



INSTITUTO FEDERAL

Norte de Minas Gerais

Campus Januária

Sistemas Distribuídos

- *Tasks && Pool de Workers* -



O que temos até aqui...

■ Paralelização de um problema...

- Analisar algoritmo em forma sequencial;
- Identificar oportunidades de paralelismo;
- Verificar (e tratar) possíveis condições de corrida;
- Manter workers ocupados com trabalho útil;



threads ou subprocessos



O que temos até aqui...

■ Paralelização de um problema...

- Analisar algoritmo em forma sequencial;
- Identificar oportunidades de paralelismo;
- Verificar (e tratar) possíveis condições de corrida;
- Manter workers ocupados com trabalho útil;



threads ou subprocessos

mas... quantos workers???



Granularidade

- A **granularidade** é um conceito que diz respeito ao **tamanho da tarefa decomposta (tasks)** e a **quantidade de workers** designados para tratá-las.
- Muitas vezes, a performance de um algoritmo depende **diretamente da granularidade definida**.



Granularidade

- Quantos girassóis há nessa imagem?





Granularidade

■ Granularidade Grossa

- **Pequeno número de grandes tarefas;**
- Baixo *overhead* (*Context switch*);
- Requer poucos processadores;
- Baixo nível de concorrência e paralelização;
- **Possibilidade de desbalanceamento de carga;**





Granularidade

■ Granularidade Fina

- Grande número de pequenas tarefas;
- Maior *overhead* (*Context switch*);
- Maior nível de concorrência e paralelização;
- Melhor utilização da qtde. de processadores (se disponíveis);
- **Melhor balanceamento de carga;**



Granularidade

- Granularidade Fina vs. Granularidade Grossa
- **Equilíbrio** é a chave!



Granularidade

- Granularidade Fina vs. Granularidade Grossa
- **Equilíbrio** é a chave!

mas...



Granularidade

- Granularidade Fina vs. Granularidade Grossa
- **Equilíbrio** é a chave!

mas...

*E se pudéssemos “definir” a
granularidade em tempo de execução?*



INSTITUTO FEDERAL
Norte de Minas Gerais
Campus Januária

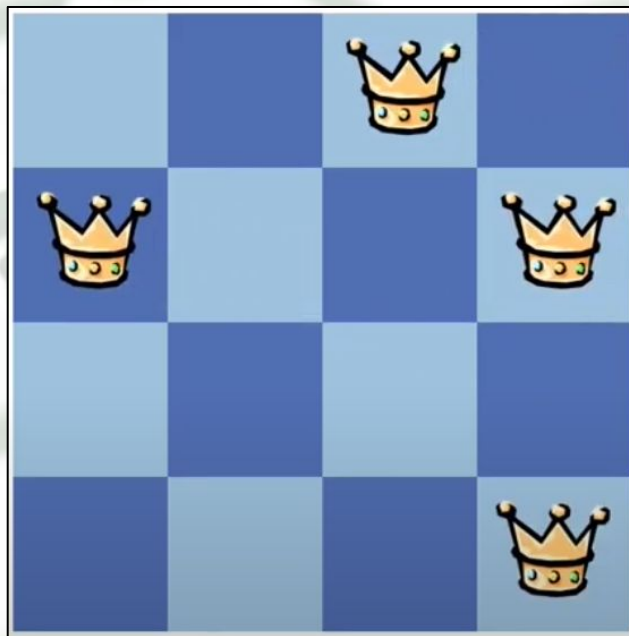
Solução Ideal?





Outro Exemplo...

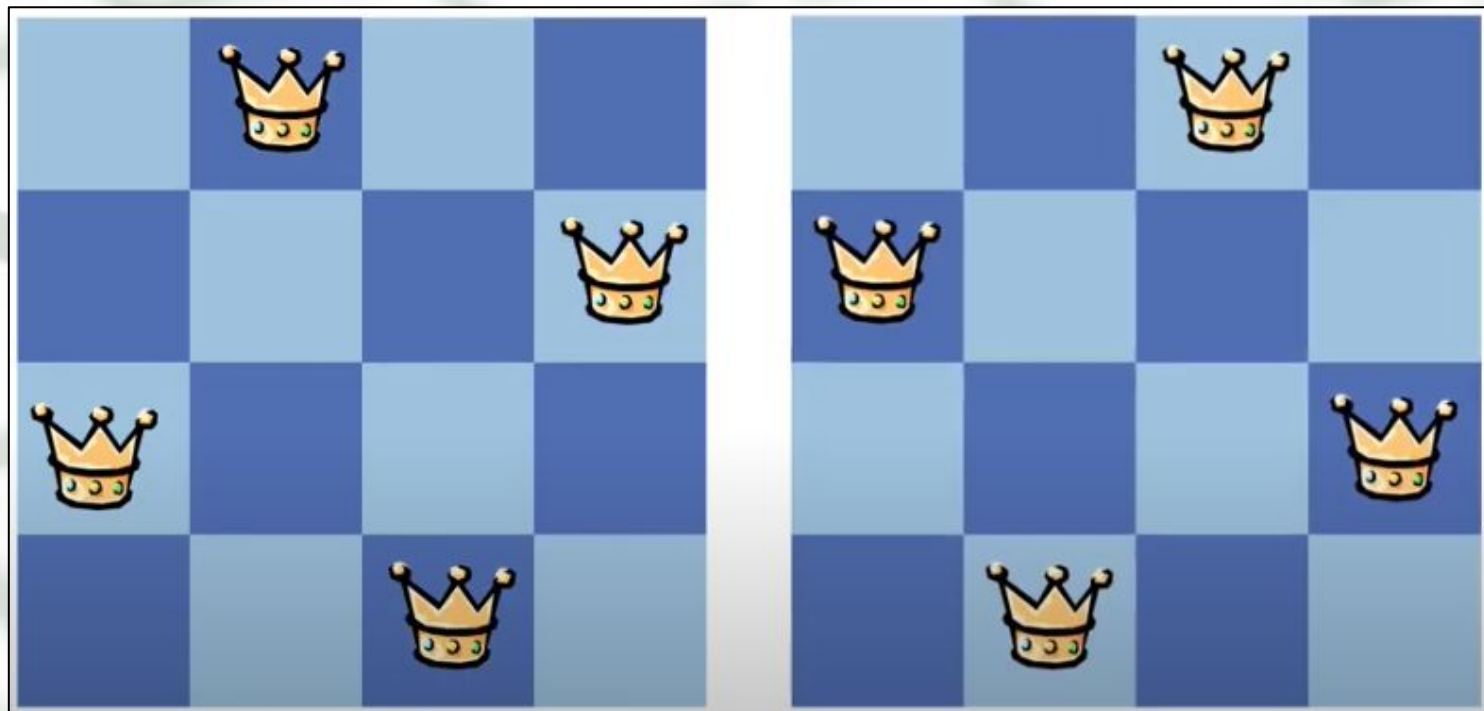
- **O Problema das N Rainhas...**
- É possível colocar N Rainhas em um tabuleiro $N \times N$ de forma que nenhuma rainha ataque outra?





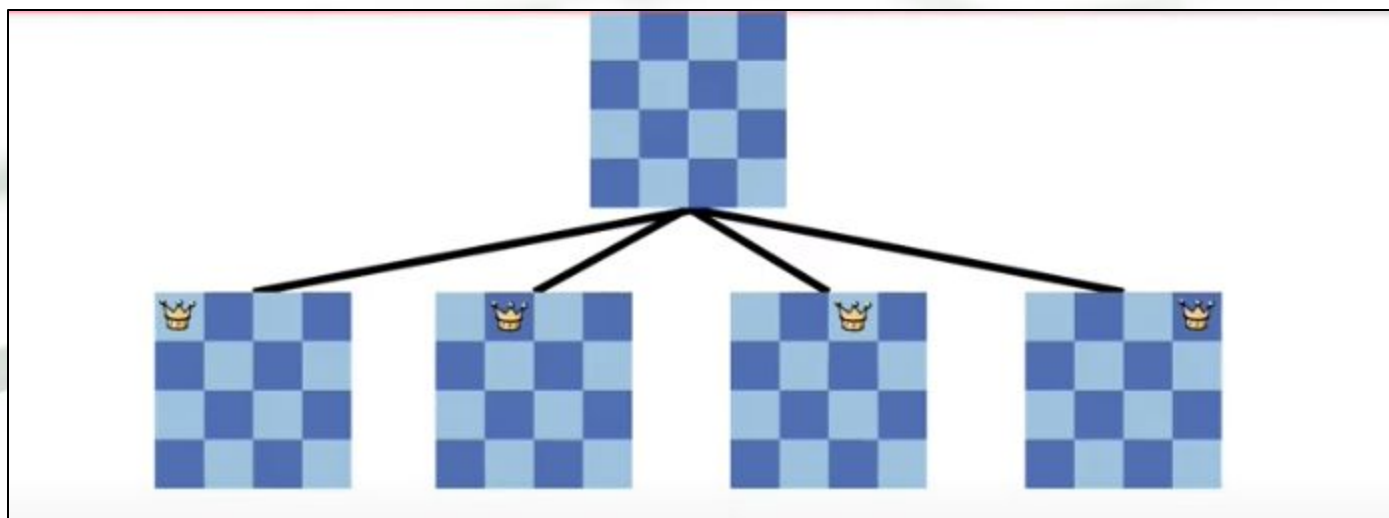
Outro Exemplo...

- **O Problema das N Rainhas...**
- Para $N = 4$, as duas únicas soluções possíveis são:



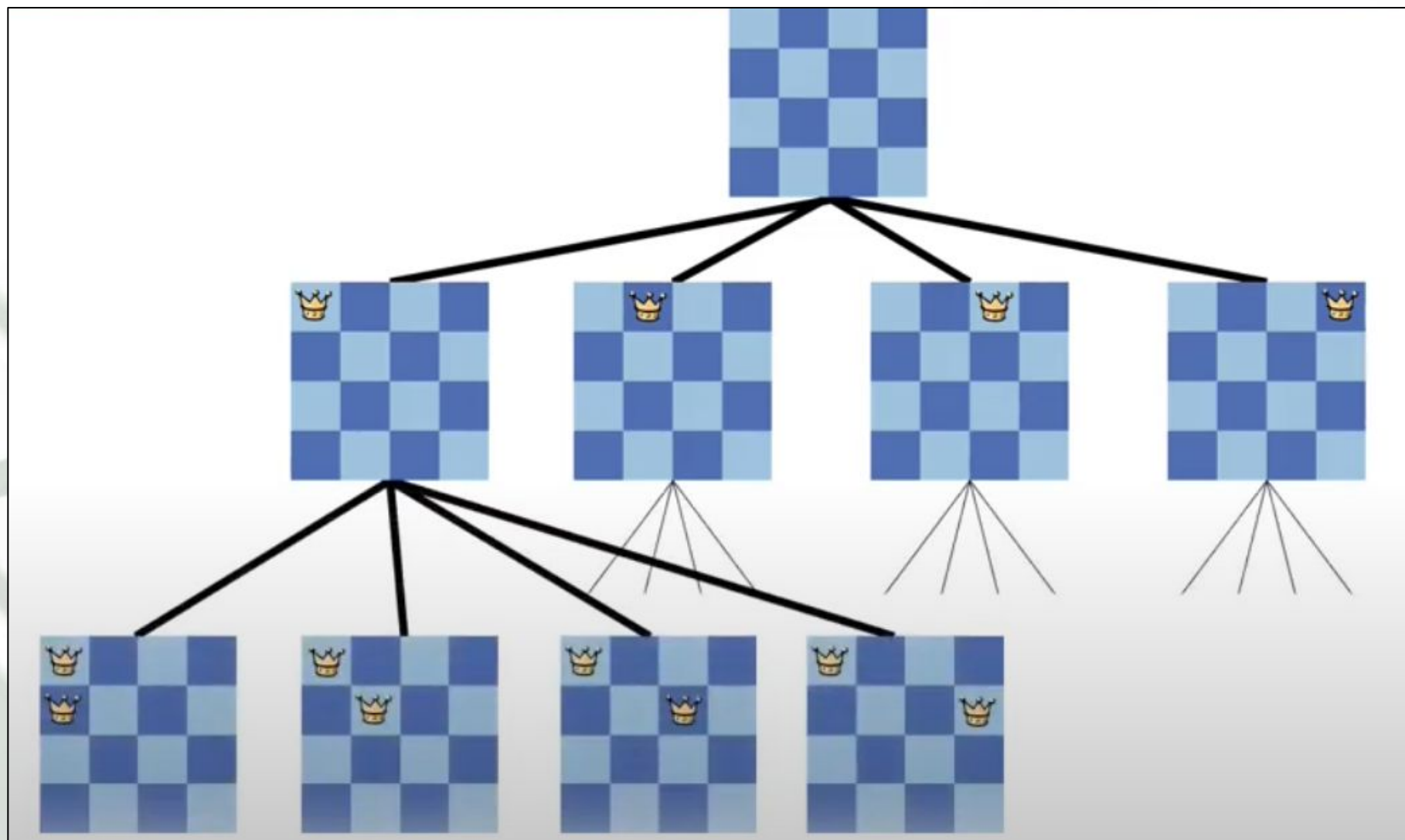


■ Busca Exaustiva



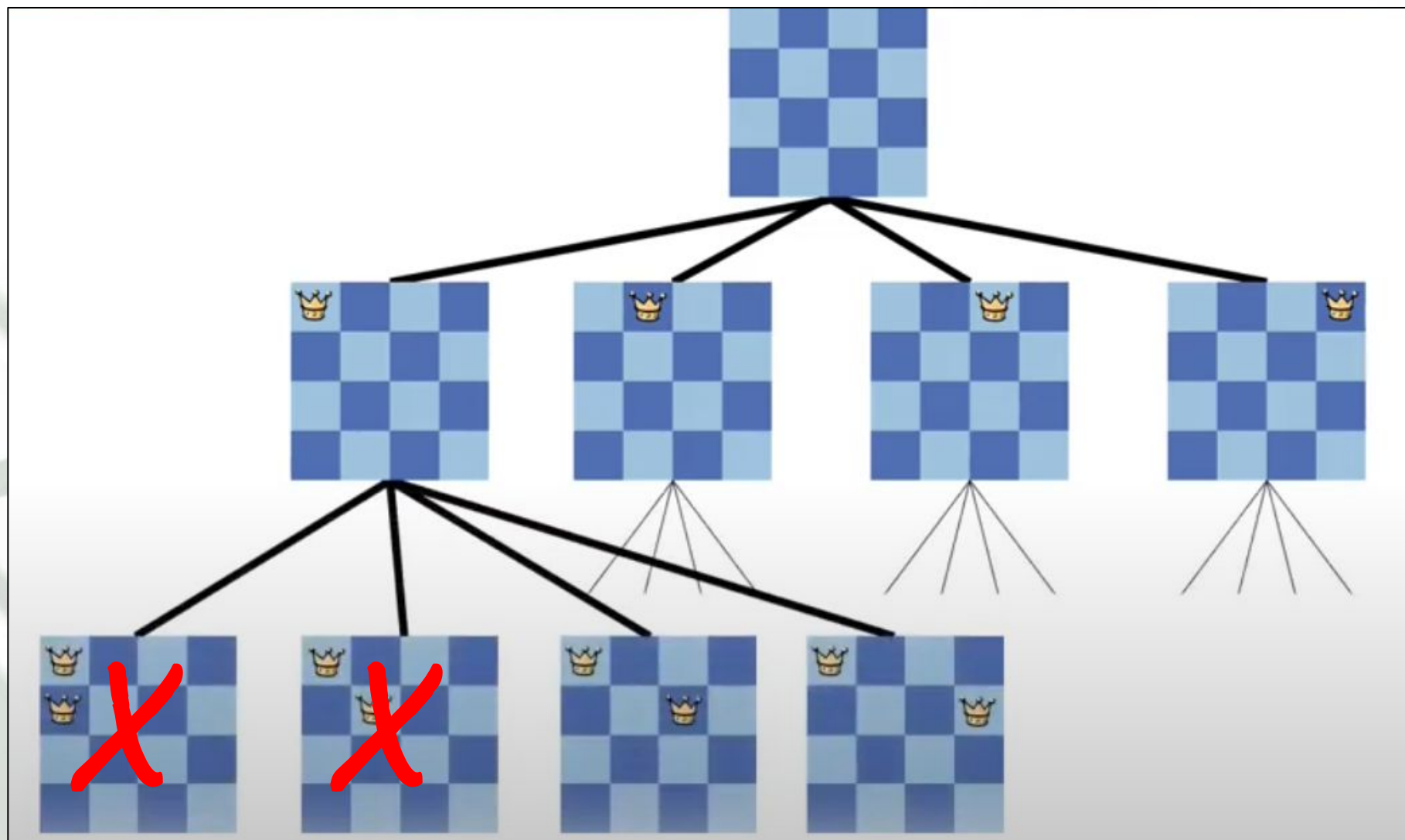


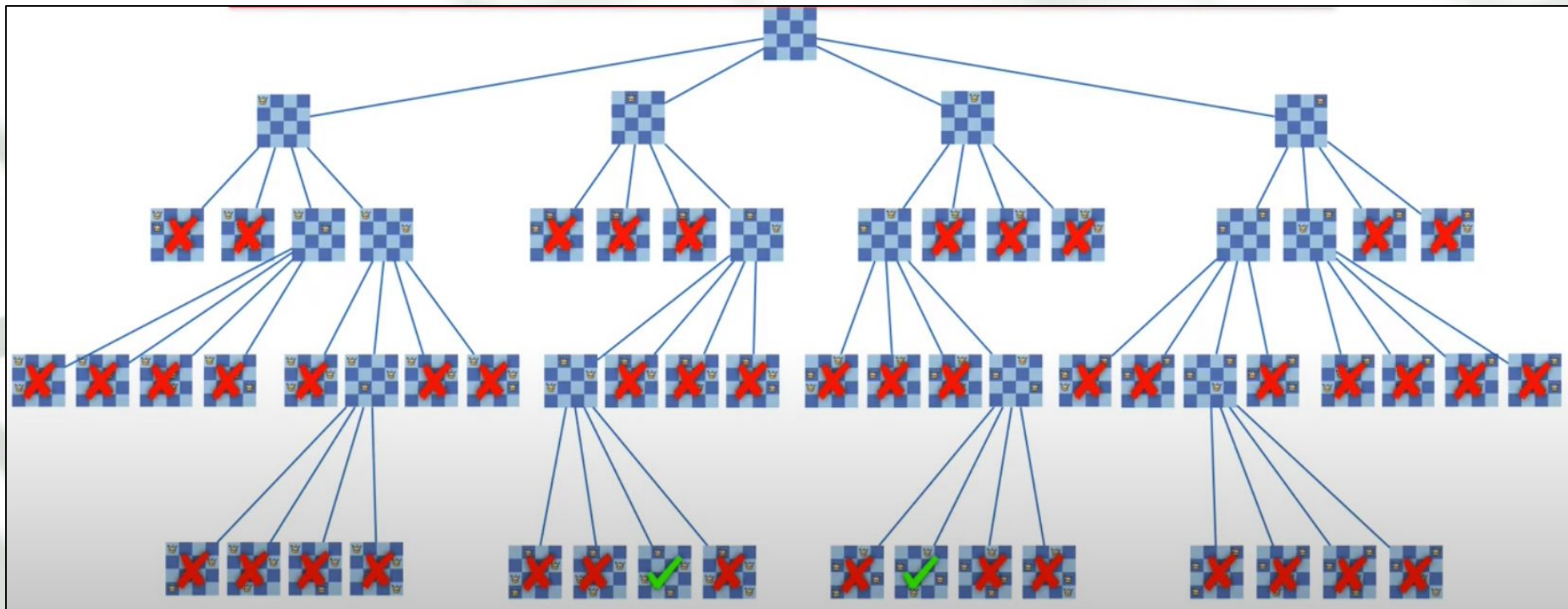
Algoritmo





Algoritmo

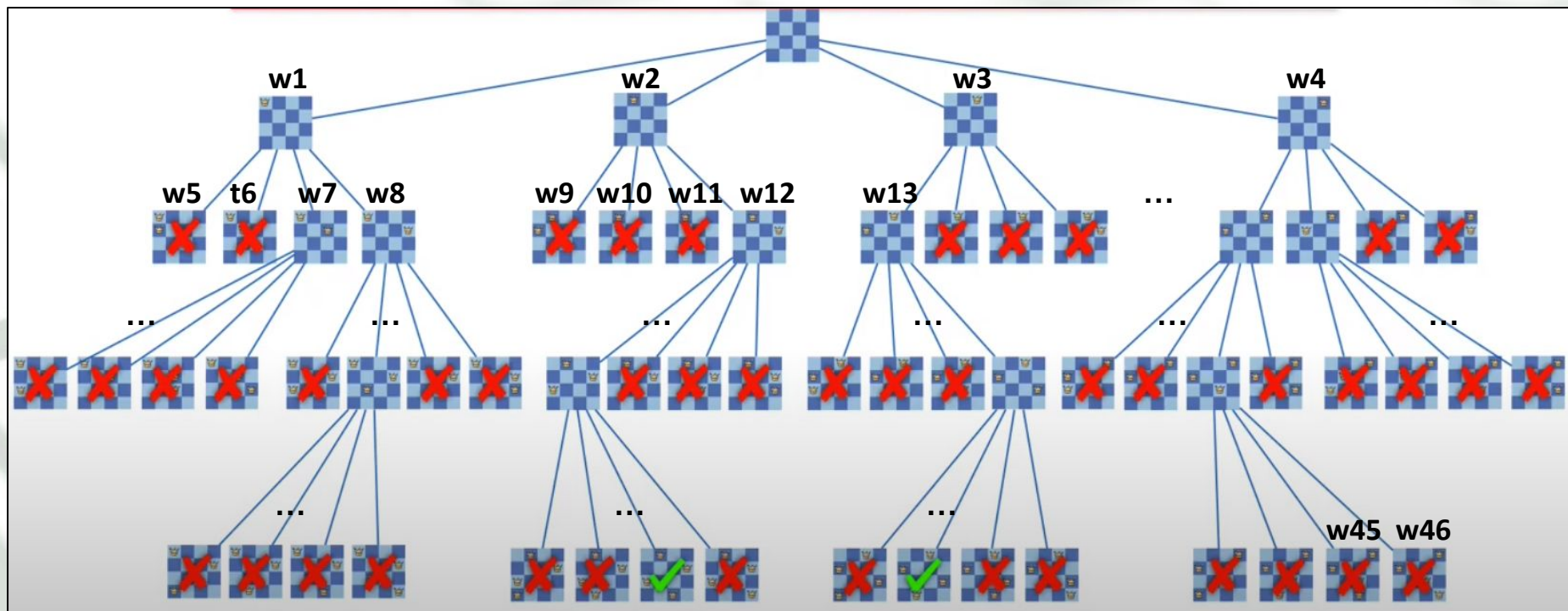






Algoritmo

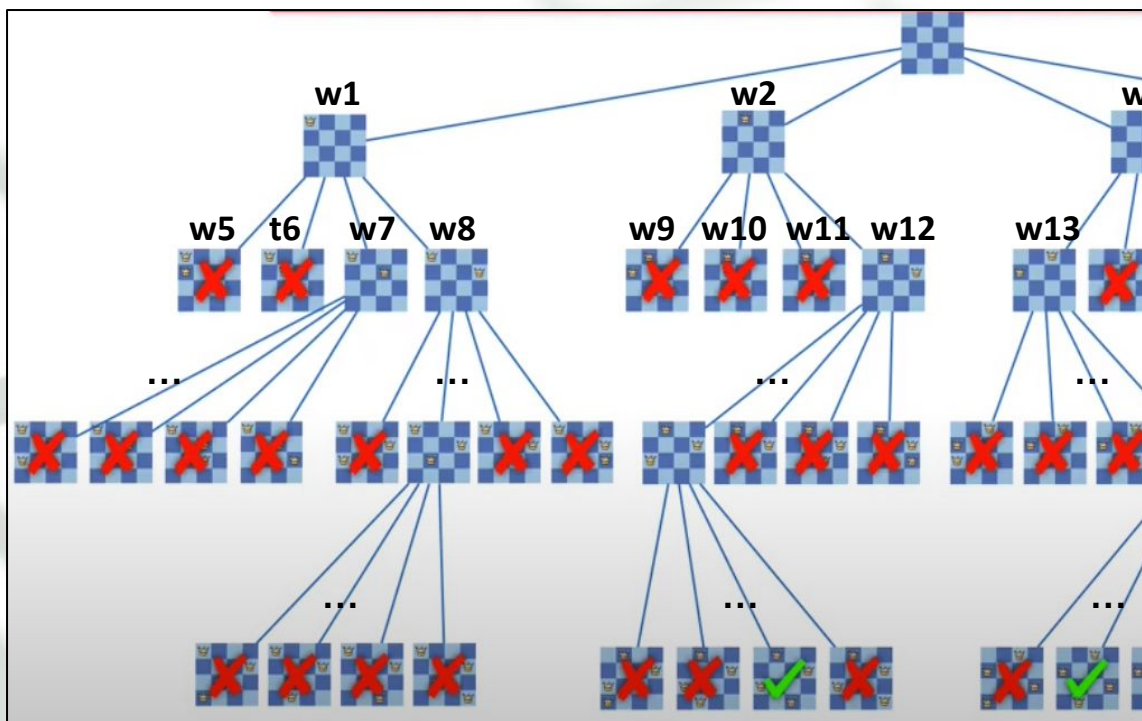
- Busca Exaustiva => Um worker por tarefa...





Algoritmo

■ Busca Exaustiva => Um worker por tarefa...



A cada nova possibilidade, um novo worker é instanciado para tratar o job.

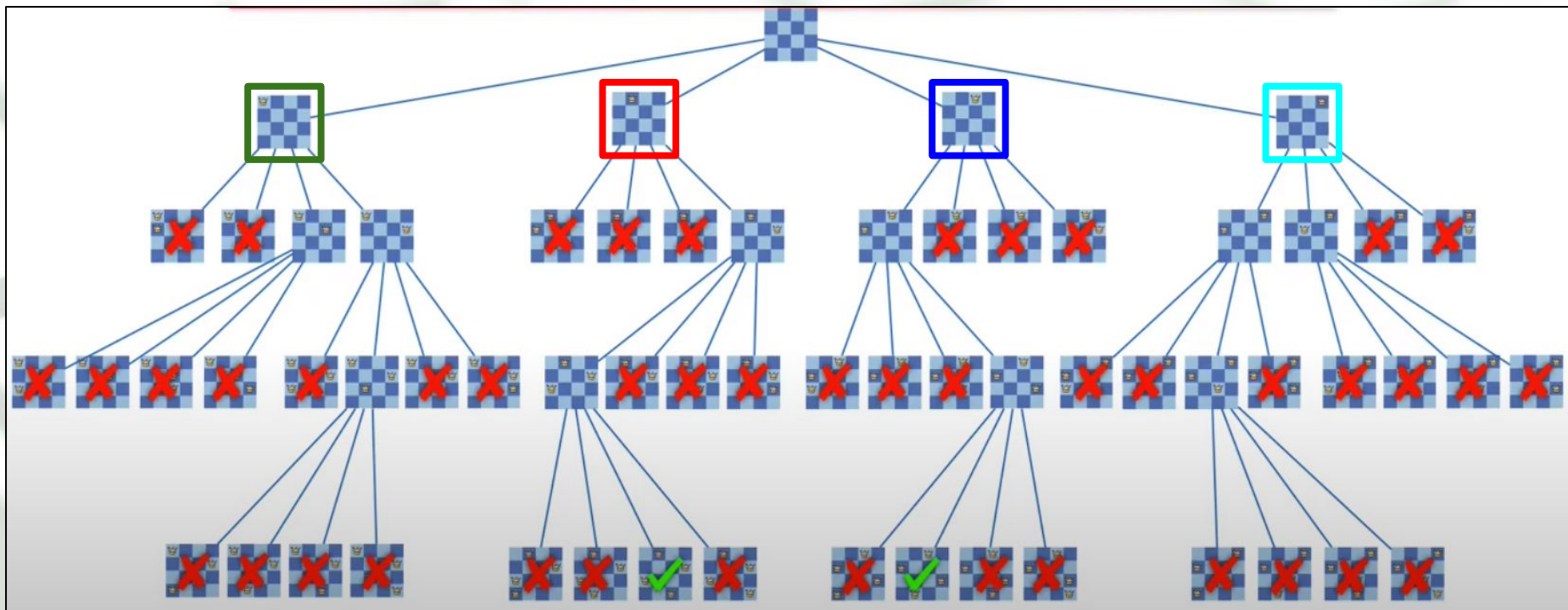
Vantagem:
Facilidade e simplicidade

Desvantagem:
Alto Overhead



Algoritmo

- **Busca Exaustiva => Particionamento Dinâmico**
 - *Workers* trabalham dinamicamente para resolver o problema.

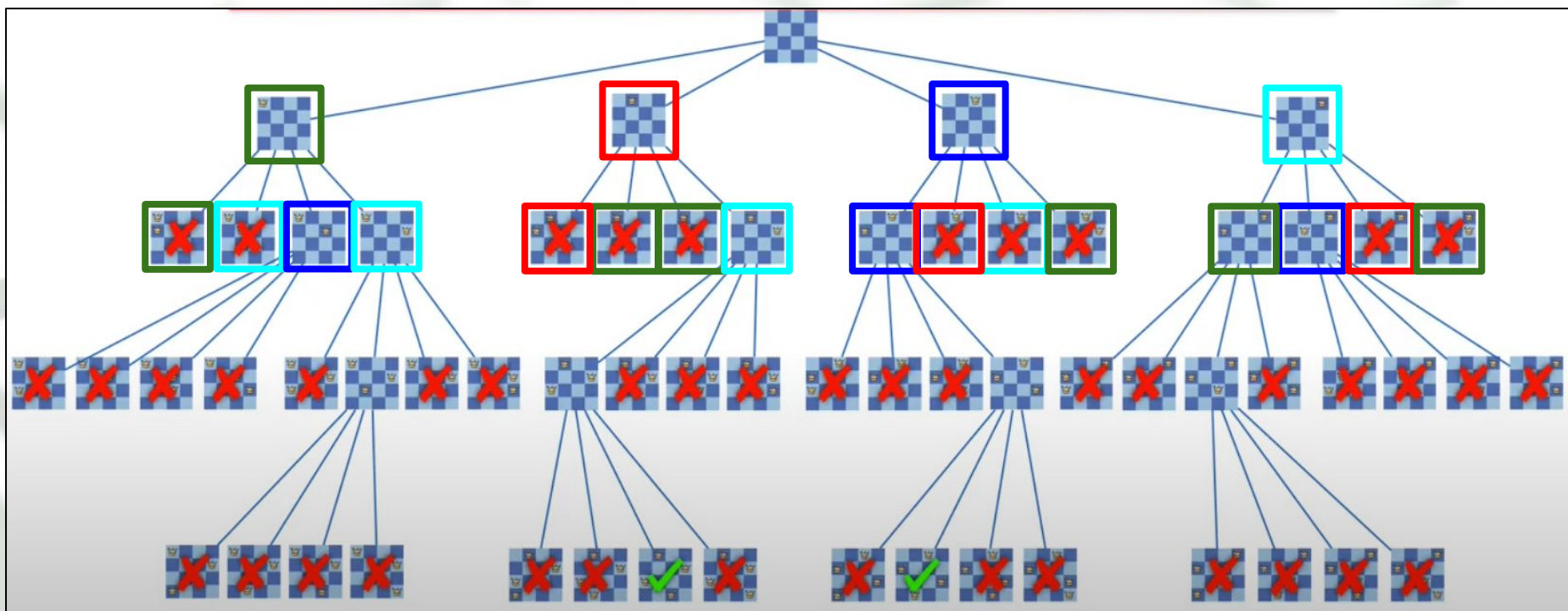




Algoritmo

■ Busca Exaustiva => Particionamento Dinâmico

- *Workers* trabalham dinamicamente para resolver o problema.

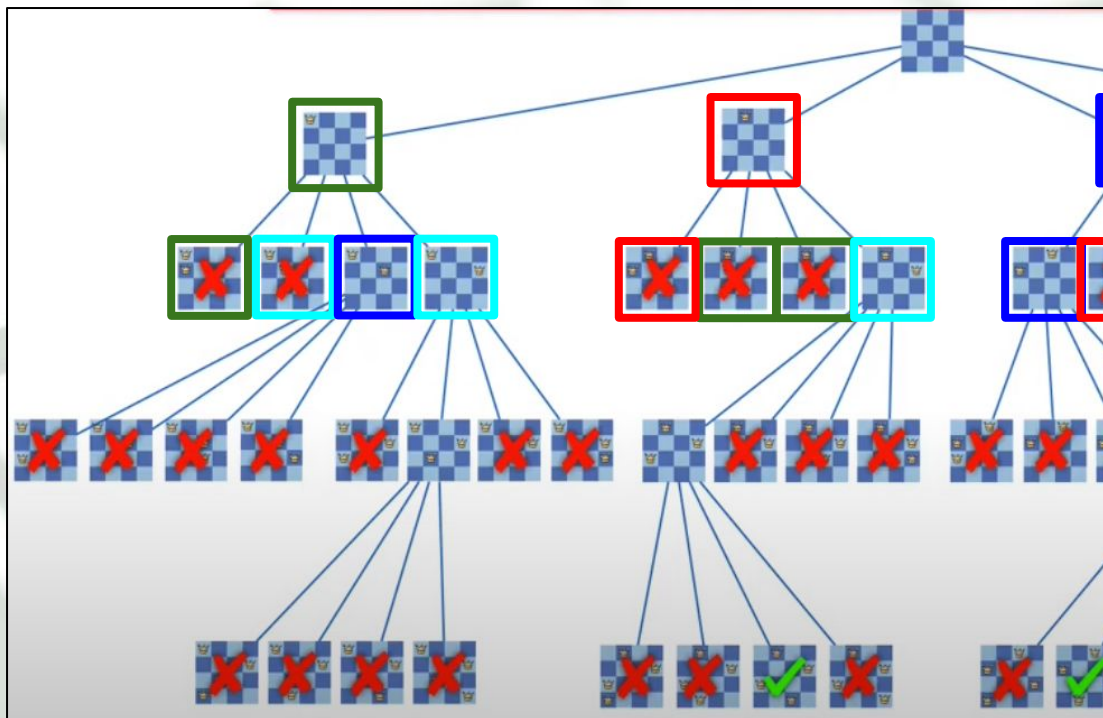




Algoritmo

■ Busca Exaustiva => Particionamento Dinâmico

- *Workers* trabalham dinamicamente p



Cada possibilidade é armazenada em uma fila de tasks, e os workers se revezam para tratar os jobs.

Vantagem:

Facilidade e simplicidade
&&

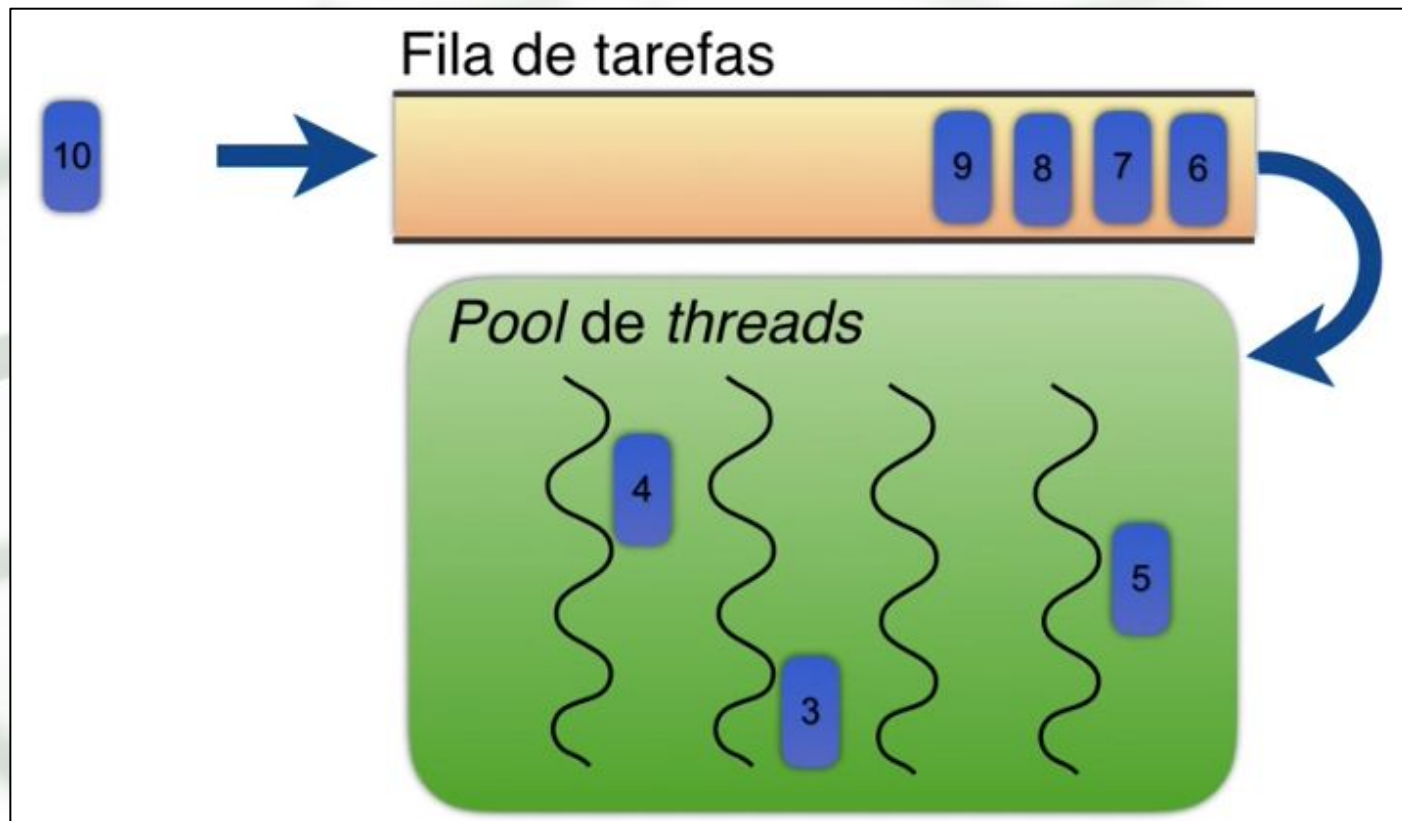
Baixo Overhead

&&

Balanceamento de Carga

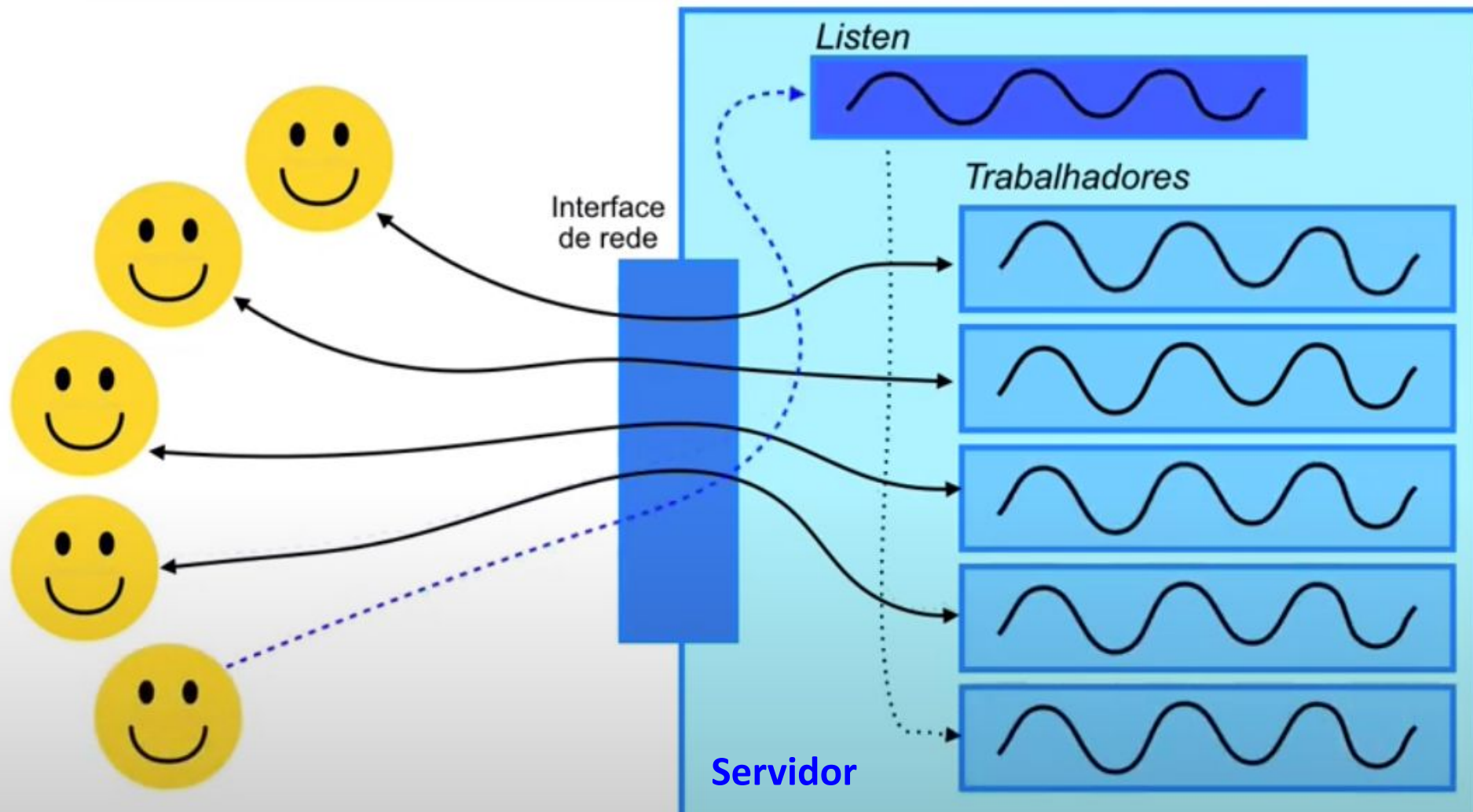


■ Fila de Tasks & Pool de Workers





Exemplo Prático





Implementação

- A implementação da técnica pode se dar por algumas alternativas...
- **Fila de Tasks:**
 - `multiprocessing.JoinableQueue()`
- **Pool de Workers:**
 - `concurrent.futures`



Joinable Queue

- A classe **JoinableQueue** é uma extensão da classe **Queue**, mas que implementa controles mais rígidos de acesso, inserção e remoção de dados, **ideal para repositório de tarefas (tasks) pendentes**.



Joinable Queue

- Existem dois novos métodos na **JoinableQueue**:

`queue.task_done()`

Que deve ser executado para cada `queue.get()`, indicando que a tarefa obtida foi processada e concluída.

`queue.join()`

Permite um processo ser bloqueado até a conclusão de todas as tasks da queue.



Laboratório #06.01

- Dada uma fila de tasks, onde cada task consiste em uma lista de 10 valores sorteados aleatoriamente, crie um programa que implementa paralelismo para realizar a ordenação de cada lista.
- Crie 02 processos para gerar tasks (300/processo).
- Crie os `.cpu_count()` processos para tratar cada task.
- Os resultados devem ser armazenados numa fila de resultados, e ser impressa ao final da execução.



Concurrent Futures

- A lib **Concurrent.Futures** (2011) é atualmente a principal biblioteca para o desenvolvimento de aplicações concorrentes/paralelas em Python.
- Oferece uma **interface de alto nível para desenvolvedores**, encapsulando e simplificando diversos mecanismos para criação, controle e sincronização de workers (*threads ou processos*).



Concurrent Futures

```
from concurrent.futures import  
ThreadPoolExecutor, ProcessPoolExecutor
```

- `workers = ThreadPoolExecutor(max_workers)`
 - Workers são instanciados como Threads
- `workers = ProcessPoolExecutor(max_workers)`
 - Workers são instanciados como SubProcessos



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import time

def do_something(i):
    time.sleep(i)
    return i*2

def when_finished(f):
    print(f'Callback => {f.result()}')

workers = ProcessPoolExecutor(5)

task = workers.submit(do_something, 4)

task.add_done_callback(when_finished)

print('Main => Callback não é bloqueante')

print('Main => Result é bloqueante...')
resp = task.result()
print(f'Main => {resp}')
```

**Implementação
de Callbacks**



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import time

def do_something(i):
    time.sleep(i)
    return i*2

def when_finished(f):
    print(f'Callback => {f.result()}')

workers = ProcessPoolExecutor(5)

task = workers.submit(do_something, 4)

task.add_done_callback(when_finished)

print('Main => Callback não é bloqueante')

print('Main => Result é bloqueante...')
resp = task.result()
print(f'Main => {resp}')
```

Retorno de Valores entre Processos



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import time

def do_something(i):
    time.sleep(i)
    return i*2

def when_finished(f):
    print(f'Callback => {f.result()}')

workers = ThreadPoolExecutor(5)

task = workers.submit(do_something, 4)

task.add_done_callback(when_finished)

print('Main => Callback não é bloqueante')

print('Main => Result é bloqueante...')
resp = task.result()
print(f'Main => {resp}')
```

Threads ou Processos...
A mesma Interface!



Concurrent Futures

- **Nova Meta: Divisão de tarefas de maneira dinâmica e eficiente** (*granularidade dinâmica*).
- **workers = ThreadPoolExecutor(4)**
 - Cria um Pool de Workers (até 4 threads neste caso)
- **workers.map(function, tasks_list)**
 - Cada WORKER disponível irá executar FUNCTION para a próxima TASK pendente.



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import threading
import time

def do_something(i):
    time.sleep(i)
    print(f'{threading.current_thread().name} => {i}')

workers = ThreadPoolExecutor(4)

tasks = [7, 6, 5, 4, 3, 2, 1, 0]
workers.map(do_something, tasks)

print('Main em execução concorrente!')

workers.shutdown()

print('Finalizado!')
```

Cada worker irá “pegar”
uma task para executar...



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import threading
import time

def do_something(i):
    time.sleep(i)
    print(f'{threading.current_thread().name} => {i}')

workers = ThreadPoolExecutor(4)

tasks = [7, 6, 5, 4, 3, 2, 1, 0]

workers.map(do_something, tasks)
print('Main em execução concorrente!')

workers.shutdown()
print('Finalizado!')
```

**Método map não é
bloqueante.**



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import threading
import time

def do_something(i):
    time.sleep(i)
    print(f'{threading.current_thread().name} => {i}')

workers = ThreadPoolExecutor(4)

tasks = [7, 6, 5, 4, 3, 2, 1, 0]

workers.map(do_something, tasks)

print('Main em execução concorrente!')

workers.shutdown() ←
print('Finalizado!')
```

**Método shutdown aguarda a
finalização de todas as tasks**



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import threading
import time

def do_something(i):
    time.sleep(i)
    print(f'{threading.current_thread()}')

workers = ThreadPoolExecutor(4)

tasks = [7, 6, 5, 4, 3, 2, 1, 0]

workers.map(do_something, tasks)

print('Main em execução concorrente!')

workers.shutdown()

print('Finalizado!')
```

```
ThreadPoolExecutor-0_3 => 4
ThreadPoolExecutor-0_2 => 5
ThreadPoolExecutor-0_1 => 6
ThreadPoolExecutor-0_0 => 7
ThreadPoolExecutor-0_0 => 0
ThreadPoolExecutor-0_1 => 1
ThreadPoolExecutor-0_3 => 3
ThreadPoolExecutor-0_2 => 2
```

Resultado



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import threading
import time

def do_something(i):
    time.sleep(i)
    print(f'{threading.current_thread()}

workers = ThreadPoolExecutor(4)

tasks = [7, 6, 5, 4, 3, 2, 1, 0]

workers.map(do_something, tasks)

print('Main em execução concorrente!')

workers.shutdown()

print('Finalizado!')
```

```
ThreadPoolExecutor-0_3 => 4
ThreadPoolExecutor-0_2 => 5
ThreadPoolExecutor-0_1 => 6
ThreadPoolExecutor-0_0 => 7
ThreadPoolExecutor-0_0 => 0
ThreadPoolExecutor-0_1 => 1
ThreadPoolExecutor-0_3 => 3
ThreadPoolExecutor-0_2 => 2
```

Resultado

Mas... e se for necessário armazenar os resultados da execução?



Concurrent Futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import threading
import time

def do_something(i):
    time.sleep(i)
    print(f'{threading.current_thread().name} => {i}')
    return i

workers = ThreadPoolExecutor(4)

tasks = [7, 6, 5, 4, 3, 2, 1, 0]

futures = workers.map(do_something, tasks)
print(Main em execução concorrente!')

resultados = list(futures)
print(f'Resultados = {resultados}')
```

Cada worker irá realizar um trabalho com resultado "futuro". Ao finalizar todos os trabalhos, obtemos a lista.

bloqueante



Concurrent Futures

```
ThreadPoolExecutor-0_3 => 4
ThreadPoolExecutor-0_2 => 5
ThreadPoolExecutor-0_1 => 6
ThreadPoolExecutor-0_0 => 7
ThreadPoolExecutor-0_0 => 0
ThreadPoolExecutor-0_2 => 2
ThreadPoolExecutor-0_1 => 1
ThreadPoolExecutor-0_3 => 3
Resultados = [7, 6, 5, 4, 3, 2, 1, 0]
```

lExecutor, ProcessPoolExecutor

e} => {i}')

```
futures = workers.map(do_something, tasks)
```

```
print(Main em execução concorrente!')
```

```
resultados = list(futures)
```

bloqueante

```
print(f'Resultados = {resultados}')
```

Cada worker irá realizar um trabalho com resultado "futuro". Ao finalizar todos os trabalhos, obtemos a lista.



Concurrent Futures

```
ThreadPoolExecutor-0_3 => 4
ThreadPoolExecutor-0_2 => 5
ThreadPoolExecutor-0_1 => 6
ThreadPoolExecutor-0_0 => 7
ThreadPoolExecutor-0_0 => 0
ThreadPoolExecutor-0_2 => 2
ThreadPoolExecutor-0_1 => 1
ThreadPoolExecutor-0_3 => 3
```

```
Resultados = [7, 6, 5, 4, 3, 2, 1, 0]
```

```
tasks = [7, 6, 5, 4, 3, 2, 1, 0]
```

```
lExecutor, ProcessPoolExecutor
```

```
e} => {i}')
```

OBSERVE! A lista de resultados obedece a ordem da fila de tasks, e não a ordem da execução em si.

```
futures = workers.map(do_something, tasks)
```

```
print(Main em execução concorrente!')
```

```
resultados = list(futures)
```

```
print(f'Resultados = {resultados}')
```



Concurrent Futures

- **concurrent.futures** torna **transparente** as soluções para problemas de acesso à região crítica (exclusão mútua) e comunicação entre processos (envio de parâmetros e resultados).





Laboratório #06.04

- Faça um programa que implementa:
 - Pool de 04 threads produtoras de tasks (números aleatórios entre 1 e 100), cada thread produz 10 tasks.
 - Pool de 08 threads consumidoras de tasks (para cada número obtido devolver como resultado a lista de seus divisores inteiros, exceto 1 e ele próprio).
 - Imprimir relação de tasks e o resultado dos processamentos.



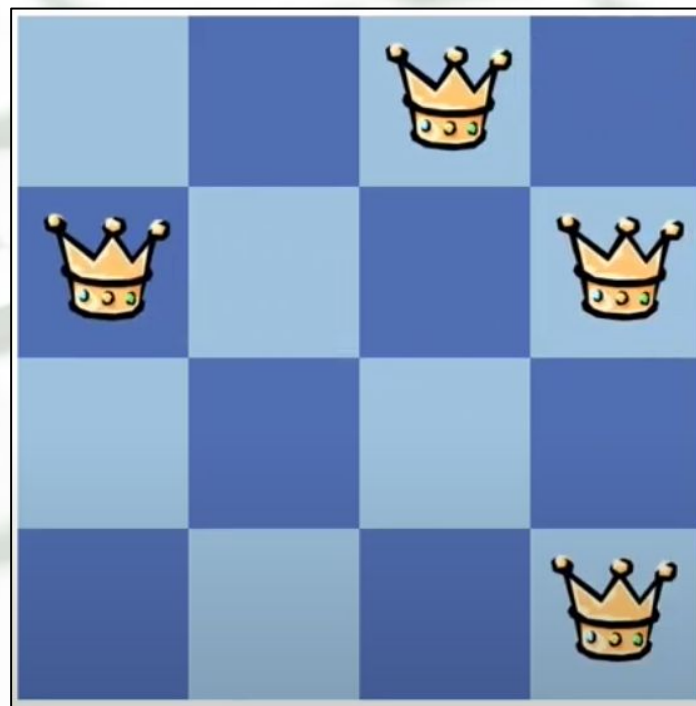
- Faça um programa
 - Pool de 04 threads para encontrar os divisores aleatórios entre 1 e 100
 - Pool de 08 threads para encontrar o maior e o menor número obtido e seus divisores
 - Imprimir relação de tarefas e resultados de processamento

```
Task 4 - Resultados = [2]
Task 47 - Resultados = []
Task 74 - Resultados = [37, 2]
Task 25 - Resultados = [5]
Task 24 - Resultados = [12, 8, 6, 4, 3, 2]
Task 46 - Resultados = [23, 2]
Task 99 - Resultados = [33, 11, 9, 3]
Task 70 - Resultados = [35, 14, 10, 7, 5, 2]
Task 85 - Resultados = [17, 5]
Task 91 - Resultados = [13, 7]
Task 86 - Resultados = [43, 2]
Task 32 - Resultados = [16, 8, 4, 2]
Task 75 - Resultados = [25, 15, 5, 3]
Task 51 - Resultados = [17, 3]
Task 31 - Resultados = []
Task 72 - Resultados = [36, 24, 18, 12, 9, 8, 6, 4, 3, 2]
Task 21 - Resultados = [7, 3]
Task 75 - Resultados = [25, 15, 5, 3]
Task 1 - Resultados = []
Task 43 - Resultados = []
Task 37 - Resultados = []
Task 90 - Resultados = [45, 30, 18, 15, 10, 9, 6, 5, 3, 2]
Task 41 - Resultados = []
Task 45 - Resultados = [15, 9, 5, 3]
```



Laboratório #06.05

- **Você consegue resolver o problema das “N Rainhas” usando Pool de Workers?**





Referências

- VAN STEEN, Maarten; TANENBAUM, Andrew S. Distributed systems. Leiden, The Netherlands: Maarten van Steen, 2017.
- MENDES, Eduardo. Lives de Python. YouTube Channel.
<https://github.com/dunossauro/live-de-python>
- GUEDES, Dorgival. Notas de aula, UFMG. YouTube Channel:
<https://www.youtube.com/channel/UCJQHsVoqmkygpOXtGfKECFw>