2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) | 978-1-6654-2705-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEAA53835.2021.00044

Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores

Gabriella Andrade*, Dalvan Griebler*†, Rodrigo Santos ‡, Marco Danelutto§, Luiz G. Fernandes*
*School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.
†Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil.
‡Department of Applied Informatics, Federal University of the State of Rio de Janeiro (UNIRIO), Rio de Janeiro, Brazil.
§Department of Computer Science, University of Pisa, Pisa, Italy
Email: gabriella.andrade@edu.pucrs.br, {dalvan.griebler,luiz.fernandes}@pucrs.br, rps@uniriotec.br, marcod@di.unipi.it

***Abstract*—From the popularization of multi-core architectures, several parallel APIs have emerged, helping to abstract the programming complexity and increasing productivity in application development. Unfortunately, only a few research efforts in this direction managed to show the usability pay-back of the programming abstraction created, because it is not easy and poses many challenges for conducting empirical software engineering. We believe that coding metrics commonly used in software engineering code measurements can give useful indicators on the programming effort of parallel applications and APIs. These metrics were designed for general purposes without considering the evaluation of applications from a specific domain. In this study, we aim to evaluate the feasibility of seven coding metrics to be used in the parallel programming domain. To do so, five stream processing applications implemented with different parallel APIs for multi-cores were considered. Our experiments have shown COCOMO II is a suitable model for evaluating the productivity of different parallel APIs targeting multi-cores on stream processing applications while other metrics are restricted to the code size.**

***Index Terms*—Development effort; Parallel computing; Parallel software; Productivity; Software metrics.**

I. Introduction

Multi-core architectures emerged in the 2000s due to the semiconductor industry's inability to increase the speed of the central processing units (CPU) using traditional approaches [1]. From that moment on, multi-core CPUs have become popular, being used from smartphones to servers. Hence, all software products should be developed using parallel programming techniques to exploit the processing power of these architectures, which integrate multiple processing cores into a single chip [2]. Developing parallel applications is a complex task because programmers must take into account various aspects in the development process, such as implementing data synchronization, dividing the computation problem among threads, exploiting concurrency, and providing low-level hardware optimization [3]. To ease such development, new parallel APIs (Application Programming Interface) have been created, which provide abstractions to release the developers from dealing with lower-level implementations.

Coding productivity is an important factor that together with effectiveness and user satisfaction are indicators of usability [4]. Based on usability or productivity indicators, it is possible to give improvement indicators for designing new parallel APIs and refine existing ones. However, obtaining usability indicators is time consuming, because empirical studies must be conducted with people. Also, an experiment must be planned and executed, and finding a representative sample of participants in parallel programming domain is quite a challenge. Coding metrics may be not the ultimate 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 [5]–[14].

Different off-line coding metrics have been used to assess parallel APIs over the years. In [6], the effort of novice programmers to develop parallel applications in MPI (Message Passage Interface) and OpenMP (Open Multi-Processing) was assessed using LOC (Lines of Code), and cost per LOC. UPC (Unified Parallel C) was compared to MPI in [5], using LOC, NOC (Number of Characters), and CCN (Cyclomatic Complexity Number). In [9], Chapel, Cilk, Go and TBB (Threading Building Blocks) were compared in the development of six benchmarks by expert and novice programmers using LOC and coding time. In [11], 7,087 programs developed in C, Go, C#, Java, F#, Haskell, Python, and Ruby were analyzed considering LOC and executable size. In [8], 556 open-source applications developed with TPL (Task Parallel Library) and PLINQ (Parallel Language Integrated Query) were analyzed using LOC and the number of parallel directives.

The effort to develop a GPU-based program using MPI and Hitmap was analyzed using LOC, TOC (Tokens of Code), CCN, and Halstead's measures in [14]. In [13], the programmability of Pthreads (POSIX Threads), TBB, FastFlow, and SPar (Stream Parallelism) to implement three real-world streaming applications was assessed using LOC and CCN. In [7], the effort to parallelize the Bzip2 compression program using OpenMP and Pthreads was compared using total, modified, added, and removed LOCs, LOCs with parallel constructs, and coding time. In [10], the DSL-POPP (Domain-Specific Language for Pattern-Oriented Parallel Programming) was compared to Pthreads using LOC and COCOMO II (Constructive Cost Model II). In [12], COCOMO II was used to evaluated the efforts of novice and professional developers to implement projects with OpenACC on GPUs.

Although coding metrics provide productivity indicators, they are designed for general purpose without considering in their evaluation factors that impact the development of parallel applications, such as the programming model and architecture. Our work aims to identify whether existing coding metrics provide useful insights for the parallel programming domain as

978-1-6654-2705-0/21/$31.00 ©2021 IEEE
DOI 10.1109/SEAA53835.2021.00044

well as highlight their limitations. So important as performance are productivity and usability analysis for parallel APIs. New initiatives are need to minimize the lack of coding metric analysis available in the literature of existing parallelism abstractions. This study contributes for evaluating the use of coding metrics when introducing parallelism in stream processing applications executing on multi-core systems.

II. Coding Metrics

Coding productivity measures the amount of resources used to develop an application, such as time and effort spent by developer [4]. Some metrics and tools allow software analysts and project managers to obtain indicators of coding productivity. LOC is a commonly used metric, which considers the total number of lines in the code, removing blank lines and comments [15]. In general, the objective is to relate the project size to the effort required to develop it. However, using only LOC metric itself can provide false or inaccurate results, because some small pieces of code can be harder to develop than a large amount of code. Also, the use of bad programming practices and the developer's programming style can also interfere in the count [16]. NOC is a similar metric that can be used to avoid these issues, since it counts the total number of characters in the code without consider new lines and blank line characters [5]. However, NOC depends on how verbose the developer is in defining identifiers, such as variable, functions and classes names.

Developers' productivity can also be analyzed based on the number of TOCs [14], which are divided between operators and operands. Operators include the keywords of a language, arithmetic (e.g., $+$, $-$, $*$, $/$, $\%$), relational (e.g., $<$, $<=$, $>$, $>=$, $==$, $!=$), logical (e.g., !, &&, ||) etc. The operands are all those that are not operators but constants, variables etc. [17]. TOC are measured from Eq. 1.

CCN uses Graph Theory to represent a program as a flow graph and measures its complexity from Eq. 2, where E is the number of edges, and N is the number of nodes. If there is a main program M calling a procedure A, there will be two flow graphs resulting in two connected components. In this case, the complexity will be equal to the sum of the individual complexities of M and A, or can be calculated from Eq. 3 [18], where P is the number of connected components.

TABLE I: Equations of coding metrics.

Metric	Equation	
TOC [14]	$TOC = Operators + Operands$	(1)
CCN [18]	$CCN = E - N + 2$	(2)
	$CCN = E - N + 2 \times P$	(3)
IFC [19]	$IFC = length \times (fan_{in} \times fan_{out})^2$	(4)
Halstead's measures [17]	$N = N_1 + N_2$	(5)
	$n = n_1 + n_2$	(6)
	$V = N \times log_2(n)$	(7)
	$D = (n_1/2) \times (N_2/n_2)$	(8)
	$E = V \times D$	(9)
	$T = E/S$	(10)
COCOMO II [20]	$S = B + 0.001 \times \sum_{j=1}^{5} F_j$	(11)
	$Effort = A \times KSLOC^S \times \prod_{i=1}^{n} M_i$	(12)
	$Time = C \times Effort^{D+0.2\times(S-B)}$	(13)
	$Team_{Size} = Effort/Time$	(14)

IFC (Information Flow Complexity) metric considers the connections among the procedures of a program to measure complexity, which are determined by the fan-in and fan-out. Fan-in is the number of local flows to the procedure plus the number of data structures from which the procedure retrieves information, and fan-out is the number of local flows of the procedure plus the number of data structures that the procedure updates [19]. IFC of a procedure is measured from Eq. 4, where the length is the LOC for the selected procedure [19].

Halstead's measures assess a program based on the number of tokens, which are measured according to: number of operators (n_1) and operands (n_2), and total occurrences of operators (N_1) and operands (N_2) [17]. From these values, it is possible to calculate the program length (Eq. 5), the vocabulary (Eq. 6), the volume in bits (Eq. 7), and the programming difficulty (Eq. 8). From volume and difficulty, the programming effort can be estimated using Eq. 9. Finally, it is possible measure the development time (in seconds) from Eq. 10, where S is the speed that the brain does elementary mental discrimination to encode a program, which is set equal to 18 [17].

COCOMO II is a model for estimating software development effort designed to add more variability and accuracy into the initial version (COCOMO 81) [20]. This model uses as input the thousand of source lines of code (KSLOC) and a set of cost drivers and scale factors for calibrate it according to the application, developers, environment, and other issues that affect the development cycle. COCOMO II has two types of cost drivers. The first one is post-architecture model, comprising: Required Software Reliability (RELY), Database Size (DATA), Documentation match to life-cycle Needs (DOCU), Product Complexity (CPLX), Required Reusability (RUSE), Execution Time Constraint (TIME), Main Storage Constraint (STOR), Platform Volatility (PVOL), Analyst Capability (ACAP), Applications Experience (APEX), Programmer Capability (PCAP), Platform Experience (PLEX), Language and Tool Experience (LTEX), Personnel Continuity (PCON), Use of Software Tools (TOOL), Multisite Development (SITE), and Required Schedule Development (SCED). The second one is early design model, comprising: Personnel Capacity (PERS), Product Reliability (RCPX), Platform Difficulty (PDIF), Personal Experience (PREX), and Facilities (FCIL). Each of them must receive a rating among: extra low, very low, low, nominal, high, very high, and extra high [20].

COCOMO II implements scale factors related to Precedentedness (PREC), Development Flexibility (FLEX), Architecture/Risk Resolution (RESL), Team Cohesion (TEAM), and Process Maturity (PMAT). Each of them also must receive a rating among very-low and extra-high. The scale factors of a project (F_j) are all summed and used to determine a scale exponent (S) from Eq. 11, where B is set equal to 0.91. After calculating S value, it is used as an exponent for calculating the effort (Person/Months) in Eq. 12, where A is set equal to

2.94, and M_i is the set of cost drivers. Moreover, COCOMO II allows us to calculate the ideal time to develop a project from the Eq. 13, where B, C, and D are set equal to 0.91, 3.67 and 0.28, respectively. Finally, the model also provides the size of development team from Eq. 14 [20].

III. Methodology

In this study, we evaluated LOC, NOC, TOC, CCN, IFC, Halstead and COCOMO II coding metrics applied to parallel programming. To do so, our experiments aim to verify the productivity of FastFlow [21], Pthreads [22], SPar [23], and TBB [24] parallel APIs for multi-core systems in the development of the following C++ stream processing applications[1]: Bzip2, Dedup, Denoiser, Person Recognition, and Lane Detection.

We used the following tools to measure the coding metrics: SLOCCount[2] to get LOC and COCOMO II, Notepad++[3] to get NOC, Lizard[4] to get CCN and IFC, and Commented Code Detector (CCD)[5] to get TOC and Halstead. Although CCD is compatible with C++, some limitations remain. It considers neither library objects as operators, e.g., the `cout` command, nor the keywords of the parallel APIs as operators, since it is not focused on evaluating parallel applications.

TABLE II: 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			

Table II 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 that will be processed. The parallel code use more than 95% of the available processors in order to achieve high performance. The development team has a few years of experience in developing these types of applications and has been able to use the parallel APIs effectively and efficiently. Also, no deadline was required for the applications developed.

To compare the estimated development time by Halstead and COCOMO II, we converted both values to days needed to develop the application. IFC and CCN were also measured as the sum of complexities from each function in the program.

[1]Codes in: https://github.com/GMAP/code-metrics-EuromicroSEAA-2021
[2]Available in: https://dwheeler.com/sloccount/.
[3]Available in https://notepad-plus-plus.org.
[4]Available in: http://www.lizard.ws/.
[5]Available in: https://github.com/dborowiec/commentedCodeDetector.

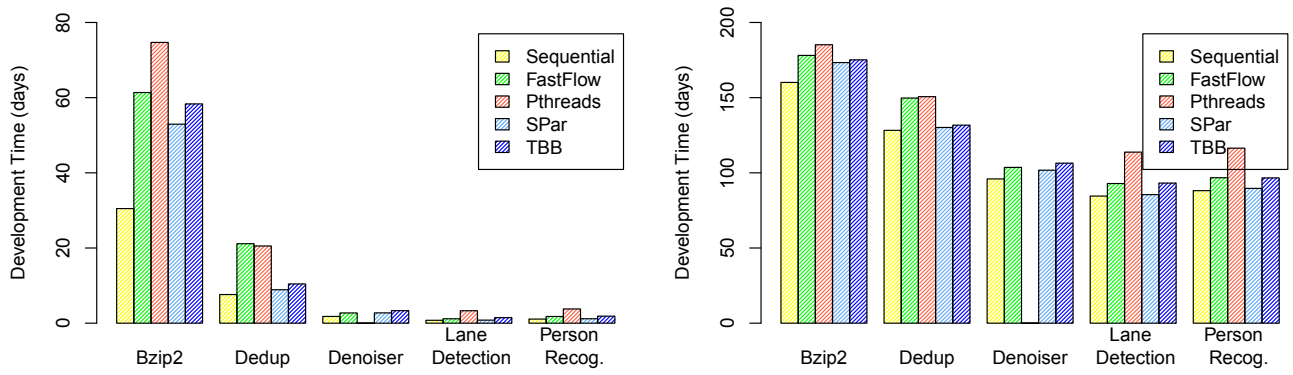
IV. Evaluation

Table III presents the results for each application implemented by the parallel APIs and the sequential version. Our results show that SPar presented the lowest number of LOC, NOC, and TOC compared to the other parallel APIs, considering all applications. This is because the SPar programming model requires only to insert annotations in the code without major changes. Compared to the sequential version, the LOC 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 LOC value increased by 35.34% and 28.40%, respectively. Similarly, there was a smaller difference between the NOC and TOC values when comparing the sequential version and the SPar version for Dedup, Lane Detection and Person Recognition. 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.

FastFlow and TBB showed similar results. The greatest difference was for Dedup, where the LOC, 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 parallel APIs have similar programming models. For both FastFlow and TBB, the application used similar data structures (`class`/`struct`) for each stage of the pipeline. In addition, communication among the stages

TABLE III: Results on code size and complexity.

Application	**APIs**	**LOC**	**NOC**	**TOC**	**CCN**	**IFC**
Bzip2	Sequential	1327	35087	9592	288	1005
	FastFlow	1991	54071	14184	431	5341
	Pthreads	2312	61223	16199	473	6289
	SPar	1796	49270	13148	392	5118
	TBB	1868	51136	13732	404	7234
Dedup	Sequential	569	18346	3902	111	2960
	FastFlow	1027	32688	6833	192	3726
	Pthreads	1052	35243	7322	196	1164
	SPar	602	20261	4313	119	3088
	TBB	629	21121	4356	116	24
Denoiser	Sequential	176	7778	1590	28	30
	FastFlow	243	9699	2006	35	643
	Pthreads	-	-	-	-	-
	SPar	226	9660	1942	34	792
	TBB	271	10718	2195	39	592
Lane Detection	Sequential	116	3344	979	13	0
	FastFlow	166	4384	1256	22	49
	Pthreads	360	9639	2187	41	888
	Spar	121	3496	1024	13	0
	TBB	168	4648	1331	22	49
Person Recognition	Sequential	136	5064	1172	18	0
	FastFlow	194	6586	1557	28	49
	Pthreads	393	11250	2476	46	888
	SPar	145	5687	1239	18	0
	TBB	193	6808	1560	25	49



(a) Halstead.

(b) COCOMO II.

Fig. 1: Results of development time estimated.

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 [21], [24], but it does not prove better productivity.

As expected, Pthreads presented the worst result among the parallel APIs, with the highest value of LOC, NOC, and TOC. In the worst case, it achieved LOC, NOC, 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.

Although program length is used as a predictor of maintainability and reability [5], 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 experience of developers. 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, LOC, NOC and TOC are only useful for measuring the manual exercised effort of developers if they are used together with other coding metrics.

Table III also presents the results of CCN and IFC. 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. Also, as with the length metrics, Pthreads shows the worst result for the CCN metric. A procedure with CNN value greater than 10 can be problematic [16]. The results in Table III show that 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 presents 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.

Fig. 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. 8. 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 parallel APIs such as operators. Thus, SPar and FastFlow presented the same number of operators ($n1 = 47$).

Fig. 1b presents the results of COCOMO II for each application implemented by the parallel APIs 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 LOC and SPar presents the smallest LOC 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 LOC value in comparison to the applications with other parallel APIs.

Although we converted the development time estimated by Halstead and COCOMO II to days, the metrics showed different results. For Lane Detection, it took approximately 85 days to develop the SPar application according to COCOMO II. On the other hand, according to Halstead, it took approximately 1 day ($\approx$ 99.52% lower). A complex application as this could not be developed from scratch in just 1 day, even if the developer had several years of experience in C++ and SPar. This occurs because the Halstead model does not consider essential aspects of software development, such as the developers' profiles. Therefore, COCOMO II seems to be a more complete model in comparison to Halstead. However, it does not consider factors that impact on the development of parallel applications, such as the developers' experience in parallel programming, the programming model used, the architecture etc. In addition, COCOMO II considers that the application will be implemented from scratch disregarding the insertion of parallel directives in the code.

V. Conclusions

LOC, NOC, and TOC metrics have shown to be insightful for comparing parallel applications with respect to code size. CCN proved to be more effective than IFC for measuring the complexity of a parallel program. However, CCN proved to be limited when evaluating SPar because it considers the annotations as another loop. Alternatively, a metric could be proposed based on the number of concurrent activities (CA) set up in the program and the number of interactions ($INTER$) among parallel activities deployed in the code. This complexity could be calculated with $CA \times INTER$.

COCOMO II proved to be the most suitable metric for evaluate the productivity of parallel APIs. However, only the CPLX cost driver assess the complex parallel computing operations. There are other factors that can impact the development of parallel applications, such as programming model, architecture, developer's background, and application domain. Performance of applications is another idea from [15] that can be included to these factors as well as the number of activities that will be executed simultaneously, or even the amount of data shared and accessed concomitantly among threads.

This study has some threats to validity. The learning effect can be a threat to internal validity, because no order for the use of the parallel APIs was specified. Another threat to internal validity is related to instrumentation, such as the use of CCD tool, which does not recognize parallel API keywords as operators. The study design can be a threat to construction validity, because FastFlow, SPar and TBB are pattern-based parallel APIs, unlike Pthreads. Finally, a threat to the validity of conclusion is the size of the applications evaluated.

A future investigation could be the extension of the COCOMO II to evaluate parallel programs by including new cost drivers concerning the factors that affect the development effort of such applications. Reusing the knowledge acquired from years now, the model could be fine-tuned.

Acknowledgment

This research is partially supported by Università di Pisa PRA_2018_66 DECLWARE, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPERGS 05/2019-PQG project ParaS (Nº 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPar4.0 (Nº 21/2551-0000725-7), Universal MCTIC/CNPq Nº 28/2018 project SParCloud (Nº 437693/2018-0), UNIRIO, and FAPERJ (Proc.211.583/2019).

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2016.

[2] T. Rauber and G. Rünger, *Parallel programming*. Springer, 2013.

[3] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: patterns for efficient computation*. Morgan Kaufmann, 2012.

[4] ISO 9241-11:2018, "Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts," Geneva, Switzerland, 2018. [Online]. Available: https://www.iso.org/standard/63500.html

[5] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the upc language," in *2004 IPDPS Conference*. Santa Fe, USA: IEEE, 2004.

[6] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel programmer productivity: A case study of novice parallel programmers," in *Proceedings of the 2005 ACM/IEEE Conf. on Supercomputing*. Seatle, USA: IEEE, 2005.

[7] V. Pankratius, A. Jannesari, and W. F. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *IEEE software*, vol. 26, no. 6, pp. 70–77, 2009.

[8] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the SIGSOFT/FSE'12*. New York, USA: ACM, 2012.

[9] S. Nanz, S. West, and K. S. Da Silveira, "Examining the expert gap in parallel programming," in *Euro-Par'13*. Aache, Germany: Springer, 2013, pp. 434–445.

[10] D. Griebler, D. Adornes, and L. G. Fernandes, "Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures," in *Proceedings of SEKE 2014*. Vancouver, Canada: KSIGS, Jul 2014, pp. 25–30.

[11] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *Proceedings of ICSE '15*, vol. 1. Florence, Italy: IEEE, 2015, pp. 778–788.

[12] J. Miller, S. Wienke, M. Schlottke-Lakemper, M. Meinke, and M. S. Müller, "Applicability of the software cost model cocomo ii to hpc projects," *IJCSE*, vol. 17, no. 3, pp. 283–296, 2018.

[13] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes, "High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2," *IJPP*, vol. 47, no. 1, pp. 253–271, 2018.

[14] J. Fernàndez-Fabeiro, A. Gonzalez-Escribano, and D. R. Llanos, "Simplifying the multi-gpu programming of a hyperspectral image registration algorithm," in *HPCS 2019*. IEEE, 2019, pp. 11–18.

[15] S. Wienke, J. Miller, M. Schulz, and M. S. Müller, "Development effort estimation in hpc," in *SC'16*. IEEE, 2016, pp. 107–118.

[16] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*, 3rd ed. CRC press, 2015.

[17] M. H. Halstead, *Elements of software science*. Elsevier, 1977.

[18] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[19] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE transactions on Software Engineering*, no. 5, pp. 510–518, 1981.

[20] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece, *Software cost estimation with COCOMO II*. Prentice Hall PTR, 2000.

[21] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: high-level and efficient streaming on multi-core," in *Programming multi-core and many-core computing systems*. Wiley, 2014.

[22] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, 1996.

[23] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. 27, no. 01, p. 1740005, 2017.

[24] M. Voss, R. Asenjo, and J. Reinders, *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019.