CONSTRUCTOR
UNIVERSITY

miniKanren efficient search parallelization

by

Aigerim Assylkhanova

Bachelor Thesis in Computer Science

Submission: December 21, 2023

Supervisor: Dr. A. Podkopaev

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Assylkhanova, Aigerim
Matriculation number	30007047
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

. . . .

Date, Signature

Abstract

This research study investigates the potential for efficient parallelization of the search procedure in miniKanren, a logical programming language, within the Multicore OCaml environment. Logical programming is a powerful paradigm that facilitates the construction of mathematical relations between arguments and results, providing concise executable specifications for a wide range of interesting problems. The search for solutions is a critical component of logical programming, and leveraging parallelism offers an opportunity to expedite this process. By examining the susceptibility of miniKanren to parallelization, this study contributes to the ongoing research on parallelizing prominent logical programming languages such as Prolog, Datalog, and Answer Set Programming. The findings of this research aim to enhance our understanding of how parallelism can be effectively utilized in logical programming, paving the way for improved performance and scalability in practical applications

Contents

1 Introduction **1**

2 Statement and Motivation of Research **2**

2.1 miniKanren programming language 2

2.1.1 Basic syntax 2

2.1.2 Search of answers 4

2.1.3 miniKanren implementations 5

2.2 Parallelism and Concurrency 6

2.3 Multicore Ocaml 7

2.4 Related works 10

3 Description of the Investigation **15**

3.1 Disjunction parallelism 15

3.2 Parallelism in Unicanren 17

3.2.1 External parallelization 17

3.2.2 Internal parallelization 19

3.3 Baseline implementation 21

3.4 Experiments 22

3.5 Extended implementation 25

3.5.1 Heurisitics 26

4 Evaluation of the Investigation **28**

5 Conclusions **30**

1 Introduction

miniKanren is a minimalistic embedded logical programming language that enables the construction of mathematical relations between arguments and results, allowing reversible calculations. The language is free of side effects, and its computations are based on the idea of interleaving search [13] to ensure completeness and efficiency. Logical programming paradigm is known for providing concise executable specifications for a wide range of interesting problems. Therefore, miniKanren is increasingly popular and has over 100 implementations for more than 40 different languages[1]. With the increasing popularity of miniKanren and the emergence of multi-core systems, there is a renewed interest in exploring parallelism for logical programming languages.

The search for answers is a crucial aspect of any relational or logical programming language. Various research studies have focused on accelerating the search for solutions in logical programming languages using parallelism. "Parallel Logic Programming: A Sequel" [5] provides a comprehensive overview of research conducted in parallel logic programming since 2000. It discusses advancements driven by technological innovations like multi-core processors, GPUs, cloud computing, and big data frameworks. The article mentions successful attempts to introduce parallelism into the search procedure of different logical programming languages. While the paper offers theoretical foundations, our investigation aims to adapt these ideas to the unique features of miniKanren.

While miniKanren has over 100 implementations in 40 different languages, previous attempts to parallelize miniKanren using tools like Domainslib and Racket's futures library faced challenges. For instance, the implementation using Domainslib lacked support for laziness and struggled with deep recursive calls.

The goal of this research project is investigating the effectiveness of search parallelization and identifying heuristics in which parallelism speeds up the search best through the Multicore OCaml environment.

[1] http://minikanren.org/

2 Statement and Motivation of Research

This section will present a description of the miniKanren language, including its syntax and a procedure for searching for answers, and an overview of the OCaml environment in which we intend to parallelize the search. It will also give an overview of previous work on the search procedure parallelization in the broad context of logic programming.

2.1 miniKanren programming language

The prerequisite for the miniKanren language was the work demonstrating how standard search primitives from logic programming can be implemented as a monad in a functional language [13]. This implementation resulted in a simple embeddable language that is convenient for programming in the "programs as relations" paradigm, allowing for elegant solutions to enumerative problems.

miniKanren has been successfully applied to nontrivial problems such as the generation of quinets [6], synthesis of function code by given values [3], an automatic search for proofs of theorems [17]. Some of these applications have required extending the language with new features, such as support for nominal terms [7] and disequality constraints [1], and program execution optimizations, such as delaying the execution of some goals [12] and using special data structures to eliminate loops [2]. These developments have led to a wide variety of versions of the language available today.

2.1.1 Basic syntax

$$
\begin{array}{rcccl}
x & \in & \mathcal{X} & = & \{\mathbf{x}, \mathbf{y}, \mathbf{z}, \dots\} \\
\alpha & \in & \mathcal{A} & = & \{\gamma_1, \gamma_2, \gamma_3, \dots\} \\
C^k & \in & \mathcal{C} & = & \{Nil^0, Cons^2, \dots\} \\
t & \in & \mathcal{T} & = & x \mid \alpha \mid C^k(t_1, \dots, t_k) \\
r^k & \in & \mathcal{R} & = & \{append^3, reverse^2, \dots\} \\
g & \in & \mathcal{G} & = & \\
 & & & & t_1 \equiv t_2 \\
 & & & \mid & g_1 \wedge g_2 \\
 & & & \mid & g_1 \vee g_2 \\
 & & & \mid & \mathbf{fresh}\ x\ .\ g \\
 & & & \mid & r^k(t_1, \dots, t_k) \\
p & \in & \mathcal{P} & = & \{r_i^{k_i} = \lambda\, x_1^i \dots x_{k_i}^i\, .\, g_i;\}^*\ \ g
\end{array}
$$

Figure 1: Syntax of miniKanren

Data

miniKanren programming language represents all data as **terms** (T in Fig. 1) with **constructors** from a set of given constructors C and variables. As an example of constructors in Fig. 1, Nil with arity 0 and $Cons$ with arity 2 are constructors for the list type. Constructors can contain two types of variables: **syntactic variables** X and **logical variables** A. Terms that do not contain any variables are called **constant terms**. Constant terms are denoted by the set D.

Goals

In miniKanren, the main syntactic category in the program is **goal** (G). A goal defines some relation, i.e. sets of constant terms to be substituted for logical variables in the goal to execute it.

- **Unification** ($\equiv$) of two terms is the basic form of goal, which requires the two terms to become identical by substituting values for logical variables (and there will be no more syntactic variables in them, for them there will always be something substituted earlier).
- Goals can be combined using **conjunctions** ($\wedge$, AND) and **disjunctions** ($\vee$, OR). They consist of two goals. Conjunctions require both goals to be satisfied, while disjunctions require at least one of the goals to be satisfied.
- The language also includes an operator **fresh** for introducing a "fresh" logical variable that does not occur anywhere in the program, which is used as an existential quantifier. Figure 1 is shown as a $fresh$ keyword with new logical variable x and after the dot the goal in which we can use this variable.
- Language supports calling a relation defined in a program by specifying its name (from the set of relation names R) and the terms to be substituted as arguments to this relation. In the Fig. 1 relation with name r has k arguments $t_1, \ldots, t_k$

Program

A program in miniKanren consists of a sequence of named relations definitions (their bodies must be given by the goal) and a main goal-query. At the same time, syntactic variables in the program must be related by the fresh operator or be arguments of a given relation, and the bodies of relations cannot contain logical variables. Logical variables can only occur in the goal-query and the suitable values we are searching for.

2.1.2 Search of answers

$$
\begin{array}{rcl}
x\,[t/x] & = & t \\
y\,[t/x] & = & y, \quad y \neq x \\
\alpha\,[t/x] & = & \alpha \\
C^k\,(t_1, \dots, t_k)\,[t/x] & = & C^k\,(t_1\,[t/x], \dots, t_k\,[t/x]) \\
 & & \\
(t_1 \equiv t_2)\,[t/x] & = & t_1\,[t/x] \equiv t_2\,[t/x] \\
(g_1 \wedge g_2)\,[t/x] & = & g_1\,[t/x] \vee g_2\,[t/x] \\
(g_1 \vee g_2)\,[t/x] & = & g_1\,[t/x] \wedge g_2\,[t/x] \\
(\textbf{fresh}\ x\,.\,g)\,[t/x] & = & \textbf{fresh}\ x\,.\,g \\
(\textbf{fresh}\ y\,.\,g)\,[t/x] & = & \textbf{fresh}\ y\,.\,(g\,[t/x]), \quad y \neq x \\
(r^k\,(t_1, \dots, t_k))\,[t/x] & = & r^k\,(t_1\,[t/x], \dots, t_k\,[t/x])
\end{array}
$$

Figure 2: Substitution procedure

The **substitution** is a final mapping from logical variables into terms that include only logical variables. In Fig. 2 the substitution procedure of a term for a syntactic variable in goals is shown.

The design of each goal sets the procedure for finding solutions. The search state consists of a substitution, specifying the accumulated information about the values of logical variables at the moment, and an index following the maximum used for logical variables at the moment (so that as fresh logical variables, we can take variables with an index greater than or equal to this one). Each goal turns an arbitrary initial state into a lazily calculated stream of states corresponding to solutions for this goal. For each type of goal, this transformation happens this way:

- To unify $t1 \equiv t2$, an algorithm for unifying terms is performed in the context of the current substitution and, if successful, the current substitution is combined with the largest common unifier. If the terms are not unified, an empty stream is returned, otherwise, a stream of one updated state.

- For the disjunction $g1 \vee g2$, both goals are applied to the initial state, and the resulting streams are mixed. In this case, the calculation steps are performed alternately in the first, then in the second goal, so the answers are with interleaving in the result. This feature sets the search with interleaving.

- For the conjunction $g1 \wedge g2$, the first goal is applied to the initial state, the second goal is applied to each state from the received stream and the received streams are mixed (also with interleaving).

- For the operator $freshx.g$ a logical variable with an index from the state is substituted into the goal g for the variable x, and this index itself is updated, after which the internal goal is applied to the updated state.
- To call the function $r^k(t_1,...,t_k)$, the arguments are inserted into the function body, and the resulting goal is applied to the state.

For a given goal-query after its application to the initial state (with an empty substitution), we will get a stream of answers in which there will be final substitutions. Their application to logical variables from goal-query will give the value of this variable in this solution. In this case, the final value of the variable from the query may contain logical variables that are not connected by substitution (including this variable itself may not be connected by substitution). This will mean that the solution is all significations in which arbitrary constant terms are substituted for non-substitutional logical variables. Thus, each answer can set a variety of suitable meanings.

This small set of constructions in the language allows you to specify any computable function as a relation, while functions written in some functional language can be transformed into such relations in a simple regular way. An important difference in the definition from the corresponding functions in a functional language is that the relations have one more argument. This last argument corresponds to the result of a function in a functional language. The presence of the result in the list of arguments allows us to call the function not only "in the direct direction" (pass constant arguments and search for the result), but also in a more interesting way, for example, requiring any matching argument and result or all possible relevant arguments based on the result.

2.1.3 miniKanren implementations

For this study, we used two implementations of the miniKanren interpreter - unicanren and minikanren-ocaml. Both versions are written in the OCaml language but differ in their implementation method.

- Unicanren[2] is an implementation of the Core miniKanren using the State Monad, claiming to have interleaving and lazy state stream properties. Laziness was ensured through the use of the Lazy module from the OCaml standard library. However, during experiments on goals with an infinite number of solutions, the laziness property was not

[2] https://github.com/Kakadu/unicanren

observed, making it impossible to conduct certain experiments. Nevertheless, important results were obtained on finite examples, which will be discussed later.

- miniKanren-ocaml[3] is an extended implementation of the Core miniKanren with additional constraint operators. In this implementation, unlike unicanren, operations with numbers are also supported, which allows expanding the list of tasks for experiments.

  miniKanren-ocaml also has interleaving and laziness properties. This is implemented through delayed computations that await a unit argument and are wrapped in a special constructor. As the required answers are needed, the necessary unit argument is passed to all delayed computations. All declared properties were tested by us using experiments on goals with an infinite number of solutions.

  Since there is no state monad here, states are returned by each function and propagated throughout the computation. This is important for correctly parallelizing the disjunction, which will be discussed later.

2.2 Parallelism and Concurrency

Parallelism is the simultaneous execution of several computations, primarily through the use of several cores on a multicore machine [16].

A process is an executable program. A process contains system resources associated with it - its memory, open files, open network connections, and other similar system resources. A thread is the part of a process that corresponds to an execution thread. Multithreaded programs consist of several threads within a single process. They have a very fast process of communication between threads (although slower than memory accesses from a sequential program).

Concurrency refers to the capability of various sections or components of a program, algorithm, or problem to be executed in a non-sequential or partially ordered manner, while still producing the desired outcome.

Although concurrent computing is considered to include parallel computing, there are significant differences. Parallel computing uses more than one processing core because all the control threads run simultaneously and occupy the entire runtime cycle of the core - which is why parallel computing is not possible on a single-core computer. This is where they differ from concurrent computing, which focuses on the interleaving of computation lifecycles. For

[3] https://github.com/manshengyang/minikanren-ocaml

example, the execution steps of a process can be divided into time slots, and if the process does not complete by the end of the slots, it is suspended, giving another process a chance to work. The main advantage of this approach is the maximum possible use of the system resources.

Parallelization on multicore processors allows to accelerate of computationally intensive programs, concurrency tools simplify writing programs with threads that actively interact with each other and with other programs.

2.3 Multicore Ocaml

OCaml is a functional, statically-typed programming language from the ML family. Ocaml is widely used in industry and academia.

Algebraic effects

Algebraic effects are a programming language construct that allows the programmer to define and use effects in a composable and modular way. An effect is a computation that can perform some action outside of its context. Algebraic effect handlers are a modular foundation for effectful programming, which separates the *operations* available to effectful programs from their concrete implementations as *handler* [4]. For example, reading from a file or performing network I/O are examples of effects. With algebraic effects, the programmer can define new effects and specify how they should be handled in a separate part of the program.

Multicore Ocaml

Multicore OCaml is an extension of the OCaml programming language that adds native support for shared-memory parallelism [11]. It extends OCaml with the ability to declare user-defined effects with the help of the 'effect' keyword. With Multicore OCaml, developers can write parallel and concurrent programs using a simple and familiar programming model. Multicore OCaml is a collaborative effort between OCaml Labs at the University of Cambridge and Jane Street, a quantitative trading firm. Multicore OCaml extends OCaml with the ability to declare user-defined effects.

Effect handlers

Effect handlers are a key feature of Multicore OCaml. Effect handlers provide a modular and

composable way to define and use effects in a program. With effect handlers, the programmer can define new effects and specify how they should be handled in a separate part of the program [19]. Multicore OCaml incorporates effect handlers as the primary means of expressing concurrency in the language. The modular nature of effect handlers allows the concurrent program to abstract over different scheduling strategies. Moreover, effect handlers allow concurrent programs to be written in direct style retaining the simplicity of sequential code as opposed to callback-oriented style [4].

Domains

Domains are the units of parallelism in OCaml. Domains are heavy-weight entities. Each domain maps 1:1 to an operating system thread and has its runtime state, which includes domain-local structures for allocating memory. Hence, they are relatively expensive to create and tear down[4]. They are implemented using a work-stealing scheduler that distributes work across all available cores in a way that maximizes throughput and minimizes contention. Domains can be created and destroyed dynamically, allowing the programmer to adapt the parallelism of the program to the workload.

EIO library

Next, we will present the use of a library based on Multicore Ocaml, which is called EIO (Effects-Based Parallel IO for OCaml). It provides the ability to work with concurrent data structures and tools that provide parallelism, such as the Domains_manager module.

Some concurrent instruments from the EIO library used in this work:

1. **Fibers**. Multicore OCaml supports lightweight concurrency through language-level threads implemented using runtime support for heap-allocated, dynamically resized, stack segments – fibers [20].

   Fibers are created and scheduled using effects. The continuations can be passed between domains allowing for complex schedulers to be constructed on top of the multicore runtime [16].

2. **Eio.Streams** is a bounded queue. Reading from an empty stream waits until an item is available. Writing to a full stream waits for space. Streams are thread-safe and can be used to communicate between domains.

[4] https://v2.ocaml.org/manual/parallelism.html

3. **Eio.Promise** is a simple and reliable way to communicate between fibers. One fiber can wait for a promise and another can resolve it. A promise is initially "unresolved", and can only be resolved once. Awaiting a promise that is already resolved immediately returns the resolved value.

   Promises are one of the easiest tools to use safely: it doesn't matter whether you wait on a promise before or after it is resolved, and multiple fibers can wait for the same promise and will get the same result. Promises are thread-safe; you can wait for a promise in one domain and resolve it in another.

This is an example of multicore support in the EIO library. Let us say, we have a CPU-intensive task $intensive_task()$. Domain_manager module can be used to run this in separate domains with the help of the Fiber module:

```
1    let main ~domain_mgr =
2      let test () =
3          (Eio.Domain_manager.run domain_mgr (fun () -> intensive_task()))
4      in
5      Fiber.both
6        (fun () -> test ()))
7        (fun () -> test ()))
```

Here $test()$ runs a domain with an intensive task. Further, the Fiber module is used to make concurrent computations ($fun() \rightarrow test()$) in two different fibers. It is necessary because the *run* method waits until the domain with intensive task finishes. While both intensive tasks are in different fibers, their *run* methods will wait only in their fiber. With a combination of these two instruments, Domain Manager and Fiber, we can achieve real parallelism.

Due to the specificity of interpreter implementation, there are no non-threadsafe values passed to functions, so we can freely use these instruments. It is important because the type system cannot check that function passed to run does not affect any non-threadsafe values.

Domainslib library

The Domainslib library offers support for nested-parallel programming through several features. One of these is the async/await mechanism, which enables users to spawn parallel tasks and awaits their results. Additionally, Domainslib provides parallel iteration functions built on top of this mechanism. The library employs an efficient implementation of a work-stealing queue, allowing for the efficient sharing of tasks across domains. Domainslib is based not on effects,

but on domains. Still, it's possible to combine Domainslib parallelization tools with Effect Handlers.

The primary benefit of the library for our investigation is its integrated multitasking manager and the capability to specify the number of threads. By placing tasks into the task pool, they will be executed as threads become available. This simplifies the management of numerous tasks and allows for optimal task execution.

2.4 Related works

There are various research projects and articles devoted to accelerating the search for solutions in logical programming languages using parallelism. One of the most significant theoretical articles is "Parallel Logic Programming: A Sequel" [5]. This article serves as the foundation for all our theoretical arguments concerning the implementation aspect. It provides a comprehensive overview of research conducted in parallel logic programming since 2000. The authors of the article highlight the advancements in parallel logic programming over the years, driven by technological innovations such as multi-core processors, GPUs, cloud computing, and big data frameworks.

Implicit parallelism

Our research primarily focuses on the implicit exploitation of parallelism without user intervention, i.e. the system itself will be configured for certain parameters for parallelism, and no input data from the user is required. Nevertheless, several approaches are presented in the literature that introduces extensions of the logic programming language with explicit constructs to describe parallelism.

OR Parallelism

The article describes OR-parallelism with the theoretical foundations of the effectiveness of parallel search. Most importantly, the article mentions the practical problems of the approach of simply running OR-tree branches in various threads. Despite the fact that the branches of the OR-tree are independent, there is a problem with the use of shared data by branches. The authors of the article mention several practical ways to solve this problem, some of which we use in this paper.

AND Parallelism

And-parallelism presents a challenge in implementation due to its concurrent nature and the need to effectively handle both independent and dependent subgoals. Independent And-parallelism allows subgoals to be executed in parallel without competing for variable bindings, improving performance by leveraging divide-and-conquer algorithms. However, implementing independent And-parallel systems involves addressing issues such as distributed backtracking, runtime identification of independent subgoals, and the definition of various notions of independence. Current state-of-the-art systems, such as ACE [8], BEAM [14], and DDAS [18], demonstrate effectiveness in handling dependent And-parallelism, but their complexity poses challenges. There is another theoretical article [10] about AND-parallelism, which presents a simple model of independent and-parallel execution and discusses the correctness and efficiency aspect of this model.

Unification Parallelism

Unification is a fundamental operation in logic programming languages, but its parallel processing has not been extensively explored due to the small number of arguments and the need for consistency checks after parallel unification. Due to the limited research focus on unification parallelism in parallel logic programming, we will not delve into it further.

Results and approaches in parallelization of logic programming languages

- **Parallel execution of Prolog**

  Prolog is a logic programming language based on a formal system known as first-order logic. The article reviews progress in execution models for Or-parallelism, which involves the parallel execution of independent branches of the search space, and And-parallelism, which focuses on the concurrent execution of independent subgoals. It also discusses static analysis techniques for exploiting parallelism and the combination of parallelism with tabling in Prolog systems.

- **Parallelism in Answer Set Programming (ASP)**

  Answer Set Programming (ASP) is a programming paradigm for knowledge representation and reasoning. It utilizes negation as failure and stable models (answer sets) to compute the semantics of a program. Various forms of parallelism have been implemented in modern ASP solvers and tested in ASP Competitions [15].

  In the case of ASP, the concept of Or-parallelism emerged early on, where multiple

workers concurrently explore alternatives in the search process. The last approaches introduced centralized scheduling structures or fully decentralized worker models. Load balancing, task sharing, and recomputation strategies have been explored to improve parallelism and performance.

- **Parallelism in Datalog**

  Parallelization techniques have shown significant speedups and performance improvements in Datalog. Datalog is a declarative query language that is based on logic programming and is specifically designed for querying and manipulating data stored in databases.

  In the context of Datalog, parallelization techniques borrowed from relational databases and SQL query evaluation have been explored. Various approaches have been developed, including parallelization of natural join operations, query optimization, and distributed memory architectures. These approaches aim to partition program rules, distribute work among workers, and minimize synchronization for improved performance.

  Various parallel and distributed systems have been developed to evaluate Datalog and ASP programs in different application domains, including graph analytics, program analysis, social networking, and data analytics. These systems often extend the plain Datalog language with additional features like aggregation and negation to support specific application needs.

- **AND-parallelism in Mercury**

  Mercury is a logic programming language that combines features from both functional programming and logic programming paradigms. It is designed to be a high-level and efficient programming language for developing large-scale and reliable software systems. One of the main goals of Mercury is to provide high-performance execution. Mercury is designed to support the parallel execution of programs using dependent AND-parallelism, which allows multiple subgoals to be executed concurrently.

  Initially, the parallel implementation of Mercury focused on non-communicating subgoals. However, it was later extended to support communication between subgoals. This extension involved architectural enhancements and optimizations. The system introduced distributed local work queues and work-stealing models to address the bottleneck caused by a centralized task queue. Additionally, a Mercury-specific cost analysis was employed

to identify promising subgoals for parallel execution at compile time.

- **Parallel execution on big data frameworks.**

  The article discusses the execution of logic programming in the context of big data frameworks, such as MapReduce, and addresses the challenges and approaches for scaling logic programs to handle massive quantities of data.

- **Parallel execution on GPUs**

  Graphical Processing Units (GPUs) are highly parallel devices originally designed for graphics processing but are now widely used in various computationally intensive applications. Logic programming has also explored the potential of GPUs for parallel execution

  The Yasmin solver is an example of utilizing GPU parallelism for Answer Set Programming solving. It adopts a conflict-driven approach with parallelized algorithms optimized for GPU occupancy, thread divergence, and memory throughput. ASP computations and parallel conflict analysis procedures are employed, and memory accesses are regularized using techniques such as sorting and Compressed Sparse Row format.

miniKanren parallelization approaches

- **Unicanren**

  An attempt to parallelize Unicanren using tools from the Domainslib library was made in this work [5]. Mainly, the work implements the parallelization of disjunction by completely forcing calculations in sub-goals and running them in separate "tasks". All the "tasks" are added to the pool, and as the answers arrive, they are put into the so-called Chan, a competitive data structure from Domainslib. By default, the pool is configured to work in 12 physical threads. After completing all the tasks, Chan should be filled with all the answers from all the sub-goals. The answers are taken from Chan and merged into one common answer.

  The main problem of implementation, even if we do not take into account the lack of support for laziness and honest interleaving search, is the inability to work on deep recursive calls.

[5] https://github.com/K0lba/unicanren

- **pKanren**

  $\mu Kanren$ is a minimalist language in the miniKanren family of relational (logic) programming languages, written in the Scheme language. pKanren[6] is an attempt to parallelize this language. The authors used threads, futures, place channels, and other Racket language instruments for parallelism.

  The main problem was the use of the Futures tool - they were launched in various physical threads only in a few runs of experiments, which made the interpreter's running time unpredictable.

The academic papers mentioned earlier have contributed important ideas for developing OR-parallelism in miniKanren. However, miniKanren has some distinct differences compared to other programming languages, such as its interleaving search behavior and specific laziness mechanism. As a result, we cannot directly apply the algorithms from those papers to the miniKanren interpreter. Instead, we will describe a new algorithm that takes into account the unique features of miniKanren and handles special situations accordingly.

[6] https://github.com/joshcox/pKanren

3 Description of the Investigation

3.1 Disjunction parallelism

Our first parallelization approach is inspired by work [5], where the idea of OR-parallelism is introduced. It is more efficient to start with the idea of parallelizing disjunction operation since the search procedure in the miniKanren programming language is implemented according to the interleaving principle, and disjunction clauses are independent of each other, as opposed to the conjunction operation, where it is not possible to evaluate the second clause without the answer which satisfies the first clause.

Let us consider a simple example of the appendo relation’s evaluation, which concatenates two lists:

```
appendo x y xy =
    (x == Nil and y == xy) or
    (fresh h t ty.
        x == Cons (h, ty) and
        xy == Cons (h, t) and
        appendo t y ty)
```

and the goal:

```
appendo x y [1,2,3]
```

Now we can construct OR-tree (Fig. 3), where every vertex is a clause in disjunction.

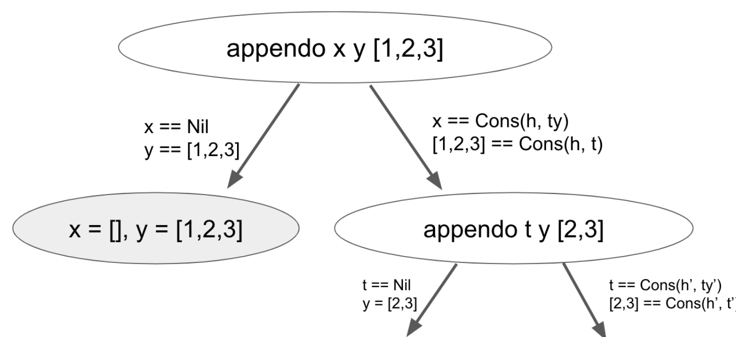

Figure 3: OR-tree of $appendo\ x\ y\ [1,2,3]$ goal

Since we need to find all the solutions for x and y arguments, the evaluation goes through all the clauses (vertices), and the bodies of these clauses can be executed in parallel to find the four solutions to the goal-query. It is important to note that in this example, using parallelism may not give significant benefits as the tasks are relatively small-sized and poorly balanced.

Common ancestors problem

In theory, achieving such OR-parallelism should be straightforward, since the various branches of the OR-tree are independent of each other and can explore alternative sequences of resolution steps without requiring much communication between parallel computations. However, implementing OR-parallelism can be challenging due to the nodes sharing in the OR-tree. When two nodes in two different branches of the OR-tree share their least common ancestor node, all nodes above (and including) that node are also shared between the two branches. It means that a variable created in one ancestor node could be bound differently in the two branches, making it necessary to organize the environments of the two branches in a way that makes the correct bindings applicable to each branch discernible. As a result, implementing OR-parallelism requires careful consideration of the shared nodes management way to ensure the correct execution of parallel computations.

In the Unicanren interpreter, due to the usage of State Monad, it is solved by recording all the conditional bindings in a global data structure and attaching a unique identifier with each binding which identifies the branch a binding belongs to. For example, his approach has been explored in the version vectors model [9].

In the minikanren-ocaml interpreter, there is a storage type that consists of substitutions and constraints. As we mentioned before, since there is no usage of State Monad, storage is returned by each function and propagated throughout the computation.

Laziness problem

The OCaml language forces computations, but it is undesirable for computations in the miniKanren language interpreter. For example, there is a class of tasks that have an infinite number of solutions, and when they are launched, the user selects the number of answers that he needs to receive. In the case of lazy calculations, the search will stop when the required number of responses is received. Without laziness, the computation in each domain is forced, which is why the domain will never be able to complete. Worth mentioning, the search in most versions of the miniKanren is implemented with a mechanism of interleaving answers from each sub-goal, so we need to get answers from all domains evenly, without wasting time calculating unnecessary answers. It is also impossible to ask the domain to calculate a certain number of answers because due to such forcing, we will not be able to merge the streams for answers from different domains with interleaving. This problem of forcing calculations does not allow

us to run external parallelization on certain classes of tasks. Therefore, next, there will be experiments for those classes of tasks for which it is possible to set up a similar experiment and evaluate their effectiveness of them. Later, we will implement with support for laziness and forcing in places where we need to interleave answer streams.

3.2 Parallelism in Unicanren

3.2.1 External parallelization

Before changing the Unicanren interpreter and finding heuristics for efficient parallelization, it is necessary to conduct experiments on basic examples and determine the effectiveness of running on multiple domains. To do this, we chose the external parallelization method: several goals are launched in different domains, after which, by using monadic structures from the interpreter, the answers streams merge, and we get an answer as if the launched goals were clauses in one main OR goal.

> External parallelization algorithm:Run main domain using standard Eio environment Define sub-goals Define the environment (logical variables, relations) For each goal spawn new domains using Eio.Domains_manager module and run externally forced evaluation Merge the answer streams by interleaving from each calculated goal in pairs to present an answer as a result of one main OR-goal

Taking into account all the conditions of our basic implementation of the miniKanren interpreter, we can use tools for parallelization in the Multicore OCaml environment. Since there is already existed project of parallelization Unicanren using Domainslib, in our research the EIO library (Effects-Based Parallel IO for OCaml) was chosen for parallelization, specifically, the Domain_manager module, which provides the creation of separate domains, and Eio.Stream for communication between domains. Other tools from the Multicore OCaml environment (Fibers, Promise, Async/Await) were also tried, but during the experiments, it turned out that their pure use does not give the effect of parallelism: they effectively cope with asynchronous work, but without an external tool for creating domains/threads, they cannot provide parallelism.

Experiments on CPU intensive tasks

Table 4 shows the data obtained when running two identical reverso relationships in an external parallelization way for lists of different lengths. The main goal is:

```
1    (reverso lst x) or (reverso lst x)
```

and the *lst* is a list of symbols 'a' with a certain length. As a result of calculating this goal, two identical answers equal to *lst* are expected. Here we used a version of reverso relation, which internally calls appendo relation:

```
reverso x y =
  x == Nil and y == Nil or
  fresh h tmp tl.
    x == Cons(h, tl) and
    reverso tl tmp   and
    appendo tmp Cons(h, Nil) y
```

The used version of the reverso relation is quite a CPU-intensive task. In the table, you can see that it takes more than a minute to turn over a list of length 1024. All experiments were run on an 8-core processor with 16 GB of RAM. On a list of length 1300, it was not possible to count the answers in a non-parallel run due to excessive load on the computer at the 196th second. However, the parallel version managed to be calculated in 105 seconds and gave an improvement in time by at least 46%.

Lists length		Parallel, sec	Non-Parallel, sec
512	37.36%	5.41	8.64
900	13.42%	35.82	41.37
1024	31.03%	51.73	75.00
1200	43.46%	80.20	141.84
1300	$>$ 46.40%	105.48	$>$ 196

Table 1: Results of running OR of two 'reverso list x' subgoals

The next table 5 shows key results of running in 2 domains with similar appendo sub-goals (each appends one list to another, lists have the same length shown in the first column of the table) in an external parallelization way. If there is no noticeable improvement in time on the lists with 4096 elements, since it is about one second, and it can be interpreted as a deviation from launch to launch, then on lists with a length of more than 8000, an improvement of about 30% is a good result.

Lists length		Parallel, sec	Non-Parallel, sec
4096	32.04%	3.01	4.43
8192	33.55%	13.64	20.52
16384	35.30%	72.74	112.43

Table 2: Key results of running OR of two 'appendo list1 list2 x' sub-goals

It is important to note that the calculations in the given experiments were balanced across domains since the branches of the OR-tree are identical, and, most likely, these may be the most effective results.

Other interesting results were obtained by launching 10 appendo relations with external parallelization, shown in Table 3. There is no improvement in time up to 7000-length lists. Intuitively, it seems that parallelizing 10 tasks at once should be much more efficient than parallelizing 2 tasks, but the results show the opposite. This may be related to how the stream of states merging works – the tools in the miniKanren language interpreter allow us to merge streams in pairs. In this experiment, the streams merged in pairs, forming a binary tree, while in the sequential version, this merge was implemented by sequential folding of the list of subgoals state streams.

Lists length		Parallel, sec	Non-Parallel, sec
6000	0.11%	58.20	58.26
7000	14.08%	77.55	90.25
8000	25.26%	102.23	136.77

Table 3: Key results of running 10 'appendo list1 list2 x'

As for the experiments in the most poorly balanced case (one of the two branches in the binary OR-tree ends with a simple calculation like the unification of the empty list), then such parallelism in Unicanren slows down the calculation by a few seconds.

3.2.2 Internal parallelization

Baseline

First of all, to make the disjunction operator parallel directly in the interpreter, it is necessary to ensure the acceleration of calculations. Before that, in external parallelization, we did not care about forcing calculations, because the external launch of individual goals already forces

these calculations by itself. When we start working with the internal disjunction operator, we do not have the tools to force the calculations, because it is provided by the external launch of the search for solutions. After all, the number of expected answers will be known at this stage.

When forcing calculations, the optimal way will be to collect answers in some data structure, in which you can put answers from different streams, and then get and merge the answers. This is convenient because in the future when we improve forcing, we will be able to maintain laziness and not wait for threads to complete all calculations. The structure from the Eio library that we will use for this purpose is called Eio.Stream, and in order not to confuse it with the state stream, we will call this structure a queue.

In Unicanren, there are only two types of stream state: **Nil**, i.e. empty stream, and **Cons (x, c)**, where **x** is an answer and **c** is a lazy calculation of the rest of the answers. For the forcing in the case of an empty stream, nothing needs to be done. If there is an answer **x** in the stream, then we put it in the queue, and then recursively force a lazy calculation **c**. If there are no responses in the queue, then we will return an empty stream of Nil states. Otherwise, to merge, you need to get one received stream state from the queue and attach a recursive call to the merge function using monadic plus.

In every spawned domain we need to force streams. After finishing calculations in all the domains, the procedure of merging streams starts.

Switching version

In the course of setting up experiments on an extended class of problems, it became clear that Unicanren does not have the declared property of laziness. Since our research focuses on the parallelism of the answer search procedure in the minikanren family of languages, we decided to switch to another version of minikanren, which is also written in OCaml. In the case of fixing the mechanism of lazy calculations in Unicanren, it will be possible to apply the algorithm developed in our study for another version of miniKanren. However, parallelism heuristics may differ, because the representation of logical operators, the use of different ways of storing substitutions, and the mechanism of laziness in these languages differ. Nevertheless, in Unicanren there is a simple way to look into the conjunct and determine the type of operation, which could be an interesting heuristic for parallelism. We hope that one day it will be possible to investigate.

3.3 Baseline implementation

Further in the study, we continue to explore OR-parallelism within the framework of the minikanren-ocaml language. To parallelize this version of the minicanren, we re-use the baseline described in the chapter about Unicanren.

As shown in the Listing. **??**, we keep the structure from the previous baseline: forcing computations of subgoals in domains, putting calculated answers into a queue, and merging all the answers.

In minikanren-ocaml, there are four types of stream states:

1. **MZero** – no answers in the stream. There is no need to force it.
2. **Unit x** – found the only answer **x**. There is no need to force it, but the answer must be added to the queue.
3. **Func f** – lazy computation **f**, which is waiting **()** argument to run. To force it, we recursively call the forcing function on the **f()** argument.
4. **Choice (x, f)** – answer **x** and lazy computation **f** with the rest of the answers. To force it, we put the answer to the queue, and recursively force

After the tasks are assembled and run, all fibers are waiting for calculations. Next, the procedure for merging takes place. We have reused the standard mplus from the minikanren-ocaml interpreter with deferred computation. During the launch of the parallel disjunction, exactly as many answers as were requested will be given. Since all threads have completed calculations by this point, there is no need to force the merging of streams.

```
1  let condePar lst s =
2        let queue = Eio.Stream.create max_int in
3        let rec force_streams x =
4          match x with
5          | Choice (x, f) -> Eio.Stream.add queue x;
6            force_streams (f ());
7          | Unit x -> Eio.Stream.add queue x;
8          | Func f -> force_streams (f())
9          | MZero -> ()
10       in
11       let make_par_task f ~domain_mgr =
12         Eio.Domain_manager.run domain_mgr (fun () ->
13           force_streams (f s))
```

```
14        in
15        let make_task_list l =
16            Eio_main.run @@ fun env ->
17            let rec iter_tasks l =
18              match l with
19              | hd :: tl -> Eio.Fiber.both
20                (fun () ->
21                  make_par_task ~domain_mgr:(Eio.Stdenv.domain_mgr env) hd)
22                (fun () -> iter_tasks tl)
23              | [] -> ()
24            in iter_tasks l
25        in
26        make_task_list (List.map all lst);
27
28        let rec merge_streams queue =
29          match Eio.Stream.take_nonblocking queue with
30          | Some x -> mplus (Unit x) (fun () -> merge_streams queue)
31          | None -> MZero
32        in
33        merge_streams queue
```

Listing 1: Baseline of minikanren-ocaml parallelization

Domainslib

We have also attempted to make a similar implementation using data structures from the Domainslib library. In comparison with the current baseline based on the Eio library, the implementation showed worse results on simple experiments with a balanced OR tree. Besides, not every answer where computed. The main problem was the launch of a large number of domains, in which case the implementation was completed with an error and the answers were not counted. In the current implementation based on the Eio library, these problems were avoided, which will be shown in the experiments below.

3.4 Experiments

The computing speed in minikanren-ocaml differs from the computing speed of Unicanren, and the baseline had some fixes, so we again experimented with CPU-intensive tasks. In addition, it is important to note that the time error from launch to launch of the same task can be up to 1 second. For this reason, experiments on tasks running for less than 1 second will not be

mentioned in the presented data.

Parallelization of two branches

Excellent results were obtained during the launch of the task "disjunction of two similar reverso tasks". This task is perfectly balanced since both branches are identical. Table 4 shows that the parallelism was almost perfect: there is an improvement in speed of about 50%.

List length		Parallel, sec	Non-Parallel, sec
100	46.67%	8	15
125	45.95%	20	37
150	50.00%	38	76
175	50.00%	70	140
200	50.21%	119	239
215	50.78%	158	321

Table 4: Disjunction of two "reverso ['a',..,'a'] q" subgoals

There was also a significant improvement when running the disjunction of two appendo tasks. Again, this goal is well balanced and managed to get almost perfect parallelism, as shown in Table 5.

Lists length		Parallel, sec	Non-Parallel, sec
950	50.00%	8	16
1350	51.06%	23	47
1650	51.16%	42	86
1950	50.36%	69	139
2350	49.59%	123	244

Table 5: Disjunction of two "appendo ['a',..,'a'] ['b',..,'b'] q" subgoals

Parallelization of 10 branches

More interesting results were obtained during the launch of the disjunction with ten branches. Tables 6 and 7 show that running a large number of branches in parallel gives an improvement of up to 80%. These results could not be obtained during experiments on Unicanren, where the results deteriorated relative to experiments with only two branches. These significant im-

provements provide grounds for investigating the heuristics of parallelization of a large number of branches at the same level at once.

Lists length		Parallel, sec	Non-Parallel, sec
50	80.00%	1	5
70	73.68%	5	19
100	72.73%	21	77
125	73.63%	48	182
150	75.33%	93	377

Table 6: Disjunction of 10 "reverso ['a',..,'a'] q" subgoals

Lists length		Parallel, sec	Non-Parallel, sec
550	75.00%	4	16
1000	75.79%	23	95
1300	77.51%	47	209
1500	78.57%	69	322

Table 7: Disjunction of 10 "appendo ['a',..,'a'] ['b',..,'b'] q" subgoals

Losses on small tasks

We set up experiments on quickly solved tasks (up to 16 seconds), and no significant loss in performance of the parallel version in comparison with the sequential one was revealed. It also applies to experiments on highly unbalanced OR-trees. Such results are pleasing because creating threads can be a cheap operation in this implementation and they do not spoil the performance of the interpreter. With this property, it will be much easier to pick up heuristics, since they will not have a strong requirement to identify and run only CPU-intensive tasks.

Infeasible Simultaneous Execution

A potential problem in such implementation may arise when generating too many domains, more than can be executed simultaneously. When we used the Domainslib library in an attempt to add parallelism to minikanren-ocaml, we encountered a similar problem. To test the interpreter behavior in this case, we set up experiments with a large amount of CPU-intensive tasks. The tables show the results of performing 50 (Table 8) and 100 (Table 9) reverso tasks on lists of lengths 50 and 100. On such tasks we have got not only a version that can

withstand a large number of intensive tasks, but a significant speed improvement – in these experiments, an average acceleration is 75%. Unfortunately, we also found a test, on which domains spawning failed, and the program ended with an error. It can be corrected by adding error handling (if it is impossible to start a new domain, just run it sequentially). Another approach, perhaps partially solving the problem, is to add heuristics that would balance the start of threads so that the system does not overload and there is still an improvement in time.

Lists length		Parallel, sec	Non-Parallel, sec
50	74.07%	7	27
100	76.62%	90	385

Table 8: Disjunction of 50 "reverso ['a',..,'a'] q" subgoals

Lists length		Parallel, sec	Non-Parallel, sec
50	73.08%	14	52
100	77.40%	179	792

Table 9: Disjunction of 100 "reverso ['a',..,'a'] q" subgoals

3.5 Extended implementation

Laziness

Laziness support is a significant aspect of languages from the miniKanren family. To maintain laziness, first of all, it is necessary, to see directly in the disjunction how many answers you need to get or when to finish all the continuations. Secondly, we need to implement a mechanism for interrupting further calculations when the required number of responses has already been collected. Additionally, with the nesting of disjunctions, there must be a coordinated work of all disjunctions at once, so that it is clear when each of them ends.

In the first case, this can be done by stretching the number of requested responses from the external triggering function (*taken* or *run*()) directly into the disjunction. To do this, the disjunction must be an argument of a special type, either stretch this value through all logical operations or add this value to the storage, since it is passed through all functions. The procedure of aborting domain calculations mostly depends on the tools used for parallelism. For the nested disjunctions, creating one common queue in the top disjunction operation could be done. As soon as the queue is filled with the required number of answers, domains will be

aborted. This should still give an acceleration in work, perhaps not as strong as full forcing on certain classes of tasks. At the same time, this will give a significant acceleration on tasks where a small subset of answers is needed, since you will not need to wait for the rest of the calculations. This property is a priority and therefore the focus of further research will be directed to achieving laziness.

In addition, the forcing mechanism could be modified: domains will calculate goals only before the first response, and deferred calculation will start later when another response is needed.

3.5.1 Heurisitics

Heuristics can be a great tool to improve performance and reduce the load on the domain manager. There are several ideas on which conditions we decide the OR-goal branch of the tree to be calculated into a separate domain: based on the depth in the tree, based on the type of clause (unification, conjunction, fresh, another disjunction), the order in the clause list of the current OR-goal, random parallelization.

Random parallelization After encountering the domain manager overload problem, attempts were made to implement selective parallelism. For example, to run parallelism only in one of several branches. This was ensured either through strict counting of the k-th branch or through a random coin toss. Experiments were launched on a disjunction of 100 reverso with a length of 100 with parallel branches. It was possible to achieve correct operation and an average improvement of 50% compared to the fully sequential version.

In comparison with full parallelization in ordinary tasks, it copes slower, but in absolute values, it is still a significant result. In the case of setting the implicit version of parallelism in the interpreter, then some significant deterioration in time should not occur. Usually, on medium-balanced tasks and well-balanced tasks, the improvement will be quite large. On poorly balanced tasks, which at the same time do not overload the domain manager, the deterioration will be observed within the constant value, which may be within the margin of error. On some of the tasks that heavily overload the domain manager, there will be a significant improvement in time.

However, in the case of explicit parallelism in the interpreter, it will be more convenient for the user to independently arrange parallel disjunctions in the code so as not to overload the domain manager, and at the same time, heuristics will not be needed. The user sees that

some branches of disjunction are simple unifications, so it makes no sense to waste time and resources on running them in separate threads. Then everything that is specified as parallel will run in separate domains, not just selective branches. Full parallelization without heuristics will make it possible to get the greatest acceleration of the program. This is not even a question of balancing tasks, but rather a question of overloading the domain manager. If a task is unbalanced and does not spawn too many domains, we will not lose performance, as indicated above.

4 Evaluation of the Investigation

The baseline implementation of minikanren-ocaml parallelization using the Eio library shows promising results. It effectively forces computations, puts answers into a queue, and merges them. It demonstrates good performance on CPU-intensive tasks and shows speed improvements compared to the non-parallel version.

Attempts to use the Domainslib library for the implementation encountered issues and showed worse results compared to the Eio library. The Eio library-based implementation avoids problems and provides better performance.

Experiments were focused on testing the performance of parallelization on different tasks. The first set of experiments involved the parallelization of two branches in a disjunction. Two types of tasks were considered: "reverso" and "appendo". In both cases, the tasks were well-balanced, with identical branches. The results showed almost perfect parallelism, with an improvement in speed of about 50%. For example, in the "reverso" task with a list length of 200, the parallel version took 119 seconds compared to 239 seconds for the non-parallel version.

The next set of experiments involved the parallelization of 10 branches in a disjunction. Again, the tasks were "reverso" and "appendo". The results demonstrated significant performance improvements, with speedup ranging from 73.68% to 80%. For instance, in the "appendo" task with a list length of 1500, the parallel version took 69 seconds compared to 322 seconds for the non-parallel version.

The experiments also investigated the performance of small tasks and highly unbalanced OR-trees. The results indicated that the parallel version did not suffer significant losses in performance compared to the sequential version in these scenarios. This is advantageous as it allows the interpreter to create threads without significantly impacting performance, making it easier to implement heuristics without strong requirements for identifying CPU-intensive tasks.

However, it should be noted that the experiments revealed a potential issue when generating a very large number of domains that exceeds the system's capacity for simultaneous execution. In some cases, the interpreter encountered errors and failed to count all the answers. This problem could be addressed by adding error handling and implementing heuristics to balance the starting of threads to prevent system overload. On the test data, we managed to avoid such an error using the heuristic of randomly selecting branches for parallelization.

The experiments also touched on the topic of laziness support in the minikanren-ocaml parallelization. Maintaining laziness is important for languages in the miniKanren family, and further research will focus on achieving laziness by incorporating interruption mechanisms and coordinated work of nested disjunctions.

In addition to the experiments conducted on the minikanren-ocaml interpreter, similar efforts were made to implement parallelization in the Unicanren interpreter. The goal of parallelizing Unicanren was to explore the potential performance gains through the parallel execution of disjunctive goals. Experiments on the Unicanren parallelization revealed some interesting findings. Initially, the experiments focused on parallelizing two branches in a disjunction, similar to the experiments conducted on the minikanren-ocaml interpreter. However, unlike the minikanren-ocaml implementation, the performance of the Unicanren parallelization deteriorated as the number of branches increased. This suggests that the parallelization strategy used in Unicanren may not be as effective in achieving performance improvements compared to the minikanren-ocaml implementation. The experiments also revealed problems related to resource allocation and load balancing when parallelizing Unicanren. In certain scenarios, the uneven distribution of workload between domains and the overhead associated with the creation of domains and their coordination reduced overall performance. These issues point to the need for further research and optimization to improve the parallel execution strategy in Unicanren.

In conclusion, the experiments demonstrated the effectiveness of parallelization in the minikanren-ocaml interpreter using the Eio library. Significant performance improvements were observed, especially when parallelizing a large number of branches. The results provided a basis for further research into heuristics and laziness support, intending to optimize parallel execution in the context of miniKanren languages.

5 Conclusions

The research project focused on exploring and evaluating the parallelization of the minikanren-ocaml language. The study revealed that parallelism is a powerful tool for speeding up the search process in minikanren. The capabilities of the Multicore OCaml language enabled significant acceleration in the execution of tasks.

The main aspects and results of the project can be summarized as follows:

- Baseline implementation: The baseline implementation of the disjunction parallelization approach in minikanren-ocaml was described. It involved forcing computations of subgoals in domains, storing calculated answers in a queue, and merging the answers.
- Performance evaluation: Experiments were conducted to evaluate the performance of the parallelization approach. The results demonstrated significant speed improvements in CPU-intensive tasks. The parallel version achieved speed improvements of around 50% for disjunctions of two similar tasks and up to 80% for disjunctions with a larger number of branches.
- Scalability: The parallelization approach showed scalability, with performance improvements maintained even when increasing the number of branches in the disjunctions. This scalability was a notable advantage compared to previous experiments with Unicanren.

By leveraging parallelism and the multicore capabilities of OCaml, the research project demonstrated the potential for substantial performance improvements in minikanren-ocaml. These findings open up avenues for future research and optimization of parallelization techniques in the context of logic programming languages.

References

[1] Claire E. Alvis, Jeremiah J. Willcock, and Kyle M. Carter. "cKanren: miniKanren with Constraints". In: (2011).

[2] William E. Byrd et al. "A Shallow Scheme Embedding of Bottom-Avoiding Streams". In: (2011).

[3] William E. Byrd et al. "A Unified Approach to Solving Seven Programming Problems". In: (2017).

[4] Stephen Dolan et al. "Concurrent System Programming with Effect Handlers". In: (Feb. 2018).

[5] Agostino Dovier et al. "Parallel Logic Programming: A Sequel". In: (Sept. 2005).

[6] Byrd William E., Holk Eric, and Friedman Daniel P. "miniKanren, Live and Untagged: Quine Generation via Relational Interpreters". In: (2012).

[7] Byrd William E. and Friedman Daniel P. "kanren: A Fresh Name in Nominal Logic Programming". In: (2007).

[8] Gopal Gupta and Enrico Pontelli. "Extended dynamic dependent and-parallelism in ACE." In: Jan. 1997, pp. 68–79.

[9] B. Hausman, A. Ciepielewski, and S. 1987. Haridi. "Or-parallel Prolog made efficient on shared memory multiprocessors". In: (1987).

[10] Manuel V. Hermenegildo and Francesca Rossi. "Strict and nonstrict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions". In: *The Journal of Logic Programming* 22.1 (1995), pp. 1–45. ISSN: 0743-1066. DOI: https://doi.org/10.1016/0743-1066(93)00007-F. URL: https://www.sciencedirect.com/science/article/pii/074310669300007F.

[11] Sadiq Jaffer et al. "Parallelising your OCaml code with Multicore OCaml". In: (Aug. 2020).

[12] Hemann Jason and Friedman Daniel P. "µKanren: A Minimal Functional Core for Relational Programming". In: (2013).

[13] Oleg Kiselyov et al. "Backtracking, interleaving, and terminating monad transformers: (functional pearl)". In: (Sept. 2005).

[14] Ricardo Lopes, Vitor Costa, and Fernando Silva. "A Design and Implementation of the Extended Andorra Model". In: *Computing Research Repository - CORR* 12 (Jan. 2011). DOI: 10.1017/S1471068411000068.

[15] Gebser M. et al. "Evaluation techniques and systems for answer set programming: a survey". In: (2018).

[16] Anil Madhavapeddy. "OCaml Multicore Wiki: Concurrency and parallelism design notes". In: (Aug. 2020).

[17] Near Joseph P., Byrd William E., and Friedman Daniel P. "alpha-leanTAP: A Declarative Theorem Prover for First-Order Classical Logic". In: (2008).

[18] Kish Shen. "Overview of DASWAM: Exploitation of dependent and-parallelism". In: *The Journal of Logic Programming* 29.1 (1996). High-Performance Implementations of Logic Programming Systems, pp. 245–293. ISSN: 0743-1066. DOI: https://doi.org/10.1016/S0743-1066(96)00079-9. URL: https://www.sciencedirect.com/science/article/pii/S0743106696000799.

[19] KC Sivaramakrishnan et al. "Retrofitting Effect Handlers onto OCaml". In: (June 2021).

[20] KC Sivaramakrishnan et al. "Retrofitting Parallelism onto OCaml". In: (Aug. 2020).