrc(num_cu); //Source edges pre-processed
std::vector<cl::Buffer> edgeDst(num_cu); //Destination edges pre-processed
std::vector<cl::Buffer> output(num_cu); //PageRank output result to write
    back into memory
std::vector<cl::Buffer> ffsize(num_cu); //Size of the each chunck of data
    to be processes

for (int i = 0; i < num_cu; i++) {
        /** Host buffers pointers */
        OCL_CHECK(err,
                        outDegree[i] =
                        cl::Buffer(context,
                                CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                                DATA_SIZE * sizeof(uint32_t),
                                outdegree.data(),
                                &err)
                );

        OCL_CHECK(err,
                        edgeSrc[i] =
                        cl::Buffer(context,
                                CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                                DATA_SIZE * sizeof(EdgeId),
                                src.data(),
                                &err)
                );

        OCL_CHECK(err,
                        edgeDst[i] =
                        cl::Buffer(context,
                                CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                                DATA_SIZE * sizeof(EdgeId),
                                dst.data(),
                                &err)
                );

        OCL_CHECK(err,
                        output[i] =
                        cl::Buffer(context,
                                CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                                DATA_SIZE * sizeof(VertexId),
                                buffer_out.data(),
                                                                &err)
                );
        OCL_CHECK(err,
                        ffsize[i] =
                        cl::Buffer(context,
                                CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                                p2 * sizeof(VertexId),
                                fsize.data(),
                                &err)
                );
}
```

Listing 5.2: The OpenCL commands used to create buffers between host and kernel.

```
std::vector<cl::Kernel> krnls(num_cu); //num_cu is the number of compute
    units
for (int i = 0; i < num_cu; i++) {
        OCL_CHECK(err, krnls[i] = cl::Kernel(program, "kernel_pagerank_0",
    &err));
}

... //rest of the host program
for (int i = 0; i < num_cu; i++) {
        int nargs = 0;
        /** setting the kernel arguments */
        OCL_CHECK(err, err = krnls[i].setArg(nargs++, edgeSrc[i]));
        OCL_CHECK(err, err = krnls[i].setArg(nargs++, edgeDst[i]));
        OCL_CHECK(err, err = krnls[i].setArg(nargs++, output[i]));
        OCL_CHECK(err, err = krnls[i].setArg(nargs++, fsize[i]));
        OCL_CHECK(err, err = krnls[i].setArg(nargs++, vertices));
        OCL_CHECK(err, err = krnls[i].setArg(nargs++, partitions));

        /** copy data to the device global memory */
        OCL_CHECK(err, err = q.enqueueMigrateMemObjects({edgeSrc[i]}, 0 ));
        OCL_CHECK(err, err = q.enqueueMigrateMemObjects({edgeDst[i]}, 0 ));
        OCL_CHECK(err, err = q.enqueueMigrateMemObjects({outDegree[i]}, 0 )
    );
        OCL_CHECK(err, err = q.enqueueMigrateMemObjects({ffsize[i]}, 0 ));
}
for (int i = 0; i < num_cu; i++) {
        /** Launch the Kernel */
        OCL_CHECK(err, err = q.enqueueTask(krnls[i]));

}
OCL_CHECK(err, err = q.finish()); //sync to execution
//and ensure all the execution is done till this point
```

Listing 5.3: The OpenCL commands to run the kernel using appropriate arguments and pointing to created buffers in Listing 5.2.

The following items highlight how to ensure that data access patterns result in the FPGA implementation [170].

1) Reduce the number of data input reads. The data received into the block can readily feed numerous parallel paths, but the hardware function's inputs can become performance bottlenecks.

2) If the data must be reused, read it once and employ a local cache.

3) Access to arrays, especially big arrays, should be kept to a minimum. Arrays are implemented in block RAM, which, similar to I/O ports, has a limited number of ports and can be a performance bottleneck. Arrays can be partitioned into smaller arrays and even single registers, although dividing huge arrays will take a lot of registers. To hold accumulations, use local localised caches and then write the final result to the array.

4) Rather than conditionally executing jobs, including pipelined tasks, try to do conditional branching inside pipelined tasks. Conditionals are implemented in the pipeline as independent pathways. Allowing input from one task to flow into the next task while applying the condition inside the next task will result in a more efficient system.

5) Shoud avoid writing unnecessary output for the same reason as input reads: ports are bottlenecks. Replicating more accesses merely exacerbates the problem.

6) Consider using a coding style that encourages read-once/write-once to function parameters in C code that processes data in a streaming fashion, since this guarantees the function can be effectively implemented in an FPGA. It is more productive to write a C method that results in a high-performance FPGA implementation than it is to figure out why the FPGA isn't doing as well as it should.

Fig. 5.5 shows a high-level overview of the design implementation. We used the *Out of order command queue* to schedule better the commands from host to the kernel. Commands from the Command queue can be issued in any order by the scheduler [150]. We explicitly built up event dependencies and synchronizations in this manner. Listing 5.4 shows the kernel declaration and its arguments.

Multi Kernel Implementation

The Vitis compiler builds a single hardware instance from a kernel. If the host program executes the same kernel multiple times due to data processing requirements, it must execute the kernel on the hardware accelerator sequentially. The order in which kernels are executed has an influence on overall application performance. Vitis, on the other hand, customises the kernel linking stage such that a single kernel can instantiate several hardware compute units (CUs). The host software may now make several overlapping kernel calls, executing kernels concurrently by running independent compute units, which can increase performance. Using parameters in the v++ config file during linking, several CUs of a kernel can be produced, as demonstrated in Listing 5.6.

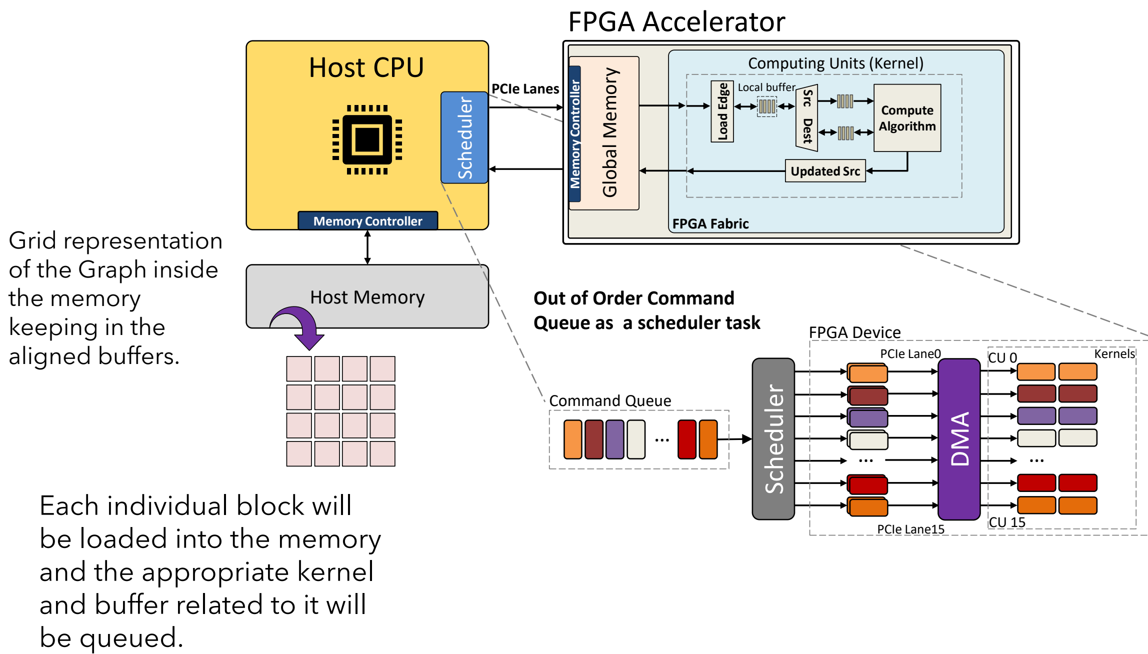


Fig. 5.5.: The hardware implementation high-level overview.

```
#define DAMPING_FACTOR 0.85
#define PE 16 //number of PEs
#define BUFFER_SIZE 64 //local arraye size
#define DATA_WIDTH 512

typedef ap_uint<DATA_WIDTH> pkt_data;
typedef unsigned int u32;

extern "C" {
        void kernel_pagerank(
                        pkt_data *edge_src, //input 1 for source edges
                        pkt_data *edge_dst, //input 2 for destination edges
                        pkt_data *out_pr, //output pagerank value
                        int size,         //size of the each block
                        int vertices,
                        int partitions
                        ) {
                          ... //rest of implementation
                        }
}
```

Listing 5.4: The kernel function declaration.

```
[connectivity]
nk=pr:15:pr_1.pr_2.pr_3.pr_4.pr_5.pr_6.pr_7.pr_8.pr_9.pr_10.pr_11.pr_12.
    pr_13.pr_14.pr_15
slr=pr_1:SLR0
slr=pr_2:SLR0
...
slr=pr_15:SLR2
sp=pr_1.m_axi_gmem:HOST[0]
sp=pr_2.m_axi_gmem:HOST[0]
...
sp=pr_15.m_axi_gmem:HOST[0]
```

Listing 5.5: The multi kernel configuration that enables running multiple instances of the kernel in parallel from the host program.

In this connectivity configuration, we also dedicate each kernel to the desired SLR (Super Logic Region) of the FPGA and also dedicate the memory channel required. In our design since host communicates with FPGA directly, all the kernel instances must be connected to `HOST[0]`. As PCIe interface has 16 lanes, the FPGA provide only 15 lanes to be connected to the kernel instances that can simultaneously communicate with the host. This technique optimize the design significantly and we achieved close to 7x speedup while using multiple kernel against one sequential kernel running on FPGA. Fig.5.6 shows the speedup gain we achieve to execute the algorithm with multiple parallel kernels (in this case 15 kernels). The real hardware implementation has been done on Alveo U250 from XACC Xilinx Adaptive Compute Clusters in Eth Zürich. The server specifications and the hardware accelerator with the resource utilization details is summarized in Table 5.5 and Table 5.6.

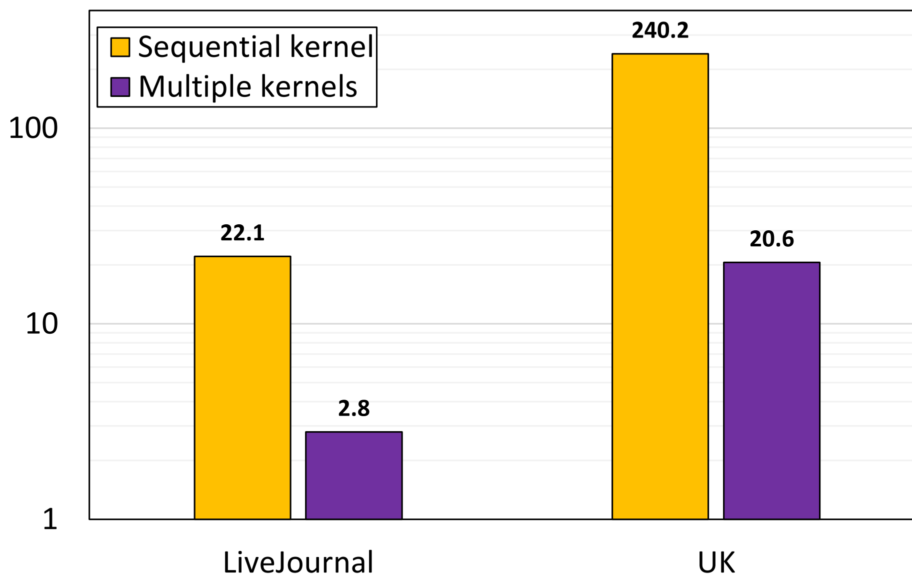


Fig. 5.6.: The speedup gained while using multiple kernel instances against running the application using sequential kernel.

Tab. 5.5.: The XACC xilinx server used to evaluate the real implementation.

Instance Name	CPU	Freq	No. of Cores	Memory	Hardware Accl.
alveo2a.ethz.ch	Intel® Xeon® Gold 6234	2.50 GHz	16	128 GiB	Alveo U250

Tab. 5.6.: The alveo U250 resource utilization in this experiment.

Resources	CLBs	BRAM	URAM	DSP	Power
Available	215777	2688	1280	12280	-
Used	55771 (25.85%)	580 (21.60%)	0	4	23.47 (w)

5.5 Evaluation and Performance Model

In order to evaluate the proposed method, we consider several steps. The first step is to evaluate the model based on the theoretical values. We defined all the metrics, system char-

Tab. 5.7.: The performance model report, which has been calculated based on the bottlenecks like PCIe Rate, Computation time, Communication time, etc. Here the number of SLR regions used is equal to 1.

Dataset	Transfer Data to Memory [1]	Optimal number of Partitions	Computation Time	Execution Time DFE	Execution Time Software Baseline	Anticipated Speedup
LiveJournal	0.81	24	2.95	**3.76**	12.86	3.42
UK	9.76	16	39.03	**48.8**	1347	27.6
Twitter	14.89	35	61.26	**76.1**	538.1	7.1
Yahoo	67.7	73	322.2	**389.9**	4719	12

[1] Reported time is seconds.

Tab. 5.8.: The datasets for evaluating our proposed study. We choose them based on the size and the structure of the datasets to be comparable with other works.

Graph dataset	Vertices	Edge	Size (GB)	Type
LiveJournal [98]	4.8 M	68.9 M	0.514	Social Web
Web-UK-2005 [18]	39 M	994 M	7.5	Web Graph
Twitter [18]	61.6 M	1.47 B	11	Web Graph
Yahoo [18]	1.41 B	6.64 B	51	Web Graph

acteristics and parameters and based on the selected datasets we achieved the performance model evaluation values.

In the performance model step, we figure out the bounds, bottlenecks, parallelism, and speedup we can achieve, and we implement the portion that needs to be mapped into FPGA by software C/C++ implementation. This implementation needs to be precise and describes exactly the FPGA execution. Then we provide some metrics to measure important elements, like speedup compared to parallelism with no partitioning. Some of the predicted system characteristics of the model has shown in Table 5.7. As can be seen, the anticipated speedup is calculated based on the computation time, communication time and other overheads and bottlenecks such as PCIe Rate (here, we consider 0.85 as the efficiency of the PCIe). Note that here we use all the SLR regions in our design aim to exploit the all capacity of resources available on the FPGA.

Based on this performance evaluation, we achieved close to $\sim$ 12 times better than baseline studies on CPU [181]. However, the status of current implementation on single FPGA is slightly slower than what we calculated and expected in the performance model. This may have serveral reasons including: **1)** The implementation is not percisely perform enough parallelism or efficiency on hardware, **2)** The software model is over optimistic and some bottlenecks in not considered correctly. **3)** Since we are highly dependant to PCIe efficiency, the PCIe efficiency might be less than 0.85, which we considered in our performance model.

In terms of evaluating the real hardware implementation, as mentioned before, we deploy the implementation on Alveo U250 from XACC Xilinx Adaptive Compute Clusters in Eth Zürich, Based on the Table 5.8, **Livejournal** and **UK** datasets have been chosen to

Tab. 5.9.: The evaluation of the hardware implementation of the GridGraph algorithm on CPU and FPGA platform.

	Baseline on CPU Intel(R) Xeon(R) Gold 6234 CPU @ 3.30GHz (seconds)			FPGA works (seconds)		
Dataset	Sequential	OpenMP	GridGraph	Foregraph[40] (simulation model 24 PE)	Fabgraph[137] (simulation model 48 (2x24) PE)	Our work (1 PE)
LiveJournal	27.01	18.96	3.54	0.578	0.27	2.78
UK	275.44	214.2	32.3	N/A	N/A	20.6

evaluate the implementation. Table 5.9 shows a comparison of the of the executing the same algorithm of GridGraph on CPU and FPGA.

5.6 Conclusion

In this work we presented a distributed large scale graph processing application. We show the advantage of using FPGAs and the reason of their growth in datacenters and cloud servers. Our architecture uses Hadoop at the higher level to map a graph to the underlying hardware. The higher layer of computation is responsible for gathering the blocks of data pre-processed and stored on the host's file system and distributing them to a lower layer of computation made of FPGAs. Some parts of the high-level implementation on Hadoop is still undone and we aim to present it in our future studies. In this work, we show how graph partitioning combined with a FPGA architecture will lead to high performance without limitation on the size of the graph, even when the graph has trillions of vertices. This method combined with a distributed high-level framework like Hadoop can significantly increase the performance especially for large scale datasets. We have implemented the proposed method on single FPGA and for the next step we extend it to the multi-FPGA distributed platform. In the current architecture the host program communicate directly with the FPGA using PCIe interface. Although network bandwidth is the limiting factor for multi-FPGA performance on most systems, this can lead to a potential increase in overall system performance. Based on the experiment that we have done on PCIe interconnection in Alveo U250, the interconnect bandwidth is measured (with dummy data and 15 kernels) not exceeding than 10 GB/s. This is less thann the best bandwidth measured on DRAM almost 14.2 GB/s. However, using directly communication within host enables many features and simplifies the design and development time. Moreover, the projection of next generation hardware accelerators with PCIe version 5.0 will provide almost 64 GB/s bandwidth which is greatly help to eliminate this interconnect bandwidth as a bottleneck.

Based on our performance model in the case of PageRank, forecasts performance improvement of up to 20 times and a cost-normalized improvement of up to 12 times when comparing the proposed approach on one Xilinx Alveo U250 FPGA accelerator against a state-of-the-art baseline graph processing software implementation on a high-end CPU like a

32-core processor at 2.2 GHz. We aim to extend our application to cover graph processing benchmarks such as WCC, SSSP and SpMV.

Dataflow Case Studies | 6

> *Pure mathematics is, in its way, the poetry of logical ideas.*
>
> — **Albert Einstein**
> (German theoretical physicist)

6.1 A Dataflow Methodology for Accelerating FFT

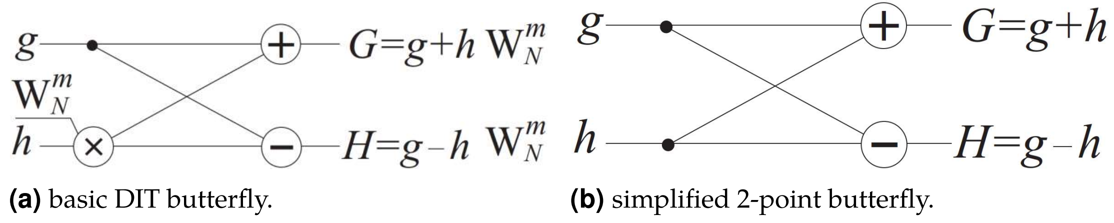
The native implementation of the N-point digital Fourier Transform involves calculating the scalar product of the sample buffer (treated as an N-dimensional vector) with N separate basis vectors. Since each scalar product involves N multiplications and N additions, the total time is proportional to N^2, in other words, it's an $O(N^2)$ algorithm. However, it turns out that by cleverly re-arranging these operations, one can optimize the algorithm down to $O(Nlog_2(N))$, which for large N makes a huge difference. The optimized version of the algorithm is called the Fast Fourier Transform, or the FFT. In this work, we discuss about an efficient way to obtain Fast Fourier Transform algorithm (FFT). According to our study, we can eliminate some operations in calculating the FFT algorithm thanks to property of complex numbers and we can achieve the FFT in a better execution time due to a significant reduction of $N/8$ of the needed twiddle factors and to additional factorizations.

6.1.1 Introduction and Theoretical background

Fast Fourier Transform (FFT) is an important signal processing algorithm widely adopted in communication systems to efficiently compute the Discrete Fourier Transform (DFT) of a signal. The computational problem for the discrete Fourier transform (DFT) is to compute the sequence X_k of N complex-valued numbers from another sequence of data x_n of length N. In general, x_n is also assumed to be complex valued. Given a finite sequence of N points x_n, $n = 0, 1, \ldots, N-1$, its DFT is by definition the finite sequence:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-j\frac{2\pi}{N}kn}, \qquad k = 0, 1, \ldots, N-1 \tag{6.1}$$

where $j = \sqrt{-1}$ and $e^{j\varphi} = \cos\varphi + j\sin\varphi$ is the Euler's formula. In the following, instead of the term $e^{-j\frac{2\pi}{N}kn}$ of Eq. 6.1, we use its equivalent term W_N^{kn} called *twiddle factor*, which represents one of the N roots of order N of the unity. The twiddle factor has three properties:



(a) basic DIT butterfly. **(b)** simplified 2-point butterfly.

Fig. 6.1.: Twiddle factor position for a length-2 DFT.

- Periodicity: $W_N^{h+N} = W_N^h$
- Symmetry: $W_N^{h+N/2} = -W_N^h$
- Recursion: $W_{N/2}^h = W_N^{2h}$

DFT has a computational complexity of $\mathcal{O}(N^2)$, since we need N^2 complex multiplication and $N \cdot (N-1)$ complex additions [82]. Based on the Cooley-Tukey decomposition [34], many FFT algorithms have been developed to reduce the number of the arithmetic operations to a computational complexity in the order of $\mathcal{O}(Nlog_2N)$ or less [131, 26, 86, 45, 47]. Without loss of generality we can assume that N is a power of two.

Given an N-point sequence where $N = 2^\nu \in \mathbb{N}$ with $\nu > 1$, then the DFT can be broken into two (N/2)-point DFT sequences. The decomposition can be performed $\nu - 1$ times, until each DFT length is 2. A length-2 DFT is also named *butterfly* for the shape of its data flow graph. The overall result is called a *radix*-2 FFT and its computation only requires $\sigma = logN$ stages, each with $N/2$-point DFT butterflies. In the following, we address more details.

Radix-2 FFT computes the DFT by dividing the N-point sequence in the even-indexed and odd-indexed points, and then combining these two results to produce the DFT of the whole sequence. In this case, Eq.6.1 becomes:

$$X_k = \sum_{n=0}^{N-1} x_n.W_N^{kn} = \sum_{n=0}^{(N/2)-1} x_{2n}.W_{N/2}^{kn} + W_N^k \sum_{n=0}^{(N/2)-1} x_{2n+1}.W_{N/2}^{kn} \tag{6.2}$$

We can derive a graphical representation of the FFT by rewriting Eq.6.2 as $A_k + {W_N}^k B_k$, and $x_{2n} = a(m)$ and $x_{2n+1} = b(m)$. Thus, DFT[$a(m)$] = $A(k)$ for even-numbered samples, and DFT[$b(m)$] = $B(k)$ for odd-numbered samples. With this assumption, represent the N/2-point DFTs of the sequences $a(m)$ and $b(m)$, respectively. Due to the periodicity of the DFT, the outputs for $N/2 \leq k < N$ from a DFT of length $N/2$ are identical to the outputs for $0 \leq k < N/2$. That is, $A(k+N/2) = A(k)$ and $B(k+N/2) = B(k)$ for $0 \leq k < N/2$. In

addition, the factor $W_N^{k+N/2} = -W_N^k$ thanks the to symmetrical property. Thus, the whole DFT can be calculated as follows:

$$\begin{gathered} X_k = A(k) + W_N^k B(k) \\ X_{k+N/2} = A(k) - W_N^k B(k) \\ 0 \leq k < N/2 \end{gathered} \tag{6.3}$$

Finally, by substituting $A(k)$ with g and $B(k)$ with h, $X(k)$ with G and $X(k+N/2)$ with H, we obtain Fig.6.1a. By recursively applying this step we arrive to $N = 2$, with the simplified implementation of Fig.6.1b. In the next sections, we describe our idea, its evaluation and more related works.

6.1.2 Reducing the needed twiddle factors

In Fig.6.2 we can observe some properties of twiddle factors that can help understand diminishing the number of operations.

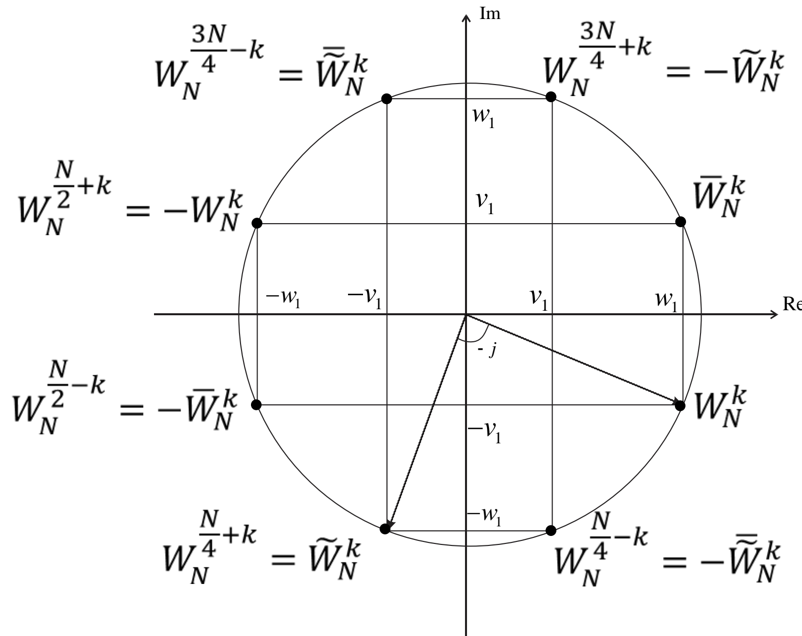


Fig. 6.2.: Relationship among twiddle factors: $\widetilde{W}$ means a rotation of $-\pi/2$, whilst $\overline{W}$ means the conjugate.

Given $W_N^k \stackrel{\text{def}}{=} w_1 - jv_1$, with $0 < k < N/2$, let us define

$$\widetilde{W}_N^k \stackrel{\text{def}}{=} (-j) \cdot W_N^k = -v_1 - jw_1 \tag{6.4}$$

which represents a rotation of $-\pi/2$ on the Re-Im plane or, in terms of "butterfly", the following diagram:

Thus, given Eq. 6.4, we can then express the following seven twiddle factors based on the single twiddle factor W_N^k as follows, where the overline represents the conjugate operation

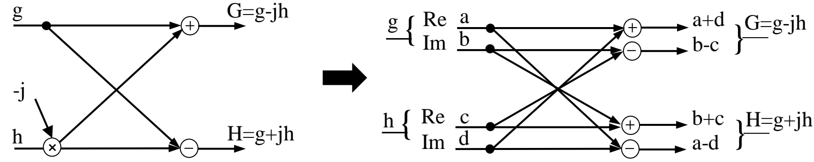


Fig. 6.3.: Eliminating Complex Multiplication based on our proposed method.

$$\begin{cases} W_N^{N/4-k} = -\overline{\widetilde{W}}_N^k \\ W_N^{N/4+k} = \widetilde{W}_N^k \\ W_N^{N/2-k} = -\overline{W}_N^k \\ W_N^{N/2+k} = -W_N^k \\ W_N^{3N/4-k} = \overline{\widetilde{W}}_N^k \\ W_N^{3N/4+k} = -\widetilde{W}_N^k \\ W_N^{-k} = \overline{W}_N^k \end{cases} \tag{6.5}$$

As can be seen in Fig. 6.3, the complex multiplication with $(-j)$ can be obtained by exchanging the Re and Im parts and adjusting signs. Similarly, for the multiplications with the twiddle factors in Eq. 6.5, we can perform just one single complex multiplication with W_N^k and make the necessary re-arrangements of Re and Im parts as well as adjusting the signs of additions and subtractions in the butterflies. So we can use just one twiddle factor instead of eight: we reduced the needed twiddle factors by a factor of 8. To further explain how we can exploit the above observations, in the next section we consider a simple example for $N = 8$ together with its data-flow program graph.

6.1.3 The 8-point FFT Data-Flow Graph

According to section 6.1.1, we can arrange the operations of FFT algorithm to obtain the butterfly diagram of Fig.6.4. In that figure, black lines carry real values, green lines imaginary values, and red lines complex numbers. Its Data-flow Program Graph (DPG) for the first two stages is shown in Fig. 6.5. In the second stage, from semantic point of view, the twiddle factor operator $W_4^1 = -j$ does not imply an actual arithmetic operation, but it only turns into a signed imaginary number its incoming real value as explained in section 6.1.2. Then, coupled to the respective real numbers in the addition and subtraction operations, these imaginary numbers form the complex numbers inside the dashed boxes. Therefore, as an example, the complex number $a_1^2 = (a_1^1 - ja_3^1)$, the complex number $a_3^2 = (a_1^1 + ja_3^1)$, and so on. In the picture the $\pi/2$ rotations and minus operations are only shown for clarity reasons, but they do not require any kind of operation, being conjugate values whose origin number is the real value a_3^1. It will be in the third stage that they actually acquire their meaning in term of sign. In fact, let us consider the two complex product $a_5^2 \times W_8^1$ and $a_7^2 \times W_8^3$. Set

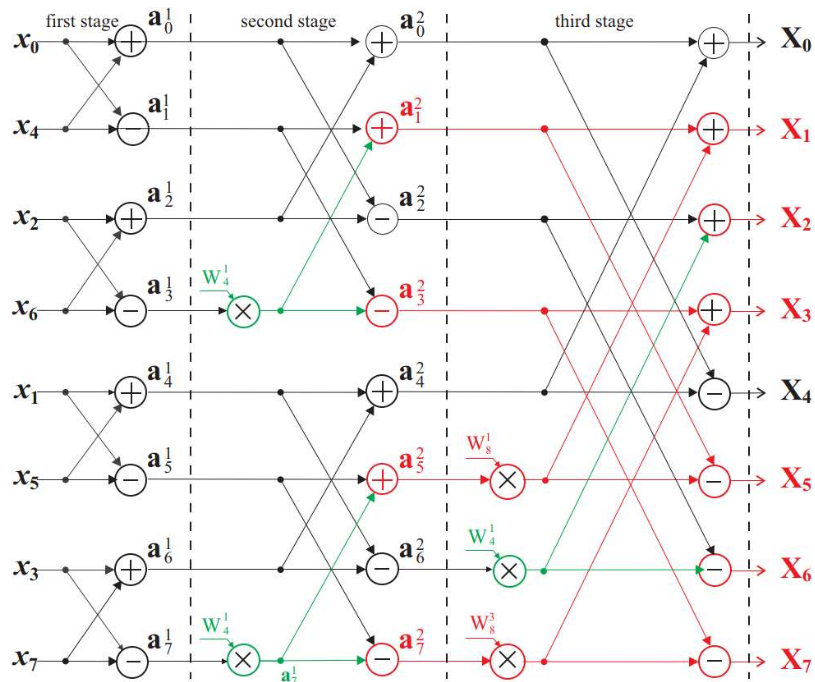


Fig. 6.4.: 8-point butterfly FFT.

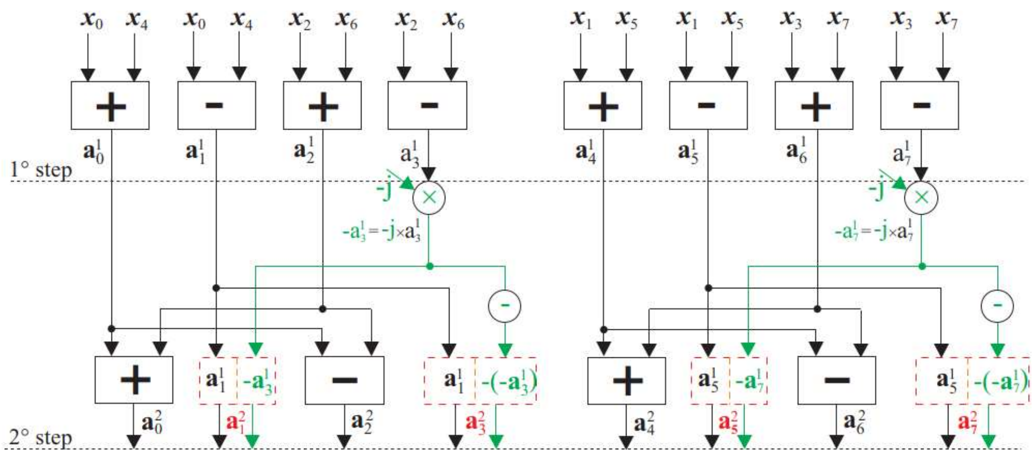


Fig. 6.5.: Data-Flow Program Graph (DPG) for the first two steps.

$W_8^1 = (w_1 - jv_1)$ and $W_8^3 = (-w_3 - jv_3)$, we have that $|w_1| = |v_1| = |w_3| = |v_3|$. Then, for the two products we have the following equations:

$$a_5^2 \times W_8^1 = (a_5^1 - ja_7^1) \times (w_1 - jv_1) = w_1(a_5^1 - a_7^1) - jw_1(a_5^1 + a_7^1) \tag{6.6}$$

$$a_7^2 \times W_8^3 = (a_5^1 + ja_7^1) \times (-w_1 - jv_1) = -w_1(a_5^1 - a_7^1) - jw_1(a_5^1 + a_7^1) \tag{6.7}$$

If in the Eq. 6.6, we name σ the real value $w_1(a_5^1 - a_7^1)$ and τ the imaginary value $-w_1(a_5^1 + a_7^1)$, then we have $a_5^2 \times W_8^1 = (\sigma + j\tau)$ and $a_7^2 \times W_8^3 = (-\sigma + j\tau)$. Consequently, the two complex multiplications are reduced to two real multiplications and additions, where the real values σ have opposite signs (π rotation) but same imaginary value τ. Now for σ we act just like a_3^1. Fig.6.6 shows the new DPG of the butterfly in Fig.6.3. In the new diagram of Fig. 6.6 the total operations on real numbers are reduced to 27 (Table 6.1). Previous work showed that the number of mathematical operations can be reduced further [59, 46], however, here we propose a practical implementation inspired by the data-flow graph instead of a purely mathematical study. Even if not shown in Fig.6.6, we could have simplified further the DPG by applying the relation $X(k) = \overline{X}(N-k)$ where required, thus saving four ADD/SUB operations; we can save also the SUB before a_6^2 thus obtaining the result reported in [59] for an 8-point DIT (20 ADD/SUBs and 2 MULTs). This DPG in Fig. 6.6 has the advantage of including only dyadic operators and the critical path is optimized: in the next section we will show that this implementation leads to a better execution time compared to [34, 54], when translated into C code.

Tab. 6.1.: Summarized number of different operations in DPG

Methodology	Complex MULT	Re and Im ADD/SUB	Total ops.
Cooley-Tukey FFT [34]	12	31	**43**
Reduced FFT	2	25	**27**

6.1.4 Experimental validation

We compare the execution time of our 8-point modified algorithm,which is implemented on Ryzen7 AMD 16 Core CPU running on 3.6 GHz clock. For measuring the performance of this proposed FFT Data-Flow graph, we calculated the result by a C/C++ code [127].

As can be seen in Table 6.2, we implemented our proposed algorithm, FFT[34] and FFTW version 3.3.8 [54] by using fftw_plan_dft_r2c_1d plan. The results show that our proposed algorithm reduces the execution time significantly, compared to FFT and FFTW. The execution time in a our proposed methodology is 0.160μs while in FFTW is 1.5x and in FFT is 2.2x slower. We used "gettimeofday" function to acquire these results and listed in Table 6.2.

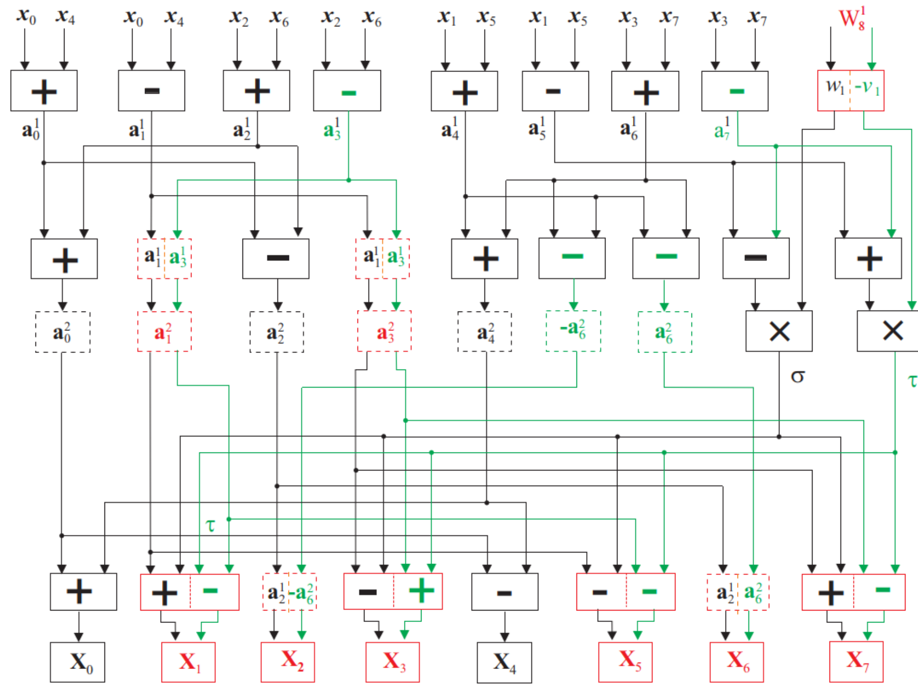


Fig. 6.6.: DPG after the third step.

Tab. 6.2.: Experiment Result for our proposed FFT Algorithm

Algorithm	Compiler	Time (μs)
Cooley-Tukey FFT [34]	GCC	0.348
FFTW [54]	GCC	0.248
Our Proposed FFT	GCC	**0.160**

6.1.5 Related work

There are many studies about implementing FFT algorithm on FPGA's and CPU's, however this study shows the efficiency of implementing FFT algorithm in a new way to eliminate some operations to achieve high performance result. Just a few years after 1965 when Cooley-Tukey FFT algorithm published, many scientists around the world start to investigate this interesting and useful algorithm in many different applications. As a result their work made FFT literature highly immense. In [172] Yavne presented what became known as the "split-radix" FFT algorithm for N=2^m obtaining an improvement by 20% over the classic "radix-2" algorithm presented by Cooley-Tukey. More recently, in [84] the Authors lowered the operations number by a further 5.6%. Although the performance of FFTs on recent computer hardware is determined by many factors besides pure arithmetic counts [3], there still remains an intriguing unsolved mathematical question: what is the smallest number of arithmetic operations required to compute a DFT of a given size? Other FFT algorithms, such as radix-4[142], radix-8[26], radix-(4+2)[86], split-radix[47] algorithms, have been proposed using the complex mathematical relationship to reduce the hardware complexity. As the algorithms were derived based on intensive mathematical manipulation, it is not straightforward to understand the mathematical meaning and apply them to derive new FFT algorithms. The computational complexity and the hardware requirement are greatly

dependent on the FFT algorithms in use. In[45], Despain showed a new method of deriving very fast FFT algorithms to be implemented in a digital hardware. Accordingly, there are other scientific researches worked on implementing algorithm on digital hardware for different applications to accelerate computation time along with reducing the computational complexity. In[147], authors showed the performance and energy efficiency of a processor-integrated FFT accelerator, designed to support efficient integration of low-level and high level signal, image, and video processing. In[27], authors show the design method of a real-time FFT processor which introduced adaptive overflow control to avoid overflow without interrupting the computing pipeline. In other hand, for Data-Flow programming, in [158] authors present a new type of soft-core processor called the "Data-Flow Soft-Core" that can be implemented through FPGA technology with adequate interconnect resources. This processor provides data processing based on data-flow instructions rather than control flow instructions [107]. Accordingly we inspired our data-flow computing procedure from which authors discussed about it basically in [160].

6.1.6 Conclusion and future works

Fast Fourier Transform is a widely used kernel in many research fields. In this study, we introduce a new methodology to reduce the arithmetic operations for a FFT algorithm, then we show the experimental results for implementing this FFT data-flow. Results show that the time required to reach the result by reducing the operations is 1.5x faster that other well-known FFT algorithm such as FFTW. As our future work we will expand this methodology to generalize the algorithm.

7 A Custom Board To Perform Distributed Computing

> *Users do not care about what is inside the box, as long as the box does what they need done.*
>
> — **Jef Raskin**
> about Human Computer Interfaces

7.1 Gluon: The High-Speed Interconnect Solution

Heterogeneous systems are one of the most discussed architectures in computer science. Their capabilities have provided many good features for researchers to use this kind of structure in their state-of-the-art works. Heterogeneous systems are flexible, cost-efficient, and well-supported by communities. They are widely used in artificial intelligence, automotive, IoT, and embedded applications. Moreover, there is also a challenge to have a sufficient, cost-efficient, and flexible structure to use heterogeneous systems. In this section, we present the Gluon board, which uses serial transceivers in Xilinx Ultrascale+ structure and facilitates using GTH transceivers in high data rate transfer applications. The possible solution would be a high data rate cluster network based on Zynq Ultrascale+ MPSoCs, which can easily deploy a multi-node, multi-code structure at a reasonable cost [7].

In recent years, there are some works to introduce a heterogeneous platform such as AXIOM [63], which can provide flexible infrastructure for AI applications as well discussed in [64]. However, there was a need to exploit all serial transceivers of Xilinx Ultrascale+, plus, can also capable of carrying fully operating support with a sufficient amount of memory. We designed the Gluon board, which can provide all these requirements. As can be seen in 7.1, the idea is to provide necessary elements to build a FPGA based cluster to accelerate applications such as AI, IoT, automotive and computing applications [68, 132, 3]. The worker Nodes in this figure, can be connected to each other and can receive orders by the Root node. The connection between each two nodes is up to 18 Gb/s and the topology of the network can be define by the user and depends on the applications such as Mesh, Start, Ring, 2D-Torus etc. This infrastructure, pave the road to have a distributed system, which is applicable to develop and test many interesting ideas.

The speed rate of GTH transceivers has been calculated by a running full Ubuntu AXIOM software stack [4] by using NETPERF measurement between two board using variety of payloads. As can be seen from Fig.7.2b, we measured payload sizes from 1024 to 524280 bytes starting with 2 software threads. Then we increased the number of threads up to

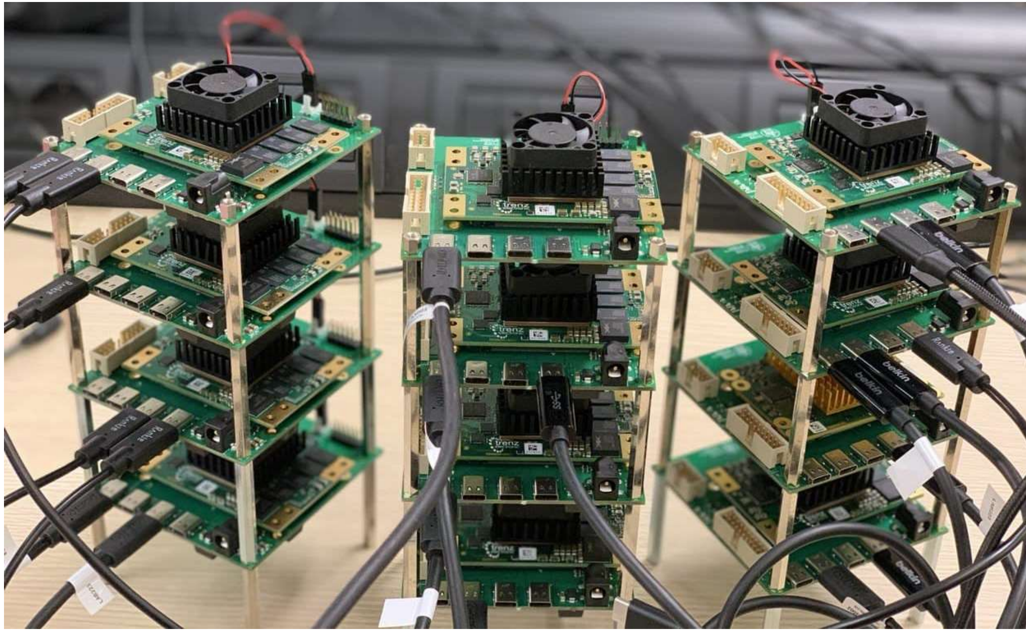
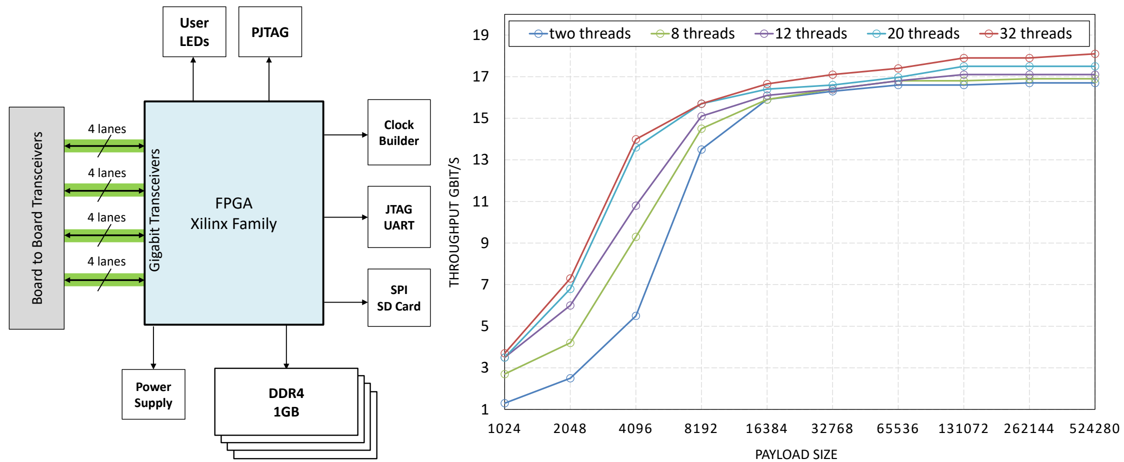


Fig. 7.1.: The cluster of FPGAs using Gluon boards. The network is capable of creating different network topologies such as Mesh, Star, 2D-Torus etc.



(a) The Gluon board block diagram. **(b)** The throughput of two connected Gluon board with 1 connection connected.

Fig. 7.2.: Gluon board block diagram and throughput to test the functionality of the board.

32 threads and see how the workload can saturate the network and reach its maximum performance. For the future work we have planned to add some useful features to Gluon board, that can cover more area of computing science some applications [3, 159]. Another future study is the RISC-V core implementation and experiment by using our dataflow execution model can be achievable by a significantly reasonable cost.

7.2 Conclusion

We designed and presented the first version of the Gluon board (see Fig.7.2a), which provides necessary elements to fully exploit Giga Transceivers of Xilinx Ultrascale+ modules. This structure is cost-effective, reliable, and flexible, a different number of modules of Xilinx MPSoCs can embark on this carrier board based on the budget of the user and just need to check the compatibility of the module with the carrier board.

Gluon board is the modified and enhanced version of the TEBT0808 board from Trenz Company [69], which with the new design is able to power up the FPGA module with unix-based operating system. Gluon enables serial transceivers in Xilinx Ultrascale+ structure and facilitates using GTH transceivers in high data rate transfer applications.

8 Conclusion and Future Studies

> *What we know is a drop, what we don't know is a ocean.*
>
> **— Isaac Newton**
> (1643-1727)

In this section, we conclude the discussion of this thesis. Here we point out the achievements and what has been left and couldn't cover during the Ph.D. period. A critical aspect of the thesis is that while it is essential to analyze the theoretical bounds of algorithms, it is also necessary that the algorithms perform well in practice. Therefore, we focused on establishing a solid baseline with notable benchmarks to stress the potential of the execution models. We performed different evaluations on the selected benchmarks and got practical conclusions within these experiments. We achieved several results regarding our Dataflow Execution Model compared to other well-known programming models, similar to our execution model. We outperform Cilk and OpenMPI with different experiments. The outstanding investigation against Cilk and OpenMPI leads us to believe the following facts:

1) Even though DF-threads do not have compiler support and has an overhead of translated function compiled by GCC compiler, it can outperform well-known parallel programming models in both multi-core and multi-node fashion.
2) DF-Threads has a great potential to be distributed over a multi-node platform. This is challenging for many parallel programming models.
3) DF-Threads performs parallelism in fine-grained level and shows great potential to execute many fine-grained elements without losing the performance.

We are particularly interested in heterogeneous platforms in this thesis. Heterogeneous systems are adaptable, cost-effective, and popular among communities. Our research focuses on CPU+FPGA heterogeneous systems, typically consisting of a general-purpose CPU (x86 or ARM) running alongside an FPGA accelerator in a general-purpose Operating System. A state-of-the-art, highly efficient graph processing software implementation on a high-end CPU such as a 32-core processor at 2.2 GHz against a Xilinx Alveo U250 FPGA accelerator. We design and fabricate the GLUON board, which employs serial transceivers in the Xilinx Ultrascale+ Heterogeneous accelerator and supports GTH transceivers in high rate data transfer applications, a necessity in our hardware platform structure.

Moreover, a practical application is shown to illustrate the usefulness of the suggested execution paradigm and tools. We design and demonstrate a graph processing implementation aim to cover very large scale graph sizes. We first choose a graph partitioning method from

the literature to do this. We ended up GridGraph partitioning method to partition the input graph and reside the chunks and other helpful information in the preprocessing phase on the file system. We use this information and data as the input files of the hardware accelerator. At a higher level, Hadoop is used to translate a graph to the underlying hardware in our design. The higher computing layer is in charge of gathering and distributing preprocessed and stored data blocks from the host's file system to a lower layer of computation made up of FPGAs.

In future work we plan to optimize the hardware kernels and deploy or hardware architecture on real cloud platform (which was one of the main motivations of this work). Another future development is to deploy our Dataflow-Threads execution model on our Gluon heterogeneous platform. In this regard, a potential RISC-V implementation of the DF-Thread on Gluon structure will be implemented and evaluated as a prototype to describe the capability of the DF-thread as a next generation of exascale high performance computing framework.

Bibliography

[1] A light-weight MPI Profiler. *http://github.com/LLNL/mpiP* (cit. on p. 113).

[2] E. Ajkunic, Hana Fatkić, K. Talic, and Novica Nosovic. "A comparison of five parallel programming models for C++". In: MIPRO, 2012 Proceedings of the 35th International Convention, Jan. 2012, pp. 1780–1784 (cit. on p. 28).

[3] Amin Sahebi Ali Soleimani. "Using neural networks to predict road roughness". In: *Journal of Solid and Fluid Mechanics* 2 (2012), pp. 63–69 (cit. on pp. 91, 92).

[4] Carlos Alvarez, Eduard Ayguade, Jaume Bosch, et al. "The AXIOM software layers". In: *Microprocessors and Microsystems* 47 (2016), pp. 262–277 (cit. on p. 91).

[5] Tiago A. O. Alves, Leandro A. J. Marzulo, Felipe M. G. Franca, and Vitor Santos Costa. "Trebuchet: Exploring TLP with Dataflow Virtualisation". In: *Int. J. High Perform. Syst. Archit.* 3.2/3 (May 2011), pp. 137–148 (cit. on p. 6).

[6] Tiago A. O. Alves, Leandro A. J. Marzulo, Felipe M. G. Franca, and Vitor Santos Costa. "Trebuchet: Exploring TLP with Dataflow Virtualisation". In: *Int. J. High Perform. Syst. Archit.* 3.2/3 (May 2011), pp. 137–148 (cit. on p. 15).

[7] Roberto Giorgi Amin Sahebi. "GLUON, The High-Speed Inexpensive and Easy Interconnect Solution". In: 2020/7 (cit. on p. 91).

[8] *Apache Hadoop Foundation*. `https://hadoop.apache.org/`. Jan. 2022 (cit. on p. 70).

[9] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. "COTSon: infrastructure for full system simulation". In: *SIGOPS Oper. Syst. Rev.* 43.1 (2009), pp. 52–61 (cit. on pp. 31, 57).

[10] Arvind. "Data Flow Languages and Architectures". In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, p. 1 (cit. on pp. 5, 8, 9).

[11] K. Arvind and Rishiyur S. Nikhil. "Executing a Program on the MIT Tagged-Token Dataflow Architecture". In: *IEEE Trans. Comput.* 39.3 (1990), pp. 300–318 (cit. on pp. 5, 8, 9).

[12] Arvind and David E. Culler. "Dataflow Architectures". In: *Annual Review of Computer Science* (1986) (cit. on pp. 5, 12).

[13] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016 (cit. on p. 18).

[14] Mikhail Asiatici and Paolo Ienne. "Large-Scale Graph Processing on FPGAs with Caches for Thousands of Simultaneous Misses". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 609–622 (cit. on pp. 66–68).

[15] Jonathan Bachrach, Huy Vo, Brian Richards, et al. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221 (cit. on pp. 18, 67).

[16] Jairo Balart, Alejandro Duran, Marco Gon, et al. "Nanos mercurium: A research compiler for OpenMP". In: 2004 (cit. on p. 19).

[17] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. *Graph Processing on FPGAs: Taxonomy, Survey, Challenges*. 2019. arXiv: 1903.06697 [cs.DC] (cit. on p. 64).

[18] Paolo Boldi and Sebastiano Vigna. "The Web Graph Framework I: Compression Techniques". In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601 (cit. on p. 79).

[19] David F. Brailsford and R. James Duckworth. "The MUSE Machine – an Architecture for Structured Data Flow Computation". In: *New Generation Computing* 3 (2 1985). Ed. by T. Moto-oka and K. Fuchi (cit. on p. 8).

[20] Joseph Buck and Edward Lee. "The Token Flow Model". In: (Dec. 1994) (cit. on p. 13).

[21] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. "Dataflow: A Complement to Superscalar". In: *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* (2005), pp. 177–186 (cit. on p. 6).

[22] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, et al. "Scaling to the end of silicon with EDGE architectures". In: *Computer* 37 (2004), pp. 44–55 (cit. on p. 6).

[23] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures". In: May 2007, p. 20 (cit. on p. 44).

[24] CAPSL. *The Codelet Execution Model*. https://www.capsl.udel.edu/codelets.shtml (cit. on p. 15).

[25] Márcia C. Cera, Guilherme Peretti Pezzi, Elton N. Mathias, Nicolas Maillard, and Philippe Olivier Alexandre Navaux. "Improving the Dynamic Creation of Processes in MPI-2". In: *PVM/MPI*. 2006 (cit. on pp. 30, 31).

[26] Y. Chandu, M. Maradi, A. Manjunath, and P. Agarwal. "Optimized High Speed Radix-8 FFT Algorithm Implementation on FPGA". In: *2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*. May 2018, pp. 430–435 (cit. on pp. 84, 89).

[27] Chu Chao, Zhang Qin, Xie Yingke, and Han Chengde. "Design of a high performance FFT processor based on FPGA". In: *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. ACM. 2005, pp. 920–923 (cit. on p. 90).

[28] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 367–379 (cit. on p. 8).

[29] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308 (cit. on pp. 8, 9).

[30] Xinyu Chen, Hongshi Tan, Yao Chen, et al. "ThunderGP: HLS-Based Graph Processing Framework on FPGAs". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 69–80 (cit. on pp. 66, 68, 70, 71).

[31] Y. Chen, J. Emer, and V. Sze. "Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators". In: *IEEE Micro* 37.3 (2017), pp. 12–21 (cit. on p. 9).

[32] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. "Detecting Data Races in Cilk Programs That Use Locks". In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '98. Puerto Vallarta, Mexico: Association for Computing Machinery, 1998, pp. 298–309 (cit. on p. 52).

[33] Cilk multithreaded programming technology. *https://cilk.mit.edu/* (cit. on pp. 22, 27).

[34] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301 (cit. on pp. 84, 88, 89).

[35] Martin Cowley and Lina Sawalha. "RISC-V Dataflow Extension". In: *Fifth Workshop on Computer Architecture Research with RISC-V (CARRV '21)*. 2021, p. 7 (cit. on pp. 17–19).

[36] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 164–175 (cit. on p. 8).

[37] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 105–110 (cit. on pp. 64, 68).

[38] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 105–110 (cit. on p. 67).

[39] Guohao Dai, Tianhao Huang, Yuze Chi, et al. "ForeGraph: Exploring Large-Scale Graph Processing on Multi-FPGA Architecture". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, p. 217226 (cit. on pp. 64, 66).

[40] Guohao Dai, Tianhao Huang, Yuze Chi, et al. "ForeGraph: Exploring Large-Scale Graph Processing on Multi-FPGA Architecture". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, p. 217226 (cit. on pp. 67, 68, 70, 80).

[41] A. L. Davis. "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine". In: *Proceedings of the 5th Annual Symposium on Computer Architecture*. ISCA '78. New York, NY, USA: Association for Computing Machinery, 1978, pp. 210–215 (cit. on pp. 7, 8).

[42] Jack B. Dennis. "Compiling Fresh Breeze Codelets". In: *Proceedings of Programming Models and Applications on Multicores and Manycores*. PMAM'14. Orlando, FL, USA: Association for Computing Machinery, 2014, pp. 51–60 (cit. on p. 12).

[43] Jack B. Dennis. "Data Flow Computation". In: *Control Flow and Data Flow: Concepts of Distributed Programming*. Ed. by Manfred Broy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 345–398 (cit. on p. 5).

[44] Jack B. Dennis and David P. Misunas. "A Preliminary Architecture for a Basic Data-Flow Processor". In: (1974) (cit. on pp. 5, 7, 8).

[45] Alvin M. Despain. "Very fast Fourier transform algorithms hardware for implementation". In: *IEEE Transactions on Computers* 5 (1979), pp. 333–341 (cit. on pp. 84, 90).

[46] P. Duhamel. “Implementation of "Split-radix" FFT algorithms for complex, real, and real-symmetric data”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 34.2 (Apr. 1986), pp. 285–295 (cit. on p. 88).

[47] P. Duhamel and H. Hollmann. “Split radix’ FFT algorithm”. In: *Electronics Letters* 20.1 (Jan. 1984), pp. 14–16 (cit. on pp. 84, 89).

[48] Nina Engelhardt and Hayden K.-H. So. “GraVF-M: Graph Processing System Generation for Multi-FPGA Platforms”. In: *ACM Trans. Reconfigurable Technol. Syst.* 12.4 (Nov. 2019) (cit. on pp. 67, 68).

[49] Nina Engelhardt and Hayden Kwok-Hay So. “GraVF: A vertex-centric distributed graph processing framework on FPGAs”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–4 (cit. on pp. 64, 67).

[50] C. Farabet, B. Martini, B. Corda, et al. “NeuFlow: A runtime reconfigurable dataflow processor for vision”. In: *CVPR 2011 WORKSHOPS*. 2011, pp. 109–116 (cit. on p. 6).

[51] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, et al. “Task-Based Programming with OmpSs and Its Application”. In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by Luís Lopes, Julius Žilinskas, Alexandru Costan, et al. Cham: Springer International Publishing, 2014, pp. 601–612 (cit. on p. 19).

[52] Alejandro Fernández, Vicenç Beltran, Sergi Mateo, Tomasz Patejko, and Eduard Ayguadé. “A Data Flow Language to Develop High Performance Computing DSLs”. In: *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. 2014, pp. 11–20 (cit. on p. 6).

[53] Eric Finnerty, Zachary Sherer, Hang Liu, and Yan Luo. “Dr. BFS: Data Centric Breadth-First Search on FPGAs”. In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6 (cit. on p. 64).

[54] Matteo Frigo and Steven G Johnson. “FFTW: An adaptive software architecture for the FFT”. In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP’98 (Cat. No. 98CH36181)*. Vol. 3. IEEE. 1998, pp. 1381–1384 (cit. on pp. 88, 89).

[55] L. Gan, H. Fu, O. Mencer, W. Luk, and G. Yang. “Chapter Four - Data Flow Computing in Geoscience Applications”. In: *Creativity in Computing and DataFlow SuperComputing*. Ed. by Ali R. Hurson and Veljko Milutinović. Vol. 104. Advances in Computers. Elsevier, 2017, pp. 125–158 (cit. on p. 6).

[56] G. R. Gao. “A Code Mapping Scheme for Dataflow Software Pipelining”. In: (Jan. 1991), p. 120 (cit. on p. 12).

[57] G.R. Gao. “An efficient hybrid dataflow architecture model”. In: *Parallel Distributed Computers* 19.4 (1993), pp. 293–307 (cit. on p. 10).

[58] Guang R. Gao, Herbert H. J. Hum, and Jean-Marc Monti. “Towards an Efficient Hybrid Dataflow Architecture Model”. In: *Proceedings on Parallel Architectures and Languages Europe Volume I. Parallel Architectures and Algorithms*. PARLE ’91. Eindhoven, The Netherlands: Springer-Verlag, 1991, pp. 355–371 (cit. on p. 10).

[59] M. Garrido, K. K. Parhi, and J. Grajal. “A Pipelined FFT Architecture for Real-Valued Signals”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 56.12 (Dec. 2009), pp. 2634–2643 (cit. on p. 88).

[60] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. “XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures”. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 1299–1308 (cit. on pp. 6, 15).

[61] R. Giorgi and P. Faraboschi. “An Introduction to DF-Threads and their Execution Model”. In: *IEEE MPP*. Paris, France, Oct. 2014, pp. 60–65 (cit. on pp. xix, 6, 12, 15, 16, 54).

[62] R. Giorgi, F. Khalili, and M. Procaccini. “A Design Space Exploration Tool Set for Future 1K-core High-Performance Computers”. In: *ACM RAPIDO Workshop*. 2019, pp. 1–6 (cit. on pp. 57, 61, 62).

[63] R. Giorgi, F. Khalili, and M. Procaccini. “AXIOM: A Scalable, Efficient and Reconfigurable Embedded Platform”. In: *IEEE Proc.DATEi*. Mar. 2019, pp. 1–6 (cit. on pp. 6, 91).

[64] R. Giorgi, Farnam. Khalili, and Marco Procaccini. “Translating Timing into an Architecture: The Synergy of COTSon and HLS (Domain Expertise – Designing a Computer Architecture via HLS)”. In: *Hindawi - International Journal of Reconfigurable Computing* 2019 (Dec. 2019), pp. 1–18 (cit. on pp. 61, 91).

[65] R. Giorgi and Marco Procaccini. “Bridging a Data-Flow Execution Model to a Lightweight Programming Model”. In: *2019 International Conference on HPCS* (2019) (cit. on p. 55).

[66] R. Giorgi and A. Scionti. “A scalable thread scheduling co-processor based on data-flow principles”. In: *ELSEVIER Future Generation Computer Systems* 53 (Dec. 2015), pp. 100–108 (cit. on pp. 15, 19, 55).

[67] Roberto Giorgi et al. “TERAFLUX: Harnessing dataflow in next generation teradevices”. In: *ELSEVIER Microprocessors and Microsystems* 38.8, Part B (2014), pp. 976–990 (cit. on pp. 2, 6, 8, 10, 12, 19).

[68] Roberto Giorgi, Marco Procaccini, and Amin Sahebi. “DRT: A Lightweight Runtime for Developing Benchmarks for a Dataflow Execution Model”. In: *Architecture of Computing Systems - 34th International Conference, ARCS 2021, Virtual Event, June 7-8, 2021, Proceedings*. Vol. 12800. Lecture Notes in Computer Science. Springer, 2021, pp. 84–100 (cit. on pp. xix, 91).

[69] Trenz Company Gmbh. *Testboard for Trenz Electronic MPSoC modules*. https://shop.trenz-electronic.de/en/TEBT0808-02-Testboard-for-Trenz-Electronic-MPSoC-modules-TE0803-TE0807-and-TE0808. Feb. 2022 (cit. on pp. 4, 93).

[70] Graham Gobieski, Amolak Nagi, Nathan Serafin, et al. “MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 670–684 (cit. on p. 6).

[71] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30 (cit. on p. 70).

[72] V.G. Grafe and J.E. Hoch. “The Epsilon-2 hybrid dataflow architecture”. In: *Digest of Papers Compcon Spring ’90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*. 1990, pp. 88–93 (cit. on p. 8).

[73] Richard L. Graham and Galen Shipman. “MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 130–140 (cit. on p. 118).

[74] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using mpi: Portable parallel programming with the message-passing interface*. The MIT Press, 2014 (cit. on p. 41).

[75] L.A. van der Gugten. *Load balancing framework comparison*. June 2020 (cit. on p. 28).

[76] J. R Gurd, C. C Kirkham, and I. Watson. "The Manchester Prototype Dataflow Computer". In: *Commun. ACM* 28.1 (1985), pp. 34–52 (cit. on p. 6).

[77] High-Performance Portable MPI. *https://www.mpich.org/* (cit. on pp. 22, 31).

[78] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. "Efficient execution of memory access phases using dataflow specialization". In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 118–130 (cit. on p. 11).

[79] H.H.J. Hum, K.B. Theobald, and G.R. Gao. "Building multithreaded architectures with off-the-shelf microprocessors". In: *Proceedings of 8th International Parallel Processing Symposium*. 1994, pp. 288–294 (cit. on p. 8).

[80] W. Hwu and Y. N. Patt. "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality". In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. ISCA '86. Tokyo, Japan: IEEE Computer Society Press, 1986, pp. 297–306 (cit. on pp. 8, 9).

[81] IBM Spectrum MPI. *https://www.ibm.com/products/spectrum-mpi* (cit. on p. 41).

[82] Freescale Semiconductor Inc. "FFT-Based Algorithm for Metering Applications," in: (2015) (cit. on p. 84).

[83] Intel® MPI Library. *https://software.intel.com/content/www/us/en/develop/tools/oneapi.html* (cit. on p. 41).

[84] S. G. Johnson and M. Frigo. "A Modified Split-Radix FFT With Fewer Arithmetic Operations". In: *IEEE Transactions on Signal Processing* 55.1 (Jan. 2007), pp. 111–119 (cit. on p. 89).

[85] Herbert Jordan, Peter Thoman, Peter Zangerl, Thomas Heller, and Thomas Fahringer. "A Context-Aware Primitive for Nested Recursive Parallelism". In: May 2017, pp. 149–161 (cit. on p. 28).

[86] Yunho Jung, Hongil Yoon, and Jaeseok Kim. "New Efficient FFT Algorithm and Pipeline Implementation Results for OFDM/DMT Applications". In: *IEEE Trans. on Consum. Electron.* 49.1 (Feb. 2003), pp. 14–20 (cit. on pp. 84, 89).

[87] David Karlbom. "A Performance Evaluation of MPI Shared Memory Programming". In: (2016) (cit. on pp. 27, 118).

[88] K.M. Kavi, R. Giorgi, and J. Arul. "Scheduled dataflow: execution paradigm, architecture, and performance evaluation". In: *IEEE Transactions on Computers* 50.8 (2001), pp. 834–846 (cit. on pp. 10, 54).

[89] Bart Kienhuis and A. Jan. "Design space exploration of stream-based dataflow architectures : methods and tools /". In: (Jan. 1999) (cit. on pp. 8, 11).

[90] Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, et al. "The EM-X Parallel Computer: Architecture and Basic Performance". In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA '95. S. Margherita Ligure, Italy: Association for Computing Machinery, 1995, pp. 14–23 (cit. on p. 10).

[91] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. “Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 162–173 (cit. on p. 19).

[92] C. Kyriacou, P. Evripidou, and P. Trancoso. “Data-Driven Multithreading Using Conventional Microprocessors”. In: *IEEE Transactions on Parallel and Distributed Systems* 17.10 (2006), pp. 1176–1188 (cit. on p. 18).

[93] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. “GraphChi: Large-Scale Graph Computation on Just a PC”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 31–46 (cit. on p. 68).

[94] HP Labs. *COTSon: INFRASTRUCTURE FOR FULL SYSTEM SIMULATION*. https://sourceforge.net/projects/cotson/files/ (cit. on pp. 61, 62).

[95] Christopher Lauderdale, Mark Glines, Jihui Zhao, Alex Spiotta, and Rishi Khan. “Swarm: A unified framework for parallel-for, task dataflow, and distributed graph traversal”. In: *ET International Inc., Newark, USA* (2013) (cit. on pp. 12, 14).

[96] B. Lee, A.R. Hurson, and B. Shirazi. “A hybrid scheme for processing data structures in a dataflow environment”. In: *IEEE Transactions on Parallel and Distributed Systems* 3.1 (1992), pp. 83–96 (cit. on p. 8).

[97] Ben Lee and A.R. Hurson. “Issues in Dataflow Computing”. In: ed. by Marshall C. Yovits. Vol. 37. Advances in Computers. Elsevier, 1993, pp. 285–333 (cit. on p. 6).

[98] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014 (cit. on p. 79).

[99] João Lima and Nicolas Maillard. “Online mapping of MPI2 dynamic tasks to processes and threads”. In: *International Journal of High Performance Systems Architecture* 2 (Mar. 2009) (cit. on p. 28).

[100] Cheng Liu, Xinyu Chen, Bingsheng He, et al. “OBFS: OpenCL Based BFS Optimizations on Software Programmable FPGAs”. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 315–318 (cit. on p. 64).

[101] Wenyan Lu, Guihai Yan, Jiajun Li, et al. “FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 553–564 (cit. on p. 6).

[102] George Matheou and Paraskevas Evripidou. “FREDDO: an efficient Framework for Runtime Execution of Data-Driven Objects”. In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. 2016, pp. 265–273 (cit. on pp. 6, 10).

[103] George Matheou and Paraskevas Evripidou. “FREDDO: an efficient Framework for Runtime Execution of Data-Driven Objects”. In: *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. 2016, pp. 265–273 (cit. on pp. 14, 28).

[104] T. G. Mattson, R. Cledat, V. Cavé, et al. “The Open Community Runtime: A runtime system for extreme scale computing”. In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 2016, pp. 1–7 (cit. on pp. 15, 16, 56, 57, 59).

[105] Maxeler Technologies. *https://www.maxeler.com/index.html* (cit. on pp. 8, 9).

[106] Farnam Khalili Maybodi. “A Data-Flow Threads Co-Processor for MPSoC FPGA Clusters”. In: *Doctoral Thesis the University of Firenze* (2021) (cit. on pp. 12, 16).

[107] Veljko Milutinović, Jakob Salom, N Trifunović, and Roberto Giorgi. *Guide to dataflow supercomputing*. Springer, 2015 (cit. on p. 90).

[108] Lucas Morais, Vitor Silva, Alfredo Goldman, et al. "Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-Core Processor". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 861–872 (cit. on pp. 18, 19).

[109] R. S. Nikhil. "Can Dataflow Subsume von Neumann Computing?" In: *Proceedings of the 16th Annual International Symposium on Computer Architecture*. ISCA '89. Jerusalem, Israel: Association for Computing Machinery, 1989, pp. 262–272 (cit. on pp. 17, 18).

[110] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. "Stream-Dataflow Acceleration". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 416–429 (cit. on pp. 8, 11).

[111] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. "Heterogeneous Von Neumann/Dataflow Microprocessors". In: *Commun. ACM* 62.6 (2019), pp. 83–91 (cit. on p. 10).

[112] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, et al. "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation". In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2014, pp. 25–28 (cit. on pp. 64, 67).

[113] OCR. *Open Community Runtime v1.0*. `https://xstack.exascale-tech.com/git/public/ocr.git`. Accessed: 2021.01 (cit. on pp. 12, 15).

[114] Tayo Oguntebi and Kunle Olukotun. "GraphOps: A Dataflow Library for Graph Analytics Acceleration". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 111–117 (cit. on pp. 64, 67, 68).

[115] Open Source high Performance Computing. *https://www.open-mpi.org/* (cit. on pp. 22, 27).

[116] Opencilk monorepo repository. *https://github.com/OpenCilk/opencilk-project* (cit. on pp. 15, 27).

[117] G.M. Papadopoulos. "Monsoon: a dataflow computing architecture suitable for intelligent control". In: *Proceedings. 5th IEEE International Symposium on Intelligent Control 1990*. 1990, 292–298 vol.1 (cit. on pp. 6, 8–10).

[118] Oliver Pell and Vitali Averbukh. "Maximum Performance Computing with Dataflow Engines". In: *Computing in Science Engineering* 14.4 (2012), pp. 98–103 (cit. on p. 9).

[119] Oleksandr Pochayevets. "BMDFM: A Hybrid Dataflow Runtime Parallelization Environment for Shared Memory Multiprocessors". In: *MS thesis in Computer Engineering*. 2006 (cit. on p. 14).

[120] Antoniu Pop and Albert Cohen. "OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs". English. In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 2013) (cit. on p. 8).

[121] Portable Hardware Locality. *https://www.open-mpi.org/projects/hwloc/* (cit. on p. 24).

[122] Marco Procaccini. "A Data-Flow Execution Engine for Scalable Embedded Computing". In: *Doctoral Thesis the University of Siena* (2020) (cit. on pp. 16, 31).

[123] European Project. *Agile, eXtensible, fast I/O Module for the cyber-physical era*. `https://git.axiom-project.eu/`. Accessed: 2021.01 (cit. on pp. 61, 62).

[124] G. Quenot and B. Zavidovique. “A data-flow processor for real-time low-level image processing”. In: *Euro ASIC ’91*. 1991, pp. 92–95 (cit. on p. 8).

[125] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. “A Distributed Algorithm for Large-Scale Graph Partitioning”. In: *ACM Trans. Auton. Adapt. Syst.* 10.2 (June 2015) (cit. on p. 68).

[126] J. E. Requa and J. R. McGraw. “The Piecewise Data Flow Architecture: Architectural Concepts”. In: *IEEE Trans. Comput.* 32.5 (May 1983), pp. 425–438 (cit. on p. 8).

[127] *RFFT source code*. `https://github.com/AminSahebi/RFFT`. Accessed: 2022-01-01 (cit. on p. 88).

[128] Juan-Antonio Rico-Gallego and Juan-Carlos Díaz-Martín. “t-Lop: Modeling performance of shared memory MPI”. In: *Parallel Computing* 46 (2015), pp. 14–31 (cit. on p. 118).

[129] L. Roh and W.A. Najjar. “Design of storage hierarchy in multithreaded architectures”. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. 1995, pp. 271–278 (cit. on pp. 8, 12, 16, 17).

[130] Lucas Roh, Walild Najjar, Bhanu Shankar, and A. Böhm. “An Evaluation of Optimized Threaded Code Generation”. In: (Sept. 1995) (cit. on p. 17).

[131] Radu Rugina and Martin Rinard. “Recursion unrolling for divide and conquer programs”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2000, pp. 34–48 (cit. on p. 84).

[132] Amin Sahebi, Lorenzo Verdoscia, and Roberto Giorgi. “A Data-Flow Approach To Accelerate Real-Valued Fast Fourier Transform”. In: *HiPEAC ACACES-2019*. poster. Fiuggi, Italy, July 2019, pp. 155–158 (cit. on p. 91).

[133] B. Salami, K. Parasyris, A. Cristal, et al. “LEGaTO: Low-Energy, Secure, and Resilient Toolset for Heterogeneous Computing”. In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020, pp. 169–174 (cit. on p. 6).

[134] s Sampath, Bharat Sagar, C Subbaraya, and B.R. Nanjesh. “Performance Evaluation of Parallel Applications using MPI in Cluster Based Parallel Computing Architecture”. In: (Sept. 2013) (cit. on p. 27).

[135] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, Yoshinori Yamaguchi, and Yasuhito Koumura. “Thread-Based Programming for the EM-4 Hybrid Dataflow Machine”. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA ’92. Queensland, Australia: Association for Computing Machinery, 1992, pp. 146–155 (cit. on pp. 8, 10).

[136] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. “Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching”. In: Feb. 2019, pp. 320–329 (cit. on pp. 64, 70).

[137] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. “Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching”. In: Feb. 2019, pp. 320–329 (cit. on pp. 66–68, 71, 80).

[138] Jurij Šilc and Borut Robič. “Synchronous dataflow-based architecture”. In: *Microprocessing and Microprogramming* 27.1 (1989). Fifteenth EUROMICRO Symposium on Microprocessing and Microprogramming, pp. 315–322 (cit. on pp. 5, 6, 8, 12, 17).

[139] R. J. N. Silva, B. Goldstein, L. Santiago, et al. “Task Scheduling in Sucuri Dataflow Library”. In: *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. 2016, pp. 37–42 (cit. on p. 14).

[140] J. Strohschneider and K. Waldschmidt. "ADARC: a fine grain dataflow architecture with associative communication network". In: *Proceedings of Twentieth Euromicro Conference. System Architecture and Integration*. 1994, pp. 445–450 (cit. on p. 8).

[141] Joshua D. Suetterlein, Stéphane Zuckerman, and Guang Rong Gao. "An Implementation of the Codelet Model". In: *Euro-Par*. 2013 (cit. on pp. 12, 13).

[142] Z. Sun, X. Liu, and Z. Ji. "The Design of Radix-4 FFT by FPGA". In: *2008 International Symposium on Intelligent Information Technology Application Workshops*. Dec. 2008, pp. 765–768 (cit. on p. 89).

[143] Xubin Tan, Jaume Bosch, Daniel Jimenez-Gonzalez, et al. "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)*. 2016, pp. 225–234 (cit. on p. 19).

[144] Xubin Tan, Jaume Bosch, Miquel Vidal, et al. "General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 244–253 (cit. on p. 18).

[145] Konstantinos Tatas and Costas Kyriacou. "Implementation of a threaded dataflow multiprocessor using FPGAs". In: *2011 6th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2011, pp. 1–6 (cit. on p. 10).

[146] T. Temma, M. Iwashita, K. Matsumoto, H. Kurokawa, and T. Nukiyama. "Data flow processor chip for image processing". In: *IEEE Transactions on Electron Devices* 32.9 (1985), pp. 1784–1791 (cit. on pp. 7, 8).

[147] Tung Thanh-Hoang, Amirali Shambayati, Calvin Deutschbein, Henry Hoffmann, and Andrew A Chien. "Performance and energy limits of a processor-integrated fft accelerator". In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2014, pp. 1–6 (cit. on p. 90).

[148] The OpenMP API specification for parallel programming. *https://www.openmp.org* (cit. on p. 27).

[149] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. "Data-Driven and Demand-Driven Computer Architecture". In: *ACM Comput. Surv.* 14.1 (Mar. 1982), pp. 93–143 (cit. on p. 8).

[150] UG1023. *SDAccel Environment User Guide*. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1023-sdaccel-user-guide.pdf`. May 2019 (cit. on pp. 64, 66, 76).

[151] Theo Ungerer. "ASTOR - An architecture of special purpose processing units with distributed control and message passing". In: *Microprocessing and Microprogramming* 23.1 (1988), pp. 227–232 (cit. on p. 8).

[152] Theo Ungerer, Jurij Silc, and Borut Robic. "Beyond Dataflow". In: *Journal of Computing and Information Technology (cit@srce.hr); Vol.8 No.2* 8 (June 2000) (cit. on pp. 5, 6).

[153] Theo Ungerer, Jurij Silc, and Borut Robic. "Processor Architecture From Dataflow to Superscalar and Beyond". In: *Springer, Berlin, Heidelberg* 8 (June 1999) (cit. on p. 10).

[154] Rex Vedder and Dennis Finn. "The Hughes Data Flow Multiprocessor: Architecture for Efficient Signal and Data Processing". In: *Proceedings of the 12th Annual International Symposium on Computer Architecture*. ISCA '85. Boston, Massachusetts, USA: IEEE Computer Society Press, 1985, pp. 324–332 (cit. on pp. 7, 8).

[155] Arthur H. Veen. “Dataflow machine architecture”. In: *ACM Comput. Surv.* 18 (1986), pp. 365–396 (cit. on p. 6).

[156] Arthur H. Veen and Reinier Van Den Born. “The RC compiler for the DTN dataflow computer”. In: *Journal of Parallel and Distributed Computing* 10.4 (1990). Data-flow Processing, pp. 319–332 (cit. on p. 8).

[157] L. Verdoscia and R. Giorgi. “A Data-Flow Soft-Core Processor for Accelerating Scientific Calculation on FPGAs”. In: *Mathematical Problems in Engineering* 2016.1 (Apr. 2016). article ID 3190234, pp. 1–21 (cit. on pp. 6, 15).

[158] Lorenzo Verdoscia and Roberto Giorgi. “A data-flow soft-core processor for accelerating scientific calculation on FPGAs”. In: *Mathematical Problems in Engineering* 2016 (2016) (cit. on pp. 6, 90).

[159] Lorenzo Verdoscia, Amin Sahebi, and Roberto Giorgi. “A Data-Flow Methodology for Accelerating FFT”. In: *The 8th Mediterranean Conference on Embedded Computing - MECO* (2019) (cit. on pp. 6, 92).

[160] Lorenzo Verdoscia and Roberto Vaccaro. “A high-level dataflow system”. In: *Computing* 60.4 (1998), pp. 285–305 (cit. on p. 90).

[161] Mário Véstias and Horácio Neto. “Trends of CPU, GPU and FPGA for high-performance computing”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–6 (cit. on p. 1).

[162] Keval Vora. “LUMOS: Dependency-Driven Disk-based Graph Processing”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 429–442 (cit. on pp. 68, 70).

[163] Dong Kai Wang and Nam Sung Kim. “DiAG: A Dataflow-Inspired Architecture for General-Purpose Processors”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 93–106 (cit. on p. 6).

[164] John Wawrzynek, David Patterson, Mark Oskin, et al. “RAMP: Research Accelerator for Multiple Processors”. In: *IEEE Micro* 27.2 (2007), pp. 46–57 (cit. on p. 8).

[165] Haitao Wei, Stéphane Zuckerman, Xiaoming Li, and Guang Gao. “A Dataflow Programming Language and Its Compiler for Streaming Systems”. In: *Procedia Computer Science* 29 (Dec. 2014), pp. 1289–1298 (cit. on p. 8).

[166] S. Weis, A. Garbade, B. Fechner, et al. “Architectural Support for Fault Tolerance in a Teradevice Dataflow System”. English. In: *Springer Int.l Journal of Parallel Programming* 44.2 (Apr. 2016), pp. 208–232 (cit. on p. 54).

[167] Michael Wilde, Mihael Hategan, Justin M. Wozniak, et al. “Swift: A language for distributed parallel scripting”. In: *Parallel Computing* 37.9 (2011). Emerging Programming Paradigms for Large-Scale Scientific Computing, pp. 633–652 (cit. on p. 14).

[168] J. M. Wozniak, T. G. Armstrong, M. Wilde, et al. “Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing”. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 2013, pp. 95–102 (cit. on pp. 6, 14).

[169] Taoran Xiang, Yujing Feng, Xiaochun Ye, et al. “Accelerating CNN Algorithm with Fine-Grained Dataflow Architectures”. In: *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2018, pp. 243–251 (cit. on p. 6).

[170] UG1270 Xilinx documentation. *Vivado HLS Optimization Methodology Guide*. https://www.xilinx.com/support/documentation. May 2017 (cit. on p. 76).

[171] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. "An Efficient Graph Accelerator with Parallel Data Conflict Management". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus: Association for Computing Machinery, 2018 (cit. on p. 64).

[172] R. Yavne. "An Economical Method for Calculating the Discrete Fourier Transform". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). San Francisco, California: ACM, 1968, pp. 115–125 (cit. on p. 89).

[173] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. "Hybrid Dataflow/von-Neumann Architectures". In: *IEEE Transactions on Parallel and Distributed Systems* (2014) (cit. on pp. 6, 53).

[174] Yijun Liu and S. Furber. "A low power embedded dataflow coprocessor". In: *IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*. 2005, pp. 246–247 (cit. on p. 6).

[175] T. Yokota, Hiroshi Matsuoka, Kazuaki Okamoto, et al. "A prototype router for the massively parallel computer RWC-1". In: *Proceedings of ICCD '95 International Conference on Computer Design. VLSI in Computers and Processors* (1995), pp. 279–284 (cit. on p. 10).

[176] Richard Yoo, Christopher Hughes, Konrad Lai, and Ravi Rajwar. "Performance evaluation of Intel® Transactional Synchronization Extensions for high-performance computing". In: Nov. 2013 (cit. on p. 28).

[177] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. "High-Throughput and Energy-Efficient Graph Processing on FPGA". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, pp. 103–110 (cit. on p. 64).

[178] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. "Optimizing memory performance for FPGA implementation of pagerank". In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2015, pp. 1–6 (cit. on p. 64).

[179] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. "HitGraph: High-throughput Graph Processing Framework on FPGA". In: *IEEE Transactions on Parallel and Distributed Systems* 30.10 (2019), pp. 2249–2264 (cit. on pp. 65, 68).

[180] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. "An FPGA Framework for Edge-Centric Graph Processing". In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. CF '18. Ischia, Italy: Association for Computing Machinery, 2018, pp. 69–77 (cit. on p. 64).

[181] Xiaowei Zhu, Wentao Han, and Wenguang Chen. "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning". In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. Santa Clara, CA: USENIX Association, 2015, p. 375386 (cit. on pp. 68, 70, 79).

[182] Yu Zou and Mingjie Lin. "GridGAS: An I/O-Efficient Heterogeneous FPGA+CPU Computing Platform for Very Large-Scale Graph Analytics". In: *2018 International Conference on Field-Programmable Technology (FPT)*. 2018, pp. 246–249 (cit. on pp. 67, 68).

[183] Stéphane Zuckerman. *DARTS: An asynchonous fine-grained runtime based on the codelet model*. https://github.com/szuckerm/DARTS. Accessed: 2021.01 (cit. on pp. 12, 15, 57, 59).

[184] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. "Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper". In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 64–69 (cit. on pp. 12, 14, 15).

Appendix A - Publications

A

A.1 Peer-reviewed Conference Papers

Best Paper Award L. Verdoscia, A. Sahebi and R. Giorgi, "A Data-Flow Methodology for Accelerating FFT," 2019 8th Mediterranean Conference on Embedded Computing (MECO), 2019, pp. 1-4. https://doi.org/10.1109/MECO.2019.8760044

Giorgi R., Procaccini M., Sahebi A. (2021) DRT: A Lightweight Runtime for Developing Benchmarks for a Dataflow Execution Model. Architecture of Computing Systems. ARCS 2021. Lecture Notes in Computer Science, vol 12800. Springer, Cham. https://doi.org/10.1007/978-3-030-81682-7_6

A.2 Workshops and Posters

Sahebi, Amin, and Roberto Giorgi. "GLUON, The High-Speed Inexpensive and Easy Interconnect Solution", HiPEAC ACACES, 2020, Fiugi, Italy.

Sahebi, A., Verdoscia, L., Giorgi, R. (2019). A data-flow approach to accelerate real-valued fast fourier transform. HiPEAC ACACES, 2019, 155-158.

Appendix B - Parallel Programming Models Notes

A

A.1 MPI profiling with mpiP

mpiP[1] is an open-source library that provides lightweight profiling of MPI applications. It uses statistical sampling to record profiling data, thus is not as accurate as other profiling tools but it is lightweight and trace files are much smaller particularly for very large MPI process runs. No code changes are required to use mpiP but a re-link is required. From the performance data provided by the tools, the POP performance metrics can be calculated. The code should be compiled with the -g flag and linked with the mpiP library. The link line is shown below:

```
#!/bin/bash

mpicc -g bmm-mpi.c -o bmm-mpi -L/user/mpiP/installation/dir/ -lmpiP
export LD_PRELOAD=/user/mpiP/installation/dir/libmpiP.so
export MPIP="-c -d -p -y -k 0"

export LD_LIBRARY_PATH=/user/mpiP/installation/dir/:$LD_LIBRARY_PATH
```

Listing A.1: How to compile and set environment variables for mpiP profiling tool

The code is executed as normal with mpirun and the performance report is saved to a file. The file name will be printed at the end of the application run. The performance report contains the following sections:

1) **MPI%**: The percentage of time each rank is spending in MPI (which includes MPI-IO) and non-MPI;

2) **Sites**: Call sites which are locations in the code containing MPI calls;

3) **AppTime**: The overall time that application spent in each rank.

4) The top 20 call sites that spend the **most time** in MPI;

5) The top 20 calls sites that send the **most data**;

6) MPI call site statistics which include number of times called, averagemaxmin time spent, and percentage of time in code and MPI;

7) MPI call site statistics which include number of bytes sent, and averagemaxmintotal bytes sent.

```
call MPI_INIT( ierr )
call MPI_PCONTROL( 0 )       ! 1. disable profiling
[ ... ]                      ! 2. The initial body of the Application
call MPI_PCONTROL( 1 )       ! 3. enable profiling
   for i = 1, Ni             ! 4. region of interest that is doing
   [ ... ]                   ! computation and communication
   end for
call MPI_PCONTROL( 0 )       ! 5. disable profiling
[ ... ]                      ! 6. some other computation
call MPI_FINALIZE( ierr )
```

Listing A.2: The command specifications to specify the region of interest to profile with mpiP

Regions of interest (ROI) in the code can be enclosed with MPI_PCONTROL to switch onoff profiling. The example code below shows how to control profiling of regions of interest: The following Listing A.3 shows the experimental results performed on tfx2 machine and profiles with mpiP, the MPIin the whole algorithm, as you can see the percentage varies from rank to rank and this is a good example of weak scheduler of the Linux system (of course this behavior depends on the Linux kernel and the tuning parameters, therefore using batch manager which perform tuned scheduling with MPI will eliminate this problem). Note that MPI time also includes MPI-IO subroutine calls (including parallel NetCDF and parallel HDF5) but not POSIX I/O. The AppTime field includes MPI time, so to calculate user-code time, It is subtracted MPITime from AppTime. The data in second part and UsrTime have been manually calculated and are not included in the output of mpiP.

```
@--------------MPI Time (seconds) ---------------
Task     AppTime     MPITime     MPI%       UsrTime
0           17        1.55       9.13        15.45
1           17         2.2      12.94        14.8
2           17        2.14      12.60        14.86
3           17        2.18      12.84        14.82
4           17        2.17      12.76        14.83
5           17        2.07      12.22        14.93
6           17        1.03       6.05        15.97
7           17        1.39       8.21        15.61
8           17        1.64       9.68        15.36
9           17       0.753       4.43        16.24
10          17        2.04      12.02        14.96
11          17        2.39      14.09        14.61
12          17        4.03      23.73        12.97
13          17        4.06      23.91        12.94
14          17        4.06      23.91        12.94
total       255       33.7      13.23(Avg)   221.29
@--------------Manually Calculated --------------
max         17        4.06      23.91        16.26
min         17        0.753     4.43         12.94
avg         17        2.24      13.23        14.7
```

Listing A.3: The output of profiling the OpenMPI benchmark with mpiP from LLNL repository. The results show the MPI overhead and UsrTime compared to the overall execution time.

The other kind of report that can be collected with mpiP profiling is shown below, in this profiling experiment, we changed the environment variables for the mpiP and let it collect the aggregate data based on each type of MPI calls, like Send, Receive, Barrier and Bcast, the report is shown in listing A.4.

```
@ mpiP
@ Command : ./bmm-mpi 2000
@ Version                  : 3.5.0
@ MPIP Build date          : Jul 26 2021, 23:39:46
@ Start time               : 2021 08 30 14:46:58
@ Stop time                : 2021 08 30 14:47:15
@ Timer Used               : PMPI_Wtime
@ MPIP env var             : -c -d -p -y -k 0
@ Collector Rank           : 0
@ Collector PID            : 1939
@ Final Output Dir         : .
@ Report generation        : Single collector task
---------------------------------------------------------------------------
@--- Task Time Statistics (seconds) ---------------------------------------
---------------------------------------------------------------------------
                     AppTime           MPITime   MPI%   App Task   MPI Task
Max                17.028874          0.970338                 0         11
Mean               16.994485          0.824369
Min                16.990061          0.189923                 2          0
Stddev              0.009534          0.188070
Aggregate         254.917268         12.365537   4.85
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                     Site        Time     App%    MPI%       Count
Recv                      135    8.44e+03     3.31   68.29          28
Send                      148    1.87e+03     0.73   15.13          28
Bcast                       8    1.77e+03     0.69   14.29          30
Barrier                     7         283     0.11    2.29          15
---------------------------------------------------------------------------
@--- Aggregate Point-To-Point Sent (top twenty, descending) ---------------
---------------------------------------------------------------------------
Call                     MPI Sent %             Comm Size            Data Size
Send                           11.1          8 -         15    2097152 -   4194303
---------------------------------------------------------------------------
@--- End of Report --------------------------------------------------------
--------------------------------------------------------------------------
```

Listing A.4: The aggregate collective report from mpiP profiling, matrix size 2000 and number of workers 15

The graphical plot of the MPI overhead (MPI% section in Listing A.3) has been illustrated in Figure A.1.

Based on the raw data profiled with mpiP, the user can calculate useful metrics to see how the profiling help to understand better the performance of the MPI application. As listed below there are three metrics that can be achieved using this raw data,

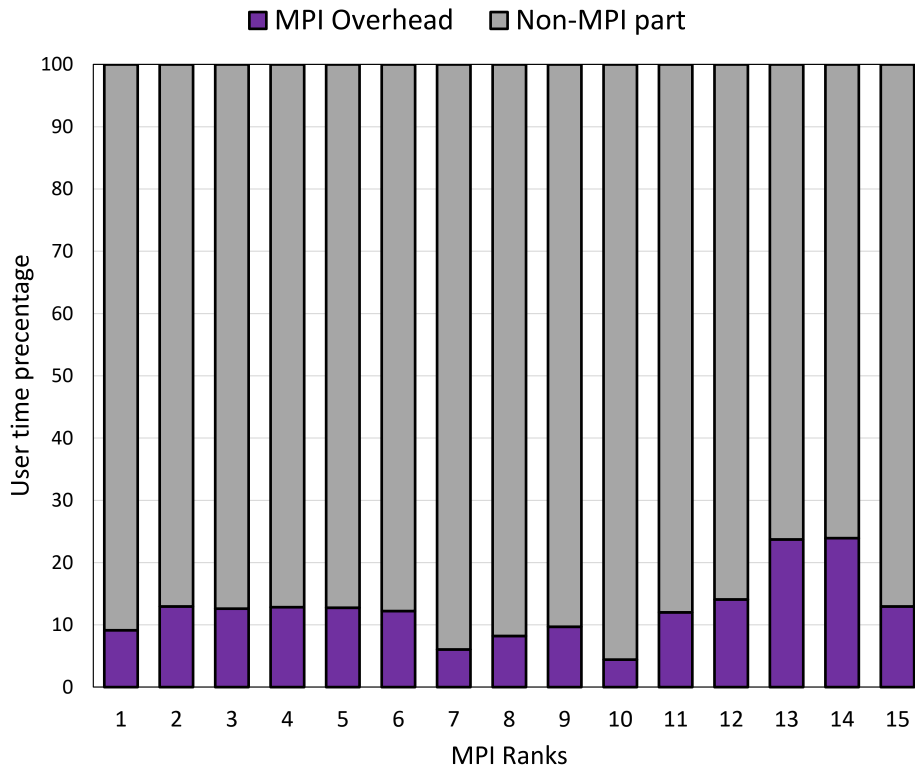


Fig. A.1.: Results of profiling BMM with MPI shows an example of the MPI overhead over 15 cores, it can be seen from the results that most of the workers have unbalanced MPI overhead across available resources, and in this case not having a ideal speedup. (Experiment has been done on TFX2)

1) Load Balance (LB)

2) Communication Efficiency (CommE)

3) Parallel Efficiency (PE)

The mentioned metrics can be calculated as follows: The **load balance (LB)** metric can be calculated by the average user-code time (average of all MPI processes) divided by the maximum user-code time. For the above example, this is going to be calculated as follows:

$$\text{LoadBalance} = \frac{\textit{Average user code}}{\textit{Maximum user code}} \times 100 = \frac{14.7}{16.26} = 90.4\% \tag{A.1}$$

The **communication efficiency (CommE)** can be calculated by maximum user code time divided by the runtime:

$$\text{CommEffeciency} = \frac{\textit{Maximum user code}}{\textit{Runtime}} \times 100 = \frac{16.26}{17} = 95.6\% \tag{A.2}$$

The runtime can be measured in the mpiP output log, just need to subtract the two stop and start timers as follows:

```
@ mpiP
@ Command          : ./bmm-mpi 1728
@ Version          : 3.5.0
@ Start time       : 13:54:49
@ Stop time        : 13:55:06
--> @ Runtime      : 17.0
```

Listing A.5: The mpiP output log and how to calculate manually the runtime, this runtime time is not a part of the mpiP output log.

The last metric that can be investigated is **Parallel Efficiency (PE)**; The parallel efficiency is a product of load balance and communication efficiency (the last two metrics):

$$\text{ParallelEffeciency} = (LB \times CommE)/100 = (90.4 \times 95.6)/100 = 86.42\% \tag{A.3}$$

The above metrics can be measured with different problem sizes, different number of processors and different environments and the user can plot them in a figure to see the behavior of the algorithm on different aspects and learn lessons from this numbers.

A.2 Pure OpenMPI on Hyper threaded Hardware

As an extension to our measurement, we studied the impact of dedicating MPI processes to one Hardware thread, it turns out for small sizes of the task (i.e., Matrix size of 512) it works fine. However, while the problem size is increased, the impact of using hwthread will slow down the performance. Note that here we leave everything to Linux Scheduler to schedule the acquisition of hwthread on cores and migrate the jobs between hwthread, and the OpenMPI options left as default except `-use-hwthread-cpus`, since pining the MPI process to the specific hwthread and use MPI options like `-map-by core/hwthread` or bind them using `-bind-to core/hwthread` will impact more negatively on the overall performance. There are some `-MCA`[1] options to give more direction to the MPI processing like `-mca btl self,sm`, however these options are dependent to the algorithm implementation and the hardware used, In our case none of these option were changed the execution performance.

Figure A.2 shows the speedup for BMM on an hyper threaded machine, the specification of the machine has been listed in Table A.1. As can be seen from Figure A.2 the trend of the speedup for all the chosen matrix sizes is close to the linear speedup, this trend continues until each hwthread from each core is used by the execution model, then (after 16 No. of processors in this case) the second hwthread on each core will be used in our experiment. It can be seen a significant slow down with big number of matrix size 4096 and no impact within small size 512. It can be due the fact that this implementation of pure

[1] An MCA framework uses the MCA's services to find and load component sat run time. An MCA component is a stand-alone collection of codes that can be inserted into the Open MPI code base at run-time and/or compile-time

Tab. A.1.: The configurations of the experiment that has been done on **TFX3** a Single Multi-core Machine.

Title	Description
Operating system	`Ubuntu 20.04.3 LTS`
OpenMPI	`version 2.1.1`
OpenMPI options	`mpirun -n $j -use-hwthread-cpus -hostfile ~/hosts ./executable $i`
GCC Version	`version 9.3.0`
GCC Flags	`-O3`
Kernel	`5.11.0-27-generic`
CPUs	`32`
Processor	`AMD Ryzen Threadripper 1950X 16-Core`
Hyper threading	`Enabled (2 Threads per Core)`
L1d cache:	`512 KiB`
L1i cache:	`1 MiB`
L2 cache:	`8 MiB`
L3 cache:	`32 MiB`
Network	`Single Multi-core Machine`

two-sided point-to-point OpenMPI can not exploit the shared memory for two hwthread in one core (L1,L2 and L3). This must be developed with the especial primitives of OpenMPI to exploit shared memory environments as discussed in detail in [87]. On the other hand, by monitoring the execution of the experiment by Linux tools like `htop`, can be seen that the kernel activity suddenly increases a lot once the second hwthread of the core is called by the execution model of the experiment. This works[87, 128, 73] discuss the implementation of OpenMPI to manage the shared memory resources in hyper threading environments. One can try OpenMPI+X, where X can be OpenMPI as a well-known shared memory programming model or shared-memory MPI implementation of the given algorithm.

A.3 Scripts

To do the experiments I use some scripts to collect the data from the execution, I keep all the benchmarks with the same scripts, the output will be a csv file that includes the execution time for ‘average’, ‘minimum’ and ‘maximum’ numbers, and a PNG file which is output of the GNUPLOT for the given csv file, I plot the bar chart plus error bars for each experiment. The user can collect the csv files and plot different figures based on the needs. A sample of the script is shown in the listing A.6,

```
#!/bin/bash
host=`hostname -s`
release=`lsb_release -r | awk 'NR==1{printf $2}' | cut -c -2`
distro=`lsb_release -d  | awk 'NR==1{print $2.$3}' | cut -c -1`
date=`date +%Y%m%d%H%M | cut -c 3-`
core=`getconf _NPROCESSORS_ONLN`
wdir=`pwd`
toolname=`pwd | awk -F/ '{print $5$6}'`
cpath="run_bmm.sh" #command path
size="3978" # input range
```

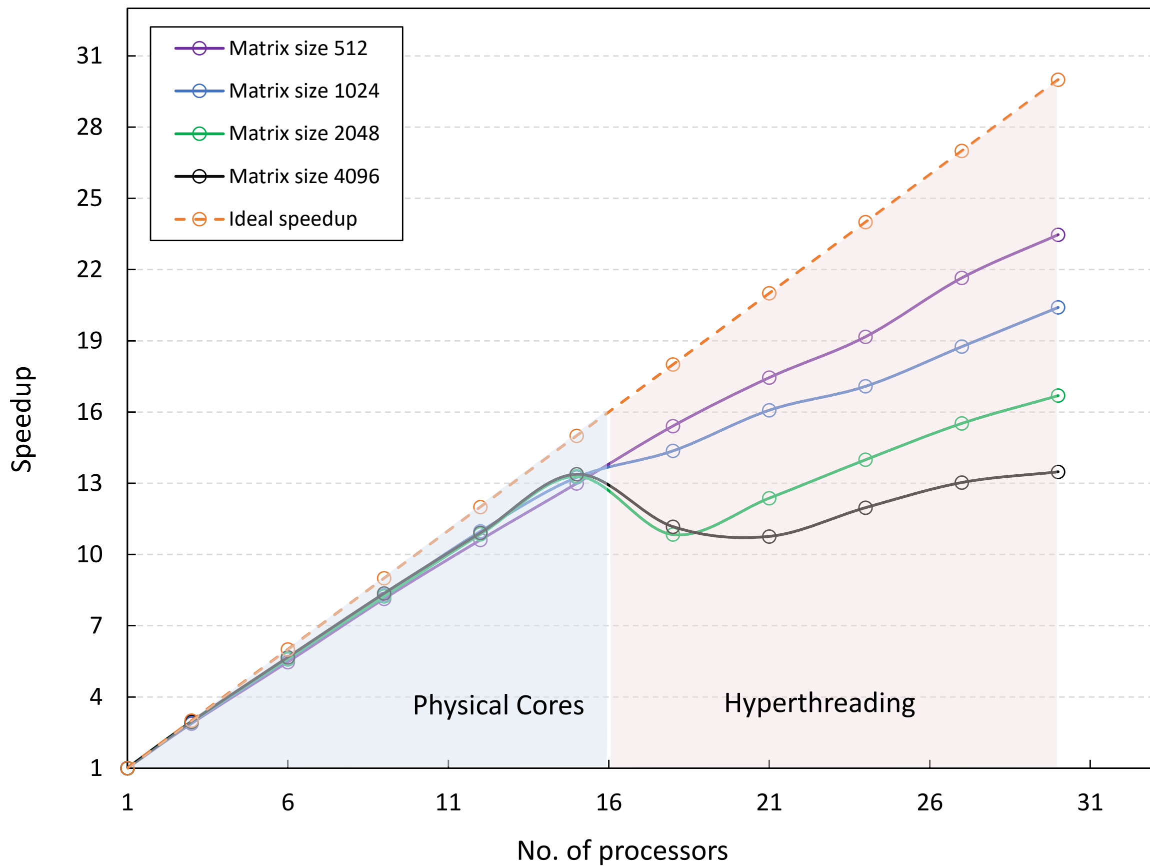


Fig. A.2.: Results obtained from the BMM with OpenMPI with different Matrix sizes. It turns out while the second thread of the core is calling, the performance decreases due to the shared-memory resources that Pure OpenMPI implementation can not manage it and moreover kernel scheduler is not optimized for this execution model, However on logical cores the trend is almost linear.

```
irange="1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20"
mstring="EXECUTION TIME" # matching string
mcolumn="5" # matching column
nrep="10" #number of repetition of the test
if [ ! -d "$wdir" ]; then
    echo "ERROR: cannot find '$wdir'"; exit 1
fi
if [ ! -s "$wdir/$cpath" ]; then
    echo "ERROR: cannot find '$cpath' in '$wdir'"; exit 1
fi
cd $wdir
FILENAME=$toolname-$host$distro$release"C"$core-$date
re='^[0-9]+$'
fl='^[+-]?[0-9]+\.?[0-9]*$'
echo "pwd=`pwd`"
echo "create log and plot file name: $FILENAME.csv"
echo "logs and plots will store into $wdir/logs/"
echo ""
declare -a value
for i in $irange; do
    a="0"; min=""; max="0"
    for r in `seq $nrep`; do
      output=`./$cpath $size $i  2>error.log`
      val=`echo "$output"|awk "/$mstring/{print \\$5}" c=$mcolumn`
      if ! [[ $val =~ $re ]] ; then echo "ERROR: The output is not a number
    ! (input=$i,output='$val')" >&2; exit 1; fi
      if [ 1 = `echo "$val > $max"|bc` ]; then max="$val"; fi
      if [ "$min" = "" ]; then min="$val"; a="$val" else
      if [ 1 = `echo "$val < $min"|bc` ]; then min="$val"; fi fi
      i1=`expr $i - 1`
      a=`echo "define trunc(x) { auto s; s=scale; scale=0; x=x/1; scale=s;
    return x } trunc($a * $i1 / $i + $val / $i)"|bc -l`
  if [ "$i1" = "0" ]; then value="$a" fi
    b=`echo "define trunc(x) { scale=2; x=x/1.0;  return x } trunc($value/
    $a)"|bc -l` #speedup
    done
    echo "$i,$a,$min,$max,$b" #printf the index, avg, min, max and speedup
    printf %s'\n' "$i,$a,$min,$max,$b" >> $FILENAME.csv done
```

Listing A.6: The script to iterate the experiment of a loop and collect the numbers, producing a suitable csv file that can be used for reports

To plot the desired 'csv' file as the output of the experiment, I prepared a GNUPLOT script as can be seen from Listing A.8, which is prepared to plot the speed and execution time. Dealing with GNUPLOT is straightforward, however, there are some ticks that the user should carefully consider them to have a nice graph.

```
for FILE in ${FILENAME}.csv; do
    gnuplot <<- EOF
  set xlabel "Numebr of workers"
    set ylabel "Execution time (ms)"
    set key left
```

```
  set term png
    set style histogram cluster gap 1
    set style fill solid 0.5
    set boxwidth 0.9
    set style histogram errorbars linewidth 1
    set errorbars linecolor black
    red = "#FF0000"; green = "#00FF00"; blue = "#0000FF"; skyblue = "#87
   CEEB" ; violet = "#FF00FF"; purple = "#440154" ;
    set grid ytics
    set format y '10^{%L}'
    set logscale y
    set autoscale x
    set yrange [1:]
  set output "${FILENAME}.png"
    set datafile separator ","
    set style data histogram
    plot "${FILENAME}.csv" using 2:3:4:xtic(1) title "Execution time (ns)"
   linecolor rgb purple linewidth 0
    EOF
done
```

Listing A.7: The script to fetch numbers from a generated csv file and plot it using GNUPLOT

```
#!/usr/bin/gnuplot -p
set terminal pngcairo size 800,800
set size square
matrixsize='45+5'
filename='RFIB-CILK-TFX3'
set title sprintf('%s, size %s', filename,matrixsize)
set xlabel "Number of Workers" font "Times-Roman,16"
set ylabel "Execution time (s)" font "Times-Roman,16" offset 0
set style increment default
set offsets 0.05, 0.05, 0, 0
set style function linespoints
set bmargin  6
unset colorbox
set key left
#set term png
set logscale y
set grid x y
set output sprintf('%s.png', filename)
set datafile separator ","
set style line 101 lc black lt 1 lw 2 pt 6 pi -1 ps 1
set key samplen 3 spacing 1.5 font ",12"
set key box width -2
set pointintervalbox 3
plot "RFIB-cilk-tfx3.csv" using 2:xtic(int($0)%2==0 ? strcol(1):'') with
    linespoints ls 101 title "execution time"
```

Listing A.8: The GNUPLOT script uses to create graphs using the csv file output. The user should take care of the name of the csv file and other decorations such as title, fonts, etc.