Paralelismo em Python: um comparativo entre diferentes abordagens para o problema da multiplicação de matrizes

Inessa Diniz Luerce[1], **Plínio Finkenauer Junior**[2]

[1]Bacharelado em Ciência da Computação
Centro de Desenvolvimento Tecnológico
Universidade Federal de Pelotas (UFPel) – Pelotas, RS – Brazil

[2]Bacharelado em Engenharia de Computação
Centro de Desenvolvimento Tecnológico
Universidade Federal de Pelotas (UFPel) – Pelotas, RS – Brazil

{idluerce,pfinkenauer}@inf.ufpel.edu.br

Abstract. *This work is an exploratory research about parallel programming in Python. To evaluate the viability we did three implementations of matrix multiplication: a) sequential; b) parallel, using native threads from Python and; c) parallel, using the API mpi4py. The matrices used for the calculation were of the 256x256 size. We run 100 times each algorithm, where we have used statistics methods for the speed-up evaluation. The results presented the MPI implementation as the one with the better performance.*

Resumo. *Este trabalho consiste de uma pesquisa exploratória sobre a utilização de programação paralela em Python. Para avaliação da sua viabilidade, foram realizadas três implementações para o problema de multiplicação de matrizes: a) sequencial; b) paralela utilizando threads nativas de Python e; c) paralela utilizando a API mpi4py. As matrizes utilizadas para cálculo eram de tamanho 256x256. Foram realizadas 100 execuções de cada implementação, onde foram utilizados métodos estatísticos para avaliação do speed-up. Os resultados apontaram a implementação de MPI como a com melhor desempenho.*

1. Introdução

Python é uma linguagem interpretada amplamente utilizada por cientistas nos dias atuais e seu número de usuários cresceu consideravelmente nos últimos anos. De acordo com relatórios do Github, Python é a terceira linguagem mais utilizada na plataforma, somando um total de mais de 164 mil repositórios ativos, ficando atrás apenas de Javascript e Java [Github 2017].

No contexto científico, ela tem se mostrado bastante popular, especialmente pelo fato de sua sintaxe ser simples e sua curva de aprendizado pequena. Além disso, conta com diversas bibliotecas disponíveis, o que contribui para que profissionais de outras áreas se interessem por programação, e possam implementar suas próprias soluções.

Entretanto, uma das carências da linguagem está relacionada ao paralelismo. Ainda que esse paradigma seja indispensável para as necessidades atuais exigidas pela computação, ele ainda não está implementado de forma robusta em Python.

O objetivo deste trabalho consiste em explorar o paralelismo disponível na linguagem Python em sua terceira versão. Para isso, é realizada uma comparação entre diferentes implementações para o problema de multiplicação de matrizes. Foram efetuadas as implementações: a) sequencial, b) utilizando o conceito nativo de Python para *threads* e c) utilizando uma implementação de *MPI (Message Passing Interface)* para Python, *mpi4py*.

O artigo está estruturado da seguinte forma: na Seção 2 é realizada a fundamentação teórica, onde aborda-se trabalhos relacionados que explorem paralelismo em Python. A seção 3 apresenta a metodologia empregada na implementação das soluções propostas e dos testes realizados. A análise dos resultados e discussão dos mesmos é exposta na seção 4. Por fim, a seção 5 apresenta as considerações finais.

2. Trabalhos Relacionados

De acordo com [Miller 2002b], algumas tarefas são resolvidas de forma bastante simples em linguagens de script, como Python. Entretanto, essas linguagens carecem de soluções para lidar com o paralelismo e acabam sendo pouco exploradas. Em virtude disso, ele desenvolveu o *pyMPI*, uma implementação distribuída de Python que extende o MPI. Além de mostrar a viabilidade da utilização de tal técnica, ele forneceu uma documentação introdutória para facilitar sua aplicação [Miller 2002a].

Estudos realizados por [Cai et al. 2005] visaram, primeiramente, buscar aplicações científicas que utilizassem Python. Foram exploradas diversas técnicas para verificar a eficiência de sua implementação serial nestas aplicações. A seguir, foram avaliadas implementações paralelas simples nas aplicações científicas.

Uma outra implementação de MPI para Python foi apresentada no trabalho de [Dalcín et al. 2005], que disponibilizou a API *mpi4py*, disponível em um repositório aberto onde a comunidade do software livre auxilia na detecção e correção de bugs [Dalcin 2017].

No trabalho realizado por [Singh et al. 2013] são apresentadas algumas abordagens utilizando programação paralela em Python para processamento de dados relativos à Astronomia. Foram propostos três diferentes problemas para serem avaliados em relação ao tempo de execução.

O trabalho aqui proposto baseia-se em um dos conceitos utilizados nestes trabalhos: comparar implementações sequenciais e paralelas, a fim de compreender primeiramente operações trivais para então, buscar soluções paralelas mais complexas. Foi escolhido utilizar a implementação do mpi4py, a fim de compará-la com a implementação sequencial e a nativamente paralela.

3. Metodologia

A metodologia empregada no trabalho foi, primeiramente, a escolha de um problema simples suficiente para ser facilmente paralelizado e compreendido. O problema definido foi a operação matemática envolvida no produto de duas matrizes, onde o tamanho de cada matriz é de 256x256.

A seguir foram realizadas três implementações, uma sequencial e duas paralelas, uma delas utilizando threads nativas de Python e a outra com a implementação

de MPi para Python, *mpi4py*. Foram efetuadas e registradas 100 execuções de cada implementação, onde foram analisadas a média, desvio padrão e os quartis correspondentes. A fim de validar os resultados, também realizou-se a análise da variância, através do teste f, seguida do teste de Tukey para comparações múltiplas (haja visto que houve diferença significativa entre as implementações).

3.1. Produto de matrizes

Para realizar a multiplicação entre duas matrizes é necessário que o número de colunas da primeira matriz seja igual ao número de linhas da segunda matriz. Realiza-se então, a soma da multiplicação de cada elemento da linha da primeira matriz pelo da segunda, como apresentado na figura 1.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} =$$

$$\begin{bmatrix} aj + bm + cp & ak + bn + cq & al + bo + cr \\ dj + em + fp & dk + en + fq & dl + eo + fr \\ gj + hm + ip & gk + hn + iq & gl + ho + ir \end{bmatrix}$$

Figure 1. Operações realizadas em uma multiplicação de duas matrizes de tamanho 3x3.

3.2. Implementações

Para a implementação sequencial, primeiramente foram inicializadas as duas matrizes de 256x256 com números aleatórios variando de 1 a 99. O algoritmo então realiza a operação de multiplicação ao decorrer de três loops do tipo *for*, onde percorre a linha, cada elemento da linha e a coluna.

Para a implementação paralela utilizando o módulo *threading* do Python foi definido o uso de duas threads. Seu procedimento para o cálculo foi semelhante ao sequencial, com o mesmo conceito para inicialização e caminho percorrido. Entretanto, as matrizes foram dividida em duas partes iguais, assim cada uma das threads ficou responsável pelo cálculo de uma metade. A primeira parte foi da linha 0 a 127 e a segunda da linha 128 a 255.

Por fim, a implementação com MPI também realizou a inicialização das matrizes com números aleatórios, porém seu cálculo foi realizado utilizando duas funções de multiplicação distintas. Elas percorrem os mesmos caminhos para o cálculo convencional (linha, elemento, coluna), porém uma delas utiliza um *offset* para definir seu área de abrangência, neste caso, a segunda metade da matriz. Cada rank fica responsável por calcular a metade da matriz, sendo o rank 0 que realiza a concateneção dos resultados.

3.3. Testes

Os testes foram realizados em um computador com processador Intel Core i7 CPU 960 3.20GHz, onde para cada implementação foram executadas 100 repetições. O tempo de

execução, medido em segundos, é calculado pelo próprio programa e corresponde apenas à execução dos cálculos, sendo o tempo despendido para a geração das matrizes desprezado. As implementações, testes, scripts e resultados estão disponíveis para consulta e reuso em [Finkenauer and Luerce 2017].

4. Resultados e Discussão

Conforme supracitado, foram realizadas 100 repetições para cada implementação desenvolvida. A descrição geral dos tempos de execução para cada implementação está representada na Tabela 1. A unidade de medida para os valores apresentados é segundo (*s*). Observa-se que entre as implementações paralelas, a versão utilizando o conceito de *thread* nativo da linguagem Python, obteve um desempenho mais lento que o algoritmo sequencial.

Table 1. Descrição dos tempos de execução dos algoritmos, por implementação.

Implementação (n = 100)	Média	DP	Mín	Q1	Md	Q3	Máx
Sequencial	4,21	0,061	4,10	4,16	4,20	4,25	4,40
Threads	5,97	0,241	4,92	5,84	6,05	6,14	6,24
MPI4py	2,38	0,048	2,30	2,34	2,37	2,40	2,53

Nota: DP= desvio padrão; Min= menor valor; Q1= primeiro quartil; Md= mediana; Q3= terceiro quartil; Max= maior valor. n = número de repetições.

A Tabela 2 expressa as médias gerais de tempo para cada implementação, além dos valores de *Speedup* (Sp). Sp é uma medida do grau de desempenho e é definido como o ganho de tempo de uma tarefa executada em paralela quando comparada à execução sequencial [Scott et al. 2005]. Calculou-se o *Speedup* a partir dos tempos de execução dos algoritmos. A fórmula empregada foi $\frac{Tseq}{Tpar}$.

Table 2. Média dos tempos de execução, por implementação dos algoritmos, e resultados de *Speedup*.

Implementação	Médias (s)	*Speedup*
Sequencial	4,21	-
Threads	5,97	0,71
MPI4py	2,38	1,76
Média Marginal	4,18	-

Conforme observado, a implementação com *MPI4py* obteve *speedup* maior que 1, constatando sua superioridade em comparação com a versão sequencial. Em contraparida, a versão *thread* obteve o valor de Sp menor que 1, confirmando que a paralelização nativa não proporcionou melhora no tempo de execução. Isso pode estar relacionado ao *Global Interpreter Lock*, um mutex que protege o acesso aos objetos em Python e previne que múltiplas threads executem Python bytecodes ao mesmo tempo. Este lock é necessário pois o gerenciamento de memória não é *thread-safe* [Python.org 2017]. Assim, ao analisar a execução do programa pode-se observar que os processadores nunca executavam em sua capacidade total, sempre chegavam a no máximo 50%.

Visando verificar a significância da desigualdade entre os tempos médios de execução dos algoritmos, realizou-se a análise de variância. A hipótese sob verificação

nesta análise supõe que as variâncias de cada implementação estimam o mesmo parâmetro. O valor observado para a estatística F expôs diferenças significativas na comparação das médias das três implementações, rejeitando, assim, a hipótese de nulidade.

Dessa forma, a significância dessas diferenças de tempo foi verificada através do teste de Tukey para comparações múltiplas. A Tabela 3 expõe os resultados obtidos pelo teste de Tukey. Observa-se que todas as implementações diferem-se entre si, e que os tempos de execução do algoritmo paralelo com *MPI4py* foram significamente inferiores aos apresentados pelos demais algoritmos, ao nível α = 0,05.

Table 3. Resultados para o teste de significância para a desigualdade entre os tempos.

Implementação	Médias		Variância
Sequencial	4,21	a	0,0037
Threads	5,97	b	0,0580
MPI4py	2,38	c	0,0023

Nota: Médias seguidas de mesmas letras não diferem entre si pelo teste de Tukey (α= 0,05).

A variabilidade entre os tempos de execução foi bastante reduzida, indicando alta homogeneidade entre os valores. A Figura 2 apresenta um gráfico de caixa no qual se observa maior variância entre os tempos da versão implementada com *threads*. Além disso, observa-se uma presença maior de valores discrepantes nessa implementação em relação às demais.

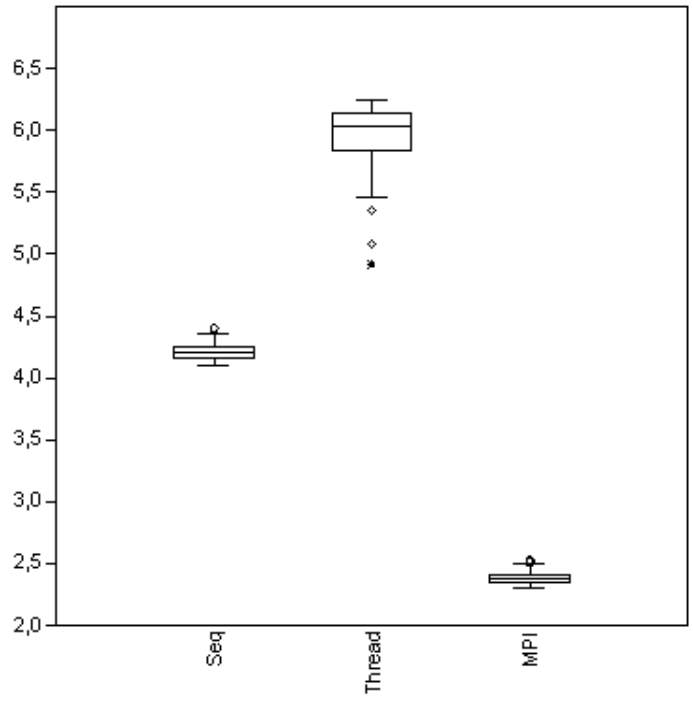

Figure 2. *Boxplot* para a variância dos tempos de execução, por implementação.

5. Considerações Finais

Uma vez que a proposta do trabalho consistia em explorar os recursos de paralelismo oferecidos por Python, acredita-se que o resultado tenha sido alcançado, visto que nos testes realizados, uma das implementações paralelas obteve desempenho superior ao da implementação sequencial.

Para trabalhos futuros, pretende-se aumentar a complexidade do problema proposto e realizar uma nova comparação entre a implementação sequencial e paralela (baseada em MPI), onde também serão exploradas outras configurações para o número de threads.

References

Cai, X., Langtangen, H. P., and Moe, H. (2005). On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming*, 13(1):31–56.

Dalcin, L. (2017). Mpi for python. Acessado em: 18 jul 2017.

Dalcín, L., Paz, R., and Storti, M. (2005). Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115.

Finkenauer, P. and Luerce, I. D. (2017). Parallel python. Acessado em: 20 ago 2017.

Github (2017). A small place to discover languages in github. Acessado em: 28 jul 2017.

Miller, P. (2002a). pympi: An introduction to parallel python using mpi. *Livermore National Laboratories*, 11.

Miller, P. J. (2002b). Parallel, distributed scripting with python.

Python.org (2017). Global interpreter lock. Acessado em: 02 jul 2017.

Scott, L. R., Clark, T., and Bagheri, B. (2005). *Scientific Parallel Computing*. Princeton University Press, Princeton, NJ, USA.

Singh, N., Browne, L.-M., and Butler, R. (2013). Parallel astronomical data processing with python: Recipes for multicore machines. *Astronomy and Computing*, 2:1–10.