Interactive Benchmark - Schedulers Performance

Utility functions and benchmark setup

Regex parsing function for the C "time done" output

In [ ]:
```
import re

def parse_process_output(output):
    match = re.search(r"\d+\.\d+", output)
    if match:
        number = float(match.group())
    else:
        number = None
    return number
```

Display current specs and initial config

In [ ]:
```
import subprocess
import multiprocessing
import matplotlib.pyplot as plt
import numpy as np

cores_count = multiprocessing.cpu_count()
print("Benchmark done on a AMD Ryzen 9 3900X 12-Core Processor with Simultaneous Mu
print(f"Detected {cores_count} cores")

core_values = list(range(1, cores_count + 1))
default_n_size = 10 * 1024 * 1024

multiplier = 10
size = default_n_size * multiplier

print(f"Benchmarking with a size of {multiplier} * 10MB (default quicksort size)")
```

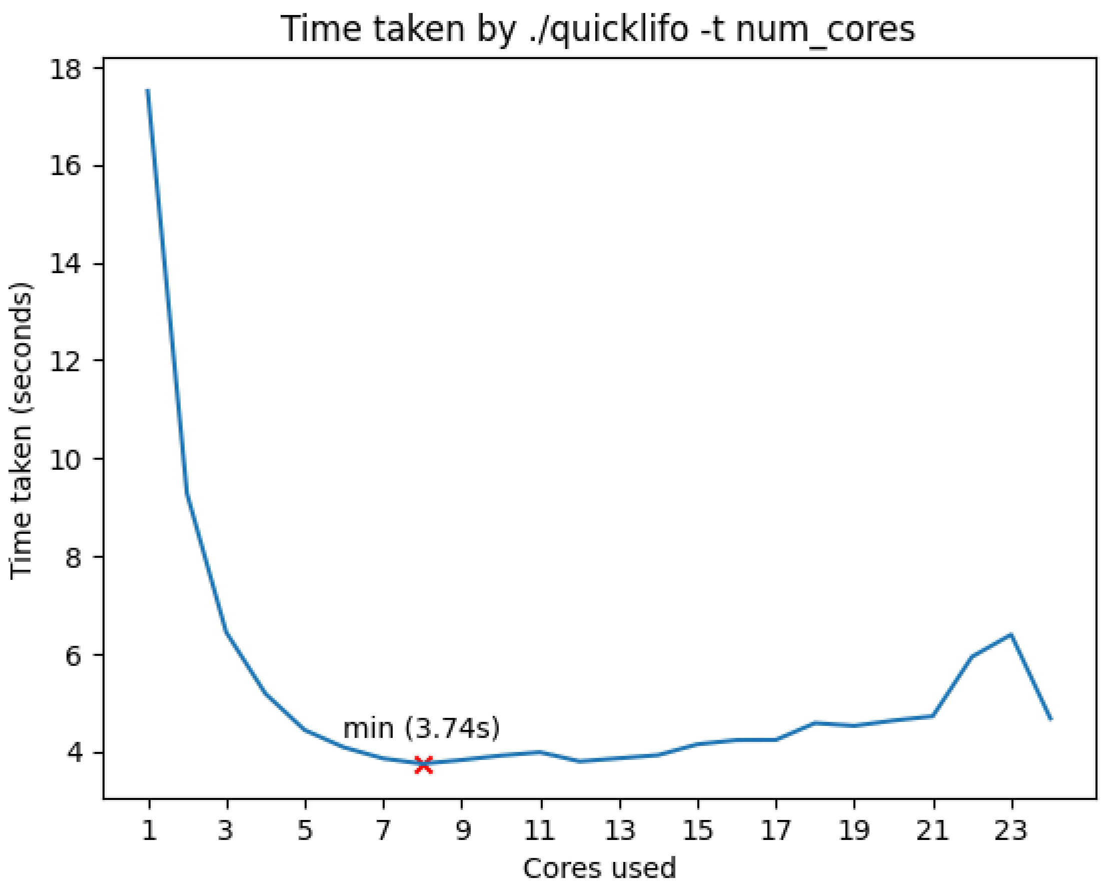
```
Benchmark done on a AMD Ryzen 9 3900X 12-Core Processor with Simultaneous Multithrea
ding (SMT) enabled
Detected 24 cores
Benchmarking with a size of 10 * 10MB (default quicksort size)
```

Benchmarking Quicksort + LIFO scheduler

In [ ]:
```
PROC_NAME = "quicklifo"

times = []
```

```python
print(f"Compiling {PROC_NAME}...")
subprocess.run(["make", f"{PROC_NAME}"], check=True, stdout=subprocess.DEVNULL, std

print(f"Running benchmarks for ./{PROC_NAME} -t num_cores -n {size}")
for num_cores in core_values:
    proc = subprocess.run([f"./{PROC_NAME}", "-t", str(num_cores), "-n", str(size)]
    output = proc.stdout.decode().strip()
    time_taken = parse_process_output(output)
    print(f"[{str(num_cores).zfill(2)}] Time taken: {time_taken} seconds")
    times.append(time_taken)

min_time = min(times)
min_index = times.index(min_time)

print(f"Minimum time taken: {min_time} seconds for cores = {core_values[min_index]}

x_ticks = np.arange(min(core_values), max(core_values) + 1, 2)

plt.plot(core_values, times)
plt.scatter(core_values[min_index], min_time, color='red', marker='x')
plt.annotate(f"min ({min_time:.2f}s)", (core_values[min_index], min_time), textcoor
plt.xlabel('Cores used')
plt.ylabel('Time taken (seconds)')
plt.title(f'Time taken by ./{PROC_NAME} -t num_cores')
plt.xticks(x_ticks)
plt.show()

quicklifo_times = times.copy()
```

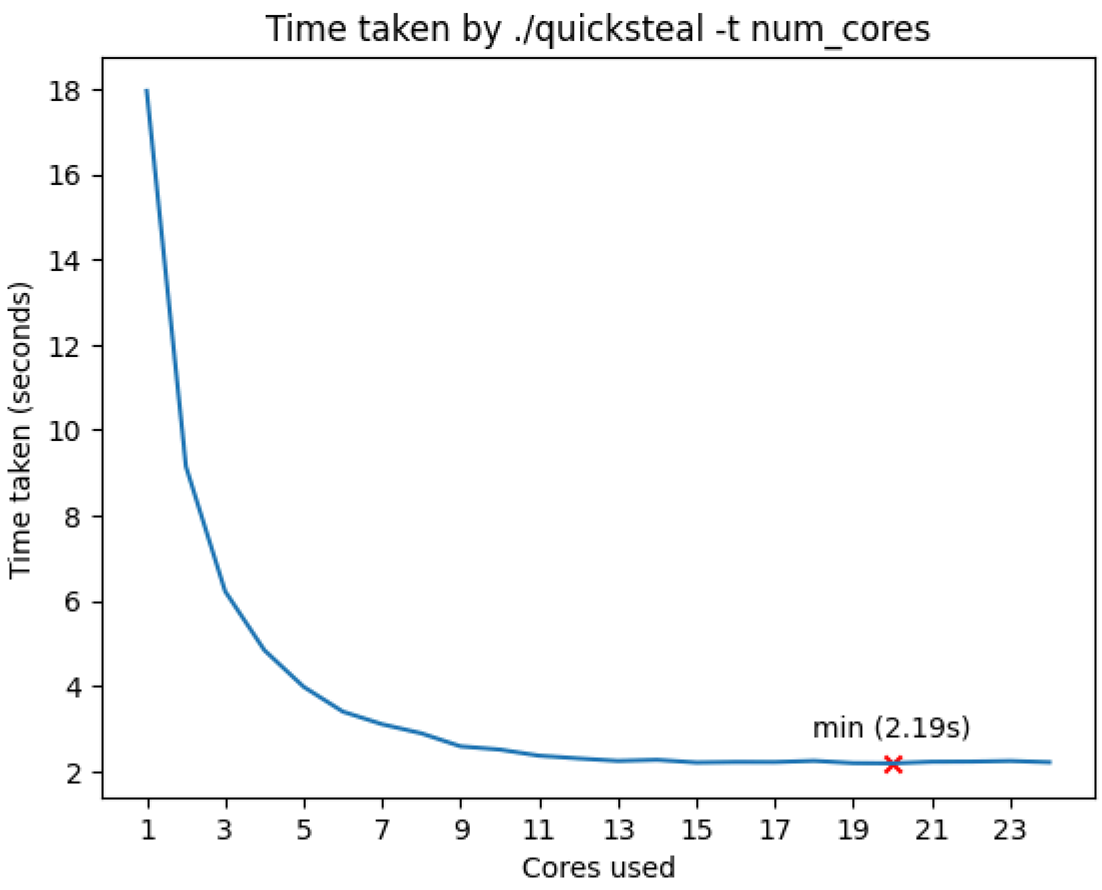
```
Compiling quicklifo...
Running benchmarks for ./quicklifo -t num_cores -n 104857600
[01] Time taken: 17.509984 seconds
[02] Time taken: 9.279284 seconds
[03] Time taken: 6.436856 seconds
[04] Time taken: 5.178791 seconds
[05] Time taken: 4.433078 seconds
[06] Time taken: 4.078858 seconds
[07] Time taken: 3.851509 seconds
[08] Time taken: 3.742197 seconds
[09] Time taken: 3.819355 seconds
[10] Time taken: 3.910319 seconds
[11] Time taken: 3.97783 seconds
[12] Time taken: 3.791003 seconds
[13] Time taken: 3.855192 seconds
[14] Time taken: 3.919726 seconds
[15] Time taken: 4.140905 seconds
[16] Time taken: 4.22738 seconds
[17] Time taken: 4.230741 seconds
[18] Time taken: 4.575803 seconds
[19] Time taken: 4.519699 seconds
[20] Time taken: 4.627701 seconds
[21] Time taken: 4.715321 seconds
[22] Time taken: 5.933553 seconds
[23] Time taken: 6.387221 seconds
[24] Time taken: 4.668794 seconds
Minimum time taken: 3.742197 seconds for cores = 8
```



Benchmarking Quicksort + Work Stealing scheduler

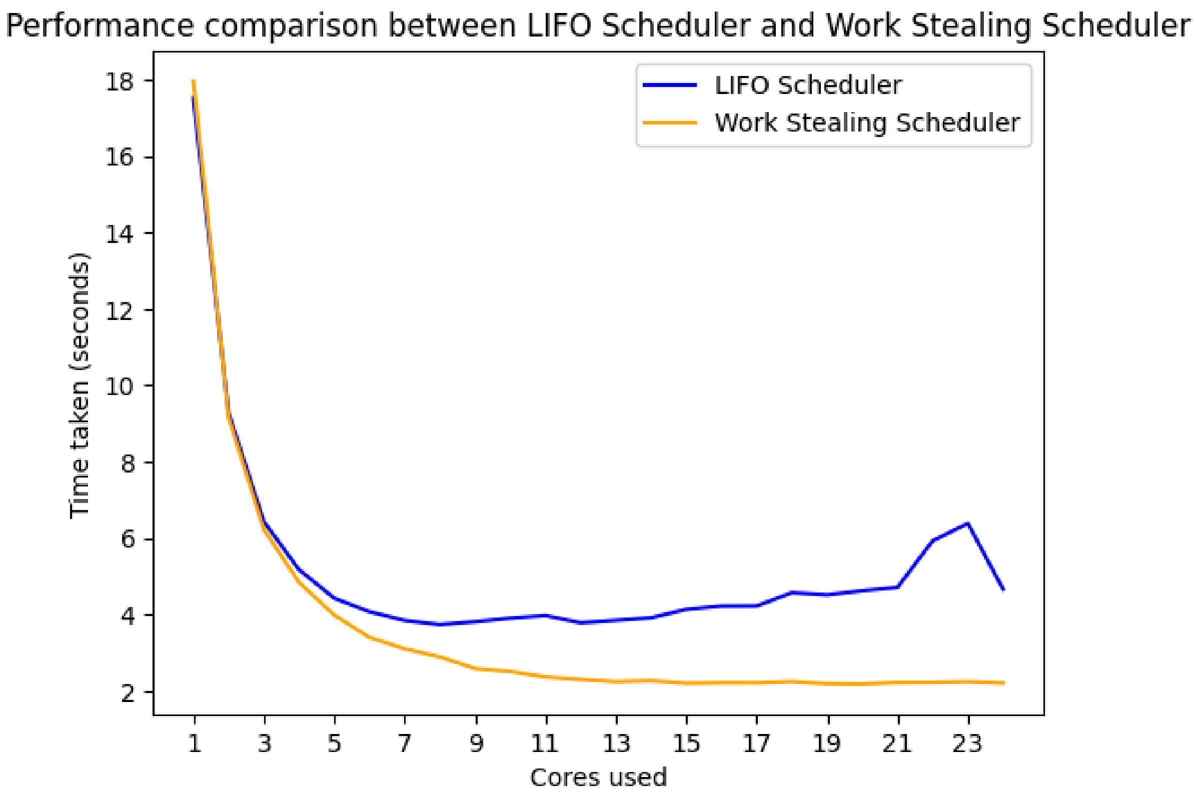
In [ ]:
```
PROC_NAME = "quicksteal"

times = []

print(f"Compiling {PROC_NAME}...")
subprocess.run(["make", f"{PROC_NAME}"], check=True, stdout=subprocess.DEVNULL, std

print(f"Running benchmarks for ./{PROC_NAME} -t num_cores -n {size}")
for num_cores in core_values:
    proc = subprocess.run([f"./{PROC_NAME}", "-t", str(num_cores), "-n", str(size)]
    output = proc.stdout.decode().strip()
    time_taken = parse_process_output(output)
    print(f"[{str(num_cores).zfill(2)}] Time taken: {time_taken} seconds")
    times.append(time_taken)

min_time = min(times)
min_index = times.index(min_time)

print(f"Minimum time taken: {min_time} seconds for cores = {core_values[min_index]}

x_ticks = np.arange(min(core_values), max(core_values) + 1, 2)

plt.plot(core_values, times)
plt.scatter(core_values[min_index], min_time, color='red', marker='x')
plt.annotate(f"min ({min_time:.2f}s)", (core_values[min_index], min_time), textcoor
plt.xlabel('Cores used')
plt.ylabel('Time taken (seconds)')
plt.title(f'Time taken by ./{PROC_NAME} -t num_cores')
plt.xticks(x_ticks)
plt.show()

quicksteal_times = times.copy()
```

```
Compiling quicksteal...
Running benchmarks for ./quicksteal -t num_cores -n 104857600
[01] Time taken: 17.951371 seconds
[02] Time taken: 9.154441 seconds
[03] Time taken: 6.228161 seconds
[04] Time taken: 4.846915 seconds
[05] Time taken: 3.989399 seconds
[06] Time taken: 3.406709 seconds
[07] Time taken: 3.111111 seconds
[08] Time taken: 2.897992 seconds
[09] Time taken: 2.592519 seconds
[10] Time taken: 2.516723 seconds
[11] Time taken: 2.37343 seconds
[12] Time taken: 2.308365 seconds
[13] Time taken: 2.250708 seconds
[14] Time taken: 2.275719 seconds
[15] Time taken: 2.212709 seconds
[16] Time taken: 2.225099 seconds
[17] Time taken: 2.223119 seconds
[18] Time taken: 2.252978 seconds
[19] Time taken: 2.199567 seconds
[20] Time taken: 2.193738 seconds
[21] Time taken: 2.230499 seconds
[22] Time taken: 2.234378 seconds
[23] Time taken: 2.250049 seconds
[24] Time taken: 2.219417 seconds
Minimum time taken: 2.193738 seconds for cores = 20
```



Comparaison Graph

In [ ]:
```
x_ticks = np.arange(min(core_values), max(core_values) + 1, 2)

plt.plot(core_values, quicklifo_times, color='blue', label='LIFO Scheduler')
plt.plot(core_values, quicksteal_times, color='orange', label='Work Stealing Schedu

plt.xlabel('Cores used')
plt.ylabel('Time taken (seconds)')
plt.title('Performance comparison between LIFO Scheduler and Work Stealing Schedule
plt.xticks(x_ticks)
plt.legend()

plt.show()
```



Conclusions

- The Work Stealing Scheduler has better average performance than the LIFO Scheduler
- Adding more cores does not increase performance linearly
- More is not always better. There seems to be an optimum < max_cores in both schedulers (overhead > perfomance gains)
- The first few additionnal cores are extremely important for performance. For example, in both cases, using two cores instead of one cuts computing time by half