RE-LANG—A PARALLEL-BY-DEFAULT PROGRAMMING LANGUAGE

Tao Tao

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the College of Arts and Sciences.

Chapel Hill
2023

Approved by:
Stan Ahalt
Gary Bishop
David Plaisted
Diane Pozefsky
David Stotts
ABSTRACT

Tao Tao: RE-LANG—A Parallel-by-default Programming Language
(Under the direction of David Plaisted)

In recent years, programming language features such as lightweight threads have gained popularity in the software development workflow. Our research takes a critical look at these recent trends, rethinking them through an academic lens.

We propose a construct called “smart assignment,” supported by rewriting semantics, which enables a novel parallel-by-default programming paradigm. We present a new programming language—RE-LANG—that implements this feature. Specifically, we demonstrate how the design philosophy of RE-LANG makes imperative, parallel programming more developer-friendly. We discuss the implementation of the language and showcase performance benchmarks, as well as overhead analysis, to demonstrate its efficiency.
To family and friends all over the world and beyond...
ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to all those who have supported me throughout my journey towards earning a Ph.D. in Computer Science. I would like to extend a special thank you to my advisor Dr. David Plaisted, for his invaluable guidance, encouragement, and support throughout the entire process. I am also grateful to the members of my thesis committee Dr. Stan Ahalt, Dr. Gary Bishop, Dr. Diane Pozefsky, and Dr. David Stotts, for their insightful comments and suggestions have greatly improved the quality of my research. I would also like to acknowledge the support and resources provided by University of North Carolina at Chapel Hill and the department of Computer Science, which made it possible for me to pursue my passion for the field. Lastly, I would like to extend my appreciation to my friends and family, who have been a constant source of motivation and encouragement throughout this journey.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS</td>
<td></td>
<td>xi</td>
</tr>
<tr>
<td>1</td>
<td>Background and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Declarative Parallel Programming</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>The Challenge to Scale</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Towards a New Parallel Programming Paradigm</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>RE-LANG—A Parallel-by-default Language</td>
<td>13</td>
</tr>
<tr>
<td>1.5</td>
<td>Summary</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>System Design</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>Input Language</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Core Language</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Compiling Smart Assignment</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>From Core Language to Token Rewrite Rules</td>
<td>31</td>
</tr>
<tr>
<td>2.5</td>
<td>Extensions</td>
<td>32</td>
</tr>
<tr>
<td>2.6</td>
<td>The Stream Rewriting Execution Model</td>
<td>37</td>
</tr>
<tr>
<td>2.7</td>
<td>Software Stream Rewriting</td>
<td>40</td>
</tr>
<tr>
<td>2.8</td>
<td>Summary</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>Evaluation</td>
<td>48</td>
</tr>
<tr>
<td>3.1</td>
<td>Suite A</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>Suite B</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>Suite C</td>
<td>57</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

3.1 Fifteen benchmark programs .............................................................. 49
3.2 Suite A execution time data ............................................................ 52
3.3 Suite B execution time data ............................................................ 56
3.4 Suite C execution time data ............................................................ 60
LIST OF FIGURES

2.1 Grammar of the core language ............................................................................. 22

2.2 Eliminating the last statement in a block into a function call. The solid frame
denotes a new unique function created at the top-level of the program, where $f$ is a
unique function name. ................................................................................................. 23

2.3 Eliminating the last destructive statement in a block into a function call. The solid
frame denotes a new unique function created at the top-level of the program, where
$f$ is a unique function name. ........................................................................................ 25

2.4 Eliminating a multi-assignment statement into a series of single assignments. Here,
$X$ is not a free variable in $TT$ ................................................................................... 26

2.5 A "don’t-care" statement is converted into an empty multi-assignment. The latter is
then transformed into a dummy variable assignment, which effectively disregards
the return value of $T$. ................................................................................................. 27

2.6 Non-valued conditional statements are converted into valued conditional statements
by merging the terminating term into the conditional. .................................................. 28

2.7 Definition of the rewrite rule generator function. Every conditional is given a
unique label and translated into two rules corresponding to the two branches. This
is necessary for disambiguating nested expressions. The variable name $n$ does not
matter, assuming unary functions and no free variables. Variable tokens should be
understood as "holes." $\alpha, \hat{\alpha}, \phi, \delta$ are native runtime functions that perform memory
manipulation. .............................................................................................................. 33

2.8 Using the naive Fibonacci number function as an example, the $f_0$ token is the
unique name for the $fib$ function. The $fb$ token is the unique name for the if-else
branching statement in $fib$. Per our compilation strategy presented earlier, they
are compiled into two separate rewriting rules representing each resulting branch
of the if-else based on the value of the condition at runtime. ........................................ 34

2.9 A while loop is eliminated by transforming it into a self-recursive function call. $f$
is a unique function name. ........................................................................................... 35

2.10 Parallel stream rewriting with four processors ...................................................... 40

2.11 Function arguments return to the parent from two processors .............................. 41

2.12 Global scheduling with four processors. Each processor attempts to rewrite
$\lceil 10/4 \rceil = 3$ tokens in the conceptual total buffer into itself. The result is not
guaranteed to be perfectly even because it is a product of a rewriting iteration. ............. 43
2.13 Stream rewriting can be viewed in terms of computational tree nodes. Note that the tree not only grows but also shrinks. After a token is reduced or returned to its parents, it is deleted from the tree.

3.1 Suite A scalability results ................................................................. 51

3.2 Suite B scalability results ................................................................. 54

3.3 primes and catalan using for every loop ............................................... 55

3.4 Suite C scalability results ................................................................. 59

4.1 Spawn-continuation decomposition of a simple fork-and-join DAG. In this figure, the path denoted by the number 0 is the critical path. The paths denoted by the number 1 and 2 are continuations ................................................................. 66

4.2 An example of global load rebalancing for 4 processors. Before load balancing, there are 13 tasks in total. The equal partitioning yields \{4,3,3,3\}, which is represented by different colors in the figure. ................................................................. 72
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>AWAR</td>
<td>Atomic Write-after-read</td>
</tr>
<tr>
<td>CAS</td>
<td>Compare-and-swap</td>
</tr>
<tr>
<td>CMP</td>
<td>Many-core Chip-multiprocessors</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-programmable Gate Arrays</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>LHS</td>
<td>Left-hand Side</td>
</tr>
<tr>
<td>LL/SC</td>
<td>Load-link/store-conditional</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low Level Virtual Machine</td>
</tr>
<tr>
<td>MFence</td>
<td>Memory Fence</td>
</tr>
<tr>
<td>NBE</td>
<td>Normalization-by-evaluation</td>
</tr>
<tr>
<td>RAW</td>
<td>Read-after-write</td>
</tr>
<tr>
<td>RHS</td>
<td>Right-hand Side</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multithreading</td>
</tr>
<tr>
<td>TBB</td>
<td>Threading Building Blocks</td>
</tr>
<tr>
<td>TSO</td>
<td>Total Store Ordering</td>
</tr>
</tbody>
</table>
CHAPTER 1: BACKGROUND AND MOTIVATION


1.1 Declarative Parallel Programming

The most declarative, task-parallel programming framework is Cilk (Blumofe, Joerg, Kuszmaul, Leiserson, Randall & Zhou 1996), which relies on annotations for task spawning and joining. However, such frameworks still require programmer intervention to generate parallelism and are not completely automatic. Research has shown that certain annotations required by these frameworks can be inferred by the compiler automatically, as seen in research conducted by Hijma et al. without loss to code quality (Hijma, Van Nieuwpoort, Jacobs & Bal 2011). Hijma et al.’s ”sync generator” aims to automatically insert sync statements in Satin code, a divide-and-conquer programming model similar to Cilk. The generator operates as an extra compiler pass for the Satin compiler, analyzing the code and inserting sync statements to ensure program correctness and parallelism. To achieve this, the sync generator relies on control flow and alias analysis. Control flow analysis helps determine the program’s execution path, ensuring that synchronization points are placed at appropriate locations to maintain correctness. Alias analysis allows the sync generator to track the relationship between
different references to objects and determine whether they point to the same memory location. This helps identify potential data dependencies and prevents incorrect synchronization placements. The sync generator’s implementation ensures that the resulting program is always correct, with sync statements placed in such a way that the parallel version delivers the same results as the sequential version. By automatically generating sync statements, the sync generator makes parallel programming more accessible to programmers who are not experts.

Algorithms such as Satin hint at the possibility of a more implicit parallel programming language, even though Cilk has already made significant advancements in this regard. This motivation is further supported by research that compares the efficacy of utilizing explicit frameworks between expert and novice parallel programmers (Nanz, West & Silveira 2013). Their study focuses on four parallel programming approaches: Chapel, Cilk, Go, and Threading Building Blocks (TBB). To determine the effectiveness of these approaches, a suite of benchmark programs is implemented and reviewed by notable experts in each language. Nanz et al. aim to answer four research questions related to code size reduction, execution time reduction, speedup increase, and the overhead of implementing experts’ corrections. The results find that the design of Chapel, Cilk, Go, and TBB allows experts to only moderately improve programs written by non-experts across the four metrics.

*Explicit parallelization burdens the programmer* not only because it is tedious to write parallel annotations such as *cilk_spawn*, but also because parallel annotations are foreign extensions to languages that do not support them natively (e.g., Cilk vis-à-vis C). This adds difficulty for language tooling and is often considered a hack. Some parallelization frameworks developed in the functional programming community share less of this problem (Marlow, Newton & Peyton Jones 2011) since they manage to embed parallel instructions as a series of composable, functional strategies (aka. parallel monads). However, these functional programs are strongly typed, and complexities are instead introduced at the type level, while still requiring programmers to compose parallelism manually. It is interesting to see that functional programming needs to
resort to explicit parallelization despite its theoretical foundations—purely functional programs possess the Church-Rosser property (Barendregt et al. 1984) and have the potential for full automation of parallelization.

In an ideal world, the programmer would only write high-level, abstract pseudocode and let algorithms handle the rest. Such is the holy grail of declarative parallelism. However, this vision is hardly realistic. A big issue for implicit parallelization lies in the difficulty to avoid concurrency errors for any non-trivial program that allocates and accesses shared heap memory.

The most popular approach involves the use of intelligent compilers that perform static analysis and extract parallelizable portions from the input program (Fonseca et al. 2016, Gupta, Mukhopadhyay & Sinha 2000, Rafael et al. 2014). Similarly, there are methods that utilize runtime feedback to adopt parallelism schemes and adjust their dynamic behavior (Duran, Corbalán & Ayguadé 2008, Harris & Singh 2007, Prechelt & Hanssgen 2002). However, being based on heuristics, these methods have a common limitation in their lack of generality: They work well only on specific types of programs and are not effective if the codebase does not fit the pre-defined stereotypes. This paradigm requires the programmer to guess the behavior of the compilers and modify their code accordingly, making the codebase fragile to changes, which goes against best practices in software engineering. Recent evaluations of these frameworks also show less than desirable performance (Harel, Mosseri, Levin, Alon, Rusanovsky & Oren 2020). On the other hand, frameworks such as Grace (Berger, Yang, Liu & Novark 2009) detect concurrency errors at runtime retroactively and roll back program states using transactional memory. However, this approach only enjoyed limited success on coarse-grained workloads. As the amount of parallelism increases, the overhead of transactional processing overwhelms the benefit.

1.2 The Challenge to Scale

In addition to correctness issues, the biggest challenge to declarative parallel programming is to efficiently schedule a large number of fine-grained parallel tasks. In the scheduling world, the directed
acyclic graph (DAG) model is commonly used to represent parallel programs, with each node representing a unit task. The directed edges between the nodes indicate sequential dependencies between the tasks, while parallelism is achieved by having multiple outgoing edges from a single task. The result of the program is produced by merging sub-DAGs that were generated through parallel execution. For the past decades, work stealing has emerged as the predominant approach for scheduling dynamic, irregular DAGs in multicore processors (Arora, Blumofe & Plaxton 1998, Blumofe, Joerg, Kuszmaul, Leiserson, Randall & Zhou 1995, Blumofe & Leiserson 1994, Fatourou & Spirakis 2000, Tchiboukdjian, Gast, Trystram, Roch & Bernard 2010). Work stealing uses concurrent deques to manage parallel tasks and assigns processors the responsibility of pushing newly created tasks to the top of the deque and popping tasks from the bottom when needed. If a processor runs out of tasks, it may steal from the bottom of another processor’s deque.

However, work stealing has a major limitation in that it is only effective for coarse-grained programs with a relatively low number of spawns. Attiya et al. (Attiya, Guerraoui, Hendler, Kuznetsov, Michael & Vechev 2011) have theoretically demonstrated that all traditional work-stealing schedulers require a minimum of one unit of synchronization cost in the form of memory fence (MFence) or compare-and-swap (CAS) operations when accessing tasks from the deque. This results in a direct correlation between the number of spawns and the amount of synchronization overhead in the system, making work stealing unsuitable for fine-grained programs with a large number of spawns.

Limiting programmers to coarse-grained workloads obstructs the ideal of declarative parallelism, where the emphasis is on defining task dependencies logically, leaving scheduling and overhead concerns to the compiler and runtime. The ideal scenario would allow programmers to concentrate solely on spawning tasks, without having to manually manage parallelization strategies and overhead. A clear example of this can be illustrated by the Fibonacci recursive program.
The Fibonacci program, as written, follows a declarative approach and should theoretically scale proportionately with the number of processors. Unfortunately, the number of tasks spawned by this program is exponential, leading to a high level of overhead that hinders work-stealing schedulers from performing effectively as the input size increases. This scenario is not unique, as many software programs encounter similar issues when attempting to achieve parallelization under a declarative approach. The challenge lies in finding ways to mitigate parallelization overhead while maintaining the simplicity and clarity of the declarative approach.

As an important note, we are abundantly aware that more efficient algorithms exist to compute Fibonacci numbers. We want to highlight that the purpose of the above example is not to discuss optimal Fibonacci implementations but to give a stereotypical declarative parallel program that spawns a large number of fine-grained tasks. Real-world software resembles much more a ”colloquially” written program that has no well-predictable structure like the recursive Fibonacci program here than a carefully thought out and optimized numerical algorithm. The program below is a representation of software development in the real world and not so much that it computes the result of Fibonacci.

As an example, a common software pattern in the real world is to traverse a JavaScript Object Notation (JSON) object, which is the predominant means for distributed web servers to communicate with each other. It is well-known that web servers spend a significant amount of time processing JSON objects and hence
parallelizing this process is a point of interest. JSON objects are untyped and arbitrarily nested. This means that a JSON object resembles a tree and does not follow a fixed structure. Therefore, in general, to parse and process a JSON object, the programmer has no choice but to adopt a divide-and-conquer, recursive approach to the tree structure of the JSON object. However, doing so generates a number of fine-grained tasks proportional to the size of the JSON tree, which is potentially very large. Existing frameworks such as work stealing would struggle to scale such a JSON processing algorithm for similar reasons to the previous Fibonacci example.

A common approach to addressing the limitations of work stealing is to implement cutoff schemes, which halt the creation of new tasks once the input size drops below a predetermined threshold. For example, Fonseca et al. (Fonseca & Cabral 2016) propose a cost model that uses a hybrid approach, combining static analysis and runtime optimizations, to decide between parallelization alternatives. The model is based on the fact that some instructions are more expensive than others. It micro-benchmarks Java operations and the overhead of creating new tasks, analyzing method invocations using runtime variables to delay task spawning decisions to runtime. The model also considers memory usage for each possible task to avoid potential issues with swapping. The authors implemented the cost model in JPar, a Java automatic parallelization compiler, and evaluated it on several benchmarks. The results show that the approach improves performance compared to not using any granularity control. In some cases, the cost model even outperforms manual approaches. However, despite past efforts in cutoff schemes, a universally effective cutoff scheme has yet to be discovered, making the development of a more generic solution a topic of active research.

In this regard, several advancements have been made in the field of work-stealing algorithms. Early implementations relied on the assumption of sequential consistency in the underlying architecture (Arora, Blumofe & Plaxton 1998, Chase & Lev 2005, Frigo, Leiserson & Randall 1998, Hendler, Lev, Moir & Shavit 2006). However, this has proven to be problematic in modern systems, as it necessitates the use of costly MFence operations (Attiya et al. 2011). To address this challenge, researchers have proposed alternatives to
traditional work-stealing. These alternatives employ either heuristics (Adnan & Sato 2011, Faxén 2010a, Hiraishi, Yasugi, Umatani & Yuasa 2009, Liu, Song, Liu & Hao 2014, van Dijk & van de Pol 2014, Vrba, Halvorsen & Griwodz 2010) or advanced algorithms (Acar, Charguéraud & Rainey 2013a, Hendler & Shavit 2002, Rito & Paulino 2022) to mitigate the overhead incurred by excessive spawns in fine-grained programs. Some have demonstrated that it is possible to eliminate synchronization overhead by either requiring specialized programming semantics (Castañeda & Piña 2021, Michael, Vechev & Saraswat 2009a) or making strong assumptions about hardware topology (Morrison & Afek 2014). However, these solutions are not suitable for general computation purposes.

To summarize, the status quo of parallel programming frameworks puts programmers in an uneasy position: To embrace the freedom of declarative parallelism, the programmer must tread carefully to obey pre-defined paradigms dictated by the compilers. At the same time, ironically, when the programmer parallelizes, he must not parallelize too much for the fear that the overhead can easily catch up with the benefit. **This situation violates the sense of control in programming:** When programmers use a tool, the tool should do exactly what it is expected to do—in our context, to parallelize. However, for all existing frameworks, the viability of the parallelization goal is highly circumstantial.

### 1.3 Towards a New Parallel Programming Paradigm

This thesis aims to create a parallel programming language and a corresponding execution model that puts control back into the hands of the programmers. Specifically, we claim that a parallelization language should function as a deterministic layer of abstraction: It should always produce the outcome an average programmer expects: The compiler and runtime must efficiently distribute parallel tasks across multiple processors (i.e. to scale). Testing the extent to which this is successful is simple: Given $P$ identical processors, the closer the execution time is decreased to $1/P$ of a single-core execution, the more successful the system is.
The abstraction also needs to be general: We need to establish confidence that it will always work for any input program. In the rare cases that it fails to work, the system should provide a clear rationale for the failure, as a reference to future improvement on the programmer’s workflow. To achieve this, we propose an alternative to the traditional, locally stealable tasks as our runtime scheduling policy. This policy differs from conventional work-stealing algorithms by distributing task workload through global rebalancing sessions, as opposed to relying on concurrent data structure sharing. We show that this policy, when applied to programs where parallelism grows with input size, results in provably minimal overhead and high scalability.

In addition to efficacy and generality, good abstraction must also be intuitive and easy to use. A good way to assess this is to look at the degree of automation achieved by employing such abstraction, where an analogy is automatic garbage collection (GC) systems in modern programming languages. Automatic garbage collection operates on the principle of tracking and deallocating memory that is no longer referenced by any active data structures or variables. This process is typically performed in the background, allowing developers to focus on implementing their algorithms and features without being burdened by manual memory management. This has led to a significant increase in programmer productivity, as well as enhanced software reliability and robustness.

In the context of parallelization, we introduce RE-LANG, an automatic parallel programming language. The defining feature of RE-LANG is its \textit{parallel-by-default} approach, where all function arguments are automatically scheduled as parallel tasks, eliminating the need for a parallelization keyword like \texttt{cilk spawn}. This design provides two main benefits.

First, it simplifies the mental model of writing software programs. Traditionally, function arguments are evaluated in a fixed order, typically from left to right. While this convention is widely practiced, it requires programmers to carefully design the order of function arguments at all times to ensure the side effects potentially induced by function calls happen in the correct order. By removing the assumption of a fixed order, the programmer can re-adapt to the simpler mental model that function arguments are, in essence,
data dependencies between different procedures in a program: The arguments block the execution of the function due to the dependencies, but the arguments themselves should not depend on each other—if they must do, the programmers should specify them by making those implicit dependencies explicit. This mental model promotes clear thinking in programming and enables a programming paradigm that lends itself easily to parallelization.

Second, it allows for the parallelization of some serial programs without any modification. This includes purely computation programs without any side effects. As an example, consider the recursive Fibonacci number program written in RE-LANG.

\[
\text{fib}(n) = \begin{cases} 
  n & \text{if } (n < 2) \\
  \text{fib}(n - 1) + \text{fib}(n - 2) & \text{else}
\end{cases}
\]

On the surface, it is exactly the same as a sequential Fibonacci program. However, since function arguments are parallelized automatically, both term \(\text{fib}(n - 1)\) and \(\text{fib}(n - 2)\) will be executed in parallel because they are arguments of the + function/operator. The Fibonacci program is just a basic example illustrating the general implication of the parallel-by-default mechanism—\textit{as long as the programmer ensures function arguments encode, and only encode, data dependencies between each argument and the called function, the program can be parallelized with zero effort.} This mechanism establishes a direct interpretation from a logically correct program to a parallelized version of the program and therefore allows programmers to focus solely on designing algorithms instead of parallelization instructions.

However, additional care must be taken when designing a new programming language. In the context of parallel programming, a new programming language that enjoys high productivity needs to satisfy the following characteristics.

First, the language should maintain a high degree of abstraction—meaning that the programmer should feel that they are writing down a pseudocode algorithmic design as opposed to a series of low-level instructions.
In this regard, rewriting semantics emerges as a candidate for the job. Specifically, rewriting semantics allows the programmer to define a program in terms of rules that describe how to transform expressions from one form to another. This approach is more closely aligned with the way that programmers think about problems and allows them to focus on expressing their intentions clearly, rather than worrying about the details of how the program will be executed. In contrast, traditional programming languages based on operational semantics require the programmer to reason about the behavior of a program in terms of a specific machine model or execution environment. This approach can be more complex and error-prone, as it requires the programmer to consider the details of the machine’s architecture, memory layout, instruction set, etc. As a result, operational semantics can be more difficult to use and understand and can lead to code that is less expressive and maintainable. This is especially a concern in declarative parallel programming since a low-level language introduces additional concerns over implementation details, making the codebase harder to read and more error-prone. In parallel programs, errors (aka. bugs) are notoriously difficult, if not practically impossible, to find and eliminate.

Second, the language should be based on relatively simple semantics—without a large set of first-class features (i.e. features available in the core\(^1\) language). The main motivations for simplicity are two: (1) A simple language is more easily adopted by the programming community. (2) A simple language semantics allows us to effectively parallelize any program. In a language with a large number of features, the complexity of the semantics grows, making it more difficult to parallelize. Traditional programming languages based on operational semantics are specified in a sequence of steps, where each step transforms the program state from one state to the next. The state includes the values of variables and the control flow of the program. Reasoning with operational semantics can be difficult due to the fact that such reasoning is constrained to each big-step or small-step language construct defined per operational semantics, where problems occur in isolated paths.

\(^1\)We will discuss the definition of the core language in the coming chapter. Generally, the core language (of a programming language) consists of the primitives of the language that are not reducible to the language itself and must be interpreted into lower-level languages or computation models.
of the program’s execution, blurring the bigger picture. On the other hand, rewriting semantics naturally supports equational reasoning, enabling programmers to reason about the correctness and equivalence of programs using algebraic techniques, leading to more reliable and maintainable code. In rewriting semantics, equational reasoning is based on transforming one expression into another using a set of rules. These rules can be expressed in a very concise manner, often using just a few lines of code. The rules are typically defined as pairs of expressions, where the first expression is called the left-hand side (LHS), and the second expression is called the right-hand side (RHS). The rule specifies that any occurrence of the LHS in an expression can be replaced by the RHS. The replacement is done by applying the rule repeatedly until no more occurrences of the LHS are present. The use of concise rewrite rules is possible because the rules operate on term expressions, which represent a program’s state. Reasoning about the transformation of term expressions is typically easier than reasoning about the mutation of state. Overall, rewriting semantics based on rewrite rules operates on a simplified representation of computation, making programs easier to write and understand.

At last, the programming language design needs to be efficient and general, supporting critical and well-received features such as destructive assignments. Purely functional programming languages do not support destructive assignments, which means that once a variable is assigned a value, it cannot be changed. This approach is based on the idea that mutable states can lead to errors and make it difficult to reason about the behavior of a program. However, this can also be inconvenient in some cases, especially in situations where the state of the program needs to be updated. For example, to encode destructive array assignment in purely functional programming, the entire array needs to be duplicated, or the runtime must resort to techniques such as immutable data structures and higher-order functions to manipulate data in a non-destructive way (Okasaki 1999). However, doing so destroys the original semantics of mutable arrays, where a strong performance guarantee is implied due to cache locality. The lack of support for destructive assignments on par with imperative paradigms in purely functional programming languages can be frustrating.
and makes it difficult to write efficient code in certain situations. In some cases, functional programming may require more overhead to achieve the same results as imperative programming, leading to longer development times and more difficult-to-maintain code. On the other hand, Plaisted et al. (Plaisted & Barnett 2021) have shown that destructive assignment can be elegantly incorporated into rewriting semantics, leading to a rewriting-based programming language design that retains the simplicity and abstract nature of rewriting semantics while yielding efficient interpretation.

Term rewriting has been recognized as a powerful formalism for computation (Willem & Klop 1990). Within a term rewriting system, terms serve as representations of data, and rewriting rules function as mechanisms to transform said terms. The utilization of term rewriting as a means of programming has been the subject of exploration in previous works (Brus, van Eekelen, Van Leer & Plasmeijer 1987, Goguen, Winkler, Meseguer, Futatsugi & Jouannaud 2000, Gräf 2009), among which the works by Plaisted et al. introduces destructive semantics, allowing adaptation of rewriting semantics to interpret imperative software programs. The programming language we aim to create is straightforward, without the presence of complex, higher-order features such as pattern matching and associative-commutative (AC) unification. These complex features can prove challenging for the average programmer to understand, and the presence of multiple unifiers can complicate the estimation of program complexity and deviate from the typical imperative approach.

Curiously, rewriting semantics is an opportune choice that suits our efficient parallel scheduling algorithm: Each processor can be easily interrupted to quickly convene at the global thread barrier by quickly checking a flag after each rewriting step. Traditional programming languages all feature run-to-completion semantics—when a function is called, it executes the full set of instructions comprising the function until termination, oblivious of the outside environment. In this case, work stealing presents as a more natural choice, given that work stealing allows the processors to continuously execute exclusively owned tasks. Work stealing, however, as previously explained, is not ideal for parallel scalability. Granted, with enough bravery, existing functional programming languages can perhaps adapt a similar suspendable semantics for parallelization.
However, it is entirely unclear how to do so for the most widely accepted functional language interpretation models. For example, Haskell, the most popular functional programming language, is interpreted based on lazy graph evaluation, which by definition, does not evaluate function arguments unless they are required to compute the weak-head normal form. Another widely used semantics is normalization-by-evaluation (NBE), which also adopts laziness in that it does not evaluate function arguments unless needed at runtime. At the core, purely functional programming languages are canonically mapped to mathematical objects such as lattices and graphs according to denotational semantics (Schmidt 1986). The development of parallelization frameworks under this paradigm heavily emphasizes safety and correctness properties as opposed to raw performance characteristics (Kuper & Newton 2013). On the other hand, rewriting semantics, particularly the semantics of stream rewriting that we will adopt in this thesis, closer aligns with our goal.

1.4 RE-LANG—A Parallel-by-default Language

We now demonstrate the design of our new programming language, RE-LANG, in more detail by giving a few example programs and how they are parallelized. We start off by quickly revisiting the Fibonacci example rewritten in RE-LANG.

\[
\text{fib}(n) = \begin{cases} 
  n & \text{if } (n < 2) \\
  \text{fib}(n - 1) + \text{fib}(n - 2) & \text{else}
\end{cases}
\]

The implementation of the Fibonacci sequence in RE-LANG appears to be identical to a sequential implementation. However, the key difference is that due to the parallel-by-default approach in RE-LANG, both calculations for \(\text{fib}(n - 1)\) and \(\text{fib}(n - 2)\) will be executed in parallel as they are the arguments of the + function. This demonstrates the ease with which RE-LANG enables the parallelization of pure computational programs, such as the Fibonacci sequence, without any additional effort. The next example is the standard merge sort algorithm.
mrgsrt(a, l, r) = if (l >= r) then Return else
    m ← l + (r - l) / 2;
    (mrgsrt(a, l, m), mrgsrt(a, m + 1, r));
    merge(a, l, m, r);

In RE-LANG, the parallelization of the two recursive calls in the merge sort algorithm,
mergesort(a, l, m) and mergesort(a, m + 1, r), is achieved by utilizing the tuple operator. This
operator allows both calls to be executed in parallel, as they are treated as arguments of a tuple constructor
function.

Should the return values of the calls be required, the programmer can assign them to variables using the
tuple operator and assignment operator, as shown below.

(x, y) ← (mrgsrt(a, l, m), mrgsrt(a, m + 1, r))

However, it is understood that the automatic parallelization mechanism may not always be appropriate,
particularly in cases where function arguments have side effects that must occur in a specific order. To address
this, RE-LANG stipulates that all assignment operations are executed sequentially. As a result, sequential
execution of the two recursive calls in merge sort can be achieved by modifying the code to consist of two
separate assignment operations.

x ← mrgsrt(a, l, m);
y ← mrgsrt(a, m + 1, r);

If the return values are not needed, they can be omitted.

mrgsrt(a, l, m);
mrgsrt(a, m + 1, r);
Written this way, the recursive call to `mrgsrt(a, m + 1, r)` is only initiated after the completion of the first call to `mrgsrt(a, l, m)`, hindering parallelization.

We need to emphasize that while RE-LANG offers a convenient parallel-by-default mechanism for effortless task spawning, it is the responsibility of the programmer to ensure the resulting parallel program is correct. In other words, any spawned tasks must not possess potential data race issues with each other, and if they do, explicit concurrency synchronization is required on behalf of the programmer. To help programmers with correctness, we can incorporate existing technologies such as static analysis or transactional memory. This thesis focuses on RE-LANG itself, and such efforts are left as future work.

RE-LANG also incorporates two structural parallelization constructs, namely the `forall` and `forevery` loops, further enhancing its parallel computing capabilities. The `forall` loop divides the loop iterations equally among the available processors, making it an ideal choice for embarrassingly parallel problems. As an illustration, consider the following parallelization of a matrix multiplication algorithm using the `forall` loop.

```plaintext
forall i = 0 to (m - 1)
  for j = 0 to (p - 1)
    // Compute the dot product of the i-th row of the first matrix and
    // the j-th column of the second matrix.
    dotproduct(mat1, mat2, m, n, p, i, j);
```

Sometimes `forall` may result in load imbalance if the amount of work required in each iteration of a loop is not consistent. For example, computing the prime numbers between 2 and 10,000 using a for loop would result in varying amounts of work for each iteration, as testing the primality of a number can require different amounts of computational effort. This imbalance is further enlarged by the fact that `forall` loop only divides the loop items into large chunks.
To mitigate this issue, RE-LANG introduces the `forevery` construct. This construct employs a binary divide-and-conquer approach to partition the loop into smaller iterations, resulting in finer-grained tasks and reducing load imbalance at runtime.

In our experiments, we have found that the overhead associated with the `forevery` loop is higher in comparison to the `forall` loop, particularly when the number of loop iterations exceeds 500,000 on our evaluation platforms. As a general advice, we recommend always using the `forevery` loop for overall good performance unless the loop is clearly too large. In practice, we found that large loops tend to only appear in the top-level program context (aka. the main function), a pattern widely used for well-structured parallel algorithms such as a graph or array processing for distributing a large number of tasks across all processors uniformly. This pattern is easy to identify, and programmers should use `forall` loop accordingly. A large loop that is hidden from the top-level program context usually indicates poor algorithm design, and is more than a problem caused by the overhead of `forevery` loop. In all, we consider this to be an acceptable level of abstraction leak since this thesis focuses on divide-and-conquer-style task parallelism. Mitigation and optimization regarding loop parallelism may be explored in future works.

1.5 Summary

In RE-LANG, the assignment operator `←` is central to the design of the programming paradigm: On one hand, without the assignment operator, the language would suffer from limited capability for sequential computation, leading to unnecessary hassles and complications in programming. On the other hand, the smart assignment operator integrates both the semantics of destructive writes and variable initialization. Additionally, by omitting the left-hand-side variable, the resulting expression looks exactly the same as a standalone statement with side effects. By stipulating that all smart assignment statements run sequentially in relation to their context, RE-LANG provides a familiar environment for sequential programming in this regard to existing languages such as C or Java. This allows for an easy transition to RE-LANG by programmers.
established in traditional languages. Due to its significance and defining nature of RE-LANG, we will refer the \( \leftarrow \) as the \textit{smart assignment} operator, lying at the core of RE-LANG’s parallelization paradigm.

In summary, our research offers a contribution to the field of parallel computing by demonstrating the feasibility of creating a declarative programming language for fine-grained parallel programs—RE-LANG. With this language, programmers can focus on declaring the parallelism in their code without being concerned about the potential performance implications, as the language is designed to efficiently schedule the program’s execution. RE-LANG is designed around rewriting semantics, with simple yet a complete set of features such as recursive functions and destructive assignments. Among them, the unique automatic, parallel-by-default mechanism is especially powerful, requiring minimal effort on behalf of the programmer to specify parallel tasks. When parallelism is not desired, programmers can always fall back to using smart assignment statements, which by definition execute sequentially, an elegant mechanism incorporated into the semantics of rewriting.
CHAPTER 2: SYSTEM DESIGN

In this chapter¹, we lay out the detailed design and implementation of our new programming language RE-LANG. As mentioned in the first chapter, one aspect of our design philosophy is simplicity for the purpose of ease of use and ease of parallelization, meaning the input language to RE-LANG does not start off with a large set of first-class features. However, the current implementation of RE-LANG² does come with all the well-established features similar to mainstream programming languages such as C or Java.

The RE-LANG compiler works in two steps. We first compile a high-level, imperative input language to a small core language. Then, the core language is translated into token rewrite rules.

\[
\text{Input Language} \xrightarrow{1} \text{Core Language} \xrightarrow{2} \text{Rewrite Rules}
\]

(Imperative) ⇒ (Equational)

In the second step, the core language is already very similar to token rewrite rules in form (Fig. 2.8). Hence, the compilation of this step is straightforward. The challenge lies in the first step of translating an imperative input language with high-level features such as loops and assignments into a low-level language that resembles functional-equational semantics. In this respect, we find Plaisted et al.’s work relevant (Plaisted & Barnett 2021). They showed that it is possible to perform this compilation through a series of program translation procedures that preserve the semantics.

¹© 2022 IEEE. Reprinted, with permission, from T. Tao and D. Plaisted, “Automatic Parallelization of Programs via Software Stream Rewriting,” 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2022

²https://github.com/uncttao/re-lang
2.1 Input Language

The input program has features similar to mainstream programming languages demonstrated below. In the following, $T$ denotes any expression in the language.

A tuple is the primary data structure to express a combination of values. For example, a ternary tuple takes the form of $(T, T, T)$. A tuple is often used as a trivial function to bootstrap its arguments for parallelism.

Arrays can be viewed as dynamic tuples. They are essentially fixed-sized allocations on the heap. An array must first be initialized by $(\ldots T)$, where the expression $T$ specifies a fixed length. To read from a tuple or array, we use the syntax $A[I]$. $A$ represents some array, and $I$ is an expression that corresponds to the array index to be accessed.

An $\text{if } T \text{ then } T \text{ else } T$ statement corresponds to the usual branching behavior in most programming languages.

The language has both $\text{for}$ and $\text{while}$ loops similar to those in C.

(Smart) assignment statements act like local variable declarations. They are written as $V \leftarrow T$. $V$ is a variable that is used later in the program.

Block statements are in the form of $S; \ldots; S; T$, where $S$ is some smart assignment statement. The semicolon $;$ is used as a separator of the assignment statements. This is similar to the syntax of a block $(\ldots)$ in mainstream languages.

Functions in the language take the syntax of $F(\overrightarrow{V}) = T$, where $\overrightarrow{V}$ is a list of parameter variables. There is no restriction on how functions can be composed. The language supports arbitrarily complex recursive structures, such as self-recursion and mutual recursion.

We stipulate that all variables in the language must be non-capitalized words (e.g., $x$, $y$, $z$). All capitalized words are treated as constant symbols (e.g., $\text{Apple}$, $\text{Orange}$). Constants can be used as values with special
fixed meanings. For example, we can use $T$ and $F$ to indicate true and false logical values, and we can use `None` as a symbolic placeholder.

The language has basic, built-in operators for natural numbers $+$, $-$, $\ast$, $/$ as well as for logical values $\&\&$, $||$, $!$. It also has an equality operator $=$, which denotes literal equality between two constants or numbers. The concrete implementation of these primitive operations is delegated to the underlying execution machine. Specifically in our case, they are delegated to equivalent operations in our compiler’s compilation target—low-level statements in C++\(^3\).

RE-LANG is dynamic and has no type system. In an untyped language, the programmer does not need to specify types for variables, function parameters, or return values. This makes RE-LANG’s code simpler and more concise, which can be especially beneficial for small scripts or rapid prototyping, where developers want to quickly iterate on ideas without being bogged down by type-related issues. Untyped languages also offer greater flexibility because they do not enforce type constraints. This allows for more dynamic behavior, where an object’s behavior determines its type, rather than a predefined type declaration that can make behaviors such as error handling more difficult.

### 2.2 Core Language

The core language is an intermediate language as a result of compiling the input language. The complete syntax for the core language is listed in Fig 2.1. A core language program consists of a set of function definitions followed by a call to the entry function. Terms consist of primitives, variables (i.e. `Var`), and larger expressions composed by arithmetic, comparison, and boolean operators. `arith`, `comp`, and `bool` refer to these grammatical constructs. Functionals correspond to function calls. Static tuples (STuple) are special data structures that allow the juxtaposition of arbitrarily many terms. Projection and destruction terms allow reading and writing to tuples, respectively. Dynamic tuples (DTuple) correspond to arrays in classic

---

\(^3\)This is not to be misunderstood as that "RE-LANG programs are transpiled into C++ programs". We only use C++ to the extent as a portable, high-performance code generation platform for the parallel term rewriting runtime.
programming concepts. The only allowed primitives are true/false boolean symbols, natural numbers, and constants. Constants must be capitalized alphabetic words, while variables are non-capitalized.

We define the semantics of the core language to be call-by-value. In other words, all function arguments must be evaluated first before the function call is evaluated. An example of a core language program is the following binary search program.

```plaintext
bsearch(i, j, a, n) = if (i > j) then (F, None)
else if (i = j) then
    if (a[i] = j) then (T, i) else (F, None)
else if (n < a[(i + j) / 2])
    then bsearch(i, (i + j) / 2, a, n)
else bsearch((i + j) / 2 + 1, j, a, n)
```

### 2.3 Compiling Smart Assignment

The core language is variable-less. The only exceptions are function parameters. However, in a practical programming setting, it is often desirable to introduce local variables to clarify the programmer’s intent and avoid duplication. For example, the programmer usually prefers the following style rather than the style presented in the same algorithm earlier.

```plaintext
bsearch(i, j, a, n) = if (i > j) then (F, None)
else if (i = j) then
    if (a[i] = j) then (T, i) else (F, None)
else
    sum ← i + j; // smart assignment 1
    mid ← sum / 2; // smart assignment 2
    if (n < a[mid])
        then bsearch(i, mid, a, n)
    else bsearch(mid + 1, j, a, n)
```
Figure 2.1: Grammar of the core language
Figure 2.2: Eliminating the last statement in a block into a function call. The solid frame denotes a new unique function created at the top-level of the program, where \( \ell \) is a unique function name.

The above version improves upon the first example by admitting a new block structure that contains smart assignment statements.

A block has at least one smart assignment statement followed by a single terminating statement \( TT \). \( TT \) does not have to be in the core language. For example, it can be a block with more complicated structures inside. However, \( TT \) itself cannot be an assignment statement. This is because a block as a whole is considered as an expression, and it must have a value. Its terminating statement determines the value of a block. Formal interpretation of the block grammar varies among common programming languages. Here, we adopt a strict, sequential interpretation: the assigned variable is available to the execution context only after the assignment statement appears. Any variable with the same name shadows all previous occurrences. This interpretation is natural and agrees with most imperative programming languages.

\[
\ell \left( FV(TT) \setminus \{V\}, \ T \right) \\
\cdots \\
\ell \left( FV(TT) \setminus \{V\}, \ V \right) \\
é T T
\]

Using this interpretation, we use the transformation strategy illustrated in Fig 2.2 to eliminate a block of assignment statements into a core language term. A strategy is simply an abstract syntax tree (AST)
transformation procedure represented by $\Rightarrow$. In this case, the strategy only eliminates the very last assignment in a block. To eliminate the entire block, we simply apply the strategy iteratively.

In the strategy, $FV(T)$ computes all the free variables that appear in $T$. $FV$ is a compiler function, and all computation associated with $FV$ is performed at compile time (i.e. the function $FV$ does not appear in the actual program). The framed box denotes a new function generated in the process, which we eventually append to the top-level program. The strategy replaces the terminating statement $TT$ in a block with a call to a newly generated function $\varepsilon$, and the new function is a wrapper for $TT$. We let the name $\varepsilon$ be unique. The call for function $\varepsilon$ becomes the new terminating statement.

It is not hard to see why this strategy is correct. (1) The evaluation order is preserved because the core language is call-by-value: $T$ is evaluated first as an argument of $\varepsilon$. Only then, $TT$ is evaluated as the right-hand side of $\varepsilon$ as the result of the function call. (2) The program’s structure is preserved: the free $V$ in $TT$ is still bound on the term $T$. This is because $T$ is passed into $\varepsilon$ as the argument for $V$. The rest of the free variables in $TT$ is passed as other arguments. Hence, their bindings remain the same in the larger context as desired. (3) The term $T$ is only evaluated once, which is obvious.

2.3.1 Destructive smart assignments

The previous strategy applies in general to the elimination of local variables. We can use the same syntax to denote destructive memory manipulation. Note that in the core language, the user must call the native function $\text{Dest}$ to perform a memory write operation. In typical programming languages, such an operation is usually carried out in a more readable form.

```plaintext
a[1] ← x;
```

We admit this syntax into the input language. Now a block contains not only local variable assignments but also destructive assignments in the form of $T_a[T_i] ← T_i$. Within the same framework of iterative reduction, if
the last assignment statement is a local variable assignment, use the previous strategy in Fig 2.2 to eliminate
the assignment. In case it is a destructive assignment, the strategy in Fig 2.3 is used instead. This strategy is
very similar to the previous strategy except that the call to \( \text{Dest} \) is positioned on a nullified argument \( X \),
where \( X \notin FV(TT) \). This is because a destructive assignment is purely side-effectful and does not return any
value.

2.3.2 Multi-smart-assignments

We consider one last kind of smart assignment statement—the multi-smart-assignment.

\[
(Assignee^*) \leftarrow T;
\]

where \( Assignee = V \mid T_a[T_i] \)

This statement accepts a list of possibly different kinds of assignees on the left-hand side instead of just one.

The elimination strategy in Fig 4 gives the multi-assignment a precise meaning: a multi-assignment unrolls
into a sequence of independent assignments whose right-hand side values are indexed based on the position
of the assignee listed. The first generated assignment is special and is always the value of \( T \).

As an example to illustrate this strategy, consider the following code snippet.
Figure 2.4: Eliminating a multi-assignment statement into a series of single assignments. Here, $X$ is not a free variable in $TT$. 

(a, b, c, d) ← (Apple, Orange, Banana, Grapes); 

In this snippet, (Apple, Orange, Banana, Grapes) is an array initialized to contain four constant values. The first statement is a multi-smart-assignment statement so that after the statement, the variables $a$, $b$, $c$, $d$ contain the value Apple, Orange, Banana, Grapes in that order. This is true because according to our strategy, this snippet is compiled to

$X ← (Apple, Orange, Banana, Grapes);$

$X ← T;$

$x ← X[0];$

$y ← X[1];$

$z ← X[k];$

We are not afraid of the newly created variable $X$ because, by definition, $X$ is a new variable that does not appear in the continuation $TT$, so neither it is used, nor can it overwrite existing variables in $TT$. 

26
The multi-assignment statement has two primary uses. The first is a consumer pattern for tuples. When writing a multi-assignment, the programmer expects the right-hand side expression $T$ to return a tuple. The multi-assignment decomposes the returned tuple by assigning different positions of the tuple to variables. This provides an easy way to consume the tuple values. Secondly, so far there is no way for a programmer to call a function without assigning a variable to the return value. In case the programmer does not care about the return value, the programmer at least has to assign the return value to an unused local variable. This approach is tedious and error-prone. With the multi-assignment syntax, the programmer can do the following.

$() \leftarrow \text{DontCareAboutResult}();$

We can verify that $T; \Rightarrow () \leftarrow T;$ reduces into $X \leftarrow \text{DontCareAboutResult}();$. Since $X$ is not free in the continuation (i.e. last statement in the block $TT$), its value is effectively discarded. To make it even simpler, we add the strategy in Fig 2.5 to gracefully handle the situations where $T$ is purely side-effectful and where the programmer does not care about the return value.

2.3.3 Non-valued conditional

The only control structure in the core language is the if-else expression, which returns a value. What is relatively more common in imperative languages is an if-else structure that returns no value and contains only assignment statements. Additionally, we often allow the else clause to be omitted. Call this structure the "non-valued conditional." An example is the following.
mergeSort(a, p, r) =
   // A conditional without terminating term inside body blocks
   if (p < r) then
      q ← (p + r) / 2;
      mergeSort(a, p, q);
      mergeSort(a, q + 1, r);
      merge(a, p, q, r);
   Return

We incorporate this syntax into the source language and eliminate it using the strategy in Fig 2.6 We first start from the end of a block with a non-valued conditional followed by a terminating term. Then subsume the terminating statement $TT$ into the clause and transform it into a regular, valued conditional. This involves duplicating $TT$. An else clause is automatically generated if it does not exist. This transformation is straightforward, and the semantics is clearly preserved in the process.
2.3.4 An example of core language compilation

To summarize, we defined the core language to be a small subset of the input language that excludes the following features: loops, smart assignment statements, and block statements. This is so that the core language can be treated as a simple term language with equational rewriting semantics.

To compile a full-featured input program into the core language, we apply a series of program transformation strategies. We now demonstrate this translation procedure using the naive Fibonacci program as an example.

\[
\text{fib}(n) = \begin{cases} 
  n & \text{if } (n < 2) \\
  \text{fib}(n - 2) + \text{fib}(n - 1) & \text{else}
\end{cases}
\]

The result of compiling the program \texttt{fib} is the following.

\[
\text{fib}(n) = \text{fb}(n < 2, n) \\
\text{fib1}(n) = n \\
\text{fib2}(n) = \text{fib}(n - 2) + \text{fib}(n - 1)
\]

In this process, we generate two continuation functions \texttt{fib1} and \texttt{fib2}. The original function \texttt{fib} is now only responsible for computing \( n < 2 \). \texttt{fb} is a special conditional function that encodes the two branches \texttt{fib1} and \texttt{fib2} behind the scenes. At runtime, \texttt{fb} dispatches either \texttt{fib1} or \texttt{fib2} depending the evaluation result of \( n < 2 \).

The reader might wonder why we break up a simple conditional statement like this. As a reminder, according to our language design, all function arguments in the program can be executed in parallel. If the conditional statement is not broken into separate continuations, the three redexes \( n < 2, n - 2, \) and \( n - 1 \)
in fib could be executed in parallel, which does not make sense for a conditional statement. This translation reflects the key idea of continuation: to impose an execution order.

If for some reason the programmer wants to enforce a sequential order with regard to two parallel redexes, such as \( \text{fib}(n - 2) \) and \( \text{fib}(n - 1) \) in \( \text{fib2} \), the programmer can rewrite the function \( \text{fib2} \) in the following way.

\[
\ldots
\]

\[
\text{fib2}(n) = \text{fib3}(n, \text{fib}(n - 2))
\]

\[
\text{fib3}(n, \text{left}) = \text{left} + \text{fib}(n - 1)
\]

One can see that in this way, \( \text{fib}(n - 2) \) is forced to be evaluated before \( \text{fib}(n - 1) \).

As a reminder, the same idea is used to compile a smart assignment. Take the following code fragment from the binary search for example.

\[
\ldots
\]

\[
\ldots
\]

\[
\text{mid} \leftarrow \text{sum} / 2;
\]

\[
\text{if} \ (n < a[\text{mid}])
\]

\[
\text{then} \ \text{bsearch}(i, \text{mid}, a, n)
\]

\[
\text{else} \ \text{bsearch}((\text{mid} + 1, j, a, n)
\]

We want to eliminate the assignment statement \( \text{mid} \leftarrow \text{sum} / 2 \) while preserving the program’s semantics. To do this, we break the code up into two functions such that \( \text{sum} / 2 \) is executed first as an argument of the generated continuation \( \varepsilon \). In the result, \( \varepsilon \) is a new, top-level function with a unique name. All local variables become function arguments of \( \varepsilon \) and automatically bind the free occurrences of themselves in the continuation.
2.4 From Core Language to Token Rewrite Rules

After obtaining a core language program using the previous compilation step, we can translate the core language program into token rewrite rules straightforwardly.

- Constants and numbers are translated to individual tokens that store their literal values.

- Tuples and arrays are allocated on the heap, a different memory area than the rewriting buffer. Their buffer representation is the same as a number token, except the number encodes their location on the heap.

- Function calls and built-in operator expressions are translated into Polish notation form and then split into individual tokens.

- Conditional statements are treated as special functions with two possible branches. The runtime dispatches different branches based on the evaluation result of the conditional parameter.

After this compilation step, token rewrite rules representing the program are produced. Then, stream rewriting can be applied to evaluate the program.

So far we detailed a compilation procedure that yields a core language program. The latter is then transformed into a set of token rewrite rules before the runtime can execute the program. Tokens are symbols that occupy one unit of space on the rewrite buffer. Denote the set of all tokens as $T$. Define a function $[\underline{\hspace{1cm}}]$
with a result type $\langle \mathcal{T}^*, 2\mathcal{T}^* \rightarrow \mathcal{T}^* \rangle$. This function maps a core language expression to a pair $\langle \_ , \_ \rangle$. The first item is a token list. The second item is a set of token rewrite rules.

Because terms have indeterminate shapes and are destructive, they must be stored in memory and represented in the main program by pointers to those memory locations. In the context of stream rewriting, this implies that the only data types represented on the buffer are the boolean symbols, natural numbers, and constants. For the sake of demonstration, we assume we are only dealing with natural numbers as the permitted data type. Boolean symbols are represented by numbers 0 and 1. We also only consider unary functions and tuples. A few additional bookkeeping is required to match against variadic-argument tuples and functionals.

The map $\llbracket \_ \rrbracket$ is formally defined in Fig 2.7. The first item in the resulting pair is accessed by $\llbracket \_ \rrbracket_0$. This item is the token representation of the input core expression. The second item in the pair is accessed by $\llbracket \_ \rrbracket_1$. This item is a set of token rewrite rules induced by recursively translating all sub-expressions of the input. A token is represented by the symbol $\llbracket \_ \rrbracket$. The juxtaposition of tokens (e.g., $\llbracket b \rrbracket \llbracket n \rrbracket \llbracket 0 \rrbracket$) can be seen as a token list, which also reflects the ordering of the tokens on the buffer from left to right. One can see that given a core language program $P$, the initial entry expression (aka. the "main" procedure call) is $\llbracket P \rrbracket_0$, and the set of all rewrite rules is $\llbracket P \rrbracket_1$.

2.5 Extensions

For programmer convenience, we introduced several minor features in our current implementation that do not belong to the set of first-class RE-LANG features defined in previous sections. We call these features, plus any additional set of features, extensions. Extensions are either implemented as a library (i.e. as RE-LANG programs themselves) or are directly delegated to the final compilation target language, which in our case is C++.  

32
\[ P : Program \] = 
\langle [F]_0, \bigcup \{ [FN]_1 \mid (FN : Function) \in P \} \rangle \text{ where } (F : Functional) \in P

\[ \text{if } (X : Term) \text{ then } (Y : Term) \text{ else } (Z : Term) : Conditional \] = 
\langle \beta [n][X]_0, \\
\{ \beta [n][0] \to [Y]_0, [\beta [n][1] \to [Z]_0 \cup [Y]_1 \cup [Z]_1 \}
\rangle 
\text{ where } \beta \text{ is a unique label, and } n \text{ is the free variable of the expression}

\[ (c : Const) (n : Var) = (T : Term) : Function \] = \langle \_, \{ [c][n] \to [T]_0 \cup [T]_1 \}

\[ (c : Const) (T : Term) : Functional \] = \langle [c][T]_0, [T]_1 \}

\[ (T : Term) : STuple \] = \langle [\alpha [T]_0, [T]_1 \rangle 
\text{ where } \alpha \text{ allocates memory that stores the following token}
\text{ and returns the memory location}

\[ (\ldots (s : Term)) : DTuple \] = \langle [\check{\alpha } [T]_0, [T]_1 \rangle 
\text{ where } \check{\alpha } \text{ allocates memory of size specified by the following token}
\text{ and returns the memory location}

\[ (A : Term) (I : Term) : Projection \] = \langle [\phi [A]_0[I]_0, [A]_1 \cup [I]_1 \rangle 
\text{ where } \phi \text{ accesses the memory content at location } A \text{ and offset } I

\[ \text{Dest } (A : Term, I : Term, X : Term) : Destruction \] = 
\langle [\delta [A]_0[X]_0[I]_0, [A]_1 \cup [I]_1 \cup [X]_1 \rangle
\text{ where } \delta \text{ replaces the memory content at location } A \text{ and offset } I \text{ with } X

\[ (A : Term) \circ (B : Term) \] = 
\langle [\circ [B]_0[A]_0, [A]_1 \cup [B]_1 \rangle \text{ where } \circ \text{ is some binary operator}

\[ pr : Primitive \] = \langle [pr], \emptyset \rangle

Figure 2.7: Definition of the rewrite rule generator function. Every conditional is given a unique label
and translated into two rules corresponding to the two branches. This is necessary for disambiguating
nested expressions. The variable name \( n \) does not matter, assuming unary functions and no free variables.
Variable tokens should be understood as "holes." \( \alpha, \check{\alpha}, \phi, \delta \) are native runtime functions
that perform memory manipulation.
fib(n) = if n < 2 then fib1(n) 
else fib2(n)

fib1(n) = n

fib2(n) = fib(n - 2) + fib(n - 1)

main() = fib(2)

\[ \downarrow \text{compile} \]

\[
\begin{array}{c}
\text{f0} & \rightarrow & \text{fb} < 2 \\
\text{fb} 1 & \rightarrow & \\
\text{fb} 0 & \rightarrow & + \text{f0} - 2 \text{f0} - 1 \\
\end{array}
\]

\[ \downarrow \text{execute} \]

\[
\begin{array}{c}
\text{f0} & \rightarrow \text{main} = \text{fib}(2) \\
\text{fb} 2 & < 1 \\
\text{fb} 2 0 & \\
+ \text{f0} - 2 2 \text{f0} - 1 \\
+ \text{f0} 0 \text{f0} 1 \\
+ \text{fb} 0 < 0 \text{fb} 1 < 1 \\
+ \text{fb} 0 1 \text{fb} 1 1 \\
+ 0 1 \\
\end{array}
\]

Figure 2.8: Using the naive Fibonacci number function as an example, the \textit{f0} token is the unique name for the \textit{fib} function. The \textit{fb} token is the unique name for the if-else branching statement in \textit{fib}. Per our compilation strategy presented earlier, they are compiled into two separate rewriting rules representing each resulting branch of the if-else based on the value of the condition at runtime.
while $T_{\text{cond}}$

$T_{\text{body}}$  

$TT$

$\text{f}(\text{FV}(T_{\text{body}})) =$

if $!T_{\text{cond}}$

then $\text{FV}(T_{\text{body}})$

else

$T_{\text{body}}$

$\text{f}(\text{FV}(T_{\text{body}}))$

Figure 2.9: A while loop is eliminated by transforming it into a self-recursive function call. $\text{f}$ is a unique function name.

2.5.1 Loops

Iterative structures such as for and while loops are commonly seen in most programming languages. In imperative languages, loops are typically supported by compiling them into comparison and jump operations at the instruction set level. We can simulate loops in the core language by creating a self-recursive function that encodes the loop body. This refers to the strategy in Fig 2.9

To see why this transformation is correct, consider $\text{f}$ as a procedure that iterates on the variable set $\text{FV}(T_{\text{body}})$ by mutating them in the $T_{\text{body}}$. When the terminating condition $T_{\text{cond}}$ is satisfied, $\text{f}$ returns all the current values of $\text{FV}(T_{\text{body}})$ in a tuple. The tuple is then deconstructed with the multi-assignment operator, and respective local variables are restored in the original context. As long as $T_{\text{cond}}$ is not satisfied, $\text{f}$ executes the loop body again. It then recursively calls itself to repeat the process.
2.5.2 Concurrent primitives

The first set of extensions is hardware primitives such as CAS and MFence operations used in concurrent and parallel programming to ensure mutual exclusion and memory consistency. These features are implemented in RE-LANG by directly compiling them into C++ OpenMP directives such as `omp flush` and `omp atomics`. So far, this is the most portable and efficient way of implementing these primitives since the specific instructions differ across architectures and the best practice of generating these instructions is already embedded in existing compiler infrastructures.

RE-LANG exposes these extensions by providing built-in functions such as `cas(...)` for CAS and `mfence()` for MFence. The use of these functions are self-explanatory and resembles widely practiced pseudocode convention in parallel and concurrent programming.

2.5.3 Parallel loops

In parallel programming, particularly interesting and useful are parallel loop constructs such as parallel `forall` loops and parallel `forreduce` loops. A parallel `forall` loop has the semantics of equally partitioning the number of loop items to execute on multi-processors. This is the canonical approach to composing structurally parallel programs such as matrix multiplication, where the computation is completely independent across all loop items. A parallel `forreduce` loop has a combining semantics: Given an associative operator (e.g. + for integers), the `forreduce` loop computes the sum of all the items associated with the `forreduce` loop. This is frequently used to accelerate the aggregation of a large number of results, such as computing the average number of a vector.

We implemented the `forall` and `forreduce` loops in RE-LANG as library functions following their semantic definition, which admits standard implementations: The `forall` loop divides \( n \) loop items into \( P \) chunks at runtime, where each chunk has up to \( \lceil n/P \rceil \) loop item workload. The `forreduce` loop combines \( n \) loop items bottom-up in a neighborly pair-wise fashion, and the total number of combining steps is up
to $\lceil \log(n) \rceil$. The forall loop potentially introduces a high load imbalance because the forall divided
chunks usually constitute coarse-grained tasks and may differ vastly in workload if each chunk is not further
parallelized. To address this issue, we also implement a forevery loop construct. The forevery loop has
the same semantics as forall loop except that it divides the $n$ loop items into $n$ parallel tasks in a binary
divide-and-conquer fashion, as opposed to $P$ parallel chunks. This introduces much finer-grained parallelism
and is crucial in accelerating several benchmarks, which we will present in the coming chapter. Nevertheless,
the forevery loop induces a runtime overhead proportional to the number of loop items. Hence, it needs to
be used with discretion when the number of loop items is very large.

We are aware that extensive studies exist on the topic of loop parallelization, but since RE-LANG’s focus
is on task parallelism and programming language design, further effort in loop parallelization is out of the
scope of this thesis. In the future, there is potential to introduce more complex parallel loop semantics that
accelerates specific workflows according to past research in this domain.

2.6 The Stream Rewriting Execution Model

Previously we laid out detailed compilation steps for compiling the RE-LANG input language to token
rewrite rules. In this section, we introduce the stream rewriting runtime model for efficient execution of the
token rewrite rules.

2.6.1 Stream Rewriting

Term rewriting is a general formalism for computation (Willem & Klop 1990). In a term rewriting system,
terms represent data, and rewriting rules are functions that transform terms.

In stream rewriting, terms are represented as contiguous tokens on a stream. A stream refers to the
abstract concept of a linear storage space infinite in size. A buffer is a practical implementation of the stream,
Algorithm 1 The stream rewriting algorithm (one iteration)

```plaintext
procedure STREAM-REWRITE(data)
    // data: THREAD-LOCAL BOOKKEEPING
    // data.rBuff: THE READ (INPUT) BUFFER
    // data.wBuff: THE WRITE (OUTPUT) BUFFER

    for (i = 0; i < data.rBuffSize and not rebalance; i++)
        // REWRITE FROM data.rBuff TO data.wBuff
        data.REWRITE(data.rBuff[i])
    end for

    data.SWAP-BUFFER()
end procedure
```

usually as a fixed-sized array. There are two buffers in a stream rewriting machine: the input buffer and the output buffer.

Rewrite rules are compiled into token transformers. A stream rewriting machine sequentially scans the input buffer and pattern-matches against tokens to detect rewrite opportunities. The resulting tokens are written onto the output buffer. An iteration finishes when all tokens on the input buffer are consumed. The two buffers are then exchanged, and the procedure repeats. The computation terminates when a fixpoint is reached. An example of this evaluation mechanism is illustrated in Fig 2.8 and Algorithm 1.

Middendorf et al.(Middendorf, Bobda & Haubelt 2012, Middendorf & Haubelt 2014, Middendorf, Zebelein & Haubelt 2013) have previously used stream rewriting for hardware-level parallelism, but their works differ from ours in significant ways.

- Their works are purely hardware-based, while ours is purely software.
- They use predefined initial partition sizes for all cores in the first iteration and allow partitions to grow based on runtime feedback. However, our system uses global scheduling and has no explicit adaptive behavior.
• Their architecture is complex with five processing stages that partition and combine token streams. On the other hand, our software system has processors running in largely independent sessions. Our system has no extra stream processing cost since global scheduling constitutes one computational step.

• They use direct buffering for stream rewriting, while we use indirect buffering. The latter is asymptotically superior.

We are also interested in a purely software approach because hardware solutions on field-programmable gate arrays (FPGA) tend to be much slower than their contemporary processor counterparts in practice. One important reason is the sheer lack of processing power of off-the-shelf FPGA products. Mid-to-high-end consumer CPUs run at a typical speed of more than 3GHz, while the maximum possible frequency of most FPGAs is strictly lower than 500MHz within the same price range. Evaluating general recursion also requires significant memory access(Nikhil et al. 1990, Saint-Mleux, Feeley & David 2006, Sklyarov 1999, Zhai, Townsend, Lairmore, Kim & Edwards 2015). However, FPGAs commonly lack large, fast on-chip memory, a deficiency much work was dedicated to address(Fang, Mulder, Hidders, Lee & Hofstee 2020, Winterstein, Fleming, Yang & Constantinides 2016). For the time being, it is unclear whether FPGA has a practical advantage over software approaches for evaluating general recursion. Partly for this reason, the primary use of FPGA has historically been centered around domain-specific, dataflow applications with an emphasis on exploiting parallelism(Nane, Sima, Pilato, Choi, Fort, Canis, Chen, Hsiao, Brown, Ferrandi et al. 2015).

2.6.2 Parallel Stream Rewriting

One key observation on stream rewriting is that it can operate on an arbitrary slice of the stream, not necessarily on the whole stream. This leads to a natural algorithm for parallel reduction. First, the machine partitions the stream into slices according to a scheduling algorithm. Then it dispatches each slice to a different processor for parallel execution (Fig. 2.10).
However, balanced stream rewriting is tricky, even in hardware (Middendorf, Bobda & Haubelt 2012, Middendorf, Zebelein & Haubelt 2013, Middendorf & Haubelt 2014). This previously led Middendorf et al. to use tiny block sizes for each rewriting core. Their design features five processing stages, including three synchronization stages. Hardware pipelining is also exploited to extract maximal performance. Due to the complex synchronization procedures, it is a challenge to transpose Middendorf’s design into an efficient software implementation.

Furthermore, although the Middendorf machine exploits parallelism, each processor only evaluates the stream naively. This is a significant drawback because naive rewriting copies all tokens in each iteration regardless of whether they are reducible. This is inefficient for most programs, not just in the asymptotic sense but also because token copying is an expensive memory operation.

2.7 Software Stream Rewriting

We seek to preserve the fundamental pleasing properties of parallel stream rewriting in a software setting. That is, token streams can be sliced and evaluated arbitrarily. At the same time, we want to avoid significant drawbacks in previous implementations and maximize performance. This leads to our software design based on the following distinct features.
2.7.1 Indirect buffering

To address the critical issue of naive stream rewriting, we use indirect buffering for both the read and write buffers as opposed to direct buffering used by the Middendorf machine.

The idea of an indirect buffer is simple. Instead of storing the tokens directly on the buffer and rewriting the entire token stream in each iteration, we only store references to the tokens. A token becomes a *redex* if all its arguments are fully evaluated. Most importantly, a reference is only added to the buffer if it points to an actual redex. Non-redex tokens are generated but stay in a separate token store until they become redexes.

Because the stream rewriter always attempts to parallelize all function arguments, a counter is maintained in each token to keep track of the number of joined arguments (Fig. 2.11). Each token also points to its parent token for returns. The returned arguments are kept in a separate memory space linked with their corresponding functional tokens and do not occupy space on the buffer. As a result, all tokens on the buffer are functional tokens. This simplifies buffer operations and improves performance.

Indirect buffering eliminates the need for copying non-redex tokens from the read buffer to the writer buffer in every iteration. Because we create and evaluate exactly the set of all redexes in each iteration and no more, indirect buffering is asymptotically efficient. On the other hand, direct buffering is asymptotically worse in general. This is crucial for both sequential performance and parallel scalability.
Indirect buffering also eliminates the need for pattern matching because all tokens on the buffer are redexes and can be reduced immediately without checking if they match a reducible pattern beforehand.

2.7.2 Specializing to stack evaluation

Relying on stream rewriting as the sole runtime mechanism has several drawbacks.

- A stream rewriter without specialized hardware support or extreme optimization is relatively slow compared to, for example, the stack machine.

- Because stream rewriting follows a breadth-first evaluation pattern, the buffer size can grow very fast, sometimes exponentially. In principle, this is not a problem because some input programs are inherently exponential (e.g., recursive Fibonacci numbers). However, an overly sized buffer can cause excessive paging and poor cache behaviors.

To address these problems, we conveniently resort to the observation that stack evaluation is a specialization of stream rewriting. This should be clear if we consider stack evaluation to be stream rewriting but only with the rightmost redex on the buffer instead of all the redexes on the buffer.

This leads to the design of a hybrid runtime evaluation mechanism. We use the stream rewriter following breadth-first evaluation to generate a sufficient number of redex tokens. When there are "enough" redexes, the processor switches into stack machine mode for maximal performance and prevents the buffer from rapid growth. After the stack machine mode exhausts the redexes below a certain threshold, the runtime switches back to stream rewriting mode to quickly generate more redexes. This process continues ad infinitum until the program terminates. A sketch implementation of the stack machine mode is shown in Algorithm 2.

The threshold for "enough" is a tunable parameter. In practice, however, we find that setting it to 1 (i.e., use stack machine mode as much as possible) is already sufficient for most programs, such as those that will be used in our evaluation benchmarks in the coming chapter.
Algorithm 2 Specializing stream rewriting to stack evaluation

procedure STACK-REWRITE(data)
  // TURN STACK MODE ON
  wBuffBackup ← data.wBuff
  wBufferSizeBackup ← data.wBuffSize
  data.wBuff ← data.rBuff
  data.wBuffSize ← data.rBuffSize
  while (enough in data.wBuffSize and not rebalance)
    data.wBuffSize ← rightmost ← data.wBuff[data.wBuffSize]
    data.REWRITE(rightmost)
  end while
  // TURN STACK MODE OFF
  data.rBuffSize ← data.wBuffSize
  data.wBuff ← wBuffBackup
  data.wBuffSize ← wBufferSizeBackup
end procedure

2.7.3 Global scheduling by stream rewriting

One particularly attractive aspect of stream rewriting is that scheduling streams is easy. In stream rewriting, there is one global stream that represents the current evaluation state (Fig. 2.8). Different parts of the stream can be partitioned equally across all processors in a parallel setting. (Fig. 2.12).

Figure 2.12: Global scheduling with four processors. Each processor attempts to rewrite ⌈10/4⌉ = 3 tokens in the conceptual total buffer into itself. The result is not guaranteed to be perfectly even because it is a product of a rewriting iteration.

In contrast to traditional work stealing schedulers, which copy tasks from one processor to another, no such data movement is necessary for scheduling in the stream rewriting machine. Each processor simply
rewrites tokens from another processor into itself. The scheduling step is equivalent to a global stream evaluation step. For the same reason, the number of scheduling steps is bounded by the length of the critical path. Therefore, the scheduler is efficient according to the work-first principle (Frigo, Leiserson & Randall 1998). A formal analysis of this will be presented in a future chapter.

Scheduling is triggered when a processor runs out of work. This sometimes causes thrashing: for strictly sequential programs, the number of redexes never exceeds one. Throughout the evaluation, some processors must naturally starve. To address this issue, we adopt a notion of stability: a processor is stable if it contains no more than one redex on its input buffer; it is unstable otherwise. An unstable processor has extra tokens that can be redistributed to other processors, while a stable processor cannot contribute to load balancing. Hence, we make it so that scheduling is triggered only if a processor runs out of work and at least one processor is unstable. This prevents thrashing in the case of sequential programs or programs with an insufficient degree of inherent parallelism, where the number of redexes is always smaller than the number of processors.

2.7.4 Join synchronization by walling

In one way, stream rewriting can be viewed as a computational tree generator in a breadth-first manner. Redexes are represented by nodes in the tree, where child redexes must be evaluated and joined back to the parent. In parallel evaluation, these redexes are distributed across different processors. For irregular programs, the placement of redexes is non-deterministic. A thread-safe counter is needed in every node to keep track of the number of finished child tasks.

Nevertheless, even a lockless counter using only atomic synchronization significantly impacts performance if this behavior is the default for every node. To minimize atomic synchronization in the system, we observe that the runtime is partitioned into sessions between global scheduling iterations. If the lifecycle of a redex is entirely within a session, it requires no synchronization because all its child nodes stay within
Figure 2.13: Stream rewriting can be viewed in terms of computational tree nodes. Note that the tree not only grows but also shrinks. After a token is reduced or returned to its parents, it is deleted from the tree.
the same processor. After the end of a session, some nodes are rescheduled onto other processors, which introduces the need to synchronize.

Using this insight, we apply the technique of **walling** to reduce the number of synchronizing nodes. On a high level, the idea is to mark all reducing tokens during global load balancing as wall nodes and properly maintain all wall nodes in different reduction scenarios. Any node to the right of the last wall does not require synchronization, while all others do. The wall is created and maintained via the following three steps illustrated in Fig. 2.13.

During global load balancing, all parent nodes of reducing nodes are marked as wall nodes. The meaning of this wall is intuitive: it demarcates the location in the computational tree where global load balancing happens. Per the mechanism of the scheduler, child nodes of the reducing nodes are distributed across different processors. Eventually, these nodes will return to their parents up the tree. The wall denotes a boundary in the tree: crossing into and beyond the wall is considered thread-unsafe since the returning nodes can come from different processors.

Once a wall is established, it can never be destroyed and must be maintained to preserve thread safety. When a wall node is synchronized and becomes a redex, it is removed from the computational tree, and new nodes are generated following its reduction. This introduces a "hole" in the wall, allowing processors to cross the wall without being notified of thread unsafety. We remedy this by marking the parent of any redex wall node as the new wall node. We call this **pushing** the wall.

In the other case, a wall node becomes a redex but generates a primitive value and returns it up the tree. If the returned node is still waiting for other children, its computation is delayed. When the returned node eventually becomes a redex, it already loses the information that a wall is crossed to reach it. This makes future returning steps between this node and the next wall up the tree thread unsafe. The solution is to mark the parent node of a returned node as a wall node. We call this **pulling** the wall.
2.8 Summary

In this chapter, we presented the compiler pipeline and key design implementations underlying the RE-LANG programming language.

On a high level, RE-LANG has two components: a frontend language and a backend runtime. The runtime is constructed based on the stream rewriting semantics: Any program is considered a set of token rewriting rules, and the set of all tokens represents the global state of the program at any given execution time step. The stream is divided and evaluated in parallel across all available processors. This evaluation is efficient since a stream rewriter can act as a stack machine by evaluating only the rightmost task. Whenever a processor runs out of tasks, it initiates a global rebalancing iteration, where all processors evaluate the global state in equal partition, distributing task workload. The walling-based synchronization allows the runtime to save atomic operation costs on task joins, thanks to the fact that in-between rebalancing iterations, each processor runs completely independently. To compile any program to token rewrite rules, the frontend language component relies on two pipeline processing steps: The first pipeline compiles the RE-LANG input language into a smaller, intermediate core language by transformation strategies aiming to eliminate smart assignment statements. The core language is then mapped to token rewrite rules in a relatively straightforward fashion.

In the next chapter, we will present a comprehensive set of benchmarks to evaluate the performance of RE-LANG.
CHAPTER 3: EVALUATION

In this chapter\(^1\), we present fifteen benchmark programs to evaluate our system in practice\(^2\). The evaluation consists of three suites. Each suite contains programs of similar characteristics for evaluating a particular aspect of our runtime system. All benchmarks are written in the parallel-by-default style in RE-LANG, except for Suite B, which utilizes parallel for loops. All benchmark programs' code are available in the Appendix.

We emphasize that the main purpose of all the benchmarks is to stress test the RE-LANG runtime so that by observing the results, programmers are able to gain confidence in the general capability of the language, particularly in how it is able to efficiently scale declarative parallel programs that generate numerous fine-grained tasks.

In other words, the goal of the benchmarks is not to construct specific parallel algorithms that compete against the state-of-the-art in raw execution time. Such a goal is out of the scope of this thesis because (1) State-of-the-art parallel algorithms are often compiled using mature compilers such as GCC or LLVM, which have undergone decades of optimization and improvement. Comparatively, our compiler is still highly experimental. (2) State-of-the-art parallel algorithms often contain tricks specific to the domain and architectures, such as GPU offloading for matrix multiplication, and therefore can obtain a massive performance advantage. On the other hand, the primary purpose of RE-LANG is to provide an infrastructure for general programming and does not employ domain-specific accelerating heuristics.

\(^1\)© 2023 IEEE. Reprinted, with permission, from T.Tao, “Synchronization Efficient Scheduling of Fine-grained Irregular Programs”, 2023 31st Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2023

\(^2\)CPU: Intel Xeon E5-2699A v4 2.40GHz, OS: RedHat 7 x86_64, Compiler: GCC 11.2.0. Test up to 16 cores/one socket. For each parallelism degree, run the benchmark ten times and take the average.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Suite A: Divide and conquer, computation-bound</strong></td>
<td></td>
</tr>
<tr>
<td>fib (n = 33)</td>
<td>Naive Fibonacci recursive program</td>
</tr>
<tr>
<td>knapsack (n = 30, value = 500)</td>
<td>Recursive knapsack solver</td>
</tr>
<tr>
<td>takeuchi (n = 11)</td>
<td>3-ary recursive program of exponential complexity</td>
</tr>
<tr>
<td>uts (n = 2,598,802)</td>
<td>Unbalanced tree search</td>
</tr>
<tr>
<td>tsp (n = 17)</td>
<td>Parallel Held–Karp traveling salesman solver</td>
</tr>
<tr>
<td><strong>Suite B: Structurally parallel</strong></td>
<td></td>
</tr>
<tr>
<td>add (n = 10,000,000)</td>
<td>Parallel add</td>
</tr>
<tr>
<td>matrixMult (n = 200)</td>
<td>Standard nested loop implementation</td>
</tr>
<tr>
<td>primes (2 - 10,000)</td>
<td>Simple for-based primality test</td>
</tr>
<tr>
<td>prefixSum (n = 1,000,000)</td>
<td>Parallel prefix sum algorithm</td>
</tr>
<tr>
<td>catalan (n = 14)</td>
<td>Recursive catalan number program</td>
</tr>
<tr>
<td>bellmanFord (n = 200)</td>
<td>Parallel Bellman-Ford algorithm on dense matrix</td>
</tr>
<tr>
<td><strong>Suite C: Divide and conquer, memory-heavy</strong></td>
<td></td>
</tr>
<tr>
<td>mergeSort (n = 400,000)</td>
<td>Standard merge sort implementation</td>
</tr>
<tr>
<td>nqueens (n = 10)</td>
<td>Recursive n-queen solver</td>
</tr>
<tr>
<td>coins (n = 6, value = 777)</td>
<td>Compute the number of ways to pay</td>
</tr>
<tr>
<td>mapReduce (n = 10,000,000, w=25,000)</td>
<td>Map-reduce for word frequency count</td>
</tr>
</tbody>
</table>
For reference in terms of raw performance characteristics, we provide a standalone benchmark that compares the sequential, single-core performance of RE-LANG against mainstream programming language runtimes towards the end of this chapter. We will further discuss the raw performance aspects of RE-LANG and their future implications in the concluding chapter.

### 3.1 Suite A

The first suite of benchmarks consists of four purely computational divide-and-conquer programs. Among them, there is the classic Fibonacci recursive program. The takeuchi program (McCarthy 1979) is similar to Fibonacci, except it features 3-ary recursion, while Fibonacci has binary recursion. The uts benchmark traverses a randomly generated binary tree with 2,598,802 nodes. The tsp benchmark is the Held-Karp dynamic programming algorithm for solving the traveling salesman problem. The graph paths in the Held-Karp algorithm are encoded as binary numbers and passed as integer arguments. This allows us to parallelize every recursive call. The executable RE-LANG code of all benchmarks are available verbatim in the Appendix.

The main purpose of this suite is to gauge the overhead of fine-grained tasks. All five programs generate an excessive number of tasks. Since they are also computation-bound, the overhead cannot be hidden away and will directly affect scalability.

The experimental results are summarized in Fig. 3.1 and Table 3.2. We can see that all benchmarks scale close to ideal. This shows that our runtime has negligible overhead, even for a large number of fine-grained tasks.

### 3.2 Suite B

The primary purpose of the second suite is to demonstrate forall and forevery loops in RE-LANG. Hence, the suite mainly consists of embarrassingly parallel programs that isolate the effect of parallel for
Figure 3.1: Suite A scalability results
Table 3.2: Suite A execution time data

<table>
<thead>
<tr>
<th></th>
<th>fib, n=33</th>
<th>knapsack, w=500, n=30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 4 8 16</td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>958</td>
<td>481 244 124 62</td>
<td>6542 3327 1684 833 419</td>
</tr>
<tr>
<td>954</td>
<td>482 245 122 62</td>
<td>6559 3313 1690 838 415</td>
</tr>
<tr>
<td>956</td>
<td>480 242 121 61</td>
<td>6558 3327 1681 839 418</td>
</tr>
<tr>
<td>959</td>
<td>484 244 121 62</td>
<td>6536 3373 1677 835 417</td>
</tr>
<tr>
<td>953</td>
<td>481 241 121 62</td>
<td>6576 3334 1668 835 418</td>
</tr>
<tr>
<td>954</td>
<td>489 245 121 62</td>
<td>6549 3323 1683 832 418</td>
</tr>
<tr>
<td>953</td>
<td>480 241 121 62</td>
<td>6540 3324 1681 838 418</td>
</tr>
<tr>
<td>954</td>
<td>483 241 121 62</td>
<td>6586 3355 1673 839 419</td>
</tr>
<tr>
<td>954</td>
<td>481 241 122 62</td>
<td>6625 3351 1691 839 416</td>
</tr>
<tr>
<td>976</td>
<td>482 244 122 62</td>
<td>6543 3323 1672 841 417</td>
</tr>
<tr>
<td><strong>957.1</strong></td>
<td><strong>482.3</strong></td>
<td><strong>242.8</strong></td>
</tr>
<tr>
<td>1.00</td>
<td>1.98</td>
<td>3.94</td>
</tr>
<tr>
<td></td>
<td>takeuchi, n=11</td>
<td>uts, n=2,598,802</td>
</tr>
<tr>
<td>1 2 4 8 16</td>
<td>1 2 4 8 16</td>
<td></td>
</tr>
<tr>
<td>2331 1166 596 301 166</td>
<td>836 425 216 108 54</td>
<td></td>
</tr>
<tr>
<td>2400 1174 601 302 165</td>
<td>873 427 225 108 54</td>
<td></td>
</tr>
<tr>
<td>2345 1167 592 300 170</td>
<td>851 428 218 107 55</td>
<td></td>
</tr>
<tr>
<td>2335 1166 594 301 165</td>
<td>837 428 216 108 55</td>
<td></td>
</tr>
<tr>
<td>2328 1174 602 301 164</td>
<td>835 425 216 108 55</td>
<td></td>
</tr>
<tr>
<td>2346 1168 590 302 165</td>
<td>853 430 215 108 55</td>
<td></td>
</tr>
<tr>
<td>2370 1166 598 303 166</td>
<td>841 427 217 108 55</td>
<td></td>
</tr>
<tr>
<td>2357 1170 597 301 166</td>
<td>845 436 216 109 55</td>
<td></td>
</tr>
<tr>
<td>2346 1221 591 301 167</td>
<td>887 425 218 107 55</td>
<td></td>
</tr>
<tr>
<td>2333 1182 593 300 167</td>
<td>833 432 217 108 55</td>
<td></td>
</tr>
<tr>
<td><strong>2349.1</strong></td>
<td><strong>1175.4</strong></td>
<td><strong>595.4</strong></td>
</tr>
<tr>
<td>1.00</td>
<td>2.00</td>
<td>3.95</td>
</tr>
<tr>
<td></td>
<td>tsp, n=17</td>
<td></td>
</tr>
<tr>
<td>1 2 4 8 16</td>
<td>1 2 4 8 16</td>
<td></td>
</tr>
<tr>
<td>3121 1568 803 413 208</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3122 1572 791 402 207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3162 1569 802 401 206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3124 1589 803 401 213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3119 1569 794 403 207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3114 1580 814 408 208</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3124 1584 828 403 205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3127 1587 816 410 207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3110 1566 807 401 207</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3112 1568 802 409 206</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3123.5</strong></td>
<td><strong>1575.2</strong></td>
<td><strong>806</strong></td>
</tr>
<tr>
<td>1.00</td>
<td>1.98</td>
<td>3.88</td>
</tr>
</tbody>
</table>

52
loops. In this regard, benchmarks such as add, matrixMult, and prefixSum are well-known and standard. The only exception is the Catalan number program (catalan\(^3\)), which utilizes both recursive and for-loop parallelism. The bellmanFord benchmark is a straightforward parallelized version of the Bellman-Ford single-source shortest path algorithm, achieved by performing all edge relaxation in parallel up to each outer loop iteration. The executable RE-LANG code of all benchmarks are available verbatim in the Appendix.

```c
// The recursive Catalan number program
main() = catalan(14)

catalan(n) = if (n <= 1) then 1 else
    o ← (...n);

// Parallel forall loop
forall i = 0 to (n - 1)
    // Recursive parallel calls
    o[i] ← catalan(i) * catalan(n - i - 1);

    sum ← 0;
    for i = 0 to (n - 1)
    sum ← sum + o[i];

free(o);
sum
```

The experimental results are summarized in Fig. 3.2. We can see that all benchmarks scale close to ideal except the primes and Catalan programs. The explanation is that both benchmarks contain irregular, coarse-grained tasks given the use of forall loops. To confirm this, we rewrote the two programs using forevery loops, rerun the experiments, and summarize the results in Fig. 3.3. From Fig. 3.3, we can see that both benchmarks now exhibit much better scalability, showing that our hypothesis is correct.

\(^3\)The program naively computes the \(i\)-th Catalan number \(C_i\) following the recursive definition \(C_0 = 1, C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}\), for \(n \geq 0\).
Figure 3.2: Suite B scalability results
Figure 3.3: primes and catalan using forevery loop
<table>
<thead>
<tr>
<th>add, n=10,000,000</th>
<th>matrixMult, 200x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 4 8 16</td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>2209 1135 562 283 144</td>
<td>1341 671 338 170 88</td>
</tr>
<tr>
<td>2243 1142 561 282 142</td>
<td>1332 667 337 168 86</td>
</tr>
<tr>
<td>2207 1142 562 284 142</td>
<td>1340 670 335 168 85</td>
</tr>
<tr>
<td>2267 1143 570 283 142</td>
<td>1331 672 336 168 85</td>
</tr>
<tr>
<td>2217 1159 568 282 142</td>
<td>1334 674 335 168 85</td>
</tr>
<tr>
<td>2209 1134 561 283 143</td>
<td>1331 670 335 168 85</td>
</tr>
<tr>
<td>2220 1140 572 284 142</td>
<td>1347 668 335 168 85</td>
</tr>
<tr>
<td>2215 1133 565 282 144</td>
<td>1349 668 335 168 85</td>
</tr>
<tr>
<td>2253 1148 564 284 142</td>
<td>1338 668 335 168 85</td>
</tr>
<tr>
<td>2219 1160 567 284 143</td>
<td>1331 670 336 168 85</td>
</tr>
<tr>
<td><strong>2225.9</strong></td>
<td><strong>1143.6</strong></td>
</tr>
<tr>
<td><strong>1.00</strong></td>
<td><strong>1.95</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prefixSum, n=1,000,000</th>
<th>primes (forevery), 2-10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 4 8 16</td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>1614 817 424 214 115</td>
<td>741 370 186 94 49</td>
</tr>
<tr>
<td>1610 814 424 210 112</td>
<td>740 369 185 93 48</td>
</tr>
<tr>
<td>1610 813 416 211 112</td>
<td>738 368 185 93 48</td>
</tr>
<tr>
<td>1615 814 415 210 111</td>
<td>733 370 185 93 48</td>
</tr>
<tr>
<td>1610 815 415 211 111</td>
<td>744 369 185 93 48</td>
</tr>
<tr>
<td>1611 815 415 211 111</td>
<td>739 368 185 93 47</td>
</tr>
<tr>
<td>1615 815 417 211 111</td>
<td>744 369 188 93 47</td>
</tr>
<tr>
<td>1610 820 417 212 112</td>
<td>738 369 185 93 48</td>
</tr>
<tr>
<td>1609 818 422 210 112</td>
<td>736 371 186 93 47</td>
</tr>
<tr>
<td>1613 819 418 216 111</td>
<td>736 371 185 93 48</td>
</tr>
<tr>
<td><strong>1611.7</strong></td>
<td><strong>816</strong></td>
</tr>
<tr>
<td><strong>1.00</strong></td>
<td><strong>1.98</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>catalan (forevery), n=14</th>
<th>bellmanFord, n=200</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 4 8 16</td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>977 493 252 125 65</td>
<td>2468 1218 623 335 179</td>
</tr>
<tr>
<td>968 480 251 123 64</td>
<td>2404 1224 654 325 187</td>
</tr>
<tr>
<td>966 484 249 124 64</td>
<td>2444 1223 634 327 177</td>
</tr>
<tr>
<td>1038 482 248 125 63</td>
<td>2495 1223 627 326 178</td>
</tr>
<tr>
<td>962 482 252 123 64</td>
<td>2431 1226 621 327 177</td>
</tr>
<tr>
<td>957 480 248 123 64</td>
<td>2415 1214 631 326 178</td>
</tr>
<tr>
<td>958 490 249 123 63</td>
<td>2450 1241 625 332 178</td>
</tr>
<tr>
<td>961 489 252 122 64</td>
<td>2451 1229 625 327 178</td>
</tr>
<tr>
<td>974 493 249 142 63</td>
<td>2424 1218 628 326 178</td>
</tr>
<tr>
<td>970 482 249 123 63</td>
<td>2427 1217 625 323 178</td>
</tr>
<tr>
<td><strong>973.1</strong></td>
<td><strong>485.5</strong></td>
</tr>
<tr>
<td><strong>1.00</strong></td>
<td><strong>2.00</strong></td>
</tr>
</tbody>
</table>

| 2440.9 | 1223.3 | 629.3 | 327.4 | 178.8 |
| 1.00   | 2.00   | 3.88  | 7.46  | 13.65 |
3.3 Suite C

The last suite consists of divide-and-conquer programs involving extensive memory reading and writing. For example, mergeSort allocates memory when merging the results of recursive calls, which is the standard implementation. The nqueens benchmark replicates the state of the chessboard on a fork, and the coins benchmark relies on linked list operations to keep track of the state of the wallet. The mapReduce benchmark is a shared-memory version of the word frequency count algorithm following the map-reduce processing paradigm (Dean & Ghemawat 2008). The benchmark is based on a paragraph of randomly generated text of length 10,000,000 using 25,000 words encoded as positive integers. The algorithm proceeds in two phases: the map phase and reduce phase. In the map phase, all processors summarize the word count of the text into independent hash maps across $P$ equal partitions of the text. The reduce phase merges all the hash maps in $\log(P)$ steps in a fashion similar to the parallel add algorithm. In reality, word count can be unevenly distributed causing workload imbalance and making the benchmark difficult to parallelize. However, our random text is uniformly generated and should allow the benchmark to scale close to ideal. The executable RE-LANG code of all benchmarks are available verbatim in the Appendix.

The memory-heavy nature of the benchmarks in suite C renders them closer to real-world algorithms. For this reason, the purpose of this suite is to provide insight into how RE-LANG performs in a context closer to real-world software programs.

```plaintext
// The coins program
pay(val, coins) =
  if (val = 0) then 1 else ( 
    if (coins = Nil) then 0 else 
      (head, tail) ← (coins[0], coins[1]); (c, q) ← head; 
      if (c > val) then pay(val, tail) else 
        if (q = 1) then
          // Recursive parallel calls
```
pay(val - c, tail) + pay(val, tail) else

// Use functional linked list to manage state.
newhead ← (c, q - 1); coinsx ← (newhead, tail);
// Recursive parallel calls
(l, r) ← (pay(val - c, coinsx), pay(val, tail));
free(newhead); free(coinsx);
1 + r
)

The experimental results are summarized in Fig. 3.4. We can see that all benchmarks scale close to ideal except for mergeSort and mapReduce. However, this is expected since mergeSort only has $\log(n)$ degree of parallelism because a significant portion of the program is the sequential merger routine.

The result of mapReduce is also expected because the reduction phase has a larger operational constant cost than the map phase. We can estimate the ratio of this constant difference by examining $r = \frac{n/P}{w \log(P)}$, where $n/P$ estimates the critical path length in the map phase, and $w \log(P)$ estimates the critical path length in the reduce phase. This estimate assumes a perfect division of the input text of length $n$ across $P$ processors. Further, it assumes the word frequencies are equally distributed across the input text so that the map phase yields $P$ hash maps of size roughly $w$.

To estimate $r$ on our evaluation machine, we constructed an emulator program running on a single core such that it first conduct $n/P$ number of hash map inserts and then performs $\log(P)$ times two-way hash-map merge of equal size $w$. By executing this program with parameters $P = 16, w = 25,000, n = 10,000,000$, we estimate that $r = \frac{\text{Time spent in map phase emulation}}{\text{Time spent in reduce phase emulation}} \approx 4.79$. As a result, we estimate that the average constant cost in the reduce phase is roughly 1.88 times more expensive than the map phase under these parameters. This is because $\frac{n/P}{1.88w \log(P)} \approx 4.79$, given $P = 16, w = 25,000, n = 10,000,000$.

Based on this estimate, we conclude that the maximum number of achievable parallelism for the entire mapReduce program under these parameters is $P = \frac{\text{Total work}}{\text{Critical path length}} = \frac{T_1}{T_\infty} \approx \frac{n + 1.88wp}{n/P + 1.88w \log(P)} \approx$.
Figure 3.4: Suite C scalability results
Table 3.4: Suite C execution time data

<table>
<thead>
<tr>
<th>nqueens, n=10</th>
<th>mergeSort, n=400,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 4 8 16</td>
</tr>
<tr>
<td>2440</td>
<td>1229 623 310 158</td>
</tr>
<tr>
<td>2448</td>
<td>1225 621 310 157</td>
</tr>
<tr>
<td>2462</td>
<td>1226 627 308 157</td>
</tr>
<tr>
<td>2511</td>
<td>1227 621 315 156</td>
</tr>
<tr>
<td>2458</td>
<td>1231 621 308 156</td>
</tr>
<tr>
<td>2446</td>
<td>1235 625 310 156</td>
</tr>
<tr>
<td>2449</td>
<td>1228 623 310 156</td>
</tr>
<tr>
<td>2460</td>
<td>1246 621 309 156</td>
</tr>
<tr>
<td>2448</td>
<td>1222 621 309 156</td>
</tr>
<tr>
<td>2468</td>
<td>1229 631 310 157</td>
</tr>
<tr>
<td>2459</td>
<td>1229.8 623.4 309.9 156.5</td>
</tr>
<tr>
<td>1.00</td>
<td>2.00 3.94 7.93 15.71</td>
</tr>
<tr>
<td>coins, large</td>
<td></td>
</tr>
<tr>
<td>18516</td>
<td>9360 4689 2343 1200</td>
</tr>
<tr>
<td>18565</td>
<td>9329 4648 2357 1196</td>
</tr>
<tr>
<td>18592</td>
<td>9598 4700 2380 1183</td>
</tr>
<tr>
<td>18524</td>
<td>9339 4660 2394 1202</td>
</tr>
<tr>
<td>18521</td>
<td>9416 4678 2371 1191</td>
</tr>
<tr>
<td>18567</td>
<td>9484 4706 2354 1184</td>
</tr>
<tr>
<td>18542</td>
<td>9299 4653 2343 1197</td>
</tr>
<tr>
<td>18530</td>
<td>9334 4686 2341 1183</td>
</tr>
<tr>
<td>18499</td>
<td>9535 4686 2378 1186</td>
</tr>
<tr>
<td>18513</td>
<td>9263 4658 2353 1196</td>
</tr>
<tr>
<td>18536.9</td>
<td>9395.7 4676.4 2361.4 1191.8</td>
</tr>
<tr>
<td>1.00</td>
<td>1.97 3.96 7.85 15.55</td>
</tr>
</tbody>
</table>
The actual scaling reported for mapReduce is approximately 12.03 (Table 3.4), which is close to ideal. The remainder of the difference can be attributed to the fact that the reduce phase loses half of the parallelism after each iteration.

### 3.4 Sequential Performance

The global scheduling mechanism requires all processors to pause their current work. This is because the synchronization iteration requires the processors to execute tasks in a breadth-first manner. Crucially, the pausing must be performed immediately—any delay adds to the overhead. For reference, we evaluate the single-core, sequential performance of our rewriting-based system and compare it against a few mainstream programming language runtimes\(^4\). For the benchmark, we use the Ackermann function executed with parameters \(m = 3, n = 10\). The result is summarized below.

\(^4\)CPU: i5-9600K 4.70GHz, OS: WSL2. Run ten times and take the average. Code and just-in-time optimizations are disabled for fair play because our compiler is non-optimizing.
As we can see, although our system cannot yet compete against fast, native languages such as C++, it is already faster than a few mainstream programming languages such as Java.

3.5 Summary

In this chapter, we presented a comprehensive set of fifteen parallel benchmarks aiming to evaluate the runtime performance of RE-LANG. The fifteen benchmarks consist of three suites: Suite A represents purely computational workloads, suite B represents structurally parallel workloads written in loop-style parallelism, and suite C represents more complex, real-world programs that contain heavy memory read and write operations. Each benchmark result is collected by averaging the wall time duration across ten executions for each parallel degree up to 16 cores. Both raw execution time and scalability charts are provided in detail for reference. The observation from our experimental results is that all benchmarks scale close to ideal.

Curious readers may wonder how the performance of RE-LANG compares with existing task-parallel systems such as Cilk or Thread Building Blocks (TBB). We did not provide such experiments for several reasons: (1) Most of the existing task-parallel frameworks have no open-sourced implementation, nor do the authors provide code ready for implementation, often due to the complex and architecture-sensitive nature of such projects. (2) The raw execution time between RE-LANG and existing frameworks built using mature compilers such as LLVM is too large to produce meaningful and conclusive comparisons. For example, the recursive Fibonacci program takes around 950 milliseconds to run on a single core using RE-LANG in our benchmarks, but the same program only takes about 100 milliseconds to run using LLVM-based Cilk implementations. (3) The runtime of existing frameworks, such as Cheetah for Cilk, is highly engineered.
and optimized. On the other hand, our runtime is yet to integrate a comparable degree of optimization. This circumstance would yield unfair comparisons.

To get a sense of how existing frameworks such as Cilk perform and scale, we refer to prior publications that focus on evaluating these algorithms (Bhattacharjee, Contreras & Martonosi 2011, Wheeler, Stark & Murphy 2012, Podobas, Brorrsson & Faxén 2015). The basic takeaway from past studies is that all existing frameworks struggle to reach ideal scaling even using cutoff schemes. This is due to synchronization and other runtime overhead.

With that being said, in the following chapter, we will establish a theoretical argument and prove the fact that RE-LANG will always outperform traditional work-stealing-based frameworks given sufficient parallelism. Compared to empirical comparisons, our theory compares both frameworks under clean formalism with minimal assumptions. This yields a clearer and more general picture of the runtime cost factors at play between traditional frameworks and RE-LANG than empirical studies.
CHAPTER 4: RUNTIME COST ANALYSIS

In the last chapter, we have provided concrete experimental benchmarks of how RE-LANG performs in the parallel setting—it enjoyed close to ideal scalability on all benchmarks. However, one may wonder how strong such a promise holds in general. In this chapter\textsuperscript{1}, we will give a formal analysis of our runtime scheduling policy. Our results will take into account all scheduling-related synchronization costs at runtime.

The most important contribution of this chapter is to provide a formalism that allows us to compare RE-LANG to traditional work-stealing. We will show that traditional work stealing algorithms must incur $S$ number of synchronization costs, where $S$ is the number of task spawns in the program. The implication is that when the input program is sufficiently parallel, such a cost is overwhelming and allows RE-LANG to outperform traditional work stealing.

4.1 Model and Concepts

This section introduces the concept of a continuation-spawn decomposition of a parallel program DAG. Using the continuation-spawn decomposition, we prove a theoretical upper bound on the speedup a parallel program will achieve in Theorem 4.1.3, which will be used in Section 4.4 to prove that a traditional work stealing incurs a higher synchronization cost than the global rebalancing policy.

We begin by modeling a single parallel program as a DAG of unit-sized tasks, which is a commonly-used model of parallelism (Blumofe & Leiserson 1994). For a given input program, there are several characteristics of the underlying DAG:

\textsuperscript{1}© 2023 IEEE. Reprinted, with permission, from T.Tao, “Synchronization Efficient Scheduling of Fine-grained Irregular Programs”, 2023 31st Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2023
• $T_1$, the total number of unit-sized tasks comprising the DAG. $T_1$ also represents the time required to execute the program on a single processor.

• $T_\infty$, the length of the longest path in the DAG, also known as the **critical path**.

• $P$, the total number of processors.

• $P = T_1 / T_\infty$, an upper bound on the **speedup** a program receives from running on any number of processors. Here, speedup refers to the ratio of a program’s running time on $P$ processors to its running time on a single processor.

Throughout this paper, we assume that the DAG is both parallel ($P > 1$) and non-empty ($T_\infty > 0$).

We will partition the DAG using a particular procedure called the spawn-continuation decomposition. The spawn-continuation decomposition consists of the critical path plus some disjoint sets of paths covering the remaining tasks. We refer to paths other than the critical path as continuations.

To construct the spawn-continuation decomposition of a DAG, we successively select the longest continuation consisting of uncovered tasks.

**Algorithm 4.1.1 (Spawn-continuation decomposition).**

1. We first select the critical path and mark all the tasks on this path as covered.

2. We select the longest path in the remaining uncovered DAG as the next continuation.

3. Mark all the tasks on this continuation as covered\(^2\).

4. Go to step 2 unless all tasks are covered.

The purpose of this decomposition algorithm is to create sets of continuations of non-increasing length. Let $L_k = T_\infty - k$. For all $k \in [1, T_\infty - 1]$, we define $c_k$ to be the number of continuations of length $L_k$ in the

\(^2\)In addition, for any spawn edge along this continuation, designate it as a non-spawn edge and change the original non-spawn sibling edge to a spawn edge. This does not change the semantics of the DAG.
Figure 4.1: Spawn-continuation decomposition of a simple fork-and-join DAG. In this figure, the path denoted by the number 0 is the critical path. The paths denoted by the number 1 and 2 are continuations.

spawn-continuation decomposition of the DAG. We can summarize the spawn-continuation decomposition of the DAG by providing $T_{\infty}$ and $C = \{c_1, c_2, \ldots, c_{T_{\infty} - 1}\}$ where $c_i \in C \geq 0$.

Based on the above definitions, we then prove a few lemmas, which leads to Theorem 4.1.3.

**Lemma 4.1.1 (The spawn lemma).**

$$\sum_k c_k = S$$

**Proof.** To see this, we will show that each thread/sub-DAG spawned by a spawn corresponds to a unique continuation produced by Algorithm 4.1.1.

In the first iteration of the algorithm, the critical path is chosen. There is no spawn in this scenario, and the critical path does not count as a continuation, so the conclusion holds. After the first step, what remains are exactly all the sub-DAGs spawned along the critical path.

In general, for iteration $i$ ($i \geq 2$) in the algorithm, it begins with $n_i$ independent sub-DAGs and selects $m_i \leq n_i$ longest paths from the sub-DAGs and produces $n_{i+1}$ independent sub-DAGs for iteration $i + 1$ such that

$$n_{i+1} = n_i - m_i + \sum_{k=1}^{m_i} T_i^k$$

where $T_i^k$ is the number of sub-DAGs spawned along the selected path in the sub-DAG $k = \{1, \ldots, m_i\}$ processed in iteration $i$. Note that $m_i > n_i$ is impossible because it would imply the algorithm selects more
than one path from some sub-DAG, where one is necessarily the parent of the others. However, these paths would not be of equal (longest) length.

The algorithm terminates therefore \( \lim_{i \to \infty} n_i = 0 \), or in other words,

\[
n_2 - m_2 + \sum_{k=1}^{m_2} T_2^k - m_3 + \sum_{k=1}^{m_3} T_3^k - \ldots = 0
\]  

(4.1)

Now observe that after the first iteration, \( n_2 \) spawns disappear from the DAG. After the 2nd iteration, \( \sum_{k=1}^{m_2} T_2^k \) spawns disappear, so on and so forth. Therefore, in total, we have

\[
n_2 + \sum_i \sum_{k=1}^{m_i} T_i^k = S
\]

(4.2)

Additionally, the number of total selected paths is exactly the total number of continuations equal to \( \sum_k c_k \). Therefore,

\[
\sum_i m_i = \sum_k c_k
\]

(4.3)

Applying equation (3) and (2) into (1) yields the conclusion that \( \sum_k c_k = S \). \( \square \)

**Corollary 4.1.1.1.**

\[
\bar{k} = \sum_k \frac{k c_k}{S}
\]

**Proof.** The average value of \( k \) (i.e., \( \bar{k} \)) is \( \sum k \cdot c_k / \sum c_k \). By the spawn lemma, \( \sum c_k = S \) and hence the conclusion. \( \square \)

We now decompose work \( T_1 \) in terms of the summation of continuation lengths.

**Lemma 4.1.2 (Decomposition of work).**

\[
T_1 = T_\infty + \sum_k c_k (T_\infty - k)
\]
Proof. This lemma says that by adding the total lengths of all the continuations, we obtain the total number of tasks. This should be clear because, by Algorithm 4.1.1, no two continuations overlap, and we cover all tasks in the DAG.

The above results lead to the following theorem.

**Theorem 4.1.3 (P for the DAG model).**

\[ P = Sz + 1 \]

where \( z = 1 - \frac{k}{T_{\infty}} \)

Proof. By the decomposition of work \( T_1 \),

\[ T_1 = T_{\infty} + \sum c_k (T_{\infty} - k) \]

Because \( \sum c_k = S \) by the spawn lemma,

\[ T_1 = T_{\infty} + T_{\infty}S - \sum k \cdot c_k \]

We also know that \( \sum k \cdot c_k = S\overline{k} \) by Corollary 4.1.1.1.

\[ T_1 = T_{\infty} + T_{\infty}S - S\overline{k} \]

By the DAG model, \( P = \frac{T_1}{T_{\infty}} \).

\[ P = 1 + S(1 - \frac{k}{T_{\infty}}) \]
Let \( z = 1 - \frac{k}{T_{\infty}} \), we conclude that

\[
\mathbb{P} = Sz + 1
\]

The significance of this theorem is that the maximum parallel speedup \( \mathbb{P} \) of any fork-and-join DAG is directly determined by the number of spawns in the DAG.

**Corollary 4.1.3.1 (The spawn utilization).**

\[
z = \frac{L}{T_{\infty}}
\]

where \( L \) is the average length of a continuation.

**Proof.** 

\[
z = 1 - \frac{k}{T_{\infty}} = \frac{T_{\infty} - k}{T_{\infty}} = \frac{L}{T_{\infty}}
\]

The value of \( z \) can be interpreted as the spawn utilization of the input program. This makes sense because assuming \( T_{\infty} \) is fixed, the longer the average continuation length \( L \) is in the program, the larger \( z \) becomes: longer continuations "utilize each spawn more."

### 4.2 Global Rebalancing Policy

We now introduce the global rebalancing policy—a formal name of the scheduling policy of the RELANG runtime. Unlike work stealing, this policy does not rely on concurrent data structures. Instead, load balancing is conducted globally for all processors in synchrony.

**Algorithm 4.2.1 (The global rebalancing policy).**

1. Scheduling is initiated if any processor runs out of tasks.
2. The scheduler invokes a global thread barrier, where all processors pause work immediately and convene. Since we assume unit task size, this is always achievable.
3. The scheduler distributes all tasks in memory equally across all $P$ processors such that any processor has at most one task more than any other, which is always achievable. After this step, denote the number of tasks in processor $i$ as $A_i$. If $\max(A_i) = 0$, the runtime terminates.

4. The scheduler commands each processor to work on at most $\max(A_i)$ tasks in a breadth-first manner. That is to say, processor $i$ will work on the exact $A_i$ tasks assigned to it in the previous step before working on any other task.

5. The scheduler invokes a second global thread barrier and waits for all processors to convene.

6. The global rebalancing finishes by exiting the global thread barrier and allowing each processor to return to independent work.

7. If the scheduling condition is satisfied again, go to (2).

### 4.2.1 Synchronization Cost

Let a *scheduling iteration* refer to a one-time execution of the scheduling steps from 1 to 7. We define synchronization cost as accumulating some specified synchronization cost in each step. Later, we justify this definition by proving that it is the only synchronization cost concerning a perfect schedule of $\mathcal{O}(T_1/P)$.

We will carefully account for all synchronization cost induced by the global rebalancing policy during scheduling iterations. We assume a tightly-coupled, fully memory-coherent machine with identical multi-processors that follows the total store ordering (TSO) consistency model. An example of this is Intel’s x86 architecture prevalent in personal and cloud computing environments today.

The *unit of synchronization cost*, which we standardize and refer to throughout the remainder of this chapter, is defined to be the cost of any CPU operation that generates coherence traffic (e.g. cacheline invalidation) or relies on hardware synchronization primitives (e.g. CAS, MFence, thread barrier). These
operations are considered expensive because they involve inter-processor communication and are limited by
the frequency and bandwidth of the interconnect or system bus. All other operations, mostly just local reads
or writes, are considered to be constant-time and cheap.

**Definition 4.2.1 (Synchronization cost).** The synchronization cost for each scheduling iteration is 
\[ g(P) = 2C_G + C_B + C_M + d(P), \]
where \( C_G \) is the cost of a global thread barrier operation, \( C_B \) is the cost of broadcasting
a one-bit message globally, \( C_M \) is the cost of a memory barrier operation, and \( d(P) \) accounts for all
operational cost.

In step 1, the processor that becomes empty the earliest broadcasts a one-bit message globally and
informs other processors to stop working and convene at the first barrier. All processors will receive the
message after \( C_B \) time steps and stop working immediately. This can be achieved by making each processor
perform a local, non-synchronizing read of a global flag that signifies whether some processor is empty after
each computation step. The cache-coherent protocol guarantees that when a processor is empty and writes to
this global flag, all other processors will receive this update after a small delay, which we account for as \( C_B \).
There is no synchronization cost to observe this flag because no coherence traffic is generated when every
processor is busy.

Step 2 corresponds to one global thread barrier operation. Let the synchronization cost of the barrier
operation be \( C_G \).

In step 3, denote the cost for each processor to find out where its partition is under equal partitioning as 
\( d(P) \). This is done by traversing through an array of size \( P \) in shared memory that contains the number of
current tasks in each processor’s task stack and computing the sum. With the sum, each processor knows
how many tasks it is assigned and from which processors and stack locations it should migrate the tasks.
This is possible based on the a priori agreement that the equal partitions are sorted in descending order in
size, and the \( i \)-th processor is assigned the \( i \)-th partition. For this step, we count a synchronization cost of
Figure 4.2: An example of global load rebalancing for 4 processors. Before load balancing, there are 13 tasks in total. The equal partitioning yields \( \{4, 3, 3, 3\} \), which is represented by different colors in the figure.

\( C_M \), corresponding to a memory fence at the beginning to ensure all processors agree with the same shared memory view.

We note that task migration can be done virtually without copying the tasks by assuming each processor has an \textit{ancillary} stack in addition to the \textit{main} stack. The main stack is the current evaluating stack with tasks in each processor, while the ancillary stack is empty. Virtual migration works in the following way: The processor reads the task through shared memory from the target processor’s main stack, executes the task, and writes the resulting tasks onto its own ancillary stack. At the end of the rebalancing iteration, the main and ancillary stack roles are flipped. This mechanism is illustrated in Fig. 4.2.

Step 4 consists of executing exactly all tasks in the current memory in a breadth-first evaluation order. This implies that although the executed tasks may spawn new tasks, the new tasks are set aside in the ancillary stack and not executed until after the scheduling iteration.

Since the tasks are equally distributed among all processors with a difference of at most one, letting all processors run \( \max(A_i) \) steps may introduce up to one task-sized idle time for some processors. However, the idle time is processed in parallel among all processors, so the total delay is 1. We count this delay in the \( d(P) \) term as the synchronization cost of step 4.

Step 5 corresponds to one global thread barrier operation, which we count as \( C_G \). We count all operational constants, including step 6 and 7, into the previous \( d(P) \) term.
4.2.2 Number of Scheduling Iterations

We now bound the maximum number of scheduling iterations, which happens to be the length of the critical path.

\textbf{Definition 4.2.2 (Span).} The span is the longest remaining path(s) in the DAG.

\textbf{Definition 4.2.3 (Active path).} A path in the DAG is active if the head of the path has zero in-degree.

Intuitively, a path with zero in-degree means its task dependency is resolved and ready to be executed, hence ”active”.

\textbf{Lemma 4.2.1.} Each scheduling iteration reduces the span of the DAG by exactly one.

\textit{Proof.} First of all, all the spans must be active. This is because the spans are the longest remaining paths in the DAG, and the longest paths in the DAG must have zero in-degree. Otherwise, following the predecessor of the path yields a longer path.

Now, per step 4 in Algorithm 4.2.1, the DAG is evaluated in a breadth-first manner, meaning all the active paths have their length reduced by exactly one. Because all the spans are active, the length of all the spans is reduced by exactly one. The conclusion follows. \hfill \Box

\textbf{Theorem 4.2.2 (Maximum scheduling iterations).} The number of scheduling iterations is at most $T_\infty$.

\textit{Proof.} The starting span of the DAG is the critical path of length $T_\infty$. According to Lemma 4.2.1, each scheduling iteration reduces the span length by exactly one. When the span is of length 0, the computation terminates. Therefore, the number of times the span can be reduced by exactly one is at most $T_\infty$. The conclusion follows. \hfill \Box
4.3 Analysis of Running Time

We now give the running time of any DAG with a total number of tasks \( T_1 \) on \( P \) processors using the global rebalancing policy. Our result takes into account all synchronization cost.

In our analysis, a schedule refers to a discrete time step at runtime with information on how tasks are assigned to all processors in the time step. Greedy schedules are time steps where if there are at least \( P \) total number of tasks, each processor works on at least one task, and if there are fewer than \( P \) tasks, all tasks are worked on. We refer to the previous as fully greedy schedules and the latter as partially greedy schedules. All other schedules are non-greedy.

Lemma 4.3.1 (Worst-case running time). Based on the global rebalancing policy, the running time \( T_P \) of any DAG using \( P \) processors is

\[
T_P \leq O\left(\frac{T_1}{P}\right) + g(P)T_\infty
\]

including all synchronization cost.

Proof. We prove this by showing that for all partially greedy or non-greedy schedules at runtime, the total time steps are counted towards the synchronization cost \( g(P) \) totaling up to \( g(P)T_\infty \). For the fully greedy schedules, the running time is clearly bounded by \( O(T_1/P) \).

Step 1 of the global rebalancing policy contains partially greedy schedules during the broadcasting delay \( C_B \) since at least one processor is empty but waits idly at the global thread barrier, not working on tasks available in the shared memory of other processors. However, this schedule takes up to \( C_B \) time steps and is subsumed into \( g(P) \) by Definition 4.2.1.

Step 2 is non-greedy, as all processors are busy entering the global thread barrier. Step 3 is also non-greedy since all processors are busy with MFence. In both cases, the delay is counted towards \( g(P) \) according to Definition 4.2.1.
In step 4, tasks are distributed across all processors equally with a difference of at most one. The equal partitioning cost is $d(P)$ and counted in $g(P)$ by Definition 4.2.1. During the first $\max(A_i) - 1$ time steps, a fully greedy schedule is computed per the equal partition policy. The final time step where some processors may become idle is partially greedy, but we count this in $g(P)$ per Definition 4.2.1.

Step 5 is a non-greedy schedule since all processors are busy entering the global thread barrier. The delay is counted in $g(P)$ by Definition 4.2.1.

All operational constants during the scheduling steps including step 6 and 7 are counted towards the $d(P)$ term in $g(P)$ by Definition 4.2.1.

All schedules computed outside the scheduling steps are, by definition, fully greedy since any out-of-work processor initiates a scheduling iteration.

Lemma 4.3.1 is significant in that it is possible to have a worst-case running time guarantee when scheduling dynamic irregular DAGs. To the best of our knowledge, no existing framework enjoys this property. Other state-of-the-art methods, such as (Acar, Charguéraud & Rainey 2013, Rito & Paulino 2022) are based on random processes and have no well-defined worst-case behavior.

It turns out that the $g(P)$ synchronization cost factor can be amortized for all properly parallel DAGs, where the number of parallelism grows to infinity as the program size grows to infinity.

Definition 4.3.1 (Properly parallel DAG). A DAG is properly parallel when

$$\lim_{n \to \infty} \mathbb{P}(n) = \infty$$

where $n$ is the input size of the program, and $\mathbb{P}(n)$ is the maximum amount of parallelism in the generated DAG.

Lemma 4.3.2. Throughout the execution of the DAG, the span is reduced exactly $T_\infty$ times.
Proof. At the start of the execution, the span is the critical path of length $T_\infty$. The computation terminates when the span length is 0. We need to show that there is no gap in the length reduction.

Suppose there is a gap: Assume at time step $t_1$, the span length is $T_\infty - i$, but at time step $t_1 + 1$, the span length becomes strictly smaller than $T_\infty - i - 1$.

The only way the span is no longer of length $T_\infty - i$ after $t_1$ is that the last path of this length is executed at $t_1$. However, at $t_1 + 1$, the length of this path must be $T_\infty - i - 1$. This is because it is impossible to make more than one unit task progress on a path within one time step, a contradiction. \hfill \Box

**Theorem 4.3.3 (Amortized running time).** Based on the global rebalancing policy, the amortized running time $T_P$ of any properly parallel DAG using $P$ processors is

$$T_P = \mathcal{O}\left(\frac{T_1}{P} + T_\infty\right)$$

including all synchronization cost.

Proof. We prove this by amortizing the $g(P)$ factor, a linear function with regard to $P$, against longer sequential time steps at runtime.

Denote the number of scheduling iterations to be $C$, where $C \leq T_\infty$ per Theorem 4.2.2. Additionally, according to Lemma 4.2.1, each scheduling iteration reduces the span length by exactly one. Denote the average number of tasks executed per processor (in step 4 of Algorithm 4.2.1) per scheduling iteration as $\bar{B}'$. Therefore, the total number of tasks executed during scheduling iterations is $\bar{B}'PC$.

However, according to Lemma 4.3.2 the span length is reduced exactly $T_\infty$ times when the computation terminates. This implies that the number of times this length is reduced outside scheduling iterations is $T_\infty - C$. Denote the average number of tasks executed per processor per reduction on the span outside scheduling iterations to be $\bar{B}$. Therefore, the total number of tasks executed outside scheduling iterations is $\bar{B}P(T_\infty - C)$.
This yields us

\[ \bar{B}P(T_\infty - C) + \bar{B}'PC = T_1 \]

Now, the \( g(P) \) cost induced by the \( C \) number of scheduling iterations is amortized when

\[ \bar{B}(T_\infty - C) + \bar{B}'C \geq PC \]

That is to say, it is amortized when the number of all time steps for executing the input program itself is greater than the synchronization cost. The number of all time steps is represented on the left-hand side as the sum of the number of time steps for executing the program outside scheduling iterations \( \bar{B}(T_\infty - C) \) and the number of time steps executing the program inside scheduling iterations \( \bar{B}'C \). To solve this condition, we start from the previous fact that

\[ \bar{B}P(T_\infty - C) + \bar{B}'PC = T_1 \]

Dividing both sides by \( P \) yields

\[ \bar{B}(T_\infty - C) + \bar{B}'C = \frac{T_1}{P} \]

Hence, the amortization condition \( \bar{B}(T_\infty - C) + \bar{B}'C \geq PC \) is satisfied when

\[ \frac{T_1}{P} \geq PC \]

Dividing both sides by \( T_\infty \) plus the fact that \( \bar{P} = \frac{T_1}{T_\infty} \) yields
\[ \frac{P}{P} \geq Pc, \text{ where } c = \frac{C}{T_\infty} \text{ is the synchronization frequency.} \]

\[ P \geq P^2 c \]

Since \( c \leq 1 \), this condition is satisfied when

\[ P \geq P^2 \]

Because \( P \) is fixed (per \( T_P \)), and \( P \) grows to infinity as the input size grows to infinity (per properly parallel DAG), and our condition for amortized running time bound is satisfied asymptotically. The conclusion follows.

Throughout the proof, we assumed that the synchronization frequency \( c > 0 \) and \( 1 - c > 0 \). In the case \( c = 0 \), there is no synchronization, and the conclusion holds trivially. In the case \( c = 1 \) (i.e., \( C = T_\infty \)), all the tasks are executed during scheduling iterations. Each iteration, on average, executes \( T_i / T_\infty = P \) tasks, taking up to \( [P/P] \) time steps. When \( P \) goes to infinity, the same conclusion follows.

Finally, in the proof, we assumed that all scheduling iterations partition tasks perfectly equally, leading to average \( \bar{B}' \) tasks per iteration per processor. In reality, there may be a one-task difference among processors. We now show that the latter does not affect the conclusion. In unequal partitioning, the only change to the above proof is to substitute \( T_i \) with \( T'_i \), where \( T'_i \geq T_i - (P - 1)T_\infty \). This is because, under unequal partitioning, we ignore up to \( (P - 1)T_\infty \) tasks given that \( \bar{B}' \) is the average size of the equal partitions per iteration per processor. Consequently, we substitute \( P \) with \( P' = T'_i / T_\infty \geq P - (P - 1) \). Therefore, the final condition becomes \( P' \geq P^2 \) if \( P - P + 1 \geq P^2 \) and if \( P \geq P^2 + P - 1 \). The conclusion follows the same logic that \( P \) goes to infinity, given fixed \( P \) and properly parallel DAGs.
Corollary 4.3.3.1 (Optimal processor number). The optimal number of processors \( P \) to use in the global rebalancing policy is \( \sqrt{P/c} \), where \( c = C/T_\infty \) is the synchronization frequency, and \( C \) is the number of synchronization iterations.

Proof. The optimal asymptotic running time is achieved when the cost of synchronization iterations is amortized. According to the previous theorem, this is satisfied when \( P \geq P^2c \), where \( c \leq 1 \) is the synchronization frequency. This implies that by using no more than \( \sqrt{P/c} \) number of processors, we achieve amortization. However, we want to use as many processors as possible to achieve parallel speedup, therefore the optimal number of processors to use is exactly \( \sqrt{P/c} \).

The significance of this corollary is that by measuring or predicting the amount of parallelism \( P \) and synchronization frequency \( c \) at runtime, we can infer the optimal number of processors in terms of the lowest synchronization overhead by the global rebalancing policy. This serves as a starting point for future research in adaptive runtime control and management.

4.4 Comparison with Traditional Work Stealing

We now compare the total running time synchronization cost required by the global rebalancing policy with the synchronization cost required by traditional work stealing on modern processor architectures.

This result relies critically on Attiya et al.’s seminal work in concurrency theory (Attiya et al. 2011). Their paper tackles the challenge of designing correct and efficient concurrent algorithms, with an emphasis on minimizing the use of expensive synchronization patterns. Attiya et al. identified two prevalent synchronization patterns: read-after-write (RAW) and atomic write-after-read (AWAR). These patterns are known to be costly on current mainstream processors, making their elimination or reduction highly desirable.

The most prevalent architecture today, such as Intel’s x86, employs the total store ordering (TSO) consistency model. In TSO, memory operations are divided into two categories: load (read) and store (write). The architecture guarantees that all processors observe a total order of store operations, but load operations
can be reordered with respect to store operations. This side effect is caused by the existence of store buffers built in each processor core, which accelerates the CPU’s processing speed by buffering all memory write instructions and allowing them to execute asynchronously. This behavior is transparent to the programmer in a single-core setting since TSO architectures often also implement store-buffer forwarding, which ensures the program order up to each core. However, the implication of this design is that multiple cores can disagree with the global memory state, leading to incorrect concurrent algorithm implementations if proper synchronization operations are not employed.

To ensure the RAW pattern in a TSO architecture, memory fence (MFence) synchronization primitives, also known as memory barriers, are required. Memory fence operations are designed to enforce ordering constraints on memory operations, preventing them from being reordered across the fence. By using these operations, developers can explicitly enforce the order in which memory operations are executed, thus preserving the read-after-write pattern and ensuring data consistency.

On the other hand, the atomic write-after-read (AWAR) pattern corresponds to hardware-level instructions such as compare-and-swap (CAS) that are ubiquitously used in the design of concurrent data structures. Almost all modern architectures include some version of the AWAR semantics such as CAS or load-link/store-conditional (LL/SC).

Attiya et al. address a key question that arises during the design process: is it inherently impossible to eliminate certain synchronizations or should designers keep attempting to remove them? They provide a formal answer to this question by proving that it is impossible to completely eliminate RAW and AWAR in the implementation of classic specifications, such as sets, queues, stacks, mutual exclusion, and read-modify-write operations. Formally, Attiya et al. proved that it is impossible to build linearizable implementations of strongly non-commutative methods without RAW and AWAR, while satisfying deterministic sequential specifications.
We now clarify some of the terminologies involved. The basic premise of Attiya et al.’s analysis relies on tracing the execution, defined as the sequence of atomic transitions of the global memory state, of any parallel program and recording the sequence of method calls, which are potentially interleaved in a multi-core setting. This is referred to as the history of the program’s execution. Deterministic sequential specification refers to all possible histories executing the parallel program non-interleaved. The specification being deterministic solidifies the fact that each method is a proper algorithm: The same set of input parameters determines a unique output. A parallel execution of the program is considered correct if its history is linearizable with respect to some sequential specification of the program, meaning the result of parallel, interleaved execution of some methods is the same as some execution of these methods one after another with the same input arguments. A method is said to be strongly non-commutative if there exists another method, possibly itself, such that when they operate on the same global state in parallel, it is possible to tell which linearized history the execution produces based on the results of all the methods’ invocations.

In our context, we are interested in applying Attiya et al.’s theoretical results to the context of traditional work-stealing data structures, which we define as the following.

**Definition 4.4.1 (Traditional work-stealing schedulers).** We define traditional work-stealing schedulers to be the ones that have the following characteristics:

1. Each processor always makes all its tasks available to be stolen in its stack except for the task it is currently working on.

2. If a task is in the stack, there are only two ways to extract it: pop or steal, where the stack owner pops, and the thieves steal. Additionally, each task is extracted exactly once deterministically.

3. All pops and steals fetch a single task per invocation. Further, assume that all such invocations are linearizable with respect to the sequential specification.
This definition is representative of most work-stealing schedulers today, including the Chase-Lev concurrent deque (i.e. doubly-linked queue) implementation (Chase & Lev 2005). Here, of particular relevance to traditional work stealing from Attiya et al.’s result is the fact that both pop and steal are strongly non-commutative methods. Therefore, for either implementation of these two methods, the runtime must ensure that any invocation of steal is atomic in relation to any other steal and that any invocation of pop is atomic in relation to any other number of steals that could deplete the queue. Otherwise, a RAW or AWAR pattern must exist in the execution, which realistically implies the use of at least one expensive MFence or CAS operation. In both cases, we count one unit of synchronization cost.

Lemma 4.4.1. Under the TSO model, the total synchronization cost by all traditional work-stealing schedulers is lower bounded by \( C_U S \), where \( S \) is the total number of spawns in the DAG, and \( C_U \) is the average synchronization cost.

Proof. According to (1) in the definition of traditional work stealing, all tasks in a concurrent deque are available to be stolen besides the task being processed. Without loss of generality, assume that the processor puts all spawned tasks in its deque and then chooses the continuation task as its next task. The total number of tasks pushed into the deque throughout the execution is therefore \( S \).

According to (2) in the definition of traditional work stealing, any task put into the deque can only be extracted in two ways: it is either popped or stolen exactly once. Under this condition, since one unit of synchronization is required under TSO for both pop and steal operations, the amount of synchronization cost under traditional work stealing is lower-bounded by \( C_U S \), as desired. \( \Box \)

3The nuance here is that Attiya et al.’s result does not immediately preclude a low-cost/constant-time algorithm or hardware design that separates safe strongly non-commutative executions from unsafe ones, which in theory remains an open problem. With that being said, we are not aware of any implementation of such a low-cost implementation without breaking the definition of traditional work stealing, hence counting one unit of synchronization cost for both pop and steal is a reasonable assumption. In fact, Attiya et al. verified that in all popular implementations of traditional work-stealing schedulers such as the Chase-Lev and Arora-Blumofe-Plaxton (ABP) deques, this assumption holds.
Based on the previous conclusions, we can provide a sufficient condition for an input program to run asymptotically faster under global rebalancing than traditional work stealing.

**Theorem 4.4.2.** Consider a parallel program that takes an input of size \( n \). Let \( T_1(n) \) and \( T_\infty(n) \) be the total number of tasks and critical path length in the DAG generated. If \( T_1(n) = \omega(T_\infty^2(n)) \), global rebalancing achieves asymptotically better running time than all traditional work-stealing schedulers.

**Proof.** By Lemma 4.4.1, the total synchronization cost by traditional work stealing is \( C_US \). In the best case, this cost can be amortized across all processors, so the total cost for traditional work stealing is at least \( C_US/P \). By Theorem 4.2.2 and 4.3.3, the total cost of the global rebalancing scheduling policy is amortized \( gT_\infty(n) \), where \( g \) is some constant.

Now, according to Theorem 4.1.3 and Corollary 4.1.3.1, \( P = Sz + 1 = SL/T_\infty(n) + 1 \). Therefore, \( S = (P - 1)T_\infty(n)/L \). It follows that

\[
\frac{C_US}{P} = \frac{C_U(P - 1)T_\infty(n)}{PL}
\]

We want to know when is \( C_US/P > gT_\infty(n) \). This holds if

\[
\frac{C_U(P - 1)T_\infty(n)}{PL} > gT_\infty(n).
\]

Rearranging, this condition gives

\[
P > \frac{gP}{C_U}L + 1.
\]

Note that \( L < T_\infty(n) \) because the average continuation length is strictly smaller than the length of the critical path. So the above holds if

\[
P > \frac{gP}{C_U}T_\infty(n) + 1.
\]
From the definition of $\mathbb{P}$, we get that
\[
\frac{T_1(n)}{T_2^\infty(n)} > \frac{gP}{C_U} + \frac{1}{T_\infty}
\]

When $T_1(n) = \omega(T_2^\infty(n))$, $\lim_{n \to \infty} T_1(n)/T_2^\infty(n) = \infty$, the above inequality holds as $P$ is fixed. The conclusion follows.

4.5 Non-traditional Work-stealing Schedulers

The synchronization cost limitation posed by traditional work-stealing schedulers has incentivized creative modifications to traditional work-stealing algorithms. This is done by breaking the usual work-stealing semantics such as stealing multiple tasks at once as opposed to just stealing one, allowing some tasks to be repeatedly extracted, always keeping some tasks private to the owner processor, etc. However, all non-traditional work-stealing schedulers suffer from major limitations due to these non-conventional modifications.

Hendler et al. (Hendler & Shavit 2002) proposed a work-stealing approach in which a thief processor steals half of the tasks from the victim processor, instead of just one. This strategy aims to reduce synchronization cost to a logarithmic function of the owner’s execution path length and the number of steals. However, the exact amount of synchronization cost incurred remains unclear, as the authors did not provide a bound on the number of steals. The main drawback of their method is that the steal-half strategy is limited to fixed-size deques, which are susceptible to overflow problems when the system overflows with a lot of fine-grained tasks.

Michael et al. (Michael, Vechev & Saraswat 2009b) introduce a technique called idempotent work stealing. The conventional semantics of work stealing guarantee that each inserted task is extracted exactly once, but many applications can tolerate repeated work or already ensure that no work is repeated. They exploit this invariant relaxation and introduce idempotent work stealing, which guarantees that each inserted task is extracted at least once. They also present several new algorithms that exploit the relaxed semantics to deliver
better performance by avoiding the use of expensive synchronization instructions in the critical path of the owner’s operations. However, Michael et al.’s approach requires the input program to be idempotent, making their solution unsuitable for many general algorithms.

Tzannes (Tzannes 2012) proposes a solution called lazy scheduling, a scheduling algorithm that adapts the granularity of parallelism to run-time load conditions. The author also provides compiler optimizations and a parametric model for the costs of the runtime to further improve the efficiency of declarative task-parallel codes. Tzannes argues that lazy scheduling, combined with these optimizations and the parametric model, can achieve very high efficiencies and improve both ease of programming and performance portability. Nevertheless, lazy scheduling, relying on control of granularity, is essentially a cutoff heuristic, where the authors provided no generalizability analysis.

Endo et al. (Endo, Taura & Yonezawa 1997) describe the implementation of a parallel mark-sweep garbage collector for shared-memory machines and report its performance. The collector stops the application program during a collection and all the processors cooperatively traverse objects in the global shared heap. The algorithm performs dynamic load balancing by exchanging objects in mark stacks. They found several factors that limit scalability and presented solutions to improve performance, such as splitting large objects into small parts, skipping queues already locked by another processor, using a non-serializing method for termination detection, and using a test-compare-and-swap sequence instead of a lock-and-test sequence. Endo et al.’s effort represents the first to popularize the use of *split deques* to improve performance: By distinguishing public and private portions of the shared deque, the scheduler can employ tricks to avoid some amount of synchronization. Dinan et al. (Dinan, Larkins, Sadayappan, Krishnamoorthy & Nieplocha 2009) further investigate split deques on large-scale distributed memory systems. They also proposed various techniques to reduce overhead and increase efficiency, such as reducing locking on the critical path, reducing task creation overhead, maximizing the availability of work, and implementing a customized locking scheme. Similar efforts were pursued by Lifflander et al. (Lifflander, Krishnamoorthy & Kale 2012), who showed
that retentive stealing improves execution efficiency by incrementally improving the load balance. The most advanced software-based split deque implementation is by Dijk et al. (Dijk & Pol 2014). The deque uses a dynamic split point between the shared and private portions. Only memory fences are required when shrinking the shared portion. The proposed design is implemented in a C library called Lace, which has a similar interface to the work-stealing library Wool (Faxen 2010b). The performance of Lace is evaluated using common benchmarks, and it is shown to be competitive with Wool and a recent approach using private deques. The work-stealing algorithm in Lace eliminates the limitations of previous approaches and allows for efficient parallel execution of small tasks. Unfortunately, the split deque design is a complex heuristic and incurs synchronization cost when adjusting the split point between public and private regions. While split deques can drastically improve scalability for certain program benchmarks, it suffers from a similar generalizability issue as cutoff schemes since no such analysis was provided by these authors.

As an alternative to split deques, Hiraishi et al. (Hiraishi et al. 2009) propose that to eliminate the cost of creating and managing logical threads, processors can perform backtracking and temporary restoration of task-spawnable states to spawn real tasks only when requested by another idle worker. The paper proposes that this approach eliminates the cost of spawning/managing logical threads and improves the reuse of workspaces and the locality of reference. The framework enables efficient backtrack search algorithms with delayed workspace copying. However, backtracking, while effective, introduces a cost in itself, potentially causing an unacceptable level of overhead for certain classes of parallel programs.

Morrison et al. (Morrison & Afek 2014) challenge the traditional belief that memory fences are necessary for the correctness of work-stealing algorithms in task parallel programming. They introduce the concept of a bounded total store ordering (TSO[S]) memory model, which is a restriction of TSO that limits the number of stores a load can be reordered with. By exploiting this reordering bound, the authors propose three fence-free work-stealing algorithms (FF-THE, FF-CL, and THEP) that eliminate the overhead of memory fences. However, Morrison’s solution requires detailed reasoning concerning architecture topology.
and compiler implementation, making their solution highly non-portable. Additionally, since their THEP algorithm blocks thieves when stealing is potentially unsafe, it necessitates a loss of parallelism proportional to the size of store buffers. As the number of processors increases, the loss of parallelism also increases because fewer tasks now reside in each processor, leading to more blocking of thieves.

The most well-developed, generic task-parallel scheduling algorithm is by Acar et al. (Acar, Charguéraud & Rainey 2013a), who presented a negotiation-based scheduling algorithm, drawing inspiration from distributed computing. In their approach, processors proactively share tasks rather than relying on stealing. As a result, all synchronization operations on work-stealing deques become unnecessary. However, the algorithm introduces additional overhead caused by the negotiation protocol. Acar et al.’s experimental results did not find a significant improvement in their method over traditional work-stealing implementations.

4.6 Summary

In this chapter, we presented a formalism that allows us to compare the running time of RE-LANG to all traditional work-stealing schedulers. This is done by first establishing a relationship between the number of spawns and the maximum amount of parallelism of a parallel program’s DAG. The relationship between the two is intuitive, showing that the more spawns there are in a DAG, the more parallelism the program has. We then refer to Attiya et al.’s impossibility results and show that all traditional work-stealing schedulers must incur running time synchronization cost proportional to the number of spawns. On the other hand, the synchronization cost of RE-LANG is a linear function with respect to the length of the critical path. Given that the DAG is sufficiently parallel, the number of spawns overwhelms the critical path-bound synchronization cost, yielding the fact that RE-LANG will outperform all traditional work-stealing schedulers.

The definition of sufficiently parallel (i.e. $T_1 = \omega(T_2^2)$) has a nice geometric meaning: A sufficiently parallel DAG is ”fat”—the $T_2^2$ can be interpreted as ”the area” onto which parallelism can be scheduled.
a DAG lacking parallelism, usually $T_1 << T_2^2$. For a DAG with sufficient parallelism, $T_1$ often grows much faster than $T_2^2$. For example, the recursive Fibonacci program has $T_\infty = \Theta(n)$, while $T_1$ is exponential.
CHAPTER 5: LIMITATIONS, FUTURE WORKS, AND CONCLUSION

In this chapter, we will conclude our work by discussing all the current limitations of RE-LANG. We will explain why a substantial amount of work of different natures is required to tackle them and point out a few starting points based on the research we have conducted in the scope of this thesis.

5.1 The Cost of Global Synchronization Barriers

First of all, in all currently existing architectures, the operational constant induced by the global thread barriers is relatively expensive\(^1\). While in the last chapter, we have shown that ample slackness can amortize this cost, it is practically still an issue when the program has insufficient parallelism: At least one processor is constantly out-of-work and triggers global rebalancing non-stop. To avoid this in practice, we can progressively add the number of processors, starting from a single core until no speedup is observed.

Future works remain to invent clever heuristics to avoid unnecessary synchronization iterations. Currently, we implemented simple heuristics in our current system that prevent rebalancing if no processor has at least two tasks. This works well for programs with sufficient parallelism but sometimes may still cause the overhead of barrier synchronization to overwhelm the parallelization benefit. Programs vulnerable to such pitfalls are those that spawn only a few short-lived tasks but do so on a frequent basis. Clearly, such programs do not possess sufficient parallelism up to our definition in the previous chapter, and we argue that they are not the most common programs in the real world: Recall that when parallel programs are written declaratively, the number of spawns usually grows very quickly.

\(^1\)On Intel(R) Xeon(R) CPU E5-2699A v4 2.40GHz, it requires approximately 2.7 microseconds to synchronize 16 processor cores using the OpenMP barrier directive.
With that being said, we acknowledge that such cases can sometimes appear at runtime, and we have the intuition that a better approach exists than simply setting a threshold for multithreading. One approach we attempted was to implement a hybrid scheme that combines work stealing with RE-LANG’s global rebalancing policy: When there is insufficient parallelism (e.g., each processor has fewer than \( P \) tasks), we exclusively use work stealing to balance workload; otherwise, the runtime switches to global rebalancing. Preliminary experimental results show that such a scheme eliminated the barrier synchronization overhead in the case of insufficient parallelism. However, it also caused a drastic slowdown in some of our benchmarks. Future research direction is to devise provably effective protocols that allow the runtime to switch between work stealing and global rebalancing with minimal cost. Recent engineering improvements in work stealing must also be fully incorporated to achieve the best possible performance. We consider this work to be out of the scope of this thesis as the primary goal of the thesis is the programming language design of RE-LANG, instead of parallel theory.

An alternative solution is to advocate the incorporation of faster implementations of global synchronization barriers. The idea is that if barrier synchronization is cheap enough, we could simply afford to not care about the overhead it induces. Perhaps to many’s surprises, past research in hardware design suggests optimism in this regard (Abellán, Fernández & Acacio 2011, Sampson, Gonzalez, Collard, Jouppi, Schlansker & Calder 2006, Sartori & Kumar 2010).

An example of such advancement is GBarrier (Abellán, Fernández & Acacio 2011). GBarrier is a novel hardware-based barrier mechanism designed specifically for efficient barrier synchronizations in future many-core chip-multiprocessors (CMPS). Traditional software-based barriers, which rely on busy waiting on shared variables that are atomically updated, can lead to memory and network contention as the number of processors increases, thereby limiting their applicability in many-core CMPS. The GBarrier mechanism consists of two main components. The first is a dedicated G-line-based network that uses the S-CSMA technique, enabling fast and efficient signaling of barrier arrival and departure. The second component is
a simple synchronization protocol that coordinates the actions of the controllers attached to the G-lines while performing barrier synchronizations. Unlike traditional approaches based on atomic read-modify-write instructions operating on shared memory positions, GBarrier does not impact the memory system. This avoids all coherence activity and barrier-related network traffic, improving scalability. GBarrier can be easily adapted to various scenarios and system configurations. The architecture can be extended to support multiple GBarriers, groups of cores, larger many-core CMPs, and simultaneous multithreading (SMT) processor cores.

In a detailed execution-driven simulation of a 32-core CMP running a set of benchmarks, GBarrier was compared against one of the most efficient software-based barrier implementations—a tree barrier—in terms of execution time, network traffic level, and energy efficiency. The results showed that GBarrier outperformed the software-based barrier implementation for both kernels and scientific applications.

Most noteworthy, Abellán et al. have shown that the latency of global thread barrier operations can be reduced to a mere dozen CPU cycles by implementing the operation at the hardware level, as opposed to thousands of cycles as the cost of software implementation of thread barriers on currently existing commercial hardware. For this reason, we argue that our global rebalancing policy has no fundamental disadvantage compared to CAS/MFence-based solutions. One may even argue that barrier-based policies are superior because CAS/MFence operations must interact with the shared-memory regime, which has its own scalability issues. On the other hand, a global barrier is a generic, fast operation that is memory-independent. Unfortunately, as of 2023, we are unaware of commercially available general-purpose CPUs that take advantage of hardware synchronization barriers. Hence, to practically and realistically demonstrate RE-LANG’s behavior on such systems lies in the scope of future works. It is our hope that the excellent parallelization capability of RE-LANG can serve as a motivation for hardware manufacturers to advance designs that incorporate hardware synchronization barriers.
5.2 Runtime Characteristics Compared to The State-of-the-art

Significant advancement on the front of work-stealing-inspired scheduler was proposed very recently. Rito et al. (Rito & Paulino 2022) presents a variant of the algorithm by Acar et al. (Acar, Charguéraud & Rainey 2013) that significantly reduces synchronization overheads in parallel computations by using a lazy approach to exposing work. The new algorithm, Low Cost Work Stealing, utilizes a data structure called SpDeque, which is split into private and public parts. The Low Cost Work Stealing algorithm improves upon traditional Work Stealing algorithms by minimizing synchronization overheads and maintaining high performance. It achieves this by introducing a new data structure called SpDeque, which consists of a private part for local operations and a public part for sharing work. Processors operate on the private part of their SpDeque by default, pushing and popping ready nodes as necessary. This allows them to avoid synchronization for local operations. They only attempt to fetch work from the public part if the private part is empty. If both the private and public parts of a processor’s SpDeque are empty, the processor becomes a thief and starts a stealing phase. During the stealing phases, thieves target victims uniformly at random and attempt to steal work from the top of their SpDeques. Work is exposed for load balancing only when a thief directly requests it, and only a single unit of work is exposed at a time. This lazy approach to synchronization is key to the improved performance of the Low Cost Work Stealing algorithm. Rito et al.’s algorithm ensures an asymptotically optimal expected runtime by provably reducing synchronization overheads. The algorithm’s expected total synchronization is $O(PT_{\infty}(CAS + MFence))$.

In comparison, our system runs just as optimally efficiently as theirs by having an amortized total running time of $O(T_1/P + T_{\infty})$. In addition, our system has a unique advantage: Rito et al.’s algorithm is based on random processes and does not have a meaningful worst-case behavior. On the other hand, RE-LANG’s runtime performance has a deterministic worst-case guarantee due to the fact that the global rebalancing policy
does not rely on random processes. Rito et al.’s work is purely theoretical and provided no experimental evaluation of their schedulers. The practical performance of their design remains to be seen.

5.3 User Study on Programmability

In this thesis, we relied on the argument that RE-LANG’s parallel-by-default mechanism allows for easier parallel programming because purely computational programs (e.g., the recursive Fibonacci function) can be parallelized with zero effort—hence it is better than annotation-based frameworks such as Cilk.

We did not formally conduct a social experiment to verify this claim in our thesis, mainly for two reasons: (1) The time and budget of this thesis project did not allow for such experiments at a meaningful scale. (2) We observe and acknowledge that how ”easy” it is to program in a certain framework or language is ultimately a subjective experience closely related to the nature of the task the programmer is working on and the flavor of judgment influenced by the programmer’s training and educational background. This observation is evident in the so-called ”language wars” in online programmer communities, where futile arguments are fervently advertised to boost or disrespect certain programming languages (e.g. C++ versus Java) solely based on a subjective ”sense of taste” and personal professional experiences according to each online commenter. However, if time and budget allow, we are interested in conducting a social experiment to confirm the effectiveness of the parallel-by-default design in the following way.

The experiment will follow an overall structure of a typical social scientific experiment design. First of all, we will sample a pool of participants from the university’s computer science department consisting of approximately 100 individuals. The idea is to select only the participants who presumably possess at least moderate exposure to computer and programming technology so that our experimental results are not affected by extra factors not related to the essential operations in the parallel programming language (e.g. struggling to understand how to program an algorithm in the first place). The participants will be split in half
into experimental and control groups. The experimental group will use RE-LANG, and the control group will use traditional annotation-based frameworks (e.g., Cilk).

Before the actual experiment, we will establish a training session to allow the participant to become familiar with the programming language and equipment used in the experiment. This can be conducted in the form of video instructions or handbooks. The session is crucial in eliminating any factor caused by "unfamiliarity" with the setup and the how-to(s) in the generic sense. Since RE-LANG is a new programming language and has a potential learning curve, we allow the participants to take the training session at their own pace for approximately one week. The same treatment will also be granted to Cilk. To ensure fairness, we will make it clear that only a subset of the Cilk/C programming language is allowed since the out-of-box C programming language that Cilk is based on is already much more powerful than RE-LANG.

On the day of the experiment, all participants will first complete a pre-experiment questionnaire, which quickly asks the participant to self-assess his/her training preparation level. Most importantly, the questionnaire will ask the programmer "Which programming language/framework do you find to be more appealing to you so far?" The purpose of this question is to compare the response to it with a corresponding question post-experiment, which helps us gauge the extent to which RE-LANG appeals to an average programmer versus its actual prefer-ability after real programming tasks are conducted.

The actual experiment will consist of approximately five to ten parallel programming tasks of gradually increasing difficulty (e.g., from Fibonacci recursive function to complex graphs and dynamic programming algorithms). The participant must complete all the tasks in a single sitting under the time pressure of approximately twenty minutes per programming task in a controlled environment without any additional help or collaboration. Since the topic of interest is parallelization, each programming task will ask the programmer to parallelize an existing algorithm, which is provided on the scene in the form of pseudocode as opposed to asking the participant to derive the algorithm from scratch. The participants will be encouraged to "think
“aloud” similar to a technical interview process. In the case of frustration, the participant has the freedom to quit the experiment at any time. The entire process will be audio and video recorded for final analysis.

Each experiment will produce two main artifacts: (1) The RE-LANG or Cilk code written by the participant for each parallel programming task. (2) The video and audio tape of each participant’s entire performance. Many dimensions of data will then be extracted from the artifact. For example, the code produced by the participant will be analyzed for algorithmic correctness, executed to assess its scalability, and compared to a standard implementation for readability. Additionally, we will review the videotape for quantitative cues that hint at the participant’s cognitive and emotional response to the task of programming in either RE-LANG or Cilk. For example, we will count the number of times the participant curses or makes annoyed sounds. We will pay special attention to any indication of frustration and difficulty, such as the number of tasks after which the participant gives up the experiment.

Finally, we will analyze the data using appropriate statistical tools such as ANOVA to determine the relationship between each parallel programming language/framework and the easiness provided by each to the parallel problem-solving process in terms of the number of setbacks and frustration induced to an average programmer confronted with increasingly challenging parallelization tasks.

5.4 Conclusion

To conclude, in this thesis, we presented a new programming language design—RE-LANG—that has a novel parallel-by-default semantics: All function arguments are considered as data dependencies independent with regard to each other and are scheduled as parallel tasks automatically. The key motivation behind this design is to enable declarative parallel programming, achieved in two aspects compared to all existing frameworks. First, the unique parallel-by-default semantics frees the programmer from the burdening, error-prone process of explicitly annotating parallel notations, such as those in the popular Cilk framework. Second, the parallel-by-default semantics provides a deterministic layer of abstraction in that it guarantees to
scale sufficiently parallel programs across tightly coupled multi-processors in modern architectures. This is a challenge since traditional schedulers, such as work stealing, struggle to cope with synchronization cost given a large number of fine-grained tasks. We successfully tackled this challenge by utilizing a global rebalancing policy that amortizes all scheduling costs. We formally showed that such a policy will always induce smaller overhead than all existing traditional work-stealing schedulers given sufficient parallelism.

In all, RE-LANG has advantages in the simplicity of use and semantics in an imperative framework and offers an intuitive and simple formalism for parallel programming, which has been implemented with good empirical results and theoretically analyzed to show its advantages. It is our hope that the design of RE-LANG inspires future system research and development: By taking advantage of fast hardware barrier synchronizations and memory accelerators, RE-LANG would be able to achieve even superior performance characteristics than our current portable implementation that targets commercially available CPUs.
1. *fib*

main() = fib(33)

fib(n) = if (n < 2) then n else (fib(n - 1) + fib(n - 2))

2. *ack*

main() = ack(3, 10)

ack(m, n) = if (m = 0) then (n + 1) else
   (if (n = 0) then ack(m - 1, 1) else
      ack(m - 1, ack(m, n - 1)))

3. *knapsack*

main() = {
    val ← (13,17,19,23,24,29,43,44,45,48,49,55,57,57,58,62,62,63,
    68,70,79,87,93,96,97,98,99,100);
    wt ← (3,6,13,19,20,28,29,37,41,43,48,49,54,63,63,68,68,70,73,77,
    79,88,88,91,94,94,98,99,99,100);
    w ← 500;
    n ← 30;
    knapsack(w, wt, val, n)
}

knapsack(w, wt, val, n) =
   if ((n = 0)  (w = 0)) then 0 else ( 
      if (wt[n - 1] > w) then knapsack(w, wt, val, n - 1) else
         max(val[n - 1] + knapsack(w - wt[n - 1], wt, val, n - 1),
            knapsack(w, wt, val, n - 1)))
main() = {
    rt ← randomtree(0);
    print(count(rt));

    t1 ← time();
    uts(rt);
    t2 ← time();
    print(t2 - t1);
}

uts(tree) = if (tree = Nil) then Return else {
    left ← tree[0];
    right ← tree[1];
    (uts(left), uts(right));
}

count(tree) = if (tree = Nil) then 0 else {
    left ← tree[0];
    right ← tree[1];
    (count(left) + count(right)) + 1
}

randomtree(d) = if (d = 24) then Nil else {
    r1 ← rand();
    r2 ← rand();
    hasleft ← r1 <= 90;
    hasright ← r2 <= 90;
    node ← (...2);
    (node[0], node[1]) ← (Nil, Nil);

    if hasleft then {
        node[0] ← randomtree(d + 1);
    }
}
if hasright then {
    node[1] ← randomtree(d + 1);
}

5. takeuchi

main() = takeuchi(n)

takeuchi(n) = tak(n, 0, n + 1)

tak(x, y, z) = if (x <= y) then y else
    tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y))

6. add

main() = {
    n ← 10000000;
    a ← (...n);
    add(a, 0, n - 1)
}

add(a, i, j) = if (i >= j) then a[i] else {
    m ← (i + j) / 2;
    add(a, i, m) + add(a, m + 1, j)
}

7. matrixMult

main() = {
    d ← 200;
    d2 ← d * d;
}

(m, n, o) ← ((...d2), (...d2), (...d2));

forevery i = 0 to (d - 1) {
  forevery j = 0 to (d - 1) {
    dotprod(m, n, d, d, i, j, o);
  }
}

dotprod(m, n, nn, pp, mvm, nvp, o) = {
  sum ← 0;
  for j = 0 to (nn - 1) {
    sum ← sum + (m[(mvm * nn) + j] * n[(j * pp) + nvp]);
  }
  o[(mvm * pp) + nvp] ← sum;
}

8. prefixSum

main() = {
  n ← 1000000;
  a ← (...n);
  prefixsum(a, n);
}

prefixsum(a, n) =
  if (n = 1) then Return else {
    if (n = 2) then {
      a[1] ← a[0] + a[1];
    } else {
      half ← n / 2;
      interim ← (...half);
for every $i = 0$ to $(\text{half} - 1)$ {
    $k \leftarrow (i \times 2) + 1$;
    interim[$i$] $\leftarrow a[k] + a[k - 1]$;
}

prefixsum(interim, half);

for every $i = 1$ to $(n - 1)$ {
    if ($(i \mod 2) = 0$) then {
        $a[i] \leftarrow a[i] + \text{interim}[(i / 2) - 1]$;
    } else {
        $a[i] \leftarrow \text{interim}[i / 2]$;
    }
}

9. primes

main() = {
    for every $i = 2$ to 10000 {
        isprime($i$);
    }
}

isprime($n$) = {
    isp $\leftarrow 1$;
    $i \leftarrow 2$;
    while ($(i \leq (n - 1)) \&\& \text{isp}$) {
        if ($(n \mod i) = 0$) then {
            isp $\leftarrow 0$;
        }
        $i \leftarrow i + 1$;
    }
10. catalan

main() = catalan(14)

catalan(n) = if (n <= 1) then 1 else {
    o ← (...n);

    forevery i = 0 to (n - 1) {
        o[i] ← catalan(i) * catalan((n - i) - 1);
    }

    sum ← 0;
    for i = 0 to (n - 1) {
        sum ← sum + o[i];
    }
    free(o);
    sum
}

11. nqueens

main() = {
    n ← 10;
    a ← (...n);
    nqueens(n, 0, a)
}

ok(n, a) = {
    isok ← 1;
    i ← 0;
}
while ((i <= (n - 1)) && isok) {
    p ← a[i];
    j ← i + 1;
while ((j <= (n - 1)) && isok) {
    q ← a[j];
    if (((q = p) (q = (p - (j - i))) (q = (p + (j - i)))) then {
        isok ← 0;
    }
    j ← j + 1;
}
i ← i + 1;
}

isok

nqueens(n, j, a) = if (n = j) then 1 else {
    count ← (...n);
    for k = 0 to (n - 1) {
        count[k] ← 0;
    }
}

torevery i = 0 to (n - 1) {
    b ← (...(j + 1));
    for k = 0 to (j - 1) {
        b[k] ← a[k];
    }
    b[j] ← i;

    if ok(j + 1, b) then {
        count[i] ← nqueens(n, j + 1, b);
    }
    free(b);
}
sum ← 0;
for k = 0 to (n - 1) {
    sum ← sum + count[k];
}
free(count);
sum

12. mergeSort

main() = {
    n ← 400000;
    a ← (...n);

    for i = 0 to (n - 1) {
        a[i] ← rand();
    }

    t1 ← time();
    mergesort(a, 0, n - 1);
    t2 ← time();
    print(t2 - t1);
}

mergesort(a, l, r) = if (l >= r) then Return else {
    m ← l + ((r - l) / 2);
    (mergesort(a, l, m), mergesort(a, m + 1, r));
    merge(a, l, m, r);
}

merge(a, l, m, r) = {
    n1 ← (m - l) + 1;
    n2 ← r - m;
\[
\text{ls} \leftarrow (\ldots n1); \\
\text{rs} \leftarrow (\ldots n2); \\
\text{for } i = 0 \text{ to } (n1 - 1) \{ \\
\hspace{1em} \text{ls}[i] \leftarrow a[l + i]; \\
\} \\
\text{for } j = 0 \text{ to } (n2 - 1) \{ \\
\hspace{1em} \text{rs}[j] \leftarrow a[(m + 1) + j]; \\
\} \\
\text{i} \leftarrow 0; \\
\text{j} \leftarrow 0; \\
\text{k} \leftarrow 1; \\
\text{while } ((i < n1) \&\& (j < n2)) \{ \\
\hspace{1em} \text{if } (\text{ls}[i] \leq \text{rs}[j]) \text{ then } \\
\hspace{2em} \text{a}[k] \leftarrow \text{ls}[i]; \\
\hspace{2em} \text{i} \leftarrow i + 1; \\
\hspace{1em} \} \text{ else } \\
\hspace{1em} \text{a}[k] \leftarrow \text{rs}[j]; \\
\hspace{1em} \text{j} \leftarrow j + 1; \\
\hspace{1em} \} \\
\text{k} \leftarrow k + 1; \\
\} \\
\text{while } (i < n1) \{ \\
\hspace{1em} \text{a}[k] \leftarrow \text{ls}[i]; \\
\hspace{1em} \text{i} \leftarrow i + 1; \\
\hspace{1em} \text{k} \leftarrow k + 1; \\
\} \\
\text{while } (j < n2) \{ \\
\hspace{1em} \text{a}[k] \leftarrow \text{rs}[j]; \\
\hspace{1em} \text{j} \leftarrow j + 1; \\
\hspace{1em} \text{k} \leftarrow k + 1; 
\]
13. coins

main() = {
    val ← 777;
    pairs ← ((250, 55), (100, 88), (25, 88), (10, 99), (5, 122), (1, 177));
    coins ← makelinked(pairs, 0, 6);
    pay(val, coins)
}

pay(val, coins) =
    if (val = 0) then 1 else {
        if (coins = Nil) then 0 else {
            (head, tail) ← (coins[0], coins[1]);
            (c, q) ← head;

            if (c > val) then pay(val, tail) else {
                if (q = 1) then {
                    (left, right) ← (pay(val - c, tail), pay(val, tail));
                    left + right
                } else {
                    newhead ← (c, q - 1);
                    coinsx ← (newhead, tail);
                    (left, right) ← (pay(val - c, coinsx), pay(val, tail));
                    free(newhead);
                    free(coinsx);
                    left + right
                }
            }
        }
    }
makelinked(a, i, n) = if (i >= n) then Nil else  
    (a[i], makelinked(a, i + 1, n))

14. mapReduce

main() = {
    n ← 25000;
    l ← 10000000;
    rt ← randomtext(n, l);

    t0 ← time();

    maps ← (...nthreads());
    forall p = 0 to (nthreads() - 1) {
        hs ← hashmap();
        maps[p] ← hs;
        domap(l, rt, p, hs);
    }

    print(time() - t0);

    pmax ← nthreads();
    pmax1 ← pmax - 1;
    reducetimes ← log(nthreads());
    for r = 1 to reducetimes {
        quickSortIntern(maps, 0, pmax1);

        pmaxlp ← pmax1;
        pmax ← pmax / 2;
        pmax1 ← pmax - 1;
        forall p = 0 to pmax1 {
            append(maps[p], maps[pmaxlp - p]);
        }
    }
}
print(time() - t0);

Return
}

domap(l, rt, p, hs) = {
    psize ← l / threads();
    size ← psize;
    if (p = (threads() - 1)) then {
        size ← l - (psize * (threads() - 1));
    }
    
    start ← p * psize;
    end ← (start + size) - 1;

    for i = start to end {
        w ← rt[i];
        if hashmapcontains(hs, w) then {
            cnt ← hashmapget(hs, w);
            hashmapput(hs, w, cnt + 1);
        } else {
            hashmapput(hs, w, 1);
        }
    }
}

randomtext(n, l) = {
    rt ← (...l);
    l1 ← l - 1;
    for i = 0 to l1 {
        rt[i] ← rand(n);
    }
    rt
quickSortIntern(a, p, r) = {
    if (p < r) then {
        q ← partition(a, p, r);
        (quickSortIntern(a, p, q - 1), quickSortIntern(a, q + 1, r));
    }
}

partition(a, p, r) = {
    x ← hashmapsize(a[r]);
    i ← p - 1;
    for j = p to (r - 1) {
        if (hashmapsize(a[j]) <= x) then {
            i ← i + 1;
            swap(a, i, j);
        }
    }
    swap(a, i + 1, r);
    i + 1
}

swap(a, i, j) = {
    temp ← a[i];
    a[i] ← a[j];
    a[j] ← temp;
}

append(hs1, hs2) = {
    keys1 ← hashmapkeys(hs1);
    ul1 ← hashmapsize(hs1) - 1;
    for i = 0 to ul1 {
        key ← keys1[i];
        v1 ← hashmapget(hs1, key);
v2 ← 0;
if hashmapcontains(hs2, key) then {
    v2 ← hashmapget(hs2, key);
}
hashmapput(hs1, key, v1 + v2);
hashmapremove(hs2, key);
}
free(keys1);

keys2 ← hashmapkeys(hs2);
u2 ← hashmapsize(hs2) - 1;
for i = 0 to u2 {
    key ← keys2[i];
    v ← hashmapget(hs2, key);
    hashmapput(hs1, key, v);
}
free(keys2);

hs1

15. bellmanFord

main() = {
    n ← 200;
    rg ← randomGraph(n);

    t0 ← time();
    ds ← (...n);
    pr ← (...n);
    for v = 0 to (n - 1) {
        ds[v] ← 0 - 1;
        pr[v] ← 0 - 1;
    }
ds[0] ← 0;

for repeat = 0 to (n - 1) {
    forall v = 0 to (n - 1) {
        uv0 ← v * n;
        uv1 ← (uv0 + n) - 1;

        u ← 0;
        for uv = uv0 to uv1 {
            if (~u = v)) then {
                w ← rg[uv];
                dsu ← ds[u];

                if (dsu >= 0) then {
                    dist ← dsu + w;
                    if (ds[v] < 0) then {
                        ds[v] ← dist;
                        pr[v] ← u;
                    } else {
                        if (dist < ds[v]) then {
                            ds[v] ← dist;
                            pr[v] ← u;
                        }
                    }
                }
            }
            u ← u + 1;
        }
    }
}

print(time() - t0);
randomGraph(n) = {
    n2 ← n * n;
    g ← (...n2);
    for i = 0 to (n2 - 1) {
        g[i] ← rand(100) + 1;
    }
    g
}

16. tsp

main() = {
    n ← 17;
    d ← randomGraph(n);
    
t0 ← time();
    expn ← 2;
    for i = 0 to (n - 1) {
        expn ← expn * 2;
    }
    g ← (...(expn * n));
    initg(g, d, n);
    for s = 2 to (n - 1) {
        subsetEnum(n, s, d, g);
    }
    
fset ← fullset(n);
    opt ← 0 - 1;
    for k = 1 to (n - 1) {
        this ← g[(fset * n) + k] + d[k * n];
        if (opt < 0) then {
            opt ← this;
        }
} else {
    opt ← mintwo(opt, this);
}

print(time() - t0);

}

randomGraph(n) = {
    n2 ← n * n;
    g ← (...n2);
    for i = 0 to (n2 - 1) {
        g[i] ← rand(100) + 1;
    }
    g
}

fullset(n) = {
    fset ← 1;
    for j = 1 to (n - 1) {
        fset ← (fset * 2) + 1;
    }
    fset
}

initg(g, d, n) = {
    for k = 1 to (n - 1) {
        kset ← 1;
        for j = 1 to (n - 1) {
            if (j = k) then {
                kset ← (kset * 2) + 1;
            } else {
                kset ← kset * 2;
            }
        }
    }
}
subsetEnum(n, s, d, g) = {
    subsetEnumRec(1, n, s, 1, 1, 0 - 1, d, g);
}

subsetEnumRec(i, n, s, a, b, taken, d, g) = {
    if (s <= 0) then {
        for k = i to (n - 1) {
            a ← a * 2;
            b ← b * 2;
        }
        tspCore(a, b, taken, n, d, g);
    } else {
        (p1(i, n, s, a, b, taken, d, g),
         p2(i, n, s, a, b, taken, d, g),
         p3(i, n, s, a, b, taken, d, g));
    }
}

p1(i, n, s, a, b, taken, d, g) = {
    if (i < (n - s)) then {
        subsetEnumRec(i + 1, n, s, a * 2, b * 2, taken, d, g);
    }
}

p2(i, n, s, a, b, taken, d, g) = {
    if ((taken >= 0) (s > 1)) then {
        subsetEnumRec(i + 1, n, s - 1, (a + 2) + 1, (b + 2) + 1, taken, d, g);
    }
}
p3(i, n, s, a, b, taken, d, g) = {
    if (taken < 0) then {
        subsetEnumRec(i + 1, n, s - 1, (a * 2) + 1, b * 2, i, d, g);
    }
}

tspCore(s, sk, k, n, d, g) = {
    n1 ← n - 1;

    min ← 0 - 1;
    skk ← sk;
    for i = 1 to n1 {
        bit ← skk % 2;

        if bit then {
            m ← n - i;

            this ← g[(sk * n) + m] + d[(m * n) + k];

            if (min < 0) then {
                min ← this;
            } else {
                min ← mintwo(min, this);
            }
        }
    }

    skk ← skk / 2;
}

g[(s * n) + k] ← min;

mintwo(x, y) = if (x < y) then x else y
APPENDIX B: USAGE OF RE-LANG AND OVERVIEW OF IMPLEMENTATION

In this chapter, we will walk through all the necessary practical steps to compile a RE-LANG program using our current implementation. By doing this, we aim to achieve two goals:

1. The reader will understand exactly how to use our current RE-LANG compiler implementation to compile and execute any RE-LANG code on any mainstream computer.

2. The reader will gain an overview knowledge of how our RE-LANG compiler works in practice.

B.1 Frontend Overview

We implemented RE-LANG in two major components—the frontend and the backend, which are both software projects based on existing programming languages. The frontend is a project based on the Scala programming language\(^1\) version 2.13.8. The frontend project\(^2\) (aka. the RE-LANG compiler) includes four sub-components organized into four separate folders of Scala files:

- The parsing folder contains the grammar parser and a complete abstract syntax tree (AST) definition of RE-LANG. We implemented the parser using the parboiled\(^3\) library in Scala.

- The core folder contains the procedures for smart assignment elimination. Functions and algorithms in the AssignmentElimination.scala file is used by the frontend toolchain to compile an imperative-style RE-LANG program into a ”pure” program based on rewriting semantics.

- The compile folder contains procedures to compile an equational-style rewriting program produced by the previous pipeline step into token-based rewrite rules. This can be understood as further lowering the semantics of a rewriting program. The key procedure is the compileAll function in the file Compiler.scala.

- The codegen folder contains implementations of the last step in the RE-LANG frontend toolchain—the EmitIntegrationFile function in the CppRuntime.scala takes the result from the previous pipeline and produces a .cpp file containing C++ code. This file will then be used to integrate with the RE-LANG backend and produce the final executable.

---

\(^1\)https://www.scala-lang.org  
\(^2\)https://github.com/uncttao/re-lang  
\(^3\)https://github.com/sirthias/parboiled2
As of the conclusion of this thesis, the current state of the frontend compiler project can be summarized as the following:

• The RE-LANG compiler is fully working and fully automatic, meaning there is no step in the compiling process that requires manual program transformation effort\(^4\).

• The compiler works on all the test programs in the `Examples.scala` file in the sense that it produces code that we have tested to work successfully on a series of moderately complex inputs (e.g., the `bubbleSort` program successfully sorts and prints out a sequence of 100 random numbers).

• Besides the test examples, the compiler should work on all RE-LANG programs, meaning there is no known bug or critical flaw in the compiler that is pending for fixing. However, such an error is possible, in which case we politely ask the reader to contact us or submit a Github issue/pull request.

A few limitations for the frontend project are (1) There is no dedicated editor, nor associated convenient development features such as syntax highlighting or code completion. (2) The `parboiled` parser does not produce a friendly enough grammar error report. We delegate these minor but admittedly essential features to future engineering side projects to be completed by the community\(^5\).

### B.2 Compile a RE-LANG program

To compile a RE-LANG program, simply observe the following few steps:

1. Download the RE-LANG project from Github by cloning the repository using Git\(^6\).

2. Download and install the Scala language toolchain from their official website\(^7\).

3. Write a RE-LANG program in the correct syntax in your favorite editor.

4. Execute the `Main.scala` file in the project using the `sbt` program in the Scala toolchain and passing in the text file containing your RE-LANG program.

---

\(^4\)Maybe to some readers’ surprise, not all research language compilers can confidently state this!

\(^5\)This includes us.

\(^6\)https://github.com/uncttao/re-lang

\(^7\)https://www.scala-lang.org
5. A C++ file representing the RE-LANG program will be generated in the project folder with the same prefix as your input program file.

However, the resulting C++ file is not a standalone program. It needs to be integrated with the RE-LANG backend in order to produce a final executable.

B.3 Backend Overview

The backend project of RE-LANG is a project implemented using the C++ programming language and relies on features in the C++20 standard and OpenMP framework. Hence, in order to execute any RE-LANG program, the user needs a fairly modern and mainstream C++ compiler, such as GCC.

The C++ project (nicknamed “Iron Stream Rewriting”) is organized in the following way.

- The *domain* folder contains C++ struct definitions that represent the stream rewriting computation model. In particular, the `LinkedToken.h` structure is the cornerstone of the model and should serve as the key reference in the case of confusion.

- The *runtime* folder contains a complete implementation of the parallel stream rewriting machine. The key file is the `streamrewriter.h`, which contains the runtime algorithm, while other files are data structures and helper procedures used as supporting roles.

- The `streamrewriter.cpp` is the "C++ integration file". The idea is to overwrite the content of this file with the generated file from the RE-LANG frontend compiler. Then compiling the C++ project would yield an executable that runs the RE-LANG program. By default, the `streamrewriter.cpp` represents the implementation of the naive Fibonacci function.

As of the conclusion of this thesis, the current state of the backend runtime project can be summarized as the following.

- The stream rewriting runtime is fully working and has no bug or critical flaw we are aware of.

---

8[https://www.openmp.org/](https://www.openmp.org/)
9[https://gcc.gnu.org/](https://gcc.gnu.org/)
10[https://github.com/uncttao/iron-stream-rewriting-cpp](https://github.com/uncttao/iron-stream-rewriting-cpp)
• The C++ project is standalone and cross-platform-friendly—we did not rely on any platform-specific features or implementation. The project is also bootstrapped using CMake\textsuperscript{11} version 3.16, a cross-platform C++ project maker. This means that in principle the RE-LANG runtime can run on any platform supported by CMake and a modern C++20 compiler.

• The project relies on mimalloc\textsuperscript{12}, a high-performance multi-threading memory allocation library. The user needs to have this library built and/or installed on the target machine.

• On any platform, the performance of the parallel stream rewriting machine is dependent on two factors: (1) How efficiently the machine interacts with shared memory. (2) How fast the processors synchronize with each other (i.e.\texttt{omp barrier} latency). Additionally, the runtime can already take advantage of single instruction, multiple data (SIMD) in modern processors for closure initialization. It is expected to see a significant performance boost for processors that feature high-performance SIMD pipelines.

B.4 Execute a RE-LANG program

To execute a RE-LANG program, simply observe the following steps.

1. Download the runtime project from Github by cloning the repository using Git\textsuperscript{13}.

2. Download and install mimalloc\textsuperscript{14} following the necessary steps according to their instructions.

3. Generate a C++ integration file following the previous frontend compilation steps.

4. Type \texttt{sh ./run.sh <Path to the integration file> <Number of processors>} in the command line to trigger the script that compiles the C++ project using CMake and runs the program using the designated number of processors.

\textsuperscript{11}https://cmake.org
\textsuperscript{12}https://github.com/microsoft/mimalloc
\textsuperscript{13}https://github.com/uncttao/iron-stream-rewriting-cpp
\textsuperscript{14}https://github.com/microsoft/mimalloc


van Dijk, Tom & Jaco van de Pol. 2014. Lace: Non-blocking Split Deque for Work-Stealing. In *Euro-Par Workshops*.


