A Comprehensive Exploration of Languages for Parallel Computing

FEDERICO CICCOZZI, LORENZO ADDAZI, SARA ABBASPOUR ASADOLLAH, BJÖRN LISPER, ABU NASER MASUD, and SAAD MUBEEN, Mälardalen University

Software-intensive systems in most domains, from autonomous vehicles to health, are becoming predominantly parallel to efficiently manage large amount of data in short (even real-) time. There is an incredibly rich literature on languages for parallel computing, thus it is difficult for researchers and practitioners, even experienced in this very field, to get a grasp on them. With this work we provide a comprehensive, structured, and detailed snapshot of documented research on those languages to identify trends, technical characteristics, open challenges, and research directions. In this article, we report on planning, execution, and results of our systematic peer-reviewed as well as grey literature review, which aimed at providing such a snapshot by analysing 225 studies.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; **Concurrent programming languages;**

Additional Key Words and Phrases: Parallel computing, programming, modelling, languages, frameworks, systematic literature review

ACM Reference format:
Federico Ciccozzi, Lorenzo Addazi, Sara Abbaspour Asadollah, Björn Lisper, Abu Naser Masud, and Saad Mubeen. 2022. A Comprehensive Exploration of Languages for Parallel Computing. *ACM Comput. Surv.* 55, 2, Article 24 (January 2022), 39 pages.
https://doi.org/10.1145/3485008

1 INTRODUCTION

There is an ever increasing demand for computational power, which has enabled us to do things that were once considered science fiction: an example is self-driving cars, which require large amounts of computing to do the necessary real-time processing of streaming data from sensors, cameras and so on, and make decisions in real-time based on these data.

Software parallelism, where multiple computations are carried out at the same time, has become the most common way to provide higher levels of computational power leveraging massively

This work was supported by the Knowledge Foundation through the projects HERO (rn. 20180039), DPAC, SACSys, and the Swedish Foundation for Strategic Research through the Serendipity project.
Authors' address: F. Ciccozzi, L. Addazi, S. A. Asadollah, B. Lisper, A. N. Masud, and S. Mubeen, Mälardalen University, Box 883, Västerås 72123, Sweden; emails: {federico.ciccozzi, lorenzo.addazi, sara.abbaspour.asadollah, bjorn.lisper, abu.naser.masud, saad.mubeen}@mdh.se.

0360-0300/2022/01-ART24 $15.00
https://doi.org/10.1145/3485008

parallel hardware computational units. Parallel hardware was first introduced in supercomputers, e.g., in the ILLIAC-IV [22]. As of the time of writing (July 2020), the supercomputer with the highest peak performance is the Japanese Fugaku [233], with 7.3 million cores, giving it a maximal speed of 514 petaFLOPS. Besides increment of performance, parallelism can be used to reduce energy consumption too; this role is becoming more and more important, as shown by several secondary studies on energy awareness [221], how to best achieve energy efficiency [122], and which energy predictive models are available [181], in HPC.

We are facing a situation where software-intensive systems have become predominantly parallel. Parallelism appears in many forms, at multiple levels, and it is supported by a variety of hardware architectures. This raises the issue how to *program* these systems. Parallel programming was known to be hard already 30 years ago [126], and it has hardly become easier since then. The programming language is an important link in the software production chain. Many parallel programming languages have been invented over the years, and there is an incredibly rich literature. Due to the importance of the topic and its breadth, we believe that the time is ripe to create a structured and detailed snapshot of this research area, to identify trends, technical characteristics, and potentially open challenges.

In this article, we report on the planning, execution, and results of our systematic literature review, which aims at providing such a snapshot. From an initial set of 3,476 papers and 72 languages, we identified 225 *primary* studies, which we analysed in detail following a precise data extraction, analysis, and synthesis process. A summary of the resulting highlights of our study is as follows:

- Since 1988, effort in this research area has been steadily growing;
- Most languages are high-level and general-purpose;
- Most languages are defined in terms of extensions of existing languages; C/C++ and Java are the most popular programming languages and UML the most popular modelling language;
- Imperative and object-oriented programming are the most popular paradigms, in that order;
- Explicit parallelism is the most commonly supported;
- Task- and data-parallelism are the most common problem decomposition approaches, possibly supported by lower-level forms of parallelism, e.g., pipeline;
- The majority of languages are compiled and the target is usually a high-level language;
- CPUs (multi- and many-core) represent by large the most commonly targeted architectures; interestingly, over 30% of the languages are not tailored to any specific target architecture;
- Support for data-parallelism and multiple hardware platforms (also heterogeneously mixed) are the most longed for features for existing languages;
- Languages and related tooling are in most cases at a prototypical level.

The remainder of the article is organised as follows. We describe our research method in Section 2. The results of our study are unwound in Sections 3 and 4. In Section 5, we provide a further discussion of the results focusing on projecting open challenges into what we believe, supported by our results, to be the most urgent matters to solve in the area. An analysis of the potential threats to the validity of our study, and how we handled them, is reported in Section 6. Works related to our study are described in Section 7, and the article is concluded with Section 8.

2 RESEARCH METHOD

This study was designed and carried out by following guidelines for systematic literature reviews [133]. To carry out this study, we followed the process depicted in Figure 1, which can be divided into the three common phases of planning, conducting, and documenting.

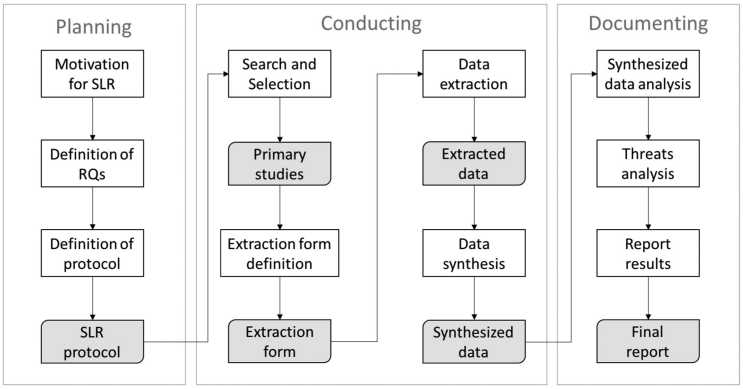


Fig. 1. Overview of the SLR process.

Planning. The objective of this phase was to

- establish the need for a review of languages for parallel computing.
- identify research goal and more importantly questions, and
- define the protocol to be followed by the research team for carrying out the work in a systematic and pinpoint manner.

The output of the planning phase is a detailed review protocol.

Conducting. The objective of this phase was to perform the SLR by carrying out all the steps defined in the review protocol, as follows:

- *Search and selection*: we performed an automatic search in the peer-reviewed literature on a set of four databases and a manual search in the so called *grey literature* (e.g., web-pages, forums) on the Google search engine. Then, identified candidate entries were filtered in order to obtain the final list of primary studies[1] to be considered in later activities of the review. After selection, we carried out an exhaustive backward and forward snowballing as integration to the search and selection process.
- *Data extraction form definition and classification framework*: we defined the set of parameters to compare and classify the primary studies based on the research questions. This was done systematically by applying a standard keywording process [186]. The set of parameters constitutes a classification framework, which was used for data extraction in our study and can be used for the classification of languages for parallel computing.
- *Data extraction*: we went into the details of each primary study, thus filling the corresponding data extraction form. Filled forms were collected and aggregated to be analyzed and synthesised.
- *Data analysis and synthesis*: we provided a comprehensive summary and analysis of the data extracted in the previous activity. The main goal of this activity was to elaborate on the extracted data in order to address each research question of the SLR. This activity involved

[1]For the sake of simplicity, in the remainder of this document, we will refer to included studies and languages from either source as *primary studies*, and will use different terms only when strictly needed to differentiate between papers and languages.

Table 1. Goal of This Study

Purpose	Identify, classify, and evaluate
Issue	the publication trends, technical characteristics, provided evidence, and limitations
Object	of existing languages for describing parallel software
Viewpoint	from researcher's and practitioner's points of view.

both quantitative and qualitative analysis of the extracted data, achieved via vertical and orthogonal analysis.

Documenting. In this phase, we thoroughly analysed and synthesised the extracted data as well as carried out a detailed analysis of possible threats to validity. Eventually, we wrote this article, which describes the performed study. We also provided a complete and public *replication package*[2] for independent replication and verification of our study. In the package, we provide the raw data of search and selection, the complete list of primary studies, as well as the raw data from data extraction.

2.1 Research Goal and Questions

When carrying out a systematic literature review, clearly defining the research goal, and questions are pivotal tasks [41]. The goal of this study was to

> *identify, classify, and evaluate trends, focus, and open challenges in existing research on (modelling and programming) languages for parallel computing in the scope of software engineering.*

The goal was defined by using the Goal-Question-Metric perspectives [24] (see Table 1). Since our aim is to classify languages for the direct use of software engineers, we discard intermediate and lower languages, which are usually employed for automatic transformations such as compilation and optimisation, or as pivot languages, as well as languages for analysis and V&V purposes only. Instead, we focus on high-level languages, that is to say third-generation programming languages and above, intended to be used by engineers to describe parallel software. We will call them **parallel languages** in the remainder of this article.

Our focus is on general-purpose and external domain-specific languages (defined as standalone languages extending or based on an existing language) [127]. These could be also be defined in terms of standalone libraries, which we consider in this study. On the other hand, we do not consider embedded/internal domain-specific languages since the focus would have become too broad and loose (embedded/internal domain-specific languages have several other interesting aspects in relation to their design/implementation and relation to the host language that would need attention).

Our goal can be refined into the following **research questions** (**RQ**), for each of which we also provide primary objective of investigation:

RQ1: *What are the **publication trends** of research on parallel languages?*
Objective: to classify primary studies in order to assess interests, relevant venues, and contribution types; depending on the number of primary studies, trends are assessed over time.

[2]The replication package is available at https://github.com/federicoCiccozzi/CSUR_parallel_languages_replicationPackage.

RQ2: *What are the **technical characteristics** of parallel languages?*
Objective: to identify and classify existing languages in terms of their technical characteristics.

- RQ2.1: *What are the **properties** of parallel languages?*
  Objective: to classify languages in terms of their core properties (e.g., programming paradigm).
- RQ2.2: *What are the characteristics concerning the **programming model** of parallel languages?*
  Objective: to classify languages according to the characteristics of the programming model that they implement (e.g., memory model).
- RQ2.3: *What are the characteristics concerning the **execution, run-time layer and implementation** of parallel languages?*
  Objective: to classify languages in terms of how they are executed (e.g., interpretation), their run-time characteristics (e.g., target architecture), and how they are implemented (e.g., standalone)

RQ3: *What are the **limitations** of parallel languages?*
Objective: to identify current gaps and limitations with respect to the state-of-the-art in languages for parallel computing.

Answering RQ1 gives us a detailed snapshot of publication trends, venues, and types. Based on the classification results obtained by answering research question RQ2, we provide a solid foundation for a thorough comparison of existing and future solutions for languages for parallel computing. Finally, answering RQ3 helps the community in understanding whether there is space for improvement in this research area.

This contribution is useful for both (i) researchers to further contribute to this research area by defining new approaches or refining existing ones, and (ii) practitioners to better understand existing methods and techniques and thereby to be able to adopt the one that better suites their business goals.

2.2 Search and Selection Strategy

In this phase, we gathered the set of research studies and languages that are relevant and representative for our purposes. Figure 2 shows our search and selection process.

Before performing the actual search and selection of relevant studies, we manually selected a set of seven pilot studies. They were selected based on the authors' knowledge of the targeted research domain (i.e., languages and frameworks for parallel programming) and on an informal preliminary screening that we performed on the available literature on the topic. Selected pilot studies fulfil our selection criteria (see below) and were selected to cover a fairly long time-span, they are presented in Table 2. Pilot studies were used to validate our search and selection strategy; more specifically, we used them to have quick feedback about the goodness of our search string to be used for the automatic search and for guiding the refinement of the selection criteria.

Two parallel activities were carried out: the *review of the peer-reviewed literature*, and the *review of the grey literature*. These activities yielded in a set of (i) papers describing languages and (ii) languages of interest.

We followed the same overall process when reviewing the peer-reviewed and grey literature. For the sake of simplicity, in the remainder of this document, we will refer to included studies and languages from either source as *primary studies*, and will use different terms only when strictly needed to differentiate between papers and languages.

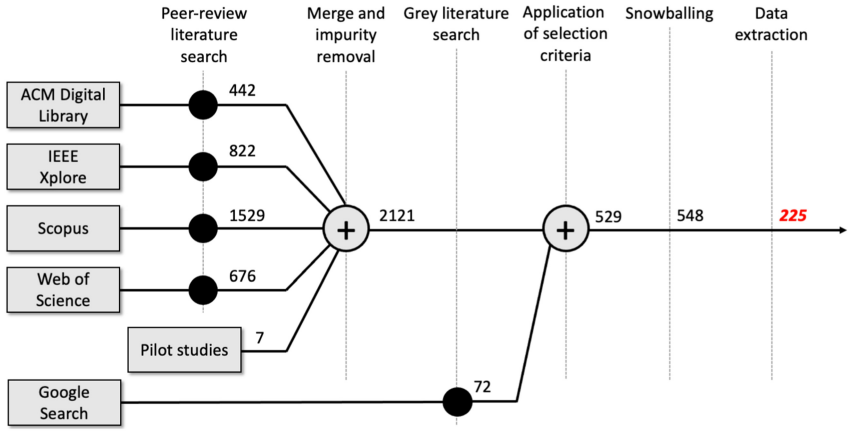


Fig. 2. Search and selection process.

Table 2. Pilot Research Studies

ID	Title	Year	References
P01	ParaSail: A Pointer-Free Pervasively-Parallel Language for Irregular Computations	2019	[223]
P02	Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates	2017	[116]
P03	Pencil: A platform-neutral compute intermediate language for accelerator programming	2015	[20]
P04	Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines	2013	[190]
P05	A Model-Driven Design Framework for Massively Parallel Embedded Systems	2011	[91]
P07	HPJava: a data parallel programming alternative	2003	[49]
P08	Parallel programming in Split-C	1993	[70]

Table 3. Electronic Databases and Indexing Systems Considered in This Research

Name	Type	URL
IEEE Xplore Digital Library	Electronic database	http://ieeexplore.ieee.org
ACM Digital Library	Electronic database	http://dl.acm.org
SCOPUS	Indexing system	http://www.scopus.com
Web of Science	Indexing system	http://webofknowledge.com

2.2.1 Automatic Search. In this stage, we performed automatic searches on the electronic databases and indexing systems [133] listed in Table 3. As suggested in [133], in order to cover as much relevant literature as possible, we chose four of the largest and most complete scientific databases and indexing systems in software engineering, that are: IEEE Xplore Digital Library, ACM Digital Library, SCOPUS, and Web of Science. The selection of these electronic databases and indexing systems was guided by (i) their high accessibility, (ii) their ability to export search results to well-defined, computation-amenable formats, and (iii) the fact that they have been recognised as being an effective means to conduct systematic literature reviews in software engineering [41].

To create the search string, we considered (i) the research questions and (ii) the set of pilot studies. Then we extracted a list of relevant concepts, their synonyms, abbreviations, and alternative spellings, and we combined them into the final search string by using logical ANDs and ORs. The search string, shown in Listing 2.2.1 was tested by executing pilot searches on the four sources, and checked against all the pilot studies, which had to be part of the obtained results. The actual search strings used for each database were obtained by syntactically adapting them to the characteristics of the specific database; the four specific strings can be found in the replication package. We sought the search string on title, abstract, and keywords of papers; the automatic searches gave us a total of **3,469** potential primary studies.

```
("data parallel" OR "task parallel" OR "process parallel" OR "control parallel" OR
    "collection oriented" OR "multi thread*" OR "parallel programming")
AND
("model* language" OR "model* framework" OR "programming language" OR "programming
    framework")
```

Listing 1. Search string used for automatic searches.

2.2.2 Impurity and Duplicates Removal. Due to the nature of the electronic databases and indexing systems, search results may include also elements that are clearly not research papers, such as conference and workshop proceedings, international standards, textbooks, book series, and so on, and duplicates. In this stage, we manually removed impurities and merged duplicates, reaching **2,121** potential primary studies.

2.2.3 Grey Literature Search. To harvest the grey literature, we targeted the *Google Search Engine*, in accordance to the guidelines for including grey literature in software engineering multivocal reviews [93]. We started with an automatic search using the same search string as for the peer-reviewed literature and completing by manual searches based on our knowledge in the field. The combination of automatic and manual searches helped us identifying relevant *lists* (e.g., Wikipedia's "List of concurrent and parallel programming languages" and *language-specific pages* (e.g., language-specific web-pages or blog posts about how specific languages are used). We selected the relevant languages using a mixture of expert knowledge, browsing the web-pages of well-known languages from the primary studies, and using lists such as Wikipedia's page on "Concurrent and parallel programming languages". The grey literature gave us 72 potential parallel languages to be included.

2.2.4 Application of Selection Criteria. Once removed impurities and duplicates, we applied our inclusion and exclusion criteria on all remaining studies to decide on their potential inclusion in the set of primary studies. Each study was analysed in two steps: first by considering its `title`, `keywords`, and `abstract`; second, if the analysis did not result in a clear decision, by `introduction`, and `conclusion` sections. The selection criteria were the following:

Inclusion Criteria for Peer-reviewed literature

(ICP1) Studies proposing a modelling/programming language/framework for parallel programming.
(ICP2) Studies proposing a formalisation and/or implementation of the proposed language.
(ICP3) Studies subject to peer review [254].
(ICP4) Studies written in English.
(ICP5) Studies available as full-text.

Inclusion Criteria for Grey literature

(ICG1) Web-pages reporting on a modelling/programming language/framework for parallel programming.
(ICG2) Web-pages reporting on a formalisation and/or implementation of the proposed language.
(ICG3) Web-pages in English.
(ICG4) Web-pages freely accessible.

Exclusion Criteria for Peer-reviewed literature

(ECP1) Secondary and tertiary studies (e.g., systematic literature reviews, surveys).
(ECP2) Studies in the form of tutorial papers, short papers (≤3 pages), poster papers, editorials, manuals, because they do not provide enough information.

Even if secondary/tertiary and other studies were excluded from the set of primary studies (see the ECP1/ECP2 exclusion criteria), we considered them as follows:

- for checking the completeness of our set of primary studies (i.e., if any relevant paper was missing from our study);
- for providing a summary of what is already known about languages for parallel programming;
- for identifying any important issues to be considered in our study;
- for defining what the contribution of our study to the literature could be;
- for identifying potential new ideas and preliminary results related to our topic.

Exclusion Criteria for Grey literature

(ECG1) Web-pages reporting exclusively on the basic principles of the language.
(ECG2) Web-pages reporting exclusively on the application of the language.
(ECG3) Peer-reviewed literature, since this type of studies is considered in a different search activity.
(ECG4) Videos, webinars, books, and so on. Since they are too time-consuming to be considered for this study.

To select studies objectively, four researchers—the selection team—actively participated in this phase. More specifically, by following the method proposed in [9], each potentially relevant study was classified by the four researchers as *relevant*, *uncertain*, or *irrelevant* according to the selection criteria above. Studies classified as *irrelevant* were immediately excluded, while those marked as *relevant* were preliminary included. For the *uncertain* cases, the selection team discussed with the mediation of a fifth researcher, the mediator. The selection team members classified a certain number of common studies (∼15%) in order to check agreement, achieving a Cohen's kappa [28] of **0.87**.[3] Out of the gathered 2,122 peer-reviewed studies and 72 languages, we eventually selected **529** potential primary studies.

2.2.5 Snowballing. To minimise potential bias with respect to construct validity of our study, we complemented the automatic search with a snowballing activity [103]. We performed a closed recursive backward and forward snowballing activity [253]. We found 19 relevant studies, which led to a total of **548** preliminary primary studies.

[3]Cohen suggested the Kappa result be interpreted as follows: values ≤ as indicating no agreement and 0.01–0.20 as none to slight, 0.21–0.40 as fair, 0.41–0.60 as moderate, 0.61–0.80 as substantial, and 0.81–1.00, which represents our case with 0.87, as almost perfect agreement.

2.3 Data Extraction

At the beginning of this phase, we created a *data extraction form*, or classification form, to be used to collect data extracted from each preliminary primary study. The data extraction form, provided in Tables 5 and 6, was composed of four facets, one for each research question. Specifically, for answering RQ1, we considered standard information such as title, authors, and publication details of each study. For RQ2, RQ3, and RQ4, we followed a systematic process based on *keywording* for: (i) defining the parameters of each facet of the data extraction form, and (ii) extracting data from the primary studies accordingly.

The goal of the keywording was to effectively develop an extraction form that could fit existing studies and took their characteristics into account [186]. More precisely, we collected keywords and concepts by reading the full text of the pilot studies. Then, we performed a clustering operation on collected keywords and concepts in order to organise them according to the identified categories. The clustering operation is very similar to the sorting phase of the grounded theory methodology [57]. During the actual extraction phase, we collected any kind of additional information that was deemed relevant but that did not fit within the data extraction form. We reviewed the collected additional information and, when needed, we refined the data extraction form to better fit the collected information; previously analysed primary studies were re-analysed according to the refined data extraction form. This process was complete only when all primary studies were analysed. The final total number of included and analysed primary studies was **225** (see Table 4).

2.4 Data Analysis and Synthesis

In this phase we gathered, analysed and synthesised the extracted data to understand and classify the current state of the art in the area of languages for parallel programming [134, § 6.5]. We designed and carried our data analysis and synthesis by following the guidelines presented by Cruzes et al. in [69]. More specifically, we focused on vertical and orthogonal (or horizontal) analysis.

Vertical analysis aims at finding trends and collect information about each category of the data extraction form. We applied the line of argument synthesis [254], meaning that we first analysed each primary study individually to classify its main features according to each specific parameter of the data extraction form. Then, we analysed the set of studies as a whole, in order to reason about potential patterns, trends and potential gaps.

Orthogonal analysis aims at identifying possible relations across different categories of the data extraction form. We cross-tabulated and grouped extracted data as well as we made comparisons between pairs of categories of the data extraction form. Through contingency tables, we extracted and evaluated relevant pair-wise relations in terms of patterns, trends, and potential gaps.

In both cases, we performed a combination of content analysis [89] (for categorising and coding approaches under broad thematic categories) and narrative synthesis [200] (for detailed explanation and interpretation of the findings coming from the content analysis).

3 RESULTS: VERTICAL ANALYSIS

In this section, we report on the results of the vertical analysis. More specifically, we analysed each primary study individually to classify its main features according to each specific parameter of the data extraction form. Then, we analysed the set of studies as a whole and here we reason and report on potential patterns, trends, and potential gaps. Note that in some cases no value or

Table 4. List of Primary Studies, with Language Name (if Avail.), Reference, and Activity Status of the Language

ID	Name ((✓) = active)	References	ID	Name ((✓) = active)	References	ID	Name ((✓) = active)	References
P01	ParaSail (✓)	[223]	P02	Futhark (✓)	[116]	P03	Pencil	[20]
P04	Halide (✓)	[190]	P05	Gaspard (✓)	[91]	–	–	–
P07	HPJava	[49]	P08	Split-C (✓)	[70]	P993	–	[184]
P1002	APARAPI-FPGA	[206]	P1011	Logtalk (✓)	[171]	P1012	PACXX	[111]
P1034	XPFortran-OpenMP	[261]	P1039	HyperPascal (✓)	[31]	P1041	IAL	[66]
P105	CARPET	[211]	P1097	AJWS (✓)	[139]	P1101	ClassiC	[175]
P1103	X-KLAIM (✓)	[30]	P1105	SuperPascal	[113]	P1113	Tetra (✓)	[84]
P1116	Seymour	[168]	P1118	IPC++	[216]	P1131	JavaNOW	[229]
P1136	Java4P	[178]	P114	JUMP (✓)	[115]	P116	ECP-C	[23]
P1172	Lemonade (✓)	[76]	P1175	Linda (✓)	[205]	P1200	NestStep	[130]
P1233	–	[189]	P1254	Modula-P (✓)	[244]	P1260	Molecule (✓)	[258]
P1285	MULTILISP	[112]	P1288	XcalableMP (✓)	[236]	P1291	Musket (✓)	[197]
P1293	NANO-2	[16]	P1304	Node.Scala	[37]	P1312	–	[208]
P1313	OP++	[220]	P1317	C++-with-Ease	[158]	P132	AMPLE (✓)	[137]
P134	LSP	[213]	P1351	–	[260]	P1371	OpenRCL	[149]
P138	P^3L	[71]	P1388	Orgel	[180]	P1389	Vector Pascal (✓)	[63]
P1395	Force	[5]	P140	–	[21]	P1412	Parallel C	[109]
P1425	–	[177]	P143	ASTOR (✓)	[238]	P1433	–	[59]
P1437	Encore (✓)	[40]	P1442	PPM	[43]	P1452	Chapel (✓)	[52]
P1464	Ensemble (✓)	[114]	P1467	Morpho (✓)	[4]	P147	–	[73]
P998	OIL (✓)	[97]	P1480	Delirium	[157]	P1495	–	[219]
P1511	ParCel-1	[242]	P1514	ParoC++	[176]	P155	Fortran M	[54]
P1550	PGASUS (✓)	[110]	P1565	Pooma	[179]	P1572	Prelude (✓)	[64]
P1586	Qlisp	[100]	P1620	Modula-2*	[187]	P1632	Arvo (✓)	[247]
P1636	PyCOMPSs (✓)	[226]	P1641	Quasar (✓)	[101]	P1649	Real-time Mentat	[106]
P1669	RELACS (✓)	[191]	P168	–	[72]	P1681	NCX	[10]
P1691	C^n	[154]	P1695	River Trail (✓)	[117]	P1700	–	[90]
P1705	SAC (✓)	[104]	P171	–	[78]	P1710	–	[36]
P1711	– (✓)	[224]	P1717	OmpSs-OpenCL	[77]	P1736	Scootr (✓)	[140]
P1738	Scout (✓)	[165]	P1752	Glasgow paral. Haskell	[153]	P1764	SHMEM+ (✓)	[3]
P1771	Sigma C (✓)	[102]	P1804	Spar (✓)	[241]	P1844	Swift/T	[255]
P1849	SyncCharts	[245]	P1867	FastFlow (✓)	[7]	P1894	Atomos (✓)	[48]
P1906	ZPL (✓)	[209]	P1910	Jade (✓)	[198]	P1912	CGIS	[156]
P1916	DSPL	[169]	P1923	ForeC (✓)	[259]	P1924	Fork95	[131]
P1928	II	[74]	P1930	JavaSymphony	[8]	P1931	JStar (✓)	[240]
P1940	OpenTM	[19]	P1941	parallel C (✓)	[47]	P1944	ParCel-2 (✓)	[46]
P1970	LM (✓)	[67]	P1976	–	[192]	P198	Polymorphic Parallel C	[162]
P1988	GridNestStep	[164]	P1991	TCF++ (✓)	[159]	P2023	TPascal	[45]
P2036	Trellis	[222]	P2055	UTC (✓)	[150]	P2072	iOberon	[135]
P208	SVM-Fortran	[96]	P2094	VersaPipe (✓)	[263]	P2113	Wysteria (✓)	[194]
P2120	Yada	[95]	P215	–	[203]	P232	Orca	[26]
P240	MANIFOLD	[17]	P252	A-NETL	[18]	P268	Accelerator	[225]
P281	mpC	[145]	P284	VPR	[65]	P292	Qu-Prolog (✓)	[60]
P296	AL1	[161]	P297	ALBA	[118]	P334	EL*	[193]
P344		[138]	P347	XMP (✓)	[147]	P36	–	[34]
P360	Gaspard2 (✓)	[201]	P363	DAPL	[250]	P364	Promoter	[29]
P367	PObC++ (✓)	[188]	P380	ALWAN	[88]	P392	Agora (✓)	[33]
P408	Augur (✓)	[235]	P415	CCMs	[107]	P443	Balinda C++	[249]
P444	Bamboo (✓)	[264]	P446	Beehive (✓)	[234]	P457	Booster	[182]
P460	Braid (✓)	[252]	P465	BSGP (✓)	[121]	P472	C**	[144]
P476	CaKernel	[35]	P479	CAOPLE (✓)	[257]	P48	DPML	[86]
P480	CAPP (✓)	[62]	P498	Chestnut	[215]	P51	–	[108]
P52	V	[141]	P545	SEPCom (✓)	[251]	P581	Delta Prolog	[185]
P586	Rolez (✓)	[80]	P605	COOL	[53]	P608	Copperhead (✓)	[50]
P619	CuPit-2	[120]	P62	Visper	[212]	P633	Firebird (✓)	[232]
P636	Dataparallel C	[173]	P650	Declarative Ada	[231]	P651	CLM (✓)	[68]
P656	Aida	[155]	P658	CL/1	[42]	P665	812	[167]
P669	OpenMPD	[146]	P692	TACT	[214]	P72	STING (✓)	[124]
P726	DPX10 (✓)	[248]	P728	DryadLINQ (✓)	[83]	P734	ICE (✓)	[98]
P735	Mentat	[105]	P736	EastFJP (✓)	[163]	P748	Skil	[39]
P764	Flat X10 (✓)	[32]	P768	ELMO (✓)	[196]	P786	MFL (✓) (✓)	[210]
P787	eskimo	[6]	P800	Eve	[85]	P805	KL	[136]
P824	StreamMDE (✓)	[217]	P825	HPF	[27]	P840	Aspen	[239]
P846	–	[148]	P852	–	[12]	P866	FastPara	[160]
P87	– (✓)	[92]	P886	FSPPL (✓)	[142]	P891	ForkLight	[132]
P90	Ellie (✓)	[13]	P907	FPMR (✓)	[207]	P910	HOE^2 (✓)	[152]
P930	Gemma in April	[256]	P931	GEMS (✓)	[38]	P949	Go!	[61]
P954	Gossamer	[199]	P964	GraphGrind (✓)	[218]	P968	GraphIt (✓)	[262]
P972	Habanero Java (✓)	[51]	P980	Ly (✓)	[237]	G1	Open MPI (✓)	[228]
G2	Joule	[128]	G3	LabVIEW (✓)	[172]	G5	Sisal	[94]
G6	Bloom (✓)	[195]	G7	Julia (✓)	[227]	G8	Limbo	[243]
G9	Sequoia++	[25]	G10	Clojure (✓)	[119]	G11	Erlang (✓)	[79]
G12	Rust (✓)	[202]	G13	Charm++ (✓)	[56]	G14	Join Java(✓)	[246]
G15	Chapel (✓)	[55]	G16	Coarray Fortran (✓)	[123]	G17	Ateji PX (✓)	[183]
G18	Scala(✓)	[204]	G19	Go (✓)	[99]	G20	JoCaml	[125]

Table 5. Data Extraction form Facets, Clusters and Categories

Facet	Cluster	Category	Description
RQ1	Publication details	Authors	identifies the list of authors
		Venue name	identifies the title of the study
		Venue type	identifies the type of venue
		Year	identifies the year of publication
RQ2	Language properties	Language name	reports the name of the language that is defined in the study
		Language abstraction	identifies the level of abstraction at which the language is defined
		Purpose	identifies whether the language is generic or domain-specific
		Parallel primitives	identifies whether the language provides implicit or explicit parallelism
		Abstract syntax	identifies how the abstract syntax is defined
		Concrete syntax	identifies how the concrete syntax is defined
		Program. paradigm	identifies the specific programming paradigm
		Communication type	identifies whether communication is synchronous or not
		Synchronisation type	if synchronous, identifies the sync. type
	Programming model	Parallel architecture	identifies the class of parallel architecture targeted in the study
		Problem decomposition inspired by Thoman et al.'s taxonomy [230]	identifies the type of parallelism. Task parallelism: a form of parallelism where the work at hand is broken down into smaller work units, called tasks, which are scheduled to run in parallel. Pipeline parallelism: form of parallelism where the computation of each task is divided into a fixed sequence of stages, the same for all tasks, and the tasks are computed in a partially parallel fashion where a task can start executing initial stages before the preceding task has finished executing its final stage. Data parallelism: a form of parallelism where operations over whole data structures are performed in parallel over the individual elements of the structures. Nested parallelism: a form of parallelism where the parallelism is exploited at several levels. An example is a cluster of powerful nodes where some parallelism appears between the nodes and some internally in the nodes, e.g., in a GPU internal to the node.
		Communication model inspired by Thoman et al.'s taxonomy [230]	identifies the the communication model. Shared memory: a form of communication where processes, or threads, communicate through writing and reading a shared memory area. Message passing: a form of communication where processes communicate, and often synchronise, by sending and receiving messages. Data flow: a computing paradigm where the nodes in a so-called Data Flow Graph communicate by passing data tokens over the edges in the graph. Shared events: a form of communication where processes/threads communicate via shared events. FIFO buffer: a form of communication via data buffering in FIFO manner.
		Memory model inspired by Thoman et al.'s taxonomy [230]	identifies the memory model. Distributed: a memory model where each processor has its own private memory, and has exclusive access to this memory. Shared: memory model where several processors share the same address space, and can read and write to the same memory area. Stack-based: way to handle memory for function calls in a structured fashion by allocating and deallocating local data and actual function arguments. Region-based: an alternative to stack-based memory handling where memory is allocated and deallocated in larger memory areas, so-called regions.
	Execution, run-time and implementa-tion	Execution mode	identifies how programs conforming to the language can be executed
		Target language (if compiled)	if compiled, identifies the compilation target language(s)
		Target architecture	identifies the hardware processor(s) targeted for execution
		Implementation type	identifies how the language is implemented
		Extended language	identifies the name of base language extended (or tailored) to define the language presented in the study
RQ3		Limitations	identifies gaps and open challenges as reported in the studies

Table 6. Category Values and Multiplicity

Category	Values	Mult.
Authors	string	1
Venue name	string	1
Venue type	Workshop, Conference, Journal, Book chapter	1
Year	numeric value (e.g., 2010)	1
Language name	string	0..1
Language abstraction	Modelling language, High-level programming language	1
Purpose	General-purpose, Domain-Specific	1
Parallel primitives	Explicit, Implicit	0..*
Abstract syntax	Metamodel, Formal specification, Context-free grammar, Informal specification	1
Concrete syntax	Textual, Diagrammatic, Graph-based, Tree-based	0..*
Programming paradigm	Object-oriented, Imperative, Declarative, Event-driven, Multi-paradigm (representing a language featuring several paradigms)	0..*
Communication type	Synchronous, Asynchronous	0..*
Synchronisation type	Rendezvous, Monitor, Lock-free data structure, Lock, Dynamic interfaces, Dependency graph, Channel, Barrier	0..*
Parallel architecture	Single instruction single data (SISD), Single instruction multiple data (SIMD), Multiple instruction single data (MISD), Multiple instruction multiple data (MIMD), Single program multiple data (SPMD), Architecture independent	0..*
Problem decomposition	Task parallelism, Data parallelism, Nested parallelism, Pipeline parallelism	0..*
Communication model	Shared memory, Message passing, Data-flow, User-defined, Shared events, FIFO buffer	0..*
Memory model	Stack-based, Distributed, Shared, Region-based	0..*
Execution mode	Compiled, Interpreted, Hybrid	0..*
Target language type (if compiled)	High-level, low-level	0..1
Target architecture	Multi-core CPU, Many-core CPU, GPU, GPGPU, FPGA, DSA, Target-independent	0..*
Implementation type	Standalone, Extension	0..1
Extended language (if extension)	string	0..1

multiple values could be extracted from the studies for specific categories. This means that the total number of occurrences in the related plots may not sum up to or be greater than the total number of primary studies (225). Note that we also provide summary tables (Tables 9–24) in supplementary material where each analysed article is connected to each specific category and its characterising value(s).

3.1 Publication Trends (RQ1)

To answer RQ1, we analysed the extracted data to identify publication trends in terms of (i) trend over time and (ii) venue types. The first, shown in terms of the number of primary studies published over the years, is depicted in Figure 3. It is interesting to note that the first study included in our investigation was published back in 1981. After that, the research area did not receive much

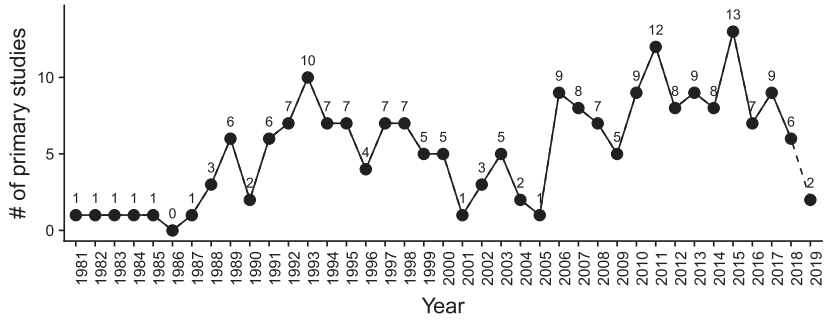


Fig. 3. Trend of publications over time.

attention for six years, which is indicated by a maximum of one primary study per year until 1987, as shown in Figure 3. Since 1988, effort in this research area started to steadily grow. This is evident from the 10-, 12- and 13-fold increase in the number of primary studies in 1993, 2011, and 2015 with respect to 1981. The growing interest of the community seems to have peaked in 2011 with 13 primary studies. Furthermore, based on the extracted data, we observe that the average number of primary studies is approximately six per year from 1981 to 2019, and approximately nine per year in the last decade (2009–2019). Given the steady increment in the last decade, we expect this trend to continue in the future. We also noted drops in the number of primary studies in 2001 and 2005; however, we could not find any clear reason for that.

We also classified the number of primary studies with respect to the type of publication venue, namely, Journal, Conference, Workshop, and Book chapter. We notice that Conference is the most common venue type, with 54% of primary studies belonging to it. The second most common venue type is Journal, which holds almost 36% of the primary studies. Workshop and Book chapter stand for 9.5% and 0.5% of the primary studies, respectively.

The top 10 venues in terms of number ($\geq$4) of primary studies are listed in Table 7. It is interesting to note that these venues only stand for 22.8% of the 225 primary studies, whereas the remaining ones are published in 134 other venues. The **International Conference on Parallel Processing** (**ICPP**) has published the highest portion (3.65%) of the primary studies. The second ranked venue, with 3.2% is the International European Conference on Parallel and Distributed Computing (Euro-Par). The ACM SIGPLAN Symposium on **Principles and Practice of Parallel Programming** (**PPOPP**) and the **International Conference on Functional Programming** (**ICFP**) published 2.74% of primary studies each. The **International Parallel and Distributed Processing Symposium** (**IPDPS**), the Conference on **Object-Oriented Programming Systems, Languages, and Applications** (**OOPSLA**), and ACM **Transactions on Programming Languages and Syste**ms (**TOPLAS**) published 2.28% of studies each. Finally, the ACM SIGPLAN Conference on **Programming Language Design and Implementation** (**PLDI**), the **Hawaii International Conference on System Sciences** (**HICSS**) and the International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) published 1.82% of studies each.

These results could be helpful in guiding new researchers in this area to identify the most relevant publication venues for searching the related work as well as publishing their research results. However, it should be noted that apart from the top nine venues (which also display a rather low concentration of published primary studies), the widely dispersed distribution of relevant publications over other venues (134) shows that the research community does not strongly favour any specific venue.

Table 7. Venues Featuring ≥4 Primary Studies

Venue name	Acronym	Type	#Papers
International Conference on Parallel Processing	ICPP	C	8
International European Conference on Parallel and Distributed Computing	Euro-Par	C	7
ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming	PPOPP	C	6
International Conference on Functional Programming	ICFP	C	6
International Parallel and Distributed Processing Symposium	IPDPS	C	5
Conference on Object-Oriented Programming Systems, Languages, and Applications	OOPSLA	C	5
ACM Transactions on Programming Languages and Systems	TOPLAS	J	5
ACM SIGPLAN Conference on Programming Language Design and Implementation	PLDI	C	4
Hawaii International Conference on System Sciences	HICSS	C	4
Workshop on High-Level Parallel Programming Models and Supportive Environment	HIPS	W	4

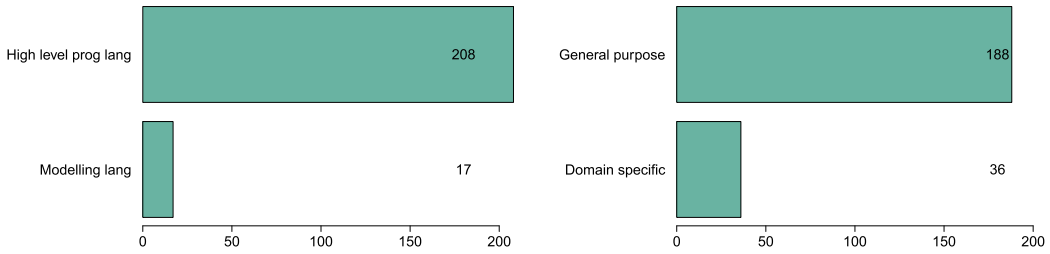
Highlights – RQ1 Publication trends

- Since 1988, effort in this research area has been steadily growing; a steady increment in the last decade suggests a continuation of this trend in the near future.
- The widely dispersed distribution of relevant publications across venues suggests that the research community does not strongly favour any specific one.
- The HIPS workshop has been continuously running for 25 years, showing an interesting continuity in the community's effort towards high-level parallel programming and related tools.

3.2 Technical Characteristics (RQ2)

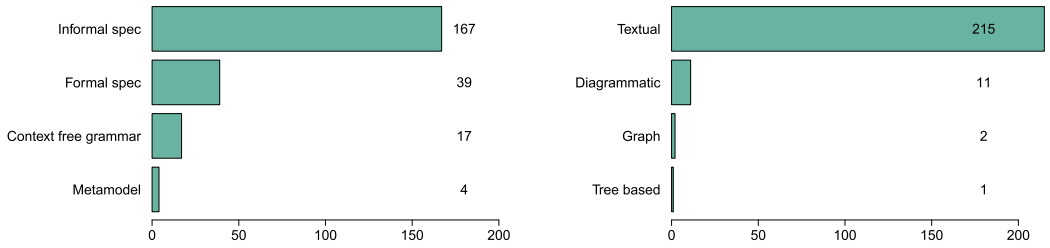
This research question focuses on a set of technical aspects of the languages proposed in the analysed primary studies. The technical aspects were either mentioned explicitly in the papers or could be easily inferred from the specification of the languages discussed in the papers.

3.2.1 Language Properties (RQ2.1). From the primary studies, we extracted and analysed some high-level features such as level of language abstraction, its purpose, and the kinds of abstract and concrete syntax of the languages. As it can be seen in Figure 4(a), most languages (208/225) are defined as *high-level programming languages*. This has to do with the syntactic abstractions provided by the language, the different kinds of data and control flow abstraction and the level of dependency on the underlying platform, such as hardware and operating systems. These languages range from *third generation* to *fifth generation* programming languages. A small number of languages (17/225) are defined as canonical *modelling languages*, in order to provide an even more abstract, and often diagrammatic, representation (i.e., model) of a program. The purpose of these modelling languages include providing efficient platform for data analytics, supporting multilevel modelling on various parallel platforms and accelerators. Interestingly, we found one concurrent constraint programming language (P633) targeting massively parallel SIMD computers. As shown in Figure 4(b), except for a few domain-specific or special-purpose languages (36/225), the majority of the studies (188/225) proposed general-purpose languages.



(a) Language abstraction (b) Purpose

Fig. 4. Language properties: language abstraction and purpose.



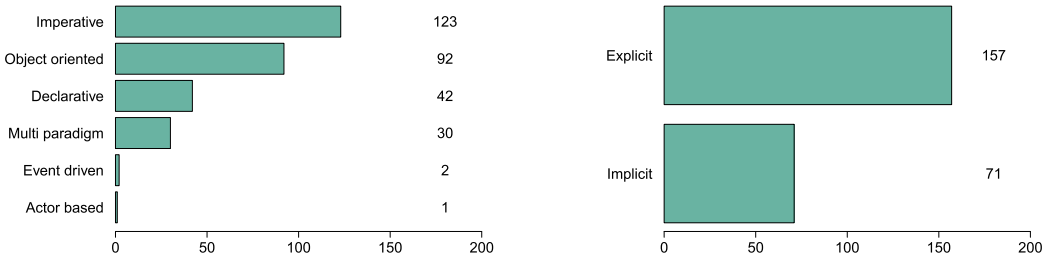
(a) Abstract syntax (b) Concrete syntax

Fig. 5. Language properties: abstract and concrete syntax.

As expected, and shown in Figure 5(b), the concrete syntax of most languages (215/225) is specified in terms of textual notations. For modelling languages in particular, diagrammatic, tree-based, or graph-based concrete syntax is provided in most cases too (14/17). The abstract syntax of most languages (approximately 73%) is provided only informally, as shown in Figure 5(a). In several studies, proposed languages are built on top of existing languages without providing the complete abstract syntax of the extended languages. The abstract syntax of modelling languages is described by means of canonical metamodel specifications only in four cases out of 17.

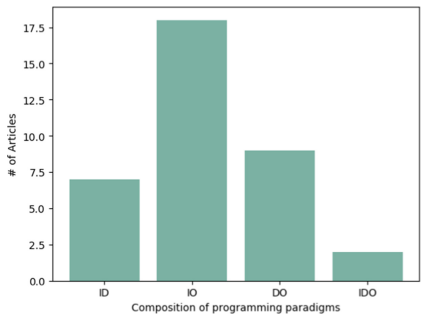
With regards to languages' programming paradigm, as it can be seen in Figure 6(a), it is evident that the procedural imperative paradigm is the most studied and used 123/225, whereas object-oriented and declarative paradigms rank second and third with 92 and 42 occurrences, respectively. Among those, a fair number of languages (30/225) support multiple programming paradigms; out of them, eighteen support imperative and object-oriented, seven support declarative and imperative, nine support declarative and object-oriented, two support combined imperative, declarative, and object-oriented, and one supports event-driven and object-oriented. The figure and table[4] in Figure 7 depicts the combination of the major programming paradigms and the associated programming languages. One event-driven programming language, called *Eve* (P800), is proposed to support the event-loop and task-based parallelism in achieving high concurrency. An interesting case is represented by *Node.Scala*, which is a dynamic multi-paradigm language and framework for enhancing the Java Virtual Machine by introducing safe stateful event-driven programming (P1304). Quite surprisingly, given the suitability of the paradigm for

[4]The table exclude languages that was not given a name in the study.



(a) Programming paradigm (b) Parallel primitives

Fig. 6. Language properties: programming paradigm and parallel primitives.



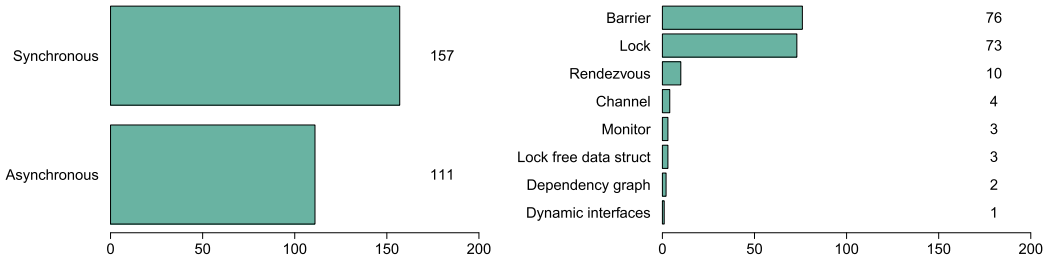
Paradigms	Languages
ID	PyCOMPSs, Scootr, DryadLINQ, Skil, FastPara, OIL, Julia
IO	PACXX, AJWS, C++-with-Ease, OpenRCL, Morpho, Pooma, Bamboo, CAPP, DryadLINQ, Mentat, GraphGrind, HPJava, Open MPI, Julia, Join Java, Chapel, Ateji PX
DO	Logtalk, Qu-Prolog, DryadLINQ, Go!, Habanero Java, Julia, Scala, JoCaml
IDO	DryadLINQ, Julia

Fig. 7. Multi-paradigm parallel languages. ID—Imperative and Declarative, IO—Imperative and Object-oriented, DO—Declarative and Object-oriented, IDO—Imperative, Declarative, and Object-oriented.

describing parallelism, only three object-oriented languages are *actor-based* (P1991, P296, and P1916).

Regarding the supported type of parallel primitives (Figure 6(b)), most languages (157/225) support *explicit* primitives for describing parallelism. Nevertheless, a noteworthy number of languages (71/225) focus on *implicit* primitives. Not so surprisingly, support for explicit or implicit primitives is in most cases mutually exclusive; only three languages (P01, P1970, and P1550) provide support for both. An interesting case is represented by Glasgow parallel Haskell (P1752), which exploits so-called *semi-explicit* parallelism, meaning that potential parallelism is explicitly annotated in the program, while all aspects of coordination are delegated to the runtime environment.

Regarding the types of communication among parallel activities offered by the languages (Figure 8(a)), we found that 42/225 studies offer both synchronous and asynchronous communication; the rest support either synchronous (157/225) or asynchronous communication (111/225). Among those supporting synchronous communication (Figure 8(b)), we found that lock- and barrier-based synchronizations are the most popular choices to synchronise parallel activities, with 76 and 73 occurrences each. A few languages (10/157) use rendezvous-based synchronisation instead. However, there exist some languages that use specialised synchronisation mechanisms, such as: channel-based (4/157), monitor-based (3/157), lock-free data structures (3/157), dependency graphs (2/157), and dynamic interfaces (1/157). Note that some languages exploit multiple types of synchronisation.



(a) Communication type (b) Synchronisation type

Fig. 8. Language properties: communication and synchronisation types.

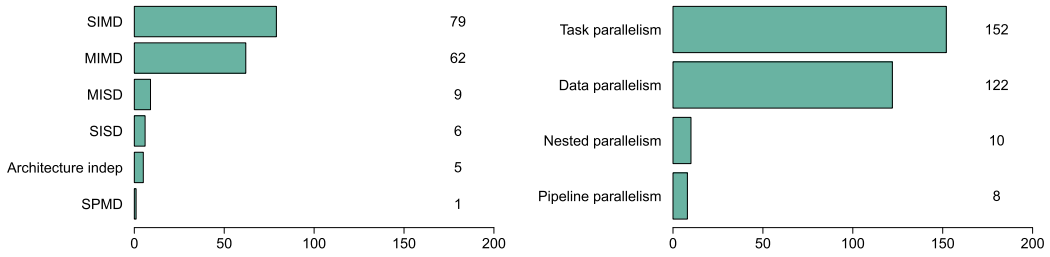
Highlights – RQ2.1 Language properties

- Most languages are high-level and general-purpose, their formal abstract syntax is often not explicitly given, and their concrete syntax is in most cases textual.
- Imperative and object-oriented programming are the most popular paradigms, in that order;
- Explicit parallelism is the most commonly supported.
- Distribution of synchronous and asynchronous communication is rather even, with lock- and barrier-based are the most common synchronisation means.

3.2.2 Programming Model (RQ2.2). The analysis of RQ2.2 covers four aspects of parallel programming models: target hardware architecture, problem decomposition, communication model, and memory model. As expected, most studies target **single instruction multiple data** (**SIMD**) or **multiple instruction multiple data** (**MIMD**) architectures, with 79/225 and 62/225 occurrences, respectively. Very few studies target **single instruction single data** (**SISD**) or **multiple instruction single data** (**MISD**) architectures, with 6/225 and 9/225 occurrences, respectively. Furthermore, five languages (P02, P03, P05, P1906, P1118, and P1136) provide *architecture-independent* solutions, meaning that their programming model is not tailored nor dependent on any specific target machine. One language (P1988) supports **single program multiple data** (**SPMD**) architecture.

The results concerning the supported problem decomposition types confirm the historical predominance of task- and data-parallelism (Figure 9(b)), with 152/225 and 122/225 entries, respectively. Task-parallelism is compatible with applications for various domains, e.g., from web applications to system programming, and often used to implement implicit (data) parallelism. In contrast, data-parallelism is generally preferred for high-performance computing applications. Lower-level forms of parallelism, nested (10/225), and pipeline parallelism (8/225) appear too, often as secondary form of parallelism. Figure 10 depicts the number of studies proposing languages supporting various composition of problem decomposition categories. In 64/225 primary studies, languages provide support for both task- and data-parallelism. Other problem decomposition combinations are proposed in relatively small number of studies: (7/225) for task and nested parallelism, (4/225) for pipeline and data parallelism, (6/225) for nested and data parallelism, (3/225) for task, nested, and data parallelism, and (1/225) for each of the combination of task and pipeline parallelism, and task, pipeline, and data parallelism.

Regarding the communication model (Figure 11(a)), message-passing is the most popular choice of communication (111/225). The use of shared memory, i.e., more or less implicitly regulated accesses to common regions of memory, is the second most popular medium of communication (107/225). Another, less common, communication means is data-flow-based (23/225), employed



(a) Parallel architecture (b) Problem decomposition

Fig. 9. Programming model: parallel architecture and problem decomposition.

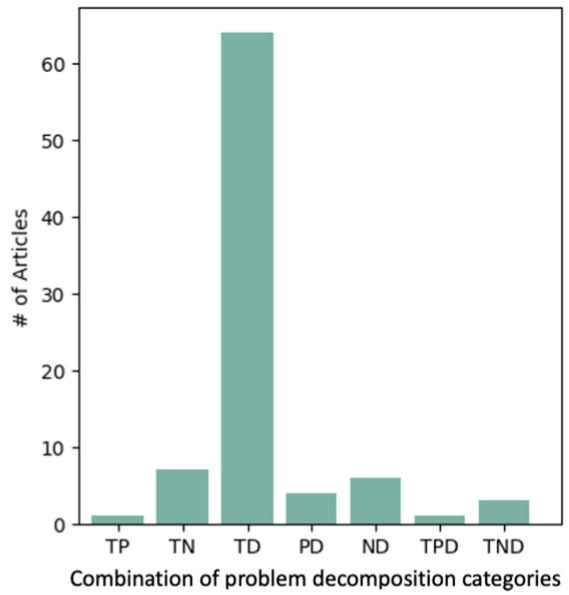
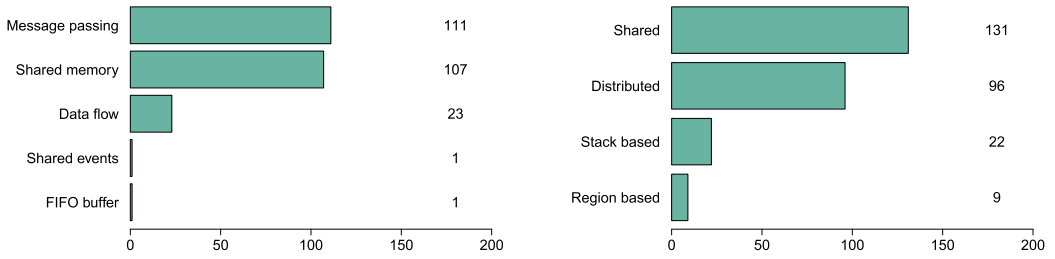


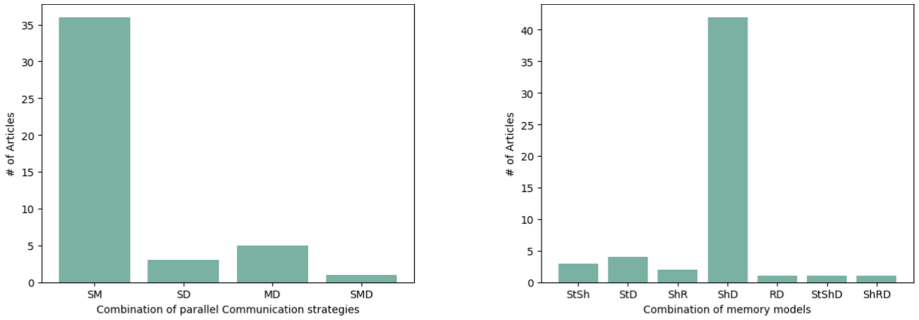
Fig. 10. Number of Languages supporting multiple problem decomposition strategies. T—Task parallelism, P—Pipeline parallelism, N—Nested parallelism, and D—Data parallelism.

by so-called data-flow languages. Communication via shared events (P1976) and via FIFO buffers (P998) occurs in one study each. As illustrated in Figure 12(a), 36/225 studies proposed languages providing parallel communication mechanism via shared memory and message passing mechanism, (3/225) proposed shared memory and data flow communication, 5/225 proposed message passing and data flow, and one study proposed the *split-C* language offering shared memory, message passing, and data flow based communication. Concerning the memory model (Figure 11(b)), the majority of languages target shared memory architectures (131/225). It is worth noticing that, in contrast with its virtual abstraction role in communication models, memory here is meant as "physical" memory. To exemplify, languages featuring shared memory in their communication model do not necessarily target architectures with shared physical memory. Distributed memory architectures, such as clusters, are also commonly targeted (96/225). Finally, these models might be enhanced by advanced variations, as in stack-based (22/225) and region-based (9/225) approaches. As with the previous categories, a given language might target



(a) Communication model (b) Memory model

Fig. 11. Programming model: communication and memory models.



(a) Combination of communication models. S - Shared memory, M - Message passing, D - Data flow

(b) Combination of memory models. St - Stack based, Sh - Shared, R - Region based, D - Distributed

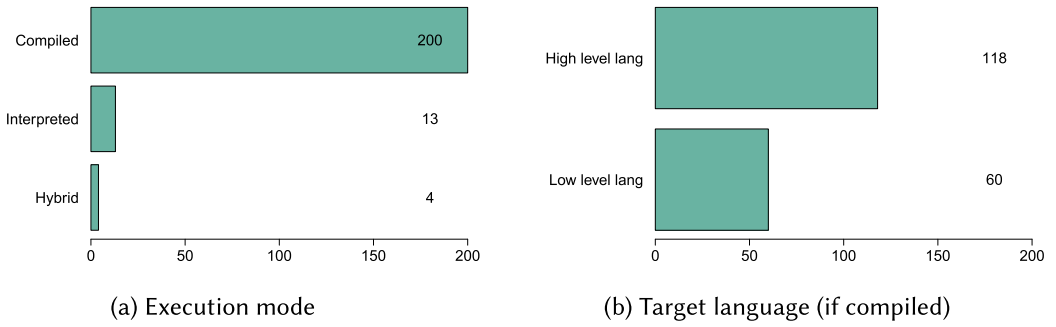
Fig. 12. Programming model combinations: communication and memory models.

more than one memory model, especially in those cases where memory access is abstracted through language constructs, i.e., implicit communication and data allocation. As depicted in Figure 12(b), in 42 primary studies, languages target both shared and distributed memory. Other combination of memory models include 3/225, 4/225, and 2/225 studies that proposed languages having stack-based and shared, stack-based and distributed, and shared and region-based memory models. There exists one study for each of the combinations of region-based and distributed, stack-based, shared, and distributed, and shared, region-based, and distributed memory models.

Highlights – RQ2.2 Programming model

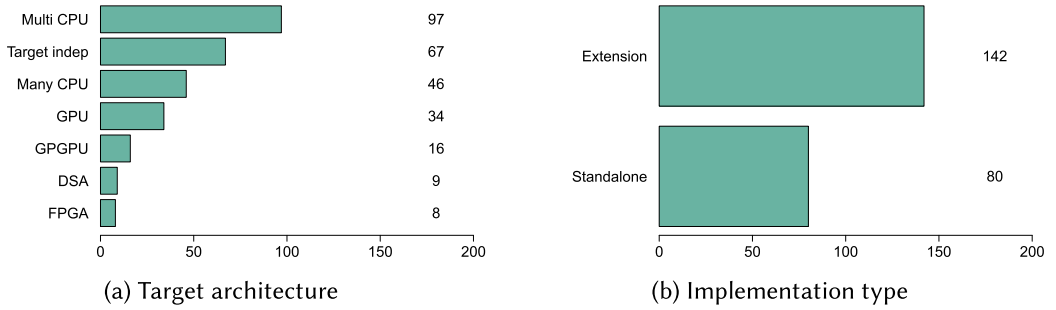
- Most languages target SIMD and MIMD architectures, while very few provide *architecture-independent* solutions.
- Task- and data-parallelism are the most common problem decomposition approaches, possibly supported by lower-level forms of parallelism, e.g. pipeline.
- Shared-memory, message-passing and data-flow are the most common communication models, in that order.
- The majority of languages target shared or distributed memory architectures, possibly enhanced into more advanced models, e.g. region-based approaches.

3.2.3 Execution, Run-time and Implementation (RQ2.3). Research question RQ2.3 concerns languages with respect to their run-time characteristics, more specifically how languages are implemented and executed. First, we classified the languages according to how they are executed, namely



(a) Execution mode (b) Target language (if compiled)

Fig. 13. Execution: mode and target language.



(a) Target architecture (b) Implementation type

Fig. 14. Target architecture and implementation type.

via interpretation, compilation or hybrid approaches. If compiled, we investigated whether the language(s) targeted by the compiler are low- or high-level languages. Eventually, we analysed which architectures are specifically targeted by the languages and how the language is implemented.

With regards to the execution mode (Figure 13(a)), it is evident that the community has heavily focused on developing compilation strategies (200/225); interpretation strategies were provided in 13 studies, while in four cases execution was achieved via hybrid approaches (e.g., mix of interpretation and JIT compilation). Looking at Figure 13(b), we can see how, in most cases (118/200), work on compilers focuses on generating programs in high-level languages (including compiler internal/intermediate languages). This shows that the bulk of the community's effort is devoted to so called front- and middle-ends, while back-ends tend to be reused. Nevertheless, there is a fair amount of compilation works (60/200) targeting low-level languages, thus focusing more on back-ends.

Regarding target architectures (Figure 14(a)), many approaches (97/225) focus on multi-core CPUs. Interestingly, the second biggest share of studies (67/225) focus on target-independent languages, followed by those focusing on many-core CPUs (46/225). GPUs are targeted by 34 studies, while GPGPUs by 16; DSAs, and FPGAs are mentioned as target architectures in nine and eight studies, respectively. Note that a number of studies (62/225) propose languages that are meant to support multiple target architectures. Concerning how languages are implemented (Figure 14(b)), the majority of the primary studies (142/225) present languages as extension of existing languages. In Table 8, we list the base languages extended to implement the languages reported in the primary studies. The most leveraged family of programming languages is C-based (49/142) followed by Java-based (20/142). On the modelling languages side, the UML family of languages is exploited in eight studies.

Table 8. Extended Languages

# occ.	Languages
25	C
21	C++
15	Java
5	Pascal
5	UML, Fortran
4	OpenMP
3	MARTE
2	Prolog, Ada, Modula-2, Haskell, Lisp, OpenCL, LM, X10
1	APARAPI, XPFortran, IA, C*, PAL, Euclid, Scala, CUDA, PL/1, C#, Insense, XSLT, Python, Mentat, JavaScript, Deterministic Parallel Java, OmpSs-OpenCL, R, SHMEM, JavaSymphony, ParCel-1, NestStep, OpenACC, TNC, Active Oberon, Orca, mpC, ReactoGraph, Qu-Prolog, PLASMA, XMP, MPL, BSP, Cactus, Rolez, CuPit, Habanero Java, occam2, Scheme, LINQ, MFL, HPF, Node.js, April, Cilk, MPI, Caml

Highlights – RQ2.3 Execution, run-time and implementation

- The large majority of parallel languages are compiled and the target is usually a high-level language, indicating a consolidated trend towards implementing front- and middle-ends rather than back-ends for new languages.
- CPUs (multi- and many-core) represent by large the most commonly targeted architectures; interestingly, over 30% of languages is not tailored to any specific target architecture.
- Most languages are defined in terms of extensions of existing languages; C/C++ and Java are the most leveraged programming languages, and UML the most used modelling language, for extension purposes.

3.3 Limitations (RQ3)

The analysis and synthesis of limitations presented in the primary studies were carried out by: (i) extracting text portions from the studies concerning limitations and planned future works, as described by the authors, and (ii) clustering them, if represented by ≥ 3 primary studies, in the comprehensive groups described next in descending order.

The most represented gaps were related to **supported data-parallelism** and expressed by the following needs:

- Support for describing data-parallelism, by providing annotations (P1097), by modelling (P140), or by programming (P268, P852, and P2023);
- Support for structured (P479), complex (P545), or user-defined (P498) data-types;
- Support for a broader set of data-parallel primitives (P608);
- Support for multi-dimensional arrays (P608);
- Support for irregular data structures (P07, P364);
- Support for data-parallelism at compilation level (P460);
- Support for data-parallel garbage collection (P633).

The second cluster of gaps was related to **supported target hardware platforms** and expressed as follows:

- Support for targeting multiple hardware platforms (P1288, P1351, P608, P268, P968, and P1752);

- Support for multi-GPU execution (P408, P2055, P1291, and P1700);
- Support for heterogeneous target platforms (P1717);
- Support for clusters of multi-cores (P1412).

The third cluster was related to **improvement of language and related tooling** and expressed as follows:

- Improvement of the specification and implementation of the language (P1632, P1804, P415, P2055, P480, and P768);
- Improvement of the provided compiler (P90, P1944, and P360);
- Extension of the language to implement full-blown OS running on bare hardware (P72);
- Support for external parallel libraries (P2120);
- Support for runtime execution (P444).

The fourth cluster was related to **provided analyses and optimisations** and expressed as follows:

- Enlargement of the set of provided program analyses, both static and dynamic, and optimisations (P05, P07, P1912, P1940, P968, and P1944);
- Support for advanced memory optimisations (such as usage of fast local memory on GPUs from high-level API in P1700, low-level memory access patterns in P02);
- Support for timing efficiency, through analysis and optimisation (P1923);
- Improvement of currently provided analyses (P846);
- Support for analysing how work-stealing affects energy consumption in power-critical mobile devices (P1097);
- Data-sharing optimisation to minimise overheads (P993).

The fifth cluster regarded the **evaluation** of the language and related tooling and was expressed as follows:

- Need of real-world evaluation (P1695, P805, P846, P980, and P1511);
- Need of scalability proof (P1738, P930, P1511, and P116);
- Need of performance proof (P498);

The sixth cluster concerned **schedulability** issues and was expressed as the need for improvement of scheduling policies (P1940, P446, P1752, and P443), also with support for priority schedulers (P1752). Improvement of **performance** in general (P1442, P2055, and P1291) and communication performance in particular (P114) as well as improvement of **portability** (P930, P116, and P1304) constitute the last two clusters.

Besides the clustered ones, there are a number of other gaps that are worth mentioning, as follows:

- Support for automatic generation of target configuration (P140, P1041), also exploit machine learning for predicting, for a given combination of application and hardware, the best way to execute the application (P1700);
- Enlargement of communication model (P52, P736);
- Trade-off between compatibility with existing compilers and performance gains (P1011, P1636);
- Improvement of usability (P2055);
- Support for interactive handling of exceptions (P1172);
- Support for both dynamic and static (at compile-time) concurrency (P1910).

Highlights – RQ3 Limitations

- Support for data-parallelism seems to be the most wished feature of existing parallel languages;
- Support for multiple hardware platforms (also heterogeneously mixed) is a longed feature too;
- In general, languages and related tooling are in most cases at a prototypical level;
- More support, especially automated, for analysis and optimisation is a common envisioned enhancement;
- There is an evident need of evaluation in real-world settings, especially to assess scalability and performance of languages for large-scale applications.

4 RESULTS: ORTHOGONAL ANALYSIS

In this section, we discuss interesting relations across categories of the extraction data. To do so, we cross-tabulated and grouped extracted data as well as we made comparisons between *pairs* of categories of the data extraction form. Through contingency tables (not included in the text for space reasons), we extracted and evaluated relevant pair-wise relations.

4.1 Problem Decomposition Vs. Parallel Primitives

Most languages offering explicit parallel primitives focus on task-parallelism (115/147). Such a high number of languages supporting explicit task-parallelism is motivated by the great expressive power offered to programmers, i.e., complete control over the parallel choreography; this comes with potential risks related to explicitly describing how tasks shall be synchronised though. In addition, introducing support for task-parallelism using explicit primitives represents a relatively simple solution for extending existing sequential languages. Explicit primitives and data-parallelism represent the second most common combination, although with much fewer occurrences (73/147). Despite being expectedly less frequent than explicit primitives, as seen in the vertical analysis (Figure 6(b)), implicit primitives in combination problem decomposition produces an interesting result. Indeed, the tendency to prefer task-parallelism (61%) rather than data-parallelism (39%) for explicit primitives is inverted when it comes to implicit primitives, where data-parallelism (55%) is slightly more common than task-parallelism (45%). Few languages support pipeline parallelism and code parallelism. Among these, the majority provides explicit primitives. It is interesting to notice that no language supporting nested parallelism provides implicit primitives.

4.2 Communication Model Vs. Memory Model

As seen in the vertical analysis (Figure 11(a)), shared-memory represents the most common communication model; the majority of languages combine this model with the shared memory model (89/105). However, it is interesting to notice that a fair number of languages also combine shared-memory communication model with distributed memory (31/105). Message-passing is the second most common communication model, but its combination with memory models provides inverted results. Most languages featuring message-passing target indeed the distributed memory model (66/99), with shared memory coming right after (41/99). Stack-based memory model is also moderately diffused in combination with shared memory (8/21) and message passing (13/21) communication models. Finally, languages adopting data flow communication tend to prefer the distributed (16/31) and shared memory (10/31) models, with a slight preference for the first.

4.3 Communication and Memory Models Vs. Target Architecture

Shared memory and message passing communication models are the most popular choices among languages targeting multi-core CPU (52/98 and 41/98, respectively). These two communication models are used a lot with many-core CPU architectures too (32/52 and 30/52, respectively). This applies even to target-independent languages (29/70 and 37/70 entries). Shared memory appears

to be the most convenient medium of communication for various accelerator architectures such as DSA, GPU, GPGPU, and FPGA than message passing or data flow. Nevertheless, the presence of data flow communication in all target architectures (including target-independent), even though with a lower number of occurrences than shared memory or message passing communication model, shows its importance as communication model.

The relation between memory model and target architecture is rather symmetric to the one between communication model and target architecture. More specifically, shared and distributed memory models are the most popular choices in languages targeting multi-core CPU (59/98 and 48/98, respectively). A similar distribution applies to languages targeting many-core CPU (31/52 and 30/52, respectively) and target-independent languages (34/70 and 29/70, respectively). As for the shared memory communication model, the shared memory model is the most common choice for accelerator architectures such as DSA, GPU, GPGPU, and FPGA.

5 OPEN CHALLENGES AND PROSPECTS

From our vertical analysis (RQ2), it appeared that languages for parallel computing are in most cases high-level and general-purpose. This is understandable since the first ensures an acceptable abstraction level for the programmer to be able to express his functions without the need to handle machine-specific details. The second makes those languages usable in multiple application domains and for multiple purposes. Nevertheless, since parallel and heterogeneous hardware is getting evermore common in fields where experts do not necessarily have a background in parallel programming (e.g., health, biology), we believe that research should direct more attention toward the definition of languages at **higher-levels of abstraction** (i.e., modelling languages). Moreover, mechanisms for specialising (and thereby shrinking) general-purpose languages to **domain-specific variants** of it would be helpful too, since they would allow domain-specific analyses and optimisations as well as require a less steep learning/training curve.

Most parallel languages provide explicit means to describe parallel computations. On the one hand, this allows experienced programmers to make fine-grained optimisations and provide very efficient parallel algorithms; on the other hand, this makes parallel languages hard to use for the less experienced users. To make parallel programming more accessible to those without parallel programming skills, it would be useful to carry out more research in the provision of **implicit parallelism**, not necessarily as a mutually exclusive alternative to explicit parallel primitives but also in combination. Efficient ways to provide data parallelism is regarded as the most wished feature for existing languages (RQ4); there is definitely space for contributing to the body of knowledge and state of the practice by focusing on **implicit data parallelism**.

Through our analysis (RQ2), we got a confirmation that multi- and many-core CPUs represent the most commonly targeted architectures. Although already growing in number of related publications, there is still space for significant contributions and advances in languages for other architectures, especially FPGAs and GPUs, and even more for languages **targeting multiple architectures**, even in combination. The latter is in fact one of the most commonly mentioned longed feature in the analysed studies (RQ4).

In the definition of new languages, we tend to believe that **a balanced mix of concepts from object-oriented and functional programming** could provide a fruitful combination of ingredients for achieving both explicit and implicit parallelism. Moreover, it would allow to push up the level of abstraction, thus making those languages more accessible and less dependent on the target architectures; multiple levels of refinement (e.g., through modelling), could permit to separate functional aspects from allocation, configuration and deployment ones [2]. In addition, we believe that **blending notations** [58] for stakeholders to interact with high-level (modelling) languages in a flexible and personalised manner has the potential to boost the usage and productivity of these

languages [1], thus making a broader set of stakeholders keen on using them, even for multiple purposes—not only programming, but also design and communication across design and developing teams. For instance, while textual notations could be used for algorithmic parts, graphical ones could be employed to model the target architecture and possibly the allocation of functional items to it; this would be particularly useful in case of complex heterogeneous architectures.

As expected, languages and supporting tools are in most cases at a prototypical level. This is a typical issue of academic research where industrial actors are not involved enough to give a hand in taking results a step (or in most cases several steps) further and to reach industrial grade. We believe that, together with classic networks of researchers and practitioners (e.g., HIPEAC[5]), the growing industrial consortia (e.g., Capella[6] and INCOSE[7] in systems engineering), which in the last few years have increasingly focused on open-source solutions, should be leveraged more by academics in this area as a precious vehicle for pushing research results to new heights and to reach a broader audience.

6 THREATS TO VALIDITY

There are two aspects that make us confident on the quality of our study. The first one is that we defined and meticulously followed a detailed research protocol document, which was subject to external reviews by independent researchers. The second one is that we conducted our study by following well-established guidelines for systematic studies. In any case, it is important to discuss the potential threats to the validity of our work and its outcomes, as well as the means undertaken to mitigate them.

6.1 External Validity

External validity refers to the generalisability of causal findings with respect to the desired population and settings [254]. In a systematic review, one of the most severe threats is that the selected set of primary studies is not representative of the state of the art on languages for parallel computing. This potential threat to validity is mitigated by targeting multiple data sources, i.e., ACM Digital Library, IEEE Xplore, Scopus, and Web of Science. The used data sources cover the area of software engineering well [41] and are five of the largest and most complete scientific databases and indexing systems in software engineering. Moreover, we further complement the results of the automatic searches by performing closed recursive backward and forward snowballing.

Only studies published in the English language were included. Although, this decision could result in a possible threat to validity due to potentially missing primary studies published in other languages, English is the de-facto standard language for scientific papers, so the threat can be reasonably considered negligible.

6.2 Internal Validity

Internal validity refers to extraneous variables and inaccurate settings that may have had a negative impact on the design of the study [254]. We mitigated this potential threat by defining meticulously the process to follow and the data extraction form according to well-established guidelines. Concerning the validity of the data analysis and synthesis, threats were minimal since we used descriptive statistics. Moreover, we cross-analysed the different categories of the data extraction form to make a sanity check of the extracted data. This task made us identify and solve potential

[5] https://www.hipeac.net/.
[6] https://www.eclipse.org/capella/.
[7] https://www.incose.org/.

issues on the consistency of the extracted data and make us confident of the internal validity of our study.

6.3 Construct Validity

Construct validity refers to the extent to which an identified causal relationship can be generalised from the particular methods and operations of a specific study to the theoretical constructs and processes they were meant to represent [254]. As already argued for external validity, the initial automated search has been performed on four different data sources and complemented with snowballing. One remaining potential threat to validity may be caused by a malformed search string. We mitigated this potential threat by (i) following a rigorous process for defining it and (ii) piloting the search string in preliminary queries using all four data sources. Once we gathered all relevant studies from the automatic search, we rigorously screened them in a collaborative fashion and according to well-documented inclusion and exclusion criteria. Moreover, the selection team members classified a certain number of common studies (~15%) in order to check agreement, achieving a Cohen's kappa [28] of **0.87** (almost perfect agreement according to Cohen's suggested interpretation).

6.4 Conclusion Validity

This refers to the relationship between extracted data and obtained findings [254]. We mitigated potential threats by systematically applying and documenting well-defined processes; we also provide a publicly accessible replication package,[8] which allows to replicate each step of our study. The data extraction form definition can be a notable threat to conclusion validity if based on own experience or other informal sources. We mitigated this by (i) letting the categories and their values emerge from the pilot studies and refining them throughout the entire data extraction activity, and (ii) making five researchers be actively involved in the definition of the form as well as the extraction and analysis/synthesis of data.

7 RELATED WORK

In the past few years, systematic literature reviews have gained considerable popularity in many research areas, in particular, software engineering [133]. In this article, we conduct a systematic literature review of programming and modeling languages for parallel computing platforms. This topic overlaps with the research areas of software engineering, computer science, and computer engineering. The research area targeted in this article is quite dispersed as there is a large body of research on programming and modelling languages for parallel computing platforms. This is indicated by 219 identified publications as shown in Figure 2 and discussed in Section 3.

There are very few systematic literature reviews, systematic mapping studies, and surveys of the state-of-the-art that discuss some aspects of parallel computing platforms. The most recent one is a survey on parallel programming models by Fang et al. [81], Brodtkorb et al. [44] conduct an investigation of the state-of-the-art in the area of heterogeneous computing. In this study, they limit the scope of their investigation by focusing on only three heterogeneous platforms, namely, the Cell Broadband Engine Architecture, GPU, and FPGA. Similar to the work in [44], Mittal and Vetter [170] present a survey of heterogeneous computing techniques, where they only consider CPU-GPU computing platform. Fernando et al. [82] perform a systematic mapping study to create a structured map of the existing research with respect to parallel computing in various execution platforms such as multi-core, cluster, and GPU. Similarly, Liu et al. [151] conduct a survey of

[8]The replication package is attached as extra files archive to this manuscript for review purposes and will be made public once the manuscript is in press.

software and hardware aspects of heterogeneous computing platforms. In this work, the authors focus only on a specific architecture, namely, the **Heterogeneous System Architecture** (**HSA**). However, these works do not investigate programming and modelling languages for the considered parallel computing platforms. In comparison to the above mentioned works, we conduct a systematic literature review of programming and modelling languages for generally a broad range of parallel computing platforms.

Andrade and Crnkovic [14] conduct a systematic mapping study of software architectures for heterogeneous computing platforms. The authors identify 28 publications of interest and using these they present various trends and identify gaps in the research area. Similarly, Andrade et al. [15] conduct a systematic mapping study that focuses on the deployment of software on heterogeneous computing platforms. In this study, the authors provide an overview of the research area with the aim at identifying and categorising the published research results according to a defined classification. Although the studies in [14, 15] focus on heterogeneous computing platforms, they do not consider programming and modelling languages. In comparison, this article presents a systematic literature review of programming and modelling languages for parallel computing while considering homogeneous as well as heterogeneous computing platforms.

There are a few works that study and survey the existing programming languages and programming models for multi-core and many-core computing platforms. For instance, Diaz et al. [75] present a survey of parallel and distributed programming models for multi-core and many-core processors. Soares et al. [143] present a systematic mapping study of parallel programming on multi-core platforms. However, their core focus is on the programming languages that are used in undergraduate education. Frank et al. [87] conduct a systematic literature review on parallelisation, modelling and performance prediction of applications that are run on multi-core and many-core processors. In this work, the authors identify and analyse 34 relevant publications that mainly focus on providing support for performance prediction of parallel applications on these parallel computing platforms. However, both these works focus only on homogeneous multi-core and many-core computing platforms. In comparison to these works, the systematic literature review conducted in our article identifies and analyses parallel programming and modeling languages for homogeneous as well as heterogeneous parallel computing platforms.

Kessler and Keller [129] review and analyse a few parallel computing models with respect to execution styles, type of parallelism (data and task parallelism), memory and communication models. Memeti et al. [166] perform a comparative evaluation of four parallel programming frameworks, namely OpenMP, OpenCL, OpenACC, and CUDA. The evaluation parameters considered in this study include programming productivity, performance, and energy. In comparison with these works, our study is comprehensive in the sense that it considers a wide range of programming and modelling languages, programming models, target hardware architectures (SISD, SIMD, MISD, and MIMD), communication and memory models, and parallel execution models.

Nestmann [174] performs a systematic literature review with the aim at identifying a taxonomy for parallel programming models. The authors identify five different taxonomies in the existing literature. Based on the identified taxonomies, the authors build a consolidated and consistent taxonomy for parallel programming models. Amaral et al. [11] conduct a systematic mapping study of programming languages for **High-performance Computing** (**HPC**) platforms. In this study, the authors identify 26 programming languages for the HPC platforms that are presented in 33 relevant publications. Although this work is closely related to the work presented in our article, there are a number of key differences between the two works. That is, Amaral et al. [11] present a systematic mapping study focusing on a large-scale cluster of computing nodes that are connected by networks. Furthermore, the heterogeneity in the parallel computing platforms is not explicitly addressed. On the other hand, we present a systematic literature review that considers

programming and modelling languages for a broad range of homogeneous and heterogeneous parallel computing platforms ranging from on-chip and on-board to large-scale cluster platforms.

To the best of our knowledge, the state-of-the-art is lacking a systematic literature review on programming and modelling languages for parallel computing platforms. There is an urgent need for constructing a structured map of the research area, analysing the existing works, and identifying various trends and gaps in the area. The work presented in this paper takes a major step in addressing these needs.

8 CONCLUSIONS AND FUTURE DIRECTIONS

In this article, we reported on the planning, execution, and results of a systematic literature review on languages for describing parallel software. From an initial set of 3,476 papers (3,469 from automatic searches plus seven pilot studies) and 72 languages, after a set of refinements, we selected 225 *primary* studies to represent the treated topic. We extracted data from them according to a thoroughly defined data extraction process; then, we analysed and synthesised the data. The result of this work was a comprehensive, structured, and detailed snapshot of the current landscape on languages for parallel computing, useful for both the more and the less experienced researcher or practitioner. Additionally, based on the study results and our own interpretation of the current and forthcoming trends in relevant application domains, we provided hints on challenges that remain open as well as on what we believe being some of the "hot" research directions to pursue in this research area. An interesting extension to this study could be a separate review of embedded/internal domain-specific languages for parallel computing, where the focus would be on aspects related to their design/implementation and the relation to the host languages. A retrospective on the host languages and the additional features brought by hosted languages could be then synthesised too. Moreover, another direction for extending this study could be represented by an in-depth analysis of the new challenges in HPC systems in terms of programming, such as those related to persistent memory.

REFERENCES

[1] Lorenzo Addazi, Federico Ciccozzi, Philip Langer, and Ernesto Posse. 2017. Towards seamless hybrid graphical-textual modelling for UML and profiles. In *Proceedings of the 13th European Conference on Modelling Foundations and Applications*. Retrieved from http://www.es.mdh.se/publications/4751-.

[2] Lorenzo Addazi, Federico Ciccozzi, and Björn Lisper. 2019. Executable modelling for highly parallel accelerators. In *Proceedings of the Workshop on Modelling Language Engineering and Execution at MoDELS*.

[3] Vikas Aggarwal, Alan D. George, Changil Yoon, Kishore Yalamanchili, and Herman Lam. 2011. SHMEM+ A multilevel-PGAS programming model for reconfigurable supercomputing. *ACM Transactions on Reconfigurable Technology and Systems* 4, 3 (2011), 1–24.

[4] Snorri Agnarsson. 2010. Parallel programming in morpho. In *Proceedings of the International Workshop on Applied Parallel Computing*. Springer, 97–107.

[5] Gita Alaghband and Harry F. Jordan. 1994. Overview of the force scientific parallel language. *Scientific Programming* 3, 1 (1994), 33–47.

[6] Marco Aldinucci et al. 2003. eskimo: Experimenting skeletons on the shared address model. In *Proceedings of the 2nd International Workshop on*, Vol. 16. Computer Science Department, University of Pisa, Italy, 44–58.

[7] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2012. Targeting distributed systems in fastflow. In *Proceedings of the European Conference on Parallel Processing*. Springer, 47–56.

[8] Muhammad Aleem, Radu Prodan, and Thomas Fahringer. 2012. The javasymphony extensions for parallel gpu computing. In *Proceedings of the 2012 41st International Conference on Parallel Processing*. IEEE, 30–39.

[9] Nauman Bin Ali and Kai Petersen. 2014. Evaluating strategies for study selection in systematic literature studies. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. ACM.

[10] Makoto Amamiya, Masahiko Satoh, Akifumi Makinouchi, Ken-ichi Hagiwara, Taiichi Yuasa, Hitoshi Aida, Kazunori Ueda, Keijiro Araki, Tetsuo Ida, and Takanobu Baba. 1994. Research on programming languages for massively parallel processing. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*. IEEE, 443–450.

[11] Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, Helen Karatza, Christoph Kessler, Peter Kilpatrick, Hugo Martiniano, Ilias Mavridis, Sabri Pllana, Ana Respício, José Simão, Luís Veiga, and Ari Visa. 2020. Programming languages for data-Intensive HPC applications: A systematic mapping study. *Parallel Computing* 91 (2020), 102584. DOI: https://doi.org/10.1016/j.parco.2019.102584

[12] Manel Ammar, Mouna Baklouti, and Mohamed Abid. 2012. Extending MARTE to support the specification and the generation of data intensive applications for massively parallel SoC. In *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*. IEEE, 715–722.

[13] Birger Andersen. 1994. A general, fine-grained, machine independent, object-oriented language. *ACM SIGPLAN Notices* 29, 5 (1994), 17–26.

[14] Hugo Andrade and Ivica Crnkovic. 2018. A review on software architectures for heterogeneous platforms. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 209–218.

[15] Hugo Andrade, Jan Schroeder, and Ivica Crnkovic. 2019. Software deployment on heterogeneous platforms: A systematic mapping study. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1–1.

[16] Keijiro Araki, Itsujiro Arita, and Masaki Hirabaru. 1984. NANO-2: HIGH-LEVEL PARALLEL PROGRAMMING LANGUAGE FOR MULTIPROCESSOR SYSTEM HYPHEN. In *Proceedings-IEEE Computer Society International Conference*. IEEE, 449–456.

[17] F. Arbab, J. W. De Bakker, M. M. Bonsangue, J. J. M. M. Rutten, A. Scutella, and G. Zavattaro. 2000. A transition system semantics for the control-driven coordination language MANIFOLD. *Theoretical Computer Science* 240, 1 (2000), 3–47.

[18] Takanobu Baba and Tsutomu Yoshinaga. 1995. A-NETL: A language for massively parallel object-oriented computing. In *Proceedings of the Programming Models for Massively Parallel Computers*. IEEE, 98–105.

[19] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. 2007. The OpenTM transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE, 376–387.

[20] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*. IEEE, 138–149.

[21] Mouna Baklouti, Manel Ammar, Philippe Marquet, Mohamed Abid, and Jean-Luc Dekeyser. 2011. A model-driven based framework for rapid parallel SoC FPGA prototyping. In *Proceedings of the 2011 22nd IEEE International Symposium on Rapid System Prototyping*. IEEE, 149–155.

[22] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. 1968. The ILLIAC IV computer. *IEEE Transactions on Computers* C-17, 8 (1968), 746–757.

[23] Pablo Basanta-Val and Marisol García-Valls. 2015. A library for developing real-time and embedded applications in C. *Journal of Systems Architecture* 61, 5–6 (2015), 239–255.

[24] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. The goal question metric approach. In *Proceedings of the Encyclopedia of Software Engineering*. Vol. 2. Wiley, 528–532.

[25] Michael Bauer, John Clark, Eric Schkufza, and Alex Aiken. 2010. Sequoia++ User Manual, University of Chicago. Retrieved from https://www.classes.cs.uchicago.edu/archive/2011/winter/32102-1/reading/sequoia-manual.pdf. (2010). Accessed: 2021-06-15.

[26] Saniya Ben Hassen, Henri E. Bal, and Ceriel J. H. Jacobs. 1998. A task-and data-parallel programming language based on shared objects. *ACM Transactions on Programming Languages and Systems* 20, 6 (1998), 1131–1170.

[27] Siegfried Benkner and Viera Sipkova. 2003. Exploiting distributed-memory and shared-memory parallelism on clusters of smps with data parallel programs. *International Journal of Parallel Programming* 31, 1 (2003), 3–19.

[28] Kenneth J. Berry and Paul W. Mielke Jr. 1988. A generalization of Cohen's kappa agreement measure to interval measurement and multiple raters. *Educational and Psychological Measurement* 48, 4 (1988), 921–933.

[29] Matthias Besch, Hua Bi, Gerd Heber, Matthias Kessler, and Matthias Wilhelmi. 1997. An object-oriented approach to the implementation of a high-level data parallel language. In *Proceedings of the International Conference on Computing in Object-Oriented Parallel Environments*. Springer, 97–104.

[30] Lorenzo Bettini, Rocco De Nicola, Rosario Pugliese, and Gian Luigi Ferrari. 1998. Interactive mobile agents in X-Klaim. In *Proceedings 7th IEEE International Workshop on Enabling Technologies: Infrastucture for Collaborative Enterprises*. IEEE, 110–115.

[31] Suchendra M. Bhandarkar and Bonnie M. Edwards. 1995. HyperPascal-an architecture independent Pascal interface for parallel programming. In *Proceedings IEEE Southeastcon'95. Visualize the Future*. IEEE, 78–84.

[32] Ganesh Bikshandi, Jose G. Castanos, Sreedhar B. Kodali, V. Krishna Nandivada, Igor Peshansky, Vijay A. Saraswat, Sayantan Sur, Pradeep Varma, and Tong Wen. 2009. Efficient, portable implementation of asynchronous multi-place programs. *ACM Sigplan Notices* 44, 4 (2009), 271–282.

[33] Roberto Bisiani and Alessandro Forin. 1987. Architectural support for multilanguage parallel programming on heterogeneous systems. *ACM SIGPLAN Notices* 22, 10 (1987), 21–30.
[34] Luc Bläser. 2006. A component language for structured parallel programming. In *Proceedings of the Joint Modular Languages Conference*. Springer, 230–250.
[35] Marek Blazewicz, Steven R. Brandt, Michal Kierzynka, Krzysztof Kurowski, Bogdan Ludwiczak, Jian Tao, and Jan Weglarz. 2011. CaKernel–a parallel application programming framework for heterogenous computing architectures. *Scientific Programming* 19, 4 (2011), 185–197.
[36] Robert L. Bocchino Jr, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. *ACM SIGPLAN Notices* 46, 1 (2011), 535–548.
[37] Daniele Bonetta, Danilo Ansaloni, Achille Peternier, Cesare Pautasso, and Walter Binder. 2012. Node.scala: Implicit parallel programming for high-performance web services. In *Proceedings of the European Conference on Parallel Processing*. Springer, 626–637.
[38] Daniele Bonetta, Luca Salucci, Stefan Marr, and Walter Binder. 2016. Gems: Shared-memory parallel programming for node. js. *ACM SIGPLAN Notices* 51, 10 (2016), 531–547.
[39] George Horaţiu Botorog and Herbert Kuchen. 1998. Efficient high-level parallel programming. *Theoretical Computer Science* 196, 1–2 (1998), 71–107.
[40] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel objects for multicores: A glimpse at the parallel language encore. In *Proceedings of the International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 1–56.
[41] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software* 80, 4 (2007), 571–583.
[42] Peter Brezány. 1983. Denotational semantics of parallel programming languages. *Kybernetika* 19, 3 (1983), 248–262.
[43] Ron Brightwell, Mike Heroux, Zhaofang Wen, and Junfeng Wu. 2009. Parallel phase model: A programming model for high-end parallel machines with manycores. In *Proceedings of the 2009 International Conference on Parallel Processing*. IEEE, 92–99.
[44] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. 2010. State-of-the-art in heterogeneous computing. *Sci. Program.* 18, 1 (Jan. 2010), 1–33. DOI: https://doi.org/10.1155/2010/540159
[45] Ansgar Brüll and Herbert Kuchen. 1996. TPascal–A language for task parallel programming. In *Proceedings of the European Conference on Parallel Processing*. Springer, 654–659.
[46] Paul-Jean Cagnard. 2000. The ParCeL-2 programming language. In *Proceedings of the European Conference on Parallel Processing*. Springer, 767–770.
[47] Ran Canetti, L. Paul Fertig, Saul A. Kravitz, Dalia Malki, Ron Y. Pinter, Sara Porat, and Avi Teperman. 1991. The parallel C (pC) programming language. *IBM Journal of Research and Development* 35, 5.6 (1991), 727–741.
[48] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. 2006. The Atomos transactional programming language. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1–13.
[49] Bryan Carpenter and Geoffrey Fox. 2003. HPJava: A data parallel programming alternative. *Computing in Science and Engineering* 5, 3 (2003), 60–64.
[50] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. 47–56.
[51] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. 51–61.
[52] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.
[53] Rohit Chandra, Anoop Gupta, and John L. Hennessy. 1994. COOL: An object-based language for parallel programming. *Computer* 27, 8 (1994), 13–26.
[54] K. Mani Chandy and Ian Foster. 1995. A notation for deterministic cooperating processes. *IEEE Transactions on Parallel and Distributed Systems* 6, 8 (1995), 863–871.
[55] Chapel, Version 1.24.1. 2021. The Chapel Parallel Programming Language. Retrieved from https://chapel-lang.org/. (2021). Accessed: 2021-06-15.
[56] Charm++ Programming Language, Release 6.10.2. 2020. Retrieved from http://charmplusplus.org/. (2020). Accessed: 2021-06-15.
[57] Kathy Charmaz and Linda Liska Belgrave. 2007. Grounded theory. *The Blackwell Encyclopedia of Sociology* (2007).

[58] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. 2019. Blended modelling – what, why and how. In *Proceedings of the MPM4CPS workshop*. Retrieved form http://www.es.mdh.se/publications/5642-.
[59] R. M. Clapp and T. N. Mudge. 1992. Parallel language constructs for efficient parallel processing. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, Vol. 2. IEEE, 230–241.
[60] Keith Clark and Peter J. Robinson. 2002. Agents as multi-threaded logical objects. In *Proceedings of the Computational Logic: Logic Programming and Beyond*. Springer, 33–65.
[61] Keith L. Clark and Francis G. McCabe. 2004. Go!–a multi-paradigm programming language for implementing multi-threaded agents. *Annals of Mathematics and Artificial Intelligence* 41, 2–4 (2004), 171–206.
[62] Robert Clucas and Stephen Levitt. 2015. CAPP: A C++ aspect-oriented based framework for parallel programming with OpenCL. In *Proceedings of the 2015 Annual Research Conference on South African Institute of Computer Scientists and Information Technologists*. 1–10.
[63] Paul Cockshott and Greg Michaelson. 2006. Orthogonal parallel processing in vector Pascal. *Computer Languages, Systems and Structures* 32, 1 (2006), 2–41.
[64] Adrian Colbrook, William E. Weihl, Eric A. Brewer, Chrysanthos N. Dellarocas, Wilson C. Hsieh, Anthony D. Joseph, Carl A. Waldspurger, and Paul Wang. 1992. Portable software for multiprocessor systems. *Computing and Control Engineering Journal* 3, 6 (1992), 275–281.
[65] Philip Cox, Simon Gauvin, and Andrew Rau-Chaplin. 2005. Adding parallelism to visual data flow programs. In *Proceedings of the 2005 ACM Symposium on Software Visualization*. 135–144.
[66] D. Crookes, P. J. Morrow, and P. J. McParland. 1990. IAL: A parallel image processing programming language. *IEE Proceedings I (Communications, Speech and Vision)* 137, 3 (1990), 176–182.
[67] Flavio Cruz, Ricardo Rocha, and Seth Copen Goldstein. 2015. Thread-aware logic programming for data-driven parallel programs. ICLP.
[68] Flavio Cruz, Ricardo Rocha, and Seth Copen Goldstein. 2016. Declarative coordination of graph-based parallel programs. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.
[69] Daniela S. Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 275–284.
[70] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. 1993. Parallel programming in Split-C. In *Supercomputing'93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. IEEE, 262–273.
[71] Marco Danelutto, Roberto Di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. 1992. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems* 8, 1–3 (1992), 205–220.
[72] Tiago José Barreiros Martins de Almeida and Nuno Filipe Valentim Roma. 2010. A parallel programming framework for multi-core DNA sequence alignment. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 907–912.
[73] Pablo de Oliveira Castro, Stéphane Louise, and Denis Barthou. 2010. A multidimensional array slicing dsl for stream programming. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 913–918.
[74] Francisco de Sande, Felix Garcia, Coromoto Leon, and Casiano Rodriguez. 1996. The II parallel programming system. *IEEE Transactions on Education* 39, 4 (1996), 457–464.
[75] J. Diaz, C. Muñoz-Caro, and A. Niño. 2012. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems* 23, 8 (2012), 1369–1386.
[76] Walter dos Santos, Luiz F. M. Carvalho, Gustavo de P. Avelar, Átila Silva, Lucas M. Ponce, Dorgival Guedes, and Wagner Meira. 2017. Lemonade: A scalable and efficient Spark-based platform for data analytics. In *Proceedings of the 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 745–748.
[77] Vinoth Krishnan Elangovan, Rosa M. Badia, and Eduard Ayguadé. 2014. Scalability and parallel execution of ompss-opencl tasks on heterogeneous cpu-gpu environment. In *Proceedings of the International Supercomputing Conference*. Springer, 141–155.
[78] Hajime Enomoto, Naoki Yonezaki, Isao Miyamura, and Masayuki Sunuma. 1980. A parallel programming language and description of scheduler. In *Proceedings of the IBM Germany Scientific Symposium Series*. Springer, 23–41.
[79] Erlang Programming Language, OTP Release 24.0. 2021. Build Massively Scalable Soft Real-time Systems. Retrieved from https://www.erlang.org/. (2021). Accessed: 2021-06-15.
[80] Michael Faes and Thomas R. Gross. 2018. Concurrency-aware object-oriented programming with roles. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.
[81] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Parallel programming models for heterogeneous many-cores: A comprehensive survey. *CCF Transactions on High Performance Computing* 2, 4 (2020), 382–400.

[82] E. Fernando, D. F. Murad, and B. D. Wijanarko. 2018. Classification and advantages parallel computing in process computation: A systematic literature review. In *Proceedings of the International Conference on Computing, Engineering, and Design*. 143–147.

[83] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2009. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proceedings of the LSDS-IR*. (2009).

[84] Ian Finlayson, Jerome Mueller, Shehan Rajapakse, and Daniel Easterling. 2015. Introducing tetra: An educational parallel programming system. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 746–751.

[85] Alcides Fonseca, João Rafael, and Bruno Cabral. 2014. Eve: A parallel event-driven programming language. In *European Conference on Parallel Processing*. Springer, Cham, 170–181.

[86] Rhys S. Francis, Ian D. Mathieson, Paul G. Whiting, Martin R. Dix, Harvey L. Davies, and Leon D. Rotstayn. 1994. A data parallel scientific modeling language. *Journal of Parallel and Distributed Computing* 21, 1 (1994), 46–60.

[87] M. Frank, M. Hilbrich, S. Lehrig, and S. Becker. 2017. Parallelization, modeling, and performance prediction in the multi-/many core area: A systematic literature review. In *Proceedings of the 7th IEEE International Symposium on Cloud and Service Computing*. 48–55.

[88] Robert Frank and Helmar Burkhart. 1997. Application support by software reuse: The ALWAN approach. In *Proceedings of the International Conference on High-Performance Computing and Networking*. Springer, 636–643.

[89] Roberto Franzosi. 2010. *Quantitative Narrative Analysis*. Number 162. Sage.

[90] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. 2015. Runtime code generation and data management for heterogeneous computing in java. In *Proceedings of the Principles and Practices of Programming on the Java Platform*. 16–26.

[91] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. 2011. A model-driven design framework for massively parallel embedded systems. *ACM Transactions on Embedded Computing Systems* 10, 4 (2011), 1–36.

[92] Sofien Gannouni. 2015. A gamma-calculus GPU-based parallel programming framework. In *Proceedings of the 2015 2nd World Symposium on Web Applications and Networking*. IEEE, 1–4.

[93] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), 101–121.

[94] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar, and Patrick Miller. 2001. *The Sisal Project: Real World Functional Programming*. Springer Berlin Heidelberg, Berlin, 45–72. DOI: https://doi.org/10.1007/3-540-45403-9_2

[95] David Gay, Joel Galenson, Mayur Naik, and Kathy Yelick. 2011. Yada: Straightforward parallel programming. *Parallel computing* 37, 9 (2011), 592–609.

[96] Michael Gerndt and Andreas Krumme. 1997. A rule-based approach for automatic bottleneck detection in programs on shared virtual memory systems. In *Proceedings 2nd International Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE, 93–101.

[97] Stefan J. Geuns, Joost P. H. M. Hausmans, and Marco J. G. Bekooij. 2014. Hierarchical programming language for modal multi-rate real-time stream processing applications. In *Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops*. IEEE, 453–460.

[98] Fady Ghanim, Uzi Vishkin, and Rajeev Barua. 2017. Easy PRAM-based high-performance parallel programming with ICE. *IEEE Transactions on Parallel and Distributed Systems* 29, 2 (2017), 377–390.

[99] Go Programming Language. Retrieved from 2021. https://golang.org/. (2021). Accessed: 2021-06-15.

[100] Ron Goldman and Richard Gabriel. 1988. Preliminary results with the initial implementation of Qlisp. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. 143–152.

[101] Bart Goossens, Hiep Luong, Jan Aelterman, and Wilfried Philips. 2018. Quasar, a high-level programming language and development environment for designing smart vision systems on embedded platforms. In *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 1310–1315.

[102] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. 2011. ΣC: A programming model and language for embedded manycores. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 385–394.

[103] Trisha Greenhalgh and Richard Peacock. 2005. Effectiveness and efficiency of search methods in systematic reviews of complex evidence: Audit of primary sources. *BMJ* 331, 7524 (2005), 1064–1065.

[104] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC–a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427.

[105] Andrew S. Grimshaw. 1993. Easy-to-use object-oriented parallel processing with Mentat. *Computer* 26, 5 (1993), 39–51.

[106] Andrew S. Grimshaw, Ami Silberman, and Jane W. S. Liu. 1989. Real-time Mentat programming language and architecture. In *Proceedings of the 1989 IEEE Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'*. IEEE, 141–147.
[107] Radu Grosu, Yanhong A. Liu, Scott Smolka, Scott D. Stoller, and Jingyu Yan. 2001. Automated software engineering using concurrent class machines. In *Proceedings 16th Annual International Conference on Automated Software Engineering*. IEEE, 297–304.
[108] Steven A. Guccione and Mario J. Gonzalez. 1993. A data-parallel programming model for reconfigurable architectures. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*. IEEE, 79–87.
[109] Vadim Guzev. 2008. Parallel C#: The usage of chords and higher-order functions in the design of parallel programming languages. In *Proceedings of the PDPTA*. 833–837.
[110] Wieland Hagen, Max Plauth, Felix Eberhardt, Frank Feinbube, and Andreas Polze. 2016. PGASUS: A framework for C++ application development on NUMA architectures. In *Proceedings of the 2016 4th International Symposium on Computing and Networking*. IEEE, 368–374.
[111] Michael Haidl and Sergei Gorlatch. 2018. High-level programming for many-cores using C++ 14 and the STL. *International Journal of Parallel Programming* 46, 1 (2018), 23–41.
[112] Robert H. Halstead Jr. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (1985), 501–538.
[113] Per Brinch Hansen. 1994. Interference control in SuperPascal–a block-structured parallel language. *Computer Journal* 37, 5 (1994), 399–406.
[114] Paul Harvey, Kristian Hentschel, and Joseph Sventek. 2015. Parallel programming in actor-based applications via OpenCL. In *Proceedings of the 16th Annual Middleware Conference*. 162–172.
[115] Kenneth A. Hawick and Heath A. James. 2000. A java-based parallel programming support environment. In *Proceedings of the International Conference on High-Performance Computing and Networking*. Springer, 363–372.
[116] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 556–571.
[117] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. 2013. River trail: A path to parallelism in JavaScript. *ACM SIGPLAN Notices* 48, 10 (2013), 729–744.
[118] J. Hernandez, Pedro de Miguel, M. Barrena, Juan Miguel Martinez, A. Polo, and M. Nieto. 1993. ALBA: A parallel language based on actors. *ACM SIGPLAN Notices* 28, 4 (1993), 11–20.
[119] Rich Hickey. 2020. A history of clojure. *Proceedings of the ACM on Programming Languages* 4, HOPL, Article 71 (June 2020), 46 pages. DOI: https://doi.org/10.1145/3386321
[120] Holger Hopp and Lutz Prechelt. 1999. CuPit-2: Portable and efficient high-level parallel programming of neural networks. *Systems Analysis Modelling Simulation* 35, 4 (1999), 379–398.
[121] Qiming Hou, Kun Zhou, and Baining Guo. 2008. BSGP: Bulk-synchronous GPU programming. *ACM Transactions on Graphics* 27, 3 (2008), 1–12.
[122] Eduardo Camilo Inacio and Mario A. R. Dantas. 2014. A survey into performance and energy efficiency in HPC, cloud and big data environments. *International Journal of Networking and Virtual Organisations* 14, 4 (2014), 299–318.
[123] Intel's Coaarray Fortran. 2021. Tutorial: Using Coarray Fortran. Retrieved from https://software.intel.com/content/www/us/en/develop/documentation/fortran-compiler-coarray-tutorial/top.html. (2021). Accessed: 2021-06-15.
[124] Suresh Jagannathan and Jim Philbin. 1992. A foundation for an efficient multi-threaded scheme system. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. 345–357.
[125] JoCaml Programming Language, Version 4.00.1.A. 2014. Retrieved from http://jocaml.inria.fr/. (2014). Accessed: 2021-06-15.
[126] Alan H. Karp. 1987. Programming for parallelism. *Computer* 20, 5 (May 1987), 43–57.
[127] Ryan D. Kelker. 2013. *Clojure for Domain-specific Languages*. Packt Publishing Ltd.
[128] C. Kessler and J. Keller. 1995. *Joule: Distributed Application Foundations*. Technical Report. Agorics Technical Report ADd003.4P, Retrieved from http://www.laputan.org/pub/papers/Joule.pdf. Accessed: 2021-06-15.
[129] C. Kessler and J. Keller. 2007. *Models for Parallel Computing: Review and Perspective*. Technical Report. Linköping University, Sweden. ISSN: 0177-0454, Retrieved from https://www.ida.liu.se/~chrke55/papers/modelsurvey.pdf.
[130] Christoph W. Kessler. 2004. Managing distributed shared arrays in a bulk-synchronous parallel programming environment. *Concurrency and Computation: Practice and Experience* 16, 2–3 (2004), 133–153.
[131] Christoph W. Keßler and Helmut Seidl. 1997. The Fork95 parallel programming language: Design, implementation, application. *International Journal of Parallel Programming* 25, 1 (1997), 17–50.
[132] Christoph W. Keßler and Helmut Seidl. 1999. ForkLight: A control-synchronous parallel programming language. In *Proceedings of the International Conference on High-Performance Computing and Networking*. Springer, 525–534.

[133] Barbara Kitchenham and Pearl Brereton. 2013. A systematic review of systematic review process research in software engineering. *Information and Software Technology* 55, 12 (2013), 2049–2075.
[134] Barbara A. Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Keele University and University of Durham.
[135] Svend E. Knudsen. 2011. Using independence to enable parallelism on multicore computers. *Software: Practice and Experience* 41, 4 (2011), 393–402.
[136] Krzysztof Kochut. 1989. Execution kernel for parallel logic programming. In *Proceedingsof the IEEE Energy and Information Technologies in the Southeast'*. IEEE, 491–495.
[137] Aaron H. Konstam. 1981. A method for controlling parallelism in programming languages. *ACM SIGPLAN Notices* 16, 9 (1981), 60–65.
[138] Sriram Krishnamoorthy, Umit Catalyurek, Jarek Nieplocha, Atanas Rountev, and P. Sadayappan. 2006. An extensible global address space framework with decoupled task and data abstractions. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 7–pp.
[139] Vivek Kumar, Julian Dolby, and Stephen M. Blackburn. 2016. Integrating asynchronous task parallelism and data-centric atomicity. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–10.
[140] Andreas Kunft, Lukas Stadler, Daniele Bonetta, Cosmin Basca, Jens Meiners, Sebastian Breß, Tilmann Rabl, Juan Fumero, and Volker Markl. 2018. Scootr: Scaling r dataframes on dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 288–300.
[141] Shigeru Kusakabe, Taku Nagai, Yoshihiro Yamashita, Rin-ichiro Taniguchi, and Makoto Amamiya. 1995. A dataflow language with object-based extension and its implementation on a commercially available parallel machine. In *Proceedings of the 9th International Conference on Supercomputing*. 308–317.
[142] V. P. Kutepov, V. N. Malanin, and N. A. Pankov. 2012. Flowgraph stream parallel programming: Language, process model, and computer implementation. *Journal of Computer and Systems Sciences International* 51, 1 (2012), 65–79.
[143] F. A. Lara Soares, C. Neri Nobre, and H. Cota de Freitas. 2019. Parallel programming in computing undergraduate courses: A systematic mapping of the literature. *IEEE Latin America Transactions* 17, 08 (2019), 1371–1381.
[144] James Larus. 1992. C**: A large-grain, object-oriented, data-parallel programming language. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. Springer, 326–341.
[145] Alexey Lastovetsky. 2002. Adaptive parallel computing on heterogeneous networks with mpC. *Parallel Computing* 28, 10 (2002), 1369–1407.
[146] Jinpil Lee, Mitsuhisa Sato, and Taisuke Boku. 2007. Design and implementation of OpenMPD: An OpenMP-like programming language for distributed memory systems. In *Proceedings of the International Workshop on OpenMP*. Springer, 143–147.
[147] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. 2011. An extension of XcalableMP PGAS lanaguage for multi-node GPU clusters. In *Proceedings of the European Conference on Parallel Processing*. Springer, 429–439.
[148] Sunwoo Lee, Byung Kwan Jung, Minsoo Ryu, and Seungwon Lee. 2009. Extending component-based approaches for multithreaded design of multiprocessor embedded software. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 267–274.
[149] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. 2010. OpenRCL: Low-power high-performance computing with reconfigurable devices. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*. IEEE, 458–463.
[150] Chao Liu and Miriam Leeser. 2016. Unified and lightweight tasks and conduits: A high level parallel programming framework. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[151] Fei Liu, Yi Yang, and Ling-ze Wang. 2016. A survey of the heterogeneous computing platform and related technologies. In *Proceedings of the International Conference on Informatics, Management Engineering and Industrial Application*. 6–12. DOI: https://doi.org/10.12783/dtetr/imeia2016/9229
[152] Ivan Llopard, Christian Fabre, and Albert Cohen. 2017. From a formalized parallel action language to its efficient code generation. *ACM Transactions on Embedded Computing Systems* 16, 2 (2017), 1–28.
[153] Hans-Wolfgang Loidl, Phil Trinder, Kevin Hammond, Abdallah Al Zain, and Clem Baker-Finch. 2008. Semi-explicit parallel programming in a purely functional style: GpH. *Process Algebra for Parallel and Distributed Processing: Algebraic Languages in Specification-Based Software Development* (2008), 47–76.
[154] Anton Lokhmotov, Benedict R. Gaster, Alan Mycroft, Neil Hickey, and David Stuttard. 2007. Revisiting SIMD programming. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. Springer, 32–46.
[155] Roberto Lublinerman, Jisheng Zhao, Zoran Budimlić, Swarat Chaudhuri, and Vivek Sarkar. 2011. Delegated isolation. *ACM SIGPLAN Notices* 46, 10 (2011), 885–902.

[156] Philipp Lucas, Nicolas Fritz, and Reinhard Wilhelm. 2006. The development of the data-parallel GPU programming language CGiS. In *Proceedings of the International Conference on Computational Science*. Springer, 200–203.
[157] Steven Lucco and Oliver Sharp. 1991. Parallel programming with coordination structures. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 197–208.
[158] Tim H. MacKenzie and Trevor I. Dix. 1998. Object-oriented ease-based parallel primitives in C++. In *Proceedings 1998 International Conference on Parallel and Distributed Systems*. IEEE, 623–630.
[159] Jari-Matti Makela, Martti Forsell, and Ville Leppanen. 2017. Towards a language framework for thick control flows. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 814–823.
[160] Yong Mao, Yunhong Gu, Jia Chen, and Robert L. Grossman. 2006. FastPara: A high-level declarative data-parallel programming framework on clusters. In *Proceedings of Parallel and Distributed Computing and Systems , November* (2006), 13–15.
[161] A. Marcoux, C. Maurel, and P. Salle. 1988. AL 1: A language for distributed applications. In *[1988] Proceedings. Workshop on the Future Trends of Distributed Computing Systems in the 1990s*. 270–276.
[162] Massimo Maresca and Pierpaolo Baglietto. 1993. A programming model for reconfigurable mesh based parallel computers. In *Proceedings of Workshop on Programming Models for Massively Parallel Computers*. IEEE, 124–133.
[163] Cristian Mateos, Alejandro Zunino, and Mat Hirsch. 2013. EasyFJP: Providing hybrid parallelism as a concern for divide and conquer Java applications. *Computer Science and Information Systems* 10, 3 (2013), 1129–1163.
[164] Håkan Mattsson and Christoph Kessler. 2004. Towards a bulk-synchronous distributed shared memory programming environment for grids. In *Proceedings of the International Workshop on Applied Parallel Computing*. Springer, 519–526.
[165] Patrick McCormick, Jeff Inman, James Ahrens, Jamaludin Mohd-Yusof, Greg Roth, and Sharen Cummins. 2007. Scout: A data-parallel programming language for graphics processors. *Parallel computing* 33, 10–11 (2007), 648–662.
[166] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. 2017. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming productivity, performance, and energy consumption. In *Proceedings of the of ARMS-CC*. 1–6.
[167] Olivier Michel. 1996. Design and implementation of 812: A declarative data-parallel language. *Computer Languages* 22, 2–3 (1996), 165–179.
[168] Russ Miller and Quentin F. Stout. 1989. An introduction to the portable parallel programming language Seymour. In *[1989] Proceedings of the 13th Annual International Computer Software and Applications Conference*. IEEE, 94–101.
[169] Andreas Mitschele-Thiel. 1993. The DSPL programming environment. In *Proceedings of Workshop on Programming Models for Massively Parallel Computers*. IEEE, 35–42.
[170] Sparsh Mittal and Jeffrey S. Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys* 47, 4, Article 69 (July 2015), 35 pages. DOI : https://doi.org/10.1145/2788396
[171] Paulo Moura, Paul Crocker, and Paulo Nunes. 2008. High-level multi-threading programming in logtalk. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*. Springer, 265–281.
[172] National Instruments. 2005. *LabVIEW Fundamentals*. Technical Report. LabVIEW Manual, 374029A-01, Retrieved form https://www.ni.com/pdf/manuals/374029a.pdf. Accessed: 2021-06-15.
[173] Nenad Nedeljkovlc and Michael J. Quinn. 1993. Data-parallel programming on a network of heterogeneous workstations. *Concurrency: Practice and Experience* 5, 4 (1993), 257–268.
[174] Markus Nestmann. 2017. Building a consistent taxonomy for parallel programming models. In *Proceedings of the GI-Jahrestagung*.
[175] Bob Newman and Martin Payne. 1994. Integration of object oriented and concurrent programming. In *Proceedings of 20th Euromicro Conference. System Architecture and Integration*. IEEE, 258–264.
[176] T.-A. Nguyen and Pierre Kuonen. 2003. ParoC++: A requirement-driven parallel object-oriented programming language. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 9–pp.
[177] Susumu Nishimura and Atsushi Ohori. 1999. Parallel functional programming on recursively defined data via data-parallel recursion. *Journal of Functional Programming* 9, 4 (1999), 427–462.
[178] Lukito Edi Nugroho and A. S. M. Sajeev. 1999. Java4P: Java with high-level concurrency constructs. In *Proceedings Fourth International Symposium on Parallel Architectures, Algorithms, and Networks*. IEEE, 328–333.
[179] Eddy A. M. Odijk. 1990. POOMA, POOL and parallel symbolic computing: An assessment. In *Proceedings of the CONPAR 90–VAPP IV*. Springer, 26–38.
[180] Kazuhiko Ohno, Shigehiro Yamamoto, Takanori Okano, and Hiroshi Nakashima. 2000. Orgel: An parallel programming language with declarative communication streams. In *Proceedings of the International Symposium on High Performance Computing*. Springer, 344–354.
[181] Kenneth O'brien, Ilia Pietri, Ravi Reddy, Alexey Lastovetsky, and Rizos Sakellariou. 2017. A survey of power and energy predictive models in HPC systems and applications. *ACM Computing Surveys (CSUR)* 50, 3 (2017), 1–38.
[182] Edwin M. Paalvast, Henk J. Sips, and Leo C. Breebaart. 1991. Booster: A high-level language for portable parallel algorithms. *Applied Numerical Mathematics* 8, 2 (1991), 177–192.

[183] Patrick Viry. 2017. Ateji PX for Java: Parallel Programming Made Simple. Retrieved form https://www.slideshare.net/PatrickViry/ateji-px-for-java. (2017). Accessed: 2021-06-15.
[184] Hervé Paulino and Eduardo Marques. 2015. Heterogeneous programming with single operation multiple data. *Journal of Computer and System Sciences* 81, 1 (2015), 16–37.
[185] L. Moniz Pereira, L. F. Monteiro, Jose C. Cunha, and Joaquim N. Aparicio. 1988. Concurrency and communication in Delta Prolog. In *Proceedings of the 1988 International Specialist Seminar on the Design and Application of Parallel Digital Processors*. IET, 94–104.
[186] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. 2008. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08)*. British Computer Society, Swinton, UK, 68–77. Retrieved form http://dl.acm.org/citation.cfm?id=2227115.2227123.
[187] Michael Philippsen, Thomas M. Warschko, Walter F. Tichy, and Christian G. Herter. 1993. Project triton: Towards improved programmability of parallel machines. In *[1993] Proceedings of the 26th Hawaii International Conference on System Sciences*, Vol. 1. IEEE, 192–201.
[188] Eduardo Gurgel Pinho and Francisco Heron de Carvalho Junior. 2014. An object-oriented parallel programming language for distributed-memory parallel computing platforms. *Science of Computer Programming* 80 (2014), 65–90.
[189] Katalin Popovici, Xavier Guerin, Lisane Brisolara, and Ahmed Jerraya. 2007. Mixed hardware software multilevel modeling and simulation for multithreaded heterogeneous MPSoC. In *Proceedings of the 2007 International Symposium on VLSI Design, Automation and Test*. IEEE, 1–4.
[190] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.
[191] Frédéric Raimbault and Dominique Lavenier. 1993. Relacs for systolic programming. In *Proceedings of International Conference on Application Specific Array Processors*. IEEE, 132–135.
[192] Rafael Ramirez. 1998. Time, communication and synchronisation in an agent-based programming language. In *Proceedings. 5th International Workshop on Temporal Representation and Reasoning (Cat. No. 98EX157)*. IEEE, 169–176.
[193] Pushpa Rao and Clifford Walinsky. 1993. An equational language for data-parallelism. *ACM SIGPLAN Notices* 28, 7 (1993), 112–118.
[194] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE, 655–670.
[195] Regents of the University of California. 2013. Bloom Programming Language, Release 0.9.7. Retrieved from http://bloom-lang.net/. (2013). Accessed: 2021-06-15.
[196] R. J. Richards, Balkrishna Ramkumar, and Sukumar G. Rathnam. 1997. ELMO: Extending (sequential) languages with migratable objects-compiler support. In *Proceedings 4th International Conference on High-Performance Computing*. IEEE, 180–185.
[197] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. 2019. Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 1534–1543.
[198] Martin C. Rinard and Monica S. Lam. 1998. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems* 20, 3 (1998), 483–545.
[199] Joseph A. Roback and Gregory R. Andrews. 2010. Gossamer: A lightweight approach to using multicore machines. In *Proceedings of the 2010 39th International Conference on Parallel Processing*. IEEE, 30–39.
[200] Mark Rodgers, Amanda Sowden, Mark Petticrew, Lisa Arai, Helen Roberts, Nicky Britten, and Jennie Popay. 2009. Testing methodological guidance on the conduct of narrative synthesis in systematic reviews: Effectiveness of interventions to promote smoke alarm ownership and function. *Evaluation* 15, 1 (2009), 49–73.
[201] A. Wendell, O. Rodrigues, Frederic Guyomarc'h, and Jean-Luc Dekeyser. 2012. An MDE approach for automatic code generation from UML/MARTE to OpenCL. *Computing in Science and Engineering* 15, 1 (2012), 46–55.
[202] RUST, Version 1.52.1. 2021. A Language Empowering Everyone to Build Reliable and Efficient Software. Retrieved from https://www.rust-lang.org/. (2021). Accessed: 2021-06-15.
[203] Shigeyuki Sato and Hideya Iwasaki. 2009. A skeletal parallel framework with fusion optimizer for GPGPU programming. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 79–94.
[204] Scala, Version 2.12.14. 2017. The Scala Programming Language. Retrieved from https://www.scala-lang.org/. (2017). Accessed: 2021-06-15.
[205] M. Schollmeyer. 1991. Linda and parallel computing-running efficiently on parallel time. *IEEE Potentials* 10, 3 (1991), 43–45.

[206] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. 2014. High level programming framework for FPGAs in the data center. In *Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–4.

[207] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. 2010. FPMR: MapReduce framework on FPGA. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 93–102.

[208] Pritam Prakash Shete, P. P. K. Venkat, Dinesh M. Sarode, Mohini Laghate, S. K. Bose, and R. S. Mundada. 2012. Object oriented framework for CUDA based image processing. In *Proceedings of the 2012 International Conference on Communication, Information and Computing Technology (ICCICT)*. IEEE, 1–6.

[209] Lawrence Snyder. 2007. The design and development of ZPL. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*. 8–1.

[210] Jon A. Solworth. 1992. Epochs. *ACM Transactions on Programming Languages and Systems* 14, 1 (1992), 28–53.

[211] Giandomenico Spezzano and Domenico Talia. 1997. A high-level cellular programming model for massively parallel processing. In *Proceedings of the 2nd International Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE, 55–63.

[212] Nenad Stankovic and Kang Zhang. 2002. A distributed parallel programming framework. *IEEE Transactions on Software Engineering* 28, 5 (2002), 478–493.

[213] Ketil Stølen. 1991. A method for the development of totally correct shared-state parallel programs. In *Proceedings of the International Conference on Concurrency Theory*. Springer, 510–525.

[214] R. F. Stone and H. S. M. Zedan. 1989. Designing time critical systems with TACT. In *[1989] Proceedingsof the EUROMICRO Workshop on Real Time*. IEEE, 74–82.

[215] Andrew Stromme, Ryan Carlson, and Tia Newhall. 2012. Chestnut: A Gpu programming language for non-experts. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. 156–167.

[216] Shelly S. Stubbs and Doris L. Carver. 1995. IPCC++: A C++ extension for interprocess communication with objects. In *Proceedings of the COMPSAC*. IEEE, 205–210.

[217] Yan Su, Feng Shi, Shahnawaz Talpur, Jin Wei, and Hai Tan. 2014. Exploiting controlled-grained parallelism in message-driven stream programs. *The Journal of Supercomputing* 70, 1 (2014), 488–509.

[218] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[219] Yuanhao Sun, Tianyou Li, Qi Zhang, Jia Yang, and Shih-wei Liao. 2007. Parallel xml transformations on multi-core processors. In *Proceedings of the IEEE International Conference on e-Business Engineering*. IEEE, 701–708.

[220] B. Suresh and R. Nadarajan. 2003. Object oriented parallel programming model on a network of workstations. In *Proceedings of the International Conference on Computational Science*. Springer, 1000–1010.

[221] A. R. Surve, A. R. Khomane, and S. Cheke. 2013. Energy awareness in HPC: A survey. *International Journal of Computer Science and Mobile Computing* 2, 3 (2013), 46–51.

[222] Lukasz G. Szafaryn, Todd Gamblin, Bronis R. De Supinski, and Kevin Skadron. 2013. Trellis: Portability across architectures with a high-level framework. *Journal of Parallel and Distributed Computing* 73, 10 (2013), 1400–1413.

[223] S. Tucker Taft. 2019. ParaSail: A pointer-free pervasively-parallel language for irregular computations. *The Art Science and Engineering of Programming* 3, 3 (2019), 7. DOI: https://doi.org/10.22152/programming-journal.org/2019/3/7

[224] S. Tucker Taft, Brad Moore, Luís Miguel Pinho, and Stephen Michell. 2014. Safe parallel programming in Ada with language extensions. In *Proceedings of the HILT*. 87–96.

[225] David Tarditi, Sidd Puri, and Jose Oglesby. 2006. Accelerator: Using data parallelism to program GPUs for general-purpose uses. *ACM SIGPLAN Notices* 41, 11 (2006), 325–335.

[226] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. 2017. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications* 31, 1 (2017), 66–82.

[227] The Julia Programming Language. 2021. Julia in a Netshell, Version 1.6.1. Retrieved from https://julialang.org/. (2021). Accessed: 2021-06-15.

[228] The Open MPI Project. 2021. Open MPI: Open Source High Performance Computing. Retrieved from https://www.open-mpi.org/. (2021). Accessed: 2021-06-15.

[229] George K. Thiruvathukal, Phillip M. Dickens, and Shahzad Bhatti. 2000. Java on networks of workstations (JavaNOW): A parallel computing framework inspired by Linda and the Message Passing Interface (MPI). *Concurrency: Practice and Experience* 12, 11 (2000), 1093–1116.

[230] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74, 4 (2018), 1422–1434.

[231] John Thornley. 1995. Declarative Ada: Parallel dataflow programming in a familiar context. In *Proceedings of the 1995 ACM 23rd Annual Conference on Computer Science*. 73–80.
[232] Bo-Ming Tong and Ho-Fung Leung. 1998. Data-parallel concurrent constraint programming. *The Journal of Logic Programming* 35, 2 (1998), 103–150.
[233] TOP500.org. TOP500 – The List. Retrieved from https://www.top500.org/lists/top500/2020/06/. (????). Accessed: 2020-07-03.
[234] Anand Tripathi, Vinit Padhye, and Tara Sasank Sunkara. 2014. Beehive: A framework for graph data analytics on cloud computing platforms. In *Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops*. IEEE, 331–338.
[235] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen Green, and Guy L. Steele. 2014. Augur: Data-parallel probabilistic modeling. In *Proceedings of the Advances in Neural Information Processing Systems*. 2600–2608.
[236] Miwako Tsuji, Mitsuhisa Sato, Maxime Hugues, and Serge Petiton. Multiple-SPMD Programming Environment Based on PGAS and Workflow Toward Post-petascale Computing. In *Proceedings of the 42 International Conference on Parallel Processing*. DOI: https://doi.org/10.1109/ICPP.2013.58
[237] David Ungar and Sam S. Adams. 2010. Harnessing emergence for manycore programming: Early experience integrating ensembles, adverbs, and object-based inheritance. In *Proceedings of the SPLASH*. 19–26.
[238] Theo Ungerer and Eberhard Zehendner. 1991. A multi-level parallelism architecture. *ACM SIGARCH Computer Architecture News* 19, 4 (1991), 86–93.
[239] Gautam Upadhyaya, Vijay S. Pai, and Samuel P. Midkiff. 2007. Expressing and exploiting concurrency in networked applications with aspen. In *Proceedings of the PPoPP*. 13–23.
[240] Mark Utting, Min-Hsien Weng, and John G. Cleary. 2013. The JStar language philosophy. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*. 31–41.
[241] Kees van Reeuwijk, Arjan J. C. van Gemund, and Henk J. Sips. 1997. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience* 9, 11 (1997), 1193–1205.
[242] Stéphane Vialle, Thierry Cornu, and Yannick Lallement. 1996. ParCeL-1: A parallel programming language based on autonomous and synchronous actors. *ACM SIGPLAN Notices* 31, 8 (1996), 43–51.
[243] Vita Nuova. 1995. Limbo Programming Language. Retrieved from http://www.vitanuova.com/inferno/licence.html. (1995). Accessed: 2021-06-15.
[244] Jürgen Vollmer and Ralf Hoffart. 1992. Modula-Pa language for parallel programming definition and implementation on a transputer network. In *Proceedings of the 1992 International Conference on Computer Languages*. IEEE, 54–64.
[245] Reinhard Von Hanxleden. 2009. SyncCharts in C: A proposal for light-weight, deterministic concurrency. In *Proceedings of the 7th ACM International Conference on Embedded Software*. 225–234.
[246] Stewart von Itzstein and David Kearney. 2002. Applications of Join Java. In *Proceedings of the 7th Asia-Pacific Computer Systems Architectures Conference*. 37–46.
[247] Benjamin J. L. Wang and Uwe R. Zimmer. 2017. Pure concurrent programming. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 824–831.
[248] Chen Wang, Ce Yu, Jizhou Sun, and Xiangfei Meng. 2015. DPX10: An efficient X10 framework for dynamic programming applications. In *Proceedings of the 2015 44th International Conference on Parallel Processing*. IEEE, 869–878.
[249] H. C. Wang, C. K. Yuen, and M. D. Feng. 1999. BaLinda c++: Run-time support for concurrent object-oriented language. In *Proceedings of the of I-SPAN'99*. IEEE, 36–41.
[250] P. Y. Wang, S. B. Seidman, M. D. Rice, and T. E. Gerasch. 1989. An object-method programming language for data parallel computation. In *Proceedings of the HICSS*, Vol. 2. IEEE, 745–750.
[251] Y. M. R. D. Wepathana, G. Anthonys, and L. S. K. Udugama. 2015. Compiler for a simplified programming language aiming on Multi Core Students' Experimental Processor. In *Proceedings of the ICIIS*. IEEE, 284–289.
[252] Emily A. West and Andrew S. Grimshaw. 1995. Braid: Integrating task and data parallelism. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 211–219.
[253] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the EASE*. ACM, Article 38, 38:1–38:10 pages.
[254] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer.
[255] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. 2013. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *Proceedings of the CCGRID*. IEEE, 95–102.
[256] Tianji Wu, Di Wu, Yu Wang, Xiaorui Zhang, Hong Luo, Ningyi Xu, and Huazhong Yang. 2011. Gemma in April: A matrix-like parallel programming architecture on OpenCL. In *Proceedings of the 2011 Design, Automation and Test in Europe*. IEEE, 1–6.

[257] Chengzhi Xu, Hong Zhu, Ian Bayley, David Lightfoot, Mark Green, and Peter Marshall. 2016. Caople: A programming language for microservices saas. In *Proceedings of the 2016 IEEE Symposium on Service-Oriented System Engineering*. IEEE, 34–43.

[258] Zhiwei Xu and Kai Hwang. 1989. Molecule: A language construct for layered development of parallel programs. *IEEE Transactions on Software Engineering* 15, 5 (1989), 587–599.

[259] Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. 2016. The forec synchronous deterministic parallel programming language for multicores. In *Proceedings of the MCSOC*. IEEE, 297–304.

[260] Mohamed Youssfi, Omar Bouattane, Mohamed O. Bensalah, ENSET Bd Hassan I. I. BP, and Mohammedia Morocco. 2010. On the object modelling of the Massively parallel architecture Computers. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. 71–78.

[261] Yuanyuan Zhang, Hidetoshi Iwashita, Kuninori Ishii, Masanori Kaneko, Tomotake Nakamura, and Kohichiro Hotta. 2010. Hybrid parallel programming on SMP clusters using XPFortran and OpenMP. In *Proceedings of the International Workshop on OpenMP*. Springer, 133–148.

[262] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[263] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. Versapipe: A versatile programming framework for pipelined computing on GPU. In *Proceedings of the MICRO*. IEEE, 587–599.

[264] Jin Zhou and Brian Demsky. 2010. Bamboo: A data-centric, object-oriented approach to many-core software. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 388–399.

Received September 2020; revised August 2021; accepted August 2021