



Universidade Federal Fluminense
Instituto de Ciências Exatas
Departamento de Física
Física Computacional

UMA SIMULAÇÃO DE MONTE CARLO DO MODELO DE ISING EM 2D

Renato Maciel Félix

Volta Redonda

2021

Sumário

1	INTRODUÇÃO	4
2	PROBLEMA	5
2.1	Modelo de Ising 2D	5
2.2	Monte Carlo e Metropolis	5
3	FLUXOGRAMA	6
4	PROGRAMA	7
5	PROFILE	11
6	OTIMIZAÇÃO	12
6.1	Otimização de escrita	12
6.2	Otimização de software	12
7	PARALELISMO	15
7.1	Paralelismo nas temperaturas	15
7.2	Paralelismo no domínio	15
8	MPI	16
8.1	Implementação do paralelismo nas temperaturas	16
8.2	Implementação do paralelismo no domínio	16
8.3	Comparações das implementações	17
9	LNCC	20
10	CONCLUSÃO	21
	BIBLIOGRAPHY	22

Resumo

A modelagem computacional nos permite grandes análises e descobertas sobre fenômenos da natureza, análise de mercado financeiro, entre outras áreas do conhecimento humano. Muitas dessas modelagem nasceram com objetivos diferentes dos quais são utilizados hoje em dia, como o Modelo de Monte Carlo, que foi criado durante a Segunda Guerra, para melhores cálculos necessários naquela época. Este modelo pode ser aplicado em modelos físicos que nos ajudam a entender melhor algumas forças da natureza. Um exemplo é o modelo de Ising 2D, que nos ajuda a entender o ferromagnetismo em uma rede de spins. Neste trabalho implementamos um programa que realiza uma simulação do modelo de Ising 2D, utilizando o modelo de Monte Carlo e o algoritmo de Metropolis para as tomadas de decisões. Buscamos também uma melhoramento do programa jeito, buscando otimizações de escrita e no nível de software também. Por fim, analisamos seus principais gargalos e criamos estratégias de paralelismo, de modo a reduzir o tempo de execução. Com essas estratégias, implementamos o paralelismo utilizando a api MPI, onde obtivemos uma redução muito singnificativa no tempo de execução.

Palavras-chave: Ising, Monte Carlo, Metropolis.

Abstract

Computational modeling allows us to make great analyzes and discoveries about the market of natural phenomena, financial analysis, among other analyzes of human knowledge. The modeling of that date was born with different objectives than those used today in Monte Carlo, which was during the Second World War, for the best that were created at that time. This model can be applied to physical models that help us better understand the force of nature. One example is the 2D Ising model, which helps us understand ferromagnetism in a spin network. In this work we implement a program that performs a simulation of the 2D Ising model, using the Monte Carlo model and the Metropolis method for decision making. We are also looking for an improvement of the Modulo program, looking for optimizations in writing and at the software level as well. Finally, we analyze its main bottlenecks and create parallelism strategies in order to reduce execution time. With strategies, we implemented parallelism using the MPI api, where we achieved a very significant reduction in execution time.

Keywords: Ising, Monte Carlo, Metropolis.

1 Introdução

Modelo de Ising é um modelo simples mas muito importante para entender os fenômenos do magnetismo. Ele foi proposto por Wilhelm Lenz (1888-1957) e Ernest Ising (1900-1998) em 1920 ([LÍBERO, 2000](#)), que era seu aluno de doutorado. Eles buscavam estudar o ferromagnetismo, mas encontraram problemas com os resultados encontrados, pois a teoria de magnetismo que tinham na época, apontavam para outros resultados.

Implementaremos este problema com o algoritmo de Monte Carlo, que foi inventado por John von Neumann e Stanislaw Ulam durante a Segunda Guerra Mundial para melhorar a tomada de decisão em condições incertas ([EDUCATION, 2020](#)). Usaremos uma das mais famosas implementações do algoritmo de Monte Carlo, que é o algoritmo de Metropolis, que foi apresentado por Nicolas Metropolis e seus colaboradores ([SATO, 2021](#)).

Neste trabalho usaremos o modelo em uma rede quadrada ($N \times N$), tendo assim, $2N$ sítios. As variáveis que definirão o andamento da simulação para uma dada temperatura T em um microestado α , são: magnetização, energia e o calor específico, como mostrado na Seção 2. Esquematizaremos a implementação deste programa apresentando um fluxograma (Seção 3) e faremos o programa na linguagem C (Seção 4), realizando uma análise (Seção 5) e uma otimização a nível de escrita e software (Seção 6). Apresentaremos propostas de paralelismo (Seção 7), que será implementada com a API OpenMPI (Seção 8). Por fim, utilizaremos as máquinas disponibilizadas do Sdumont (Seção 9)

2 Problema

2.1 Modelo de Ising 2D

Os spins assumem valores os seguintes valores: $s = +1, -1$, portanto, a magnetização assumirá valores inteiros e é definida pela soma dos spins na malha:

$$M_\alpha = \sum_i s_i. \quad (2.1)$$

A energia será dada pela Eq.(2.2), onde J é a constante de troca, H é um campo magnético externo, μ o momento magnético associado ao spin. Somente foram consideradas as interações entre o spin e seus vizinhos mais próximos.

$$E = -J \sum_{\langle i,j \rangle} s_i s_j - \mu H \sum_i s_i. \quad (2.2)$$

Partindo do teorema de flutuação-dissipação da mecânica estatística, a relação entre o calor específico e a variação de energia é dada por:

$$C = \frac{(\Delta E)^2}{k_B T^2} \quad (2.3)$$

2.2 Monte Carlo e Metropolis

O modelo de Monte Carlo consiste em uma criação de resultados possíveis, utilizando uma distribuição de probabilidade, estimando uma determinada variável que possua uma incerteza probabilística. Este modelo utiliza de números aleatórios para calcular estes resultados possíveis, repetindo a conta por um determinado número de vezes, chamado de Número de Monte Carlo nMC . No algoritmo de Metropolis, é realizada uma alteração em uma variável do problema, no nosso caso do modelo de Ising, um "flip" em um spin da malha. Se essa mudança diminui a energia do sistema, aceitamos essa mudança. Caso contrário, geramos um número aleatório (rn) e comparamos com o peso de Boltzmann p , dado pela Eq.2.4, onde dE é a variação da energia, k_B a constante de Boltzmann e T é a temperatura.

$$p = \exp\{-1 * dE/k_B T\} \quad (2.4)$$

Caso $rn < p$, aceitamos a mudança do spin. Do contrário, recusamos a mudança e partimos para o próximo spin da malha.

3 Fluxograma

Analisado o problema na Seção 2, esquematizamos um fluxograma para implementação da simulação de Monte Carlo do Modelo Ising 2D. Partimos das definições das variáveis utilizadas, até toda a dinâmica necessária.

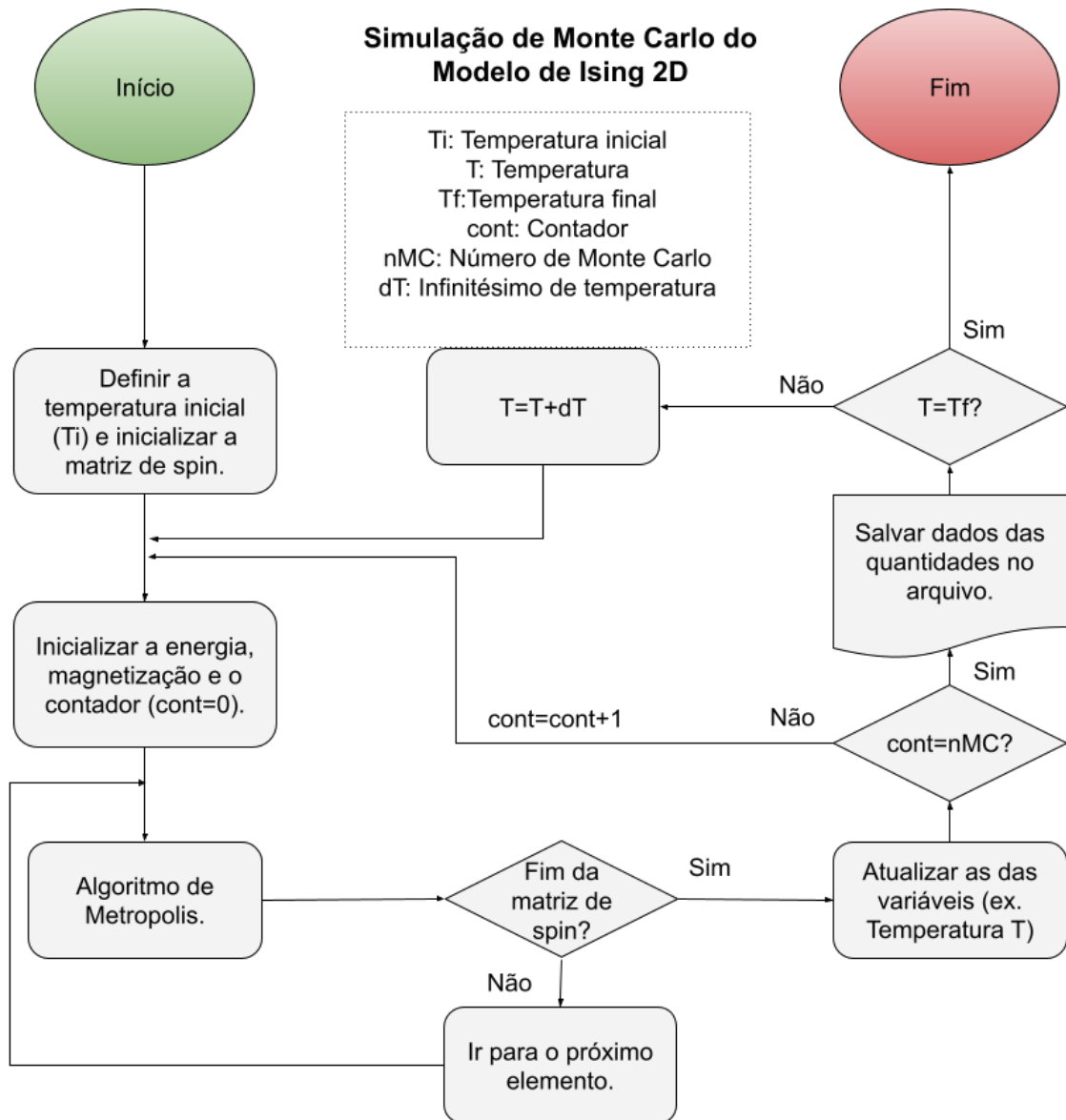


Figure 1 – Fluxograma de uma simulação de Monte Carlo do Modelo de Ising.

4 Programa

Utilizando as definições mostradas na Seção 2 e o esquema da Seção 3, implementamos na linguagem C, um programa que realiza a simulação de Monte Carlo do Modelo de Ising 2D. Foi criada uma malha de tamanho $N \times N$, onde $N = 256$. Esta malha foi preenchida aleatoriamente com 90% de spin "up", com magnetização $s = +1$, e o restante com spin "down", com magnetização $s = -1$. Não inserimos um campo magnético externo, portanto, $H = 0$, na Eq.2.2. Foi definida a temperatura inicial de $T_i = 1$ e a final de $T_f = 5$. Foi escolhido o número de Monte Carlo $nMC = 5000$ e número de passos de Monte Carlo sob a temperatura $nP = 250$, que divide os intervalos de temperatura dT . De modo a ilustrar este fenômeno, foi criada uma função neste programa que salva a configuração da malha para cada acréscimo de temperatura, de modo a criar um Gif com a dinâmica resultante.

Utilizando a máquina disponibilizada no Lab 107, que possui um processador Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, a primeira versão do programa demorou 7530 segundos para gerar os resultados. Podemos ver dois "frames" do Gif resultante da simulação, mostrado nas Figuras 2 e 3.

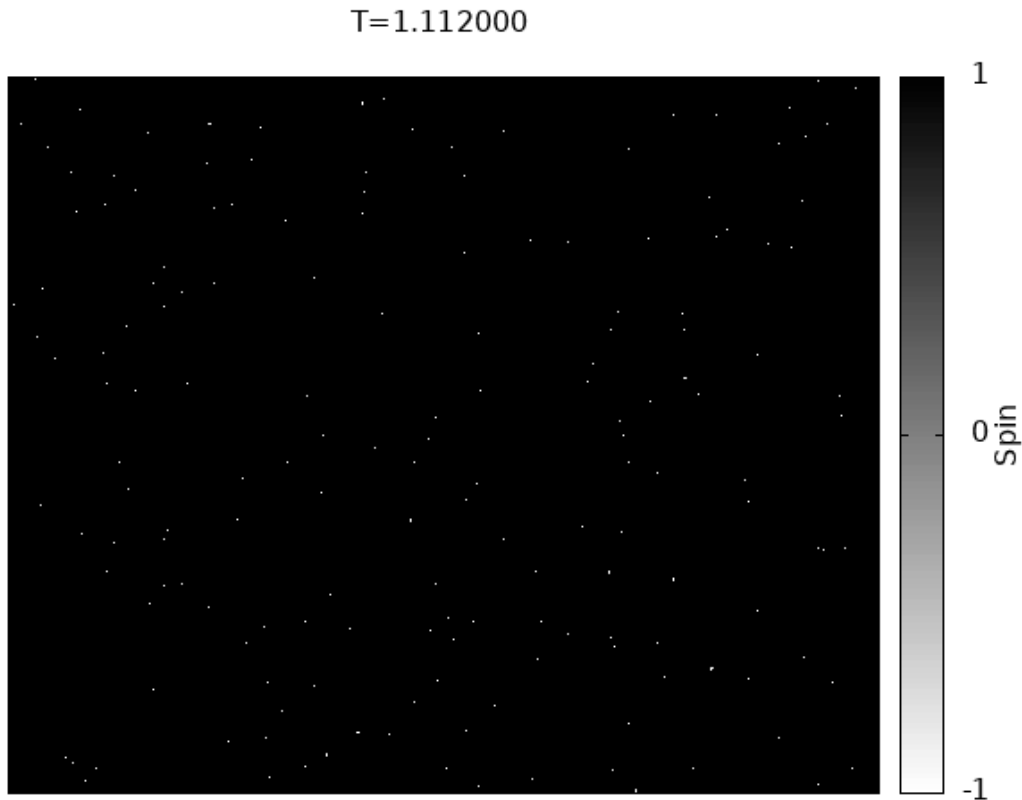


Figure 2 – Configuração inicial da malha de spins.

Nota-se da Figura 2 a configuração inicial, onde a maioria dos spins, possuem magnetização $s = +1$. Já na Figura 3, temos a configuração próxima da temperatura final, onde temos uma distribuição mais homogênea entre os spins "up" e "down".

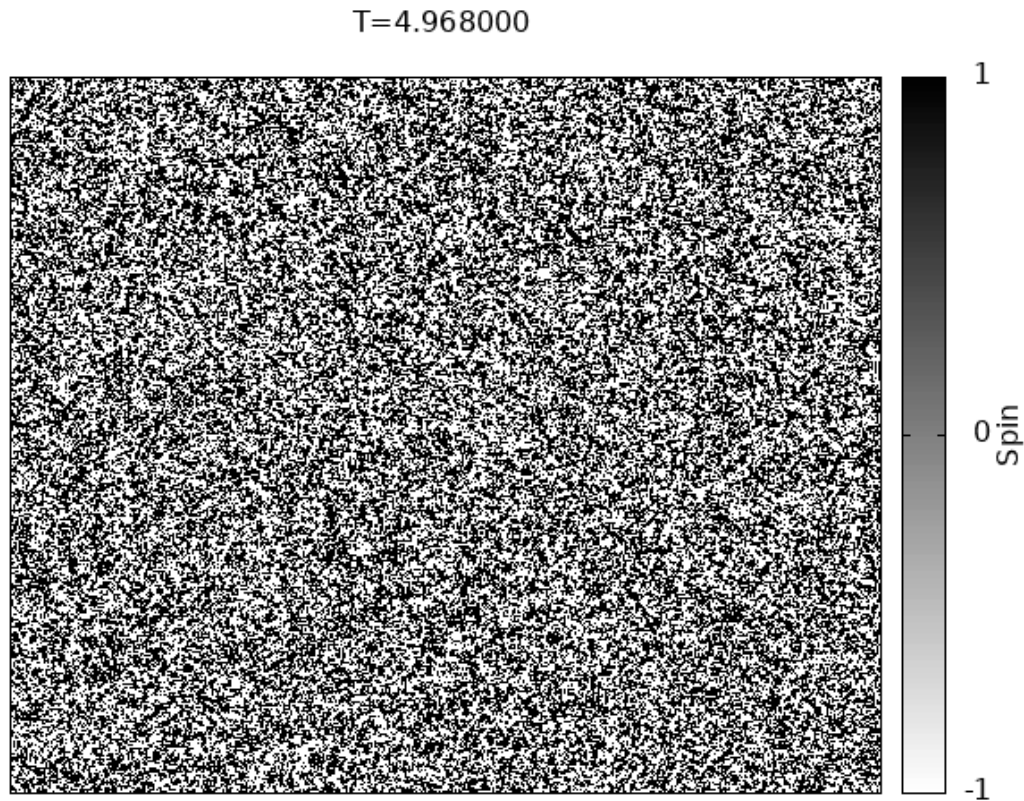


Figure 3 – Configuração final da malha de spins.

A simulação também gerou os valores da magnetização através da Eq.2.1 pela temperatura da malha, onde criamos o gráfico mostrado na Figura 4. Vemos que a magnetização da malha decai com o avanço da temperatura, chegando em um estado de equilíbrio, onde a magnetização flutua em zero. Este resultado era esperado, pois não temos campo magnético externo neste sistema, portanto, ele iria ao equilíbrio naturalmente, como foi mostrado.

Com os valores gerados pela simulação para a energia segunda a Eq.2.2, criamos o gráfico mostrado na Figura 5. Vemos que a energia começa em um valor baixo e vai aumentando enquanto aumentamos a temperatura.

Por fim, também foi coletado os valores referentes ao calor específico, dado pela Eq.2.3, onde estão sendo mostrados no gráfico da Figura 6. Vemos que o calor específico da malha vai aumentando até atingir um valor máximo e inverte a tendência e começa a diminuir. Sabemos que o calor específico é dado também por $C = \frac{dE}{dT}$, logo, como vimos que no gráfico da energia temos uma inversão de concavidade, era esperado que o calor específico apresenta-se um máximo.

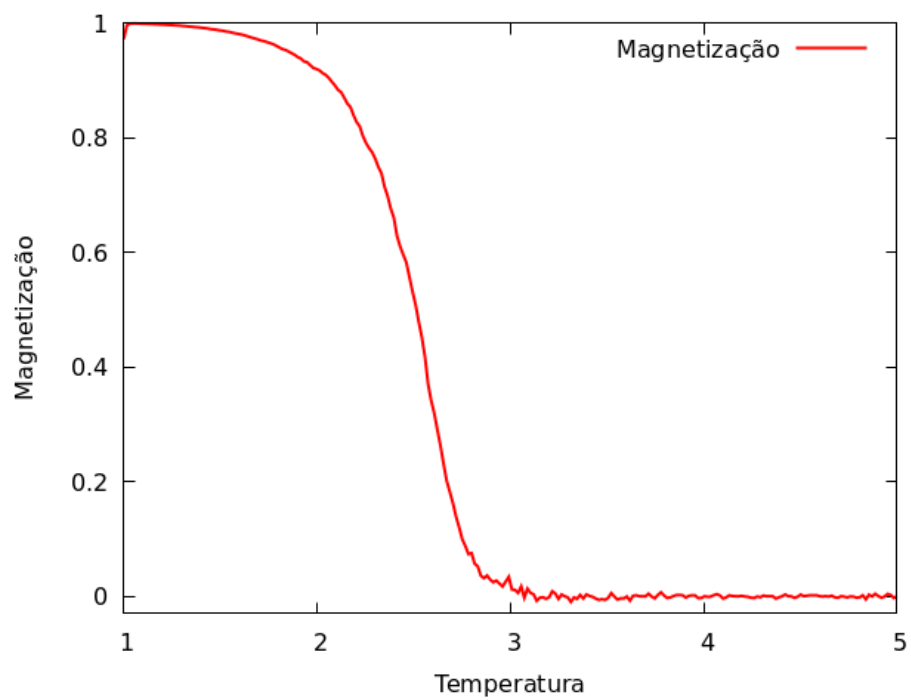


Figure 4 – Magnetização da malha pela temperatura.

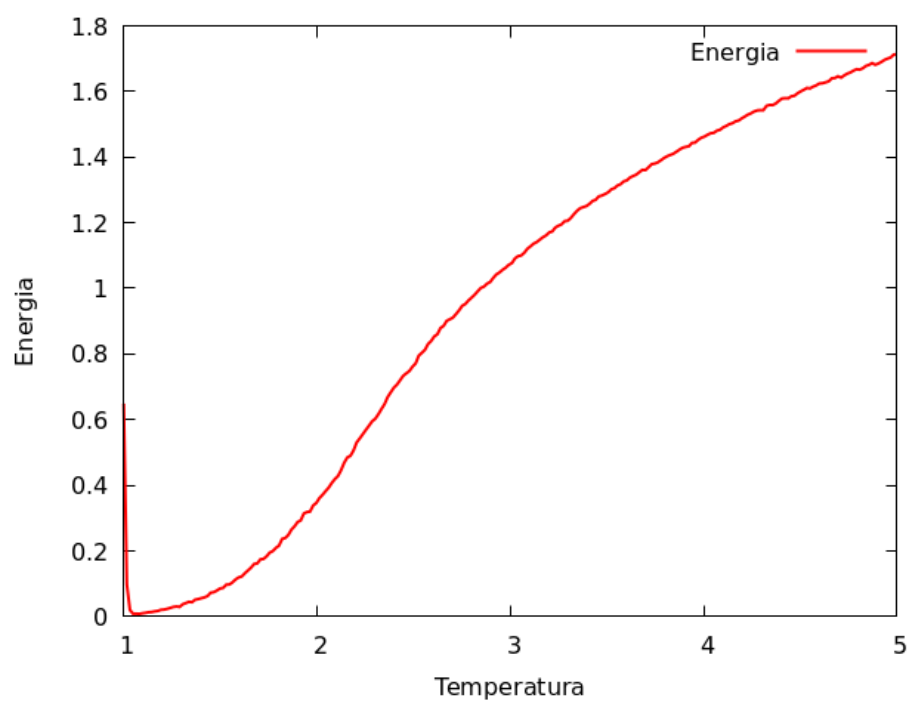


Figure 5 – Energia da malha pela temperatura.

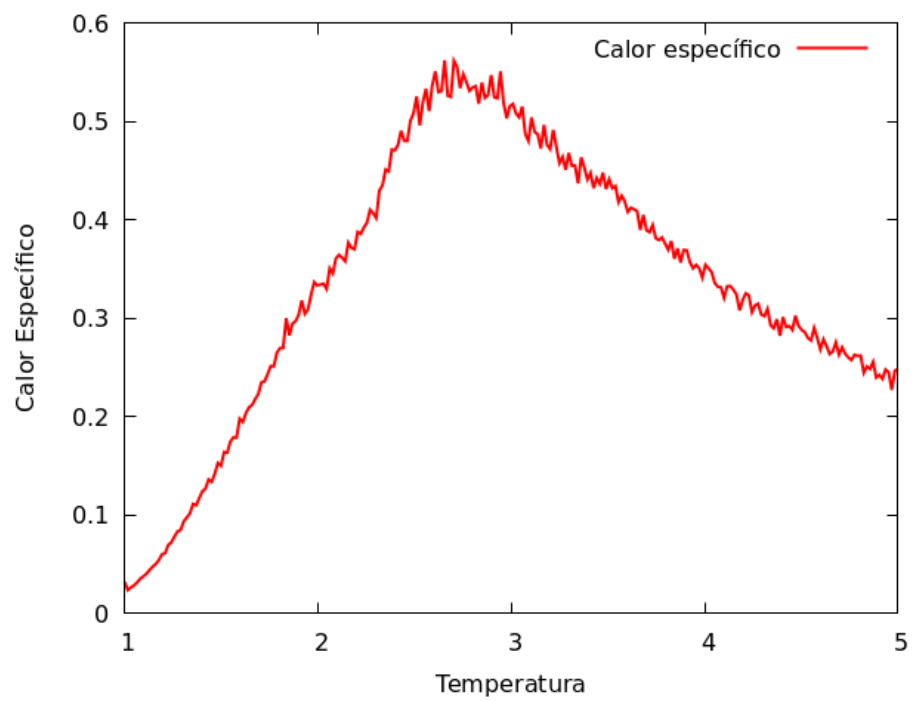
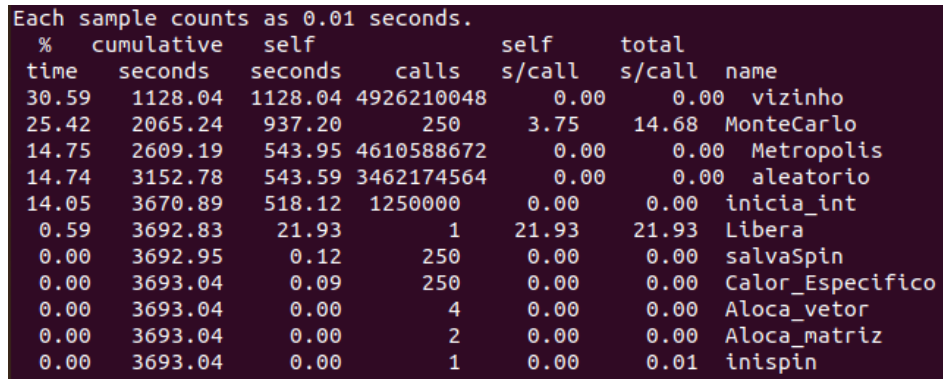


Figure 6 – Calor específico da malha pela temperatura.

5 Profile

De modo a analisar o programa criado, utilizamos a técnica de *profile* para observar onde nosso código leva mais tempo para realizar as tarefas. Com isso, tivemos o resultado mostrado na Figura 7.



```
Each sample counts as 0.01 seconds.
```

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
30.59	1128.04	1128.04	4926210048	0.00	0.00	vizinho
25.42	2065.24	937.20	250	3.75	14.68	MonteCarlo
14.75	2609.19	543.95	4610588672	0.00	0.00	Metropolis
14.74	3152.78	543.59	3462174564	0.00	0.00	aleatorio
14.05	3670.89	518.12	1250000	0.00	0.00	inicia_int
0.59	3692.83	21.93	1	21.93	21.93	Libera
0.00	3692.95	0.12	250	0.00	0.00	salvaSpin
0.00	3693.04	0.09	250	0.00	0.00	Calor_Especifico
0.00	3693.04	0.00	4	0.00	0.00	Aloca_vetor
0.00	3693.04	0.00	2	0.00	0.00	Aloca_matriz
0.00	3693.04	0.00	1	0.00	0.01	inispin

Figure 7 – Profile da versão básica do programa.

Vemos que o nosso programa passa 30% do tempo na função *vizinho*, que calcula a magnetização dos vizinhos próximos de um spin da malha. É uma função de cálculo simples e está em primeiro lugar pois é chamada diversas vezes durante a execução.

Em segundo lugar, vemos que a função *MonteCarlo* ocupa 25% da execução, em poucas chamadas ($nP = 250$, número de passos sob a temperatura). Portanto, temos o principal gargalo do nosso programa, onde temos as principais operações sendo feitas por ali. A cada variação de temperatura, chamamos essa função, e nela, percorremos a malha analisando os spins na malha e realizando os cálculos necessários para o andamento da simulação. A terceira função encontrada é *Metropolis*, que é chamada dentro da função *MonteCarlo*. Temos o mesmo caso da função *vizinho*, onde temos muitas chamadas, e neste caso, temos os condicionais necessários do algoritmo de Metropolis, por este tempo nessa função.

6 Otimização

De modo a conseguir um menor tempo de execução do nosso programa, realizaremos algumas mudanças na escrita do código e a utilização de *flags*, afim de conseguir um melhor desempenho na máquina disponibilizada do laboratório 107.

6.1 Otimização de escrita

Buscou-se reduzir o número de ciclos matemáticos na execução, evitando o uso de repetições de cálculos que envolvesse divisões, ou cálculos que estavam dentro de *loops* sem a necessidade. Desta forma, transformamos todas as divisões em multiplicações, principalmente nas funções que foram mais chamadas mostrado no *profile* realizado na Seção 5. Também realizamos mudanças na forma de criação dos arquivos necessários para confecção do Gif. Com tudo isso, executamos a nova versão do código na mesma máquina do laboratório 107. O novo tempo de execução foi de 4919.4 segundos, o que nos deu 65,3% de performance. Um resultado muito satisfatório.

6.2 Otimização de software

De modo a utilizar a máquina de melhor maneira possível para execução do programa, realizamos três baterias de execução com diferentes flags, variando em cada bateria o nível de otimização $-OX$, com $X = 1, 2, 3$. As três baterias foram com as flags:

1. OX : $-OX$, com $X = 1, 2, 3$.
2. Avx : $-OX$, com $X = 1, 2, 3$ + `-fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=avx -march=avx`
3. $Native$: $-OX$, com $X = 1, 2, 3$ + `-fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native`

Com isso, tiramos o tempo de execução de cada caso e montamos um gráfico mostrado na Figura 8. Na Figura 9, temos o gráfico que mostra as eficiências encontrada para cada nível de otimização com as flags, em relação ao tempo de execução de 4919.4 segundos.

Nota-se que flags com *Native* obtiveram um menor tempo nos três níveis de otimização, o que de fato era esperado, já que essa flag busca as melhores flags do processador. Notamos também, que o nível de otimização $X = 3$ foi o melhor encontrado

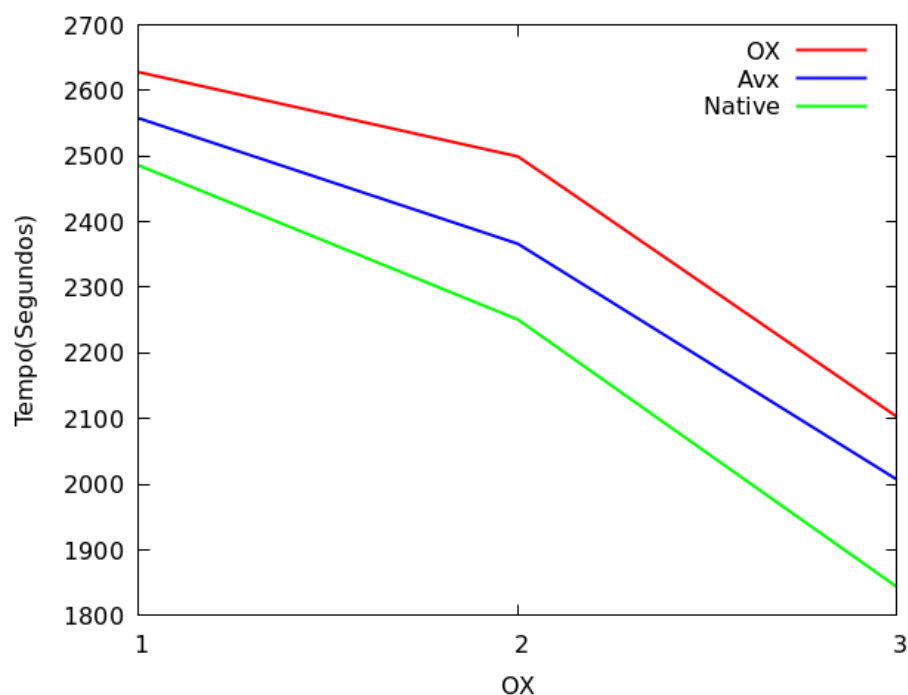


Figure 8 – Tempos por nível de otimização das três etapas.

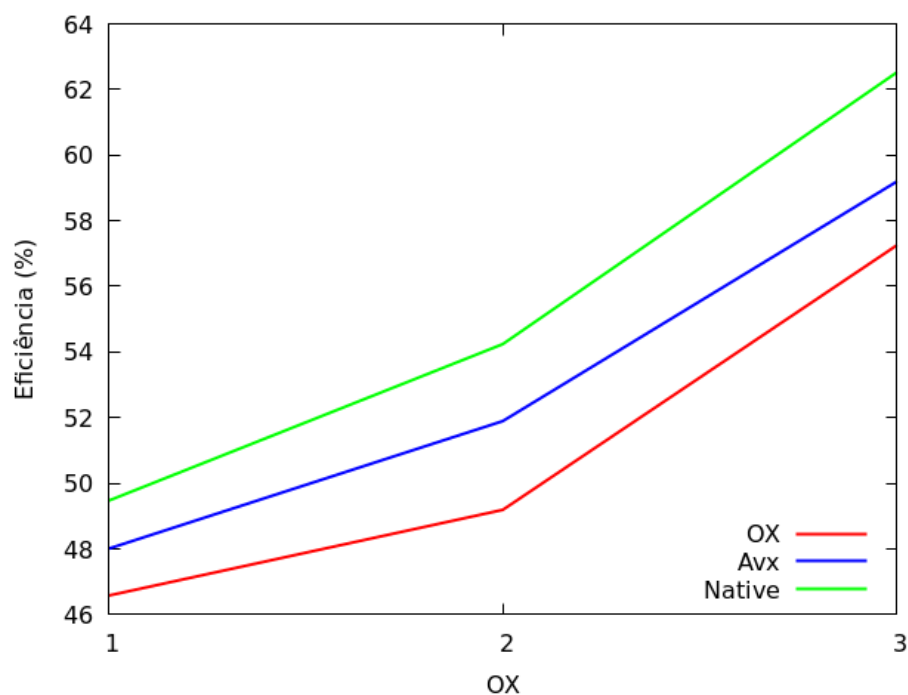


Figure 9 – Eficiência por nível de otimização das três etapas em comparação com o tempo sem flags.

em todos os casos, tendo o menor tempo de 1843.6 segundos para a flag *Native*, o que nos dá uma eficiência de 62.5% em relação ao tempo sem flag. Este nível de otimização é conhecido por ser o mais agressivo e podendo até interferir nos resultados. A forma que escrevemos a função de número aleatório, em todas as execuções, partimos com a mesma semente, tendo assim que o esperado é ter os mesmo resultados em todas as execuções

realizadas, interferindo somente na precisão a mais que cada flag pode trazer. Porém, não foi encontrado uma grande discrepância entre os resultados das baterias realizadas em cada nível de otimização, como podemos ver na Tabela 1.

	Magnetização	Energia	Calor Específico
O1	0.17%	0.15%	0.18%
O2	0.32%	0.29%	0.33%
O3	0.29%	0.25%	0.23%

Table 1 – Erro percentual para os resultados obtidos entre as execuções em cada etapa com flags diferentes.

Com isso, podemos utilizar o terceiro nível de otimização com a flag *Native*, que apresentou um menor tempo, sem se preocupar com a influência nos resultados obtidos.

7 Paralelismo

Com a análise feita na Seção 5, vemos na Figura 7 que o principal "gargalo" estava na função *MonteCarlo*. Nessa função, temos as principais operações realizadas, os cálculos necessários e percorremos toda a malha de spins. Visto isso, esta é a principal parte do código que podemos investigar uma possível aplicação do paralelismo. Veremos agora duas propostas de paralelismo, que serão implementadas com a API OpenMPI na Seção 8.

7.1 Paralelismo nas temperaturas

Como vemos no fluxograma apresentado na Figura 1, um dos loops presentes neste problema é regido pelo acréscimo de temperatura. Assim como neste fluxograma, em nosso código, este loop é o mais externo do programa. O acréscimo dT é definido pelo o número de passos sob a temperatura nP , como:

$$dT = \frac{T_f - T_i}{nP - 1} \quad (7.1)$$

onde T_f e T_i são a temperatura final e inicial, respectivamente. Dessa forma, podemos paralelizar o nosso problema dividindo a temperatura entre os *cores*, buscando definir um valor de nP que ajuste perfeitamente o número de *cores* disponível na máquina.

7.2 Paralelismo no domínio

Como dito, na função *MonteCarlo*, percorremos todos os spins da malha diversas vezes, aplicando as operações necessárias. Dessa forma, a maneira mais intuitiva é buscar uma divisão do domínio entre o número de *cores*. O grande desafio seria implementar uma comunicação entre as divisões. Também temos que levar em conta a forma em que geramos os números aleatórios, de modo a mudar a semente entre os *cores*, para continuar com o bom andamento da simulação.

Iremos, portanto, dividir em tiras horizontais o domínio e passar para os *cores*, se preocupando com a comunicação entre eles. Mudaremos a semente que gera os números aleatórios multiplicando ela pela a identificação do *core* utilizado. Ajustaremos o tamanho do domínio com o número de *cores* disponível, de modo a obter o melhor resultado.

8 MPI

Dado a discussão na Seção 7, implementamos cada uma das propostas com a API MPI. De forma a investigar o andamento das paralelizações, inicialmente, usaremos somente o computador do laboratório 107. Para poder comparar a melhor forma de paralelismo desse problema, ajustaremos as definições dos parâmetros de modo a ter o mesmo problema para as duas propostas. isto é, tanto o número de passos sob a temperatura nP e o tamanho do domínio terão o mesmo valor nos dois casos. Definimos então $nP = 192$ e $N = 192$, com $nMC = 5000$. Escolhemos estes valores pois é o máximo de *cores* que estarão disponíveis no LNCC, além de ajustar bem a capacidade do laboratório 107. Conhecendo as limitações de comunicação entre os *cores* e, posteriormente, entres nos *nós*, removemos a função que gerava os arquivos para confecção do Gif, focando assim nos cálculos do problema. O tempo de execução dessa versão em serial foi de 795.4 segundos.

8.1 Implementação do paralelismo nas temperaturas

Utilizamos a diretiva *MPI_Scatter* para vetorizar a temperatura, dividindo ela entre os *cores* disponíveis, onde cada parte fará um processo de maneira idêntica como da versão em serial, calculando as saídas do programa. Dessa forma, não precisamos retornar para o processo principal (*root*), de forma a agilizar o nosso problema. Porém, temos que tomar cuidado na hora de registrar os resultados da simulação, se preocupando com a ordem em que os valores serão armazenados.

8.2 Implementação do paralelismo no domínio

Neste caso, tivemos que utilizar diversas diretivas para dividir o domínio da malha de spins e possibilitar uma comunicação entre essas subdivisões. Criamos as subdivisões com um novo tipo de vetor utilizando *MPI_Type_vector*, sendo essa subdivisão com o tamanho disponível pelo número de *cores* com acréscimo de dois, onde será armazenada a informação da subdivisão anterior e posterior. Nisso, vemos que quanto mais subdivisões forem feitas, maior será a comunicação, onde podemos esperar uma estabilização do tempo de execução. Para o sincronismo dos valores do programa, utilizamos barreiras (*MPI_Barrier*) entre os processos antes de calcular as variáveis do programa, pois será o *root* que fará os cálculos, de maneira diferente da implementação anterior. Para os cálculos das variáveis, foi utilizado o *MPI_Reduce* com *MPI_SUM*. Para a comunicação entres as subdivisões, utilizamos *MPI_Send* e *MPI_Recv*.

8.3 Comparações das implementações

No gráfico mostrado na Figura 10, vemos que as duas implementações apresentam um ganho de tempo considerável com o aumento do número de *cores*. Os dois programas reduziram seus tempos de maneira similar, mostrando que o objetivo nos dois casos foram concluídos. Os menores tempos foram encontrados ambos para oito *cores*, com 163.3 segundos para a temperatura e 143.7 segundos para o domínio.

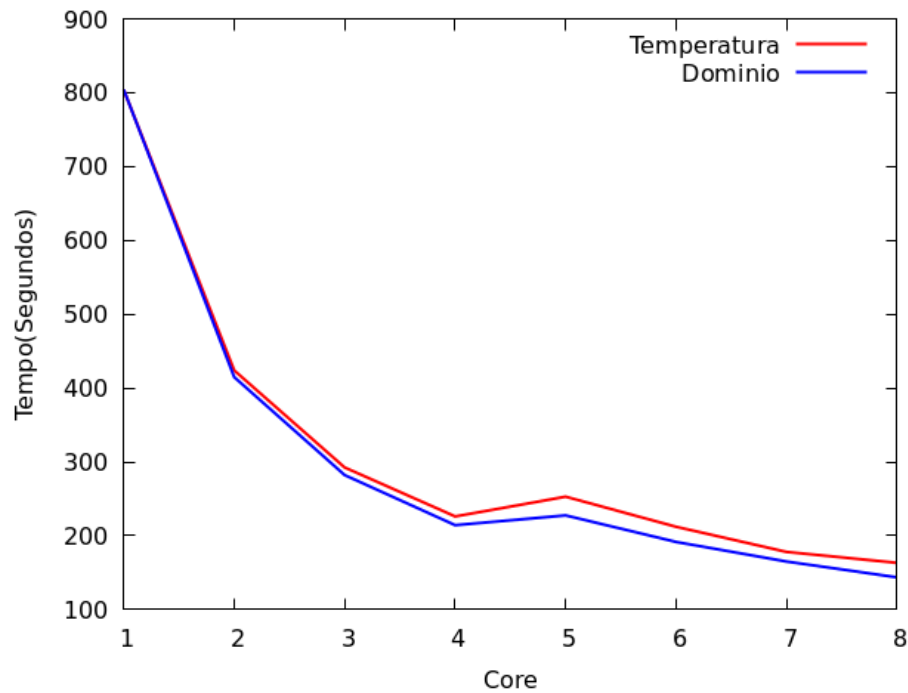


Figure 10 – Tempo de execução pelo número de *cores*.

O gráfico mostrado na Figura 11 mostra a eficiência encontrada levando como referência o tempo em serial de 795.4 segundos. O máximo de eficiência foi obtida nos dois casos com oito *cores*, com 79.5% para paralelismo com a temperatura e 81.9% para o domínio.

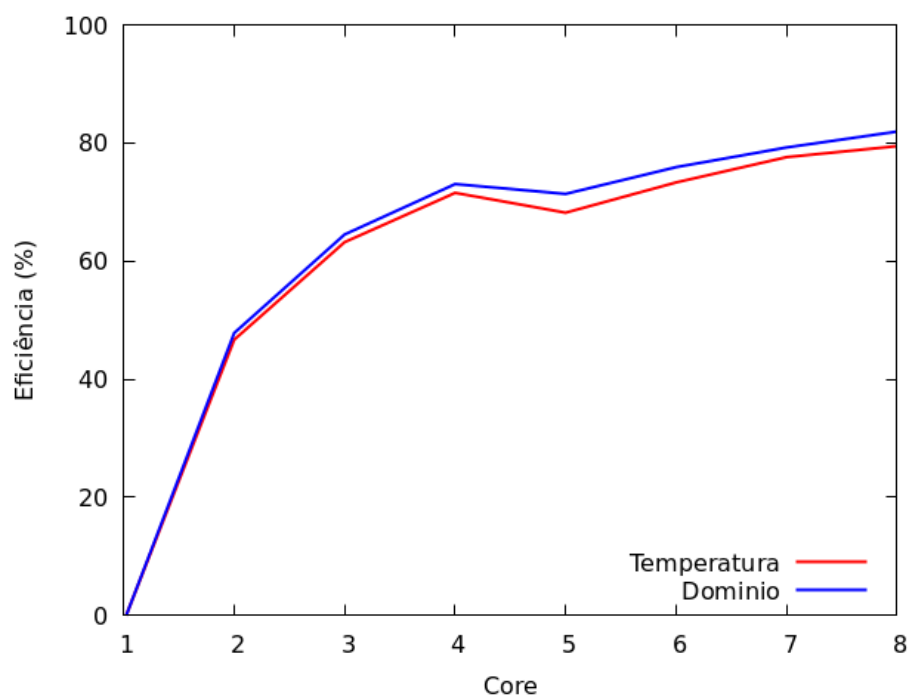


Figure 11 – Eficiência pelo número de *cores*.

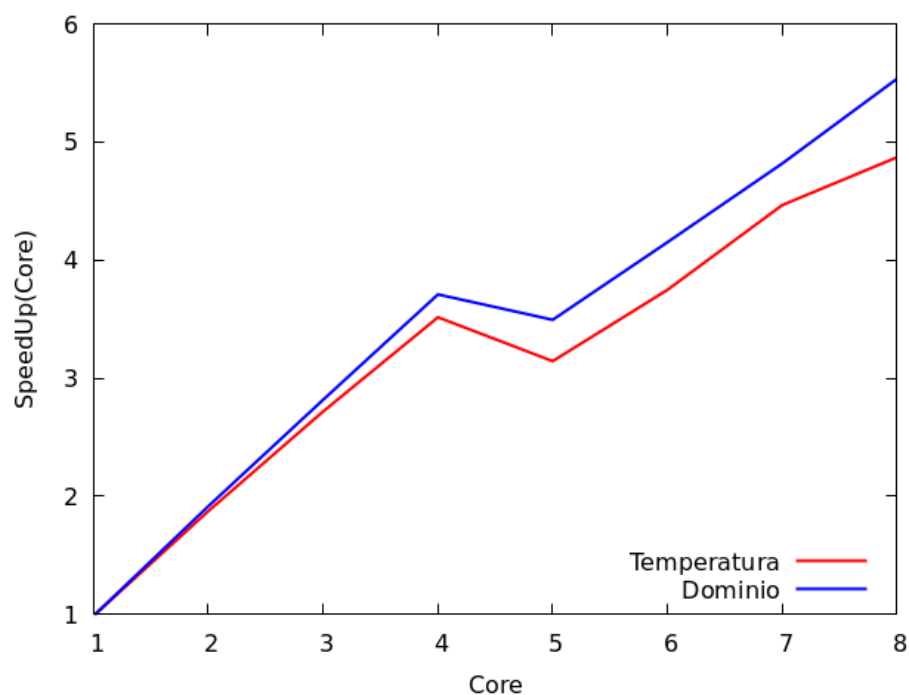


Figure 12 – *SpeedUp* pelo número de *cores*.

É mostrado no gráfico da Figura 12 o *SpeedUp* por *core* dos dois programas paralelizados. O objetivo dessa análise é encontrar até onde y é aproximadamente x . Nos dois casos, com o aumento do número de *cores*, temos essa aproximação até $core = 3$, começando a reduzir logo após. Houve uma queda em $core = 5$ e depois continua a crescer, mas sem apresentar o comportamento desejável. Com isso, vemos a limitação da máquina

do laboratório 107, onde a comunicação entre os *cores* começa a influenciar no tempo.

9 LNCC

10 Conclusão

Em relação aos resultados encontrados na implementação do modelo de Ising 2D utilizando Monte Carlo e o algoritmo de Metropolis, tivemos resultados satisfatórios, que ilustraram o comportamento do fenômeno físico. As técnicas utilizadas para uma otimização de escrita e de software foram fundamentais para o nosso trabalho, onde conseguimos reduzir vertiginosamente o tempo de execução, sem ter muita influência nos resultados da simulação.

A análise feita usando a técnica de *Profile* no nosso programa, foi de muita importância para encontrar caminhos para o paralelismo, onde, conseguimos traçar hipóteses de como aplicar esta técnica. Conseguimos, com sucesso aplicar a API MPI, conseguindo reduzir ainda mais o tempo de execução, podendo perceber a influência do uso do número de *cores* e a as dificuldades na hora de comunicar as divisões de tarefas para cada processador. Por fim, temos como satisfeito com os resultados encontrados nesse trabalho.

Bibliography

EDUCATION, I. C. Simulação de monte carlo. *IBM*, 2020. Disponível em: <https://www.ibm.com/br-pt/cloud/learn/monte-carlo-simulation>. Acesso em: 20 de Janeiro de 2022. Citado na página 4.

LÍBERO, V. L. De ising a metropolis. *Revista Brasileira de Ensino de Física*, v. 22, n. 3, p. 346–352, 2000. Disponível em: http://www.sbfisica.org.br/rbef/pdf/v22_346.pdf. Acesso em: 3 de Novembro de 2021. Citado na página 4.

SATO, F. Algoritmo de metropolis. *Departamento de Física – Instituto de Ciências Exatas, Universidade Federal de Juiz de Fora – UFJF*, 2021. Disponível em: <https://www.fisica.ufjf.br/~sjfsato/fiscomp1/node58.html>. Acesso em: 20 de Janeiro de 2022. Citado na página 4.