ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

GABRIELLA LOPES ANDRADE

IMPROVING PARALLEL PROGRAMMING ASSESSMENT: CHALLENGES, METHODS, AND OPPORTUNITIES IN CODING PRODUCTIVITY

Porto Alegre
2023

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

IMPROVING PARALLEL PROGRAMMING ASSESSMENT: CHALLENGES, METHODS, AND OPPORTUNITIES IN CODING PRODUCTIVITY

GABRIELLA LOPES ANDRADE

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Dr. Luiz Gustavo Leão Fernandes
Co-Advisor: Prof. Dr. Dalvan Jair Griebler

Porto Alegre
2023

Ficha Catalográfica

A553i Andrade, Gabriella Lopes

Improving parallel programming assessment : Challenges, methods, and opportunities in coding productivity / Gabriella Lopes Andrade. – 2023.
199p.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.
Coorientador: Prof. Dr. Dalvan Jair Griebler.

1. Coding metrics. 2. Development effort. 3. GPU. 4. Multi-core. 5. Parallel computing. I. Fernandes, Luiz Gustavo Leão. II. Griebler, Dalvan Jair. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

GABRIELLA LOPES ANDRADE

IMPROVING PARALLEL PROGRAMMING ASSESSMENT: CHALLENGES, METHODS, AND OPPORTUNITIES IN CODING PRODUCTIVITY

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 29, 2023.

COMMITTEE MEMBERS:

I dedicate this work to my family.

“We’ve all got both light and dark inside us. What matters is the part we choose to act on. That’s who we really are.”
(Harry Potter and the Prisoner of Azkaban)

ACKNOWLEDGMENTS

Firstly, I would like to thank God, for without Him, nothing would be possible. God has blessed my path every day.

I thank my parents, Pedro Andrade and Rozangila Andrade, for all their support during these years. I especially thank my boyfriend, Gabriel Quintana, who always encouraged me to follow my dreams.

I thank my advisors, Luiz Gustavo Fernandes and Dalvan Griebler, for their support during this journey. I also thank the support of all my friends and colleagues.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

MELHORANDO A AVALIAÇÃO DA PROGRAMAÇÃO PARALELA: DESAFIOS, MÉTODOS E OPORTUNIDADES NA PRODUTIVIDADE DE CODIFICAÇÃO

RESUMO

O desenvolvimento de aplicações paralelas não é uma tarefa fácil, pois os desenvolvedores devem lidar com várias questões como a implementação da sincronização de dados, a divisão do problema de computação entre as threads e a exploração da concorrência. Para facilitar essa tarefa, surgiram novas Interfaces de Programação Paralela (IPPs). Ao avaliar essas IPPs, a maioria dos estudos na área de programação paralela se concentra na avaliação do tempo de execução e desempenho dessas IPPs. Entretanto, a produtividade é um fator importante que, juntamente com a eficácia e a satisfação do usuário, são indicadores de usabilidade. A partir da avaliação da produtividade e da usabilidade, é possível continuar aumentando as abstrações do paralelismo e criar IPPs melhores e simples de usar sem comprometer o desempenho das aplicações. Logo, o principal objetivo dessa tese de doutorado é prover metodologias e técnicas para melhorar e suportar a avaliação da produtividade na área de programação paralela. Para atingir esse objetivo, inicialmente conduzimos uma revisão da literatura para descobrir como a usabilidade e produtividade tem sido avaliada na área de programação paralela. A partir dessa revisão identificamos que a fim de avaliar a produtividade na programação paralela, alguns pesquisadores estão realizando estudos envolvendo pessoas, geralmente desenvolvedores de aplicações, os quais demandam certo tempo para serem planejados e executados. Por outro lado, alguns pesquisadores têm se concentrado no uso de métricas de Engenharia de Software (por exemplo, CCN, COCOMO II e Halstead), as quais não foram projetadas para avaliar especificamente o desenvolvimento de aplicações paralelas. Em relação ao processo de experimentação, nessa tese de doutorado, apresentamos uma metodologia para orientar outros pesquisadores de programação paralela durante o planejamento, execução e análise dos resultados dos experimentos. Para validar essa metodologia, conduzimos experimentos com iniciantes em programação paralela ao explorar o paralelismo em aplicações de processamento de stream em ambientes multi-core e o paralelismo de dados em arquiteturas com GPU. Em relação às métricas de codificação, realizamos um estudo com o objetivo de verificar a eficácia dessas métricas ao avaliar a produtividade de IPPs. A partir desse estudo, identificamos uma série de métricas populares na área de Engenharia de Software ainda não exploradas na área de programação paralela, incluindo o modelo de Putnam, Pontos de Função, Pontos de Casos de Uso e Planning Poker. Para identificar as limitações e oportunidades de melhorias dessas métricas, verificamos a acurácia ao estimar o tempo de desenvolvimento de aplicações paralelas. Além disso, conduzimos uma pesquisa de opinião com desenvolvedores de aplicações paralelas para identificar os fatores que impactam na produtividade de desenvolvimento. Nosso objetivo era propor melhorias para as métricas de codificação com base nos fatores identificados. Os resultados dos experimentos com desenvolvedores iniciantes mostraram que, conforme o esperado, IPPs com um nível mais alto de abstração tendem a aumentar a produtividade do desenvolvedor em ambientes multi-core e GPU. Esse resultado também foi confirmado através da pesquisa de opinião realizada. Além disso, os resultados desta pesquisa de opinião confirmaram que a experiência do desenvolvedor é um dos principais fatores que influenciam o desenvolvimento de aplicações paralelas. Os resultados da avaliação das métricas de codificação mostraram que o Planning Poker se mostrou uma métrica promissora, pois considera as opiniões de desenvolvedores experientes ao estimar o esforço de desenvolvimento. Nesse sentido, nós propusemos uma modificação à métrica Planning Poker ao considerar a opinião de apenas um desenvolvedor ao invés de uma equipe de desenvolvimento. Os resultados mostraram que o Planning Poker é um método eficaz e que exige menos esforço para ser utilizado na prática em comparação com experimentos controlados com estudantes que visam coletar dados de tempo de desenvolvimento. Logo concluímos, que essa métrica pode ser usado como um substituto para medir o tempo de desenvolvimento de aplicações paralelas.

Palavras-Chave: Esforço de desenvolvimento, Computação paralela, GPU, Métricas de codificação, Multi-core.

IMPROVING PARALLEL PROGRAMMING ASSESSMENT: CHALLENGES, METHODS, AND OPPORTUNITIES IN CODING PRODUCTIVITY

ABSTRACT

Developing parallel applications is a challenging task because the developers must be able to deal with several issues, such as implementing data synchronization, dividing the computation problem among threads, and exploiting concurrency. New Parallel Programming Interfaces (PPIs) have emerged to facilitate this task. When evaluating these IPPs, most studies in the parallel programming area focus on assessing the execution time and performance of these IPPs. However, productivity is an important factor that, together with effectiveness and user satisfaction, are usability indicators. From evaluating productivity and usability, it is possible to continue to increase the abstractions of parallelism and create better and simple-to-use PPIs without compromising application performance. Therefore, the main goal of this Ph.D. thesis is to provide methodologies and techniques to improve and support productivity evaluation in parallel programming. To achieve this goal, we initially conducted a literature review to determine how usability and productivity have been evaluated in parallel programming. From this review, we identified that to assess productivity in parallel programming, some researchers are conducting studies with people, usually application developers, which require some time to be planned and executed. On the other hand, some researchers have focused on using Software Engineering metrics (for example, CCN, COCOMO II and Halstead), which were not designed to evaluate parallel application development specifically. Regarding the experimentation process, in this Ph.D. thesis, we presented a methodology to guide other parallel programming researchers during the planning, execution, and analysis of experiment results. We conducted experiments with beginners in parallel programming to validate this methodology by exploring parallelism in stream processing applications in multi-core environments and data parallelism in GPU architectures. Regarding coding metrics, we conducted a study to verify the effectiveness of these metrics when evaluating the productivity of IPPs. From this study, we identified some popular metrics in the Software Engineering area not yet explored in the parallel programming area, including Putnam's model, Function Points, Use Case Points, and Planning Poker. To identify the limitations and opportunities for improvement of these metrics, we verified their accuracy when estimating the development time of parallel applications. Furthermore, we conducted a survey with parallel application developers to identify the factors that impact development productivity. We aimed to propose improvements to coding metrics based on the identified factors. The results of the experiments with beginners showed that, as expected, PPIs with a higher level of abstraction tend to increase developer productivity in both multi-core and GPU environments. This result was also confirmed by surveying parallel programming developers. Furthermore, the results of this survey confirmed that the developers' experience is one of the main factors influencing parallel application development. The evaluation of the coding metrics showed that Planning Poker proved to be a promising metric because it considers the opinions of experienced developers when estimating the development effort. In this regard, we proposed modifying the Planning Poker metric by considering the opinion of only one developer instead of a development team. The results showed that Planning Poker is an effective method that requires less effort to use in practice compared to controlled experiments with students that aim to collect development time data. Therefore, this metric could be used as an alternative to measuring development time for parallel applications.

Keywords: Coding metrics, Development effort, GPU, Multi-core, Parallel computing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACAP	Analyst Capability
AEXP	Application experience
AFP	Adjusted Function Point
ALU	Arithmetic and logic unit
APEX	Applications Experience
API	Application Programming Interface
ASLOC	Added Source Lines of Code
CARI	Collective Asynchronous Remote Invocation
CCD	Commented Code Detector
CCN	Cyclomatic Complexity Number
CM	Complexity Metrics
CMMI	Capability Maturity Model Integration
COCOMO	Construction Cost Model
CPLX	Complexity of the Product
CPU	Central Process Unit
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DATA	Database Size
DET	Data Element Type
DOCU	Documentation match to life-cycle needs
DRAM	Dynamic random-access memory
DSL	Domain-Specific Language
DSL-POPP	Domain-Specific Language for Pattern-Oriented Parallel Programming
ECF	Environment Complexity Factor

EI	External Input
EIF	External Interface Files
EO	External Output
EQ	External Inquirie
FCIL	Facilities
FLEX	Development Flexibility
FPA	Function Point Analysis
FPGA	Field-Programmable Gate Array
FTR	File Type Referenced
GDDR	Graphics Double Data Rate
GPU	Graphics Processing Unit
GQM	Goal Question Metric
GrPPI	Generic Reusable Parallel Pattern Interface
GSC	General Systems Characteristics
GT	Grounded Theory
HDNN	Heterogeneous Deep Neural Network
Hi-PaL	High-Level Parallelization Language
HPC	High Performance Computing
HPF	High Performance Fortran
ILF	Internal Logical Files
IFC	Information Flow Complexity
KSLOC	Kilo Source Lines of Code
LAN	Local area network
LEO	Language Extensions for Offload
LTEX	Language and Tool Experience
MdMRE	Median magnitude of relative error

MIMD	Multiple instruction multiple data stream
MMRE	Mean Magnitude of Relative Error
MODP	Use of modern programming practices
MPI	Message Passage Interface
MRE	Magnitude of Relative Error
MSLOC	Modified Source Lines of Code
NOC	Number of Characteres
OmpSs	OpenMP Superscalar
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision Library
OpenMP	Open Multi-Processing
OSL	Orléans Skeleton Library
ParSoDA	Parallel Social Data Analytics
PCAP	Programming Capability
PCON	Personnel Continuity
PCRM	Parallel COCOMO II Reuse Model
PERS	Personnel Capacity
PDIF	Platform Difficulty
PI	Productivity Index
PInT	Pattern Instrumentation Tool
PLINQ	Parallel Language Integrated Query
PLEX	Platform Experience
PMAT	Process Maturity
PHalstead	Parallel Halstead
PPI	Parallel Programming Interface
PPGCC	Graduate Program in Computer Science

PPL	Parallel Pattern Language
PREC	Precedentedness
PRED	Percentage Relative Error Deviation
PREX	Personnel Experience
Pthreads	POSIX Threads
PUCRS	Pontifical Catholic University of Rio Grande do Sul
RCPX	Product Reliability and Complexity
PVOL	Platform Volatility
RCPX	Product Reliability
RELY	Required Software Reliability
RESL	Architecture/Risk Resolution
RET	Record Element Type
RMI	Remote Method Invocation
RUSE	Required Reusability
SCED	Required Development Schedule
SE	Software Engineering
SEER-SEM	SEER – Software Estimation Model
SIMD	Single instruction multiple data
SITE	Multisite operation
SLOC	Source Lines of Code
SM	Stream Multiprocessor
SP	Streaming Processor
SPar	Stream Parallelism
STOR	Main Storage Constraint
SVM	Support vector machine algorithm
TBB	Threading Building Blocks

TCF	Technical Complexity Factor
TEAM	Team Cohesion
TIME	Execution Time Constraint
TOC	Tokens of Code
TOOL	Use of Software Tools
TPL	Task Parallel Library
TPU	Tensor Processing Unit
TURN	Computer turnaround time
UCP	Use Case Points
UFP	Unadjusted Function Points
UPC	Unified Parallel C
USC	University of Southern California
UUCP	Unadjusted Use Case Points
VAF	Value Adjustment Factor
VLSI	Very large scale integration
WOC	Words of Code

CONTENTS

1. INTRODUCTION

One of the main reasons for the popularization of parallel architectures in the last decade has been the limitations faced by the silicon industry in the design of the CPU and the requirements to increase performance [185, 163]. As such, the multi-core CPU architectures containing several cores on a single chip, as well as the many-core Graphics Processing Unit (GPU) co-processors, have emerged [119, 163, 185]. To enable the parallelism exploration of such architectures, the programmer must use parallel programming techniques, libraries, structures, mechanisms, and paradigms to develop the applications. In addition, the programmer must know the characteristics of the computer architecture in which they will develop the software, which varies between vendors and platform types [76, 119]. Developing parallel applications is a complex task for application programmers, who usually focus on developing business logic code. It is also a challenging task for system programmers who are experts in parallel programming because they need to be concerned about several parallelism aspects throughout the parallel application development process. For example, they have to think in parallel, implement data synchronization, split the computing problem among threads, exploit concurrency, and provide low-level hardware optimization such as memory [152, 204].

New Parallel Programming Interfaces (PPIs) have been created to facilitate the development of parallel applications. PPIs allow the programmer to deal with low-level implementations and architecture-specific optimizations. In addition, there are PPIs based on structured and non-structured approaches. Unlike the non-structured approaches, structured parallel programming is a high-level approach that uses parallel patterns that can be a receipt/guide for writing efficient parallel software or provided as ready-to-use templates that already implement lower-level parallelisms, such as the communication and synchronization threads, regardless of the target architecture [150, 152]. In addition, these patterns facilitate the exploitation of parallelism in applications such as stream processing [77], which are easily available in our daily lives through their execution on personal computers, cell phones, and servers [43]. Some examples of stream processing applications include video and audio processing, and data compression and analysis.

Most studies in the parallel programming domain focus on evaluating the execution time and performance of an application without considering the human effort involved in the development, making it difficult to determine which PPI offers the best productivity [96]. Coding productivity is an important factor that, together with effectiveness and user satisfaction, are usability indicators [108]. Based on productivity and usability indicators, it is possible to give improvement indicators for designing new PPIs and refining existing ones. As such, it is possible to continue increasing the abstractions of parallelism and create better and simple-to-use PPIs without compromising the performance of the applications.

Productivity is commonly measured by human effort in relation to the results achieved [108, 107], including the time to develop a software [78]. Experiments should be conducted in controlled environments to evaluate productivity, in which human subjects (usually students) resolve small programming tasks using different PPIs [78, 173, 171, 189, 211, 94, 96, 158]. However, experimentation is time-consuming because it must be carefully planned and executed [252], and it is a challenge to find a representative sample of participants in the parallel programming domain. Therefore, many parallel programming researchers instead use established off-line code metrics to estimate the development time and facilitate productivity evaluation.

Different coding metrics, such as those based on code size, complexity evaluation, and development effort, have been used to evaluate productivity in the parallel programming domain. However, these metrics target the assessment of general-purpose software without considering the particular characteristics of parallel applications. Usually, these metrics consider the development of the application from scratch without considering the parallelization of an existing sequential application. Therefore, there is still a need to investigate the effectiveness of these metrics to assess the productivity of parallel applications and opportunities for improvement.

In this work, we aimed to map the parallel programming area to identify the challenges and opportunities for improvement in evaluating coding productivity. Section 1.1 presents the objectives of this study in more detail. Based on the defined goals, scientific contributions are provided in Section 1.2. In addition, the organization of this paper is presented in Section 1.3.

1.1 Research goals

In this doctoral thesis, we address productivity evaluation in developing parallel applications. We aim to provide methodologies and techniques to facilitate and support productivity evaluation in the parallel programming domain. The use of coding metrics for evaluating the productivity of parallel applications is promising since they can provide specific insights into coding productivity. There are opportunities to propose improvements to existing coding metrics to make them more suitable for estimating the effort required to develop parallel applications. Therefore, the research goals can be summarized as follows:

1. Increase knowledge regarding the evaluation of PPIs productivity in the literature;
2. Improve the execution of software experiments in order to evaluate the productivity of PPIs;

3. Identify the factors impacting the coding productivity of parallel application developers;
4. Identify the limitations of existing coding metrics and opportunities for improvement;
5. Find a less costly method or technique for estimating the effort required to develop parallel applications.

1.2 Contributions

Based on the goals previously defined, the scientific contributions provided in this Ph.D. thesis are as follows.

- Categorizations and discussions of metrics and methodologies that have been used to evaluate the productivity and usability of PPIs (goal 1). More details can be seen in Chapter 3, where we present a literature review.
- A parallel programming assessment methodology to guide other researchers based on the limitations found through the literature review (goal 2). In Chapter 4, this methodology is presented in detail.
- Mapping quantitative and qualitative usability indicators related to parallel programming in stream processing applications for multi-core environments (goal 3). More details regarding how this study was conducted can be seen in Chapter 4.
- Mapping quantitative and qualitative usability indicators related to parallel programming for GPU environments (goal 3). Chapter 4 shows more details regarding this study and its results.
- Mapping and qualitatively analyzing developers' perceptions of productivity in parallel programming (goal 3). More details about the survey research results can be seen in Chapter 6.
- Analysis and evaluation of metrics used to evaluate the programming effort of parallel applications (goal 4). In Chapter 5, we describe and discuss this contribution.
- A less costly method for estimating development effort for parallel applications (goal 5). In Chapter 7, we present a methodology for applying the Planning Poker method in the context of parallel programming.

1.3 Document organization

The content of this Ph.D. thesis is organized as follows. Chapter 2 presents a background of this study, where we introduced parallel programming concepts (Sections 2.1, 2.2, 2.3, and 2.4), standard metrics and tools to assess software development (Sections 2.5.1), and common metrics used to assess the accuracy of software development estimation models (Section 2.5.2). In addition, Chapter 3 presents a literature review regarding usability and productivity evaluation in parallel computing.

Next, after presenting the background and state of the art, Chapter 4 presents a proposed methodology to help researchers conduct experiments to evaluate the usability of PPIs. On the other hand, Chapter 5 considers the use of coding metrics for evaluating parallel programming interfaces and presents their advantages and limitations. Next, Chapter 6 presents a survey to identify factors that can affect the programming productivity of parallel applications.

Chapter 7 presents proposed metrics and techniques for evaluating the development productivity of parallel applications. Finally, Chapter 8 concludes this work, which includes concluding observations, limitations, and potential future work.

2. BACKGROUND

In this chapter, we introduce the main concepts related to the objectives of our study. In Section 2.1, we start introducing parallel architectures. Next, Section 2.2 presents the main concepts related to structured parallel programming. Section 2.3 introduces the stream processing applications, and Section 2.4 introduces some structured programming interfaces to explore this applications domain. Next, Section 2.5 shows an overview of the metrics to evaluate software development. Finally, in Section 2.5.2, we present the main metrics to measure the accuracy of the estimation development effort models presented in the previous section.

2.1 Parallel Architectures

Moore's law [163] is a prediction made by American engineer Gordon Moore in 1965 that the number of transistors on a computer chip was doubling yearly. In practice, the number of transistors doubled every 18 – 24 months, where hardware manufacturers significantly increased performance for application programs. However, currently, there is a physical limitation on the size of the processor chip that limits the number of pins that can be used, causing a bandwidth limitation between the CPU and the main memory. In addition, there are other problems that the processor designer will have to face, such as increasing the complexity of the processor architecture [204].

The inability to increase the speed of processors using traditional techniques has driven the emergence of new architectures. Two main approaches in the semiconductor industry have been used to design microprocessors: multi-core CPU and many-core GPU. Figure 2.1 shows the fundamental differences between these architectures in a simplified way [119]. Multi-core CPUs integrate multiple processing cores into a single chip [204], which may be of general purpose or highly specialized in nature [54]. Multi-core CPUs are designed to increase the performance of sequential codes by executing them in parallel at multiple cores, even out of their sequential order, while maintaining the appearance of sequential execution [126, 119]. Multi-core CPU architecture is briefly composed of control units, Arithmetic and logic units (ALUs), and multiple levels of caches (L1, L2, and L3) [119]. Each core can have a private L1 cache and share the L2 cache with other cores. In addition, a shared L3 cache is used frequently for highly used data [204].

In contrast to multi-core CPU architectures, many-core GPUs focus on the throughput of parallel applications [119]. This architecture comes from the video game industry, where it processes video graphics and has been increasingly used for programmable computing [54]. A typical Nvidia GPU is organized in a set of highly threaded Stream Multiprocessors (SMs) [179], an architecture of type Single instruction multiple data (SIMD) or

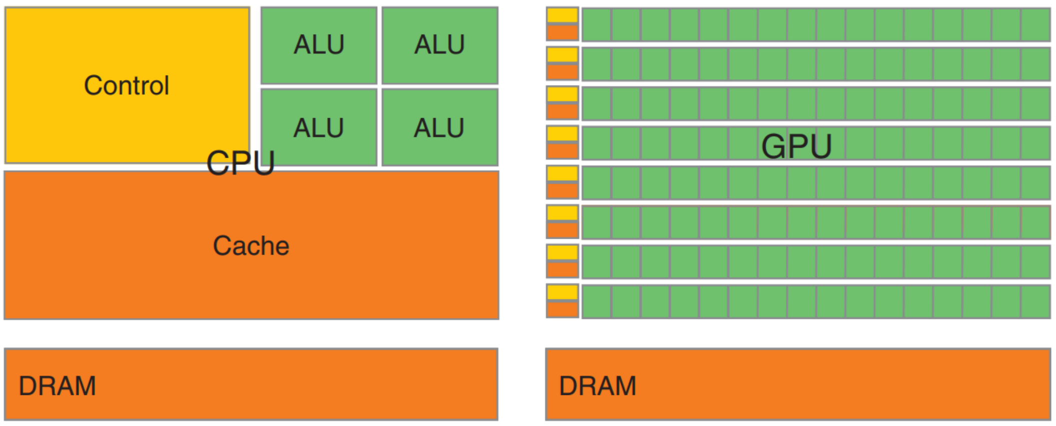
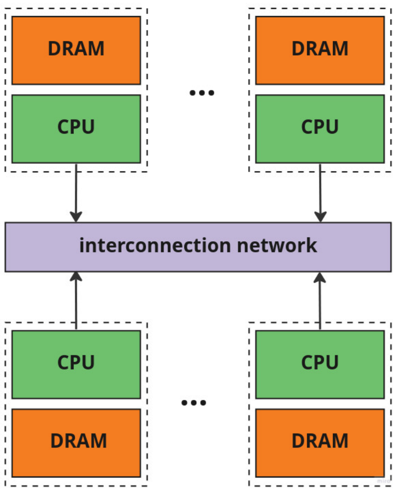


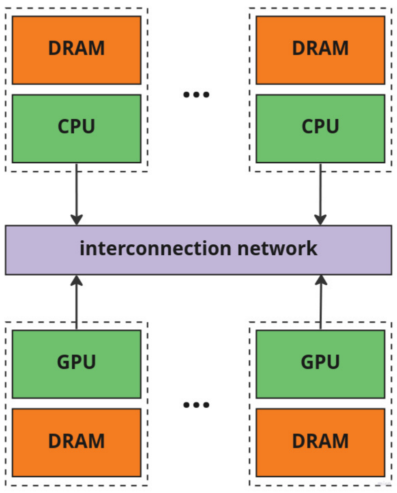
Figure 2.1: Architectural differences between CPU and GPU. Extracted from [119].

Multiple instruction multiple data stream (MIMD) composed of several Streaming Processors (SPs) or thread processors [70]. Each SMs performs the computation independently. However, the SPs within the single SM execute instructions synchronously since they share control logic and an instruction cache [126, 119]. As illustrated in Figure 2.1, the many-core GPU architecture is designed in order to have more transistors dedicated to data processing (SP) instead of cache and control units [179].

There is another difference between the multi-core CPU and the many-core GPU, which is related to global memory. In Figure 2.1, the global memory is represented as Dynamic random-access memory (DRAM). GPUs have multiple gigabytes of Graphics Double Data Rate (GDDR) DRAM, which, unlike system DRAMs on the CPU motherboard, it is optimized for higher bandwidth workloads. For graphics applications, the GPU GDDR DRAM maintains video images and texture information for 3D rendering [119].



(a) Homogeneous architecture.



(b) Heterogeneous architecture. Adapted from [126].

Figure 2.2: Architectural differences between homogeneous and heterogeneous cluster.

In addition to shared memories, there are architectures with distributed memories, such as clusters. A computer cluster comprises a collection of two or more computers (nodes), which are used to execute a particular problem or section. In a computing cluster, the interconnection network that connects the nodes is usually a Local area network (LAN) [70]. In this type of computer cluster, the architectures are homogeneous, that is, all nodes have similar architectures [54]. Figure 2.2a illustrates the architecture of a homogeneous cluster, which is composed by several nodes with multi-core CPUs. Nowadays, many High Performance Computing (HPC) clusters have heterogeneous architectures due to the flexibility of adding new nodes with different and more updated architectures [54]. Therefore, a homogeneous cluster can become heterogeneous. Figure 2.2b shows an example of heterogeneous HPC cluster with both CPU and GPU nodes [126, 119].

2.2 Structured parallel programming

In Section 2.1, the parallel architectures were presented, which are widely available today. To enable the exploration of parallelism, different IPPs have been designed over the years. Using these PPIs efficiently is not a simple task for application programmers, who usually focus on developing the business logic code. This is also a challenging task for system programmers who are experts on parallel programming because they need to be concerned low-level architectural details as well as address parallelism-specific aspects, such as load balancing, synchronization etc.

In order to provide abstractions and release programmers from dealing with lower-level implementations and architecture-specific optimizations, new PPIs have been created based on structured approaches. Structured parallel programming has a higher-level approach for writing efficient, structured, and maintainable programs using a set of parallel patterns, also called algorithmic skeletons [152]. This programming model came from Software Engineering, where design patterns are widely used in object-oriented programming [68]. Such patterns are offered as pre-defined templates that support modeling business logic code. On the other hand, through the algorithmic skeletons, common parallel programming paradigms are captured and made available to the programmer as high-level programming constructions equipped with well-defined functional and extra-functional semantics [5].

Figure 2.3 visually presents an overview of the main parallel programming patterns [77], which can be classified mainly into data (e.g., map, reduce, stencil, scan, and others) and stream (e.g., pipeline, farm, and others) parallel patterns [152]. A pipeline pattern is used to exploit in the form of a traditional manufacturing assembly line. It has well-defined tasks to be performed on data to produce transformed data, which are sent to the next stage/workstation [207, 152, 237, 5]. The pipeline pattern applies a sequence of operations simultaneously to each data element, so it is possible to compute each operation in a different data element at each point and in a given time. The parallel activity graph in Figure 2.3 is a pipeline with three independent stages (kernels or filters) that communicate explicitly through data channels, where the output of one stage feeds the input of the next stage, finite or infinitely.

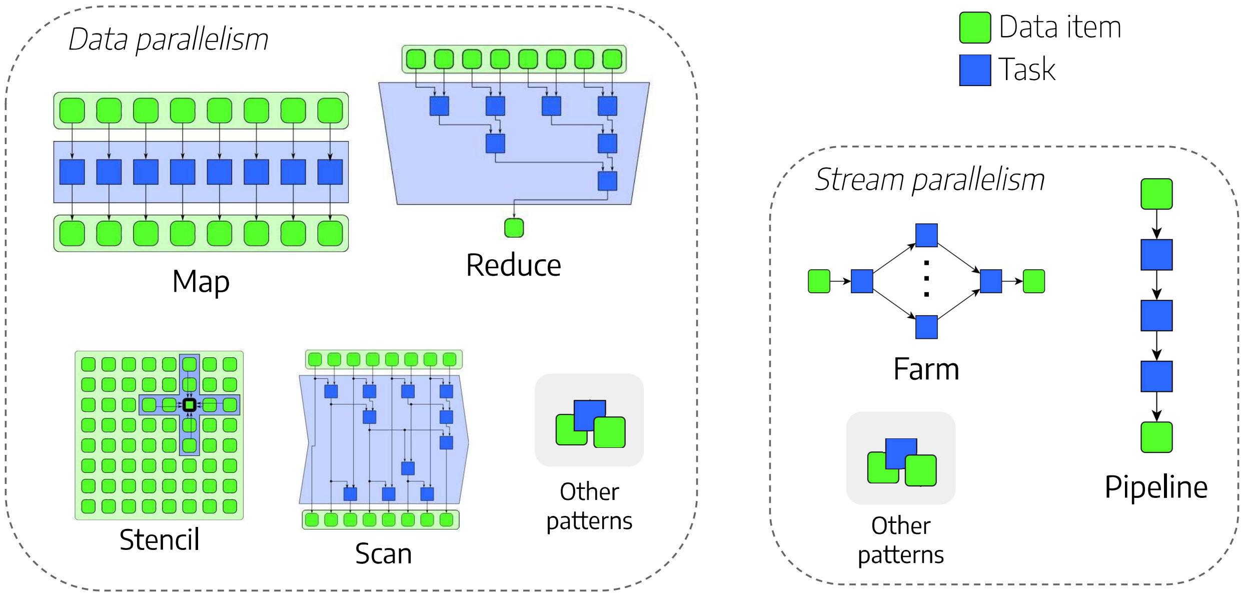


Figure 2.3: Overview of parallel patterns. Adapted from [152].

The farm pattern is also called split-join [237]. This pattern is similar to the pipeline pattern and can be implemented with two or three stages. The first stage performs as the data item emitter or scheduler. The second stage performs as stage replicas called workers. Optionally, the last stage acts as a data item collector [5, 77]. The parallel activity graph in Figure 2.3 is a farm pattern with its three components (emitter, worker, and collector). In addition, both pipeline and farm patterns are widely used to implement parallelism exploitation in stream processing applications [77]. These applications comprise of collecting, processing, and analyzing high volume, heterogeneous, and continuous data streams in real-time [13].

The map pattern, also called loop parallelism [150], replicates a function on each element of a set that can be abstract or associated with the elements of a collection. Since the replicated function applies to the elements of a real input data collection, it is called elementary function [152]. The map pattern can be used to replace an independent loop in a program whose number of iterations is known previously, and the computation only depends on the iteration count as an index in a collection. The elementary function must not modify any global data on which other instances of this function depend. In addition, the map pattern can be used, for example, for correction and thresholding in images, color space conversions, Monte Carlo sampling, and radius tracing [152].

There is a generalization of the map pattern called the stencil. In this pattern, the elementary function can access a set of neighbors and not just a single element in a collection of inputs, as seen in Figure 2.3. In the stencil pattern, the neighborhoods are given by a set of relative offsets. The threshold conditions for access to the matrix need to be considered for this pattern. Moreover, this pattern is frequently used for image filtering, for example, convolution, median filtering, motion estimation in video encoding, and isotropic diffusion noise reduction [152].

The reduce pattern uses an associative combinator function to combine each collection element into a single output element. Many different ordinances are possible. In addition, the reduction pattern is usually used with a map pattern. Reduction applications include averaging of Monte Carlo samples for integration, convergence testing in iterative solution of systems of linear equations (e.g., conjugate gradient), image comparison metrics (e.g., video encoding), and dot products and row–column products in matrix multiplication [152].

The scan pattern, also called prefix sum [119], is considered a special case of the series pattern called bending. The scan pattern calculates all partial reductions of a collection. This means that, for every output position, a reduction of the input to that point is computed. Describing a computation as a mathematical recursion is necessary to parallelize it using a scan pattern [119]. The scan is often used to convert seemingly sequential operations, such as resource allocation, work assignment, and polynomial evaluation [152].

2.3 Parallel stream processing

Streams are data generated continuously by network services, cameras, sensors, and other data sources. Stream processing is a computing paradigm for collecting, processing, and analyzing high-volume, heterogeneous, and continuous data streams in real-time [13, 239]. The data items are processed or consumed by the so-called stream processing applications. Usually, they are continuously collected and processed as a sequence of stages (also called operators) that apply computations (filter and analyze) over the items, which can later be stored in a permanent file system [237, 13]. In addition, the stages tend to be organized as a Directed Acyclic Graph (DAG) where the stream item flows through the graph [118].

Stream processing applications can be found in several domains, including the stock market, natural systems, transportation, manufacturing, health and life sciences, law enforcement, defense and cybersecurity, fraud prevention, e-Science, telephony, and others. Figure 2.4 shows examples of these applications in different domains. Stream processing applications can process structured and unstructured data. Most stream processing applications process structured data (e.g., relational database style records), which share a common schema or structure with data items organized into names/types/value triples. On the other hand, commercial stream processing applications usually process unstructured data like image, audio, and video, mainly executing tasks, such as compression, filtering, reproduction, etc. [13, 237].

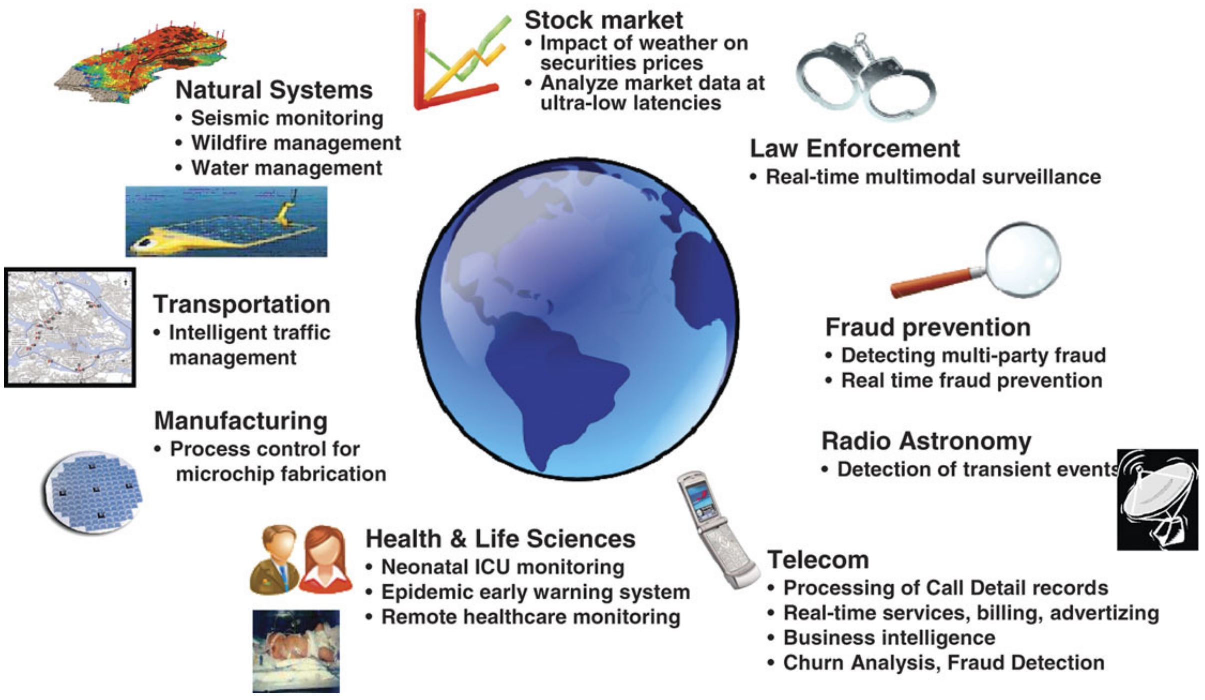
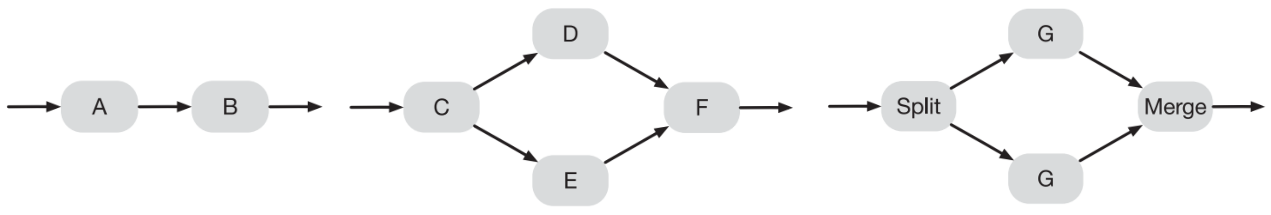

Stock market
• Impact of weather on securities prices
• Analyze market data at ultra-low latencies
Natural Systems
• Seismic monitoring
• Wildfire management
• Water management
Law Enforcement
• Real-time multimodal surveillance
Transportation
• Intelligent traffic management
Fraud prevention
• Detecting multi-party fraud
• Real time fraud prevention
Manufacturing
• Process control for microchip fabrication
Radio Astronomy
• Detection of transient events
Health & Life Sciences
• Neonatal ICU monitoring
• Epidemic early warning system
• Remote healthcare monitoring
Telecom
• Processing of Call Detail records
• Real-time services, billing, advertizing
• Business intelligence
• Churn Analysis, Fraud Detection

Figure 2.4: Stream applications. Extracted from [239].

Compared to traditional applications, the volume of data to be processed in stream processing applications is unknown and cannot be calculated. Storing the stream items in a database and processing it using traditional approaches is not feasible because, in many cases, this data comes from sensor measurements. Furthermore, there are stringent requirements on latency and throughput of processing these data [33, 106]. Therefore, parallel computing can be necessary to reach the quality of service requirements such as real-time response or high throughput [83].

Parallel stream processing applications execute as stream graphs composed of operators or stages and FIFO (First In, First Out) communication queues [5, 93]. The stream input is an infinite sequence of items (or data) stream, and the queues contain a finite number of items waiting to be consumed by each stage [222]. The computation waits until the batched operator has received enough items. Therefore, when the batch of items is complete, the operator will perform the entire batch computation at once [93]. In addition, each operator can process a different item than the previous operator. Figure 2.5a illustrates this type of parallelism, where the sequence of producer A and consumer B operators can process different items concurrently.



(a) Pipeline parallel. (b) Task parallel. (c) Data parallel.

Figure 2.5: Types of parallelism in the context of data stream processing. Extracted from [93].

The parallelism can be increased by replicating one stateless operator to process multiple items simultaneously (data parallelism) or by performing different computations simultaneously (task parallel) [93]. Figure 2.5b illustrates task parallelism, where the different operators D and E that do not constitute a pipeline can be processed concurrently. Figure 2.5c shows the data parallelism, where the execution of the stateless operator G is replicated to process different portions of the same data [93].

As can be noticed, developing parallel stream processing applications is a challenging activity. Therefore, the structured parallel programming approach presented in Section 2.2 alleviates these complexities for different application domains. It supports developers with parallel patterns, which are parallelism strategies to write efficient, structured, and maintainable programs [152]. The two parallel patterns commonly used when implementing parallel stream processing are pipeline and farm. The parallelism type shown in Figure 2.5a is implemented using the Pipeline pattern. However, the number of operators usually limits the parallelism of stream processing. Therefore, the data parallelism could be applied or combined to increase the degree of parallelism [77]. In the farm pattern, the parallelism can be increased through a combination of pipeline and data parallelisms. Therefore, the stateless operator can be replicated in order to perform the same computation on different data items.

2.4 Structured parallel programming interfaces

Different PPIs have been proposed to explore parallel architectures. For multicore architectures the well-established models are Open Multi-Processing (OpenMP) and POSIX Threads (Pthreads) [113], although they are only suitable for data parallelism exploitation and requires the programmer to implement extra synchronization mechanisms in the stream parallelism exploitation. For stream parallelism exploitation based on the structured programming approach, one remarkable PPI is StreamIt [237]. It is an external Domain-Specific Language (DSL) (new language and compiler), and its research activities were discontinued in 2013. Maintained by Intel, Threading Building Blocks (TBB) [207, 244] is an open-source and general-purpose C++ template-based PPI from the industry. TBB offers a Pipeline pattern constructor that can also perform as the Farm pattern.

FastFlow [5] is a representative PPI from the scientific community. It has a similar C++ template-based interface to TBB. However, their runtime parallelism systems are implemented differently. While the TBB interface works on top of an unchangeable work-stealing task scheduler and building blocks, the FastFlow interface works on top of customizable lock-free FIFO queues, building blocks, parallel patterns, and task scheduler. A research initiative that promises to leverage higher-level and productive stream parallelism on multi-core systems is Stream Parallelism (SPar) [80, 82]. It is an internal DSL (embedded in the C++ language) in the form of C++ annotation to avoid sequential code rewriting. SPar explore stream parallelism in multi-core systems through parallel code generation with FastFlow [79], TBB [98], and more recently with OpenMP [99] runtimes.

For heterogeneous programming, the most popular PPIs are Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) [22]. As low-level models, they require specific control of coordination and computation, such as controlling access to different levels of the memory hierarchy [22]. OpenACC is another low-level model for heterogeneous programming [155]. SkePU is a C++ template library that provides a higher abstraction level through skeletons building on top of OpenCL and CUDA for heterogeneous architectures [60]. It provides a task-parallel skeleton (Farm) and a set of data-parallel skeletons (Map, Reduce, MapReduce, MapOverlap, and Scan). SkePU uses smart data containers and provides custom data structures resident on the host memory. However, providing a non-native C++ data structure is one of the SkePU drawbacks. In addition, when using GPUs, smart data containers automatically manage the memory device, which may not perform the best choice to optimize memory transfers [48]. A newer research approach that does not have these drawbacks is GSParLib [208], a structured PPI for exploiting data parallelism in C++ applications executing in GPU systems. GSParLib also provides abstraction through CUDA and OpenCL code generation.

In the next sections we will present the workings of TBB, FastFlow, and SPar for exploiting stream processing parallelism on multi-core architectures. In addition, we will present the workings of GSParLib for exploiting GPU data parallelism.

2.4.1 Intel TBB

This section describes how to use TBB to expressing stream parallelism in C++ stream processing applications in multi-core CPU systems. In TBB, some stages can operate in `parallel`, and others can not (`serial`). The concept of the stage in TBB is known by the name `filter`. Listing 2.1 shows how to create the first stage using TBB. Although TBB supports modeling stages using the lambda function interface, we concentrate on the default interface, in which stages are modeled as classes extending the `tbb::filter` class, where one of the following filter types should be specified as an argument to indicate the stage behavior (line 3) [244]:

- **`tbb::filter::serial_out_of_order:`** It is used to process the items one at a time without preserving the processing order;
- **`tbb::filter::serial_in_order:`** It is used to process the items one at a time in the same order. The processing order is implicitly defined by the first filter and respected by the other ones;
- **`tbb::filter::parallel:`** It is used to process multiple items in parallel and in no particular order.

Moreover, each stage class needs to implement the virtual `operator` method (lines 4-11) in which a task or stream item is processed. Every time this method returns a `void` pointer (line 8), it is implicitly sends the stream item to the next stage. When `NULL` is returned (line 10), it indicates the end of the stream to stop the stream processing [244].

```
 1 class first() : public tbb::filter{
 2 public:
 3   first():tbb::filter(tbb::filter::serial_in_order) {}
 4   void* operator() (void*){
 5     while(1){
 6       // computation
 7       if(stop) break;
 8       return item;
 9     }
10     return NULL;
11   }
12 };
13 class middle() : public tbb::filter{
14 public:
15   middle(): tbb::filter(tbb::filter::parallel) {}
16   void* operator() (void *item) {
17     // computation
18     return item;
19   }
20 };
21 class last() : public tbb::filter{
22 public:
23   last() : tbb::filter(tbb::filter::serial_in_order) {}
24   void* operator() (void *item) {
25     // computation
26     delete item;
27     return NULL;
28   }
29 };
```

Listing 2.1: First, middle and last stages using TBB. Extracted from [11].

Listing 2.1 also shows how to create the middle and last stage using TBB. The middle stage only computes the stream items and sends them to the next stage using the `return` operation (line 19). The programmer can create as many as necessary middle stages. For the last stage, the programmer has to manage the stream item data, deallocate the memory for the input item (line 28), and return a specific value to skip sending items to subsequent stages (line 29).

Therefore, to build the parallel activity graph as a Farm pattern in TBB, the programmer specifies the `parallel` filter in the middle stage and uses the same Pipeline object to instantiate a traditional Pipeline pattern, as shown in Listing 2.2. In TBB, the programmer can also specify how many concurrent threads are created in line 1. The `run` method is used to indicate the beginning of the Pipeline execution, receiving as an argument the number of concurrent tokens (it can also be understood as the number of items in the shared queue). Moreover, the class object `tbb::pipeline` is first declared (line 2) to build a parallel activity graph. Next, the objects of the three stages are declared and added to the Pipeline object using `add_filter` (lines 3-6) in the correct sequence. Lastly, by calling the `run` method in the `pipeline` object, the parallel computing will start and keep executing until a stop condition, which is a `NULL` pointer [244].

```
1 tbb::task_scheduler_init init(3);
2 tbb::pipeline pipeline;
3 first S1;
4 middle S2;
5 last S3;
6 pipeline.add_filter(S1);
7 pipeline.add_filter(S2);
8 pipeline.add_filter(S3);
9 pipeline.run(3);
```

Listing 2.2: Parallel activity graph modeled according to the Farm pattern while instantiating stages using TBB. Extracted from [11].

2.4.2 FastFlow

This section describes how to use FastFlow for exploiting stream processing on C++ applications in multi-core CPU environments. Listing 2.3 show how to create the first stage using FastFlow. In FastFlow, although a stage can be modeled using the lambda function interface, we focus on the default interface where a stage is modeled as a class or struct, extending the `ff_node` class (line 1). Inside the stage, the virtual `svc` method has to be implemented (lines 3-10). The first stage may produce tasks (stream items) inside the `svc` method and send the produced stream items to the next stage using the `ff_send_out` method (line 7). If there are no more stream items, it is possible to return `EOS` (line 9) to propagate the end of the stream processing to the subsequent stages [5].

Listing 2.3 also shows how to create the middle and last stage using FastFlow. Disassociating the specific syntax and semantics, the principle for modeling the middle and last stage is similar to the TBB presented previously. The middle stage also computes the stream items and sends them to the next stage using the `return` operation (line 16). For the last stage, the programmer has to manage the stream item data, deallocate the memory for the input item (line 23). However, unlike TBB, the last FastFlow stage must return the tag **GO_ON** (line 24), which is used to inform the runtime support that other tasks should be expected from the input channel and that the computation is not finished [5]. Another difference is that in TBB the programmer defines the stage behavior when writing the class and passing as an argument the filter, while in FastFlow, this is done when instantiating the parallel pattern.

```
 1 class first : public ff::ff_node_t<int>{
 2 public:
 3   int *svc (int *){
 4     while (1){
 5       // computation
 6       if(stop) break;
 7       ff_send_out(item);
 8     }
 9     return EOS;
10   }
11 };
12 class middle: public ff::ff_node_t<int>{
13 public:
14   int *svc (int *item){
15   // computation
16     return item;
17   }
18 };
19 class last : public ff::ff_node_t<int>{
20 public:
21 int *svc (int *item){
22   // computation
23   delete item;
24   return GO_ON;
25   }
26 };
```

Listing 2.3: First, middle, and last stages using FastFlow. Extracted from [11].

Listing 2.4 shows how to build a Pipeline pattern using the `ff::ff_Pipe`, where the arguments are stage object classes (line 4). Differently from TBB, in FastFlow there is a specific object class to build the parallel activity graph for the Farm pattern. Listing 2.5 shows that the three entities of the Farm pattern (Emitter, Worker, and Collector) receive the class objects declared as an argument. For the Worker entity, we must create a vector that has many replicas as parallel workers are intended (`nthreads`) (lines 1-3). After, the stage object classes are declared (line 4). Next, the `ff::ff_Farm` template class is used to build the parallel activity graph, where the Worker entity is passed as an argument (a vector of workers) (line 5). The Emitter and Collector entities are added using the respective routine (lines 6-7). Then, the parallel computing will start and wait until finished at the call of the `run_wait_end` routine.

```
1 first S1;
2 middle S2;
3 last S3;
4 ff:ff_Pipe<int> pipeline(S1, S2, S3);
5 pipeline.run_wait_end();
```

Listing 2.4: Parallel activity graph modeled according to the Pipeline pattern while instantiating stages using FastFlow. Extracted from [11].

```
1 std::vector<std::unique_ptr<ff_node>> S2;
2 for(int i=0; i<nthreads; i++)
3 S2.push_back(std::make_unique<middle>());
4 first S1;
5 last S3;
6 ff::ff_OFarm<int> farm(std::move(w));
7 farm.add_emitter(S1);
8 farm.add_collector(S3);
9 farm.run_wait_end();
```

Listing 2.5: Parallel activity graph modeled according to the Farm pattern while instantiating stages using FastFlow. Extracted from [11].

FastFlow allows us to create other parallel patterns, combining Farm and Pipeline object classes, and reusing the same stage classes. A new parallel pattern can be created by transforming one or more stages of a simple Pipeline pattern (Listing 2.4) into a Farm. The example on the Listing 2.6 shows how this can be done. The middle stage becomes the Worker entity, and the last stage becomes the Collector entity of a Farm. Then, we build the Pipeline with two stages, where the first is the serial while the second is a Farm. Other patterns and activity graphs could be created using FastFlow, which does not necessarily optimize the parallelism exploitation. It will depend on several aspects regarding the environment and application.

```
1 std::vector<std::unique_ptr<ff_node>> S2;
2 for(int i=0; i <nthreads; i++)
3 S2.push_back(std::make_unique<middle>());
4 first S1;
5 last S3;
6 ff::ff_OFarm<int> farm(std::move(w));
7 farm.add_collector(S3);
8 ff:ff_Pipe<int> pipeline(S1, farm);
9 pipeline.run_wait_end();
```

Listing 2.6: Parallel activity graph modeled according to the Pipeline pattern and combined with Farm while instantiating stages using FastFlow. Extracted from [11].

2.4.3 SPar

This section describes how to use SPar for exploiting stream processing on C++ applications in multi-core CPU environments, which is a scientific DSL proposed by Griebler [77]. SPar is C++ embedded DSL using standard C++11 attributes annotations to express stream parallelism, which is parsed by its own compiler to generate parallel code automatically [80]. A C++11 annotation is declared using double brackets (`[[attr-list]]`) where there are one or more attributes. In SPar, the first attribute in the list is called an identifier (`ID`), and the rest are auxiliary (`AUX`). All attributes are part of the stream parallelism namespace (named as `spar`). The SPar `ID` attributes are `ToStream` and `Stage`, while `AUX` attributes are `Input`, `Output`, and `Replicate`.

`ToStream` indicates that a given C++ program region is going to be stream parallelism. The annotated region can be a loop or a code block. `Stage` denotes a phase within `ToStream`, where operations are computed over the stream items. At least one stage must be within a `ToStream` region. In addition, SPar supports any number of stages inside a `ToStream` region. `Input` is used to indicate the variables that will be consumed by `ID` attributes, and `Output` is used to indicate the variables that will be produced by `ID` attributes. When using these attributes, at least one argument must be present. `Replicate` is used to replicate a `Stage`. This attribute allows programmers to scale the performance on stateless stages. This attribute receives a constant value delimiting the number of workers for the stage as an argument. Moreover, this attribute can also be left empty to use the environment variable `SPAR_NUM_WORKERS`.

```
 1 [[spar::ToStream]]
 2 while(1){
 3     // computation
 4     if (stop) break;
 5     [[spar::Stage, spar::Input(item), spar::Output(item), spar::Replicate(4)]]{
 6         // computation
 7     }
 8     [[spar::Stage, spar::Input(item)]]{
 9         // computation
10     }
11 }
```

Listing 2.7: Example of Pipeline using SPar. Extracted from [11].

Listing 2.7 presents a pseudocode example of the SPar use. In line 1, note that `ToStream` annotation was inserted in front of a loop, indicating the beginning of the stream parallelism region. Since stream items are not consumed from and produced outside the region, auxiliary attributes are unnecessary. The codes left between `ToStream` and the first `Stage` annotation (line 5) are always an implicit serial stage that produces stream items to the following stages. The first `Stage` annotation will actually be the second `Stage` of the parallel activity graph, where if there is another `Stage` in sequence, `Input` and `Output` are required. Since this `Stage` is a stateless computation, we can add the `Replicate` attribute to increase the degree of parallelism. By the way, the last stage is stateful and does not produce anything outside. Therefore, `Output` is not needed, and `Replicate` does not apply.

Additionally, SPar provides compiler flags if programmers want to change the runtime system behavior, which may improve the performance of the application:

- **`-spar_blocking`:** It is used to activate the blocking mode on the scheduler when the communication queues are full (by default, it is not locked);
- **`-spar_ondemand`:** It is used to activate the on-demand scheduler by setting the queue size to one (by default, it is round-robin);
- **`-spar_ordered`:** It is used to allow stream elements to be processed in order (by default, they are processed without preserving the order).

These flags can be used individually or combined to optimize the code during compilation and provide speed.

2.4.4 GSParLib

This section describes how to use GSParLib for exploiting data parallelism in GPU environments, a scientific library proposed by Rockenbach [208]. GSParLib is a C++ library divided into two layers, a lower-level and a higher-level. The low-level layer of GSParLib is called Driver Application Programming Interface (API), which acts as a wrapper over CUDA [179] and OpenCL. The high-level layer, on the other hand, focuses on structured parallel programming. This layer is called the Pattern API [208].

The goal of the Driver API is to act as a unified interface to the different GPU PPIs, currently CUDA and OpenCL. In addition, it abstracts from the programmer the complexities of lower-level language constructs, such as C-like error handling. Programming an application with the Driver API follows the standard GPU programming flow defined by PPIs such as CUDA and OpenCL. Programming with Driver API can be summarized in four steps [208]:

- **Step 1:** initialize the interface and identify the devices;
- **Step 2:** allocate the device memory and copy the necessary data;
- **Step 3:** prepare the Kernel object and invoke the GPU kernel;
- **Step 4:** copy results back to main memory (host) and release resources used.

```
 1 #if defined(GSPARDRIVER_CUDA) //CUDA gpu kernel
 2 #include "GSPar_CUDA.hpp"
 3 using namespace GSPar::Driver::CUDA;
 4 const char* gpu_kernel_source = GSPAR_STRINGIZE_SOURCE(
 5 extern"C"
 6 __global__ void matrix_multiplication(int* matix1, int* matrix2, int* matrix3, intN){
 7     int i = blockIdx.x * blockDim.x + threadIdx.x;
 8     int j = blockIdx.y * blockDim.y + threadIdx.y;
 9     for(intk=0; k<N; k++){
10         matrix3[i*N+j] += (matrix1[i*N+k]* matrix2[k*N+j]);
11     }
12 });
13 #elifdefined(GSPARDRIVER_OPENCL) // OpenCL gpu kernel
14 #include "GSPar_OpenCL.hpp"
15 using namespace GSPar::Driver::OpenCL;
16 const char* gpu_kernel_source = GSPAR_STRINGIZE_SOURCE(
17 __kernel void matrix_multiplication(__global int* matrix1,__global int* matrix2,__global int* matrix3,intN){
18     int i = global_id(0);
19     int j = global_id(1);
20     for(int k=0; k<N; k++){
21         matrix3[i*N+j] += (matrix1[i*N+k] * matrix2[k*N+j]);
22     }
23 });
24 #endif
25 int main(){
26     // initialization of the steps
27     Instance* driver=Instance::getInstance();
28     driver->init();
29     auto gpus = driver->getGpuList();
30     auto gpu = driver->getGpu(0);
31     matrix1_device = gpu->malloc(N*N*sizeof(double), matrix1_host); // gpu memory allocation
32     matrix2_device = gpu->malloc(N*N*sizeof(double), matrix2_host);
33     matrix3_device=gpu->malloc(N*N*sizeof(double), matrix3_host);
34     matrix1_device->copyIn(); // memory transfer, copy memory to gpu
35     matrix2_device->copyIn();
36     matrix3_device->copyIn();
37     gpu_kernel = newKernel(gpu, gpu_kernel_source, "matrix_multiplication"); // gpu kernel, compiling
38     gpu_kernel->setNumThreadsPerBlockForX(1024);  // gpukernel, setting parameters
39     gpu_kernel->setParameter(matrix1_device);
40     gpu_kernel->setParameter(matrix2_device);
41     gpu_kernel->setParameter(matrix3_device);
42     gpu_kernel->setParameter(sizeof(int),&N);
43     // gpukernel, setting total amount of threads
44     unsigned long dimensions[3] = {N,N,0};  // N*N threads
45     gpu_kernel->runAsync(dimensions); // gpu kernel, running
46     gpu_kernel->waitAsync();  // gpu kernel, wait the finish of computations
47     matrix1_device->copyOut(); // memory transfer, copy memory to cpu (host)
48     matrix2_device->copyOut();
49     matrix3_device->copyOut();
50 }
```

Listing 2.8: Matrix multiplication with the Driver API of GSParLib. Extracted from [48].

Listing 2.8 presents the matrix multiplication applications using GSParLib Driver API [48], in which these steps can be seen. First, the GPU kernels with CUDA (lines 1-12) and OpenCL (lines 13-24) are defined. These definitions allows the programmer to switch between CUDA and OpenCL only to change the compilation flags. From the lines 27 to 30, the Driver API is initialized. Next, the GPU memory is allocated (lines 31-33), and the data are copied from the host memory to the GPU memory (lines 34-36). The GPU kernel is compiled in line 37, and the its parameters are set from lines 38 to 42. The total number of threads used to execute the GPU kernel also must be defined (line 44). Then, the GPU kernel will start and wait until finished at call of the `runAsync` and `waitAsync` routines (lines 45-46). Finally, the `copyOut` routine is used to copy the memory from the GPU to the CPU (lines 47-49).

The Pattern API of the GSParLib provides a higher-level structured programming interface using the capabilities of the Driver API to support both CUDA and OpenCL. The Pattern API provides the Map and Reduce parallel patterns, which can also be grouped into a pattern composition [208].

Listing 2.9 presents the matrix multiplication applications using the Pattern API [48]. From lines 2 to 7, the GPU kernel is defined and the Map pattern is instanced. To do so, first, the functions provided by the Pattern API abstractions are called (lines 3-4), which are replaced by CUDA or OpenCL syntax and collect the id of the threads. From lines 8 to 11, the patterns of the GPU kernel are defined, for which are defined the amount of GPU memory that must be allocated. In addition, it is necessary to define whether the matrices should be copied to the GPU before running the GPU kernel (`GSPAR_PARAM_IN`), or whether they should be copied from the GPU to the CPU after finishing computations (`GSPAR_PARAM_OUT`). In line 12, the GPU kernel is compiled and executed. Finally, the Map pattern is deleted, releasing the related resources.

```
1  void matrix_multiplication(double* matrix1, double* matrix2, double* matrix3, int N){
2    Map* map = new Map(GSPAR_STRINGIZE_SOURCE(
3      int i = gspar_get_global_id(0);
4      int j = gspar_get_global_id(1);
5      for(intk=0;k<N;k++){
6        matrix3[i*N+j]+=(matrix1[i*N+k] * matrix2[k*N+j]);
7      }));
8    map->setParameter("matrix1", sizeof(double)*N*N, matrix1, GSPAR_PARAM_IN)
9    .setParameter("matrix2", sizeof(double)*N*N, matrix2, GSPAR_PARAM_IN)
10   .setParameter("matrix3", sizeof(double)*N*N, matrix3, GSPAR_PARAM_OUT)
11   .setParameter("N", sizeof(int), N)
12   .run<Instance>({N,N});  //N*Nthreads
13   delete map;
14 }
15 int main(){
16   // initialization of the steps
17   matrix_multiplication(matrix1, matrix2, matrix3, N);
18 }
```

Listing 2.9: Matrix multiplication with the Pattern interface of GSParLib. Extracted from [48].

2.5 Coding productivity assessment

Before actually implementing a given software, one of the main concerns related to the software design is how much implementation effort (e.g., working hours) will be required [245]. In order to solve this problem, Software Engineering (SE) researchers have proposed several quantitative metrics to measure specific software characteristics and estimate factors such as cost, effort, quality, and reliability [248]. Most coding metrics use at least one parameter as a basis, such as the number of lines in the source code and system functional requirements [245]. In addition to these techniques, there are metrics of complexity based on the information flow of code, which can also be used as indicators of the development effort. Over the years, several metrics have been proposed. In this context, this section aims to present some of the classical metrics already well established in the SE field.

Several coding metrics are available for estimating software development efforts. Therefore, the use of evaluation metrics is essential to measure the performance and accuracy of the proposed estimation methods [242]. ISO/TS 24541 [110] defined accuracy as the "*closeness of agreement between a measured quantity value and a true quantity value of a measurand*". Therefore, from accuracy assessment, we can verify the proximity between the estimated and actual values, which should be as small as possible [242]. In this context, this section also presents some classical metrics for accuracy evaluation.

2.5.1 Metrics for estimating development effort

This section presents some coding metrics used to estimate the effort and time required to develop software applications.

Source lines of code

Source Lines of Code (SLOC) is perhaps the oldest of the software evaluation metrics and is the easiest to measure [121, 210]. This technique emerged when programming languages like FORTRAN were strongly oriented to SLOC. At that time, the programs were registered on perforated cards, with one line per card. Therefore, the height of the stack of cards was a very natural metric to evaluate the complexity of a program. This metric evolved to Kilo Source Lines of Code (KSLOC), as the size of most programs is now measured in thousands of lines of code.

According to Laplante [121], one of the main disadvantages of using SLOC as a metric is that it can only be measured after the code has been written. However, there is a technique for estimating KSLOC. In this technique there is a meeting with the development team to discuss the system to be developed, in which the participants will give their opinions about the number of KSLOC needed to develop the system considering three values [245]:

- **Optimist:** the minimum number of lines expected to develop if all conditions are favorable;
- **Pessimistic:** the maximum number of lines expected to develop under unfavorable conditions;
- **Expected:** the number of lines expected to develop under normal conditions.

The KSLOC can be calculated from from Equation (2.1):

$$KSLOC = \frac{(4 \times KSLOC_{expected} + KSLOC_{optimist} + KSLOC_{pessimist})}{6} \tag{2.1}$$

McCabe's Cyclomatic Complexity Number

McCabe's Cyclomatic Complexity Number (CCN) [151] is a widely used metric to measure program complexity. This complexity metric is based on the graph theory to measure the number of paths through a program. Measuring the total number of paths may be impractical since any program with a backward branch potentially has an infinite number of paths. Therefore, this measure is defined in terms of basic paths, that when combined, will generate all possible paths.

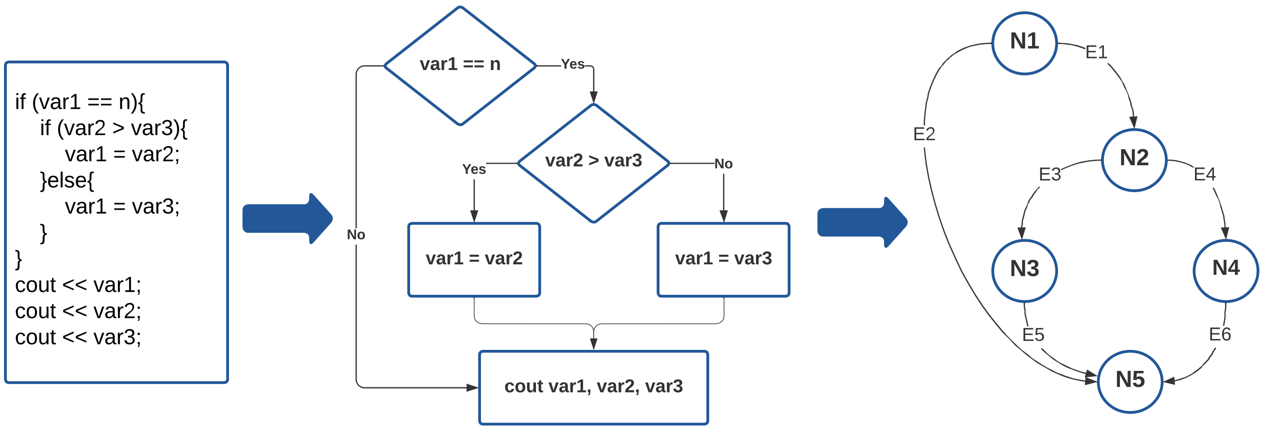
In the CCN, a program must be represented by a flow graph, in which the nodes represent all commands and the edges represent the control flows. In addition, conditional (`if`, `else`) and repetition (`while`, `do`, `for`) structures should be represented through distinct nodes with edges indicating decision and repetition.

CCN can be calculated from the Equation (2.2):

$$CCN = E - N + 2 \tag{2.2}$$

where E is the number of edges, and N is the number of nodes. Fig. 2.6 illustrates the flow-graph of a small piece of a program with five nodes (N), and six edges (E). From Equation 2.2, we achieve CCN equal to 3.

The way a flow graph is defined with a single input and output node results in all flow graphs having only one connected component. However, for example, if there is a main program (M) calling a function (F), there will be two graphs resulting in two connected components (P). In this case, the complexity will be equal to the sum of the individual complexities of M and F, calculated from Equation 2.2 [151]. In addition, the



```
if (var1 == n){
    if (var2 > var3){
        var1 = var2;
    }else{
        var1 = var3;
    }
}
cout << var1;
cout << var2;
cout << var3;
```

Figure 2.6: Flow-graph of a piece of code.

CCN of a collection of flow graphs can also be calculated from Equation 2.3:

$$CCN = E - N + 2 \times P \tag{2.3}$$

where E is the number of edges, N is the number of nodes, and P is the number of connected components [151].

Information Flow Complexity

In 1981, Henry and Kafura proposed a metric to measure the complexity of a code based on the information flow concepts [91]. For this measure, a program must be represented by a flow graph, in which the nodes represent all procedures (or functions), and the edges represent the connections between these procedures. The fan-in and fan-out terms are used to determine the connections between the procedures of a program, which are defined as follows:

- **Fan-in:** The number of local flows to the procedure plus the number of data structures from which the procedure retrieves information;
- **Fan-out:** The number of local flows of the procedure plus the number of data structures the procedure updates.

The Equation 2.4 defines the complexity value of a procedure:

$$HK = length \times (fan_{in} \times fan_{out})^2 \tag{2.4}$$

where the length of a procedure is defined as the SLOC for the selected procedure [91].

Figure 2.7 shows an example of information flow with six procedures A, B, C, D, E, and F. This figure also had a data structure DS and the connections among DS and the procedures. In Figure 2.7, procedure A retrieves information from the DS data structure and then calls procedure B, which updates the DS. Procedure F also updates DS. Therefore, the fan-in of A is equal to 2 because A receives information from B and F. In addition, the fan-out of A is equal to 1 because A sends information only to B [91].

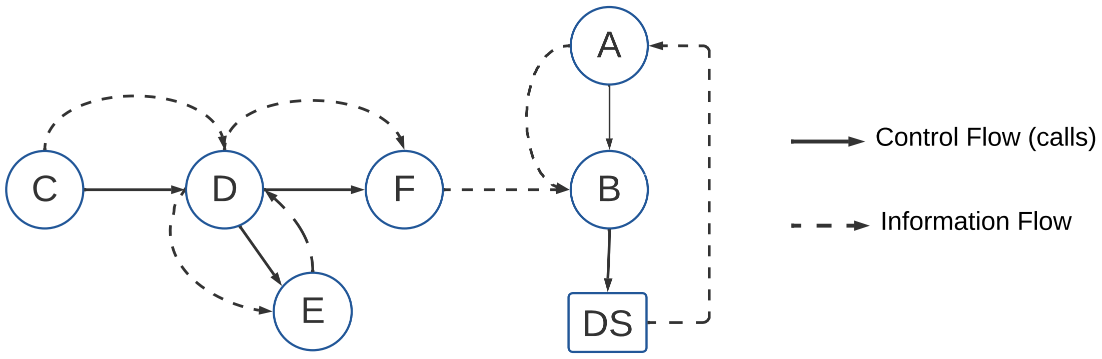


Figure 2.7: Example of information flow graph. Adapted from [91].

Putnam’s model

Putnam’s model is one of the first to estimate the development effort [201]. This model is simple and easily calibrated because it estimates the development time in years based on a Productivity Parameter (PP) and the SLOC. PP can be derived from SLOC, development effort, and development time (in years) of previous projects using the Equation 2.5:

$$PP = \frac{SLOC}{\left[\frac{E}{B}\right]^{\frac{1}{3}} \times T^{\frac{4}{3}}}, \tag{2.5}$$

where B scale factor is defined equal to 16. On the other hand, PP can be determined from the type of development environment according to Pressman [200]: 2000 is used for poor environments, 8000 is used for environments with adequate documentation and reviews, and 11000 is used for environments with automated tools and techniques.

Putnam’s model can be used to estimate the development effort in person-years refactoring the Equation 2.5 [201]:

$$E = \left[\frac{SLOC}{PP \times T^{\frac{4}{3}}}\right]^{3} \times B \quad \text{(person-years)} \tag{2.6}$$

On the other hand, the Equation 2.7 can be used to estimate the minimum time required to develop an application in years [201]:

$$t_{d-min} = 0.68 \times (SLOC/PP)^{0.43} \quad \text{(years)} \tag{2.7}$$

Halstead's Measures

Halstead introduced several measures for a software evaluation, including program length, program vocabulary, program volume (in bits), programming difficulty, development effort, and development time [87]. Aiming to perform his evaluations, Halstead initially defined a program as a collection of tokens classified as operators or operands. However, there is no standardized definition of an operand or operator for all programming languages [125]. In C++ language, for example, there are arithmetic operators (e.g., $+$, $-$, $*$, $/$, %), relational operators (e.g., $<$, $<=$, $>$, $>=$, ==, !=), logical operators (e.g., !, &&, $||$), and bitwise operators (e.g., &, $|$, $\wedge$, $\sim$, $<<$, $>>$). The keywords of the C++ language are also considered operators, such as the native functions, the class specifiers (e.g., `static` and `virtual`), type qualifiers (e.g., `const` and `friend`), and reserved instructions (e.g., `for`, `if`, `while`, `struct`, and `namespace`). In addition, ";" and parenthesis are considered operators. The operands are all those that are not operators, but constants, variables, typenames, and user-defined identifiers [125, 53].

Then, Halstead proposed the tokens measurement according to [183]:

- n_1: number of unique operators;
- n_2: number of unique operands;
- N_1: total occurrences of operators;
- N_2: total occurrences of operands.

Listing 2.10 presents a function in C++ language that calculates the factorial of a number. Table 2.1 shows the number of operators and operands, their occurrences in this code, and the total occurrences. This code has 12 operators (n_1) with 21 occurrences in total (N_1), and it has four operands (n_2) with 11 occurrences in total (N_2).

```
int factorial(int n){
    assert(n >= 0);
    if(n == 0){
        return 1;
    }
    else{
        return n * factorial(n - 1);
    }
}
```

Listing 2.10: Factorial function. Adapted from [180].

From the metrics proposed by Halstead, it is possible to calculate the program length (N). According to [121], the length can be used to estimate costs, schedule, and defect rates. The total number of operator occurrences and the total number of operand

Table 2.1: The number of tokens of a factorial function.

ID	**Operators**	**Occurrences**	**ID**	**Operands**	**Occurrences**
1	int	2	1	factorial	2
2	if	1	2	n	5
3	else	1	3	0	2
4	assert	4	1	1	2
5	return	2	-	-	-
6	>=	1	-	-	-
7	==	1	-	-	-
8	$*$	1	-	-	-
9	$-$	1	-	-	-
10	( )	4	-	-	-
11	$\{\}$	3	-	-	-
12	;	3	-	-	-
Total	n_1 = 12	N_1 = 21	Total	n_2 = 4	N_2 = 11

occurrences can be calculated from Equation (2.8) [87]:

$$N = N_1 + N_2 \tag{2.8}$$

The vocabulary (n) is measured through the total number of unique operators and unique operands using the Equation (2.9) [87]:

$$n = n_1 + n_2 \tag{2.9}$$

Using Halstead's metrics, it is also possible to measure the program volume (V) in bits. The program volume can be calculated from Equation (2.10) using the program length (N) and program vocabulary (n) [87]:

$$V = N \times log_2(n) \tag{2.10}$$

The program difficulty (D) measures how in terms of difficult it is to deal with the program, which is calculated from Equation (2.11) [87]:

$$D = \left(\frac{n_1}{2}\right) \times \left(\frac{N_2}{n_2}\right) \tag{2.11}$$

From Halstead's measures, it is possible to calculate the programming effort (E), which measures the number of elementary mental discriminations required to create a program [89]. We refer to elementary mental discrimination as the programmer's mental ability to make decisions during development. The programming effort can be obtained by multiplying the difficulty and volume of the program:

$$E = D \times V \tag{2.12}$$

It is also possible to measure the programming time (T) required to translate the existing algorithm that is being implemented into the specified programming language. The programming time is measured in seconds from Equation (2.13) [89]:

$$T = \frac{E}{S} \tag{2.13}$$

where the Stroud number S is the speed at which the brain makes elementary mental discrimination defined by the psychology domain as $5 \leq S \leq 20$ discrimination per second [75]. In software science, it has been defined that a program requires 18 discrimination per second to be encoded [75].

Function points analysis

In 1983, Albrecht and Gaffney Jr. [4] proposed a parametric technique to estimate the software development effort called Function Point Analysis (FPA). This technique is, in principle, independent of the programming language and the technology used in the development because it is based on the analysis of the functional requirements of the software. Therefore, the software does not need to be developed, only the requirements that define how the software will work. These requirements will be converted into numerical values, representing the effort needed to develop the application [245].

FPA can be used to measure the size of an application before its development. This technique can also be used to estimate the maintenance effort of software in case it is necessary to implement some change or add new functionality. Therefore, initially, it is necessary to determine the type of function point count among [67, 245]:

- **Development:** it is used to estimate the effort for the development of a new project;
- **Improvement:** it is used to estimate the evolution of software, in which features added, changed, and removed are counted;
- **Existing application:** it is used to count function points of existing applications.

After defining the function type, the function categories need to be identified [67, 203]. In FPAs, there are five categories mapping how functions and data elements are supported in an application: External Inputs (EIs), External Outputs (EOs), External Inquiries (EQs), Internal Logical Files (ILF), and External Interface Files (EIF). EIs, EOs, and EQs are called transaction functions and represent the functionalities provided to the user for data processing by an application. Whereas ILF and EIF are called data functions and are the structural representation of data [203]. Figure 2.8 illustrates each category and its functionality. In this figure, the user provides inputs to the system through EIs, receives outputs through EOs, and interacts with system functions through EQs. The inputs provided by the user are transformed via ILFs. In addition, the system interacts with other applications and external systems through EIFs, EIs, and EOs [67].

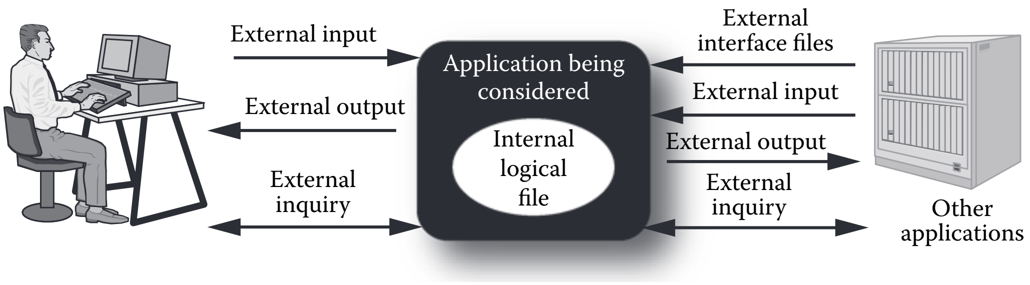


Figure 2.8: FPAs function units systems. Extract from [67].

After defining the category of functional requirement, its complexity can be calculated based on the number of following factors [203], according to Table 2.2:

- **Record Element Type (RET):** is the user-recognizable subset of data within an internal or external file (any class);
- **File Type Referenced (FTR):** is an internal or external file used in a transaction (a class that is not a component of another);
- **Data Element Type (DET):** is a unit of information, indivisible and recognizable by the user. It is usually the field of a table, a class attribute, or a function parameter.

After determining the complexity of the functions among low, medium, and high, the respective value can be obtained from Table 2.3. The number of Unadjusted Function Points (UFP) for the system as a whole will be the sum of the UFP obtained for each of the system functionalities. The UFP must be adjusted to take into account the internal technical complexity of the functions using the Value Adjustment Factor (VAF), which is calculated based on 14 General Systems Characteristics (GSC): data communications; distributed processing; performance objectives; operational configuration load; transaction rate; online data entry; end-user efficiency; online update; complex processing logic; reusability; installation ease; operational ease; multiple sites; and desire to facilitate change [132]. The Equation 2.14 must be used calculate VAF:

$$VAF = 0.65 + \left(0.01 \times \sum_{i=1}^{14} GSC_i\right), \tag{2.14}$$

where each of the GSCs should be given a score from zero to five to indicate their influence on the development process. From Adjusted Function Point (AFP) (Equation 2.15), it is possible to estimate the effort (Equation 2.16) needed to develop the application.

$$AFP = UFP \times VAF, \tag{2.15}$$

$$E = AFP \times PI \quad \text{(person-months)}, \tag{2.16}$$

where Productivity Index (PI) is obtained by evaluating previous projects (FPA/E). From the effort it is possible to calculate the development time using the Equation 2.17:

$$T = 2.5 \times \sqrt[3]{E} \quad \text{(months)}. \tag{2.17}$$

Table 2.2: Complexity of EIs, EOs, EQs, ILFs and EIFs. Adapted from [245].

		DET arguments		
	FTR classes	**1 to 4**	**5 to 15**	**16 or more**
EIs	**0 to 1**	low	low	average
	2	low	average	high
	3 or more	average	high	high
	FTR classes	**1 to 5**	**6 to 19**	**20 or more**
EOs and EQs	**1**	low	low	average
	2 to 3	low	average	high
	4 or more	average	high	high
	RET classes	**1 to 19**	**20 to 50**	**51 or more**
ILF and EIF	**1**	low	low	average
	2 to 5	low	average	high
	6 or more	average	high	high

Table 2.3: UFP by function type and complexity. Adapted from [245].

Function type	**Low**	**Average**	**High**
EI	3	4	6
EO	4	5	7
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

Use Case Points

Use Case Points (UCP) [112] is a model inspired by FPA, but focusing on the analysis of software use cases. First, the model measures the number of actors and use cases and their complexities (simple, average, or complex). Actors could be, for example, users or systems with which the application communicates. Use cases could be, for example, a customer balance query in a banking system. The complexities of the actors and the use cases are computed according to the weighting factors in the Table 2.4:

Table 2.4: Weighting factor of actors and use cases. Adapted from [112].

	Complexity	Definition	Weight
Actor	Simple	The actor represents another system with a defined application programming interface	1
	Average	The actor is an interaction with another system through a protocol, or a human interaction with a line terminal	2
	Complex	The actor interacts through a graphical user interface	3
Use case	Simple	Three or less transactions	1
	Average	Between four and seven transactions	2
	Complex	More than seven transactions	3

After define the weight of each actor and use case, the Unadjusted Use Case Points (UUCP) can be calculated using Equation 2.18 [112]:

$$UUCP = \sum_{i=1}^{6} n_i \times W_i, \tag{2.18}$$

where n are the numbers of actors and use cases of the simple, medium and complex types, and W is their respective weight.

Then, the impact of a series of technical factors (Tfactor) and environmental factors (Efactor) are evaluated. The set of 13 TFactors is: distributed systems, application performance objectives, end user efficiency (online), complex internal processing, reusability, installation ease, operational ease, portability, changeability, concurrency, special security features, provide direct access for third parties, and special user training facilities. The set of eight Efactors is: familiar with objectory, part-time workers, analyst capability, application experience, object-oriented experience, motivation, difficult programming language, and stable requirements. Each of the Tfactors and Efactors should be rated from zero to five, considering its impact on the development of the evaluated project, where zero means it is irrelevant and five means it is essential. These evaluation is used to get the Technical Complexity Factor (TCF) and Environment Complexity Factor (ECF) from the equations below:

$$TCF = 0.6 + (0.01 \times Tfactor), \tag{2.19}$$

$$ECF = 1.4 + (-0.03 \times Efactor), \tag{2.20}$$

Then the adjusted UCP can be obtained from the Equation 2.21:

$$UCP = UUCP \times TCF \times EFC, \tag{2.21}$$

Finally, it is possible to measure the development effort from the Equation 2.20:

$$E = UCP \times PI \quad \text{(person-hour).} \tag{2.22}$$

Constructive Cost Model

In 1981, Barry W. Boehm [25] proposed a model to estimate the effort and cost of software development called Construction Cost Model (COCOMO). This model was created from an empirical study in 63 projects at TRW Aerospace, where programs with 2 to 100 KSLOC and written in various languages (such as Assembly and PL/I) were examined [245]. COCOMO 81 is already obsolete and replaced by a new version in real applications. In 2000, COCOMO II was proposed by the University of Southern California (USC) to add more variability and accuracy into the initial version [26].

COCOMO II is designed to measure the effort and size of the development team for the elaboration and construction phases of the Unified Process, but it can also be used for the waterfall and Spiral models. Figure 2.9 presents the effort estimation region covered by COCOMO II. COCOMO II is applied to determine the effort required to develop the elaboration and construction phases of a project. The duration of the incept and transition phases of the unified process should be calculated by applying a percentage on the development effort of the elaboration and construction phases [245].

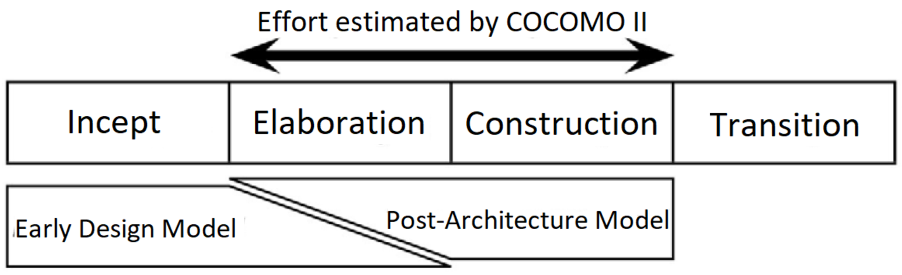


Figure 2.9: COCOMO II effort estimation region and application timing of early design and post-architecture models. Adapted from [245].

COCOMO II evaluates the development effort through the number of SLOC. In addition, through COCOMO cost drivers, it is possible to perform a detailed analysis of the development effort according to the application, developers, environment, and other issues that affect the development cycle. COCOMO II has cost drivers of the post-architecture and early design model types. Figure 2.9 shows that the Early Design Model should be applied in the incept and elaboration phases, and the Post-Architecture Model should be applied in the elaboration and construction phases [245].

The cost divers of the Post-architecture Model are divided into four groups: Product Factors, Platform Factors, Personnel Factors, and Project Factors. As seen in Table 2.5, each cost driver should receive a value from very low to extra high. Each of these cost drivers is described briefly below:

- Required Software Reliability (RELY): It evaluates the consequences regarding losses and risk to human life in case of any software failure;
- Complexity of the Product (CPLX): It evaluates project complexity from the subjective mean of five descriptors: control operations, computational operations, device-dependent operations, data management operations, and interface management operations;
- Required Reusability (RUSE): It evaluates the components produced by the software in relation to reuse;
- Database Size (DATA): It evaluates the size of the database used to test the program;
- Documentation match to life-cycle needs (DOCU): It assesses the documentation needed for development is produced;
- Execution Time Constraint (TIME): It evaluates the expected percentage of use of available processors by the application;
- Main Storage Constraint (STOR): it evaluates the expected percentage of use of the main memory by the application;
- Platform Volatility (PVOL): It evaluates the update period of the platform (hardware and software) on which the application is developed;
- Analyst Capability (ACAP): It evaluates the ability of analysts (in percentage) to analyze and model applications, cooperate, and communicate;
- Programming Capability (PCAP): It evaluates the ability of programmers (in percentage) to analyze and model applications, cooperate, and communicate;
- Personnel Continuity (PCON): It evaluates the exchange of developers (in percentage) in the development team in one year;
- Applications Experience (APEX): It evaluates the average time of experience of the team (in years) in developing applications similar to the one that will be developed;
- Platform Experience (PLEX): It evaluates the average time of experience of the team (in years) in using the development platform, such as hardware, libraries, operating system, database, and other related items;
- Language and Tool Experience (LTEX): It evaluates the average time of experience of the team (in years) in the CASE languages and tools used for the development of the application;
- Use of Software Tools (TOOL): It evaluates the quality of computational support to the development environment concerning the tools used;

- Multisite operation (SITE): It evaluates the influence of the distribution of the development team concerning the co-location (international, same country, same city, etc.) and communication (via telephone, e-mail, etc.);
- Required Development Schedule (SCED): It evaluates the percentage of acceleration of the schedule followed for the development of the application.

Table 2.5: Cost drivers values for the post-architecture and early design models. Adapted from [26].

Post Architecture Model							
Factor	**Name**	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
Product	RELY	0.82	0.92	1	1.1	1.26	NA
	CPLX	0.73	0.87	1	1.17	1.34	1.74
	RUSE	NA	0.95	1	1.07	1.15	1.24
	DATA	NA	0.9	1	1.14	1.28	NA
	DOCU	0.81	0.91	1	1.11	1.23	NA
Platform	TIME	NA	NA	1	1.11	1.29	1.63
	STOR	NA	NA	1	1.05	1.17	1.46
	PVOL	NA	0.87	1	1.15	1.3	NA
Personnel	ACAP	1.42	1.19	1	0.85	0.71	NA
	PCAP	1.34	1.15	1	0.88	0.76	NA
	PCON	1.29	1.12	1	0.9	0.81	NA
	APEX	1.22	1.1	1	0.88	0.81	NA
	PLEX	1.19	1.09	1	0.91	0.85	NA
	LTEX	1.2	1.09	1	0.91	0.84	NA
Project	TOOL	1.17	1.09	1	0.9	0.78	NA
	SITE	1.22	1.09	1	0.93	0.86	0.8
	SCED	1.43	1.14	1	1	1	NA
Early Design Model							
Name	**Extra Low**	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
PERS	2.12	1.62	1.26	1	0.83	0.63	0.5
RCPX	0.49	0.60	0.83	1	1.33	1.91	2.72
PDIF	NA	NA	0.87	1	1.29	1.81	2.61
PREX	1.59	1.33	1.22	1	0.87	0.74	0.62
FCIL	1.43	1.3	1.1	1	0.87	0.73	0.62

The early design models are: Personnel Capacity (PERS), Product Reliability (RCPX), Platform Difficulty (PDIF), Personnel Experience (PREX), Facilities (FCIL), RUSE and SCED. RUSE and SCED are the same in post-architecture model. As seen in Table 2.5, each cost driver should receive a value from extra low to extra high. The values of the cost drivers of the early design model result from combinations of the cost drivers of the post-architecture model and other information about the environment[1].

[1]For more details about the evaluation of cost drivers see [26].

COCOMO II model also implements a series of scale factors. As can be seen in Table 2.6, each of these scale factors should receive a value that represents very low, low, nominal, high, very high and extra high. Each one them is described briefly below:

- Precedentedness (PREC): It evaluates if the application developed is similar to applications previously developed by the same team;
- Development Flexibility (FLEX): It evaluates flexibility in development in relation to the requirements of the application;
- Architecture/Risk Resolution (RESL): It evaluates the existence of architecture or support system for risk resolution;
- Team Cohesion (TEAM): It evaluates the cohesion of the development team concerning to their experience and ability in working as a team;
- Process Maturity (PMAT): It evaluates the Capability Maturity Model Integration (CMMI) maturity level obtained by the company.

Table 2.6: Scale Factors values. Adapted from [26].

Scale Factors	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
PREC	6.2	4.96	3.72	2.48	1.24	0
FLEX	5.07	4.05	3.04	2.03	1.01	0
RESL	7.07	5.65	4.24	2.83	1.41	0
TEAM	5.48	4.38	3.29	2.19	1.1	0
PMAT	7.8	6.24	4.68	3.12	1.56	0

The scale factors of a project denoted as F_j, are summed across all of the factors and used to determine a scaling exponent S from Equation (2.23):

$$S = B + 0.001 \times \sum_{j=1}^{5} F_j \tag{2.23}$$

where B is a constant that must be calibrated according to historical values. According to [245] the suggested initial value for B is 0.91.

After calculating S value, it is used as an exponent for calculating the development effort in person-months from Equation (2.24):

$$E = A \times KSLOC^{S} \times \prod_{i=1}^{n} M_i \quad \text{(person-month)} \tag{2.24}$$

where A is a constant that must be calibrated from historical data and M_i is the set of cost drivers. According to [245] the suggested initial value for A is 2.94. Moreover, it is possible to calculate the ideal linear time to develop a project in running months from the Equation (2.25):

$$T = C \times E^{D+0.2\times(S-B)} \quad \text{(months)} \tag{2.25}$$

where B, C, and D are constants that must be calibrated from historical data. According to [245] the suggested initial values for C and D are 3.67 and 0.28, respectively. Finally, COCOMO II model also provides the number of people recommended for the development team. To do so, Equation (2.26) is used:

$$Team = \frac{E}{T} \tag{2.26}$$

COCOMO II is used to evaluate software development from scratch. However, variants of the model aim to model the development effort from existing software (reuse model) and for improvements or corrections to an already developed software (maintenance model). For this purpose, only the post-architecture model cost drivers are employed, and the size code must be measured again [26]. The maintenance size is calculated from the number of lines added to the original code (Added Source Lines of Code (ASLOC)) and the number of lines modified in the original code (Modified Source Lines of Code (MSLOC)) using the Equation 2.27:

$$SLOC_{Maintenance} = (ASLOC + MSLOC) \times MAF. \tag{2.27}$$

where MAF is the adjusted maintenance factor, calculated from the degree of understanding of the existing software and the programmer's relative lack of familiarity with the software developed. Finally, to measure the development effort using the COCOMO II maintenance model, substitute KSLOC for $SLOC_{Maintenance}$ in Eq 2.24.

Unlike the maintenance model, to estimate the development effort from software reuse, the COCOMO II is based only on the amount of software adapted (ASLOC). The automatically translated (AT) and adaptive modifier (AMM) factors are also considered together with ASLOC to calculate the reuse size from Equation 2.28:

$$SLOC_{Reuse} = ASLOC \times (1 - AT/100) \times AAM, \tag{2.28}$$

To measure the development effort using the COCOMO II reuse model, substitute KSLOC for $SLOC_{Reuse}$ in Eq 2.24.

SEER Software Estimating Model

SEER – Software Estimation Model (SEER-SEM) [67] is a parametric model developed by Galorath Inc. to estimate and analyze the effort, cost, staffing, schedule, and risk of a software project. To make an estimation, SEER-SEM has as input a series of factors divided into the following categories: effective size, effective technology, effective complexity, constraints, and probability. Regarding to development effort and time, their measures are based on size, complexity, and effective technology factors.

Effective size factor is calculated from the following equation:

$$S_e = L_x \times (AdjFactor \times size)^{\frac{Entropy}{1.2}} \tag{2.29}$$

where L_x is the language expansion factor, *AdjFactor* is the adjust factor of the program, and *Entropy* ranges from 1.04 to 1.2. To measure *size* variation, can be used the number of SLOC, function points, use cases, or other size metrics. In addition, when measuring the size of program code, it is necessary to consider whether it is new or reused.

Effective technology measures the developer's propensity for productivity based on the program's requirements to be developed. Its is calculated using an intermediate value called basic technology (C_{tb}). The intermediate technology is based on some factors similar to COCOMO cost drivers such: ACAP, PCAP, TOOL, Use of modern programming practices (MODP), and Application experience (AEXP), Computer turnaround time (TURN), and Terminal response time (TERM). Each of these factors should receive a value that represents very low, low, nominal, high, very high and extra high.

The intermediate technology can be calculated using the Eq 2.30:

$$C_{tb} = 200 \times exp\left(\frac{-3.70945 \times \left(\frac{ctbx}{4.11}\right)}{5 \times TURN}\right) \tag{2.30}$$

where *ctbx* factor is calculate from:

$$ctbx = ACAP \times AEXPAPPL \times MODP \times PCAP \times TOOL \times TERM \tag{2.31}$$

Next, effective technology factor can be calculated using the Eq 2.32:

$$C_{te} = \frac{C_{tb}}{ParmAdjustment} \tag{2.32}$$

where *ParmAdjustment* is calculated from a set of evaluation factors, such as language complexity, volatility, quality, and reuse.

Effective complexity measures the programming difficult, and can be calculated from:

$$D = K^{\frac{5}{2}} \times \frac{S_e}{C_{te}}^{3} \tag{2.33}$$

where K is the total Life-cycle effort (in person years) including development and maintenance. Finally, development time can be achieved from the Equation 2.34:

$$t_d = D^{0.2} \times \frac{S_e}{C_{te}}^{0.4} \quad \text{(months)} \tag{2.34}$$

Planning Poker

Planning poker is a variation of the Delphi technique [136], created by the RAND Corporation in the 1950s to refine group judgments. Nowadays, Planning Poker [39] is a metric commonly used by software agile teams to estimate the user story, which defines features and requirements that provide information to the user or customer [56]. It relies on experts' opinions about the software to be developed to guess the development effort. Participants in the planning poker method include all persons in the developing team, such as the developers, testers, engineers, analysts, and others. In addition, there is a moderator to coordinate the execution of the method.

Before starting, the moderator should prepare a deck of cards with a valid sequence of numbers written on each card. Planning poker decks are usually based on Fibonacci sequence [69], as can be seen in Figure 2.10, which are presented different options of decks. The original Fibonacci sequence also can be used, although modifying the Fibonacci sequence allows development team members to estimate projects with development effort close to 1/2. In addition, each number must have a meaning, such as story points, number of Product Backlog, and development time. Next, the moderator or team leader should explain the requirements of the application or project to be developed. Then, each participant receives a deck of cards and selects a card representing their estimation opinion. All participants must discuss to justify their choice. If the experts disagree with the estimates, they can repeat the process until the results converge. On the other hand, averaging the estimates can be done to avoid too many rounds [39]. Figure 2.11 illustrates the steps required to use the Planning Poker method to estimate story points.

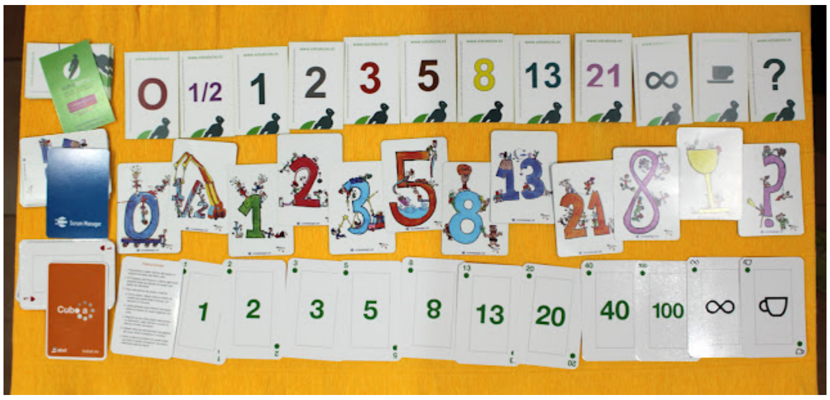


Figure 2.10: Different decks of planning poker cards. Extracted from [156].

According to Cohn [39], Planning Poker is a metric that works by several factors. First, it gathers various opinions from estimating experts, who form a cross-functional team from all stages of a software project. Therefore, they are better suited to the estimating task than anyone else. Secondly, estimators are called upon to justify their estimates, a factor that improves the estimation's accuracy. Finally, studies have shown that individual averaging estimates lead to better results, as well as, does holding group discussions.

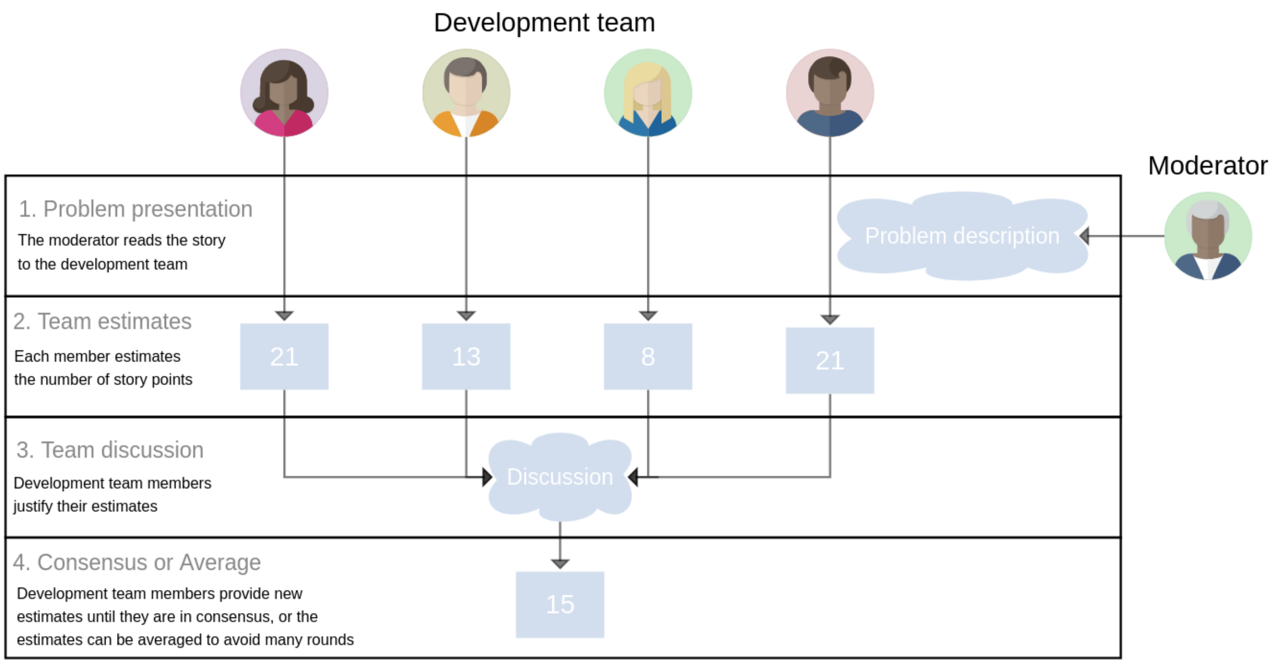

Development team
Moderator
1. Problem presentation
The moderator reads the story to the development team
Problem description
2. Team estimates
Each member estimates the number of story points
21
13
8
21
3. Team discussion
Development team members justify their estimates
Discussion
4. Consensus or Average
Development team members provide new estimates until they are in consensus, or the estimates can be averaged to avoid many rounds
15

Figure 2.11: Poker planning steps for estimating the number of story points.

2.5.2 Accuracy Methods

It is necessary to measure the accuracy of estimation models to verify the proximity between the estimated and actual values, which should be as small as possible [242]. This section presents some of the most popular metrics for accuracy evaluation according to Venkataiah *et al.* [242]. One of the first metrics to evaluate accuracy was introduced in 1981 by Bohen *et al.* called percentage error, which measures the percentage of the error relative to the actual value. The percentage error can be calculated from the Equation (2.35):

$$Percentage\_Error = \frac{Actual\_Effort - Estimated\_Effort}{Actual\_Effort} \times 100 \tag{2.35}$$

Conte *et al.* [41] introduced two methods to assess the accuracy commonly used: Magnitude of Relative Error (MRE) and Mean Magnitude of Relative Error (MMRE). MRE can be calculated from Equation (2.36):

$$MRE = \frac{|Actual\_Effort - Estimated\_Effort|}{Actual\_Effort} \tag{2.36}$$

According to [198], MRE is useful because it does not penalize excessively large projects. Moreover, it is unit-less, that is, it is independent of the scale of the project.

MMRE is a method based on MRE and can be calculated from Equation (2.37):

$$MMRE = \frac{1}{n}\sum_{i=1}^{n} MRE_i \tag{2.37}$$

where n is the number of projects and i is the project identifier. According to [41], MMRE ≤ 0.25 is considered an acceptable performance level for models and effort estimation. However, the value of the MMRE can be strongly influenced by some very high values of MRE, which can be a problem [111].

Median magnitude of relative error (MdMRE) is the median of all MRE's, which can be calculated from Equation (2.38) [242]:

$$MdMRE = median(MRE_j) \tag{2.38}$$

MdMRE has been used as an accuracy criterion instead of MMRE because it is less sensitive to outliers [176]. In addition, if the estimation model has MdMRE ≤ 0.25 it is considered a good predicition model [41].

Percentage Relative Error Deviation (PRED) within x is another method based on MRE, which can be calculated from Equation (2.39) [198]:

$$PRED(x) = \frac{\sum_{i=1}^{n}}{n} \begin{cases} i & if\ MRE \leq x \\ 0 & otherwise \end{cases} \tag{2.39}$$

where n is the number of projects, and the fraction numerator is the sum of the number of projects that MRE is less than or equal to x. Generally, the accuracy comparison is based on PRED(0.25) (the percentage of the tasks with MRE ≤ 0.25) [111] [198]. However, some studies also analyze PRED(0.3) with little difference in results compared to PRED(0.25). Generally, PRED(0.25) ≤ 0.75 (or PRED(0.3) ≤ 0.75) is considered an acceptable model accuracy [198].

2.6 Final remarks

This chapter overviewed parallel architectures and patterns in structured parallel programming. We presented the domain of stream processing, in which parallelism is exploited from these parallel patterns. We also presented some interfaces based on structured parallel programming to explore multi-core and heterogeneous architectures. Furthermore, we presented a set of code metrics used to obtain productivity indicators and some metrics used to measure their accuracy. In the next chapter, we will review the literature to understand the process of evaluating productivity in the parallel programming domain.

3. LITERATURE REVIEW

The popularization of parallel architecture in our daily computing systems leveraged parallel programming, which is well-known to be a complex task and most reserved for specialists. Consequently, research and industry have promoted and developed multiple PPIs to ease this task. Unfortunately, only a few studies aimed to evaluate the productivity of such PPIs. Most studies focused on evaluating their PPIs regarding the performance, where only runtime, speedup, and efficiency of the parallel software are analyzed [15, 134, 227, 230]. It is still a task for experts, as only a few studies care about productivity. One way to develop more parallel code is to make parallel programming easier for the application developers, who are not experts or specialists in system programming. Therefore, they could develop parallel applications productively.

According to ISO 9241-11 [108], productivity or efficiency is a subset of the usability properties of software. Usability is generically described by ISO 9241-11 [108] how the "*extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use*". Translating to parallel programming, we describe that **usability extent to which a PPI or tool can be used by developers to make source code execute computations in parallel with effectiveness, efficiency, and satisfaction in a specified context of use**. Effectiveness is described as "*accuracy and completeness with which users achieve specified goals*". In parallel programming, effectiveness can be measured by the accuracy of the application developed by the programmer, in other words, whether the application returns the expected results. Efficiency, also called productivity [107], is defined concerning the "*resources used in relation to the results achieved*" [108]. These resources include the time required to complete a task [18], human effort, costs, and materials [108]. In the context of parallel programming, development time and development effort are measures that can be used to evaluate the efficiency of PPIs. Satisfaction is defined as "*extent to which the user's physical, cognitive and emotional responses that result from the use of a system, product or service meet the user's needs and expectations*". In the parallel programming domain, satisfaction assesses whether the PPIs meets the programmer's needs and expectations.

In parallel programming, the development time required by developers to parallelize an application has been a relevant metric used to evaluate productivity [78]. In addition, metrics commonly used in SE, such as SLOC, McCabe's CCN, Halstead, and COCOMO, have been used in the parallel programming domain to evaluate the productivity. Through initial exploratory research, we have identified some studies that also refer to such metrics as usability indicators [38, 173, 171, 186, 233]. In this context, the question we aim to answer in this chapter is "*how are productivity and usability evaluated in the parallel programming domain?*". Therefore, this chapter presents a literature review to map how productivity and usability have been evaluated in parallel programming and the metrics used for this assessment. In addition, we aim to discover the PPIs considered, the architectures explored, and the applications type evaluated.

3.1 Search method

In this section, we presented the search method used in the literature review conducted.

3.1.1 Research questions

We structured the research questions based on three viewpoints according to Kitchenham and Charters [120]: Population, Intervention, and Outcomes (PIO).

Table 3.1: Research questions structured using PIO criteria.

Population	Software/application development
Intervention	Methods/techniques/models used for estimating the productivity and usability of PPIs
Outcomes	Methods/techniques/models used for estimating the productivity and usability of PPIs, architectures and domain evaluated

We aim answer the follow research questions: How is performed the evaluation of productivity and usability of PPIs? From this research questions we defined the following three sub-questions:

- **RQ1:** What approaches are used to evaluate the usability of the PPIs?
- **RQ2:** What methods and metrics are used to evaluated the productivity of the PPIs?
- **RQ3:** What are the PPIs evaluated?
- **RQ4:** What are the architectures evaluated?
- **RQ5:** What are the applications evaluated?
- **RQ6:** What are the types of developers?

RQ1 aims to discover the approaches used to evaluate the usability of PPIs. That is, whether effectiveness, efficiency, and user satisfaction are considered in this evaluation. RQ2 seeks to discover the methods and metrics used to evaluate the productivity of PPIs. RQ2 also aims to categorize the types of productivity methods and metrics used. RQ3 and RQ4 aim to show the already evaluated PPIs concerning productivity and which architectures they exploit. RQ5 seeks to describe the target applications of productivity assessment. Furthermore, RQ6 aims to categorize the type of developers of the evaluated applications, whether they are novice developers in the parallel programming domain or whether they are experts in the area.

3.1.2 Data bases

We performed the search in the following databases: ACM Digital Library[1], IEEE *Xplore*[2], Scopus[3], Engineering Village[4], and Web of Science[5]. We selected the Scopus database because it is the largest indexing database [20]. ACM Digital Library, Engineering Village, and IEEE *Xplore* were selected because they are predominately cited in computer science-related sources [216].

For the selected databases, we considered the search string presented in Table 3.2, developed based on the research questions. To compose the search sequence, we divided the search questions into three main aspects: evaluation, productivity/usability, and parallel programming. In addition, we considered the search string in the title, abstract, and keywords of the studies. In Scopus, Engineering Village, and Web of Science, we restricted the search to the Computer domain because they returned studies from other areas, such as business.

Table 3.2: Search terms used for the literature review.

Search term	**Related terms**
Evaluation	(“assess*” OR “evaluate*” OR “examin*”)
Productivity	(“development effort” OR “programming effort” OR “productivity” OR “usability”)
Parallel programming	(“parallel programming” OR “parallel computing” OR “hpc” OR “high-performance computing”)

3.1.3 Studies selection criteria

This section presents the study selection criteria, which aim to identify primary studies that provide direct evidence about the research questions [120].

[1]Available at: https://dl.acm.org/
[2]Availableat:https://ieeexplore.ieee.org/
[3]Available at: https://www.scopus.com
[4]Available at: https://www.engineeringvillage.com/
[5]Available at: https://www.webofscience.com/

We considered the following inclusion criteria in the study:

1. Studies that performed experiments with developers to assess PPIs usability.
2. Studies that use coding metrics to assess PPIs usability or productivity.
3. Journal articles and conference papers because they are reviewed more rigorously.

We considered the following exclusion criteria in the study:

1. Studies that talk about productivity assessment but do not actually assess it.
2. Duplicates studies, when there are many versions of a study in different journals. In this case, we considered the full version of the study.
3. Studies not available for download.
4. Studies not written in the English language.
5. Abstracts, technical reports, thesis, and books because journal articles and conference papers are more rigorously reviewed.

3.1.4 Studies selection and extraction process

We selected the studies in five steps according to Figure 3.1: search execution, duplicate removal, a first filter, a second filter, and snowballing method [228]. The search on the databases returned 688 studies. We exported them in BibTeX format to the Zotero tool[6] to help us in the review process. Using Zotero, we first removed the duplicate studies, resulting in 356 studies. In the first filter, we analyzed the studies using the inclusion criteria from reading the title, abstract, and keywords, resulting in 133 studies. From the second first, we analyzed the studies' full text using the inclusion and exclusion criteria, resulting in 57 studies.

To complement our literature review, we used the snowballing procedure (backward and forward) to include new papers using the inclusion and exclusion criteria [251]. First, we used forward snowballing to identify new papers based on the citations to the 57 selected papers. The citations were analyzed using Google Scholar[7]. Next, we used backward snowballing to analyze the selected studies' reference lists to identify new studies. We used the inclusion and exclusion criteria for backward and forward snowballing to decide if the papers would be included. Finally, we selected 110 studies to analyze and extract data.

[6]Available at: https://www.zotero.org/
[7]Available at https://scholar.google.com.br/

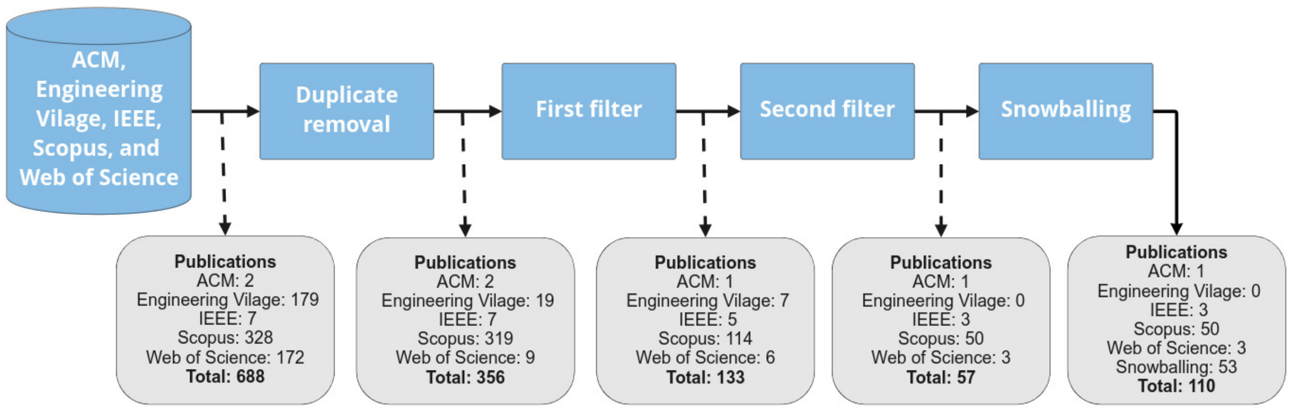

ACM, Engineering Vilage, IEEE, Scopus, and Web of Science
Duplicate removal
First filter
Second filter
Snowballing
Publications ACM: 2 Engineering Vilage: 179 IEEE: 7 Scopus: 328 Web of Science: 172 Total: 688
Publications ACM: 2 Engineering Vilage: 19 IEEE: 7 Scopus: 319 Web of Science: 9 Total: 356
Publications ACM: 1 Engineering Vilage: 7 IEEE: 5 Scopus: 114 Web of Science: 6 Total: 133
Publications ACM: 1 Engineering Vilage: 0 IEEE: 3 Scopus: 50 Web of Science: 3 Total: 57
Publications ACM: 1 Engineering Vilage: 0 IEEE: 3 Scopus: 50 Web of Science: 3 Snowballing: 53 Total: 110

Figure 3.1: Paper selection process.

3.2 Result analysis and discussion

In this section, we discuss the studies found in the literature.

3.2.1 Overview of the usability approach used

According to ISO 9241-11 [108], three factors must be evaluated to determine the usability of a system, product, or service: effectiveness, efficiency, and user satisfaction. In the parallel programming domain, different approaches have been used to evaluate usability (RQ1). In [233], the authors defined usability as how easy it is for a graduate student to design, develop, code, test and debug an application. Among the 110 articles from the literature review, we identified only 40 studies that performed experiments with people aimed to evaluate usability. Yet claiming to do usability experiments with people, [3, 38, 47, 57, 58, 95, 104, 135, 142, 159, 158, 160, 169, 173, 172, 188, 187, 190, 213, 225, 233, 235, 240, 249, 248, 257, 258] performed software experimentation without taking into account all the best practices, such as develop an experimental plan, performed a hypotheses test, and evaluate the threats to validity [252]. On the other hand, [32, 45, 78, 96, 94, 128, 127, 171, 189, 191, 211, 229, 247] considered the best practices for evaluating their solutions.

ISO 9241-11 [108] describes effectiveness as the precision and completeness with which users achieve specified goals. In parallel programming, the goal is generally achieved when performance is obtained by parallelizing an application using a specific PPIs. In order to evaluate the performance of the target PPIs, works found in the literature use metrics such as execution time and speedup. In addition, in the studies that conduct experiments with people, the performance achieved by the participants when performing the proposed activities is evaluated.

Different metrics have been used in parallel programming regarding efficiency or productivity evaluation. On the one hand, some studies perform experiments on people to collect the actual time taken to parallelize applications. However, experiments with persons must be well planned and controlled [252] and involve ethical issues. Therefore, many parallel programming researchers instead use established code metrics to facilitate productivity evaluation, such as SLOC, Halstead, and COCOMO. We will discuss these metrics in more detail in the next section.

User satisfaction is the last factor to be evaluated to assess usability. However, 70 of the articles that claim to evaluate PPIs usability do not conduct experiments with people and consequently do not consider their opinions in the evaluations. On the other hand, of the 40 studies that conducted experiments with people, only [38, 173, 160, 171, 172, 188, 189, 213, 225, 233, 235] considered the participants' satisfaction when using the PPIs evaluated.

3.2.2 Overview of the productivity metrics used

Figure 3.2 summarizes the productivity metrics used (RQ2). The metrics were divided into the following categories: *real-time based*, *number of activities performed*, *code correctness*, *code size*, *code occurrences*, *control flow*, *development estimation metrics*, and *productivity increase*. The researchers collected the real-time based metrics through experiments with people, where was measured the actual time spent by the participants to complete a specific task (i.e., parallelize an application, solve a problem, and solve a test). Most studies focus on measuring the total time required by the experiment participants to complete the target task. While other works evaluate the time necessary to perform different activities, such as study activity, development, testing, debugging, and others [57, 95, 187, 190, 248, 258]. On the other hand, some researchers decided to measure the number of times the experiment participant performed a given task. These metrics include the number of commits, edits, compiles, and executions of the parallel application [169, 171, 233]. Furthermore, in [47], a method was proposed to measure productivity based on the number of steps to complete a given task, the number of context shifts, and the working memory load (that derives from data operations) required at each step. Danis *et al.* [47] defined context shifts as the time and mental effort required to orient to the new context, for example, clicking on the browser icon and changing the context to a browser window.

Figure 3.2 also presents metrics used to get productivity indicators from the evaluation of code correctness, where higher code correctness indicates higher coding productivity. For this purpose are used metrics such as the number of correct programs, number of incorrect programs, number of programming errors or bugs in the code, and type of errors [235]. The types of errors evaluated by the papers in the literature are divided between compile-time, logical, hanging, and non-hanging errors [32, 45]. Compile-time errors are detected while a program is being compiled and are usually caused by syntax errors [27]. A logical error occurs due to some flaw in the program's logic and generates incorrect, unexpected, or unintended output [27]. In addition, Castor *et al.* [32] defined hanging and non-hanging errors as errors related to PPIs. Errors such as deadlocks and infinite loops are hang errors that cause the program to hang. On the other hand, non-hanging errors are concurrency errors that do not cause the program to hang, such as race conditions.

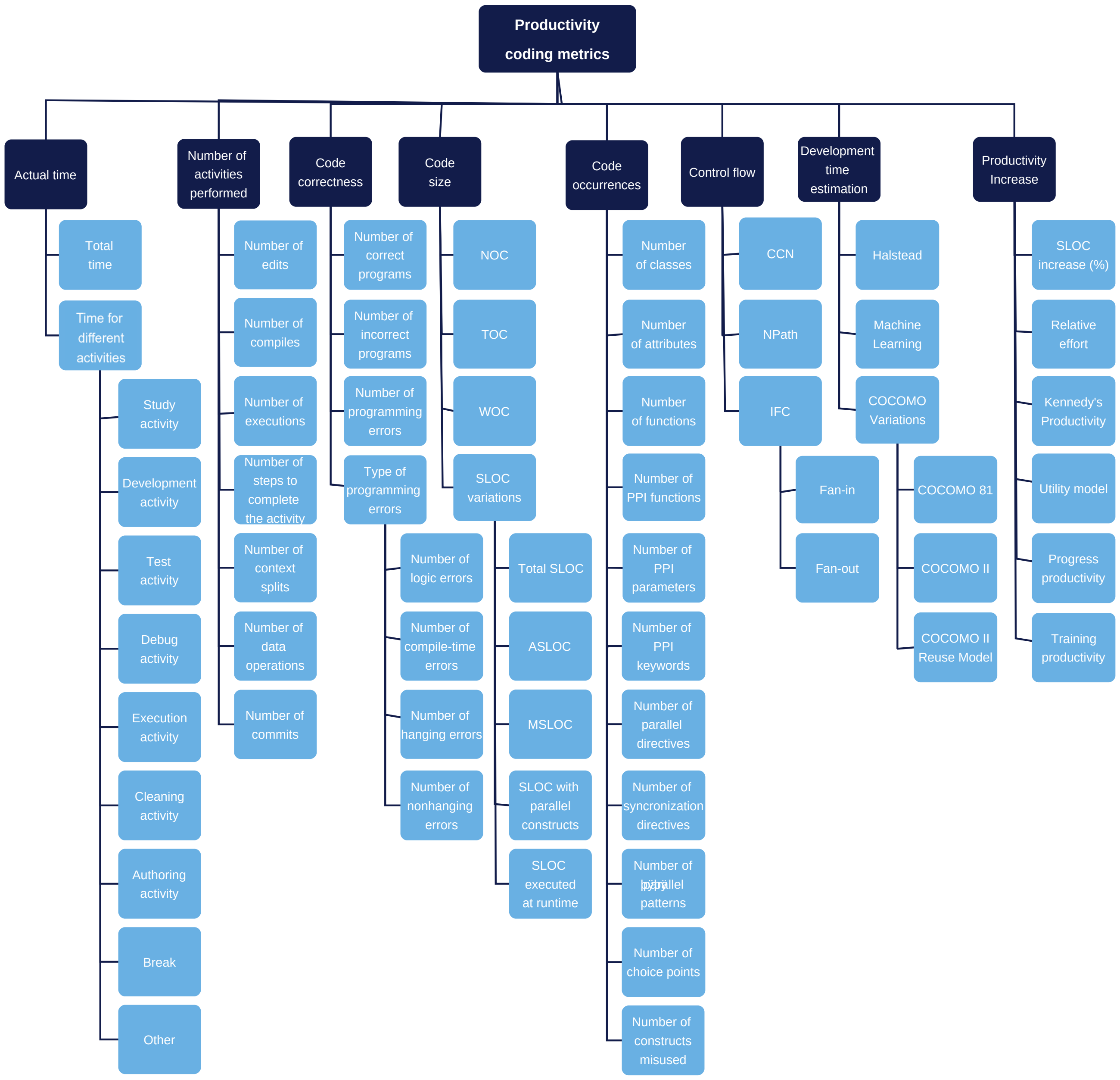


Figure 3.2: Metrics used to evaluate coding productivity in parallel programming assessment.

Still, regarding the evaluation of parallel application code, some metrics aim to evaluate code size to obtain productivity indicators. The most classical are SLOC, ASLOC, MSLOC, Number of Characteres (NOC), Tokens of Code (TOC), and Words of Code (WOC). In addition, SLOC is the most widely used metric for evaluating code size, which is used by 76 articles. There are variations of SLOC less used in the literature, such as the SLOC with parallel directives [155, 168], and the SLOC executed at runtime [129]. Lima and Domenico [129] defined the SLOC executed at runtime as the number of lines that will be executed at runtime ignoring comments, blank lines, data declarations and headers.

Some researchers have also measured the occurrence of structures related to the PPIs used, such as the number of PPI functions [3, 139], number of PPI parameters [139], number of PPI keywords [139], number of parallel directives, number of synchronization directives, and number of parallel patterns (pipeline, map-reduce, or other) [181]. MacDonald *et al.* measured the number of choice points, any point in a program where the control flow can be altered and no longer be sequential. There are other metrics related to the sequential code, such as the number of classes, number of attributes, and number of functions[135, 178]. In addition, some studies use metrics that evaluate code complexity from its representation as control flow graphs, such as CCN [37, 105, 104, 130, 143, 146, 169, 178, 184, 196, 194, 195, 209, 211], fan-in [195], fan-out [195], and NPath [37].

Some researchers evaluate coding productivity without considering the human factor in their evaluations. Instead, they have used predictive metrics of development effort in order to evaluate PPIs. Halstead's measure has been the most widely used to compare the productivity of PPIs [65, 125, 130, 143, 146, 164, 184, 192, 196, 194, 195, 209]. However, Halstead only evaluates the number of TOC without taking into account factors that impact the effort of developing parallel applications, such as the programming model and architecture. A recent study [143] proposed a metric based on the effort estimated by Halstead's metric to develop a sequential CPU application to predicts the percentage of extra effort to develop GPU code. The metrics used as features in the regression prediction model were the number of hotspots, number of SLOC, number of hotspots' SLOC, number of hotspots' statements, distinct and total operators of CPU version, CPU CCN, and Halstead's volume, length and difficulty of the CPU version. Different regression models were used to make the predicition, such as Randon Forest, Bagging Trees, Gradient Boosting, SVR, Bayessian Ridge, Decission Tree, and K-nearest neighboors. The results showed that Random forest produced the highest accuracy.

Still, due to the limitations of metrics like Halstead, recent studies have aimed to evaluate PPIs using more robust metrics as some variations of COCOMO: COCOMO 81 organic model [1, 74, 105, 103, 174], COCOMO II [78, 159], and COCOMO II reuse model [177]. In addition, in [248] was proposed an extension of the COCOMO II. The new model was built on regression analysis identifying and evaluating effort cost drivers important in HPC, based on a performance life-cycle through a survey posed to HPC developers.

The authors identified 11 cost drivers: pre-knowledge of architecture/hardware and parallel programming model, pre-knowledge of the numerical algorithm used, code work, architecture/hardware, parallel programming model and compiler/runtime system, tools, performance, energy efficiency, kind of algorithm, code size, and portability and maintainability over code's lifetime. However, the COCOMO II extension was not evaluated through accuracy metrics as it is commonly performed in SE (Section 2.5.2). Instead, Wienke *et al.* [248] have separately evaluated the factors they consider to most affect the development effort of parallel applications, such as pre-knowledge (surveys) and parallel programming model and architecture (student experiment) [247].

To measure the increase in productivity of parallel applications compared to sequential applications, some researchers use metrics based on the increase in SLOC, development time, and execution time. Some works [8, 44, 50, 124, 155, 181, 231] consider the increase in the number of SLOC of the parallel application over the sequential application as a productivity factor (Eq (3.1)). Kennedy's productivity [116] is another metric used [73, 257], which calculated a productivity factor for parallel programming from speedup and the relative effort (Eq (3.4)). The relative effort can be calculated from the development of the parallel and sequential applications or from their SLOC. In addition, [257] proposed a modification in Kennedy's productivity metric. Equation 3.5 presents the modification method, where r is the number of executions of the program over its lifetime and varies between 0 and 1 [257].

Gmys *et al.* [73] used a modification of the productivity model proposed by Snir and Bader [226], called Utility model (Eq. 3.6). In Gmys *et al.* [73] modification S_p is the Speedup, E is the efficiency achieved (S_p/number of threads), A is the availability of the system, C_s is the software cost measure in SLOC, C_M is the machine cost, and C_o and ownership cost. In addition, A was set equal to 100%, and both C_M and C_o were set equal to zero. Miller and Arenaz [158] also proposed a new methodology to measure the productivity based on two key components: the productivity of progress and the productivity of training. Productivity progress is defined as a ratio between the percentage of milestones completed during the training event and the cost of the event. Eq. 3.7 can be used to measure the productivity progress, where T is the number of trainees, M is the number of milestones, and $M_{t,comp}$ is the number of milestones completed by t-th trainee. On the other hand, training productivity can be calculated using the Eq. 3.8, considering the following factors: total human knowledge improvements (K), total software improvements (S), total long-term improvements (L), and associated cost of the training (C).

Table 3.3: Metrics for quantifying productivity in parallel applications.

Metric	**Equation**	
SLOC increase (%)	$SLOC\ increase = \frac{Parallel\ SLOC}{Sequential\ SLOC} \times 100$	(3.1)
Speedup	$Speedup = \frac{Sequential\ execution\ time}{Parallel\ execution\ time}$	(3.2)
Relative effort	$Relative\ effort = \frac{Parallel\ effort}{Sequential\ effort}$	(3.3)
Kennedy's productivity	$Productivity = \frac{Speedup}{Relative\ Effort}$	(3.4)
Kennedy's productivity modification	$Relative\ effort = \frac{(Parallel\ effort + r \times Sequential\ effort)}{(Sequential\ effort + r \times Parallel\ effort)}$	(3.5)
Utility model	$\psi = \frac{S_p \times E \times A}{C_s \times C_o \times C_M}$	(3.6)
Progress productivity	$PP = \frac{\sum_t^T M_{t,comp}}{M \times T \times Cost}$	(3.7)
Training productivity	$TP = \frac{\sum(\alpha_{\kappa_i} \times \kappa_i) + \sum(\alpha_{S_i} \times S_i) + \sum(\alpha_{L_i} \times L_i)}{Cost}$	(3.8)

3.2.3 Studies overview and their evaluation environments

In the section 3.2.2, we showed an overview of the metrics used to evaluate the usability of PPIs. On the one hand, some researchers have conducted empirical studies to assess usability, measuring the real-time required by developers to parallelize applications. On the other hand, some studies claimed to do usability evaluation without actually performing software experimentation with people. In this context, this section presents an overview of these studies divided between studies that perform controlled experiments with people and studies that do not consider the human factor in their evaluations.

Parallel programming evaluation through software experimentation

Table 3.4 presents the studies that perform experiments with people, which contain the PPIs evaluated, participants, metrics used, target architecture, and the activity performed by the participants. These studies are sorted by publication year. In 1996, Szafron *et al.* [233] conducted one of the first controlled experiments with graduate students to evaluate the usability of PPIs. In this experiment, Szafron *et al.* [233] aimed to evaluate the usability of two PPIs for programming in distributed memory computing environments, which are the Enterprise PPS interface and the PVM-like (NMP) library of message-passing routines. Using a similar approach, Singh *et al.* [225] compared the development effort of Enterprise and PVM with a commercially available tool that allows loop iterations to be done in parallel (PAMS). After the initial studies by Szafron *et al.* [233], other researchers have also conducted empirical studies with students to evaluate the usability of PPIs.

Still, on usability evaluation of interfaces to distributed memory systems, a study was performed to evaluate the effort spent by novices to develop Message Passage Interface (MPI) programs in [3]. Patel *et al.* [191] aimed to compare the performance and productivity of MPI and Unified Parallel C (UPC) programs. In [57], another experiment was performed to compare the productivity of C+MPI, UPC, and the x10 language of the IBM PERCS project. In [57], productivity was evaluated through an experiment with undergraduate students with little or no parallel programming experience. In [235], the authors presented a methodology for evaluating UPC programmability against MPI through classroom studies with a group of novice programmers. Speyer *et al.* [229] evaluated the productivity and usability of Charm++, UPC, SCOOP, MATLAB+Star-P, and SHMEM through an experiment with computer science and engineering students.

Zhang *et al.* [258] proposed a methodology for determining heuristics to identify the workflow of parallel application programmers from captured low-level data. To evaluate the accuracy of this heuristic, Zhang *et al.* conducted two case studies with graduate students using MPI to develop the applications. In [169], a pilot study was conducted to assess the usability of MPI when implementing design patterns in contrast to alternative implementations of the parallel program. Zelkowitz *et al.* [257] evaluated the productivity of MPI in developing three different applications using a series of measures, including Kennedy's productivity model. In addition, they proposed a modification to Kennedy's productivity assessment model.

The main goal in [96] was to evaluate the effort of beginners in parallel programming to develop parallel applications in MPI and OpenMP. The authors concluded that MPI requires more programming effort when compared to OpenMP based on the number of SLOC and the time required for participants to develop the applications. To complement the previous study, Hochstein *et al.* [95] combined two techniques for collecting the data regarding development time: self-reported and automatically collected. In addition, Hochstein *et al.* [95] evaluated the time spent by the participants to perform different development tasks, such as understanding the target problem, thinking about the solution, debugging the code, among others. This was a way to increase the accuracy of the data collected, because the self-reported data was not always correct.

In [188] the main goal was to evaluate the effort of graduate students to develop an actual program for multi-core computers using OpenMP and Pthreads. Coblenz *et al.* [38] compared Cilk Plus and OpenMP to evaluate the design trade-offs in the usability and security of these approaches. In [142], Pthreads and OpenMP were compared to OpenMP _XN programming model. OpenMP _XN is an extension of OpenMP with Atomic Sections of code code executed atomically and mutually exclusive from other conflicting atomic operations.

In [213], an empirical study was conducted with novice and expert Java programmers to identify multithreaded bugs in Java Threads code examples. The results were obtained through self-evaluation questionnaires, where feedback from the participants was also collected. In [171], the authors compare Java Threads and SCOOP for comprehending and debugging existing programs and writing correct new programs. The participants were undergraduate students in the software architecture course. In [189], an experiment was conducted with Master’s students who are, on average, in their fourth year of Computer Science studies to compare the development of parallel applications for multi-core systems using Scala and Java Threads. MacDonald *et al.* [135] evaluated the usability of CO_2P_3S, a tool that implements the parallel design patterns process, to build correct and functional parallel programs. To do so, MacDonald *et al.* [135] performed a study with 20 undergraduate students divided into two groups. The first group used non-CO_2P_3S Java (with a barrier class provided), and the other group used CO_2P_3S.

Rossbach *et al.* [211] performed a study with 237 undergraduate students of an Operating System course. The main goal of this study was to verify and compare the different techniques used in transactional memory programming with Java Threads: using coarse- and fine-grain locks, monitors, and transactions. In [187], a study was performed to compare teams of programmers developing a parallel program from scratch using Pthreads and Intel Software Transactional Memory (STM) compiler. Similar to the previous study, the experiment performed by Castor *et al.* [32] also aimed to evaluate the use of locks and transactional memory. Castor *et al.* [32] evaluated the effort spent by novices to develop a simple program with mutual exclusion and synchronization requirements using Halskell’s transaction memory and lock-based concurrency control mechanisms.

Nanz *et al.* [173] compared Chapel, Cilk, Go, and TBB in a study based on a sequential and parallel implementation of six benchmarks created by notable programmers with more than six years of experience, while Nanz *et al.* [172] performed an experiment that explores the claimed gap between expert and novice parallel programmers. In [172], the fraction of the original development time spent on implementing the corrections suggested by experts was also measured. Moreover, in [173], the Wilcoxon signed-rank test (two-sided variant) was used to evaluate the results.

In [240], the goal was to evaluate three types of platforms in terms of application performance, programming effort, and cost: generic multi-core CPU, GPU, and STI Cell Broadband Engine (a heterogeneous multi-core processor, designed by Sony, Toshiba and IBM). The participants in this study used Pthreads to evaluate the multi-core environment, Cell to evaluate STI Cell Broadband Engine, and CUDA to evaluate the GPU [240]. Li *et al.* [128] conducted an empirical investigation to compare the productivity of the OpenACC and CUDA GPU programming interfaces when used by undergraduate students in a classroom environment. In 2018, the previous study was complemented by evaluating the performance and code size [127]. Another more recent study was conducted to compare the scheduling productivity of CUDA with Thrust library [45].

Wienke *et al.* [249] compared the effort required to develop parallel applications for multi-core environments versus GPU environments through hackathons with beginners students. To do this, the students parallelized the applications using OpenCL and OpenACC for GPU, and OpenMP and a combination of OpenMP and Intel's Language Extensions for Offload (LEO) in multi-core. In [247], the goal was to evaluate and compare the effort required by hackathon participants to parallelize applications using OpenMP, OpenACC, and CUDA. In 2016, Wienke *et al.* [248] proposed an extension of the COCOMO II to evaluate parallel programming development. However, the proposed method was not evaluated in the present study because according to Wienke *et al.* [248] it is difficult to translate the COCOMO cost drivers to the parallel programming area. Finally, Wienke *et al.* [248] introduced EffortLog, a tool that allows comprehensive data collection of development time and performance measures with little overhead. Aiming to evaluate EffortLog, Wienke *et al.* [248] assess the productivity of OpenACC and CUDA through hackathons with beginner students in this domain.

In [159], an experiment was conducted with undergraduate students to compare the accuracy of the development time reported by the participants with the development time estimated by COCOMO II. Initial results showed that the development time estimated by COCOMO II showed inaccuracy. Miller *et al.* [158] also conducted a study to quantify the impact of HPC training by measuring learners' productivity in heterogeneous systems. Miller *et al.* [158] compared applications developed by beginners students using MPI, OpenMP, OpenACC, and OpenMP+OpenACC to assess productivity. To evaluate training effectiveness, the authors proposed a new methodology with two key components: progress productivity and training productivity.

In [78], the authors aimed to evaluate the performance and usability of the primary/secondary pattern of the Domain-Specific Language for Pattern-Oriented Parallel

Table 3.4: Studies performing human experimentation to evaluated PPIs usability and productivity.

Work	PPIs	Participants	Metrics	Architecture	Activity
[233]	Enterprise PPS and PVM-like	Graduate students	Login hours, SLOC, number of edits, number of compiles, and number of execution	Cluster of Workstations	Transitive closure
[225]	Enterprise, PVM and PAMS	Graduate students	Login hours, and SLOC	Distributed	Graph theory related, sorting and tree searching
[135]-2002	CO_2P_3S, and non-CO_2P_3S Java	Undergraduate students	SLOC, number of classes, and number of choice points	Multi-core	Laplace and reaction-diffusion problems
[96]-2005	MPI and OpenMP	Graduate students	SLOC and development time per LOC	Cluster	Game of life and grid of resistors
[95]-2005	MPI and OpenMP	Graduate students	Development time, time understanding the problem, time designing a solution, time experimenting, time adding functionality, time parallelizing, time tuning, time debugging, time testing, and other specific activity	Multi-core and distributed	small programming problem
[142]-2005	OpenMP, OpenMP _XN and Pthreads	Undergraduate and graduate students	Development time	Multi-core	Benchmark applications
[257]-2005	MPI	Graduate students	SLOC, development time, relative effort (SLOC and development time), Kennedy's productivity (SLOC and development time), Kennedy's productivity modification	Distributed	Game of life and Buffon Laplace needle problem
[57]-2005	MPI, UPC and IBM PERCS x10	Undergraduate students	Time executing the application, cleaning time, parallelization time, debugging time, authoring time, and time accessing documentation	Distributed	Smith-Waterman algorithm
[58]-2006	-	Undergraduate and graduate students	Cards organization	-	Use the card sorting method to organize twenty-four parallel problems
[3]-2007	MPI+C	Undergraduate students	Development time, and number of MPI functions divided into basic functions, non-blocking functions, and collective functions	Cluster	Game of life
[229]-2008	Charm++, UPC, SCOOP, Star-P, and SHMEM	Groups of both novice and expert programmers	Development time (hrs), SLOC, and SLOC increase (%)	Distributed	N-body problem and Pi using a Monte Carlo method
[94]-2008	PRAM-like and MPI	Graduate students	Number of correctness programs and development time reported, instrumented and combined	Linux cluster and class server	Sparse matrix and dense vector multiplication
[47]-2008	IBM PPIs	Novices and experts developers	Development time, the number of steps to complete the activity, number of context splits, and the number of data items operated	Workstation	Use of an MPI function
[191]-2008	MPI and UPC	Graduate students	SLOC	Multi-core and Linux cluster	Power method algorithm
[240]-2009	Pthreads, Cell, and CUDA	Graduate students	Development time	Multi-core, STI Cell/B.E., and GPU	Optimizing the gridding kernel
[235]-2009	UPC and MPI	Undergraduate students	SLOC, development time, number of correct and incorrect implementations	Cluster	Minimum distance problem
[188]-2009	Pthreads and OpenMP	Graduate students	SLOC, SLOC with parallel constructs, and development time	Multi-core	Bzip2
[258]-2009	MPI	Undergraduate students	Time thinking/problem, time thinking/-solution, functionality tim, parallelizing tim, debugging time, testing time, tuning tim, and other time	Linux cluster	Small parallel programming problems
[211]-2010	Java Threads	Undergraduate students	Design time, development time, debug time, programming errors, and CCN	Multi-core	Implement sync-gallery Java program using coarse and fine-grain locks, monitors, and transactions.
[213]-2010	Java threads	Students and professionals	Number of errors	Multi-core	ArchivalList, IntList and StringBuffer

Table 3.5 continued from previous page

Work	PPIs	Participants	Metrics	Architecture	Activity
[32]-2011	Haskell	Undergraduate students	Logic and Compilation errors, hanging and non-hanging errors, development time, and SLOC	Multi-core	Program with synchronization and mutex
[189]-2012	Scala and Java Threads	Masters students	Development time, SLOC, NOC, functional styles (%), imperative style (%), and number of errors	Multi-core	Dining Philosophers, mergesort, and Parallel DRC Project
[173]-2013	Chapel, Cilk, Go, and TBB	Experts in each PPI tested	SLOC, and development time	Multi-core	Six micro-benchmark programs
[172]-2013	Chapel, Cilk, Go and TBB	Experts and non-expert developers	SLOC, development time, and correction time	Multi-core	Randmat, thresh, winnow, outer, product, and chain problems
[171]-2013	Java Threads and SCOOP	Graduate students	Time to complete the test, error types, number of classes, number of attributes, number of functions, and SLOC	Multi-core	Not informed
[169]-2013	MPI	Undergraduate students	SLOC, CCN, development time, and number of compilations	Linux cluster	Game of Life
[249]-2013	OpenCL, OpenACC, OpenMP, and OpenMP+LEO	Hackathons participants	Total SLOC, MSLOC, and development time (days)	Multi-core CPU and GPU	KegelSpan and Neuromagnetic Inverse problems
[187]-2014	Pthreads and Intel STM	Graduate students	SLOC, reading time, design time, dev. time, testing time, debugging time, number of parallel constructs, and number of critical sections	Multi-core	Parallel desktop search engine
[78]-2014	DSL-POPP and Pthreads	Graduate students	Development time, SLOC, and COCOMO II	Multi-core	Matrix multiplication
[38]-2015	CilkPlus and OpenMP	Master students	Number of correct programs, and development time	Multi-core	Program that finds anagrams
[160]-2015	Patty and Parallel Studio	Experienced developers	Clarity, complexity, perceivability, learnability, correctness, and development time	Multi-core	RayTracing benchmark
[247]-2015	OpenMP, OpenACC, and CUDA	Hackathons participants	Development time	Multi-core and GPU	Not informed
[128]-2016	OpeanACC and CUDA	Undergraduate and graduate students	Development time	GPU	Heat transfer and message encryption
[248]-2016	OpenACC and CUDA	hackathons participants	EffortLog metrics: Break time, thinking time, serial time, parallelizing time, testing time, debugging time, tuning time, experimenting time	Hybrid multi-core CPU and GPU	Not informed
[127]-2018	OpeanACC and CUDA	Undergraduate students	SLOC, and SLOC increase (%)	GPU	Heat transfer and message encryption
[159]-2018	OpenACC	Undergraduate students	Development time and COCOMO II	GPU	CG solver and Aeroacoustics simulation application ZFS
[158]-2019	MPI, OpenMP and OpenACC	Hackathons participants	Development time, process productivity, and training productivity	Multi-core and GPU	Two real-world training activities
[45]-2020	CUDA and Thrust	Graduate students	Development time, number of compiler errors, and number of successful results	GPU	Six programs
[190]-2021	OpenMP, CUDA, OpenCL, MPI, SkePU, StarPU, StarPU-MPI, EXA2PRO	Senior engineer	Training time, Development time, and SLOC	CPU and GPU	CO2 capture , Metalwalls , Brain modeling, ODE solver
[104]-2022	Taskflow, oneTBB, StarPU, HPX, and OpenMP	Ph.D. level	SLOC, TOC, CCN for a single function, CCN for the whole program, development time, and percentage of bugs	CPU and GPU	Micro-benchmarks and two real-world applications

Programming (DSL-POPP) in comparison to the Pthreads library. In [160], a new pattern-based process model called Patty was introduced. Patty was compared with Intel Parallel Studio in an experiment with experienced developers to evaluate its effectiveness and productivity. Papadopoulos *et al.* [190] proposed EXA2PRO, which is a framework to improve developers' productivity for parallelizing applications in several environments, such as GPU and FPGA. In order to assess the productivity of the EXA2PRO, Papadopoulos *et al.*

[190] compared it with other models such as OpenMP, CUDA, OpenCL, MPI, SkePU, StarPU, and StarPU+MPI. In addition, Huang *et al.* [104] performed a quasi-experiment with five Ph.D. level C++ programmers to compare the expressiveness and programmability of a set of task graph computing systems: Taskflow, oneTBB, StarPU, HPX, and OpenMP.

In [47], the authors proposed a methodology for measuring the complexity of programming-related tasks in HPC. Complexity Metrics (CM) was proposed to help determine the productivity impact of new PPIs developed by IBM through a series of real-world observations with HPC experts and experiments with novice programmers. Given a task that the programmer must perform, [47] defines productivity as the performance to complete that task. Although the CM method has shown promise, it does not include development time, which is a key productivity measure. Differently from the other studies, Eccles and Stacey [58] performed an experiment using the open card sorting method. In this experiment, the participants were required to sort 24 index cards with typical parallel problems. The participants organized the problems into categories and ranked them according to the difficulty of their solving.

Parallel programming evaluation only using coding metrics

Some works claimed to do usability evaluation of PPIs without actually performing software experimentation with people. Table 3.5 presents a summary of these works, which contain the PPIs evaluated, metrics used, target architecture, and the applications evaluated. In addition, these studies are sorted by publication year. As seen in this table, most of these works evaluate the usability of PPIs for multi-core environments. Okur *et al.* [181] analyzed the productivity of Microsoft's parallel libraries Task Parallel Library (TPL) and Parallel Language Integrated Query (PLINQ) in comparison with C++ threads and .NET in the development of large-scale multi-core applications. In [8], Pthreads productivity was compared with OpenMP Superscalar (OmpSs) that combines advanced features, such as automated runtime dependency resolution, while maintaining simple programming using pragmas. Danelutto *et al.* [46] evaluate the effort required by the number of SLOC to develop parallel applications using FastFlow, TBB, OpenMP, and OmpSs. In [157], Pthreads, OpenMP, Cilk Plus, TBB, SWARM, and FastFlow were compared in developing a series of numerical linear algebra problems.

Griebler *et al.* [79] also proposed a new PPI for exploiting stream parallelism in multi-core environments called SPar, which supports parallel code generation with the FastFlow runtime. In order to evaluate the productivity of SPar, Griebler *et al.* [79] compared it to FastFlow in terms of the SLOC. Other studies has been done to evaluate the productivity of SPar against other parallel programming models. In [82], the main goal was to asses the programmability and performance of SPar, FastFlow, TBB, and Pthreads considering a set of real-world streaming applications, of which only the versions with the best performance were chosen to perform the productivity evaluations. In [98], a new version of SPar that supports parallel code generation with both FastFlow and TBB runtimes was compared with FastFlow, TBB, and Pthreads. In addition, Hoffmann *et al.* [99] implemented a new compiler algorithm in SPar for automatically generating parallel code in the OpenMP runtime using source-to-source code transformations. This modification was compared with the other SPar runtime versions (FastFlow and TBB), TBB, FastFlow, OpenMP, and Pthreads.

Huang *et al.* [105] introduced Cpp-Taskflow, which is a gereral-purpose parallel task programming system to facilitate the construction of large and complex parallel applications using task dependency graphs. In order to evaluate the usability of Cpp-Taskflow, Huang *et al.* [105] compared it with OpenMP and TBB. Huang *et al.* [103] also developed a new version of OpenTimer, which is a high-performance timing analysis tool for Very large scale integration (VLSI) systems. The goal of Huang *et al.* [103] was to evaluate the usability of the two versions of OpenTimer, where OpenMP tasking clauses were adopted to implement the pipeline-based incremental timing in the first version and Cpp-Taskflow to implement the task-based incremental timing in the second version. In [130], the productivity of a proposed new version of the Cpp-Taskflow was compared to the TBB.

The effort required to develop a cryptographic algorithm based on the Brahmagupta-Bhaskara equation using OpenMP was evaluated in [174]. In order to evaluate the effort into learning and using a new programming language, Gmys *et al.* [73] compared the performance, scalability and productivity of Chapel, Julia, Python and C with OpenMP. Schmitz *et al.* [221] proposed Parallel Pattern Language (PPL), which closely follows mathematical notations. Schmitz *et al.* [221] evaluated the usability of PPL against OpenMP. In [148], a new framework for developing parallel simulations called Pyne was compared to PETSc in order to evaluate its productivity.

Legaux *et al.* [125] designed and developed Orléans Skeleton Library (OSL), a new C++ algorithmic skeleton library. Aim to evaluate the productivity of OSL, Legaux *et al.* [125] used Halstead's measures. In 2016, del Rio Astorga *et al.* [50] proposed a Generic Reusable Parallel Pattern Interface (GrPPI) for stream-based C++ applications. GrPPI allows users to easily explore parallelism in sequential applications using existing frameworks (e.g., threads C++, OpenMP and TBB) thanks to its high-level C++ PPI. Aiming to evaluate the usability of GrPPI, the authors evaluated the increase in lines of code compared to the C++ threads, OpenMP and TBB versions [50]. Guo and Agrawal [85] evaluated the programmability of MapReduce interface against two proposed variations called MR-like and Reduction Object. Adornes *et al.* [1] proposed a unified MapReduce domain-specific language to reduce the programming effort and improve code reuse between multi-core and distributed environments. Adornes *et al.* [1] compared their solution to Phoenix++ and Hadoop MapReduce. In addition, Hadoop productivity was compared to

Parallel Social Data Analytics (ParSoDA), a high-level library to reduce the programming skills needed for implementing scalable social data mining applications [21].

Pankratius [186] assessed the productivity of Java threads to develop multi-core applications. Kepner [117] also evaluated Java threads against OpenMP, MPI, and High Performance Fortran (HPF) to compare parallel application development for multi-core and distributed environments. In addition, other studies also aim to compare the usability and productivity of PPIs for multi-core environments against distributed environments. In [220], OpenMP and MPI applications were evaluated to show the applicability and functionality of a proposed tool called Pattern Instrumentation Tool (PInT). This tool automatically calculates pre-implemented metrics (e.g. SLOC, CCN, and the number of pattern occurrences) from instrumented source code. In [164], MPI productivity was compared with a new framework called Trasgo, which was used to generate automatically MPI applications from abstract data-parallel expressions.

There are several other studies aimed to evaluate MPI productivity and compared it to other PPIs. Hochstein *et al.* [97] evaluated the use and productivity of MPI in a project of Flash Center at the University of Chicago. In 2004, UPC was analyzed and compared to MPI in parallelizing applications for distributed environment [31]. In [14], a domain-specific language proposed to automate the generation and insertion of code required for parallelization into an existing sequential application (High-Level Parallelization Language (Hi-PaL)) is compared to MPI. Baek *et al.* [17] evaluated the productivity of Cilk and MPI. The productivity of Collective Asynchronous Remote Invocation (CARI) a collective variant of the paradigm to distributed systems Remote Method Invocation (RMI) was compared to MPI by Ahmad *et al.* [2]. In [86], a version of MPI with Fortran, MPI together OpenMP, and basic OpenMP were compared in terms of SLOC to evaluated the produtivity. In [145], the productivity of the parallelization of divide-and-conquer algorithms using two libraries developed for this purpose (parallel_recursion and parallel_stack_recursion), OpenMP and Cilk. An updated version of these libraries (dparallel_recursion and dparallel_stack_recursion) was also compared with MPI and MPI+OpenMP [146].

Some new approaches have been proposed to exploit parallelism in distributed systems. Tejedor *et al.* [236] introduced Cluster Superscalar (ClusterSs), a runtime for Java applications based on automatic function-level parallelism designed to execute on clusters of symmetric multiprocessors. Cid-Fuentes *et al.* [37] introduced PyCOMPSs to propose a solution for distributed big data processing in HPC infrastructures, which is a task-based programming model for Python. O'Donncha *et al.* [184] proposed AllScale, which is a toolchain that aims to simplify the development of highly scalable parallel applications by dividing development responsibilities into silos. Löff *et al.* [131] proposed DSParLib, which is a high-level parallel programming abstraction to exploit stream parallelism in distributed environments. In addition, Cid-Fuentes *et al.* [37], O'Donncha *et al.* [184], and Löff *et al.* [131] compared their solutions with MPI to evaluated the productivity.

Some researchers also compared MPI productivity to PPIs for GPU programming. In [65], an experimental study was conducted to compare the MPI and Hitmap libraries for the implementation of a GPU-based algorithm to solve the hyperspectral image registration. In [84], MPI, OpenSHMEM, Charm++, and Legion were compared regarding their productivity, performance, scalability, and load-balancing capacity on homogeneous clusters with GPUs. Nakao *et al.* [168] performed a study to evaluate a rective-based language for accelerated clusters (XACC) compared to CUDA with MPI and OpenACC with MPI in clusters with GPUs.

Several other studies have aimed to evaluate the productivity of interfaces to GPU systems. Herdman *et al.* [92] evaluated the productivity of OpenACC, OpenCL, and CUDA. Malik *et al.* [139] evaluated the productivity of CUDA, OpenCL, PGI, and Matlab. Wienke *et al.* [250] compared OpenCL, PGI Acc, and OpenACC to develop GPU applications. In [177], the effort required to used OpenACC for developing an aeroacoustics simulation framework was evaluated. Memeti *et al.* [155] assessed the productivity of OpenCL, OpenACC, OpenMP, and CUDA to GPU programming. CUDA was compared to OpenCL C++ and Python versions to develop a Mandelbrot application by Holm et al [101]. In [165], OpenACC was compared to a new approach called Hybrid Fortran, which enables a reduction in the code changes required to achieve parallelization in hybrid architectures. In addition, Marantos *et al.* [143] evaluated CUDA productivity using a proposed estimation metric based on effort estimated by Halstead's metric to develop a sequential CPU application to predict the percentage of extra effort to develop GPU code.

GPU PPIs have also been compared to multi-core CPU PPIs to evaluate productivity. In addition, Lee and Vetter [124] evaluated the increase in SLOC of the Rondina benchmarks developed with OpenMP compared to the versions developed with PGI Accelerator, OpenACC, HMPP, OpenMPC, and R-Stream. In [44], the authors' goal was to compare the productivity of the C++ heterogeneous programming paradigms C++ AMP and OpenACC with OpenMP for multi-core CPU and OpenCL for GPU systems. Wei *et al.* [246] compared multi-core OpenMP applications with CUDA and OpenACC applications concerning SLOC. In [144], the productivity of a set of PPIs for CPU, GPU, or hybrid (both CPU and GPU) systems were compared: OpenMP 3.0, OpenMP 4.0 (support to GPU), OpenACC, RAJA, Kokkos (lambdas, functors and nested), CUDA (only GPU), and OpenCL. Sakdhnagool *et al.* [214] compared the productivity of PPIs running on systems with multi-core CPUs (OpenCL and LE-OpenMP) versus systems with GPU (OpenCL, CUDA, and OpenMP). In [231], CUDA was compared with OpenACC versions for multi-core and GPU systems concerning increasing the number of SLOC. Di *et al.* [51] evaluated the productivity of parallelizing C benchmark applications using CUDA and OpenACC and Python benchmark applications using Numba. Pennycook *et al.* [197] evaluated the productivity of Kokkos, mxhMD, OpenMP 4.5, and OpenMP 5.0.

Table 3.5: Studies evaluation of PPIs productivity without considering the human factor.

Work	PPIs	Metrics	Architecture	Application
[31]-2004	UPC and MPI	SLOC, NOC, and CCN	Distributed	NAS Benchmark (C and Fortran)
[17]-2004	Cilk and MPI	SLOC	Heterogeneous Workstations	Fibonacci, Travelling Salesman, Matrix Multiplication, N-Queens, Laplace problems
[117]-2004	MPI, OpenMP, HPF, and Java Threads	SLOC	Multi-core and Cluster	NAS parallel benchmarks
[97]-2008	MPI	SLOC, number of commits, number of MPI function calls, median developer activity	Distributed	FLASH code
[36]-2010	OpenCL, XLC, and CUDA	Kernel SLOC, and host SLOC	GPU and Cell Broadband Engine	Vector application
[2]-2011	MPI and CARI	SLOC	Cluster	Gadget-2 code
[186]-2011	Java Threads	Parallel constructs	Multi-core	40 Java projects
[181]-2012	C++ threads, .NET 4.0, TPL and PLINQ	SLOC, SLOC increase (%), number of parallel directives, number of parallel patterns, number of synchronization directives, and number of constructs misused	Multi-core	655 open-source applications found in GitHub
[14]-2012	MPI and Hi-PaL	SLOC	Cluster	Circuit satisfiability, game of life, Poisson solver, image processing, and prime numbers
[236]-2012	Java ClusterSs and X10	SLOC	Cluster	Matmul, Sparse LU, and K-means
[124]-2012	OpenMP, PGI, OpenACC, HMPP, OpenMPC, and R-Stream	SLOC Increase (%)	Multi-core (OpenMP) and GPU	Thirteen OpenMP programs
[92]-2012	OpenACC, OpenCL and CUDA	Number of words of code (WOC) (Total, device and host)	GPU	Hydrocode benchmark
[8]-2012	OmpSs and Pthreads	Increase of SLOC	Multi-core	K-type benchmarks
[139]-2012	CUDA, OpenCL, PGI, Matlab	SLOC, NOC, number of PPIs functions, number of PPIs parameters, and number of PPIs keywords	GPU	Four of the five NAS Parallel Benchmark kernels: EP, CG, FT, and MG
[250]-2012	OpenCL, PGI Acc, and OpenACC	MSLOC	GPU	Simulation of Bevel Gear Cutting, and Neuromagnetic Inverse Problem
[125]-2014	OSL	Halstead distinct and total operators, distinct and total operands, length, vocabulary, difficulty, and effort	Multi-core	Low-level and high-level BSP implementations
[44]-2015	OpenMP, OpenCL, C++ AMP, and OpenACC	SLOC, and Kennedy's productivity	Multi-core CPU, GPU, and heteregeneous system	Five proxy applications: read-benchmark, LULESH, CoMD, XSBench, and miniFE
[1]-2015	Phoenix++, Hadoop, and Unified domain-specific language	SLOC, and Basic COCOMO model	Multi-core and distributed	Word Count, Word Length, Histogram, K-means and Linear Regression applications
[79]-2015	SPar, and FastFlow	SLOC	Multi-core	Sobel and prime number applications
[50]-2016	GrPPI, C++ Threads, OpenMP and TBB	SLOC increase (%)	Multi-core	OpenCV video stream processing application with two filters: Gaussian Blur and Sobel
[157]-2016	Pthreads, OpenMP, Cilk Plus, TBB, SWARM, and FastFlow	SLOC and development time	Multi-core	Nine benchmark applications
[177]-2016	OpenACC	SLOC, ASLOC, COCOMO II, and COCOMO II reuse model	Multi-core and GPU	ZF aeroacoustic simulation software kernels
[246]-2016	OpenACC, CUDA, and OpenMP	SLOC	Multi-core and Hybrid (multi-core and GPU)	GTC-P application
[86]-2016	MPI (Fortran), OpenMP, and MPI+OpenMP	SLOC	Hybrid cluster	NAS BT benchmark
[155]-2017	OpenCL, OpenACC, OpenMP, and CUDA	SLOC, SLOC with parallel directives, and SLOC increase (%)	GPU	Rodinia benchmark suites
[144]-2017	Kokkos, RAJA, OpenACC, OpenMP 3.0, OpenMP 4.0, CUDA, and OpenCL	SLOC	Multi-core CPU and GPU	Tealeaf miniature proxy application
[82]-2018	Pthreads, TBB, FastFlow and SPar	SLOC and CCN	Multi-core	Dedup, Ferret and Bzip2

Table 3.6 continued from previous page

Work	PPIs	Metrics	Architecture	Activity
[220]-2018	OpenMP and MPI	SLOC, Fan-In, Fan-Out, CCN, and number of pattern occurrences collected from PInT	Multi-core and distributed	Breadth-first Search (OpenMP) and LAMMPS (MPI)
[192]-2018	PHAST, CUDA, OpenCL, Kokkos, and SYCL	SLOC, Halstead's development effort, and CCN	GPU	Triad, DCT8x8, Black-scholes, and N-Body
[85]-2018	MapReduce, and MR-like, and Reduction Object	SLOC	Multi-core	K-means, Support vector machine algorithm (SVM), Linear Regression, and Logistic Regression
[165]-2018	Hybrid Fortran and OpenACC	ASLOC	GPU	weather prediction model
[197]-2018	Kokkos, mxhMD, OpenMP 4.5, OpenMP 5.0	SLOC	Heteregeneous (multi-core CPU and GPU)	miniMD benchmark from the Mantevo suite
[159]-2018	OpenMP, CUDA, and MPI	SLOC	Multi-core and GPU	Boltzmann Transport Equation
[164]-2019	MPI and Trasgo	SLOC, Halstead's development effort, and CCN	Hybrid cluster and multi-core	Ronate, Cannon, and NAS MG benchmark
[214]-2019	OpenCL, LE-OpenMP, OpenMP, and CUDA	SLOC	Multicore and GPU	seven Rondina Benckmarks applications
[46]-2019	FastFlow, TBB, OpenMP, and OmpSs	SLOC, and MSLOC	Multi-core	Parsec benchmark
[254]-2019	Ruler and OpenCL	SLOC	Multi-core and GPU	Seven micro-benchmark programs
[65]-2019	MPI and Hitmap	SLOC, TOC, CCN, and Halstead's development time	GPU	Hyperspectral image registration problem
[231]-2019	CUDA and OpenACC	Execution time, speedup, SLOC, SLOC increase (%)	multi-core CPU (OpenACC) and GPU (OpenACC and CUDA)	Multiple Flow Direction (MFD) algorithm
[130]-2019	Cpp-Taskflow v2 and TBB	SLOC, Halstead's development effort, and CCN	Multi-core	Parallel machine learning hyperparameter search and a Very-large-scale integration (VLSI) circuit timing analysis.
[168]-2019	XACC, CUDA+MPI, and OpenACC+MPI	SLOC with parallel directives, total SLOC, ASLOC, MSLOC, and delete SLOC	Cluster with GPU	LQCD mini-application
[148]-2019	Pyne and PETSc	SLOC	Multi-core	Fluid simulation application
[196]-2019	PHAST and CUDA	SLOC, Halstead's development effort, and CCN	Multi-core CPU and GPU	Cache-timing-attack resistant AES-based pseudo random number generator application
[193]-2019	PHAST and SYCL	SLOC, Halstead's development effort, and CCN	Heterogeneous (multi-core and GPU)	Histogram-stretch application
[194]-2019	PHAST and SYCL	SLOC, Halstead's development effort, and CCN	Heterogeneous (multi-core and GPU)	Histogram-stretch application
[84]-2019	MPI, OpenSHMEM, Charm++, and Legion	SLOC	Cluster with GPUs	Five open-source application
[129]-2019	HPSM, OpenMP, and StarPU	SLOC, and SLOC that will be executed at runtime (it ignores comments, blank lines, data declarations and headers)	Multi-core (OpenMP) and heterogeneous (multi-core and GPU)	AXPY program
[21]-2019	ParSoDA library and Hadoop	SLOC	Cluster	Parallel social data analysis applications based on the ParSoDA library
[37]-2020	PyCOMPSs and MPI	SLOC, CCN and NPath	Multi-core Cluster	Cascade SVM and K-means
[73]-2020	Chapel, Julia, Python and OpenMP	SLOC, Kennedy's productivity, and Utility model	Multi-core	3D Quadratic Assigment Program
[184]-2020	AllScale, and MPI	SLOC, Halstead's development effort, average CCN across all modules, and the sum total CCN	Cluster	AMDADOS and iPIC3D
[238]-2020	Fleet and CUDA	SLOC	FPGA and GPU	Six micro-benchmark programs
[103]-2020	OpenMP and Cpp-Taskflow	SLOC and COCOMO 81 organic model	Multi-core	OpenTimer v1 and v2
[105]-2020	Cpp-Taskflow, OpenMP, and TBB	SLOC, CCN, maximum CCN, COCOMO development effort, time and cost ($)	Multi-core	wavefront computing and graph traversal
[178]-2020	OpenCL and EngineCL	SLOC, CCN, C++ TOC, number of structs/classes and methods used, and number of sections with error checking	Heterogeneous (multi-core and GPU)	Five benchmarks

Table 3.6 continued from previous page

Work	PPIs	Metrics	Architecture	Activity
[98]-2020	SPar, TBB, FastFlow, and Pthreads	SLOC	Multi-core	Bzip2, Ferret, Lane Detection, and Person Recognition
[101]-2020	CUDA and OpenCL for C++ and Python	SLOC and subject development time (medium or fast)	GPU	Mandelbrot
[145]-2021	parallel_recursion, parallel_stack_recursion, OpenMP and Cilk	Multi-core cluster	SLOC and Halstead's development time	T2XL, N Queens, fib, floorplan, and knapsack benchmarks
[221]-2021	OpenMP and PPL	SLOC	Multi-core	19 benchmarks of the Rodinia OpenMP benchmark suite
[209]-2021	Controller model, Intel oneAPI, and OpenCL	SLOC, TOC, CCN, and Halstead's development effort	Heterogeneous system with multi-core CPU and FPGA	Hotspot, Matrix Pow, and Sobel Filter
[195]-2022	PHAST, PHAST SA (single architecture), and SYCL	SLOC, Halstead's Mental Discriminations, CCN, Fan-in, and Fan-out	multi-core CPU and GPU	Six benchmarks
[174]-2022	OpenMP	basic COCOMO (effort, time, and team size)	Multi-core CPU	Brahmagupta-Bhaskara Equation Based Algorithm
[146]-2022	MPI, MPI+OpenMP, dparallel_recursion, and dparallel_stack_recursion,	SLOC, Halstead's development effort, and CCN	Multi-core cluster	uts, N Queens, fib and topsorts benchmarks
[131]-2022	MPI (on-demand and round-robin) and DSParLib	SLOC	Cluster	Mandelbrot, prime numbers, face recognition, lane detection, and PBzip2
[143]-2022	CUDA	Number of hotspots, SLOC, number of hotspots' SLOC, number of hotspots' statements, CPU distint and total operations, CPU CCN, and Halstead's measures of CPU version: volume, length, and difficulty	Multi-core CPU and GPU	Hotspots from Polybench and Rodinia benchmark suites
[74]-2022	Verilog, OpenCL, and oneAPI	Kernel SLOC, host SLOC, total SLOC, and COCOMO development time	FPGA	Sobel filter program
[51]-2022	CUDA and OpenACC for C, and Numba for Python	SLOC for operations type	Multi-core CPU and GPU	NPB kernels
[195]-2022	PHAST, PHAST SA (single architecture), and SYCL	SLOC, Halstead's development effort, and CCN	multi-core CPU and GPU	Several benchmarks
[99]-2022	SPar, TBB, FastFlow, OpenMP, and Pthreads	SLOC	Multi-core	Bzip2, Ferret, Lane Detection, and Person Recognition
[147]-2022	HDNN, DeepDSL, and oneAP	SLOC	Multi-core CPU, GPU, and TPU service in Google Cloud Platform	Softmax layer

Some new interfaces have been proposed to exploit parallelism on heterogeneous architectures. In [254], Ruler was presented, which is a core language that eases the conflict between productivity and performance with quantitative parallelism expression. To evaluate the productivity of Ruler, it was compared to OpenCL regarding the number of SLOC [254]. In [129], HPSM was introduced, a high-level C++ framework to enable the execution of parallel loops and reductions simultaneously over multi-core CPUs and GPUs. To evaluate HPSM, Lima and Domenico compared its productivity with OpenMP for multi-core CPU systems and StarPU for heterogeneous systems. Nozal *et al.* [178] present EngineCL, a new OpenCL-based C++ PPI for heterogeneous systems. To evaluate the productivity of their solution, Nozal *et al.* [178] compared it to OpenCL.

Peccerillo and Bartolini [192] proposed a framework for multi-core CPU and GPU systems called PHAST library. Peccerillo and Bartolini [192] evaluated the productivity and complexity of their approach compared to CUDA, OpenCL, Kokkos, and SYCL. In [196], PHAST was compared to CUDA in developing a cache-timing-attack-resistant AES-based pseudo-random number generator, which is a challenging real-world application with several solutions available. PHAST receives a series of updates, which enables task-agnostic fashion coding [193], is automatically optimized for a specific platform [194], and allows the choice of the architecture at runtime [195]. The effort required to use these new features was evaluated against the old PHAST version and SYCL.

Finally, the productivity of PPIs for Field-Programmable Gate Array (FPGA) systems has been less explored in the literature. Thomas *et al.* [238] proposed a framework for massively parallel streaming on FPGAs called Fleet, whose productivity in developing six micro-benchmarks was compared with the sequential CPU and CUDA GPU version. In [209], a new Controller model for FPGAs was presented based on OpenCL for GPUs. The effort of developing the controller model was compared to the effort of developing applications with Intel oneAPI and the reference codes programmed directly with OpenCL [209]. Gondhalekar *et al.* [74] evaluated the effort required to develop the Sobel filter program using Verilog, OpenCL, and Intel oneAPI for FPGA systems. In addition, only one study evaluates development productivity in environments with Tensor Processing Unit (TPU) accelerators. Martinez *et al.* [147] proposed a domain-specific language to exploit parallelism in CPU, GPU, and TPU systems called Heterogeneous Deep Neural Network (HDNN). The productivity of HDNN was compared to a DSL for deep learning (DeepDSL) and oneAPI interfaces.

3.3 Final remarks

This chapter aimed to determine how productivity and usability have been evaluated in parallel programming. For this purpose, we conducted a review of the literature regarding the usability and productivity evaluation of PPIs. From this literature review, we identified that there are studies that claim to perform a usability evaluation, but do not actually do so. Since only a few studies consider the opinion of the participants in their evaluations. In addition, the vast majority of studies do not conduct experiments with people. Although some of these studies mention usability, in fact they only evaluate productivity by considering metrics such as SLOC, CCN, COCOMO, and Halstead's measures. These classical metrics can rapidly provide some programming indicators based on code size, complexity assessment, and development effort estimation. However, they are designed to evaluate general-purpose software and do not consider factors that impact the parallel programming effort. As these metrics are widely used in the literature, verifying their effectiveness when evaluating parallel applications is necessary.

We identified only two effort estimation model in the context of parallel programming. Wienke *et al.* [248] proposed a COCOMO II extension to parallel programming, considering factors that affect the development cycle of such applications. However, this is a conceptual model and it has not been validated in practice, so its accuracy cannot be confirmed. This prevent a fair comparison with the original COCOMO II model. On the other hand, Marantos [143] proposed a model to estimate the percentage of extra effort to parallelize an existing sequential CPU application with CUDA and the energy efficiency. This model performs its estimations using different machine learning algorithms based on Halstead, CCN, and SLOC measures of the target sequential application. Marantos [143] designed a dataset for training and testing its models. However, there needs to be documentation regarding this data, making it difficult to replicate the study results.

We have also identified some productivity metrics specific to parallel applications, such as Kennedy's productivity [73, 257], Utility model [73], Progress productivity [158], and Training productivity [158]. These metrics have proven helpful in comparing parallel interfaces concerning increased productivity. However, these metrics perform evaluations based on programming effort and other factors such as speedup, efficiency, and the number of threads. Usually, the programming effort is measured by SLOC or development time. Therefore, to use such metrics, it is necessary to know the development effort of the evaluated applications.

Regarding the studies conducted with people, we observe several challenges to be addressed in the future. We observed that few studies [32, 45, 78, 96, 94, 128, 127, 171, 189, 191, 211, 229, 247] considered the practices recommended by Wohlin *et al.* [252] when conducting software experiments, such as developing an experimentation plan, and conducting hypothesis testing. Furthermore, we observe that there are some studies evaluating the productivity of developing parallel stream processing applications [46, 50, 79, 82, 98, 99, 131], none of them conduct experiments with people for this purpose. On the other hand, Huang *et al.* [104] performed a quasi-experiment with five participants to evaluate the expressiveness and programmability Tasflow, oneTBB, StarPU, HPX, and OpenMP in developing task-graph applications.

We also observed that few studies aim to analyze the productivity of PPIs targeting FPGA architectures [238, 209, 74], and none of them have conducted experiments with students. While there are a series of studies that evaluate the productivity of developing parallel applications for GPU and heterogeneous environments, few perform experiments on people for this purpose [240, 249, 247, 128, 248, 127, 159, 158, 45, 190]. Only one study has been conducted to evaluate the productivity of PPIs for systems with cloud TPU, which is an accelerators developers by Google to machine learning. Moreover, there is a gap in evaluating emerging PPIs for HPC clusters, such as HPX and Apache interfaces for big data processing (e.g., Spark and Flink). Since few studies explore such architectures, there is also room for creating methodologies to assist parallel programming usability experimentation.

4. ASSESSMENT OF PARALLEL PROGRAMMING PRODUCTIVITY

From usability assessments, it is possible to create better and simpler-to-use PPIs and manage their quality [52]. Usability is defined by ISO 9241-11 [108] as the extent to which specific users can use a system, product, or service to achieve specific goals with effectiveness, efficiency, and satisfaction in a specific context of use. In parallel programming literature, we observed that most authors claim to perform usability evaluations of PPIs without considering users in their evaluations. Furthermore, among the studies with users, most claim to conduct usability assessments of PPIs without considering all the recommended practices (e.g., develop an experiment plan, perform a hypotheses test, and evaluate the validity) [3, 38, 47, 57, 58, 95, 135, 142, 159, 158, 160, 169, 173, 172, 188, 187, 190, 213, 225, 233, 235, 240, 249, 248, 257, 258]. Only few works consider it [32, 45, 78, 96, 94, 128, 127, 171, 189, 191, 211, 229, 247]. Among these papers, only a few works considered programmers' satisfaction when evaluating usability [38, 173, 160, 171, 172, 188, 189, 213, 225, 233, 235].

In this context, the research question we aim to answer in this chapter is: "*How to improve usability evaluation of PPIs?*". The limitations found in the literature led us to present a methodology to help other researchers evaluate usability in parallel programming based on the experimentation practices recommended by Wohlin *et al.* [252]. According to Wohlin *et al.* [252], one of the advantages of conducting a planned experiment is the control of subjects, objects, and instrumentation, ensuring that more general conclusions can be drawn. In addition, a further advantage is the ability to perform statistical analysis using hypothesis testing methods and opportunities for replication of the experiment. To ensure that the experiment is performed correctly, a process is required to prepare and conduct the experiment and analyze its results [252]. In addition, according to ISO 9241-11 [108], our methodology proposes the evaluation of usability based on three factors: efficiency, effectiveness, and user satisfaction. Finally, we aim to provide this methodology to guide other parallel programming researchers in usability analysis, which is presented in Section 4.1.

In this chapter, experiments were also conducted with students to demonstrate the use of the methodology presented. Section 4.2 presents an experiment to assess the usability of three PPIs based on structured parallel programming for expressing parallelism in stream processing applications targeting multi-core systems. From the literature review, no studies were identified aiming to evaluate the usability of structured PPIs for parallel stream processing. Moreover, Section 4.3 presents an initial effort to evaluate the usability of a new parallel programming model for GPU against classical models. Unlike studies that perform experiments with experienced developers on parallel programming [172, 173], participants of these studies were intentionally beginner developers to understand the challenges faced by them.

This chapter is an abridged version of the results presented in the paper [11][1], and has been reproduced here in accordance with the signed copyright agreement and the copyright holder. The results of the study with GPU PPIs (Section 4.3) have been added in this chapter to complement the results in [11]. In addition, the studies were approved by the Pontifical Catholic University of Rio Grande do Sul (PUCRS) research ethics committee with CAAE number 52635421.3.0000.5336.

4.1 Research Methodology

To ensure that the experiment is performed correctly, a process is needed to provide steps to support the activity execution. A process can be used as a checklist and guideline on what to do and how to do it [252]. Figure 4.1 presents the methodology proposed to evaluate the usability of PPIs. The first step is called the planning phase. In this phase, the problem to be solved and the experiment plan must be defined. The target problem can be defined through a gap found in a literature review. In addition, the goals, context, hypothesis, procedure, study activity, and instruments must also be defined in the planning phase.

Still, in the planning phase, the study goals are formulated from the problem to be solved and should reflect the purpose of the experiment. The context of the experiment is also defined in planning phase, which can be characterized according to four dimensions as [252]:

- **Offline versus online:** Defines the environment of the experiment. Whether the experiment is being conducted in a laboratory under controlled conditions or is a real-world project;
- **Student versus professional:** Defines the degree of knowledge of the experiment participant;
- **Classroom (or toy) versus real problems:** Defines the size of the problem being evaluated in the experiment;
- **Specific versus general problem:** Defines whether the results of the experiment will be valid in the general context of the area studied or in a specific context.

The next step in the planning phase is the hypothesis formulation. Usually, the definition of the experiment is formalized into two hypotheses:

- **Null hypothesis (H_0):** there is no statistically significant relationship between cause and effect;

[1]A Parallel Programming Assessment for Stream Processing Applications on Multi-core Systems, Computer Standards & Interfaces - ®2022 Elsevier

- **Alternative hypothesis (H_1):** there is statistically significant relationship between cause and effect.

H_0 asserts that there are no real underlying trends or patterns in the experiment environment, and the differences between the observations are coincidental [252]. For example, suppose an experiment is conducted to compare a new failure inspection method with the old method. When defining the hypotheses, the H_0 defines that a new inspection method finds, on average, the same number of failures as the old one ($H_1 : \mu_{N_{old}} = \mu_{N_{new}}$). On the other hand, the H_1 defines that a new inspection method finds, on average, more faults

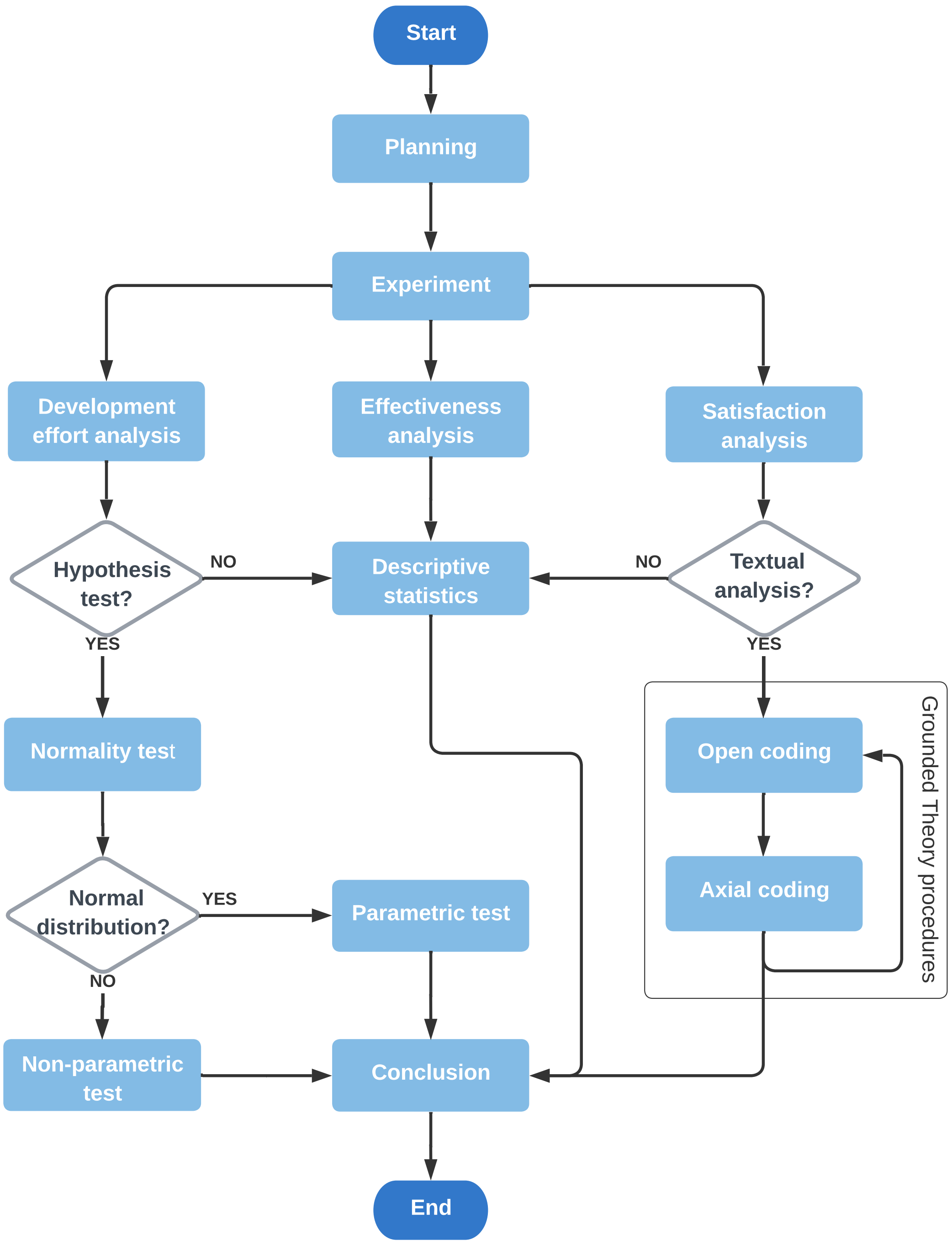


Figure 4.1: Research method flowchart.

than the old one ($H_0 : \mu_{N_{old}} < \mu_{N_{new}}$) [252]. Therefore, the goal of an experiment is to reject H_0 in favor of the H_1.

After formulating the hypotheses, the study activity and the procedure are also defined in the planning phase. The procedure presents all the rules and steps to be followed by the participants when performing the study activity. For example, whether participants will be able to access the Internet during the activity or whether there will be materials available for access, how much time they will be given to complete the activity, and other considerations to ensure there is no bias.

In the planning phase, the experimenters also choose and develop the instruments for the study, which can be objects, guidelines, or measuring instruments [252]. The experience objects can be, for example, specification documents or code. Guidelines are used to guide participants and may include, for example, procedure description and checklists. Measuring instruments are used for data collection, which can be done through forms, interviews, and instrumentation of the machines used by the participants, such as scripts to record the participants' screens and capture log files [252]. Also, before conducting the experiment, a pilot study is performed to test the experiment plan.

After the planning phase, the experiment is conducted, and the results achieved are analyzed. Effectiveness, efficiency, and user satisfaction must be evaluated to determine the usability of a PPI [108]. Effectiveness should assess the accuracy and completeness with which participants achieve the goals specified in the study [108]. Descriptive statistics used to organize and summarize a data set [100] can be applied to evaluate effectiveness. For example, using the percentage, it is possible to express the proportion of participants who achieved the study goal concerning the total number of participants.

Efficiency or productivity can be evaluated through the effort spent to develop an application [108]. Descriptive statistics or hypothesis testing can be applied to evaluate the development effort results. Although descriptive statistics help to organize and summarize the evaluated data set, just the average time spent by the participants to develop a parallel application may not be enough to determine which PPIs provide the best productivity. Then, through a hypothesis test, it is possible to determine if there is a significant difference between the average times needed to develop the applications with each of the PPIs evaluated. If a hypothesis test is necessary, a normality test should first be performed to verify whether the sample has a normal distribution. A parametric test is applied if the sample has a normal distribution. Otherwise, the non-parametric test is applied [34, 224].

Participants' satisfaction can be evaluated both qualitatively and quantitatively using forms [19]. For qualitative data, a textual analysis can be performed using a subset of the procedures for coding from Grounded Theory (GT) [42], which is a specific methodology developed with the objective of building theory from data [49]. However, this study did not fully explore the GT methodology, since we were not interested in generating theories. Since this metric has been popularly used in software engineering for qualitative analysis [223], we aimed to use it to evaluate the qualitative perception of the participants over the activity performed [49]. Therefore, our methodology approached only a textual analysis based on the open and axial coding steps of GT methodology (Figure 4.2). Open coding is an interpretive process by which data is analytically divided. At this step, responses are analyzed accurately, and relevant events, actions, or interactions are compared to identify similarities and differences. Those quotes that are conceptually similar are grouped and receive conceptual labels (code) and form categories and subcategories, as illustrated in Figure 4.2. Figure 4.2 also illustrates the axial coding, where the categories are related to their subcategories (codes) [42].

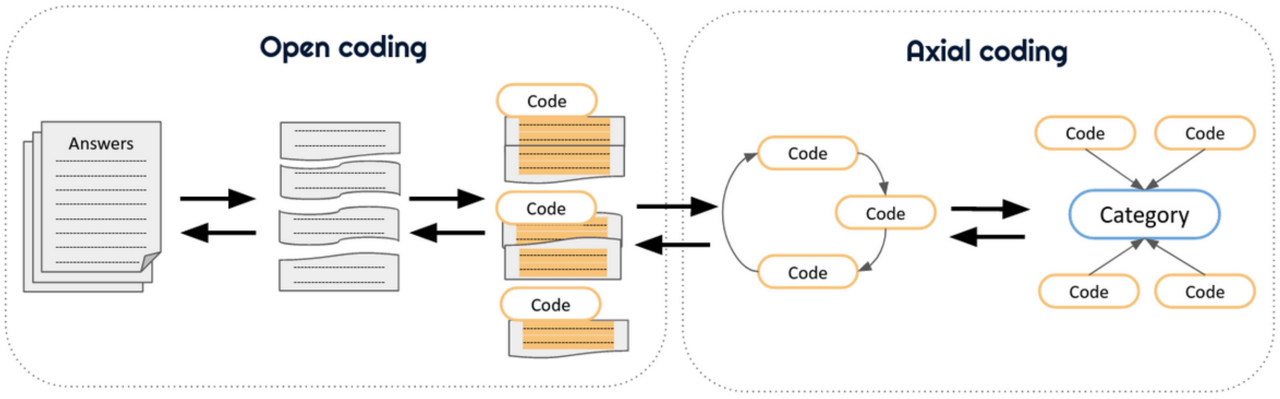


Figure 4.2: Open and axial coding from GT procedures.

After analyzing the effectiveness, development effort, and participants' satisfaction, the conclusions over the results are presented. In addition, the study's limitations and threats to validity are discussed, and lessons learned are presented.

4.2 Experiment to evaluate the usability of PPIs for CPU environments

Stream processing is one of the programming paradigms that has gained prominence over the years [13]. Stream processing applications are present in our day-to-day life, for example, on the Internet, where millions of data sources collect and exchange information through devices and social media [77]. The literature review in the previous chapter (Chapter 3) showed that the evaluation of developer productivity in the stream processing domain had been explored [46, 50, 79, 82, 98, 99, 131]. However, studies that perform controlled experiments with people have yet to be conducted for this purpose. Therefore, this section presents an experiment to assess the usability of three PPIs based on structured parallel programming from academia and industry for expressing parallelism in stream processing applications targeting multi-core systems: FastFlow, SPar, and TBB.

TBB [207, 244] is an open-source and general-purpose C++ template-based PPI from the industry, which provides patterns for stream parallelism exploration. FastFlow [5] is a representative PPI from the scientific community, which has a similar C++ template-based interface to TBB. A more recent research initiative for high-level stream parallelism in multicore systems is SPar [80, 82], an internal DSL (embedded in the C++ language) in the form of C++ annotation to avoid sequential code rewriting.

Some other PPIs can be used to exploit parallelism in multicore systems, of which OpenMP and Pthreads are the most popular. However, this study aims to use the structured programming approach to explore parallelism in stream processing applications. Therefore, as mentioned in Section 2.4, OpenMP and Pthreads were not used because it is only suitable for data parallelism exploitation and requires the programmer to implement extra synchronization mechanisms for stream parallelism exploitation. On the other hand, StreamIt is a remarkable PPI for stream parallelism exploitation based on the structured programming approach. However, the activities with StreamIt were discontinued in 2013, so this language was not used in this study. Therefore, in this study, a comparison of usability considered only SPar, FastFlow, and TBB.

This section is organized as follows. Section 4.2.1 presents the experimentation plan, Section 4.2.2 presents the results evaluation, and Section 4.2.3 presents some threats to validity.

4.2.1 Experimentation plan

This section presents the experimentation plan for this study, which was designed according to the methodology presented in Section 4.1. The experimentation plan includes the variables, the objectives, the hypotheses, the context, the activity given to the participants, the procedure followed, and the instruments used during this study.

Independent and dependent variables

The PPIs evaluated in this study are independent variables. In this study, we evaluated the usability of three PPIs for stream processing on multi-core systems: FastFlow, SPar, and TBB. Each of them has specific characteristics that influence the development of the applications. The experience of the programmer is one of the important independent variables [96]. In our study, the participants were intentionally beginners developers in the parallel programming domain to consider the impact of learning and help us understand the challenges faced by beginners. The study environment is also an independent variable. Our study is conducted in a university and not in an industrial environment. Therefore, the study is not affected by factors in the industry environment. Even so, a deadline of one day was set for the participants to complete the activity.

To evaluate the usability of PPIs, we assess the learnability, the effort required to parallelize a stream processing application, user errors, and satisfaction. The participants' learnability was assessed by the time accessing the material provided. The effort required to develop the parallel application includes the following activities: reading the procedure description, understanding the sequential application code, coding, debugging, and testing. The time in seconds was considered to evaluate the first five activities. However, the time testing the application was not considered because the participants left the application running and continued to perform other activities. Therefore, the number of executions was measured.

Goals

This section presents the objectives of this study, which were formulated from the problem to be solved according to the methodology presented in Figure 4.1. The main goal of this study is to compare the usability of SPar, TBB, and FastFlow PPIs for implementing stream parallelism in C++ applications for multi-core systems. The specific goals are as follows:

- Measure the learnability;
- Measure the time spent to exploit parallelism;
- Report the implementation errors;
- Report the users' satisfaction.

Hypothesis

We consider the following seven hypotheses in our experiment based on the goals. The first four refer to learnability, and the others to the development effort.

- H_{0_answer}**:** The time to answer the form is the same for FastFlow, SPar, and TBB;
- H_{0_study}**:** The time to access and study the material provided is the same for FastFlow, SPar, and TBB;
- H_{0_under}**:** The time to understand the code is the same for FastFlow, SPar, and TBB;
- H_{0_read}**:** The time to read the procedure description is the same for FastFlow, SPar, and TBB;
- H_{0_dev}**:** The development time is the same for FastFlow, SPar, and TBB;
- H_{0_debug}**:** The debugging time is the same for FastFlow, SPar, and TBB;
- H_{0_exec}**:** The number of execution is the same for FastFlow, SPar, and TBB.

Context of study

The participants were 15 graduate students from the Graduate Program in Computer Science (PPGCC) of the PUCRS, in Porto Alegre city South of Brazil. This study was part of the parallel programming course at PUCRS. Moreover, this study is conducted with participants having experience in the industry but beginners in parallel programming. The environment of this experiment is offline because it was conducted in an academic environment under controlled conditions and not in the industry. This study is specific because it focuses on assessing the usability of PPIs for stream processing in an academic environment. In addition, this study addresses a real problem common in the stream processing area: a video Open Source Computer Vision Library (OpenCV) processing application, which aims to extract a RBG channel from a video.

Activity of study

The activity given to the participants was to implement stream parallelism in an OpenCV video processing application, which aims to extract an RGB channel from a video. Listing 4.1 presents a piece of the RGB channel extraction application. The video processing application receives an input video and reads each video frame (line 6). A frame is processed through a series of operations to extract only the green channel (lines 9-18). Next, the frame with the green channel is written to the output video (line 19). This process is repeated until all frames have been processed (line 7) [182].

```
 1 int main(){
 2     // initialization of the steps
 3     while (1){
 4         Mat src, res;
 5         total_frames++;
 6         input Video >> src; // read frame
 7         if (src.empt y()) break; // check if end of video
 8         vector<Mat> spl;
 9         split (src, spl); // process - extract only the correct channel
10         for(int i =0; i < 3; ++i){
11             if (i != channel){
12                 spl[i] = Mat ::zeros(S, spl[0].t ype());
13             }
14         }
15         merge(spl, res);
16         cv::GaussianBlur(res, res, cv::Size(0, 0), 3);
17         cv::addWeight ed(res, 1.5, res, -0.5, 0, res);
18         Sobel(res,res,-1,1,0,3);
19         outputVideo << res; // write frame
20     }
21 }
```

Listing 4.1: RGB channel extraction application. Adapted from [182].

Experimental setup

The study participants used multi-core workstations with an Intel® Core^{TM} i7-4790 processor with eight cores (four physical), 3.6 GHz, and 15.6 GB of RAM. The operating system was Linux Mint 17.3 and G++ compiler version 5. The OpenCV version used was 3.1.0. The PPIs used were FastFlow 2.1.3, TBB 4.4.6, and SPar 1.

Procedure and execution

A folder with materials about each evaluated PPI was provided for the participants to access during the activity. Therefore, it was forbidden to consult any material on the Internet and use a cell phone to have more control over the material accessed by the participant. The measuring instruments used were questionnaires and a script to record the screens of the participants' machines. In addition, we defined some criteria for the activity to be considered complete: The participants could only complete the activity if the parallel application achieved a speedup greater than or equal to 3; if the parallel application produced the same result as the sequential version; and if the questionnaires were correctly filled out.

Initially, a pilot study was conducted to to adjust environmental problems and fix software issues, which was not included in the final results. Next, the selected participants answered a questionnaire to characterize them (Table 4.1). The characterization questionnaire was used to evaluate the participants' experience with parallel programming, video processing, and others. The participants should report their level of experience among:

- **0:** None, because I have never participated in such activities;
- **1:** I studied it in the classroom or in a book (I have only theoretical knowledge);
- **2:** I have practiced it in classroom projects (I have theoretical knowledge applied only in the university);
- **3:** I used it in personal projects (I have theoretical knowledge and individual practical experiences);
- **4:** I used it in some projects in industry or research (I have theoretical knowledge and little practical experience);
- **5:** I have used it in many projects in industry or research (I have theoretical knowledge and many real practical experiences).

The participants were divided into three groups (each group with five students), varying the sequence of using the PPIs to parallelize the same OpenCV video processing application. The first group used SPar, TBB, and FastFlow; the second group used TBB, SPar, and FastFlow; and the third group used SPar, FastFlow, and TBB. After finishing the

Table 4.1: Questionnaire used to characterize the experiment participants in a multi-core environment.

ID	Question
Q1.	Which is your academic background?
Q2.	Which is the name of your course?
Q3.	Which is your level of experience with command line and text editor on the Linux operating system? (From 0 to 5)
Q4.	Which is your level of experience with the C++ programming language? (From 0 to 5)
Q5.	Which is your level of experience with PPIs? (From 0 to 5)
Q6.	If you have any experience in Q5, please inform the PPIs for multi-core systems you have used and the features explored in your previous experiences:
Q7.	Which is your level of experience with developing stream processing applications (reading, writing, and processing files, network, video, audio etc.)? (From 0 to 5)
Q8.	If you have any experience in Q7, please specify which one you worked on:

Table 4.2: Questionnaire used to collect the results achieved by the participants.

ID	Question
Q1.	Which time did you start your activity?
Q2.	Which time did you finish your activity?
Q3.	Does the parallel program produce the correct result (the same as the sequential program)? (Yes/No)
Q4.	Which is the execution time of the sequential program (in seconds)?
Q5.	Which is the parallelism degree that reached the shortest execution time in the parallelized version? (From 0 to 8)
Q6.	Which is the execution time of the parallel program (in seconds)?
Q7.	Was the manual helpful in performing the activity? (Yes/No)
Q8.	Was the activity successfully completed? (Yes/No)
Q9.	Which were your main difficulties in implementing parallelism in this activity?

activity, the participants should report the results achieved and their satisfaction with each PPIs used through questionnaires (Tables 4.2 and 4.3). Finally, the participants' answers to questionnaires and screen-capture videos were analyzed to obtain data that can be used to evaluate the usability of the PPIs.

4.2.2 Evaluation

This section presents the results of this experiment, including the participants' profile, effectiveness analysis, productivity analysis, and satisfaction analysis. In addition,

Table 4.3: Questionnaire used to evaluate the satisfaction of the participants.

ID	**Question**
Q1.	With which PPIs did you perform your first activity?
Q2.	With which PPIs did you run your second activity with?
Q3.	If you had problems understanding the video application, what were they (please provide details)?
Q4.	Did you become uncomfortable with the screen-capture during the activity? Yes/No)
Q5.	Do you consider yourself able to perform activities with this PPI after completing this activity? (From 0 to 5)
Q6.	Do you consider yourself capable of performing activities with TBB after completing this activity? (From 0 to 5)
Q7.	Do you consider yourself capable of performing activities with FastFlow after completing this activity? (From 0 to 5)
Q8.	Which is your level of experience with the C++ programming language after completing this activity? (From 0 to 5)
Q9.	Which is your level of experience with developing stream applications after completing this activity? (From 0 to 5)
Q10.	Which PPI did you find the hardest?
Q11.	Justify your choice for question Q10 (please provide details):
Q12.	Which PPI did you find the easiest?
Q13.	Justify your choice for question Q12 (please provide details):
Q14.	If you need to parallelize an application similar to the application used in the activities, which interface would you choose?
Q15.	Justify your choice for question Q14 (please provide details):

we used the methodology shown in Figure 4.1 to perform the analysis of the effectiveness, productivity, and satisfaction of the participants.

Participants' profile

This section presents the profile of the participants in this study. All participants are graduate students familiar with command line and text editor in the Linux operating system, using it in the classroom and personal, industry or research projects. Most of the participants (9 participants) have experience in developing applications with the C++ programming language. However, three participants have no experience developing C++ applications, and three have only theoretical knowledge. In addition, most of the participants (10 participants) have no experience developing stream processing applications or only theoretical knowledge (three participants). Although two participants had practical experience in developing stream processing applications, they are not considered experienced developers in this domain because they usually focus on developing the business logic code and not on developing these types of applications. Therefore, the participants in this group are considered beginners in the stream processing domain and satisfy the target sample of this experiment.

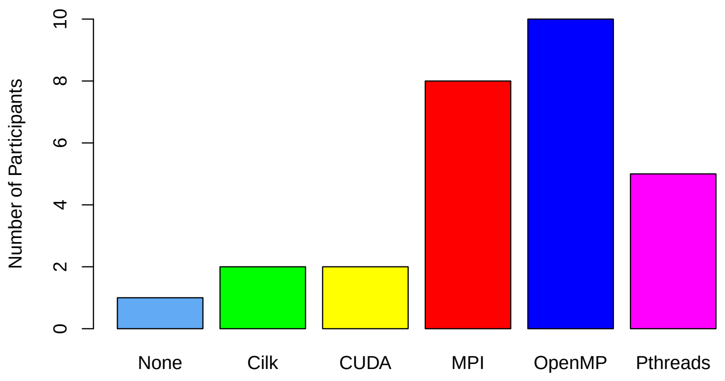


Figure 4.3: PPIs already used by the participants of the experiment.

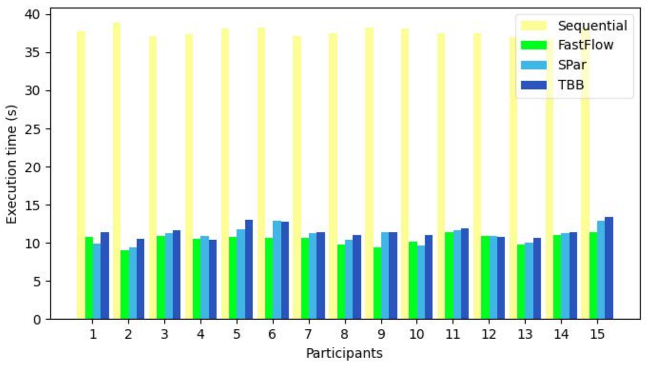
Figure 4.3 shows that only one participant has no experience with PPIs. Most of the participants have already used PPIs for multi-core (OpenMP, Pthreads, and Cilk), and distributed (MPI) systems. Only two participants have already developed GPU applications using CUDA. However, these participants are not experts in parallel programming because they have had a short time in contact with the PPIs, developing only basic activities in classroom, personal and research projects. In addition, only two participants have never used PPIs, having only theoretical knowledge about them.

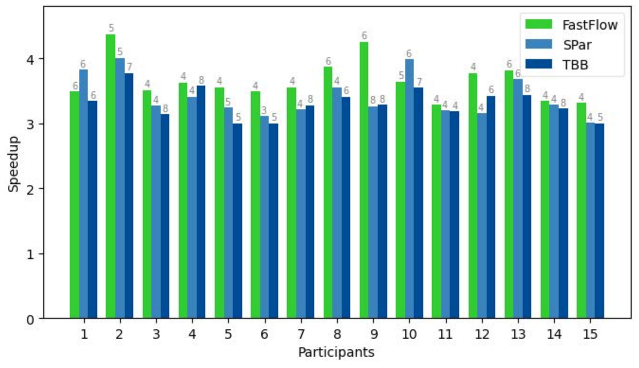
Effectiveness analysis

This section presents the effectiveness evaluation, which refers to the accuracy and completeness in which the participants achieve the specified objectives [108]. In this study, the participants' objective was to parallelize a video processing application using FastFlow, SPar, and TBB. Furthermore, this goal would only be achieved if the participant met the three criteria presented in Section 4.2.1: performance, program correctness, and the correct completion of the questionnaire (Table 4.2).

Figure 4.4a shows that all participants could reduce the execution time of the parallel applications. Figure 4.4b shows that all applications developed by the participants achieved the minimum required performance (speedup $\geq$ 3) using the three PPIs. Each participant used a certain number of threads shown on each bar's label in the graph. However, this speedup is not necessary the optimal performance. To achieve more performance would require the participants to have more knowledge about the architecture, which is beyond the scope of this study. Therefore, we did not ask for the maximum speedup but only the minimum speedup as an indicator of parallelization success and the conclusion of the activity.

To evaluate the correctness of a program, it was necessary to verify if its execution against a known input generates the expected output [94]. Our results showed that all participants were able to parallelize the application in order to produce the expected





(a) Average execution time of the sequential applications and execution time of the parallelized applications by the participants.

(b) Speedup achieved by the applications parallelized by each participant using FastFlow, SPar and TBB. The labels on each bar represent the number of threads used to run each of the versions.

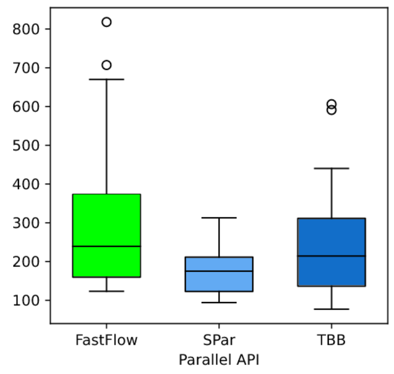
Figure 4.4: Execution time and performance achieved by each participant using FastFlow, SPar, and TBB.

output using the three PPIs. In addition, the questionnaire was adequately filled out by all participants. Therefore, FastFlow, SPar, and TBB showed effectiveness in this study.

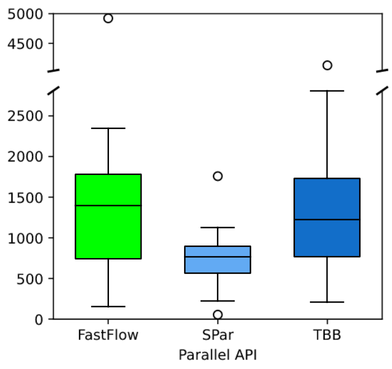
Productivity analysis

To assess the productivity, eight factors were collected from the screen-captured videos: time to answer the form (in seconds), time accessing the material available (in seconds), time to understand the application code (in seconds), time to read the procedure description (in seconds), development time (in seconds), debugging time (in seconds), and number of executions. Figure 4.5 shows the box plots for each of the metrics evaluated. Based on the box plots, it can be seen that the lowest medians for the time to answer the form, time to access the provided material, development time, debugging time, and the number of executions were obtained when participants used SPar. On the other hand, when they use FastFlow, the lowest medians were obtained for the time spent understanding the application code and reading the procedure description. Since the value of the medians alone can not determine which PPI provides the best usability, it is necessary to perform a hypothesis test.

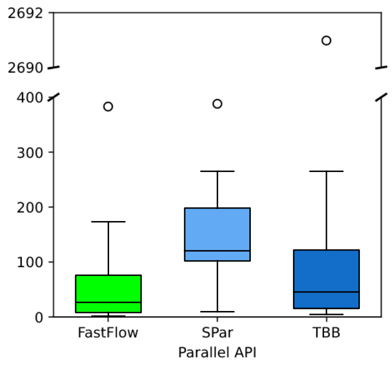
We performed the hypothesis test according to the methodology presented in Figure 4.1. Initially, we performed a normality test to verify whether the data collected had a normal distribution or not, in order to decide whether a parametric or non-parametric test would be performed. A parametric test should be used if the sample has a normal distribution (P-value $\geq$ 0.05). Otherwise, a non-parametric test is applied (P-value $<$ 0.05) [34, 224]. Therefore, we used the Shapiro-Wilk test at the conventional significance level (α = 0.05) [224] to verify whether the collected data had a normal distribution. In



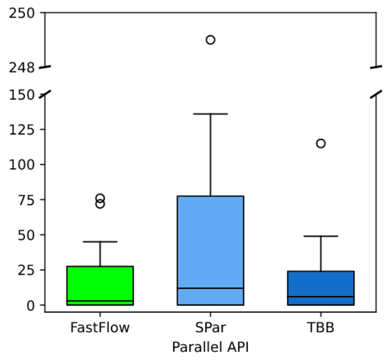
(a) Time to answer the form.



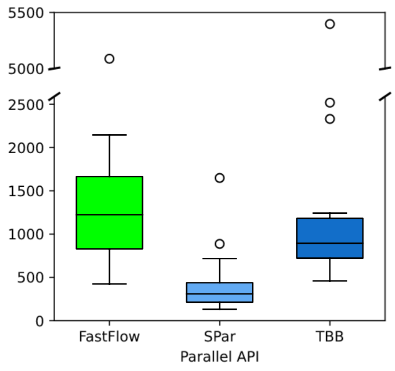
(b) Time accessing the material provided.



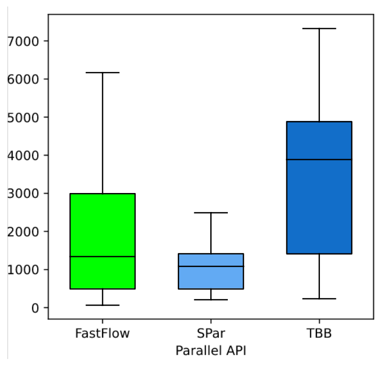
(c) Time to understand the application code.



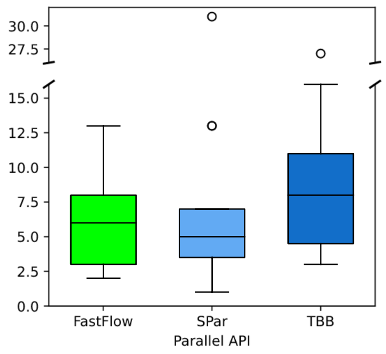
(d) Time to read the procedure description.



(e) Development time.



(f) Debugging time.



(g) Number of executions.

Figure 4.5: Box Plot for the times collected in the experiment.

Table 4.4: P-value of shapiro-wilk, wilcoxon and students's t tests.

Shapiro-Wilk	SPar	FastFlow	TBB
	p-value	p-value	p-value
Time to answer the form	**0.5166**	0.0033	0.0286
Time accessing the material provided	**0.4828**	0.0085	**0.0811**
Time to understand the application code	**0.3586**	0.0001	4.06E-07
Time to read the procedure description	0.0010	0.0002	0.0001
Development time	0.0003	0.0004	4.82E-05
Debugging Time	**0.5247**	0.0333	**0.2026**
Number of Executions	0.0003	**0.1289**	0.0068
Wilcoxon and Student's t tests	SPar x FastFlow	SPar x TBB	FastFlow x TBB
	p-value	p-value	p-value
Time to answer the form	0.0353	**0.1914**	**0.6788**
Time accessing the material provided	0.0026	0.0147	**1.0000**
Time to understand the application code	0.0170	**0.2524**	**0.2524**
Time to read the procedure description	**0.0618**	**0.1959**	**0.8753**
Development time	0.0006	0.0067	**0.6788**
Debugging Time	**0.0730**	0.0012	**0.1688**
Number of Executions	**0.7257**	**0.2778**	**0.1015**

addition, we chose this test because it is one of the most efficient tests and is independent of sample sizes [205].

Table 4.4 shows the results of the Shapiro-Wilk test, where *P*-values greater than α (in bold) indicate data with a normal distribution and *P*-values less than α indicate the opposite. In order to compare the PPIs, we performed a paired hypothesis test for two samples [212] considering the hypotheses presented in the Section 4.2.1: SPar versus FastFlow, SPar versus TBB, and FastFlow versus TBB. The hypothesis test between SPar and TBB was performed using the parametric Student's *t*-test for the time to consult material and debugging time because the samples have a normal distribution. For all other cases, the non-parametric Wilcoxon test was performed.

The results in the Table 4.4 showed a significant difference only between the time spent by the participants to answer the form in the activities using SPar and FastFlow (*P*-value $< \alpha$). This time may have been shorter for the SPar activity due to the lower volume of information reported by the participants. Similarly, there is a significant difference only in the time spent by the participants to understand the sequential application when comparing SPar with FastFlow (*P*-value $< \alpha$). These results highlight that the effort to understand the sequential application was the lowest in the activity using FastFlow because two groups finished the activity using it. Moreover, there is no significant difference between the efforts spent by the participants to understand the procedure in each of the activities because *P*-value is greater than α in all cases.

The hypothesis test showed a significant difference between the time spent by the participants studying SPar versus FastFlow and TBB, confirming the results in Figure

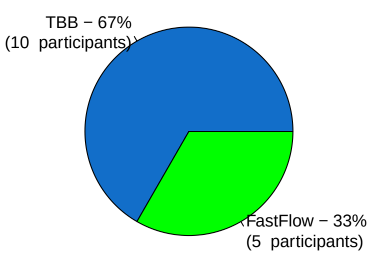
4.5b. Therefore, SPar is the easiest language to use for parallel stream processing because it presented a lower learning curve in relation to the other PPIs. There is a significant difference in the time required to develop an application using SPar regarding FastFlow and TBB (P-value $< \alpha$). with a short time of access to the material provided, the participants were already able to parallelize the application using SPar with less effort. Although, from the hypothesis test, it was possible to show that there is a significant difference only between the debugging times of SPar and TBB (P-value = 0.0012 $<$ 0.05). There is no significant difference between the debugging times of SPar and FastFlow, highlighting that the effort to correct programming errors is lower for SPar and FastFlow. Although, due to SPar's programming model, many programming errors were avoid. With both FastFlow and TBB there were compile-time errors caused by syntax errors in the use of C++ pointers and classes. Many participants had difficulty in passing a variable as a parameter to a FastFlow and TBB stage. In addition, some participants had difficulties in instantiating the stages, and running the pipeline using FastFlow and TBB. For more details about the challenges faced by the participants in this experiment, see the following paper [11].

The time testing the application was not considered because the participants left the application running and continued to perform other activities. Instead, the number of runs was measured to evaluate the effort to test the parallel applications. The results showed that the effort to test the applications parallelized with FastFlow, SPar, and TBB after fixing the debugs was equal (P-value $\geq \alpha$).

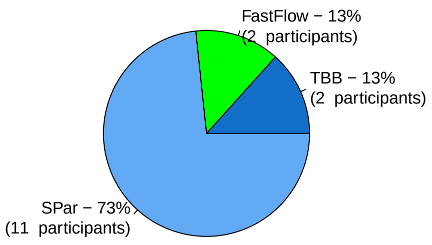
Satisfaction analysis

We performed qualitative and quantitative analyses to evaluate the participants' satisfaction according to the methodology shown in Figure 4.1. On the one hand, for the qualitative analysis, we constructed pie charts. On the other hand, for the quantitative analysis, we used the initial procedures (open coding and axial coding) of the GT methodology to perform a textual analysis of the participants' answers.

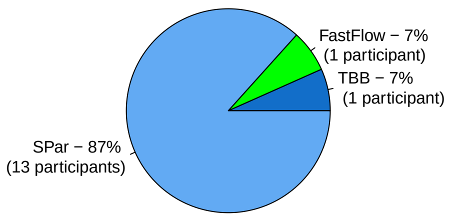
In the evaluation experiment questionnaire (Table 4.3), participants were asked which PPI they had more difficulty performing the activity. Figure 4.6a shows that none



(a) More difficult PPI to use.

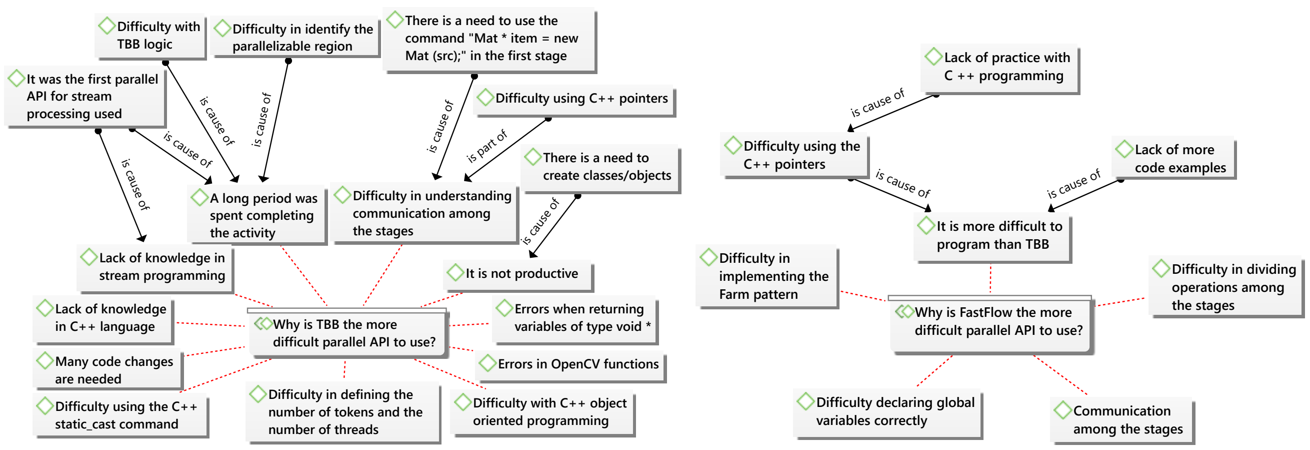


(b) Easiest PPI to use.



(c) PPI chosen by the participants.

Figure 4.6: Participants' opinion regarding the PPIs used.


Difficulty with TBB logic
Difficulty in identify the parallelizable region
There is a need to use the command "Mat * item = new Mat (src);" in the first stage
It was the first parallel API for stream processing used
is cause of
is cause of
is cause of
is cause of
Difficulty using C++ pointers
is cause of
is part of
There is a need to create classes/objects
A long period was spent completing the activity
Difficulty in understanding communication among the stages
is cause of
Lack of knowledge in stream programming
It is not productive
Lack of knowledge in C++ language
Why is TBB the more difficult parallel API to use?
Errors when returning variables of type void *
Many code changes are needed
Errors in OpenCV functions
Difficulty using the C++ static_cast command
Difficulty in defining the number of tokens and the number of threads
Difficulty with C++ object oriented programming
Lack of practice with C ++ programming
is cause of
Difficulty using the C++ pointers
Lack of more code examples
is cause of
is cause of
It is more difficult to program than TBB
Difficulty in implementing the Farm pattern
Difficulty in dividing operations among the stages
Why is FastFlow the more difficult parallel API to use?
Difficulty declaring global variables correctly
Communication among the stages

(a) Why is TBB harder? (b) Why is FastFlow harder?

Figure 4.7: Reasons reported by participants for why they find it more difficult to program stream processing applications with FastFlow and TBB.

of the participants found SPar more challenging to program than FastFlow and TBB. On the other hand, ten participants found it more difficult to program with TBB, and five with FastFlow. From a qualitative analysis of the participants' answers using the initial coding procedures of GT it was possible to identify the main reasons reported by the participants. Figures 4.7a and 4.7b show the graphical representations created using ATLAS.ti tool[2]. In these figures, a red line represents the relationships between codes and categories, and a black line represents the relationships between the codes. Moreover, the black lines have a label describing the type of relationship.

Figure 4.7a shows that participants found it more difficult to program with TBB due to their difficulties in programming pointers and structures with C++, understanding the TBB logic, and identifying the program's parallel region. Moreover, some participants needed more time to complete the activity since TBB was the first PPI used. In summary, the main difficulty reported by the participants was to understand the communication among the stages. The quotes below highlight that it may be related to their difficulty with C++:

> "I took time to understand how to correctly pass the values through pointers among the TBB stages."
> [Participant 1]

> "My biggest difficulty was to notice the details in the definition of the variables shared among the stages. For example, it was necessary to instantiate the shared variable with new in the first stage. In the following stages, the `static_cast` was used. I was having problems because I was not doing `Mat res = new Mat` in the first stage."
> [Participant 2]

[2]ATLAS.ti is a software used for qualitative data analysis and mixed methods research in academic, market, and user experience research. Available at: https://atlasti.com/.

Figure 4.7b shows that regarding FastFlow, the participants found it more difficult due to their lack of practice with C++ programming and difficulties understanding the logic required to implement the Farm pattern, communication, and operations division among the stages. Due to the lack of practice with C++ programming, there were difficulties in using C++ pointers. In addition, participants reported that due to the lack of more FastFlow code examples available, it became more difficult to use than the TBB.

Participants were also asked which PPI they found easiest to perform the activity. Figure 4.6b shows that 11 participants found it easier to program with SPar. Figure 4.8 shows the reasons given by the participants. In summary, SPar has a simple syntax, where there is no need for significant code changes, such as creating a class or structure for each stage. The quotes below highlight this aspects:

> "Its syntax is extremely simple, it does not require the use of structure or classes as in FastFlow and TBB, making the code easier to understand and apply parallelism."
> [Participant 4]
>
> "With SPar, there were no major changes in the code, only new annotations for the parts that can be parallelized."
> [Participant 6]

Despite these advantages, two participants found it easier to program using FastFlow or TBB because these were the last PPIs they used and consequently made the activity easier to perform.

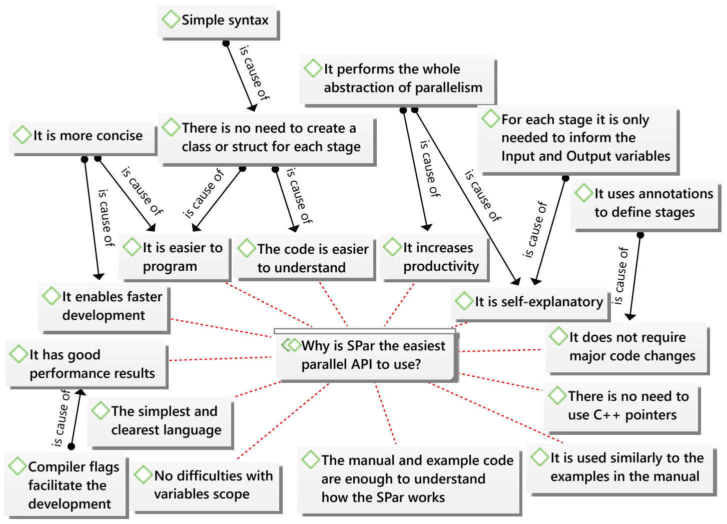

Simple syntax
is cause of
It performs the whole abstraction of parallelism
It is more concise
There is no need to create a class or struct for each stage
For each stage it is only needed to inform the Input and Output variables
It uses annotations to define stages
It is easier to program
The code is easier to understand
It increases productivity
It enables faster development
It is self-explanatory
Why is SPar the easiest parallel API to use?
It has good performance results
It does not require major code changes
The simplest and clearest language
There is no need to use C++ pointers
Compiler flags facilitate the development
No difficulties with variables scope
The manual and example code are enough to understand how the SPar works
It is used similarly to the examples in the manual

Figure 4.8: Reasons reported by participants for why they find SPar easier to program stream processing applications than FastFlow and TBB.

Participants also should inform which PPI they would choose to parallelize a stream processing application in a real-life situation in the future. Figure 4.6c shows that most participants would have chosen SPar. The main reason reported by the participants to choose SPar are presented in Figure 4.9. In summary, the participants chose SPar due to its annotation-based model, where the programmer only inserts annotations in the code to enable parallelism without requiring significant code changes. Despite the SPar advantages pointed out, TBB was chosen by one participant due to its simplicity, programmability, reliability, Intel support, number of users, and number of tutorials and examples available on the Internet. In addition, FastFlow was chosen by one participant due to its versatility, maturity, and learnability close to SPar.

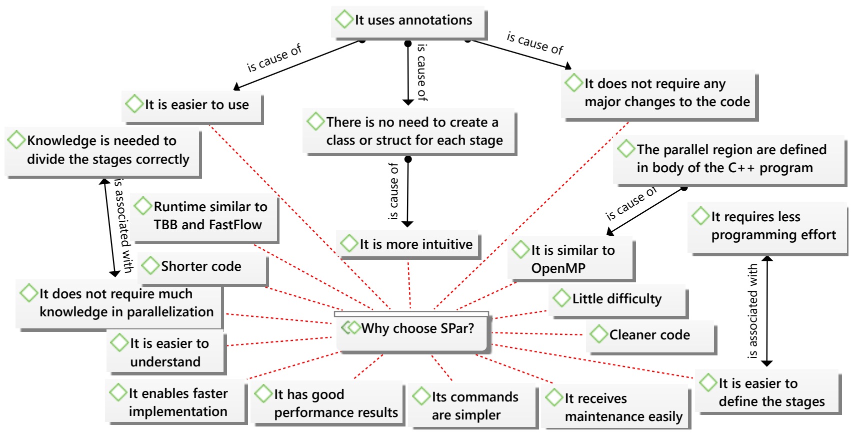

It uses annotations
is cause of
is cause of
is cause of
It is easier to use
There is no need to create a class or struct for each stage
It does not require any major changes to the code
Knowledge is needed to divide the stages correctly
The parallel region are defined in body of the C++ program
is associated with
is cause of
is cause of
Runtime similar to TBB and FastFlow
It requires less programming effort
It is more intuitive
It is similar to OpenMP
Shorter code
It does not require much knowledge in parallelization
Little difficulty
is associated with
Why choose SPar?
Cleaner code
It is easier to understand
It is easier to define the stages
It enables faster implementation
It has good performance results
Its commands are simpler
It receives maintenance easily

Figure 4.9: Reasons reported by the participants for choosing SPar if they needed to develop a stream processing application in a real-life situation.

The evaluation of the participants' satisfaction confirmed the results concerning productivity. When using SPar, the participants did not have to make significant modifications to the code to provide parallelism. With FastFlow and TBB, it was necessary to create structures (`class` and `struct`) for each stage, resulting in errors related to the C++ programming language. In addition, the need for the `return` command on a stage implemented through C++ structures is related to one of the main programming logic errors made by the participants using FastFlow and TBB. From the satisfaction analysis, we observed that using C++ structures and pointers is one of the main difficulties reported by the participants when performing the activity using FastFlow and TBB. In addition, participants found it easier to use SPar (11 participants) and would choose it if they needed to solve a real problem in future activities (13 participants) since it is not necessary to use pointers and create structures to implement parallelism with this PPI.

4.2.3 Discussion and threats to validity

The results show that SPar, FastFlow, and TBB offer effectiveness in solving the activity given to the participants because all applications produced the output video correctly with a minimum speedup of 3. However, SPar is the easiest language to use for parallel stream processing because it presented a lower learning curve in relation to the other PPIs and provided more productivity to the developers. With a short time of access to the material provided, the participants could already parallelize the application using this DSL. In addition, the learning curve for SPar is shorter since the simplicity of this PPI.

When using SPar, the participants did not have to make major modifications to the code to provide parallelism. With FastFlow and TBB, it was necessary to create structures (`class` and `struct`) for each stage, resulting in errors related to the C++ programming language. In addition, the need for the `return` command on a stage implemented through C++ structures is related to one of the main programming logic errors made by the participants using FastFlow and TBB. From the satisfaction analysis, we observed that the use of C++ structures and pointers is one of the main difficulties reported by the participants when performing the activity using FastFlow and TBB. In addition, participants found it easier to use SPar (11 participants) and would choose it if they needed to solve a real problem in future activities (13 participants) since it is not necessary to use pointers and create structures to implement parallelism with this PPI.

From the participants' point of view, SPar is easier to use PPI because of its annotation-based programming model. This feature allowed the participants to develop parallel stream processing applications with less effort. Evaluating the participants' satisfaction confirmed the results concerning the development effort and productivity. Therefore, SPar offers the best usability in developing stream processing applications for multi-core systems because it provides the best productivity indicators and user satisfaction.

This study showed promising results. However, some threats to validity remain. The learning effect is a threat to internal validity because no group started the activity with FastFlow, and it was the last PPI used by two groups. Therefore, participants may have learned faster how to solve the problem with FastFlow, which is a threat to internal validity. Another threat to internal validity is related to instrumentation. The participants may enter incorrect data into the self-assessment questionnaire, such as the activity start and end time. To face this threat, in addition to the questionnaires, the machine screens used by the participants during the study were automatically recorded using a monitoring script and later checked. However, using a human observer has some challenges because it is time-consuming and requires double-checking. One approach that could reduce this effort is automatically generating logs from the experiment execution. However, none of the tools available met our requirements.

As a threat to external validity, it is not possible to generalize the results as only stream processing in multi-core environments was evaluated. The design of the experiment may be a threat to construct validity because the SPar programming model is different from the TBB and FastFlow programming models. However, all of them are based on structured parallel programming and support stream parallelism. Finally, a threat to conclusion validity is the sample size, which may need to be more representative. Therefore, an experiment with a larger sample of participants may be required to confirm the results. However, these threats to validity do not discredit our research and the conclusions, where several lessons-learn were taken from qualitative and quantitative results.

4.3 Initial study to evaluate the usability of PPIs for GPU environments

This section presents a initial study to investigate the usability of the PPIs in the GPU environment. GSParLib is recent approach proposed by Rockenbach [208] that provides programmers with abstractions through CUDA and OpenCL code generation (see Section 2.4.4). Therefore, this study aims to assess the usability of GSparLib compared to well-established GPU programming interfaces. For this purpose, we evaluate the development of parallel applications for GPU in a graduate course using different PPIs: CUDA [35], GSPaLib [208], OpenACC [61], and OpenCL [166]. Section 4.3.1 presents the experimentation plan, Section 4.3.2 presents the results evaluation, and in Section 4.3.3 are some discussions and threats to validity.

4.3.1 Experimentation plan

This section presents the experimentation plan for this initial study, which was designed according to the methodology presented in Section 4.1. This experimentation plan presents the variables, the objectives, the hypotheses, the context, the activity given to the participants, the procedure, and the instruments used for data collection.

Independent and dependent variables

The following metrics used in this study to evaluate GPU programming are considered dependent variables: development time, SLOC, difficulties faced by students, and participant satisfaction. Unlike the experiment in the multi-core environment presented in Section 4.2, the participants' machine screens were not recorded in this study. This study presents an initial effort to evaluate the usability of GPU PPIs. Therefore, the total development time was considered for the productivity evaluation instead of the time spent by the participants to complete each activity performed, such as the time to consult and study the material, development time, and debugging time. Therefore, to complement our analysis, we included the metric the number of SLOC. The PPIs used in this study are independent variables because they have specific characteristics that may impact the development effort. The students' experience can also affect the development effort as they do not have extensive experience in GPU programming. The study environment is also an independent variable.

Goals

The main goal of this study is to compare the usability of CUDA, GSParLib, OpenACC, and OpenCL PPIs for implementing parallelism in C/C++ applications for GPU. The specific goals are as follows:

- Measure the time spent to exploit parallelism;
- Measure the number of SLOC;
- Report the implementation errors;
- Report the users' satisfaction

Hypotheses

Based on the goals, we consider the following two hypotheses in our experiment:

- H_{0_effort}**:** The effort required to implement parallelism is the same for CUDA, GSParLib, OpenACC, and OpenCL
- H_{0_sloc}**:**The number of SLOC required to implement parallelism is the same for CUDA, GSParLib, OpenACC, and OpenCL

Context of study

This study is offline because it was conducted in an academic environment under controlled conditions. The participants were graduate students from the Heterogeneous Parallel Programming course of the PPGCC at the PUCRS in Porto Alegre city, South of Brazil. This study is specific because it focuses on assessing the usability of PPI for GPU programming in an academic environment.

Activity of study

The activity given to the students was to implement parallelism in the Animal Rescue with Drones Problem program to accelerate the calculations. This problem uses drones to find environments to reintroduce animals into the wild rescued from various adverse situations. There are three groups of animals, based on where they need to be introduced: a place with the lowest possible height from sea level, a place with the highest possible height from sea level, or a specific height from sea level. The drones perform a series of computations to find the most suitable environment to reintroduce them.

Experimental setup

To perform the activity using CUDA, one of the students used a machine equipped with an Intel® Core^{TM} E5-2698 v3 processor (16 physical cores and 32 threads) with 2.40 GHz, 16 GB of RAM memory, and a GPU NVIDIA V100 Volta (5,120 CUDA cores) with 32 GB of VRAM memory. The operating system was Ubuntu 18, GCC compiler 7.3, and NVCC compiler 11.3. For the other PPIs, this same participant used a machine equipped with an Intel® Core^{TM} i7-6700k (4 cores/8 threads) with 32 GB of RAM, and a GPU NVIDIA GTX-1070 Pascal (1,920 CUDA cores) with 8 GB of DRAM. The operating system was Linux Mint 20.2, GCC compiler 9, and NVCC compiler 11.

The other students used a machine equipped with an Intel® Core^{TM} E5-2620 v3 processor (6 cores) with 2.40 GHz, 64 GB of RAM memory, and a GPU Nvidia Titan X Pascal (3,584 CUDA cores) with 12 GB of VRAM memory. The operating system was Ubuntu 20.04, GCC 9.3, and NVCC compiler 11.3. The -O3 compiler flag was set when compiling all the applications. The PPIs used were CUDA 11.4, GSParLib 1, OpenACC, and OpenCL 1.2.

Procedure and execution

Initially, the students answered a questionnaire to characterize them. Table 4.5 shows the questions asked of the participants. This questionnaire was refined from the questionnaire used in the previous experiment (Table 4.1) in order to assess the participants' level of specific experience in different programming environments, such as multi-core, GPU, cluster, and FPGA.

Before starting the activity, an order was established for the participants to use each PPI to parallelize the Animal Rescue application: CUDA, OpenACC, OpenCL, and GSParLib. For GSParlib, students were given the option of using the Driven or Pattern interface. During the activity, the students could access any online material to study and learn about the PPIs and help them perform the activities. Unlike the previous study (Section 4.2), where internet access was prohibited, in this study, it was allowed due to the added complexity of developing parallel applications for GPU. After paralleling the application with each PPI, the students were to answer a questionnaire to report productivity data and evaluate the activity, which is showed in Table 4.6. This questionnaire was designed by merging and reformulating the questionnaires of the Tables 4.2 and 4.3 used in the experiment in the multi-core environment. Moreover, the students had one semester to finish the activities.

Table 4.5: Questionnaire used to characterize the experiment participants in a GPU environment.

ID	Question
Q1.	What is your nationality?
Q2.	Which is your academic background?
Q3.	What is your affiliation (company or educational institution)?
Q4.	In which country is your affiliation located?
Q5.	Which is your level of experience with command line and text editor on Linux operating system? (From 0 to 5)
Q6.	Which is your level of experience with the C++? (From 0 to 5)
Q7.	Which is your experience level with parallel programming for multi-core systems? (From 0 to 5)
Q7.1	If your answer in Q7 was other than “none”, please inform which PPIs for multi-core systems (e.g. Pthreads and OpenMP) you have used, and the features explored in your previous experiences:
Q8.	Which is your experience level with parallel programming for GPU systems? (From 0 to 5)
Q8.1.	If your answer in Q8 was other than “none”, please inform which PPIs for GPU systems (e.g. CUDA and OpenACC) you have used and the features explored in your previous experiences:
Q9.	Which is your experience level with parallel programming for cluster? (From 0 to 5)
Q9.1.	If your answer in Q9 was other than “none”, please inform which PPIs for cluster (e.g. MPI) you have used and the features explored in your previous experiences:
Q10.	Which is your experience level with parallel programming for FPGA?
Q10.1.	If your answer in Q10 was other than “none”, please inform which PPIs for FPGA systems (e.g. OpenCL) you have used and the features explored in your previous experiences:

4.3.2 Results evaluation

This section presents the evaluation of the results: profile of the participants, effectiveness analysis, productivity analysis, and satisfaction analysis. In addition, we used the methodology shown in Figure 4.1 to analyze the effectiveness, productivity, and satisfaction.

Participants’ profile

The students have experience with the GNU/Linux operating system, command line, and C++ application development. They have experience in developing parallel applications for multi-core systems, mostly using OpenMP, Pthreads, and TBB as can be seen

Table 4.6: The questionnaire used to collect the results achieved by the participants and assess their satisfaction.

ID	Question
Q1.	In how many hours could you parallelize the application?
Q2.	Which PPI do you use?
Q3.	Were you successful in doing the activity? That is, did the parallel application produce the expected result (the same as the sequential application)?
Q4.	What is the execution time of the sequential application (in seconds)?
Q5.	What was the shortest execution time of the parallel application (in seconds)?
Q6.	What was the degree of parallelism (number of workers) that achieved the shortest execution time in the parallelized version?
Q7.	How easily were you able to perform the activity?
Q8.	What were your main difficulties in implementing parallelism in this activity?
Q9.	If you have accessed any documentation/materials during the activity, please provide the references or links used:
Q10.	How helpful was the documentation/material?
Q11.	Overall, how satisfied or dissatisfied are you with the parallel programming interface used?
Q12.	What are the reasons for your satisfaction or dissatisfaction?
Q13.	How likely are you to recommend this parallel programming interface to friends and colleagues?
Q14.	What are the reasons why you would or would not recommend this PPI to friends and colleagues?

in Figure 4.10. They have prior knowledge about parallel programming for cluster using PPIs, such as MPI, OpenMPI and HPX (Figure 4.10). However, one of the students has only theoretical knowledge about MPI. Only two students have experience with GPU. One of the students has done simple classroom activities using CUDA, and the other has done classroom activities using OpenACC and OpenCL and has done research activities with CUDA. In addition, none of the students developed parallel applications to FPGA.

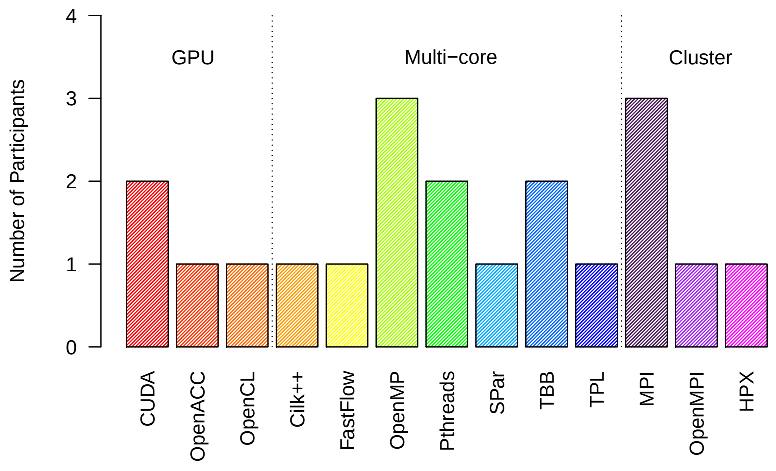


Figure 4.10: PPIs already used by the students of the GPU study.

Effectiveness analysis

In this study, the participants' objective was to parallelize a video processing application using CUDA, GSParLib, OpenACC, and OpenCL. Inicially, the study had six participants, whose were enrolled in the Heterogeneous Parallel Programming course. However, only four students were able to complete the activity with correctness, where the parallel application produced the same result as the sequential version.

Table 4.7 shows the average execution time obtained by the four participants. CUDA was the PPI that provided the best performance based on a sequential application execution time of 165.94 seconds (average of the four participants). On the other hand, when using OpenACC the participants had the worst results regarding performance. This happened because the execution time with OpenACC was five times longer than the sequential version for one of the participants because he was not able to implement GPU optimizations using this interface. Despite the poor result regarding OpenACC performance, this participant was considered able to complete the activity because the parallel application produced the correct result for the Animal Rescue problem, which was the same as the sequential application. However, since the sample size is very small (four participants) further study is needed to conclude which interfaces provide the best performance since CUDA, GSParLib, and OpenCL showed close speedups.

Table 4.7: Results (mean and standard deviation) of the execution time, SLOC and development time for the GPU study.

		CUDA	GSParLib	OpenACC	OpenCL
Execution time (s)	Mean	4.35	5.67	146.06	4.75
	σ	2.09	4.67	261.14	3.17
SLOC	Mean	999.75	1005.25	**803.00**	998.50
	σ	481.38	189.82	183.48	288.51
Development time (hr)	Mean	18.33	**6.17**	9.25	9.25
	σ	3.56	5.45	4.27	5.74

Productivity analysis

The metrics used to evaluate the productivity were SLOC and development time measured in hours. Table 4.7 presents the mean and standard deviation of the four graduate students' results for the SLOC and development time. The results showed that OpenACC required fewer SLOC to parallelize the application. On the other hand, GSParLib required more SLOC, since the participants chose to use the Driven interface for parallelization. GSParLib's Pattern interface would have reduced the number of SLOC, as seen in section 2.4.4. However, the participants chose the Driven interface over higher levels of abstraction because of the flexibility to manipulate GPU resources.

Table 4.8: P-value of the Shapiro-Wilk test and Students' t-test.

	GSParlib	OpenACC	OpenCL	CUDA		
	p-value	p-value	p-value	p-value		
SLOC	0.3018	0.4050	0.1092	0.0622		
Dev. time	0.5041	0.0657	0.4162	0.2041		
	GSParLib x OpenACC	GSParLib x OpenCL	GSParLib x CUDA	OpenACC x OpenCL	OpenACC x CUDA	OpenCL x CUDA
	p-value	p-value	p-value	p-value	p-value	p-value
SLOC	**0.0212**	0.9330	0.9733	0.0760	0.2885	0.9920
Dev. time	0.2992	**0.0076**	**0.0052**	1	**0.0443**	**0.0099**

The project size is usually related to the effort needed to develop it. However, as can be seen, the shortest development time was achieved using GSParLib. CUDA also requires fewer SLOC to parallelize the application than GSParLib and presents the longest development time. Therefore, fewer SLOC does not indicate less development effort but can be an indicator of verbosity. Nevertheless, averages of the SLOC and development time alone can not determine which PPI is more verbose and provides the best productivity and a hypothesis testing is necessary.

We performed the hypothesis test according to the methodology presented in Figure 4.1. Initially, to verify which type of hypothesis test must be conducted (parametric or non-parametric), we performed the Shapiro-Wilk normality test at the conventional significance level (α) of 0.05 [224]. We chose this test because it is one of the most efficient tests for all distribution types and can be used regardless of sample size [205].

The results in Table 4.8 shows that all samples have a normal distribution, since for all cases the P-value is greater than α (in bold). Therefore, we performed a paired Student's t-test for two samples: GSParLib versus OpenACC, GSParLib versus OpenCL, GSParLib versus CUDA, OpenACC versus OpenCL, OpenACC versus CUDA, and OpenCL versus CUDA. Table 4.8 also shows the results achieved for the Student's t-test at the conventional α equal to 0.05. For the SLOC metric, the hypothesis test showed a significant difference only between GSParLib and OpenACC (P-value = 0.212). Therefore, statistically, it could only be confirmed that OpenACC is less verbose than GSParLib because in all other cases the P-value is greater than α.

Regarding development time, when we analyzed only the average of the participants, GSParLib required less development effort. However, the hypothesis test showed no significant difference between the average development times for GSParLib and OpenACC (p-value $> \alpha$). This occurred due to the large data spread, as can be seen from the standard deviation in Table 4.7. This result may have occurred due to the sample size, which is composed of only four students. Therefore, more students are needed to obtain more concise conclusions. Moreover, there is also no significant difference between

OpenACC and OpenCL (P-value = 1) because the average development time required by the participants to complete the activity was the same for both PPIs. For all other cases (P-value $< \alpha$), the hypothesis test showed that there is a significant difference between the effort employed by the participants to perform the activity. These results confirm that developing an application with CUDA requires more effort and lower productivity.

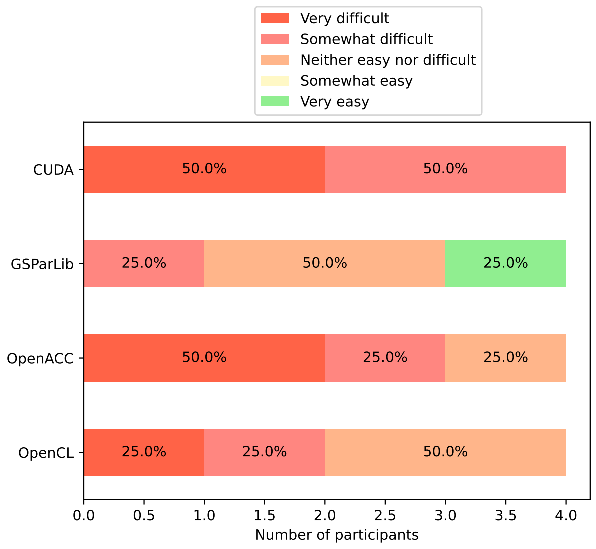
Satisfaction analysis

We performed qualitative and quantitative analyses to evaluate the participants' satisfaction according to the methodology shown in Figure 4.1. For the qualitative analysis, we constructed stacked charts. On the other hand, for the quantitative analysis, we used the initial procedures of the GT methodology to perform a textual analysis of the participants' answers.

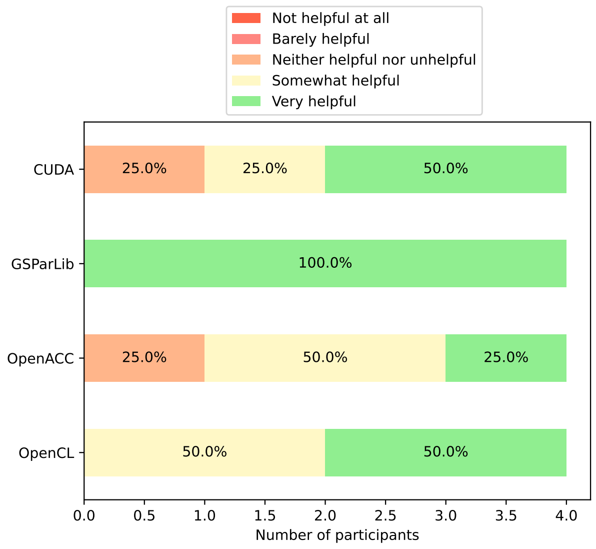
In the evaluation questionnaire (Table 4.6), the participants were asked which PPI they found most challenging to complete the proposed activity, which provided the most helpful documentation, which they were most satisfied using, and which they would recommend for friends and colleagues to use. GSParLib was the easiest-to-use PPI for developing applications for GPU systems from the participants' opinion (Figure 4.11a), and consequently the PPI with which the most students were satisfied to use (Figure 4.11c). From a qualitative analysis we identified the main reasons for the participants' satisfaction with GSParLib, which can be seen in Figure 4.12. This library is already designed to make it easier to develop applications by abstracting CUDA and OpenCL code. This advantage was one of the main reasons reported by the students to justify their satisfaction with GSParLib. In addition, the students pointed out that GSParLib provides several resources to implement the parallel strategy, such as functions and synchronization.

The students also pointed out the verbosity of this interface as a negative point, which may explain why parallel applications with GSParLib have more SLOC. Nevertheless, Figure 4.11d shows that 75% of the participants find it very likely to recommend GSParLib to others looking to develop parallel applications for GPU environments. In addition, Figure 4.11b shows that all participants felt the material available for consultation regarding GSParLib made programming easier when using it.

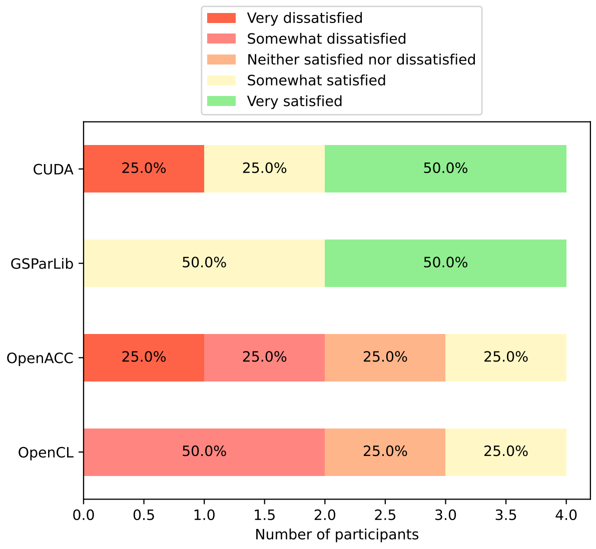
CUDA was the most challenging PPI to use from the students' opinion, as can be seen in the Figure 4.11a. Nevertheless, most of the participants felt satisfied to use it (Figure 4.11c). Figure 4.13 presents the main reasons for the participants' satisfaction. The students found CUDA easy to use, although it requires an initial learning curve. Furthermore, they found it very powerful and very flexible to program. In addition, the participants found CUDA simpler because it is similar to C, since it is an extension of this programming language.



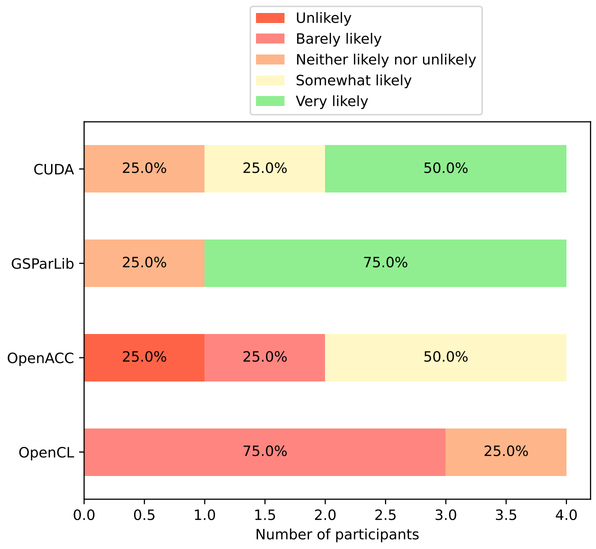
(a) Participants' difficulty level with GPU PPIs.



(b) Participants' satisfaction level with the material available on GPU PPIs.



(c) Participants' satisfaction level with GPU PPIs.



(d) Recommendation level of GPU PPIs by the participants.

Figure 4.11: Evaluating the satisfaction level of participants in a GPU study.

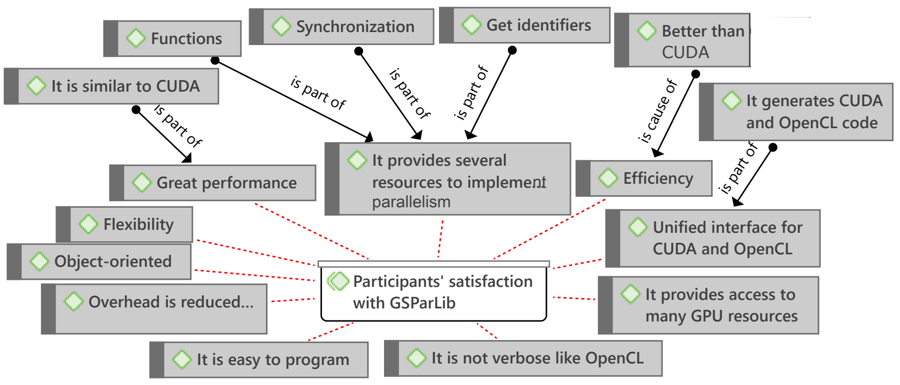

Functions
Synchronization
Get identifiers
Better than CUDA
It is similar to CUDA
is part of
is cause of
It generates CUDA and OpenCL code
Great performance
It provides several resources to implement parallelism
Efficiency
Flexibility
Unified interface for CUDA and OpenCL
Object-oriented
Participants' satisfaction with GSParLib
Overhead is reduced...
It provides access to many GPU resources
It is easy to program
It is not verbose like OpenCL

Figure 4.12: Reasons for participants' satisfaction with GSParLib.

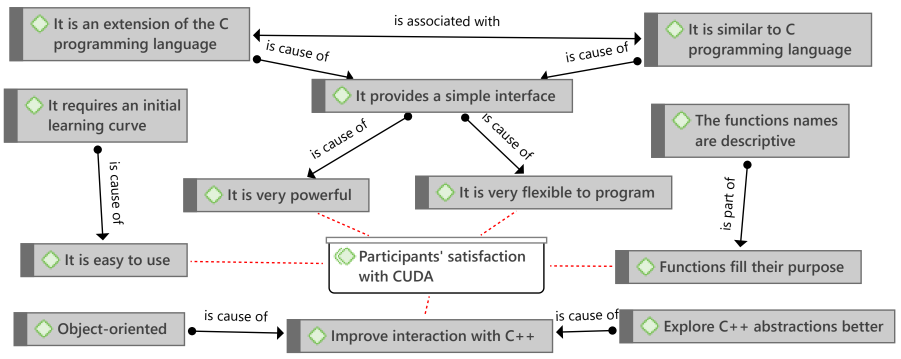

It is an extension of the C programming language
is associated with
It is similar to C programming language
is cause of
is cause of
It provides a simple interface
It requires an initial learning curve
is cause of
is cause of
The functions names are descriptive
is cause of
It is very powerful
It is very flexible to program
is part of
It is easy to use
Participants' satisfaction with CUDA
Functions fill their purpose
Object-oriented
is cause of
Improve interaction with C++
is cause of
Explore C++ abstractions better

Figure 4.13: Reasons for participants’ satisfaction with CUDA.

Regarding the dissatisfaction with CUDA, the main reason reported was participants’ difficulty in restructuring the code to remove dependencies. The students also had difficulties debugging the code, dealing with architecture-specific details, understanding the programming logic, and choosing the parallelization strategy. Although, the documentation available on CUDA was useful for doing the activity in the opinion of most participants (Figure 4.11b).

The participants had a lot of difficulties when programming with OpenACC and OpenCL (Figure 4.11c). These were the interfaces that the students were least satisfied using (Figure 4.11c), and consequently were the least recommended by them (Figure 4.11d). Only one student was somewhat satisfied with OpenCL because of its portability. On the other hand, the other three participants pointed out several reasons for their dissatisfaction with OpenCL, such as the lack of code readability and verbosity (Figure 4.14). Furthermore, students highlighted that they spent much time implementing the applications due to OpenCL verbosity.

Regarding OpenACC, one student was somewhat satisfied because of its similarity with OpenMP. On the other hand, Figure 4.15 shows that the other students reported dissatisfaction with OpenACC due to their difficulty implementing GPU optimization, manipulating the code flexibly, and understanding the code generated. The main reason for the lack of flexibility was that OpenACC is a very high-level PPI. Hence, it is not as flexible as CUDA and OpenCL. Moreover, the students could not use the same parallelism

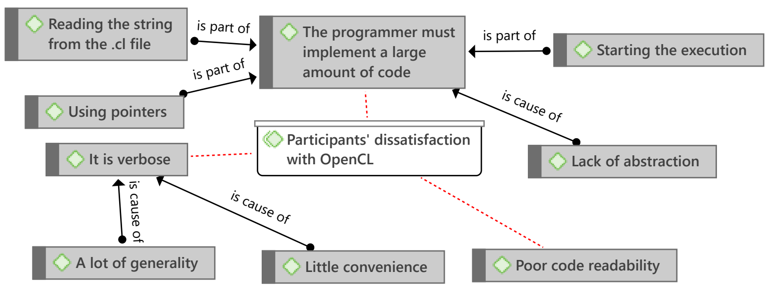


Figure 4.14: Reasons for participants’ dissatisfaction with OpenCL.

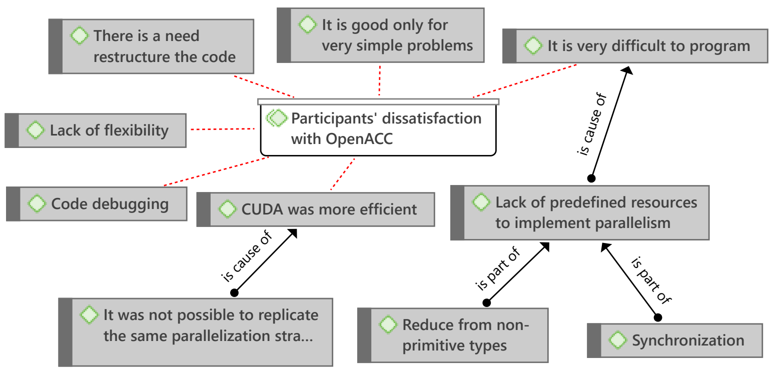

There is a need restructure the code
It is good only for very simple problems
It is very difficult to program
Participants' dissatisfaction with OpenACC
Lack of flexibility
is cause of
Code debugging
CUDA was more efficient
Lack of predefined resources to implement parallelism
is cause of
It was not possible to replicate the same parallelization stra...
is part of
Reduce from non-primitive types
is part of
Synchronization

Figure 4.15: Reasons for participants' dissatisfaction with OpenACC.

strategy used in the parallelized versions with CUDA and OpenCL, resulting in a higher development effort.

4.3.3 Discussion and threats to validity

The results show that CUDA, GSParLib, and OpenCL effectively solved the activity given to the participants because all applications produced the output correctly and reduced the execution time required. On the other hand, OpenACC presented the worst results regarding performance. However, we have not aimed to evaluate the execution time or the speedup achieved by the students when parallelizing the applications in this empirical study since not all students advanced in this regard. They were beginning GPU programming students, so they needed to learn about the architecture to perform optimizations to get the best performance.

From the results, we conclude that GSParLib provides the best productivity in developing parallel applications for GPU systems. GSParLib productivity occurred because the library is designed to make it easier to develop applications by abstracting CUDA and OpenCL code. This GSParLib feature was the main reason the students reported to justify their satisfaction with using GSParLib. In addition, more students were satisfied with it compared to the other PPIs. Therefore, we concluded that GSParLib provides the best usability in developing parallel applications for GPU environments.

This empirical study has some threats to validity. The learning effect threatens internal validity due to the order in which the participants use the PPIs: CUDA, OpenACC, OpenCL, and GSParLib. Therefore, the development time of GSParLib may be affected because the students already knew the target problem before using it. Another threat to internal validity can be the self-assessment questionnaires used to collect students' results, as they may report incorrect data. We cross-checked the students' responses to overcome this limitation.

The developer profile can be also considered a threat to external validity since the results cannot be generalized. However, our goal was to evaluate the learning of beginning students in parallel programming for GPU (context-specific). Finally, the sample size is a threat to the conclusion validity. Several lessons were learned from the qualitative and quantitative results, although only four students may not be as statistically representative. This study was conducted remotely during the COVID-19 pandemic. The remote activities during the pandemic period made obtaining participants for our empirical study difficult, justifying the participant sample size. Therefore, it is necessary to conduct a new experiment with a larger sample of participants to confirm this study's results.

4.4 Final remarks

This chapter proposed improving the usability evaluation of PPIs. For this purpose, we presented a methodology to encourage and support the conduct of experiments with people in order to determine the usability of PPIs. The proposed methodology was used to conduct two studies with graduate students. This methodology supported the experimentation process in all stages of the experiments: planning, execution, and analysis of results. Therefore, our experiments were conducted more easily since we used the proposed methodology to guide what should be done and how it should be done. In addition, this methodology can also serve as a guide to other researchers in the parallel programming field.

In this chapter, a controlled experiment was conducted to assess the usability of three PPIs for expressing parallelism in stream processing applications that execute on multi-core CPU environments: FastFlow, SPar, and TBB. From the results, we concluded that SPar offered the best usability in developing stream processing applications for multi-core systems. The results showed that SPar offers efficiency, the best productivity indicators, and the highest user satisfaction. This occurred due to SPar's annotation-based programming model, which required a few code changes by the programmer. In addition, our results may also help in teaching parallel programming because the participants were beginners in this domain, and we identified the main challenges faced by them.

An initial study was also conducted to evaluate usability of four PPIs for exploring parallelism in GPU environments: CUDA, GSParLib, OpenACC, and OpenCL. From the results obtained we concluded that GSParLib provided the best usability in developing parallel applications for GPU environments. Our results showed that GSParLib obtained the best productivity indicators and the highest user satisfaction, although it does not show the best results regarding performance. However, the execution time results are very close to CUDA and OpenCL. Therefore, GSParLib reduces the development effort without significant loss in performance. These initial results give indications that high-level abstractions can reduce the effort of developing parallel applications.

The results showed that parallelizing applications for multi-core CPU environments requires less programming effort than for many-core GPU environments. The total development time, including all tasks performed by the 15 participants, was around 1.72 hours for FastFlow, 1.67 hours for SPar, and 2.17 hours for TBB. The efforts required to parallelize applications in a multi-core environment were at least 64.83% lower when compared to the GPU results. Although preliminary, these results confirm the ease of programming for multi-core environments. In addition, FastFlow, SPar, and TBB are structured PPIs, which may influence the ease of programming. Nevertheless, further investigation is necessary since the results of only four students were considered in the GPU study. In addition, the applications evaluated in the studies for multi-core CPU and GPU are different. Therefore, a new experiment is also needed to compare the effort required to develop a parallel application for multi-core CPU systems versus many-core GPU systems.

5. CODING PRODUCTIVITY METRICS EVALUATION

Coding productivity is a factor that, together with effectiveness and users' satisfaction, are indicators of usability [108]. Based on usability or productivity indicators, it is possible to give improvement indicators for designing new parallel PPIs and refine existing ones. As such, it is possible to continue increasing the abstractions of parallelism and create better and simple-to-use PPIs without compromising the performance of the applications. However, obtaining productivity indicators is time-consuming because empirical studies such as those presented in Sections 4.2 and 4.3 must be performed. Moreover, an experiment must be planned and executed, and finding a representative sample of participants in the parallel programming domain is quite a challenge, as seen in the initial study presented in Section 4.3. Coding metrics may not be the final solution, but they at least provide specific coding productivity insights that can be comparable in certain parallel programming scenarios and taken into account for usability analysis [31, 96, 188, 181, 172, 78, 170, 159, 82, 65].

Although coding metrics provide productivity indicators, they were designed for general purposes without considering factors that impact the development of parallel applications, such as characteristics of the PPI, the programming model, and the target architecture. We aimed to identify whether existing coding metrics provide valuable insights for the parallel programming domain, as well as highlight their limitations. Therefore, one of the research questions we aim to answer in this chapter is "*how effective are existing coding metrics for evaluating the coding productivity of parallel applications?*". Aiming to answer this question, we selected a set of metrics commonly used to evaluate the coding productivity of parallel applications, which we identified from a literature review presented in Chapter 3.

From the literature review, we identified three types of commonly used coding metrics, which are based on code size evaluation, complexity evaluation, and those that estimate development effort. In Section 5.1, we present a study to evaluate the feasibility of these coding metrics for assessing the coding productivity of parallel applications. This section is a slightly modified version of the paper [9][1], in [9] the interested reader can access the version in an article format.

Since we have identified metrics to estimate the effort and time required to develop applications, the other research question we aim to answer in this chapter is: "*How accurate are the effort estimates produced by existing metrics when evaluating parallel applications?*". Therefore, in Section 5.2, we aim to compare the development effort estimated by a set of coding metrics with the actual effort spent by the programmers to develop parallel applications. This section is a slightly modified version of the paper

[1]Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores, 2021 Euromicro Conference on Software Engineering and Advanced Applications (SEAA) - ®2021 IEEE

[12][2], in [12] the interested reader can access the version in an article format. Finally, Section 5.3 presents the final remarks.

5.1 Assessing coding metrics for parallel programming

Different offline coding metrics have been used in parallel programming to obtain productivity indicators. From the literature review (Section 3), we identified the metrics most commonly used by researchers in the parallel programming domain to evaluate programming productivity. There are metrics based on size code analysis (SLOC, NOC, and TOC), complexity evaluation (CCN), and development effort estimation (Halstead, COCOMO 81, and COCOMO II). Since these coding metrics are designed to evaluate general-purpose software development, we aim to identify whether they provide helpful information for the parallel programming domain and highlight their limitations and opportunities for improvement.

5.1.1 Methodology

In this study, we evaluated the use of SLOC, NOC, TOC, CCN, Information Flow Complexity (IFC), Halstead and COCOMO II coding metrics applied to parallel programming. Moreover, our evaluations did not consider COCOMO 81 because it is obsolete. To evaluate these metrics, our experiments aim to verify the productivity of FastFlow, Pthreads, SPar, and Intel TBB PPIs for multi-core systems in the development of the following C++ stream processing applications:

- **Bzip2:** It is an application designed for data compression and decompression in Bzip2 format [82];
- **Dedup:** It is a PARSEC application designed to compress data streams based on the deduplication method, where it combines local and global compression to achieve high compression ratios [82];
- **Denoiser:** It is a filter application designed to restore digital images with salt and pepper noise [6];
- **Person Recognition:** It is a video application designed to recognize people [81];
- **Lane Detection:** It is a video application designed for detecting road lanes on video feeds [81].

[2]Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing, 2022 Euromicro Conference on Software Engineering and Advanced Applications (SEAA) - ®2022 IEEE

We used SLOCCount[3] tool to get SLOC and COCOMO, Notepad++[4] to get NOC, Lizard[5] tool to get CCN and IFC, and Commented Code Detector (CCD)[6] to get TOC and Halstead's measures. To compare the estimated development time between Halstead and COCOMO II, we converted both measures to days needed to develop the applications. Moreover, IFC and CCN were measured as the sum of individual complexities from each function in the source code.

Table 5.1 presents the values of the cost drivers and scale factors used for COCOMO II. These values were chosen because the applications evaluated in this study are data compression and video processing applications, which require more than 95% of the main memory due to the large volume of data they process. The parallel codes use more than 95% of the available processors to achieve high performance. The development team has some experience developing these types of applications and has been able to use the evaluated PPIs effectively and efficiently. In addition, the programmers could develop

[3]Available at: https://dwheeler.com/sloccount/.
[4]Available at: https://notepad-plus-plus.org.
[5]Available at: http://www.lizard.ws/.
[6]Available at: https://github.com/dborowiec/commentedCodeDetector.

Table 5.1: COCOMO II cost drivers and scale factors.

Post Architecture Model			**Early Design Model**		
Name	**Option**	**Value**	**Name**	**Option**	**Value**
RELY	Low	0.92	PERS	Low	1.26
DATA	Very high	1.28	RCPX	Nominal	1
DOCU	Nominal	1.00	PDIF	High	1.29
CPLX	High	1.17	PREX	Very high	0.74
RUSE	Nominal	1	FCIL	Nominal	1
TIME	Extra high	1.63			
STOR	Extra high	1.43			
PVOL	Low	0.87	**Scale Factors**		
ACAP	Very High	0.71	**Name**	**Option**	**Value**
APEX	High	0.88	PREC	High	2.48
PCAP	Very high	0.76	FLEX	Nominal	3.04
PLEX	Very high	0.85	RESL	Nominal	4.24
LTEX	High	0.91	TEAM	Very high	1.1
PCON	Nominal	1	PMAT	Nominal	4.68
TOOL	Very high	0.78			
SITE	Nominal	1			
SCED	Nominal	1			

the applications without worrying about deadlines, since there were no deadlines for delivery to customers in this experimental study. For more details about the parameters of COCOMO II, see Section 2.5.1 of Chapter 2.

5.1.2 Results analysis

Table 5.2 presents the results for each application implemented using the target PPIs, and the sequential version. Our results showed that SPar presented the lowest number of SLOC, NOC, and TOC compared to the other PPIs for all applications evaluated. This is because the SPar's programming model requires only to insert annotations in the code without major changes. Compared to the sequential version, the SLOC value is only 5.80% greater for Dedup, 4.31% greater for Lane Detection, and 6.62% greater for Person Recognition. For Bzip2 and Denoiser, more code changes were required, where the SLOC value increased by 35.34% and 28.40%, respectively. Bzip2 is an application for compressing and decompressing data in the bzip format. This application required more modifications in the code since it was necessary to implement two pipelines for it, one for the compress stages (read, compress, and write) and one for the decompress stages (read, decompress, and write) [82]. Similarly, Bzip2 application also required a greater number of NOC and TOC than the other applications. In addition, NOC was not impacted by the developers' verbosity since it is not necessary to create any data structures to parallelize the application using SPar .

There was a minor difference between the SLOC, NOC, and TOC values when comparing the sequential version and the SPar version for Lane Detection and Person Recognition. These results occurred because there is only the main function in the sequential versions of the Lane Detection and Person Recognition applications. Given the programming model of SPar (annotation-based), it was not necessary to make many modifications to the codes to provide parallelism.

FastFlow and TBB showed similar results, as seen in Table 5.2. The greatest difference was for Dedup, where the SLOC, TOC, and NOC values are 38.75%, 35.40% and 36.25% lower for TBB. This is due to the structures required for building the Farm pattern in FastFlow. However, both PPIs have similar programming models. For both FastFlow and TBB, the application used similar data structures (`class`/`struct`) for each stage of the Farm pattern. In addition, communication among the stages is also performed in a similar way, where the current stage returns it as a pointer to be processed by the next stage after processing a task. This number could be improved if the application would be written using lambda functions [5, 244], but it does not prove better productivity.

As expected, Pthreads presented the worst result among the PPIs, with the highest value of SLOC, NOC, and TOC (Table 5.2). In the worst case, it achieved SLOC, NOC,

Table 5.2: Results on productivity and complexity.

		SLOC	NOC	TOC	CCN		IFC	
					Sum	Mean	Sum	Mean
Bzip2	Sequential	1327	35087	9592	288	24	1005	561.02
	FastFlow	1991	54071	14184	431	12.3	5341	145.04
	Pthreads	2312	61223	16199	473	21.5	6289	641.20
	SPar	1796	49270	13148	392	26.1	5118	800.11
	TBB	1868	51136	13732	404	13.9	7234	180.47
Dedup	Sequential	569	18346	3902	111	10.1	2960	59.41
	FastFlow	1027	32688	6833	192	5.3	3726	8.60
	Pthreads	1052	35243	7322	196	12.2	1164	139.69
	SPar	602	20261	4313	119	10.8	3088	62.28
	TBB	629	21121	4356	116	6.8	24	6.94
Denoiser	Sequential	176	7778	1590	28	5.6	30	11.55
	FastFlow	243	9699	2006	35	2.9	643	3.55
	SPar	226	9660	1942	34	3.4	792	42.34
	TBB	271	10718	2195	39	2.6	592	5.20
Lane Detection	Sequential	116	3344	979	13	13	0	0
	FastFlow	166	4384	1256	22	3.1	49	8.54
	Pthreads	360	9639	2187	41	3.2	888	23.33
	SPar	121	3496	1024	13	13	0	0
	TBB	168	4648	1331	22	2.4	49	6.53
Person Recognition	Sequential	136	5064	1172	18	9	0	3.66
	FastFlow	194	6586	1557	28	2.8	49	9.07
	Pthreads	393	11250	2476	46	3.3	888	23.03
	SPar	145	5687	1239	18	9	0	3.94
	TBB	193	6808	1560	25	2.8	49	7.27

and TOC values 171.03%, 175.71%, and 113.57% greater than SPar, respectively. This is because of its low-level programming model and programmers are required to explicitly implement and manage parallelism techniques, strategies, and mechanisms. The parallel stream processing applications execute as stream graphs composed of operators or stages and FIFO communication queues [5]. Since Pthreads is an unstructured PPI, the developers must implement this communication manually through chained queues, where it was necessary to manually create the mechanisms to order the insertion of the item into the chained queues.

Although program length is used as a predictor of maintainability and reability [31], it is not possible to predict how a parallel application will behave based on its length alone. There are other factors that directly impact the development effort of parallel applications. Relevant examples are programming model, architecture, and developers' experience. Each of these factors has its own particular characteristics that impact on the development effort differently. While experienced parallel programmers are more aware of the problems faced, novice parallel programmers may not know to follow the correct path for parallelizing the code. Moreover, code size alone does not guarantee that the application is concise and delivers better performance. Therefore, SLOC, NOC and TOC are only useful for measuring the manual exercised effort of developers if they are used together with other coding metrics.

Table 5.2 also presents the results of the CCN and IFC complexity metrics, in which the complexity of an application is evaluated from its representation in a flowchart (see Section 2.5.1 of Chapter 2). Our results show that the applications parallelized with SPar present the smaller CCN for Bzip2 (36.11% greater than sequential), Denoiser (21.43% greater than sequential), Lane Detection (same as sequential), and Person Recognition (same as sequential). Dedup parallelized with TBB has CCN 2.52% lower than SPar. It occurs because the CCN metric considers the annotations of SPar as nodes with possible paths. Similarly as with the length metrics, Pthreads shows the worst result for the CCN metric.

According to Felton et. al [62], a procedure or function with CCN value greater than 10 can be problematic. The results in Table 5.2 shows that the average CCN per function in the parallelized applications is highest for the Bzip2 and Dedup applications, which are data compression and decompression applications. If we consider the sum of the complexities of the functions of the application codes, all analyzed case obtained CCN in such situation. It confirms that parallel applications are more complicated to implement.

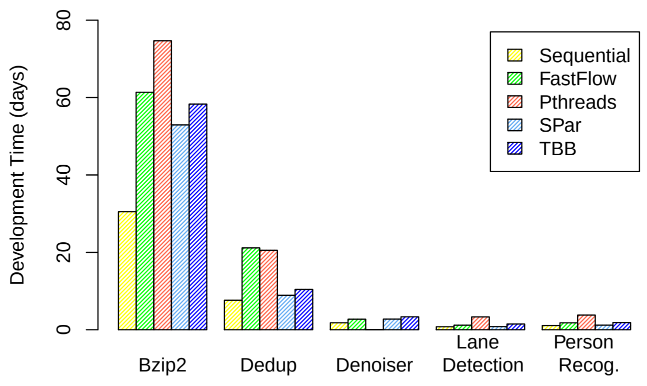
Considering IFC, the results regarding complexity are different from those obtained with CCN. For Bzip2, Lane Detection (same as sequential) and Person Recognition (same as sequential), SPar presented the smaller IFC. For Dedup, TBB presents the smaller IFC, which is 99.22% lower than SPar. For Denoiser, Fastflow presents the smaller IFC, which is 18.81% lower than SPar. Moreover, IFC presents results related to complexity equal to zero. It occurs Lane Detection and Person Recognition applications, in which the sequential application has only the main function. When parallelizing Lane Detection and Person Recognition, the application structure is maintained because it is not necessary to create any data structure to parallelize the application with SPar - we just need to insert the annotations in the code. If the program contains only the main function in the code, IFC is equal to zero even though there is a complexity in adding the parallel directives. In order to get overcome this limitation, we calculated the IFC by considering the average number of SLOC, fan-in, and fan-out.

Table 5.2 also presents the results for the average IFC. FastFlow presented the smaller IFC for Bzip2, Dedup, and Denoiser. On the other hand, SPar continued to show the lowest IFC for the Lane Detection and Person Recognition applications. However, SPar complexity continued equal to zero only for Lane Detection application using IFC metric. Moreover, IFC does not consider any parallelism factor when evaluating code. This metric only considers the code size and flow in each function of the application. Despite the results obtained, this metric did not prove to be as effective when evaluating parallel applications.

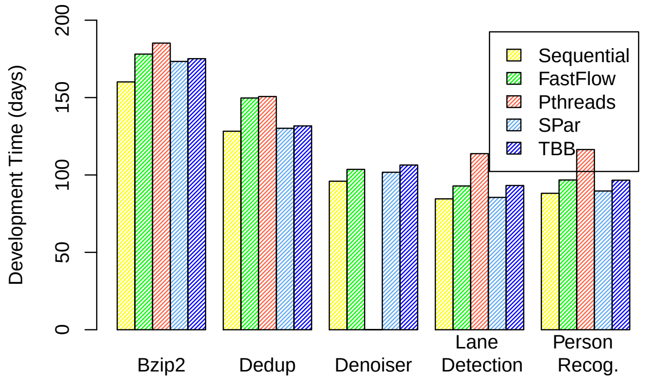
Figure 5.1a shows the results of Halstead's development time. Results show that SPar presents the smaller estimated development time for Bzip2, Dedup, Lane Detection, and Person Recognition. Denoiser implemented with Fastflow has a estimated development time 0.72% lower than SPar. Although very small, such difference is due to the way the programming difficulty is calculated in Eq. 2.11 presented in chapter 2. While the FastFlow version has more number of operands and total occurrences of operands than the version implemented with SPar, the estimated programming difficulty is lower for FastFlow. This may have occurred because the tool used to calculate the Halstead's measures does not consider the keywords of the PPIs such as operators. Thus, SPar and FastFlow presented the same number of operators ($n1 = 47$).

Fig. 5.1b presents the results of COCOMO II for each application implemented by the PPIs and the sequential version. Results shows that SPar presents the smaller estimated development time using COCOMO II for all the applications. As COCOMO II evaluates the development effort based on the SLOC and SPar presents the smallest SLOC for all applications, SPar then should present the smallest development time. On the other hand, Pthreads requires the larger times to develop a parallel application, as it has the greater SLOC value in comparison to the applications with other PPIs.

Although we converted the development time estimated by Halstead and COCOMO II to days, the metrics showed different results. Lane Detection required approximately 85 days to develop the SPar application, according to COCOMO II. On the other hand, according to Halstead, Lane Detection could be developed in around one day (≈ 99.52% lower). A complex application like this could not be developed from scratch in just one day, even if the developer had several years of experience in C++ and SPar. These results occur because the Halstead model does not consider essential aspects of software development, such as the developers' profiles. Hence, COCOMO II seems to be a more complete model than Halstead because COCOMO II considers some parameters in its evaluation, such as the project's complexity, the documentation, the experience and skill of the developer,



(a) Halstead.



(b) COCOMO II.

Figure 5.1: Results of development time estimated using Halstead and COCOMO II metrics.

and others. However, it was not developed to evaluate the development of parallel applications. In addition, COCOMO II considers that the application will be implemented from scratch disregarding the insertion of parallel directives in the code. Therefore, further investigation of this metric is necessary.

5.1.3 Discussion and threats to validity

In this study, we investigated and assessed different coding metrics applied to parallel programming of stream processing applications on multi-cores systems, considering different well-known PPIs. From these PPIs, SPar showed the best results in the most coding metrics used. We conclude that the values of SLOC, NOC and TOC tends to be lower with the SPar programming model since it requires minor code changes due to its high-level abstraction. CCN has been more effective than IFC for measuring the complexity of a parallel program since none of the metrics consider parallel directives. However, both proved to be limited when evaluating parallel applications because, in some cases, the complexity of the parallel applications was lower than the complexity of the sequential application. A possible cause of the CCN and IFC results is how these metrics evaluate the complexity of applications, which should be represented as a flow graph.

Halstead's measures are widely used in the parallel programming literature [65, 125, 130, 143, 146, 164, 184, 192, 196, 194, 195, 209]. This metric can help determine code size (TOC), programming difficulty, and development effort. This metric is based only on analyzing operators and operands in the code to perform its estimations. However, there is yet to be a consensus on what should be considered an operator or operand in a code. In addition, there is yet to be a consensus on whether the keywords of parallel programming models should be evaluated as code operators. However, further analysis of the Halstead metric is required.

COCOMO II proved to be the most promising metric for evaluating the productivity of PPIs. Unlike other metrics, COCOMO II has a collection of cost drivers and scale factors to be calibrated according to the software development cycle in order to estimate the programming effort. Moreover, only the CPLX cost driver considers the evaluation of task synchronization operations and complex parallel computing operations. There are other COCOMO II cost evaluation factors as seen in Chapter 2, such as the skills of the programmers (PCAP), the programmer's experience in developing applications similar to the target application (APEX), the programmer's knowledge of the target architecture (PLEX), and the programmer's expertise in using the target languages and tools (PLEX). However, COCOMO II considers that the application will be implemented from scratch. There are other COCOMO II variations called reuse and maintenance models presented in Chapter 2. COCOMO II reuse model was designed to evaluate development from existing applications. While the maintenance model was build to estimate the effort required to implement improvements or corrections to already developed software. In the literature review, we identified only one study using the COCOMO II reuse model to evaluate GPU programming effort [177]. Therefore, it is still necessary to investigate the use of these COCOMO variations in parallel application evaluation.

Despite the results provided in this section, some threats to validity remain. The learning effect can threaten internal validity because we specified no order for the use of the PPIs evaluated. Another threat to internal validity relates to instrumentation, such as using the CCD tool, which does not recognize PPI keywords as operators. The study design can be a threat to construction validity because FastFlow, SPar, and TBB are pattern-based PPIs, unlike Pthreads. Finally, a threat to the conclusion validity is the size of the applications evaluated, although they are real-world stream processing applications.

5.2 Productivity Estimation with Cross Analysis

In the previous section, the results obtained showed that while code metrics based on code size and complexity can be helpful for evaluating PPIs, it is impossible to predict the effort required to develop a parallel application based on these factors alone. Other factors influence the parallel development cycle, such as the development environment, and developer experience. Thus, Halstead and COCOMO II showed more promise for evaluating PPIs, although they also have limitations. In this section, we presented a initial efforts to overcome some of these limitations. We proposed an approach to evaluate parallel applications using Halstead and a refined version of COCOMO II reuse model. There are other predictive metrics that, to our knowledge, have not yet been used to evaluate parallel applications: FPA, Planning Poker, Putnam, SEER-SEM, and UCP. Therefore, the goal of this study is to evaluate the *accuracy* of such metrics compared to the actual effort required to develop parallel stream applications using FastFlow, SPar, and TBB.

5.2.1 Development Effort Metrics for Parallel Programming

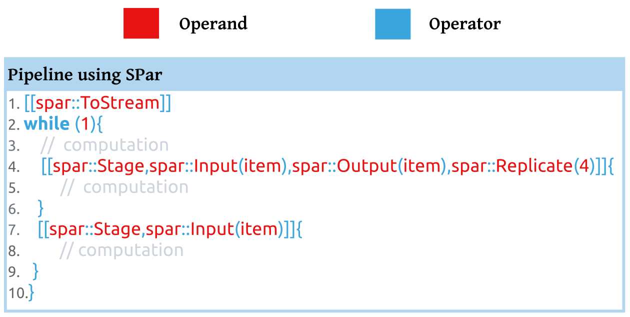
Parallel Halstead

In the Section 5.1, we used the CCD[7] tool to obtain Halstead's measures [9]. This tool allows the analysis of code written in the C, C++, and Java programming languages. However, we identified some limitations while using it to evaluate parallel stream processing applications developed in C++. There is a lack of user support since CCD has not

[7]Available at: https://github.com/dborowiec/commentedCodeDetector.

received updates since 2014. This is a CCD limitation because the C++ language has received two update versions since 2014 (C++17 and C++20), in which new attributes and operators have been added. In addition, the CCD tool does not consider any of the PPIs's keywords as operators (e.g., `spar`, `ff_node`, `tbb`, and `pipeline`), because its focus is not on evaluating parallel applications. Figure 5.2a illustrates the operand and operator count performed by the CCD tool for the SPar interface, where all SPar keywords are considered as operands (in red). However, since C++ keywords can be considered as operators [53], we also consider the keywords of the PPIs as operators.

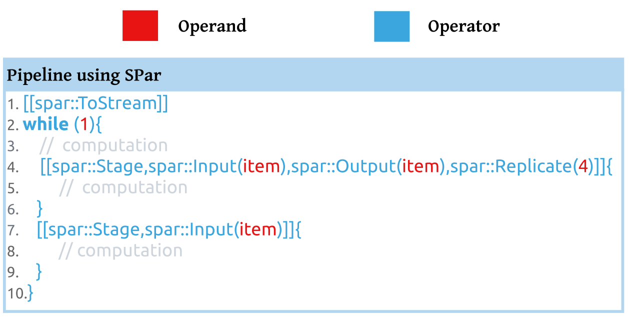
From the literature review, we have identified 14 studies using Halstead's measures to evaluate PPIs. However, only two of them cited the tools used for this purpose. CMetrics[8] tool was used by [192, 184], which is a tool to measure SLOC, CCN, and Halstead development effort in applications developed with C programming language. There are other more recent tools for getting Halstead measurements on code written in C++, such as Testwell CMT++[9] and IBM Rational Test RealTime[10]. However, like the CCD tool, these other tools were not developed to evaluate parallel code. To overcome these limitations, we developed the *Parallel Halstead (PHalstead)*[11] tool, a Python script to obtain Halstead's measures in C++ applications parallelized with FastFlow, SPar, TBB. Figure 5.2b illustrates the operands and operators count performed by the CCD tool for the SPar, where all SPar keywords are considered as operators (in blue), as well as the C/C++ keywords. In addition, the current version allows the analysis of Java code parallelized using Flink and Storm, and C/C++ applications parllelized with GrPPI, OpenMP, and C++ threads.



Pipeline using SPar

```
[[spar::ToStream]]
while (1){
  // computation
  [[spar::Stage,spar::Input(item),spar::Output(item),spar::Replicate(4)]]{
     // computation
  }
  [[spar::Stage,spar::Input(item)]]{
     // computation
  }
}
```

(a) Operators and operands from Halstead metric.



Pipeline using SPar

```
[[spar::ToStream]]
while (1){
  // computation
  [[spar::Stage,spar::Input(item),spar::Output(item),spar::Replicate(4)]]{
     // computation
  }
  [[spar::Stage,spar::Input(item)]]{
     // computation
  }
}
```

(b) Operators and operands from PHalstead metric.

Figure 5.2: Difference between Halstead and PHalstead.

Parallel COCOMO II Reuse Model

From the results presented in the Section 5.1, we identified COCOMO II as a good metric for evaluating parallel applications [9]. However, some of its parameters are not

[8]Available at: https://github.com/MetricsGrimoire/CMetrics.
[9]Available at: http://www.verifysoft.com/en.html.
[10]Available at: https://help.blueproddoc.com/rationaltest/rationaltestrealtime/8.3.0/index.html.
[11]Available at: https://github.com/GMAP/phalstead .

usually applied in the development cycle of parallel applications. Some efforts have been made by Wienke *et al.* [248] in order to extend COCOMO II to evaluate parallel applications. However, this model is challenging to apply in practice because it uses linear regression and the dataset used was not available. In addition, it is not easy to translate the parameters of COCOMO II to the parallel programming scenario [159].

In this study, we also make an initial effort to refine the COCOMO II reuse model. Initially, we identified some scaling factors without impacting the development cycle of parallel applications. The scale factors identified were the following:

- **RESL:** evaluates whether there is good support for resolving risks;
- **TEAM:** evaluates the cohesion of the development team;
- **PMAT:** evaluates the CMMI maturity level.

.

To do our modification, we removed these three scale factors from the Eq. 2.23, which is used to generate the S scaling exponent. In other words, the new set of scale factors will consist only of the following scale factors:

$$F^{new} = \{PREC, FLEX\} \tag{5.1}$$

. Consequently, the new equation to calculated the S scaling exponent will be:

$$\sum_{j=1}^{5} F_j^{new} = PREC \times FLEX \tag{5.2}$$

$$S = B + 0.001 \times \sum_{j=1}^{5} F_j^{new} \tag{5.3}$$

Since the reuse model applies only the cost drivers of the post-architecture model, we also identified some of these cost drivers that are not relevant in parallel application development:

- **RUSE:** evaluates whether the project is designed to generate components to be reused;
- **ACAP:** evaluates the software analysts' ability to analyze and model applications, efficiency, and effectiveness;
- **PCON:** evaluates the percentage of developer changes in a one year.

To do this modification, we removed the three cost drivers identified from the multiplication performed in Eq. 2.24. Thus, the new set of cost drivers M to be used is:

$$\begin{gathered} M^{new} = \{RELY, CPLX, DATA, DOCU, TIME, STOR, PVOL, \\ ACAP, PCAP, APEX, PLEX, LTEX, TOOL, SITE, SCED\} \end{gathered} \tag{5.4}$$

Consequently, the new equation to calculate the development effort will be:

$$\prod_{i=1}^{n} M_i^{new} = RELY \times CPLX \times ... \times SCED \tag{5.5}$$

$$Effort = A \times KSLOC^S \times \prod_{i=1}^{n} M_i^{new} \tag{5.6}$$

We refer to this modification of the COCOMO II Reuse model, which does not include these cost drivers and scale factors in its evaluations, as the *Parallel COCOMO II Reuse Model (PCRM)*.

5.2.2 Methodology

We aimed to evaluated the accuracy of development effort estimation models applied to the parallel programming domain: Putnam's model, Halstead's measures, PHalstead, COCOMO II using early design and post-architecture models together, COCOMO II using post-architecture model alone, COCOMO II reuse model, COCOMO maintenance model, PCRM, FPA, UCP, SEER-SEM, and Planning Poker. For the COCOMO II, we did not use the early design model alone because the applications developed are not in the design phase since the architecture was already defined (multi-core).

We used the following tools to measure the code metrics: CCD to get Halstead, PHalstead[12] to get Halstead, SLOCCount to get COCOMO II variations, Function Point Calculator[13] to get FPA, Use Case Point Calculator[14] to get UCP, SEER-SEM 8.4[15] trial version, and a spreadsheet for the other metrics. We used the modified Fibonacci sequence (0, 1/2, 1, 3, 5, 8, ...) for the Planning Poker method to perform the estimation. The Planning Poker estimates were obtained by averaging the guesses of three stream processing experts. In addition, for Putnam's model we considered a development environment with adequate

[12]Available at: https://github.com/GMAP/phalstead.

[13]Available at: https://w3.cs.jmu.edu/bernstdh/web/common/webapps/oop/fpcalculator/FunctionPointCalculator.html

[14]Available at: http://groups.umd.umich.edu/cis/tinytools/cis375/f17/team9-use-case-pts/Use_Case_Point_Calculator/

[15]Available at: https://galorath.com/seer-for-software/.

documentation and reviews (8000), and and there is no use of automated development tools and techniques.

We applied each metric described above to estimate the time required to develop the RGB channel extraction application, described in the Section 4.2.1. To compare the results, we converted the times estimated by each of the metrics to development hours. According to Boehm *et al.* [26], we considered 152 working hours per month already excluding weekends and holidays. To evaluate the accuracy, we used the data of the experiment conducted with beginners in parallel programming presented in Section 4.2. Therefore, we compared the actual development time with that estimated by each model. For this purpose, we used MMRE and PRED accuracy metrics [198].

5.2.3 Results analysis

Table 5.3 presents the average of SLOC, ASLOC and MSLOC for the RGB channel extraction applications developed using FastFlow, SPar, and TBB. Our results showed that SPar presented the lowest number of SLOC. Compared to the sequential application, the SLOC value is only 6.66% greater for SPar. As seen in this table, SPar requires adding only five lines (average) to explore the parallelism. In addition, only two lines (average) were modified in the original application to parallelize it using SPar. These results occurred because of the high-level abstraction provided by SPar's annotation-based programming model. On the other hand, for FastFlow and TBB more code changes were required, where the SLOC value increased by 44% and 57.33%, respectively.

Table 5.3 also shows the average time taken by each of the 15 participants to implement stream parallelism. These results showed that SPar requires less effort to develop parallel stream processing applications than the other PPIs (see Section 4.2). Table 5.3 also shows the average development times estimated by each of the models used in this study. To verify the accuracy, we converted the development times estimated by each metric into development hours. This time represents the average estimated development time from the applications developed by the participants using FastFlow, SPar, and TBB.

The Planning Poker metric got the best result than the other estimation metrics. However, the estimated value is considered acceptable only for FastFlow and TBB. Table 5.4 shows the MMRE, MdMRE, and PRED for these results. According to Port and Korte [198], MMRE and MdMRE less than or equal to 0.25, and PRED greater than or equal to 0.75 are values considered an acceptable accuracy level for models and effort estimation [198] (see Section 2.5.2 of Chapter 2). The estimated development times for SPar are close to the actual times, although they do not meet the accuracy criteria. These results occurred because we obtained Planning Poker estimations from the opinions of three programmers experienced in developing stream processing applications. This also explains

Table 5.3: Results obtained for the number of SLOC (total, added and modified), as well as the actual development time (in hours) and the estimated development times for each of the metrics evaluated (in hours).

Code Metric	**Sequential**	**FastFlow**	**SPar**	**TBB**
SLOC	75	108	80	118
ASLOC	-	33	5	43
MSLOC	-	21	2	20
Actual development time	-	1.72	1.01	2.17
COCOMO II (post-archit.)	264.34	570.44	525.75	582.90
COCOMO II (early and post-archit.)	198.24	550.12	507.74	562.11
COCOMO II maintenance model	-	506.81	287.41	526.80
COCOMO II reuse model	-	428.50	253.15	454.30
Parallel COCOMO II reuse model	-	447.32	267.69	473.57
FPA	-	127.54	101.62	137.32
PHalstead	-	12.96	9.18	13.74
Planning Poker	-	1.00	0.50	2.00
Putnam’s model	-	195.06	171.20	202.05
SEER-SEM	31.00	58.07	42.60	63.33
UCP	-	162.06	129.72	175.14

why the estimated time for FastFlow and SPar was shorter than that of beginner developers. Since the experts are already used to developing this type of application, they thought they would need less effort to develop the evaluated video processing application.

The results show that PHalstead’s development effort was the second-best result, although it does not meet the accuracy criteria (MMRE and MdMRE $\geq$ 5.59 and PRED = 0). This metric considers only the number of tokens in its evaluation without considering any factors that impact the development effort of parallel applications. Therefore, this metric can help measure code size, just as the SLOC.

The FPA and UCP models showed similar results, where both did not meet the accuracy criteria (MMRE and MdMRE $\geq$ 67.92 and PRED = 0). This behavior is because these models were designed to evaluate user interaction systems in which there is an interface where users enter, delete, and query data. They also do not consider any aspect of the programming language used, so they are not suitable for evaluating parallel applications.

Putnam’s model showed results close to FPA and UCP (MMRE and MdMRE $\geq$ 100.26 and PRED = 0). Despite this, the evaluation of Putnam’s model differs from the FPA and UCP models. It uses the Productivity Parameter (PP) obtained through development effort, development time, and SLOC of previously developed applications. However, we use the PP value suggested by Pressman [200], which was applied in the Equation 2.7 to estimate the minimum time required to develop the target application. The PP value may have been a factor that increased the estimated development time. If we consider an ideal development environment (PP equal to 11000) the development time reduces by about 13%: 149.29 hours for SPar, 170.10 hours for FastFlow, and 176.20 hours for TBB.

Table 5.4: Accuracy results for the development times estimated by each of the code metrics evaluated using MMRE, MdMRE, and PRED metrics.

	FastFlow			**SPar**			**TBB**		
	MMRE	MdMRE	PRED	MMRE	MdMRE	PRED	MMRE	MdMRE	PRED
COCOMO II (post-archit.)	533.68	420.49	0.00	600.63	573.93	0.00	442.27	297.98	0.00
COCOMO II (early and post-archit.)	511.71	388.83	0.00	577.85	572.82	0.00	427.99	283.84	0.00
COCOMO II maintenance model	472.12	374.99	0.00	330.96	319.74	0.00	398.90	268.42	0.00
COCOMO II reuse model	398.97	321.54	0.00	291.19	286.27	0.00	344.74	229.29	0.00
Parallel COCOMO II reuse model	416.46	342.60	0.00	307.65	302.07	0.00	358.94	238.50	0.00
FPA	133.02	96.57	0.00	115.25	107.72	0.00	102.61	67.92	0.00
PHalstead	11.11	10.04	0.00	9.54	8.37	0.00	9.61	5.59	0.00
Planning Poker	0.53	0.71	**1.00**	0.50	0.50	0.00	0.31	**0.08**	**2.00**
Putnam’s model	182.61	146.93	0.00	195.10	182.30	0.00	151.74	100.26	0.00
SEER-SEM	53.38	46.82	0.00	47.79	45.01	0.00	46.97	30.62	0.00
UCP	169.29	122.98	0.00	147.40	137.78	0.00	116.07	86.90	0.00

However, the actual development time of the RGB channel extraction application remains about 99% shorter than the values estimated by Putnam’s model.

The worst results were observed with the traditional COCOMO II, using the cost drivers of the post-architecture model and early design and post-architecture models together. The conventional COCOMO II estimates the effort required to develop an application from scratch [9]. Therefore, we already expected that these results would be higher than the actual ones (MMRE and MdMRE $\geq$ 283.84 and PRED = 0). As seen in Section 2.5.1, the original COCOMO II model evaluates the effort to develop software from scratch. Therefore, we used the COCOMO II maintenance and reuse models to address this limitation because the developers implemented the parallel applications from a sequential application rather than from scratch.

The COCOMO II reuse model showed better results compared to the maintenance model. There is still a big difference between the development time estimated by the reuse model and the actual development time (MMRE and MdMRE $\geq$ 229.29 and PRED = 0). Therefore, we refined the reuse model, removing the cost drivers and scale factors that do not apply to parallel application development. However, this increased development time because adapting existing models to the parallel programming domain is a complex task. It is not possible to just remove some of their parameters.

The SEER-SEM method considers several factors that impact the software development cycle in its evaluation, such as complexity and effective technology. SEER-SEM also uses a database of already developed software to calibrate its parameters, requiring further analysis. There are video processing applications in its database, but not parallel applications. Refrain from considering such applications in its evaluations may be one of the reasons why the model also fails to estimate a development time close to the real one (MMRE and MdMRE $\geq$ 30.62 and PRED = 0). In addition, this is a proprietary tool, making it difficult to adapt for parallel programming evaluation.

5.2.4 Discussion and threats to validity

Our results showed that Planning Poker showed the best results because it relies on the experts' opinions to guess the development effort. However, finding professionals to apply this method in practice can be challenging. On the other hand, using PHalstead can be an alternative for studies where it is impossible to conduct controlled experiments with students and parallel application developers. PHalstead has proven to be a helpful metric for evaluating parallel code, although it does not get the best results. The current PHalstead version considers other PPIs, such as GrPPI, OpenMP, C++ threads, Flink, and Storn. In the future, we aim to extend PHalstead to consider other PPIs, such as Pthreads for multi-core programming and CUDA, OpenCL, OpenACC, and GSParLib for heterogeneous programming.

Putnam's model has yet to prove a suitable method for estimating the effort to develop parallel applications, which use data from previously developed applications to calibrate its PP parameter. Our results showed that a better development environment tends to increase the PP value, and consequently reduce the estimated development time. However, the development time estimated by Putnam's model continued at about 99%. These results may have occurred due to the PP value used, which was established by Pressman [200]. In their original model, Putnam and Myers [201] suggested deriving the PP value from the development time, effort, and SLOC of previously developed projects. However, the development effort also depends on the PP value to be estimated (Equation 2.6). Therefore, we chose to estimate the minimum development time in this study, which does not consider the development effort in its equation (Equation 2.7). Deriving the PP value from other projects can be a challenge since there is no available dataset composed of such data. In addition, the main Putnam's limitation is to evaluate the development time of an application based on the code size [215], which analysis performs better for larger projects [141].

SEER-SEM uses a similar technique to calibrate its parameters, although this method did not show the best result. Several public domain data sets are available in the software engineering area to evaluate the effort estimation models. However, no such data set is available in the parallel programming domain, making it difficult to use and evaluate these techniques. A data set is also important to make it easier for researchers in the parallel programming area to propose new methods for estimating development effort. This avoids the need to perform further experiments to validate these metrics, which is a time-consuming process.

Our results showed that the actual time to develop an RGB channel extraction application using SPar, FastFlow and TBB is about 99.6% less than the values estimated by the different variations of COCOMO II evaluated. Since the parallel applications evaluated were developed from sequential applications, we evaluated the use of the COCOMO II reuse and maintenance models. However, the actual development time remained about 99.6% less than the values estimated by reuse and maintenance model. Since the development cycle of parallel applications could be one of the reasons for these results, we have identified some cost drivers and scale factors without impacting the development cycle of parallel applications. Despite our efforts, our adaptation of the COCOMO II reuse model (PCRM) was not suitable for parallelization using high-level PPIs since the estimated effort is much higher than the actual effort. Moreover, refining its parallel application development scenario parameters is difficult. Therefore, creating a parallelism-sensitive model (e.g. a COCOMO extension) to evaluate applications in this domain is necessary since its development involves factors that are not addressed by the models considered in this chapter.

This study has some threats to validity. The learning effect is a threat to internal validity because of the order in which the participants use the PPIs. Therefore, the time spent by the participants developing applications using FastFlow may have been affected since two of the three groups already knew the target problem before using it. In addition, there is no control group since the three groups used all three PPMs (construct validity). The study design is a threat to construct validity because SPar is an annotation-based PPI, unlike FastFlow and TBB. The application size is another threat to construct validity, although it is a standard application in real-world stream processing. In addition, the sample size of 15 participants and the participants' experience level are threats to the conclusion validity. Therefore, it is not possible to generalize the results.

5.3 Final remarks

In this chapter, we aimed to evaluate the feasibility and accuracy of coding metrics for assessing productivity when applied to the parallel programming domain. Initially, we presented a study aimed at evaluating the feasibility of coding metrics commonly used to assess productivity in parallel programming: SLOC, TOC, NOC, CCN, IFC, Halstead, and

COCOMO II. Our results showed that while code metrics based on code size and complexity can be helpful for evaluating PPIs, it is difficult to predict the effort required to develop a parallel application based on these factors alone. On the other hand, Halstead and COCOMO II showed promise for evaluating parallel applications development, although they also have some limitations. Therefore, we proposed variations of these metrics to overcome some of these limitations, called PHalstead and PCRM.

To evaluate the proposed modifications, we conducted a second study to determine their accuracy against the actual effort required to develop parallel stream applications using FastFlow, SPar, and TBB. In addition, we compared the results with other predictive metrics that have not yet been used to evaluate parallel applications: FPA, Planning Poker, Putnam, SEER-SEM, and UCP. Our results showed that Planning Poker was an approach that showed the most promising results among the metrics evaluated. This metric showed the best development time estimation results over the other estimation metrics evaluated. The Planning Poker accuracy was because its estimates were derived from experts' opinions in parallel stream processing applications. In addition, these results showed that the developer's expertise can directly influence the development effort.

PHalstead has proven a helpful metric for evaluating parallel code, although it does not get the best results. Our results showed that this metric could be a simpler alternative for estimating the development time of parallel applications because it is based only on source code analysis. Unlike Planning Poker, it is not necessary to consider the opinion of other developers to use PHalstead. Moreover, it is simple since the user should only inform the source code and the interface targeted for analysis as input to this metric.

PCRM proposed was not suitable for parallelization using high-level PPIs since the estimated effort is much greater than the actual effort. Moreover, refining the COCOMO II parameters to evaluate the development effort of parallel applications is a challenge. Therefore, creating a parallelism-sensitive model to evaluate applications in this domain is necessary since its development involves factors that are not addressed by the models considered in this chapter. Such factors may include programmer experience, architecture, and programming model [248]. Performance of applications is another idea from Wienke *et al.* [248] that can be included in these factors as well as the number of activities that will be executed simultaneously or even the amount of data shared and accessed concomitantly between threads. Therefore, in the next chapter, we will survey parallel application developers in order to identify the factors that most impact the development effort of such applications. By identifying these factors, we can propose and design a new model to estimate the effort of parallel application development.

6. PRODUCTIVITY SURVEY RESEARCH

In the last chapter, we discussed the limitations of standard offline coding metrics for evaluating the programming productivity of parallel applications. From the preliminary investigation presented in Section 5.1, we can conclude that the assessed metrics are not suitable for evaluating the usability of parallel applications. However, these metrics can be improved and used in parallel programming. SLOC, NOC, and TOC are valuable metrics for measuring code size. However, they have limitations in providing productivity indicators in developing parallel applications because the development of this type of application involves other factors besides the code size. Similarly, CCN and IFC are not effective metrics for measuring the complexity of parallel programs because none of these metrics consider the complexity of the parallel directives. Therefore, in Section 5.2, we proposed some modifications to Halstead and COCOMO metrics because we identified some potential to evaluate parallel applications, which were called PHalstead and PCRM. PHalstead has proven to be a helpful metric for evaluating parallel code than PCRM, although it does not get the best results compared with Planning Poker. Despite our efforts, the PCRM was not suitable for parallelization using high-level PPIs since the estimated effort is much higher than the actual effort. Moreover, it is not easy to refine its parameters for the parallel application development scenario.

Creating a parallelism-sensitive model to evaluate applications in the parallel programming domain is necessary since its development involves factors that are not addressed by the models considered in the last chapter. Relevant examples are the PPI used, the programming model, and the target architecture because each of these factors has its own particular characteristics that influence the development effort differently. For example, programming with Pthreads requires several subroutines for threads management, synchronization, and mutex [28]. On the other hand, in OpenMP, there are directives to control the distribution of tasks among threads [149]. In structured parallel programming, there are concerns that are not considered in non-structured parallel programming. For example, a developer must be aware of the communication between data channels where the output of one stage feeds the input of the next stage [5]. In addition, the scheduling on multi-core CPU is easier, due to flexible PPIs that bind the threads to a specific core. However, on the many-core GPU this task is more complicated due to the hierarchy of threads, in which some threads compose a warp, some warps compose a block of threads, and some blocks of threads compose a grid [253].

In this context, this chapter presents an international survey research to identify “*what are the factors impacting the coding productivity of parallel applications?*”. In parallel programming, different surveys and interviews have already been conducted to determine the community’s opinion on a specific issue. Meade *et al.* [154] conducted an exploratory study and a survey to investigate the tools and practices used by experts to perform data decomposition when parallelizing applications. Meade *et al.* [153] also conducted a multi-method empirical study (interview study, participant-observer case study, focus group study, and a sample survey) to understand the task of data decomposition as part of applications parallelization and identify the main requirements for tools to assist developers in this task. In [138], a survey was conducted to discover the energy and power-aware job scheduling and resource management techniques used in nine HPC centers in the United States, Europe, and Asia. Amaral *et al.* [7] conducted a survey with experts to evaluate the mapping study results that address High-Performance Modeling and Simulation for Big Data applications. Ferdinandy *et al.* [63] conducted a survey with researchers not in the Information and Communication Technology field to identify the main advantages and drawbacks these researchers face when having their first contact with HPC.

Lynn *et al.* [133] identified the determinants of cloud computing adoption for HPC by surveying 121 HPC decision-makers worldwide. Schlagkamp and Renker [219] used a questionnaire survey to investigate cluster users' satisfaction regarding the waiting time required to execute parallel applications. Schlagkamp *et al.* [218] also investigated several influences on the work behavior of computer cluster users based on the Computer Cluster User Habits Questionnaire (QUHCC), including the impact of slow responses on work times, strategies for dealing with high contention and poor performance, user experience, and user satisfaction. In [23], a survey was conducted to identify the use of MPI among applications and software technology efforts in the United States Exascale Computing Project. Hori *et al.* [102] conducted an international online questionnaire survey to analyze the adoption of MPI in parallel application development.

Regarding parallel programming productivity, Danis *et al.* [47] interviewed tool developers to identify the tools that offered the most significant potential to increase programmer productivity. Based on these interviews, Danis *et al.* [47] created a spreadsheet related to tools, availability, users, and workflows to guide the development of their CM approach (presented in Chapter 3), which is based on the number of steps to complete a given task, the number of context changes, and the working memory load required at each step. Wienke *et al.* [248] performed a series of surveys with experienced developers to identify the factors that most impact the development of parallel applications. From the survey results, Wienke *et al.* [248] made an initial effort to propose a model to estimate the effort required to develop parallel applications based on COCOMO II (presented in the Chapter 3). Although the results of Wienke *et al.* [248] provided valuable insights into parallel productivity, the COCOMO II extension is only a conceptual model and cannot be used in practice. Therefore, creating a parallelism-sensitive model to evaluate applications in this domain is still necessary.

Our survey also focuses on identifying factors that impact the productivity of parallel application development. However, differently from Wienke *et al.* [248], we also aim to discover "*what can be done to increase developer productivity?*". In addition, we also aim to find the PPIs that can increase the productivity of novice and experienced developers. This chapter is an updated version of the paper [10][1], in [10] we considered only the Brazilians' opinion. In addition, this survey was approved by the PUCRS research ethics committee with CAAE number 52635421.3.0000.5336.

Section 6.1 discusses the methodology applied to the survey in detail. Next, Section 6.2 presents the quantitative and qualitative results. Finally, Section 6.3 presents the final remarks.

6.1 Research method

From survey research, it is possible to statistically quantify the opinion of a certain population [72]. We aimed to discover the factors that prevent the development of parallel applications productively from the developers' opinion. For this purpose, our survey was built from three stages: planning, execution, and analysis of the results. Each of the stages is detailed below. In addition, the study was approved by the PUCRS research ethics committee.

6.1.1 Planning

The first stage of the survey research was the planning phase. To define the research instrument we used the Goal Question Metric (GQM) approach [29], which is a measurement system for planning which metrics will be used to interpret the data based on a set of questions and specific goals. GQM is a hierarchy with three levels, as shown in Figure 6.1: measurement goals (G), questions (Q), and evaluation metrics (M). In this survey, three goals were defined based on this approach:

- **G1:** Identify the factors that impact the development effort of parallel applications;
- **G2:** Profile parallel application developers;
- **G3:** Classify the parallel applications developed.

Table 6.1 presents the 18 questions derived from the goals. The questionnaire have two types of questions: closed-ended questions (Q1 to Q13), and open-ended questions (Q14 to Q18). Closed-ended questions were classified as single answer (Q1 to Q6),

[1]Opinião de Brasileiros Sobre a Produtividade no Desenvolvimento de Aplicações Paralelas, 2022 Symposium on High Performance Computing Systems (WSCAD) - ®2022 SBC

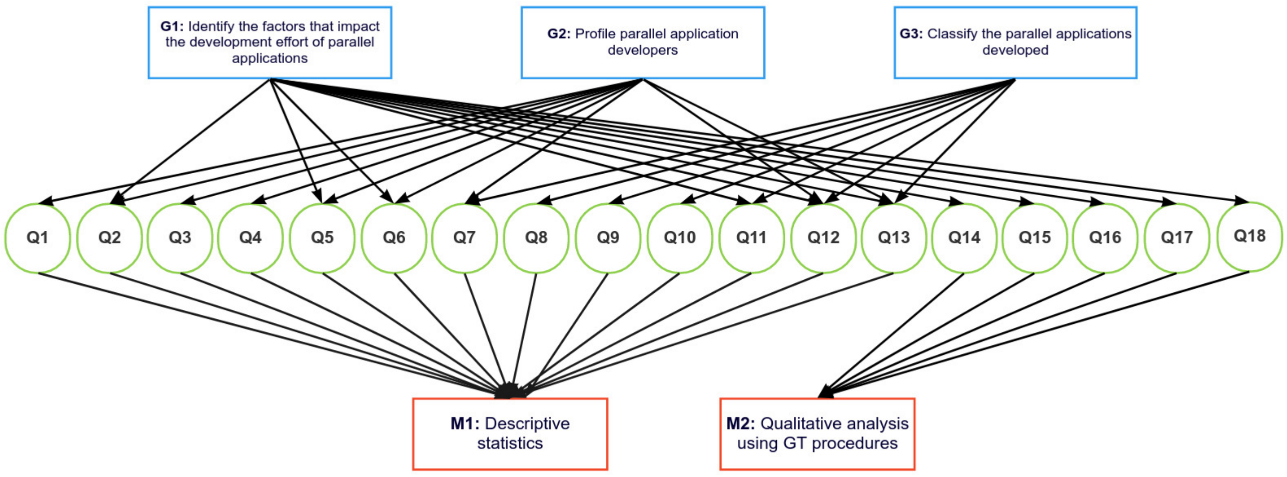

G1: Identify the factors that impact the development effort of parallel applications
G2: Profile parallel application developers
G3: Classify the parallel applications developed
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9
Q10
Q11
Q12
Q13
Q14
Q15
Q16
Q17
Q18
M1: Descriptive statistics
M2: Qualitative analysis using GT procedures

Figure 6.1: GQM approach used.

and multiple choice questions (Q7 to Q13). These questions were used to facilitate reading and synthesize the information required to characterize the participants and the applications developed by them. On the other hand, the open-ended questions allowed the participants to describe their opinions and experiences regarding parallel application development productivity. In addition, only Q18 was not mandatory.

The survey questionnaire was constructed using the Google Forms platform. The questionnaire was divided into four sections. The first section has a brief presentation of the survey context, and the term of free and informed consent. The second section presents the closed questions for the characterization of the participants and the third section presents the closed questions for the characterization of the developed applications. Finally, the fourth section presents the open questions related to the productivity of developing parallel applications.

A pilot study was conducted with eight participants before making the questionnaire available. These responses served to evaluate and refine the questionnaire and were not included in the final result. The questionnaire was then sent to the PUCRS Research Ethics Committee for evaluation. After the approval of the research by the ethics committee we started the selection of the participants.

Initially, we identified the study's target population from their attendance at HPC conferences, such as Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD), Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD), and Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP). The participants were also identified by analyzing the editorial board of journals in the HPC area, such as the Journal of Parallel and Distributed Computing, IEEE Transactions on Parallel and Distributed Systems, International Journal of Parallel of Programming, and ACM Transactions on Parallel Computing. Moreover, the participants' Google Scholar[2] profiles were also evaluated to select the participants.

[2]Available at: https://scholar.google.com.br

Table 6.1: Survey questionnaire.

Section	ID	Question
Participants' characterization	**Q1.**	What is your nationality?
	Q2.	What is your academic background? (High school, Technical course, Undergraduate, Specialization, Master's degree, or Ph.D. degree)
	Q3.	What is your affiliation (company or educational institution)?
	Q4.	In what country is your affiliation located?
	Q5.	What is your experience with parallel programming (in years)? (From 0 to 1, from 1 to 2, from 2 to 5, from 5 to 10, or more than 10 years)
	Q6.	What type of developer do you consider yourself? (Beginner, Intermediate, Expert, or Other)
Applications' characterization	**Q7.**	What type of appplications do you develop?
	Q8.	What is the domain of these applications?
	Q9.	Can you specify the applications' names or describe what the applications do?
	Q10.	How do you develop the applications?
	Q11.	In what are the programming languages in which the applications are developed?
	Q12.	What PPIs or tools do you use to parallelize applications?
	Q13.	In what architectures do you implement parallelism in your applications?
Productivity features	**Q14.**	In your opinion, what affects the productivity of parallel application development?
	Q15.	In your opinion, what can be done to make parallel programming more productive?
	Q16.	In your opinion, what PPIs would you recommend for a beginning programmer to develop parallel applications more productively? Please justify your choice.
	Q17.	In your opinion, what PPIs do you think a developer needs to be more experienced in parallel programming to be able to parallelize an application more productively? Please justify your choice.
	Q18.	If you have any other opinions about productivity in PPI, please send us an audio file.

6.1.2 Execution

We selected 725 participants to this survey. However, for 42 of the selected participants the e-mail address was incorrect or could not be found. We have also tried to contact them through other social media (e.g., Linked In) but without success. Therefore, we sent this survey to 683 people via e-mail, Linked In or Research Gate. In addition, we sent this survey to 30 research labs in the HPC area. The survey was available from January 28th, 2022, to November 31st, 2022. During this period, we sent some reminders to the participants. After this deadline, we had 131 responses, representing a response percentage of around 19.18%.

6.1.3 Analysis

In this study, we performed a quantitative and qualitative analysis of the data (Figure 6.1). We used descriptive statistics to summarize and describe the data obtained through the closed-ended questions. We used GT procedures to analyze the qualitative data: open coding to create codes related to specific excerpts from the responses and axial coding to identify the categories and subcategories and create relationships between them [42]. The qualitative analysis was performed using ATLAS.ti tool[3], in which graphical representations were generated to represent the relationships identified.

6.2 Results evaluation

This section presents a quantitative (Section 6.2.1) and qualitative (Section 6.2.2) analysis of the results of this survey.

6.2.1 Quantitative analysis

This section presents this study's variables, objectives, hypotheses, and context. This section presents a quantitative analysis of the participants' responses regarding their backgrounds and the applications developed by them.

Participants' profile

This section presents the profile of the 131 participants of this survey, whose are from 26 different countries. Figure 6.2 shows that most of the participants are Brazilians (41%). Although in smaller numbers, we also have quite a few Italian (10%), American (9%), Spanish (8%), German (5%), and French (3%). About 2% of the participants are Argentinian, Austrian, Colombian, British, Dutch, Mexican, Portuguese, Romanian, and Venezuelan. Finally, the smallest part of the participants is from Costa Rica, Cuba, Greece, India, Lebanon, Norway, Peru, and Poland (1% each).

Figure 6.3a shows that most participants have Ph.D. and Master's degrees (89 and 35). Only one participant has a specialization, and six participants have an undergraduate degree. Figure 6.3b shows that most participants work in educational intuitions, such as universities and technological institutes from different (Figure 6.4). Most of these participants work at universities in Brazil, Italy, Spain, United Kindgdom, and United States of

[3]Available at: https://atlasti.com/.

America. Twenty-one participants work in research centers, such as Argonne National Laboratory, Barcelona Supercomputing Center, Galicia Supercomputing Centre, Laboratório Nacional de Computação Científica, Lawrence Berkeley National Laboratory, National Research Council of Italy, and others. Only seven participants work in companies, which are Google, HPE, Intel Corporation, Nvidia, Pagonxt, and SERPRO.

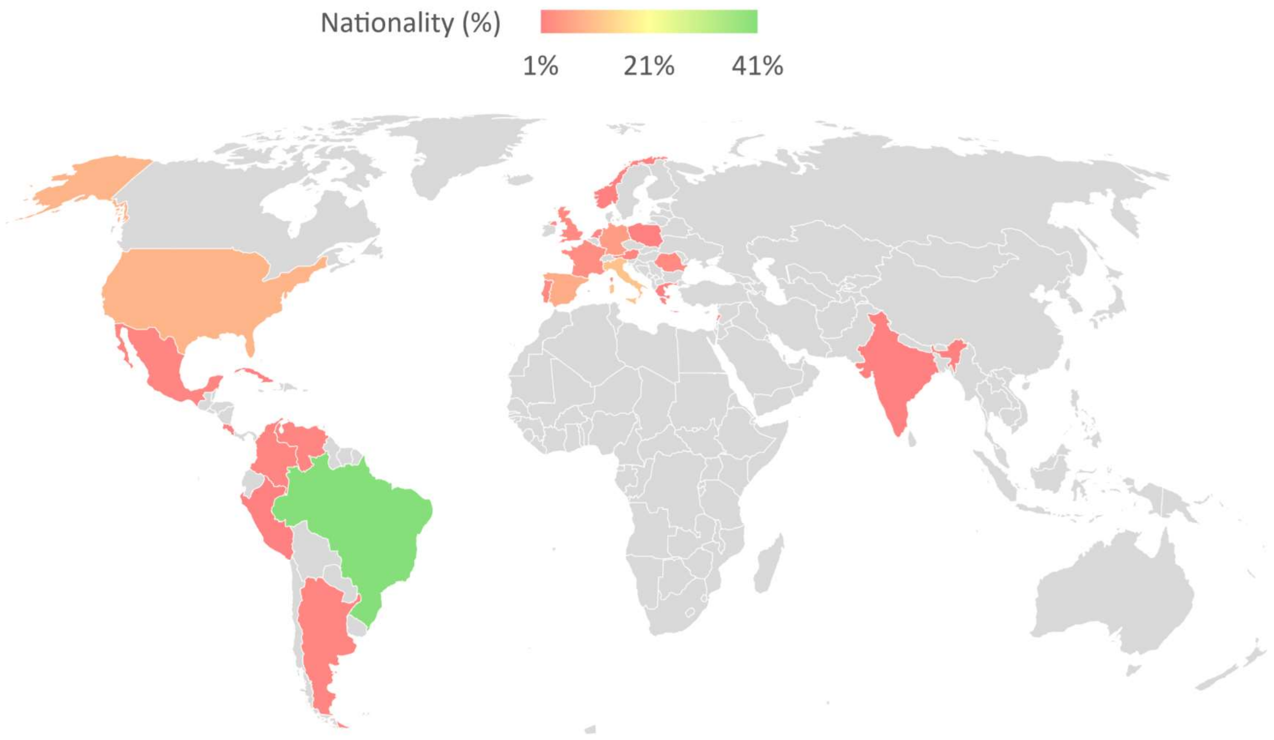
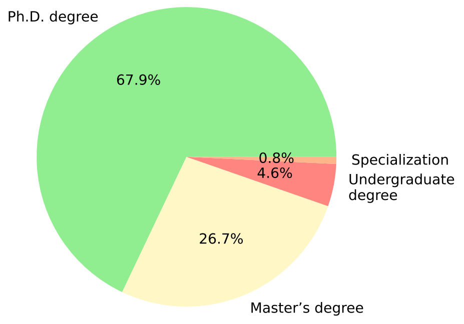
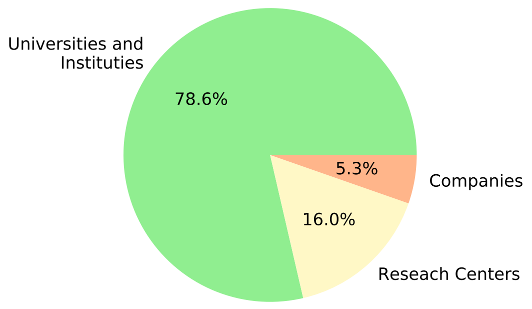


Figure 6.2: Nationality of survey participants.

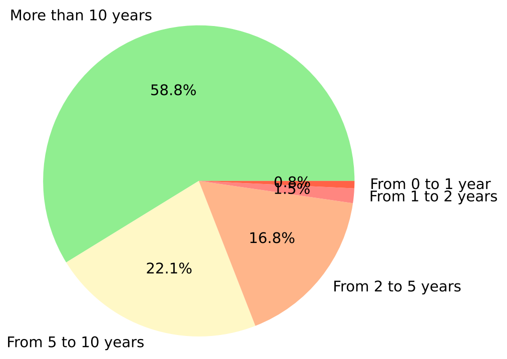
Figure 6.3c shows that most of the participants are parallel programming developers with many years of experience: 29 have between two and five years of experience, 22 have between five and ten years of experience, and 77 have more than ten years of experience. Only two participants have worked in parallel application development for one to two years. In addition, one participant has little expertise in parallel programming, having worked in this field for less than one year. However, years of experience can not prove the developer’s expertise. Therefore, we asked the participants to rate their level of knowledge in parallel programming between beginner, intermediate, and expert. As seen in Figure 6.3d, most participants consider themselves expert developers (84 participants), and 41 participants believe they are intermediate developers in the field. Only three respondents feel that their level of expertise is between intermediate and expert, and two consider themselves beginners. In addition, one of the respondents considers himself a supervisor, as he only supervises the developed applications and does not develop them.



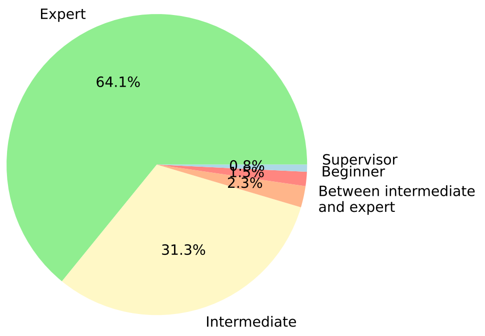
(a) Academic background.



(b) Affiliation.



(c) Years of experience.



(d) Level of experience.

Figure 6.3: Survey participants' profile.

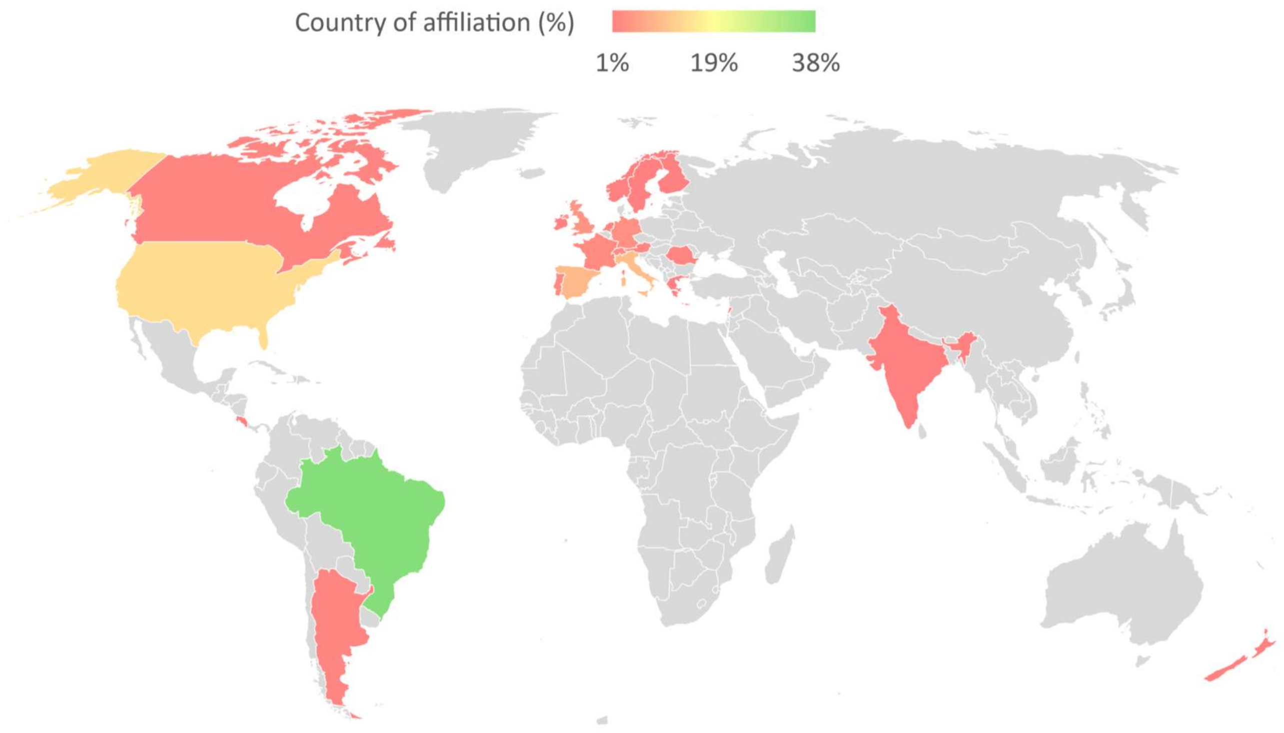


Figure 6.4: Country of participants' affiliation.

Application features

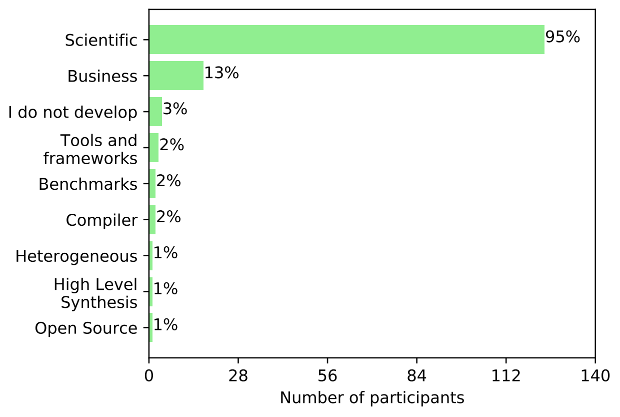
This section presents the applications developed by the participants of this survey. Figure 6.5a shows the type of applications developed by the participants (Q7), of which most are scientific applications (95%). Smaller in number are business applications, which 17 participants develop. Benchmarks to evaluate the PPIs, compiler, and tools/frameworks to exploit parallelism are developed by 2% of the participants each. Only one participant said that they develop quite variable and heterogeneous applications (between science and business), and one said that they develop open-source applications. In addition, 4 participants state that they do not currently develop any parallel application. They have developed in the past, or they just supervise and teach other programmers.

Figure 6.5b shows the domain of these applications (Q8). Most of the participants have developed Scientific Computing applications (110 participants). Computer Simulation is the second field more explored, which is developed by 56 participants. Many participants have explored data processing domains, such as big data (27%), stream processing (24%), and data stream (11%). Computer vision field is also explored by a portion of participants (10%). In addition, the other nine fields are less explored: compilers, HPC, parallel computing, artificial intelligence, Computational Fluid Dynamics (CFD), data parallelism, high intensity servers, system and runtime, and version control.

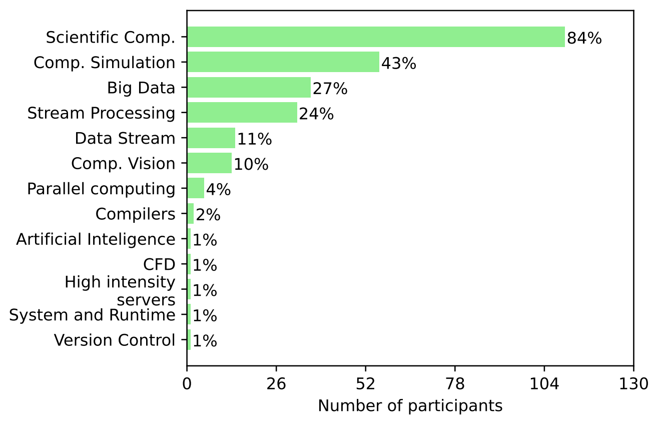
Concerning exploring parallelism (Q11), most of the respondents (96) have developed parallel applications from sequential applications. On the other hand, 62% of them have implemented the parallelism from the beginning of the development process, as seen in Figure 6.5c. In addition, Figure 6.5d shows that these applications are majority developed with C and C++ programming languages (71%). Python is another programming language very used (32%), followed by Fortran (17%) and Java (12%). Scala, Go, Erlang, R, Chapel, DSL, Haskell, Lua, Mathematic, NodeJS, Reference nets, Verilog, and VHDL are the fewer programming languages used by the participants.

Figure 6.5e shows the PPIs used by the participants to explore parallelism (Q12). OpenMP is one of the most popular PPIs for multi-core environment and consequently is the most used (100 participants). Similarly, MPI and CUDA, popular PPIs for distributed and GPU environment, are the second and third most used (92 and 86 participants each). OpenMP, MPI, and CUDA are consolidated PPIs, which justifies these results. Consequently, the most exploited architectures by the participants when implementing parallelism in their applications (Q13) are multi-core CPU, GPU, and Cluster (Figure 6.5f). Although Pthreads and OpenACC are also well-established interfaces, they are used by less than half of the participants (52 and 33 participants each).

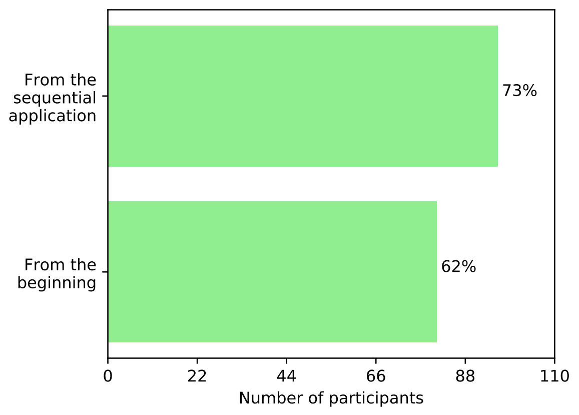
TBB, FastFlow, and SPar are PPIs cited by a group of participants who develop applications for the strem processing domain (16, 14, and 8, respectively). Other PPIs for data processing, such as Spark, Flink, and Storn, are less frequently mentioned. In addition, there are PPIs used only by one participant, such as Vulcan, SkePU, Python Threading, PHAST, oneAPI, Hitmap, Cilk, Chapel, and others. Therefore, there is a need for more dissemination of emerging and academic PPIs. Participants will probably choose to use interfaces such as OpenMP, CUDA and MPI because they have greater support and are already well-established in parallel programming.



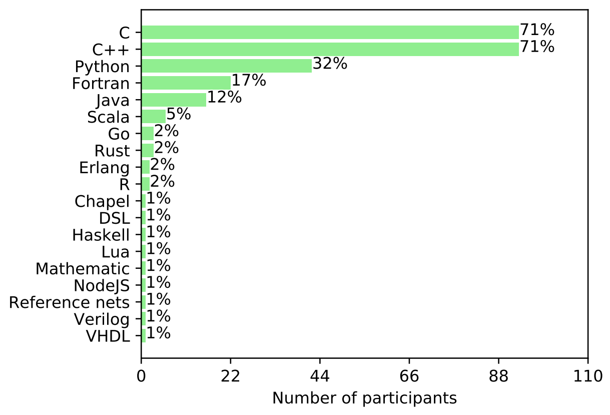
(a) Applications type.



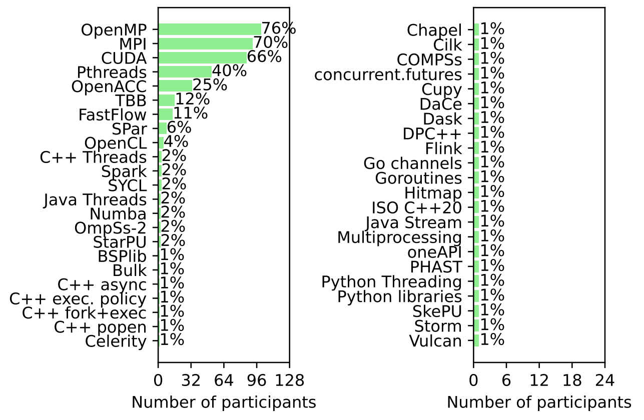
(b) Applications domain.



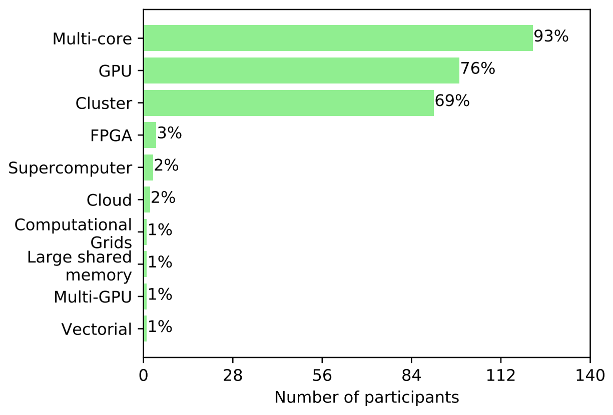
(c) Development process.



(d) Programming language.



(e) Used PPIs.



(f) Target architectures.

Figure 6.5: Characteristics of the applications developed by the participants of this survey research.

Other architectures are less explored by the participants of this survey. Figure 6.5f shows that four participants developing parallel applications for FPGA architectures. Supercomputers and cloud environment are explored by three participants each. Moreover, only one participant develop parallel applications for computational grids, large shared memory, multi-GPU and vectorial architectures.

6.2.2 Qualitative analysis

This section presents a qualitative analysis of the participants' opinions regarding productivity.

Q14. What affects the productivity of parallel application development?

Figure 6.6 shows the factors that prevent productivity according to the respondents' point of view (Q14). The participants reported different reasons, such as those related to PPIs and their programming models and architecture. One of the main reasons given by the respondents was the difficulty in understanding the application to be parallelized. In addition, it is necessary to understand the application's behavior to identify possible data dependencies, critical regions, etc. It is often necessary to modify and refactor the code to enable parallelization. The quotes below highlight that this is one of the main factors that complicate application parallelization:

> *"The need to refactor a large sequential code base which may not be prepared for parallelism."*
> [Participant 23]

> *"The other fact that impacts the productivity, in my opinion, it is the refactoring of the serial code. Commonly, an optimized version targeting GPUs is unrecognizable, it means, you see the code and it is completely different from the serial code."*
> [Participant 46]

> *"Some algorithms are recursive and difficult to parallelize without extensive changes."*
> [Participant 62]

The participants also reported problems related to the difficulties they face when debugging and testing parallel applications, an activity on which they spend a lot of time. There are different factors that hinder the testing and debugging activities of parallel applications, such as lack of proper testing and debugging tools and environments, difficulty and lack of knowledge to use existing profiling tools, difficulty to interpret profiler output, and benchmarking setup. In addition, the occurrence of low-level errors makes it difficult

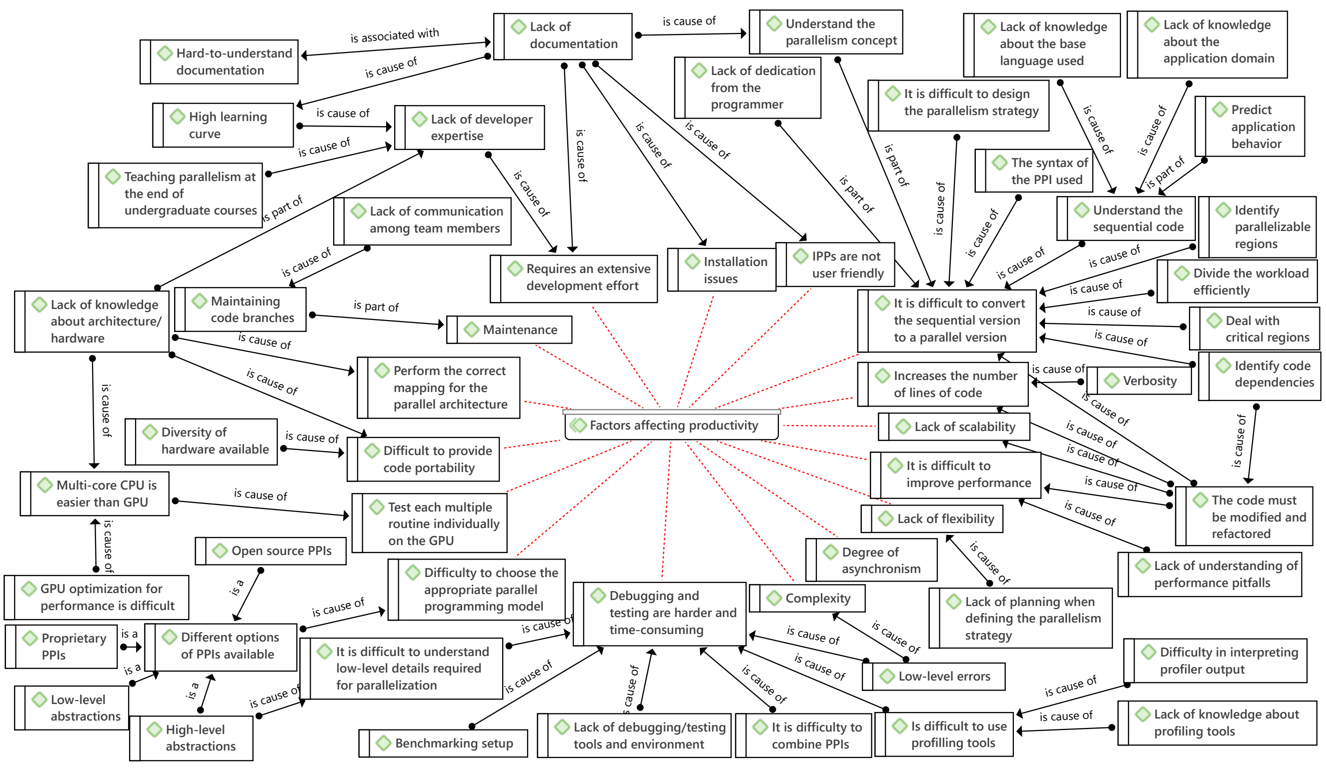

Factors affecting productivity
Hard-to-understand documentation
Lack of documentation
Understand the parallelism concept
Lack of knowledge about the base language used
Lack of knowledge about the application domain
Lack of dedication from the programmer
It is difficult to design the parallelism strategy
High learning curve
Lack of developer expertise
Teaching parallelism at the end of undergraduate courses
Predict application behavior
The syntax of the PPI used
Lack of communication among team members
Understand the sequential code
Identify parallelizable regions
Requires an extensive development effort
Installation issues
IPPs are not user friendly
Divide the workload efficiently
Lack of knowledge about architecture/ hardware
Maintaining code branches
Maintenance
It is difficult to convert the sequential version to a parallel version
Deal with critical regions
Identify code dependencies
Perform the correct mapping for the parallel architecture
Increases the number of lines of code
Verbosity
Diversity of hardware available
Difficult to provide code portability
Lack of scalability
Multi-core CPU is easier than GPU
Test each multiple routine individually on the GPU
It is difficult to improve performance
The code must be modified and refactored
Lack of flexibility
Open source PPIs
Degree of asynchronism
Lack of understanding of performance pitfalls
GPU optimization for performance is difficult
Difficulty to choose the appropriate parallel programming model
Debugging and testing are harder and time-consuming
Complexity
Lack of planning when defining the parallelism strategy
Proprietary PPIs
Different options of PPIs available
It is difficult to understand low-level details required for parallelization
Difficulty in interpreting profiler output
Low-level errors
Low-level abstractions
High-level abstractions
Benchmarking setup
Lack of debugging/testing tools and environment
It is difficulty to combine PPIs
Is difficult to use profiling tools
Lack of knowledge about profiling tools
is cause of
is part of
is associated with
is a

Figure 6.6: Factors that prevent programmers from developing parallel applications productively.

to debug applications because it increases their complexity. The use of PPIs that provide high-level abstractions also affects the debugging of applications, as they can make it difficult for programmers to understand the low-level details needed for parallelization. Therefore, when the programmer has to deal with lower-level details such as memory management, they will have difficulty debugging the application. Some quotes highlight these issues:

> *"Lack of debugging support to detect race conditions and other ill concurrency phenomenon."*
> [Participant 35]

> *"The second obstacle is the time spent needed for testing and debugging, e.g., developing test cases, troubleshooting crashes.."*
> [Participant 61]

> *"The lack of a good debugging environment."*
> [Participant 104]

The developer's expertise was also one of the main factors identified to an extensive development effort. The participants highlight that it is challenging to perform optimizations to obtain high-performance without knowing about the PPIs used and the target architecture. It is difficult for the programmers to choose the right programming model for each case since there are different programming interfaces available with different programming models.

> "*Different options of parallel programming tools.*"
> [Participant 3]

> "*Choosing the proper model.*"
> [Participant 24]

> "*Diversity in hardware and programming models.*"
> [Participant 91]

One of the main reasons pointed out by the survey participants for the lack of experience of programmers is the lack of documentation about PPIs and the lack of courses that address parallel programming properly during undergraduate studies. The lack of proper documentation hinders students' learning, and consequently their understanding of parallelism concepts. Furthermore, due to poor documentation, the interfaces are not user-friendly, and installation issues can be occurred. The following quotes highlight the importance of documentation for efficient use of PPIs:

> "*Like in any other software development team, lack of documentation cause delays etc.*"
> [Participant 13]

> "*Lack of documentation, especially for open source libraries.*"
> [Participant 93]

Q15. What can be done to make parallel programming more productive?

Figure 6.7 shows possible solutions pointed out by the participants to improve developer productivity (Q15), such as more expressive and concise PPIs in order to avoid rewriting code. The use of PPIs with higher-level abstractions, based on annotations or templates, tends to make them easier to learn and use. In addition, standardized and cross-platform PPIs are some of the solutions pointed out by respondents to provide portability to applications.

The participants point out that developing smarter PPIs that automatically parallelize the entire code would be a perfect solution. However, there are limitations to the design of such a tool. A solution proposed to address this problem is the development of tools to support the development, debugging, testing of parallel applications. If such tools do not further affect developer productivity, they can avoid programming errors, reduce the time spent debugging code, and facilitate parallelization of applications. The quotes below indicate these aspects:

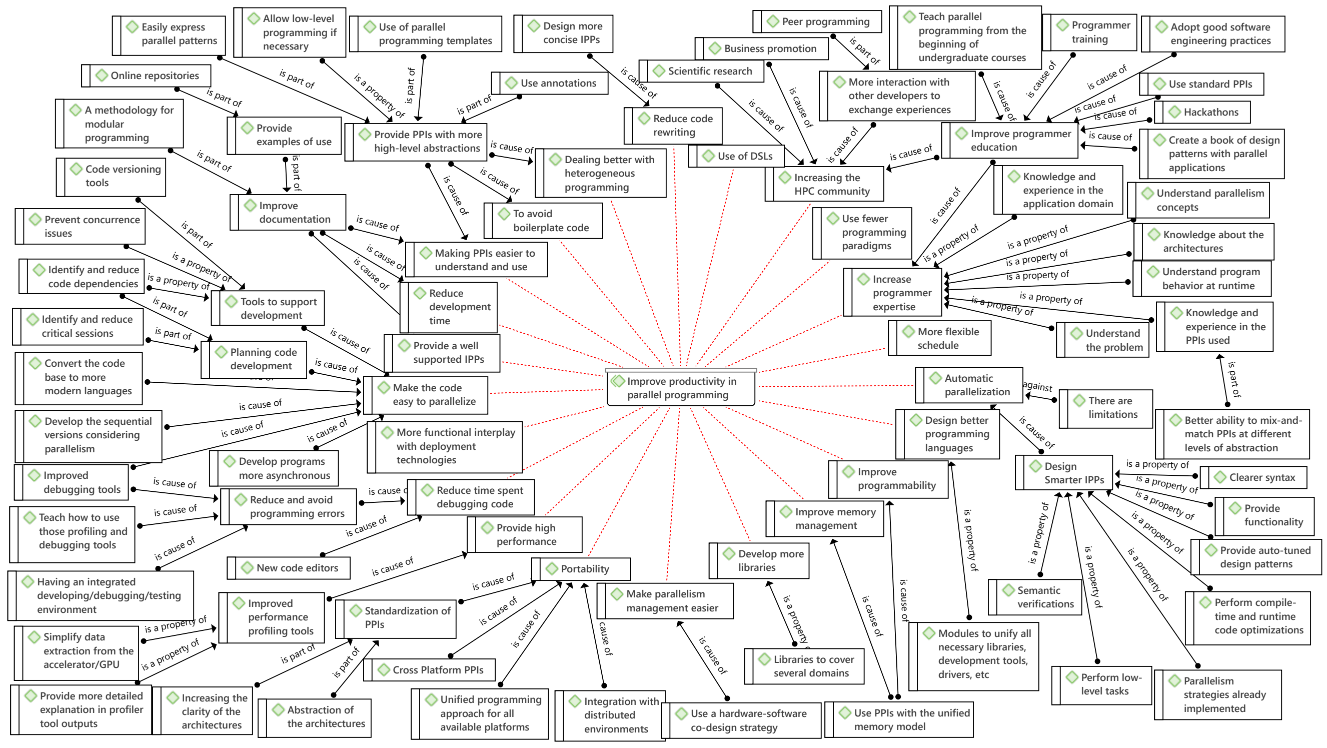

Improve productivity in parallel programming
Easily express parallel patterns
Allow low-level programming if necessary
Use of parallel programming templates
Design more concise IPPs
Peer programming
Teach parallel programming from the beginning of undergraduate courses
Programmer training
Adopt good software engineering practices
Online repositories
Business promotion
Scientific research
Use annotations
More interaction with other developers to exchange experiences
Use standard PPIs
Hackathons
A methodology for modular programming
Provide examples of use
Provide PPIs with more high-level abstractions
Reduce code rewriting
Improve programmer education
Create a book of design patterns with parallel applications
Dealing better with heterogeneous programming
Use of DSLs
Increasing the HPC community
Knowledge and experience in the application domain
Code versioning tools
Understand parallelism concepts
Improve documentation
To avoid boilerplate code
Use fewer programming paradigms
Prevent concurrence issues
Knowledge about the architectures
Making PPIs easier to understand and use
Identify and reduce code dependencies
Increase programmer expertise
Understand program behavior at runtime
Reduce development time
Tools to support development
Knowledge and experience in the PPIs used
Identify and reduce critical sessions
More flexible schedule
Understand the problem
Provide a well supported IPPs
Planning code development
Convert the code base to more modern languages
Make the code easy to parallelize
Automatic parallelization
There are limitations
Develop the sequential versions considering parallelism
More functional interplay with deployment technologies
Design better programming languages
Better ability to mix-and-match PPIs at different levels of abstraction
Develop programs more asynchronous
Design Smarter IPPs
Clearer syntax
Improved debugging tools
Reduce time spent debugging code
Improve programmability
Reduce and avoid programming errors
Improve memory management
Provide functionality
Teach how to use those profiling and debugging tools
Provide high performance
Develop more libraries
Provide auto-tuned design patterns
New code editors
Portability
Having an integrated developing/debugging/testing environment
Make parallelism management easier
Semantic verifications
Improved performance profiling tools
Standardization of PPIs
Perform compile-time and runtime code optimizations
Simplify data extraction from the accelerator/GPU
Modules to unify all necessary libraries, development tools, drivers, etc
Libraries to cover several domains
Cross Platform PPIs
Perform low-level tasks
Parallelism strategies already implemented
Provide more detailed explanation in profiler tool outputs
Increasing the clarity of the architectures
Abstraction of the architectures
Unified programming approach for all available platforms
Integration with distributed environments
Use a hardware-software co-design strategy
Use PPIs with the unified memory model
is part of
is a property of
is cause of
against

Figure 6.7: Possible solutions to increase the productivity of parallel application development.

"Tools that read the code and visualize the dependencies would greatly help."
[Participant 1]

"Tools that aid in debugging libraries and frameworks that smartly takes care of many of the responsibilities of the programmer (Like communication, data copies, scheduling)."
[Participant 4]

"Develop programming support tools, which perform low-level tasks such as data partitioning and distribution or workload balancing."
[Participant 128]

The use of profiling tools also contributes to increasing developer productivity. According to the participants in this survey, proposed improvements to profiling tools help develop high-performance applications. These improvements include simplifying data extraction for accelerator environments and providing more detailed explanations in the output of profiling tools. In addition, participants reported that developers should be instructed how to use such tools effectively in order to increase their productivity.

Regarding the education of parallel programmers, increasing their expertise also allows them to better understand the concepts of parallelism and how to use PPIs efficiently. It is necessary to improve the programmers' knowledge about architectures in order to better exploit their potential to provide performance either by using only one PPI or mix-and-match PPIs of different abstraction levels. It is also necessary that programmers know how to recognize the characteristics of the application to be parallelized, understanding the behavior of the code at runtime, and acquiring more knowledge about its domain. Furthermore, the participants pointed out different ways to improve the teaching of programmers, such as conducting hackathons and training, improving the teaching of parallel programming in undergraduate courses (from the beginning), and adopting good software development practices. The quotes below highlight ways to improve programmers' learning.

> “*An internship that begins right at the beginning of an undergraduate course. You need to shape parallel computational thinking early on in your study career.*”
> [Participant 12]

> “*Teach parallelism from the very beginning. There's a need of mindset change in the community.*”
> [Participant 68]

> “*I think that Hackathons and events where people can get exposed to these problems in a hands on matter and talk to experts help.* ”
> [Participant 131]

According to the survey participants, improving the teaching of parallel programming is also a way to increase the HPC community. It is also important to have more interactions between developers to exchange experiences, either through scientific research or by promoting companies. In addition, improving the documentation of available PPIs and provide more usage examples can make them easier to understand and use.

Q16. What PPIs would you recommend for a beginning programmer?

We asked the participants which PPI they would recommend for beginners to develop parallel applications productively (Q16). Figure 6.8 shows the PPIs indicated. Most participants ($\approx$ 64%) suggest that beginners start developing parallel applications using OpenMP. MPI, OpenACC, and CUDA also were recommended for more than 11% of the participants. Other PPIs (e.g., Pthreds, SPar, TBB, etc.) were less recommended, which were recommended for less than eight participants. In addition, some participants did not suggest the use of any specific PPI by beginners (14 participants), of which two did not feel able to answer this question. For three participants, the PPI used for parallelization depends on the goals of the developer and the target platform. Some participants indicated the use of PPIs with certain characteristics such as ease of use, low and high level of abstractions, and the use of new PPIs not attached to the sequential language. Three

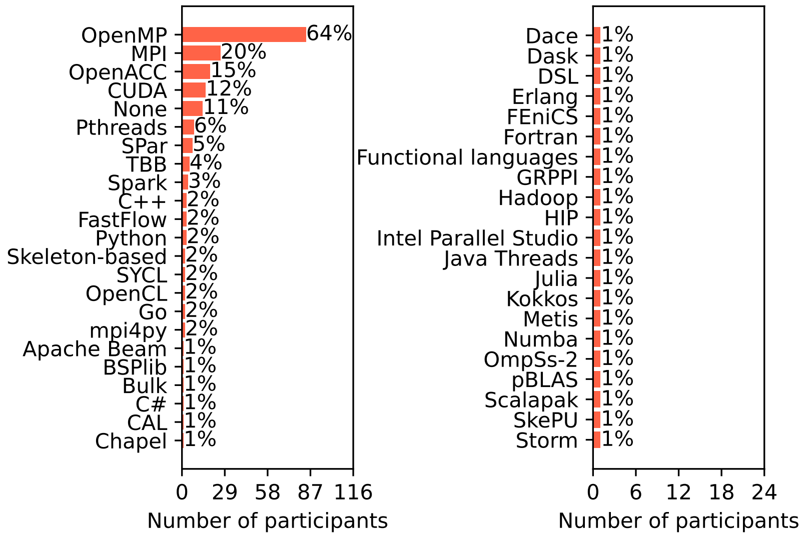


Figure 6.8: Recommended PPIs for beginner to develop parallel applications more productively.

participants suggested improving the training of developers initially. Moreover, one participant thought that beginners should not deal with parallel applications, contrary to the opinion of most other participants.

As previously mentioned, OpenMP was the PPI majority indicated by the survey participants to beginners develop parallel applications productively. Figure 6.9 presents the main reasons for the survey participants suggest OpenMP, such as ease of understanding and use. It is easier for novice developers to develop parallel applications for multi-core environments. In addition, using `pragmas` is simpler for beginner programmers because it provides high-level abstractions with a basic syntax and less vocabulary. Since OpenMP provides high-level abstractions, its learning curve is shorter and consequently allows developers to quickly understand the essential concepts of parallelism. Therefore, OpenMP allows developers to quickly parallelize a block of code without the need to modify and restructure all the sequential code in order to provide parallelism. The quotes below highlight these aspects:

> "*I recomend C+OpenMP, since the use of pragmas are considered easy for beginners.*"
> [Participant 12]

> "*OpenMP because this tool offers resources allowing to implement a parallel code based on the serial code (incremental parallelism) So, this makes the programming easier than other tools that apply an approach based on the all-or-nothing conversion of an entire program.*"
> [Participant 31]

> "*Annotation-based approaches such as OpenMP pragmas and SPar tend to be easier to use.*"
> [Participant 49]

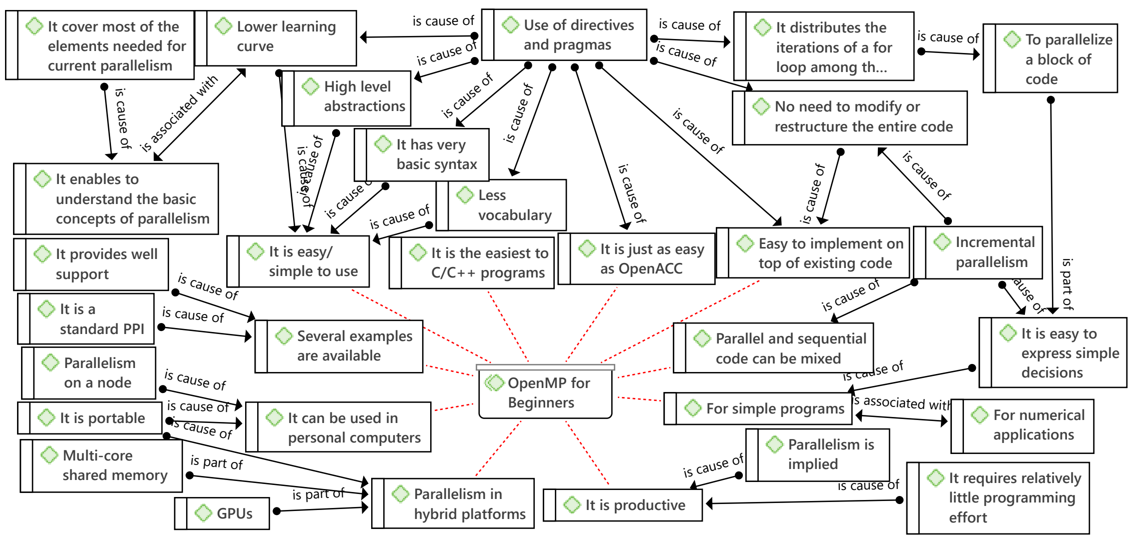

It cover most of the elements needed for current parallelism
Lower learning curve
Use of directives and pragmas
It distributes the iterations of a for loop among th...
To parallelize a block of code
High level abstractions
No need to modify or restructure the entire code
It has very basic syntax
It enables to understand the basic concepts of parallelism
Less vocabulary
It provides well support
It is easy/ simple to use
It is the easiest to C/C++ programs
It is just as easy as OpenACC
Easy to implement on top of existing code
Incremental parallelism
It is a standard PPI
Several examples are available
Parallel and sequential code can be mixed
It is easy to express simple decisions
Parallelism on a node
OpenMP for Beginners
For simple programs
For numerical applications
It is portable
It can be used in personal computers
Parallelism is implied
Multi-core shared memory
It requires relatively little programming effort
Parallelism in hybrid platforms
It is productive
GPUs
is cause of
is associated with
is part of

Figure 6.9: The reasons why survey participants recommend OpenMP for beginners to develop parallel applications more productively.

Other factors related to the popularization of OpenMP make it easier for beginners programmers to use it, such as good support provided, many code examples available on the Internet, and its portability. OpenMP portability allows it to be used on both personal computers and hybrid platforms consisting of multi-core CPUs and GPUs. Therefore, the survey participants think that it is better for beginners programmers to start developing parallel applications with OpenMP because it allows heterogeneous programming and is easier to learn than other interfaces like OpenACC. In addition, OpenMP is the PPI most used by the participants as seen in Section 6.2.1, which may explain its recommendations.

Q17. What PPIs would you recommend for a experts programmer?

Figure 6.10 shows the PPIs indicated by respondents for experts to develop parallel applications productively. They indicated that experienced developers should explore architectures other than the traditional multi-core, such as GPU and Cluster. The two PPIs most indicated were CUDA and MPI (both 34%), as they require more experience to be used efficiently by the developers. CUDA and MPI are also widely used by the participants, as seen in Section 6.2.1, which may explain their recommendations. OpenMP and Pthreads were recommended for more than 11% of the participants. Other PPIs (e.g., OpenACC, OpenCL, etc.) were for fewer than five participants. In addition, 23% of the participants did not indicate any specific interface because the PPI to be used for parallelization depends on the goals of the developer and the target platform.

Figure 6.11 presents the main reasons pointed out by the respondents to indicate CUDA. One of the main reasons given by the participants for using CUDA is to exploit the processing power of GPU architectures to achieve high performance, mainly in NVIDIA GPUs. However, developing efficient parallel applications using CUDA requires the devel-

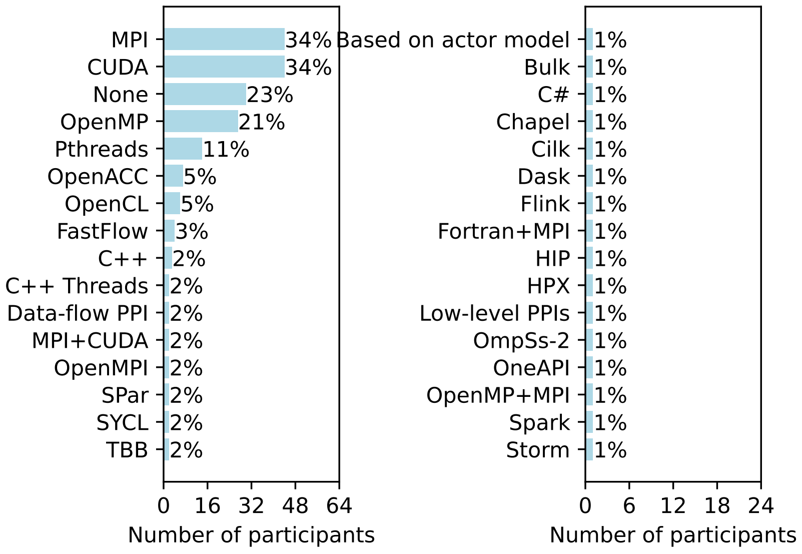


Figure 6.10: Recommended PPIs for experts to develop parallel applications more productively.

oper to have more experience in parallel application development. The developer must know the properties of the hardware in order to exploit concurrency. The following quotes give reasons why developers should have knowledge about GPU operation in order to develop parallel applications with good performance:

> *"I think CUDA requires more specialization, since it is necessary to adjust the data structure to fit the hardware."*
> [Participant 30]

> *"In my opinion, CUDA is one of the best parallel frameworks, but it requires experience to produce good speedups When using CUDA, developers need to understand the process of writing and reading from a GPU memory. These GPU communications are time-consuming processes, which can surpass the parallel speedup of the GPU parallel execution."*
> [Participant 33]

According to the survey participants, developing applications with CUDA requires more effort from the programmer and more debugging time because CUDA is a lower-level interface. CUDA has several advantages, although the learning curve is higher. One of the advantages pointed out by the participants is the use of CUDA in conjunction with other interfaces, such as OpenMP and MPI, to exploit parallelism in hybrid architectures. In addition, CUDA enables efficient parallelization of more complex applications, such as robotics, computer vision, and machine learning applications.

> *"Today, I think CUDA is very important Real-time applications in computer vision and robotics demands efficient parallel execution Also, training deep learning models re-*

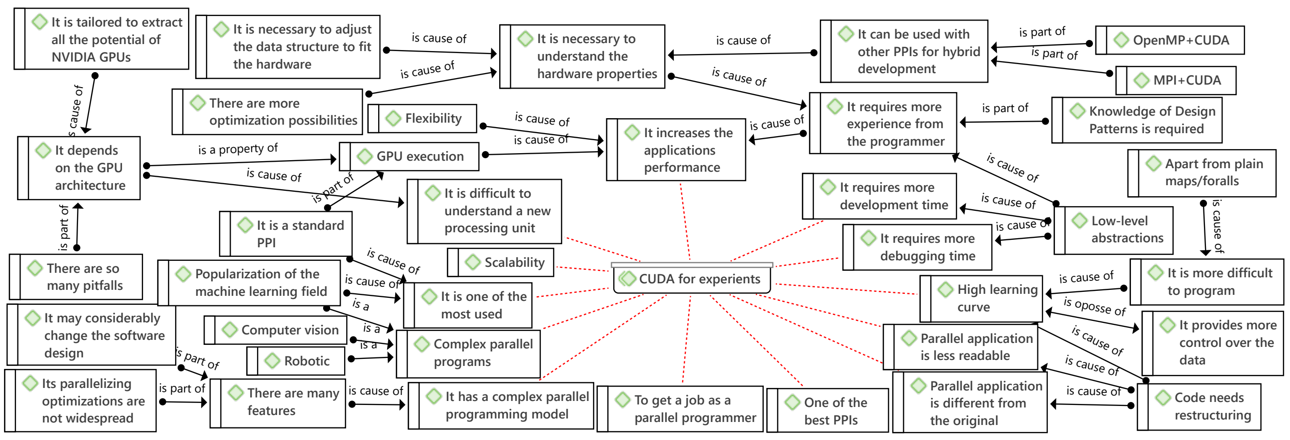


Figure 6.11: The reasons why survey participants recommend CUDA for experts developers.

quires fast implementations once it uses a massive amount of data"
[Participant 26]

"CUDA and GPU's are becoming more used with the spread of the machine learning. CUDA parallelizing optimizations are not widespread and many frameworks haven't this into account."
[Participant 121]

Figure 6.12 presents the main reasons pointed by the respondents to indicate MPI. This is a standard PPI for distributed memory. Running in clusters can bring advantages for data-intensive processing applications. The participants point out that the use of MPI can bring good performance results for scientific and big data processing applications, as can be seen in the following quotes:

"In addition to OpenMP, and optionaly OpenACC and CUDA, it is necessary to know MPI, because some applications processing large data and could be run in a cluster."
[Participant 8]

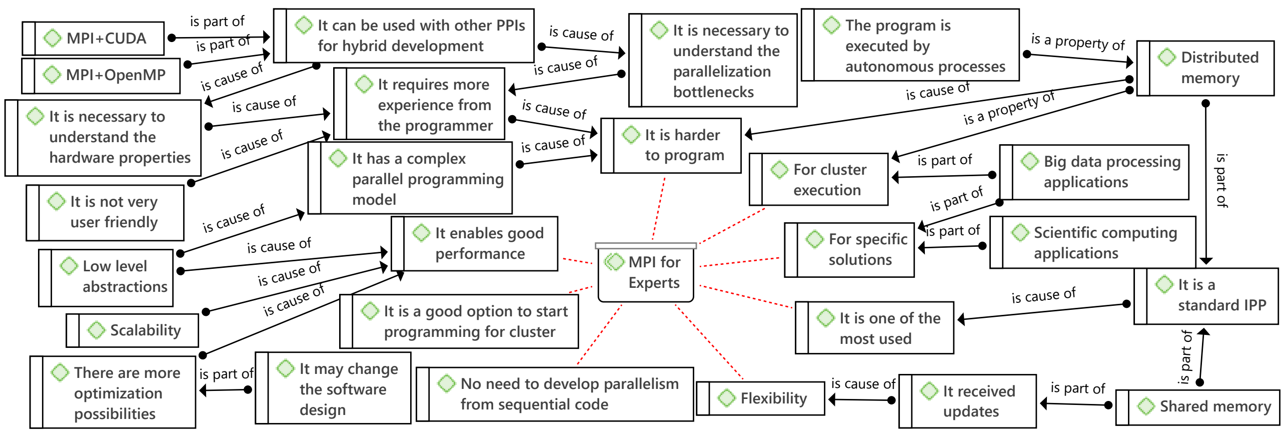


Figure 6.12: The reasons why survey participants recommend MPI for expert developers.

> “*MPI and OpenMP is essential for scientific computing*”
> [Participant 36]

Despite the MPI advantages, it requires more experience from the developer. MPI is not very user-friendly and is harder to program because it has a complex parallel programming model, which provides low-level abstractions and requires more knowledge about the distributed hardware. In addition, there are code optimization possibilities, which can be better explored by more experienced programmers. The quotes below highlight these aspects:

> “*For top performance, specific solutions may need to be developed in a more low-level (more hardware details exposed) and thus more complex parallel programming model, such as MPI for clusters.*”
> [Participant 86]

> “*I think that conventional approaches like MPI, OpenMP, CUDA, etc. require more experience to use effectively because they use disparate concepts and syntactic forms to express the same basic ideas (e.g., parallelism, locality), simply at different levels of granularity.*”
> [Participant 57]

The participants also highlighted the flexibility of MPI, which now allows the exploration of shared memory architectures. In addition, two respondents indicated using MPI combined with CUDA to get more performance. As can be seen in the following quotes, it is necessary for the programmer to have enough experience to know how to use the hardware resources efficiently.

> “*MPI+CUDA. Bad code can be made. But to truly use it and create a decent MPI+CUDA application, the developer needs to understand a lot about the specific hardware, manually optimize things like memory/cache in all devices, DMA communications, the network conditions that they are in, what can be done in CPU or GPU, and general distributions and scheduling problems tailored for their application.*”
> [Participant 16]

> “*If a set of multiple interacting levels of parallelization is required (i.e. the common MPI+CUDA) then that needs further experience on part of the developer.*”
> [Participant 46]

6.3 Final remarks

In this chapter, we aimed to identify the factors that hinder the productivity of parallel programming, as well as identify ways to improve the productivity of programmers. For this purpose, we conducted an international survey with developers of parallel applications. Our results showed that lack of experience is one of the main reasons for the increased development effort. These results showed a gap in teaching parallel programming in universities, where concepts related to parallel computing are covered only at the end of undergraduate courses. The survey participants suggest teaching parallel programming from the beginning of undergraduate courses to overcome this problem. Teaching programming to beginner students can be a complex task. However, the results of Conte *et al.* [40] showed that it is possible to teach parallel programming to students with no prior knowledge of computing, obtaining high scores and interest in this learning. The HPC community can also promote hackathons and training events to improve programmers' education and improve the interaction between developers to encourage an exchange of experiences. Moreover, the designers of the PPIs must improve documentation to facilitate the students' learning process.

The results show that GPU programming is more complex than multi-core programming from the participants' perspective of this survey. The survey results confirm the results presented in Chapter 4, where the efforts to explore parallelism in multi-core environments were approximately 64.83% lower than for GPU environments. The developer's experience with the architecture and programming model used also negatively impacted the development effort. Developing effective GPU applications requires more understanding and knowledge of how the architectures work to perform optimizations that exploit their full processing power. PPIs that provide higher-level abstractions also facilitate the development of parallel applications for multi-core environments.

From this survey, we also identified possible solutions to improve productivity based on the participants' opinions. While high-level abstractions can increase developer productivity, their use makes it difficult to understand the low-level details required for parallelization. For more complex architectures, such as GPU, the developer needs to know architecture detail to avoid programming errors and perform the necessary optimizations to achieve performance. The results of the GPU study presented in Chapter 4 confirm these assertions. In this study, the students used the GSParLib Driver API instead of the Patterns API. Patterns can increase the productivity of the developers. However, they chose to program at a lower level due to the flexibility to manipulate the architecture. In parallel programming, programmers can only develop applications faster if they achieve the expected performance. Therefore, coding productivity and application performance must be complementary.

Understanding how sequential code works is also essential to achieve good performance in the parallel version. When developers understand the behavior of the sequential application, they can avoid many programming errors. Moreover, profiling tools can help programmers identify bottlenecks in the application and facilitate parallelization since they know how to use them without further hindering their productivity. Therefore, we concluded that the developer's experience level is the main factor that impacts productivity. The greater their knowledge about parallel architectures, PPIs, and programming skills, the higher their development productivity.

According to most participants, OpenMP is one of the easiest PPIs for parallel programming, while CUDA and MPIs require more experience from developers to be used efficiently. We have identified some studies from the literature review that show the productivity-related advantages of OpenMP over other interfaces, such as CUDA [247], FastFlow [46, 157], Kokkos [197], LE-OpenMP [214, 249], mxhMD [197], MPI [117, 96], OpenCL [155, 249], OpenACC [249], and Pthreads [157]. However, Hoffmann *et al.* [99] implemented an update in SPar to automatically generate parallel code at the OpenMP runtime, which in previous versions only generated parallel code at the FastFlow and TBB runtimes. Hoffmann *et al.* [99] showed that in addition to reducing the SLOC number, SPar reduced the complexity of switching between runtime versions (FastFlow, TBB, and OpenMP) since the SPar annotations were the same for each of these PPIs. Furthermore, the performance of the SPar versions was very similar to their handwritten implementations, with performance differences of less than 2.49%. Therefore, these results show that using interfaces with a higher level of abstraction can increase developer productivity without significant performance losses.

This study presents some threats to validity. The collection method may threaten the study's internal validity, as participants may report incorrect data. The participants' profiles can be considered a threat to external validity, as most participants are from the American and European continents. There are no participants from the African continent. Therefore, we cannot generalize this study to the international level. In addition, the sample of participants can be considered a threat to the conclusion validity, as most participants are from academia and not industry. This study provides valuable insights regarding productivity in parallel programming despite these limitations. In future investigations, we plan to include more participants from industry and other nationalities.

7. PLANNING POKER APPLIED TO PARALLEL PROGRAMMING

Over the years, some effort has been dedicated to improving metrics for estimating development effort in the SE [243]. On the one hand, some researchers have proposed extensions and improvements to already established models, such as COCOMO and FPA, using bio-inspired algorithms [71, 114, 115], fuzzy logic [88, 175], and machine-learning techniques [16, 122, 206, 256]. On the other hand, new models are also conceived based on these algorithm techniques [55, 59, 217, 241]. From a literature review, we identified only two studies aiming to propose models for estimating parallel programming effort. The first is an extension of COCOMO II [248], which presents only a conceptual model. The other study uses machine learning algorithms to estimate the increase in effort required to convert C++ applications to CUDA using Halstead's measures.

To create a model to estimate the effort required to develop applications is necessary to create a dataset to train the model. However, collecting datasets is a challenging task. In the SE domain, there are several public domain datasets available to evaluate effort estimation models. One of the most used is the PROMISE repository, which includes a collection of datasets based on COCOMO (COCOMO81, COCOMO NASA, COCOMO NASA 2) and FPA (Desharnais) models and tools to help researchers aim to build predictive software models. In the parallel programming domain, no dataset is available to evaluate the effort estimation models. Wienke *et al.* [248] performed a research survey with experts to collect factors impacting the development of parallel applications. However, the datasets produced by Wienke *et al.* [248] are not available. On the other hand, Marantos *et al.* [143] provided the dataset used to train the regression models they used. However, Marantos *et al.* [143] still need to provide a data dictionary to facilitate the replication of the study. In addition, Marantos *et al.* [143] did not consider other factors that may impact the development effort of parallel applications, such as the developer's experience, the target architecture, and the programming model used. The previous chapter provided a more detailed discussion regarding the impact of these factors on developer productivity.

In our work, we conducted an experiment with 15 beginners in parallel programming to understand the main challenges in implementing a video processing application (Section 4.2). This experiment was performed to evaluate specific PPIs (SPar, FastFlow, and TBB), which were used to develop a stream processing application. In addition, we conducted a initial study with four graduate students who were beginners in GPU programming to understand the challenges of developing parallel applications in architecture with accelerators (Section 4.3). To do so, we compared the effort required to program the Animal Rescue problem using CUDA, GSParLib, OpenACC, and OpenCL. However, the two studies were conducted with a tiny sample concerning the population of parallel application developers. Therefore, only these experiments cannot be considered a dataset to assess the accuracy of a technique for development effort estimation. New experiments are needed to build a dataset for the parallel programming domain.

Performing an experiment with people is challenging because the experimentation process takes time to be planned and conducted. The investigation must be well-planned to minimize threats to its validity. We presented a methodology based on the best experimentation practices in Section 4.1 to facilitate the experimentation process. Moreover, one primary difficulty when carrying out experiments in the parallel programming domain is finding experienced developers in that area. Experiments can be carried out with beginner students in parallel programming to overcome this problem. Thus, it is possible to evaluate the ease of learning, which according to ISO/IEC 25010 [109], is one of the main characteristics of software usability. However, the small number of participants in our experiments showed that finding participants for this type of study in parallel programming is still challenging.

In this context, the research question we aim to answer in this chapter is: *How to reduce the effort devoted to collecting development time in parallel programming?*. The results presented in Chapter 5 showed the effectiveness of the Planning Poker method in estimating the development effort of parallel stream processing applications. Although this method requires input from expert developers, it requires less effort to be applied in practice than experiments that collect the actual development time. Therefore, we proposed a new methodology in this chapter to measure the effort required to develop parallel applications based on the Planning Poker method.

Based on the following literature reviews [56, 64], the Planning Poker is the most widely used metric for estimating software projects' complexity and development effort based on the Scrum methodology. Regarding development effort, Moloken *et al.* [161, 162] conducted empirical studies to compare the Planning Poker estimation with the estimation performed by individual experts, which showed similar accuracy. Finco [66] proposed the combination of Planning Poker with machine learning to assess the development effort required for software development teams. Haugen [90] investigated whether using the Planning Poker estimation process could improve the ability of XP teams to estimate story points. Tamrakar and Jorgensen [234] conducted an empirical study to estimate software development effort using Planning Poker with a linear scale instead of the usual Fibonacci sequence. Gandomani *et al.* [69] evaluate the use of average or consensus opinion in Planning Poker to estimate user stories.

In order to produce more accurate effort and time estimates using Poker Planning, Zahraoui and Idrissi [255] proposed an adjustment to the story point calculations using priority, size, and complexity factors. Power [199] presented a technique called Silent Clustering, which can complement Poker Planning to size large sets of user stories quickly. Sudarmaningtyas and Mohamed [232] proposed a new model to improve the actual performance of planning poker by modifying the estimation process and the consensus process of this method. Mahnivc *et al.* [136] conducted an experiment with students and experts to estimate the number of user stories through the Planning Poker method, whose results showed that the experienced participants' estimates were more accurate than the students' estimates. Mahnic *et al.* [137] performed an empirical study with students to identify the essential practices for successful Scrum-based software projects, among which Planning Poker was evaluated. Raith *et al*. [202] evaluated the accuracy of the effort estimates using the Planning Poker method, which led them to develop a prototype to apply this method to a student project. In addition, there are other studies aiming to provide tools to support Planning Poker method execution [30, 167].

Unlike previous works, our study aims to apply a modification of the Planning Poker method to the parallel programming domain, which is discussed in further detail in Section 7.1. Next, Section 7.2 evaluates parallel stream processing applications in order to validate the proposed Planning Poker methodology. Finally, Section 7.3 presents the final remarks.

7.1 Planning poker for parallel programming estimation

The planning poker method was presented in Section 2.5.1, which is usually used by software agile teams to estimate the development effort from experts' opinions about the software to be developed. In Section 5.2, we showed the accuracy of the Planning Poker method to estimate the time required to develop parallel applications for stream processing when comparing the estimated time to the actual time collected from an experiment with beginners' programmers. These results lead us to create an extension of the Planning Poker method to evaluate the development of parallel applications.

In the original Planning Poker (see Figure 2.11), the estimation is performed for all the people who compose the development team, such as the developers, testers, engineers, analysts, and others. In parallel programming, mainly in academic environments, the application will be developed by only one developer. The researchers usually develop their applications without being in a development team, although there are research groups. In other words, the same person will be responsible for the coding, debugging, testing, and evaluating an application. Therefore, the Planning Poker adaptation estimation will be performed only by a developer, as seen in Figure 7.1. Furthermore, the research results presented in Chapter 6 highlight developer experience as an essential factor impacting the parallel application development effort. These results led us to consider only the opinion of experienced developers for the Planning Poker method.

Unlike the original Planning Poker method, which aims to estimate the number of story points or Product Backlog, our goal is to estimate the number of hours required for parallel application development. Furthermore, we will not perform our estimations based on a non-linear sequence like the Fibonacci sequence. The Tamrakar and Jorgensen [234] study showed a decrease in estimated development effort by up to 60% when using a Fibonacci scale instead of the traditional linear scale. Using a Fibonacci scale and other non-linear scales probably affect development effort estimates because they induce people to make biased estimates, especially when the uncertainty is substantial. When considering the modified Fibonacci sequence (0, 1, 1/2, 2, 3, 5, 8, 13, 20, 40, 100), a developer may be in doubt when estimating the number of development hours for a particular application. For example, in the developer's opinion, a specific application requires about 60 hours to develop. Therefore, they might choose the value 100 from the Fibonacci sequence, resulting in an inaccurate estimate. On the other hand, the developer could choose the specific value of the evaluation (60 hours in this case) if a linear scale was used.

The estimators will be also freer to choose their estimates and can compare the estimated values between the evaluated interfaces considering their possible difficulties. There are more straightforward applications where parallelization can be realized within one to two hours of development time, such as the RGB channel extraction application presented in Section 4.2. Therefore, a linear scale can accurately estimate development time for more straightforward applications like this.

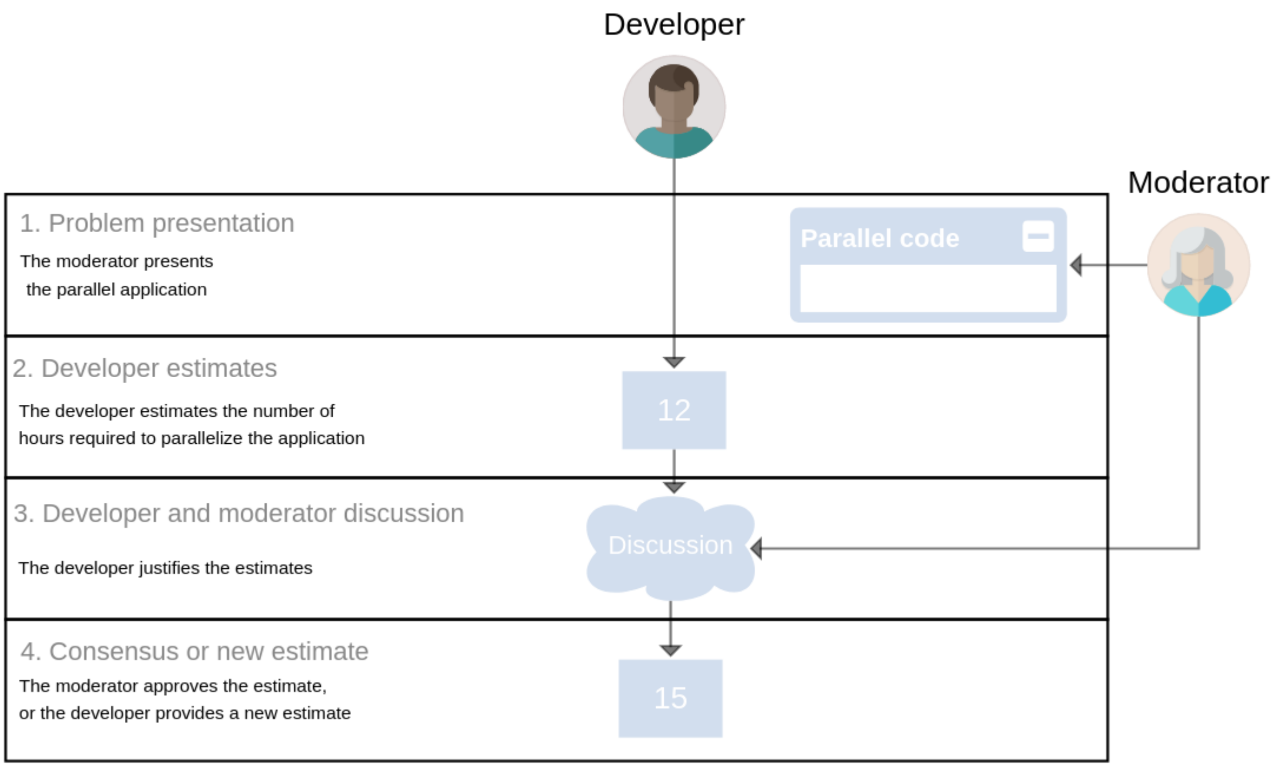

Developer
Moderator
1. Problem presentation
The moderator presents the parallel application
Parallel code
2. Developer estimates
The developer estimates the number of hours required to parallelize the application
12
3. Developer and moderator discussion
The developer justifies the estimates
Discussion
4. Consensus or new estimate
The moderator approves the estimate, or the developer provides a new estimate
15

Figure 7.1: Poker planning method for estimating parallel development time.

In the proposed Planning Poker adaptation, there is still a moderator to coordinate the execution of the method. Figure 7.1 presents the step-by-step to be followed through the proposed methodology. Initially, the moderator should show the code to be parallelized with a given PPI. Soon after, the developer should analyze the code and give the estimate. If there are any inconsistencies with the estimate made by the developer, a discussion should occur between the developer and the moderator. For example, the development time estimated by the developer was only two hours to parallelize a stream processing application with OpenMP. During the discussion, the moderator should give the developer reasons to disagree with the estimate. For example, the moderator should explain that the OpenMP programming model is not based on structured programming (such as TBB). Therefore, the developer should consider the time needed to implement extra synchronization mechanisms when using OpenMP to exploit stream parallelism. From the moderator's exposition, the developer should make a new estimate. Moreover, the estimation will be based only on the developer's opinion, and the moderator should only explain the characteristics of the application to be parallelized and the PPI to be considered in parallelization.

7.2 Planning Poker evaluation

This section evaluated the Planning Poker adaptation applied to parallel programming. To evaluate this metric, we performed an experiment to verify the productivity of FastFlow, OpenMP, Pthreads, SPar, and Intel TBB PPIs for multi-core systems in the development of the following C++ stream processing applications: Bzip2, Person Recognition, Lane detection, and Ferret. The following sections present the experimentation plan and the results obtained.

7.2.1 Experimentation plan

Independent and dependent variables

The development time estimated using the Planning Poker method is a dependent variable. The independent variables include the PPIs evaluated, the parallel applications evaluated, the participant's experience, and the study environment because they impact the dependent variable tested.

Goals

The main goal of this study is to use the Planning Poker method to estimate the development effort required for implementing parallelism in C++ stream processing applications for multi-core environments using FastFlow, SPar, TBB, OpenMP, and Pthreads. The specific goals are as follows:

- Measure the effort required to exploit parallelism using the Planning Poker method for Bzip2 compress and decompress, Person Recognition, Lane Detection, and Ferret applications.

Hypotheses

Based on the goals, we consider the following hypothesis in our experiment:

- H_{0_bzipC}**:** The effort required to implement parallelism on the Bzip compress application is the same for FastFlow, SPar, TBB, OpenMP, and Pthreads;
- H_{0_bzipD}**:** The effort required to implement parallelism on Bzip decompress application is the same for FastFlow, SPar, TBB, OpenMP, and Pthreads;
- H_{0_person}**:** The effort required to implement parallelism on the Person Recognition application is the same for FastFlow, SPar, TBB, OpenMP, and Pthreads;
- H_{0_lane}**:** The effort required to implement parallelism on the Lane Detection application is the same for FastFlow, SPar, TBB, OpenMP, and Pthreads;
- H_{0_ferret}**:** The effort required to implement parallelism on the Ferret application is the same for FastFlow, SPar, TBB, OpenMP, and Pthreads.

Context of the study

This study is offline because it was conducted in an academic environment under controlled conditions. The participants were five graduate students from the PPGCC at the PUCRS in Porto Alegre city, South of Brazil. They are experts in parallel-stream processing programming. Furthermore, this is a specific study because it focuses on evaluating the productivity of PPIs for parallel programming of stream processing applications in an academic environment.

Activity of study

The activity given to the participants was to estimate the time required to develop a series of parallel stream processing applications using FastFlow, OpenMP, Pthreads, SPar, and Intel TBB. The following C++ stream processing applications for multi-core systems were evaluated: Bzip2 data compression and decompression application, Person Recognition application, Lane detection application, and a PARSEC application called Ferret, whose goal is to content similarity search in data such as video, audio, and images [24]. In addition, we evaluate the effort to develop the compression and decompression of the Bzip2 application data separately.

Procedure and execution

Participants should estimate the time required to develop the parallel applications considering an existing sequential application. Participants should also consider that it is their first time parallelizing the application regardless of the PPI used. They also should not consider reusing code from another parallel version to avoid contaminating the experiment with the learning effect. After analyzing the target application, the developers reported their estimates according to the instrument shown in Table 4.6. From this questionnaire, only Q1 and Q2 were applied to the participants because the goal of this experiment is only to collect the estimated development time and not to evaluate user satisfaction or usability.

7.2.2 Productivity evaluation

Table 7.1 presents the development time estimated using the Planning Poker method by five graduate students (anonymized) who are expert developers in the parallel stream processing domain. Figure 7.2 shows the box plots of the estimated Planning Poker development time for each application evaluated. As seen in in the Table 7.1, our results presented the smallest averages for the parallel applications developed using SPar. FastFlow and TBB showed similar results, requiring slightly more development effort than SPar. OpenMP and Pthreds required more development effort, of which Pthreads showed the worst results. OpenMP and Pthreads also showed close results. Therefore, it was necessary to perform a hypothesis test to see if there was a significant difference between the results obtained.

Initially, we performed a normality test to verify that the data collected had a normal distribution. A parametric test should be performed if the samples have a normal distribution (P value $\geq$ 0.05). Otherwise, a non-parametric test should be performed (P-value $<$ 0.05) [34, 224]. Therefore, we used the Shapiro-Wilk test at the conventional significance level (α = 0.05) [224] to verify whether the collected data had a normal distribution. Moreover, we chose the Shapiro-Wilk test because it is an efficient test for all distribution types and can be used regardless of sample size [205].

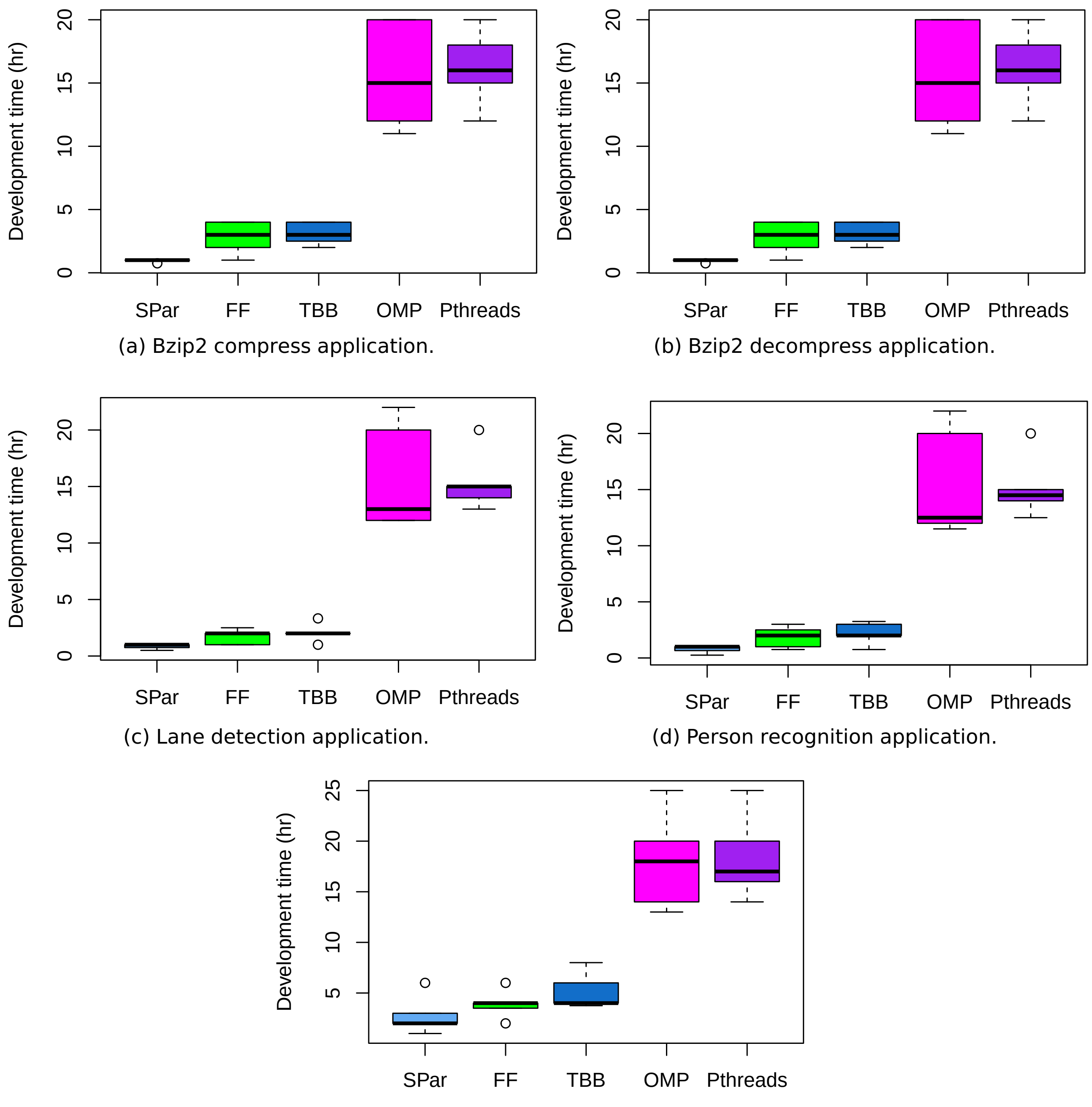
Table 7.2 shows the results achieved for the Shapiro-Wilk tests. As seen in this table, only the SPar results for the Bzip2 compress and decompress applications do not show a normal distribution since, for these cases, the P-values are close to zero (in bold). Therefore, we used the non-parametric Wilcoxon test to compare the results of the SPar with the results of the other interfaces for the Bzip2 compress and decompress applications. We performed a Student's t-test for two samples in all other cases. In addition, we performed paired tests as the samples were not independent since we collected the data from the same participants.

Table 7.3 shows the results for the Students' t and Wilcoxon tests. The average time to develop the applications with SPar is shorter due to the annotation-based programming model of this DSL. However, through hypothesis testing, it was possible to observe that there is no significant difference between SPar and FastFlow development times for the applications Bzip2 Compress, Bzip2 Decompress, Person Recognition, and Ferret. This was because the FastFlow development model is based on the use of templates for designing parallel patterns. The results presented in section 4.2 showed that FastFlow could provide similar productivity for developers as SPar.

Regarding Bzip2 compress and decompress applications, when we analyzed only the participants' average, SPar required less development effort. However, the hypothesis test showed no significant difference between the average development times for SPar and the others PPIs evaluated. This may have happened because the SPar samples for the Bzip compress and decompress applications do not have a normal distribution. In these

Table 7.1: Planning poker estimated development time in hours.

Application	PPIs	Dev. 1	Dev. 2	Dev. 3	Dev. 4	Dev. 5	Average
Bzip2 compress	SPar	1	1	1	1	0.75	0.95
	FastFlow	3	4	1	4	2	2.80
	TBB	3	4	2	4	2.50	3.10
	OpenMP	20	15	12	20	11	15.60
	Pthreads	20	18	15	16	12	16.20
Bzip2 decompress	SPar	1	1	1	1	0.75	0.95
	FastFlow	3	4	1	4	2	2.8
	TBB	3	4	2	4	2.50	3.1-
	OpenMP	20	15	12	20	11	15.60
	Pthreads	20	18	15	16	12	16.20
Lane Detection	SPar	1	0.50	1	1	0.75	0.85
	FastFlow	2	1	1	2	2.50	1.70
	TBB	2	1	2	2	3.33	2.07
	OpenMP	20	13	12	22	12	15.80
	Pthreads	20	15	15	14	13	15.40
Person Recognition	SPar	1	0.31	1	1	0.66	0.78
	FastFlow	2	0.75	1	3	2.50	1.85
	TBB	2	0.75	2	3	3.25	2.20
	OpenMP	20	12.50	12	22	11.50	15.60
	Pthreads	20	14.50	15	14	12.50	15.20
Ferret	SPar	6	3	2	1	2	2.80
	FastFlow	4	6	2	4	3.50	3.90
	TBB	8	6	4	4	3.75	5.15
	OpenMP	25	18	14	20	13	18
	Pthreads	25	20	17	16	14	18.40



(a) Bzip2 compress application.

(b) Bzip2 decompress application.

(c) Lane detection application.

(d) Person recognition application.

(e) Ferret application.

Figure 7.2: Box Plot for the development times collected using the planning poker method.

cases, it was necessary to use the Wilcoxon test to compare them with the results of the other interfaces. We normalized these data using the Min-Max normalization [123] to overcome this limitation. With the normalized samples, we used the t-test to compare them. Table 7.3 shows that when comparing the normalized samples, the SPar development time differs statistically from FastFlow TBB, OpenMP, and Pthreads (P-value ≤ 0.0217).

FastFlow and TBB showed close results, as shown in Figure 7.2. For all the applications evaluated, the hypothesis test showed that there is no significant difference between the development time of FastFlow and TBB, as shown in Table 7.3 (P-value > 0.05).

Table 7.2: P-value of the shapiro-wilk test for planning poker evaluation.

	Bzip2 compress	Bzip2 decompress	Lane detection	Person recog.	Ferret
	p-value	p-value	p-value	p-value	p-value
SPar	**0.0001**	**0.0001**	0.05	0.05	0.22
FastFlow	0.42	0.42	0.20	0.61	0.68
TBB	0.38	0.38	0.28	0.55	0.10
OpenMP	0.21	0.21	0.07	0.08	0.70
Pthreads	0.99	0.99	0.12	0.18	0.64

Table 7.3: P-value of the student's t-test and wilcoxon test for planning poker evaluation.

Wilcoxon	Bzip2 compress	Bzip2 decompress	Lane detection	Person recog.	Ferret
	p-value	p-value	p-value	p-value	p-value
SPar x FastFlow	**0.0975**	**0.0975**	-	-	-
SPar x TBB	**0.0579**	**0.0579**	-	-	-
SPar x OpenMP	**0.0579**	**0.0579**	-	-	-
SPar x Pthreads	**0.0625**	**0.0625**	-	-	-
T-test - P-value	Bzip2 compress	Bzip2 decompress	Lane detection	Person recog.	Ferret
	p-value	p-value	p-value	p-value	p-value
SPar x FastFlow	0.0217	0.0217	0.0434	**0.05272**	**0.3131**
SPar x TBB	0.0036	0.0036	0.0265	0.0224	0.0009
SPar x OpenMP	0.0012	0.0012	0.0021	0.0024	0.0008
SPar x Pthreads	0.0002	0.0002	0.0002	0.0003	0.0002
FastFlow x TBB	**0.2080**	**0.2080**	**0.1803**	**0.1836**	**0.1855**
FastFlow x OpenMP	0.0012	0.0012	0.0024	0.0023	0.0022
FastFlow x Pthreads	0.0003	0.0003	0.0004	0.0007	0.0013
TBB x OpenMP	0.0018	0.0018	0.0036	0.0034	0.0012
TBB x Pthreads	0.0004	0.0004	0.0006	0.0009	0.0003
OpenMP x Pthreads	**0.6657**	**0.6657**	**0.8486**	**0.8486**	**0.7572**

In addition, both interfaces require less programming effort than OpenMP and Pthreads (P-value < 0.05). These results occurred due to the programming models of both interfaces. Intel TBB [244] is an open-source and general-purpose C++ template-based PPI from the industry, which provides a Pipeline pattern constructor that can also perform as the Farm pattern. While FastFlow [5] is a representative interface from the scientific community with a model-based C++ interface similar to the TBB. However, each model has particularities, although parallelism is implemented similarly. Nevertheless, the results showed that they require the same programming effort.

OpenMP and Pthreads also presented similar results, as seen in Table 7.3. There is no significant difference in development time when comparing OpenMP with Pthreads for all applications evaluated (P-value > 0.1018). Moreover, developing the applications with OpenMP and Pthreads requires more programming effort, in the participants' opinion. The OpenMP and Pthreads interfaces have unstructured programming models, each with its characteristics. Parallelization in OpenMP is exploited through compilation directives or pragmas defined in the C and C++ standards. The Pthreads library is based on the POSIX specification that defines a set of types, functions, and macros for creating and controlling multiple threads. Therefore, they do not have parallelism patterns implemented to develop stream processing applications (e.g., Pipeline and Farm), unlike the FastFlow, TBB, and SPar PPIs. In the stream processing applications evaluation, ordering the items before sending them to the next stage was necessary. Therefore, with Pthreads and OpenMP, it was required to manually create mechanisms like ordered insertion into chained queues, increasing the development effort.

7.3 Final remarks

In this chapter, we aimed to find a way to reduce the effort devoted to collecting development time in parallel programming. For this purpose, we proposed extending the Planning Poker method to estimate the time required to develop parallel applications. It is already a well-established method in agile development, widely used and spread among agile development teams. In Chapter 5, we evaluated the accuracy of the Planning Poker method for estimating the time needed to develop parallel stream processing applications using three PPIs: FatsFLow, SPar, and TBB. Through this analysis, we concluded the method's effectiveness since the estimates made by experts in the area showed results close to those of an experiment conducted with 15 beginner in parallel programming (Section 4.2). Among the metrics evaluated, Planning Poker was the only one that met the accuracy criteria, although these values were considered acceptable for FastFlow and TBB. The potential of Poker Planning motivated us to create a methodology for its use in the parallel programming area.

This chapter evaluated the proposed methodology through a quasi-experiment with five developers experienced in developing stream processing applications. The participants used the Planning Poker method to estimate the time required to develop five parallel stream processing applications using the FastFLow, SPar, TBB, OpenMP, and Pthreads interfaces. The Planning Poker results were similar to the experiment results with 15 students for FastFlow, SPar, and TBB. The results showed that SPar is the interface that requires the least programming effort to use, followed by FastFlow and TBB. However, for the applications Bzip2 compress, Bzip2 decompress, Person Recognition, and Ferret, the hypothesis test did not show a significant difference between FastFlow and SPar development times. The same also occurred in the experiment with novice students due to how the experiment was conducted. Many participants parallelized the RGB channel extraction application with FastFlow after already parallelizing it with SPar and TBB. In this way, parallelizing with FastFlow was easier because they already knew how to parallelize the application. Therefore, developers with some experience can develop applications using FastFlow as productively as SPar.

The hypothesis test also showed no significant difference between the estimated development times for FastFlow and TBB due to the similarities in their programming models. The same occurred in the experiment with beginner developers. The hypothesis test also showed no significant difference between the estimated development times for FastFlow and TBB due to the similarities in their programming models. The same behavior also occurred for OpenMP and Pthreads, which require at least 85% more effort to parallelize the applications evaluated from the opinion of the participants of this study. OpenMP and Pthreads do not have similar programming models as FastFlow and TBB. Despite the differences between OpenMP and Pthreads, both have similar programming efforts to parallelize stream applications by not providing a structured programming model. It is necessary to implement parallelism patterns like Pipeline and Farm, which are composed of different processing stages, to parallelize stream applications. Therefore, to implement stream parallelism with Pthreads and OpenMP, it is necessary to develop mechanisms such as ordered insertion into chained queues manually. Experienced developers would only realize such features. Therefore, considering the opinion of beginner developers in the Planning Poker method could result in incorrect estimates.

Planning Poker has proven to be a promising method for estimating the time required to develop parallel applications since applying it in practice requires less effort. However, its use still has some limitations. By performing an experiment with people to collect the real-time needed to develop a given application, it is possible to collect information about the difficulties and challenges faced by them during the execution of the task. This experiment allows us to collect the participants' opinions about their satisfaction with the interfaces. A qualitative analysis of the participants' answers can also provide valuable insights, such as those presented in Chapter 4. To address this limitation, we propose using a questionnaire for the participants to report the reasons for the programming effort related to a given interface, its limitations, and problems. In addition, it is essential that participants also report their satisfaction with the evaluated PPIs.

8. CONCLUSION

In this Ph.D. thesis, we discussed opportunities, methods, and techniques to improve the evaluation of parallel programming productivity. This work began with a literature review aimed at mapping the techniques and metrics for evaluating the productivity and usability of parallel programming interfaces (Chapter 3). We found that most studies that claim to evaluate usability in parallel programming do not assess user satisfaction. Furthermore, to simplify the evaluations, most studies use only coding metrics to measure productivity rather than conducting experiments with people. We have seen that there are different metrics to evaluate coding productivity. Therefore, in Chapter 3, we also presented a classification of these metrics. Furthermore, due to the limitations found in the literature review, we introduce a methodology to guide other parallel programming researchers in experimenting with people (Chapter 4).

Chapter 4 details two studies to assess parallel programming usability using software engineering methodologies [252] for a quantitative and qualitative investigation. The scope of this study comprised three PPIs (FastFlow, SPar, and TBB) based on structured parallel programming to express parallelism in stream processing applications targeting multi-core systems. SPar showed the best usability indicators in this study because of its annotation-based programming model. In Chapter 4, we also presented an initial study with beginners developers in parallel programming for GPU systems to identify the usability of different PPIs (CUDA, GSParLib, OpenACC, and OpenCL). In this study, GSParLib showed the best usability indicators due to its programming model that abstracts the parallelism complexity by generating CUDA and OpenCL code. These findings suggest that PPIs with a higher level of abstraction can reduce the effort required to develop parallel applications.

Unlike studies that perform experiments with experienced developers on parallel programming [172, 173], the participants of the studies presented in Chapter 4 were intentionally beginner developers to consider the impact of learning, difficulties faced, and programming errors. These results may also help teach parallel programming because we identified the main challenges they faced in the study. In addition, through these studies, it was possible to test and validate the proposed methodology to facilitate the evaluation of the usability of PPIs.

Conducting experiments on people is essential to evaluate development productivity, although time-consuming. An alternative evaluation is the use of offline coding metrics. In Chapter 5, we assessed the feasibility of different coding metrics (SLOC, NOC, TOC, CCN, IFC, Halstead, and COCOMO II) when evaluating parallel applications. The results showed that Halstead and COCOMO II metrics were more promising for evaluating parallel applications, although they also have limitations. Therefore, in Chapter 5, we also tried overcoming some Halstead and COCOMO II limitations.

We proposed an approach to evaluate the development effort of parallel applications using Halstead and a refined COCOMO II reuse model version. We identified other predictive metrics that have not previously been used to estimate the development effort of parallel applications, such as FPA, Planning Poker, Putnam's model, SEER-SEM, and UCP. Aimed to evaluate such metrics, we measured their accuracy against the effort required to develop parallel stream applications. Our results showed that Planning Poker got the best result than the other estimation metrics from the accuracy evaluation (MMRE, MdMRE, and PRED values). In addition, Planning Poker showed the best results because it relies on the experts' opinions to guess the development effort.

Our results also showed that PHalstead has proven to be a helpful metric for evaluating parallel code, although it does not get the best results. PHalstead may be an alternative for studies for which it is not possible to conduct controlled experiments with students and developers of parallel applications. This tool can be easily used by users, who only have to provide the source code and the target PPI. In addition, currently, PHalstead can be used to estimate the effort to develop C and C++ applications with FastFlow, SPar, TBB, OpenMP, C++ threads and GrPPI, and Java applications with Flink and Storn.

Our adaptation of the COCOMO II reuse model (PCRM) did not prove adequate for estimating the time required to develop parallel applications due to a lack of accuracy since the estimated effort was much higher than the actual effort. Moreover, it is not easy to refine the parameters of COCOMO for the parallel application development scenario. Therefore, we identified the need to create an estimation model based on evaluating factors not considered in COCOMO.

Our initial idea was to design a model for estimating the development effort of parallel applications based on COCOMO II. For this purpose, initially, we conducted an international survey to identify the factors hindering parallel application developers' productivity (Chapter 6). This survey also identified some possible solutions based on the participants' opinions to improve developer productivity in parallel programming. Our results showed that lack of experience is one of the main reasons for the extensive effort to develop parallel applications. These results showed a gap in teaching parallel programming in the universities, where concepts related to PPIs are covered only at the end of undergraduate courses. Better PPI documentation also helps the students' learning process. Using profiling and debugging tools can help develop parallel applications more productively. Finally, the results confirm the importance of creating programming models that can abstract away the complexities of parallelization and increase developer productivity.

From the survey presented in Chapter 6, we identified some factors that impact the development time of parallel applications, including developer experience, programming model, architecture, development environment, and documentation. Using a machine learning model, we could predict development time from the factors identified by the survey. However, building a dataset containing such information for training the model is a complex task since it is necessary to perform experiments with people to collect the development time. Furthermore, due to the events of COVID-19, it was not possible to conduct more experiments to collect such data. Instead, an alternative approach was to focus on proposing improvements to coding metrics that did not require many participants to use.

Planning Poker was the most effective method for estimating parallel application development time among the coding metrics evaluated. Planning Poker's accuracy occurred because it relied on the opinions of experienced parallel application developers to make the estimates. In addition, the survey results highlight the developer's experience as one of the main factors impacting productivity. Therefore, in Chapter 7, we have proposed a modification of Planning Poker to estimate the development effort of parallel applications based on the opinion of an experienced developer instead of a development team. We performed an experiment with experienced parallel programming developers to validate the proposed methodology. The participants used the modified Planning Poker method to estimate the development time of stream processing applications. Our results showed that the Planning Poker method has promising results comparable to the actual development time collected through controlled experiments like those presented in Chapter 4. In addition, Planning Poker requires less effort to use in practice than such experiments. Therefore, we concluded that this method is a more effective and less costly alternative for measuring development time, which is the main contribution of this Ph.D. thesis.

Finally, there is potential to expand and evolve the methodologies and techniques presented above. There are many research opportunities to be explored regarding the productivity of coding parallel applications. Therefore, we discuss relevant existing limitations and opportunities for future work in section 8.1.

8.1 Limitations and future work

From the literature review, we identified some little or not yet explored parallel architectures to evaluate productivity from the literature review. Most of the studies explore multi-core environments. On the other hand, few studies assess the coding productivity of parallel application systems with cloud TPU and FPGA architectures [238, 209, 74]. Furthermore, no studies have been conducted to evaluate emerging PPIs for HPC clusters, such as HPX interfaces and Apache Spark and Flink. Hence, there is an opportunity to explore such architectures.

We presented a methodology to guide and help researchers evaluate the usability of PPIs. This methodology was used for an initial study and a controlled experiment with graduate students. These studies showed promising results. However, one of the limitations found is related to the size of the samples evaluated. It is not easy to find a representative sample of participants in the parallel programming area to perform experiments like those in this work. To try to overcome this problem, we proposed using Planning Poker. However, the problem related to sample size still needs to be solved.

On the other hand, PHalstead is a solution proposed for when it is impossible to conduct experiments with people, although it still has limitations. There is a need to design coding metrics evaluating factors other than lines of code and the number of tokens. Other factors, such as developer experience, the target architecture, the programming model, and the application domain, should also be considered when proposing a parallel application evaluation model.

By conducting experiments with people, it is possible to collect information about the difficulties and challenges faced by the participants during the activity, and their satisfaction or dissatisfaction with the interfaces used. It was not possible to collect such information from participants using the proposed modification of Planning Poker. Consequently, it was impossible to perform a qualitative analysis using Grounded Theory procedures, such as those presented in Chapters 4 and 6. To overcome this limitation, we proposed modifying this method, in which we request further information from the participant. Besides the estimated development time for a given interface, it is possible to ask the developer the reasons for this development time and the PPIs' problems and limitations. It is also possible to ask participants to report difficulties that a less experienced programmer might face during parallelization. Furthermore, it will also be possible to evaluate user satisfaction.

We have identified a model from the literature review to estimate the percentage of extra effort to develop CUDA applications [143]. Using different machine-learning algorithms, this model performs its estimations based on code metrics (Halstead, CCN, and SLOC). There needs to be documentation regarding the dataset used for training and testing the models, making it difficult to replicate the study results. In addition, this dataset comprises only the CUDA interface. We initially tried composing a dataset, including the applications evaluated in this study. Unfortunately, the experiments with people presented in this paper show a sample that needs to be more representative to compose a dataset. Designing a dataset with more PPIs is still necessary to train a machine learning model. A dataset is also essential to make it easier for researchers in the parallel programming area to propose new methods for estimating development efforts. It avoids the need to perform new experiments with students to validate these metrics, which is time-consuming. In addition, this data must be made available to the HPC community in a repository online with well-documented documentation.

We evaluated several metrics for estimating the development effort of parallel applications, among which Planning Poker was the most prominent. Agile development teams widely use this method. Several other estimation models for agile projects are available, such as t-shirt sizing, dot voting, bucket system, large/uncertain/small, and affinity mapping [140]. Therefore, the evaluation of such models in the parallel programming domain still needs to be explored.

8.2 Publications

The following are research papers published during the doctoral period that are directly related to this Ph.D. thesis:

- Andrade, G.; Griebler, D.; Santos, R.; Danelutto, M.; Fernandes, L. G. “**Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores**”. In: 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2021 [9];
- Andrade, G.; Griebler, D.; Santos, R.; Kessler, C.; Ernstsson, A.; Fernandes, L. G. “**Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing**”. In: 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2022 [12];
- Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “**Opinião de Brasileiros Sobre a Produtividade no Desenvolvimento de Aplicações Paralelas**”. In: Symposium on High Performance Computing Systems (WSCAD), 2022. [10];
- Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “**A parallel programming assessment for stream processing applications on multi-core systems**”, Computer Standards Interfaces, vol. 84, March 2023 [11].

REFERENCES

[1] Adornes, D.; Griebler, D.; Ledur, C.; Fernandes, L. G. “A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures”. In: Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering, 2015, pp. 1–6.

[2] Ahmad, W.; Carpenter, B.; Shafi, A. “Collective Asynchronous Remote Invocation (CARI): A High-Level and Effcient Communication API for Irregular Applications”, *Procedia Computer Science*, vol. 4, Jun 2011, pp. 26–35.

[3] Alameh, R.; Zazworka, N.; Hollingsworth, J. p. A. P. s. f. b. m. K. “Performance Measurement of Novice HPC Programmers Code”. In: Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications, 2007, pp. 5.

[4] Albrecht, A. J.; Gaffney, J. E. “Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation”, *IEEE Transactions on Software Engineering*, vol. 9–6, Nov 1983, pp. 639–648.

[5] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “FastFlow: High-level and Efficient Streaming on Multi-core”. In: *Chapter in Programming Multi-core and Many-core Computing Systems*, John Wiley, 2017, pp. 261–280.

[6] Aldinucci, M.; Pezzi, G. P.; Drocco, M.; Tordini, F.; Kilpatrick, P.; Torquati, M. “Parallel Video Denoising on Heterogeneous Platforms”. In: Proceedings of the 3rd Internacional Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems, 2014, pp. 1–8.

[7] Amaral, V.; Norberto, B.; Goulão, M.; Aldinucci, M.; Benkner, S.; Bracciali, A.; Carreira, P.; Celms, E.; Correia, L.; Grelck, C.; et al.. “Programming Languages for Data-Intensive HPC Applications: A Systematic Mapping Study”, *Parallel Computing*, vol. 91, Mar 2020, pp. 1–17.

[8] Andersch, M.; Chi, C. C.; Juurlink, B. “Using OpenMP Superscalar for Parallelization of Embedded and Consumer Applications”. In: Proceedings of the 12th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2012, pp. 23–32.

[9] Andrade, G.; Griebler, D.; Santos, R.; Danelutto, M.; Fernandes, L. G. “Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores”. In: Proceedings of the 47th Euromicro Conference on Software Engineering and Advanced Applications, 2021, pp. 291–295.

[10] Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “Opinião de Brasileiros Sobre a Produtividade no Desenvolvimento de Aplicações Paralelas”. In: Proceedings of the 23th Brazilian Symposium on High Performance Computing Systems, 2022, pp. 276–287.

[11] Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “A Parallel Programming Assessment for Stream Processing Applications on Multi-core Systems”, *Computer Standards & Interfaces*, vol. 84, Mar 2023, pp. 1–25.

[12] Andrade, G.; Griebler, D.; Santos, R.; Kessler, C.; Ernstsson, A.; Fernandes, L. G. “Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing”. In: Proceedings of the 48th Euromicro Conference on Software Engineering and Advanced Applications, 2022, pp. 229–232.

[13] Andrade, H. C.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 558p.

[14] Arora, R.; Bangalore, P.; Mernik, M. “Raising the Level of Abstraction for Developing Message Passing Applications”, *The Journal of Supercomputing*, vol. 59–2, Nov 2012, pp. 1079–1100.

[15] Atashpendar, A.; Dorronsoro, B.; Danoy, G.; Bouvry, P. “A Scalable Parallel Cooperative Coevolutionary PSO Algorithm for Multi-objective Optimization”, *Journal of Parallel and Distributed Computing*, vol. 112, Feb 2018, pp. 111–125.

[16] Azzeh, M.; Nassif, A. B. “A Hybrid Model for Estimating Software Project Effort from Use Case Points”, *Applied Soft Computing*, vol. 49, Dec 2016, pp. 981–989.

[17] Baek, S.; Lee, K.; Kim, J.; Morris, J. “Heterogeneous Networks of Workstations”. In: Proceedings of the 9th Asia-Pacific Conference on Advances in Computer Systems Architecture, 2004, pp. 426–439.

[18] Barbosa, S.; Silva, B. “Interação Humano-Computador”. Elsevier Brasil, 2010, 384p.

[19] Barnum, C. M. “Usability Testing Essentials: Ready, Set... Test!” Morgan Kaufmann, 2010, 408p.

[20] Barros-Justo, J. L.; Benitti, F. B.; Tiwari, S. “The Impact of Use Cases in Real-world Software Development Projects: A Systematic Mapping Study”, *Computer Standards & Interfaces*, vol. 66–103362, Dec 2019, pp. 1–16.

[21] Belcastro, L.; Marozzo, F.; Talia, D.; Trunfio, P. “A High-Level Programming Library for Mining Social Media on HPC Systems”. In: Proceedings of the 1st Future Trends of HPC in a Disruptive Scenario, 2019, pp. 3–21.

[22] Belikov, E.; Deligiannis, P.; Totoo, P.; Aljabri, M.; Loidl, H.-W. “A Survey of High-level Parallel Programming Models”, Technical report, School of Mathematical and Computer Sciences, Heriot-Watt University, 2013, 46p.

[23] Bernholdt, D. E.; Boehm, S.; Bosilca, G.; Gorentla Venkata, M.; Grant, R. E.; Naughton, T.; Pritchard, H. P.; Schulz, M.; Vallee, G. R. “A Survey of MPI Usage in the US Exascale Computing Project”, *Concurrency and Computation: Practice and Experience*, vol. 32–3, Sep 2018, pp. 1–16.

[24] Bienia, C.; Kumar, S.; Singh, J. P.; Li, K. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 72–81.

[25] Boehm, B. W. “Software Engineering Economics”. Prentice Hall PTR, 1981, 800p.

[26] Boehm, B. W.; Abts, C.; Brown, A. W.; Chulani, S.; Clark, B. K.; Horowitz, E.; Madachy, R.; Reifer, D. J.; Steece, B. “Software Cost Estimation with COCOMO II”. Prentice Hall, 2000, 544p.

[27] Bronson, G. J. “A First Book of C++”. Cengage Learning, 2011, 4th ed., 816p.

[28] Buttlar, D.; Farrell, J.; Nichols, B. “Pthreads Programming: A POSIX Standard for Better Multiprocessing”. O’Reilly Media, 1996, 288p.

[29] Caldiera, V. R. B. G.; Rombach, H. D. “The Goal Question Metric Approach”, *Encyclopedia of Software Engineering*, vol. 1, Nov 1994, pp. 528–532.

[30] Calefato, F.; Lanubile, F. “A Planning Poker Tool for Supporting Collaborative Estimation in Distributed Agile Development”. In: Proceedings of the 6th International Conference on Software Engineering Advances, 2011, pp. 14–19.

[31] Cantonnet, F.; Yao, Y.; Zahran, M.; El-Ghazawi, T. “Productivity Analysis of the UPC Language”. In: Proceedings of the 18th International Symposium on Parallel and Distributed Processing, 2004, pp. 254–260.

[32] Castor, F.; Oliveira, J. a. P.; Santos, A. L. “Software Transactional Memory vs. Locking in a Functional Language: A Controlled Experiment”. In: Proceedings of the 1st International Workshop on Transitioning to Multicore, 2011, pp. 117–122.

[33] Chakravarthy, S.; Jiang, Q. “Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing”. Springer Science & Business Media, 2009, 324p.

[34] Chan, Y. “Biostatistics 102: Quantitative Data–Parametric & Non-parametric Tests”, *Singapore Medical Journal*, vol. 44–8, Sep 2003, pp. 391–396.

[35] Cheng, J.; Grossman, M.; McKercher, T. “Professional CUDA C Programming”. John Wiley & Sons, 2014, 528p.

[36] Cho, S. M.; Im, D.; Jang, O.; Song, H. J.; Paulovicks, B.; Sheinin, V.; Yeo, H. “OpenCL and Parallel Primitives for Digital TV Applications”, *IBM Journal of Research and Development*, vol. 54–5, Sep 2010, pp. 506–519.

[37] Cid-Fuentes, J. Á.; Alvarez, P.; Amela, R.; Ishii, K.; Morizawa, R. K.; Badia, R. M. “Efficient Development of High Performance Data Analytics in Python”, *Future Generation Computer Systems*, vol. 111, Oct 2020, pp. 570–581.

[38] Coblenz, M.; Seacord, R.; Myers, B.; Sunshine, J.; Aldrich, J. “A Course-based Usability Analysis of Cilk Plus and OpenMP”. In: Proceedings of the 12th International Symposium on Visual Languages and Human-Centric Computing, 2015, pp. 245–249.

[39] Cohn, M. “Agile Estimating and Planning”. Pearson Education, 2005, 330p.

[40] Conte, D. J.; de Souza, P. S. L.; Martins, G.; Bruschi, S. M. “Teaching Parallel Programming for Beginners in Computer Science”. In: Proceedings of the 14th International Conference on Frontiers in Education, 2020, pp. 1–9.

[41] Conte, S. D.; Dunsmore, H. E.; Shen, V. Y. “Software Engineering Metrics and Models”. Benjamin-Cummings Publishing Company, 1986, 396p.

[42] Corbin, J. M.; Strauss, A. “Grounded Theory Research: Procedures, Canons and Evaluative Criteria”, *Qualitative sociology*, vol. 13–1, Mar 1990, pp. 3–21.

[43] Corral-Plaza, D.; Medina-Bulo, I.; Ortiz, G.; Boubeta-Puig, J. “A Stream Processing Architecture for Heterogeneous Data Sources in the Internet of Things”, *Computer Standards & Interfaces*, vol. 70, Jun 2020, pp. 1–13.

[44] Daga, M.; Tschirhart, Z. S.; Freitag, C. “Exploring Parallel Programming Models for Heterogeneous Computing Systems”. In: Proceedings of the 16th International Symposium on Workload Characterization, 2015, pp. 98–107.

[45] Daleiden, P.; Stefik, A.; Uesbeck, P. M. “GPU Programming Productivity in Different Abstraction Paradigms: A Randomized Controlled Trial Comparing CUDA and Thrust”, *ACM Transactions on Computing Education*, vol. 20–4, Dec 2020, pp. 1–27.

[46] Danelutto, M.; De Matteis, T.; De Sensi, D.; Mencagli, G.; Torquati, M.; Aldinucci, M.; Kilpatrick, P. “The Rephrase Extended Pattern Set for Data Intensive Parallel Computing”, *International Journal of Parallel Programming*, vol. 47–1, Nov 2019, pp. 74–93.

[47] Danis, C.; Thomas, J.; Richards, J.; Brezin, J.; Swart, C.; Halverson, C.; Bellamy, R.; Malkin, P. “Towards Applying Complexity Metrics to Measure Programmer Productivity in High Performance Computing”. In: Proceedings of the 30th International Conference on Software Engineering, 2008, pp. 1–8.

[48] De Araujo, G. A. “Data and Stream Parallelism Optimizations on GPUs”, Master’s thesis, School of Technology, Graduate Program in Computer Science, PUCRS, 2022, 111p.

[49] De França, B. B. N.; Jeronimo, H.; Travassos, G. H. “Characterizing DevOps by Hearing Multiple Voices”. In: Proceedings of the 30th Brazilian Symposium on Software Engineering, 2016, pp. 53–62.

[50] Del Rio Astorga, D.; Dolz, M. F.; Sanchez, L. M.; Blas, J. G.; García, J. D. “A C++ Generic Parallel Pattern Interface for Stream Processing”. In: Proceedings of the 16th International Conference on Algorithms and Architectures for Parallel Processing, 2016, pp. 74–87.

[51] Di Domenico, D.; Cavalheiro, G. G.; Lima, J. V. “NAS Parallel Benchmark Kernels with Python: A Performance and Programming Effort Analysis Focusing on GPUs”. In: Proceedings of the 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2022, pp. 26–33.

[52] Domínguez-Mayo, F.; Escalona, M.; Mejías, M.; Ross, M.; Staples, G. “A Quality Management Based on the Quality Model Life Cycle”, *Computer Standards & Interfaces*, vol. 34–4, Jun 2012, pp. 396–412.

[53] Dong, Y.; Yang, F. “C++ Programming”. De Gruyter, 2019, 390p.

[54] Dongarra, J.; Lastovetsky, A. L. “High Performance Heterogeneous Computing”. John Wiley & Sons, 2009, 280p.

[55] Dragicevic, S.; Celar, S.; Turic, M. “Bayesian Network Model for Task Effort Estimation in Agile Software Development”, *Journal of Systems and Software*, vol. 127, May 2017, pp. 109–119.

[56] Durán, M.; Juárez-Ramírez, R.; Jiménez, S.; Tona, C. “Taxonomy for Complexity Estimation in Agile Methodologies: A Systematic Literature Review”. In: Proceedings of the 7th International Conference on Software Engineering Research and Innovation, 2019, pp. 87–96.

[57] Ebcioglu, K.; Sarkar, V.; El-Ghazawi, T.; Urbanic, J.; Center, P. S. “An Experiment in Measuring the Productivity of Three Parallel Programming Languages”. In: Proceedings of the 3rd International Workshop on Productivity and Performance in High-End Computing, 2006, pp. 30–36.

[58] Eccles, R.; Stacey, D. A. “Understanding the Parallel Programmer”. In: Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment, 2006, pp. 1–7.

[59] Elyassami, S.; Idri, A. “Applying Fuzzy ID3 Decision Tree for Software Effort Estimation”, *International Journal of Computer Science Issues*, vol. 8–1, Nov 2011, pp. 131–138.

[60] Enmyren, J.; Kessler, C. W. “SkePU: A Multi-backend Skeleton Programming Library for Multi-GPU Systems”. In: Proceedings of the 4th International Workshop on High-level Parallel Programming and Applications, 2010, pp. 5–14.

[61] Farber, R. “Parallel Programming with OpenACC”. Morgan Kaufmann Publishers, 2016, 326p.

[62] Fenton, N.; Bieman, J. “Software Metrics: A Rigorous and Practical Approach”. CRC press, 2015, 3rd ed., 618p.

[63] Ferdinandy, B.; Guerrero-Higueras, Á. M.; Verderber, É.; Miklósi, Á.; Matellán, V. “Analysis of Users’ First Contact with High-Performance Computing: First Approach with Ethology Researchers”. In: Proceedings of the 7th International Conference on Technological Ecosystems for Enhancing Multiculturality, 2019, pp. 554–557.

[64] Fernández-Diego, M.; Méndez, E. R.; González-Ladrón-De-Guevara, F.; Abrahão, S.; Insfran, E. “An Update on Effort Estimation in Agile Software Development: A Systematic Literature Review”, *IEEE Access*, vol. 8, Sep 2020, pp. 166768–166800.

[65] Fernàndez-Fabeiro, J.; Gonzalez-Escribano, A.; Llanos, D. R. “Simplifying the Multi-GPU Programming of a Hyperspectral Image Registration Algorithm”. In: Proceedings of the 11th International Conference on High Performance Computing & Simulation, 2019, pp. 11–18.

[66] Finco, D. A. “Combinando Planning Poker e Aprendizado de Máquina para Estimar Esforço de Software”. In: Proceedings of the 5th Regional School on Software Engineering, 2021, pp. 129–138.

[67] Galorath, D. D.; Evans, M. W. “Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves”. Auerbach Publications, 2006, 576p.

[68] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley Publishing Company, 1994, 416p.

[69] Gandomani, T. J.; Faraji, H.; Radnejad, M. "Planning Poker in Cost Estimation in Agile Methods: Averaging vs. Consensus". In: Proceedings of the 5th International Conference on Knowledge Based Engineering and Innovation, 2019, pp. 66–71.

[70] Gebali, F. "Algorithms and Parallel Computing". John Wiley & Sons, 2011, 364p.

[71] Gharehchopogh, F. S.; Maleki, I.; Talebi, A. "Using Hybrid Model of Artificial Bee Colony and Genetic Algorithms in Software Cost Estimation". In: Proceedings of the 9th International Conference on Application of Information and Communication Technologies, 2015, pp. 102–106.

[72] Glasow, P. A. "Fundamentals of Survey Research Methodology", Technical report, Mitre, Washington C3 Center, 2005, 28p.

[73] Gmys, J.; Carneiro, T.; Melab, N.; Talbi, E.-G.; Tuyttens, D. "A Comparative Study of High-Productivity High-Performance Programming Languages for Parallel Metaheuristics", *Swarm and Evolutionary Computation*, vol. 57, May 2020, pp. 1–14.

[74] Gondhalekar, A.; Twomey, T.; Feng, W.-c. "On the Characterization of the Performance-Productivity Gap for FPGA". In: Proceedings of the 26th International Conference on High Performance Extreme Computing, 2022, pp. 1–8.

[75] Gordon, R. D.; Halstead, M. H. "An Experiment Comparing Fortran Programming Times with the Software Physics Hypothesis". In: Proceedings of the 4th National Conference on Computer, 1976, pp. 935–937.

[76] Gregory, K.; Miller, A. "C++ AMP". O'Reilly Media, 2012, 356p.

[77] Griebler, D. "Domain-Specific Language & Support Tool for High-Level Stream Parallelism", Ph.d. thesis, Faculdade de Informática, Graduate Program in Computer Science, PUCRS, 2016, 243p.

[78] Griebler, D.; Adornes, D.; Fernandes, L. G. "Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures". In: Proceedings of the 26th International Conference on Software Engineering & Knowledge Engineering, 2014, pp. 25–30.

[79] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. "An Embedded C++ Domain-Specific Language for Stream Parallelism". In: Parallel Computing: On the Road to Exascale, Proceedings of the 13th International Conference on Parallel Computing, 2015, pp. 317–326.

[80] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. "SPar: A DSL for High-Level and Productive Stream Parallelism", *Parallel Processing Letters*, vol. 27–1, Mar 2017, pp. 1–14.

[81] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Higher-Level Parallelism Abstractions for Video Applications with SPar”. In: Parallel Computing is Everywhere, Proceedings of the 14th International Conference on Parallel Computing, 2017, pp. 698–707.

[82] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2”, *International Journal of Parallel Programming*, vol. 47–1, Feb 2018, pp. 253–271.

[83] Griebler, D.; Vogel, A.; De Sensi, D.; Danelutto, M.; Fernandes, L. G. “Simplifying and Implementing Service Level Objectives for Stream Parallelism”, *Journal of Supercomputing*, vol. 76, Jun 2019, pp. 4603–4628.

[84] Gu, R.; Becchi, M. “A Comparative Study of Parallel Programming Frameworks for Distributed GPU Applications”. In: Proceedings of the 16th International Conference on Computing Frontiers, 2019, pp. 268–273.

[85] Guo, J.; Agrawal, G. “Achieving Performance and Programmability for Mapreduce (-Like) Frameworks”. In: Proceedings of the 25th International Conference on High Performance Computing, 2018, pp. 314–323.

[86] Habel, R.; Silber-Chaussumier, F.; Irigoin, F.; Brunet, E.; Trahay, F. “Combining Data and Computation Distribution Directives for Hybrid Parallel Programming: A Transformation System”, *International Journal of Parallel Programming*, vol. 44–6, May 2016, pp. 1268–1295.

[87] Halstead, M. H. “Elements of Software Science (Operating and Programming Systems Series)”. Elsevier, 1977, 127p.

[88] Hamdy, A. “Fuzzy Logic for Enhancing the Sensitivity of COCOMO Cost Model”, *Journal of Emerging Trends in Computing and Information Sciences*, vol. 3–9, Sep 2012, pp. 1292–1297.

[89] Hamer, P. G.; Frewin, G. D. “MH Halstead’s Software Science - A Critical Examination”. In: Proceedings of the 6th International Conference on Software Engineering, 1982, pp. 197–206.

[90] Haugen, N. C. “An Empirical Study of Using Planning Poker for User Story Estimation”. In: Proceedings of the 4th International Conference on Agile Development, 2006, pp. 1–9.

[91] Henry, S.; Kafura, D. “Software Structure Metrics Based on Information Flow”, *IEEE Transactions on Software Engineering*, vol. 7–5, Sep 1981, pp. 510–518.

[92] Herdman, J.; Gaudin, W.; McIntosh-Smith, S.; Boulton, M.; Beckingsale, D. A.; Mallinson, A.; Jarvis, S. A. “Accelerating Hydrocodes with OpenACC, OpenCL and CUDA”. In: Proceedings of the 25th International Conference on High Performance Computing, Networking, Storage, and Analysis, 2012, pp. 465–471.

[93] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R. “A Catalog of Stream Processing Optimizations”, *ACM Computing Surveys*, vol. 46–4, Mar 2014, pp. 1–34.

[94] Hochstein, L.; Basili, V. R.; Vishkin, U.; Gilbert, J. “A Pilot Study to Compare Programming Effort for Two Parallel Programming Models”, *Journal of Systems and Software*, vol. 81–11, Nov 2008, pp. 1920–1930.

[95] Hochstein, L.; Basili, V. R.; Zelkowitz, M. V.; Hollingsworth, J. K.; Carver, J. “Combining Self-reported and Automatic Data to Improve Programming Effort Measurement”, *ACM SIGSOFT Software Engineering Notes*, vol. 30–5, Sep 2005, pp. 356–365.

[96] Hochstein, L.; Carver, J.; Shull, F.; Asgari, S.; Basili, V.; Hollingsworth, J. K.; Zelkowitz, M. V. “Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers”. In: Proceedings of the 18th International Conference on Supercomputing, 2005, pp. 35.

[97] Hochstein, L.; Shull, F.; Reid, L. B. “The Role of MPI in Development Time: A Case Study”. In: Proceedings of the 21st International Conference on Supercomputing, 2008, pp. 1–10.

[98] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism Annotations for Multi-Core Frameworks”. In: Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity, 2020, pp. 48–55.

[99] Hoffmann, R. B.; Löff, J.; Griebler, D.; Fernandes, L. G. “OpenMP as Runtime for Providing High-level Stream Parallelism on Multi-cores”, *The Journal of Supercomputing*, vol. 78, Jan 2022, pp. 7655–7676.

[100] Holcomb, Z. C. “Fundamentals of Descriptive Statistics”. Routledge, 2016, 103p.

[101] Holm, H. H.; Brodtkorb, A. R.; Sætra, M. L. “GPU Computing with Python: Performance, Energy Efficiency and Usability”, *Computation*, vol. 8–1, Jan 2020, pp. 1–24.

[102] Hori, A.; Jeannot, E.; Bosilca, G.; Ogura, T.; Gerofi, B.; Yin, J.; Ishikawa, Y. “An International Survey on MPI Users”, *Parallel Computing*, vol. 108, Dec 2021, pp. 1–14.

[103] Huang, T.-W.; Guo, G.; Lin, C.-X.; Wong, M. D. “Opentimer v2: A New Parallel Incremental Timing Analysis Engine”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40–4, Apr 2020, pp. 776–789.

[104] Huang, T.-W.; Lin, D.-L.; Lin, C.-X.; Lin, Y. “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 33–6, Jun 2022, pp. 1303–1320.

[105] Huang, T.-W.; Lin, Y.; Lin, C.-X.; Guo, G.; Wong, M. D. “Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40–8, Aug 2021, pp. 1687–1700.

[106] Hueske, F.; Kalavri, V. “Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications”. O’Reilly Media, 2019, 220p.

[107] ISO. “ISO/IEC TR 9126-4:2004 – Software Engineering – Product Quality – Part 4: Quality in Use Metrics”. Source: https://www.iso.org/standard/39752.html, Jan 2023.

[108] ISO. “ISO 9241-11:2018 – Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts”. Source: https://www.iso.org/standard/63500.html, Jan 2023.

[109] ISO. “ISO/IEC 25010:2011 – Systems and Software Engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models”. Source: https://www.iso.org/standard/35733.html, Jan 2023.

[110] ISO. “ISO/TS 24541:2020 - Service Activities Relating to Drinking Water Supply, Wastewater and Stormwater Systems — Guidelines for the Implementation of Continuous Monitoring Systems for Drinking Water Quality and Operational Parameters in Drinking Water Distribution Networks”. Source: https://www.iso.org/standard/74482.html, Jan 2023.

[111] Jorgensen, M. “Experience with the Accuracy of Software Maintenance Task Effort Prediction Models”, *IEEE Transactions on software engineering*, vol. 21–8, Aug 1995, pp. 674–681.

[112] Karner, G. “Resource Estimation for Objectory Projects”, *Objective Systems SF AB*, vol. 17–1, Sep 1993, pp. 1–9.

[113] Kasim, H.; March, V.; Zhang, R.; See, S. “Survey on Parallel Programming Model”. In: Proceedings of the 7th International Conference on Network and Parallel Computing, 2008, pp. 266–275.

[114] Kaushik, A.; Verma, S.; Singh, H. J.; Chhabra, G. “Software Cost Optimization Integrating Fuzzy System and COA-Cuckoo Optimization Algorithm”, *International Journal of System Assurance Engineering and Management*, vol. 8–2, Apr 2017, pp. 1461–1471.

[115] Kennedy, J.; Eberhart, R. “Particle Swarm Optimization”. In: Proceedings of the 3rd International Conference on Neural Networks, 1995, pp. 1942–1948.

[116] Kennedy, K.; Koelbel, C.; Schreiber, R. “Defining and Measuring the Productivity of Programming Languages”, *The International Journal of High Performance Computing Applications*, vol. 18–4, Nov 2004, pp. 441–448.

[117] Kepner, J. “High Performance Computing Productivity Model Synthesis”, *The International Journal of High Performance Computing Applications*, vol. 18–4, Nov 2004, pp. 505–516.

[118] Khare, S.; Sun, H.; Gascon-Samson, J.; Zhang, K.; Gokhale, A.; Barve, Y.; Bhattacharjee, A.; Koutsoukos, X. “Linearize, Predict and Place: Minimizing the Makespan for Edge-Based Stream Processing of Directed Acyclic Graphs”. In: Proceedings of the 4th International Symposium on Edge Computing, 2019, pp. 1–14.

[119] Kirk, D. B.; Hwu, W.-m. W. “Programming Massively Parallel Processors: A Hands-on Approach”. Morgan Kaufmann, 2016, 576p.

[120] Kitchenham, B.; Charters, S. “Guidelines for Performing Systematic Literature Reviews in Software Engineering”, Technical report, School of Computer Science and Mathematics, Keele University, 2007, 57p.

[121] Laplante, P. A. “What Every Engineer Should Know About Software Engineering”. CRC Press, 2007, 328p.

[122] Laqrichi, S.; Marmier, F.; Gourc, D.; Nevoux, J. “Integrating Uncertainty in Software Effort Estimation Using Bootstrap Based Neural Networks”. In: Proceedings of the 15th International Symposium on Information Control Problems in Manufacturing, 2015, pp. 954–959.

[123] Larose, D. T.; Larose, C. D. “Discovering Knowledge in Data: An Introduction to Data Mining”. John Wiley & Sons, 2014, 2nd ed., 316p.

[124] Lee, S.; Vetter, J. S. “Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing”. In: Proceedings of the 25th International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11.

[125] Legaux, J.; Loulergue, F.; Jubertie, S. “Development Effort and Performance Trade-off inHigh-Level Parallel Programming”. In: Proceedings of the 12th International Conference on High Performance Computing & Simulation, 2014, pp. 162–169.

[126] Lei, G.; Dou, Y.; Wan, W.; Xia, F.; Li, R.; Ma, M.; Zou, D. “CPU-GPU Hybrid Accelerating the Zuker Algorithm for RNA Secondary Structure Prediction Applications”, *BioMed Central*, vol. 13–1, Jan 2012, pp. 1–11.

[127] Li, X.; Shih, P.-C.; Li, X.; Seals, C. “A Case Study of Novice Programmers on Parallel Programming Models”, *Journal of Computers*, vol. 13–5, May 2018, pp. 490–502.

[128] Li, X.; Shih, P.-C.; Overbey, J.; Seals, C.; Lim, A. “Comparing Programmer Productivity in OpenACC and CUDA: An Empirical Investigation”, *International Journal of Computer Science, Engineering and Applications*, vol. 6–5, Oct 2016, pp. 1–15.

[129] Lima, J. V. F.; Domenico, D. D. “HPSM: A Programming Framework to Exploit Multi-CPU and Multi-GPU Systems Simultaneously”, *International Journal of Grid and Utility Computing*, vol. 10–3, May 2019, pp. 201–211.

[130] Lin, C.-X.; Huang, T.-W.; Guo, G.; Wong, M. D. “An Efficient and Composable Parallel Task Programming Library”. In: Proceedings of the 23rd International Conference on High Performance Extreme Computing, 2019, pp. 1–7.

[131] Löff, J.; Hoffmann, R. B.; Pieper, R.; Griebler, D.; Fernandes, L. G. “DSParLib: A C++ Template Library for Distributed Stream Parallelism”, *International Journal of Parallel Programming*, vol. 50, Oct 2022, pp. 454–485.

[132] Low, G. C.; Jeffery, D. R. “Function Points in the Estimation and Evaluation of the Software Process”, *IEEE transactions on Software Engineering*, vol. 16–1, Jan 1990, pp. 64–71.

[133] Lynn, T.; Fox, G.; Gourinovitch, A.; Rosati, P. “Understanding the Determinants and Future Challenges of Cloud Computing Adoption for High Performance Computing”, *Future Internet*, vol. 12–8, Aug 2020, pp. 1–17.

[134] Löff, J.; Griebler, D.; Mencagli, G.; Araujo, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. “The NAS Parallel Benchmarks for Evaluating C++ Parallel Programming Frameworks on Shared-Memory Architectures”, *Future Generation Computer Systems*, vol. 125, Dec 2021, pp. 743–757.

[135] MacDonald, S.; Anvik, J.; Bromling, S.; Schaeffer, J.; Szafron, D.; Tan, K. “From Patterns to Frameworks to Parallel Programs”, *Parallel Computing*, vol. 28–12, Dec 2002, pp. 1663–1683.

[136] Mahnič, V.; Hovelja, T. “On Using Planning Poker for Estimating User Stories”, *Journal of Systems and Software*, vol. 85–9, Sep 2012, pp. 2086–2095.

[137] Mahnic, V.; Rozanc, I. “Students’ Perceptions of Scrum Practices”. In: Proceedings of the 35th International Convention on Information and Communication Technology, Electronics and Microelectronics, 2012, pp. 1178–1183.

[138] Maiterth, M.; Koenig, G.; Pedretti, K.; Jana, S.; Bates, N.; Borghesi, A.; Montoya, D.; Bartolini, A.; Puzovic, M. “Energy and Power Aware Job Scheduling and Resource Management: Global Survey – Initial Analysis”. In: Proceedings of the 32nd International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2018, pp. 685–693.

[139] Malik, M.; Li, T.; Sharif, U.; Shahid, R.; El-Ghazawi, T.; Newby, G. “Productivity of GPUs Under Different Programming Paradigms”, *Concurrency and computation: practice and experience*, vol. 24–2, Dec 2011, pp. 179–191.

[140] Mallidi, R. K.; Sharma, M. “Study on Agile Story Point Estimation Techniques and Challenges”, *International Journal of Computer Applications*, vol. 174–13, Jan 2021, pp. 9—14.

[141] Mansor, Z. B.; Kasirun, Z. M.; Arshad, N. H. H.; Yahya, S. “E-Cost Estimation Using Expert Judgment and COCOMO II”. In: Proceedings of the 4th International Symposium on Information Technology, 2010, pp. 1262–1267.

[142] Manzano, J. B.; Zhang, Y.; Gao, G. R. “P3I: The Delaware Programmability, Productivity and Proficiency Inquiry”. In: Proceedings of the 2nd International Workshop on Software Engineering for High Performance Computing System Applications, 2005, pp. 32–36.

[143] Marantos, C.; Papadopoulos, L.; Tsintzira, A.-A.; Ampatzoglou, A.; Chatzigeorgiou, A.; Soudris, D. “Decision Support for GPU Acceleration by Predicting Energy Savings and Programming Effort”, *Sustainable Computing: Informatics and Systems*, vol. 34, Apr 2022, pp. 1–13.

[144] Martineau, M.; McIntosh-Smith, S.; Gaudin, W. “Assessing the Performance Portability of Modern Parallel Programming Models Using TeaLeaf”, *Concurrency and Computation: Practice and Experience*, vol. 29–15, Mar 2017, pp. 1–15.

[145] Martínez, M. A.; Fraguela, B. B.; Cabaleiro, J. C. “A Parallel Skeleton for Divide-and-Conquer Unbalanced and Deep Problems”, *International Journal of Parallel Programming*, vol. 49–6, May 2021, pp. 820–845.

[146] Martínez, M. A.; Fraguela, B. B.; Cabaleiro, J. C. “A Highly Optimized Skeleton for Unbalanced and Deep Divide-and-Conquer Algorithms on Multi-core Clusters”, *The Journal of Supercomputing*, vol. 78, Jan 2022, pp. 10434–10454.

[147] Martínez, P. A.; Bernabé, G.; García, J. M. “HDNN: A Cross-Platform MLIR Dialect for Deep Neural Networks”, *The Journal of Supercomputing*, vol. 78–11, Mar 2022, pp. 13814–13830.

[148] Matsuba, H.; Matsuda, M.; Kawai, M. “Pyne: A Programming Framework for Parallel Simulation Development”. In: Proceedings of the 48th International Conference on Parallel Processing: Workshops, 2019, pp. 1–10.

[149] Mattson, T. G.; He, Y.; Koniges, A. E. “The OpenMP Common Core: Making OpenMP Simple Again”. MIT Press, 2019, 320p.

[150] Mattson, T. G.; Sanders, B.; Massingill, B. “Patterns for Parallel Programming”. Pearson Education, 2004, 384p.

[151] McCabe, T. J. “A Complexity Measure”, *IEEE Transactions on Software Engineering*, vol. SE-2–4, Dec 1976, pp. 308–320.

[152] McCool, M.; Reinders, J.; Robison, A. “Structured Parallel Programming: Patterns for Efficient Computation”. Morgan Kaufmann Publishers, 2012, 406p.

[153] Meade, A.; Deeptimahanti, D. K.; Buckley, J.; Collins, J. “An Empirical Study of Data Decomposition for Software Parallelization”, *Journal of Systems and Software*, vol. 125, Mar 2017, pp. 401–416.

[154] Meade, A.; Deeptimahanti, D. K.; Johnston, M.; Buckley, J.; Collins, J. “Data Decomposition for Code Parallelization in Practice: What Do the Experts Need?” In: Proceedings of the 10th International Conference on High Performance Computing and Communications, 2013, pp. 754–761.

[155] Memeti, S.; Li, L.; Pllana, S.; Kołodziej, J.; Kessler, C. “Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption”. In: Proceedings of the 4th International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, 2017, pp. 1–6.

[156] Menzinsky, A. “¿Cuáles son los Beneficios de la Estimación con Planning Poker?” Source: https://scrum.menzinsky.com/2017/08/cuales-son-los-beneficios-de-la.html, Mar 2023.

[157] Michailidis, P. D.; Margaritis, K. G. “Scientific Computations on Multi-Core Systems Using Different Programming Frameworks”, *Applied Numerical Mathematics*, vol. 104, Jun 2016, pp. 62–80.

[158] Miller, J.; Arenaz, M. “Measuring the Impact of HPC Training”. In: Proceedings of the 6th International Workshop on Education for High-Performance Computing, 2019, pp. 58–67.

[159] Miller, J.; Wienke, S.; Schlottke-Lakemper, M.; Meinke, M.; Müller, M. S. “Applicability of the Software Cost Model COCOMO II to HPC Projects”, *International Journal of Computational Science and Engineering*, vol. 17–3, Oct 2018, pp. 283–296.

[160] Molitorisz, K.; Müller, T.; Tichy, W. F. “Patty: A Pattern-based Parallelization Tool for the Multicore Age”. In: Proceedings of the 6th International Workshop on Programming Models and Applications for Multicores and Manycores, 2015, pp. 153–163.

[161] Moløkken-Østvold, K.; Haugen, N. C. “Combining Estimates with Planning Poker–An Empirical Study”. In: Proceedings of the 18th Australian Conference on Software Engineering, 2007, pp. 349–358.

[162] Moløkken-Østvold, K.; Haugen, N. C.; Benestad, H. C. “Using Planning Poker for Combining Expert Estimates in Software Projects”, *Journal of Systems and Software*, vol. 81–12, Dec 2008, pp. 2106–2117.

[163] Moore, G. “Cramming More Components onto Integrated Circuit”, *Electronics Magazine*, vol. 38–8, Apr 1965, pp. 114–117.

[164] Moreton-Fernandez, A.; Gonzalez-Escribano, A. “Automatic Runtime Calculation of Communications for Data-parallel Expressions with Periodic Conditions”, *Concurrency and Computation: Practice and Experience*, vol. 31–5, Mar 2019, pp. 1–13.

[165] Müller, M.; Aoki, T. “Hybrid Fortran: High Productivity GPU Porting Framework Applied to Japanese Weather Prediction Model”. In: Proceedings of the 4th International Workshop on Accelerator Programming Using Directives, 2018, pp. 20–41.

[166] Munshi, A.; Gaster, B.; Mattson, T. G.; Ginsburg, D. “OpenCL Programming Guide”. Pearson Education, 2011, 648p.

[167] Naik, N. G.; Divya, T. “Planning Poker Tool for Story Point Estimation”, *RV Journal of Science Technology Engineering Arts and Management*, vol. 3–2, Jul 2022, pp. 82–92.

[168] Nakao, M.; Odajima, T.; Murai, H.; Tabuchi, A.; Fujita, N.; Hanawa, T.; Boku, T.; Sato, M. “Evaluation of XcalableACC with Tightly Coupled Accelerators/InfiniBand Hybrid Communication on Accelerated Cluster”, *The International Journal of High Performance Computing Applications*, vol. 33–5, Jan 2019, pp. 869–884.

[169] Nanthaamornphong, A. “A Pilot Study: Design Patterns in Parallel Program Development”. In: Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, 2013, pp. 17–20.

[170] Nanz, S.; Furia, C. A. “A Comparative Study of Programming Languages in Rosetta Code”. In: Proceedings of the 37th International Conference on Software Engineering, 2015, pp. 778–788.

[171] Nanz, S.; Torshizi, F.; Pedroni, M.; Meyer, B. “Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages”, *Information and Software Technology*, vol. 55–7, Jul 2013, pp. 1304–1315.

[172] Nanz, S.; West, S.; Da Silveira, K. S. “Examining the Expert Gap in Parallel Programming”. In: Proceedings of the 19th International Conference on Parallel and Distributed Computing, 2013, pp. 434–445.

[173] Nanz, S.; West, S.; Da Silveira, K. S.; Meyer, B. “Benchmarking Usability and Performance of Multicore Languages”. In: Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 183–192.

[174] Narayanan, V.; Kavitha, R.; Srikanth, R. “Performance Evaluation of Brahmagupta-Bhaskara Equation Based Algorithm Using OpenMP”. In: Proceedings of the 2nd International Conference on Data Analytics and Management, 2022, pp. 21–28.

[175] Nassif, A. B.; Azzeh, M.; Capretz, L. F.; Ho, D. “A Comparison Between Decision Trees and Decision Tree Forest Models for Software Development Effort Estimation”. In: Proceedings of the 3rd International Conference on Communications and Information Technology, 2013, pp. 220–224.

[176] Nassif, A. B.; Capretz, L. F.; Ho, D.; Azzeh, M. “A Treeboost Model for Software Effort Estimation Based on Use Case Points”. In: Proceedings of the 11th International Conference on Machine Learning and Applications, 2012, pp. 314–319.

[177] Nicolini, M.; Miller, J.; Wienke, S.; Schlottke-Lakemper, M.; Meinke, M.; Müller, M. S. “Software Cost Analysis of GPU-Accelerated Aeroacoustics Simulations in C++ with OpenACC”. In: Proceedings of the 31st International Conference on High Performance Computing, 2016, pp. 524–543.

[178] Nozal, R.; Bosque, J. L.; Beivide, R. “EngineCL: Usability and Performance in Heterogeneous Computing”, *Future Generation Computer Systems*, vol. 107, Jun 2020, pp. 522–537.

[179] NVIDIA. “CUDA C++ Programming Guide 12.1”. Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/, Feb 2023.

[180] Nyhoff, L. “Programming in C++ for Engineering and Science”. CRC Press, 2012, 730p.

[181] Okur, S.; Dig, D. “How do Developers use Parallel Libraries?” In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11.

[182] OpenCV. “Creating a video with OpenCV”. Source: https://docs.opencv.org/2.4/doc/tutorials/highgui/video-write/video-write.html, Aug 2020.

[183] Ottenstein, L. M.; Schneider, V. B.; Halstead, M. H. “Predicting the Number of Bugs Expected in a Program Module”, Technical report, Computer Sciences Department, Purdue University, 1976, 20p.

[184] O’Donncha, F.; Iakymchuk, R.; Akhriev, A.; Gschwandtner, P.; Thoman, P.; Heller, T.; Aguilar, X.; Dichev, K.; Gillan, C.; Markidis, S.; et al.. “AllScale Toolchain Pilot Applications: PDE Based Solvers Using a Parallel Development Environment”, *Computer Physics Communications*, vol. 251–107089, Jun 2020, pp. 1–10.

[185] Pacheco, P. S. “Introduction to Parallel Programming”. Morgan Kaufmann, 2011, 370p.

[186] Pankratius, V. “Automated Usability Evaluation of Parallel Programming Constructs (NIER Track)”. In: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 936–939.

[187] Pankratius, V.; Adl-Tabatabai, A.-R. “Software Engineering with Transactional Memory Versus Locks in Practice”, *Theory of Computing Systems*, vol. 55–3, Mar 2014, pp. 555–590.

[188] Pankratius, V.; Jannesari, A.; Tichy, W. F. “Parallelizing Bzip2: A Case Study in Multicore Software Engineering”, *IEEE Software*, vol. 26–6, Nov-Dec 2009, pp. 70–77.

[189] Pankratius, V.; Schmidt, F.; Garreton, G. “Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java”. In: Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 123–133.

[190] Papadopoulos, L.; Soudris, D.; Kessler, C.; Ernstsson, A.; Ahlqvist, J.; Vasilas, N.; Papadopoulos, A. I.; Seferlis, P.; Prouveur, C.; Haefele, M.; Thibault, S.; Salamanis, A.; Ioakimidis, T.; Kehagias, D. “EXA2PRO: A Framework for High Development

Productivity on Heterogeneous Computing Systems”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 33–4, Apr 2021, pp. 792–804.

[191] Patel, I.; Gilbert, J. R. “An Empirical Study of the Performance and Productivity of two Parallel Programming Models”. In: Proceedings of the 22nd International Symposium on Parallel and Distributed Processing, 2008, pp. 1–7.

[192] Peccerillo, B.; Bartolini, S. “PHAST - A Portable High-Level Modern C++ Programming Library for GPUs and Multi-Cores”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 30–1, Jan 2018, pp. 174–189.

[193] Peccerillo, B.; Bartolini, S. “Single-source Library for Enabling Seamless Assignment of Data-parallel Task-DAGs to CPUs and GPUs in Heterogeneous Architectures”. In: Proceedings of the 10th and 8th International Workshops on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms, 2019, pp. 1–4.

[194] Peccerillo, B.; Bartolini, S. “Task-DAG Support in Single-Source PHAST Library: Enabling Flexible Assignment of Tasks to CPUs and GPUs in Heterogeneous Architectures”. In: Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, 2019, pp. 91–100.

[195] Peccerillo, B.; Bartolini, S. “Flexible Task-DAG Management in PHAST Library: Data-Parallel Tasks and Orchestration Support for Heterogeneous Systems”, *Concurrency and computation: Practice and experience*, vol. 34–2, Jan 2022, pp. 1–20.

[196] Peccerillo, B.; Bartolini, S.; Koç, Ç. K. “Parallel Bitsliced AES through PHAST: A Single-Source High-Performance Library for Multi-cores and GPUs”, *Journal of Cryptographic Engineering*, vol. 9–2, Oct 2019, pp. 159–171.

[197] Pennycook, S. J.; Sewall, J. D.; Hammond, J. R. “Evaluating the Impact of Proposed OpenMP 5.0 Features on Performance, Portability and Productivity”. In: Proceedings of the 1st International Workshop on Performance, Portability and Productivity in HPC, 2018, pp. 37–46.

[198] Port, D.; Korte, M. “Comparative Studies of the Model Evaluation Criterions MMRE and PRED in Software Cost Estimation Research”. In: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement, 2008, pp. 51–60.

[199] Power, K. “Using Silent Grouping to Size User Stories”. In: Proceedings of the 12th International Conference on Agile Software Development, 2011, pp. 60–72.

[200] Pressman, R. S. “Engenharia de Software”. Pearson Makron Books, 1995, 3rd ed., 1056p.

[201] Putnam, L. H.; Myers, W. “Measures for Excellence: Reliable Software on Time, within Budget”. Prentice Hall PTR, 1991, 400p.

[202] Raith, F.; Richter, I.; Lindermeier, R.; Klinker, G. “Identification of Inaccurate Effort Estimates in Agile Software Development”. In: Proceedings of the 20th Asia-Pacific Conference on Software Engineering, 2013, pp. 67–72.

[203] Raju, H.; Krishnegowda, Y. “Software Sizing and Productivity with Function Points”, *Lecture Notes on Software Engineering*, vol. 1–2, May 2013, pp. 204–208.

[204] Rauber, T.; Rünger, G. “Parallel Programming: For Multicore and Cluster Systems”. Springer, 2013, 2nd ed., 532p.

[205] Razali, N. M.; Wah, Y. B.; et al.. “Power Comparisons of Shapiro-wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests”, *Journal of Statistical Modeling and Analytics*, vol. 2–1, Jan 2011, pp. 21–33.

[206] Reddy, P.; Sudha, K.; Sree, P. R.; Ramesh, S. N. S. V. S. C. “Software Effort Estimation using Radial Basis and Generalized Regression Neural Networks”, *Journal of Computing*, vol. 2–5, Jul 2010, pp. 87–92.

[207] Reinders, J. “Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism”. O’Reilly Media, 2007, 334p.

[208] Rockenbach, D. A. “High-Level Programming Abstractions for Stream Parallelism on GPUs”, Master’s thesis, School of Technology, Graduate Program in Computer Science, PUCRS, 2020, 163p.

[209] Rodriguez-Canal, G.; Torres, Y.; Andújar, F. J.; Gonzalez-Escribano, A. “Efficient Heterogeneous Programming with FPGAs using the Controller Model”, *The Journal of Supercomputing*, vol. 77–12, May 2021, pp. 13995–14010.

[210] Rosenberg, J. “Some Misconceptions about Lines of Code”. In: Proceedings of the 4th International Symposium on Software Metrics, 1997, pp. 137–142.

[211] Rossbach, C. J.; Hofmann, O. S.; Witchel, E. “Is Transactional Programming Actually Easier?” In: Proceedings of the 15th International Symposium on Principles and Practice of Parallel Programming, 2010, pp. 47–56.

[212] Rubert, M.; Farias, K. “On the Effects of Continuous Delivery on Code Quality: A Case Study in Industry”, *Computer Standards & Interfaces*, vol. 81, Apr 2022, pp. 1–20.

[213] Sadowski, C.; Yi, J. “User Evaluation of Correctness Conditions: A Case Study of Cooperability”. In: Proceedings of the 2nd International Workshop on Evaluation and Usability of Programming Languages and Tools, 2010, pp. 1–6.

[214] Sakdhnagool, P.; Sabne, A.; Eigenmann, R. “Comparative Analysis of Coprocessors”, *Concurrency and Computation: Practice and Experience*, vol. 31–e4756, Jan 2019, pp. 1–13.

[215] Saleem, M. A.; Ahmad, R.; Alyas, T.; Idrees, M.; Farooq, A.; Khan, A. S.; Ali, K.; et al.. “Systematic Literature Review of Identifying Issues in Software Cost Estimation Techniques”, *International Journal of Advanced Computer Science and Applications*, vol. 10–8, Jan 2019, pp. 341–346.

[216] Salisbury, L. “Inspec on Two Platforms: Elsevier’s Engineering Village and Clarivate Analytics’ Web of Science”, *The Charleston Advisor*, vol. 20–3, Jan 2019, pp. 5–13.

[217] Sarro, F.; Petrozziello, A. “Linear Programming as a Baseline for Software Effort Estimation”, *ACM Transactions on Software Engineering and Methodology*, vol. 27–3, Sep 2018, pp. 1–28.

[218] Schlagkamp, S.; da Silva, R. F.; Renker, J.; Rinkenauer, G. “Analyzing Users in Parallel Computing: A User-Oriented Study”. In: Proceedings of the 13th International Conference on High Performance Computing and Simulation, 2016, pp. 395–402.

[219] Schlagkamp, S.; Renker, J. “Acceptance of Waiting Times in High Performance Computing”. In: Proceedings of the 17th International Conference on Human-Computer Interaction, 2015, pp. 709–714.

[220] Schlebusch, F.; Müller, Y.; Wienke, S.; Miller, J.; Müller, M. S. “PInT: Pattern Instrumentation Tool for Analyzing and Classifying HPC Applications”. In: Proceedings of the 5th International Workshop on the LLVM Compiler Infrastructure in HPC, 2018, pp. 71–80.

[221] Schmitz, A.; Miller, J.; Trümper, L.; Müller, M. S. “PPIR: Parallel Pattern Intermediate Representation”. In: Proceedings of the 6th International Workshop on Hierarchical Parallelism for Exascale Computing, 2021, pp. 30–40.

[222] Schneider, S.; Hirzel, M.; Gedik, B.; Wu, K.-L. “Safe Data Parallelism for General Streaming”, *IEEE Transactions on Computers*, vol. 64–2, Feb 2013, pp. 504–517.

[223] Sharma, P.; Sangal, A. L. “Building a Hierarchical Structure Model of Enablers that Affect the Software Process Improvement in Software SMEs–A Mixed Method Approach”, *Computer Standards & Interfaces*, vol. 66–103350, Oct 2019, pp. 1–23.

[224] Sheskin, D. J. “Handbook of Parametric and Nonparametric Statistical Procedures”. CRC Press, 2004, 3rd ed., 1016p.

[225] Singh, A.; Schaeffer, J.; Szafron, D. “Experience with Parallel Programming using Code Templates”, *Concurrency: Practice and Experience*, vol. 10–2, Dec 1998, pp. 91–120.

[226] Snir, M.; Bader, D. A. “A Framework for Measuring Supercomputer Productivity”, *The International Journal of High Performance Computing Applications*, vol. 18–4, Nov 2004, pp. 417–432.

[227] Soni, V.; Hadjadj, A.; Roussel, O.; Moebs, G. “Parallel Multi-core and Multi-processor Methods on Point-value Multiresolution Algorithms for Hyperbolic Conservation Laws”, *Journal of Parallel and Distributed Computing*, vol. 123, Jan 2019, pp. 192–203.

[228] Souza, E. T. B.; Conte, T. “Estimativa de Projetos de Aplicativos Móveis: Um Mapeamento Sistemático da Literatura”. In: Proceedings of the 16th Brazilian Symposium on Software Quality, 2017, pp. 206–220.

[229] Speyer, G.; Freed, N.; Akis, R.; Stanzione, D.; Mack, E. “Paradigms for Parallel Computation”. In: Proceedings of the 20th International Conference on DoD High Performance Computing Modernization Program Users Group, 2008, pp. 486–494.

[230] Spiliotis, I. M.; Bekakos, M. P.; Boutalis, Y. S. “Parallel Implementation of the Image Block Representation using OpenMP”, *Journal of Parallel and Distributed Computing*, vol. 137, Mar 2020, pp. 134–147.

[231] Stojanovic, N.; Stojanovic, D. “Parallelizing Multiple Flow Accumulation Algorithm using CUDA and OpenACC”, *International Journal of Geo-Information*, vol. 8–9, Sep 2019, pp. 386.

[232] Sudarmaningtyas, P.; Mohamed, R. B. “Extended Planning Poker: A Proposed Model”. In: Proceedings of the 7th International Conference on Information Technology, Computer, and Electrical Engineering, 2020, pp. 179–184.

[233] Szafron, D.; Schaeffer, J. “An Experiment to Measure the Usability of Parallel Programming Systems”, *Concurrency: Practice and Experience*, vol. 8–2, Mar 1996, pp. 147–166.

[234] Tamrakar, R.; Jørgensen, M. “Does the Use of Fibonacci Numbers in Planning Poker Affect Effort Estimates?” In: Proceedings of the 16th International Conference on Evaluation & Assessment in Software Engineering, 2012, pp. 228–232.

[235] Teijeiro, C.; Taboada, G. L.; Tourino, J.; Fraguela, B. B.; Doallo, R.; Mallón, D. A.; Gómez, A.; Mourino, J. C.; Wibecan, B. "Evaluation of UPC Programmability Using Classroom Studies". In: Proceedings of the 3rd International Conference on Partitioned Global Address Space Programing Models, 2009, pp. 1–7.

[236] Tejedor, E.; Farreras, M.; Grove, D.; Badia, R. M.; Almasi, G.; Labarta, J. "A High-Productivity Task-Based Programming Model for Clusters", *Concurrency and Computation: Practice and Experience*, vol. 24–18, Mar 2012, pp. 2421–2448.

[237] Thies, W.; Amarasinghe, S. "An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design". In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 365–376.

[238] Thomas, J.; Hanrahan, P.; Zaharia, M. "Fleet: A Framework for Massively Parallel Streaming on FPGAs". In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 639–651.

[239] Turaga, D.; Andrade, H.; Gedik, B.; Venkatramani, C.; Verscheure, O.; Harris, J. D.; Cox, J.; Szewczyk, W.; Jones, P. "Design Principles for Developing Stream Processing Applications", *Software: Practice and Experience*, vol. 40–12, Aug 2010, pp. 1073–1104.

[240] Van Amesfoort, A. S.; Varbanescu, A. L.; Sips, H. J.; Van Nieuwpoort, R. V. "Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels". In: Proceedings of the 6th International Conference on Computing Frontiers, 2009, pp. 207–216.

[241] Venkataiah, V.; Mohanty, R.; Pahariya, J.; Nagaratna, M. "Application of Ant Colony Optimization Techniques to Predict Software Cost Estimation". In: Proceedings of the 3rd International Conference on Computer and Communication Technologies, 2017, pp. 315–325.

[242] Venkataiah, V.; Ramakanta, M.; Nagaratna, M. "Review on Intelligent and Soft Computing Techniques to Predict Software Cost Estimation", *International Journal of Applied Engineering Research*, vol. 12–22, Jul–Dec 2017, pp. 12665–12681.

[243] Vera, T.; Ochoa, S. F.; Perovich, D. "Survey of Software Development Effort Estimation Taxonomies", Technical report, Computer Science Department, University of Chile, 2017, 27p.

[244] Voss, M.; Asenjo, R.; Reinders, J. "Pro TBB: C++ Parallel Programming with Threading Building Blocks". Apress, 2019, 754p.

[245] Wazlawick, R. “Engenharia de Software: Conceitos e Práticas”. Elsevier, 2013, 360p.

[246] Wei, Y.; Wang, Y.; Cai, L.; Tang, W.; Wang, B.; Ethier, S.; See, S.; Lin, J. “Performance and Portability Studies with OpenACC Accelerated Version of GTC-P”. In: Proceedings of the 17th International Conference on Parallel and Distributed Computing, Applications and Technologies, 2016, pp. 13–18.

[247] Wienke, S.; Cramer, T.; Müller, M. S.; Schulz, M. “Quantifying Productivity–Towards Development Effort Estimation in HPC”. In: Proceedings of the 27th International Conference on High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–4.

[248] Wienke, S.; Miller, J.; Schulz, M.; Müller, M. S. “Development Effort Estimation in HPC”. In: Proceedings of the 28th International Conference on High Performance Computing, Networking, Storage and Analysis, 2016, pp. 107–118.

[249] Wienke, S.; Müller, M. S.; et al.. “Accelerators for Technical Computing: Is It Worth the Pain? A TCO Perspective”. In: Proceedings of the 28th International Conference on Supercomputing, 2013, pp. 330–342.

[250] Wienke, S.; Springer, P.; Terboven, C.; an Mey, D. “OpenACC – First Experiences with Real-World Applications”. In: Proceedings of the 18th European Conference on Parallel Processing, 2012, pp. 859–870.

[251] Wohlin, C. “Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering”. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, 2014, pp. 1–10.

[252] Wohlin, C.; Runeson, P.; Høst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. “Experimentation in Software Engineering”. Springer Science & Business Media, 2012, 236p.

[253] Wu, B.; Shen, X. “Software-Level Task Scheduling on GPUs”. In: *Chapter in Advances in GPU Research and Practice*, Morgan Kaufmann Publishers, 2017, pp. 83–103.

[254] Wu, S.; Dong, X.; Wang, Y.; Chen, W. “Language Constructs and Semantics for Runtime-independent Parallelism Expression on Heterogeneous Systems”. In: Proceedings of the 5th International Conference on Computer and Communications, 2019, pp. 1269–1275.

[255] Zahraoui, H.; Idrissi, M. A. J. “Adjusting Story Points Calculation in Scrum Effort & Time Estimation”. In: Proceedings of the 10th International Conference on Intelligent Systems: Theories and Applications, 2015, pp. 1–8.

[256] Zare, F.; Zare, H. K.; Fallahnezhad, M. S. “Software Effort Estimation Based on the Optimal Bayesian Belief Network”, *Applied Soft Computing*, vol. 49, Dec 2016, pp. 968–980.

[257] Zelkowitz, M.; Basili, V.; Asgari, S.; Hochstein, L.; Hollingsworth, J.; Nakamura, T. “Measuring Productivity on High Performance Computers”. In: Proceedings of the 11th International Symposium on Software Metrics, 2005, pp. 1–10.

[258] Zhang, M.; Hochstein, L. “Fitting a Workflow Model to Captured Development Data”. In: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, 2009, pp. 179–190.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br