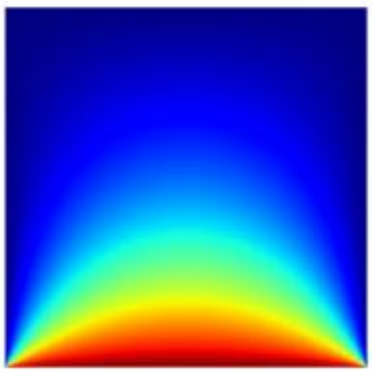
Universidade do Minho
Escola de Engenharia

UNIVERSIDADE DO MINHO

TRABALHO - OpenMP

SIMULAÇÃO DO PROCESSO DE DIFUSÃO DE CALOR

para N_MAX iterações



Bruno Rodrigues pg41066

Carlos Alves pg41840

UC: Computação Paralela

2019/2020

ÍNDICE

1 INTRODUÇÃO

Primeiramente, a maioria dos programas de computador são desenvolvidos para computação sequencial, visto que as instruções destes programas somente podem ser executadas por vez, isto depois da instrução for concluída. A computação paralela aparece propositadamente para evitar a execução de instruções de forma sequencial, isto é, usa vários elementos de processamento simultaneamente para resolver um problema. Basicamente o que ele faz é dividir o problema em partes independes, permitindo que cada elemento de processamento possa "correr" a sua parte do algoritmo simultaneamente com os diversos elementos. Computação Paralela é também uma forma de programação que oferece os mesmos resultados que a versão sequencial, mas em muito menos tempo e com mais eficiência (Mais ganho).

Com isto, este trabalho prático desenvolvido na Unidade Curricular Computação paralela focar-se-á principalmente na abordagem do algoritmo de difusão de calor(2D) em **OpenMP**, onde será possível concluir objetivamente resultados entre duas versões do algoritmo: Sequencial e Paralela. Afirmando uma vez mais os benefícios de se colocar em prática os diversos conceitos de paralelismo nos algoritmos sequências.

2 REPRESENTAÇÃO E CALCULO DIFUSÃO DE CALOR

Primitivamente, o caso em questão trata-se da simulação do processo de difusão de calor numa matriz de dimensão 2, para **N_MAX** iterações. O processo de difusão de calor/processo de propagação de calor, pode ser definido como a distribuição de calor evolui ao longo de um determinado material, seja sólido, líquido ou mesmo um material gasoso. O tipo de material influência em grande parte o tempo que demora a ocorrer a dispersão de calor.

Dito isto, existem diversas formas como o calor se propaga, mas visto que o caso em questão é baseado numa forma de difusão de calor especifica, *steady-state conduction*, apenas nos iremos focar nesta forma.

Este processo consiste numa região quadrada de duas dimensões que inicialmente se encontra fria, valor zero na matriz, e aquecida no topo. Neste caso, consideramos que este processo não fosse afetado por qualquer variante, incluindo o tipo de material e ainda o local que se encontra. Deste modo, é possível observar que a quantidade de energia que lhe é colocada e a que sai é a mesma, sendo assim o resultado final é a região quadrada ficar com uma temperatura constante.

Como referido acima, a região que irá ser representada é de uma região quadrada de duas dimensões, para isto foi então desenvolvida uma matriz quadrada ***M[N,N]***, sendo o *N* o número de linhas e colunas da matriz. De seguida, passou-se a definir os valores adotados nesta implementação, de modo que fosse possível determinar quais zonas da matriz se encontravam quentes ou frias. Para isto o valor 0 representa o estado (mínimo) mais frio e o valor 100 o estado(máximo) mais quente.

Com a representação desenvolvida, foi necessário definir o estado inicial da matriz. Como considerado anteriormente, a matriz seria apenas aquecida no seu topo ou na sua base, deste modo apenas uma linha é preenchida com o valor máximo - 100, caraterizando essa linha como

a mais quente da matriz, enquanto que o resto da matriz estaria preenchida com valor mínimo possível - 0, neste caso estaria totalmente fria.

100	100	100	100	100	100	100
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Fig. 1 – Estado inicial da matriz

No cálculo da difusão do calor foi seguida a equação descrita no enunciado, que consiste essencialmente numa expressão de propagação. Afetando cada posição da matriz, isto é, para cada posição***[i,j]*** da matriz é calculado a média das posições imediatamente acima, abaixo, direita e esquerda e a própria posição em questão.

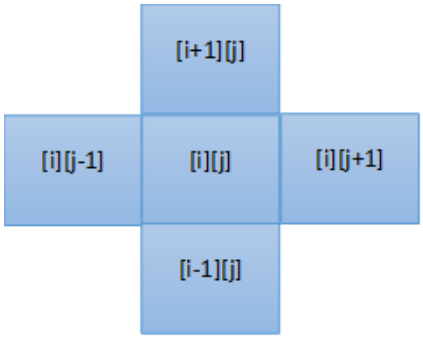


Fig. 2 – Cálculo da difusão

Mas como era necessário calcular sucessivamente os diversos "estados" de cada posição da matriz sem que esta não fosse influenciado pelo novo resultado, foi então incorporado uma outra matriz, onde fosse armazenado os resultados de cada posição da última iteração da matriz inicial. Para tal, **G1** é representada como a nova matriz e **G2** como a matriz "antiga". Deste modo, é possível em cada iteração efetuada seja copiado os dados de **G1** para **G2** de forma a esta ser usada nos cálculos das iterações seguintes. Este processo pode ser visto no código (Anexo-A).

$$\boldsymbol{G1[i][j] = \frac{G2[i-1][j] + G2[i+1][j] + G2[i][j-1] + G2[i][j+1] + G2[i][j]}{5}}$$

3 ALGORITMO

No que diz respeito à comparação, foram desenvolvidas duas versões: uma sequencial e outra paralela. Deste modo, foi possível observar diferenças na execução nos dois códigos que iremos descrever mais a baixo.

3.1 SEQUENCIAL

Para a implementação de uma simulação de difusão de calor, começamos por declarar e alocar espaço em memória para duas matrizes, *heat_new* e *heat_old*.

Para inicializar as matrizes, foi necessário preencher as linhas *heat_new[0][i]* a 100, e o resto a 0. Estes serão os valores iniciais que permitirão começar o processo de iteração.

Existe uma função que receberá o número de iterações, como também o tamanho da matriz, e irá iterar até N iterações, sendo que para cada iteração, são calculados novos valores para os pontos interiores, e depois guardados em *heat_old*.

O facto de o código ser de complexidade reduzida, e mais o facto de nos termos guiado pelo pseudocódigo fornecido em aula, a quantidade de otimizações possíveis é limitada, sendo que umas das alterações efetuadas foi na maneira como a alocação das matrizes eram alocadas.

Inicialmente as matrizes estavam a ser alocadas da seguinte forma:

```
float (*heat_new) [n] = malloc(sizeof(float[n][n]))
```

Sendo que, no código sequencial, comparado com as formas mais comuns de inicializar *arrays*, como esta,

```
float **heat_new = malloc(n * sizeof(float *));
       for(i = 0; i < nrows; i++)
       heat_new[i] = malloc(matrix_size * sizeof(float));
```

os tempos de execução eram semelhantes, e um pouco melhores, esta era a forma como também tinha sido colocado no código paralelizado. Mais tarde, na execução dos testes, verificamos que o código paralelizado conseguia ser 3 vezes mais lento devido a maneira como a memória para as matrizes era alocada, e como tal, foi modificado, e alterado para o que agora se encontra.

Em anexo pode ser consultado o código completo da implementação sequencial.

3.2 PARALELO

Para o algoritmo paralelizado foram apenas adicionadas as diretivas do **OpenMP**, sem se presenciar quaisquer alterações significativa do código. Foi adicionado a diretiva *#pragma omp parallel num_threads(N)*, que define uma região paralela, que vai ser executado por N *threads*.

Neste sentido, o uso da diretiva *#pragma omp parallelshared(new_M,old_M) private(i,j) num threads(N)* também foi testada, permitindo especificar que as variáveis *new_M* e *old_M* seriam compartilhadas entre as *threads*, e que cada *thread* teria também uma estância própria de cada variável "i" e "j", que seria executado em N *threads*.

Após a realização de alguns testes com cada uma das formas de aplicar as diretivas, foi concluído que a diferença entre ter e não ter as clausulas *private* e *shared* não eram significativas, e como tal não foram incorporadas no código. Para cada ciclo for é também acrescentado a diretiva *#pragma omp for*. Esta diretiva instrui o compilador a distribuir as iterações do *loop* dentro da região paralela entre as *threads*.

Esta implementação encontra-se no anexo A-2, demonstrando exatamente o que anteriormente foi descrito.

4 TESTES E RESULTADOS

Após desenvolvidos as duas versões do código, sequencial e Paralelizado, iniciamos a fase de testes de modo a ser possível analisá-los posteriormente, a todos os níveis. Para isso fizemos uso do Search-computer-652-2, onde realizamos os testes.

Nos testes foi utilizado o contador da biblioteca **OpenMP** *omp_get_wtime()* de forma a ser possível obter resultados dos tempos de execução, em milissegundos, da versão sequencial e paralela do algoritmo. Com base no - tempo de execução - a métrica utilizada para medir o desempenho das duas versões foi possível calcular os ganhos obtidos em cada execução paralela e compara-los com os valores da execução sequencial. Os valores registados e calculados podem ser consultados no Anexo B.

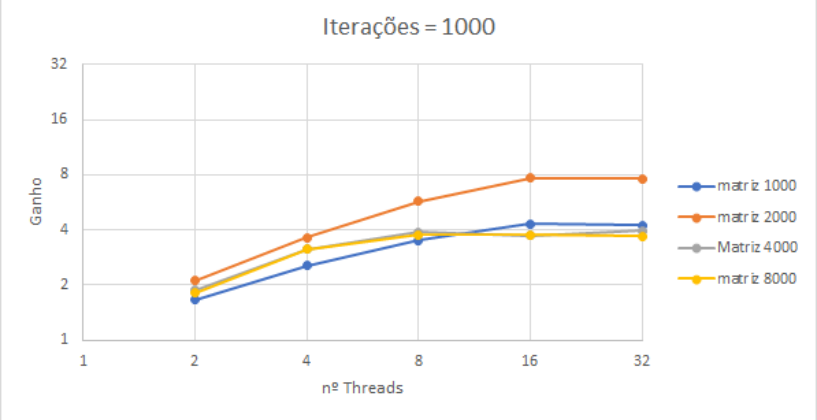


Fig. 3 – Ganhos para N_Max 1000

4.1 ANÁLISE

Para entender melhor os resultados obtidos, faremos uma análise onde iremos discutir os resultados consoante o número de *threads* usadas:

4.1.1 2 *threads*

Para duas *threads*, o ganho ideal esperado seria duas vezes mais rápido do que o código sequencial.

Podemos observar que houve um ganho significativo perante o código sequencial, sendo que para cada tamanho de matriz e números máximos de iterações, os ganhos foram entre 1,5 a 2 vezes mais rápido. A matriz de 2000x2000 atingiu o ganho ideal sendo que independente do número máximo de iterações efetuadas, o ganho perante o sequencial foi sempre 2 vezes maior.

A matriz com a menor performance foi a matriz de 1000x1000, onde o ganho perante a sequencial manteve-se dos 1,5, sendo a matriz que mantem a pior performance perante todas as outras testadas.

4.1.2 4 *threads*

Para quatro *threads*, o ganho ideal esperado seria quatro vezes mais rápido do que o código sequêncial.

Comparativo ao ganho conseguido com duas *threads*, ao utilizar quatro *threads* conseguimos resultados que são proporcionalmente menores do que os obtidos anteriormente, sendo que o maior ganho atingido foi pela matriz de 2000x2000, onde os ganhos encontram-se entre 3 e 3,5 vezes maiores. Ao contrário do que se verificava para duas *threads*, os ganhos começam agora também a variar mais conforme o número máximo de iterações efetuados, onde podemos verificar que quando maior é o número de iterações efetuado, menor é o ganho conseguido em relação ao código sequencial. ´

Podemos também verificar que esta situação não se aplica a matriz de 8000x8000, onde os ganhos desta são maiores quando o número máximo de iterações se encontra nos 8000.

4.1.3 8 *threads*

Para oito *threads*, existe ganhos maiores comparativamente ao uso de menos *threads*, mas os ganhos por *thread* são menores do que os anteriores. A matriz de 2000x2000 é a única que continua a ter *speedups* mais acentuados do que os outros tamanhos testados.

4.1.4 16 – 32 threads

Podemos observar nos gráficos que os ganhos obtidos quando usados 16 e 32 *threads* são praticamente semelhantes em todas as matrizes, até havendo perda de performance com a utilização de 32 *threads*, como acontece no caso da matriz 2000x2000, quando as iterações máximas são 1000 e 2000.

4.2 CONCLUSÃO DOS RESULTADOS

Observando os gráficos, podemos constatar que a matriz que obtém maiores ganhos em relação ao código sequencial é a matriz de 2000x2000, que mantém ganhos acentuados até serem utilizados 16 *threads*, onde depois observamos para todos os tamanhos uma estabilização dos *speedups* conforme o aumento de *threads* usadas.

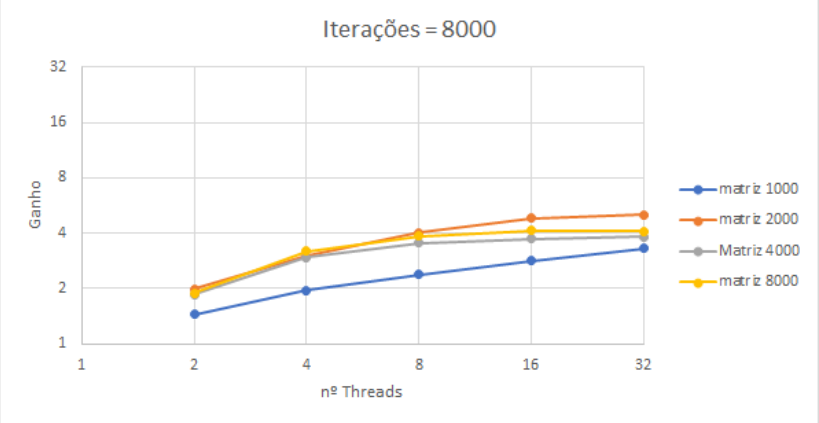


Fig. 4 – Ganhos para N_Max = 8000

Podemos também ver que o maior ganho por *thread* acontece quando são utilizadas duas *threads*. Isto verifica-se pelo facto de começar a existir mais memória cache para a realização de

operações (comparado com o código sequencial), sendo que devido à utilização de mais cores a memória cache destes passa a estar disponível para utilização, o que irá evitar menos chamadas à memória *ram*. Foi observado que quando são usados 16 *threads* para cima, existe uma estabilização dos ganhos, sendo que os valores passam a ser constantes (muito semelhantes).

Acreditamos que isto acontece devido ao facto que quando utilizamos 16 *threads* para cima (32 *threads* foi o máximo que testamos) os *threads* utilizados começam a ser os virtuais, e logo não possuem memória cache própria, para além que a largura de banda começa a deixar de ser o suficiente, começando a existir uma restrição na transição de dados. Problemas associados ao acesso a memória também afetam a performance.

5 CONCLUSÃO

Em suma, acreditamos que os resultados obtidos nos testes foram o esperado, tendo em conta o algoritmo e as capacidades da máquina utilizada, sabendo que estes são os maiores fatores para a obtenção de valores desejáveis. A limitação do algoritmo em si impende grandes otimizações, e como tal será sempre limitando.

Para concluir, gostaríamos de mencionar o quão impressionados ficamos com a velocidade obtida com uma simples adição de diretivas de **OpenMP**. Para pessoas como nós, que nunca tínhamos trabalhado com paralelização de códigos sequencias, ficamos surpresos que os ganhos, mesmo que não tenham sido os ideais, sejam ainda bastante significativos, sendo que é um método bastante eficaz e demonstra o quão essencial é existir uma boa otimização por parte do programador.

6 REFERÊNCIAS

Quinn, M. J. (2013). *Parallel Programming in C with MPI and OpenMP.* Oregon: MC Grawn Hill.

(Quinn, 2013) (Quinn, 2013)

7 ANEXOS

7.1 ANEXO A - CÓDIGO

7.1.1 Sequencial

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

void iteration(int size, float **new_M, float **old_M,  int maxIt)
{

    //Itera ate maxIt iteracoes

    for(int i = 0; i < maxIt; i++)
    {
            //Calcula os novos valores dos pontos interiores para n
ew_M.
            for(int i = 1; i < size-1; i++)
            {
                  for(int j = 1; j < size-1; j++)
                       {
                         new_M[i][j] =(old_M[i-1][j]
                             + old_M[i+1][j]
                             + old_M[i][j-1]
                             + old_M[i][j+1]
                             + old_M[i][j])/5;
                  }
            }

        //Guarda a ultima solucao em old_M

        for(int i = 0; i < size; i++)
        {
                for(int j = 0; j < size; j++)
                      {
                        old_M[i][j] = new_M[i][j];
                   }
         }
    }
}
```

```
void printMatrix(int size,float (*M_New)[size], FILE *file)
{
    for(int i = 0; i < size; i++)
  {
            for(int j = 0; j < size; j++)
           {
                fprintf(file, "%.7f ", M_New[i][j]);
            }
        fprintf(file, "\n");
    }
}

int main(int argc, char * argv[])
{

    int matrix_size;
    int n_max_it;
    double start, final;
    FILE *file = NULL;

     //arg1 - tamanho matrix,
    matrix_size = atoi(argv[1]);
    //arg2 - numero maximo de iteraçeos
    n_max_it = atoi(argv[2]);

    //alocação das matrizes
      float **heat_new = malloc(matrix_size * sizeof(float *));
    float **heat_old = malloc(matrix_size * sizeof(float *));

    for(int i = 0; i < matrix_size; i++)
    {
        heat_new[i] = malloc(matrix_size * sizeof(float));
        heat_old[i] = malloc(matrix_size * sizeof(float));
    }
     //Preenche a primeira linha a 100
    for(int i = 0; i < matrix_size; i++)
    {
        heat_new[0][i] = (float) 100;
    }
    //Preenche o restante a 0
    for(int i = 1; i < matrix_size; i++)
    {
          for(int j = 0; j < matrix_size; j++)
           {
                heat_new[i][j] = (float) 0;
            }
    }
```

```
    //Executa N_max iteraçoes, indicando o tempo de execução
    start = omp_get_wtime();
    iteration(matrix_size,heat_new,heat_old,n_max_it);
    final = omp_get_wtime();
    //Imprimir em milsegundos o tempo de execuçao
    printf("%.5f\n", (final -start)*1000);

     for(int i = 0; i < matrix_size; i++)
    {
        free(heat_new[i]);
         free(heat_old[i]);
    }
    //Libertar memoria utilizada
    free(heat_new);
    //Libertar memoria utilizada
    free(heat_old);

    return 1;

}
```

7.1.2 Paralelo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

void iteration(int size, float **new_M, float **old_M,  int maxIt,
int threads)
{

    //Itera ate maxIt iteracoes

    for(int i = 0; i < maxIt; i++)
    #pragma omp parallel num_threads(threads)
    {
            //Calcula os novos valores dos pontos interiores para n
ew_M.
            #pragma omp for
            for(int i = 1; i < size-1; i++)
        {
                  for(int j = 1; j < size-1; j++)
            {
                        new_M[i][j] =(old_M[i-1][j]
                            + old_M[i+1][j]
                            + old_M[i][j-1]
                            + old_M[i][j+1]
                            + old_M[i][j])/5;
                  }
            }

        //Guarda a ultima solucao em old_M
        #pragma omp for
        for(int i = 0; i < size; i++)
        {
                 for(int j = 0; j < size; j++)
            {
                        old_M[i][j] = new_M[i][j];
                  }
            }
       }
}

void printMatrix(int size,float (*M_New)[size], FILE *file)
{
    for(int i = 0; i < size; i++)
  {
            for(int j = 0; j < size; j++)
        {
```

```
                fprintf(file, "%.7f ", M_New[i][j]);
            }

        fprintf(file, "\n");
    }

}

int main(int argc, char * argv[])
{

    int matrix_size;
    int n_max_it;
    int num_threads;
    double start, final;
    FILE *file = NULL;

     //arg1 - tamanho matrix,
    matrix_size = atoi(argv[1]);
    //arg2 - numero maximo de iteraçeos
    n_max_it = atoi(argv[2]);

    num_threads = atoi(argv[3]);
    //alocação das matrizes
    float **heat_new = malloc(matrix_size * sizeof(float *));
    float **heat_old = malloc(matrix_size * sizeof(float *));

    for(int i = 0; i < matrix_size; i++)
    {
        heat_new[i] = malloc(matrix_size * sizeof(float));
        heat_old[i] = malloc(matrix_size * sizeof(float));
    }

     //Preenche a primeira linha com 100
    for(int i = 0; i < matrix_size; i++)
    {
        heat_new[0][i] = (float)100;
    }
    //Preenche o resto com 0
    for(int i = 1; i < matrix_size; i++)
    {
          for(int j = 0; j < matrix_size; j++)
        {
                heat_new[i][j] = (float)0;
            }
    }
```

```
    //Executa n_max_it iteraçoes, indicando o tempo de execução
    start = omp_get_wtime();
    iteration(matrix_size,heat_new,heat_old,n_max_it, num_threads);
    final = omp_get_wtime();
    //Imprimir em milsegundos o tempo de execuçao
    printf("Time seq: %.5f ms \n", (final -start)*1000);

    for(int i = 0; i < matrix_size; i++)
    {
        free(heat_new[i]);
         free(heat_old[i]);
    }
    //Libertar memoria utilizada
    free(heat_new);
    //Libertar memoria utilizada
    free(heat_old);

    return 1;

}
```

7.2 ANEXO B - DADOS

Matriz	Iterações	Threads	Tempo (Média /ms)	Ganho (Sequencial / Paralelizado)
1000	1000	0	1640.042	0
		2	993,4783	1,650808075
		4	640,556	2,560341328
		8	469,0245	3,496708594
		16	379,1949	4,325063444
		32	385,7995	4,25102158
	2000	0	3529.162	0
		2	2241.436	1,574509377
		4	1539.074	2,29304244
		8	1153.776	3,058793041
		16	953.5363	3,701130203
		32	922.544	3,825467403
	4000	0	7699.352	0
		2	5016.035	1,534947822
		4	3635.139	2,118035101
		8	2908.886	2,646838687
		16	2458.984	3,131111061
		32	2205.602	3,490816566
	8000	0	17109.32	0
		2	11818.83	1,447632295
		4	8767.153	1,951525199
		8	7188.492	2,380098635
		16	6047.26	2,829268131
		32	5193.281	3,294510734
2000	1000	0	7012.216	0
		2	3329.865	2,105855943
		4	1932.646	3,628298198
		8	1225.471	5,722057886
		16	915.3598	7,660611707
		32	918.8645	7,631392877
	2000	0	14418.24	0
		2	6893.434	2,091590345
		4	4264.2595	3,381182595
		8	2954.563	4,879990713
		16	2249.414	6,409776057
		32	2262.453	6,372835148
	4000	0	30790.8682	0
		2	14872.226	2,070360429
		4	9668.587	3,184629584
		8	6985.864	4,407596283
		16	5708.599	5,39376968
		32	5710.321	5,392143139
	8000	0	68188.3812	0
		2	34310.598	1,98738539
		4	22748.477	2,997492149
		8	17015.356	4,007461331
		16	14166.554	4,813335777

		32	13559.912	5,028674316
4000	1000	0	31719.0856	0
		2	16984.83	1,867495029
		4	10087.93	3,144261072
		8	8172.019	3,881425826
		16	8519.443	3,723140773
		32	8007.415	3,96121415
	2000	0	63446.0864	0
		2	34444.3	1,841990878
		4	20403.7	3,10953829
		8	17246.85	3,678705758
		16	16503.14	3,84448574
		32	16680.63	3,803578546
	4000	0	136039.25	0
		2	71990.61	1,889680474
		4	43348.59	3,138262398
		8	37108.28	3,666007964
		16	33944.0379	4,007750946
		32	33910.921	4,011664856
	8000	0	278795.8711	0
		2	150901.831	1,8475314
		4	94174.181	2,960427881
		8	79154.2361	3,522185101
		16	74926.54	3,720922801
		32	72830.4797	3,828010913
8000	1000	0	115105.0189	0
		2	63554.184	1,81113204
		4	36702.1333	3,136194236
		8	30775.365	3,740167465
		16	30546.5477	3,768184216
		32	31213.943	3,687615464
	2000	0	227465.5508	0
		2	129361.913	1,758365701
		4	75262.5026	3,02229587
		8	61136.6398	3,720609303
		16	61902.382	3,674584781
		32	61412.477	3,703898001
	4000	0	469837.8846	0
		2	268007.331	1,753078481
		4	156115.077	3,009561239
		8	125385.1103	3,747158522
		16	128120.135	3,667166637
		32	124397.9448	3,776894267
	8000	0	1041548.221	0
		2	553833.8371	1,880614999
		4	327901.7105	3,176403744
		8	271134.796	3,841440628
		16	253313.784	4,111691849
		32	255606.116	4,07481729

7.3 ANEXO C - GRÁFICOS

Com 1000 iterações

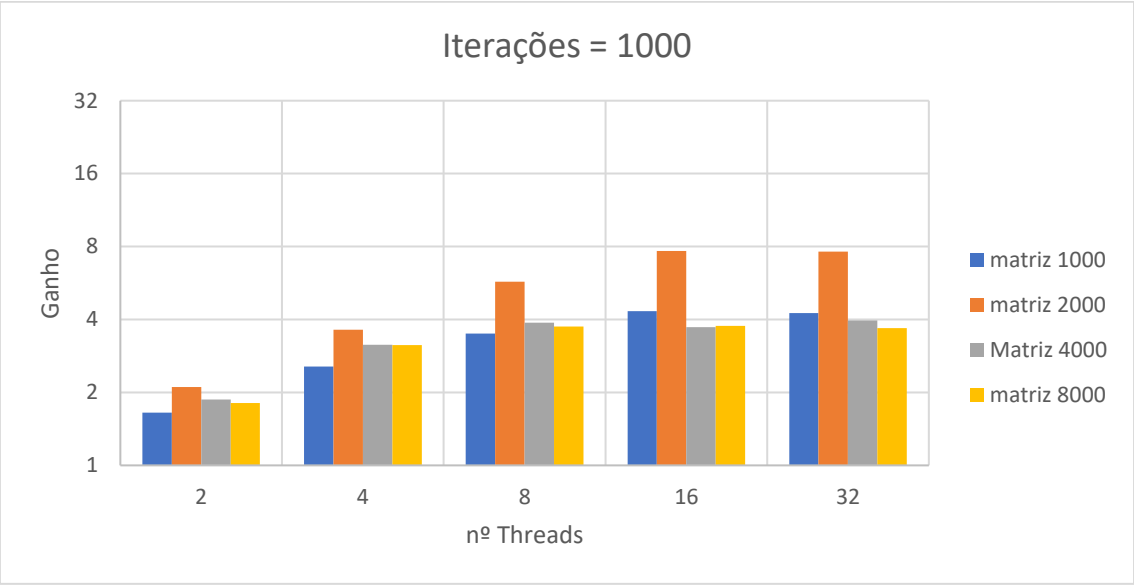


Gráfico de Colunas

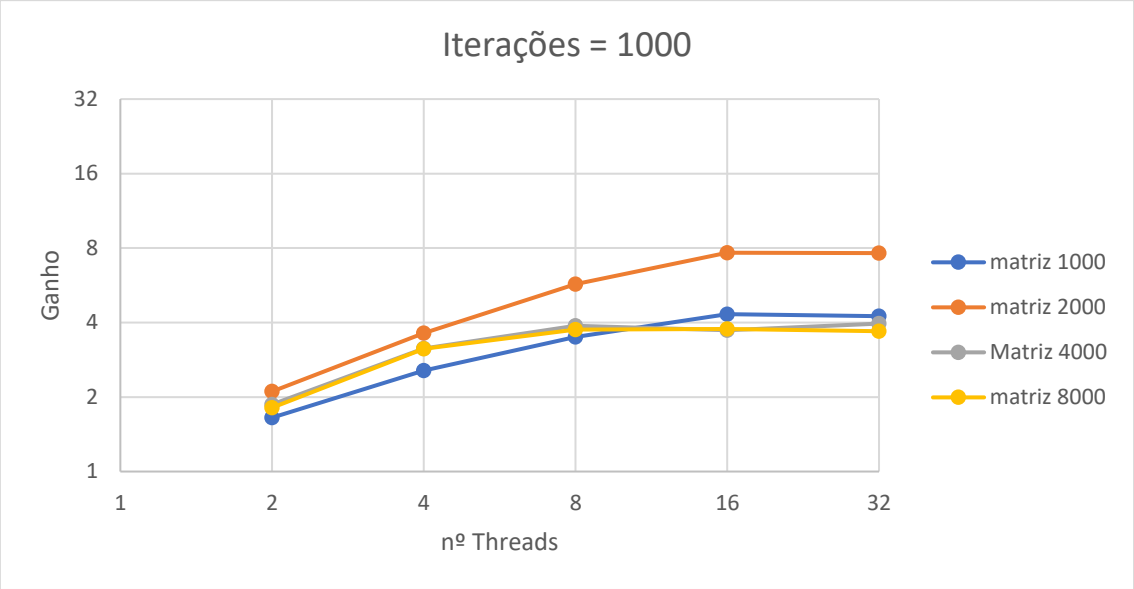


Gráfico de Linhas

Com 2000 iterações

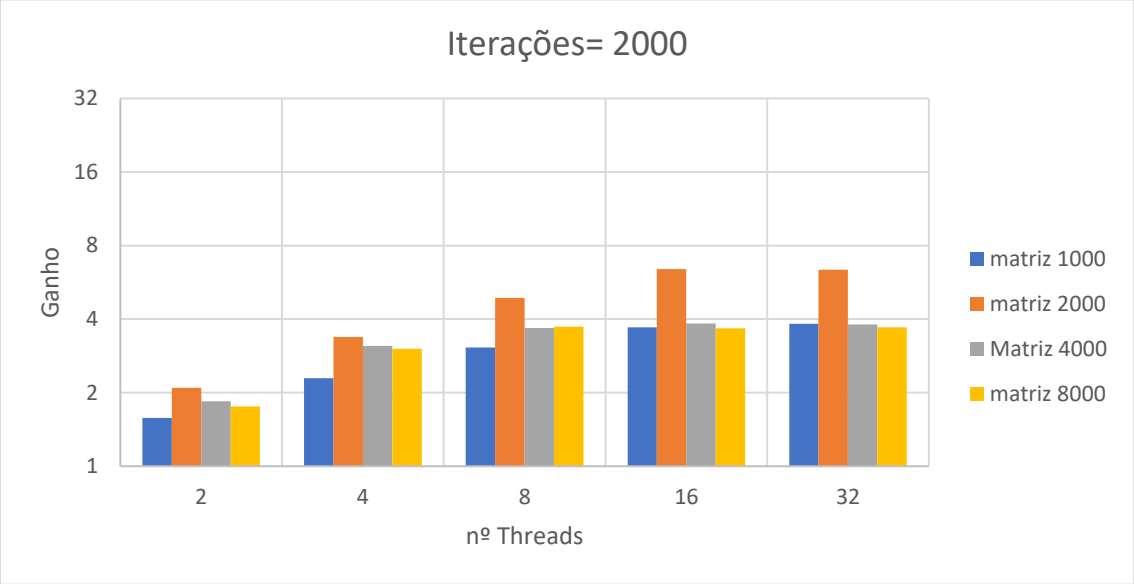


Gráfico de Colunas

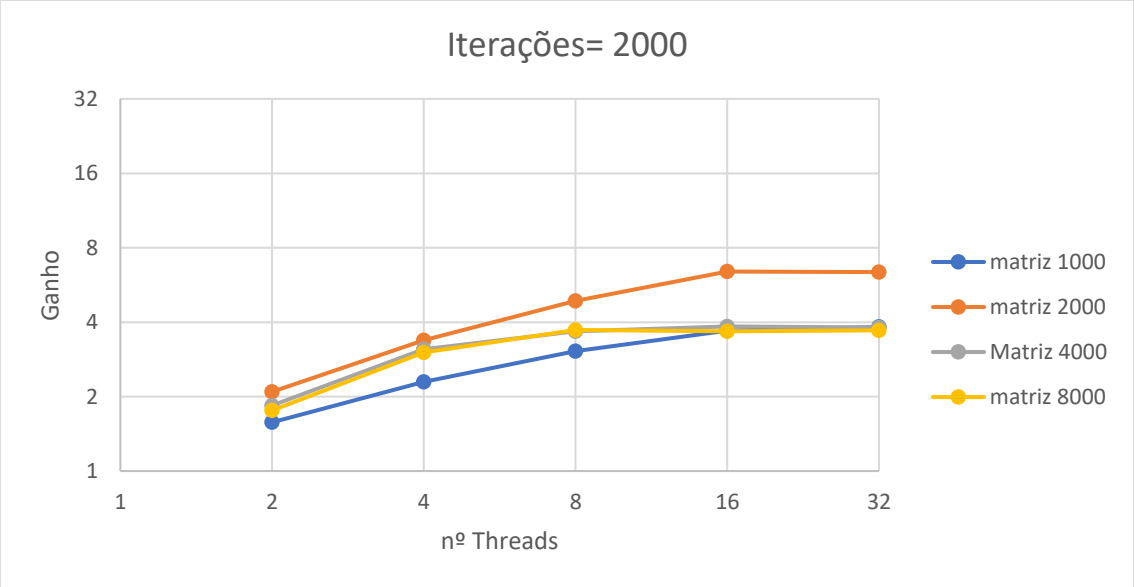


Gráfico de Linhas

Com 4000 iterações

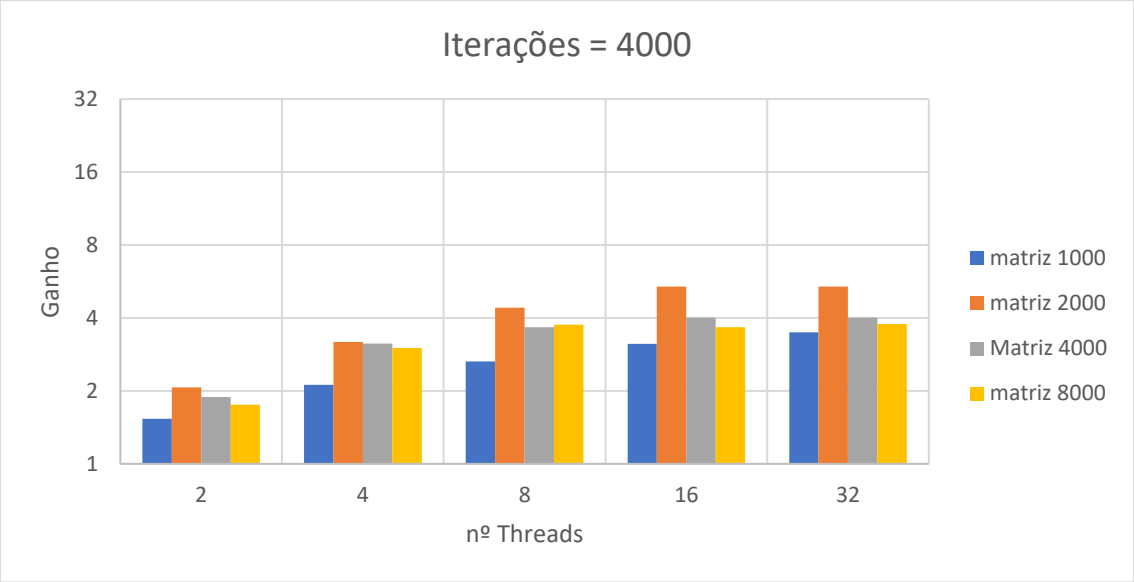


Gráfico de Colunas

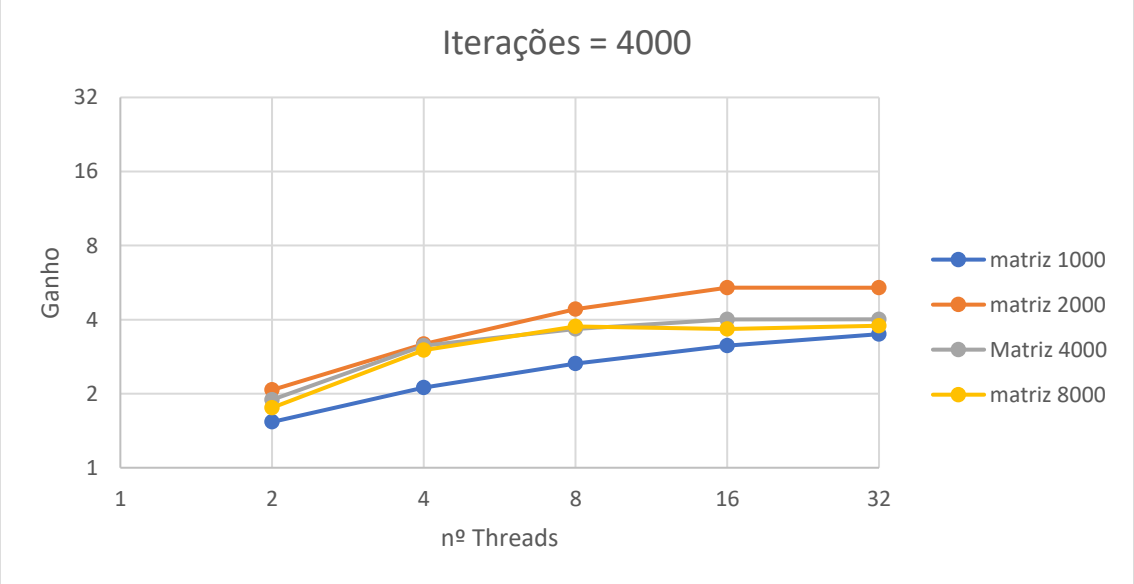


Gráfico de Linhas

Com 8000 iterações

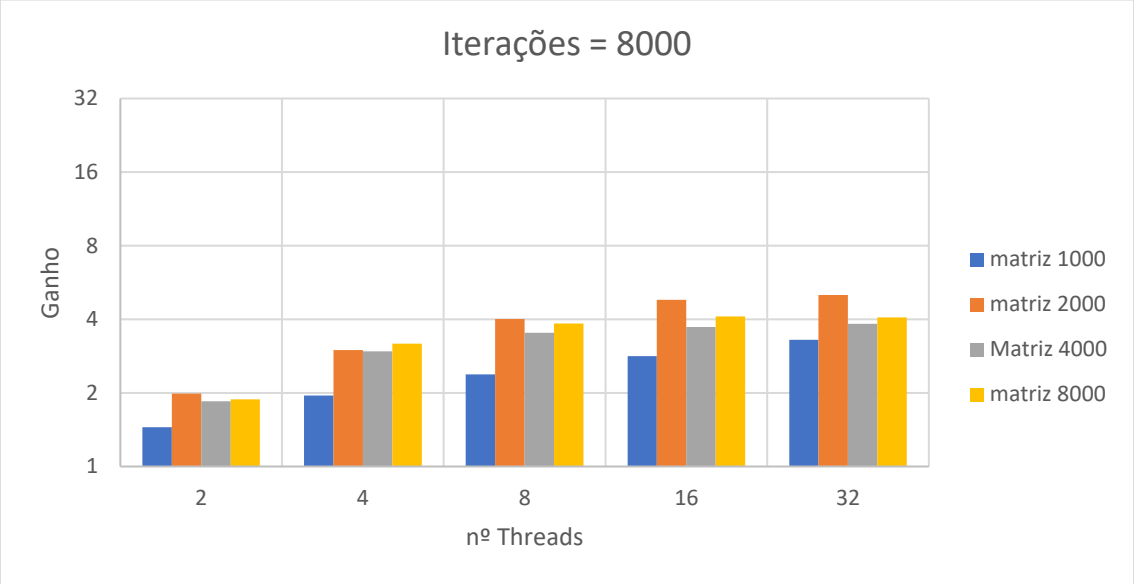


Gráfico de Colunas

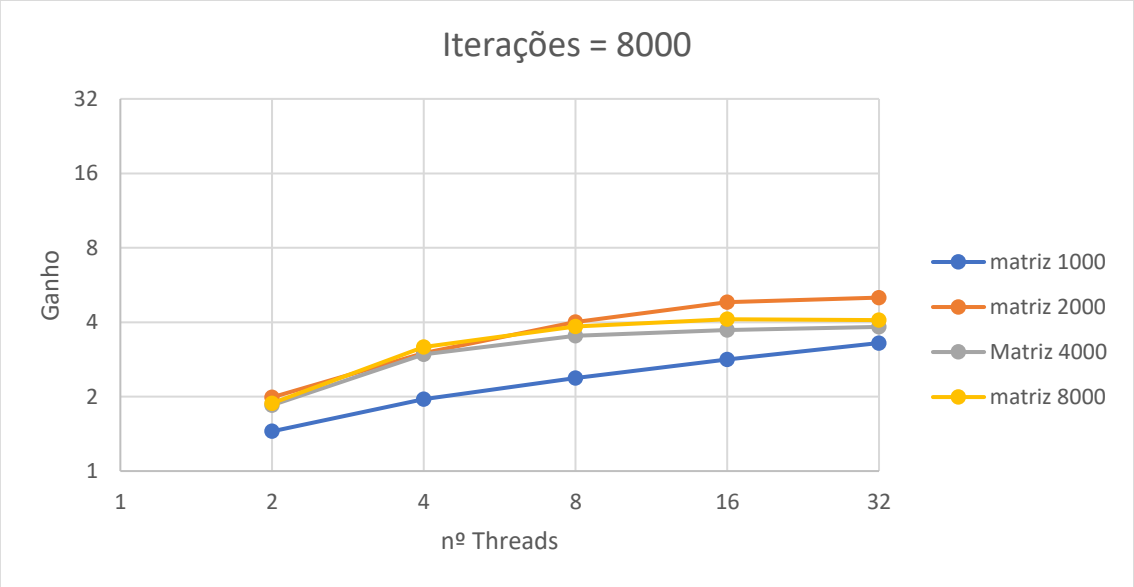


Gráfico de Linhas