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
Nour Ayman Amin

Supervised by
Dr/Lydia Wahid

1/7/2025

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

I would like to express my deepest gratitude to the individuals who have supported and guided me throughout this project. Without their invaluable assistance, this work would not have been possible.
First and foremost, I am profoundly grateful to my supervisor, **Dr. Lydia Wahid**, for their unwavering guidance, insightful feedback, and constant encouragement. Their expertise in distributed systems was instrumental in shaping the direction of this research, and their patience and mentorship provided me with the confidence to tackle the complex challenges of this project.

I would also like to extend my sincere thanks to my colleagues and teammates, **Omar Said, Abdelrahman Mohamed, and Mostafa Magdy**, who developed the upstream analysis modules and the communication protocols. Their diligent work on the Code Structure Analyzer, the Memory Footprint Profiler and the Remote Producer Server provided the critical requirements upon which this scheduler is built. Our collaborative sessions and technical discussions were essential for ensuring seamless integration and a cohesive system architecture.

My gratitude also goes to the faculty and staff of the **Computer Department at the Faculty of Engineering Cairo University** for providing a stimulating academic environment and the necessary resources for this research.
Finally, I wish to thank my family and friends for their endless support, patience, and belief in me. Their encouragement was a constant source of motivation during the most challenging phases of this project.

Table of Contents

List of Figures

List of Abbreviations

Abbreviation	Description
JSON	JavaScript Object Notifier
DDG	Data Dependency Graph
CSV	Comma-Separated Values
AST	Abstract Syntax Tree

Contacts

Team Member

Name	Email	Phone Number
Nour Ayman Amin	nour.el-dabaa02@eng-st.cu.edu.eg	+20 10 2400 3992

Supervisor

Name	Email	Number
Lydia Wahid	LydiaWahid@outlook.com	-

This page is left intentionally empty

Chapter 1: Introduction

The efficient execution of Python code in distributed environments remains a significant challenge, largely due to the language's inherent sequential nature and the complexities of managing resources across multiple machines. While Python is the language of choice for data science and analytics, scaling its execution requires developers to manually parallelize their code—a process that is intricate, error-prone, and demands deep expertise in distributed systems. This manual effort involves identifying data dependencies, estimating memory consumption, partitioning data, and orchestrating task execution, creating a substantial barrier to leveraging the full power of distributed computing.

This report details the design and implementation of the RunSurge Scheduler, a sophisticated orchestration module at the heart of a system designed to automate this complex process. The Scheduler's primary role is to act as the central intelligence that transforms a standard, sequential Python script into a parallel, distributed execution plan. It achieves this by consuming the outputs of two critical pre-analysis modules: a Code Structure Analyzer, which produces a block-level Data Dependency Graph (DDG), and a Memory Footprint Profiler, which provides detailed estimates of memory usage.

Armed with this structural and resource information, the RunSurge Scheduler makes intelligent, dynamic decisions to generate an optimal execution strategy, demonstrating a significant advancement over manual parallelization techniques.

1.1. Motivation and Justification

The core motivation for the RunSurge Scheduler is to **automate the complex decision-making involved in distributed program execution**. While other systems may provide tools for parallel programming, they often leave the most difficult questions to the developer: How should the program be split into tasks? Which tasks can run in parallel? How much memory will each task require? Where should each task be run to maximize efficiency?
Data Dependency Graph (DDG) analyzes variable interactions to capture dependencies between code segments, ensuring correctness during parallel execution.

The RunSurge Scheduler is justified by its ability to answer these questions automatically through a novel, multi-level scheduling algorithm. Its key contributions are:

- **Intelligent Resource-Aware Scheduling:** It dynamically matches the memory requirements of code blocks, as determined by the Memory Footprint Profiler, against the available resources of worker nodes, preventing out-of-memory failures and ensuring efficient allocation.
- **Automated Data Parallelism:** For tasks that are too large for any single node, the scheduler automatically devises a data-parallelism strategy. It calculates the optimal number of data chunks required to ensure each parallel task can execute within the memory constraints of the available nodes.
- **Advanced Pipeline Optimization with "Stage Fusion":** A key innovation is the scheduler's ability to handle chained, multi-stage parallel pipelines. Rather than naively scheduling each stage independently, it identifies entire parallel chains, determines the maximum required degree of parallelism for the whole chain, and "fuses" the stages into a single, highly efficient set of parallel tasks. This advanced optimization minimizes disk I/O and network overhead, which are often the primary bottlenecks in distributed data processing.

By abstracting away these complex scheduling and optimization decisions, the RunSurge Scheduler empowers developers to scale their Python applications with minimal effort, making distributed computing more practical and accessible.

1.2. Document Organization

This document details the design, implementation, and verification of the RunSurge Scheduler, the core orchestration component of the distributed execution system for Python. The report is structured to provide a clear and comprehensive overview of the Scheduler's architecture and functionality.

- **Chapter 2** provides a comprehensive overview of the **System Design and Architecture**. It begins by outlining the high-level goals and operational assumptions of the RunSurge system. It then focuses in detail on the **Scheduler module**, describing its inputs, its multi-level scheduling algorithm, and its advanced "Stage Fusion" logic for optimizing parallel pipelines.
- **Chapter 3** describes the **System Testing and Verification** methodology. This chapter details the setup for unit tests that validate specific scheduling heuristics and presents the results of end-to-end integration tests that confirm the system's ability to correctly generate complete, executable plans for a variety of program structures.
- **Chapter 4** concludes the report by summarizing the key achievements and contributions of the RunSurge Scheduler. It discusses the primary **Challenges Faced** during development and outlines promising

directions for **Future Work**, such as implementing dynamic load balancing and more advanced parallel execution patterns.

Chapter 2: 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.

2.1. Overview and Assumptions

The RunSurge system is designed to automate the parallel execution of sequential Python programs on a distributed cluster of worker nodes. The primary goal is to minimize manual intervention by dynamically analyzing program structure and memory requirements to generate an efficient, parallel execution schedule. The system operates on a master-worker architecture where a central scheduler (the Master) dispatches tasks to a pool of available workers.

The system relies on two preceding analysis modules that provide critical inputs for the scheduling process:

1. **Code Structure Analyzer (Successor to DDG):** This module analyzes the input Python script and produces a block-based Data Dependency Graph (DDG). It identifies sequential blocks of code and explicitly defines their data dependencies (e.g., Block 2 depends on variable x from Block 1). This structural information is provided in the main_lists.json file.

2. **Memory Footprint Profiler (Successor to Memory Estimator):** This module performs a static or profile-guided analysis of the code to estimate its memory consumption. It produces two key artifacts:
    - **Live Variable Footprint (main_lines_footprint.json):** For each line of code, this file details the size and length of all variables that are "live" (in memory) at that point.
    - **Function Execution Footprint (func_lines_footprint.json):** This file provides a detailed trace of the peak memory usage inside each function call, line by line.

The scheduler makes the following key assumptions:

- A shared file system (e.g., NFS) is accessible by the master and all worker nodes, located at the path defined by JOBS_DIRECTORY_PATH.
- Workers have a gRPC interface for receiving task assignments and data notifications.
- The input data for a job is provided as a single CSV file.

2.2. System Architecture

The system architecture consists of a **Master Node** and multiple **Worker Nodes**.

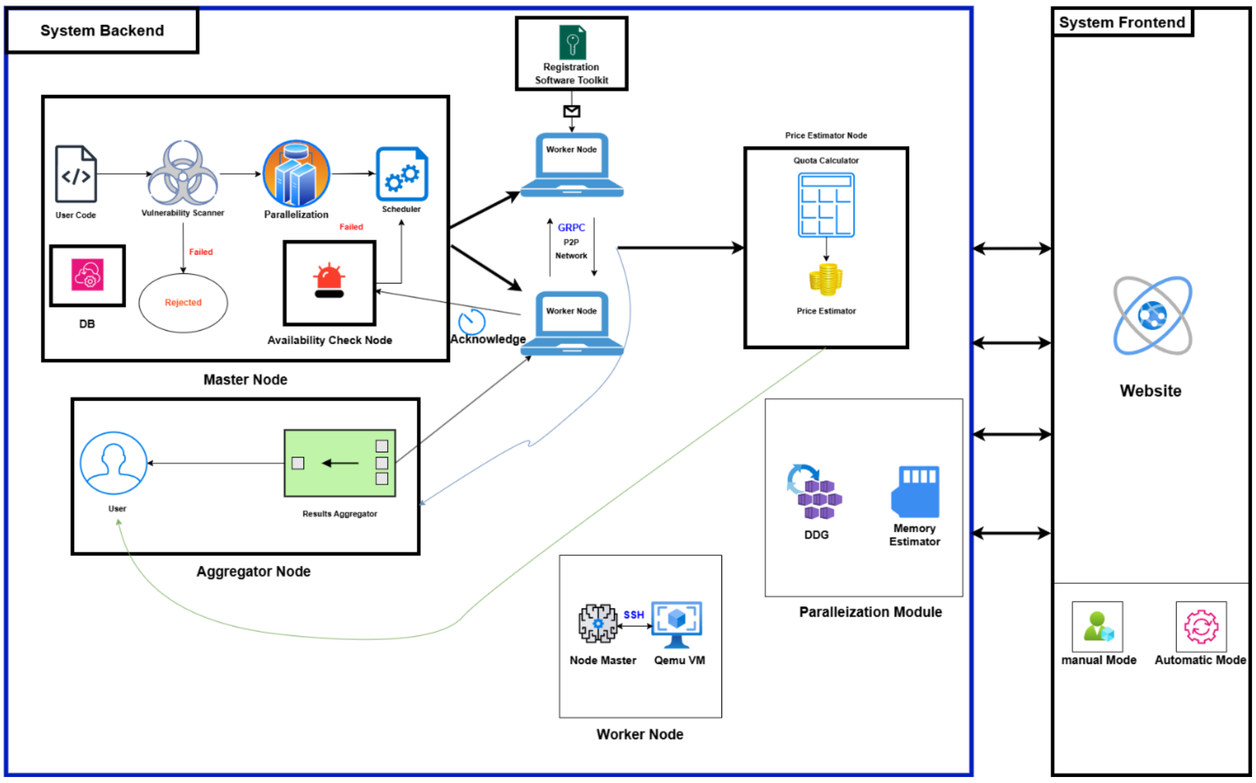
1. **User Interaction:** A user submits a Python script and an input data file to the system, creating a new Job.

2. **Analysis Phase:**
    - The **Code Structure Analyzer** processes the script to create the main_lists.json DDG.
    - The **Memory Footprint Profiler** processes the script to create the main_lines_footprint.json and func_lines_footprint.json profiles.

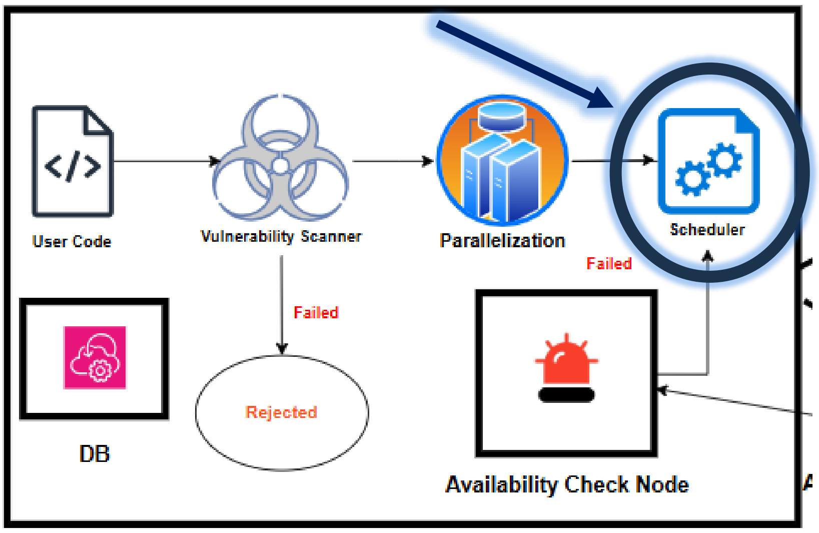
3. **Scheduling Phase (The scheduler.py logic):**
    - The **Master Scheduler** polls for new, unscheduled jobs.
    - It retrieves the job's analysis files and queries the database for available worker node resources (RAM, IP address).
    - It executes a multi-level scheduling algorithm to create an Execution Plan.
    - It generates Python scripts for each task defined in the plan.

4. **Execution Phase:**
    - The **Master** assigns tasks to **Worker Nodes** via gRPC, sending the generated script.
    - The **Master** sends DataNotification messages for data that is ready at schedule time (initial input, master-created chunks).
    - **Workers** execute their assigned scripts. They wait for data dependencies (either via notification or file polling), perform computations, and write outputs (intermediate partial files or the final result) to the shared file system.

2.2.1. Block Diagram



2.1 System Block Diagram



2.2 Master Node

2.3. The Scheduler Module

The Scheduler is the core orchestration component of the RunSurge system. Its primary function is to transform the high-level dependency graph and memory profiles into a concrete, executable plan distributed across available worker nodes. This is achieved through a hierarchical, multi-level scheduling strategy designed to maximize resource utilization and enable parallelism where possible.

2.3.1. Functional Description

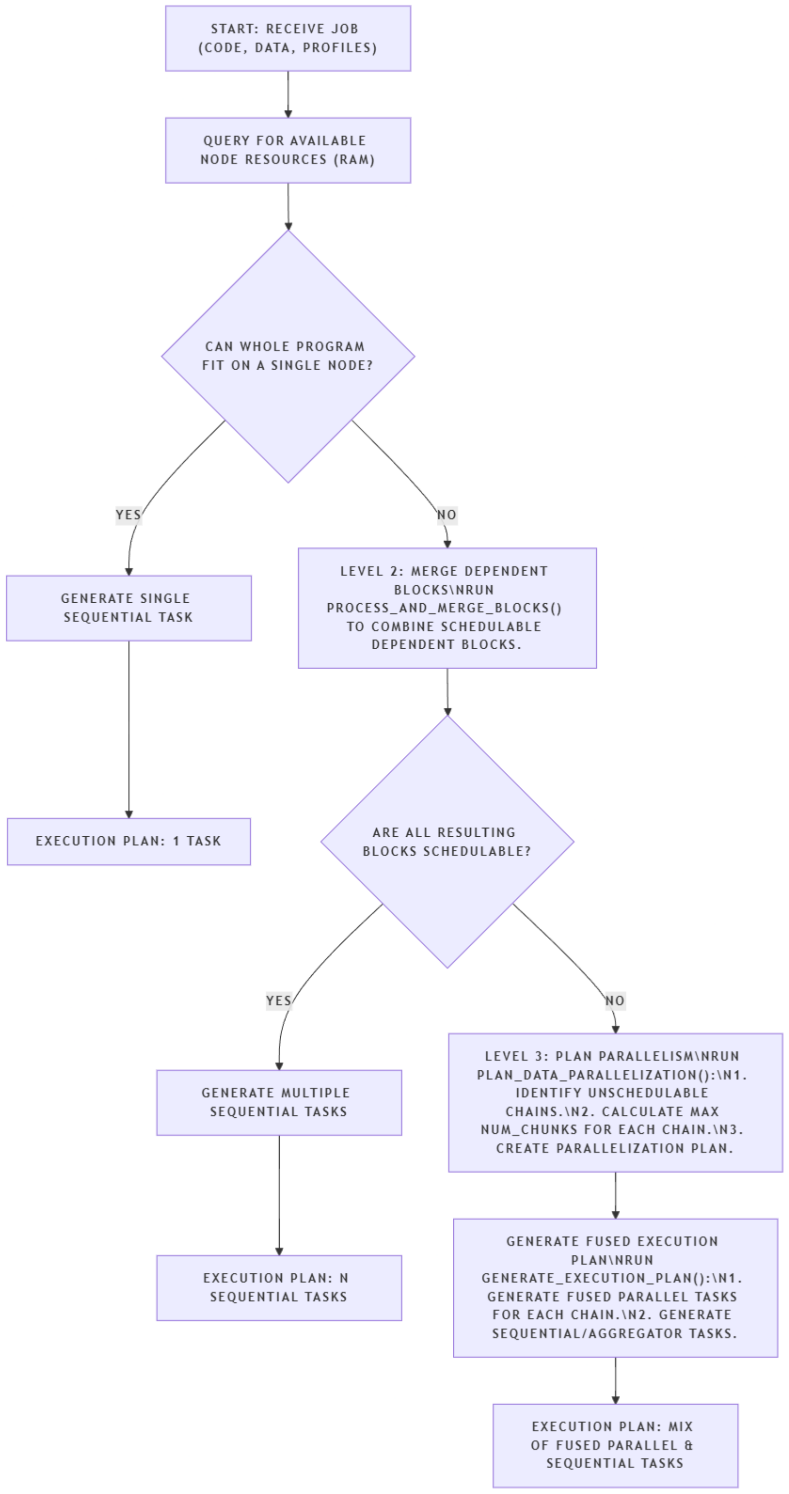
The scheduling process operates on three distinct levels:

- **Level 1: Whole Program Scheduling:** The scheduler first performs a holistic analysis to determine if the entire program can be executed sequentially on a single, sufficiently powerful worker node. It calculates the total peak memory requirement for the whole program by simulating its execution using the provided footprints. If a node with adequate memory is available, the entire program is assigned as a single task. This is the most efficient approach for smaller jobs as it avoids all parallelization overhead.

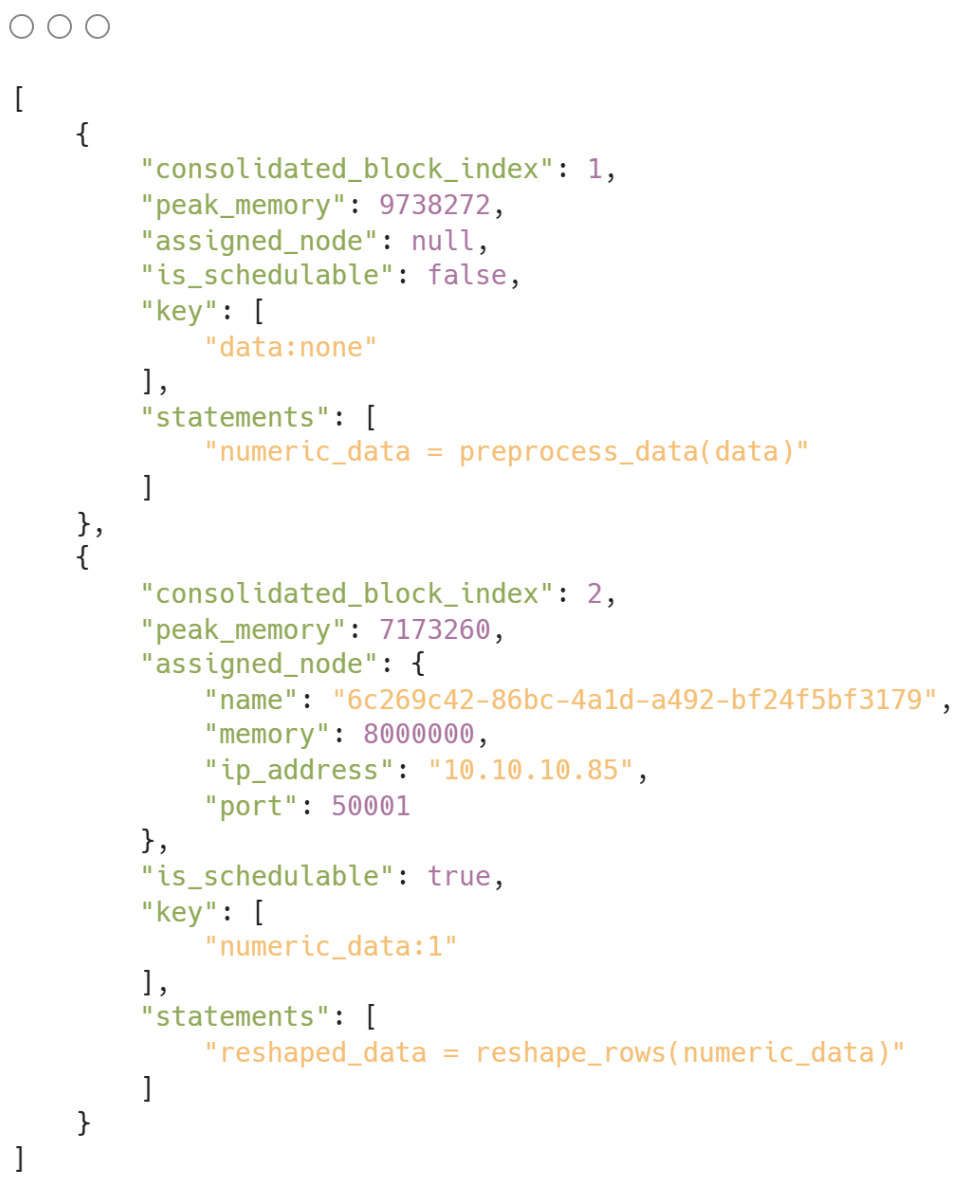
- **Level 2: Sequential Block Merging and Scheduling:** If Level 1 fails, the scheduler treats the program as a Directed Acyclic Graph (DAG) of code blocks, as defined by the main_lists.json. It attempts to merge dependent blocks (process_and_merge_blocks) where the combined memory footprint of the merged block still fits on an available node. This reduces task-switching overhead and network communication. After merging, it attempts to schedule each of the resulting sequential blocks on the smallest possible fitting node. Blocks that still cannot be scheduled are passed to the next level.

- **Level 3: Parallelization and Stage Fusion:** This level handles blocks that are too large to run on any single node. It employs a data-parallelism strategy based on a **"Maximum Required Parallelism"** model.
    1. **Chain Identification:** The scheduler first identifies "chains" of unschedulable blocks that are dependent on one another.
    2. **Parallelism Calculation:** For each stage in the chain, it calculates the required degree of parallelism (num_chunks) needed to fit a single data chunk on the smallest available node. It then takes the *maximum* of these required values as the definitive division factor for the *entire* chain, ensuring all stages can execute without memory errors.
    3. **Stage Fusion:** Instead of creating separate tasks for each link in the parallel chain (e.g., Task A for func1, Task B for func2), the scheduler intelligently fuses these stages. It generates a single set of parallel

tasks where each task executes the entire chain's logic (e.g., runs func1 then immediately runs func2 on the result in-memory), minimizing disk I/O and maximizing pipeline efficiency.

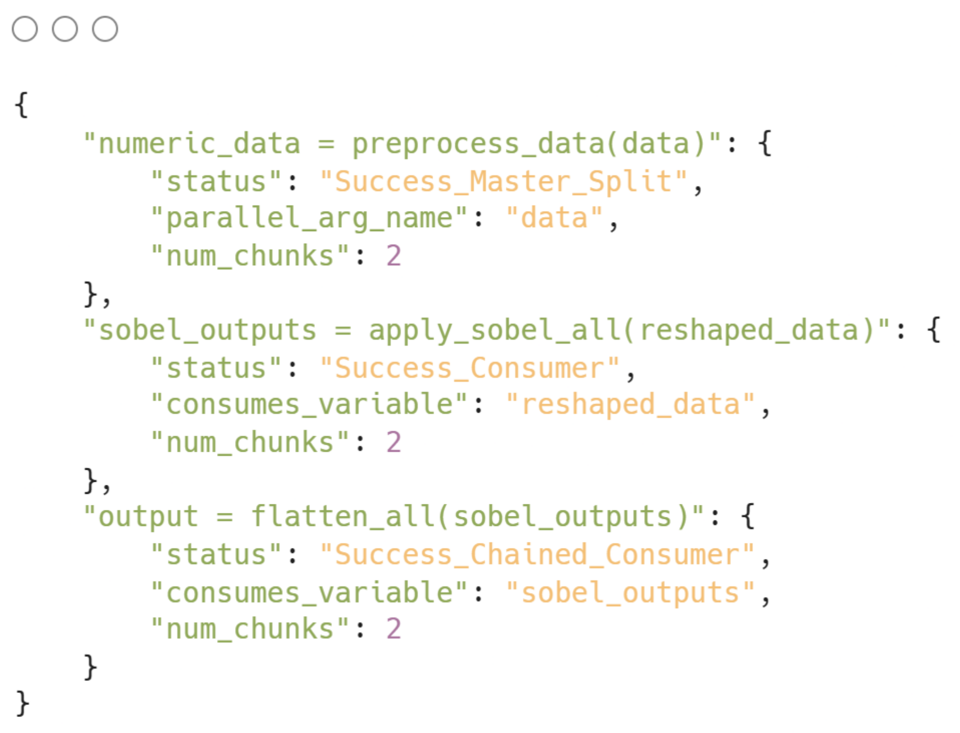

START: RECEIVE JOB (CODE, DATA, PROFILES)
QUERY FOR AVAILABLE NODE RESOURCES (RAM)
CAN WHOLE PROGRAM FIT ON A SINGLE NODE?
YES
NO
GENERATE SINGLE SEQUENTIAL TASK
EXECUTION PLAN: 1 TASK
LEVEL 2: MERGE DEPENDENT BLOCKS\NRUN PROCESS_AND_MERGE_BLOCKS() TO COMBINE SCHEDULABLE DEPENDENT BLOCKS.
ARE ALL RESULTING BLOCKS SCHEDULABLE?
YES
NO
GENERATE MULTIPLE SEQUENTIAL TASKS
EXECUTION PLAN: N SEQUENTIAL TASKS
LEVEL 3: PLAN PARALLELISM\NRUN PLAN_DATA_PARALLELIZATION():\N1. IDENTIFY UNSCHEDULABLE CHAINS.\N2. CALCULATE MAX NUM_CHUNKS FOR EACH CHAIN.\N3. CREATE PARALLELIZATION PLAN.
GENERATE FUSED EXECUTION PLAN\NRUN GENERATE_EXECUTION_PLAN():\N1. GENERATE FUSED PARALLEL TASKS FOR EACH CHAIN.\N2. GENERATE SEQUENTIAL/AGGREGATOR TASKS.
EXECUTION PLAN: MIX OF FUSED PARALLEL & SEQUENTIAL TASKS

2.3 Scheduler Flow



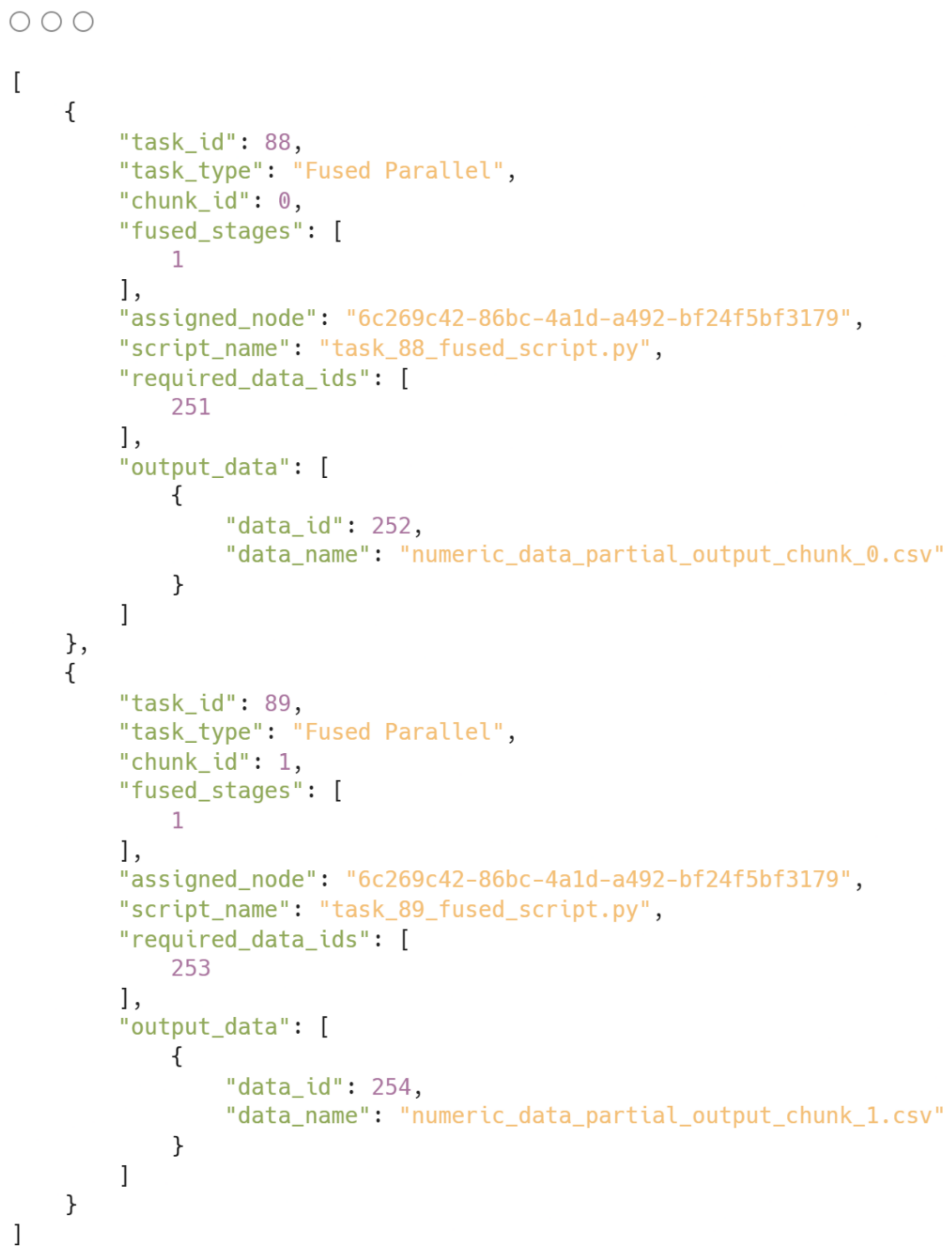
```
[
    {
        "consolidated_block_index": 1,
        "peak_memory": 9738272,
        "assigned_node": null,
        "is_schedulable": false,
        "key": [
            "data:none"
        ],
        "statements": [
            "numeric_data = preprocess_data(data)"
        ]
    },
    {
        "consolidated_block_index": 2,
        "peak_memory": 7173260,
        "assigned_node": {
            "name": "6c269c42-86bc-4a1d-a492-bf24f5bf3179",
            "memory": 8000000,
            "ip_address": "10.10.10.85",
            "port": 50001
        },
        "is_schedulable": true,
        "key": [
            "numeric_data:1"
        ],
        "statements": [
            "reshaped_data = reshape_rows(numeric_data)"
        ]
    }
]
```

2.4 Level 2 Output



```
{
    "numeric_data = preprocess_data(data)": {
        "status": "Success_Master_Split",
        "parallel_arg_name": "data",
        "num_chunks": 2
    },
    "sobel_outputs = apply_sobel_all(reshaped_data)": {
        "status": "Success_Consumer",
        "consumes_variable": "reshaped_data",
        "num_chunks": 2
    },
    "output = flatten_all(sobel_outputs)": {
        "status": "Success_Chained_Consumer",
        "consumes_variable": "sobel_outputs",
        "num_chunks": 2
    }
}
```

2.5 Level 3 Output



```json
[
    {
        "task_id": 88,
        "task_type": "Fused Parallel",
        "chunk_id": 0,
        "fused_stages": [
            1
        ],
        "assigned_node": "6c269c42-86bc-4a1d-a492-bf24f5bf3179",
        "script_name": "task_88_fused_script.py",
        "required_data_ids": [
            251
        ],
        "output_data": [
            {
                "data_id": 252,
                "data_name": "numeric_data_partial_output_chunk_0.csv"
            }
        ]
    },
    {
        "task_id": 89,
        "task_type": "Fused Parallel",
        "chunk_id": 1,
        "fused_stages": [
            1
        ],
        "assigned_node": "6c269c42-86bc-4a1d-a492-bf24f5bf3179",
        "script_name": "task_89_fused_script.py",
        "required_data_ids": [
            253
        ],
        "output_data": [
            {
                "data_id": 254,
                "data_name": "numeric_data_partial_output_chunk_1.csv"
            }
        ]
    }
]
```

2.6 Execution Plan

2.3.2. Modular Decomposition

The scheduler is composed of several key Python functions:

- **scheduler():** The main async loop that polls for jobs and orchestrates the entire process.
- **plan_data_parallelization():** The core "brain" for Level 3. It implements the robust chain analysis and "Maximum Required Parallelism" calculation to create a high-level parallelization plan.
- **generate_execution_plan():** The primary orchestrator that executes the plans. It implements the two-pass "Stage Fusion" logic, first generating all parallel fused tasks and then generating the sequential/aggregator tasks that depend on them.
- **generate_sequential_assignment():** A specialized function responsible for generating the script and database records for a single schedulable task, including the complex aggregation logic.
- **consolidate_to_block_format() & process_and_merge_blocks():** These functions implement the Level 2 block merging and scheduling logic.
- calculate_peak_memory_for...(): Utility functions that use the profiler's output to determine memory requirements for different code segments.
- The scheduler currently assumes a 1-to-1 mapping for data flow in parallel chains. It does not implement a generic M-to-N shuffle, which would require a more complex data redistribution mechanism.
- The system's correctness depends on the accuracy of the memory profiles provided by the upstream **Memory Footprint Profiler**.

2.3.3. Design Constraints

1. **Dependency on Upstream Analysis Modules:**

   The scheduler's effectiveness is fundamentally dependent on the quality and accuracy of the inputs it receives from the two upstream analysis modules.

- **Code Structure Analyzer:** The scheduler assumes that the main_lists.json file provides a correct and complete Data Dependency Graph (DDG) at the block level. It does not perform its own dependency analysis and will produce an incorrect schedule if the provided dependencies are flawed.
- **Memory Footprint Profiler:** Scheduling decisions, particularly for data parallelism, are highly sensitive to the memory estimates in main_lines_footprint.json. Inaccurate or non-representative profiling

can lead to either sub-optimal parallelization (over-provisioning chunks) or, more critically, out-of-memory failures on worker nodes.

2. **1-to-1 Data Flow in Parallel Pipelines:**

   The current implementation of data parallelism is based on a **1-to-1 mapping** between the chunks of consecutive parallel stages. This "stage fusion" model is highly efficient for embarrassingly parallel pipelines. However, it does not support more complex data flow patterns:

- **Fan-in / Aggregation (M-to-1):** The system cannot automatically generate a parallel aggregation stage where, for example, 10 parallel tasks feed their results into a single task that performs a sum or average. This type of aggregation must be performed by a final *sequential* block.
- **Fan-out / Re-shuffle (M-to-N):** The scheduler does not implement a generic data shuffle. It cannot, for instance, take the output of 4 parallel tasks and redistribute or re-partition that data across 7 new parallel tasks. The degree of parallelism, once set at the beginning of a parallel chain, remains constant throughout that fused chain.

3. **Nature of Parallelism:**

   The system's parallelization strategy is exclusively **data parallelism**. It excels at splitting large input datasets across multiple nodes. It does not currently support **task parallelism**, where truly independent function calls (e.g., a = func1() and b = func2(), with no dependency) could be executed concurrently on different nodes. All scheduling decisions are driven by data dependencies and memory constraints, not by identifying independent computational tasks.

4. **Reliance on Filesystem Polling for Inter-Task Dependency:**

   For dependencies between tasks (e.g., a sequential task waiting for the partial outputs of a parallel chain), the system relies on the wait_for_data function, which polls the shared filesystem.

- **Constraint:** This introduces a potential latency, as the consumer task actively checks for the existence of files instead of being passively notified.
- **Trade-off:** This design was chosen for its simplicity and robustness, avoiding the need for a complex, stateful master or a message bus to

manage inter-worker notifications, which would significantly increase the system's architectural complexity.

5. **Static Scheduling:**

   The entire execution plan is generated **statically** before any tasks are executed. The scheduler does not adapt to dynamic changes in the cluster during runtime.

- **Constraint:** It cannot react to a worker node failing or becoming slow mid-job. There is no mechanism for fault tolerance or re-scheduling failed tasks.
- **Constraint:** It does not perform dynamic load balancing. If one parallel task takes significantly longer than others (due to data skew), the overall job completion time will be dictated by this "straggler," as the system cannot re-distribute its work.

Chapter 3: 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.

3.1. 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.

3.2 Module Testing

- **calculate_peak_memory_for_merged_block:** Unit tests were designed to verify the correctness of context-aware memory calculation. Test cases included:
    - A block with no external dependencies.
    - A block with a single external dependency.
    - A block with multiple external dependencies, ensuring only the sizes of relevant live variables were summed.
    - Verification that the internal function execution footprint was correctly added to the live variable sum.
- **plan_data_parallelization:** Unit tests focused on the "Maximum Required Parallelism" logic:
    - **Test Case 1 (Single Parallel Block):** A single unschedulable block. Verified that num_chunks was calculated correctly based on its input size.
    - **Test Case 2 (Chained, First Stage is Bottleneck):** A two-stage chain where func1 required 4 chunks and func2 required 2. Verified that the final num_chunks for both plans was 4.
    - **Test Case 3 (Chained, Second Stage is Bottleneck):** A two-stage chain where func1 required 4 chunks and func2 required 7. Verified that the final num_chunks for both plans was 7.

3.3 Integration Testing

Integration tests were designed to validate the end-to-end workflow of the scheduler for representative job structures. For each test, the final execution_plan.json was inspected, and the generated worker scripts were manually verified for correctness.

- **Test Case 1: Simple Sequential Job:** A program small enough to fit on a single node.
    - **Expected Outcome:** The schedule_program_whole (Level 1) logic succeeds. A single sequential task is created.
      The execution_plan.json shows one entry.
    - **Result:** As expected.
- **Test Case 2: Multi-Block Sequential Job:** A program with multiple dependent blocks, each schedulable on its own.
    - **Expected Outcome:** Level 1 fails. Level 2 (process_and_merge_blocks) successfully creates a schedule with multiple sequential tasks assigned to different nodes.
    - **Result:** As expected. The generated scripts correctly contained wait_for_data calls for intermediate files (e.g., x.csv).

- **Test Case 3: Chained Parallel Job with Final Aggregator:** The three-stage program (data -> x -> y -> output) where the first two stages are parallel-fused and the final stage is a sequential aggregator.
  - **Expected Outcome:** The plan_data_parallelization function correctly identifies the [Block 1, Block 2] chain and computes a single num_chunks factor. The generate_execution_plan function generates one set of "fused" parallel scripts for func1 and func2. It then generates a single sequential script for the final block (output = func2(y)) which contains the aggregation logic to wait for all y_partial_output_chunk_i.csv files.
  - **Result:** As expected. The execution_plan.json correctly detailed the two fused parallel tasks and the final single aggregator task.

Chapter 4: Conclusions and Future Work

4.1. Faced Challenges

- **Accurate Memory Estimation:** The entire scheduling logic is predicated on the accuracy of the upstream Memory Footprint Profiler. Early iterations struggled with the dynamic nature of Python, where object sizes can change unexpectedly. A profile-guided approach proved more reliable than purely static analysis.
- **Dependency Complexity:** Managing the dependencies between sequential tasks, parallel tasks, and aggregator tasks was a significant challenge. The transition from a simple task-based model to a "stage fusion" model required a complete rework of the execution planner to correctly track data flow and populate the necessary instructions (split_instructions) for downstream stages.
- **State Management:** Designing a stateless master scheduler that "fires and forgets" was a key design goal for simplicity. This necessitated a robust polling mechanism (wait_for_data) on the workers, creating a trade-off between implementation simplicity and execution latency.

4.2. Gained Experience

This project provided invaluable experience in the design and implementation of distributed computing systems. Key takeaways include the importance of a layered, heuristic-based approach to complex problems like scheduling. Instead of seeking a single perfect algorithm, building a hierarchy of simpler, specialized strategies proved more robust and effective. Furthermore, the project underscored the critical need for clear data contracts and communication mechanisms (like the parallelization_plan and split_instructions dictionaries) between different modules of a complex system.

4.3. Future Work

:

- **Implementation of a Full M-to-N Shuffle:** The current 1-to-1 pipeline model is effective but cannot handle cases where the optimal degree of parallelism changes between stages. The next major step would be to implement a true shuffle mechanism. This would involve upgrading the planner to generate a detailed shuffle map and enhancing worker scripts to handle splitting their output for multiple consumers and aggregating input from multiple producers.
- **Task Completion Notifications:** To move away from filesystem polling, a task completion notification system could be implemented. When a worker finishes a task, it would send a signal back to the master (or directly to a message bus). The master could then send DataNotification messages to dependent tasks, enabling a more efficient, event-driven execution model.
- **Dynamic Node Selection and Fault Tolerance:** The current node selection is a simple round-robin. A more advanced implementation would consider data locality (i.e., preferring to schedule a task on a node that already holds its input data) and resource heterogeneity. Additionally, implementing mechanisms for detecting failed tasks and rescheduling them would significantly improve the system's resilience.

References

[1] T. El-Ghazaly, T. El-Gogary, and M. G. El-Hariry, "Pydron: A Semi-Automatic Parallelization Tool for Python," in 2018 13th International Conference on Computer Engineering and Systems (ICCES), Cairo, Egypt, 2018, pp. 248-253.

[2] A. S. S. Al-Hayali and S. K. A. Al-Dulaimi, "PyParallelize: A Python Tool for Dynamic AST-Based Parallelization of Sequential Code," International Journal of Computer Science and Network Security, vol. 22, no. 10, pp. 52-58, Oct. 2022.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in OSDI'04: 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, 2004, pp. 137-150.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in HotCloud'10: 2nd USENIX Workshop on Hot Topics in Cloud Computing, Boston, MA, 2010.

[5] Z. Wang, K. Chen, and H. Jin, "Towards Parallelism Detection of Sequential Programs with Graph Neural Network," in Proceedings of the 30th International Conference on Program Comprehension (ICPC '22), 2022, pp. 586-590.

[6] D. G. Lowery and G. F. Riley, "A Profile-Guided Approach for the Parallelization of Sequential Code," in Proceedings of the 2008 Spring simulation multiconference (SpringSim '08), 2008, pp. 15-22.