Universidade do Minho
Escola de Engenharia

Universidade do Minho
Mestrado Integrado Engenharia Informática

Paradigmas de Computação Paralela

Simulação do Processo de Difusão de Calor

Paralelismo em OpenMP ®

João Lopes **A61077**
Nuno Moreira **A61017**

Braga, 8 de Novembro de 2016

Conteúdo

1. Introdução

A utilização de paradigmas de computação paralela pode ser, em muitos casos, uma forma de conseguir paralelizar códigos sequenciais por forma a reduzir os seus tempos de execução elevados. Contudo, nem sempre o expectável ocorre quando um certo código é paralelizado: na utilização de, por exemplo, quatro threads num código paralelo muitas vezes espera-se que o tempo de execução seja quatro vezes melhor quando comparado com o tempo de execução sequencial. Esta expectativa de ganho muito dificilmente acontece (dependendo dos algoritmos) devido a vários tipos de limitações de hardware da máquina (roofline, largura de banda dos barramentos, tamanho das memórias cache, etc) bem como à implementação do código paralelo que não tire proveito de todo o seu potencial (redução de acessos à memória, operações com dados contíguos, tamanho dos dados adequado para a memória cache, etc).

O trabalho desenvolvido consiste na aplicação de paradigmas de computação paralela, em sistemas de memória partilhada, por forma a avaliar o nível de escalonamento obtido numa versão de código paralelo em comparação à versão do mesmo código em formato sequencial, para o caso de estudo a tratar.

Por forma a poder obter resultados foi necessário começar por perceber o caso de estudo - Simulação do Processo de Difusão de Calor para N_MAX Iterações - seguido por uma possível implementação do seu código sequencial e paralelo, o mais eficiente possível.

Para perceber se e quanto a versão paralela do código é mais eficiente foi necessário executar ambas as versões do código, para diferentes valores de input, e comparar resultados. Esta comparação é então analisada e discutida no final do trabalho por forma a perceber quais as limitações no hardware utilizado e no código desenvolvido que são responsáveis pelos resultados obtidos.

1.1. Caso de Estudo

O caso de estudo desenvolvido é referente à simulação do processo de difusão de calor, para N_MAX iterações.

O processo de difusão de calor pode ser definido como a transferência de calor ao longo de um material. Existe uma grande quantidade de materiais que podem ser sujeitos a este tipo de processo, como sólidos, líquidos, gases, etc. A medida de tempo e dispersão pela qual o calor é propagado por um material é dependente das propriedades físicas desse material.

Existem diversas formas de difusão de calor sobre um material. A uma dessas formas é dado o nome de *steady-state conduction*, pela qual o caso de estudo é baseado. Assim, o processo de difusão de calor considerado consiste na existência de uma superfície quadrada, inicialmente fria, que é aquecida no seu lado superior. É considerado que o processo de difusão de calor não é influenciado pela propriedade do material da superfície e que a quantidade de calor que entra numa dada região é igual à quantidade de calor que sai. Isto leva a que ao fim de um certo tempo de difusão do calor sobre o material (N_MAX iterações), o gradiente de temperatura ao longo do material deixe de sofrer alterações, permanecendo constante.

Desta forma, foi necessário começar por representar esta superfície e a forma de propagação do calor sobre a mesma utilizando um código sequencial. Após isto, pela utilização da *API* ***OpenMP***, que explora paralelismo em modelos de memória partilhada, foi possível realizar um conjunto de medidas que permitem tirar conclusões acerca do comportamento das duas versões implementadas, para diferentes dados de entrada (tamanho da superfície, numero máximo de iterações sobre a superfície, numero de cores/threads, características da memória da máquina, etc).

2. Dados de Input

Para que seja possível calcular a difusão de calor sobre uma superfície quadrada foi necessário começar por representar essa superfície. A representação adotada é a de uma matriz quadrada de [N,N], onde N representa o tamanho dos lados da mesma.

Foi necessário definir ainda uma escala numérica de temperaturas representativas da forma de calor, onde um numero que seja superior a outro é considerado estar num estado mais quente que o numero menor.

Após isto foi ainda necessário definir um estado inicial para as matrizes de modo a representar a forma como a superfície está a ser aquecida (fonte de calor) bem como a compreensão e representação da propagação do calor a partir da fonte de calor.

Desta forma, uma matriz quadrada, representativa da superfície, é constituída por um tamanho N e por um conjunto de valores numéricos em cada posição da matriz que representam o seu estado inicial (fonte de calor) e ao longo do tempo a forma como o calor é difundido da fonte para as restantes posições.

2.1. Estado Inicial da Matriz e Calculo da Difusão de Calor

O processo de difusão de calor vai então ser suportado por uma matriz quadrada de tamanho N onde cada posição da matriz possui um valor numérico representativo da temperatura dessa posição. Desta forma, foi necessário começar por determinar um intervalo de valores que representem a variação de temperatura. A escala adotada foi o intervalo de valores de 0 a 100, onde 0 representa o valor de temperatura mínimo (frio) e 100 representa o valor de temperatura máximo (quente):

100	99	98	...	2	1	0

Após determinar o intervalo de valores de temperatura foi necessário definir um estado inicial para a matriz. Como referido anteriormente, é considerado que a matriz/superfície é aquecida apenas no seu lado superior, ou seja, considerando uma matriz representada por [i,j], apenas a linha i=0 sofre um aquecimento inicial. Os valores escolhidos de temperatura para essa linha foi o maior valor possível de temperatura: 100. Nas restantes posições da matriz foi considerado que possuem valor 0 (menor valor de temperatura). Desta forma, o caso descrito aproxima-se do caso de uma superfície ser aquecida de forma constante, num dos seus lados, observando depois a forma como o calor é difundido ao longo da mesma.

Por forma a simplificar o processo, foi considerado ainda que os lados/bordas da matriz são imutáveis, ou seja, os valores nas linhas i=0 e i=N-1 e colunas j=0 e j=N-1, permanecem iguais aos valores iniciais.

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**

Table 1: Estado inicial da matriz. As bordas permanecem imutáveis.

Para efetuar o calculo da difusão do calor é necessário que cada posição da matriz (exceto as bordas) seja afetada consoante uma forma de propagação. A forma de propagação adotada para cada posição [i,j] da matriz é a média das posições imediatamente a cima, baixo, esquerda, direita e da própria posição:

$$matriz[i][j] = \frac{matriz[i-1][j] + matriz[i+1][j] + matriz[i][j-1] + matriz[i][j+1] + matriz[i][j]}{5} \quad (1)$$

Contudo, para que o calculo na posição sucessiva a uma posição já atualizada não seja influenciada, pelo novo resultado, calculado nessa posição, é necessário que cada posição seja calculada com os valores anteriores da matriz, ou seja, os valores que a matriz possui-a antes de ser iniciada uma nova iteração. Para tal é necessário a utilização de outra matriz que possui o resultado da ultima iteração da matriz inicial. Existem assim duas matrizes a que se dão o nome de "matriz_nova" (M_New) e "matriz_nantiga" (M_Old).

Cada iteração é efetuada na matriz M_New e sempre que termina, é copiado o estado da matriz M_New para a matriz M_Old que vai ser utilizada para os cálculos da próxima iteração de M_New:

$$M\_New[i][j] = \frac{M\_Old[i-1][j] + M\_Old[i+1][j] + M\_Old[i][j-1] + M\_Old[i][j+1] + M\_Old[i][j]}{5} \quad (2)$$

3. Implementação Algoritmo

Para avaliar o ganho de tempo de execução, do processo de difusão de calor desenvolvido, quando se aplica técnicas de paralelização, foi necessário desenvolver uma versão de código sequencial e outra versão de código paralelo, que se apresentam abaixo.

3.1. Algoritmo Sequencial

A implementação do algoritmo sequencial começa por alocar memoria para cada uma das matrizes, consoante o seu tamanho. Após a alocação das matrizes, a matriz M_New é iniciada da forma descrita anteriormente. Quando M_New possui os valores iniciais pode então ser iniciado o processo de calculo da difusão de calor.

O processo de difusão de calor é efetuado para N_MAX iterações, numero que é passado como argumento à função responsável por efetuar o processo. Assim, para cada iteração começa-se por efetuar a cópia de M_New para M_Old passando depois ao calculo da difusão de calor de acordo com a fórmula apresentada acima:

```
//Itera ate N_MAX iteracoes
for(int iteration = 0; iteration < N_MAX; iteration++) {

    //Guarda/copia a ultima solucao em M_Old
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            M_Old[i][j] = M_New[i][j];
        }
    }

    //Calcula os novos valores dos pontos interiores em M_New.
    for(int i = 1; i < N-1; i++) {
        for(int j = 1; j < N-1; j++) {
            M_New[i][j] = (M_Old[i-1][j] + M_Old[i+1][j] + M_Old[i][j-1] + M_Old[i][j↩
                +1] + M_Old[i][j])/5;
        }
    }
}
```

O código sequencial completo pode ser consultado no anexo D.

A escolha de efetuar primeiro a cópia de M_New para M_Old a cada iteração foi com o propósito de evitar efetuar esta cópia quando já não é necessário. Se primeiro fosse efetuado o calculo da difusão de calor e depois a cópia de M_New para M_Old, então na ultima iteração, esta cópia iria ser efetuada sem necessidade uma vez que não são efetuadas mais iterações. Analisando a complexidade das linhas de código responsáveis pela cópia a mesma é sempre N^2, que para tamanho de matrizes grandes (valores de N), consome uma quantidade de tempo elevado.

Para além da otimização referida acima e atendendo ao facto de o código ser de complexidade baixa e simples, o que limita a quantidade de otimizações possíveis de implementar, foi procurado de diversas formas tirar mais partido da hierarquia de memória, como por exemplo, percorrer a matriz na diagonal e reutilizar acessos à memória (por exemplo, matriz[i][j] acede às posições [i+1][j] e [i][j+1] que são também

acedidas, mais tarde, na posição matriz[i+1][j+1]), contudo, o código destas versões revelou-se extenso e de difícil implementação. Apesar disto, a solução apresentada consegue tirar partido da localidade espacial, ou seja, o acesso aos dados é efetuado, de uma forma geral, pela mesma ordem que estão armazenados, o que aumenta a probabilidade de na execução seguinte do ciclo *for* interior os dados necessários já estarem na *cache*, diminuindo assim tempos de acesso à memória *RAM*.

3.2. Algoritmo Paralelo

Em comparação com o código sequencial, o código paralelo desenvolvido consiste basicamente na adição de diretivas **OpenMP**.

Após várias tentativas e testes de várias diretivas disponibilizadas pelo **OpenMP** o resultado final foi:

```
int i, j;

//Itera ate N_MAX iteracoes
for(int iteration = 0; iteration < N_MAX; iteration++) {
    //Criacao de um conjunto de threads
    #pragma omp parallel shared(M_New, M_Old) private(i, j)
    {
        //Guarda a ultima solucao em M_Old
        #pragma omp for
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
                M_Old[i][j] = M_New[i][j];
            }
        }

        //Calcula os novos valores dos pontos interiores para M_New.
        #pragma omp for
        for(i = 1; i < N-1; i++) {
            for(j = 1; j < N-1; j++) {
                M_New[i][j] = (M_Old[i-1][j] + M_Old[i+1][j] + M_Old[i][j-1] + M_Old[↩
                    i][j+1] + M_Old[i][j])/5;
            }
        }
    }
}
```

O código paralelo completo pode ser consultado no anexo E.

Estão a ser criadas um conjunto de threads a partir da diretiva *#pragma omp parallel*. Para além da criação das threads, esta diretiva tem ainda mais informação associada: *shared(M_New, M_Old) e private(i,j)*. A primeira indica que ambas as matrizes são partilhadas pelas várias threads, isto é necessário uma vez que todas as threads devem ter acesso às matrizes para leitura e escrita. A segunda indica que cada thread vai possuir o seu próprio valor das variáveis i, j, que são responsáveis por fazer o controlo dos ciclos.

Existem ainda mais duas diretivas, *#pragma omp for*, que são colocadas antes do inicio dos ciclos *for* externos. Isto leva à paralelização do ciclo *for* externo, que se traduz na atribuição dos cálculos efetuados em cada linha para cada thread, ou seja, cada thread é responsável por cada linha da matriz.

Em termos de problemas comuns de controlo de concorrência não foram encontrados nenhuns. Os códigos paralelos responsáveis pela cópia das matrizes e pelo calculo da difusão de calor em cada posição estão separados o que não origina problemas de concorrência. No código responsável pelo calculo da difusão também não vão ocorrer problemas de concorrência uma vez que, embora cada linha aceda a elementos das linhas imediatamente acima e abaixo, esses elementos estão apenas a ser lidos de M_Old, não havendo problemas de leitura e escrita concorrentes nem inconsistência de valores, de facto, M_Old no processo de calculo da difusão nunca é escrito mas apenas lido.

4. Testes: SeARCH - compute-641-8

Nesta secção vai ser descrito todo o processo e tomada de decisão dos testes realizados. Todos os testes foram efetuados no **SeARCH - compute-641-8** cujas especificações relevantes podem ser encontradas no anexo A.

4.1. Compilação dos Algoritmos

O compilador da *GNU* para C (*gcc*) inclui alguns níveis de otimização que quando utilizados é aplicado um conjunto de técnicas automáticas que otimizam o código sem qualquer input do utilizador. Para tal são utilizadas flags de compilação que especificam a profundidade da otimização. O nível de otimização utilizado foi definido pela flag "-O3". A versão do *gcc* utilizada para compilação no *SeARCH* foi **gcc 4.9.0**.

```
$ gcc -O3 -Wall -Wextra -std=c99 -fopenmp -o heatDiff_<Seq><Par> ↩
    heatDiff_<Seq><Par>.c
```

Para que os resultados dos tempos de execução, de ambas as versões do código, fossem fidedignos utilizou-se a mesma diretiva de compilação tanto para a versão sequencial como paralela.

4.2. Tamanhos de Input

Foi necessário tomar decisões acerca do tamanho de input. Este tamanho inclui o tamanho das matrizes e ainda o numero de iterações que vão ser efetuadas até que o processo de difusão de calor esteja concluído.

O tamanho de input a utilizar deve ser relativamente razoável por forma a que seja possível maximizar o numero de medições possíveis de efetuar mas que ao mesmo tempo permita que se tenha algum esforço de computação para que seja possível avaliar o comportamento/penalização de chamadas à memória. Neste sentido, foi escolhido um tamanho para as matrizes que ultrapasse o tamanho da memória cache dos processadores, levando à existência de misses nos vários níveis da cache bem como consequentes chamadas à memória RAM. Assim, foram efetuados testes para tamanhos de matrizes de 1024x1024, 2048x2048, 4096x4096 e ainda 8192x8192.

Em relação ao numero de iterações que vão ser efetuadas até que o processo de difusão de calor termine foram escolhidas 1000, 2000, 4000 e 8000 iterações. Foi escolhido este conjunto de valores uma vez que é necessário um tempo razoável até que o processo de difusão estabilize, ou seja, até que por cada iteração os valores deixem de sofrer alterações relevantes.

Para a versão paralela do código foi necessário definir ainda o numero de threads a utilizar por forma a compreender o ganho, em tempo, da utilização de mais threads. Assim, o código paralelo vai ser executado para 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22 e 24 threads.

4.3. Metodologia e Condições de Medição

Atendendo aos tamanhos de input descritos acima, foram então realizadas medidas de tempos de execução e ganhos, retirados para cada tamanho de matriz com cada tamanho de iterações máximas para cada numero de threads possíveis.

A métrica utilizada para medir o desempenho do algoritmo sequencial e paralelo foi o **tempo de execução**. É com base neste tempo que se vão calcular os ganhos obtidos de cada execução paralela em comparação com a execução sequencial. Alguma informação adicional, relevante, para descrever as condições de teste:

- Foi utilizado o contador de tempo da biblioteca *OpenMP*: **omp_get_wtime()**;
- A resolução de tempos utilizada é o **milissegundo (ms)**;
- As áreas de código delimitadas para a medição dos tempos não incluem qualquer tipo de I/O (ex: *printf's*);

- Foram recolhidas 10 medições para cada teste;
- Para cada teste foi calculada a **mediana** dos valores para atenuar as maiores oscilações de valores.

As recolhas dos tempos de execução, para cada teste efetuado, podem ser consultadas no anexo B.

5. Resultados: SeARCH - compute-641-8

Por forma a fazer uma análise mais simples e agregada de todos os valores recolhidos foram criados quatro gráficos. Cada gráfico representa os SpeedUps obtidos para cada numero de iterações (N_MAX) de cada matriz para cada numero possível de threads. Estes gráficos podem ser consultados no anexo C.

5.1. Discussão

Observando os gráficos, é possível constatar que os melhores ganhos aconteceram para o numero máximo de iterações igual a 2000. Nesse mesmo gráfico é possível constatar um ganho acentuado para a matriz 2048x2048 que é possível observar também nos restantes gráficos. De facto, a matriz 2048x2048 é a que mais ganhos possui em qualquer numero máximo de iterações. Tendo em conta que existe uma matriz de 1024x1024 (menor tamanho da matriz), era expectável que esta fosse a que melhores ganhos possui, o que não acontece. A única explicação que o grupo chegou para justificar isto é que para matrizes de pequena dimensão é gasto mais tempo na criação da *team* de threads e escalonamento de trabalho entre elas do que na execução do código do programa, o que inevitavelmente se traduz na quantidade de ganho obtido.

Atendendo a todos os gráficos é possível observar que as matrizes de 4096x4096 e 8192x8192 possuem um comportamento semelhante independentemente do numero máximo de iterações. As linhas de tendência do ganho das duas andam sempre próximas e de forma aproximadamente paralela. Isto acontece possivelmente quando o tamanho das matrizes começa a ser considerado grande, o que leva a que a quantidade de largura de banda e memória cache disponíveis não sejam suficientes para suportar o volume de dados necessários.

Fazendo agora uma análise por numero de threads utilizadas pode-se inferir que para:

- **0 a 4 threads**
  Existe uma subida acentuada do ganho para todos os tamanhos das matrizes e números máximos de iterações. No melhor caso, seria expectável de se obter um ganho máximo proporcional ao numero de threads utilizadas, o que não acontece.

  Existe pelo menos uma matriz em cada gráfico que, para duas threads, possui um ganho superior a dois. Isto ocorre devido à quantidade de memoria cache disponível. Quando o código passa da versão sequencial para a versão paralela, passa a existir uma quantidade de memória cache igual à soma da memória cache disponível em cada core, o que leva a uma subida acentuada de ganhos e possivelmente ganhos superiores ao que é teoricamente expectável, devido a menores chamadas à memoria RAM.

  Para quatro (e mais threads) já não ocorre o descrito acima, ou seja, o ganho começa a ser menor do que o numero de threads utilizadas. Apesar de à medida que se vai utilizando mais threads, a quantidade de cache disponível aumenta (apenas até ao numero de cores físicos), a quantidade de dados a serem transportados pelos barramentos são elevados o que leva a que a largura de banda disponível deixe de ser suficiente, ocorrendo bottlenecks na transmissão dos dados.

  Ainda neste intervalo de threads, no gráfico de N_MAX iterações igual a 4000, ocorre um ganho praticamente linear para a matriz de tamanho 2048x2048. Isto vem apoiar os comentários anteriores na medida em que para que tal ocorra, é necessário que todos os dados iniciais do calculo efetuado se encontrem na memória cache.

- **4 a 14 threads**
  Neste intervalo de utilização de threads, de uma forma geral para todos os valores de N_MAX, continuam a existir speedups, mas pouco acentuados (exceto para a matriz 2048x2048). Aqui, os ganhos conseguidos deixam de ser acentuados devido, principalmente, à quantidade de vezes que é necessário buscar dados à memória RAM.

- **14 a 24 threads**
  Em todos os gráficos é possível observar que, nesta gama de threads, os ganhos começam a tender para constantes, havendo até perdas de ganho em alguns tamanhos de matrizes. Isto deve-se sobretudo por entre as 16 e 32 threads/cores serem virtuais (hyper-threading), ou seja, não possuem memória cache própria (especificações da máquina de teste utilizada no anexo A). Um outro motivo de influencia para o tempo constante dos ganhos neste intervalo de threads é a falta de largura de banda suficiente para transmissão dos dados (bottleneck no acesso aos dados).

É ainda possível concluir pela observação dos ganhos que o numero máximo de iterações (N_MAX) não influencia de forma significativa os ganhos possíveis de obter. Isto é especialmente observável para as matrizes de tamanho 4096x4096 e 8192x8192 (maiores matrizes) que independentemente do numero máximo de iteração que são efetuadas, o ganho máximo obtido mantém-se por volta de 3.5. Como tal é possível concluir que o tamanho de matriz escolhido tem mais influência nos possíveis ganhos de speedup do que o numero de iterações que vão ser efetuadas.

Para concluir, de uma forma geral foram obtidos bons speedups, ou seja, foi conseguida uma boa escalonagem do código. Para inputs pequenos como a matriz de 1024x1024 não se obtiveram muitos ganhos devido ao tempo necessário de escalonamento de trabalho entre threads. Para inputs de tamanho médio, matriz de 2048x2048, conseguiu-se os melhores ganhos uma vez que possui um tamanho propício para a quantidade de memória e threads utilizadas. Por fim, para matrizes de tamanhos maiores (4096x4096 e 8192x8192) os ganhos passam a ser constantes devido aos problemas inerentes de acesso à memória bem como falta de largura de banda.

6. Conclusão

A utilização de paradigmas de computação paralela é um método muito eficaz para conseguir paralelizar um código sequencial. Contudo, a sua aplicabilidade nem sempre é trivial para todos os casos, como a existência de *data races* que é, geralmente, um problema de difícil resolução.

Os resultados expectáveis/desejáveis muito raramente são obtidos devido a várias limitações de hardware (roofline, largura de banda, tamanho da memória cache, etc) bem como à implementação do código paralelo. De facto, o programador tem um papel essencial na obtenção de ganhos uma vez que é responsável por desenvolver código com o conjunto de diretivas que tirem o melhor proveito possível para o problema a tratar.

Apesar destas possíveis limitações, foi conseguido de uma forma geral um bom ganho em speedup do código paralelo em relação ao código sequencial, especialmente para a matriz de tamanho 2048x2048 e ainda foi possível constatar que o numero máximo de iterações efetuadas para cada matriz não influencia significativamente o ganho obtido (mas influencia o tempo de execução sequencial e paralelo), estando a maior limitação no tamanho das matrizes.

Como trabalho futuro seria interessante desenvolver uma versão ainda mais otimizada do código sequencial e paralelo por forma a avaliar que tipo de ganhos seriam possíveis de obter.

A. Máquina de Teste

Hardware	SeARCH (compute-641-8)
Processador	Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
Cache	20MB
Memória	64GB
#Cores	16 físicos + 16 virtuais
Sistema Operativo	Rocks 6.1 (Emerald Boa)

B. Recolha de Tempos SeARCH - compute-641-8

Tam. Matriz	N_MAX	#Threads	Mediana Tempos	Ganho (Sequencial vs Paralelo)
1024*1024	1000	0 (Seq.)	1551,904	0
		2	1003,346	1,546728646
		4	629,359	2,465848586
		6	481,186	3,225164489
		8	416,713	3,724155474
		10	391,122	3,967825896
		12	362,655	4,27928472
		14	366,842	4,230442534
		16	357,173	4,344964485
		18	364,761	4,254577655
		20	371,072	4,182218006
		22	358,602	4,327650153
		24	358,41	4,329968472
	2000	0 (Seq.)	2651,03	0
		2	1888,91	1,403470785
		4	1266,701	2,092861693
		6	1109,219	2,389996926
		8	1032,906	2,566574306
		10	990,457	2,676572532
		12	936,333	2,831289723
		14	894,026	2,965271704
		16	901,947	2,939230354
		18	895,663	2,959852087
		20	909,564	2,914616234
		22	884,301	2,997881943
		24	883,423	3,000861422
	4000	0 (Seq.)	5705,429	0
		2	4903,473	1,163548571
		4	3203,942	1,780752898
		6	2994,84	1,905086415
		8	2606,455	2,188961252
		10	2428,118	2,349733003
		12	2333,177	2,445347695
		14	2279,152	2,503312197
		16	2327,634	2,451171017

		18	2293,144	2,488037821
		20	2225,301	2,563890907
		22	2126,374	2,683172857
		24	2139,179	2,667111541
	8000	0 (Seq.)	12614,995	0
		2	10368,873	1,216621614
		4	8418,503	1,49848435
		6	7002,28	1,801555351
		8	6316,452	1,997164706
		10	6162,336	2,047112491
		12	5795,737	2,176598938
		14	5685,639	2,218747092
		16	5536,685	2,278438271
		18	5534,618	2,279289194
		20	5279,782	2,389302248
		22	5275,847	2,391084313
		24	5065,876	2,490190245
2048*2048	1000	0 (Seq.)	6705,539	0
		2	2900,253	2,312053121
		4	1792,309	3,741285124
		6	1432,133	4,682204097
		8	1231,881	5,443333406
		10	1120,853	5,982532054
		12	1049,455	6,389544097
		14	999,976	6,705699937
		16	1066,039	6,290144169
		18	1098,814	6,102524176
		20	1056,37	6,347718129
		22	1082,112	6,196714388
		24	943,29	7,108671776
	2000	0 (Seq.)	17611,092	0
		2	7476,855	2,355414409
		4	4652,284	3,785472254
		6	3611,19	4,876811245
		8	3213,054	5,48110676
		10	2699,612	6,523564127
		12	2480,882	7,098722148
		14	2325,437	7,573239782
		16	2340,91	7,52318201
		18	2457,636	7,165866711
		20	2516,94	6,997024959
		22	2380,993	7,396532455
		24	2336,661	7,536862215
	4000	0 (Seq.)	35847,241	0
		2	15909,037	2,253262784
		4	8821,048	4,063830171
		6	7752,67	4,623857458
		8	6739,509	5,318969231
		10	6316,998	5,674727299
		12	5908,101	6,067472611
		14	5747,976	6,23649803

		16	5753,6	6,230402009
		18	6148,554	5,830190481
		20	5794,621	6,186296049
		22	5819,135	6,160235327
		24	5678,253	6,313075694
	8000	0 (Seq.)	76970,775	0
		2	35507,039	2,167761018
		4	21156,347	3,638188341
		6	17985,053	4,279707989
		8	16386,459	4,697218295
		10	15673,827	4,910783754
		12	14729,818	5,225507539
		14	14049,191	5,478662437
		16	14508,239	5,305314794
		18	14260,046	5,397652644
		20	13939,447	5,521795449
		22	15285,188	5,035644638
		24	13890,995	5,541055554
4096*4096	1000	0 (Seq.)	27495,882	0
		2	17997,604	1,527752361
		4	11277,254	2,438171739
		6	9700,616	2,834447008
		8	9655,98	2,847549601
		10	9180,728	2,994956609
		12	9373,269	2,93343571
		14	9576,276	2,871249951
		16	9841,561	2,793853739
		18	9853,24	2,790542197
		20	9587,452	2,867902963
		22	9557,819	2,876794591
		24	9535,045	2,883665678
	2000	0 (Seq.)	67101,656	0
		2	33621,163	1,995816028
		4	22113,856	3,034371572
		6	19980,774	3,358311145
		8	19482,919	3,444127443
		10	19109,485	3,511431941
		12	19218,781	3,491462648
		14	18364,945	3,653790196
		16	19825,1	3,384681843
		18	19706,526	3,405047445
		20	19584,417	3,426277943
		22	19471,976	3,446062998
		24	18790,281	3,571083157
	4000	0 (Seq.)	148321,93	0
		2	72081,295	2,057703458
		4	47811,072	3,102250667
		6	42550,5	3,485785831
		8	39967,809	3,711034798
		10	39362,201	3,768131005
		12	39708,895	3,735231867

		14	39213,516	3,782418542
		16	42872,289	3,459622368
		18	39961,155	3,711652729
		20	40665,58	3,647358036
		22	41136,98	3,605561954
		24	40672,839	3,646707081
	8000	0 (Seq.)	294494,758	0
		2	151651,07	1,941923377
		4	103850,835	2,835747618
		6	90306,575	3,261055554
		8	79928,264	3,684488356
		10	91650,129	3,213249793
		12	83034,474	3,546656513
		14	80505,417	3,658073816
		16	90230,57	3,263802478
		18	85844,439	3,430563021
		20	84826,697	3,471722564
		22	79838,995	3,688608029
		24	83260,806	3,53701546
8192*8192	1000	0 (Seq.)	99155,322	0
		2	68253,6	1,452748602
		4	41529,055	2,387613251
		6	36127,681	2,744580312
		8	35376,71	2,802841813
		10	34451,845	2,878084526
		12	34598,737	2,865865364
		14	35306,38	2,80842505
		16	38689,116	2,562873807
		18	39939,683	2,482626665
		20	37291,254	2,658943086
		22	36976,416	2,681582823
		24	37585,827	2,638104039
	2000	0 (Seq.)	275572,822	0
		2	138810,715	1,985241716
		4	84026,68	3,279587174
		6	72277,577	3,812701441
		8	72291,137	3,811986274
		10	74157,751	3,71603532
		12	71858,265	3,834949564
		14	72245,94	3,81437105
		16	80389,576	3,427967104
		18	79003,83	3,488094463
		20	77356,277	3,562384756
		22	76686,094	3,593517516
		24	75951,218	3,628287067
	4000	0 (Seq.)	622577,989	0
		2	285748,467	2,178762306
		4	168989,366	3,684125243
		6	148801,237	4,18395708
		8	144504,389	4,308367333
		10	135245,044	4,60333311

		12	146249,965	4,256944533
		14	145693,463	4,273204687
		16	152960,716	4,070182236
		18	151938,656	4,097561512
		20	152602,797	4,079728558
		22	148817,017	4,183513428
		24	146936,903	4,237043086
	8000	0 (Seq.)	1108701,053	0
		2	531168,379	2,087287378
		4	355472,716	3,118948384
		6	300444,05	3,690208054
		8	240299,33	4,613833309
		10	321978,842	3,443397231
		12	281833,817	3,93388226
		14	293020,581	3,783696863
		16	328870,617	3,371237793
		18	327745,207	3,382813934
		20	319480,117	3,470328806
		22	329611,19	3,363663269
		24	312137,647	3,551961975

C. Comparação SpeedUp's SeARCH - compute-641-8

C.1. N_MAX Iterações: 1000

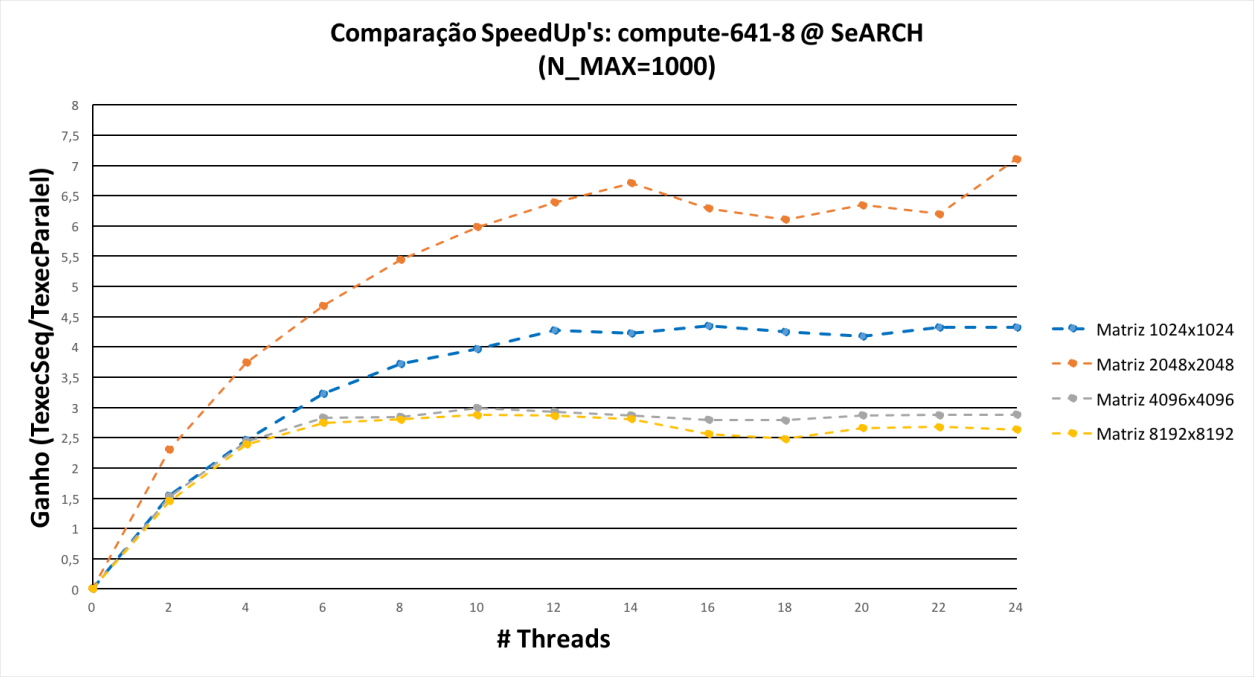


Figure 1: Comparação dos SpeedUp's obtidos para cada tamanho de matriz, com 1000 N_MAX iterações

C.2. N_MAX Iterações: 2000

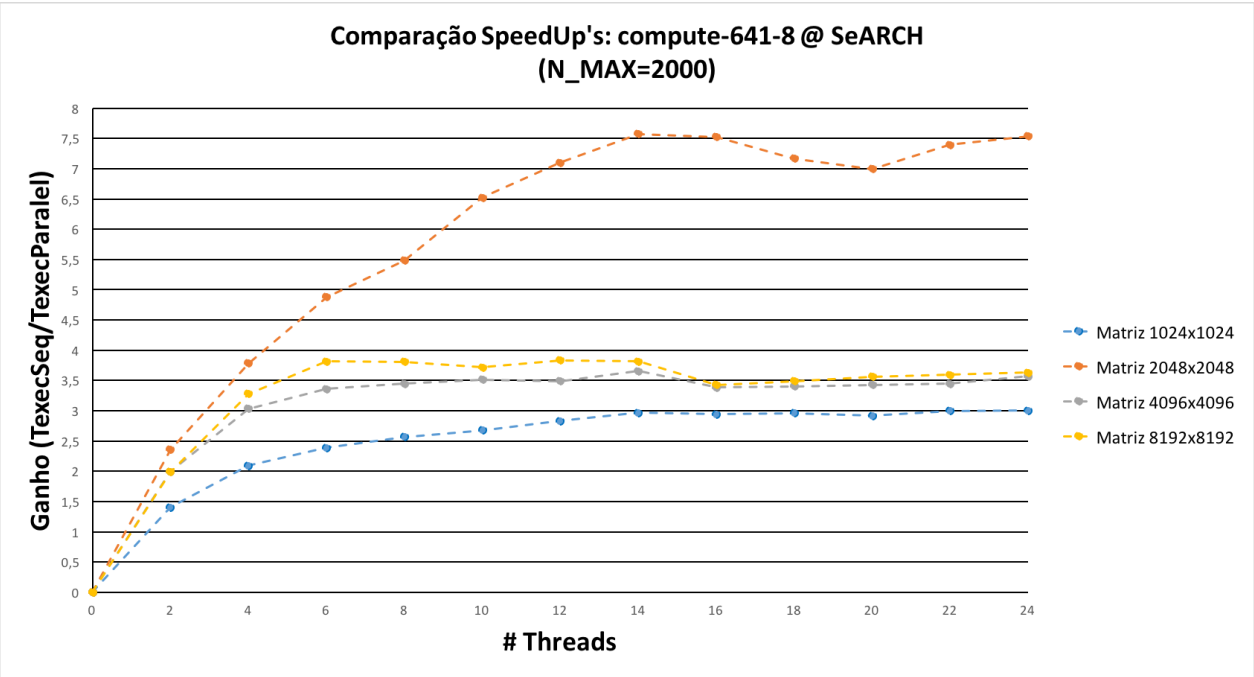


Figure 2: Comparação dos SpeedUp's obtidos para cada tamanho de matriz, com N_MAX iterações igual a 2000

C.3. N_MAX Iterações: 4000

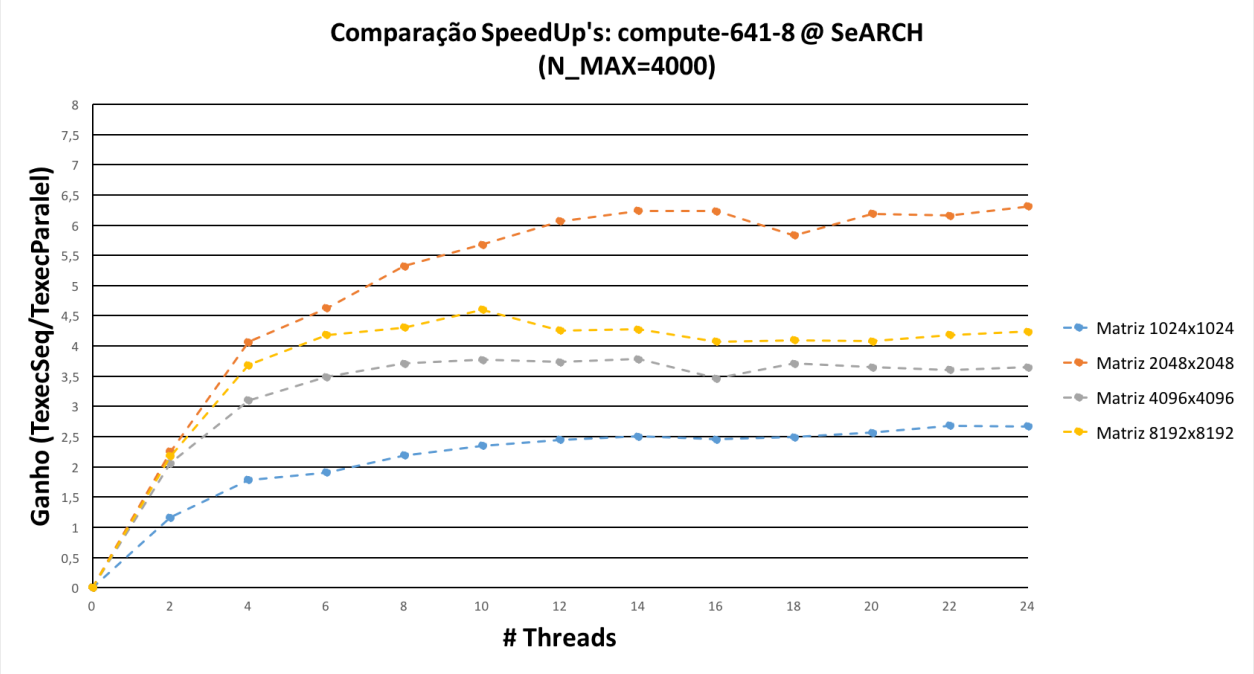


Figure 3: Comparação dos SpeedUp's obtidos para cada tamanho de matriz, com N_MAX iterações igual a 4000

C.4. N_MAX Iterações: 8000

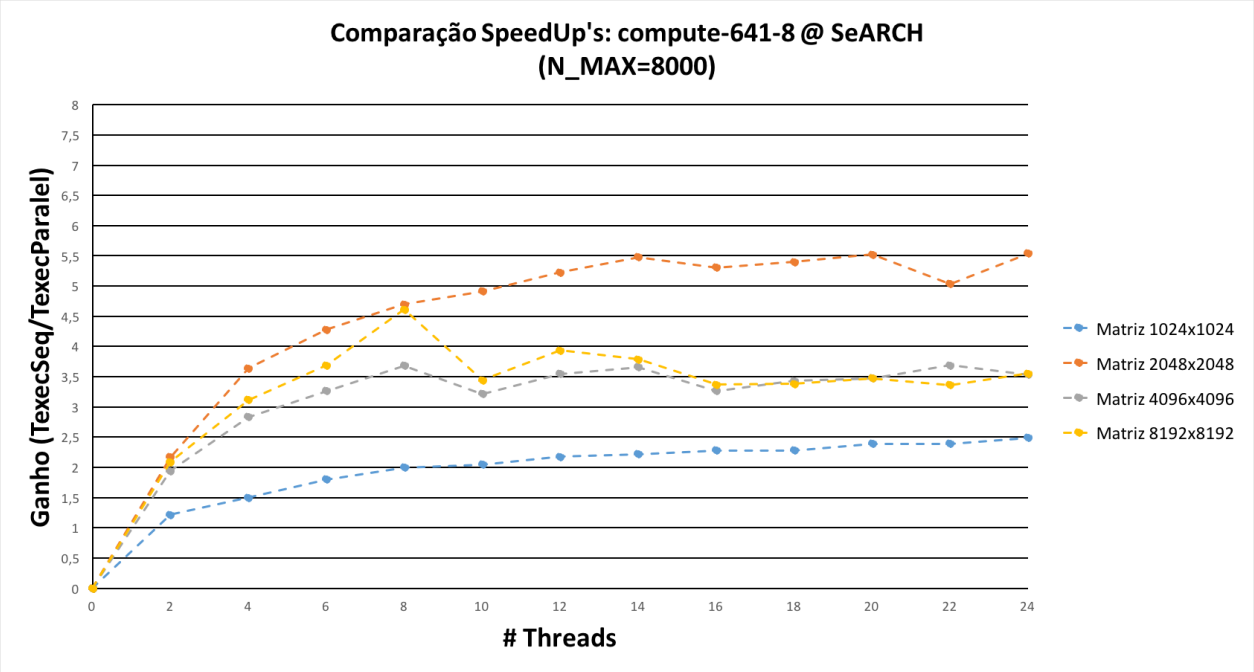


Figure 4: Comparação dos SpeedUp's obtidos para cada tamanho de matriz, com N_MAX iterações igual a 8000

D. Código Sequencial

```
//*********************************************
//*********************************************
//
//      VERSAO SEQUENCIAL - Heat Diffusion
//
//*********************************************
//*********************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>


/*** Inicia a matriz. Primeira linha a 100, restantes a 0 ***/
void initiateMatrix_new(float **M, int N) {
    //Preenche a primeira linha a 100
    for(int i = 0; i < N; i++) M[0][i] = (float) 100;

    //Preenche o restante a 0
    for(int i = 1; i < N; i++) {
        for(int j = 0; j < N; j++) {
            M[i][j] = (float) 0;
        }
    }
}

/*** Efetua as iteracoes ate N_MAX ***/
void iterate(float **M_New, float **M_Old, int N, int N_MAX) {
    for(int iteration = 0; iteration < N_MAX; iteration++) {

        //Guarda a ultima solucao em M_Old
        for(int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++) {
                M_Old[i][j] = M_New[i][j];
            }
        }

        //Calcula os novos valores dos pontos interiores para M_New.
        for(int i = 1; i < N-1; i++) {
            for(int j = 1; j < N-1; j++) {
                M_New[i][j] = (M_Old[i-1][j] + M_Old[i+1][j] + M_Old[i][j-1] + M_Old[↩
                    i][j+1] + M_Old[i][j])/5;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    float **heatPlate_new;
    float **heatPlate_old;
    int matrix_size;
```

```
    int n_max_iterations;
    double startTime, finalTime;
    FILE *f = NULL;

    matrix_size = atoi(argv[1]);
    n_max_iterations = atoi(argv[2]);

    /*** Alocacao das matrizes ***/
    heatPlate_new = (float **)malloc(matrix_size * sizeof(*heatPlate_new));
    heatPlate_old = (float **)malloc(matrix_size * sizeof(*heatPlate_old));
    //Alocacao das "linhas"
    for(int i = 0; i < matrix_size; i++) {
        heatPlate_new[i] = (float *)malloc(matrix_size * sizeof(**heatPlate_new));
        heatPlate_old[i] = (float *)malloc(matrix_size * sizeof(**heatPlate_old));
    }

    /*** Iniciar valores matriz_new ***/
    initiateMatrix_new(heatPlate_new, matrix_size);

    /*** Efetua N_MAX iteracoes. Inicia a contagem do tempo de execucao. ***/
    startTime = omp_get_wtime();
    iterate(heatPlate_new, heatPlate_old, matrix_size, n_max_iterations);
    finalTime = omp_get_wtime();

    printf("Time seq: %.3f ms\n", (finalTime - startTime)*1000);

    /*** Liberta memoria utilizada ***/
    free(heatPlate_new);
    free(heatPlate_old);

    return 1;
}
```

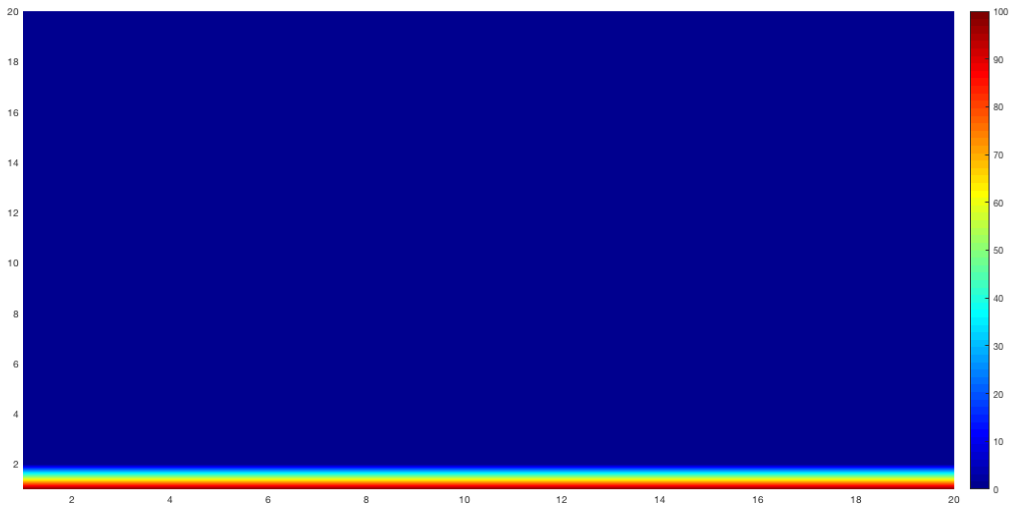
E. Código Paralelizado

```
//***********************************************
//***********************************************
//
//      VERSAO PARALELA V1.0 - Heat Diffusion
//
//***********************************************
//***********************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>



/*** Inicia a matriz. Primeira linha a 100, restantes a 0 ***/
void initiateMatrix_new(float **M, int N) {
    //Preenche a primeira linha a 100
    for(int i = 0; i < N; i++) M[0][i] = (float) 100;

    //Preenche o restante a 0
    for(int i = 1; i < N; i++) {
        for(int j = 0; j < N; j++) {
            M[i][j] = (float) 0;
        }
    }
}

/*** Efetua as iteracoes ate N_MAX ***/
void iterate(float **M_New, float **M_Old, int N, int N_MAX) {
    int i, j;

    //Itera ate N_MAX iteracoes
    for(int iteration = 0; iteration < N_MAX; iteration++) {
        //Criacao de um conjunto de threads
        #pragma omp parallel shared(M_New, M_Old) private(i, j)
        {
            //Guarda a ultima solucao em M_Old
            #pragma omp for
            for(i = 0; i < N; i++) {
                for(j = 0; j < N; j++) {
                    M_Old[i][j] = M_New[i][j];
                }
            }

            //Calcula os novos valores dos pontos interiores para M_New.
            #pragma omp for
            for(i = 1; i < N-1; i++) {
                for(j = 1; j < N-1; j++) {
                    M_New[i][j] = (M_Old[i-1][j] + M_Old[i+1][j] + M_Old[i][j-1] + ↩
                        M_Old[i][j+1] + M_Old[i][j])/5;
                }
```

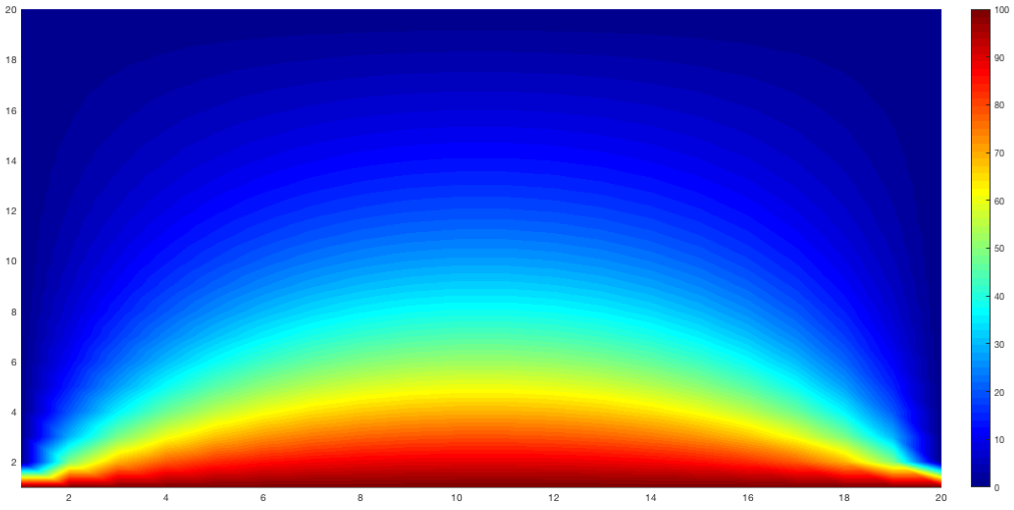
```
            }
        }
    }
}

int main(int argc, char *argv[]) {
    float **heatPlate_new;
    float **heatPlate_old;
    int matrix_size;
    int n_max_iterations;
    double startTime, finalTime;
    FILE *f = NULL;

    matrix_size = atoi(argv[1]);
    n_max_iterations = atoi(argv[2]);

    /*** Alocacao das matrizes ***/
    heatPlate_new = (float **)malloc(matrix_size * sizeof(*heatPlate_new));
    heatPlate_old = (float **)malloc(matrix_size * sizeof(*heatPlate_old));
    //Alocacao das "linhas"
    for(int i = 0; i < matrix_size; i++) {
        heatPlate_new[i] = (float *)malloc(matrix_size * sizeof(**heatPlate_new));
        heatPlate_old[i] = (float *)malloc(matrix_size * sizeof(**heatPlate_old));
    }

    /*** Iniciar valores matriz_new ***/
    initiateMatrix_new(heatPlate_new, matrix_size);

    /*** Numero de threads a utilizar ***/
    omp_set_num_threads(atoi(argv[3]));

    /*** Efetua N_MAX iteracoes. Inicia a contagem do tempo de execucao. ***/
    startTime = omp_get_wtime();
    iterate(heatPlate_new, heatPlate_old, matrix_size, n_max_iterations);
    finalTime = omp_get_wtime();

    printf("Time par: %.3f ms\n", (finalTime - startTime)*1000);

    /*** Liberta memoria utilizada ***/
    free(heatPlate_new);
    free(heatPlate_old);

    return 1;
}
```

F.
Representação Gráfica do Resultado

Apresenta-se abaixo uma representação gráfica da matriz inicial e da matriz final do processo de difusão de calor efetuado 8000 vezes, para uma matriz de tamanho 20x20 (equivalente para o código sequencial e paralelo):



(a) Matriz inicial 20x20



(b) Matriz final 20x20, para 8000 N_MAX iterações

Figure 5: Representação gráfica do processo de difusão de calor para uma matriz de 20x20 com 8000 N_MAX iterações