Liquid IT : Toward a better compromise between development scalability and performance scalability not definitive

Etienne Brodu

December 23, 2015

Abstract

TODO translate from below when ready

Résumé

Internet étend nos moyens de communications, et réduit leur latence ce qui permet de développer l'économie à l'échelle planétaire. Il permet à chacun de mettre un service à disposition de milliards d'utilisateurs, en seulement quelques heures. La plupart des grands services actuels ont commencé comme de simples applications créées dans un garage par une poignée de personnes. C'est cette facilité à l'entrée qui a permis jusqu'à maintenant une telle croissance sur le web. Google, Facebook ou Twitter en sont quelques exemples. Au cours du développement d'une application, il est important de suivre cette croissance, au risque de se faire rattraper par la concurrence. Le développement est guidé par les besoins en terme de fonctionnalités, afin de vérifier rapidement si le service peut satisfaire l'audience. On parle d'approche modulaire des fonctionnalités. Des langages tel que Ruby ou Java se sont imposés comme les langages du web, justement parce qu'ils suivent cette approche qui permet d'intégrer facilement de nouvelles fonctionnalités.

Si une application répond correctement aux besoins, elle atteindra de manière virale un nombre important d'utilisateurs. Son audience peut prendre plusieurs ordres de grandeurs en quelques jours seulement, ou même en quelques heures suivant comment elle est relayée. Une application est dite *scalable* si elle peut absorber ces augmentations d'audience. Or il est difficile pour une application suivant l'approche modulaire d'être *scalable*.

Au moment où l'audience commence à devenir trop importante, il est nécessaire de modifier l'approche de développement de l'application. Le plus souvent cela implique de la réécrire complètement en utilisant des infrastructures *scalables* qui imposent des modèles de programmation et des API spécifiques, qui représentent une charge de travail conséquente et incertaine. De plus, l'équipe de développement dois concilier cette nouvelle approche de développement *scalable*, avec la demande en fonctionnalités. Aucun langage n'a clairement réussi le compromis entre ces deux objectifs.

Pour ces raisons, ce changement est un risque pour la pérennité de l'application. D'autant plus que le cadre économique accorde peu de marges d'erreurs, comme c'est le cas dans la plupart des start-up, mais également dans de plus grandes structures.

Cette thèse est source de propositions pour écarter ce risque. Elle repose sur les deux observations suivantes. D'une part, Javascript est un langage qui a gagné en popularité ces dernières années. Il est omniprésent sur les clients, et commence à s'imposer également sur les serveurs avec Node.js. Il a accumulé une communauté de

développeurs importante, et constitue l'environnement d'exécution le plus largement déployé. De ce fait, il se place maintenant de plus en plus comme le langage principal du web, détrônant Ruby ou Java. D'autre part, l'exécution de Javascript s'assimile à un pipeline. La boucle événementielle de Javascript exécute une suite de fonctions dont l'exécution est indépendante, mais qui s'exécutent sur un seul cœur pour profiter d'une mémoire globale.

L'objectif de cette thèse est de maintenir une double représentation d'un code Javascript grâce à une équivalence entre l'approche modulaire, et l'approche pipeline d'un même programme. La première répondant aux besoins en fonctionnalités, et favorisant les bonnes pratiques de développement pour une meilleure maintenabilité. La seconde proposant une exécution plus efficace que la première en permettant de rendre certaines parties du code relocalisables en cours d'exécution.

Nous étudions la possibilité pour cette équivalence de transformer un code d'une approche vers l'autre. Grâce à cette transition, l'équipe de développement peut continuellement itérer le développement de l'application en suivant les deux approches à la fois, sans être cloisonné dans une, et coupé de l'autre.

Nous construisons un compilateur permettant d'identifier les fonctions de Javascript et de les isoler dans ce que nous appelons des Fluxions, contraction entre fonctions et flux. Un conteneur qui peut exécuter une fonction à la réception d'un message, et envoyer des messages pour continuer le flux vers d'autres fluxions. Les fluxions sont indépendantes, elles peuvent être déplacées d'une machine à l'autre.

Nous montrons qu'il existe une correspondance entre le programme initial, purement fonctionnel, et le programme pivot fluxionnel afin de maintenir deux versions équivalentes du code source. En ajoutant à un programme écrit en Javascript son expression en Fluxions, l'équipe de développent peut le rendre *scalable* sans effort, tout en étant capable de répondre à la demande en fonctionnalités.

Ce travail s'est fait dans le cadre d'une thèse CIFRE dans la société Worldline. L'objectif pour Worldline est de se maintenir à la pointe dans le domaine du développement et de l'hébergement logiciel à travers une activité de recherche. L'objectif pour l'équipe Dice est de conduire une activité de recherche en partenariat avec un acteur industriel.

Contents

List of Figures

List of Tables

Chapter 1

Software Design, State Of The Art

Contents

"A designer is responsible for producing the greatest benefit for any given investment of time, talent, money, and other resources."

— K. Sullivan, W. Griswold, Y. Cai, B. Hallen [149]

The growth of the web and Software as a Service (SaaS) revealed the importance of previously unknown economic constraints. The same company carries both development and exploitation of an application at scale of unprecedented size. To allow a continuous growth and sustainability of an application, it needs to address two contradictory goals : maintainability and performance efficiency. These goals needs to be enforced by the platform supporting the application to build good development habits for the developers. A platform designates any solution that allows to build an application on top of it, including programming languages, compilers, interpreters, frameworks, runtime libraries and so on.

75% of your budget is dedicated to software maintenance.[1] The maintainability of an application relies on the modularity enforced by the platform used to build it. Especially, higher order programming is crucial to build and compose modules efficiently. It relies either on mutable states, or immutable states, but hardly on a combination of both.

However, neither mutable nor immutable states allow performance efficiency. Mutable states leads to synchronization overhead at coarser-grain level, while immutable states leads to communication overhead at finer-grain level. Performance efficiency relies on a combination of shared mutable state at a fine-grain level, and immutable message passing at a coarse-grain level. This combination breaks the modularity, hence the maintainability of an application. A company has no choice but to commit huge development efforts to get efficient performances.

illustration: virtuous circle between community and industry

Moreover, a balance between these two contradictory goals is required for a platform to enter a virtuous circle of adoption. The maintainability is required to be appealing to gather a community to support the ecosystem around the platform. This community is appealing for the industry as a hiring pool. The performance efficiency is required to be adopted by the industry to be economically viable. And the industrial relevance provides the reason for this ecosystem to exist and the community to gather.

This chapter presents a broad view of the state of the art in the compromises between maintainability and performance efficiency. It defines software maintainability, performance efficiency, and adoption in section 1.1 and all the underlying concepts, such as higher order programming and state mutability. It then analyzes different platforms according to their focus. platforms focusing on maintainability

[1] `http://www.castsoftware.com/glossary/software-maintainability`

are addressed in section 1.2, those focusing on performance efficiency in section 1.3 and those focusing on a compromise between the two in section 1.4.

1.1 Definitions

The continuous growth and sustainability offered by a platform relies on three criteria. This section defines these tree criteria, as well as all the underlying concepts.

- Maintainability
- Performance Efficiency
- Adoption

1.1.1 Maintainability

Software maintainability is defined as the degree to which an application can be understood, fixed, and extended. For an application to be maintainable, its frameworks need to enforce modularity directly in the design.

1.1.1.1 Modularity

The modularity of a software implementation is about encapsulating subproblems and composing them to allow greater design to emerge. It allows to limit the understanding required to contribute to a module [146], which helps developers to repair and enhance the application. Additionally, it reduces development time by allowing several developers to simultaneously implement different modules [168, 23].

The criteria to define modules to improve maintainability are low coupling and high cohesion [146]. Coupling defines the strength of the interdependence between modules. Cohesion defines how strongly the features inside a module are related. The composition of modules help decrease coupling, and encapsulation helps increase their cohesion. Encapsulation and composition improves maintainability.

- Encapsulation $\rightarrow$ High Cohesion
- Composition $\rightarrow$ Low Coupling

1.1.1.2 Encapsulation

illustration: spaghetti programming

Boundary Definition Modular Programming stands upon Structured Programming [43]. It draws clear interfaces around a piece of implementation so that the execution is enclosed inside. At a fine level, it helps avoid spaghetti code [46], and at a coarser level, it structures the implementation [47] into modules, or layers.

illustration: lasagna programming

Data Protection Modular programming encapsulates a specific design choice in each module, so that it is responsible for one and only one concern. It isolates its evolution from impacting the rest of the implementation [131, 153, 92]. Examples of such separation of concerns are the separation of the form and the content in HTML / CSS, or the OSI model for the network stack.

1.1.1.3 Composition

Higher-Order Programming Higher-order programming allows to manipulate functions like any other primary value : to store them in variables, or to pass them as arguments. It replaces the need for most modern object oriented programming design patterns [2] with Inversion of Control [96], the Hollywood Principle [152], and Monads [164]. Higher-order programming help loosen coupling, thus improve maintainability [77].

Closures In languages allowing mutable state, higher-order functions are implemented as closure, to preserve the lexical scope [151]. A closure is the association of a function and a reference to the lexical context from its creation. It allows this function to access variable from this context, even when invoked outside the scope of this context.

Lazy Evaluation Lazy evaluation is an evaluation strategy allowing to defer the execution of a function only when its result is needed. The lazy evaluation of lists is equivalent to a stream with a null-sized buffer [162]. Both are powerful tools for structuring modular programs [1]. They allow the execution to be organized as a concurrent pipeline, as the stages are executed independently for each element of the stream. But this concurrency requires the isolation of side-effects to avoid conflicts between stages executions.

[2] `http://stackoverflow.com/a/5797892/933670`

The criteria to analyze the maintainability of platforms are the following.

- Encapsulation → High Cohesion

  Boundary definition

  Data protection

- Composition → Low Coupling

  Higher-order programming, Lambda Expressions

  Lazy evaluation, Stream composition

1.1.2 Performance Efficiency

The performance efficiency of a software project is the relation between the usage made of available resources and the delivered performance. For an application to perform efficiently, the frameworks used need to enforce scalability directly in its design.

Scalability relies on the commutativity of operations execution [33]. An operation is a sequence of statements. Operations are commutative if the order of their executions is irrelevant for the correctness of their results. Commutativity assures the independence of operations.

1.1.2.1 Independence

The independence, and commutativity of an operations depends on its accesses to shared state. If the operations doesn't rely on any shared state, it is independent. The independence of operations allows to execute them in parallel, hence to increase performance proportionally to occupied resources [8, 70]. But if they rely on shared state, they need to coordinate the causal ordering of their executions to avoid conflicting accesses. This causality between the operations can be defined in two ways.

illustration: Synchronization vs Message-passing

Synchronization Operations are scheduled sequentially to have an exclusive access to a centralized state, or

Message-passing Operations communicate their local modifications of the state to other operations, in a decentralized fashion.

If the operations access the state too frequently, the communication overhead exceed the performance gains of parallelism. And if operations access the state too rarely, the centralization of the state limits the parallelism. Additionally, because of the latency of the web, synchronization between remote locations obliterates performance.

Operations tend to share state closely at a fine-grain level and less at a coarser-grain level. Therefore, performance efficiency requires the combination of fine-level state sharing to avoid communication overhead, and coarse-level independence to allow parallelization [72, 71, 123, 69].

The criteria to analyze the performance of platforms are the following.

- Fine-level state sharing

  State mutability → Synchronization

- Coarse-level independence

  State immutability → Message-passing

1.1.2.2 Granularity

The threshold determining frequent or rare access to the state determines the granularity level between synchronization and parallelization of tasks. This granularity needs to be free from the decomposition into modules. If the too are related, they interfere with each other, and impact either maintainability, or performance.

1.1.2.3 Atomicity

Atomicity of the memory accesses is assured by either the causality between operations or exclusivity of their memory accesses. It is the assurance that the operation happens in a single bulk ; the beginning and end are indistinguishable for an external observer. It assures the developer of the invariance of the memory during the operation.

1.1.3 Adoption

An application is sustainable only if the frameworks used to build it generates reinforcing interactions between a community of passionate and the industry. A framework needs to present a balance between maintainability and performance efficiency to be adopted by both the community and the industry. The maintainability is

required for a framework to be appealing to gather a community to support the ecosystem around it. And the performance efficiency is required to be economically viable and needed by the industry, and to provide the reason for this ecosystem to exist. Additionally, the web acts as a tremendous catalyst fueling these interactions.

The criteria to analyze the adoption of platforms are the following.

- Community Support
- Industrial Need

Adoption requires a balance between performance efficiency and maintainability. This incentive to balance between maintainability and performance efficiency is illustrated in figure 1.1. This figure is used throughout this chapter to graphically represent all the platforms analyzed.

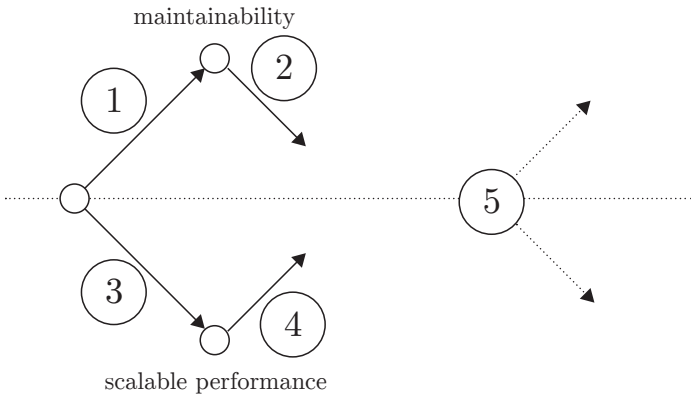


Figure 1.1 – Balance between Performance Efficiency and Maintainability

1.2 Maintainability Focused Platforms

"It is becoming increasingly important to the data-processing industry to be able to produce more programming systems and produce them with fewer errors, at a faster rate, and in a way that modifications can be accomplished easily and quickly."
— W. P. Stevens, G. J. Myers, L. L. Constantine [146].

In order to improve and maintain a software system, it is important to holds in mind a mental representation of its implementation. As the system grows in size, the mental representation becomes more and more difficult to grasp. Therefore, it is crucial to decompose the system into smaller subsystem easier to grasp individually.

Section 1.2.1 presents the modular programming paradigms, and their programming models, oriented toward maintainability. Section 1.2.2 presents the adoption of the implementations of modular programming languages. Section 1.2.3 presents the consequences of the modularity on performance. Finally, section 1.2.4 summarizes the three previous sections in a table.

1.2.1 Modular Programming

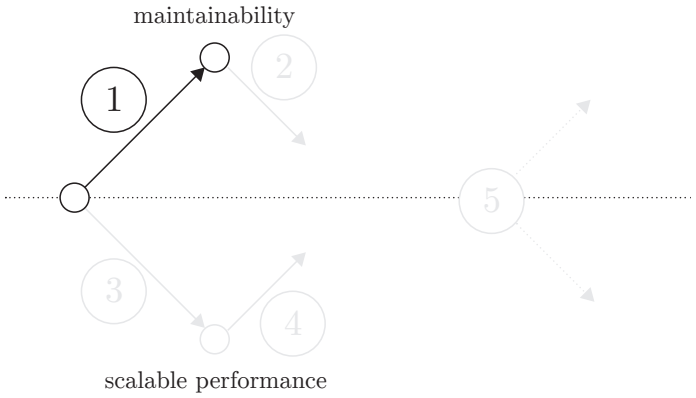


Figure 1.2 – Focus on Maintainability

The next paragraphs presents the different programming model regarding their support to modular programming and maintainability.

Imperative Programming Imperative programming is the very first programming paradigm, as it evolves directly from the hardware architectures. It allows to express the suite of operation to carry sequentially on the computing processor. Most imperative languages provide encapsulation with modules but not higher-order programming, nor lazy evaluation.

illustration: multiple cells communicating

Object Oriented Programming The very first OOP language was Smalltalk [62]. It defined the core concepts of OOP as message passing, encapsulation, and late-binding[3]. Nowadays, the major emblematic figures of OOP in the software industry are C++ and Java [64, 148]. They provide encapsulation with Classes, and higher-order programming with objects, but not lazy evaluation.

Functional Programming The definition of pure Functional Programming resides in manipulating only mathematical expressions - functions - and forbidding state mutability, replaced by message passing. The absence of state mutability makes a function side-effect free, hence their execution can be scheduled in parallel. But it implies heavy message passing, which negatively impact performances. The most important pure Functional Programming languages are Scheme [139], Miranda [159], Haskell [90] and Standard ML [119]. They provide encapsulation, higher-order programming and lazy evaluation.

Multi-Paradigm The functional programming concepts are also implemented in other languages along with mutable states and object-oriented concepts. Major recent programming languages, including Java 8 and C++ 11, now commonly present **higher-order functions** and **lazy evaluation**. *In fine*, it helps developers to write applications that are more maintainable, and favorable to evolution [91, 160]. These recent multi-paradigms languages like Javascript, Python, Ruby and Go combine the different paradigms to help developer building applications faster.

Table 1.15 presents a summary of the analysis of the programming models presented in the previous paragraphs.

[3] `http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en`

Model	Implementations	Composition	Encapsulation	→	Maintainability
Imperative Programming	C	3	4		3
Object-Oriented Programming	C++	5	5		5
Functional Programming	Scheme	5	5		5
Multi Paradigm	Javacript, Python, Ruby, Go	5	5		5

Table 1.1 – Maintainability of Modular Programming Platforms

1.2.2 Adoption

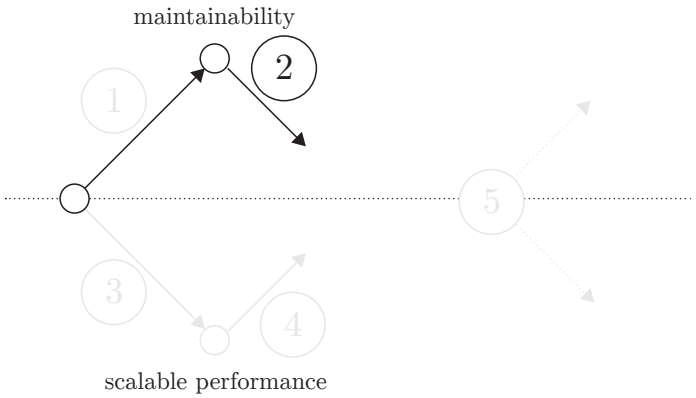


Figure 1.3 – Steer back toward Performence Efficiency

The next paragraphs presents the adoption of the implementation of the previously presented programming model.

1.2.2.1 Community

Available Resources According to the TIOBE Programming Community index, Javascript ranks 8th, as of October 2015, and was the most rising language in 2014. This index measure the popularity of a programming language with the number of results on many search engines. Alternatively, Javascript ranks 7th on the PYPL, as of October 2015. The PYPL index is based on Google trends to measure the number of requests on a programming language.

From these indexes, the major programming languages are Java, C++, C, C# and Python. These languages are still widely used by their communities and in the industry.

*

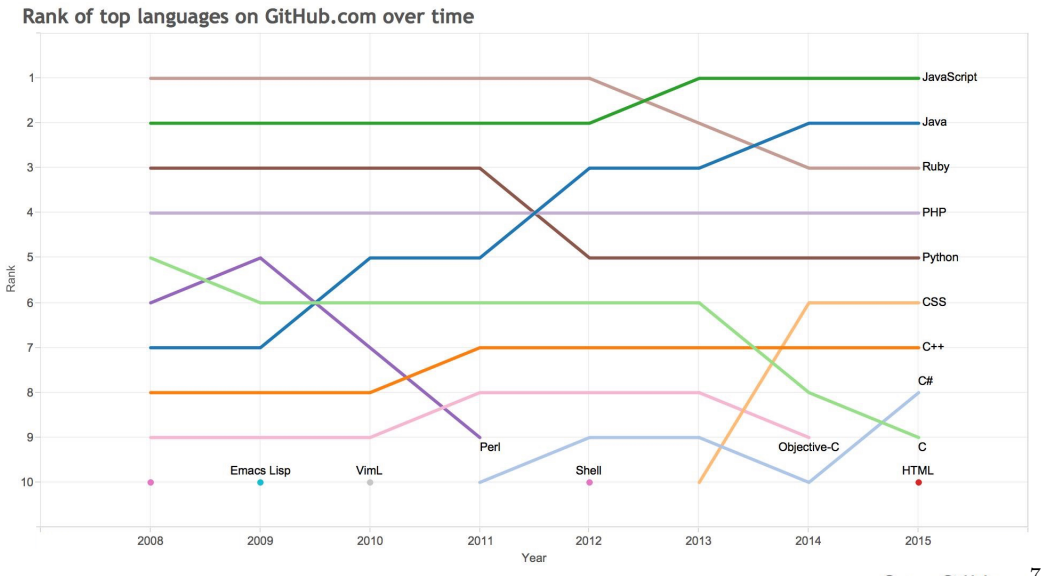
TODO graphical ranking of TIOBE and PYPL

Developers Collaboration Platforms Online collaboration tools give an indicator of the number of developers and projects using certain languages. Javascript is the most used language on *Github*[4] and the most cited language on *StackOverflow*[5] It represents more than 320000 repositories on *Github*. The second language is Java with more than 220000 repositories. It is cited in more than 960000 questions on *StackOverflow* while the second is Java with around 940000 questions. And according to a survey by *StackOverflow*, it is currently the language the most popular[6]. Moreover, the Javascript package manager, *npm*, has the most important and impressive package repository growth.

*

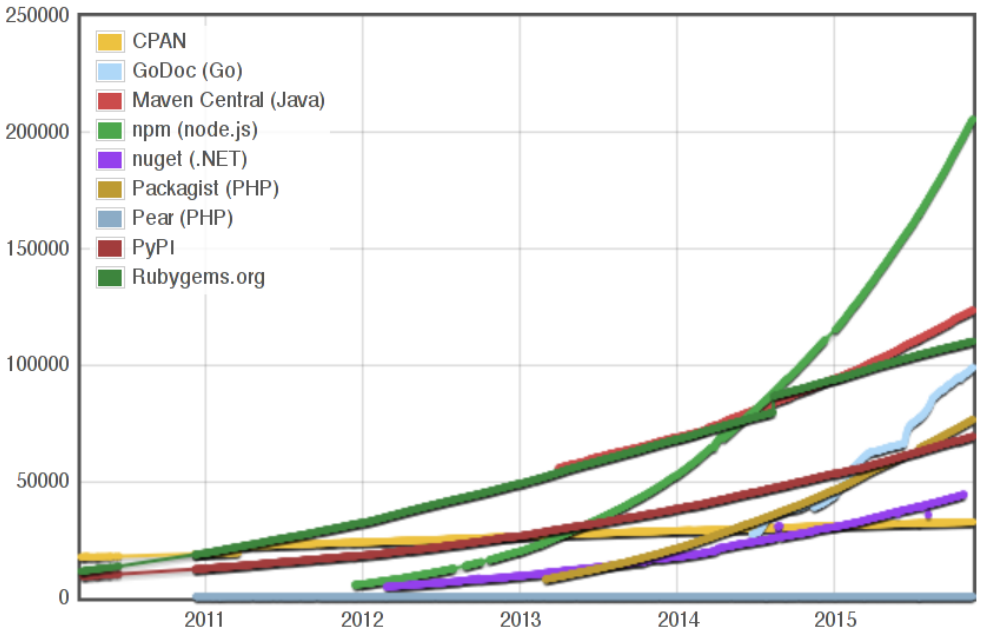
TODO include so survey graph

[4] themostimportantcollaborativedevelopmentplatformgatheringabout9millionsusers.
[5] themostimportantQ&Aplatformfordevelopers.
[6] http://stackoverflow.com/research/developer-survey-2015



Source: GitHub.com [7]

TODO redo this graph, it is ugly.



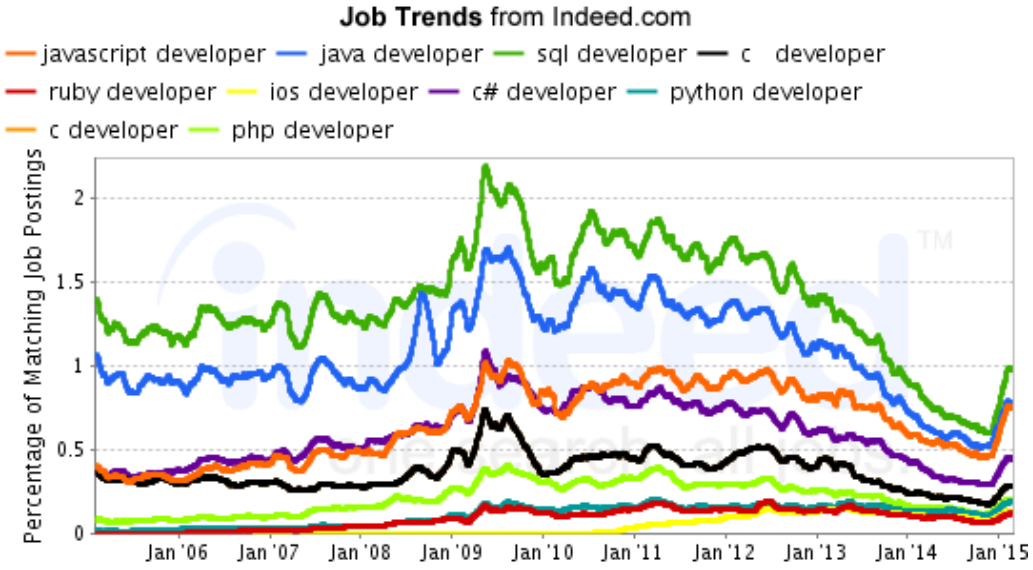
TODO graphical ranking of the tags in StackOverflow

TODO redo this graph, it is ugly.

[7] https://github.com/blog/2047-language-trends-on-github

1.2.2.2 Industry

The actors of the software industry tends to hide their activities trying to keep an edge on the competition. The previous metrics represent the visible activity but are barely representative of the software industry. The trends on job opportunities give some additional hints on the situation. Javascript is the third most wanted skill, according to *Indeed*[8], right after SQL and Java.[9] Moreover, according to *breaz.io*[10], Javascript developers get more opportunities than any other developers. Javascript is increasingly adopted in the software industry.



*

TODO redo this graph, it is ugly.

Table 1.2 presents a summary of the analysis of the programming models presented in the previous paragraphs.

[8] http://www.indeed.com

[9] http://www.indeed.com/jobtrends?q=Javascript%2C+SQL%2C+Java%2C+C%2B%2B%2C+C%2FC%2B%2B%2C+C%23%2C+Python%2C+PHP%2C+Ruby&l=

[10] https://breaz.io/

Model	Implementations	Community support	Industrial need	→	Adoption
Imperative Programming	C	3	4		3
Object-Oriented Programming	C++	4	4		4
Functional Programming	Scheme	0	0		0
Multi Paradigm	Javacript, Python, Ruby, Go	5	4		4

Table 1.2 – Adoption of Modular Programming Platforms

1.2.3 Performance Limitations

Eventually, the presented languages are hitting a wall on their way to performance.

All the languages presented in this section provide either mutable state or immutable state on which to rely to assure encapsulation and composition. Functional programming rely on immutable state, comparable to message-passing, which might impacts performance at fine-grain level when used inefficiently, because of heavy memory usage. On the other hand, the synchronization required by mutable state is often hard to develop with [3], or avoid parallelism [130, 104].

The only solution to provide performance efficiency is to combine mutable state at a fine-grain level, with synchronization, and immutable state at a coarse-grain level, with message-passing.

The table 1.3 presents the performance limitations of the languages presented in this section. It only presents programming languages. All platforms improving on these languages to provide performances are addressed in the next section.

Model	Implementations	Fine-grain level synchronization	Coarse-grain level message passing	→	Performance Efficiency
Imperative Programming	C	0	0		0
Object-Oriented Programming	C++	0	0		0
Functional Programming	Scheme	0	0		0
Multi Paradigm	Javacript, Python, Ruby, Go	0	0		0

Table 1.3 – Performance Efficiency of Modular Programming Platforms

1.2.4 Summary

Table 1.4 summarizes the characteristics of the solutions presented in this section.

Model	Implementations	Maintainability	Adoption	Performance Efficiency
Imperative Programming	C	3	3	0
Object-Oriented Programming	C++	5	4	0
Functional Programming	Scheme	5	0	0
Multi Paradigm	Javacript, Python, Ruby, Go	5	4	0

Table 1.4 – Summary of Modular Programming Platforms

1.3 Performance Efficiency Focused Platforms

With the limitations on performance presented in the previous section, both academia and industry communities proposed alternative solutions with performance in mind. Section 1.3.1 presents the concurrent and parallel programming paradigms, and their programming models. Section 1.3.2 presents the adoption steered by the performance efficiency of parallel programming. Section 1.3.3 presents the consequences of parallelism on maintainability. Finally, section 1.3.4 summarizes the three previous sections in a table.

1.3.1 Concurrency

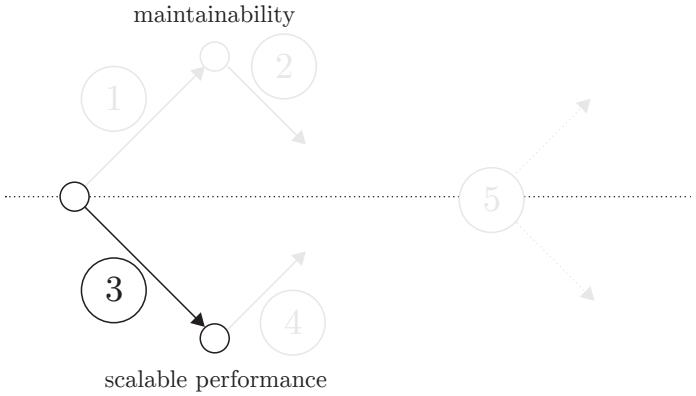


Figure 1.4 – Focus on Performance Efficiency

Web servers need to be able to process huge amount of concurrent operations in a scalable fashion. Concurrency is the ability to make progress on several operations roughly simultaneously. It implies to draw boundaries in the memory to define independent regions, and causality in the execution to define independent tasks. When both boundaries and causality are clearly defined, the tasks can be scheduled in parallel to make progress strictly simultaneously.

The definition of independent tasks allows the fine level synchronization within a task, and coarse level message passing between the tasks required for performence efficiency. The synchronization of execution at a fine level assures the invariance on the shared state, and avoid communication overhead. The message-passing at

a coarser level assures the parallelism. The two are indispensable for performance efficiency.

1.3.1.1 Concurrent Programming

illustration: feu rouge et rond point

Concurrent programming provides the mechanisms to define the causal ordering of execution and assure the invariance of the global memory. There are two scheduling strategies to execute tasks sequentially on a single processing unit, cooperative scheduling and preemptive scheduling.

Cooperative Scheduling allows a concurrent execution to run until it yields back to the scheduler. Each concurrent execution has an atomic, exclusive access on the memory.

Preemptive Scheduling allows a limited time of execution for each concurrent execution, before preempting it. It assures fairness between the tasks, such as in a multi-tasking operating system. But the unexpected preemption breaks atomicity, the developer needs to lock the shared state to assure atomicity and exclusivity.

The next paragraphs presents the event-driven programing model, based on cooperative scheduling, and the multi-threading programming model, based on preemptive scheduling. Additionally, they present lock-free data-structures, which is independent from the scheduling strategy, as they rely on atomic memory operations.

Event-Driven Programming Event-driven execution model queues explicitly defined concurrent tasks needing access to shared resources. The concurrent tasks are schedule sequentially to assure exclusivity, and cooperatively to assure atomicity. It is very efficient for highly concurrent applications, as it avoids contention due to waiting for shared resources like disks, or network. Several execution model rely on this execution model, like TAME [104], Node.js[11] and Vert.X[12]. As well as some web servers like Flash [130], Ninja [65] thttpd[13] and Nginx[14].

But the event-driven model is limited in performance. The concurrent tasks share the same memory, and cannot be scheduled in parallel. The next paragraph present work intending to improve performance.

[11] `https://nodejs.org/en/`
[12] `http://vertx.io/`
[13] `http://acme.com/software/thttpd/`
[14] `https://www.nginx.com/`

Lock-Free Data-Structures The wait-free and lock-free data-structures reduce the exclusive execution to a minimum of instructions [106, 82, 80, 81, 10]. They arrange these instructions into small atomic operations to avoid the need to lock. They are based on atomic read and write operations provided by transactional memories [75]. They provide concurrent implementation of basic data-structures such as linked list [161, 156], queue [150, 167], tree [136] or stack [79].

However these atomic operations are uncommon hence difficult to develop with. The next paragraph present the multi-threading improving parallelism with explicit synchronization from the developer.

Multi-Threading Programming Threads are light processes sharing the same memory execution context within an isolated process [47], and scheduled in parallel with fork/join operations [137, 58, 109]. They execute statements sequentially waiting for completion, and are scheduled preemptively to avoid blocking the global progression. The preemption breaks the atomicity of the execution, and the parallel execution breaks the exclusivity of memory accesses. To restore atomicity and exclusivity, hence assure the invariance, multi-threading programming models provide synchronization mechanisms, such as semaphores [44], guarded commands [45], guarded region [74] or monitors [87].

Developers tend to use the global memory extensively, and threads require to protect each and every shared memory cell. This heavy need for synchronization leads to bad performances, and is difficult to develop with [3].

Fibers Cooperative threads, or fibers proposed to join the advantage of threads sequentiality, with the advantage of cooperative scheduling [3, 19]. It avoids splitting the execution into atomic tasks nor use synchronization mechanisms to protect the memory. A fiber yields the execution to another fiber for a long-waiting operation to avoid blocking the execution, and recovers it at the same point when the operation finishes.

However, developers need to be aware of these yielding operation to preserve the atomicity[15].

Limitation of Concurrent Programming Concurrent programming provides synchronization of execution within a task, at the fine-grain level. Multi-threading imposes synchronization between threads as well. This global ordering is excessive ; it impacts performance, and is difficult to manage efficiently.

[15] `https://glyph.twistedmatrix.com/2014/02/unyielding.html`

The causal ordering between tasks proposed by the event-driven execution model is sufficient to assure correctness of execution [105, 138]. But because of the lack of memory isolation, the concurrent tasks are not scheduled in parallel.

Parallel programming is the only solution for performance efficiency, at the expense of development efforts to explicitely define the memory isolation of concurrent tasks and their communications by message pasing.

The table 1.5 presents a summary of the analysis of performance of the platforms presented in this section.

Model	Implementations	Fine-grain level synchronization	Coarse-grain level message passing	→	Performance Efficiency
Event-driven programming	Node.js, Vert.X	5	0		2
Lock-free Data Structures	list [161, 156], queue [150, 167], tree [136], stack [79]	5	0		2
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	4	0		1

Table 1.5 – Performance Efficiency of Concurrent Programming Platforms

1.3.1.2 Parallel Programming

Concurrent programming is based on the causal ordering of tasks. Concurrent tasks can be scheduled in parallel if their memory are isolated.

The Flynn's taxonomy [53] is commonly used to categorize parallel executions in function of the multiplicity of their flow of instruction and data. Parallel programming models belong to the category Multiple Instruction Multiple Data (MIMD), which is further divided into Single Program Multiple Data (SPMD) [13, 40, 41] and Multiple Program Multiple Data (MPMD) [28, 26]. SPMD defines a single program

replicated on many processing units [38, 95, 27]. While MPMD defines multiple processes in the implementation [66, 56, 55].

*

This section presents MPMD platforms allowing to define isolated tasks. It presents theoretical and programming models on asynchronous communication and isolated execution for parallel programming. It then presents with stream processing programming model. And finally, it concludes on the limitations of parallel programming regarding maintainability.

⚠ schema of SPMD and MPMD

Theoretical Models The causal ordering possible with asynchronous communication is sufficient to assure correctness in a distributed system [105, 138]. And the communication in reality are too slow compared to execution to be synchronous, and are subject to various faults and attacks [107]. The Actor model takes these physical limitations in account [85]. It allows to express the causal ordering of computation as a set of parallel actors communicating by asynchronous messages [83, 84, 34]. In reaction to a received message, an actor can create other actors, send messages, and choose how to respond to the next message.

Similarly, coroutines are autonomous programs which communicate with adjacent modules as if they were input and output subroutines [37]. It defines a pipeline to implement multi-pass algorithms. Similar works include the Communicating Sequential Processes (CSP) [86, 21], and the Kahn Networks [99, 100].

1.3.1.3 Summary of Concurrent and Parallel Programming Models

Table 1.6 presents a summary of the analysis of the paradigm presented in the previous paragraphs.

Model	Implementations	Fine-grain level synchronization	Coarse-grain level message passing	→	Performance Efficiency
Event-driven programming	Node.js, Vert.X	5	0		2
Lock-free Data Structures	list [161, 156], queue [150, 167], tree [136], stack [79]	5	0		2
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	4	0		1
Actor Model	Scala, Akka, Play, Erlang	5	5		5
Skeleton	MapReduce	4	4		4
Service Oriented Architecture	OSGi, EJB, Spring	4	4		4
Microservices	Seneca	4	4		4

Table 1.6 – Performance Efficiency of Concurrent and Parallel Programming Platforms

1.3.2 Adoption

illustration: mars rover

If there is industrial need, there will be maintenance. The languages on the Mars Rover or in banking systems are 30 years old, and there is no community to maintain it. Yet the industry continue to maintain these languages.

When the need for performance is higher than the need for maintainability, the adoption is steered by the industry more than the community. The platform presented in this section emerged from the academia and the industry and did not always gather huge community enthusiasm. The more the platform abandons maintainability, the less it will be supported by the community.

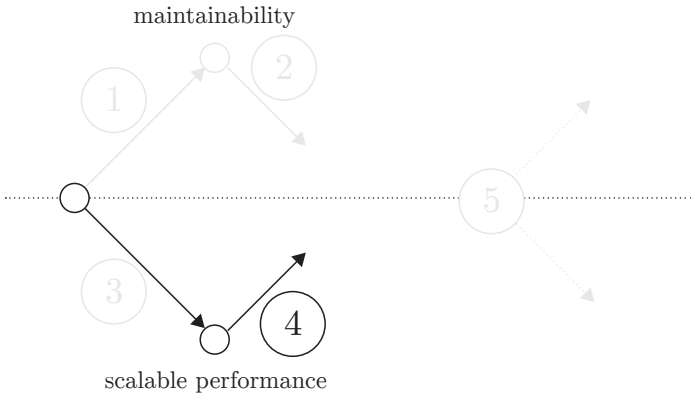


Figure 1.5 – Steer back toward Maintainability

1.3.2.1 Concurrent Programming

Most programming languages implementation supports concurrent programming somehow. Either with multi-threading or event-driven programming. These two are highly adopted by both the industry and the community.

On the other hand, lock-free data structures and cooperative threads comes from the academia, similarly to functionnal programming, and did not encounter yet significant adoption from the industry not the community.

Table 1.7 presents a summary of the adoption of concurrent programming models.

Model	Implementations	Community support	Industrial need	→	Adoption
Event-driven programming	Node.js, Vert.X	5	5		5
Lock-free Data Structures	list [161, 156], queue [150, 167], tree [136], stack [79]	0	0		0
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	0	0		0

Table 1.7 – Adoption of Concurrent Programming Platforms

1.3.2.2 Parallel Programming

There exists several programming languages directly inspired by the actors model, like Akka Scala and Erlang [**JoeArmstrong**]. There are as well some programming languages inspired by other theoretical model, like Go, inspired by Coroutines and CSP.

Scala is an attempt at unifying the object model, and functional programming [128]. Akka[16] is a framework based on Scala, following the actor model to build highly scalable and resilient applications. Play[17] is a web framework based on top of Akka.

Erlang is a functional language designed by Ericsson to operate networks of telecommunication devices [**Armstrong2014**, 11, 122]

Go is an open source language initiated by Google to build highly concurrent services [18].

These examples of implementation are heavily used in the industry. They are backed by strong, but small communities of passionate people.

However, the organization in independent tasks is hardly compatible with the modular organization presented in the previous section. It is difficult for developers

[16] `http://akka.io/`
[17] `https://www.playframework.com/`
[18] `https://golang.org/`

to manage the superposition of these two organizations, tasks and modules. The next paragraphs presents the platforms mitigating the problems stemming from the duality between execution decomposition and modularity.

Tasks Organization and Communications To reduce the difficulties of the superposition of tasks and modules, algorithmic skeletons propose predefined patterns of organization to fit certain types of problems [35, 42, 117, 63]. Developers specialize a skeleton and focus on their problem independently of the required communication. These solutions are hardly used by the community, but are crucial in some industrial contexts. A famous example is the map/reduce pattern introduced by Google [42].

Tasks Granularity The Service Oriented Architectures (SOA), and more recently Microservices [52, 57, 121] allow developers to express an application as an assembly of services connected to each others. Some examples of frameworks are OSGi[19], EJB[20], Spring[21], and Seneca[22] It allows to adjust the granularity of tasks to help developers to better fit the tasks organization with the modular organization [2]. Microservices are very recent, and it is difficult to asses their usage in the community nor the industry. But they seems to be increasingly adopted, both in the industry and in the community.

The parallel programming platform previously presented allow to build generic distributed systems. In the context of the web, a real-time application must process high volumes streams of requests within a certain time. Because these systems are key to business, their reliability and latency are of critical importance. The next paragraphs present the platform meeting these requirements required for industry adoption. Again, these platforms emerged from the academia and the industry and did not always gather huge community enthusiasm.

1.3.2.3 Stream Processing Systems

Data-stream Management Systems Database Management Systems (DBMS) historically processed large volume of data, and they naturally evolved into Data-stream Management System (DSMS) to processed data streams as well. Because of

[19] `https://www.osgi.org/developer/specifications/`
[20] `http://www.oracle.com/technetwork/java/javaee/ejb/index.html`
[21] `http://projects.spring.io/spring-framework/`
[22] `http://senecajs.org/`

this evolution, they are in rupture with imperative languages presented until now, and borrow the syntax from SQL.

DSMS concurrently run SQL-like requests on continuous data streams. The computation of these requests spread over a distributed architecture. Some recent examples are DryadLINQ [93, 171], Apache Hive [155][23], Timestream [133] and Shark [169].

Pipeline Architecture As presented in the previous section, streaming composition allows a loosely coupled yet efficient composition. The pipeline architecture takes advantage of this. It organizes an application as a network of event-driven stages connected by explicit queues, the output of one feeding the input of the next. The event-driven paradigm of a stage is similar to work like Ninja and Flash [65, 130] previously presented.

SEDA is a precursor in the design of pipeline-based architecture for real-time web applications [166]. StreaMIT is a language to help the programming of large streaming application [154]. Storm [157] is designed by and used at Twitter to process the heavy streams of tweets. Among other works, in the industry, there are CBP [111] and S4 [124], that were designed at Yahoo, Millwheel [5] designed at Google and Naiad [120] designed at Microsoft.

In the litterature, there are Spidle [36], Pig Latin [129], Piccolo [132], Comet [78], Nectar [68], SEEP [118], Legion [16], Halide [135], SDG [51] and Regent [143]

Parallel programming is not heavily supported by the community, but emerges mainly from the industrial needs and academic research. Parallel programming improves performance scalability, but prevent the adoption by the community. Despite the performance limitation, the event-driven programming model is the best candidate for a concurrent programming model supported by the community, and needed in the industry. Table 1.8 summarize the adoption of the platform oriented toward performance presented in this section.

[23] `https://hive.apache.org/`

Model	Implementations	Community support	Industrial need	→	Adoption
Event-driven programming	Node.js, Vert.X	5	5		5
Lock-free Data Structures	list [161, 156], queue [150, 167], tree [136], stack [79]	0	0		0
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	0	0		0
Actor Model	Scala, Akka, Play, Erlang	1	5		1
Skeleton	MapReduce	2	5		2
Service Oriented Architecture	OSGi, EJB, Spring	3	4		3
Microservices	Seneca	3	3		3

Table 1.8 – Adoption of Concurrent and Parallel Programming Platforms

1.3.3 Maintainability Limitations

Parallel programming requires the organization of execution and memory into independent tasks. It allows the different granularity of state accessibility required for performance efficiency. At a fine level, the state is shared, while at a coarser level, it is isolated. This difference in state access id dictated by the execution organization in tasks. It prevents high cohesion and low coupling in modules, hence impacts maintainability. Moreover, it makes more difficult higher-order programming.

It implies to keep two mental representation of the implementation, one for the module organization and one for the tasks organization. It makes parallel programming accessible only to an elite of developers. It is incompatible with the economical requirement of this thesis, which focus on platforms accessible for a large community of developer to stimulate the economy around web development.

To fit the economical context of this thesis, a solution must provide performance efficiency while avoiding the developers to keep a double mental representation of the

implementation. It comes with an abstraction for the tasks and memory organization, for the developer to focus only on module organization. The next section presents some works that provides such an abstraction.

Model	Implementations	Composition	Encapsulation	→	Maintainability
Event-driven programming	Node.js, Vert.X	3	3		3
Lock-free Data Structures	list [161, 156], queue [150, 167], tree [136], stack [79]	1	1		1
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	2	2		2
Actor Model	Scala, Akka, Play, Erlang	1	1		1
Skeleton	MapReduce	2	2		2
Service Oriented Architecture	OSGi, EJB, Spring	3	3		3
Microservices	Seneca	3	3		3
Data Stream System Management	DryadLINQ [93, 171], Apache Hive [155], Timestream [133], Shark [169]	2	2		2
Pipeline Stream Processing	SEDA, Storm, Spark Streaming, Spidle [36], Pig Latin [129], Piccolo [132], Comet [78], Nectar [68], SEEP [118], Legion [16], Halide [135], SDG [51], Regent [143]	4	4		4

Table 1.9 – Maintainability of Concurrent, Parallel and Stream Programming Platforms

1.3.4 Summary

Table 1.10 summarizes the characteristics of the platforms presented in this section.

Model	Implementations	Maintainability	Adoption	Performance Efficiency
Event-driven programming	Node.js, Vert.X	3	5	2
Lock-free Data Structures	list [161, 156], queue [150, 167], tree [136], stack [79]	1	0	2
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	2	0	1
Actor Model	Scala, Akka, Play, Erlang	1	1	5
Skeleton	MapReduce	2	2	4
Service Oriented Architecture	OSGi, EJB, Spring	3	3	4
Microservices	Seneca	3	3	4
Data Stream System Management	DryadLINQ [93, 171], Apache Hive [155], Timestream [133], Shark [169]	2	4	3
Pipeline Stream Processing	SEDA, Storm, Spark Streaming, Spidle [36], Pig Latin [129], Piccolo [132], Comet [78], Nectar [68], SEEP [118], Legion [16], Halide [135], SDG [51], Regent [143]	4	2	5

Table 1.10 – Summary of Concurrent and Parallel Programming Platforms

1.4 Adoption Focused Platforms

Section 1.2 and section ?? presented the platforms focusing on maintainability and performance efficiency, and showed that focusing on one negatively impacts the other. A balance between maintainability and performance efficiency is required to have both the community support and the industrial need, required to be widely adopted. This section presents platforms featuring an abstraction of the tasks organization to propose a compromise between maintainability and performance efficiency. Section ?? presents Compilers, and section ?? presents Runtime.

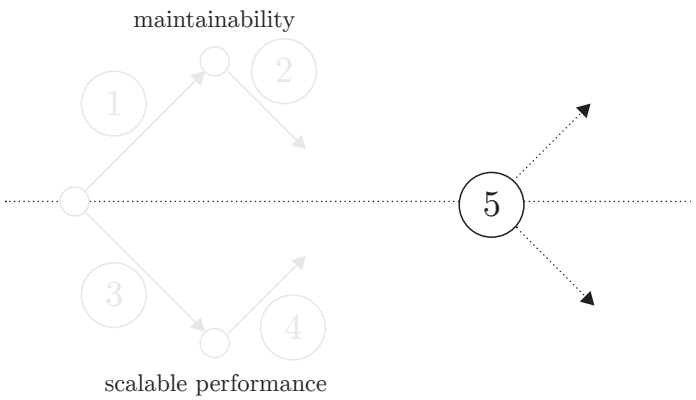


Figure 1.6 – Focus on Adoption

1.4.1 Abstraction of Tasks Organization

1.4.1.1 Compilers

"It is a mistake to attempt high concurrency without help from the compiler"
— R. Behren, J. Condit, E. Brewer [18]

.

As soon as the incompatibility between the modules and the tasks organizations were presented, it was suggested to use a compilation approach to mitigate this incompatibility [131]. This section presents the state of the art to extract parallelization from sequential programs through code transformation and compilation.

*

read and include [24]

Parallelism Extraction Extracting parallelism from a sequential implementation is a hard problem [97]. A compiler needs to identify the commutative operations to parallelize their executions [140, 33].

An important work was done to parallelize loop iterations [116, 7, 31, 14, 134], particularly using the polyhedral compilation method [172, 67, 158, 15]. To improve performance gains further, some compilers identify the data-flow inside sequential programs to allow parallelism on the whole program, and not only on its loops [17, 24, 110]. Moreover, the data-flow representation and execution of a program is well suited for modern data processing applications [51], as well as web services [141].

Mutable closures required for higher-order programming remains a challenge to parallelize because of the memory references shared across the program [76, 125, 115]. The next paragraphs present some improvements in parallel compilation.

Static analysis Compilers statically analyze the control-flow of a program to detect commutative operations [6]. The point-to analysis is a popular approach to identify side-effects [9, 94, 145, 165]. However, this analysis is not sufficient to track the dynamic control-flow of higher-order functions [142] like used in Javascript.

Another approach is the abstract interpretation of the program. Abstract interpretation techniques are more adapted for dynamic languages like Javascript, and are successfully used for security applications [89, 98, 170, 113, 32, 48]*. It allows to statically reason on the behavior of dynamic program [**Raychev2013**, 112, 144, 60, 73, 59, 20].

⚠ Update the citation for Dolby2015

However, static analysis techniques are too imprecise, and expensive for the performance gain to be profitable. Instead, some compilers relies on annotations from the developers.

Annotations Some works proposed to rely on annotations from the developer to identify the commutativity of operations or the shared data structures [163, 51]. Such annotations are especially relevant for accelerators such as GPUs or FPGAs, because the development effort yield huge performance improvements [**Tarditi2006**]. Examples of such compilers are OpenMP [39], OpenCL [147], CUDA [127] Cg [114], Brook [22], Liquid Metal [**Huang2008**].

Compilation Limitations The static analysis of static, low level languages like FORTRAN or C, brings performance improvements. However for more dynamic, higher-level languages like Javascript, the static analysis is not sufficient to identify correctly the dependencies between operations to parallelize them. And parallel compilers often fall back on relying on annotation provided by developers. Hence,

the burden of detecting commutativity of operations falls back to the developer, similarly to the platforms presented in the section 1.3, focusing on performance efficiency.

Alternatively, another approach is to dynamically distribute the commutative operations, and assure the communications. The next paragraphs present runtime allowing this dynamic distribution.

1.4.1.2 Runtimes

Partitioned Global Address Space The Partitioned Global Address Space (PGAS) provides the developers with a uniform memory access on a distributed architecture. It attempts to combine the performance efficiency of distributed memory systems, with the maintainability of shared memory systems. Each computing node executes the same program, and provide its local memory to be shared with all the other nodes. The PGAS platform assures the remote accesses and synchronization of memory across nodes. Examples of implementation of the PGAS model are CoArray Fortran [126], X10 [30]. Unified Parallel C [61], Chapel[25], OpenSHMEM [29]. Kokko [49], UPC++ [173], RAJA [88], ACPdl [4] and HPX [**Kaiser2014**, 101]

Dynamic Distribution of Execution An interesting work following SEDA, is Leda [**Salmito2014**, 141]. It proposes a model where the stages of the pipeline are defined only by their role in the application. The actual execution distribution is defined automatically during deployment. This automation manages the execution organizations to help the developer focus on the modular organization.

Tables 1.11 and 1.13 presents the platforms presented in this section regarding maintainability and performance.

Model	Implementations	Composition	Encapsulation	→	Maintainability
PGAS	CoArray Fortran [126], X10 [30], Unified Parallel C [61], Chapel[25], OpenSHMEM [29], Kokko [49], UPC++ [173], RAJA [88], ACPdl [4], HPX [101]	0	0		0
Dynamic Distribution	Leda [**Salmito2014**, 141]	0	0		0
Polyhedral Compiler	-	0	0		0
Annotation Compiler	OpenMP [39], OpenCL [147], CUDA [127], Cg [114], Brook [22], Liquid Metal [**Huang2008**]	0	0		0

Table 1.11 – Maintainability of Compilation and Runtime Platforms

Model	Implementations	Fine-grain level synchronization	Coarse-grain level message passing	→	Performance Efficiency
PGAS	CoArray Fortran [126], X10 [30], Unified Parallel C [61], Chapel[25], OpenSHMEM [29], Kokko [49], UPC++ [173], RAJA [88], ACPdl [4], HPX [101]	0	0		0
Dynamic Distribution	Leda [**Salmito2014**, 141]	0	0		0
Polyhedral Compiler	-	0	0		0
Annotation Compiler	OpenMP [39], OpenCL [147], CUDA [127], Cg [114], Brook [22], Liquid Metal [**Huang2008**]	0	0		0

Table 1.12 – Performance Efficiency of Compilation and Runtime Platforms

1.4.2 Adoption Limitations

All the platforms presented in this section come from the need to reduce the development commitment required for performance efficiency. However, none of these platforms are highly supported by the community because they respond exclusively to industrial needs.

They are limited to scientific applications.

The balance between performance efficiency and maintainability is not sufficient for a community of passionate to gather around the platform. The platform needs to allow the community to experiment and to start projects. The context of web development is particularly suited to experiment and start projects.

Model	Implementations	Community support	Industrial need	→	Adoption
PGAS	CoArray Fortran [126], X10 [30], Unified Parallel C [61], Chapel[25], OpenSHMEM [29], Kokko [49], UPC++ [173], RAJA [88], ACPdl [4], HPX [101]	0	0		0
Dynamic Distribution	Leda [**Salmito2014**, 141]	0	0		0
Polyhedral Compiler	-	0	0		0
Annotation Compiler	OpenMP [39], OpenCL [147], CUDA [127], Cg [114], Brook [22], Liquid Metal [**Huang2008**]	0	0		0

Table 1.13 – Adoption of Compilation and Runtime Platforms

1.4.3 Summary

Table 1.14 summarizes the characteristics of the platforms presented in this section.

Model	Implementations	Maintainability	Adoption	Performance Efficiency
PGAS	CoArray Fortran [126], X10 [30], Unified Parallel C [61], Chapel[25], OpenSHMEM [29], Kokko [49], UPC++ [173], RAJA [88], ACPdl [4], HPX [101]	0	0	0
Dynamic Distribution	Leda [**Salmito2014**, 141]	0	0	0
Polyhedral Compiler	-	0	0	0
Annotation Compiler	OpenMP [39], OpenCL [147], CUDA [127], Cg [114], Brook [22], Liquid Metal [**Huang2008**]	0	0	0

Table 1.14 – Summary of Compilation and Runtime Platforms

1.5 Analysis

Model	Implementations	Maintainability	Adoption	Performance Efficiency
Imperative Programming	C	3	3	0
Object-Oriented Programming	C++	5	4	0
Functional Programming	Scheme	5	0	0
Multi Paradigm	Javacript, Python, Ruby, Go	5	4	0

Event-driven programming	Node.js, Vert.X	3	5	2
Lock-free Data Structures	list [161, 156], queue [150, 167], tree [136], stack [79]	1	0	2
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	2	0	1
Actor Model	Scala, Akka, Play, Erlang	1	1	5
Skeleton	MapReduce	2	2	4
Service Oriented Architecture	OSGi, EJB, Spring	3	3	4
Microservices	Seneca	3	3	4
Data Stream System Management	DryadLINQ [93, 171], Apache Hive [155], Timestream [133], Shark [169]	2	4	3
Pipeline Stream Processing	SEDA, Storm, Spark Streaming, Spidle [36], Pig Latin [129], Piccolo [132], Comet [78], Nectar [68], SEEP [118], Legion [16], Halide [135], SDG [51], Regent [143]	4	2	5
PGAS	CoArray Fortran [126], X10 [30], Unified Parallel C [61], Chapel[25], OpenSHMEM [29], Kokko [49], UPC++ [173], RAJA [88], ACPdl [4], HPX [101]	0	0	0
Dynamic Distribution	Leda [**Salmito2014**, 141]	0	0	0
Polyhedral Compiler	-	0	0	0

Annotation Compiler	OpenMP [39], OpenCL [147], CUDA [127], Cg [114], Brook [22], Liquid Metal [**Huang2008**]	⓪	⓪	⓪

Table 1.15 – Maintainability of Modular Programming Platforms

TODO

Bibliography

[1] H Abelson, G J Sussman, and J Sussman. *The Structure and Interpretation of Computer Programs*. Vol. 9. 3. 1985, p. 81. DOI: 10.2307/3679579.

[2] Sebastian Adam and Joerg Doerr. "How to better align BPM & SOA - Ideas on improving the transition between process design and deployment". In: *CEUR Workshop Proceedings*. Vol. 335. 2008, pp. 49–55.

[3] A Adya, J Howell, and M Theimer. "Cooperative Task Management Without Manual Stack Management." In: *USENIX Annual Technical Conference* (2002).

[4] Yuichiro Ajima, Takafumi Nose, Kazushige Saga, Naoyuki Shida, and Shinji Sumimoto. "ACPdl". In: *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware - ESPM '15*. New York, New York, USA: ACM Press, Nov. 2015, pp. 11–18. DOI: 10.1145/2832241.2832242.

[5] T Akidau and A Balikov. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment 6.11* (2013).

[6] Frances E. Allen. "Control flow analysis". In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. DOI: 10.1145/390013.808479.

[7] SP Amarasinghe, JAM Anderson, MS Lam, and CW Tseng. "An Overview of the SUIF Compiler for Scalable Parallel Machines." In: *PPSC* (1995).

[8] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *AFIPS Spring Joint Computer Conference, 1967. AFIPS '67 (Spring). Proceedings of the*. Vol. 30. 1967, pp. 483–485. DOI: doi:10.1145/1465482.1465560.

[9] LO Andersen. "Program analysis and specialization for the C programming language". In: (1994).

[10] James H. Anderson and Mohamed G. Gouda. *The virtue of Patience: Concurrent Programming With And Without Waiting*. 1990.

[11] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in ERLANG*. 1993.

[12] Krste Asanovic, Bryan Christopher Catanzaro, David a Patterson, and Katherine a Yelick. "The Landscape of Parallel Computing Research : A View from Berkeley". In: *EECS Department University of California Berkeley Tech Rep UCBEECS2006183* 18 (2006), p. 19. DOI: 10.1145/1562764.1562783.

[13] Michel Auguin and Francois Larbey. "OPSILA: an advanced SIMD for numerical analysis and signal processing". In: *Microcomputers: developments in industry, business, and education*. 1983, pp. 311–318.

[14] U Banerjee. *Loop parallelization*. 2013.

[15] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. "Putting Polyhedral Loop Transformations to Work". In: *LCPC '04 Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science 2958.Chapter 14 (2004). Ed. by Lawrence Rauchwerger, pp. 209–225. DOI: 10.1007/b95707.

[16] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing Networking Storage and Analysis SC 12* (Nov. 2012), pp. 1–11. DOI: 10.1109/SC.2012.71.

[17] Micah Beck, Richard Johnson, and Keshav Pingali. "From control flow to dataflow". In: *Journal of Parallel and Distributed Computing* 12.2 (1991), pp. 118–129. DOI: 10.1016/0743-7315(91)90016-3.

[18] JR von Behren, J Condit, and EA Brewer. "Why Events Are a Bad Idea (for High-Concurrency Servers)." In: *HotOS* (2003).

[19] R Von Behren, J Condit, and F Zhou. "Capriccio: scalable threads for internet services". In: *ACM SIGOPS ...* (2003).

[20] M Bodin and A Charguéraud. "A trusted mechanised JavaSript specification". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014).

[21] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. "A Theory of Communicating Sequential Processes". In: *Journal of the ACM* 31.3 (June 1984), pp. 560–599. DOI: 10.1145/828.833.

[22] I Buck, T Foley, and D Horn. "Brook for GPUs: stream computing on graphics hardware". In: *... on Graphics (TOG)* (2004).

[23] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. "Identification of coordination requirements". In: *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work - CSCW '06*. New York, New York, USA: ACM Press, Nov. 2006, p. 353. DOI: 10.1145/1180875.1180929.

[24] Bryan Catanzaro, Shoaib Kamil, and Yunsup Lee. "SEJITS: Getting productivity and performance with selective embedded JIT specialization". In: *... Models for Emerging ...* (2009), pp. 1–10. DOI: 10.1.1.212.6088.

[25] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *International Journal of High Performance Computing Applications* 21.3 (Aug. 2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[26] F Chan, J N Cao, A T S Chan, and M Y Guo. "Programming support for MPMD parallel computing in ClusterGOP". In: *IEICE Transactions on Information and Systems* E87D.7 (2004), pp. 1693–1702.

[27] K. Mani Chandy and Carl Kesselman. "Compositional C++: Compositional parallel programming". In: *Languages and Compilers for Parallel Computing*. Vol. 757. 2005, pp. 124–144. DOI: 10.1007/3-540-48319-5.

[28] Chi-Chao Chang, G. Czajkowski, T. Von Eicken, and C. Kesselman. "Evaluating the Performance Limitations of MPMD Communication". In: *ACM/IEEE SC 1997 Conference (SC'97)* (1997), pp. 1–10. DOI: 10.1109/SC.1997.10040.

[29] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. "Introducing OpenSHMEM". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model - PGAS '10*. New York, New York, USA: ACM Press, Oct. 2010, pp. 1–3. DOI: 10.1145/2020373.2020375.

[30] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*. Vol. 40. 10. New York, New York, USA: ACM Press, Oct. 2005, p. 519. DOI: 10.1145/1094811.1094852.

[31] Chun Chen, Jacqueline Chame, and Mary Hall. "CHiLL: A framework for composing high-level loop transformations". In: *U. of Southern California, Tech. Rep* (2008), pp. 1–28. DOI: 10.1001/archneur.64.6.785.

[32] Andrey Chudnov and David A. Naumann. "Inlined Information Flow Monitoring for JavaScript". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Oct. 2015), pp. 629–643. DOI: 10.1145/2810103.2813684.

[33] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. "The scalable commutativity rule". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. New York, New York, USA: ACM Press, Nov. 2013, pp. 1–17. DOI: 10.1145/2517349.2522712.

[34] William Douglas Clinger. "Foundations of Actor Semantics". eng. In: (May 1981).

[35] M. I. Cole. *Algorithmic skeletons : A structured approach to the management of parallel computation*. eng. 1988.

[36] C Consel, H Hamdi, and L Réveillère. "Spidle: a DSL approach to specifying streaming applications". In: *Generative ...* (2003).

[37] Melvin E. Conway. "Design of a separable transition-diagram compiler". In: *Communications of the ACM* 6.7 (July 1963), pp. 396–408. DOI: 10.1145/366663.366704.

[38] David E. Culler, A. Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. "Parallel programming in Split-C". English. In: (), pp. 262–273. DOI: 10.1109/SUPERC.1993.1263470.

[39] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". English. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[40] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. "A single-program-multiple-data computational model for EPEX/FORTRAN". In: *Parallel Computing* 7.1 (Apr. 1988), pp. 11–24. DOI: 10.1016/0167-8191(88)90094-4.

[41] Frederica Darema. "The SPMD Model: Past , Present and Future". In: *Parallel Computing*. 2001, p. 1. DOI: 10.1007/3-540-45417-9{_}1.

[42] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proc. of the OSDI - Symp. on Operating Systems Design and Implementation*. Vol. 51. 1. 2004, pp. 137–149. DOI: 10.1145/1327452.1327492. arXiv: 10.1.1.163.5292.

[43] E W Dijkstra. *Notes on structured programming*. 1970.

[44] Edsger Dijkstra. "Over de sequentialiteit van procesbeschrijvingen". In: ().

[45] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975.

[46] Edsger W. Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Communications of the ACM* 11.3 (Mar. 1968), pp. 147–148. DOI: 10.1145/362929.362947.

[47] Edsger W. Dijkstra. "The structure of the "THE"-multiprogramming system". In: *Communications of the ACM* 11.5 (May 1968), pp. 341–346. DOI: 10.1145/363095.363143.

[48] Julian Dolby. "A History of JavaScript Static Analysis with WALA at IBM". In: (2015).

[49] H. Carter Edwards and Daniel Sunderland. "Kokkos Array performance-portable manycore programming model". In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '12*. New York, New York, USA: ACM Press, Feb. 2012, pp. 1–10. DOI: 10.1145/2141702.2141703.

[50] Dawson R. Engler and Todd A. Proebsting. "DCG". In: *ACM SIGOPS Operating Systems Review* 28.5 (Dec. 1994), pp. 263–272. DOI: 10.1145/381792.195567.

[51] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. "Making state explicit for imperative big data processing". In: *USENIX ATC* (2014).

[52] JI Fernández-Villamor. "Microservices-Lightweight Service Descriptions for REST Architectural Style." In: *... 2010-Proceedings of ...* (2010).

[53] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". English. In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[54] Ian Foster. *Designing and Building Parallel Programs*. 1995.

[55] Ian Foster, Carl Kesselman, and Steven Tuecke. "The Nexus Approach to Integrating Multithreading and Communication". In: *Journal of Parallel and Distributed Computing* 37.1 (Aug. 1996), pp. 70–82. DOI: 10.1006/jpdc.1996.0108.

[56] I.T. Foster and K M Chandy. "Fortran M: A Language for Modular Parallel Programming". In: *Journal of Parallel and Distributed Computing* 26.1 (Apr. 1995), pp. 24–35. DOI: 10.1006/jpdc.1995.1044.

[57] M Fowler and J Lewis. "Microservices". In: *... . http://martinfowler. com/articles/microservices. html [...* (2014).

[58] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The implementation of the Cilk-5 multithreaded language". In: *ACM SIGPLAN Notices* 33.5 (May 1998), pp. 212–223. DOI: 10.1145/277652.277725. arXiv: 9809069v1 [arXiv:gr-qc].

[59] P Gardner and G Smith. "JuS: Squeezing the sense out of javascript programs". In: *JSTools@ ECOOP* (2013).

[60] PA Gardner, S Maffeis, and GD Smith. "Towards a program logic for JavaScript". In: *ACM SIGPLAN Notices* (2012).

[61] Tarek El-Ghazawi and Lauren Smith. "UPC: unified parallel C". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06*. New York, New York, USA: ACM Press, Nov. 2006, p. 27. DOI: 10.1145/1188455.1188483.

[62] Adele Goldberg. *Smalltalk-80 : the interactive programming environment*. 1984, xi, 516 p.

[63] Horacio González-Vélez and Mario Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers". In: *Software: Practice and Experience* 40.12 (Nov. 2010), pp. 1135–1160. DOI: 10.1002/spe.1026.

[64] J Gosling. *The Java language specification*. 2000.

[65] Steven D. Gribble, Matt Welsh, Rob Von Behren, Eric a. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. "Ninja architecture for robust Internet-scale systems and services". In: *Computer Networks* 35.4 (2001), pp. 473–497. DOI: 10.1016/S1389-1286(00)00179-1.

[66] Andrew S. Grimshaw. "An Introduction to Parallel Object-Oriented Programming with Mentat". In: (Apr. 1991).

[67] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. "Polly - Polyhedral optimization in LLVM". In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT '11)* (2011), None.

[68] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan a Thekkath, Yuan Yu, and Li Zhuang. "Nectar : Automatic Management of Data and Computation in Datacenters". In: *Technology* (2010), pp. 1–8.

[69] NJ Gunther. "A New Interpretation of Amdahl's Law and Geometric Scalability". In: *arXiv preprint cs/0210017* (2002).

[70] NJ Gunther. "A simple capacity model of massively parallel transaction systems". In: *CMG-CONFERENCE-* (1993).

[71] NJ Gunther. "Understanding the MP effect: Multiprocessing in pictures". In: *In other words* (1996).

[72] JL Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* (1988).

[73] B Hackett and S Guo. "Fast and precise hybrid type inference for JavaScript". In: *ACM SIGPLAN Notices* (2012).

[74] P.B. Hansen and J. Staunstrup. "Specification and Implementation of Mutual Exclusion". English. In: *IEEE Transactions on Software Engineering* SE-4.5 (Sept. 1978), pp. 365–370. DOI: 10.1109/TSE.1978.233856.

[75] Tim Harris, James Larus, and Ravi Rajwar. "Transactional Memory, 2nd edition". en. In: *Synthesis Lectures on Computer Architecture* 5.1 (Dec. 2010), pp. 1–263. DOI: 10.2200/S00272ED1V01Y201006CAC011.

[76] Williams Ludwell Harrison. "The interprocedural analysis and automatic parallelization of Scheme programs". In: *Lisp and Symbolic Computation* 2.3-4 (Oct. 1989), pp. 179–396. DOI: 10.1007/BF01808954.

[77] CT Haynes, DP Friedman, and M Wand. "Continuations and coroutines". In: *... of the 1984 ACM Symposium on ...* (1984).

[78] B He, M Yang, Z Guo, R Chen, and B Su. "Comet: batched stream processing for data intensive distributed computing". In: *... on Cloud computing* (2010).

[79] Danny Hendler, Nir Shavit, and Lena Yerushalmi. "A scalable lock-free stack algorithm". In: *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures - SPAA '04*. New York, New York, USA: ACM Press, June 2004, p. 206. DOI: 10.1145/1007912.1007944.

[80] M. Herlihy. "A methodology for implementing highly concurrent data structures". In: *ACM SIGPLAN Notices* 25.3 (Mar. 1990), pp. 197–206. DOI: 10.1145/99164.99185.

[81] Maurice Herlihy. "Wait-free synchronization". In: *ACM Transactions on Programming Languages and Systems* 13.1 (Jan. 1991), pp. 124–149. DOI: 10.1145/114005.102808.

[82] Maurice P. Herlihy. "Impossibility and universality results for wait-free synchronization". In: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing - PODC '88*. New York, New York, USA: ACM Press, Jan. 1988, pp. 276–290. DOI: 10.1145/62546.62593.

[83] C Hewitt, P Bishop, and R Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence* (1973).

[84] Carl Hewitt. "Viewing control structures as patterns of passing messages". In: *Artificial intelligence* (1977).

[85] Carl Hewitt and Jr Baker Henry. "Actors and Continuous Functionals," in: (Dec. 1977).

[86] C. A. R. Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677. DOI: 10.1145/359576.359585.

[87] C. A. R. Hoare. "Monitors: an operating system structuring concept". In: *Communications of the ACM* 17.10 (Oct. 1974), pp. 549–557. DOI: 10.1145/355620.361161.

[88] R D Hornung and J A Keasler. "The RAJA Portability Layer : Overview and Status". In: (2014).

[89] YW Huang, F Yu, C Hang, and CH Tsai. "Securing web application code by static analysis and runtime protection". In: *Proceedings of the 13th ...* (2004).

[90] Paul Hudak, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, John Peterson, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, and John Hughes. "Report on the programming language Haskell". In: *ACM SIGPLAN Notices* 27.5 (May 1992), pp. 1–164. DOI: 10.1145/130697.130699.

[91] John Hughes. "Why functional programming matters". In: *The computer journal* 32.April 1989 (1989), pp. 1–23. DOI: 10.1093/comjnl/32.2.98.

[92] Walter Hürsch and Cristina Videira Lopes. *Separation of Concerns*. Tech. rep. NU-CCS-95-03. 1995.

[93] M Isard, M Budiu, Y Yu, A Birrell, and D Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In: *ACM SIGOPS Operating ...* (2007).

[94] D Jang and KM Choe. "Points-to analysis for JavaScript". In: *Proceedings of the 2009 ACM symposium on Applied ...* (2009).

[95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. "CRL: High-Performance All-Software Distributed Shared Memory". In: *ACM SIGOPS Operating Systems Review* 29.5 (Dec. 1995), pp. 213–226. DOI: 10.1145/224057.224073.

[96] Ralph E. Johnson and Brian Foote. "Designing Reusable Classes Abstract Designing Reusable Classes". In: *Journal of Object-Oriented Programming* 1 (1988), pp. 22–35.

[97] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in dataflow programming languages". In: *ACM Computing Surveys* 36.1 (Mar. 2004), pp. 1–34. DOI: 10.1145/1013208.1013209.

[98] N Jovanovic, C Kruegel, and E Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities". In: *Security and Privacy, 2006 ...* (2006).

[99] Gilles Kahn. "The semantics of a simple language for parallel programming". In: *In Information Processing'74: Proceedings of the IFIP Congress* 74 (1974), pp. 471–475.

[100] Gilles Kahn and David Macqueen. *Coroutines and Networks of Parallel Processes*. en. Tech. rep. 1976, p. 20.

[101] Hartmut Kaiser, Thomas Heller, and Daniel Bourgeois. *Higher-level Parallelization for Local and Distributed Asynchronous Task-Based Programming*. 2015.

[102] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. "Load-balanced pipeline parallelism". English. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*. New York, New York, USA: ACM Press, 2013, pp. 1–12. DOI: 10.1145/2503210.2503295.

[103] Dejan Kovachev, Yiwei Cao, and Ralf Klamma. "Mobile Cloud Computing: A Comparison of Application Models". In: (July 2011). arXiv: 1107.4940.

[104] MN Krohn, E Kohler, and MF Kaashoek. "Events Can Make Sense." In: *USENIX Annual Technical Conference* (2007).

[105] L Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* (1978).

[106] Leslie Lamport. "Concurrent reading and writing". In: *Communications of the ACM* 20.11 (Nov. 1977), pp. 806–811. DOI: 10.1145/359863.359878.

[107] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (July 1982), pp. 382–401. DOI: 10.1145/357172.357176.

[108] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. "On-the-fly pipeline parallelism". In: *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures* (2013), p. 140. DOI: 10.1145/2486159.2486174.

[109] Charles E. Leiserson. "The Cilk++ concurrency platform". In: *Journal of Supercomputing* 51.3 (Mar. 2010), pp. 244–257. DOI: 10.1007/s11227-010-0405-3.

[110] Feng Li, Antoniu Pop, and Albert Cohen. "Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs". English. In: *IEEE Micro* 32.4 (July 2012), pp. 19–31. DOI: 10.1109/MM.2012.49.

[111] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. "Stateful bulk processing for incremental analytics". In: *International Conference on Management of Data* (2010), pp. 51–62. DOI: 10.1145/1807128.1807138.

[112] S Maffeis, JC Mitchell, and A Taly. "An operational semantics for JavaScript". In: *Programming languages and systems* (2008).

[113] S Maffeis, JC Mitchell, and A Taly. "Isolating JavaScript with filters, rewriting, and wrappers". In: *Computer Security–ESORICS 2009* (2009).

[114] WR Mark and RS Glanville. "Cg: A system for programming graphics hardware in a C-like language". In: *... Transactions on Graphics (...* (2003).

[115] Nicholas D Matsakis. "Parallel Closures A new twist on an old idea". In: *HotPar'12 Proceedings of the 4th USENIX conference on Hot Topics in Parallelism* (2012), pp. 5–5.

[116] Christophe Mauras. "Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones". PhD thesis. Jan. 1989.

[117] MD McCool. "Structured parallel programming with deterministic patterns". In: *Proceedings of the 2nd USENIX conference on Hot ...* (2010).

[118] M Migliavacca and D Eyers. "SEEP: scalable and elastic event processing". In: *Middleware'10 Posters ...* (2010).

[119] R. Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. 1997, p. 128.

[120] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. "Naiad". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13* (Nov. 2013), pp. 439–455. DOI: 10.1145/2517349.2522738.

[121] Dmitry Namiot and Manfred Sneps-Sneppe. *On Micro-services Architecture*. en. Aug. 2014.

[122] Jay Nelson. "Structured programming using processes". In: *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang - ERLANG '04*. New York, New York, USA: ACM Press, Sept. 2004, pp. 54–64. DOI: 10.1145/1022471.1022480.

[123] R Nelson. "Including queueing effects in Amdahl's law". In: *Communications of the ACM* (1996).

[124] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. "S4: Distributed stream computing platform". In: *Proceedings - IEEE International Conference on Data Mining, ICDM*. 2010, pp. 170–177. DOI: 10.1109/ICDMW.2010.172.

[125] Jens Nicolay. "Automatic Parallelization of Scheme Programs using Static Analysis". PhD thesis. 2010.

[126] Robert W. Numrich and John Reid. "Co-array Fortran for parallel programming". In: *ACM SIGPLAN Fortran Forum* 17.2 (Aug. 1998), pp. 1–31. DOI: 10.1145/289918.289920.

[127] C Nvidia. "Compute unified device architecture programming guide". In: (2007).

[128] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. "An Overview of the Scala Programming Language". In: *System* Section 2 (2004), pp. 1–130.

[129] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig Latin: A Not-So-Foreign Language for Data Processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08* (June 2008), p. 1099. DOI: 10.1145/1376616.1376726.

[130] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. *Flash : An Efficient and Portable Web Server*. 1999. DOI: 10.1.1.119.6738.

[131] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058. DOI: 10.1145/361598.361623.

[132] R Power and J Li. "Piccolo: Building Fast, Distributed Programs with Partitioned Tables." In: *OSDI* (2010).

[133] Z Qian, Y He, C Su, Z Wu, and H Zhu. "Timestream: Reliable stream computation in the cloud". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)* (2013).

[134] C Radoi, SJ Fink, R Rabbah, and M Sridharan. "Translating imperative code to MapReduce". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2014).

[135] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Frédo Durand, Connelly Barnes, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), pp. 519–530. DOI: 10.1145/2491956.2462176.

[136] Arunmoezhi Ramachandran and Neeraj Mittal. "A Fast Lock-Free Internal Binary Search Tree". In: *Proceedings of the 2015 International Conference on Distributed Computing and Networking - ICDCN '15*. New York, New York, USA: ACM Press, Jan. 2015, pp. 1–10. DOI: 10.1145/2684464.2684472.

[137] K.H. Randall. "Cilk: Efficient Multithreaded Computing". PhD thesis. 1998.

[138] DP Reed. "" Simultaneous" Considered Harmful: Modular Parallelism." In: *HotPar* (2012).

[139] J Rees and W Clinger. "Revised report on the algorithmic language scheme". In: *ACM SIGPLAN Notices* 21.12 (Dec. 1986), pp. 37–79. DOI: 10.1145/15042.15043.

[140] MC Rinard and PC Diniz. "Commutativity analysis: A new analysis framework for parallelizing compilers". In: *ACM SIGPLAN Notices* (1996).

[141] Tiago Salmito, Ana Lucia de Moura, and Noemi Rodriguez. "A Flexible Approach to Staged Events". English. In: *2013 42nd International Conference on Parallel Processing* (Oct. 2013), pp. 661–670. DOI: 10.1109/ICPP.2013.80.

[142] O. Shivers. “Control-flow analysis of higher-order languages”. PhD thesis. 1991, pp. 1–186.

[143] Elliott Slaughter, Wonchan Lee, Sean Treichler, and Michael Bauer. “Regent : A High-Productivity Programming Language for HPC with Logical Regions”. In: *SC* (2015). DOI: 10.1145/2807591.2807629.

[144] GD Smith. “Local reasoning about web programs”. In: (2011).

[145] M Sridharan, J Dolby, and S Chandra. “Correlation tracking for points-to analysis of JavaScript”. In: *ECOOP 2012–Object- ...* (2012).

[146] W. P. Stevens, G. J. Myers, and L. L. Constantine. “Structured design”. English. In: *IBM Systems Journal* 13.2 (1974), pp. 115–139. DOI: 10.1147/sj.132.0115.

[147] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (May 2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[148] B Stroustrup. “The C++ programming language”. In: (1986).

[149] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. “The structure and value of modularity in software design”. In: *ACM SIGSOFT Software Engineering Notes* 26.5 (Sept. 2001), p. 99. DOI: 10.1145/503271.503224.

[150] H. Sundell and P. Tsigas. “Fast and lock-free concurrent priority queues for multi-thread systems”. In: *Proceedings International Parallel and Distributed Processing Symposium* 00.C (2003), p. 11. DOI: 10.1109/IPDPS.2003.1213189.

[151] Gerald Jay Sussman and Jr Steele, Guy L. “Scheme: A interpreter for extended lambda calculus”. In: *Higher-Order and Symbolic Computation* 11 (1998), pp. 405–439. DOI: 10.1023/A:1010035624696.

[152] Richard E Sweet. “The Mesa programming environment”. In: *ACM SIGPLAN Notices*. Vol. 20. 7. 1985, pp. 216–229. DOI: 10.1145/17919.806843.

[153] P. Tarr, H. Ossher, W. Harrison, and Jr. Sutton, S.M. “N degrees of separation: multi-dimensional separation of concerns”. In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)* (1999), pp. 107–119. DOI: 10.1145/302405.302457.

[154] William Thies, Michal Karczmarek, and Saman Amarasinghe. "StreamIt: A language for streaming applications". In: *Compiler Construction* LNCS 2304 (2002), pp. 179–196. DOI: 10.1007/3-540-45937-5.

[155] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive". In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1626–1629. DOI: 10.14778/1687553.1687609.

[156] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. "Wait-free linked-lists". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7702 LNCS. 2012, pp. 330–344. DOI: 10.1007/978-3-642-35476-2{_}23.

[157] Ankit Toshniwal, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, and Maosong Fu. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*. New York, New York, USA: ACM Press, June 2014, pp. 147–156. DOI: 10.1145/2588555.2595641.

[158] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. *GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation*. en. Jan. 2010.

[159] D Turner. "An overview of Miranda". In: *ACM SIGPLAN Notices* 21.12 (Dec. 1986), pp. 158–166. DOI: 10.1145/15042.15053.

[160] D. A. Turner. "The semantic elegance of applicative languages". In: *Proceedings of the 1981 conference on Functional programming languages and computer architecture - FPCA '81*. New York, New York, USA: ACM Press, Oct. 1981, pp. 85–92. DOI: 10.1145/800223.806766.

[161] John D. Valois. "Lock-free linked lists using compare-and-swap". In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing - PODC '95*. New York, New York, USA: ACM Press, Aug. 1995, pp. 214–222. DOI: 10.1145/224964.224988.

[162] Peter Van Roy and Seif Haridi. "Concepts, Techniques, and Models of Computer Programming". In: *Theory and Practice of Logic Programming* 5 (2003), pp. 595–600. DOI: 10.1017/S1471068405002450.

[163] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. "The Paralax infrastructure: automatic parallelization with a helping hand". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, New York, USA: ACM Press, Sept. 2010, pp. 389–399. DOI: 10.1145/1854273.1854322.

[164] Philip Wadler. "The essence of functional programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92*. New York, New York, USA: ACM Press, Feb. 1992, pp. 1–14. DOI: 10.1145/143165.143169.

[165] S Wei and BG Ryder. "State-sensitive points-to analysis for the dynamic behavior of JavaScript objects". In: *ECOOP 2014–Object-Oriented Programming* (2014).

[166] M Welsh, D Culler, and E Brewer. "SEDA: an architecture for well-conditioned, scalable internet services". In: *ACM SIGOPS Operating Systems Review* (2001).

[167] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. "The lock-free k-LSM relaxed priority queue". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2015*. New York, New York, USA: ACM Press, Jan. 2015, pp. 277–278. DOI: 10.1145/2688500.2688547.

[168] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. "Design Rule Hierarchies and Parallelism in Software Development Tasks". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Nov. 2009, pp. 197–208. DOI: 10.1109/ASE.2009.53.

[169] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Shark". In: *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*. New York, New York, USA: ACM Press, June 2013, p. 13. DOI: 10.1145/2463676.2465288.

[170] D Yu, A Chander, N Islam, and I Serikov. "JavaScript instrumentation for browser security". In: *ACM SIGPLAN Notices* (2007).

[171] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, Jon Currey, Frank McSherry, Kannan Achan, and Christophe Poulain. "Some sample programs written in DryadLINQ". In: *Microsoft Research* (2009).

[172] Tomofumi Yuki, Gautam Gupta, Daegon Kim, Tanveer Pathan, and Sanjay Rajopadhye. "AlphaZ: A system for design space exploration in the polyhedral model". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7760 LNCS (2013), pp. 17–31. DOI: 10.1007/978-3-642-37658-0{_}2.

[173] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. "UPC++: A PGAS Extension for C++". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 1105–1114. DOI: 10.1109/IPDPS.2014.115.