

# Actor Model Performance in Different Scenarios

1<sup>st</sup> João Bernardo Serrão Ganhão  
*Escola Superior de Gestão e Tecnologias de Santarém*  
*Instituto Politecnico de Santarém*  
Santarém, Portugal  
180100312@esg.ipsantarem.pt

2<sup>nd</sup> David do Rosário Alcobia Madeira Carreira  
*Escola Superior de Gestão e Tecnologias de Santarém*  
*Instituto Politecnico de Santarém*  
Santarém, Portugal  
200100294@esg.ipsanatem.pt

**Abstract**—O trabalho em questão tem como área de estudo programação avançada, tendo como objetivo comparar resultados de performance de três modelos, caracterizá-los e, por meio das mesmas, elaborar tabelas comparativas, escrever os pontos fortes e fracos de cada modelo e apresentar um possível resultado que consiste em mencionar um contexto possível onde esses modelos podem ser implementados.

**Index Terms**—Actores, Pool, Tempo, Resultados, model

## I. INTRODUÇÃO

Atualmente, têm-se verificado grandes avanços na área da computação que têm permitido a criação e desenvolvimento de sistemas mais eficientes e de alta performance com capacidade de lidar com grandes volumes de dados e realização de múltiplas tarefas em simultâneo. Estes avanços estão diretamente ligados à necessidade de otimização de desempenho em aplicações modernas. O modelo de atores (Actor Model) neste contexto apresenta-se com uma abordagem eficiente para lidar com a concorrência e a comunicação assíncrona entre processos. Esta abordagem pode apresentar diferentes níveis de desempenho derivados ao facto de ocorrer variações nos tipos de aplicação e no ambiente de execução.

Em síntese, este trabalho tem como objetivo analisar o desempenho do modelo de atores em diferentes cenários (Actor Model Performance in Different Scenarios), verificando os principais fatores que influenciam a sua eficiência.

## II. CONCEITOS

### A. Modelo de Atores

O modelo de atores é um modelo matemático de computação concorrente que propõe uma forma diferente de compreender a execução de programas quando múltiplas atividades ocorrem simultaneamente [1].

Diferentemente da programação tradicional, baseada no compartilhamento de dados entre diferentes partes do sistema, o modelo de atores segue um princípio fundamental: todas as entidades computacionais são atores, e a comunicação entre eles ocorre exclusivamente por meio do envio de mensagens assíncronas.

A motivação para o surgimento do modelo de atores está diretamente relacionada aos problemas enfrentados na programação concorrente tradicional. Em sistemas baseados em threads e memória compartilhada, é necessário utilizar mecanismos de sincronização, como locks, para evitar

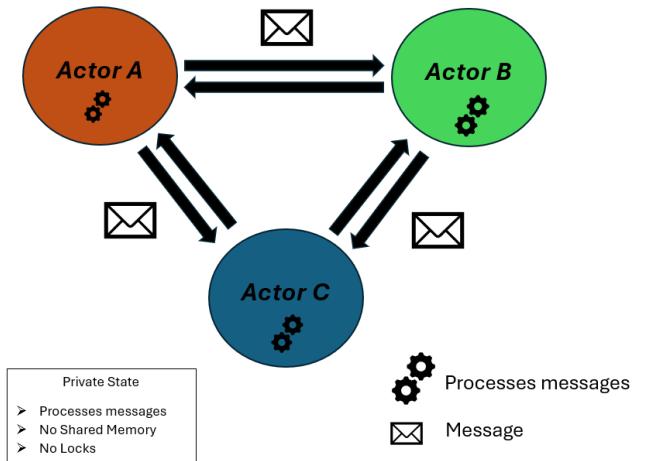


Fig. 1. Modelo de atores

condições de corrida (race conditions). No entanto, essa abordagem introduz diversos desafios, como a necessidade de lembrar explicitamente de adquirir e liberar locks, a possibilidade de deadlocks quando múltiplas threads aguardam indefinidamente umas pelas outras, e o aumento da complexidade do código, que se torna mais difícil de compreender e manter [2].

O modelo de atores aborda esses problemas ao eliminar completamente o uso de memória compartilhada. Em vez disso, cada ator possui seu próprio estado privado, que não pode ser acessado diretamente por outros atores [3].

A interação entre atores ocorre exclusivamente por meio do envio de mensagens, e cada ator processa essas mensagens sequencialmente, uma de cada vez. Essa característica garante que não haja acesso concorrente ao estado interno do ator, eliminando a necessidade de mecanismos explícitos de sincronização, como locks [4].

Dessa forma, o modelo de atores oferece uma abstração mais segura e modular para o desenvolvimento de sistemas concorrentes e distribuídos, reduzindo erros comuns associados à concorrência e facilitando o raciocínio sobre o comportamento do sistema como um todo [5].

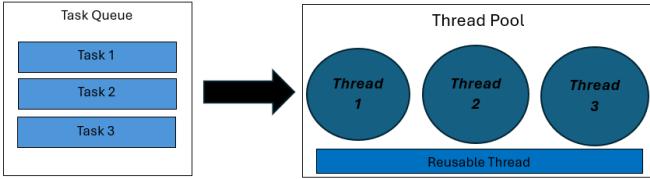


Fig. 2. Thread Pool

### B. Threads e Threads e Thread pools

Na programação concorrente tradicional, a execução paralela de tarefas é frequentemente realizada por meio de threads, que representam fluxos de execução independentes dentro de um mesmo processo e compartilham o mesmo espaço de memória [2]. Embora o uso de threads permita explorar paralelismo e melhorar o desempenho, ele também introduz desafios significativos, como condições de corrida, necessidade de sincronização explícita e risco de deadlocks quando múltiplas threads acessam dados compartilhados simultaneamente [6]. Além disso, a criação e destruição frequente de threads pode gerar um custo computacional elevado, impactando negativamente a eficiência do sistema.

Para mitigar esses problemas, utiliza-se o conceito de thread pool, que consiste em um conjunto fixo ou dinâmico de threads previamente criadas e reutilizadas para executar múltiplas tarefas ao longo do tempo [7].

Em vez de criar uma nova thread para cada tarefa, as tarefas são colocadas em uma fila e atribuídas a threads disponíveis no pool, reduzindo o overhead de gerenciamento de threads e melhorando a escalabilidade da aplicação [8]. Apesar dessas vantagens, o uso de thread pools não elimina os problemas inerentes à memória compartilhada, sendo ainda necessário empregar mecanismos de sincronização adequados para garantir a consistência dos dados [2], [6].

### III. METODOLOGIA

Neste capítulo vamos descrever a metodologia utilizada, através da apresentação de três abordagens diferentes, nomeadamente o modelo de atores, threads e threadingpool, tendo como objetivo uma comparação justa com diversos níveis de complexidade e cálculos. A primeira abordagem consiste numa versão tradicional de um sistema em que esta aplicado uma thread para cada tarefa, ou seja, cada thread tem como responsabilidade o processamento de um ficheiro. A segunda abordagem consiste na utilização de uma pool de threads com um número fixo de “workers”, criando a classe `thereadPoolExecutor` para fazer a gestão interna das tarefas. A terceira abordagem retrata basicamente uma versão das duas abordagens anteriores mas aplicando o modelo de atores, com foco principal na troca de mensagens sem que o seu estado interno seja partilhado.

Processo de execução definido nos seguintes passos:

- Definir o número de tarefas a testar.
- Executar a tarefa usando o modelo Thread por tarefa, registando tempo e memória.

- Executar a mesma tarefa usando o ThreadPool (5 workers).
- Executar a mesma tarefa usando o Modelo de atores (5 atores)
- Repetir os passos para os diferentes números de tarefas e para os três tipos de tarefa.
- Guardar todos os resultados numa tabela para comparação.

Para assegurar uniformidade, todos os testes foram realizados no mesmo computador, em condições idênticas (sem programas adicionais em funcionamento e utilizando a mesma versão do Python).

Características do computador:

- Memória RAM: 8 GB
- Processador: Intel® Core™ i5-10210U CPU @ 1.60GHz (até 2.11 GHz)
- Sistema operativo: Windows 10/11 (64 bits)
- Arquitetura: processador baseado em x64
- Armazenamento: 250 GB (disco disponível)
- Placa gráfica: 2 GB hgfd
- Versão: Python 3.13.9
- Versão: psutil 7.1.3
- Versão: pykka 4.4.0
- IDE: VsCode

Foram testados os seguintes números de tarefas:

- 100 tarefas
- 500 tarefas
- 1000 tarefas
- 5000 tarefas
- 10000 tarefas

Em cada execução foram registados:

- Tempo total de execução (em segundos) usando `time.time()`.
- Consumo de memória antes e depois do processamento, usando `psutil`.
- Número de resultados obtidos para confirmar que todas as tarefas foram processadas.

Cada teste foi repetido para os três tipos de tarefa e para cada modelo de execução.

### IV. RESULTADOS

Nomeadamente a análise sobre o nível de dificuldade sobre a escrita do código pode-se afirmar que existe diferenças significativas nos três modelos. No primeiro modelo que consiste na criação de uma thread por tarefa, verificou-se que a sua implementação foi mais intuitiva no ponto de vista da sua compreensão visto que cada tarefa está diretamente associada a uma thread, mas este modelo obriga a um esforço maior ao nível do controlo manual. Sendo necessário criar threads realizando a sua gestão através de listas e utilizando métodos como `join` para sincronização. Resumindo a medida que o número de tarefas aumenta o nível de complexidade aumenta e o número de erros torna-se maior. Nos três modelos apresentados o `ThreadPoolExecutor` demonstra ser a implementação mais simples e direta. A biblioteca fornece uma abstração maior, ou

seja, permite um foco maior na definição de tarefas a executar e o próprio executor foca-se na gestão das threads, reutilização e sincronização. Obtendo um código mais legível e compacto, reduzindo a probabilidade de erros de implementação. O modelo de atores com implementação da biblioteca Pykka mostrou ser mais complexo e difícil de entender e escrever, isto porque foi preciso compreender o funcionamento e conceito de atores, troca de mensagens e seu ciclo funcionamento, sendo que o nível de aprendizagem é mais elevado. Por outro lado, o modelo em si demonstra uma organização mais clara do código e ajuda a utilizar boas práticas na programação, permitindo uma evolução do sistema a longo prazo visto que apresentar uma melhor capacidade para um elevado número de tarefas.

#### A. Tabelas de Tempo (s):

Tabela 1 - Calculo simples

Nº ficheiros	Modelo de Atores	Thread por tarefa	ThreadPool
100	0.01 s	0.06 s	0.00 s
500	0.04 s	0.08 s	0.01 s
1000	0.07 s	0.20 s	0.01 s
5000	0.31 s	0.76 s	0.05 s
10000	0.62 s	1.60 s	0.10 s

Tabela 2 - Calculo de Ficheiros (CPU elevado)

Nº ficheiros	Modelo de Atores	Thread por tarefa	ThreadPool
100	0.01 s	1.71 s	1.59 s
500	0.03 s	8.40 s	8.50 s
1000	0.06 s	16.89 s	17.01 s
5000	0.32 s	79.41 s	79.51 s
10000	0.61 s	194.80 s	162.83 s

Tabela 3 - Operações de I/O

Nº ficheiros	Modelo de Atores	Thread por tarefa	ThreadPool
100	0.01 s	0.03 s	0.21 s
500	0.03 s	0.11 s	1.05 s
1000	0.06 s	0.17 s	2.15 s
5000	0.29 s	0.70 s	10.75 s
10000	0.58 s	1.49 s	21.52 s

#### B. Tabelas de Memória (s):

Tabela 1 - Calculo simples

Nº ficheiros	Modelo de Atores	Thread por tarefa	ThreadPool
100	+0.15 MB	+0.20 MB	+0.08 MB
500	+0.15 MB	+1.18 MB	+0.25 MB
1000	+0.13 MB	+2.14 MB	+0.50 MB
5000	+0.21 MB	+11.43 MB	+1.64 MB
10000	+0.16 MB	+21.70 MB	+8.47 MB

Tabela 2 - Calculo de Ficheiros (CPU elevado)

Nº ficheiros	Modelo de Atores	Thread por tarefa	ThreadPool
100	+0.04 MB	-0.57 MB	+0.01 MB
500	+0.04 MB	-0.59 MB	+0.01 MB
1000	+0.04 MB	-0.81 MB	+0.06 MB
5000	+0.03 MB	+0.15 MB	-0.28 MB
10000	-0.35 MB	+2.49 MB	+0.16 MB

Tabela 3 - Operações de I/O

Nº ficheiros	Modelo de Atores	Thread por tarefa	ThreadPool
100	+0.03 MB	+0.39 MB	+0.01 MB
500	+0.03 MB	+0.45 MB	-0.36 MB
1000	+0.04 MB	+0.55 MB	-0.42 MB
5000	-0.45 MB	+0.76 MB	-0.62 MB
10000	+0.05 MB	+2.71 MB	-0.41 MB

Tabela - Numero linhas de codigo

Nº Linhas de Codigo	Modelo de Atores	Thread por tarefa	ThreadPool
76	62	58	

A análise de resultados foca-se em mostrar as diferenças entre os três modelos de execução: Thread por tarefa, ThreadPool e Actor Model. Aplicando três tipos de tarefas distintas: cálculo simples, cálculo intensivo de CPU e operação de I/O. No tópico “tempo” o modelo de atores apresenta-se o mais eficiente dos três modelos de execução, demonstrando tempos muito baixos para todas as cargas de trabalho, mesmo testando com 10.000 tarefas. O modelo ThreadPool também revelou resultados satisfatórios para tarefas leves, mas apresenta ineficiência para tarefas de CPU intenso, onde ocorre um aumento de tempo de execução elevado. No modelo de thread por tarefa é o mais lento, especialmente quando ocorre aumento no número de tarefas, este acontecimento deve-se ao overhead de criação e gestão de milhares de threads. No que diz respeito à memória, o modelo de atores se destaca por apresentar um uso relativamente modesto e consistente, apresentando incrementos mínimos, mesmo com o aumento da carga de trabalho. O ThreadPool exibe um comportamento mais equilibrado e estável, apresentando pequenas variações de memória, enquanto o modelo de thread por tarefa demonstra um aumento significativo no consumo de memória em tarefas leves (especialmente em cálculos simples), o que sugere que a geração de diversas threads ao mesmo tempo requer mais recursos do sistema.

Para atividades de cálculo simples, tanto o modelo de atores quanto o ThreadPool apresentam tempos de execução bastante baixos, sendo que o ThreadPool é um pouco mais rápido em determinadas situações, enquanto o modelo de atores preserva um uso de memória mais controlado.

Em operações que exigem grande poder de processamento do CPU, o modelo baseado em atores se destaca de maneira significativa em relação aos outros dois, mostrando tempos de execução muito inferiores e um consumo de memória menor, ao passo que o ThreadPool e a abordagem de uma thread por tarefa demonstram tempos bastante maiores e uma capacidade de escalabilidade restrita.

Nas operações de I/O, o modelo baseado em atores ainda demonstra ser eficaz, sustentando baixos períodos de espera e consistência na memória, enquanto o ThreadPool começa a apresentar uma queda de rendimento à medida que a quantidade de tarefas cresce, indicando que sua habilidade de administrar tarefas de I/O simultaneamente é menor do que a do modelo de atores.

O código desenvolvido apresenta uma estrutura simples e bem organizada, com tarefas independentes que não compartilham recursos críticos entre as threads. A operação de I/O foi simulada usando a função sleep, eliminando potenciais falhas externas relacionadas a arquivos ou rede. Como resultado, nenhum bug foi identificado durante os testes, e os resultados demonstraram consistência em todas as execuções.

Motivos por não ocorrem bugs:

- funções simples e isoladas
- Não há partilha de dados entre threads
- Não existem operações reais de I/O
- A estrutura de execução está correta
- O código está a testar apenas o desempenho

Resumindo, os resultados mostram que o modelo de atores é o mais apropriado para gerir um alto volume de tarefas ao mesmo tempo, proporcionando uma mistura superior de eficiência e uso de memória. O ThreadPool representa uma opção interessante para executar tarefas de leve a média complexidade, porém apresenta restrições em situações que exigem alto uso de CPU ou operações de I/O em larga escala. O modelo que utiliza uma thread para cada tarefa é o menos eficiente em várias circunstâncias, sendo que é ideal apenas para um número limitado de tarefas devido ao alto custo relacionado à criação e administração de threads.

## V. DISCUSSÃO

Os resultados indicam que o modelo de atores (Actor Model) obteve um desempenho destacado em todos os testes, com destaque para situações em que o volume de tarefas crescia. Em operações simples e relacionadas a I/O, o tempo de execução permaneceu extremamente reduzido, enquanto o aumento no consumo de memória foi apenas moderado. Este comportamento pode ser atribuído à eficiência do modelo de atores na gestão de mensagens e à reutilização dos mesmos ao longo de todo o processo.

O modelo de uma thread por tarefa mostrou-se o menos eficiente, especialmente quando há um grande volume de tarefas. A criação de milhares de threads eleva significativamente o overhead do sistema, devido à gestão de contexto e memória, tornando esse modelo pouco adequado para lidar com cargas elevadas. Por outro lado, o ThreadPool também apresentou um desempenho satisfatório, especialmente em tarefas simples. No entanto, revelou limitações à medida que o número de tarefas aumentava ou quando estas demandavam processamento intensivo pela CPU. Isso se deve à presença do GIL no Python, que restringe o desempenho real em operações intensivas de CPU, mesmo com a utilização de múltiplas threads.

As limitações a nível técnico deste estudo envolvem sua realização em um único computador, utilizando um número fixo de threads/atores e tarefas simuladas. Em cenários reais, com operações genuínas de I/O e variadas cargas de trabalho, os resultados podem apresentar diferenças. Além das limitações técnicas do ambiente de execução, também houve desafios pessoais ligados à programação e ao entendimento dos

modelos concorrentes. O modelo de atores, em especial, demandou um esforço maior de aprendizagem devido à sua abordagem baseada na comunicação por mensagens e à arquitetura específica de atores, o que pode ter influenciado o formato da implementação. Além disso, o processo de implementar e depurar a concorrência, envolvendo threads e pools, exigiu atenção redobrada para prevenir problemas como deadlocks, perda de dados ou falhas de sincronização. Apesar dessas dificuldades, elas não comprometem os resultados obtidos, mas é importante considerá-las ao avaliar a eficiência e a praticidade de cada modelo.

## VI. CONCLUSÃO

De forma resumida, o modelo de atores se destacou como a solução mais eficiente para gerir um grande volume de tarefas concorrentes, oferecendo melhor desempenho e uma utilização de memória mais consistente. O uso de ThreadPool é uma opção adequada para tarefas de intensidade leve a moderada, embora mostre menor eficiência em operações que exigem alto processamento da CPU. Já o modelo de uma thread por tarefa é indicado exclusivamente para cargas pequenas, dado o elevado custo associado à criação e gestão de threads.

## REFERENCES

- [1] C. Hewitt, P. Bishop e R. Steiger, “A Universal Modular Actor Formalism for Artificial Intelligence,” Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI), Stanford, CA, USA, pp. 235–245, 1973
- [2] A. S. Tanenbaum e H. Bos, Modern Operating Systems, 4th ed. Upper Saddle River, NJ, USA: Pearson, 2015.
- [3] G. A. Agha, Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA, USA: MIT Press, 1986.
- [4] C. Hewitt, “What is computation? Actor Model versus Turing’s Model,” in A Computable Universe: Understanding and Exploring Nature as Computation, H. Zenil, Ed. Singapore: World Scientific, pp. 159–186, 2012.
- [5] J. Armstrong, Programming Erlang: Software for a Concurrent World. Dallas, TX, USA: Pragmatic Bookshelf, 2013.
- [6] B. Goetz et al., Java Concurrency in Practice, Addison-Wesley, 2006.
- [7] Wikipedia, “Thread pool.”
- [8] Microsoft, “Thread Pools,” Microsoft Learn.