Parallélisation maximale automatique

MOULAI HACENE Rania
Terfi Mohammed Wassim
Zeggane Walid

Mars 2025

Sommaire

Objectifs du projet

- Générer automatiquement un graphe de précédence.
- Exécuter un système de tâches séquentiellement ou en parallèle.
- Tester le déterminisme du système.
- Comparer les performances des deux modes d'exécution.

Fonctionnalités principales

- Déclaration des tâches (`name`, `reads`, `writes`, `run`).
- Construction automatique des dépendances.
- Exécution séquentielle : `runSeq()`.
- Exécution parallèle : `run()`.
- Affichage du graphe : `draw()`.

Exécution séquentielle

```python
def runSeq(self):

    G = nx.DiGraph()
    for task_name in self.tasks:
        G.add_node(task_name)

    for task, deps in self.max_parallelism.items():
        for dep in deps:
            G.add_edge(dep, task)

    execution_order = list(nx.topological_sort(G))

    for task_name in execution_order:
        task = self.tasks[task_name]
        if task.run:
            task.run()

    return True
```

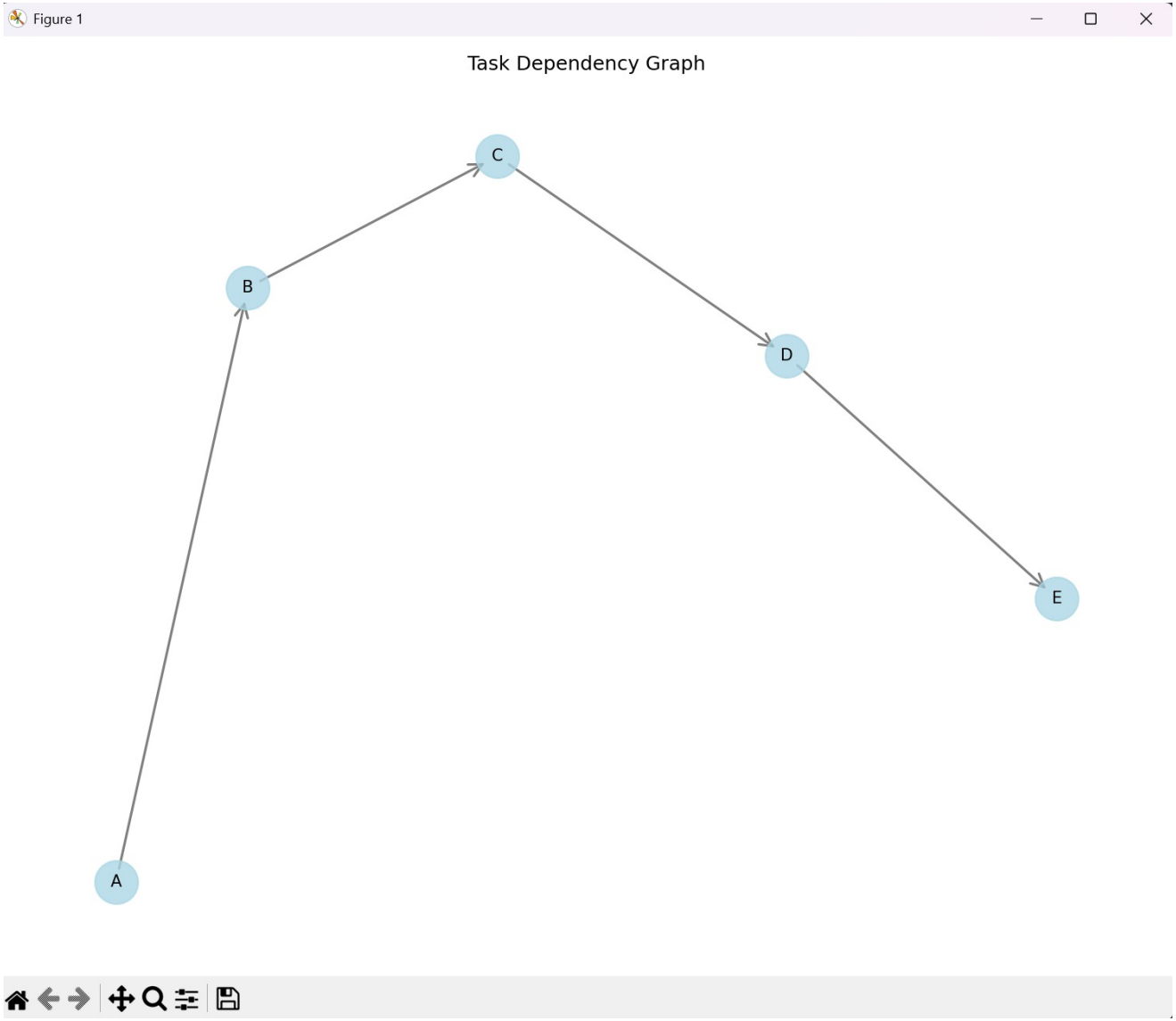
Exécution parallèle

```python
def run(self):
    """
    Exécute les tâches en parallèle tout en respectant les dépendances.
    """
    # Construit le graphe de dépendances
    G = nx.DiGraph()
    for task_name in self.tasks:
        G.add_node(task_name)

    for task, deps in self.max_parallelism.items():
        for dep in deps:
            G.add_edge(dep, task)

    # Ensemble des tâches complétées et verrou pour l'accès concurrent
    completed = set()
    completed_lock = threading.Lock()

    def execute_task(task_name):
        """Fonction pour exécuter une tâche et marquer comme complétée"""
        task = self.tasks[task_name]
        if task.run:
            task.run()
        # Ajout sécurisé à l'ensemble des tâches complétées
        with completed_lock:
            completed.add(task_name)

    # Utilise un pool de threads pour exécuter les tâches en parallèle
    with ThreadPoolExecutor() as executor:
        # Continue jusqu'à ce que toutes les tâches soient complétées
        while len(completed) < len(self.tasks):
            # Trouve les tâches prêtes à être exécutées
            ready_tasks = []
            for task_name in self.tasks:
                if task_name not in completed:
                    deps = self.max_parallelism[task_name]
                    # Une tâche est prête si toutes ses dépendances sont complétées
                    if all(dep in completed for dep in deps):
                        ready_tasks.append(task_name)

            if ready_tasks:
                # Exécute toutes les tâches prêtes en parallèle
                futures = [executor.submit(execute_task, task_name) for task_name in ready_tasks]
                for future in futures:
                    future.result()  # Attend que toutes les tâches soient terminées
            else:
                # S'il n'y a pas de tâches prêtes mais certaines ne sont pas encore complétées,
                # c'est un deadlock
                if len(completed) < len(self.tasks):
                    raise RuntimeError("Deadlock detected: no ready tasks but not all tasks completed")

    return True
```

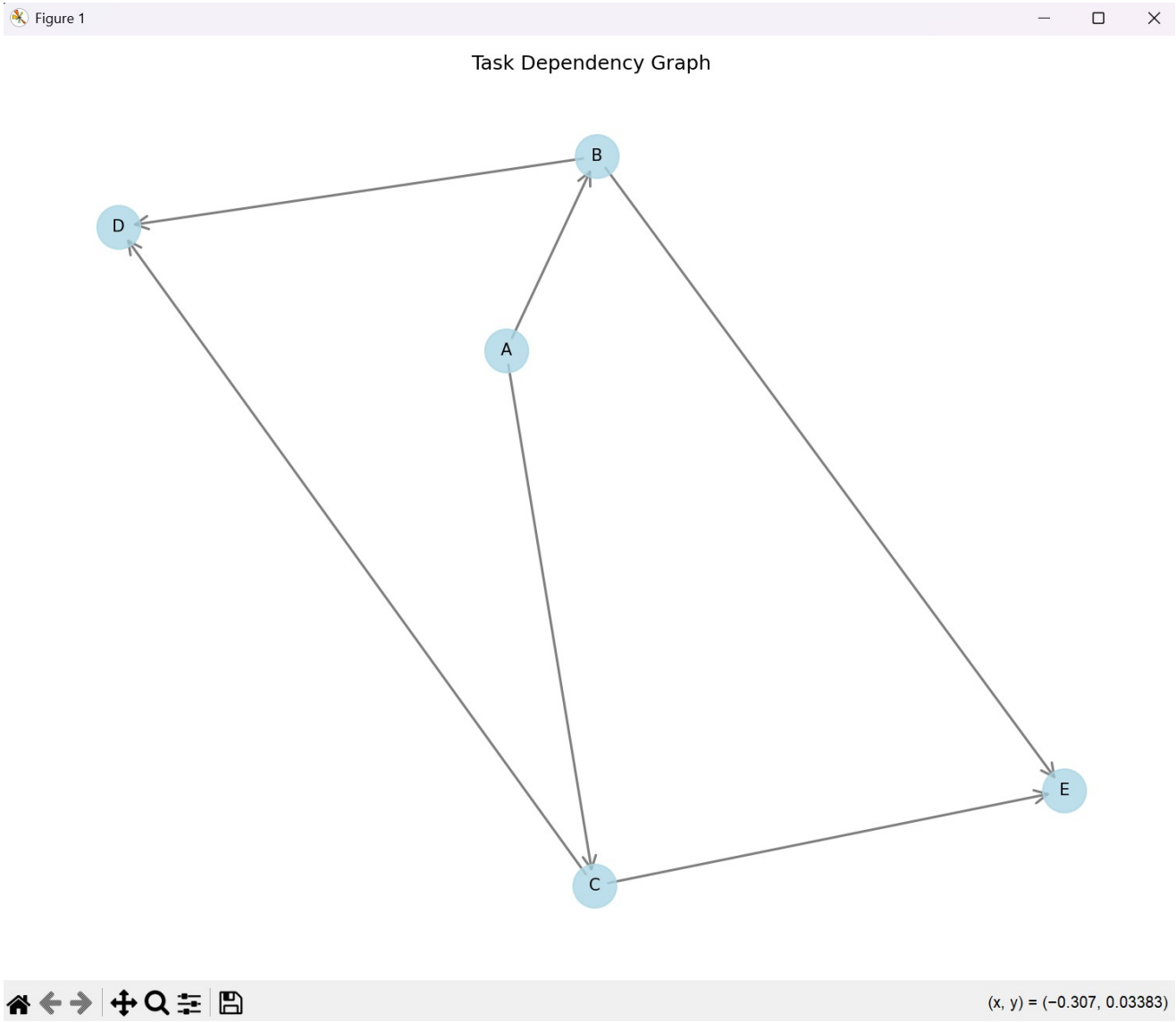
Draw

```python
def draw(self, filename=None):
    """
    Dessine le graphe de dépendances pour visualisation.
    """
    plt.figure(figsize=(10, 8))
    G = nx.DiGraph()

    # Ajoute tous les nœuds (tâches)
    for task_name in self.tasks:
        G.add_node(task_name)

    # Ajoute toutes les arêtes (dépendances)
    for task, deps in self.max_parallelism.items():
        for dep in deps:
            G.add_edge(dep, task)

    # Définit la disposition du graphe
    pos = nx.spring_layout(G, seed=42)  # Disposition cohérente avec graine fixe
    # Dessine le graphe
    nx.draw(G, pos, with_labels=True, node_color='lightblue',
            node_size=700, font_size=10, font_weight='bold',
            edge_color='blue', width=2, alpha=0.7, arrows=True)

    plt.title("Task Dependency Graph (Maximum Parallelism)")

    # Sauvegarde l'image si un nom de fichier est fourni
    if filename:
        plt.savefig(filename, dpi=300)
    plt.show()
```

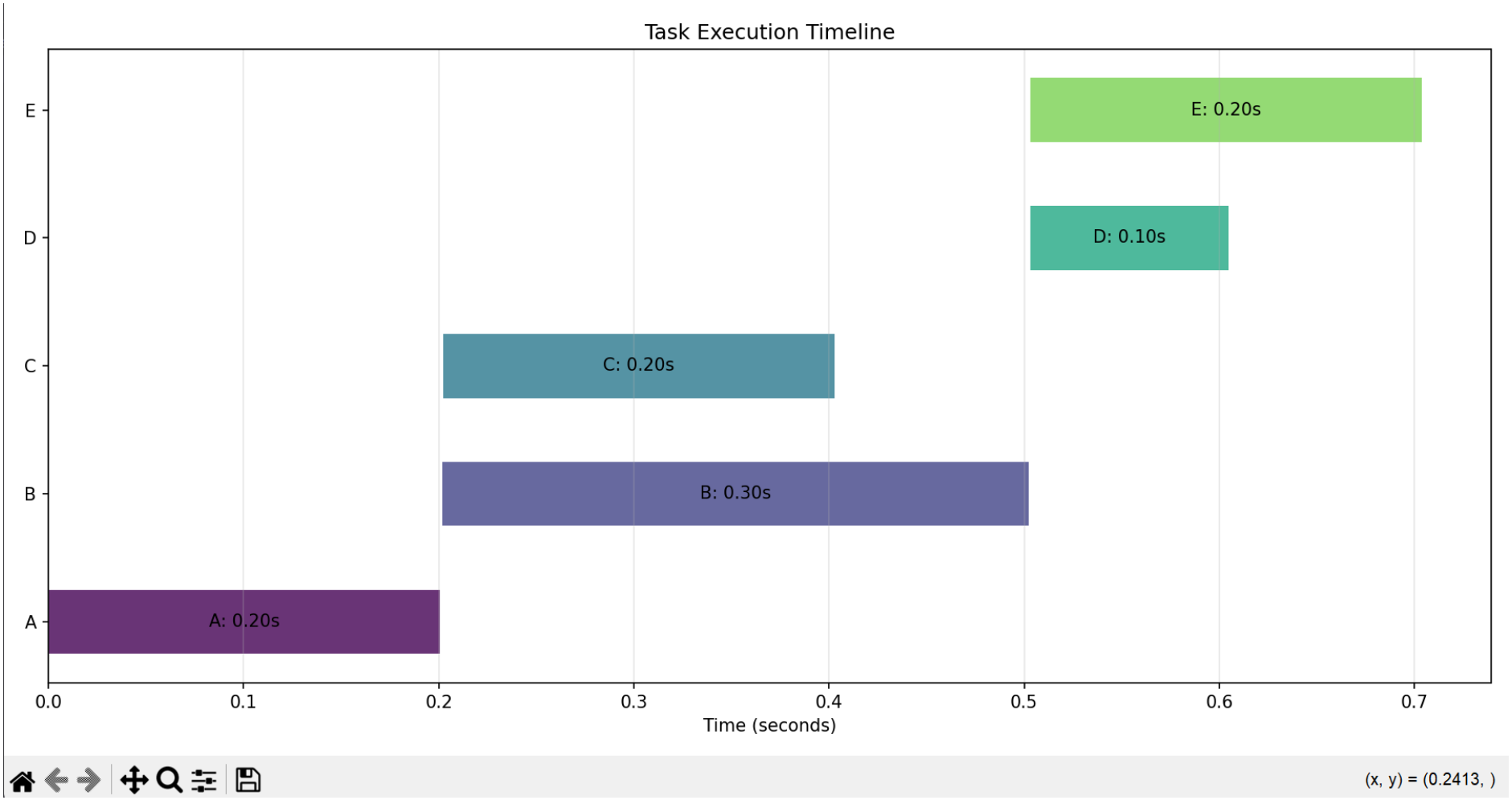
Test



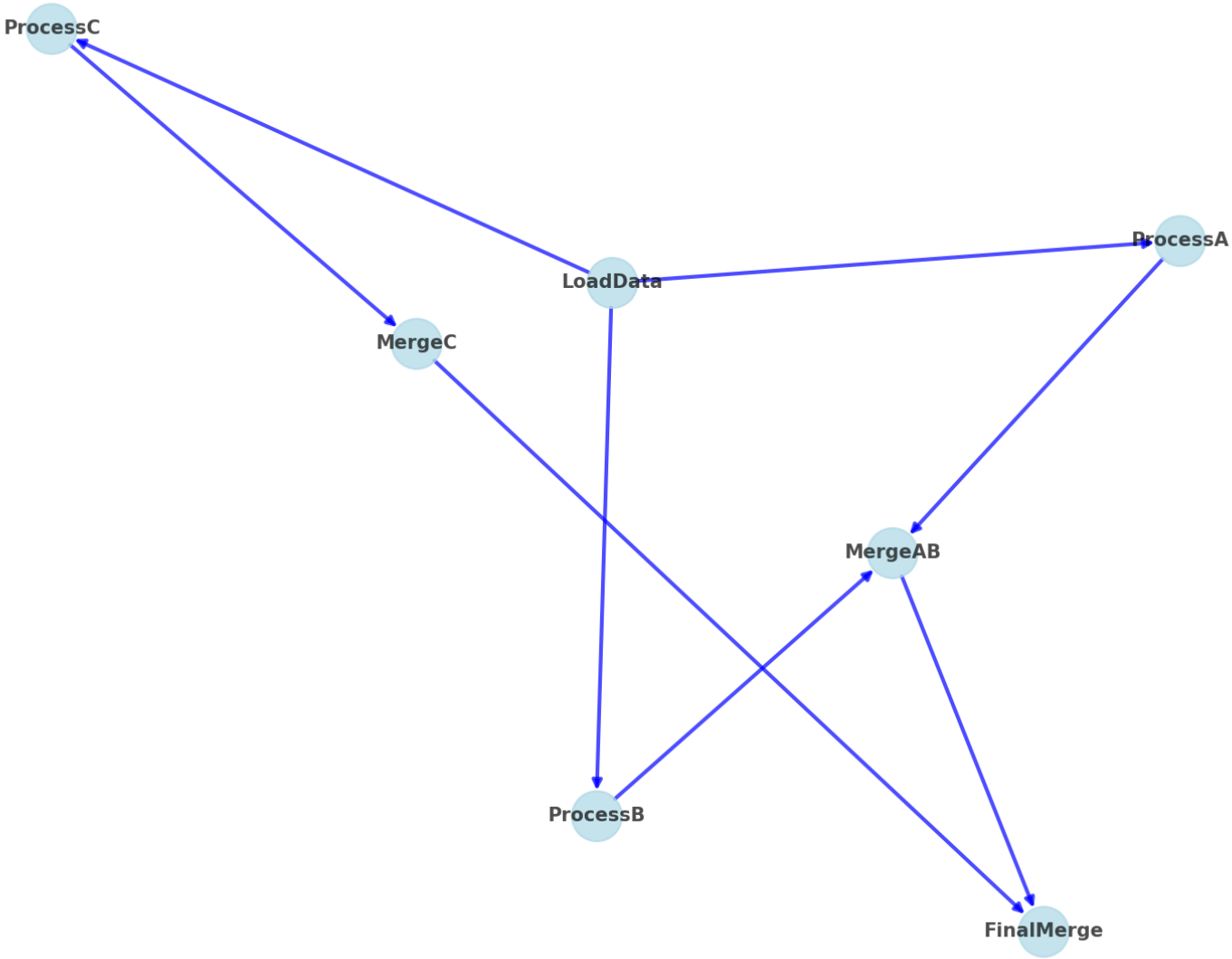
Test



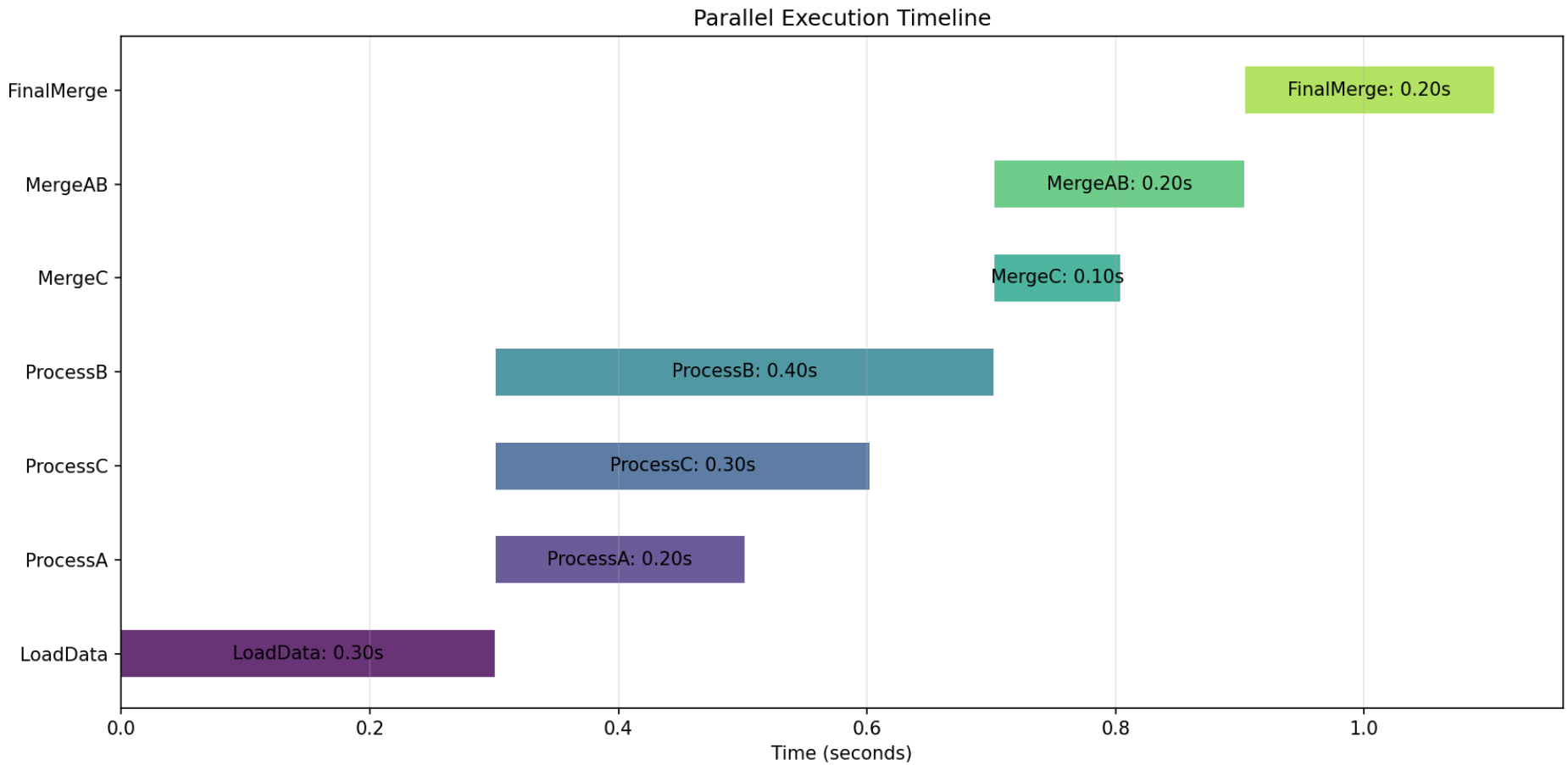
Test


Task Execution Timeline
E
D
C
B
A
E: 0.20s
D: 0.10s
C: 0.20s
B: 0.30s
A: 0.20s
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
Time (seconds)
(x, y) = (0.2413,)

Test


ProcessC
LoadData
ProcessA
MergeC
MergeAB
ProcessB
FinalMerge

Test


Parallel Execution Timeline
FinalMerge
MergeAB
MergeC
ProcessB
ProcessC
ProcessA
LoadData
FinalMerge: 0.20s
MergeAB: 0.20s
MergeC: 0.10s
ProcessB: 0.40s
ProcessC: 0.30s
ProcessA: 0.20s
LoadData: 0.30s
0.0
0.2
0.4
0.6
0.8
1.0
Time (seconds)

Test

```
--- TEST: Basic Sequential Execution ---
Generated dependency graph for basic sequential test
Running sequentially:
  Task A completed
  Task B completed
  Task C completed
  Task D completed
  Task E completed
Execution order: ['A', 'B', 'C', 'D', 'E']
Running with parallelism:
  Task A completed
  Task B completed
  Task C completed
  Task D completed
  Task E completed
Execution order: ['A', 'B', 'C', 'D', 'E']

--- TEST: Parallel Execution ---
Generated dependency graph for parallel execution test
Running with parallelism:
  Task A started at 1743760321.6924129
  Task A completed after 0.20 seconds
  Task B started at 1743760321.8935242
  Task C started at 1743760321.8936944
  Task C completed after 0.20 seconds
  Task B completed after 0.30 seconds
  Task E started at 1743760322.1945367
  Task D started at 1743760322.1946616
  Task D completed after 0.10 seconds
  Task E completed after 0.20 seconds
B and C overlap: 0.20 seconds
D and E overlap: 0.10 seconds
Generated execution timeline visualization

--- TEST: Resource Interference ---
Checking task interference:
A and B interfere: True
A and C interfere: False
A and D interfere: True
B and C interfere: False
B and D interfere: True
C and D interfere: True
Generated resource interference graph
```

Test

```
Generated derived dependency graph
Running sequentially:
  Task A started at 1743760323.243891
  Task A completed after 0.20 seconds
  Task C started at 1743760323.4443297
  Task C completed after 0.20 seconds
  Task B started at 1743760323.644918
  Task B completed after 0.30 seconds
  Task D started at 1743760323.9456594
  Task D completed after 0.10 seconds
  A: 1743760323.24 - 1743760323.44
  B: 1743760323.64 - 1743760323.95
  C: 1743760323.44 - 1743760323.64
  D: 1743760323.95 - 1743760324.05

--- TEST: Error Cases ---
1. Testing duplicate task names
  Correct error: Duplicate task names found: ['A', 'A']
2. Testing missing task in precedences
  Correct error: Tasks missing from precedences: {'B'}
3. Testing non-existent dependency
  Correct error: Non-existent dependency 'C' for task 'B'
4. Testing cyclic dependencies
  Correct error: Cyclic dependencies detected: [['A', 'B', 'C']]

--- TEST: Complex Workflow ---
Drawing the dependency graph
Generated complex workflow dependency graph
Running sequentially:
  Task LoadData started
  Task LoadData completed after 0.30 seconds
  Task ProcessA started
  Task ProcessA completed after 0.20 seconds
  Task ProcessB started
  Task ProcessB completed after 0.40 seconds
  Task ProcessC started
  Task ProcessC completed after 0.30 seconds
  Task MergeAB started
  Task MergeAB completed after 0.20 seconds
  Task MergeC started
  Task MergeC completed after 0.10 seconds
  Task FinalMerge started
  Task FinalMerge completed after 0.20 seconds
Sequential execution time: 1.71 seconds
Sequential execution order: ['LoadData', 'ProcessA', 'ProcessB', 'ProcessC', 'MergeAB', 'MergeC', 'FinalMerge']
Running with parallelism:
```

Difficultés rencontrées

- La construction du parallélisme maximal
- La synchronisation dans l'exécution parallèle

Conclusion

- Définition de tâches avec leurs dépendances
- Validation de la cohérence du système
- Exécution séquentielle et parallèle des tâches
- Visualisation du graphe de dépendance
- Test de déterminisme du système
- Mesure des performances du parallélisme

Merci pour votre attention !