UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Pthreads 1 - Matriz quadradas

Pedro Cardoso Carvalho

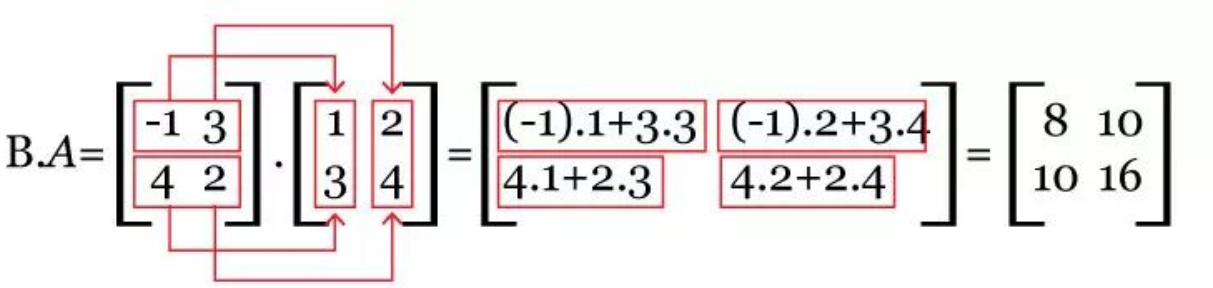
NATAL-RN
2020

Sumário

Introdução

Neste relatório iremos realizar um comparativo entre a utilização da programação paralela à tradicional programação serial, aplicado na resolução de uma multiplicação de matriz quadrada.Como demonstrado na figura 1, a resolução da multiplicação de matrizes que possuem mesmo número de linhas e colunas, chamada de matrizes quadradas, é realizado por uma somatório da multiplicação da coluna da primeira matriz com a linha da segunda matriz.

Figura 1- Demonstração de resolução de multiplicação de matriz quadrada



Fonte Só matemática

Desenvolvimento

Ambiente de teste

Todos os teste foram realizados em uma máquina do supercomputador disponibilizada pelo NPAD sendo para os códigos seriais a utilização de um núcleo e para o código paralelo uma maquina de até 32 cores(figura 2 possui o código do cabeçalho onde temos a configuração de uso da máquina paralela solicitada) . Foram selecionados 4 tamanhos de problemas que correspondem os valores M para duas matrizes MxM, sendo eles 1440, 1600, 1800 e 2080 , para cada uma desses casos foram realizados 10 execuções e para o caso do código paralelo para cada ambiente deste temos os casos de utilização de 4, 8, 16 e 32 processos. Para geração dos valores da matriz serem iguais para todos os casos foi utilizado a semente com valor fixo de 10 em hardcode.

Figura 2- Configuração da maquina virtual do codigo paralelo

```
#SBATCH --partition=cluster
#SBATCH --job-name=Parallel
#SBATCH --output=ParallelOutput.out
#SBATCH --error=ParallelError.err
#SBATCH --time=0-02:0
#SBATCH --hint=compute_bound

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32
```

Fonte: Autor

Programa serial

Para código serial, como forma de melhorar o estudo do desempenho, foram criados dois codigo. Um que realizaria o cálculos das matrizes seguindo a ordem linhas da primeira matriz

Para código serial, seguindo a ideia do problema relatado na introdução, foi gerado três sequência de loopings onde o primeiro responsável por percorrer os index das linhas da matriz resultante, segundo os index das colunas da matriz resultante e o terceiro looping que iria percorrer os valores das linha da primeira matriz e da coluna da segunda matriz realizando a multiplicação dos valores e somatório total desses valores, como demonstrado na figura 3

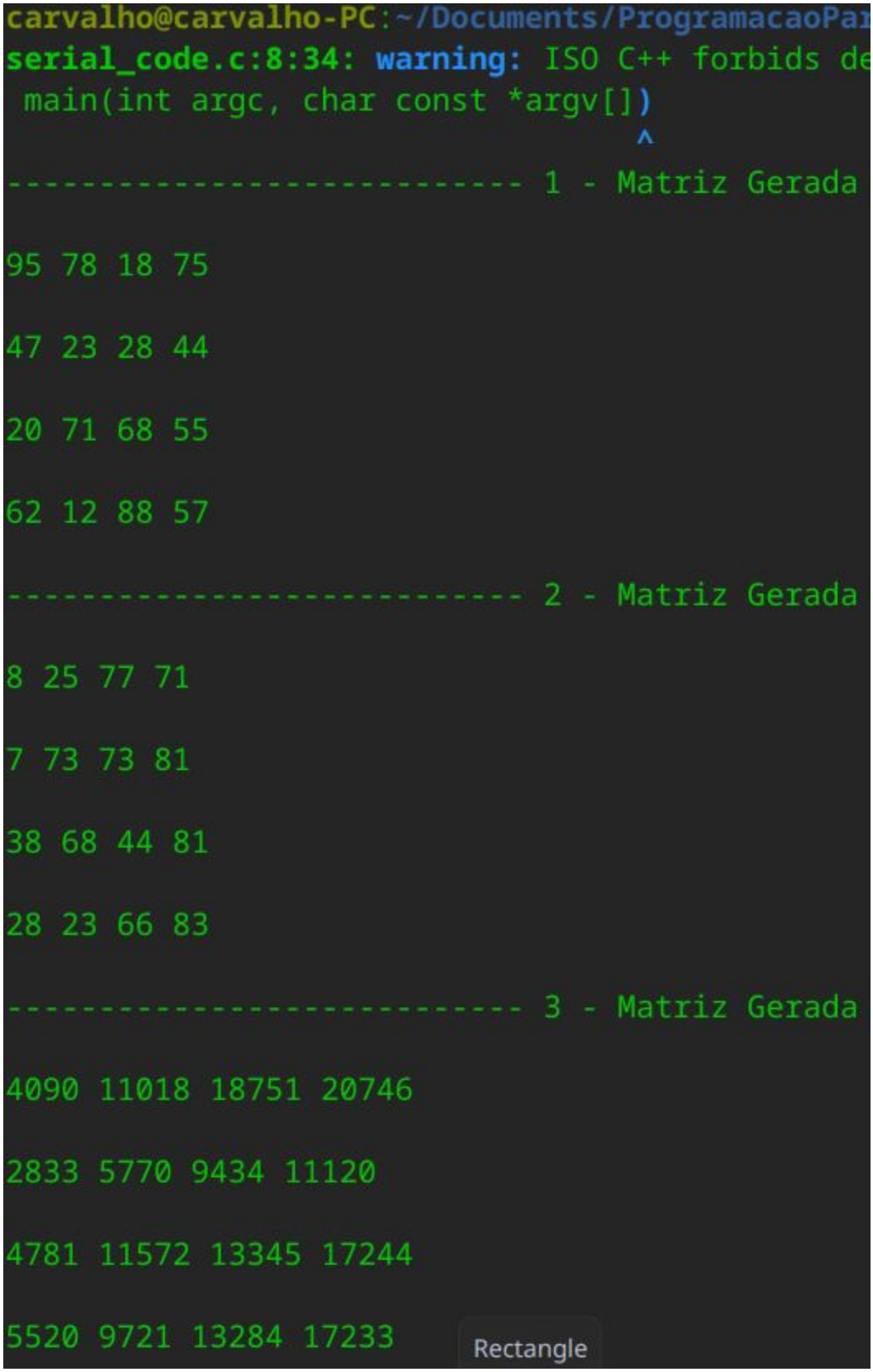
Figura 3- Código serial de multiplicação de matriz quadrada

```
for (i = 0; i < linhaA; i++)
{
    for (j = 0; j < linhaA; j++)
    {

        matrizC[i][j] = 0;
        for (x = 0; x < linhaA; x++)
        {
            aux += matrizA[i][x] * matrizB[x][j];
        }

        matrizC[i][j] = aux;
        aux = 0;
    }
}
```

Fonte -Autor

Um exemplo de execução do código mostrado pode ser visto na figura 4 com matriz valor M de 4.

Figura 4- Execução Código serial de multiplicação de matrizes quadradas


carvalho@carvalho-PC:~/Documents/ProgramacaoPar
serial_code.c:8:34: warning: ISO C++ forbids de
main(int argc, char const *argv[])
^
---------- 1 - Matriz Gerada
95 78 18 75
47 23 28 44
20 71 68 55
62 12 88 57
---------- 2 - Matriz Gerada
8 25 77 71
7 73 73 81
38 68 44 81
28 23 66 83
---------- 3 - Matriz Gerada
4090 11018 18751 20746
2833 5770 9434 11120
4781 11572 13345 17244
5520 9721 13284 17233
Rectangle

Fonte -Autor

Como forma de estudar o comportamento do princípio da localidade espacial, que em resumo diz que a probabilidade de acessar um dado e e instruções são maiores em endereços próximos àqueles acessados recentemente. Foi criado um segundo código serial onde a seleção do index da linhas realizado pelo primeiro looping seria de forma aleatória. Para isso foi criado uma lista de index de 0 até o valor total de linhas da matriz menos um e a partir de uma função de embaralhamento, mostrada na figura 5, seria embaralhada. Assim o primeiro looping iria percorrer essa lista de index no final calculando aleatoriamente as linhas da matriz resultante.

Figura 5- Função emparelhamento de listas

```
void shuffle(int *array, int n)
{
    srand(10);
    if (n > 1)
    {
        int i;
        for (i = 0; i < n - 1; i++)
        {
          int j = i + rand() / (RAND_MAX / (n - i) + 1);
          int t = array[j];
          array[j] = array[i];
          array[i] = t;
        }
    }
}
```

Fonte: Autor

Por fim será analisando o código que possuir o melhor tempo de execução será utilizado para análise com o código paralelo.

Programa Paralelo

Para o modelo paralelo foi pensado em uma divisão do número de linhas igualmente para cada core da aplicação(contando com o processo pai) e assim cada thread iria calcular de forma serial cada bloco de linhas que ficou responsável. Como podemos ver na figura 6.

Figura 6- função utilizado por cada thread

```c
void *MultMatrix(void * rank){
    long my_rank = (long) rank;
    int i,j,x;
    int aux = 0;
    int local_m = row/(thread_count+1);

    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m -1 ;
    printf("%d - first %d - last %d - local_m %d -\n",my_rank,my_first_row,my_last_row,local_m);
    for ( i = my_first_row; i <= my_last_row; i++)
    {

        for (j = 0; j < row; j++)
        {

            for (x = 0; x < row; x++)
            {
                aux += matrizA[i][x] * matrizB[x][j];
            }
            matrizC[i][j] = aux;
            aux = 0;
        }

    }
    return NULL;
```

Fonte - Autor

Com a utilização da biblioteca pthread foi realizada a criação das threads(figura 7) e o enviou das função mostrada na figura 6, posterior a isso o processo main realiza o cálculos das linhas que ficou responsável e ao final espera as demais threads finalizarem sua execução para finalizar o processo.

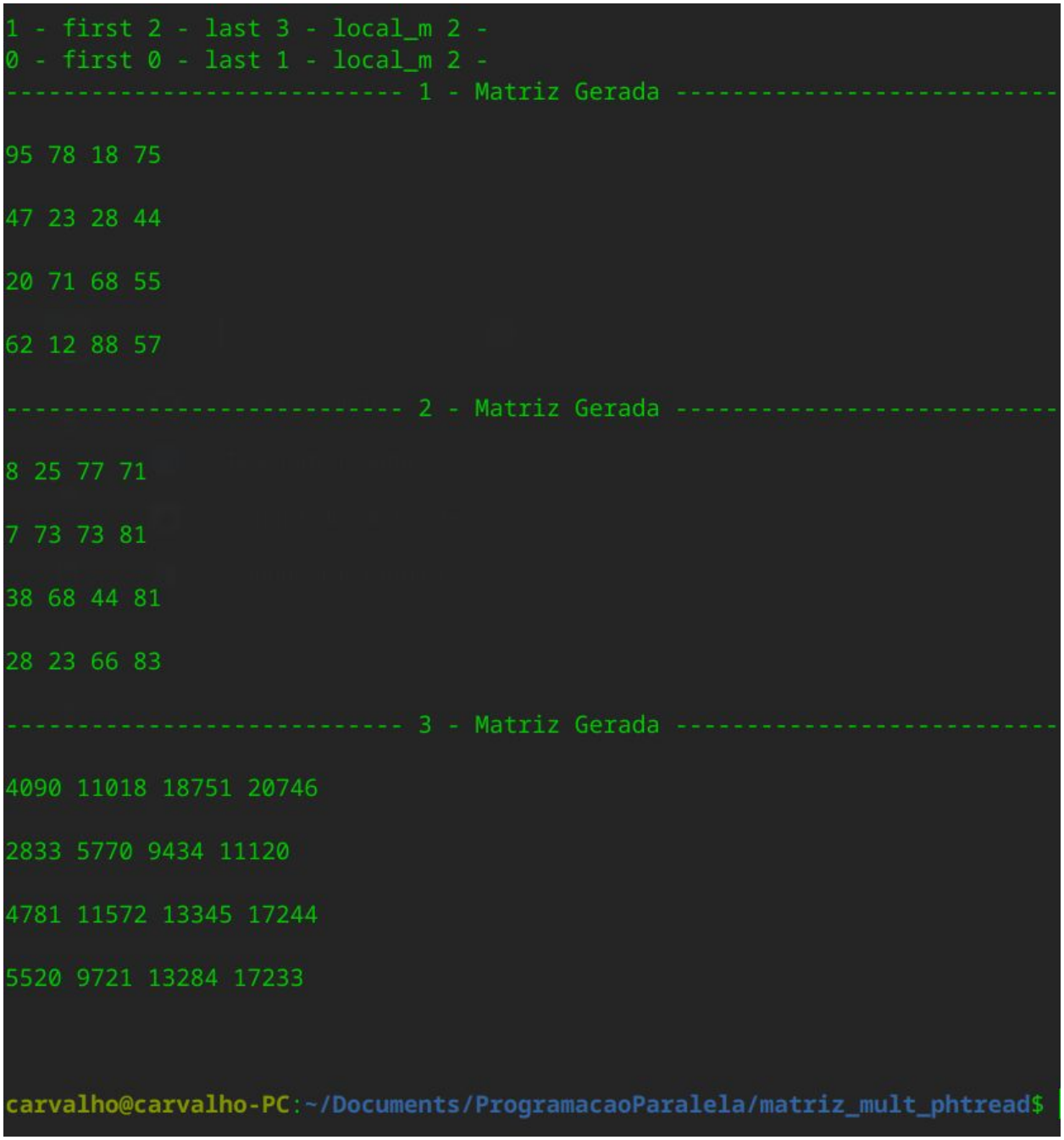
Figura 7- Código da criação de threads e execução da função

```c
pthread_t* thread_handles;
thread_count = strtol(argv[1],NULL,10);
thread_handles = (pthread_t*)malloc(thread_count*sizeof(pthread_t));
for (thread = 0; thread < thread_count; thread++)
{
    pthread_create(&thread_handles[thread],NULL,MultMatrix,(void*)thread);
}
MultMatrixMain(thread_count);
for (thread = 0; thread < thread_count; thread++)
{
    pthread_join(thread_handles[thread],NULL);
}
```

Fonte -Autor

Um exemplo de execução do código pode ser visto na figura 8, com a utilização de uma matriz 4x4 e utilização de 2 cores.

Figura 8- Execução Código paralelo



Fonte -Autor

Resultados

A primeira análise que iremos realizar é o comparativos dos tempos médios resultados da execução dos dois códigos seriais.

Figura 9- Comparativo do tempos de execução dos códigos seriais

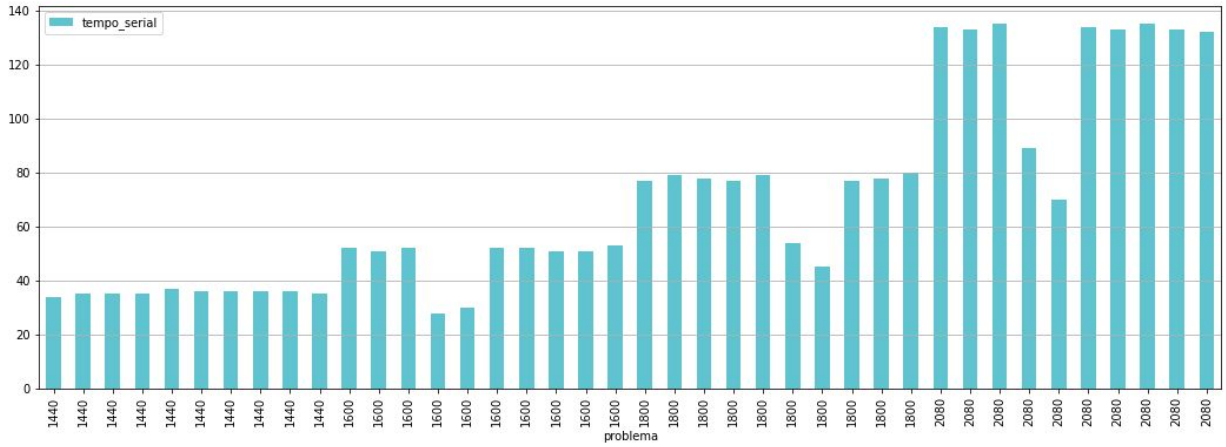
problema	tempo_serial	tempo_serial_rand
1440	35.5	36.3
1600	47.2	51.6
1800	72.4	79.8
2080	122.8	135.5

Fonte -Autor

Na figura 9 temos o “tempo_serial” como tempo de execução das linhas das matrizes em sequência enquanto “tempo_serial_rand” o tempo de execução da seleção aleatórias de linhas. Como já previsto pelo princípio da localidade espacial, a diferença das médias dos tempos de execução do primeiro código se tornaram cada vez mais relevante de acordo com aumento do problema, chegando a uma diferença de aproximadamente 13 segundos na execução de uma matriz de 2080 x 2080.

Tendo confirmado isso, iremos utilizar ,a partir de agora, somente os tempos de execução do primeiro código para quesito de análises. A figura 10 temos o gráfico de barras que representa todos tempos de execução do código serial.

Figura 10- Tempos de execução código serial



Fonte -Autor

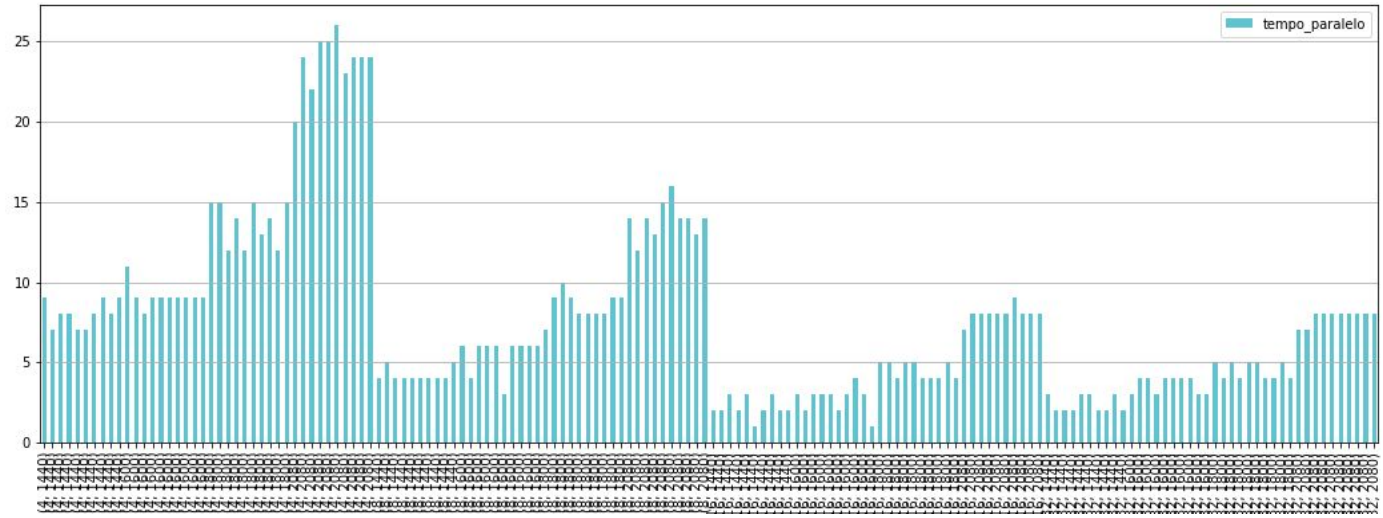
Nas figuras 11 vemos os tempos médios de execução do código paralelo e em seguida figura 12 temos os histórico de tempos de execução mostrados por barrar de todas as execuções em cada quantidade de cores e tamanho de problema.

Figura 11- Tabela de tempos médios de execução do código paralelo

core	problema	tempo_paralelo
4	1440	8.0
4	1600	9.1
4	1800	13.7
4	2080	23.7
8	1440	4.2
8	1600	5.5
8	1800	8.5
8	2080	13.9
16	1440	2.2
16	1600	2.7
16	1800	4.5
16	2080	8.0
32	1440	2.4
32	1600	3.6
32	1800	4.5
32	2080	7.8

Fonte -Autor

Figura 12- Tempos de execução código paralelol



Fonte -Autor

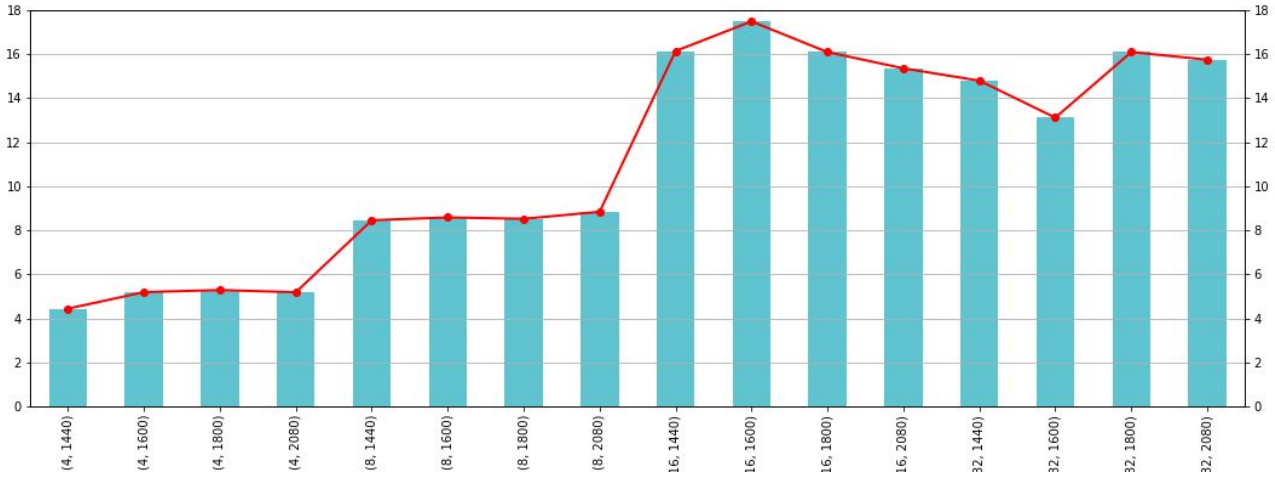
Foi concatenado as tabelas de média de tempos mostrados nas figuras 11 e 9 e calculamos o Speedup (tempo serial/tempo paralelo) e a eficiência (Speedup/ números de processos) e geramos a tabela a seguir(figura 13). E para melhor visualização foi gerado os gráficos de Speedup e eficiência(figura 14 e 15, respectivamentes).

Figura 13- tabela resultante do estudo

core	problema	tempo_paralelo	tempo_serial	speedup	eficiencia
4	1440	8.0	35.5	4.437500	1.109375
4	1600	9.1	47.2	5.186813	1.296703
4	1800	13.7	72.4	5.284672	1.321168
4	2080	23.7	122.8	5.181435	1.295359
8	1440	4.2	35.5	8.452381	1.056548
8	1600	5.5	47.2	8.581818	1.072727
8	1800	8.5	72.4	8.517647	1.064706
8	2080	13.9	122.8	8.834532	1.104317
16	1440	2.2	35.5	16.136364	1.008523
16	1600	2.7	47.2	17.481481	1.092593
16	1800	4.5	72.4	16.088889	1.005556
16	2080	8.0	122.8	15.350000	0.959375
32	1440	2.4	35.5	14.791667	0.462240
32	1600	3.6	47.2	13.111111	0.409722
32	1800	4.5	72.4	16.088889	0.502778
32	2080	7.8	122.8	15.743590	0.491987

Fonte -Autor

Figura 14- gráfico de Speedup



Fonte -Autor

Figura 15- gráfico de eficiencia

Fonte -Autor

Em uma análise partindo da figura 11. Vemos uma queda expressiva nos tempos de execução em relação aos códigos em paralelo para o em série, mostrando um bom aproveitamento de soluções paralelas para multiplicações de matrizes quadradas. O que mais chama atenção são os valores speedup e eficiência onde obtemos valores de ganhos perfeitos nas utlização de 4, 8 e 16 threads tendo tendo valores de eficiência de 0,95 até 1,32 e Speedup com valores próximos ao números de cores.

Quando visualizamos os resultados de 32 cores percebemos valores de tempo iguais aos 16 e uma queda brusca nos valores Speedup e eficiência, que em um ambiente perfeito seriam de aproximadamente 1 e 32 temos os resultados aproximados de 15 e 0,50. Esse efeito pode ser justificado pelo limitação na diminuição da parcela de tempo da execução em paralelo fazendo com que seu tempo não consiga ser diminuído e que possivelmente grande parte do tempo final é resultante do tempo serial do código.

Ao fim podemos classificar o código como escalável por ter conseguido manter aproximadamente constante os valores de eficiência enquanto aumentava o número de cores e tamanho do problema .

Considerações Finais

Em resumo os resultados dos estudo se apresentam gratificantes, obtendo resultados perfeito na utilização do codigo paralelo em até 16 cores .Para melhor visualizão os códigos (serial e paralelo) juntamente com o notebook em python, utilizado para realizar a analise dos dados e construção dos graficos, estão disponibilizados no github pelo link :

https://github.com/PdrCarvalho/matriz_mult_phtread