High Level Languages: A Better Fit for Parallel and Distributed Programming

Christopher Esterhuyse* Dalia Papuc* Zakarias Nordfäldt–Laws*
* *Vrije Universiteit Amsterdam*

Abstract

Why stick to primitives developed decades ago when building complex applications on modern parallel and distributed systems? Our work demonstrates how higher-level tools are a better fit for this purpose. The programming languages[1] Reo, Chapel and Linda are examples that manage to compete with widely used message-passing approaches such as MPI. We identify four effective requirements for languages intended for this purpose: reliability, productivity, expressivity and performance. This work examines case studies of novel, applicable programming languages, and discusses how they satisfy the requirements and which trade-offs are still necessary. We attempt to break the *illusion* that alternative programming languages and tools are less performant than established message passing frameworks.

1 Introduction

Sequential programming has changed significantly in the last twenty years. Even conventionally low-level, imperative languages known for performance (C++, for example) have been adopting more expressive, high-level constructs. Maps, iterators and anonymous closures are commonplace, often integrated into the language's syntax and well-supported by compiler optimization [13]. These features are favourites among programmers, allowing them to avoid error-prone repetition and express high-level concepts concisely. Due to their ubiquity, programmers are usually familiar with these tools by the time they graduate from university [11].

In contrast, high-performance parallel and distributed programming use MPI and other granular message-passing frameworks as the de-facto standard [20]. By working on this sub-field, programmers have no choice but to step decades into the past, implementing complex communication protocols in terms of low-level concurrency primitives, such as locks [11]. This work is exceptionally error-prone for even the most well-educated experts [7]. Programmers must manually ensure that their protocol, distributed all over the source files, solves the intended problem as expected [7]. Historically, applicable high-level programming tools were unable to compete in terms of performance [9].

In recent years, academia has been alive with work exploring new options for feature-rich novelties. Sequential languages serve as inspiration, offering high-level expressivity largely by developing powerful compilers and languages rich enough to facilitate them [23]. These languages bring with them new, desirable features such as verifiability and safety. As work continues, the shortfall in performance is shrinking, and in some cases we have found benchmarks rivaling that of established low-level message-passing approaches (such as MPI). We suggest that both academia and the industry should explore these new alternatives, as it is likely that a better fit for the programmer's use case already exists.

In our work, we identify four necessary requirements for a parallel and distributed programming language; namely, such languages should enhance the programs' **reliability**, i.e. result in verifiably-correct implementations, enhance the programmers' **productivity** through high-level constructs, be rich enough to **express** real world problems, and achieve competitive **performance**. We present a variety of languages, such as Reo, Chapel, Linda, which exemplify these features and our approach to parallel and distributed programming.

The rest of the paper is structured as follows: Section 2 explains concepts relevant to this work. Section 3 defines our set of desirable language requirements. Section 4 explores existing languages that exhibit these requirements. Section 5 discusses trade-offs that these languages make in exchange for their features, and what steps are taken (or could be taken in the future) to minimize the cost. Finally, Section 6 concludes our paper.

[1] This work often uses the terms 'language', 'library', 'tool', etc. interchangeably as this detail is irrelevant to our argument.

2 Background

This section introduces concepts and terminology relevant for the majority of the sections to follow. In addition to theoretical concepts, the languages **Reo** and **Linda** are outlined and they differ significantly from most general-purpose programming languages the reader may be most familiar with.

2.1 Scaling beyond Moore's Law

As Moore's law is grinding to a halt, we are forced to move beyond single-computer architectures and develop applications on parallel and distributed systems for greater speedup. This paradigm shift brings new complications to the table such as verifiability of distributed systems, high- and low-level synchronization and correctness. Amdahl's law seeks to represent the scalability of a system by analyzing its speedup capabilities as a function of its non-parallelizable bottleneck. Gustafson's law redefines scalability when problems are allowed to scale up with the infrastructure [22]. In practice, scalability is hard to achieve, as adding concurrency to a program requires extra consideration of memory access and synchronization patterns. Message passing frameworks such as **MPI** have been dominating this new era of computing [20]. As argued in Section 1, programmers are required to use primitives developed decades ago, as well as understand the complex execution structure of such a massive system. The new age of computing requires programming languages to facilitate these complex programs.

2.2 Declarative Languages

Declarative programming represents a paradigm of expressing computational results without specifying control flow. This approach is relevant for its association with highly concise high-level code as well as lending itself well to formal verification. While theoretical computer scientists are most familiar with these side-effect-free approaches, as they adhere more closely to the logical models with which they work, even fundamentally-imperative languages like C usually rely on declarative auxiliary tools such as **Make** in their everyday workflows [34].

Listing 1 shows a code snippet written in the declarative programming language **Prolog**, along with an explanation, showcasing the fact-based structure of the paradigm.

```
good(whiskas).
good(milk).
hungry(alice).
likes(alice, milk).
likes(alice, whiskas).
eat(X, Y) :- hungry(X), likes(X,Y), good(Y).
```

Listing 1: Example Prolog program that declares the relationships between abstract variables and constants. When queried with `?- eat(X,Y).`, Prolog checks whether there is a substitution for which the expression is `True`.

2.3 Coordination Languages

Coordination languages facilitate the cooperation of several components, concurrently working on a task. Often, they manifest as part of a larger tool-chain that offers a high-level means of communication between seemingly-isolated computation 'components' [5]. Typically, the coordination code is compiled down to the target language, generating the needed low-level *glue code* automatically. Here, computation and coordination code are entirely entwined.

Endogenous coordination languages are characterized by facilitating communication by injecting simple primitives into the target language that the programmer can weave into their code, hence the name meaning 'of the inside' [5]. **Linda** is an example of such a language, providing an abstract notion of some shared-memory data structure called the 'tuple-space' which can store and provide tuples of data [4, 10]. Linda can thus 'lift' sequential code to a component in a concurrent program by providing stub functions that allow it to read or write to the tuple-space. Upon compilation, tuple operations are translated to message-passing primitives. The tuple-space itself is translated to a potentially-distributed memory store across physical nodes [10].

Exogenous communication languages prefer to isolate the coordination and computation components from one another, allowing some view 'of the outside' [4]. This results in a clear place to find the implementation *of the protocol*, while computation code sees only communication endpoints, but not how and to whom they communicate. **Reo** is such a language, which allows a programmer to translate their protocol to a high-level hypergraph[2] representation that represents channels or relations between abstract graph nodes [33]. Reo focuses on *interactions* rather than *actions*, coordinating the activity of computational entities such as sequential code, passive or active objects, threads, processes or software components. These entities communicate though con-

[2]Hypergraphs generalize graphs, allowing edges that connect an arbitrary number of vertices.

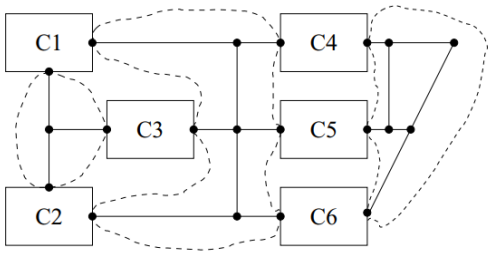


Figure 1: A system containing compute components $\{C1, C2, ..., C6\}$ with three complex connectors. Each connector is composed of multiple channels, each channel with its own semantics. Image from [6].

nectors. Complex connectors are formed of simpler connectors, ultimately composed of channels [6]. A channel's end-nodes can be either of type 'sink' (which supply data from a channel), or 'source' (which accept data into a channel) [7].

A channel describes its behaviour via constraints or relations on the flow of data at its ends (giving the language its name, named after the word 'flow' in Greek) [6, 33]. There is liberty in defining the behaviour of a channel as well as the possibility to use multiple types together.

As desired, components can be plugged into the positions of some of these graph nodes. Figure 1 illustrates the relationship between connectors, nodes and components. At compile time, the entirety of the system once again compiles down to granular message-passing between these sequential processes in the target language. At time of writing, Reo can target C, Java, C++ and Fortran [4, 26].

3 Effective Language Requirements

The community sentiment is that the programmability of distributed and parallel systems is deficient [5, 7, 11, 31]. To establish a different perspective, we define requirements we value in a language for well suiting distributed and parallel systems, which allow the description of complex real-world problems. Development of large, reliable applications requires a clear, expressive language. Such systems are more effective when they utilize all the available resources.

3.1 Reliability

We all desire our written programs to solve the intended problem without any bugs. This requires that the programming language and the constructs that compose it facilitate this reliability. M. Lohstroh [33] claims protocols are hard to understand and that proving their correctness is challenging. Formal verification techniques can check whether the specified requirements are met. A protocol may satisfy safety properties ('something bad will never happen'), or liveness properties ('something good will eventually happen'). Even if a protocol is proven to behave as expected, formal methods can not guarantee for this holds once implemented [33]. The task of proving correctness becomes even more difficult if the utilized tools do not contain the required constructs for isolating protocol code.

"In the absence of proper structures to enforce (or at least encourage) good protocol programming practices, programmers frequently succumb to the temptation of not isolating protocol code." – Jongmans et al. [27]. We need a reliable way to ensure that the implementation phase does not modify the correctness of a protocol. This necessitates models which facilitate building correct programs, free of bugs. Smart compilers may understand whether bugs occur and prevent them from unintentionally passing through the compilation phase. Alternatively, the language constructs can lead the programmer to approach the problem in a way that avoids the introduction of errors at all.

3.2 Productivity

Parallel and distributed programming is unlike sequential programming; developers must express complex notions of parallelism, data or task distribution, communication and synchronization. These processes involve a lot of moving parts and can result in highly variable outcomes depending on how computation steps are ultimately interleaved. As such, this work can easily overwhelm a programmer. We wish to provide abstractions for such cases to make these tasks as easy as possible to facilitate productivity [11].

E. A. Lee [31] and M. Lohstroh [33] consider concurrent programs extremely difficult to be understood by humans due to their nontransparent, non-deterministic behaviour; fundamentally, concurrent programs can result in more possible outcomes owing to the additional complexity introduced by arbitrary *interleavings* of actions. Larger programs result in more complex code; in the concurrent and distributed context, this problem is amplified. B. L. Chamberlain et. al. [11] state that the gap between the developers that can effectively program parallel machines and sequential ones enlarges with the time. To combat this growing divide, programmers need all the help they can get, necessitating a more natural, intuitive way of understanding their programs. A language facilitating clearer, more abstract code also achieves productivity in another sense: non-technical people can reason about the implementation.

In Subsection 3.1, we asserted that protocols are difficult to understand and check for correctness. As fur-

ther transformations of these protocols during implementation only make matters worse, we suggest a need to reduce these transformations by preferring higher-level code, closer to the original protocol.

3.3 Expressivity

A necessary requirement for solving real-world problems is the ability to effectively –and with minimal friction– express them using the constructs provided by a language. In case a language alone does not provide enough powerful constructs to illustrate a given problem, it must inter-operate with libraries or allow for a language extension for a collective effort in illustrating the problem.

Furthermore, considering the already existing systems, migrating to a completely new environment is not always possible as it requires resources and willingness to do so. Many components have been developed without considering their integration in a medium where they must communicate and collaborate with other modules. Thus, the designed model must be able to accommodate such components [8].

3.4 Performance

We wish the performance of our systems to be comparable to the performance achievable with the available languages. A language that systemically inhibits the achievable performance of a distributed system cannot always compensate by providing other desirable features such as reliability. As such, we require that performance is not lost in the attempt to satisfy other requirements.

In the long term, we believe higher performance is naturally expected as a large user community brings more support, contribution and improvement to a language. In the short term, a solution is to offer features at multiple levels, higher-level features for easy programming but not a guaranteed excellent performance and lower-level features for accommodating parts of code which wish to be optimized for performance [11].

4 Satisfying Requirements

Significant progress has been made in the development of productive languages for concurrent computing. This section explores the desirable language requirements defined in Section 3; examples of available languages are provided, along with insight into how they accomplish these requirements and how they inter-relate.

4.1 Satisfying Reliability

For a program to be reliable, it should perform its task consistently, as expected. Languages need to facilitate their programmers writing the code they intend to write. This simple goal has implications for a language's comprehensibility both for compilers (so that correctness can be definitively verified or errors identified) and humans (so that programs are written correctly in the first place).

Reo, **Manifold**, along with other exogenous communication languages provide this property in a way most languages do not: protocol code is located in one place [7]. This isolation succeeds in more effectively capturing and preserving the *intent* the programmer has in mind. Focusing on this idea of intent makes the mapping from the abstract algorithm to the Reo implementation easier and require less decision-making for the developer; additionally, it allows the problem to be expressed more coarsely, granting the compiler more freedom to optimize the details of the resulting binary. For example, Reo is able to perform *protocol* optimizations such as merging distinct message channels onto one physical link [4]. Traditionally, *actions* are first-class primitives and *interactions* are derived concepts i.e. a message emerges from a *send* and *receive*; this spreads the interaction all over the source code. Reo leans on the notion of upgrading *interactions* to a first-class primitives instead [6].

Rust is an imperative systems programming language with neither explicit memory management, nor a garbage collector. Its 'ownership system' implements affine types, which requires that all data is associated with precisely one 'owner' at a time, but may be 'borrowed' (passed by reference) temporarily [9]. These borrows are gouverned by rules akin to those for the *readers-writer lock*[3], requiring that no value is ever aliased and mutable at the same time. An example of the relationship between the ownership system and Rust syntax is given in Listing 2. This system prevents a program with dangling pointers or race conditions from compiling [39]. Along with (optional) run-time array bounds-checking, this provides memory safety [28, 39].

Session types are another paradigm for guaranteeing adherence to a communication protocol, often possible even at compile-time. In a nutshell, session types allow the programmer to express the type of a *session* (representing a communication link) in a type system similar to that typically used for data types. Participants in the protocol can then only participate if the data they pass in or out is the correct type. Multi-party session types extend this notion to links with an arbitrary number of participants [24]. Implementations for the various session type schemes either exist already or have been defined for languages Scribble, Rust, C, Erlang, Go, Java, Python, Scala and more [25, 41]. Its wide adoption can be largely attributed the broad applicability of the concept. They can

[3]The readers-writer lock is a concurrency mechanism for allowing multiple references without data races.

```
fn foo(x: Data) { // foo consumes given x
    if bar() {
        let y = &mut x; // x aliased mutibly
    } // y goes out of scope
    baz(&x, &x); // x aliased with read-only refs
    // x references out of scope
} // x is dropped
```

Listing 2: Illustration of the Rust ownership system (implementing affine types). References alias values with pointers until the references go out of scope. A mutable reference may not coexist with any other reference. Owned values that go out of scope are dropped implicitly.

```
config const m = 1000, alpha = 3.0;

const ProblemSpace = {0..m+1} dmapped Block(..), //
 ↪  allow algorithm to run on multiple locales. If
 ↪  unspecified, the domain maps on one locale only
     SubProblemSpace: subdomain(ProblemSpace) =
      ↪  {1..m};

var A, Temp: [ProblemSpace] real; // array of size
 ↪  ProblemSpace with elements of type real

//forall = concurrent alternative of for loop
A = alpha * A; // = forall a in A do alpha * a
[i in SubProblemSpace] Temp[i] = (A[i-1] + A[i+1])/2;
 ↪  // = forall over SubProblemSpace indices
```

Listing 3: Illustration of Chapel global view syntax. The code runs on multiple locales as the domain map was explicitly specified. The last two lines of code imply concurrent execution, each locale executes the code in parallel.

be implemented via libraries or natively, and checked statically or dynamically, etc. Consequently, languages without native support can be augmented by the addition of session types in a way that best suits the language.

Not only do we need the tools to create a reliable system, but we also need to be able to verify that it adheres to the protocol. This requires a good understanding of the execution semantics of our system. Analyzing models of the system's behaviour provides this understanding in a way that can be reproduced and easily communicated to others. However, modeling a complex concurrent system can be a challenging task, as was found by numerous sources [3, 16, 17]. The nature of exogenous coordination languages such as **Manifold** and **Reo** lends themselves well to graphical representations, making visual modelling of complex behavioral protocols a much easier task. For example, the authors of [29] present a tool for converting Reo into mCRL2 [19]. mCRL2 is a specification language that can be used to specify and analyze the behavior of distributed systems in Eclipse [4]. Thanks to the user-friendly environment of mCRL2 in the Eclipse Coordination Tools (ECT), the implemented protocols can be visualized, verified and checked for correctness. In addition to validation, model-based checking easily maps to declarative models, reducing the gap between specification and implementation. This benefits developers as it is easier to maintain an overview of the implementation, as well as other non-technical stakeholders as they can more easily understand it.

4.2 Satisfying Productivity

Programming in a distributed setting is very challenging. It requires programmers to make the considerations inherent to sequential programming, with the added challenges of correct concurrency on top. Overcoming this difficulty requires the language to add minimal friction or risk overwhelming a programmer, leaving them unproductive [31].

Transactional Memory was introduced in the early 1990s to facilitate abstracting away from the tedious task of manually achieving mutual exclusion. Instead of locks, transactions allow multiple threads to initiate transactions in memory concurrently, keeping track of all read and write operations performed in the *critical section*. Each thread *commits* its changes when done, after which the transaction is irreversible. A check is first performed to determine if an involved memory location was modified by two threads; if so, the thread *rolls back* the transaction, giving another thread an opportunity to commit [21]. Transactional memory is still an active area of research. However, the high cost of rollbacks inhibits its adoption as the primary solution to highly-concurrent applications [30]. Nevertheless, this type of abstraction was greatly appreciated by the development community, which still today utilizes transactional memory for specific purposes. We are still today working with the primitives that transactional memory tried to abstract away from, reflecting the necessity of another replacing language, tool or construct.

Where applicable, nothing achieves conciseness quite like a good abstraction. **Linda** manages to frame all communication operations in terms of four primitive actions on tuples, all variants of getting and setting tuples to and from a single persistent tuple store in the centralized, virtual shared memory 'tuple space' [10]. During development, Linda developers need only think in terms of these ubiquitous tuple operations, making it very easy to understand how the problem is being solved. It isn't until performance optimization becomes the priority that programmers have to look past the tuple abstraction. Even

[4] The most widely-used Java integrated development environment (IDE).

then, the compiler will do all the heavy lifting, distributing tuples between machines and optimizing ordering according to how they are accessed.

Chapel is a language that attempts to provide the benefits of a high-level global view approach to distributed programming (comparable to an exogenous coordination language) as well as the performance that sometimes requires fine control. At the higher levels, programs look more like pragma-augmented sequential code (familiar to developers that have worked with an OpenMP library), with specialized concurrent loops containing the data-parallel sections of code. An example of high-level global view Chapel can be seen in Listing 3. Whenever the higher levels differ from the programmer's needs in terms of data distribution, memory reuse etc., programmers can 'dive down' to a lower layer, altering data distribution patterns or employing task parallelism [11]. This approach ensures Chapel code is only ever as complex as it needs to be, and granular details are present only because they represent deviations from the norm.

Rust offers a similar duality with 'unsafe' code blocks. By marking a block with an explicit keyword, some operations that are not checked by Rust's ownership system are permitted [9]. This allows a programmer to explicitly manipulate raw pointers or statically type-cast values [39]. The Rust community encourages the use of 'pure safe rust', with popular libraries touting this safety in their doc pages. Unsafe Rust is used for cases where the compiler cannot be convinced of safety without turning to a sub-optimal formulation. Consider, for example, *hogwild*, where threads write to a shared variable without locking, as the algorithm converges correctly regardless of arbitrary overwrites [38]. The ownership system would prohibit these unsafe writes, as correctness cannot be inferred by the compiler [35].

4.3 Satisfying Expressivity

The architectural expressiveness of **Reo** make it well-suited for our purpose. The communication topology of components can be naturally established and described [33] making Reo accessible to developers. Figure 2 shows an example of an expressive communication model, a point-to-point model. Modifying one of the components make it easy to reason on how it may impact the other components [6].

A study on the expressive power of Reo and Linda shows that the channel-based coordination model is surely more expressive as it allows mixing synchronous and asynchronous behaviour [4]. For the same reason, Reo is considered more expressive than Manifold which only allows asynchronous behaviour [33]. Furthermore, Reo is more expressive than dataflow models, Kahn networks, synchronous languages, stream processing languages, workflow models and Petri nets [7].

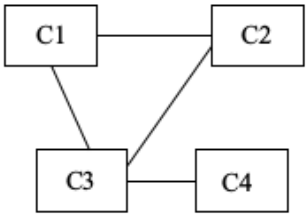


Figure 2: Architectural expressiveness. Communication pattern of a peer-to-peer model. The effect of modifying or replacing a component can be seen from this model, namely which other components will be affected. Image from [6].

Another prime example of an expressive language is **Chapel**. The global view programming model together with Chapel's language constructs for achieving data and task parallelism as described in Subsection 4.2 represent principles that allow meaningful expression of real-world problems.

Additionally, Chapel wishes to achieve interoperability with other languages such as C, to accommodate existing components [11]. The support for calling between Chapel and C was already accomplished. Chapel is able to use external declaration of C concepts, i.e. functions, variables and types [1]. At the time of writing, the interface with C++, Java, Python, Fortran is in the process of being achieved through Babel, a high-performance language interoperability tool. Babel 2.0 includes an experimental version of Braid, which is an ongoing effort to support, among other languages, Chapel [2].

4.4 Satisfying Performance

The ownership system of the **Rust** programming language, first outlined in Section 4.1 and exemplified in Listing 2, is a purely compile-time construct. Rust is intended as a systems programming language; this incentivizes strong memory safety, while also necessitating competitive performance. While safe, the use of a garbage collector is prohibitively expensive. Instead, a Rust binary looks much like one created by a C compiler, with explicit allocation and de-allocation. These calls are inserted by the compiler at positions implicit by the structure of the source code [9]. Further language-integrations such as string slices allow generically-written iterator code without incurring runtime cost [32, 36]. This feature along with powerful and safe abstract macros[5] suit Rust well for the application of

[5]Macros in Rust are not based on string preprocessing. Rather, macros define operations on tokens in subtrees of the abstract syntax tree at compile time.

```
fn dot(a: &[i32], b: &[i32], n: i32) -> i32 {
    let mut sum = 0;
    for i in range(0, n) {
        sum += a(i) * b(i);
    }
    sum
}

fn main() -> i32 {
    let (a, b) = /* (omitted) */;
    let c = @dot(a, b, 4); // @ marks partial eval.

    /*  This generates:
    let mut c = 0;
    c += a(0) * b(0);
    c += a(1) * b(1);
    c += a(2) * b(2);
    c += a(3) * b(3);
    */
}
```

Listing 4: Example of an annotation in Impala resulting in partial evaluation of a function. Combined with normal compiler optimizations this results efficient GPU code. Source: `https://anydsl.github.io/Impala`.

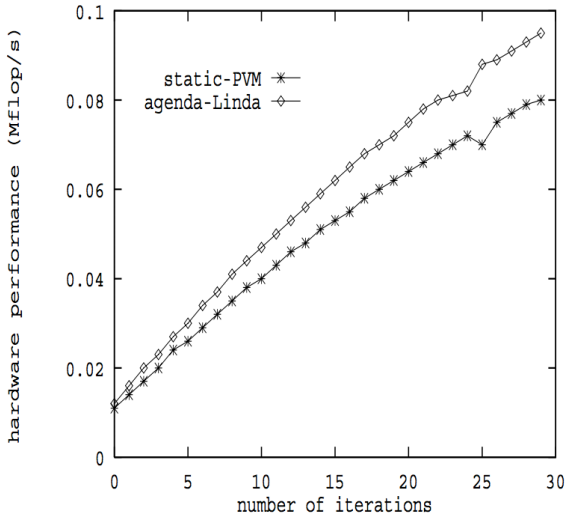


Figure 3: Hardware performance of Linda and PVM applications for increasing number of time steps in the TDSL phase, using a 9-node multi-computer. Image from [40].

writing combinatory parsers for things like VLC Media Player's demuxers [12, 14].

Impala is a domain-specific dialect of Rust that seeks to take all of the syntax and much of the foundations of Rust to GPU programming. As with Rust, Impala leverages a novel, semantic-rich language extension to allow the compiler to optimize code extensively. Impala provides pragma[6]-like annotations which allow a compiler to evaluate some statements at compile-time, exploding control structures ahead of time; an example is provided in Listing 4. Other instances generate code that exhibits 'continuation-passing style', chaining functions together by passing subsequent code into functions as call arguments. These optimizations are exploited extensively by various GPU architectures, allowing further coalescing of task granularity for GPU threads. These features allow Impala to compile down to OpenGL and CUDA targets, making it portable while performing competitively [23].

DREAM is an aerodynamics application used to simulate the behaviour of fluids around moving bodies. During the development, the authors report an extensive comparison between the message passing framework, PVM, and **Linda** [18, 40]. The evaluation is made on a cluster and both *hardware performance* and *elapsed time* are taken into consideration. The authors report consistent results of Linda outperforming PVM when scaling up the number of nodes and/or workload. This is mainly due to Linda's virtual shared memory system, offering automatic *load balancing* capabilities in the tuple-space. This does not only allow for better scaling, but also requires less effort when developing. The results from the paper when running 9 compute nodes are visualized in Figure 3.

Linda outperforms PVM in scalability and execution speed. Furthermore, the authors stress that Linda was especially useful as 'rapid application development platform', significantly decreasing the development time in comparison to PVM. Thanks to Linda's structure, the authors could perform different experiments by altering the coordination, a task which would take much more time in message passing framework due to its fine-grained structure. This case study shows how an abstract coordination language can be well-suited for the development of a complex distributed system containing many different tasks. Stitching components together is typically easier in a language that decouples process coordination and computation, since this provides a more easily maintained overview and more loosely coupled components. This holds also when using different programming languages [4]. As a result of their experience, the authors chose to continue to develop their system DREAM, with the tools offered by the Linda programming environment.

Despite its high-level approach, **Reo** has been shown to be capable of competing with hand-optimized C code. Reo is able to take the high-level protocol definition and apply *protocol-level optimization* (deduplicating communication links, etc.) during compilation. Even more importantly, Reo is capable of leveraging the *Proto-*

[6]Pragmas are instructions in code that typically involve *meta*-programming on other code constructs rather than mapping to anything in the binary; For example, inlining functions.

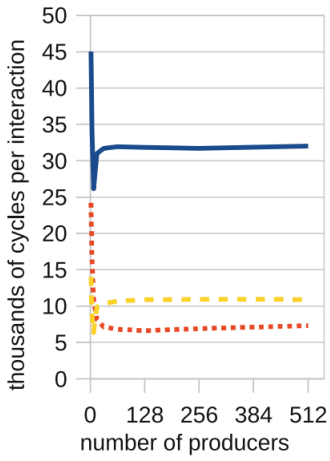
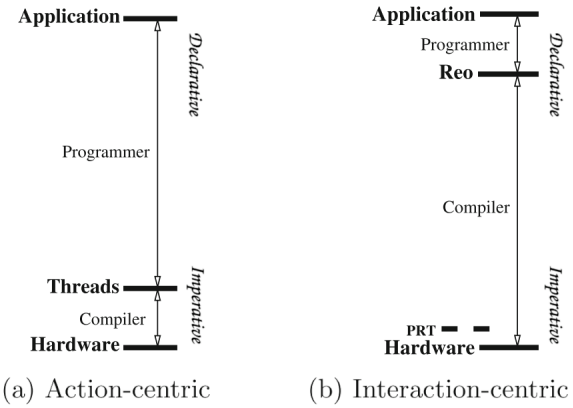


Figure 4: CPU cycles per interaction for N-Producer-1-Consumer implementation with naïve C implementation in solid navy, hand-optimized C implementation in dashed yellow and Reo in dotted red.. Image from [7].



(a) Action-centric (b) Interaction-centric

Figure 5: Visualization of the transformations necessary from the abstract algrithm to the ultimate imperative machine code for (a) a traditional, imperative language like C (b) Reo. Image from [7].

Runtime-Toolkit (PRT) for all its granular concurrency primitives. PRT offers control over system resources *without* involvement of the OS scheduler [7]; Figure 4 illustrates this positive result. Arbab et al. argue that many imperative languages cannot hope to meaningfully compile down to PRT the way Reo does, as the languages are too fine-grained for the compilers to be able to effectively map native source code to PRT. PRT is far too platform-specific and low-level for any programmer to interact with productively for any implementation of realistic complexity. They summarize this sentiment in Figure 5; for the imperative example, the large effort of the programmer is illustrated by the large 'distance' from the high-level declarative algorithm to the low-level imperative implementation in a traditional language. The corresponding compiler has little room to apply optimizations as the code it is provided with is already highly specified. By contrast, the Reo-approach affords a smaller step from algorithm to implementation by the programmer, relegating more granular work to the compiler. For this reason, the compiler is at liberty to perform optimizations at the protocol level as well as those used in both cases to make best use of the quirks of the target machine (cache coherence, etc.). Arbab et al. recognize that in both cases optimizations are performed, but in the latter case, they are performed by the less error-prone *compiler* rather than a human [7].

5 Trade-offs

This section explores the weaknesses of the languages from Section 4. We discuss how these weaknesses are currently mitigated and how they might be improved further in the future.

Chapel's layered approach to the granularity of code, as discussed in 4.2, allows the programmer to gloss over details when a high-level approach is sufficient. Unfortunately, Chapel's high-level 'global view' has been found to under-perform when compared to the same algorithm implemented in the lower, 'local view'. This is a result of the compiler's shortcoming, and perhaps a mismatch between the expectations of the programmer and the actions of the compiler. With refinement, development of the Chapel language hopes to incrementally close this performance gap [11, 15].

Coordination languages have shown immense promise, with **Linda** and **Reo** both showcasing all of our selected requirements. **Linda** has been shown to out-perform a message-passing competitor significantly, particularly for larger-scale systems [40]; this is explored further in Section 4.4. However, these findings are out of date, while comparisons to more modern competitors are scarce. **Reo** is a newer language, and it benefits from more up-to-date performance measurements to relevant technologies (such as C with MPI) [7]. Overall, it demonstrates competitive performance, as described in Section 4.4. Unfortunately, an experiment found Reo to be between 10% and 40% slower in 25% of cases, suggesting that performance is not guaranteed to consistently exceed that of imperative languages [7]. Concerns have also been raised about the **reliability** of Reo, as the exploration of their formal correctness has not yet been thoroughly proven. This is the case for many languages while still in early development. Owing to its verifiable nature, we believe Reo will overcome this limitation given time to develop.

The ownership system present in **Rust** and **Impala** does not incur any run-time performance penalty, as explored in 4.4. However, memory safety necessitates that the programmer specifies the *lifetimes* of variables in scope in relation to one another [9]. In practice, this means code is sometimes peppered by lifetime variables, which leads to a loss of **clarity** at first glance. The Rust developers have considered this, and have added *lifetime elision rules* that allow the compiler to implicitly infer omitted lifetimes in most cases [35]. Programmers also experience 'fighting with the compiler', attempting to express something not inherently memory-safe (as with the *hogwild* example in Section 4.2). This can be considered a lack of **expressiveness** in the language. Fortunately, Rust's *unsafe* language subset is intended for precisely this purpose, ensuring such restrictions in expressivity are more like speed-bumps, and less like road-blocks.

A more general problem when programming with high-level tools that abstract away from an underlying low-level implementation, it is desired that the resulting implementation is less complex than if the same task would be solved in this underlying language. This closely relates to the *complexity gap*, stating that there is an arbitrarily large gap between two complexity classes and discussing the bound relating the two [37]. In some cases one could argue that a task could be more trivially solved in a low-level language such as C, rather than using a more abstract language or tool, compiling down to a similar code base. Acknowledging that this in some cases can be true [11]; we should not focus on these individual cases but on the bigger picture. In, for example, a declarative system, one can more effectively maintain the separation of semantics and implementation. This implies less implementation complexity for future reference or when adding extra functionality. Also, systems tend to grow in complexity as functionality is added [5]. With this, the complexity gap between the two quickly diminishes, especially when a system scales in a more complex setting, such as parallel and distributed computing. See Section 4 for more insight into why abstracting away from low-level language is desirable.

6 Conclusions

With parallel and distributed computing systems becoming increasingly mainstream, we need the appropriate programming languages and tools to accommodate for this paradigm shift. Message passing frameworks such as MPI, based on primitives developed decades ago, are the most widely-used solutions for this concurrent context. However, after the examination of several novel programming languages, we argue that modern, high-level languages are just as good, if not better at solving these problems.

We identify and elaborate on four crucial requirements in modern concurrent systems: reliability, productivity, expressivity and performance. Several modern languages make meaningful, innovative contributions in these areas. Reo, in particular, distinguished itself in making significant strides in all requirements at once, while Linda shows excellent scalability in memory management using clever load-balancing. We emphasize the importance of using a tool which can be applied intuitively in a complex setting, maintaining a comprehensible overview of the code and facilitating formal protocol validation. Languages such as Reo, Chapel and Linda accomplish this by abstracting away from low-level details while still providing the necessary expressiveness. This results in developers benefiting from a faster and smoother development process, arriving at a more structured implementation. We observe how high-level languages have begun to close the performance gap with low-level languages. For example, Rust, Impala and Reo achieve strong memory and protocol safety without performance cost by relying on well-defined type systems at compile-time. Furthermore, Reo's nature afforded the Proto-Runtime-Toolkit as a compilation target, resulting in programs that use system resources without OS scheduling overhead, circumstantially outperforming hand-written optimized C-code. We identified several such viable alternatives to established messages passing frameworks.

We encourage developers to not stick to what they know and investigate these high-level programming languages. In many cases, it could result in a maintainable system with better reliability and performance.

7 Acknowledgments

Many thanks to Roy Overbeek for his guidance on the topic of coordination languages, particularly in his knowledge of Reo and the research behind it.

References

[1] C interoperability. `https://chapel-lang.org/docs/1.12/technotes/extern.html#working-with-c`. Accessed: 14.10.2018.

[2] High-performance language interoperability. `https://computation.llnl.gov/projects/babel-high-performance-language-interoperability/#page=home`. Accessed: 14.10.2018.

[3] AGHA, G. Modeling concurrent systems: Actors, nets, and the problem of abstraction and composition. In *Application and Theory of Petri Nets 1996 - 17th International Conference, Proceedings* (Germany, 1996), vol. 1091 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, pp. 1–10.

[4] AMARO, S., PIMENTEL, E., AND ROLDAN, A. M. A preliminary comparative study on the expressive power of reo and linda. *Electronic Notes in Theoretical Computer Science 180*, 2 (2007), 3–19.

[5] ARBAB, F. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI) 19* (1998).

[6] ARBAB, F. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science 14*, 3 (2004), 329–366.

[7] ARBAB, F. Proper protocol. In *Theory and Practice of Formal Methods*. Springer, 2016, pp. 65–87.

[8] ARBAB, F., HERMAN, I., AND SPILLING, P. Overview of manifold and its implementation. 23 – 70.

[9] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARI, Z., AND RYZHYK, L. System programming in rust: Beyond safety. *ACM SIGOPS Operating Systems Review 51*, 1 (2017), 94–99.

[10] CARRIERO, N. J., GELERNTER, D., MATTSON, T. G., AND SHERMAN, A. H. The linda alternative to message-passing systems. *Parallel computing 20*, 4 (1994), 633–655.

[11] CHAMBERLAIN, B. L., CALLAHAN, D., AND ZIMA, H. P. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[12] CHIFFLIER, P., AND COUPRIE, G. Writing parsers like it is 2017. In *Security and Privacy Workshops (SPW), 2017 IEEE* (2017), IEEE, pp. 80–92.

[13] COPLIEN, J. ch. In *tools* (1997), IEEE, p. 352.

[14] COUPRIE, G. Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In *Security and Privacy Workshops (SPW), 2015 IEEE* (2015), IEEE, pp. 142–148.

[15] DEITZ, S. J., CALLAHAN, D., CHAMBERLAIN, B. L., AND SNYDER, L. Global-view abstractions for user-defined reductions and scans. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2006), PPoPP '06, ACM, pp. 40–47.

[16] FOKKINK, W. Modelling distributed systems. In *Texts in Theoretical Computer Science. An EATCS Series* (2007).

[17] GANAI, M. K., AND GUPTA, A. Efficient modeling of concurrent systems in bmc. In *SPIN* (2008).

[18] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[19] GROOTE, J. F., MATHIJSSEN, A., RENIERS, M., USENKO, Y., AND VAN WEERDENBURG, M. The formal specification language mcrl2. In *Methods for Modelling Software Systems (MMOSS)* (Dagstuhl, Germany, 2007), E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, Eds., no. 06351 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[20] GROPP, W., LUSK, E. L., DOSS, N. E., AND SKJELLUM, A. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing 22* (1996), 789–828.

[21] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *ISCA* (1993).

[22] HILL, M. D. Amdahl's law in the multicore era. *Computer 41* (2008).

[23] HOLK, E., PATHIRAGE, M., CHAUHAN, A., LUMSDAINE, A., AND MATSAKIS, N. D. Gpu programming in rust: Implementing high-level abstractions in a systems-level language. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013), IEEE, pp. 315–324.

[24] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty asynchronous session types. *ACM SIGPLAN Notices 43*, 1 (2008), 273–284.

[25] JESPERSEN, T. B. L., MUNKSGAARD, P., AND LARSEN, K. F. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (2015), ACM, pp. 13–22.

[26] JONGMANS, S.-S. T., AND ARBAB, F. Global consensus through local synchronization. In *European Conference on Service-Oriented and Cloud Computing* (2013), Springer, pp. 174–188.

[27] JONGMANS, S. T. Q., AND ARBAB, F. Modularizing and specifying protocols among threads. In *Proceedings Fifth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2012, Tallinn, Estonia, 31 March 2012.* (2012), pp. 34–45.

[28] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 66.

[29] KOKASH, N., KRAUSE, C., AND DE VINK, E. P. Time and data-aware analysis of graphical service models in reo. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods* (Washington, DC, USA, 2010), SEFM '10, IEEE Computer Society, pp. 125–134.

[30] LARUS, J. R., AND RAJWAR, R. Transactional memory. *Commun. ACM 51* (2006), 80–88.

[31] LEE, E. Are new languages necessary for multicore?

[32] LI, H., AND THOMPSON, S. Safe concurrency introduction through slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (2015), ACM, pp. 103–113.

[33] LOHSTROH, M. The critical path toward the development of reo.

[34] MARTIN, D. H., AND CORDY, J. R. On the maintenance complexity of makefiles. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics* (2016), ACM, pp. 50–56.

[35] MATSAKIS, N. D., AND KLOCK II, F. S. The rust language. In *ACM SIGAda Ada Letters* (2014), vol. 34, ACM, pp. 103–104.

[36] PÉRARD-GAYOT, A., MEMBARTH, R., SLUSALLEK, P., MOLL, S., LEISSA, R., AND HACK, S. A data layout transformation for vectorizing compilers. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing* (2018), ACM, p. 7.

[37] RACCOON, L. B. S. The complexity gap. *SIGSOFT Softw. Eng. Notes 20*, 3 (July 1995), 37–44.

[38] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems* (2011), pp. 693–701.

[39] REED, E. Patina: A formalization of the rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015).

[40] SANCESE, S., CIANCARINI, P., AND MESSINA, A. Message passing vs tuple space coordination in an aerodynamics application*. In *Parallel Computing Technologies* (Berlin, Heidelberg, 1999), V. Malyshkin, Ed., Springer Berlin Heidelberg, pp. 320–334.

[41] VAN WALREE, F. Session types in cloud haskell. Master's thesis, 2017.