Uma análise de de ferramentas de multiprocessamento e *multithreading* para Python

Pedro H. de Almeida[1] **, Álvaro L. Fazenda**[1]

[1]Instituto de Ciência e Tecnologia — Universidade Federal de São Paulo (UNIFESP)
Campus São José dos Campos – SP — Brazil

{henrique.almeida, alvaro.fazenda}@unifesp.br

Resumo. *Este trabalho compara três ferramentas em Python para multiprocessamento e multithreading: Mpi4Py, Charm4Py e Numba, analisando sua eficácia no método de Jacobi aplicado a matrizes de tamanhos variáveis. A análise foi realizada com base no tempo de execução, utilizando métricas de Speedup e Eficiência, para diferentes números de processos/threads. Os resultados mostraram que Numba se apresentou superior dentre todas as ferramentas analisadas em termos de velocidade e escalabilidade. Por outro lado, a ferramenta com menor desempenho foi o Charm4Py, enquanto o Mpi4Py ficou como meio termo entre os seus comparativos.*

1. Introdução

Como previsto pela Lei de Moore, proposta por Gordon E. Moore (1965), o número de transistores em um circuito integrado dobra a cada, aproximadamente, dois anos [Moore 2006]. Tal fenômeno vêm sido observado até os anos atuais: "Clearly, transistor counts still follow the exponential growth line" [Rupp 2018]. Frente a este dilema, abordagens para Processamento de Alto Desempenho (do inglês, *High Performance Computing - HPC*) se evidenciam. Neste trabalho, compara-se 3 ferramentas em Python para para processamento paralelo: Mpi4Py, Charm4Py e Numba, sendo as duas primeiras voltadas para multiprocessamento e a última para *multithreading*.

Antes de prosseguir para o caso de estudo, ressalta-se as diferenças entre Mpi4Py e Charm4Py: apesar de ambos serem capazes de lidar com orientação a objetos, Mpi4Py não foi construído com esse propósito. Charm4Py por outro lado, é fortemente orientado a objetos, e construído a partir de objetos denominados *Chares*, garantindo maior abstração e prometendo maior escalabilidade. Também vale ressaltar que Charm4Py possui suporte para balanceamento dinâmico de carga, através da migração de *threads* entre unidades de processamento, quando se usa uma superdecomposição de domínio, com mais *threads* do que processadores. Apesar desta funcionalidade estar disponível, ela não foi usada neste caso de estudo.

2. Metodologia

Para a análise das ferramentas, seguiremos uma abordagem semelhante a proposta por Nunes [Nunes 2023]. O experimento é constituído por variações do método de Jaocbi, com matrizes quadradas $n \times n$, para valores de n iguais a 512, 1024 e 2048, com m processos e *threads*, variando entre 1, 2, 4, 8 e 16.

No contexto de computação em *stencil*, o método de Jacobi é aplicado em uma matriz, onde cada ponto da grade computacional é atualizado com base em uma média ponderada dos valores dos seus vizinhos adjacentes, Figura 1, até atingir um critério de convergência. As aplicações práticas deste metódo geralmente são voltadas para computação científica, processamento de imagens e em arquiteturas paralelas, podendo ser utilizado para se implementar uma grande variedade de operações como convoluções, filtro de imagens, processamento de sinais e simulações intensivas [Nunes 2023].

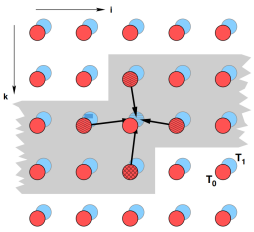

Figura 1. Visualização do método de Jacobi em 2D [Wellein et al. 2009].

As métricas utilizadas para análise serão o $Speedup$ e a Eficiência:

$$Speedup(P) = \frac{T_{exec}(1)}{T_{exec}(P)} \qquad Eficiência(P) = \frac{Speedup(P)}{P} \tag{1}$$

Onde, P representa o número de processos/*threads*, $T_{exec}(1)$ é o tempo de execução com um único processo (versão serial) e $T_{exec}(P)$ é o tempo de execução com P processos/*threads*.

A configuração do ambiente de testes está especificado a seguir:

- **OS:** Microsoft Windows 10 Pro 64 bits
- **CPU:** Intel(R) Core(TM) i7-10700KF CPU @ 3.80GHz (8 cores e 16 *threads*)
- **Python:** Versão 3.12.3
- **gcc:** (MinGW.org GCC-6.3.0-1) 6.3.0
- **Bibliotecas:** Numba (0.59.1), Charm4Py (1.0) e Mpi4Py(3.1.6)
- **MPI:** Microsoft MPI Startup Program [Version 10.0.12498.5]

3. Resultados

Todos os códigos utilizados para o testes podem ser encontrados no repositório online: `https://github.com/qedrohenrique/charm4py-vs-mpi-vs-numba`.
As Tabelas 1, 2 e 3, descrevem os resultados da média do tempo de execução para 5 rodadas, com o respectivo desvio padrão, para cada tamanho de matriz. As Tabelas 4, 5 e 6 descrevem os valores de *Speedup* e Eficiência para os mesmos casos.

Tabela 5. Speedup e Eficiência do método de Jacobi em uma matriz 1024x1024

Processos	Charm4Py		Mpi4Py		Numba	
	Speedup	Eficiência	Speedup	Eficiência	Speedup	Eficiência
1	1	1	1	1	1	1
2	1.71	0.86	1.78	0.89	1.94	0.97
4	2.25	0.56	2.80	0.70	2.80	0.70
8	2.03	0.25	2.91	0.36	3.23	0.40
16	0.90	0.06	1.16	0.07	3.48	0.22

Tabela 1. Tempo de execução do método de Jacobi em uma matriz 512x512

Processos	Tempo de execução ± Desvio Padrão em segundos			
	Charm4Py	Mpi4Py	Numba	Serial
1	2.4318 ± 0.0244	2.4744 ± 0.0721	2.5545 ± 0.03770	2.0355 ± 0.1807
2	1.8755 ± 0.0785	1.4143 ± 0.0398	0.9963 ± 0.2193	-
4	1.6326 ± 0.0406	0.9982 ± 0.0473	0.5253 ± 0.1482	-
8	2.3292 ± 0.1799	1.1009 ± 0.1422	0.512 ± 0.0035	-
16	6.2458 ± 0.2997	3.1557 ± 0.3182	0.394 ± 0.0683	-

Tabela 2. Tempo de execução do método de Jacobi em uma matriz 1024x1024

Processos	Tempo de execução ± Desvio Padrão em segundos			
	Charm4Py	Mpi4Py	Numba	Serial
1	9.7460 ± 0.0725	9.4305 ± 0.3496	8.9713 ± 0.0728	7.6811 ± 0.0501
2	5.6886 ± 0.1445	5.2923 ± 0.1230	4.6255 ± 0.5343	-
4	4.3404 ± 0.2400	3.3702 ± 0.1024	3.2020 ± 0.2997	-
8	4.8022 ± 0.1948	3.2354 ± 0.1112	2.7767 ± 0.1502	-
16	10.8742 ± 0.9021	8.1165 ± 1.2776	2.5782 ± 0.3478	-

Tabela 3. Tempo de execução do método de Jacobi em uma matriz 2048x2048

Processos	Tempo de execução ± Desvio Padrão em segundos			
	Charm4Py	Mpi4Py	Numba	Serial
1	41.9939 ± 1.2017	37.9197 ± 0.4729	36.5087 ± 0.0859	31.8358 ± 0.3082
2	25.1737 ± 0.2460	20.3484 ± 0.2788	18.8941 ± 0.4890	-
4	20.9186 ± 0.3521	15.0605 ± 0.0368	15.1097 ± 0.6511	-
8	19.9391 ± 0.7933	16.1490 ± 0.9043	14.9773 ± 0.7878	-
16	31.4449 ± 1.9490	35.0982 ± 2.4216	16.4116 ± 0.5240	-

Tabela 4. *Speedup* e Eficiência do método de Jacobi em uma matriz 512x512

Processos	Charm4Py		Mpi4Py		Numba	
	Speedup	Eficiência	Speedup	Eficiência	Speedup	Eficiência
1	1	1	1	1	1	1
2	1.30	0.65	1.75	0.87	2.17	1.09
4	1.49	0.37	2.48	0.62	4.86	1.22
8	1.04	0.13	2.25	0.28	4.99	0.62
16	0.39	0.02	0.78	0.05	6.48	0.41

4. Conclusão e trabalhos futuros

Este estudo comparou três ferramentas para multiprocessamento e *multithreading*: Mpi4Py, Charm4Py e Numba, utilizando o método de Jacobi aplicado a matrizes de diferentes tamanhos. A análise foi focada no desempenho restrito a medidas de tempo de execução para diferentes números de processos e *threads*.

Numba apresentou o melhor desempenho entre as três ferramentas, com tempos de execução menores, *speedups* e eficiências superiores. Sua vantagem se deve à menor

Tabela 6. Speedup e Eficiência do método de Jacobi em uma matriz 2048x2048

Processos	Charm4Py		Mpi4Py		Numba	
	Speedup	Eficiência	Speedup	Eficiência	Speedup	Eficiência
1	1	1	1	1	1	1
2	1.67	0.83	1.86	0.93	1.93	0.97
4	2.01	0.50	2.52	0.63	2.42	0.60
8	2.11	0.26	2.35	0.29	2.44	0.30
16	1.34	0.08	1.08	0.07	2.22	0.14

sobrecarga na criação de *threads*, que facilita a troca de dados por memória, evitando transmissões por mensagens. Charm4Py mostrou pior escalabilidade em comparação com Mpi4Py, registrando maiores tempos de execução. Uma possível explicação para isso pode estar na ineficiência da conversão de código para C++, já que Charm4Py é baseado no Charm++ [Galvez 2019]. Apesar disso, tanto Mpi4Py quanto Charm4Py oferecem processamento paralelo em sistemas de memória distribuída, ampliando a escalabilidade, recurso não disponível no Numba.

Trabalhos futuros podem explorar variações deste formato de teste, aferindo medidas de espaço em memória, observando a performance em sistemas distribuídos, implementando a funcionalidade de balanceamento de carga provida pelo Charm4Py e investigando se a perda de performance para maiores números de *threads* ou processos está atrelada ao sistema computacional usado nesta pesquisa ou a outros fatores como o próprio balanceamento de carga ou trechos sequênciais do código.

5. Agradecimentos

Esse trabalho contou com o suporte financeiro parcial dos projetos FAPESP $2023/00782-0$, $2023/00811-0$ e $2019/26702-8$.

Referências

Galvez, J. (2019). `https://charm4py.readthedocs.io/en/latest/index.html/`. Charm4Py Docs.

Moore, G. E. (2006). Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35.

Nunes, L. G. (2023). Multithread com python. Disponível em: `https://repositorio.unifesp.br/handle/11600/66871`. 38 f. TCC (Graduação) - Curso de Ciência da Computação, Instituto de Ciência e Tecnologia, Universidade Federal de São Paulo, São José dos Campos.

Rupp, K. (2018). `www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`. 42 Years of Microprocessor Trend Data.

Wellein, G., Hager, G., Zeiser, T., Wittmann, M., and Fehske, H. (2009). Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. volume 1, pages 579–586.