Comparação entre algoritmos sequencial e paralelo utilizados na resolução de multiplicação entre matrizes

Taylan Branco Meurer[1], Leandro Loffi[1], Rodrigo Curvello[2]

[1]Acadêmico Ciência da Computação – Instituto Federal Catarinense – Campus Rio do Sul (IFC)

[2]Professor Ciência da Computação – Instituto Federal Catarinense – Campus Rio do Sul

tbmeurer@gmail.com, leandroloffi3@gmail.com, rodrigo.curvello@ifc-riodosul.edu.br

Abstract. *This paper compares the performance of serial and parallel algorithms . The algorithms perform a multiplication of square matrices and will be written in C , Java and Python. Parallelism was performed by OpenMP library, Jomp , the multiprocessing and Cython language. The runtimes were obtained by the time command from a bash terminal. The metric used is based on Amdahl Act, so the language with better performance in parallel has the largest speedup . The algorithms were run on a machine with Intel Core I7-3630QM with SSD (mSATA) and Debian stretch Operating System and Gnome 3.20.*

Resumo. *Neste trabalho compara-se o desempenho de algoritmos seriais e paralelos. Os algoritmos realizam uma multiplicação entre matrizes quadradas e serão escritos em C, Java e Python. O paralelismo foi efetuado através da biblioteca OpenMP, do Jomp, do Multiprocessing e da linguagem Cython. Os tempos de execução foram obtidos por meio do comando time de um terminal bash. A métrica utilizada é baseada na Lei de Amdahl, portanto a linguagem com melhor desempenho em paralelo tem o maior speedup. Os algoritmos foram executados em uma máquina com Intel Core I7-3630QM, com SSD (mSATA) e com Sistema Operacional Debian stretch e Gnome 3.20.*

1. Introdução

O trabalho trata sobre serialização e paralelismo de algoritmos. Os algoritmos em série são aqueles executados linha após linha por um único processador e algoritmos paralelos são aqueles executados por dois ou mais processadores ao mesmo tempo.

As linguagens de programação paralela surgiram como um novo recurso para facilitar a tarefa de construir programas que utilizem os recursos de paralelismo de equipamentos com arquitetura de computação paralela [Sato]. Para utilizar plenamente os recursos de um equipamento multiprocessado, existem basicamente duas opções, construir programas com controle sobre os trechos de paralelismo, utilizando recursos de uma Linguagem de Programação para a criação das threads e processos necessários, ou utilizar-se de uma biblioteca, a qual, normalmente, é composta por diretivas de compilador para uma linguagem de programação pré-existente, como C.

Desde a criação dessas arquiteturas, surgiram diversas bibliotecas ou APIs, entre elas a OpenMP, que será utilizada nesse trabalho. O desenvolvimento desta API foi realizado através de um trabalho colaborativo entre os diversos parceiros interessados no

projeto, entre eles: Compaq/Digital, Hewlett-Packard, Intel, IBM, KAI, Silicon Graphics, Sun entre outros [Sato].

O trabalho busca analisar e comparar o desempenho entre as linguagens Java, Python e C para execução de um algoritmo de multiplicação entre matrizes. As matrizes serão quadradas e terão ordem 500, 1500 e 2000. O desempenho será avaliado com emprego da lei de Amdahl e com o comando time do shell GNU/Linux em bash.

Conforme Albuquerque[1] (2004), a lei de amdahl é uma maneira de expressar o speedup máximo como uma função da quantidade de paralelismo e da fração da computação que é inerentemente sequencial. O speedup máximo (S) alcançado por um computador paralelo com p processadores executando a computação é:

$$S \leq \frac{1}{f + \frac{(1-f)}{p}} \tag{1}$$

A linguagem C fará uso da API OpenMP, a Java do Jomp e a Python do OpenMP com Cython.

O trabalho irá no capítulo seguinte tratar sobre a API OpenMP e a linguagem Cython. Em seguida, abordará a metodologia empregada para a construção do trabalho. Posteriormente, apresentará os resultados e, por último, relatará as considerações finais e as referências bibliográficas.

2. Processamento Paralelo

Processamento paralelo consiste na divisão de uma determinada tarefa em tarefas menores e na execução de cada uma dessas tarefas em diferentes processadores [Costa and Sena 2008]. Conforme [William 2010], existem dois conceitos de paralelismo: de instruções e de máquina. O paralelismo de instruções existe quando as instruções de uma sequência são independentes e o de máquina é definido como uma medida da capacidade do processador em aproveitar o paralelismo no nível de instruções.

O processamento paralelo pode acontecer em ambientes de memória distribuída ou compartilhada. Segundo [Costa and Sena 2008], processamento paralelo em memória distribuída consiste na distribuição de tarefas para serem executadas em diferentes processadores com recursos próprios de memória. Processamento paralelo em ambientes de memória compartilhada consiste na divisão das tarefas entre vários processadores que compartilham o mesmo recurso de memória global.

Nesse capítulo será apresentado de forma genérica informações sobre OpenMP, Multiprocessing e Cython.

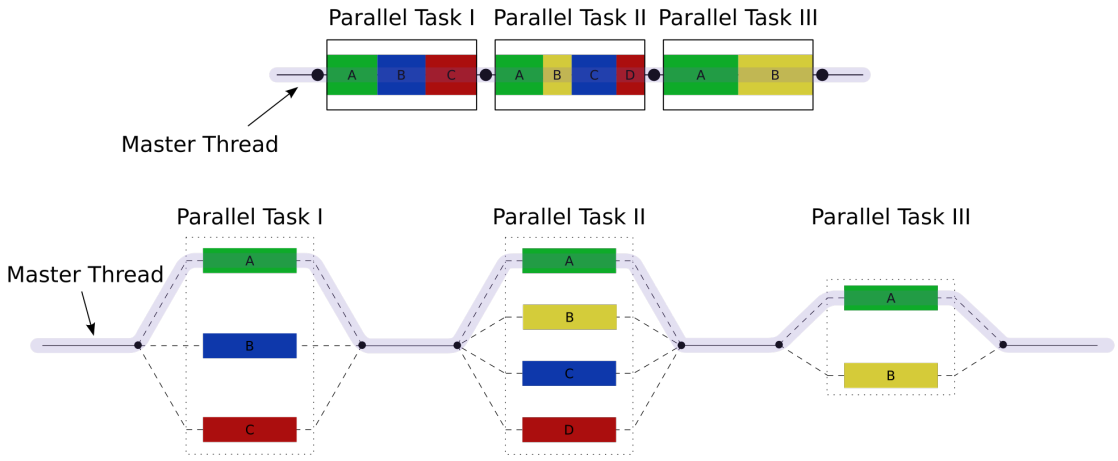
2.1. OpenMP

A API OpenMP oferece um conjunto de diretivas para a programação paralela em sistemas multiprocessados com memória compartilhada, realizando para tal a criação e o controle de Threads [Sato]. As diretivas da OpenMP podem ser incluídas em programas escritos nas linguagens "C" e FORTRAN para especificar pedaços de programas que devem ser executados em paralelo.

[1] ALBUQUERQUE, Jones. **Lei de Amdahl.**. Recife: DFM-UFRPE: 2004

O Open specification for Multiprocessing, simplesmente OpenMP, é um modelo de programação em memória compartilhada, que faz uso da especificação Pthreads. Ele surgiu a partir da cooperação de grandes fabricantes, como: Sun, IBM, Intel, AMD, HP, etc. Projetada para operar em C e Fortran, as especificações são diretivas que dizem ao compilador como gerar códigos paralelos.

Figura 1. Paralelismo.



Fonte: OpenMP[2]

Conforme a figura 1, pode-se visualizar a forma como pode ser realizado a divisão e como funciona um programa com tarefas paralelas. A primeira tarefa é executada por 3 núcleos, a segunda tarefa em 4 núcleos e, por último, a terceira tarefa foi dividida para ser executada em paralelo por 2 núcleos. Entre as tarefas, existe uma área comum serial, a qual pode representar um local de memória compartilhada.

Internamente a OpenMP trabalha com um modelo "FORK-JOIN" onde a partir de uma "thread master" (aquela que iniciou o programa) cria-se um time de threads para a realização de tarefas em paralelo, e depois é realizado um join onde todas as threads são sincronizadas e destruídas, sobrando somente a master thread.

O algoritmo 1 demonstra um exemplo básico do uso de uma instrução em paralelo com OpenMP. A partir do algoritmo é possível verificar o emprego das diretivas e dos parâmetros da API. Na linha 2, verifica-se a diretiva do construtor paralelo, o mais importante do OpenMP, uma vez que é o responsável pela indicação da região do código que será executado em paralelo [Costa and Sena 2008]. Ainda na linha 2, estão inclusas três cláusulas do construtor, a saber, shared, private e num_threads. O shared diz respeito às variáveis que são compartilhadas, o private recebe como parâmetros as variáveis privadas e o num_threads recebe o número de threads que serão utilizadas para execução paralela. Na linha 4 e 12, emprega-se o construtor for, que faz com que as iterações da estrutura de repetição situada logo abaixo da diretiva sejam executadas em paralelo [Costa and Sena 2008].

A próxima seção irá contemplar breves informações sobre o módulo de paralelismo para Python - multiprocessing e Cython.

[2]Disponível em:<https://upload.wikimedia.org/wikipedia/commons/thumb/f/f1/Fork_join.svg/2000px-Fork_join.svg.png> Acessado em: 05 de julho de 2016.

2.2. Multiprocessing

Multiprocessing[3] é um módulo Python para processamento paralelo. Ele faz uso de uma API semelhante ao threading. O pacote multiprocessing oferece concorrência local e remota, contornando o Global Interpreter Lock através de subprocessos no lugar de threads. Por essa razão, esse módulo permite ao programador tirar proveito total dos múltiplos processadores em uma determinada máquina. O multiprocessing também apresenta APIs com características distintas do módulo threading. Um bom exemplo disso é o objeto Pool, o qual oferece um meio conveniente de paralelização por meio de uma função com vários valores de entrada, distribuindo esses dados de entrada nos processos (paralelismo de dados).

2.3. Cython

Cython é um compilador estático otimizado para Python [Behnel et al. 2011]. A linguagem Cython[4] é um superconjunto da linguagem Python que suporta chamar funções e declarar tipos de dados C. Isso permite ao compilador gerar códigos em C eficientes a partir de um código Cython. Quase todo código em Python é válido em Cython.

Com o emprego do Cython, garante-se, no mínimo, 25% de ganhos com relação ao Python puro. A virtude do Cython é a sua capacidade de usar variáveis e parâmetros como tipos de dados em C. As variáveis e parâmetros podem ser misturadas sem afetar a execução [Behnel et al. 2011]. Além disso, o Cython suporta paralelismo nativo através do módulo **cython.parallel**, que emprega a API OpenMP.

3. Algoritmo

A seguir será demonstrado o pseudocódigo do algoritmo utilizado no trabalho para execução paralela em C. Para obter o serial basta remover as diretivas do openmp: #pragma omp argumento.

4. Metodologia

A metodologia faz uso da linguagem C, Java e Python (versão 2.7). Cada linguagem é composta por dois algoritmos, um serial e outro paralelo. O código paralelo em C é implementado com uso da API OpenMP. Em Java, o paralelismo é feito com Jomp e em Python com multiprocessing e Cython.

O algoritmo implementa uma multiplicação entre matrizes. Essa implementação é bastante conhecida e generalista. Além disso, exige grande esforço computacional. Os tempos de execução foram obtidos por meio do comando de terminal *time*[5]. Este comando registra o tempo de CPU utilizado e retorna o valor em segundos.

Os testes foram executados em uma máquina Intel Core I7 3630QM[6], CPU 3.4 GHz, 6MB de cache. A máquina possui 8GB de memória RAM e faz uso de um SSD mSATA. O Sistema Operacional instalado é o Debian stretch com kernel 4.6.0-1-amd64 e Gnome.

[3]Disponível em:<https://docs.python.org/2/library/multiprocessing.html> Acessado em: 23 de junho de 2016.

[4]Disponível em:<http://cython.org/> Acessado em: 05 de julho de 2016

[5]Disponível em: <http://ss64.com/bash/time.html> Acessado em: 22 de junho de 2016.

[6]Disponível em:<http://ark.intel.com/pt-br/products/71459/Intel-Core-i7-3630QM-Processor-6M-Cache-up-to-3_40-GHz> Acessado em: 20 de junho de 2016.

Algoritmo 1: algoritmo paralelo

Entrada: Definir variáveis (i,j,k) e a constante SIZE
Inicializar as matrizes (A,B,C) de ordem SIZE
Definir número de threads (NUM)

```
1  início
2      #pragma omp parallel shared(A,B,C) private(i,j,k) num_threads(NUM)
3      {
4      #pragma omp for
5      para cada i até SIZE faça
6          para cada j até SIZE faça
7              A[i][j] = 3 * i + j
8              B[i][j] = i + 3 * j
9              C[i][j] = 0
10         fim
11     fim
12     #pragma omp for
13     para cada i até SIZE faça
14         para cada k até SIZE faça
15             para cada j até SIZE faça
16                 C[i][j] = C[i][j] + A[i][k] * B[k][j]
17             fim
18         fim
19     fim
20     }
21 fim
```

5. Resultados e discussões

Os resultados foram atingidos por meio do comando time para o terminal em bash. Um exemplo de execução é demonstrado a seguir: $\$ \; time \; python \; programa.py$

O comando time retorna o tempo de execução do programa em segundos. A definição do tempo de execução se deu conforme a equação 2.

$$T_{medio} = \frac{T_1 + T_2 + T_3}{3} \tag{2}$$

Uma vez descoberto os tempos de execução, o próximo cálculo realizado foi o do speedup. Este demonstra o ganho de desempenho obtido com a paralelização. A equação 1 demonstra como se deu o procedimento de obtenção do speedup.

Os tempos de execução e os speedups calculados serão demonstrados a seguir por meio de tabelas e gráficos.

Tabela 1. Tempo de execução – C

Matriz(ordem)	Serial(s)	Paralelo(s)
500	0.446	0.127
1000	3.292	0.898
1500	11.461	3.189
2000	26.847	7.235

Tabela 2. Tempo de execução – Java

Matriz(ordem)	Serial(s)	Paralelo(s) – Jomp
500	0.228	0.189
1000	0.845	0.413
1500	2.590	1.000
2000	6.274	2.157

Tabela 3. Tempo de execução – Cython

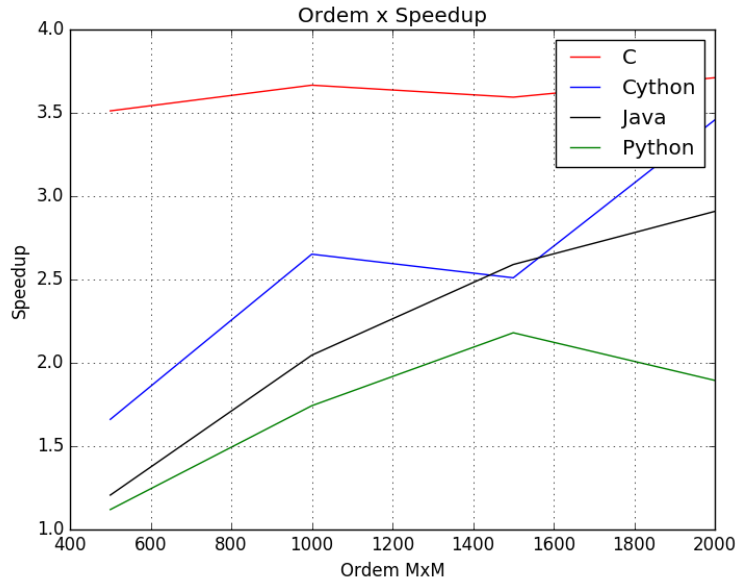
Matriz(ordem)	Serial(s)	Paralelo(s) – Cython
500	0.837	0.504
1000	10.186	3.841
1500	30.911	12.314
2000	1min41.794	29.442

Na execução do python com multiprocessing foi efetuada uma alteração no algoritmo serial. No lugar de uma multiplição entre matrizes, realizou-se duas multiplicações. Na prática, uma repetição foi inclusa na função de multiplicação. Dessa forma, tornou-se mais convincente o uso do paralelismo com multiprocessing, pois foi possível fazer com que cada processador executasse em paralelo uma multiplicação entre matrizes. Os tempos podem ser observados na tabela 4.

Tabela 4. Tempo de execução – Python2.7

Matriz(ordem)	Serial(s)	Paralelo(s) – Multiprocessing
500	1.021	0.912
1000	17.073	9.797
1500	1min7.847	31.116
2000	2min27.214	1min17.700

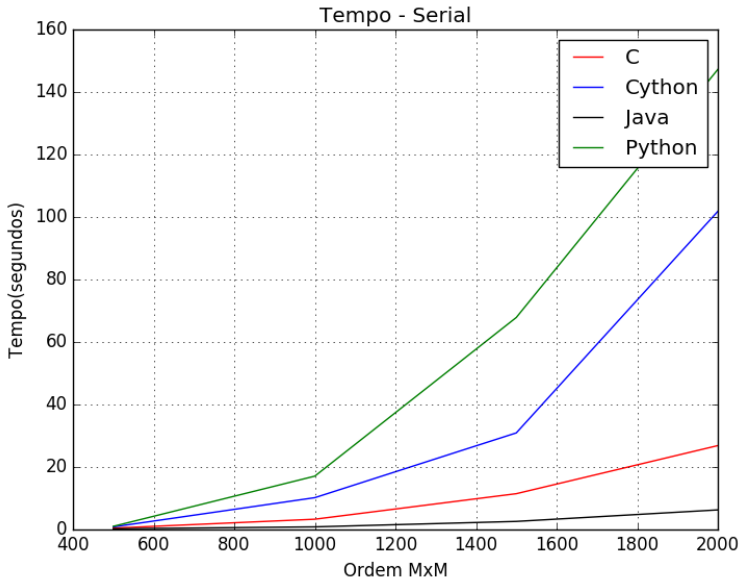
Figura 2. Ordem x Speedup



Fonte: Elaborada pelo autor.

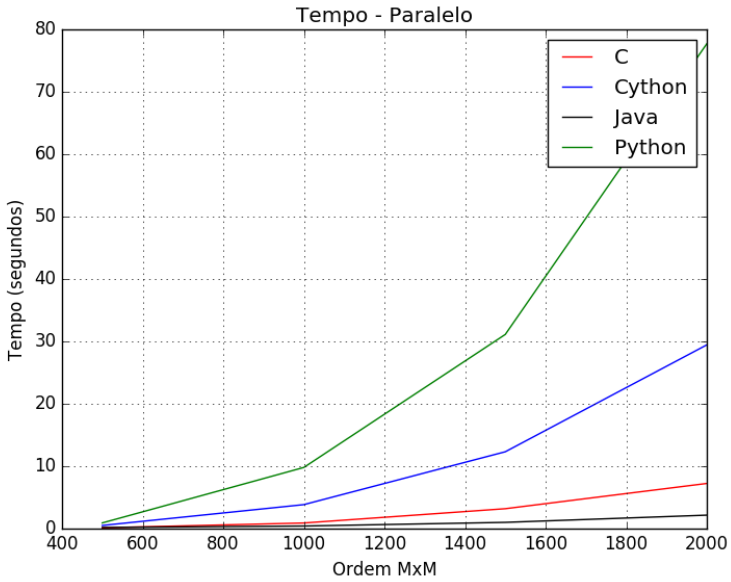
Conforme os valores obtidos e os gráficos gerados, é possível observar que a linguagem C obteve speedup entre 3.5 e 3.7, aproximadamente. O Python com multiprocessing obteve os menores valores de speedup, os quais variaram de 1 a 2, aproximadamente. Já o Cython e o Java ficaram entre 1 e 3, com uma leve vantagem para o Cython. A figura 2 ilustra o speedup de cada linguagem.

Figura 3. Tempo sequencial



Fonte: Elaborada pelo autor.

Figura 4. Tempo paralelo



Fonte: Elaborada pelo autor.

6. Considerações finais

O trabalho comparou o desempenho de algoritmos seriais com paralelos. Os algoritmos implementaram uma multiplicação entre matrizes quadradas de ordem 500, 1000, 1500 e 2000.

Em termos de tempo de execução, a linguagem Java foi a que obteve melhores resultados. Todavia, em termos de speedup, a linguagem C conseguiu atingir os valores mais altos. Lembrando, quanto maiores os valores, melhor desempenho aconteceu durante execução paralela. O speedup do Cython atingiu bons resultados e demonstrou que a linguagem cumpre as promessas de grande desempenho com relação ao Python.

Exceto o Python com multiprocessing, as demais linguagens realizaram somente uma multiplicação entre matrizes. Isso aconteceu para que o módulo do multiprocessing fosse aproveitado de forma convincente, caso contrário não haveria ganhos. Por essa razão é possível notar um tempo serial de execução superior ao tempo serial da base de cálculo do Cython.

A partir dos resultados, que são bem limitados, nota-se que o emprego do OpenMP é mais vantajoso para a linguagem nativa, a saber, a C. Embora haja diversos projetos, bibliotecas ou módulos eficientes para as linguagens apresentadas nesse artigo, a interação entre API e linguagem ainda não é madura suficiente.

Referências

[Behnel et al. 2011] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

[Costa and Sena 2008] Costa, J. and Sena, M. (2008). Tutorial openmp c/c++. *Laboratório de Computação Científica e Visualização, Maceió, AL*, 1.

[Sato] Sato, L. M. Análise comparativa de desempenho de um algoritmo paralelo implementado nas linguagens de programação cpar e openmp.

[William 2010] William, S. (2010). Arquitetura e organização de computadores.