Cairo University
Faculty of Engineering
Department of Computer Engineering

RunSurge

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by
Omar Said Mohammed

Supervised by
Dr/Lydia Wahid

July 2025

Abstract

RunSurge addresses the imbalance between users with idle computational resources and those requiring processing power by introducing the Surge Network, a distributed system that bridges this gap. The project connects contributor users offering excess resources—with users needing to run their code, enabling efficient, decentralized computation. The system is composed of three main modules: the Master Module, which acts as the central orchestrator; the Worker Module, operated by resource-contributing users; and the Parallelization Module, which takes user-submitted code in a predefined format and transforms it into a parallelizable form, with synchronization support, when possible, for distributed execution. The Parallelization Module comprises three important subcomponents: the Data Dependency Graph (DDG) for analyzing task dependencies, the Memory Estimator for resource requirement prediction, and the Parallelizer for decomposing and synchronizing tasks. These components were fully implemented in Python and validated through a set of sample test cases conforming to the required code format. The project demonstrates a functional, peer-assisted computing framework designed to facilitate scalable, distributed code execution using only internal scripts and tools.

الملخص

يعالج مشروع RunSurge مشكلة التفاوت بين المستخدمين الذين يمتلكون موارد حوسبة غير مستغلة، وأولئك الذين يحتاجون إلى قوة معالجة، من خلال تقديم شبكة Surge، وهي نظام موزّع يهدف إلى سد هذه الفجوة. يربط المشروع بين المستخدمين المساهمين الذين يوفرون موارد فائضة والمستخدمين الذين يحتاجون إلى تشغيل شيفراتهم، مما يمكّن من تنفيذ عمليات حسابية فعّالة بطريقة موزعة.
يتكوّن النظام من ثلاث وحدات رئيسية: وحدة Master، وهي المنسق المركزي للنظام، ووحدة Worker، التي يتم تشغيلها من قبل المستخدمين المساهمين بالموارد، ووحدة التوازي (Parallelization)، التي تستقبل الشيفرة من المستخدم بصيغة محددة مسبقًا وتحوّلها إلى شكل يمكن تنفيذه بالتوازي مع دعم المزامنة عندما يكون ذلك ممكنًا، ليتم تنفيذها عبر العقد المتاحة. تتكون وحدة التوازي من ثلاث وحدات فرعية أساسية: رسم تبعية البيانات (Data Dependency Graph - DDG) لتحليل الترابط بين المهام، مقدّر الذاكرة (Memory Estimator) لتوقّع المتطلبات من الموارد، ووحدة التوازي (Parallelizer) لتقسيم المهام وتزامنها. تم تنفيذ هذه المكونات بالكامل باستخدام لغة Python، وتم اختبارها من خلال مجموعة من حالات الاختبار المعدة مسبقًا وفقًا للصيغة المطلوبة للشيفرة. يمثل المشروع إطارًا وظيفيًا للحوسبة التعاونية يهدف إلى تمكين تنفيذ الشيفرات بشكل موزع وقابل للتوسع، باستخدام أدوات ونصوص داخلية فقط.

ACKNOWLEDGMENT

We would like to express our sincere appreciation to everyone who contributed to the successful completion of this graduation project.

We extend our heartfelt gratitude to Dr. Lydia for her valuable guidance, continuous encouragement, and thoughtful feedback throughout the development of this work. Her support has been instrumental in shaping the direction and quality of our project, and we are deeply thankful for her dedication and mentorship.

We are equally grateful to the respected faculty members and doctors who have accompanied us throughout our academic journey. Their expertise, commitment to teaching, and willingness to share knowledge have significantly enriched our learning experience. The lectures, practical sessions, and open discussions provided by them have helped cultivate both our technical and critical thinking skills.

We also wish to thank our families and loved ones for their unwavering support, patience, and belief in our capabilities. Their encouragement has been a source of strength during moments of challenge and growth.

In conclusion, this project stands as a reflection of the collective support, knowledge, and mentorship we have received. We are truly thankful to everyone who has contributed to our journey.

Table of Contents

Abstract ii
الملخص iii
Acknowledgment iv
Table of Contents v
List of Figures vii
List of Abbreviations vii
Contacts ix
Chapter 1: Introduction
1.1. Motivation and Justification 1
1.2. Document Organization 1
Chapter 2: Literature Survey
2.1. Comparative Study of Existing Approaches to Python Code Parallelization 2
2.1.1 Pydron: Semi-Automatic Parallelization 2
2.1.2 Dynamic AST-Based Parallelization: PyParallelize 2
2.1.3 Towards Parallelism Detection of Sequential Programs with Graph Neural Network 4
2.2. Implemented Approach 5
Chapter 3: System Design and Architecture
3.1. Overview and Assumptions 7
3.2. System Architecture 8
3.2.1. Block Diagram 8
3.3. DDG 9

3.3.1. Functional Description
.... 9
3.3.2. Modular Decomposition
.... 11
3.3.3. Design Constraints
.... 12
3.4. Memory Estimator
.... 12
3.4.1. Functional Description
.... 12
3.4.2. Modular Decomposition
.... 13
3.4.3. Design Constraints
.... 13

Chapter 4: System Testing and Verification
4.1. Testing Setup
.... 15
4.2 Module Testing
.... 4.3.
Integration Testing
.... 15

Chapter 5: Conclusions and Future Work
5.1. Faced Challenges
.... 17
5.2. Gained Experience
.... 17
5.3. Future Work
.... 17
References
.... 19

List of Figures

Figure 2.1: Pydron Overview 2
Figure 2.2: Pyparallelize Overview 3
Figure 2.3: Parallelism Detection Workflow 5
Figure 3.1: System Block Diagram 8
Figure 3.2: Master Node 8
Figure 3.3: DDG 9
Figure 3.4: Example DDG Input 10
Figure 3.5: Example Graph Output 10
Figure 3.6: Example Nodes Output 11
Figure 3.7: Example Edges Output 11
Figure 3.8: Memory Estimator 12
Figure 3.9: Example Memory Estimator Output 13

List of Abbreviation

AST	Abstract Syntax Tree
DDG	Data Dependency Graph
SSA	Static Single Assignment

Contacts

Team Members

Name	Email	Phone Number
Omar Said Mohammed	Omar.Aziz02@eng-st.cu.edu.eg	+2 01120551134

Supervisor

Name	Email	Number
Lydia Wahid	LydiaWahid@outlook.com	-

This page is left intentionally empty

Chapter 1: Introduction

This part of the report focuses on the Parallelization Module, a key component within the RunSurge distributed system. Specifically, it introduces and explains the three submodules that make up the Parallelization Module: The Data Dependency Graph (DDG), which identifies task dependencies in the input code; the Memory Estimator, which predicts memory usage to support efficient resource allocation; and the Parallelizer, which restructures the code into parallel tasks with synchronization where necessary. These submodules collectively enable the transformation of user-submitted code into a form that can be executed across multiple distributed nodes.

1.1. Motivation and Justification

Transforming sequential code into parallelizable form is a key challenge in distributed systems, requiring careful handling of data dependencies, memory usage, and task decomposition. To address this, the RunSurge system introduces three core submodules:

- Data Dependency Graph (DDG) analyzes variable interactions to capture dependencies between code segments, ensuring correctness during parallel execution.
- Memory Estimator predicts the memory footprint of code blocks to guide efficient and balanced distribution across nodes.
- Parallelizer combines outputs from DDG and Memory Estimator to generate a validated, parallel-compatible version of the input code.
- These components collectively enable scalable and automated parallelization, forming the backbone of RunSurge's distributed execution pipeline.

1.2. Document Organization

This report is organized into five main chapters, followed by several appendices that provide supporting information and documentation.

Chapter 2: reviews related work on distributed execution and code parallelization, outlining background concepts and positioning our approach.

Chapter 3 details the design of the system: including its architecture, key modules, and design constraints.

Chapter 4 explains the testing process: covering both unit and integration testing along with evaluation results.

Chapter 5 summarizes the outcomes: discusses limitations, and suggests future improvements.

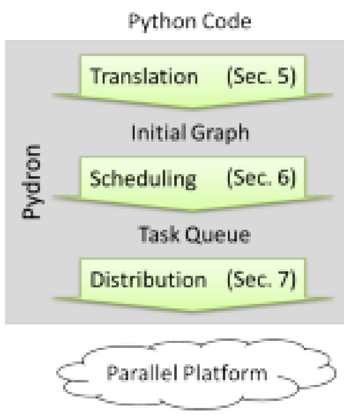
Chapter 2: Literature Survey

This chapter reviews literature relevant to the design of the Parallelization Module, focusing on data dependency analysis and automatic code parallelization. It examines approaches for constructing Data Dependency Graphs (DDGs), identifying code dependencies, and enabling safe parallel execution. Existing frameworks and their trade-offs are also discussed to justify the design choices made in the RunSurge system.

2.1. Comparative Study of Existing Approaches to Python Code Parallelization

2.1.1: Pydron: Semi-Automatic Parallelization:

Pydron (2014) introduced a semi-automatic approach to Python parallelization by translating decorated functions into a runtime-generated data-flow graph. While effective, it relies on function-level runtime analysis and lacks detailed static dependency resolution.
In contrast, our system performs statement-level static analysis via a Data Dependency Graph (DDG), enabling finer control and early detection of synchronization points. Our Memory Estimator adds heuristic memory profiling, and the Parallelizer ensures code validity before execution, offering more reliable and scalable parallelization than Pydron's runtime-based model



2.1 Pydron Overview

2.1.2: Dynamic AST-Based Parallelization: PyParallelize:

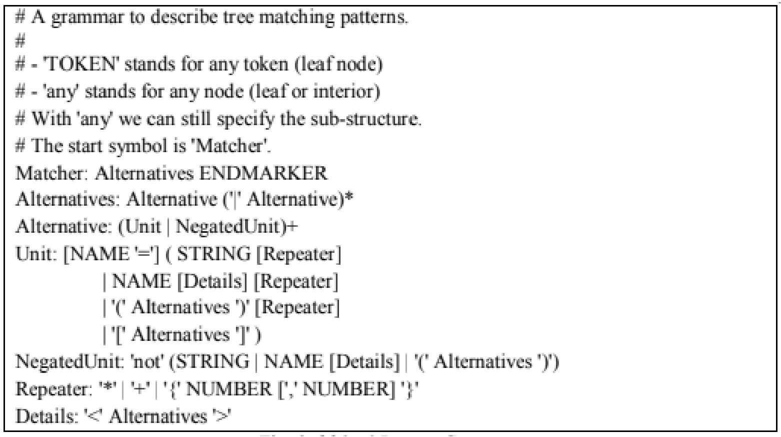
In 2017, PyParallelize was introduced as a dynamic model for automatic Python code parallelization, requiring no manual code changes. It operates by analyzing and transforming the abstract syntax tree (AST) of input code to identify and wrap parallelizable constructs.

Core Approach:

- AST Generation: Converts source code into an AST using tools like lib2to3 and graphviz.
- Pattern Matching: Identifies parallelizable structures (e.g., loops, conditionals) using rule-based EBNF-style grammar.
- Rule Definition: Applies specific grammar constraints to ensure code segments are safe for parallel execution.
- AST Transformation: Rewrites identified code segments into callable functions.
- Parallel Wrapping: Wraps constructs with decorators (e.g., @parallel) and uses batchOpenMPI for execution.
- Synchronization Handling: Skips unsafe regions (e.g., shared memory operations) to preserve correctness.
- SSA Conversion: Transforms variables into Static Single Assignment form for better dependency tracking.
- Control Flow Handling: Converts constructs like return or break into flag-based conditions to maintain flow.

Limitations Compared to Our Approach:

- Minimal Restructuring: Avoids complex synchronization handling rather than resolving it.
- Tooling Issues: Relies on deprecated modules like lib2to3, affecting long-term support.



```
# A grammar to describe tree matching patterns.
#
# - 'TOKEN' stands for any token (leaf node)
# - 'any' stands for any node (leaf or interior)
# With 'any' we can still specify the sub-structure.
# The start symbol is 'Matcher'.
Matcher: Alternatives ENDMARKER
Alternatives: Alternative ('|' Alternative)*
Alternative: (Unit | NegatedUnit)+
Unit: [NAME '='] ( STRING [Repeater]
          | NAME [Details] [Repeater]
          | '(' Alternatives ')' [Repeater]
          | '[' Alternatives ']' )
NegatedUnit: 'not' (STRING | NAME [Details] | '(' Alternatives ')')
Repeater: '*' | '+' | '{' NUMBER [',' NUMBER] '}'
Details: '<' Alternatives '>'
```

2.2 Pyparallelize Overview

2.1.3: Towards Parallelism Detection of Sequential Programs with Graph Neural Network (2021):

A data-driven framework was proposed for automatic parallelism detection in sequential C/C++ programs using a Contextual Flow Graph (XFG) and a Deep Graph Convolutional Neural Network (DGCNN). To generate training data, the authors built a pipeline using the Pluto compiler to annotate loops as parallelizable and then transformed the code into LLVM IR and XFGs.

Code Representation and Dataset Generation:

- C/C++ code is refactored to isolate loops.
- Pluto inserts OpenMP annotations to label parallel loops.
- Code is compiled to LLVM IR and converted to XFGs.
- Invalid or large loops are filtered.

Result: a labeled dataset (SSPD) with 900+ graphs.

Neural Network Architecture:

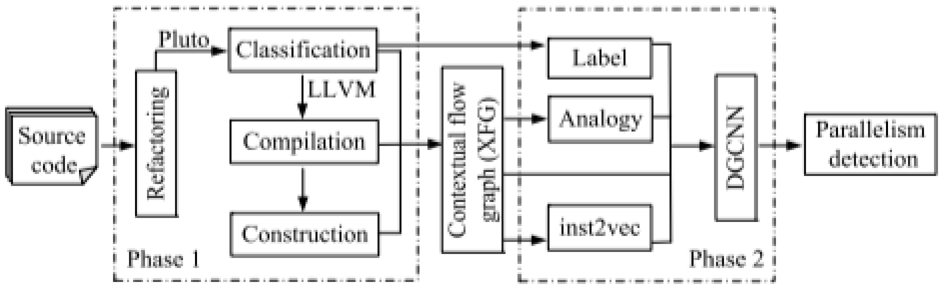
- XFGs are input to a DGCNN consisting of:
    - Graph convolutions
    - SortPooling
    - 1D convolutions and fully connected layers
    - Nodes are embedded using inst2vec and op-type tags.
    - Trained using bi-tempered logistic loss for noise robustness.

Results:

- High detection accuracy on both synthetic and real datasets (e.g., NPB).
- Model outperforms Pluto in several cases, validated by manual checks.
- Demonstrates practical speedups using Pthreads and OpenMP.

Limitations:

- Static-Only Semantics: No runtime behavior is considered (misses data-dependent patterns).
- Semantic Loss: Abstract tokens reduce variable-level insight, causing ambiguities.
- Pluto Dependence: Limited to affine, static loops; skips complex or large ones.
- Single Graph Usage: Lacks deeper semantic capture; multi-graph architectures are suggested.
- Language Limitation: Entire toolchain supports only C/C++; not applicable to Python without major redesign.



2.3 *Parallelism Detection Workflow*

2.2. Implemented Approach

Based on the comparative review of existing approaches for automatic code parallelization, we adopted a **static code analysis framework tailored for Python programs**. The choice was motivated by the need for lightweight, dependency-aware parallelization that avoids the overhead of runtime profiling, while still maintaining a broad applicability to real-world Python codebases. Our framework decomposes the parallelization process into three major modules, each designed to capture key characteristics of Python code prior to scheduling execution on parallel systems.

- **Rule-Based Dependency Analysis via AST-DDG:**
  The first component of the framework is a **rule-based Dependency Directed Graph (DDG)** construction module, which is built upon Python's Abstract Syntax Tree (AST). This module analyzes data dependencies between individual lines of code by statically inspecting variable definitions and usages. Through this analysis, it constructs a directed graph indicating the ordering constraints necessary to preserve program semantics. The design leverages the structured representation of the AST to extract accurate control and data dependencies, which is essential for determining potential parallel regions.
- **Memory Access Estimation Module**
  To complement the dependency analysis, we developed a **static memory access estimator**, also based on the AST representation. This module approximates memory behavior and interaction patterns for each line of code using a pessimistic (conservative) rule-based analysis. It considers variable lifetimes, scope, and mutability to estimate conflicts or side effects that may arise in concurrent execution. By incorporating scope-aware memory tracking, the estimator increases the reliability of subsequent parallelization decisions.
- **Parallelizer**
  The third and final component is the **Parallelizer Module**, which **integrates the outputs** of the DDG and memory estimator to produce a consolidated representation of the program's parallelization potential. It does **not perform independent decision-making**, but rather coordinates the information obtained from dependency and memory analysis to generate **annotated intermediate representations** of code segments. This module is also responsible for **resolving overlaps, aligning analysis results, and handling additional language-specific cases**.

Justification and Design Considerations:
The primary motivation behind choosing a static, rule-based approach lies in its **efficiency, interpretability, and extensibility**. Unlike dynamic methods requiring runtime instrumentation, our solution provides immediate analysis results, making it suitable for integration into development tools or CI pipelines. Moreover, by operating directly on AST representations, it avoids the complexity of intermediate representations such as bytecode or LLVM IR, which are often language-agnostic and less expressive for Python-specific constructs.
The modular design further supports extensibility. While the current system relies on a handcrafted rule set, these rules can be systematically expanded to support additional language constructs or domain-specific patterns. Furthermore, since the framework is grounded in Python's native syntax tree, it remains aligned with evolving language features.

Limitations:
The principal limitation of the implemented framework is its **rule-based nature**, which inherently constrains the scope of syntax and semantic structures it can handle. Complex constructs involving dynamic typing, reflection, or non-deterministic control flow may fall outside the capabilities of the current rule set. Additionally, while the static memory estimator is conservative to ensure correctness, this may lead to overly cautious results, preventing parallelization in cases where it is actually safe. Nonetheless, these limitations are not fundamental. The framework is designed to be extensible and could benefit from future integration with **machine learning-based prediction modules or symbolic execution engines** to improve coverage and accuracy.

Chapter 3: System Design and Architecture

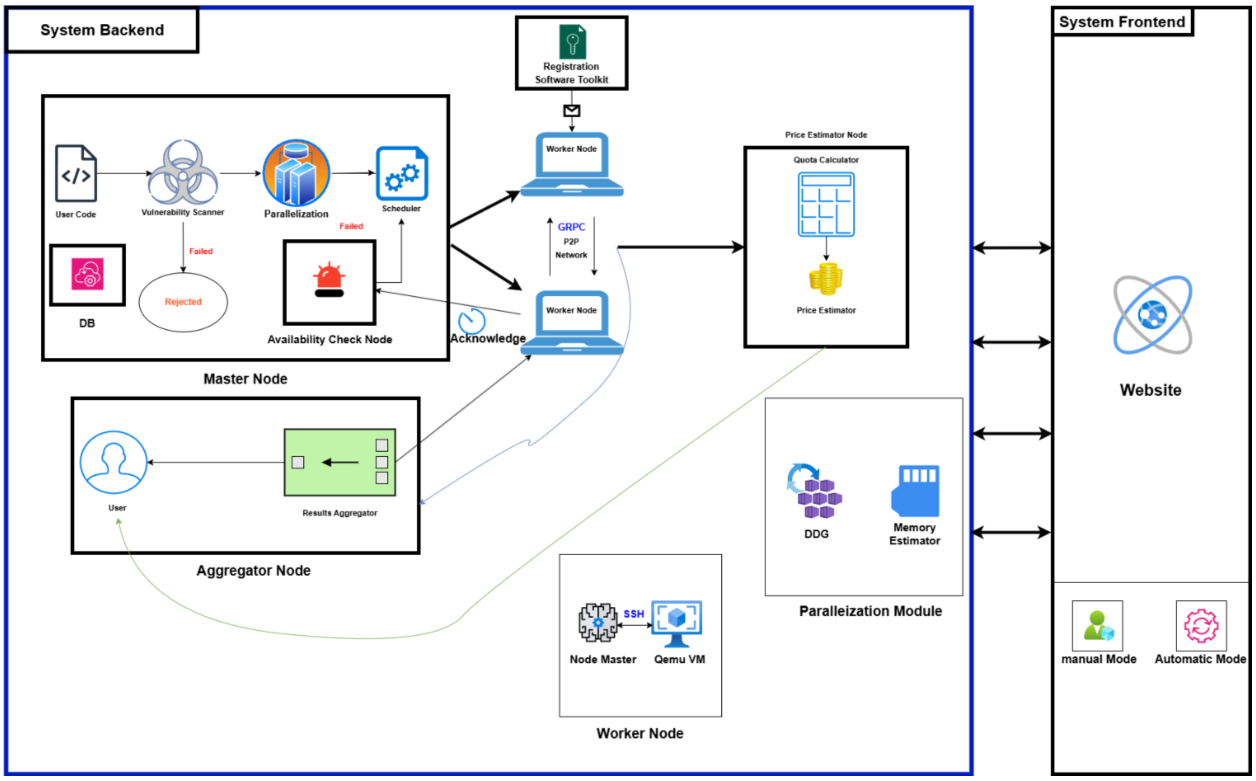
This chapter outlines the design and integration of the three foundational submodules that collectively support the proposed approach for static parallelization of Python programs. Each submodule is designed with a specific analytical responsibility, contributing unique insights toward understanding code structure and behavior.

3.1. Overview and Assumptions

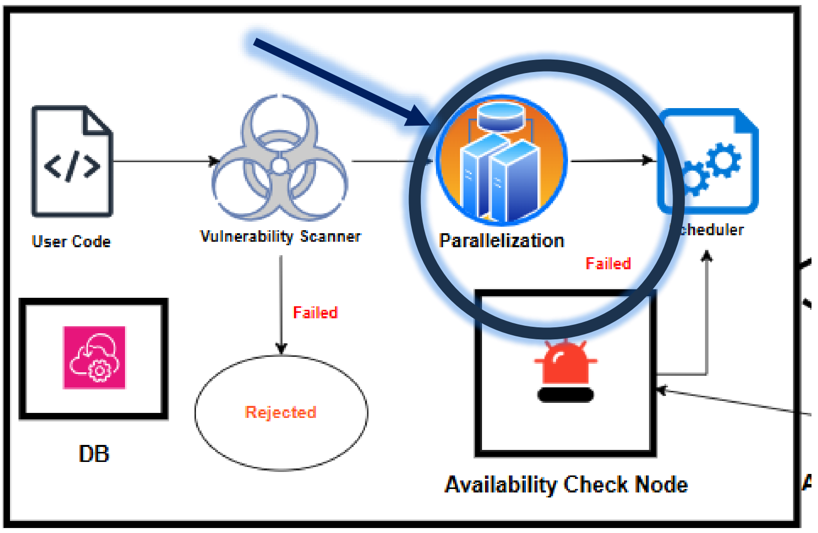
1. **Python AST Sufficiency**: It is assumed that the Python AST provides sufficient semantic and syntactic information to perform static dependency and memory analysis, even in the absence of dynamic execution data.
2. **Rule-Based Dependency Modeling**: The system assumes that most common data dependencies (read-after-write, write-after-read, write-after-write) can be detected using rule-based heuristics applied to individual AST nodes and their surrounding context.
3. **Static Memory Behavior**: It is assumed that a pessimistic approximation of memory access behavior can be inferred statically by analyzing variable scopes, lifetimes, and mutability patterns.
4. **Sequential Purity of Input**: Input programs are assumed to be sequential in nature, with no pre-existing parallel constructs (e.g., multithreading, multiprocessing) embedded.
5. **No Runtime Reflection or Dynamic Code Execution**: Constructs like eval(), exec(), dynamic attribute access (getattr, setattr), or dynamically imported modules are assumed to be absent or ignored, due to their incompatibility with static analysis.
6. **Error-Free Input**: The system assumes that the provided Python source code is syntactically and semantically valid (i.e., does not contain runtime errors or unparseable fragments).

3.2. System Architecture

3.2.1. Block Diagram

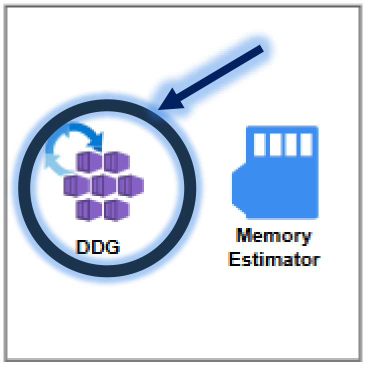


3.1 System Block Diagram



3.2 Master Node

3.3. DDG



3.3 DDG

3.3.1. Functional Description

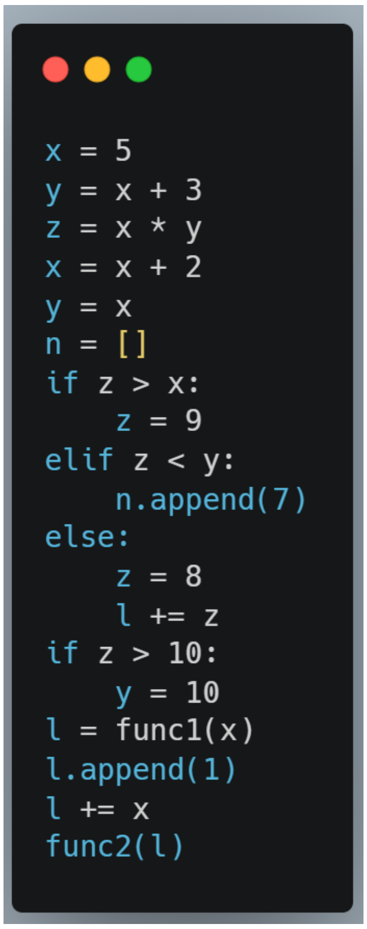
The DDG module statically analyzes Python code to detect data dependencies between lines and represent them as a directed graph. Each node corresponds to a code line, and edges reflect dependency constraints that impact parallelization.

Using Python's AST, the module identifies:

- "has": Variables defined or modified.
- "needs": Variables read or referenced.

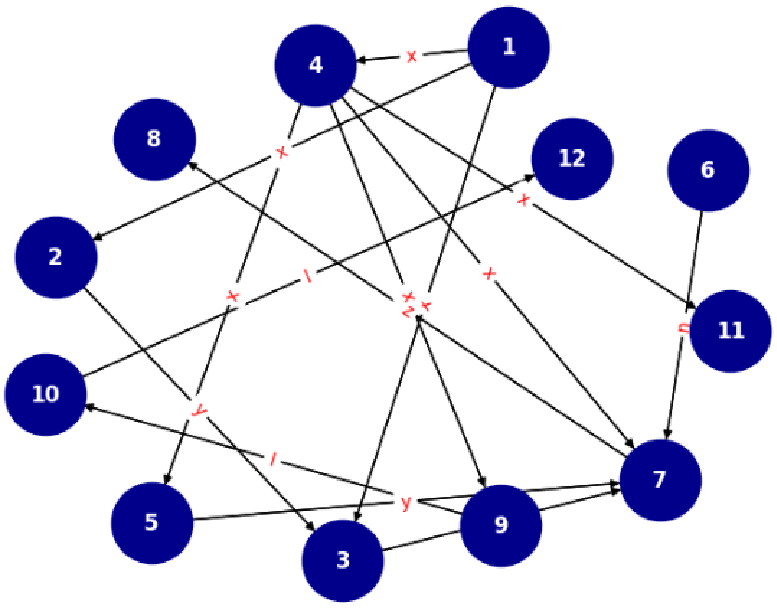
An edge from line i to line j is added if j needs a variable that i has, capturing Read-After-Write (RAW) dependencies. Other types (WAR, WAW) are included where relevant. Control structures (e.g., loops, conditionals, function calls) are analyzed conservatively to maintain correctness.

The output is a directed acyclic graph (DAG) that encodes dependency structure, exported in a format usable by downstream components.

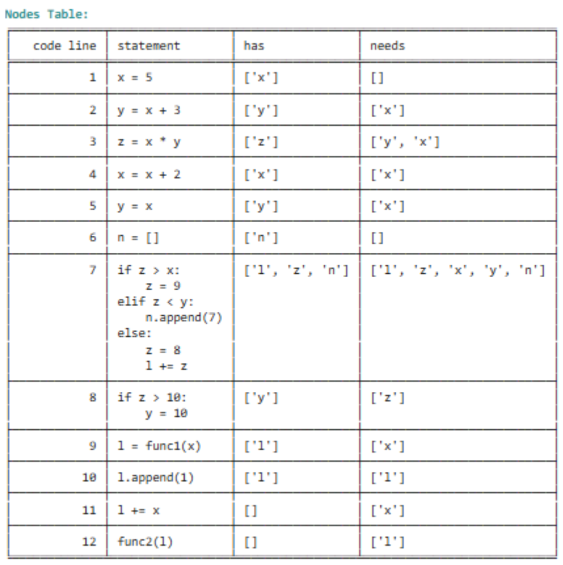


```
x = 5
y = x + 3
z = x * y
x = x + 2
y = x
n = []
if z > x:
    z = 9
elif z < y:
    n.append(7)
else:
    z = 8
    l += z
if z > 10:
    y = 10
l = func1(x)
l.append(1)
l += x
func2(l)
```

3.4 *Example DDG Input*

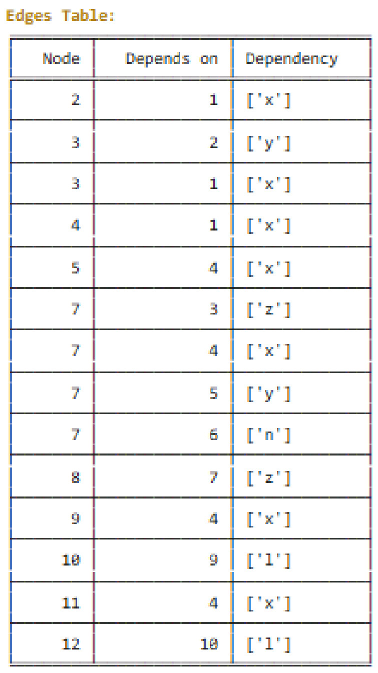


3.5 *Example Graph Output*



```
Nodes Table:
| code line | statement      | has             | needs                          |
|         1 | x = 5          | ['x']           | []                             |
|         2 | y = x + 3      | ['y']           | ['x']                          |
|         3 | z = x * y      | ['z']           | ['y', 'x']                     |
|         4 | x = x + 2      | ['x']           | ['x']                          |
|         5 | y = x          | ['y']           | ['x']                          |
|         6 | n = []         | ['n']           | []                             |
|         7 | if z > x:      | ['l', 'z', 'n'] | ['l', 'z', 'x', 'y', 'n']      |
|           |     z = 9      |                 |                                |
|           | elif z < y:    |                 |                                |
|           |     n.append(7)|                 |                                |
|           | else:          |                 |                                |
|           |     z = 8      |                 |                                |
|           |     l += z     |                 |                                |
|         8 | if z > 10:     | ['y']           | ['z']                          |
|           |     y = 10     |                 |                                |
|         9 | l = func1(x)   | ['l']           | ['x']                          |
|        10 | l.append(1)    | ['l']           | ['l']                          |
|        11 | l += x         | []              | ['x']                          |
|        12 | func2(l)       | []              | ['l']                          |
```

3.6 *Example Nodes Output*



```
Edges Table:
| Node | Depends on | Dependency |
|    2 |          1 | ['x']      |
|    3 |          2 | ['y']      |
|    3 |          1 | ['x']      |
|    4 |          1 | ['x']      |
|    5 |          4 | ['x']      |
|    7 |          3 | ['z']      |
|    7 |          4 | ['x']      |
|    7 |          5 | ['y']      |
|    7 |          6 | ['n']      |
|    8 |          7 | ['z']      |
|    9 |          4 | ['x']      |
|   10 |          9 | ['l']      |
|   11 |          4 | ['x']      |
|   12 |         10 | ['l']      |
```

3.7 *Example Edges Output*

3.3.2. Modular Decomposition

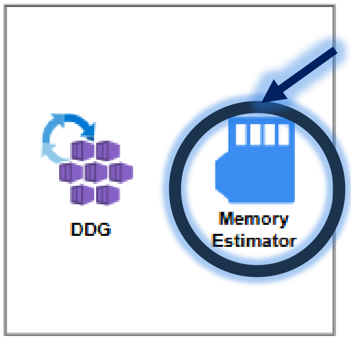
1. **Entry Point Parser:** Identifies and parses the top-level code outside function definitions.
2. **Function Parser:** Extracts and analyzes individual function bodies and their scopes.
3. **Dependency Extractor:** Determines variable definitions and usages per statement.

4. **DDG Builder:** Constructs the data dependency graph using extracted information.

3.3.3. Design Constraints

1. **Lack of Explicit Declarations:** Python code does not require variable declarations, and scopes are often inferred from context. Additionally, the use of implicit globals and built-in functions complicates dependency tracking.
2. **Complex Control Structures:** Control flow constructs like if, for, while, try-except, and comprehension expressions can introduce non-linear paths and conditional dependencies.
3. **Mutable Shared Data Structures:** Many parallelization issues arise not just from scalar dependencies, but from shared mutable containers like lists or dictionaries.
4. **Modularity and Extensibility:** The goal is to support incremental rule definition and easy expansion for new Python constructs or patterns in future work.
5. **Analysis Granularity:** To enable fine-grained parallelism decisions, dependency analysis must operate at the level of individual statements, and ideally at the expression level.
6. **Static-Only Assumption:** The entire system is based on static code analysis, meaning no runtime profiling, type resolution, or memory access tracing is available.

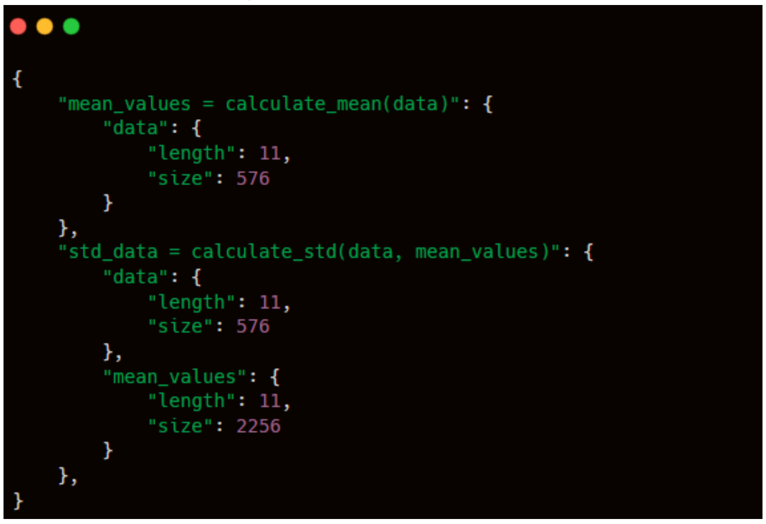
3.4. Memory Estimator



3.8 Memory Estimator

3.4.1. Functional Description

The Memory Estimator module is responsible for statically analyzing how each line of Python code interacts with memory—specifically, how it reads from or writes to program variables and data structures. Its primary goal is to produce a pessimistic memory trace in bytes for each line of code, capturing the set of memory-related side effects that must be considered when determining safe regions for parallel execution. This module plays a crucial role in enabling conservative yet effective parallelization decisions by assessing the potential memory overlap and interference between code segments. Its analysis is based entirely on Python's Abstract Syntax Tree (AST), Python runtime memory profilers and rule-based inference mechanisms, allowing for deterministic and extensible analysis.



```
{
    "mean_values = calculate_mean(data)": {
        "data": {
            "length": 11,
            "size": 576
        }
    },
    "std_data = calculate_std(data, mean_values)": {
        "data": {
            "length": 11,
            "size": 576
        },
        "mean_values": {
            "length": 11,
            "size": 2256
        }
    },
}
```

3.9 *Example Memory Estimator Output*

3.4.2. Modular Decomposition

1. **Primitives Estimator:** Analyzes assignments involving primitive types and estimates memory usage.
2. **List Estimator:** Handles list creation, growth patterns, and associated memory behavior.
3. **File Handler:** Detects and estimates memory implications of file I/O operations.
4. **Assignment Handler:** Processes assignment statements to track variable state changes.
5. **Insertion Handler:** Identifies insert operations (e.g., append, insert) and their impact on memory.
6. **Deletion Handler:** Tracks memory reclaimed or released by del statements or equivalent operations.
7. **Loop Handler:** Estimates repeated memory usage patterns in loops based on iteration bounds.
8. **Condition Handler:** Evaluates memory branching implications in if, elif, and else structures.

3.4.3. Design Constraints

1. **Static Analysis Constraints:** Operates on AST only; lacks runtime type and behavior information, cannot resolve dynamic features (e.g., eval, getattr, function pointers), Fails to trace aliases or side effects from function calls.
2. **Ambiguous Data Semantics:** Cannot reliably detect shared references or aliasing between variables.

3. **Pessimistic Estimation Strategy:** Over-approximates memory impact to ensure safety, limiting parallelization in uncertain but possibly safe cases.

4. **Rule-Based Architecture:** Requires manual updates for new structures (e.g., set, deque, custom types), Limited generalization; rules don't capture semantic meaning beyond syntax.
5. **Granularity of Estimation:** The estimator analyzes code at the line level, which introduces granularity constraints:
   - Compound Statements: Lines containing multiple operations (e.g., x = a + b; y = x + 1) require additional splitting or loss of detail.
   - Control Flow Insensitivity: Branching and loops are handled syntactically without full control-flow resolution, which may lead to inaccurate memory estimations for conditionally executed code.

Chapter 4: System Testing and Verification

This chapter outlines the process undertaken to validate the correctness and functionality of the developed system. It describes the testing setup, environment, and strategy used to verify each module individually and as part of the integrated framework. Both unit testing and integration testing were performed to ensure accurate behavior across a range of scenarios. The outcomes of these tests are presented and analyzed to demonstrate the reliability and robustness of the system.

4.1. Testing Setup

To verify the correctness and robustness of each module, a collection of carefully selected Python test cases was used. These test cases were designed to evaluate the system's ability to parse, analyze, and process a wide range of syntactic structures and data patterns. Each module was independently tested using representative code samples to ensure correct extraction of dependencies, memory behavior estimation, and integration logic. Successful parsing and accurate analysis of these test cases confirmed the functional readiness of the system's components.

4.2 Module Testing

- **Test Case Selection:** For each module, multiple Python code samples were prepared, covering a variety of scenarios including edge cases and common patterns.
- **Execution and Analysis:** The selected code samples were passed through the relevant module (e.g., DDG or Memory Estimator). The module's output—such as dependency mappings or memory usage predictions—was then collected.
- **Validation Against Python Runtime:** The actual runtime behavior of each code sample was observed by executing it in a controlled Python environment. Results such as variable access patterns, memory interactions, and evaluation order were recorded.
- **Comparison and Verification:** The outputs generated by the modules were compared against the observed runtime behavior. Discrepancies were analyzed to identify issues in the rule logic or parsing strategy.
- **Iteration:** Based on mismatches or incorrect predictions, the underlying rules and logic were refined to improve accuracy.

4.3 Integration Testing

The outputs of the DDG and Memory Estimator modules were fed into the Parallelizer module, to get one integrated output, that is then fed into the system (master) to carry out scheduling on it. This integration was designed to simulate real-world usage where dependency analysis and memory estimation inform the generation of parallelizable code blocks.
The results were compared against the runtime results.

Chapter 5: Conclusions and Future Work

This chapter should summarize the whole project; it features and limitation. Moreover, you should give directions for future work

5.1. Faced Challenges

During the development of the static parallelization framework, several challenges were encountered and addressed as follows:

- **Defensive Memory Estimation for Each Line:** Estimating line-level memory impact without execution was difficult due to Python's dynamic nature. AI and existing frameworks failed to provide accurate or reliable estimations.
- **Limitations of AI/GNN Methods:** ML and GNN models required labeled datasets that don't exist for Python. Their predictions lacked safety guarantees, making them unsuitable for dependable scheduling.

5.2. Gained Experience

The development of the static parallelization framework provided valuable skills and insights in Python semantics, memory modeling, and static analysis:

- **Understanding of Python AST and Syntax** Learned to parse, traverse, and normalize AST nodes for rule-based inspection, Handled diverse syntactic constructs and extended support for less common patterns.
- **Understanding of Python Memory Model:** Studied how Python stores primitives and data structures (especially lists), Gained insight into scoping, reference counting, and garbage collection.
- **Rule-Based Static Analysis:** Translated language behavior into safe, deterministic heuristics, Balanced precision and safety through empirical refinement.
- **Modular Design & Integration:** Built loosely coupled modules (DDG, memory estimator, aggregator), Designed metadata and interfaces for smooth inter-module communication.
- **Testing and Validation:** Created AST-specific test cases, Diagnosed mismatches between static analysis and runtime behavior.

5.3. Future Work

This project establishes a foundational static parallelization framework for Python using rule-based analysis. Future work can expand its capabilities in the following directions:

- **Extending Rule Coverage:** Support additional data structures (e.g., dict, set, deque, tuple, custom classes), Add rules for nested and aliased structures (e.g., list of dicts), Handle idiomatic constructs like comprehensions and merges.
- **Library Integration:** Model behavior of common libraries (e.g., NumPy, pandas) for static analysis, detect parallelizable patterns in functional constructs like map or reduce, Add support for parallel-aware libraries (e.g., Dask, joblib).

References

[1] S. C. Müller, G. Alonso, A. Amara, and A. Csillaghy, “Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud,” in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14), Broomfield, CO, USA, October 6–8, 2014, pp. 645–659.

[2] Afif J. Almghawish, Ayman M. Abdalla, Ahmad B. Marzouq, “An Automatic Parallelizing Model for Sequential Code Using Python,” *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 7, no. 3, pp. 276–282, March 2017.

[3] Yuanyuan Shen, Manman Peng, Shiling Wang, Qiang Wu, “Towards parallelism detection of sequential programs with graph neural network,” *Future Generation Computer Systems*, vol. 125, pp. 515–525, 2021.