NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

PyJedAI Parallelization with MPIRE

Ilias A. Kontonis

Supervisors: **Manolis Koubarakis,** Professor
George Papadakis, Senior Researcher
Konstantinos Nikoletos, M.Sc. student

ATHENS

JANUARY 2023

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Παραλληλοποίηση του PyJedAI με τη χρήση του MPIRE

Ηλίας Α. Κοντονής

Επιβλέποντες: **Μανόλης Κουμπαράκης,** Καθηγητής
Γιώργος Παπαδάκης, Ανώτερος Ερευνητής
Κωνσταντίνος Νικολέτος, Φοιτητής M.Sc.

ΑΘΗΝΑ

ΙΑΝΟΥΑΡΙΟΣ 2023

BSc THESIS

PyJedAI Parallelization with MPIRE

Ilias A. Kontonis
S.N.: 1115201700055

SUPERVISORS: **Manolis Koubarakis,** Professor
George Papadakis, Senior Researcher
Konstantinos Nikoletos, M.Sc. student

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Παραλληλοποίηση του PyJedAI με τη χρήση του MPIRE

Ηλίας Α. Κοντονής
Α.Μ.: 1115201700055

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Μανόλης Κουμπαράκης,** Καθηγητής
Γιώργος Παπαδάκης, Ανώτερος Ερευνητής
Κωνσταντίνος Νικολέτος, Φοιτητής M.Sc.

ABSTRACT

Entity resolution is a critical task in various applications, but it faces quadratic complexity. To make entity resolution scalable to large datasets, blocking is typically employed. Syntactic blocking methods usually group similar entities into overlapping blocks, reducing the number of necessary comparisons. Further efficiency gains are achieved with Meta-blocking, which prunes unnecessary comparisons in overlapping blocks, significantly improving precision without sacrificing much recall.

However, despite its time efficiency, applying Meta-blocking to solve entity resolution problems on very large datasets remains a challenge. For instance, processing 7.4 million entities can take almost eight full days on a high-end server.

In this thesis, we work with the parallelization of the python framework PyJedAI. Python introduces new challenges due to the Global Interpreter Lock (GIL) which forces us to implement a fork-join model instead of generating multiple threads. We use the MPIRE python module to implement the parallel Meta-blocking algorithms.

The experimental analysis validates the scalability of the parallel implementation as well as the significant time reduction in certain steps of the Meta-blocking. We also analyze the deadlocks we encountered in the time efficiency of our implementation due to the fork-join model and how it is possible to get over them.

SUBJECT AREA: PyJedAI parallelization

KEYWORDS: Entity Resolution, Meta-blocking, parallelization, fork-join, GIL, MPIRE, PyJedAI

ΠΕΡΙΛΗΨΗ

Η ανάλυση οντοτήτων είναι μια κρίσιμη εργασία σε διάφορες εφαρμογές, αλλά αντιμετωπίζει την τετραγωνική πολυπλοκότητα. Για να καταστεί εφικτή η ανάλυση οντοτήτων με μεγάλα σύνολα δεδομένων, χρησιμοποιείται η ομαδοποίηση. Συντακτικές μέθοδοι ομαδοποίησης (blocking) συνήθως οργανώνουν παρόμοιες οντότητες σε αλληλοκαλυπτόμενα μπλοκ, μειώνοντας τον αριθμό των απαραίτητων συγκρίσεων. Περαιτέρω κέρδη απόδοσης επιτυγχάνονται με τη μετα-ομαδοποίηση (meta-blocking), το οποίο περιορίζει τις περιττές συγκρίσεις σε επικαλυπτόμενα μπλοκ, βελτιώνοντας σημαντικά την ακρίβεια χωρίς να μειώνεται πολύ η ανάκληση.

Παρά τη χρονική του απόδοση, η εφαρμογή της μετα-ομαδοποίησης (meta-blocking) για την επίλυση προβλημάτων επίλυσης οντοτήτων σε πολύ μεγάλα σύνολα δεδομένων παραμένει μια πρόκληση. Για παράδειγμα, η επεξεργασία 7,4 εκατομμυρίων οντοτήτων μπορεί να διαρκέσει σχεδόν οκτώ ολόκληρες ημέρες σε έναν διακομιστή υψηλής τεχνολογίας.

Σε αυτή τη διατριβή, εξετάζουμε την παραλληλοποίηση του python πακέτου PyJedAI. Η Python εισάγει νέες προκλήσεις λόγω του Global Interpreter Lock (GIL) και της ανάγκης να ενσωματωθεί ένα μοντέλο fork-join αντί της δημιουργίας πολλαπλών νημάτων. Χρησιμοποιούμε τη βιβλιοθήκη MPIRE για την υλοποίηση των παράλληλων αλγορίθμων μετα-ομαδοποίησης σε python.

Η πειραματική ανάλυση επικυρώνει την επεκτασιμότητα της παράλληλης υλοποίησης καθώς και τη σημαντική μείωση χρόνου σε ορισμένα στάδια της μετα-ομαδοποίησης. Επίσης, θα αναλύσουμε και τα αδιέξοδα που συναντήσαμε στη χρονική απόδοση της υλοποίησής μας λόγω του μοντέλου fork-join και πώς είναι δυνατόν να τα ξεπεράσουμε.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Παραλληλοποίηση του PyJedAI

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ανάλυση οντοτήτων, Meta-blocking, παραλληλοποίηση, fork-join, GIL, MPIRE, PyJedAI

ACKNOWLEDGEMENTS

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα θέλαμε να ευχαριστήσουμε τους επιβλέποντες, Μανόλη Κουμπαράκη, Γεώργιο Παπαδάκη, Κωνσταντίνο Νικολέτο για τη συνεργασία και την πολύτιμη συμβολή τους στην ολοκλήρωση της.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In the realm of data-intensive applications, Entity Resolution (ER) emerges as a pivotal and recurring challenge, essential for maintaining data quality and enabling efficient data analysis. This task involves identifying and linking disparate data records that refer to the same real-world entities [6] [8]. While conceptually straightforward, ER grapples with an inherent quadratic complexity, as its brute-force approach considers all possible pairs of input entities, thus rendering it computationally demanding as data volumes continue to surge [10].

To tame the quadratic time complexity of ER, blocking is typically used to reduce the processing to the pairs that are most likely to be matching [5] [26]. Meta-blocking refines the original set of blocks to further reduce these candidate pairs. As an example, consider Figure 1.1, where we initially had an heterogeneous set of entities and we derive the blocks out of the values of each attribute of each entity. We store the entities that contain a specific blocking key to the corresponding block. Then we build a graph whose nodes are entities and the link between two entities, represents the number of times they were in the same block. In the end, we prun the edges whose counter is below the average ranking position and the remaining edges indicate the possible entity duplicates.

Nevertheless, even with the efficient time management that Meta-blocking offers, the time complexity remains notably high. For instance, processing 7.4 million entities can consume nearly eight full days on a high-end server. This challenge calls for innovative strategies to accelerate the entity resolution process, especially when managing large datasets [11].

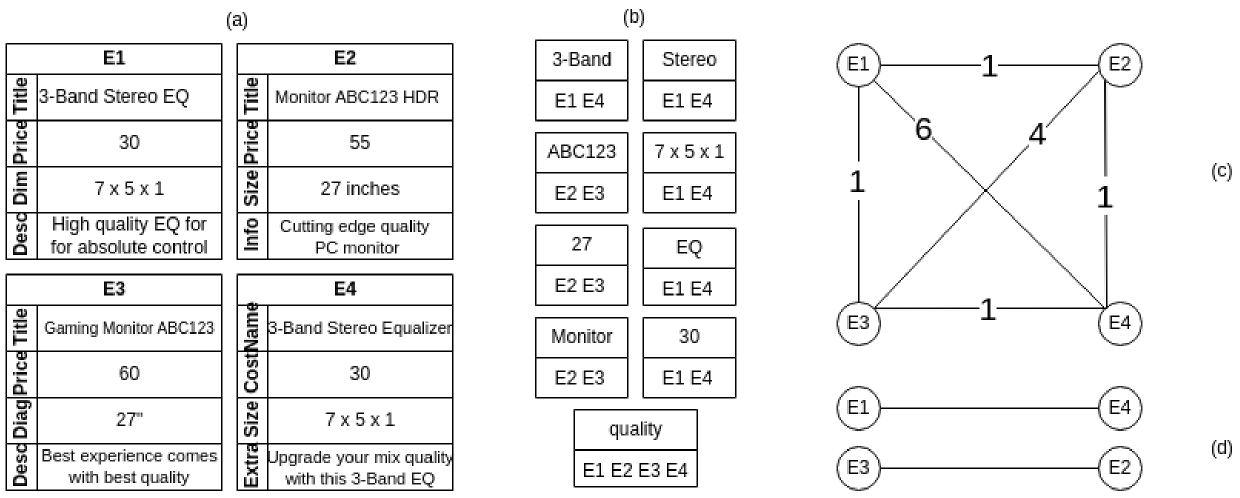

Figure 1.1: Simplified Meta Blocking Example:
(a) Heterogeneous entity collection, (b) the resulting set of attribute-agnostic blocks, (c) the blocking graph corresponding to it, (d) the pruned blocking graph.

In this thesis, we embark on the formidable task of parallelizing the Python framework PyJedAI [21], a robust tool for entity resolution. The parallelization endeavor encounters hurdles due to Python's Global Interpreter Lock (GIL) and necessitates a migration from the conventional multithreading approach in favor of a fork-join model. Our work is driven by the objective of making entity resolution more accessible and efficient for large-scale applications.

The empirical analysis within this thesis serves as a testament to the scalability and time reduction achieved through the parallel implementation of Meta-blocking. We present evidence of enhanced performance in several critical aspects. By distributing the workload across multiple processes, we harness the computational power of multiple cores, resulting in improved efficiency. The essence of this approach lies in the strategic division of tasks and the orchestration of parallel processes, which allows us to exploit parallelism bypassing the GIL barrier.

As we will see in Section 4 we achieve different efficiency for each step of the entity resolution process. We manage to reach an almost linear speed up for the heaviest step of the algorithm. However, we also delve into the intricacies of certain deadlocks that emerged due to the implementation of the fork-join model. Communication between processes introduces another aspect to the parallel implementation: serialization and deserialization. We have many cases where this task requires more time than the actual processing of the algorithm leading to underperformance.

This thesis navigates the complex terrain of entity resolution, exploring the promise of parallelization within the PyJedAI framework while candidly addressing the obstacles encountered in our pursuit of efficiency. The ensuing chapters provide a comprehensive journey through our methodology, experiments, and findings, shedding light on both the accomplishments and the challenges in the field of entity resolution at scale.

2. PROBLEM STATEMENT

In the era of big data, the task of ER, also known as record linkage or deduplication [6], holds a pivotal role in maintaining data quality, enabling effective data analysis, and supporting a myriad of applications across various domains. ER involves identifying and linking records that correspond to the same real-world entities, despite inconsistencies, variations, and errors in the data. While conceptually straightforward, the computational demands of ER are considerable, especially when dealing with large datasets [12].

Traditional ER techniques face a significant challenge due to their inherent quadratic complexity, which makes them impractical for processing massive volumes of data. To overcome this challenge, practitioners often employ blocking techniques [5] [22], which group similar entities into overlapping blocks, reducing the number of necessary pairwise comparisons. Further efficiency gains are achieved through Meta-blocking [25] [27] [28], which prunes redundant comparisons in overlapping blocks, resulting in significantly improved precision without sacrificing recall [7] [13] [29].

However, despite the compelling advantages of Meta-blocking in enhancing time efficiency, applying these techniques to solve ER problems on very large datasets remains a formidable challenge [11]. For instance, processing millions of entities can be a time-consuming endeavor even on high-end servers, limiting the applicability of ER to a broader spectrum of applications [24].

An open-source library that leverages Python's data science ecosystem to build powerful end-to-end ER workflows is PyJedAI. A large variety of methods is also implemented by JedAI [26]. However, JedAI, like most Link Discovery tools, constitutes an isolated system, implemented in Java, which cannot be easily extended with existing state-of-the-art techniques from other domains, like Deep Learning and Natural Language Processing (NLP). To address this issue, pyJedAI was developed, a new open-source system that implements the same methods as JedAI, but is capable of combining them with any package from Python's data science ecosystem [21]. In this thesis, we focus on pyJedAI, due to its capabilities and the widest coverage of the literature.

Moreover, the adoption of the PyJedAI framework for ER introduces additional complexities due to the presence of Python's Global Interpreter Lock (GIL), which hampers straightforward multithreading [4] [14] [15] [19]. To tackle these challenges effectively, there is a growing need for innovative strategies that can parallelize PyJedAI while addressing the limitations imposed by the GIL and harnessing the power of multi-core and distributed computing.

The problem at hand, therefore, revolves around the development of scalable parallelization strategies for PyJedAI that can enable efficient ER on large datasets while mitigating the adverse effects of the GIL. The successful resolution of this problem promises to revolutionize the field of ER, making it more accessible, efficient, and practical for a diverse range of applications, from data integration and cleansing to information retrieval and knowledge discovery.

This thesis endeavors to explore these challenges and pave the way for the development of innovative solutions that enhance the efficiency and scalability of ER through the parallelization of the PyJedAI framework.

3. PRELIMINARIES

In this section, we lay the foundational groundwork for understanding the core concepts and the background that is essential to our research in parallelization and ER using the PyJedAI framework. We introduce the significance of ER in the context of handling large and heterogeneous datasets [7], setting the stage for our research. We also delve into the PyJedAI framework, emphasizing its role as a versatile and robust Python-based tool that extends the multiprocess python framework [20]. Additionally, we highlight the Python Global Interpreter Lock (GIL) and its implications for parallelization, and we explore the fork-join model with processes, focusing on its interaction with the GIL. These preliminary insights provide readers with a solid foundation for the subsequent in-depth discussions and findings in our research.

3.1 Parallel programming

Parallel programming is a computational paradigm designed to enhance performance by breaking down tasks into smaller, independent units that can be executed simultaneously across multiple processing units, such as CPU cores or distributed computing nodes [9]. The principles underlying parallel programming emphasize the efficient utilization of available resources to accomplish tasks faster and more effectively. These principles include:

- **Task decomposition**, which involves dividing a problem into smaller, manageable subtasks that can be executed concurrently
- **Data decomposition**, which focuses on distributing data across processing units to enable parallel processing
- **Synchronization**, which ensures proper coordination and communication among parallel tasks to maintain consistency and avoid conflicts

Metrics used to evaluate parallel programs encompass various aspects, including:

- **Speedup**, which measures the performance improvement achieved by parallel execution compared to sequential execution
- **Scalability**, which assesses how effectively a parallel program maintains performance as the problem size or number of processing units increases
- **Efficiency**, which evaluates the utilization of resources relative to the desired outcome, considering factors like overhead and communication costs

These principles and metrics form the foundation for designing, analyzing, and optimizing parallel programs across diverse computing architectures and application domains [33] [18] [30].

3.2 GIL (Global Interpreter Lock)

The Python Global Interpreter Lock (GIL) is a fundamental concept that influences the parallelization of tasks within Python [4]. GIL is an intrinsic feature of the Python programming language. It is a mutex (or lock) that allows only one thread to execute in a Python process at a given time. This design ensures thread safety but can also introduce complexities when it comes to leveraging multi-threading for parallel computing.The presence of the GIL has significant implications for parallelization efforts within Python-based applications. While Python threads can offer concurrency, true parallel execution is impeded by the GIL, which restricts multiple threads from executing Python bytecodes in parallel [2] [32]. As a result, multi-threading may not fully harness the capabilities of multi-core processors, limiting the efficiency and performance of parallelization. In the context of Python and the GIL, achieving efficient parallelization becomes a more challenging task. Traditional multi-threading approaches may not yield the expected performance gains due to the limitations imposed by the GIL. Consequently, this limitation prompts researchers and developers to explore alternative strategies for parallel execution in Python. In response to the challenges posed by the GIL, researchers and practitioners have explored alternative parallelization strategies, including multi-processing, asynchronous programming, and shared memory approaches. These strategies aim to work around the GIL's constraints and maximize the utilization of multi-core processors for improved parallel computing [2] [4] [16] [17].

3.3 Fork-Join Model of MPIRE

The fork-join model is a parallel programming paradigm that divides a task into multiple independent processes, each of which executes concurrently. These processes are often used to leverage the capabilities of multi-core processors efficiently. Each process operates separately and communicates with others as needed to complete the overall task [30]. We can see in Figure 3.1 an simple example of the fork-join model where the master process forks the slave process and when a slave finishes, master acquires their results (join). Finally, **in a multi-process approach, each process has its own GIL**, which enables more effective parallelization in Python [20].

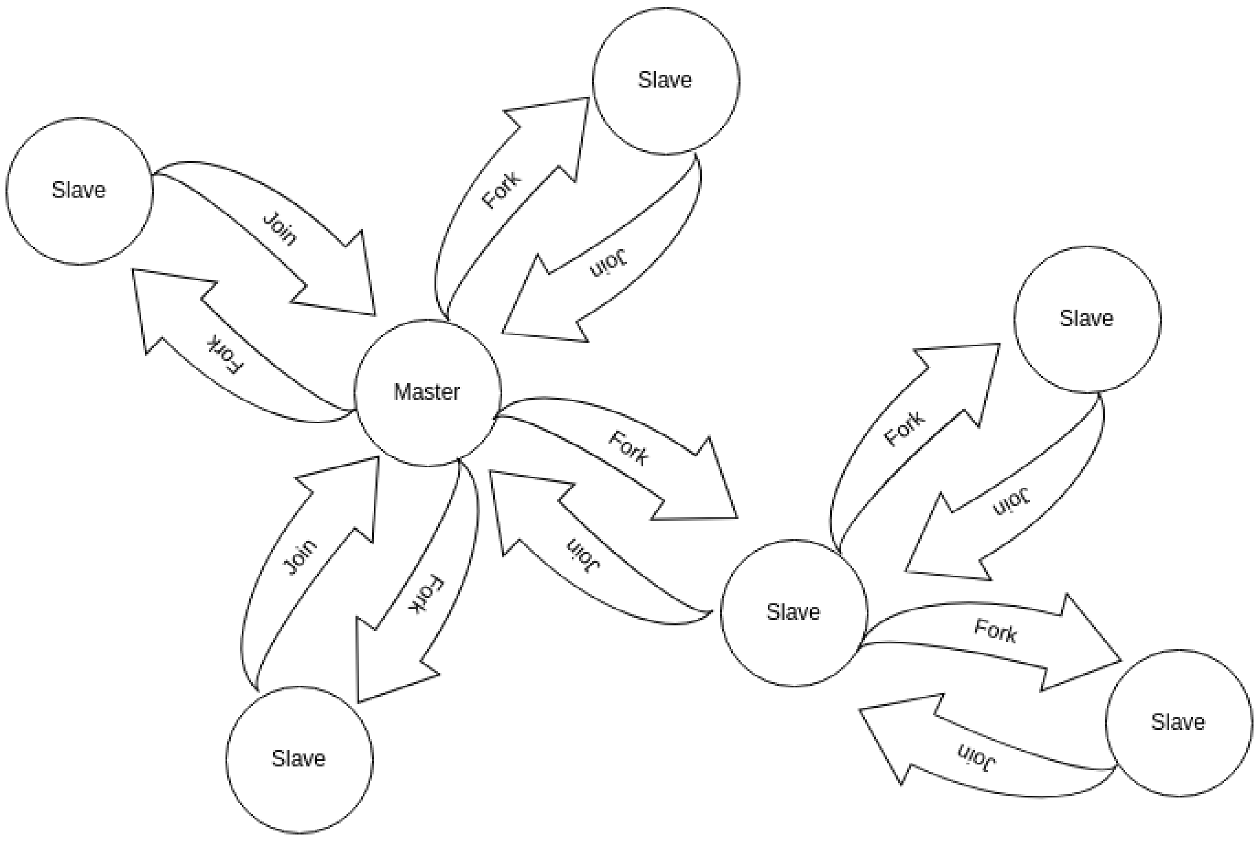

Figure 3.1: Generic Fork-Join Example.

3.4 PyJedAI Framework

PyJedAI is a dynamic Python framework engineered to provide robust and efficient solutions for a broad spectrum of ER challenges. It caters to both experienced practitioners and novices, offering state-of-the-art tools and features that streamline the ER process. PyJedAI is built on the foundation of cutting-edge Python frameworks, harnessing the latest advancements in data science, machine learning, deep learning, and natural language processing (NLP). These technologies are readily available within the Python data science ecosystem, ensuring that PyJedAI integrates the best tools for tackling ER tasks effectively [21]. For the parallel approach we based our implementation on version 0.1.2.

In Figure 3.2, we can see the ER pipeline diagram. When a rectangle, which represents a step, is bypassed by an arrow, it means the the step is optional. In summary, we have:

- **Block Building**: the dataset is divided into blocks, which are formed to group similar records together, aiming to reduce the number of comparisons needed.
- **Block Cleaning**: involves the removal of entire blocks that are dominated by repeated comparisons or comparisons between non-matching entities, aiming to improve the efficiency and accuracy of subsequent comparison processes. It ensures that only relevant records are considered for further analysis.
- **Comparison Cleaning**: operates at the level of individual records within each block, involving preprocessing such as standardizing formats, resolving inconsistencies, or handling missing data. Like block cleaning, its goal is to ensure that records are in a consistent and comparable format before similarity comparison.

- **Entity Matching**: it is the core process of comparing records within each block to identify potential matches. Similarity metrics or algorithms are applied to compare attributes of pairs of records. Records that exceed a certain similarity threshold are considered potential matches.
- **Clustering**: the matched records are grouped to form clusters of entities. This step organizes the matched records into groups that represent the same real-world entity.

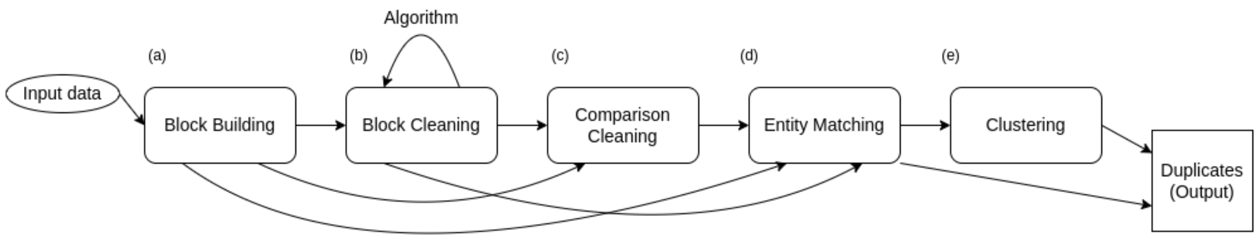

Figure 3.2: ER pipeline diagram
(a) First required step of Block Building (b) Optional step of Block Cleaning which may be recurring (c) Optional step of Comparison Cleaning (d) Required step of Entity Matching (e) Optional step of Clustering

We combine the ER pipeline depicted in Figure 3.2 with the the fork-join model. Generally, in each step of the pipeline we break the input data into N chunks and fork (create and execute) N processes. Each process takes as input a chunk and follows the procedure of the current algorithm of the step. The master process gathers and merges the results of each process. When all process have finished, the master process moves to the next ER step and so on.

4. PARALLELIZATION OF PYJEDAI WITH MPIRE

In this section, we present our efforts to parallelize the PyJedAI framework using the MPIRE library, implementing the existing algorithms introduced in [11], [23]. Our objective is to enhance the efficiency and scalability of PyJedAI's ER processes when handling large and complex datasets using the MPIRE framework.

4.1 MPIRE for Multiprocessing

4.1.1 General Structure

The implementation code has been designed with scalability in mind, offering an extensible architecture. This design allows future enhancements or additional features. The codebase is structured in a way that promotes ease of modification and integration, making it straightforward for developers to extend functionality and incorporate new components as needed.

First of all, the core code is structured in a way where each step of the ER algorithm is mapped to a class (aka a Python file). We follow the same logic in the parallel implementation, creating a class for each algorithm. For example, we have the class MultiprocessCEP, that extends the class MultiprocessComparisonCleaning, that extends the class AbstractMultiprocess.

The main methodology behind a multiprocess execution of a particular step of the algorithm is that we break the data into chunks and we generate new partial objects (e.g. in MultiprocessCEP we have many CardinalityEdgePruning objects) that are assigned to each chunk. This way we only need to do minor changes in the core implementation or re-write some parts of code when the parallel algorithm is different than the serial approach. One future-work task, that is not currently implemented, is load balancing, which can improve the overall performance by ensuring that each core has more or less the same workload.

4.1.2 SharedData

In each multiprocess class we define a SharedData object, which is passed to each process and contains read-only data that are not going to be modified. For example, this object can be used to share the dataset between the processes. This way we can reduce memory consumption and speed up the initialization step of a process as it takes some time to serialize/deserialize data in order to achieve process communication. If there is a need to use any of multiprocess oriented tools like a mutex, a queue, etc, we will use this way of sharing objects between processes as it is a restriction of the inner infrastructure of the multiprocess package. In the existing classes we use the SharedData objects when we can avoid making different copies of large data for each process, as for example the dataset or the set of blocks. [20]

If we want to have shared data between processes that are going to be modified we must use some lower level tools like shared_memory of the multiprocess package. This is actually considered as another future-work task to improve significantly the overall performance by limiting the serializations/deserializations needed.

4.2 Core methods

Each multiprocess class has 4 main methods:

1. The constructor.

   When a multiprocess object is initialized, it prepares the chunks of data that will be assigned to each process, while ensuring that our SharedData object, worker pool and the partial objects are properly set.

2. The init_class_object() method.

   It initializes the partial objects. This method is called within the constructor, and is the step where we copy the necessary information of the initial global object to the partial objects.

3. The generate_task_objects() method.

   It is the method that breaks the data into chunks and generates the necessary parameters for the processes we intend to fork.

4. The run() method.

   It is where the fork-join model is actually implemented. For some steps of the algorithm this method can be written in one line (e.g. MultiprocessEntityMatching), but for others that we need to perform some actions during the join phase, we must write some extra code. For example, in MultiprocessCEP, when a process is finished, we retrieve its top-k stack and we push each element in our global top-k stack, which is the final result.

4.3 Parallelization of ER steps

Each ER step of Figure 3.2 has different time and memory complexity. We will see that Block Building and Block Cleaning steps need far less amount of time to be computed in comparison to Comparison Cleaning and Entity Matching. Also, the Clustering step's time scale is in order of milliseconds even for large datasets, so we did not implement any parallel execution for this. Unfortunately, we noticed that in most cases we have underperformance in Block Building and Block Cleaning due to the latency in the joining phase.

4.3.1 Block Building

Block building is not a very time consuming task compared to Comparison Cleaning or Entity Matching, although for large datasets this time might need to be cut down. For this ER step we split the dataset to N parts where N is the number of processes that will be executed. We assign the indices slice of the dataset to each process to create a partial dataset for the process to work with. In order to expand the parallel execution of each different block building algorithm all we need to do is to override the init_class_object() method. The SharedData object we use for this one, contains the dataset.

In the join phase we take the block dictionary produced by each process and we merge it using the methods:

```
def merge_dicts(dict1, dict2, is_dirty_er):
    dict1_keys = dict1.keys()
    dict2_keys = dict2.keys()

    for key in dict2_keys:
        if key in dict1_keys:
            dict1[key].concat(dict2[key], is_dirty_er)
        else:
            dict1[key] = dict2[key]

# Block method
def concat(self, other_block: 'Block', is_dirty_er) -> None:
    self.entities_D1.update(other_block.entities_D1)
    if is_dirty_er:
        return
    self.entities_D2.update(other_block.entities_D2)
```

We observed that there is latency when a process finishes its job and it is very significant compared to the actual computation time. This occurs because serializing/deserializing the blocks that are calculated within each process takes more time than the block building processing.

4.3.2 Block Cleaning

Block cleaning is an even lighter task than block building, nevertheless, we implemented the parallel approach. For this one we have two algorithms: Block Filtering (BF) and Block Purging (BP). The SharedData object for both of them contains the block dictionary that the Block Building step previously produced. Similarly with Block Building, we divide the block dictionary to N chunks, one for each process. BP is very simple, all we need to do is to override the init_class_object() method and each process will filter out entire blocks whose cardinality exceeds a certain threshold. When it comes to BF it is little bit more complicated: we need to follow the procedure as it is explained in [11]. Each process iterates over its assigned blocks, and for each entity, it filters out records that do not occur frequently enough in blocks, based on a ratio. The master process, must collect and merge the filtered blocks and then it removes blocks that contain only one entity.

So the run method for BF is:

```
def run(self):
  self.blocks = {}
  for res in self.pool.imap_unordered(self.apply_processing, self.param):
    for key, value in res.items():
      self.blocks.setdefault(key, Block())
                 .concat(value, self.data.is_dirty_er)

  new_blocks = drop_single_entity_blocks(
                 self.blocks, self.data.is_dirty_er)
  self.blocks = new_blocks
```

while in BP master process just collects and merges the blocks returns by each child.

Again, we observed that there is high latency in the joining phase which leads to under-performance.

4.3.3 Comparison Cleaning

We implemented two algorithms for the parallel Comparison Cleaning: Cardinality Edge Pruning (CEP) and Weighted Edge Pruning (WEP). The SharedData object for CEP contains the initial Comparison Cleaning class, which includes all the data and as well as parameters like threshold. The SharedData object for WEP contains whatever the CEP contains, plus 3 more objects that we will explain shortly. For both we override the methods generate_task_objects(), init_class_object() and run() methods. The key difference is that in CEP we need a Priority Queue for each process and a global one for the joining phase. As explained in [11], we store the top-K edges in a local priority queue and in the joining phase with pop the items into the global priority queue.

So, the run() method for CEP is:

```
def run(self):
  threshold = self.shared_data.main_object._threshold
  top_k_edges = PriorityQueue(threshold * 2)
  minimum_weight = self.shared_data.main_object._minimum_weight
  for res in self.pool.imap_unordered(self.apply_processing, self.param):
    for comparison in res:
      weight = comparison[0]
      if weight >= minimum_weight:
        top_k_edges.put((weight, comparison[1], comparison[2]))
        if threshold < top_k_edges.qsize():
          minimum_weight = top_k_edges.get()[0]
      else:
        break

  self.blocks = defaultdict(set)
  while not top_k_edges.empty():
      comparison = top_k_edges.get()
      self.blocks[comparison[1]].add(comparison[2])
```

When it comes to WEP, we have a simple join phase where we just update the block dictionary with the block dictionary - result of each process. As mentioned in [11], we must calculate the global threshold first in order the proceed to the pruning. This is achieved with the 3 objects included in the SharedData object and are used as following in the method executed by each child process:

```
shared_data.weight += cc._threshold
shared_data.edges += cc._num_of_edges
shared_data.barrier.wait()
cc._threshold = shared_data.weight/shared_data.edges
```

while the entire method is:

```
def apply_processing(shared_data, pid, cc, entity_start, entity_end):

  if cc.data.is_dirty_er or cc._node_centric:
    cc._limit = cc.data.num_of_entities
  else :
    cc._limit = cc.data.dataset_limit

  cc._first_entity = entity_start
  cc._limit = entity_end
  cc._entity_index = shared_data.main_object._entity_index
  cc._num_of_blocks = len(shared_data.main_object._blocks)
  cc._blocks = shared_data.main_object._blocks

  cc._counters = np.empty([cc.data.num_of_entities], dtype=float)
  cc._flags = np.empty([cc.data.num_of_entities], dtype=int)
  if(cc._comparisons_per_entity_required()):
      cc._set_statistics()

  cc._num_of_edges = 0.0
  cc._threshold = 0.0

  for i in range(cc._first_entity, cc._limit):
      cc._process_entity(i)
      cc._update_threshold(i)

  shared_data.values["weight"].value += cc._threshold
  shared_data.values["edges"].value += cc._num_of_edges
  shared_data.barrier.wait()

  weight = shared_data.values["weight"].value
  edges = shared_data.values["edges"].value
  cc._threshold = weight / edges

  return cc._prune_edges()
```

In this ER step, we have reduced the overall time but there is also significant latency in the joining phase because of the serialization/deserialization of the returned results.

4.3.4 Entity Matching

The parallel Entity Matching is quite straight forward. Each core gets some blocks to work on. It iterates over the canditate pairs in these blocks. For every pair <pi, pj>, it adds two nodes ei, ej on a graph G. It also connects them with an edge <ei, ej>. The weight of the edge depends on how well the entities match. This graph is simple, meaning there's only one edge between any two nodes. The SharedData object contains the blocks and in the joining phase we merge the graphs returned from each process. The merge is done using the method:

```
def merge_graphs(self, other_pairs):
    self.pairs.add_nodes_from(other_pairs.nodes())
    self.pairs.add_edges_from(other_pairs.edges())
```

This is a latency-free step. In contrary with the other ER steps, the result of the Entity Matching is a Graph, which programatically needs very little memory space compared to the dictionaries of Blocks that the other ER steps return. Of course, this leads to easier serialization/deserialization, so less latency.

5. EVALUATION

The evaluation of our parallelization efforts for entity resolution in PyJedAI has provided valuable insights into the project's outcomes. Our parallelization strategy excels in entity matching, but in other steps, the serial approach continues to exhibit superior performance. We run the experiments with up to 16 processes and we had at our disposal 128Gb of memory.

5.1 Experimental Setup

5.1.1 CPU Configuration

The experiments were conducted on a server with a total of 16 processors. Each processor belongs to the Intel Xeon E5-4603 v2 family. Each individual processor has the following characteristics (content of /proc/cpuinfo):

- **CPU Family:** 6
- **Model:** 62
- **Stepping:** 4
- **Microcode:** 0x42e
- **CPU MHz:** 1200.601
- **Cache Size:** 10240 KB
- **Physical ID:** 0
- **Siblings:** 8
- **CPU Cores:** 4
- **APIC ID:** 0
- **Initial APIC ID:** 0
- **Flags:** fpu, vme, de, pse, tsc, msr, pae, mce, cx8, apic, sep, mtrr, pge, mca, cmov, pat, pse36, clflush, dts, acpi, mmx, fxsr, sse, sse2, ss, ht, tm, pbe, syscall, nx, pdpe1gb, rdtscp, lm, constant_tsc, arch_perfmon, pebs, bts, rep_good, nopl, xtopology, nonstop_tsc, cpuid, aperfmperf, pni, pclmulqdq, dtes64, monitor, ds_cpl, vmx, smx, est, tm2, ssse3, cx16, xtpr, pdcm, pcid, dca, sse4_1, sse4_2, x2apic, popcnt, tsc_deadline_timer, aes, xsave, avx, f16c, rdrand, lahf_lm, cpuid_fault, pti, ssbd, ibrs, ibpb, stibp, tpr_shadow, vnmi, flexpriority, ept, vpid, fsgsbase, smep, erms, xsaveopt, dtherm, arat, pln, pts, md_clear, flush_l1d
- **Bogomips:** 4400.41
- **clflush size:** 64
- **cache_alignment:** 64
- **address sizes:** 46 bits physical, 48 bits virtual

5.1.2 Memory Configuration

The setup of the server has the following memory configuration (content of /proc/meminfo):

- **Total Memory (MemTotal):** 131,966,744 kB (126.08 GB)
- **Available Memory (MemAvailable):** 110,837,680 kB (105.65 GB)
- **Buffers:** 1,671,224 kB (1.59 GB)
- **Cached:** 5,150,164 kB (4.91 GB)
- **Swap Total:** 8,388,604 kB (7.99 GB)
- **Swap Free:** 446,756 kB (0.43 GB)
- **Dirty:** 456 kB
- **Writeback:** 17,038,592 kB (16.26 GB)
- **Anon Pages:** 15,928,924 kB (15.20 GB)
- **Mapped:** 1,677,000 kB (1.60 GB)
- **Shmem:** 2,230,172 kB (2.13 GB)
- **Slab:** 1,544,536 kB (1.47 GB)
- **SReclaimable:** 949,568 kB (0.91 GB)
- **SUnreclaim:** 594,968 kB (0.57 GB)
- **Kernel Stack:** 22,944 kB (0.02 GB)
- **Page Tables:** 589,420 kB (0.56 GB)
- **Commit Limit:** 74,371,976 kB (70.85 GB)
- **Committed AS:** 353,551,364 kB (337.12 GB)
- **Vmalloc Total:** 34,359,738,367 kB (32,750.00 GB)
- **Vmalloc Used:** 0 kB
- **Vmalloc Chunk:** 0 kB
- **Huge Pages Total:** 0
- **Huge Pages Size:** 2,048 kB
- **Direct Map 4k:** 33,037,120 kB (31.51 GB)
- **Direct Map 2M:** 101,134,336 kB (96.50 GB)
- **Direct Map 1G:** 2,097,152 kB (2.00 GB)

5.2 Performance Metrics

The metrics that we use to evaluate our parallel implementation are: efficiency, speed-up and memory consumption. Time metrics measure the extent to which our parallelization strategy has decreased the overall execution time compared to a serial execution, but also, increased in some other steps of the entity matching process. Efficiency is a metric that gauges the resource utilization of our parallel implementation. We assess the trade-offs between resource consumption and task completion. A similar measure is speed-up which is a fundamental indicator of how effectively our parallelization strategy utilizes multiple processing units. We measure the speed-up achieved and analyze its variations as the dataset size and complexity change.

For Speedup S, we have:

$$S = T_{serial}/T_{parallel} \tag{5.1}$$

For Efficiency E, we have:

$$E = S/P \tag{5.2}$$

where P is the number of cores used.

Finally, memory usage is a crucial consideration in parallel computing as it monitors the second important resource of computing system, the memory and helps us analyze possible trade-offs.

5.3 Results and Plots

In the following tables, we see the results for each step of the algorithm for different datasets. In total, we used 8 Clean-Clean datasets (CC) and 2 Dirty (10k and 100k entities).

Table 5.1: Statistics of the CC datasets

Dataset	Name 1	Name2	Ent. 1	Ent. 2	Attr. 1	Attr. 2	N-V Pairs 1	N-V Pairs 2	Cart. Prod.
1	rest1	rest2	339	2256	4	4	1356	9024	764784
2	abt	buy	1076	1076	4	4	4304	4304	1157776
3	amazon	gp	1354	3039	5	5	6770	15195	4114806
4	acm	dblp	2294	2616	5	5	11470	13080	6001104
5	imdb	tmdb	5118	6056	12	15	61416	90840	30994608
6	imdb	tvdb	5118	7810	12	8	61416	62480	39971580
7	tmdb	tvdb	6056	7810	15	8	90840	62480	47297360
8	amazon	walmart	22074	2554	7	7	154518	17878	56376996

Table 5.2: Statistics of the dirty datasets

Dataset	Entities	Attributes	N-V Pairs	Cart. Prod.
10Kfull	10000	5	50000	100000000
100Kfull	100000	5	500000	10000000000

N-V: Name-Value

As mentioned before, the metrics for the evaluation are the execution time, Speed-up and Efficiency.

Table 5.3: CEP - CC 1

N	BB	BC	CC	EM	Total
1	5.77	2.71	91.11	131.57	231.2
2	11.36	4.21	76.08	82.39	174.07
4	11.69	4.38	53.0	45.17	114.28
8	12.59	4.86	41.05	25.34	83.86
16	11.71	5.79	38.21	17.42	73.15

Table 5.4: WEP - CC 1

N	BB	BC	CC	EM	Total
1	6.28	3.07	77.51	847.75	934.64
2	11.46	4.46	58.93	432.54	507.42
4	11.27	5.46	31.63	219.35	267.74
8	11.48	5.36	17.19	116.83	150.89
16	15.13	8.25	13.37	73.74	110.52

Table 5.5: CEP - CC 1 Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.51	0.64	1.2	1.6	1.33
4	0.49	0.62	1.72	2.91	2.02
8	0.46	0.56	2.22	5.19	2.76
16	0.49	0.47	2.38	7.55	3.16

Table 5.6: WEP - CC 1 Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.55	0.69	1.32	1.96	1.84
4	0.56	0.56	2.45	3.86	3.49
8	0.55	0.57	4.51	7.26	6.19
16	0.42	0.37	5.8	11.5	8.46

Table 5.7: CEP - CC 1 Effiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.25	0.32	0.6	0.8	0.66
4	0.12	0.15	0.43	0.73	0.51
8	0.06	0.07	0.28	0.65	0.34
16	0.03	0.03	0.15	0.47	0.2

Table 5.8: WEP - CC 1 Effiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.27	0.34	0.66	0.98	0.92
4	0.14	0.14	0.61	0.97	0.87
8	0.07	0.07	0.56	0.91	0.77
16	0.03	0.02	0.36	0.72	0.53

Table 5.9: CEP - CC 2

N	BB	BC	CC	EM	Total
1	3.25	2.04	28.41	69.4	103.11
2	5.75	4.38	26.37	53.64	90.15
4	5.33	5.49	19.61	31.01	61.45
8	7.69	6.85	17.18	18.22	49.94
16	7.76	7.53	18.25	12.19	45.73

Table 5.10: WEP - CC 2

N	BB	BC	CC	EM	Total
1	3.2	1.96	21.47	59.32	85.96
2	5.47	4.82	19.06	31.73	61.09
4	5.87	5.85	9.81	17.15	38.68
8	8.26	6.84	6.96	11.61	33.68
16	10.11	9.51	4.48	10.69	34.8

Table 5.11: CEP - CC 2 Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.57	0.47	1.08	1.29	1.14
4	0.61	0.37	1.45	2.24	1.68
8	0.42	0.3	1.65	3.81	2.06
16	0.42	0.27	1.56	5.69	2.25

Table 5.12: WEP - CC 2 Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.59	0.41	1.13	1.87	1.41
4	0.55	0.34	2.19	3.46	2.22
8	0.39	0.29	3.08	5.11	2.55
16	0.32	0.21	4.79	5.55	2.47

Table 5.13: CEP - CC 2 Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.28	0.23	0.54	0.65	0.57
4	0.15	0.09	0.36	0.56	0.42
8	0.05	0.04	0.21	0.48	0.26
16	0.03	0.02	0.1	0.36	0.14

Table 5.14: WEP - CC 2 Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.29	0.2	0.56	0.93	0.7
4	0.14	0.08	0.55	0.86	0.56
8	0.05	0.04	0.39	0.64	0.32
16	0.02	0.01	0.3	0.35	0.15

Table 5.15: CEP - CC 3

N	BB	BC	CC	EM	Total
1	1.62	0.73	13.36	49.86	65.58
2	2.65	1.71	13.47	31.19	49.03
4	3.65	2.31	9.47	16.45	31.89
8	3.62	3.56	9.69	10.05	26.93
16	5.19	4.37	10.22	6.51	26.29

Table 5.16: WEP - CC 3

N	BB	BC	CC	EM	Total
1	1.28	1.24	12.56	782.83	797.92
2	3.17	2.33	11.73	406.3	423.54
4	2.63	2.95	7.0	269.86	282.43
8	3.98	2.94	4.96	184.08	195.97
16	5.64	4.68	3.78	112.31	126.41

Table 5.17: CEP - CC 3 Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.61	0.43	0.99	1.6	1.34
4	0.44	0.32	1.41	3.03	2.06
8	0.45	0.21	1.38	4.96	2.44
16	0.31	0.17	1.31	7.66	2.49

Table 5.18: WEP - CC 3 Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.4	0.53	1.07	1.93	1.88
4	0.49	0.42	1.79	2.9	2.83
8	0.32	0.42	2.53	4.25	4.07
16	0.23	0.26	3.32	6.97	6.31

Table 5.19: CEP - CC 3 Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.31	0.21	0.5	0.8	0.67
4	0.11	0.08	0.35	0.76	0.51
8	0.06	0.03	0.17	0.62	0.3
16	0.02	0.01	0.08	0.48	0.16

Table 5.20: WEP - CC 3 Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.2	0.27	0.54	0.96	0.94
4	0.12	0.11	0.45	0.73	0.71
8	0.04	0.05	0.32	0.53	0.51
16	0.01	0.02	0.21	0.44	0.39

Table 5.21: CEP - Dirty 10k

N	BB	BC	CC	EM	Total
1	63.67	15.5	21.28	403.4	503.85
2	182.48	45.52	33.76	339.73	601.49
4	180.41	22.27	27.06	189.04	418.78
8	175.33	31.44	20.63	154.76	382.17
16	176.05	90.07	21.89	82.23	370.24

Table 5.22: WEP - Dirty 10k

N	BB	BC	CC	EM	Total
1	66.19	6.07	14.7	556.01	642.98
2	206.54	55.84	19.36	364.34	646.08
4	176.59	48.22	12.83	208.69	446.33
8	170.42	40.64	9.08	111.03	331.17
16	181.08	53.58	8.93	67.31	310.9

Table 5.23: CEP - Dirty 10k Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.35	0.34	0.63	1.19	0.84
4	0.35	0.7	0.79	2.13	1.2
8	0.36	0.49	1.03	2.61	1.32
16	0.36	0.17	0.97	4.91	1.36

Table 5.24: WEP - Dirty 10k Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.32	0.11	0.76	1.53	1.0
4	0.37	0.13	1.15	2.66	1.44
8	0.39	0.15	1.62	5.01	1.94
16	0.37	0.11	1.65	8.26	2.07

Table 5.25: CEP - Dirty 10k Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.17	0.17	0.32	0.59	0.42
4	0.09	0.17	0.2	0.53	0.3
8	0.05	0.06	0.13	0.33	0.16
16	0.02	0.01	0.06	0.31	0.09

Table 5.26: WEP - Dirty 10k Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.16	0.05	0.38	0.76	0.5
4	0.09	0.03	0.29	0.67	0.36
8	0.05	0.02	0.2	0.63	0.24
16	0.02	0.01	0.1	0.52	0.13

Table 5.27: CEP - 100k

N	BB	BC	CC	EM	Total
1	669.56	648.42	1012.23	18090.43	20420.64
2	667.34	764.89	1210.28	9512.34	12154.85
4	617.63	786.76	840.7	5741.97	7987.06

Table 5.28: CEP - 100k Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	1.0	0.85	0.84	1.9	1.68
4	1.08	0.82	1.2	3.15	2.56

Table 5.29: CEP - 100k Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.5	0.42	0.42	0.95	0.84
4	0.27	0.21	0.3	0.79	0.64

Table 5.30: WEP - 100k

N	BB	BC	CC	EM	Total
1	592.54	700.81	591.56	153.46	2038.38
2	600.65	595.77	502.86	99.13	1798.41
4	640.59	680.38	350.84	66.73	1738.54
8	574.04	750.01	257.44	54.58	1636.07

Table 5.31: WEP - 100k Speed-up

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.99	1.18	1.18	1.55	1.13
4	0.92	1.03	1.69	2.3	1.17
8	1.03	0.93	2.3	2.81	1.25

Table 5.32: WEP - 100k Efficiency

N	BB	BC	CC	EM	Total
1	1.0	1.0	1.0	1.0	1.0
2	0.49	0.59	0.59	0.77	0.57
4	0.23	0.26	0.42	0.57	0.29
8	0.13	0.12	0.29	0.35	0.16

In the plots below, the y-axis represents time while the x-axis denotes the number of cores utilized. There are 4 different lines:

1. The red dotted line: this one depicts the serial execution time.
2. The blue line, Exec Time: this one depicts the total execution time, counting from the very beginning of the step to the very ending.
3. The green line, (Time) With Latency: this one depicts the time, counting from the point the data of each process is initialized.
4. The orange line, Calc Time: it represents the maximum value of the time that each process consumed on doing its assigned task, plus the time required for merging the results. This visualization allows us to observe the optimal execution time if there were no latency from the interprocess communication or any other factors. However, in some cases, it might be misleading because the time for merging, that is included, is counted as the sum of the time of each merged result. The problem with this is that merging and computing is happening in parallel, as a result, we add up time that overlaps with the execution eitherway. So we may observe a worse time performance than the actual, but it ensures that it is not any worse than that.

5.3.1 Block Building

The first step of the ER algorithm is Blocking Building. Generally, this step is not very time consuming in comparison with Comparison Cleaning and Entity Matching. We can clearly see the underperformance in this step as the red line which is the serial execution time is below every other.

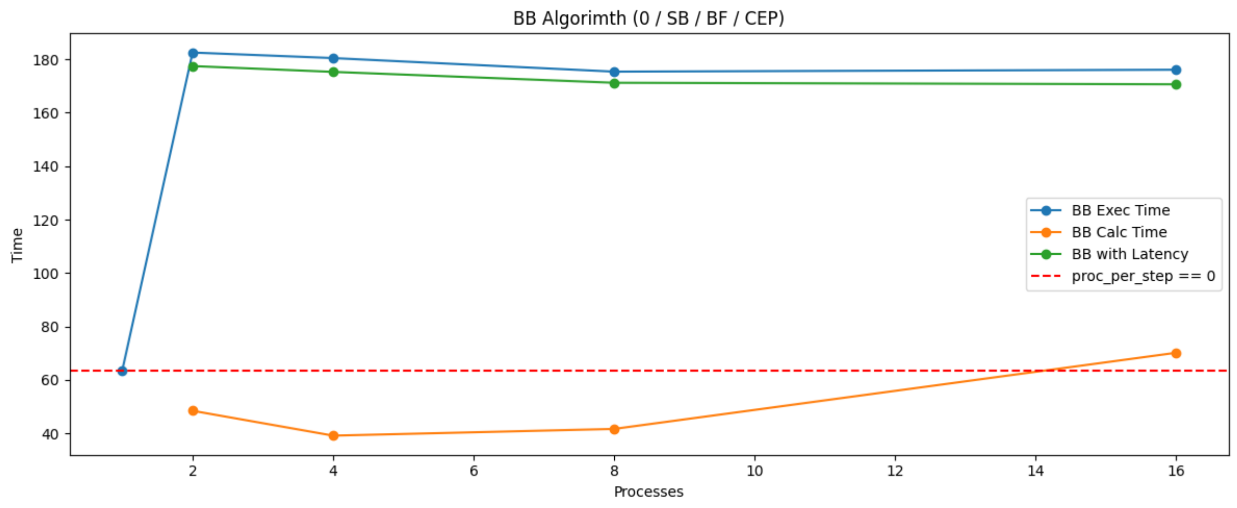

Figure 5.1: Block Building for dataset 8

5.3.2 Block Cleaning

The second step of the ER algorithm is Block Cleaning, which is even lighter, in terms of time complexity, than the first one. We can also see the underperformance as the efficiency is getting lower when the the number of processes increases.

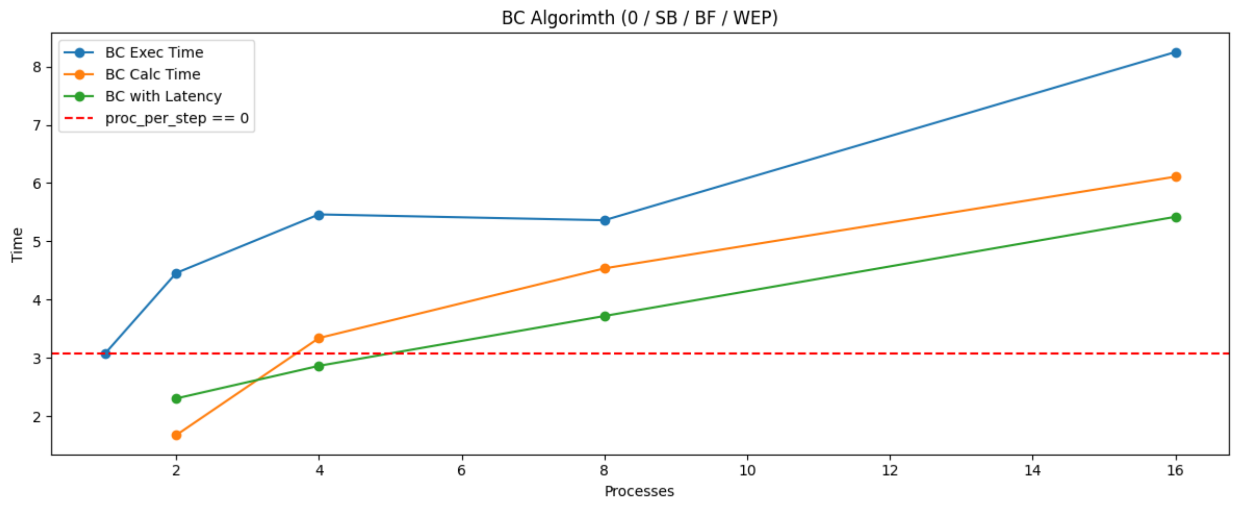

Figure 5.2: Block Cleaning for dataset 1

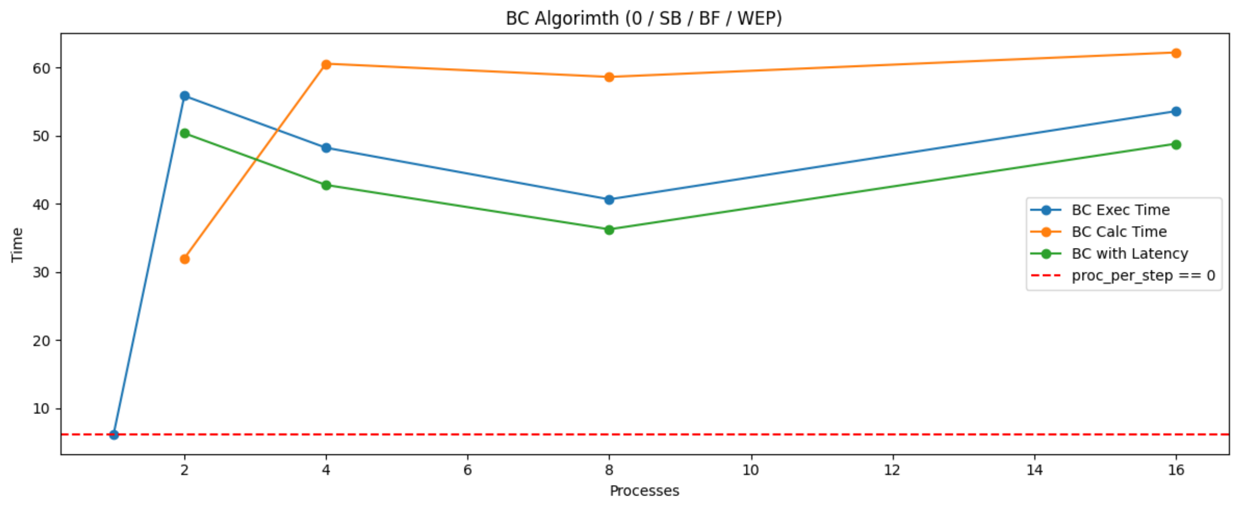

Figure 5.3: Block Cleaning for dataset 7

5.3.3 Comparison Cleaning

The third step of the ER algorithm is Comparison Cleaning where the pruning of entities is taking place. In this one, the efficiency is low, but in most cases there is no underperformance. Also, keep in mind that the load balancing algorithm is not implemented, so different data chunk sizes will distribute differently the data to each process which will lead to different performance.

The following plots are depicting the performance of the Weighted Edge Pruning (WEP) algorithm.

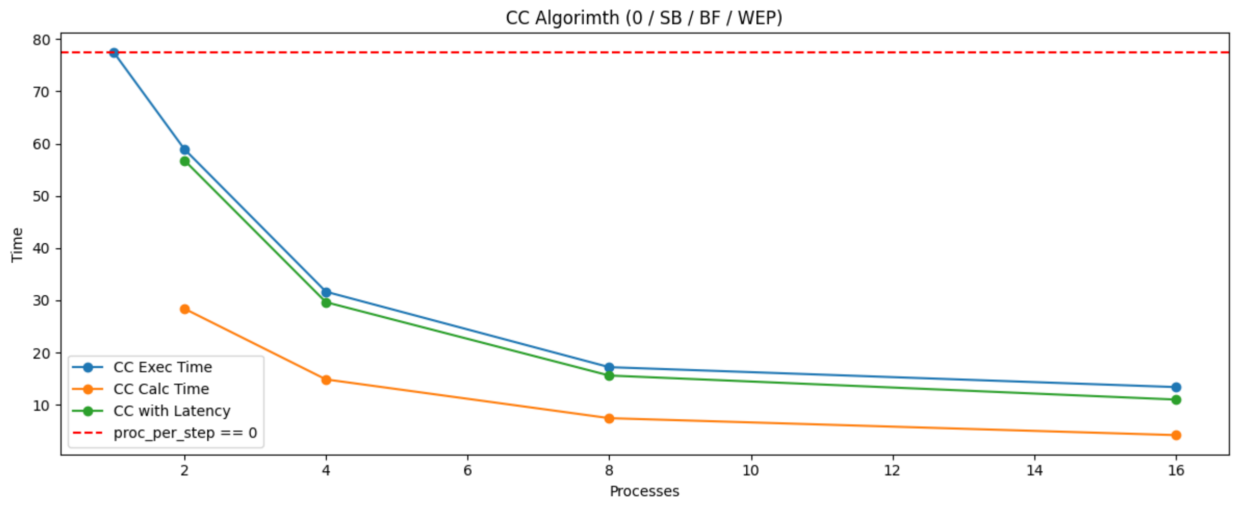

Figure 5.4: WEP for dataset 1

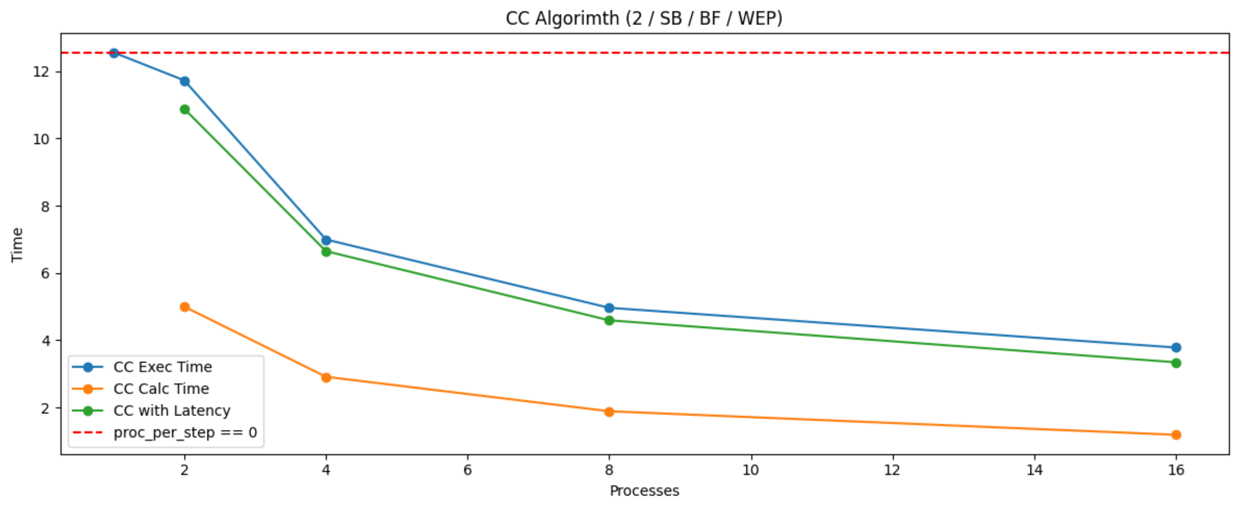

Figure 5.5: WEP for dataset 5

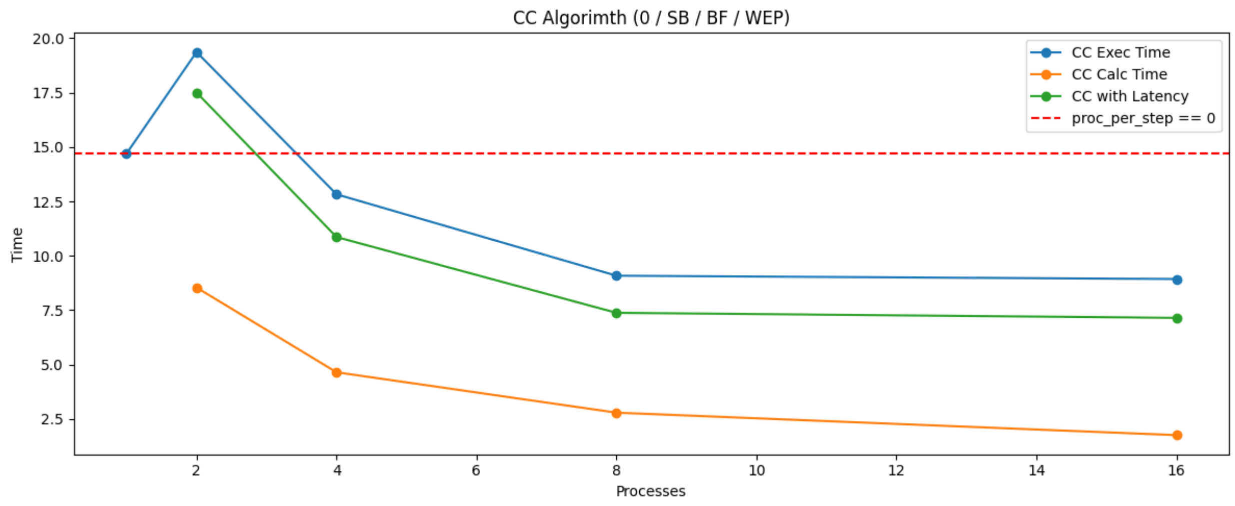

Figure 5.6: WEP for dataset 7

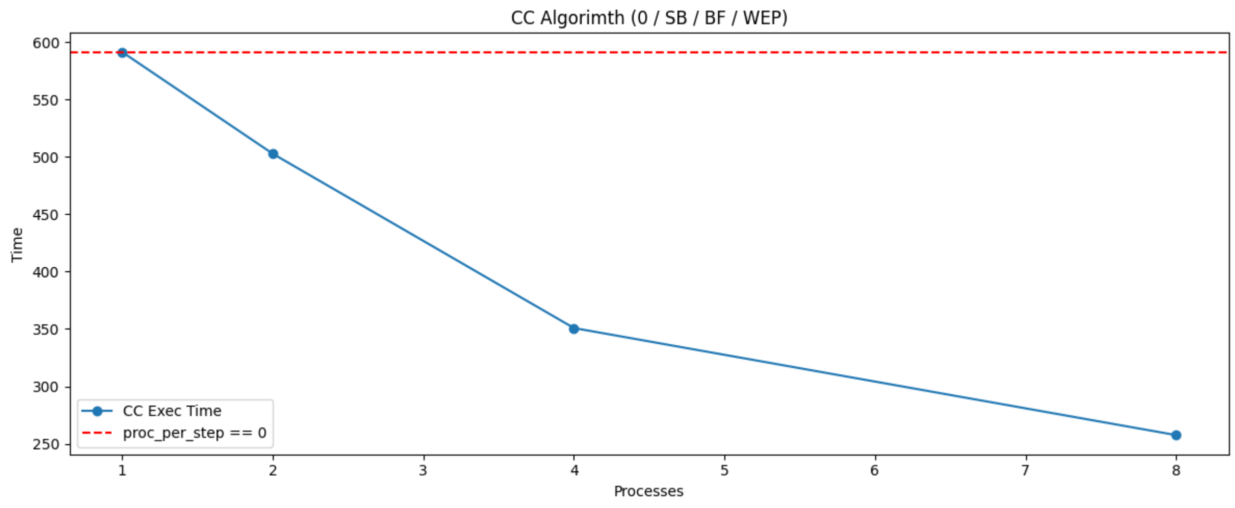

Figure 5.7: WEP for dataset 9 (Dirty 10k)

The following plots are depicting the performance of the Cardinality Edge Pruning (CEP) algorithm.

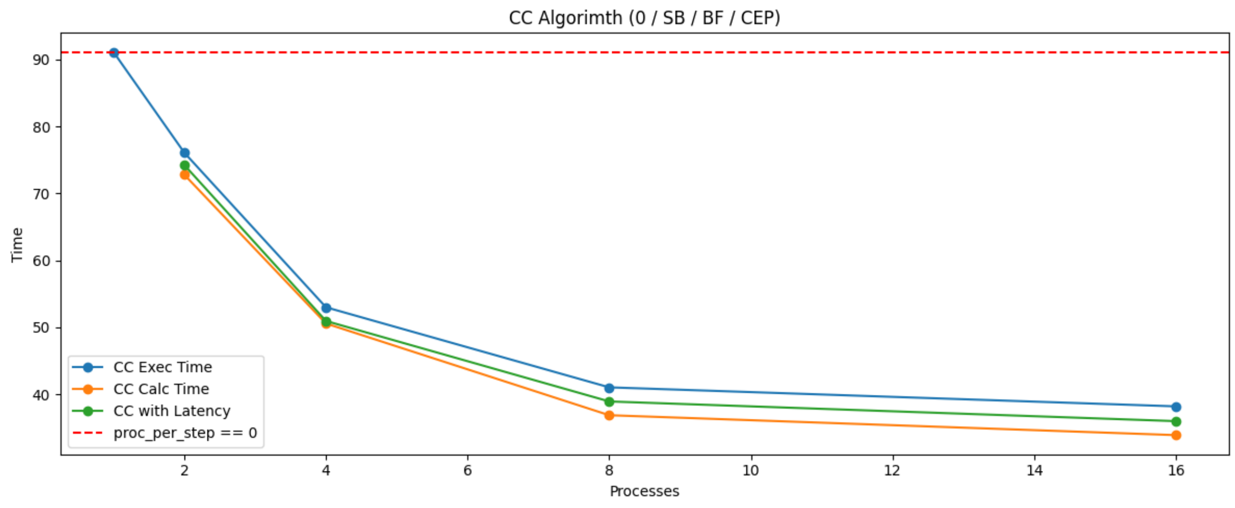

Figure 5.8: CEP for dataset 2

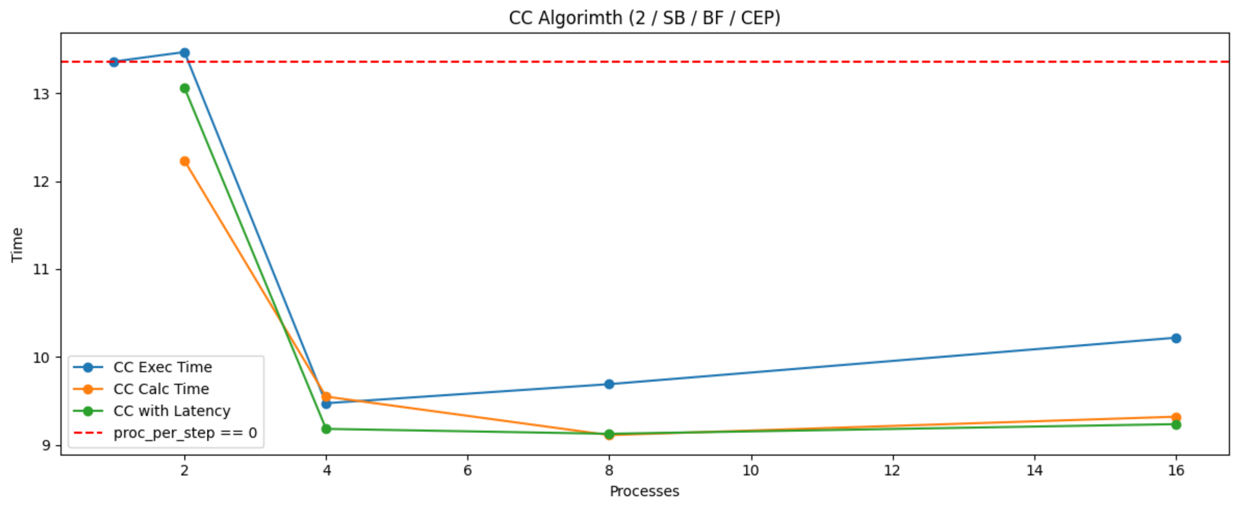

Figure 5.9: CEP for dataset 6

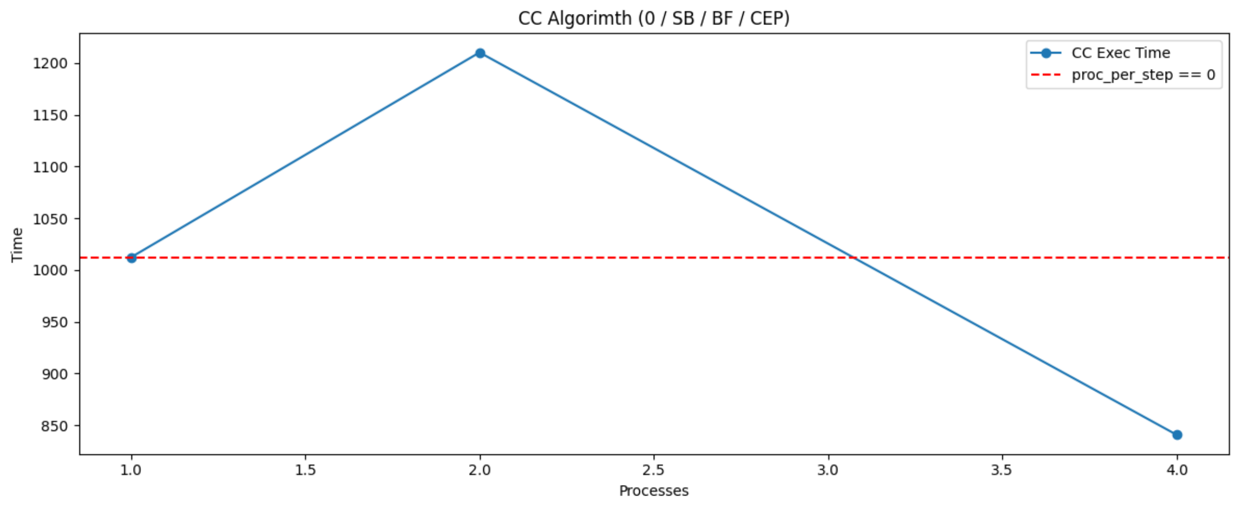

Figure 5.10: CEP for dataset 10 (Dirty 100k)

5.3.4 Entity Matching

The fourth step is Entity Matching, which is the computationally heaviest step. As we can see in the plots below, the 3 lines are overlapping with each other which means that latency due to the serialization is very low.

Now lets take a look at our greatest results.

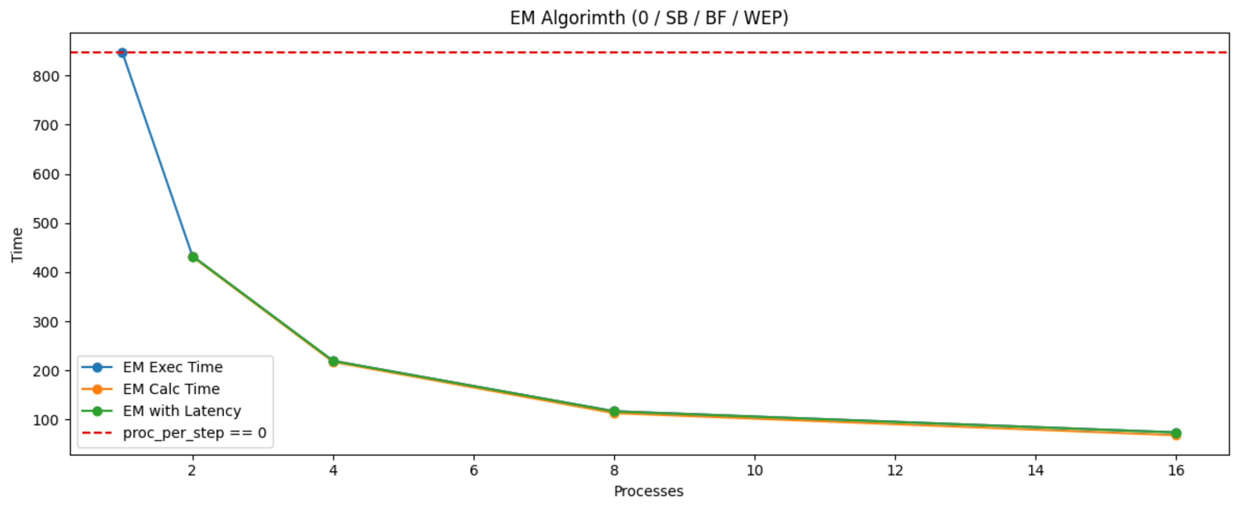

Figure 5.11: Entity Matching for dataset 1

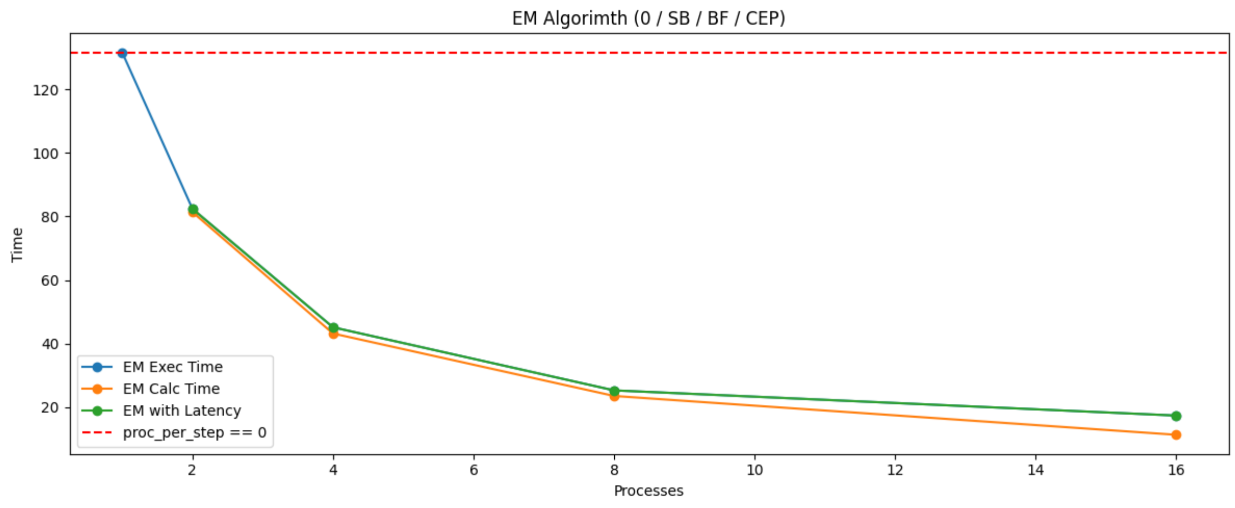

Figure 5.12: Entity Matching for dataset 2

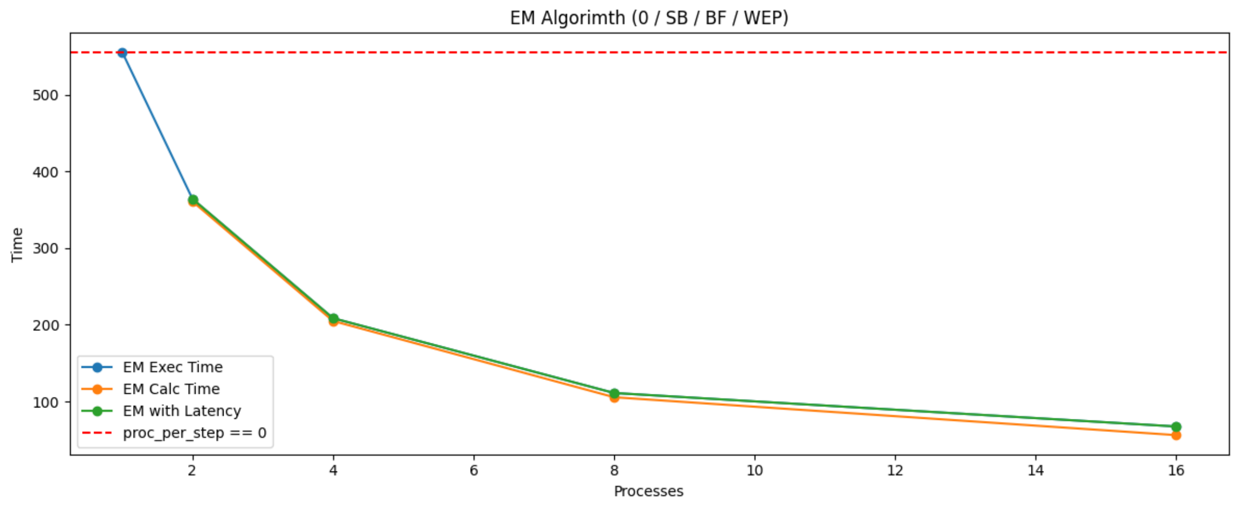

Figure 5.13: Entity Matching for dataset 7

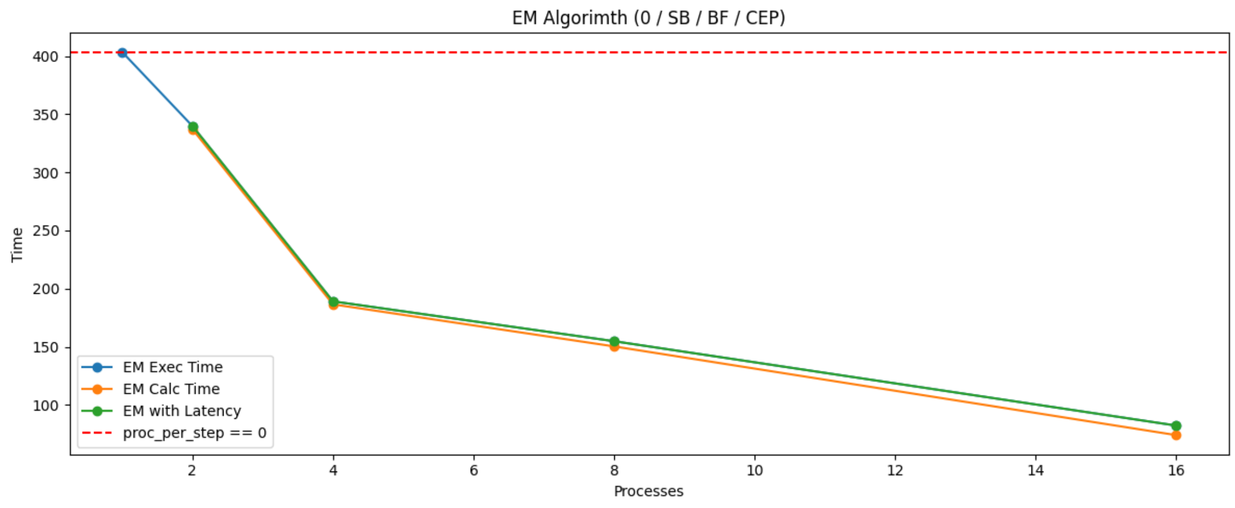

Figure 5.14: Entity Matching for dataset 8

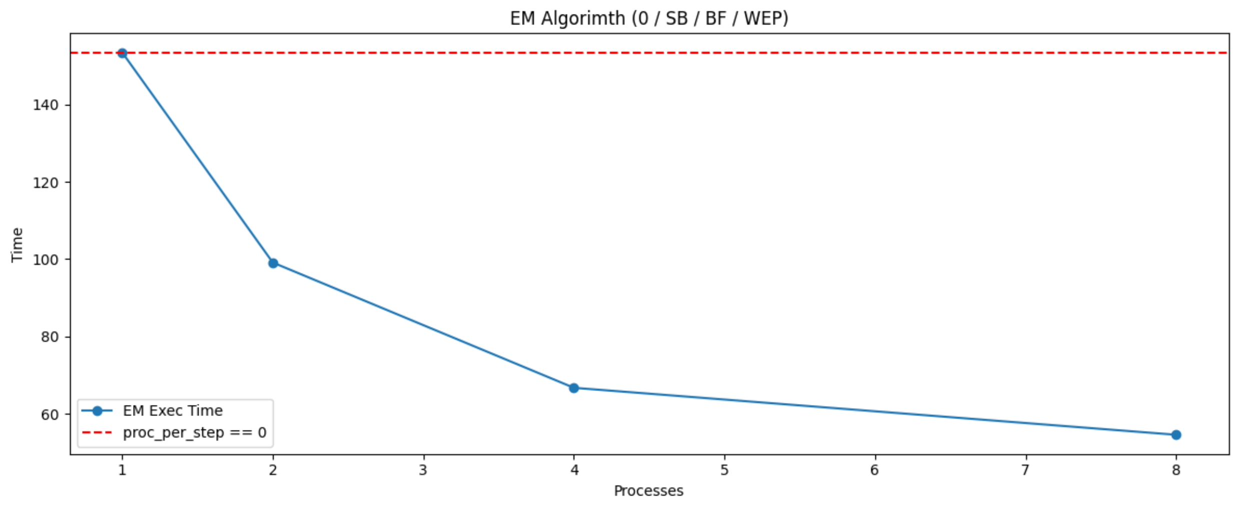

Figure 5.15: Entity Matching for dataset 9 (Dirty 10k)

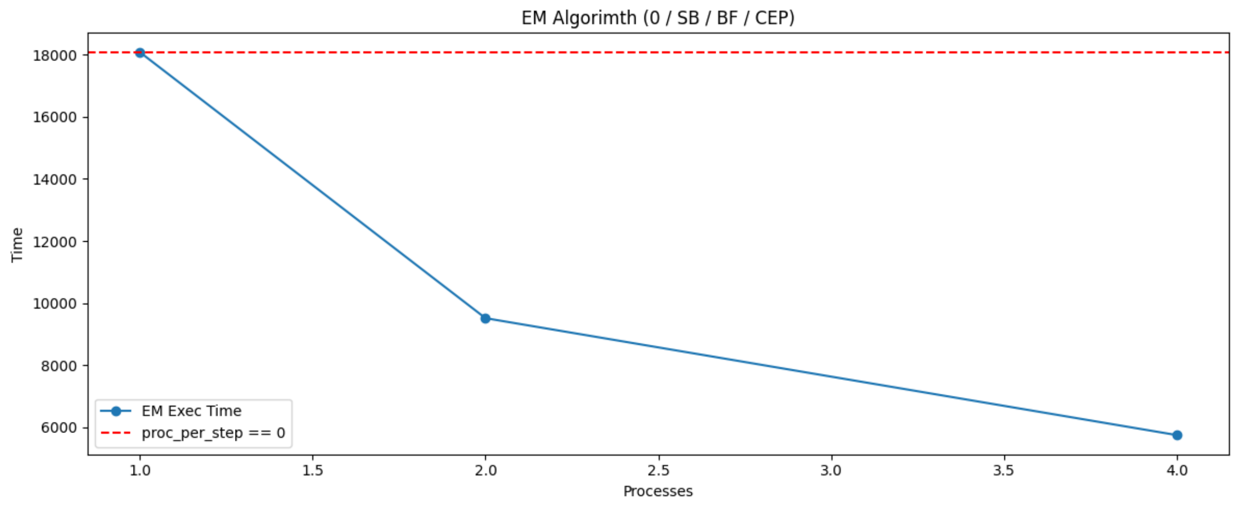

Figure 5.16: Entity Matching for dataset 10 (Dirty 100k)

5.4 Discussion of Findings

In conclusion, MPIRE is an optimization of the multiprocessing python library, plus it takes a step further the simplicity of its use, as it provides an implementation of higher level calls. In the official documentation of MPIRE [20], it is stated that in most scenarios MPIRE is faster than the multiprocessing library, Nevertheless, we cannot expect to reduce the time complexity of an algorithm with an optimization of the library we use, we must optimize our algorithm. In our case, interprocess communication, that is necessary for the parallel implementation in python, introduces a bottleneck to our algorithm. Using any python package for multiprocessing that requires this kind of communication and not a shared memory solution for example, would lead to the same bottleneck. So, the evaluation of our implementation using MPIRE is more or less the same as using any multiprocessing package.

As we can see, Entity Matching and Comparison Cleaning are the steps that bring in time reduction in the parallel implementation. These results would be even better if the load balancing algorithm of [11] was implemented too. For example, in Figure 5.42, it seems that for 2 processes the execution time is increased, but for 4 processes it is decreased, which means that this is not a bottleneck of the parallel algorithm but probably an inefficient data distribution. In such cases, most probably, there is poor load balancing. What happens, is that the time of the process with the most work sets the wall clock, delaying the entire processing. But, when it comes to poor load balancing, it means that there will be processes/threads that will finish too early, as a result being idle. Also, in most plots, like Figure 5.36, we can clearly see that the calculation time (the time for the longest job plus the total time needed for merging the results in the join phase) is much smaller than the execution time [1]. As we mentioned before, this is due to the serialization/deserialization of the data when two processes communicate.

6. CONCLUSIONS AND FUTURE WORK

In the course of our research into parallelization in Python, we have unearthed several crucial insights that shed light on the intricacies and challenges associated with harnessing parallel computing power within a high-level language like Python. These findings are instrumental in understanding the inherent complexities and exploring potential solutions:

Challenges of Parallelization in Python: It has become evident that parallelization in Python can be more intricate and demanding compared to lower-level languages. The presence of the GIL and other language-specific constraints can restrict the full potential of parallel computing, making it imperative to explore alternative strategies to maximize performance.

Latency in Data Communication: A significant contributor to latency in our parallel processing is the serialization and deserialization of data. The necessity to convert data into a transportable format for communication between processes can introduce overhead, leading to huge delays in the overall execution of parallel tasks.

Exploring Shared Memory Solutions: As we look toward addressing the issue of serialization-induced latency, one promising avenue involves the implementation of a shared memory approach. This strategy allows processes to access and manipulate data within a shared memory space, eliminating the need for resource-intensive serialization and deserialization. By enabling direct and efficient data exchange, shared memory solutions hold the potential to substantially enhance the speed and efficiency of parallelized tasks. [33] [31] [19] [3]

In conclusion, our research underscores the challenges posed by parallelization in a language like Python and highlights the role of data serialization in introducing latency within parallel processes. The pursuit of solutions, such as shared memory implementations, offers a promising pathway to mitigate these challenges, ultimately paving the way for more efficient and scalable parallelization in the context of Python-based applications like entity resolution. These insights are invaluable for researchers and practitioners aiming to harness the full potential of parallel computing within high-level programming languages.

ABBREVIATIONS - ACRONYMS

GIL	Global Interpreter Lock
WEP	Weighted Edge Pruning
CEP	Cardinality Edge Pruning
N-V	Name-Value

BIBLIOGRAPHY

[1] *Amdalh's Law Wikipedia*. URL: `https://en.wikipedia.org/wiki/Amdahl%27s_law`.

[2] Jim Anderson. "An Intro to Threading in Python – Real Python". In: *MArzo* (2019).

[3] Todd Anderson and Tim Mattson. "Multithreaded parallel Python through OpenMP support in Numba". In: 2021. DOI: `10.25080/majora-1b6fd038-012`.

[4] Zina A. Aziz, Diler Naseradeen Abdulqader, Amira Bibo Sallow and Herman Khalid Omer. "Python Parallel Processing and Multiprocessing: A Rivew". In: *Academic Journal of Nawroz University* 10 (3 Aug. 2021). ISSN: 2520-789X. DOI: `10.25007/ajnu.v10n3a1145`.

[5] Peter Christen. "A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication". In: *IEEE Transactions on Knowledge and Data Engineering* 24 (9 Sept. 2012), pp. 1537–1555. ISSN: 1041-4347. DOI: `10.1109/TKDE.2011.127`.

[6] Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, and Duplicate Detection*. Springer, 2012, pp. I–XIX, 1–270. ISBN: 978-3-642-31163-5.

[7] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis and Kostas Stefanidis. "An Overview of End-to-End Entity Resolution for Big Data". In: *ACM Comput.* 53.6 (2021), 127:1–127:42.

[8] Vassilis Christophides, Vasilis Efthymiou and Kostas Stefanidis. *Entity Resolution in the Web of Data*. Springer International Publishing, 2015. ISBN: 978-3-031-79467-4. DOI: `10.1007/978-3-031-79468-1`.

[9] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler and Alejandro Cosimo. "Parallel distributed computing using Python". In: *Advances in Water Resources* 34 (9 Sept. 2011). ISSN: 03091708. DOI: `10.1016/j.advwatres.2011.04.013`.

[10] Xin Luna Dong and Divesh Srivastava. *Synthesis Lectures on Data Management*. Morgan Claypool Publishers, 2015, pp. 1–198. ISBN: 978-3-031-00725-5.

[11] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis and Themis Palpanas. "Parallel meta-blocking for scaling entity resolution over big heterogeneous data". In: *Information Systems* 65 (Apr. 2017). ISSN: 03064379. DOI: `10.1016/j.is.2016.12.001`.

[12] Papadakis George, Ioannou Ekaterini and Palpanas Themis. "Entity Resolution: Past, Present and Yet-to-Come". In: *Proceedings of the EDBT/ICDT 2020 Joint Conference*. Copenhagen, Denmark, Mar. 2020.

[13] Papadakis George and Palpanas Themis. "Web-scale, Schema-Agnostic, End-to-End Entity Resolution". In: *Proceedings of the World Wide Web (WWW) – The Web Conference 2018*. Lyon, France, Apr. 2018.

[14] Kazuo Goda, Yuto Hayamizu, Hiroyuki Yamada and Masaru Kitsuregawa. "Out-of-order execution of database queries". In: *Proceedings of the VLDB Endowment* 13 (12 Aug. 2020), pp. 3489–3501. ISSN: 2150-8097. DOI: `10.14778/3415478.3415571`.

[15] Steve Kleiman, Devang Shah and Bart Smaalders. *Programming with threads*. Sun Soft Press Mountain View, 1996.

[16] Ami Marowka. “On parallel software engineering education using python”. In: *Education and Information Technologies* 23 (1 Jan. 2018). ISSN: 1360-2357. DOI: `10.1007/s10639-017-9607-0`.

[17] Stefano Masini and Paolo Bientinesi. “High-Performance Parallel Computations Using Python as High-Level Language”. In: 2011. DOI: `10.1007/978-3-642-21878-1_66`.

[18] Berna L. Massingill, Timothy G. Mattson and Beverly A. Sanders. “Parallel programming with a pattern language *”. In: *International Journal on Software Tools for Technology Transfer* 3 (2 2001). ISSN: 14332779. DOI: `10.1007/s100090100045`.

[19] Timothy G. Mattson, Todd A. Anderson and Giorgis Georgakoudis. “PyOMP: Multithreaded Parallel Programming in Python”. In: *Computing in Science Engineering* 23 (6 Nov. 2021), pp. 77–80. ISSN: 1521-9615. DOI: `10.1109/MCSE.2021.3128806`.

[20] *MPIRE Official Documentation*. URL: `https://sybrenjansen.github.io/mpire/index.html`.

[21] Konstantinos Nikoletos, George Papadakis and Manolis Koubarakis. “pyJedAI: a Lightsaber for Link Discovery”. In: *Proceedings of the ISWC 2022 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 21st International Semantic Web Conference (ISWC 2022), Virtual Conference, Hangzhou, China, October 23-27, 2022*.

[22] George Papadakis, George Alexiou, George Papastefanatos and Georgia Koutrika. “Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data”. In: *Proceedings of the VLDB Endowment* 9 (4 Dec. 2015), pp. 312–323. ISSN: 2150-8097. DOI: `10.14778/2856318.2856326`.

[23] George Papadakis, Konstantina Bereta, Themis Palpanas and Manolis Koubarakis. “Multi-core Meta-blocking for Big Linked Data”. In: ACM, Sept. 2017. ISBN: 9781450352963. DOI: `10.1145/3132218.3132230`.

[24] George Papadakis, Ekaterini Ioannou, Emanouil Thanos and Themis Palpanas. “Possible Directions for Future Work”. In: 2021, pp. 119–120. DOI: `10.1007/978-3-031-01878-7_9`.

[25] George Papadakis, Georgia Koutrika, Themis Palpanas and Wolfgang Nejdl. “Meta-Blocking: Taking Entity Resolutionto the Next Level”. In: *IEEE Transactions on Knowledge and Data Engineering* 26 (8 Aug. 2014), pp. 1946–1960. ISSN: 1041-4347. DOI: `10.1109/TKDE.2013.54`.

[26] George Papadakis, George Mandilaras, Luca Gagliardelli, Giovanni Simonini, Emmanouil Thanos, George Giannakopoulos, Sonia Bergamaschi, Themis Palpanas and Manolis Koubarakis. “Three-dimensional Entity Resolution with JedAI”. In: *Information Systems* 93 (Nov. 2020), p. 101565. ISSN: 03064379. DOI: `10.1016/j.is.2020.101565`.

[27] George Papadakis, George Papastefanatos, Themis Palpanas and Manolis Koubarakis. “Boosting the Efficiency of Large-Scale Entity Resolution with Enhanced Meta-Blocking”. In: *Big Data Research* 6 (Dec. 2016), pp. 43–63. ISSN: 22145796. DOI: `10.1016/j.bdr.2016.08.002`.

[28] George Papadakis, George Papastefanatos, Themis Palpanas and Manolis Koubarakis. “Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking”. In: vol. 2016-March. 2016. DOI: `10.5441/002/edbt.2016.22`.

[29] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos and Themis Palpanas. “Blocking and Filtering Techniques for Entity Resolution: A Survey”. In: *ACM Comput.* 53.2 (2021), 31:1–31:42.

[30] Thomas Rauber and Gudula Rünger. *Parallel Programming*. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-37800-3. DOI: `10.1007/978-3-642-37801-0`.

[31] Ariya Shajii, Ibrahim Numanagić, Alexander T. Leighton, Haley Greenyer, Saman Amarasinghe and Bonnie Berger. “A Python-based optimization framework for high-performance genomics”. In: *bioRxiv* (2020).

[32] L.M. SHavtikova and M.B. Tekeev. “Parallel computation of threads in the Python programming language”. In: *TRENDS IN THE DEVELOPMENT OF SCIENCE AND EDUCATION* (2020). DOI: `10.18411/lj-12-2020-46`.

[33] Giancarlo Zaccone. *Python Parallel Programming Cookbook*. 2015.