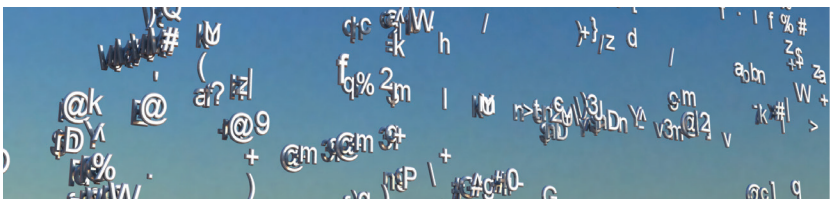
Scripting Multiple CPUs with Safe Data Sharing

Alexandre Skyrme, Noemi Rodriguez, and Roberto Ierusalimschy,
Pontifical Catholic University of Rio de Janeiro

// Scripting languages have become increasingly popular. At the same time, multicore processors are everywhere. However, concurrency support in scripting languages is often limited and difficult to use. Combining several key emerging concurrency patterns could change that, by ensuring safe data sharing in scripting languages. //



SCRIPTING LANGUAGES have become increasingly popular, as their high ranking on public programming language indexes clearly shows.[1,2] In January 2014, the TIOBE, PYPL (Popularity of Programming Language), and LangPop indexes each included between three and five scripting languages among the top 10 most popular languages. Programmers have successfully used scripting languages to implement a range of applications, from games to scientific simulations to computer networking software.

At the same time, multicore processors are everywhere, from desktop computers to mobile devices, and the so-called *concurrency revolution* implies the "free performance lunch" is over[3]—that is, programmers will have to turn to concurrency as a means to improve performance. A survey on parallel programming conducted for Intel[4] shows that, among surveyed software developers, 82 percent felt parallel programming was important to their software. Still, concurrent programming remains difficult,[3,5,6] which has prompted research into new concurrency models that emphasize safety and make it easier for programmers to write correct parallel applications.

Considering the popularity and wide applicability of scripting languages, it's important that they offer suitable concurrency support to allow for parallel execution on multiple processors and processor cores. Most, if not all, popular scripting languages already offer some concurrency support. However, the support is often limited and doesn't let them benefit from parallel execution. This support is also often based on dated, complex constructs for synchronization and communication. Even so, the interest in researching how scripting languages can allow for safe parallel programming is still lagging behind that of compiled languages.

One of the biggest challenges to allowing safe parallel programming lies in how to provide programmers with disciplined mechanisms for sharing data among execution flows. Data sharing is a key matter for parallel programming because it serves as a foundation for communication among execution flows. It also requires programmer attention because the lack or incautious use of synchronization constructs when accessing shared data can lead to data races, generating unpredictable results and making programs impossible to reason about.[7] The rules for using shared data depend on memory models and can be confusing, even for expert programmers.[8]

 0740-7459/14/$31.00 © 2014 IEEE

This article explores language-based mechanisms that allow for safe data sharing among parallel execution flows in scripting languages. After discussing the landscape of conventional concurrency support in popular scripting languages and pointing out existing limitations, we describe some emerging concurrency patterns in scripting languages that can improve concurrency support. We also analyze some of the limitations in current mechanisms and discuss how to combine the patterns we've identified to provide safe data sharing.

Conventional Concurrency Support

Conventionally, scripting languages offer concurrency support by means of multiprocessing with fork-like functions, or of multithreading with shared memory. These forms have several shortcomings, however. Multiprocessing support ensures that it's technically possible to execute code in parallel. Nevertheless, most implementations offer only a thin layer that calls operating system interprocess communication methods, such as sockets, pipes, and shared-memory regions. These low-level methods usually require more effort from programmers to implement communication among execution flows in applications. In addition, the cost of creating and destroying processes can make multiprocessing unsuitable for applications with large quantities of simultaneous execution flows or dynamic workloads that demand the creation and destruction of execution flows during runtime.

Scripting languages that include multiprocessing support include PHP, Python, Ruby, and Perl, all of which provide the standard **fork** function. Among its process control extensions, PHP includes at least three extensions that support multiprocessing: **PCNTL** (process control), **Semaphore** (semaphore, shared memory, and interprocess communication), and **Shared Memory**. **PCNTL** provides the **pcntl_fork** function; **Semaphore** provides semaphores, shared memory, regions, and message queues for interprocess communication and synchronization; and **Shared Memory** provides another implementation for shared-memory regions. Python supports pipes, sockets, and memory-mapped files for interprocess communication. Ruby supports pipes and sockets for interprocess communication and uses copy-on-write to optimize process creation. Perl 5 supports pipes, sockets, shared-memory regions, and message queues for interprocess communication. When the operating system doesn't have a fork system call, Perl 5 emulates the **fork** function by cloning the running interpreter and all its state to another system thread.

Apart from multiprocessing, many scripting languages also support preemptive multithreading. However, this doesn't necessarily mean they can use it for parallel execution. Preemptive multithreading support is also commonly combined with shared memory, so it inherits all the related complexities that make concurrent programming difficult.[3,5,6] The PHP, Python, and Ruby scripting languages include preemptive multithreading support. Perl 5 also includes preemptive multithreading support, but it uses its own model, called the *interpreter threads model*, which we describe in the next section.

PHP includes, among its process control extensions, a pthreads extension that provides an object-oriented API for userland multithreading. It supports mutual exclusion and conditional variables as well as Java-inspired synchronization constructs, such as synchronized code blocks and the wait and notify methods. The PHP manual warns users against sharing resources among contexts and states that restrictions and limitations are necessary to provide a stable environment when pthreads is used.

Apart from multiprocessing, many scripting languages also support preemptive multithreading.

The reference implementations for Python (CPython) and Ruby (MRI) support userland threads that are mapped to system threads. They include standard shared-memory synchronization constructs such as mutual exclusion (Ruby and Python), semaphores, and conditional variables (Python).

Despite their multithreading support, parallel execution in Python and Ruby is severely hampered because both include a Global Interpreter Lock (GIL). The GIL is a mutual exclusion lock used by the interpreter that only allows a single userland thread to execute at a time. It simplifies implementation, because all code is inherently

thread safe, but it can be a performance bottleneck, resulting in performance degradation when programmers use more threads to run a script. The only way to circumvent the GIL is to use alternative language implementations, such as Jython and IronPython (for Python) or JRuby and Rubinius (for Ruby). These alternative implementations, however, don't seem to have mainstream adoption.

Other than preemptive multithreading, some scripting languages support cooperative (or collaborative) multithreading. This form of multithreading isn't designed to support parallel execution, but rather to allow for explicitly coordinated threads that take turns to execute. The Ruby and Lua scripting languages support cooperative multithreading. Ruby supports fibers, which are lightweight userland threads scheduled cooperatively. Likewise, Lua supports coroutines, which follow the same paradigm as fibers. Neither fibers nor coroutines can be executed in parallel.

Emerging Concurrency Patterns

Although most scripting languages still rely on conventional concurrency support, some new patterns from compiled languages are finding their way into scripting languages as well. The first and most important pattern we've identified is *no-default sharing*. It implies no data is shared among execution flows unless explicitly requested by the programmer. This is a fundamental pattern because it serves as the basis for safe data sharing. When data isn't shared by default, the language can enforce a discipline for sharing, and language-based mechanisms can prevent data races. Moreover, explicit data sharing also helps programmers reason about program execution and consequently assists with testing and debugging. For example, Perl 5 doesn't share data by default. Instead, it supports multithreading through its `threads` module and provides interpreter-based threads (or ithreads), which are userland threads mapped 1:1 to system threads and scheduled preemptively.

> New patterns from compiled languages are finding their way into scripting languages.

A remarkable exception to conventional concurrency support is JavaScript. Albeit a popular scripting language, it doesn't support multiprocessing or traditional multithreading. Unlike the other scripting languages we've described here, JavaScript was developed for client-side scripting, which probably explains why it lacks such low-level concurrency support.

For each new thread, Perl creates a new interpreter instance and copies all the existing data from the current thread. This results in no data being shared among threads—that is, all variables are local to threads by default. It also results in a higher cost to create threads because all thread data must be copied.

Despite not sharing data by default, Perl 5 allows variables to be explicitly shared through the `threads::shared` module. This lets programmers create shared variables with a shared attribute or share function. The module supports sharing scalars, arrays, and hashes as well as references to these data types. Only scalars and references to shared variables can be assigned to shared arrays and hash elements.

Nevertheless, the only guarantee that Perl provides for shared variables is that their internal state won't become corrupt in case of conflicting accesses. Applications are still vulnerable to data races when multiple threads access shared data without proper synchronization. The `threads::shared` module provides two constructs for synchronization: locks and conditional variables. Locks are implemented with the lock function, which places an advisory lock on a shared variable. An advisory lock doesn't prevent other threads from accessing the shared variable unless they first try to lock the same variable, in which case their execution is suspended until the variable is unlocked. There is no unlock function; a locked variable is automatically unlocked when the lock goes out of scope—when the code block from where it was called finishes executing. The Perl 5 tutorial on threads cautions programmers against standard synchronization constructs (such as locks) and recommends instead the `Thread::Queue` module, which provides thread-safe queues and frees programmers from the complexities associated with the synchronization of shared data access.

A somewhat obvious approach to implementing the no-default sharing pattern is to share nothing at all. The Lua scripting language takes such an approach. For a C application to interact with Lua, it must first create a

Lua state, a lightweight data structure that defines the interpreter's state and keeps track of functions and global variables, among other interpreter-related information. States are independent—they share no data. The Lua C API allows C applications to create and manipulate multiple states at a time. In addition, when Lua code is loaded in a Lua state, its execution can be controlled just like a coroutine: it can be suspended, resumed, and yielded. The combination of multiple independent states and system threads allows for the implementation of Lua libraries for concurrent programming that support parallel execution.

Examples of such libraries include luaproc[9] and Lua Lanes.[10] The luaproc library implements Lua processes, which are lightweight execution flows of Lua code able to communicate only by message passing. Each Lua process runs in its own Lua state, and thus no data is shared by default. A set of workers—system threads implemented in C with pthreads—is responsible for executing Lua processes. There's no direct relation between workers and Lua processes; they're independent of each other (*N*:*M* mapping). Workers repeatedly take a Lua process from a ready queue and run it either to completion or until a potentially blocking operation is performed, in which case the corresponding process can be suspended and placed in a blocked queue. The only potentially blocking operations in luaproc are the standard message-passing primitives: send and receive.

Message addressing in luaproc relies on communication channels, and Lua processes have no unique identifiers. Each message carries a tuple of values with basic Lua data types. Lua processes can transmit more complex or structured data either by serializing it beforehand or encoding it as a string of Lua code that will later be executed by the receiving process.

Similar to luaproc, the Lua Lanes library uses multiple Lua states to host independent execution flows of Lua code (or lanes). However, unlike luaproc, in Lua Lanes each execution flow of Lua code is associated with a single system thread—there's a 1:1 mapping.

Communication among lanes is based on Linda[11]—that is, on tuple spaces. Each tuple in Lua Lanes must have a number, string, or Boolean as its key. Basic Lua data types, with the exception of coroutines, can be used as values in tuples. Two operation sets are available for lanes to interact with tuple spaces. The `send` and `receive` operations are analogous to the `out` and `in` operations in Linda. The `send` operation queues a tuple, and the `receive` operation consumes a tuple specified by a key. The `set` and `get` operations are used to access a tuple without queuing or consuming it. The `set` operation writes a value to a tuple specified by a key; it overwrites existing values and clears queued tuples with the same key. The `get` operation is analogous to the `rd` operation in Linda; it reads the value of a tuple specified by a key without consuming the tuple.

JavaScript also doesn't share data by default. However, JavaScript takes it a step further by combining it with another pattern: *data ownership*. The data ownership pattern prevents conflicting accesses by assigning an owner to each piece of data and ensuring that only the execution flow that owns the data may access it.

In JavaScript, data ownership is used in the communication among *Web workers*, which are currently defined in a draft HTML5 specification. They're intended to run scripts in the background while the main (user interface) thread responsible for handling visual elements and user interaction is executed. They can't directly interfere with the user interface because that could lead to race conditions with the user interface thread. For each new Web worker, a new JavaScript virtual machine (VM) instance running on a new system thread is created. This results in no data being shared among Web workers, but it also results in a higher cost to create them. Web workers communicate via message passing.

> The data ownership pattern prevents conflicting accesses by assigning an owner to each piece of data.

Web workers can exchange messages among themselves using message channels, and Web workers and their spawning scripts can simply post messages directly to each other. The default method for sending a message among Web workers is to use cloning (create a copy of the data in the message). Alternatively, instead of cloning, messages can be sent by transferring ownership (sometimes also referred to as transferable objects), or sending a reference to an object. This approach

avoids the overhead necessary to copy objects. Because data can't be owned by more than one execution flow at a time, the sending Web worker loses access to the data once it transfers its ownership and any access attempt results in an exception.

Web workers in JavaScript allow background tasks to perform computations, possibly in parallel. A similar concept, which involves coordinating asynchronous computations that can potentially be executed in parallel, is the basis for our next pattern: *futures*. A future works like a read-only view for the resulting value of a computation that will be executed at some point during a program's execution.[12] Futures are generally used for asynchronous, nonblocking, parallel computations. Implementations usually allow execution flows to check whether a future's computation is complete or wait for its completion to retrieve its value; sometimes they also allow the use of callbacks to avoid blocking while waiting for a future to complete.

> Immutability is a simple, yet effective way to safely share data.

The terms future and promise are sometimes used interchangeably. In fact, a *promise* works like a single-assignment container that completes a future—it sets a future's value. Promises are generally used to deliver a value from one execution flow to another. Neither futures nor promises are directly related to data sharing because a language could support either of them and still allow for unsafe data sharing. Still, futures are a relevant emerging pattern because they structure a mechanism for asynchronous parallel tasks.

Futures are explored in Deterministic Parallel Ruby (DPR),[13] a Ruby library that provides parallel constructs with a focus on ensuring determinism. These constructs include *isolated futures*, which are futures with a deterministic behavior. To ensure determinism, DPR requires that isolated futures represent pure functions (functions that always generate the same output when given the same input and have no observable side effects) and that the arguments to these functions are passed by deep copy to prevent concurrent changes.

Python and Clojure also support the futures pattern. Python supports futures with its **concurrent.futures** module. It allows the asynchronous execution of callables using either a pool of threads or a pool of processes. A dialect of Lisp, Clojure runs on a Java VM and builds on the Java concurrency support to offer a robust infrastructure for multithreaded programming. Although Clojure is a compiled language, we chose to consider it in our study because it has features commonly found in scripting languages and was designed to support concurrency. Clojure supports both futures and promises and uses an API similar to the one included in Java's **java.util.concurrent** package. However, Clojure is better known for its strong support of *data immutability*, the last pattern that we discuss in this section.

Immutability is a simple, yet effective way to safely share data. No synchronization is needed to access immutable objects, which makes them inherently thread-safe. Also, it's easier to ensure object consistency when working with immutability because it essentially only admits a single state after initialization. The main disadvantage of immutability is the need to create a new object each time an existing immutable object must be altered, which can be a performance bottleneck.

Clojure is inspired by functional programming languages and, as such, provides strong support for immutability. It offers a range of immutable persistent data structures (lists, vectors, sets, and maps). Only references mutate in Clojure, and they do so in a controlled way. Clojure supports four different types of mutable references to its immutable data structures: **Vars**, **Refs**, **Agents**, and **Atoms**. They differ in how they're changed. Changes to **Vars** are isolated on a per-thread basis, whereas changes to **Refs** must be executed within transactions using Clojure's Software Transactional Memory (STM) system. **Refs** allow for synchronous, coordinated state changes. **Agents** must receive functions (called actions in Clojure), with message-passing style semantics, to perform changes. **Actions** are executed asynchronously and can alter an agent's state. Each agent works by sequencing operations to a data structure instance. Agents allow for asynchronous, independent state changes. Finally, changes to **Atoms** must be executed by using simple atomic operations (such as swap and compare-and-set). **Atoms**

allow for synchronous, independent state changes.

Immutability is also explored in JavaScript by the River Trail programming API.[14] River Trail introduces a new parallel array data type (**ParallelArray**) with a set of methods that define array-processing patterns (**map**, **combine**, **reduce**, **scan**, **filter**, and **scatter**). Operations on individual elements in a **ParallelArray** can be executed in parallel by specifying a processing pattern and an elemental function, which defines the operation to be performed. To increment by one all elements of a **ParallelArray** in parallel, for instance, a programmer could use the map-processing pattern and an elemental function that receives a single array element and returns its value incremented by one. Elemental functions can't communicate directly with each other, but they have read-only access to the global state. When elemental functions are executing, the parent execution flow is suspended and thus it doesn't change its own (global) state. Because neither the parent execution flow nor child execution flows spawned to run elemental functions change the global state while elemental functions are running, River Trail calls this approach *temporal immutability*.

Toward Safe Data Sharing

Although emerging concurrency patterns are finding their way into scripting languages, there's still room for improvement. As we've briefly discussed, even scripting languages that embrace emerging patterns have their limitations. Recent research results such as River Trail (for JavaScript) and DPR confirm that promoting safe data sharing in scripting languages is a pressing matter.

Not sharing data by default is the first and most important step that programming languages must take to allow for safe data sharing. However, entirely disallowing shared data isn't always a viable approach because, in many cases, the performance overhead of cloning data among execution flows is unacceptable. Instead, proper language-level mechanisms should prevent conflicting concurrent accesses to shared data. In fact, enforcing disciplined data sharing[8] is critical to making it easier for programmers to write correct parallel applications and is complementary to no-default data sharing.

Even scripting languages that embrace emerging patterns have their limitations.

Although not sharing data by default diminishes the chances of programmers introducing data races in applications, it doesn't eliminate them. Take, for instance, Perl 5: by default, data is local to each thread, but programmers may still declare shared data and access it without proper synchronization. We need mechanisms that ensure that all accesses to shared data are safe.

One way to make data sharing explicit that is consistent with the dynamic nature of scripting languages is to combine data sharing with message passing, as JavaScript does. More than simply using messages to transfer data, JavaScript uses it to transfer ownership rights; Web workers can transfer the ownership of a shared object, via message passing, to allow another Web worker to write to that object. This works well for read-write accesses, but it isn't suitable for the common case of read-only access. To allow multiple Web workers to read a shared object, a programmer must either sequentially transfer the object's ownership, precluding parallelism, or clone the object multiple times, impacting performance.

A convenient way to deal with this limitation is to use immutability. When used by itself, immutability is typically required for all shared objects. However, we can combine it with the notion of ownership rights to improve flexibility. We can express this combination by using dynamically assigned capabilities.

Two basic capabilities can be used to express the set of rights that an execution flow can have over shared objects: *read-only* and *read-write*. The lack of capabilities means the execution flow has no access to an object. When a new, explicitly shareable object is created, the creating execution flow should have the read-write capability for that object by default. The execution flow can then share the object by sending a reference to another execution flow via message passing. When it does so, it must choose whether the reference to the object should be shared as read-write (mutable) or read-only (immutable).

If a reference is shared as read-write, the sending execution flow must lose access to the object, and

ABOUT THE AUTHORS

ALEXANDRE SKYRME is a doctoral student at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). His research interests include concurrency, parallelism, distributed computing, computer networks, and information security. Skyrme has an MSc in computer science from PUC-Rio. Contact him at askyrme@inf.puc-rio.br.

NOEMI RODRIGUEZ is an associate professor of computer science at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). Her research interests include concurrent and distributed programming. Rodriguez has a PhD in computer science from PUC-Rio. Contact her at noemi@inf.puc-rio.br.

ROBERTO IERUSALIMSCHY is an associate professor of computer science at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). His research interests include programming-language design and implementation. Ierusalimschy has a PhD in computer science from PUC-Rio. Contact him at roberto@inf.puc-rio.br.

the receiving execution should obtain the read-write capability for the object. This behavior is similar to transferring ownership of a JavaScript object between Web workers. When a reference is shared as read-only, the sending execution flow must lose the read-write capability, which becomes a read capability, and the receiving execution flow should receive the read capability to the same object. This allows multiple, safe, parallel accesses to the object. Once a reference is shared as read-only (once it becomes immutable), it can never become read-write (mutable) again, but it can be freely shared as read-only among multiple execution flows.

The combination of message passing with the emerging patterns of no-default data sharing, data ownership (expressed by means of capabilities), and immutability can allow for safe data sharing in scripting languages. It also provides a solid foundation for implementing higher-level mechanisms such as futures and data-parallelism constructs. Message passing simplifies reasoning about the interaction among execution flow and data sharing because the **send** and **receive** operations are explicit in the source code. Using messages to transfer ownership rights aligns with the dynamic nature of scripting languages because access control is performed during runtime. No-default data sharing ensures that data must be explicitly shared and allows for the enforcement of disciplined sharing. Immutability allows for safe, parallel, read-only access to shared data.

Several of the scripting languages we discussed here rely on conventional shared-memory preemptive multithreading, leaving the programmer to deal with the well-known complexity of data races. In addition, several languages don't allow programmers to explore the benefits of multicore architectures because they rely on a Global Interpreter Lock to guarantee that the interpreter's internal state isn't corrupted. However, we've identified two patterns that appear in one or more scripting languages that do allow programmers to explore multiprocessor architectures using shared data in a more disciplined way: no-default sharing and data immutability. Understanding the concepts and ideas behind the different mechanisms is an important tool for extracting the ideas that we can incorporate to make concurrency easier on the programmer.

A key point to reiterate is that no data should be shared by default, which allows disciplined data sharing to be enforced. Message-passing semantics can help programmers reason about execution flow interaction and data sharing. Capabilities that somehow extend the idea of data ownership can help control access to shared objects by associating different sets of rights to object references. Programmers can then use read-only (immutable) and read-write (mutable) capabilities to express these rights.

This proposal could be directly incorporated into a scripting language so that the language guarantees that

all communication between execution flows respects these capabilities. However, we believe that it's best to keep programming language cores as small as possible, incorporating only basic mechanisms upon which we can build different concurrency models. Future research should identify which mechanisms are needed at the core of a scripting language for it to support the construction of libraries or modules with guarantees against unsafe data sharing.

References

1. J. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *Computer*, vol. 31, no. 3, 2008, pp. 23–30.
2. L. Prechelt, "Are Scripting Languages Any Good? A Validation of Perl, Python, Rexx, and Tcl Against C, C++, and Java," *Advances in Computers*, vol. 57, 2003, pp. 205–270.
3. H. Sutter and J. Larus, "Software and the Concurrency Revolution," *ACM Queue*, vol. 3, Sept. 2005, pp. 54–62.
4. UBM TechWeb, "The Parallel Programming Landscape: Multicore Has Gone Mainstream—But Are Developers Ready?" survey sponsored by Intel and conducted with *Dr. Dobb's* readers, 2012; https://software.intel.com/sites/billboard/sites/default/files/PDFs/TW_1111059_StOfParallelProg_v6.pdf.
5. E.A. Lee, *Disciplined Message Passing*, tech. report, EECS Dept. Univ. California Berkeley, Jan. 2009.
6. J. Ousterhout, "Why Threads Are a Bad Idea (for Most Purposes)," presentation, 1996 Usenix Ann. Tech. Conf., Jan. 1996; www.stanford.edu/~ouster/cgi-bin/papers/threads.pdf.
7. H-J. Boehm, "Position Paper: Nondeterminism Is Unavoidable, But Data Races Are Pure Evil," *Proc. 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability* (RACES 12), 2012, pp. 9–14.
8. H.-J. Boehm and S.V. Adve, "You Don't Know Jack about Shared Variables or Memory Models," *Comm. ACM*, vol. 55, no. 2, 2012, pp. 48–54.
9. A. Skyrme, N. Rodriguez, and R. Ierusalimschy, "Exploring Lua for Concurrent Programming," *J. Universal Computer Science*, vol. 14, no. 21, 2008, pp. 3556–3572.
10. A. Kauppi, "Lua Lanes—Multithreading in Lua," 2009; http://kotisivu.dnainternet.net/askok/bin/lanes/.
11. D. Gelernter and N. Carriero, "Coordination Languages and their Significance," *Comm. ACM*, vol. 35, 1992, pp. 97–107.
12. H.C. Baker, Jr. and C. Hewitt, "The Incremental Garbage Collection of Processes," *Proc. 1977 Symp. Artificial Intelligence and Programming Languages*, 1977, pp. 55–59.
13. L. Lu, W. Ji, and M.L. Scott, "Dynamic Enforcement of Determinism in a Parallel Scripting Language," *Proc. ACM SIGPLAN 2014 Conf. Programming Language Design and Implementation* (PLDI 14), 2014, pp. 519–529.
14. S. Herhut et al., "Parallel Programming for the Web," *Proc. 4th Usenix Workshop on Hot Topics in Parallelism*, 2012; https://www.usenix.org/system/files/conference/hotpar12/hotpar12-final39.pdf.