OS202 Travaux dirigés n°2

10 février 2025

Xingzi ZHANG
2024-2025

Table des matières

1. Parallélisation ensemble de Mandelbrot

1.1. Répartition par blocs de lignes

L'objectif de cette étude est d'accélérer le calcul de l'ensemble de Mandelbrot en utilisant la parallélisation avec MPI. L'approche choisie consiste à diviser le calcul en blocs suivant les lignes de l'image et à distribuer ces blocs entre plusieurs processus. Le processus maître récupère ensuite les résultats pour reconstruire l'image complète.

Nous avons utilisé MPI avec la bibliothèque `mpi4py` en Python. La stratégie adoptée est la suivante :

- Chaque processus calcule une portion de l'image en fonction de son rang.
- Les lignes sont distribuées équitablement entre les processus.
- Le processus 0 collecte les résultats et assemble l'image.
- Le temps d'exécution est mesuré pour différents nombres de processus afin de calculer l'accélération obtenue.

Les temps d'exécution obtenus sont présentés dans le tableau ci-dessous :

Nombre de processus	Temps d'exécution (s)	Accélération (Speedup)
1	7.8349	1.00
2	4.6980	1.67
4	1.5130	5.18
8	0.9146	8.57

TABLE 1.1 : Temps d'exécution et accélération en fonction du nombre de processus

On observe que l'accélération est significative, mais ne suit pas exactement la loi idéale de speedup linéaire. Cela peut être expliqué par :

- Les coûts de communication entre les processus.
- Une distribution des charges qui peut ne pas être parfaitement équilibrée.
- La présence d'une partie séquentielle inévitable, comme l'assemblage final de l'image.

1.2. Répartition statique améliorée

Le tableau suivant présente les temps d'exécution pour différentes valeurs de p ainsi que le speedup calculé :

Nombre de processus (p)	Temps d'exécution $T(p)$ (s)	Speedup $S(p) = \frac{T(1)}{T(p)}$
1	7.5993	1.00
2	4.1259	1.84
4	1.6146	4.71
8	1.0279	7.39

TABLE 1.2 : Temps d'exécution et accélération en fonction du nombre de processus

Les résultats montrent que la nouvelle répartition des tâches améliore légèrement l'exécution pour un faible nombre de processus ($p = 2$), mais devient moins efficace que l'ancienne méthode pour un nombre élevé de processus ($p = 4$ et $p = 8$). Cela peut être attribué aux raisons suivantes :

- **Augmentation du coût de communication** : Avec plus de processus, les échanges de données entre eux deviennent plus coûteux, réduisant l'efficacité du speedup.

- **Fragmentation mémoire** : La nouvelle répartition impose une gestion de la mémoire plus fragmentée, réduisant la localité des données et augmentant les accès mémoire non optimaux.
- **Déséquilibre dans les calculs** : Certaines zones de l'ensemble de Mandelbrot nécessitent plus d'itérations que d'autres, ce qui peut causer un déséquilibre entre les processus.

Bien que la répartition équilibrée ait permis d'améliorer le speedup pour un faible nombre de processus, l'ancienne répartition semble plus efficace pour un nombre plus important de tâches. Une solution hybride combinant répartition statique et dynamique pourrait être envisagée pour optimiser les performances.

1.3. Stratégie Maître-Esclave

Dans cette section, nous avons mis en œuvre une stratégie maître-esclave pour paralléliser le calcul de l'ensemble de Mandelbrot. Contrairement aux approches statiques précédentes, où les lignes de l'image étaient distribuées de manière fixe entre les processus, cette approche adopte une distribution dynamique du travail.

L'objectif est d'assigner progressivement les tâches aux processus esclaves afin d'équilibrer la charge de calcul, notamment dans les zones où la convergence est plus lente.

Nous avons mesuré le temps d'exécution pour différents nombres de processus, et les résultats obtenus sont présentés dans le tableau suivant :

Nombre de processus	Temps d'exécution (s)	Speedup
1	0.0000	-
2	7.3550	1.00
4	3.2503	2.26
8	2.8416	2.59

Table 1.3 : Temps d'exécution et speedup en fonction du nombre de processus

Le speedup est défini comme :

$$S(p) = \frac{T(2)}{T(p)} \tag{1.1}$$

où $T(2)$ est le temps d'exécution avec 2 processus, et $T(p)$ est le temps avec p processus.

Nous comparons ces résultats avec les approches de répartition statique :

Nombre de processus	Répartition statique 1 (s)	Répartition statique 2 (s)	Maître-esclave (s)
1	7.8349	7.5993	7.3550
2	4.6980	4.1259	7.3550
4	1.5130	1.6146	3.2503
8	0.9146	1.0279	2.8416

Table 1.4 : Comparaison des différentes stratégies de parallélisation

Nous observons plusieurs phénomènes intéressants :

- Pour un petit nombre de processus (2 ou 4), le speedup n'est pas optimal. La répartition dynamique entraîne un coût de communication plus élevé par rapport à la répartition statique.
- Pour 8 processus, l'amélioration est moins significative qu'avec les approches statiques. Cela suggère que la gestion des tâches par le maître devient un goulot d'étranglement.
- Contrairement aux approches statiques, la charge est mieux équilibrée dans les zones complexes de Mandelbrot.

2. Produit matrice-vecteur

Le produit matrice-vecteur est une opération fondamentale en algèbre linéaire et est utilisée dans de nombreux domaines tels que l'analyse numérique, la modélisation scientifique et l'apprentissage automatique. L'objectif de cette étude est de comparer deux stratégies de parallélisation :

- **Par colonnes** : chaque processus MPI traite un sous-ensemble de colonnes.
- **Par lignes** : chaque processus MPI traite un sous-ensemble de lignes.

Nous mesurerons les temps d'exécution et calculerons le speedup obtenu pour chaque approche.
Nous avons utilisé la bibliothèque MPI (`mpi4py`) en Python pour paralléliser le calcul. La taille de la matrice utilisée est $N = 120$. Le programme est exécuté sur un nombre variable de processus p, et nous analysons le speedup en fonction de p.

2.1. Produit parallèle matrice-vecteur par colonne

Dans cette approche, la matrice A est partitionnée en blocs de colonnes, et chaque processus traite $N_{\text{loc}} = \frac{N}{p}$ colonnes. Chaque processus effectue ensuite le produit des colonnes associées avec le vecteur u, et les résultats partiels sont ensuite combinés via une opération `MPI.Allreduce`.
Temps d'exécution mesuré :

Nombre de processus (p)	Temps d'exécution $T(p)$ (s)	Speedup $S(p)$
1	5.1771	1.00
2	3.7934	1.36
4	2.2147	2.34
8	2.8416	1.82

TABLE 2.1 : Résultats pour la parallélisation par colonnes

Analyse :
L'approche par colonnes montre une amélioration des performances en augmentant le nombre de processus, mais le speedup est sous-linéaire. Avec 8 processus, on observe même une dégradation des performances. Cela est dû à :

- Un coût de communication accru avec `MPI.Allreduce`.
- Une répartition de charge moins efficace, car certaines colonnes nécessitent plus de calculs que d'autres.
- Une augmentation des conflits d'accès mémoire, réduisant les gains de parallélisation.

2.2. Produit parallèle matrice-vecteur par ligne

Dans cette approche, la matrice A est partitionnée en blocs de lignes, et chaque processus traite $N_{\text{loc}} = \frac{N}{p}$ lignes. Chaque processus effectue le produit matrice-vecteur pour ses propres lignes et envoie le résultat au processus maître via une opération `MPI.Gather`.
Temps d'exécution mesuré :

Nombre de processus (p)	Temps d'exécution $T(p)$ (s)	Speedup $S(p)$
1	4.1824	1.00
2	2.2617	1.85
4	1.2080	3.46
8	0.6130	6.83

TABLE 2.2 : Résultats pour la parallélisation par lignes

Analyse :
L'approche par lignes offre de bien meilleures performances, avec un speedup proche de la valeur idéale. La répartition des tâches est plus équilibrée et nécessite moins de communication MPI, ce qui explique la meilleure scalabilité.

Comparaison des temps d'exécution :

Nombre de processus (p)	Temps (Colonnes) $T_c(p)$	Temps (Lignes) $T_l(p)$
1	5.1771	4.1824
2	3.7934	2.2617
4	2.2147	1.2080
8	2.8416	0.6130

TABLE 2.3 : Comparaison des temps d'exécution des deux approches

Interprétation des résultats :

- La parallélisation par lignes est nettement plus efficace, avec un speedup qui se rapproche du speedup idéal.
- La parallélisation par colonnes est limitée par un coût de communication plus élevé et une mauvaise répartition de la charge.
- L'approche par lignes minimise les échanges MPI et offre une meilleure localité mémoire, ce qui explique son efficacité.

La parallélisation par lignes est clairement préférable pour cette opération, car elle offre un meilleur équilibre de charge et réduit les communications MPI inutiles. Pour aller plus loin, nous pourrions :

- Mettre en place une distribution dynamique des tâches (work stealing) pour équilibrer la charge.
- Utiliser des communications asynchrones (`MPI.Isend` et `MPI.Irecv`) pour réduire l'attente entre les processus.
- Comparer cette approche avec une implémentation utilisant OpenMP pour voir si le parallélisme mémoire partagée est plus efficace.

3. Entraînement pour l'examen écrit

Alice a parallélisé une partie de son code sur une machine à mémoire distribuée. Elle observe que la partie exécutée en parallèle représente **90%** du temps d'exécution du programme séquentiel. L'objectif de cette analyse est de :

- Estimer l'**accélération maximale** que pourra obtenir Alice selon la **loi d'Amdahl**.
- Déterminer un **nombre raisonnable de nœuds de calcul** pour ne pas gaspiller de ressources CPU.
- Utiliser la **loi de Gustafson** pour estimer l'accélération maximale si la quantité de données double.

La loi d'Amdahl permet d'estimer l'**accélération maximale** ($S_{\max}$) d'un programme en fonction de la fraction parallélisable P et du nombre de processeurs n, selon la formule :

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}} \tag{3.1}$$

3.1. Accélération maximale pour $n \to \infty$

Lorsque n tend vers l'infini ($n \gg 1$), le terme $\frac{P}{n}$ devient négligeable, et l'accélération maximale théorique devient :

$$S_{\max} = \frac{1}{1-P} \tag{3.2}$$

Dans notre cas, $P = 0.9$ (90% du programme est parallélisable), donc :

$$S_{\max} = \frac{1}{1-0.9} = \frac{1}{0.1} = 10 \tag{3.3}$$

Ainsi, **même avec un nombre infini de nœuds de calcul, l'accélération maximale est limitée à 10**.

3.2. Choix d'un nombre raisonnable de nœuds

Un nombre raisonnable de nœuds correspond à une valeur de n où l'on approche déjà bien $S_{\max}$, sans gaspiller trop de ressources.

Prenons quelques valeurs typiques pour n :

Nombre de nœuds n	Accélération $S(n)$
2	$\frac{1}{0.1+\frac{0.9}{2}} = 1.82$
4	$\frac{1}{0.1+\frac{0.9}{4}} = 3.08$
8	$\frac{1}{0.1+\frac{0.9}{8}} = 5.00$
16	$\frac{1}{0.1+\frac{0.9}{16}} = 6.90$

TABLE 3.1 : Accélération en fonction du nombre de nœuds

On observe que le gain diminue rapidement après 8 nœuds. Un choix raisonnable serait donc entre 8 et 16 nœuds, car ajouter davantage de nœuds n'apporte qu'un gain marginal.

Alice observe qu'en pratique, l'accélération est limitée à **4** lorsqu'elle augmente le nombre de nœuds.

Si elle **double la quantité de données**, nous pouvons utiliser la **loi de Gustafson** pour estimer l'accélération maximale.

La loi de Gustafson est définie par :

$$S_G(n) = n - (1-P)(n-1) \tag{3.4}$$

Avec $P = 0.9$ et $n = 8$ (supposons qu'Alice ait testé avec 8 nœuds), nous avons :

$$S_G(8) = 8 - (1 - 0.9)(8 - 1) = 8 - 0.1 \times 7 = 8 - 0.7 = 7.3 \tag{3.5}$$

Si elle double la quantité de données, et que l'algorithme parallèle suit une **complexité linéaire**, alors la partie parallélisable devient dominante ($P \approx 1$). La loi de Gustafson donne alors :

$$S_G(8) \approx 8 \tag{3.6}$$

Ainsi, Alice peut espérer une accélération proche du nombre de nœuds, soit environ 8 fois plus rapide avec 8 nœuds.

A. Annexes

A.1. Code 1.1

```
from mpi4py import MPI
import numpy as np
from PIL import Image
import matplotlib.cm
from math import log
import time

class MandelbrotSet:
    def __init__(self, max_iterations=50, escape_radius=10):
        self.max_iterations = max_iterations
        self.escape_radius = escape_radius

    def convergence(self, c: complex, smooth=True) -> float:
        z = 0
        for iter in range(self.max_iterations):
            z = z*z + c
            if abs(z) > self.escape_radius:
                if smooth:
                    return iter + 1 - log(log(abs(z)))/log(2)
                return iter
        return self.max_iterations

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nbp = comm.Get_size()

width, height = 1024, 1024
scaleX = 3.0 / width
scaleY = 2.25 / height

rows_per_proc = height // nbp
start_row = rank * rows_per_proc
end_row = (rank + 1) * rows_per_proc if rank != nbp - 1 else height

mandelbrot_set = MandelbrotSet(max_iterations=100, escape_radius=10)
local_convergence = np.empty((rows_per_proc, width), dtype=np.float64)

start_time = time.time()
for i, y in enumerate(range(start_row, end_row)):
    for x in range(width):
        c = complex(-2.0 + scaleX * x, -1.125 + scaleY * y)
        local_convergence[i, x] = mandelbrot_set.convergence(c, smooth=True)
end_time = time.time()
local_time = end_time - start_time

if rank == 0:
    global_convergence = np.empty((height, width), dtype=np.float64)
else:
    global_convergence = None

comm.Gather(local_convergence, global_convergence, root=0)

if rank == 0:
    image = Image.fromarray(np.uint8(matplotlib.cm.plasma(global_convergence) * 255))
```

```
    image.save("mandelbrot_mpi.png")
    print(f"Temps d'ex\'ecution (nbp={nbp}) : {local_time:.4f} s")
```

A.2. Code 1.2

```
from mpi4py import MPI
import numpy as np
from PIL import Image
import matplotlib.cm
from math import log
import time

class MandelbrotSet:
    def __init__(self, max_iterations=50, escape_radius=10):
        self.max_iterations = max_iterations
        self.escape_radius = escape_radius

    def convergence(self, c: complex, smooth=True) -> float:
        z = 0
        for iter in range(self.max_iterations):
            z = z*z + c
            if abs(z) > self.escape_radius:
                if smooth:
                    return iter + 1 - log(log(abs(z)))/log(2)
                return iter
        return self.max_iterations

# Initialisation MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nbp = comm.Get_size()

# D finition des param tres de l'image
width, height = 1024, 1024
scaleX = 3.0 / width
scaleY = 2.25 / height

# Meilleure r partition statique : Distribution des lignes de mani re    quilibre
num_rows = height // nbp
remainder = height % nbp

if rank < remainder:
    start_row = rank * (num_rows + 1)
    end_row = start_row + num_rows + 1
else:
    start_row = rank * num_rows + remainder
    end_row = start_row + num_rows

local_convergence = np.empty((end_row - start_row, width), dtype=np.float64)

start_time = time.time()
for i, y in enumerate(range(start_row, end_row)):
    for x in range(width):
        c = complex(-2.0 + scaleX * x, -1.125 + scaleY * y)
        local_convergence[i, x] = MandelbrotSet(max_iterations=100, escape_radius=10)
            .convergence(c, smooth=True)
end_time = time.time()
local_time = end_time - start_time

# Collecte des r sultats sur le processus ma tre
```

```
if rank == 0:
    global_convergence = np.empty((height, width), dtype=np.float64)
else:
    global_convergence = None

comm.Gather(local_convergence, global_convergence, root=0)

# Enregistrement de l'image sur le processus ma tre
if rank == 0:
    image = Image.fromarray(np.uint8(matplotlib.cm.plasma(global_convergence) * 255))
    image.save("mandelbrot_mpi_optimized.png")
    print(f"Temps d'ex cution (nbp={nbp}) : {local_time:.4f} s")
```

A.3. Code 1.3

```
from mpi4py import MPI
import numpy as np
from PIL import Image
import matplotlib.cm
from math import log
import time

class MandelbrotSet:
    def __init__(self, max_iterations=50, escape_radius=10):
        self.max_iterations = max_iterations
        self.escape_radius = escape_radius

    def convergence(self, c: complex, smooth=True) -> float:
        z = 0
        for iter in range(self.max_iterations):
            z = z*z + c
            if abs(z) > self.escape_radius:
                if smooth:
                    return iter + 1 - log(log(abs(z))) / log(2)
                return iter
        return self.max_iterations

# Initialisation MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Param tres de l'image
width, height = 1024, 1024
scaleX = 3.0 / width
scaleY = 2.25 / height
max_iterations = 100

mandelbrot_set = MandelbrotSet(max_iterations=max_iterations, escape_radius=10)

if rank == 0:
    # ---- PROCESSUS MA TRE ----
    start_time = time.time()
    # Liste des t ches (num ros de lignes    traiter)
    task_list = list(range(height))
    num_workers = size - 1
    active_workers = num_workers

    # Stockage des r sultats
    result_image = np.zeros((height, width), dtype=np.float64)
```

```
46
47      # Envoi initial des premi res t ches
48      for worker in range(1, num_workers + 1):
49          if task_list:
50              task = task_list.pop(0)
51              comm.send(task, dest=worker, tag=1)
52          else:
53              comm.send(None, dest=worker, tag=2)  # Pas de t ches restantes
54              active_workers -= 1
55
56      # R ception et distribution des t ches
57      while active_workers > 0:
58          status = MPI.Status()
59          data = comm.recv(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG, status=status)
60          worker = status.Get_source()
61          row, values = data
62
63          # Stocker les r sultats
64          result_image[row, :] = values
65
66          # Envoyer une nouvelle t che s'il en reste
67          if task_list:
68              task = task_list.pop(0)
69              comm.send(task, dest=worker, tag=1)
70          else:
71              comm.send(None, dest=worker, tag=2)  # Plus de travail -> fin
72              active_workers -= 1
73
74      end_time = time.time()
75      print(f"Temps d'ex cution (Master-Worker, {size} processus) : {end_time -
            start_time:.4f} s")
76
77      # Sauvegarde de l'image
78      image = Image.fromarray(np.uint8(matplotlib.cm.plasma(result_image) * 255))
79      image.save("mandelbrot_master_worker.png")
80
81  else:
82      # ---- PROCESSUS ESCLAVE ----
83      while True:
84          task = comm.recv(source=0, tag=MPI.ANY_TAG)
85          if task is None:
86              break  # Fin des t ches
87
88          # Calculer la ligne demand e
89          row_values = np.zeros(width, dtype=np.float64)
90          y = -1.125 + scaleY * task
91          for x in range(width):
92              c = complex(-2.0 + scaleX * x, y)
93              row_values[x] = mandelbrot_set.convergence(c, smooth=True)
94
95          # Envoyer les r sultats au ma tre
96          comm.send((task, row_values), dest=0, tag=3)
```

A.4. Code 2.1

```
1  from mpi4py import MPI
2  import numpy as np
3  from PIL import Image
4  from math import log
5  import time
```

```
import matplotlib.cm

class MandelbrotSet:
    def __init__(self, max_iterations: int, escape_radius: float = 2.):
        self.max_iterations = max_iterations
        self.escape_radius = escape_radius

    def convergence(self, c: np.ndarray, smooth=True) -> np.ndarray:
        iter_counts = np.full(c.shape, self.max_iterations, dtype=np.float64)
        z = np.zeros(c.shape, dtype=np.complex128)
        mask = np.ones(c.shape, dtype=bool)

        for it in range(self.max_iterations):
            z[mask] = z[mask] * z[mask] + c[mask]
            has_diverged = np.abs(z) > self.escape_radius
            iter_counts[has_diverged & mask] = it
            mask &= ~has_diverged
            if not np.any(mask):
                break

        if smooth:
            iter_counts[has_diverged] += 1 - np.log(np.log(np.abs(z[has_diverged])))
                / log(2)

        return iter_counts / self.max_iterations

# Initialisation MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Param tres de l'image
width, height = 1024, 1024
max_iterations = 200
escape_radius = 2.0

scaleX = 3.0 / width
scaleY = 2.25 / height

# R partition des colonnes
cols_per_proc = width // size
start_col = rank * cols_per_proc
end_col = (rank + 1) * cols_per_proc if rank != size - 1 else width

# Calcul Mandelbrot pour les colonnes assign es
mandelbrot_set = MandelbrotSet(max_iterations=max_iterations, escape_radius=
    escape_radius)
local_convergence = np.zeros((height, cols_per_proc), dtype=np.float64)

start_time = time.time()
for x in range(start_col, end_col):
    c = np.array([complex(-2.0 + scaleX * x, -1.125 + scaleY * y) for y in range(
        height)])
    local_convergence[:, x - start_col] = mandelbrot_set.convergence(c, smooth=True)
end_time = time.time()

# R colte des r sultats
global_convergence = None
if rank == 0:
    global_convergence = np.zeros((height, width), dtype=np.float64)

comm.Gather(local_convergence, global_convergence, root=0)
```

```
65
66 # Processus 0 sauvegarde l'image
67 if rank == 0:
68     image = Image.fromarray(np.uint8(matplotlib.cm.plasma(global_convergence) * 255))
69     image.save("mandelbrot_parallel_columns.png")
70     print(f"Temps de calcul ({size} processus) : {end_time - start_time:.4f} s")
```

A.5. Code 2.2

```
1  from mpi4py import MPI
2  import numpy as np
3  from PIL import Image
4  from math import log
5  import time
6  import matplotlib.cm
7
8  class MandelbrotSet:
9      def __init__(self, max_iterations: int, escape_radius: float = 2.):
10         self.max_iterations = max_iterations
11         self.escape_radius = escape_radius
12
13     def convergence(self, c: np.ndarray, smooth=True) -> np.ndarray:
14         iter_counts = np.full(c.shape, self.max_iterations, dtype=np.float64)
15         z = np.zeros(c.shape, dtype=np.complex128)
16         mask = np.ones(c.shape, dtype=bool)
17
18         for it in range(self.max_iterations):
19             z[mask] = z[mask] * z[mask] + c[mask]
20             has_diverged = np.abs(z) > self.escape_radius
21             iter_counts[has_diverged & mask] = it
22             mask &= ~has_diverged
23             if not np.any(mask):
24                 break
25
26         if smooth:
27             iter_counts[has_diverged] += 1 - np.log(np.log(np.abs(z[has_diverged])))
                   / log(2)
28
29         return iter_counts / self.max_iterations
30
31 # Initialisation MPI
32 comm = MPI.COMM_WORLD
33 rank = comm.Get_rank()
34 size = comm.Get_size()
35
36 # Param tres de l'image
37 width, height = 1024, 1024
38 max_iterations = 200
39 escape_radius = 2.0
40
41 scaleX = 3.0 / width
42 scaleY = 2.25 / height
43
44 # R partition des lignes
45 rows_per_proc = height // size
46 start_row = rank * rows_per_proc
47 end_row = (rank + 1) * rows_per_proc if rank != size - 1 else height
48
49 # Calcul Mandelbrot pour les lignes assign es
```

```
mandelbrot_set = MandelbrotSet(max_iterations=max_iterations, escape_radius=
    escape_radius)
local_convergence = np.zeros((rows_per_proc, width), dtype=np.float64)

start_time = time.time()
for y in range(start_row, end_row):
    c = np.array([complex(-2.0 + scaleX * x, -1.125 + scaleY * y) for x in range(
        width)])
    local_convergence[y - start_row, :] = mandelbrot_set.convergence(c, smooth=True)
end_time = time.time()

# R colte des r sultats
global_convergence = None
if rank == 0:
    global_convergence = np.zeros((height, width), dtype=np.float64)

comm.Gather(local_convergence, global_convergence, root=0)

# Processus 0 sauvegarde l'image
if rank == 0:
    image = Image.fromarray(np.uint8(matplotlib.cm.plasma(global_convergence) * 255))
    image.save("mandelbrot_parallel_rows.png")
    print(f"Temps de calcul ({size} processus) : {end_time - start_time:.4f} s")
```