TÓPICOS ESPECIAIS EM INTERNET DAS COISAS "C"

Multiplicação de Matrizes - Implementação Serial e Paralela em linguagem C utilizando POXIS Threads

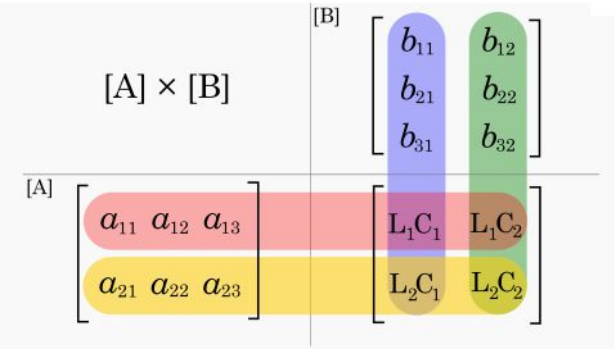
Leandro Silva Ferreira Junior

1. Introdução

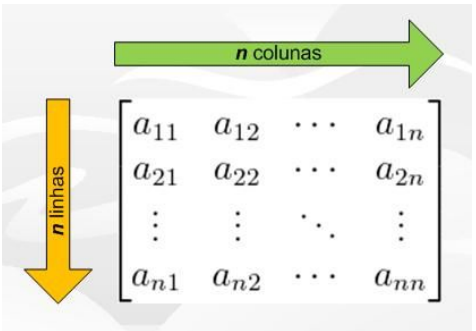
Quando consideramos fazer a multiplicação de duas matrizes, é necessário entender que o produto delas não é determinado por meio do produto dos seus respectivos elementos. Assim, ao multiplicarmos uma matriz A = (a_{ij})m,r com B = (b_{ij})r,n teremos uma matriz resultante C = (c_{ij}) m x n, onde cada elemento c_{ij} é obtido pelo produto dos elementos da linha **i** de A com os elementos correspondentes da coluna **j** da matriz B.

$$(AB)_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

Para que esta multiplicação seja possível é necessário que o número de colunas da matriz A seja igual ao número de linhas da matriz B. Dessa forma, teremos como resultado a matriz C com o mesmo número de linhas da matriz A e o mesmo número de colunas da matriz B.

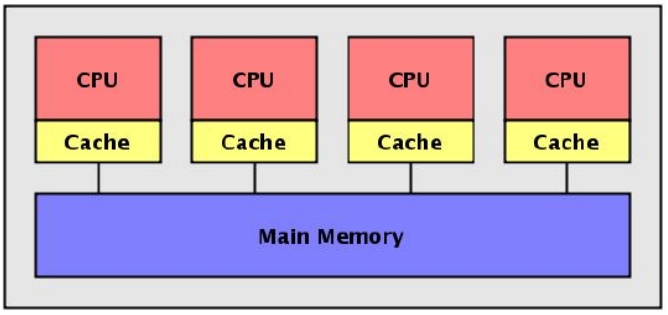


Como no trabalho em questão foram usadas apenas matrizes quadradas, não houve limitações quanto a possibilidade de multiplicação, e as matrizes resultantes possuem a mesma dimensão das usadas no produto.

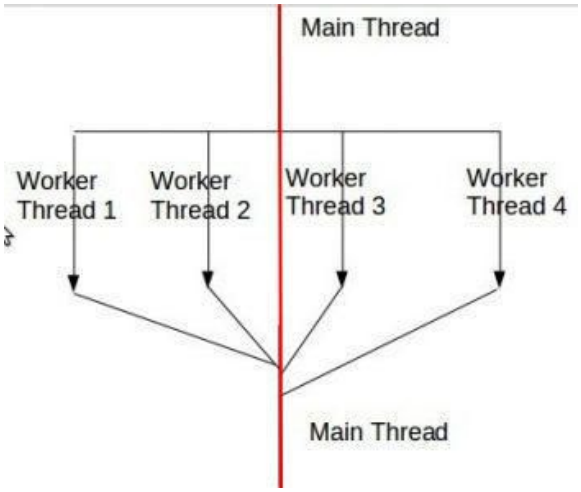


Quando comparamos a multiplicação de matrizes para tamanhos diferentes, é notório que o aumento do tamanho exige um número de cálculos envolvidos muito maior que o tamanho anterior em uma proporção alta. Isso gera grande impacto no tempo de execução dos cálculos, para os quais uma solução alternativa é a paralelização, dividindo os cálculos na região paralela e otimizando o tempo de resposta total.

Para isso, foi utilizada a estrutura computacional de memória compartilhada, e que será administrada pelo modelo POSIX Threads, onde dividiremos o número de linhas da matriz resultante a ser calculadas pelo número de threads disponíveis.



Na utilização desse tipo de estrutura, a abertura das threads é independente dos processos da main, que continua sendo executada paralelamente e a comunicação entre as threads se dá por meio de variáveis globais, o que gera regiões críticas.



Essas regiões críticas devem ser controladas de modo que uma thread não acesse uma variável global enquanto outra thread estiver utilizando a mesma variável, e para realizar esse controle, foi implementado o uso de semáforos, que vão informar se aquele dado está disponível ou não para ser acessado. Dessa forma evitamos a ocorrência de “ghost values”, ou seja, o acesso a valores e informações desatualizadas durante a execução.

2. Desenvolvimento

2.1 Implementações

2.1.1 Implementação Serial (linguagem C)

Na implementação padrão da multiplicação de matrizes, foram definidas variáveis globais para os ponteiros das respectivas matrizes A, B e C e uma variável de inteiro onde será guardado o tamanho das matrizes quadradas que será recebido por parâmetro na execução da main. Foi utilizada também uma função *print*, definida para imprimir as matrizes.

Ao executarmos a main, de início, para cada uma das 3 matrizes A, B e C, foi alocado dinamicamente um vetor ponteiro de ponteiros, onde logo em seguida para cada índice desse vetor, foi alocado um vetor de mesmo tamanho. Desta forma estamos alocando toda a matriz em 2D utilizando o tamanho definido.

Em seguida foi inserido um laço que percorre todos os elementos a_{ij} de uma matriz. Este laço é utilizado para preencher os elementos das 3 matrizes, onde para as matrizes A e B, são gerados números entre 0 e 9, e para a matriz C, todos os elementos recebem valor -1.

No passo seguinte há um laço de mesma estrutura do anterior, porém sua função neste caso é a multiplicação das matrizes, onde A*B=C. Para cada elemento C(a_{ij}) a ser calculado, inicialmente ele recebe o valor 0 (pois antes era -1), e em seguida executa outro laço que irá percorrer a linha A_i e a coluna B_j multiplicando seus respectivos valores e atribuindo-os a um somatório no elemento C_{ij}. A estrutura de medição de tempo foi implementada de modo a calcular somente o tempo desta multiplicação.

```
gettimeofday(&start, NULL);

//multiplicação das matrizes
for (i=0; i<n_size; i++){
    for (j=0; j<n_size; j++){

        C[i][j]=0;
        for(r=0; r<n_size; r++){
            C[i][j] += A[i][r] * B[r][j];
        }
    }
}

gettimeofday(&stop, NULL);
```

O tempo será impresso logo em seguida pela main no arquivo de saída txt, e após esta etapa, no caso da execução no supercomputador, será impressa no arquivo de saída .out. A execução finaliza logo após a liberação das matrizes alocadas dinamicamente.

2.1.2 Implementação Paralela (Utilizando POSIX Threads):

Na implementação paralela, nosso objetivo é dividir os cálculos da matriz entre as threads criadas, de modo que o número de linhas da matriz seja dividido igualmente pelo número de threads disponíveis.

Para que esta lógica fosse implementada corretamente, foi criada uma função responsável pela multiplicação, a qual recebe como parâmetro o rank da thread que está executando-a. Nesta função, são definidas variáveis de sinalizam a linha inicial e a linha final para qual a thread deve realizar os cálculos deste intervalo, e além disso, a cada linha [i] calculada a função realiza um "post" no semáforo[i]. O algoritmo garante que se o tamanho da matriz não for exatamente divisível pelas threads, a última thread pega as linhas restantes.

```
void *multiplication(void* rank) {

    long my_rank = (long) rank;
    long inicio = my_rank*(n_size/m_threads);
    long final = inicio+(n_size/m_threads)-1;

    if(my_rank == m_threads-1){
        final=n_size-1;
    }

    int i,j,r;
    for (i=inicio; i<=final; i++){
        for (j=0; j<n_size; j++){

            C[i][j]=0;
            for(r=0; r<n_size; r++){
                C[i][j] += A[i][r] * B[r][j];
            }
        }
        sem_post(&semaphore[i]);
    }
    return NULL;
}
```

O vetor de semáforos foi implementado com o objetivo de sinalizar se a linha [i] já foi calculada, e como é um dado comum às threads, foi inserido como variável global, e alocado dinamicamente com o tamanho do problema, sendo definido em seguida com todos os seus valores iguais a zero.

Apesar de ser acessado por todas as threads, ele não é uma região crítica, pois cada linha possui o seu elemento de índice [i] do vetor. Esta definição explica o que foi dito anteriormente na função de multiplicação, onde para cada linha completamente calculada, a função muda o semáforo de 0 para 1.

Para a implementação, a main do código aloca também um vetor de endereço das threads com o tamanho igual ao número de threads inserido na execução, o qual será usado como referência aos endereços nos laços de pthread_create e pthread_join, que abrem e fecham a região paralela.

Fazendo um panorama mais geral, ao ser executada, a main recebe por parâmetros o número de threads e o tamanho do problema, respectivamente. Em seguida inicia o vetor de semáforos com valor 0 para todos, e posteriormente aloca e preenche as matrizes. Porém,

neste caso estaremos alocando mais uma matriz D, que será preenchida inicialmente com todos os elementos iguais a zero.

Esta matriz D tem o objetivo de simular um programa externo ao código tentado resgatar valores calculados pela execução, e tudo isso durante a execução. Este papel será desempenhado pela main assim que a região paralela for criada.

Após o preenchimento das matrizes, o tempo começa a ser contado, e a região paralela das threads é então aberta, sendo passado como parâmetro a execução da função responsável pela multiplicação que já irá direcionar as partes cabíveis a cada thread.

A partir deste ponto entra o trabalho da main de simular o resgate de dados enquanto os cálculos estão sendo feitos. Para isso, a main define um vetor de booleanos com o tamanho do problema(representando as linhas neste caso) e atribui valor falso a todos os elementos, onde este booleano indica se a linha em questão já foi copiada pela main.

A main então executa um laço definido com o tamanho do problema, onde inicialmente é sorteado aleatoriamente uma linha, e uma função (da biblioteca dos semáforos) resgata o valor do semáforo desta linha. Este semáforo é verificado, e se já possuir o valor 1, indica que a linha já foi toda calculada, e que portanto já podemos copiá-la. Após este passo, é verificado se a linha já foi copiada para que uma linha não seja copiada mais de uma vez, e caso ainda não tenha sido copiada, o elemento a_{ij} da matriz C é copiado respectivamente para o elemento a_{ij} da matriz D, e é indicado ao vetor booleano que a linha foi copiada.

```
//main part
bool* foicopiado = malloc(n_size*sizeof(bool));
for(i=0; i<n_size; i++){
    foicopiado[i]=false;
}

for(int cont=0; cont<n_size; cont++){

    valor=0;
    i=rand()%n_size;
    sem_getvalue(&semaphore[i], &valor);

    if(valor==1){
        if(foicopiado[i]){} //nao faz nada
        else {
            for(j=0; j<n_size; j++){
                D[i][j]=C[i][j];
            }

            foicopiado[i]=true;
        }
    }
}
```

Após esta parte, mesmo que as threads ainda não tenham terminado todos os cálculos, como a main já executou suas tarefas, a região paralela é fechada e o tempo final é medido. O algoritmo então salva no arquivo txt o tempo de execução calculado, onde é feita uma verificação se não há elementos negativos na matriz D, e se ocorrer, o tempo impresso será zero. Está verificação de números negativos foi implementada usando uma função booleana que percorre toda a matriz D.

Para finalizar, são impressos na saída .out as matrizes C e D para efeitos de comparação. Os semáforos são então destruídos, os alocamentos dinâmicos liberados e a execução encerra.

2.2. Resultados Experimentais

2.2.1 Configuração do problema

Para realizar as análises de speedup e eficiência, é necessário realizar execuções para o algoritmo serial e comparar os tempos de execução com os tempos do algoritmo paralelo. Essa comparação foi realizada para 4 tamanhos de problema de matrizes diferentes: 700, 900, 1100 e 1400, onde o tamanho n é $M_{n,n}$.

A organização de todas as execuções dos códigos paralelo e serial foram feitas a partir de dois shellscripts (serial e paralelo) destinados ao supercomputador, que já possuem as notações necessárias para carregar os módulos de execução do sistema, além da compilação e execução do código. A configuração do shellscript garante que a execução do código seja feita 15 vezes, gerando assim 15 tempos diferentes, dos quais iremos obter uma média que será utilizada para fins de cálculo. O shellscript também imprime no arquivo txt uma tabela com o número de cores utilizados, e o tamanho utilizado a cada 15 execuções, das quais o próprio programa escreve no mesmo arquivo o tempo de cada uma.

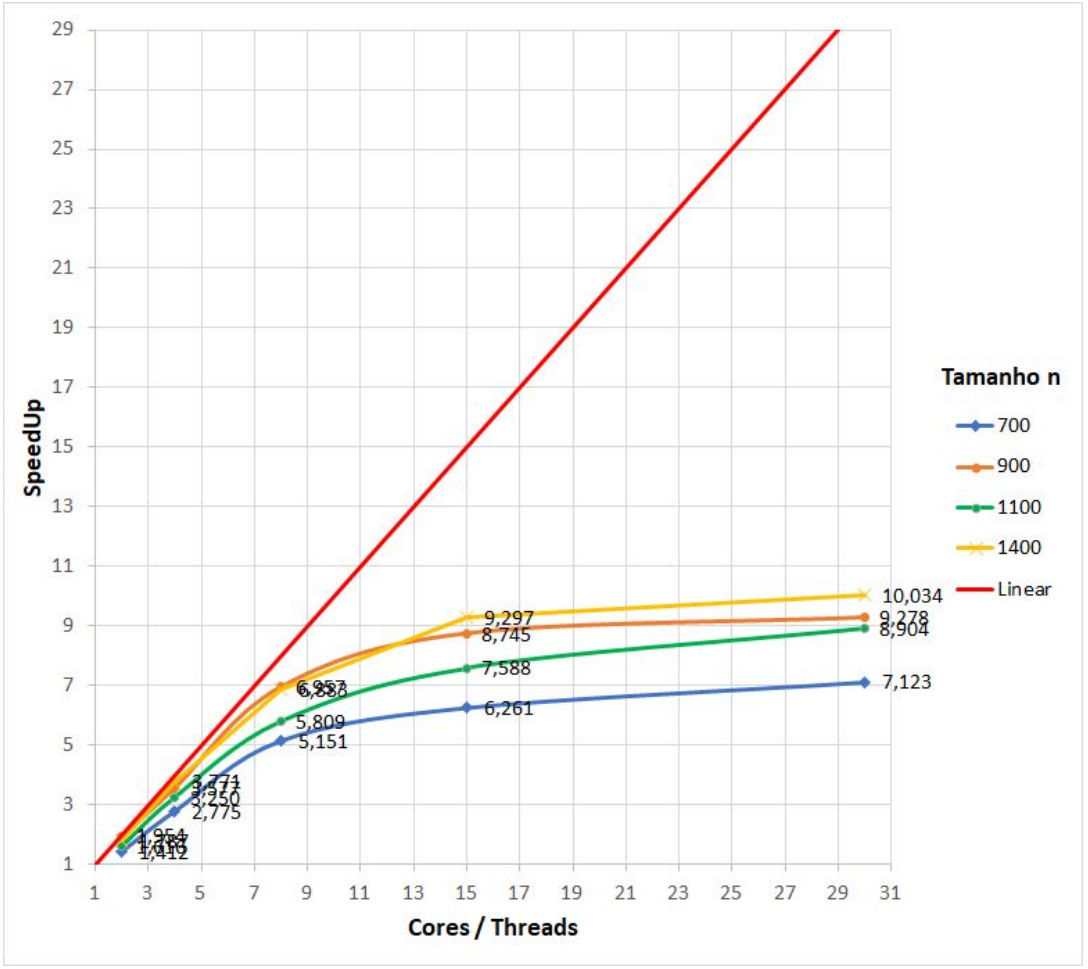
No caso do script paralelo, foi inserido mais um laço para o número de threads que serão utilizadas, desta forma, os testes para os 4 tamanhos serão realizados utilizando 2, 4, 8, 15 e 30 cores. Foi utilizado um valor ímpar para garantir que a lógica se aplica também no caso em que o tamanho do problema não é exatamente divisível pelo número de threads.

2.2.2 Análise de Escalabilidade (Speedup e Eficiência)

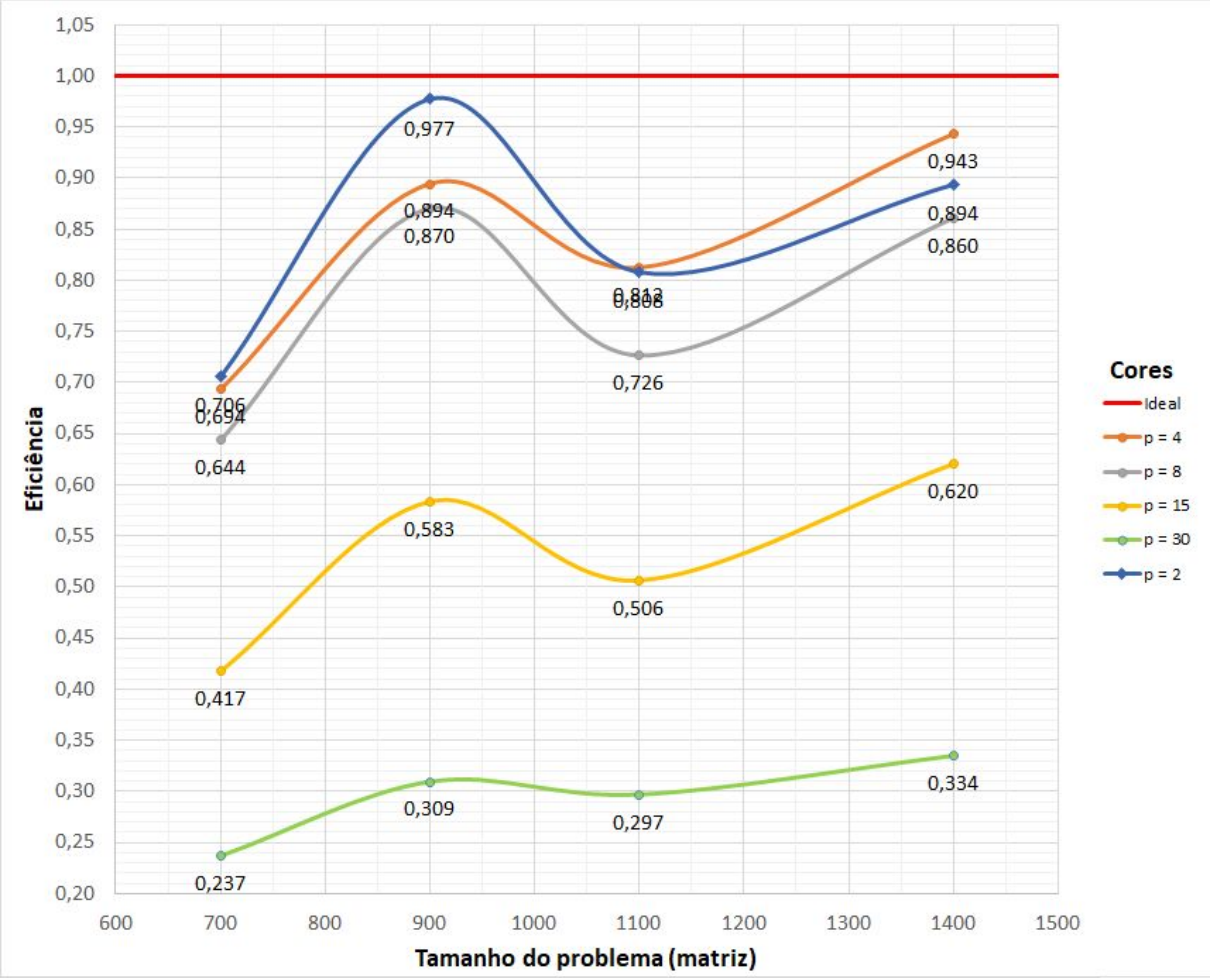
Após concluída a execução dos scripts, os arquivos txt de saída dos dados contendo as tabelas com os tempos das execuções e seus parâmetros foram importados para a o Excel, onde foi calculado a média de cada conjunto de 15 tempos, os valores de SpeedUp, e os valores de Eficiência:

N Cores	Tamanho		Média Tempo	SpeedUp	Eficiência
Serial	700		3,197	--	--
Serial	900		7,063	--	--
Serial	1100		16,201	--	--
Serial	1400		36,174	--	--
2	700		2,264	1,412	0,706
2	900		3,614	1,954	0,977
2	1100		10,027	1,616	0,808
2	1400		20,240	1,787	0,894
4	700		1,152	2,775	0,694
4	900		1,975	3,577	0,894
4	1100		4,986	3,250	0,812
4	1400		9,592	3,771	0,943
8	700		0,621	5,151	0,644
8	900		1,015	6,957	0,870
8	1100		2,789	5,809	0,726
8	1400		5,255	6,883	0,860
15	700		0,511	6,261	0,417
15	900		0,808	8,745	0,583
15	1100		2,135	7,588	0,506
15	1400		3,891	9,297	0,620
30	700		0,449	7,123	0,237
30	900		0,761	9,278	0,309
30	1100		1,819	8,904	0,297
30	1400		3,605	10,034	0,334

O gráfico das SpeedUps foi plotado para os 4 tamanhos de problemas analisados, e mostra a evolução das SpeedUps em relação ao número de cores, ou threads, utilizados:



O gráfico de eficiência mostra a sua evolução em relação aos tamanhos de problema utilizados, e foi plotado em análise para os 5 valores de cores, e consequentemente threads utilizados nas execuções:



Foi notado um comportamento padrão para todos os tamanhos de problema de testados, e isso pode ser percebido através dos gráficos plotados, onde as speedups possuem tendências muito próximas, e apesar disso, não seguem o comportamento padrão de que os maiores tamanhos de problema possuem os melhores valores de speedup. Houveram oscilações nos valores de eficiência, e isso mostra que para alguns determinados tamanhos de problema, os cores são melhor utilizados do que em outros tamanhos.

Fazendo uma análise mais ampla dos resultados, as eficiências têm uma tendência de crescimento se considerarmos a totalidade do experimento realizado, o que mostra que este algoritmo possui sua escalabilidade, porém, o algoritmo não é nem fortemente nem fracamente escalável seguindo os parâmetros de análise, e isso fica evidenciado na redução drástica da eficiência quando utilizamos mais núcleos de processamento.

4. Conclusão.

Durante a execução do código paralelo, ao analisar e comparar a matriz C e a matriz D impressa na tela, foi possível perceber que algumas linhas da matriz D continuavam com todos os valores zerados. Isso evidencia que a main tentou resgatar algumas linhas da matriz C, porém não resgatou todas, isso por dois motivos: um deles é que as linhas foram sorteadas aleatoriamente, e que por isso, algumas linhas podem não ter sido escolhidas. Além disso, pode ocorrer de que ao tentar resgatar uma linha, a main não encontrou o seu semáforo indicando que a linha estava disponível e então não a copiou.

Esse bloqueio realizado pelo semáforo, garante o controle das regiões críticas do código, e que consequentemente, a main não copie uma linha da matriz C enquanto ela estiver sendo calculada.

Para garantir que essa parte não ocorre-se foi implementada a condição de verificar números negativos em D, o que a lógica garante que não ocorre. Porém, se ocorresse, isso significaria que a main copiou dados de C para D enquanto a linha estava sendo calculada, ou seja, entrou na região crítica.

A paralelização utilizando PThreads otimizou consideravelmente o tempo de execução dos cálculos, pois como estamos trabalhando com matrizes, o número de interações aumenta drasticamente ao aumentarmos sua dimensão.