UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

INTITUTO METRÓPOLE DIGITAL

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

SISTEMAS OPERACIONAIS

RELATÓRIO: IMPLEMENTAÇÃO DE LOOP UNROLLING

NOÉ FERNANDES CARVALHO PESSOA
PABLO EMANUELL LOPES TARGINO
RAFAELLA SILVA GOMES

Natal/RN
Setembro de 2019

Sumário

INTRODUÇÃO

Na disciplina de Sistemas Operacionais, teremos nosso primeiro trabalho prático envolvendo processos, memória compartilhada e sincronização.

O trabalho consiste na criação de um programa em linguagem Python, que deve realizar comunicação entre processos. E apesar de podermos aplicar computação sequencial por meio de sinais ou arquivos, devido à suas desvantagens com relação a possíveis perdas de desempenho, nosso foco será na utilização de memória compartilhada. Tal recurso acarreta na necessidade de utilizar sincronização de acesso á memória por estes processos. Esta sincronização pode ser feita por meio de semáforos ou *mutexes*, que servem para bloquear o acesso á memória compartilhada, enquanto o processo está em uso.

Um bom exemplo de programação paralela é o *loop unrolling*, onde sua função consiste de converter trechos do código inicialmente constituído de *loop*, em um número fixo e finito de instruções de forma sequencial. Desta forma, nosso programa terá um ganho significativo.

IMPLEMENTAÇÃO

O programa compõe-se de três partes, que deverão ser descritas em tópicos que seguem abaixo:

1. Definindo funções

Nesta primeira parte do trabalho, nós devemos implementar algumas funções seguindo as seguintes especificações:

- *Definindo uma função unroll que recebe uma matriz (LxN);*

Como solicitado, criamos a função *unroll*, que recebe como argumentos duas matrizes, uma função (que pode ser da thread ou processo), uma *string* que indica o método a utilizar (thread ou processo) e uma lista que receberá os resultados obtidos. Tendo-se a *unroll* pronta, decidimos que a operação que ela executaria seria a soma para cada linha da matriz que *unroll* recebe e guardaria na lista *results* o resultado de cada operação.

No caso dos processos, na função *unroll* fizemos com que fossem criados **n** processos filhos (1 para cada linha) através da função *fork*. Cada um desses filhos terá a missão de executar a função auxiliar *proc_func*, que calcula a soma dos elementos da linha. Sabendo-se da característica do problema, percebe-se que será preciso comunicação entre os processos já que precisaremos fazer a operação e atribuir os resultados para a lista *results* depois que todos os filhos terminarem suas tarefas. Para tornar essa comunicação possível foi usada uma memória compartilhada, e para organizar a disputa pela memória foi usado um semáforo.

Tendo a função *unroll* já pronta, faltava apenas criar a função *proc_func*. Foi decidido, então, que a função *proc_func* (ver Figura 1) receberia como argumentos a linha atual da matriz e a posição que ela ocupa. Dessa forma, dentro da função, usamos um *acquire* antes da atribuição da linha na memória compartilhada e um *release* logo após, assim permitindo exclusividade ao processo que estivesse executando a função no momento atual

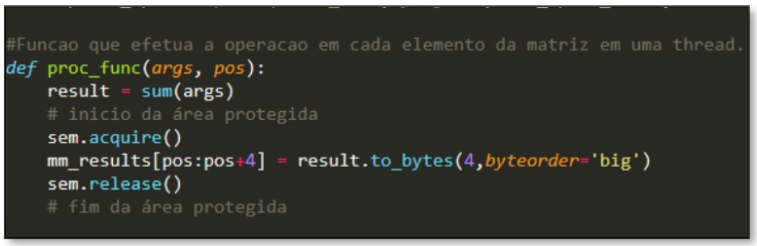


Figura 1: Função proc_func

No caso dos threads, a função *unroll* trabalha da seguinte forma: criamos uma lista *threads* onde alocamos a lista *results*. Então, para cada linha da matriz, cria-se um novo thread, colocando-os na lista *threads* e executando-os.

O fato de armazenar os threads criados é importante para que assim que possível o programa principal possa aguardar ser finalizado, usando a função *join*. Assim, a função *thread_func* (ver Figura 2) ficou da seguinte forma:

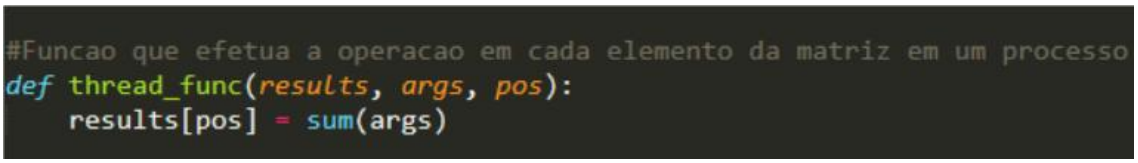


Figura 2: Função thread_func

Nos *threads*, pôde-se usar diretamente a lista global *results*, o que tornou a implementação mais fácil que a de processos. Dessa forma, com o auxílio da posição, a atribuição da soma foi realizada para cada linha da matriz.

- *Implementando soma entre duas matrizes:*

Para realizar a operação de soma, modificamos a estrutura da função *unroll*, onde agora ela recebe uma matriz quadrada e armazenará em *results* a matriz resultante.

No caso dos processos, houve modificação apenas na função *proc_func* (ver Figura 3), que agora recebe como parâmetro um elemento de cada matriz e sua posição, realiza a soma e a coloca na memória compartilhada.

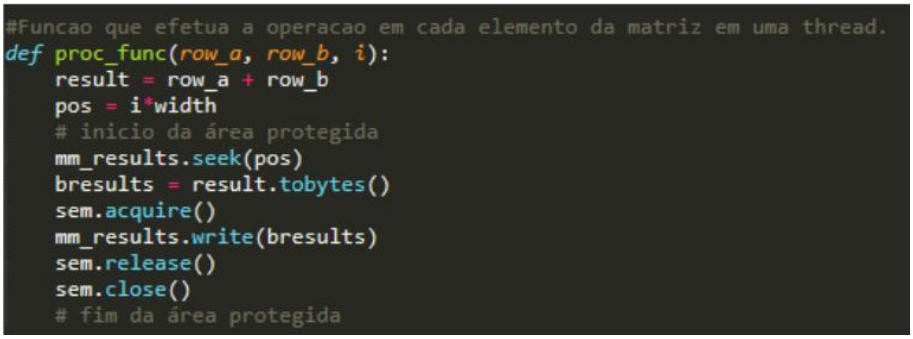


Figura 3: Operação soma de matrizes da função proc_func.

Para os *threads*, analogamente aos processos, houve modificação significativa apenas nos parâmetros da função *thread_func* (ver Figura 4), que agora recebe dois elementos (um de cada matriz), sua posição e a lista *results*. Em seguida, é realizada a soma e atribuição à lista global.

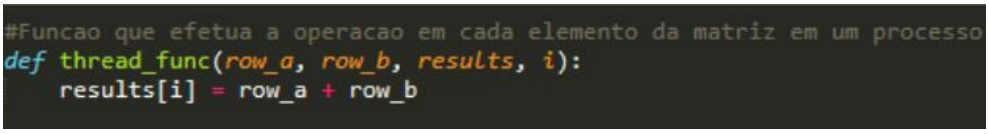


Figura 4: Operação soma de matrizes da função thread_func.

➢ *Implementando multiplicação entre duas matrizes:*

De maneira semelhante às alterações realizadas nas funções *proc_func* e *thread_func* para a operação de soma, o mesmo acontecerá nestas funções, alterando apenas o método operatório. Como isso, teremos as seguintes alterações na função *func_proc:*

```
def proc_func(row_a, mat_b, i):
        result =  np.zeros(width_b, dtype=np.uint32)
        for j in range(width_b):
                result[j] = np.sum(row_a*mat_b[:,j])
        pos = 4*i*width_a
        mm_results.seek(pos)
        bresult = result.tobytes()
        # inicio da área protegida
        sem.acquire()
        mm_results.write(bresult)
        sem.release()
        # fim da área protegida
        sem.close()
```

Figura 5: Operação multiplicação de matrizes da função proc_func.

Enquanto que a função *thread_func* ficará da seguinte forma:

```
def thread_func(row_a, mat_b, results, i):
        for j in range(width_b):
                results[i][j] = np.sum(row_a*mat_b[:,j])
```

Figura 6: Operação multiplicação de matrizes da função thread_func.

➢ *Modelagem em redes petri:*

Para a solução aplicada no item anterior, podemos demonstrá-la por meio de uma modelagem em redes de petri da seguinte maneira:

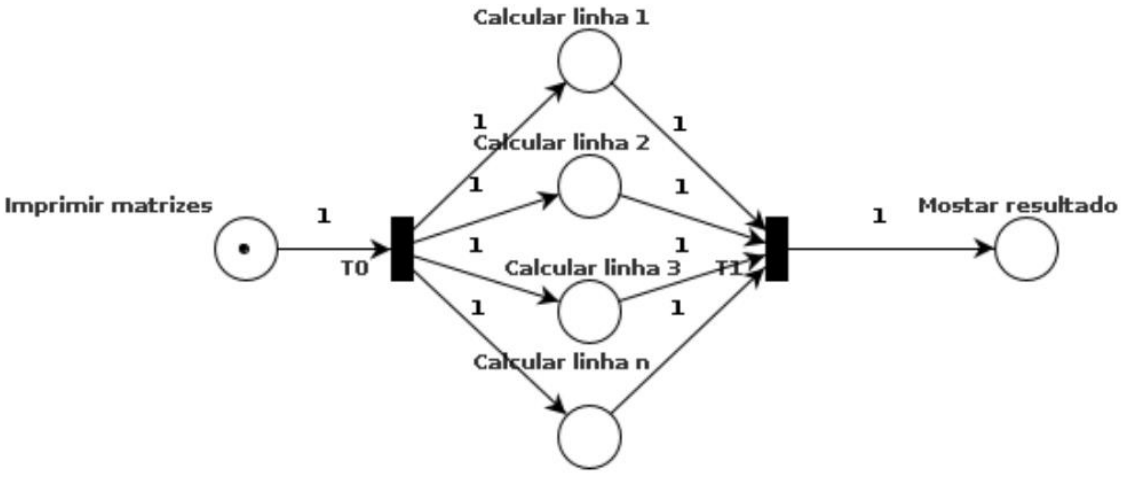


Figura 7: Modelagem em redes de petri.

2. Comparação entre implementação paralela e sequencial

Na segunda parte do trabalho, nós devemos comparar as execuções paralelas com thread ou processos e sequenciais. Esta comparação serve para verificar o ganho por meio da verificação do tempo que cada implementação leva para realizar as mesmas operações. Desta forma, utilizaremos dos seguintes requisitos:

- *Programa sequencial com as operações soma e multiplicação de matrizes:*

Para realizar a comparação entre as operações foram crias as funções *mat_sum* e *mat_product*, onde ambas recebem os valores das matrizes e os valores de retorno, realizam as operações de soma e produto, respectivamente, e por meio da função *time.process_time()* onde são capturados os momentos iniciais e finais de cada processo, retornando por fim o tempo total gasto pelos mesmos.

Desta forma, segue abaixo as funções *mat_sum* (ver Figura 8) e *mat_product* (ver Figura 9), respectivamente.

```
def mat_sum(args, results):
        t_start = time.process_time()
        results = np.zeros((width_a, height_a), dtype=np.uint32)
        for i in range(height_a):
                results[i] = args[0][i] + args[1][i]
        t_end = time.process_time()
        print(t_end - t_start)
```

Figura 8: Função mat_sum.

```
def mat_product(args, results):
        t_start = time.process_time()

        results =  np.zeros((width_a, height_a), dtype=np.uint32)
        for i in range(height_a):
                for j in range(width_b):
                        results[i] = np.sum(args[0][i]*args[1][:j])
        t_end = time.process_time()
        print(t_end - t_start)
```

Figura 9: Função mat_product.

- *Teste de tempo de execução das operações entre matrizes de ordem crescente:*

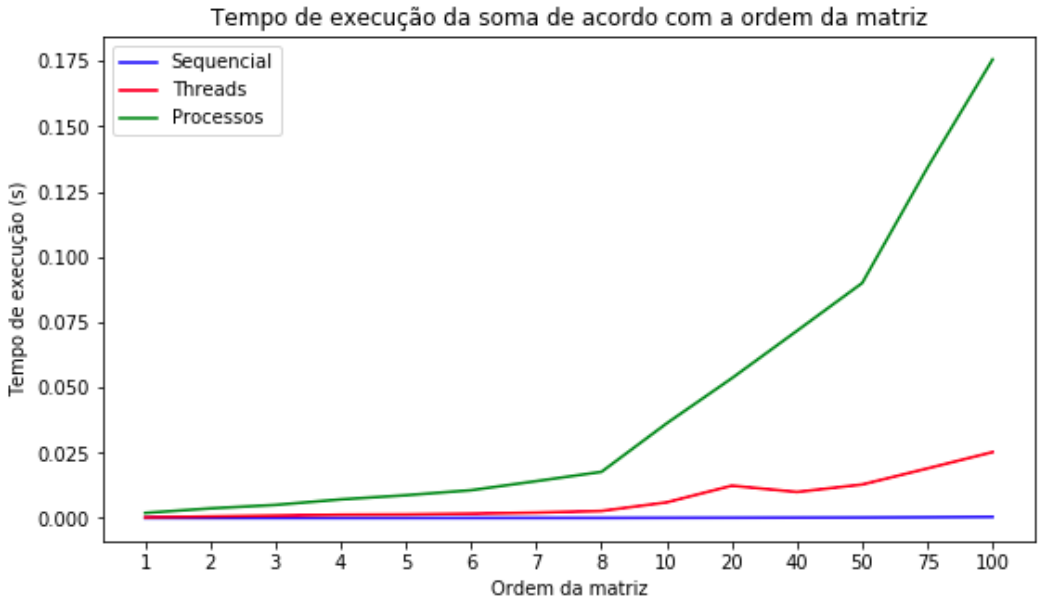
Por meio da função *random.randint*() serão gerados elementos aleatórios das matrizes que terão seus tamanhos incrementados linearmente. Estas matrizes criadas deverão ser adicionadas às funções *mat_sum* e *mat_product*, onde as operações serão realizadas e seus tempos serão calculados. Assim, nossa implementação se apresenta da seguinte maneira:

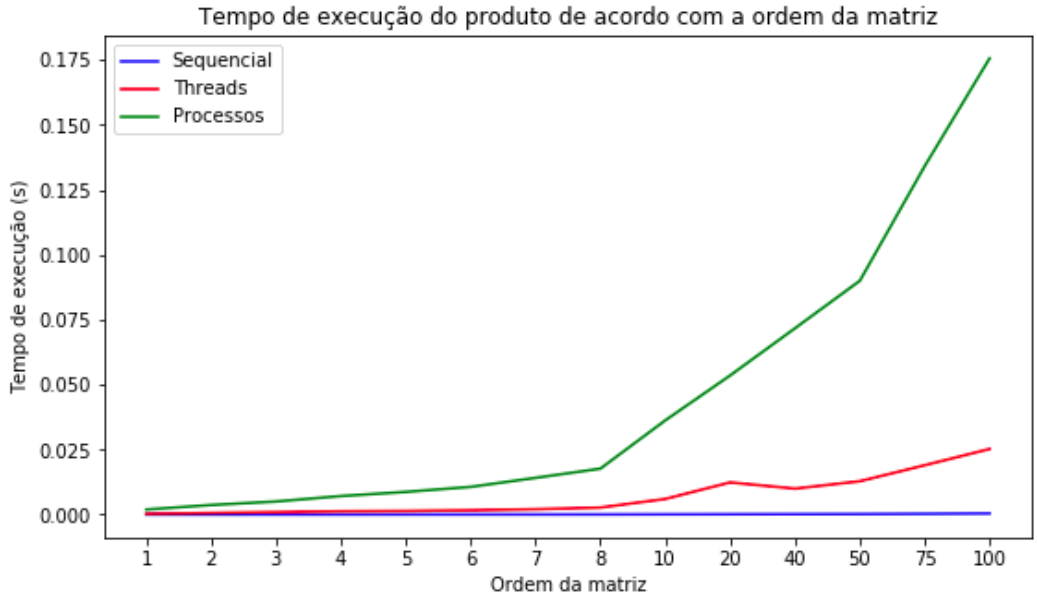
```
if (__name__ == "__main__"):
        results = []
        if(width_a != height_b):
                print('A matriz não pode ser multiplicada, pois a largura de a:{} != altura de b:{}.'.format(width_a,height_b))
                exit(1)

        #para testar os tempos de execução da operação para cada ordem de matriz.
        for i in [1, 2, 3, 4, 5, 6, 8, 10, 20, 30, 40, 50, 75, 100]:
                matrizA = np.random.randint(0, np.iinfo(np.uint16).max, (i,i), np.uint16)
                matrizB = np.random.randint(0, np.iinfo(np.uint16).max, (i,i), np.uint16)
                height_a, width_a, height_b, width_b = i, i, i, i
                # mat_sum([matrizA, matrizB],results)
                mat_product([matrizA, matrizB],results)
```

Figura 10: Gerando matrizes aleatórias de tamanhos crescentes.

Logo após, utilizando a função process_time da biblioteca time, conseguimos o tempo que cada operação é realizada considerando a execução com processos, threads ou sequencial. Tivemos, então, os seguintes gráficos:





- *Sincronizando atividades paralelas*

A terceira e última parte do trabalho consiste na sincronização das atividades. Esta funcionalidade se faz necessário uma vez que os *loops* apresentam dependência de dados, como mencionado na introdução deste relatório.

O nosso programa deverá gerar e multiplicar N matrizes utilizando *unrolling* da maneira mais paralela possível. E para isto, utilizaremos vários conceitos abordados anteriormente, bem como a criação das funções *unroll* (ver Figura 11), que deve receber a matriz e aplicando os conceitos de thread por meio do *Lock*() deve realizar o devido controle do acesso a recursos compartilhados, para evitar a corrupção de dados, que é efetivamente o objetivo deste trabalho. Abaixo segue nossa implementação:

```
def unroll(matrizes):
        print(" · ".join(str(len(m)) + 'x' + str(len(m[0])) for m in matrizes))
        print('=')
        while(len(matrizes) > 1):
                lock = threading.Lock()
                n = len(matrizes)
                while(n > 1):
                        prev = lock
                        lock = threading.Lock()
                        lock.acquire()
                        thread = threading.Thread(target=multiplicar_par, args=(matrizes.popleft(), matrizes.popleft(), lock, prev))
                        thread.start()
                        n -= 2
                lock.acquire()
                if(n == 1):
                        matrizes.append(matrizes.popleft())
                print('=')
        print(str(len(matrizes[0]))+'x'+str(len(matrizes[0][0])))
```

Figura 11: Função unroll.

Já a função *gerar_matrizes_multiplicativas* (ver Figura 12), é responsável por gerar o resultado da multiplicação das N matrizes. Segue sua implementação:

➢ *Modelagem em redes de petri:*

A aplicação deste método utilizando cinco matrizes pode ser vista na modelagem de petri a seguir:

```
def gerar_matrizes_multiplicativas(amount, min_height = 1, min_width = 1, max_height = 5, max_width = 5,
                                                                        range_start = 0, range_end=10):
        matrizes = deque()
        height = randrange(min_height, max_height+1)
```

Figura 12: Modelagem em redes de petri.

```
                width = randrange(min_width, max_width+1)
                matrizes.append(np.random.randint(0, np.iinfo(np.uint8).max+1, (height,width), np.uint8))
                height = width
        return matrizes
```

Figura 13: Modelagem de redes de petri.

Conclusão

Ao fim do trabalho, percebemos que nem sempre a paralelização vai representar um ganho significativo no tempo de execução. Por exemplo, nos gráficos gerados para soma e multiplicação, tivemos que a execução paralela foi bem mais lenta do que a sequencial, pois o custo de criar os processos e threads não compensou qualquer ganho que pudesse haver neste caso. Também tivemos problemas quando tentamos realizar as operações com paralelização elemento por elemento já que a partir de matrizes de ordem 75 o sistema operacional bloqueou a criação de novos processos. No final, o projeto foi uma boa experiência e oportunidade de verificar como se comportam na prática as estruturas que vimos ao longo da unidade 1.