Avaliação de Modelos de Paralelismo para Rastreabilidade de Dados em Saúde: Um Estudo de Caso sobre a Interpolação de Sinais Médicos

Regisson C. P. Aguiar[1], **Roberta A. A. Fagundes**[1]

[1] Departamento de Engenharia da Computação
Universidade de Pernambuco (UPE) – Recife, PE – Brazil

{regisson.aguiar, roberta.fagundes}@upe.br

Resumo. *O monitoramento de fluxos de dados em saúde é vital para garantir a integridade de predições críticas, como as do framework MedLens, exigindo alta precisão e baixa interferência no desempenho clínico. Contudo, a execução de rotinas de monitoramento em Python enfrenta gargalos de escalabilidade devido ao Global Interpreter Lock (GIL), que limita o paralelismo de threads em tarefas intensivas de CPU. Este trabalho apresenta uma avaliação de novas interfaces de monitoramento baseadas no módulo concurrent.futures (ThreadPoolExecutor e ProcessPoolExecutor) para a extração de logs de tempo em sinais médicos. A metodologia consistiu em um estudo de caso multi-core ($p = 6$) analisando métricas de Speedup (S_p) e Eficiência (E_p). Os resultados revelam comportamentos opostos: enquanto o modelo de threads sofreu degradação de desempenho ($S_p = 0,85$) devido ao overhead de troca de contexto, o modelo de processos alcançou ganho de escala real com $S_p = 3,14$, reduzindo o tempo de execução em mais de 295 segundos. Conclui-se que, dentre as abordagens avaliadas, o uso de processos demonstrou-se a alternativa mais eficaz para garantir que a rastreabilidade não comprometa a latência das predições clínicas, permitindo a adaptação da infraestrutura hospitalar às demandas de alto desempenho.*

1. Introdução

O avanço de modelos de aprendizado de máquina na área da saúde tem permitido predições cada vez mais precisas sobre a condição clínica de pacientes críticos. O framework MedLens [Ye et al. 2024] exemplifica essa evolução ao utilizar registros clínicos de alta granularidade provenientes de bases de dados como o MIMIC-III [Johnson et al. 2016], para a predição de mortalidade em Unidades de Terapia Intensiva (UTI) por meio da seleção rigorosa de sinais médicos e técnicas de regressão para o tratamento de dados ausentes. Entretanto, a aplicação prática de modelos preditivos complexos exige uma infraestrutura de software capaz de gerenciar a coleta de dados e a geração de logs de desempenho sem comprometer a latência da aplicação principal. Em cenários de Processamento de Alto Desempenho (PAD), a escalabilidade dessa camada de monitoramento torna-se um fator crítico.

O uso de Python como base para este desenvolvimento justifica-se por seu ecossistema robusto para a análise de dados estatísticos e ciência de dados [McKinney 2010]. Entretanto, a principal barreira para o paralelismo nesta linguagem é o Global Interpreter

Lock (GIL). O GIL é um mecanismo de sincronização do interpretador CPython que garante a segurança de memória por contagem de referências, permitindo que apenas uma *thread* execute *bytecode* Python por vez [Beazley 2010]. Embora essa restrição não afete tarefas de entrada/saída — nas quais as *threads* liberam o GIL voluntariamente durante operações bloqueantes —, ela serializa a execução em tarefas CPU-bound, impedindo o aproveitamento de múltiplos núcleos. Nesse cenário, *threads* adicionais competem pela aquisição do lock, gerando *overhead* de troca de contexto (*context switching*) que pode degradar o desempenho abaixo do baseline serial [Beazley 2010]. Essa restrição pode transformar o registro de métricas e o pré-processamento de sinais em um gargalo computacional [Batista et al. 2020]. A transição para um modelo de paralelismo baseado em processos, utilizando a interface ProcessPoolExecutor, surge como uma alternativa para contornar essa limitação, permitindo que a coleta de métricas e o processamento de sinais médicos ocorram de forma independente e paralela.

Neste contexto, este trabalho propõe o desenvolvimento e a avaliação de novas interfaces de monitoramento para fluxos baseados no MedLens, utilizando as abstrações da biblioteca concurrent.futures. O objetivo é prover suporte para execução via threads e processos, permitindo que o sistema se adapte à infraestrutura disponível e à complexidade das séries temporais médicas. A distinção entre esses modelos é essencial para garantir que a rastreabilidade dos dados não prejudique o desempenho da predição de mortalidade sob monitoramento.

A biblioteca `concurrent.futures` provê as interfaces `ThreadPoolExecutor` e `ProcessPoolExecutor` para lidar com essa problemática. A rotina de *Hourly Sampling* do MedLens segmenta as séries temporais em janelas de 1 hora e os dados ausentes são preenchidos por Regressão por Floresta Aleatória (*Random Forest Regression*) [Ye et al. 2024]. Ambas são CPU-bound e constituem o principal gargalo computacional do *pipeline* [Batista et al. 2020, Groner et al. 2012]. A avaliação fundamenta-se nas métricas de Speedup (S_p) e Eficiência (E_p) de [Eager et al. 1989], que expandem os limites teóricos da Lei de Amdahl [Amdahl 1967].

2. Metodologia

O desenvolvimento da ferramenta expandiu as capacidades do framework MedLens ao integrar as bibliotecas ThreadPoolExecutor e ProcessPoolExecutor para a extração de logs de tempo e análise de desempenho paralelo. Enquanto a versão original do MedLens operava exclusivamente com `ThreadPoolExecutor` para paralelizar as rotinas de *Hourly Sampling* e interpolação — submetendo cada série temporal de paciente a uma *thread* distinta —, essa abordagem não contorna o GIL em tarefas CPU-bound, o que motivou a introdução do suporte ao multiprocessamento via `ProcessPoolExecutor`. A ferramenta foi projetada para calcular o Tempo de Execução Médio (T_p) e o desvio padrão (σ), permitindo medir a consistência das rotinas de Hourly Sampling e interpolação de dados. As operações de log têm complexidade linear $O(n)$, tornando os resultados representativos para cenários de escala massiva [Cormen et al. 2009].

A experimentação compreendeu 5 execuções para cada nível de processamento ($p = \{1, 2, 4, 6\}$), buscando estabilidade estatística na coleta de dados durante a geração das séries temporais de 30 dias para os sinais médicos. A partir dos tempos coletados, foram calculados o Speedup (S_p) e a Eficiência (E_p) para ambos os modelos (Threads

e Processos). Essas métricas são essenciais para identificar o ponto de saturação dos núcleos e o ganho real de desempenho, evidenciando como a escolha da interface de execução influencia a escalabilidade do sistema em relação ao tempo de execução base.

3. Resultados e Discussão

Os experimentos foram conduzidos em um ambiente com processador *12th Gen Intel® Core™ i7-1270P*, 32 GB de RAM e 6 núcleos físicos, processando as séries temporais do MIMIC-III [Johnson et al. 2016]. Foram realizadas 5 execuções independentes para cada nível de concorrência ($p \in \{1, 2, 4, 6\}$), garantindo estabilidade estatística na coleta de dados.

Os experimentos revelaram comportamentos distintos entre as interfaces de execução em função das restrições do interpretador Python. Como observado na Tabela 1, e consolidado visualmente na Figura 1, os modelos de paralelismo apresentam trajetórias opostas de desempenho à medida que o grau de concorrência (p) aumenta.

Tabela 1. Comparativo de Desempenho entre Processos e Threads no MedLens

	Processos					**Threads**				
p	T_p **(s)**	σ	$\widetilde{T}_p$	S_p	E_p	T_p **(s)**	σ	$\widetilde{T}_p$	S_p	E_p
1	433,18	2,15	432,84	1,00	1,00	434,22	1,97	435,06	1,00	1,00
2	249,32	0,08	249,30	1,74	0,87	448,30	1,71	447,36	0,97	0,48
4	163,24	1,47	162,68	2,65	0,66	483,85	1,40	483,12	0,90	0,22
6	138,07	0,32	138,23	3,14	0,52	511,62	1,56	510,83	0,85	0,14

3.1. Análise da Escalabilidade Real com Processos

O `ProcessPoolExecutor` foi a única abordagem que apresentou ganho de escala real. O tempo de execução médio (T_p) caiu de 433,18s para 138,07s com 6 núcleos, representando uma redução drástica no tempo de processamento das séries temporais médicas. Esse resultado ocorre porque cada processo utiliza uma instância separada do interpretador Python, o que elimina a contenção de recursos e permite o paralelismo real em tarefas intensivas de CPU, como a interpolação por Random Forest Regression. Um ponto fundamental para a validação desta proposta é que a complexidade das operações de log é linear em relação ao volume de dados processados. Essa natureza algorítmica sugere que os ganhos observados no estudo de caso tendem a se manter em problemas de escala massiva, confirmando a robustez do modelo de processos para o monitoramento hospitalar em tempo real [Cormen et al. 2009].

3.2. Fenômeno da Degradação por Threads

É crucial detalhar por que o modelo de threads falhou, apresentando um comportamento oposto ao esperado para sistemas paralelos. Piora no desempenho: o T_p aumentou de 434,22s para 511,62s com o aumento de núcleos ($p = 6$), resultando em um Speedup (S_p) de 0,85, o que caracteriza uma degradação de desempenho em relação à execução

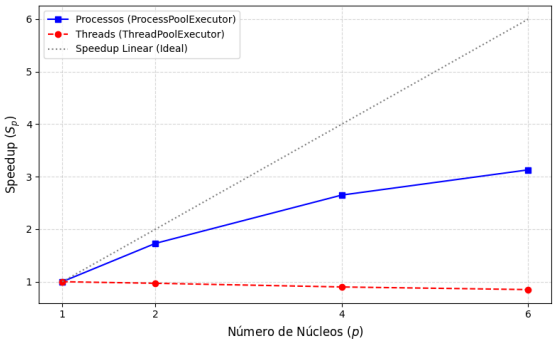

Figura 1. Comparação de Speedup entre Processos e Threads

serial. A causa raiz deste comportamento é o GIL, que serializa a execução de *bytecode* Python ao permitir que apenas uma *thread* detenha o lock por vez [Beazley 2010]. Em tarefas CPU-bound, as *threads* não liberam o GIL voluntariamente (o que ocorreria em operações de I/O), forçando trocas periódicas de contexto (*context switching*) gerenciadas pelo escalonador do sistema operacional. Cada troca implica salvar e restaurar o estado de registradores e pilha, além de disputas repetidas pela aquisição do lock, consumindo ciclos de CPU sem progresso computacional [Beazley 2010]. À medida que p aumenta, o número de contenções cresce proporcionalmente, explicando a degradação progressiva observada: S_p cai de 0,97 ($p = 2$) para 0,85 ($p = 6$), com T_p crescendo 77,40s em relação ao baseline serial. Na Figura 1, esse fenômeno é representado pela curva descendente, evidenciando a ineficiência do modelo frente à execução serial.

3.3. Análise de Eficiência e Impacto Clínico

Embora o modelo de processos tenha atingido um S_p de 3,14, a Eficiência (E_p) declinou para 0,52 em $p = 6$. Esse comportamento é atribuído ao custo de Comunicação Interprocessual (IPC) e à serialização de dados para centralização dos logs. Contudo, a redução de mais de 295 segundos no tempo de execução justifica esse trade-off de subutilização de hardware. Para o monitoramento do MedLens em UTIs, o uso de processos confirma-se, dentre as abordagens avaliadas, como a alternativa mais eficaz para garantir rastreabilidade precisa sem comprometer a latência crítica das predições de mortalidade.

4. Conclusão

Este trabalho validou o desenvolvimento de interfaces de monitoramento baseadas no módulo `concurrent.futures`, expandindo a rastreabilidade do MedLens para além das limitações do paralelismo por threads em Python. A análise revelou que o modelo de processos foi a única abordagem, dentre as avaliadas, capaz de proporcionar ganho de escala real ($S_p = 3,14$ com $p = 6$ núcleos), reduzindo o tempo de execução em mais de 295 segundos. Embora com menor eficiência ($E_p = 0,52$) devido ao custo de IPC, essa redução justifica sua adoção para garantir rastreabilidade precisa sem comprometer a latência crítica das predições de mortalidade em UTIs, fornecendo o embasamento técnico necessário para infraestruturas hospitalares de alto desempenho. Estes resultados demonstram que a escolha criteriosa da interface de execução, fundamentada na complexidade da carga de trabalho, é essencial para a escalabilidade e integridade de sistemas de suporte à decisão clínica baseados em machine learning.

Como trabalhos futuros, pretende-se investigar o uso de memória compartilhada para mitigar o overhead de comunicação e desenvolver um mecanismo de seleção adaptativa entre threads e processos. Por fim, busca-se validar a escalabilidade em sistemas distribuídos, testando a premissa de linearidade do custo de log em cenários massivos. Adicionalmente, pretende-se investigar a viabilidade de aceleração via GPU para as rotinas de interpolação, potencialmente contornando as limitações do GIL por meio de extensões que liberam o lock durante computação em hardware dedicado.

Referências

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30:483–485.

Batista, J. et al. (2020). Performance evaluation of parallel programming models in python: A case study. In *Proceedings of the Conference on Parallel and Distributed Computing*.

Beazley, D. (2010). Understanding the python global interpreter lock. Technical report, David Beazley Courses.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition.

Eager, D. L., Zahorjan, J., and Lazowska, E. D. (1989). Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423.

Groner, L. et al. (2012). A performance comparison of python interpreter implementations. In *Proceedings of the International Conference on High Performance Computing (HiPC)*.

Johnson, A. E., Pollard, T. J., Shen, L., Lehman, L.-w. H., Feng, M., Ghassemi, M., Moody, B., Szolovits, P., Celi, L. A., and Mark, R. G. (2016). Mimic-iii, a freely accessible critical care database. *Scientific Data*, 3(1):1–9.

McKinney, W. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56.

Ye, X., Wu, J., Mou, C., and Dai, W. (2024). Medlens: Improve mortality prediction via medical signs selecting and regression. *arXiv preprint arXiv:2305.11742v2*.