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

PROJETO: COMPUTAÇÃO CIENTÍFICA DE ALTO DESEMPENHO

Autor: Daniel Santiago da Silva
Professor: Rodrigo Amorim

Volta Redonda

2022

Sumário

1 INTRODUÇÃO 5

2 OBJETIVO 6

3 METODOLOGIA 7
3.1 Ferromagnetos 7
3.2 O modelo de Ising 8
3.3 Método de Monte Carlo 9
3.3.1 Algoritmo de Metropolis 9

4 IMPLEMENTAÇÃO 11
4.1 Fluxograma 11
4.2 Condições iniciais 12

5 BENCHMARK 13
5.1 Comparação entre as máquinas 21

6 PROFILE 23

7 TÉCNICAS DE OTIMIZAÇÃO DE SOFTWARE 24

8 COMPARAÇÃO DO PROGRAMA BASE VS PROGRAMA OTIMIZADO 27

9 IMPLEMENTAÇÃO DO PROGRAMA COM OPENMP 28

10 BENCHMARK DO PROGRAMA COM VÁRIAS THREADS 32
10.1 Lab107C 32
10.2 LNCC - B710 37
10.3 LNCC - SequanaX 38

11 VALIDAÇÃO DOS RESULTADOS 40
11.1 Serial 41
11.2 Paralelo 43
11.3 Análise do comportamento do modelo 44
11.3.1 Magnetização 45
11.3.2 Energia 45
11.3.3 Calor específico 46

11.3.4 Susceptibilidade Magnética 47

12 CONCLUSÕES 49

REFERÊNCIAS 50

Resumo

O presente trabalho descreve um estudo sobre o modelo de Ising 2D utilizando-se do método de Monte Carlo para se obter os resultados de interesse. Aliado a isso, diversas técnicas de otimização de compilação e software foram aplicadas com o objetivo de se reduzir o tempo de execução do programa. Mais adiante, técnicas de paralelismo também são aplicadas afim de se obter um ganho ainda maior no tempo de execução do programa (mantendo os resultados confiáveis). Os resultados gerais obtidos serão analisados de forma a entender se o modelo foi aplicado corretamente e se as técnicas de ganho de desempenho aplicadas realmente tiveram o efeito esperado.

Abstract

The present work describes a study on the 2D Ising model using the Monte Carlo method to obtain the results of interest. Allied to this, several compilation and software optimization techniques were applied in order to reduce program execution time. Further on, parallelism techniques are also applied in order to obtain an even greater gain in program execution time (keeping results reliable). The general results obtained will be analyzed in order to understand if the model was applied correctly and if the applied performance gain techniques really had the expected effect.

1 Introdução

O comportamento crítico pode ser notado em diversos sistemas físicos, e em especial, os sistemas que exibem magnetismo vem sendo objeto de grande estudo nas últimas décadas. O aumento no interesse nesses sistemas se deve a possibilidade de se utilizar métodos computacionais para estudar esse fenômeno, tais como o modelo de Ising, modelo de Heisenberg e etc. Esse modelos são utilizados já que obter uma solução analítica para um sistema grande pode ser algo extremamente complexo ou até mesmo inviável. Juntamente com a utilização de métodos como o de Monte Carlo, modelos computacionais são ferramentas extremamente poderosas para se lidar com sistemas de muitas partículas. Neste trabalho, o foco é o modelo de Ising, que convencionalmente é utilizado para investigar o comportamento de um ferromagneto.

As propriedades magnéticas dos materiais já eram conhecidas desde a antiguidade clássica e foram estudadas de maneira mais profunda a partir do século XX. Todas substâncias apresentam características magnéticas em todas temperaturas, mostrando que o magnetismo é uma propriedade básica de qualquer material, portanto, entender essa característica se torna extremamente importante. O comportamento magnético tem origem quântica vindo exclusivamente de duas fontes: momento angular orbital atômico e momento angular do spin dos férmions que compõem a matéria. Quando exposto a um campo magnético externo $\vec{B}$, o material irá reagir de uma certa forma e isso determinará seu comportamento magnético, que pode ser dividido em sete tipos diferentes: ferromagnetismo, ferrimagnetismo, antiferromagnetismo, paramagnetismo, diamagnetismo, vidro de spin e gelo de spin. Como comentado anteriormente, ao utilizar o modelo de Ising, o foco geralmente são os ferromagnetos.

2 Objetivo

O objetivo deste trabalho é aplicar técnicas de otimização a nível de compilador e software e técnicas de paralelismo em um algoritmo para checar se há ganho desempenho no tempo de execução do mesmo. O algoritmo escolhido é uma aplicação do Modelo de Ising 2D, portanto, além de aplicar as técnicas para melhorias no tempo, iremos estudar o comportamento de um ferromagneto utilizando-se do método de Monte Carlo para se obter os observáveis. Nesse caso, as observáveis de interesse serão: Magnetização absoluta média por spin, energia média por spin, calor específico por spin e susceptibilidade magnética por spin. Essas observáveis possuem valores analíticos para um malha 2x2 que será utilizada para a validação dos resultados posteriormente.

3 Metodologia

Para a realização deste trabalho é necessário entender o comportamento de um ferromagneto e como o modelo de Ising 2D e o método de Monte Carlo funcionam, portanto, uma breve discussão acerca destes três tópicos será realizada.

3.1 Ferromagnetos

Em um ferromagneto, os dipolos magnéticos vizinhos se alinham paralelos uns aos outros, formando domínios (regiões microscópicas magnetizadas) mesmo sem um campo externo estar atuando sobre o material, isto significa que um ferromagneto possui magnetização natural não nula. Além disso, esse tipo de material quando exposto a um campo magnético é fortemente magnetizado no mesmo sentido do campo aplicado e como possui uma "memória magnética", quando esse campo externo for retirado, o alinhamento dos domínios permanece no mesmo sentido. A figura [1] exemplifica esse comportamento.

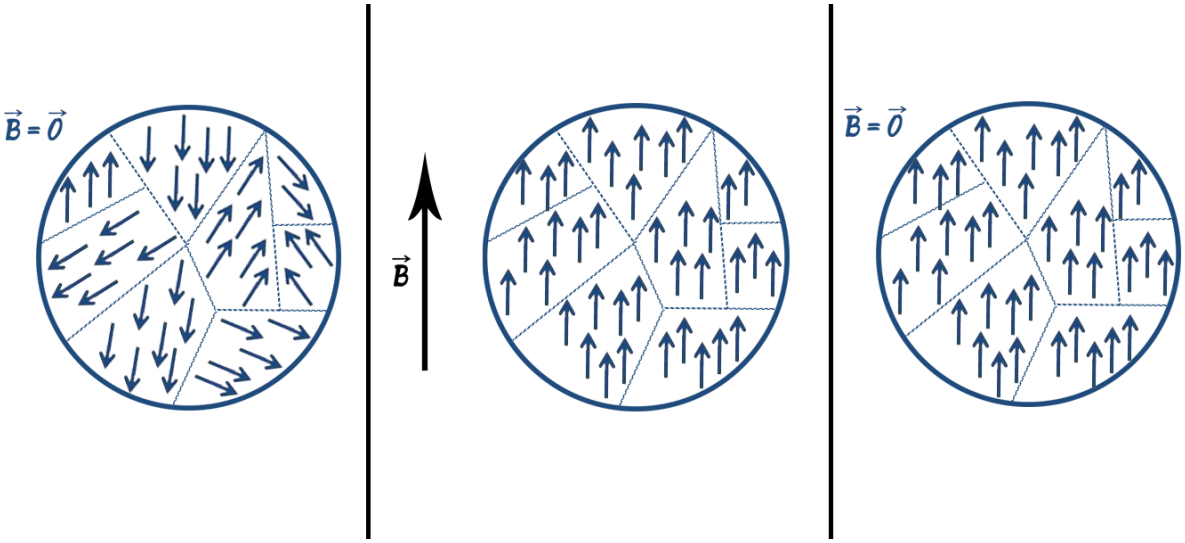

Figura 1 – Comportamento ferromagnético

Esse alinhamento é mantido indeterminadamente ou até que esse material seja elevado a uma temperatura crítica T_c, denominada Temperatura de Curie, onde essa magnetização se torna nula, e, acima dessa temperatura, o ferromagneto se transforma em um paramagneto, isto é, ocorre uma transição de fase. A figura [2] exemplifica essa transição.

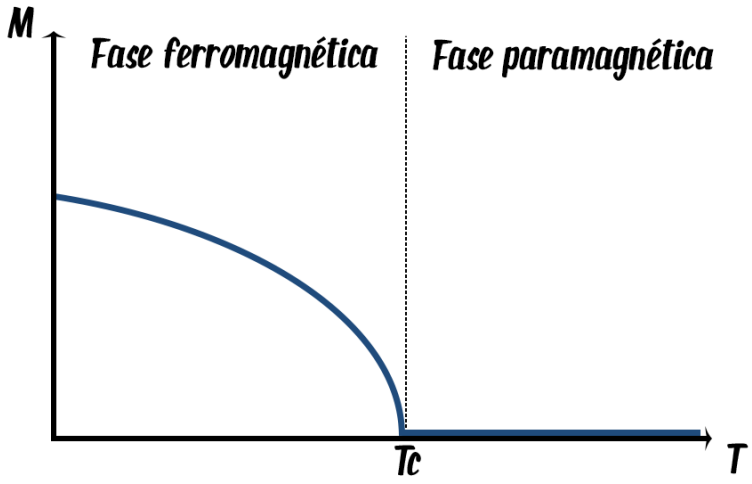

Figura 2 – Comportamento crítico de um ferromagneto

Essa magnetização natural que os ferromagnetos possuem pode não ser percebida mesmo abaixo da temperatura T_c, esse fato ocorre por esses materiais serem divididos em domínios magnetizados que apontam em sentidos aleatórios de forma que a magnetização resultante se torna praticamente nula. Porém, essa magnetização natural dos ferromagnetos pode ser observada utilizando-se o modelo de Ising 2D (o modelo 1D não apresenta essa possibilidade).

3.2 O modelo de Ising

O modelo de Ising (SCHROEDER, 1999; GIORDANO, 1997) foi proposto em 1920 com o objetivo de estudar fenômenos magnéticos nos materiais. Este modelo utiliza uma rede quadrada com N spins onde as orientações dos spins estão limitadas a duas possibilidades: apontar para cima ou para baixo. A hamiltoniana deste modelo é descrita por

$$H = -J \sum_{<i,j>} \sigma_i \sigma_j - B \sum_i \sigma_i \tag{3.1}$$

onde J é a energia de interação entre os spins vizinhos e B representa o campo magnético externo.

Para um spin *up*, é atribuído valor 1, e para um spin *down*, é atribuído valor -1, de forma que $\sigma_i = \pm 1$. Sendo que a primeira soma é feita sobre os pares de spins vizinhos e a segunda sobre o número N de spins (atuação do campo externo em cada posição). O termo <i,j> significa que a soma é feita somente nos spins vizinhos.

3.3 Método de Monte Carlo

O Método de Monte Carlo recebe esse nome aproximadamente no meio do século XX, onde foi usado para se obter soluções numéricas de integrais, isto é, o método não surgiu para resolver problemas físicos mas foi melhorado posteriormente e se tornou uma poderosa ferramenta para resolver problemas de mecânica estatística. Esse método define uma classe de algoritmos baseados em amostragem estocástica massiva para obter aproximações numéricas e são uma forma de resolver quaisquer problemas que possuem natureza probabilística. Em problemas físicos geralmente esses métodos seguem uma determinada receita:

- Define-se um sistema. Neste caso a especificação de uma configuração inicial de *spins* $\{\mathbf{S_i}\}$;
- Um dos *spins* é escolhido e seu valor modificado, e a aceitação ou não desta mudança irá depender do chamado algoritmo de Metropolis.
- Após N tentativas, com N sendo o número de *spins*, temos um passo de Monte Carlo, e o processo então é repetido. Após vários passos de MC, o valor esperado de uma quantidade X é estimado.

Dessa forma, as simulações de Monte Carlo geram consecutivas configurações de *spin* que não guardam encadeamento temporal entre si, formando uma Cadeia de Markov. No equilíbrio termodinâmico, os valores das grandezas físicas para cada configuração (microestado) irão oscilar em torno dos correspondentes valores médios, com a amplitude de oscilação proporcional à temperatura.

A vantagem desses métodos estocásticos para o cálculo de integrais, por exemplo, é que a convergência só depende do número de passos utilizados, diferentemente do método de diferenças finitas que dependem da dimensão espacial. Como consequência disso, o método de Monte Carlo é extremamente útil em sistemas com várias partículas e por isso é vantajoso se utilizar desse método em um sistema magnético.

3.3.1 Algoritmo de Metropolis

Um dos algoritmos utilizados nas simulações de Monte Carlo é o algoritmo de Metropolis. Esse algoritmo gera configurações utilizando uma distribuição de Boltzmann, fazendo uma série de transições aleatórias entre os estados, desta forma, há a garantia de que a configuração final do sistema seja a de equilíbrio. Pode-se representar o funcionamento desse algoritmo da seguinte forma:

- A configuração inicial do sistema é definida. Neste caso, a configuração inicial de *spins* $\{\mathbf{S_i}\}$;

- Uma nova configuração é gerada. Neste caso uma nova configuração de *spins* $\{\mathbf{S_j}\}$;
- Calcula-se a variação de energia entre essas configurações;
- Se essa variação for negativa, a nova configuração $\{\mathbf{S_j}\}$ é mantida;
- Caso a variação seja positiva, um número aleatório r no intervalo de [0, 1] é gerado;
- Se esse número r for menor que a probabilidade p gerada pela distribuição de Boltzmann, a nova configuração $\{\mathbf{S_j}\}$ é mantida;
- Caso contrário, a nova configuração $\{\mathbf{S_j}\}$ é descartada;
- Esse processo é repetido N vezes, onde N é o número de spins do sistema.

4 Implementação

4.1 Fluxograma

Antes da elaboração do algoritmo que irá ser utilizado neste trabalho, é interessante construir um fluxograma de forma a se visualizar como o funcionamento irá ocorrer. Desta forma, o fluxograma é descrito na figura [3].

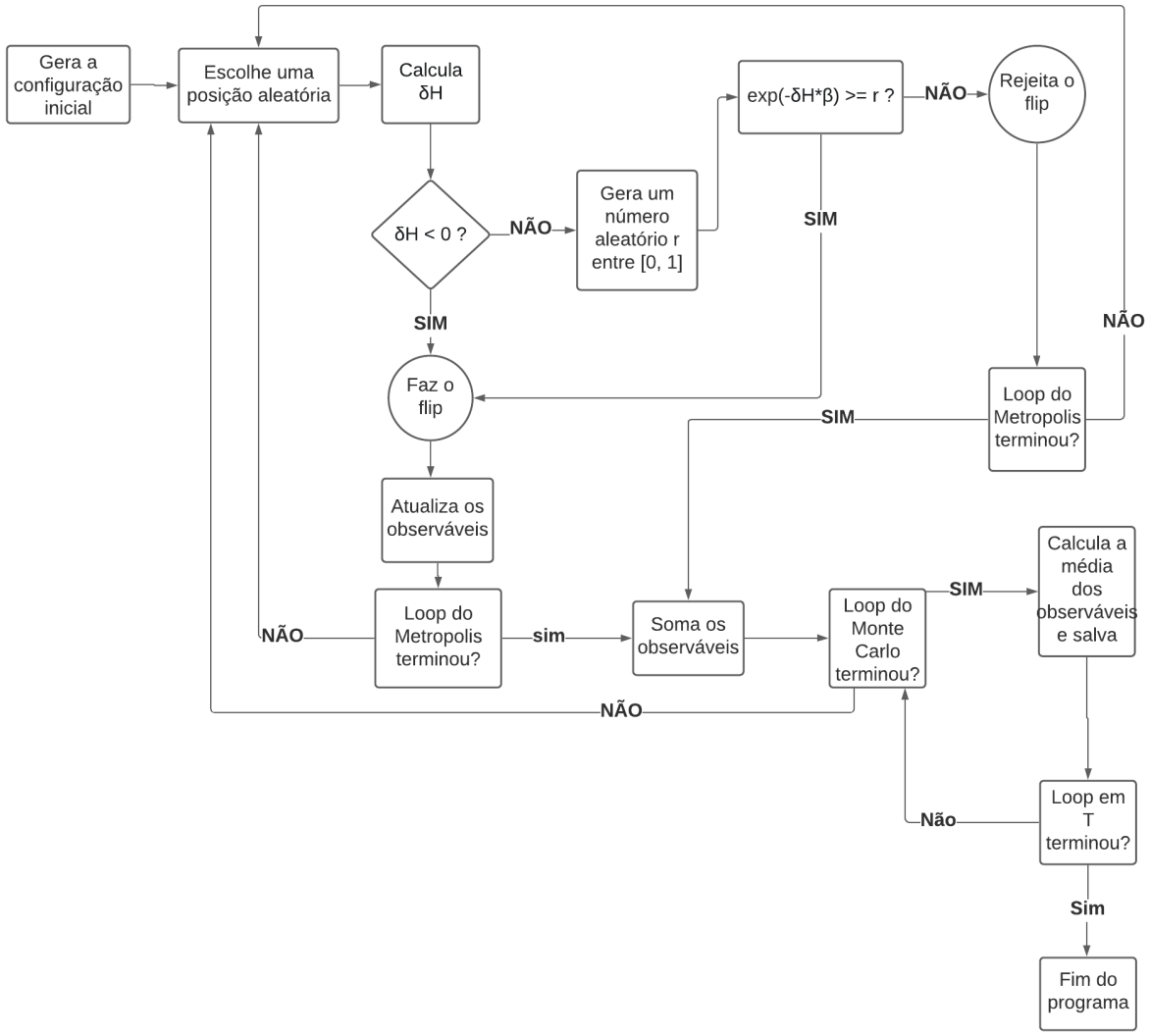

Figura 3 – Fluxograma

Onde T é a temperatura, δH é a variação de energia e $\exp(-\delta H * \beta)$ é a probabilidade p gerada pela distribuição de Boltzmann.

4.2 Condições iniciais

A Hamiltoniana do modelo de Ising geral é dada como mostra a equação (3.1), porém, uma solução analítica desse caso nunca foi realizada. Com base nisso, a forma utilizada no algoritmo deste trabalho foi o modelo de Ising simplificado (que teve a solução analítica obtida por Osanger) cuja Hamiltoniana é definida por

$$H = -J \sum_{<i,j>} \sigma_i \sigma_j \tag{4.1}$$

Isto é, o termo do efeito Zeeman, causado pelo campo externo, foi desprezado ($B = 0$).

Além disso, o termo da energia de troca J e a constante de Boltzmann k_B foram tomadas com valor 1. Desta forma, temos que

$$\beta = \frac{1}{k_B T} = \frac{1}{T} \tag{4.2}$$

E a equação (4.1), efetivamente se torna

$$H = -\sum_{<i,j>} \sigma_i \sigma_j \tag{4.3}$$

Ademais, utiliza-se no algoritmo de Metropolis a distribuição de Boltzmann, que nesse caso será dada por

$$P(\alpha) = e^{-\beta \delta H(\alpha)} \tag{4.4}$$

A rede de spins é iniciada de forma aleatória, isto é, os spins assumem valores de 1 ou -1 aleatoriamente ao iniciar o programa (com probabilidade de 50%). Inicialmente também é definido um número de passos necessários para que o sistema entre em equilíbrio. Além disso, tratar as bordas da rede é importante, uma vez que ao passar por spins nas extremidades, sem um tratamento adequado, o modelo não irá funcionar corretamente. Tendo isso em vista, condições de contorno foram aplicadas de forma que se o spin selecionado estiver em alguma extremidade a soma é feita com o spin da extremidade oposta, isso faz com que a forma geométrica da rede se comporte como um toróide.

5 Benchmark

LAB 107C

A tabela [1] mostra as configurações da máquina utilizada para a realização do benchmark.

Processador	Intel(R) Core(TM) i7-2600
Número de núcleos	4
Número de threads (Hyper-threading)	8
Clock Base	3.40GHz
Clock Turbo Máximo	3.80GHz
Cache L1	32KB
Cache L2	256KB
Cache L3	8MB
BoboMips	6784.77
Memória RAM	12GB DDR3

Tabela 1 – Configurações da máquina LAB 107C

Além das condições iniciais já descritas em (4.2), a malha utilizada foi de $L = 528$, isso implica que o número de spins foi de $N = 528$x528; além disso, o número de passos de Monte Carlo foi de $N_{MC} = 9600$ e o número de passos para a termalização foi setado em $N_{skip} = 96$; sendo a temperatura inicial definida em $T_i = 0.5$ e a temperatura final em $T_f = 5$ com o $dT = 0.1$. Esses valores não foram escolhidos aleatoriamente e serão explicados mais adiante. Vale ressaltar que o algoritmo utilizado no benchmark foi o algoritmo já com as técnicas de otimização de software aplicadas, que estão explicadas em (7).

Então, para a realização do benchmark, primeiramente aplicou-se somente as flags **-OX**, onde $X = \{0, 1, 2, 3\}$.

Posteriormente, mais 4 rodadas foram realizadas utilizando as flags **-OX -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native**.

Os tempos obtidos nas 8 rodadas estão na tabela [2].

Flags	0	1	2	3
OX	10039s	7993s	8011s	7885s
OX + Flags	10019s	7775s	7897s	7623s

Tabela 2 – Tempos obtidos com as compilações usando o gcc

Pode-se comparar o tempo obtido em cada compilação a fim de entender as diferenças variando as flags, isso está demonstrado em [4].

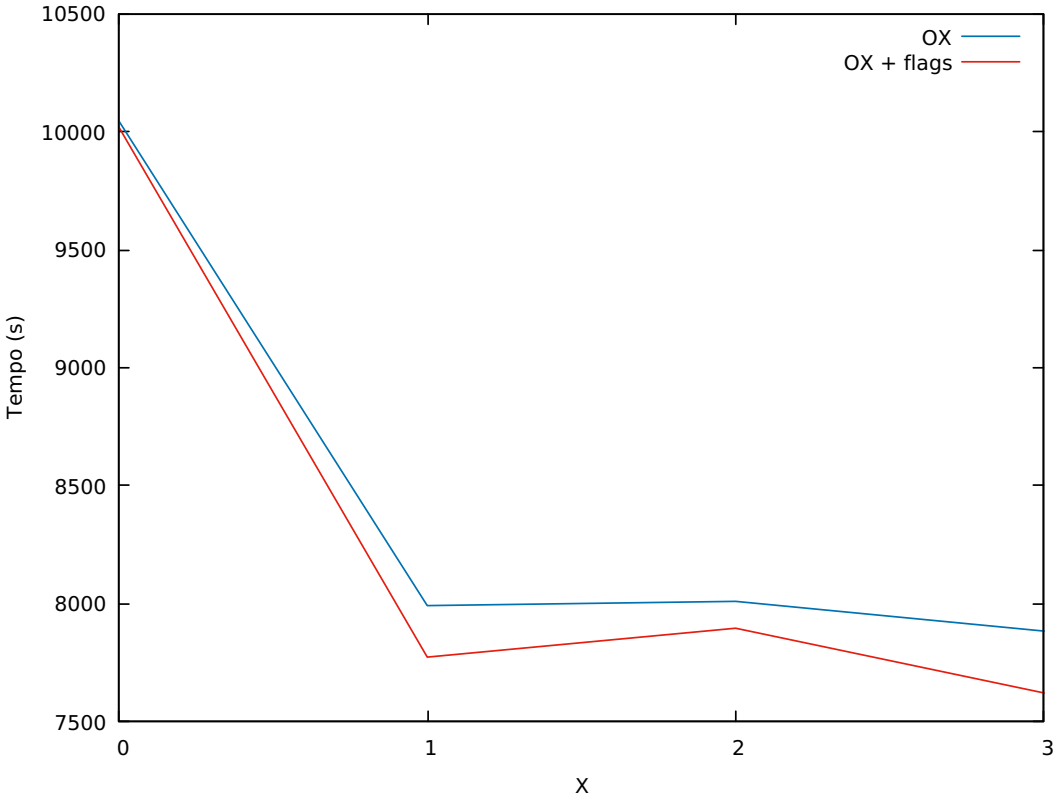

Figura 4 – Comparação geral entre o tempo pelo X

Aqui, pode-se inicialmente notar que, para todos valores de **X**, as compilações com as demais flags (além do **-OX**) resultaram em tempos menores, mesmo que as diferenças não sejam muito grandes (a maior diferença ficou com **262s** no caso **O3**). Além disso, fica claro que o padrão é o mesmo: o pior tempo fica com **O0**, seguido de **O2**, depois **O1** e o melhor tempo sempre fica com **O3**. Portanto, o melhor tempo acontece no conjunto de flags **-O3 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native** sendo de **7623s**, que equivale a **2h:7m:3s**.

Fixando o pior tempo como o tempo do programa base, discutido em (6), isto é, **10480**, podemos calcular a eficiência que cada compilação resulta. Ou seja, plotar o gráfico da eficiência vs otimização, onde a eficiência é dada por

$$\eta[\%] = \left(1 - \frac{T_{opt}}{T_{no-opt}}\right) \cdot 100 \tag{5.1}$$

onde T_{opt} é o tempo que deseja-se saber a eficiência e T_{no-opt} é o pior tempo obtido.

Portanto, neste caso, $T_{no-opt} = 10480$s e o gráfico da comparação entre as eficiências em função das flags é descrito em [5].

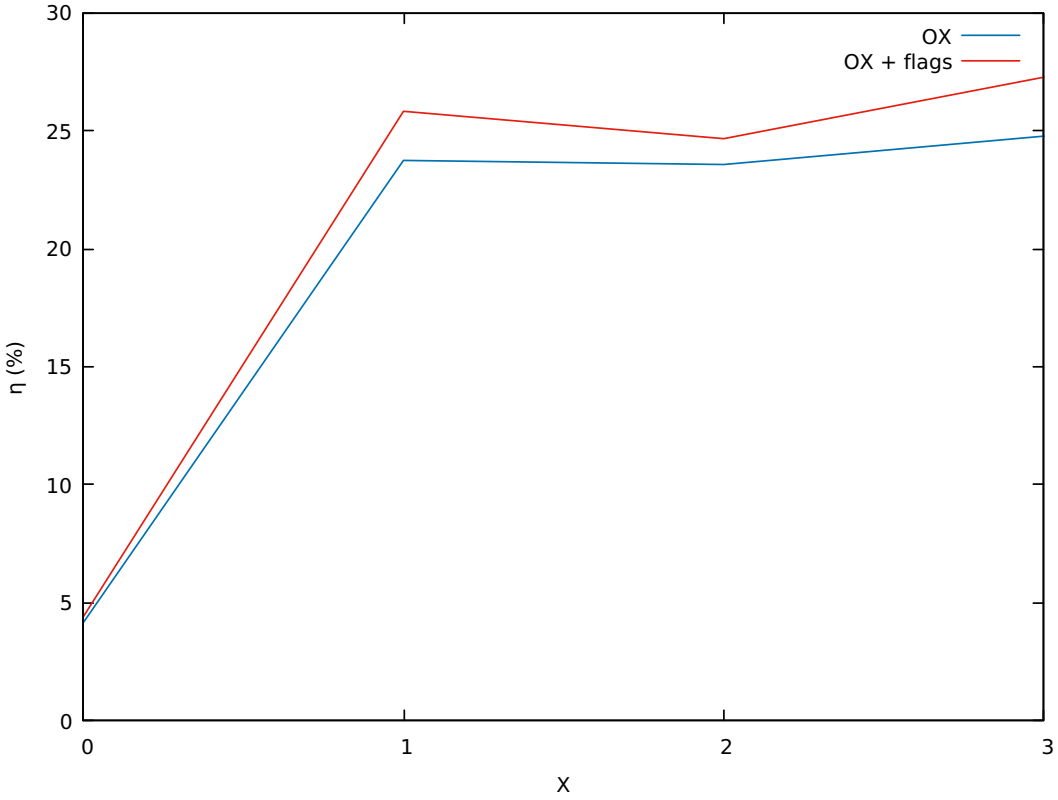

Figura 5 – Comparação geral entre a eficiência pelo X

Onde pode-se notar uma eficiência de aproximadamente 5% nos casos com **O0**, entre $23 - 26\%$ nos casos com **O1** e **O2** e no caso **O3** temos uma eficiência de 24.76% sem as demais flags e o melhor caso com as demais flags, mostrando uma eficiência de 27.26%, que equivale a uma redução de aproximadamente **50m** no tempo de execução total usando a malha com $L = 528$.

LNCC - B710

A tabela (3) mostra as configurações da máquina utilizada para a realização do benchmark.

Processador	Intel(R) Xeon(R) E5-2695 v2
Número de núcleos	12
Número de threads (Hyper-threading)	24
Clock Base	2.40GHz
Clock Turbo Máximo	3.20GHz
Cache L1	32KB
Cache L2	256KB
Cache L3	30MB
BoboMips	4799.86
Memória RAM	64GB DDR3

Tabela 3 – Configurações da máquina

As condições iniciais utilizadas para a execução nesta máquina foram as mesmas utilizadas anteriormente no LAB 107C, com exceção do tamanho da malha, que foi de $L = 128$ (pelo limite de tempo para rodar no LNCC de 20 minutos). Os conjuntos de flags utilizados também foram os mesmos usados anteriormente no LAB 107C.

Os tempos obtidos nas 8 rodadas estão na tabela [4].

Flags	0	1	2	3
OX	558s	454s	459s	450s
OX + Flags	540s	434s	455s	431s

Tabela 4 – Tempos obtidos com as compilações usando o gcc

Com os tempos em mãos, pode-se comparar os resultados obtidos para cada compilação a fim de entender as diferenças variando as flags, isso está demonstrado em [6].

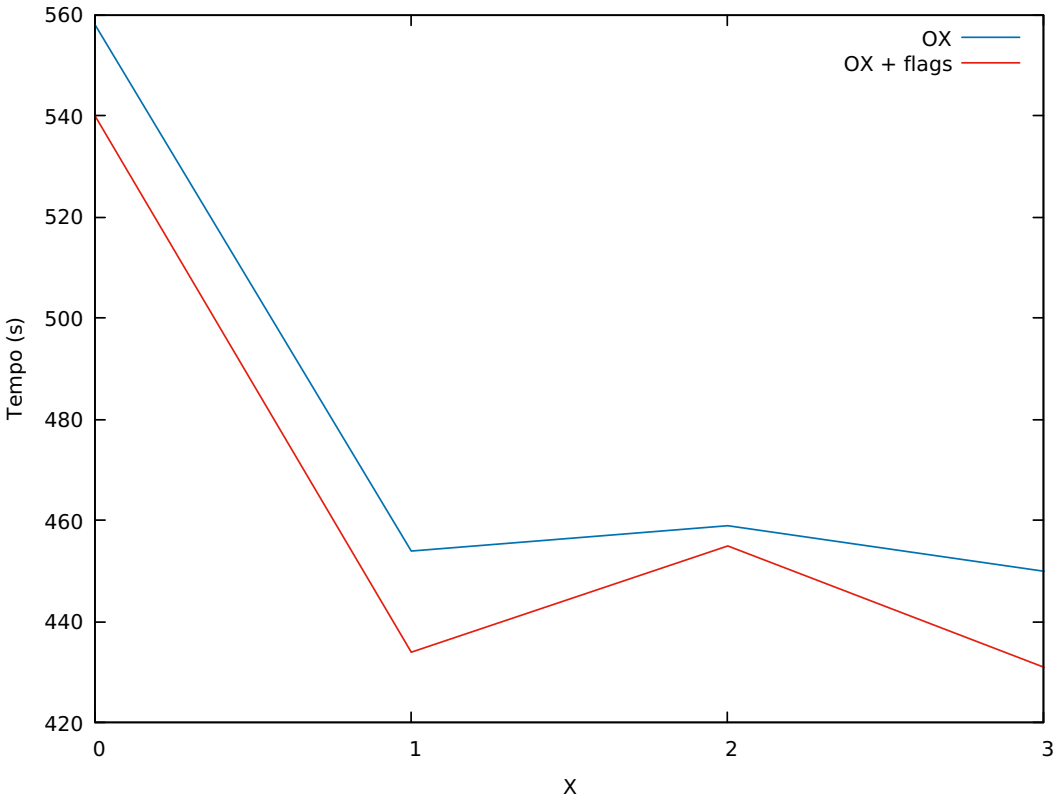

Figura 6 – Comparação geral entre o tempo pelo X

O padrão de formação do gráfico segue o mesmo obtido no LAB 107C: O pior caso acontece utilizando as flags **-O0**, depois **-O2**, seguido de **-O1** e o melhor caso ocorre utilizando **-O3**. Esse padrão se repete utilizando somente as flags **-OX** ou em conjunto com as demais flags. Dessa forma, o melhor tempo ocorre com o conjunto **-O3 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native** sendo de **431s**.

Dada a equação (5.1) e os tempos obtidos em cada compilação, pode-se calcular as eficiências usando o $T_{no-opt} = 563$s (tempo de execução do programa base). Isso está demonstrado em [7].

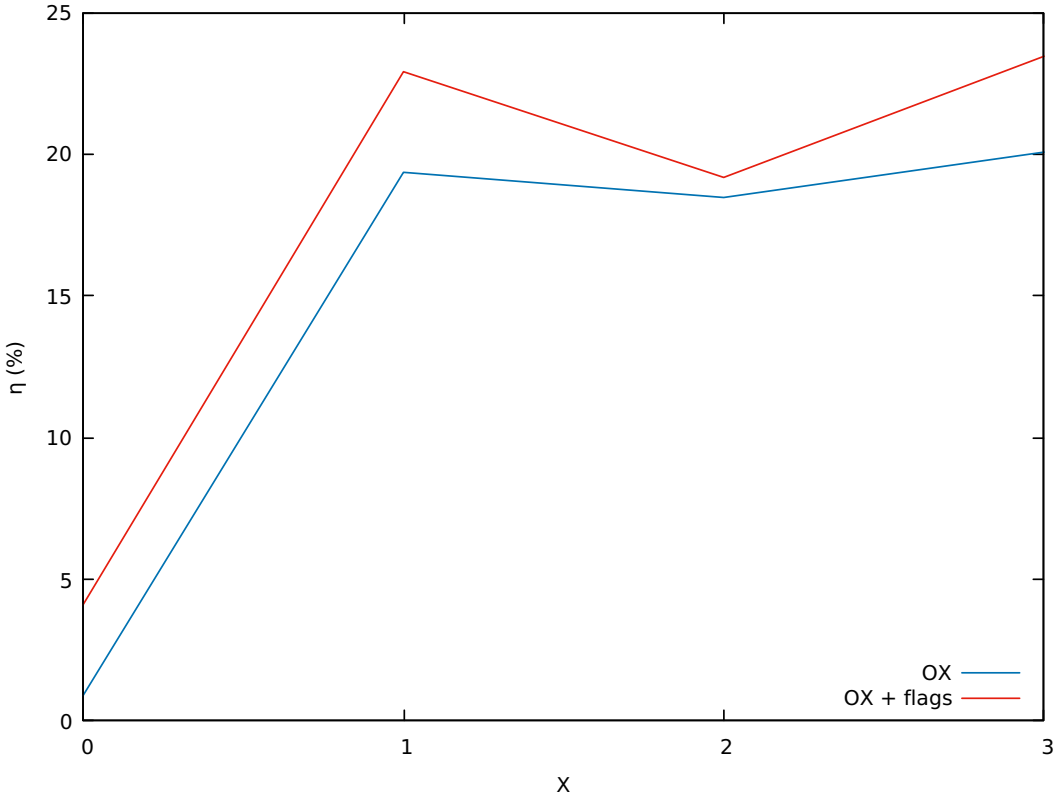

Figura 7 – Comparação geral entre a eficiência pelo X

Onde pode-se notar que, utilizando **-O0** as eficiências obtidas chegam próximo de 5% no máximo; com **-O1** ficam entre 19 - 23% aproximadamente; entre 18 - 20% com **-O2**; aproximadamente 20% utilizando somente **-O3** e 23% utilizando **-O3 + flags**, isto é, confirmando o resultado de que o melhor conjunto é **-O3 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native**. O resultado encontrado foi bem próximos do obtido anteriormente (as pequenas flutuações provavelmente ocorrem pela malha utilizada aqui ser menor.

LNCC - SequanaX

A tabela (5) mostra as configurações da máquina utilizada para a realização do benchmark.

Processador	Intel(R) Xeon(R) Gold 6152
Número de núcleos	24
Número de threads (Hyper-threading)	48
Clock Base	2.40GHz
Clock Turbo Máximo	3.70GHz
Cache L1	32KB
Cache L2	1MB
Cache L3	30MB
BoboMips	4200
Memória RAM	384GB DDR4

Tabela 5 – Configurações da máquina

As mesmas condições utilizadas no B710 foram aplicadas nesta máquina (pelo mesmo motivo anterior) e os conjuntos de flags também foram os mesmos.

Os tempos obtidos nas 8 rodadas estão na tabela [6].

Flags	0	1	2	3
OX	425s	361s	360s	357s
OX + Flags	411s	345s	356s	343s

Tabela 6 – Tempos obtidos com as compilações usando o gcc

Dados os tempos, pode-se comparar os resultados obtidos para cada compilação a fim de entender as diferenças variando as flags, isso está demonstrado em [8].

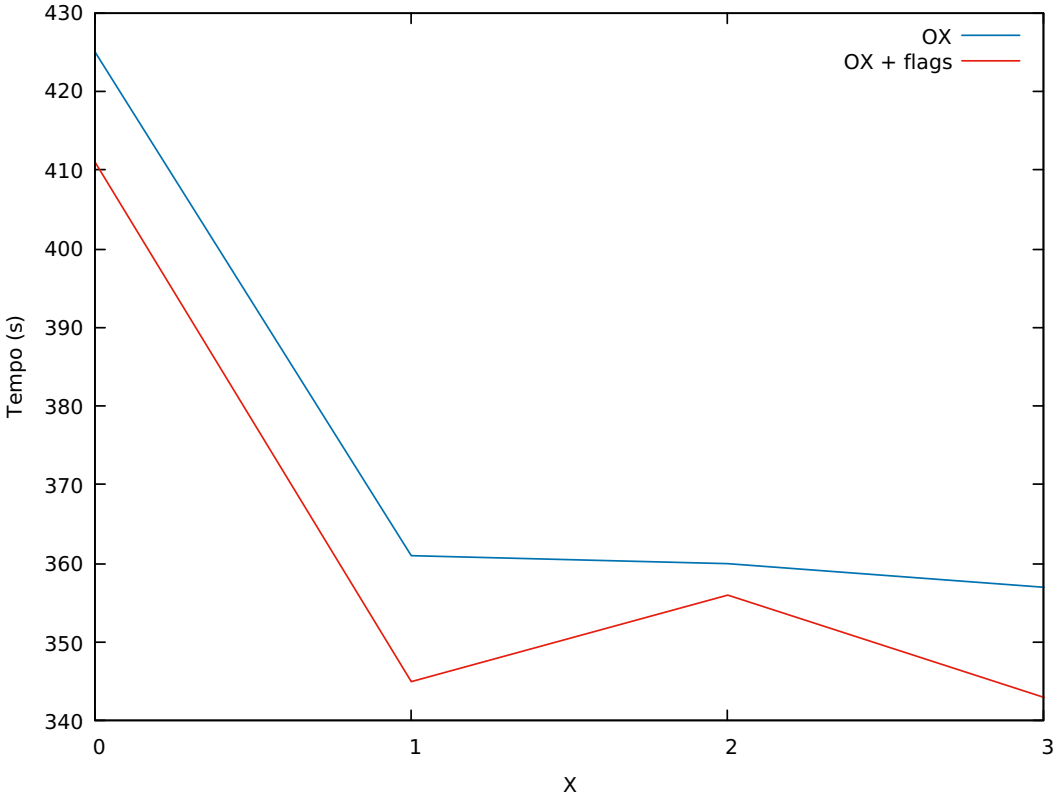

Figura 8 – Comparação geral entre o tempo pelo X

O padrão de formação do gráfico segue o mesmo obtido no LAB 107C e no nó B710 com exceção somente de uma coisa, nas compilações utilizando somente **-OX** o caso com **-O2** foi melhor que o **-O1**, um resultado que não aconteceu anteriormente. Independente disso, o melhor tempo continua com o conjunto **-O3 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native** sendo de **343s**.

Usando a equação (5.1) e com os tempos obtidos em cada compilação, pode-se calcular as eficiências usando o $T_{no-opt} = 448$s (tempo de execução do programa base). Isso está demonstrado em [9].

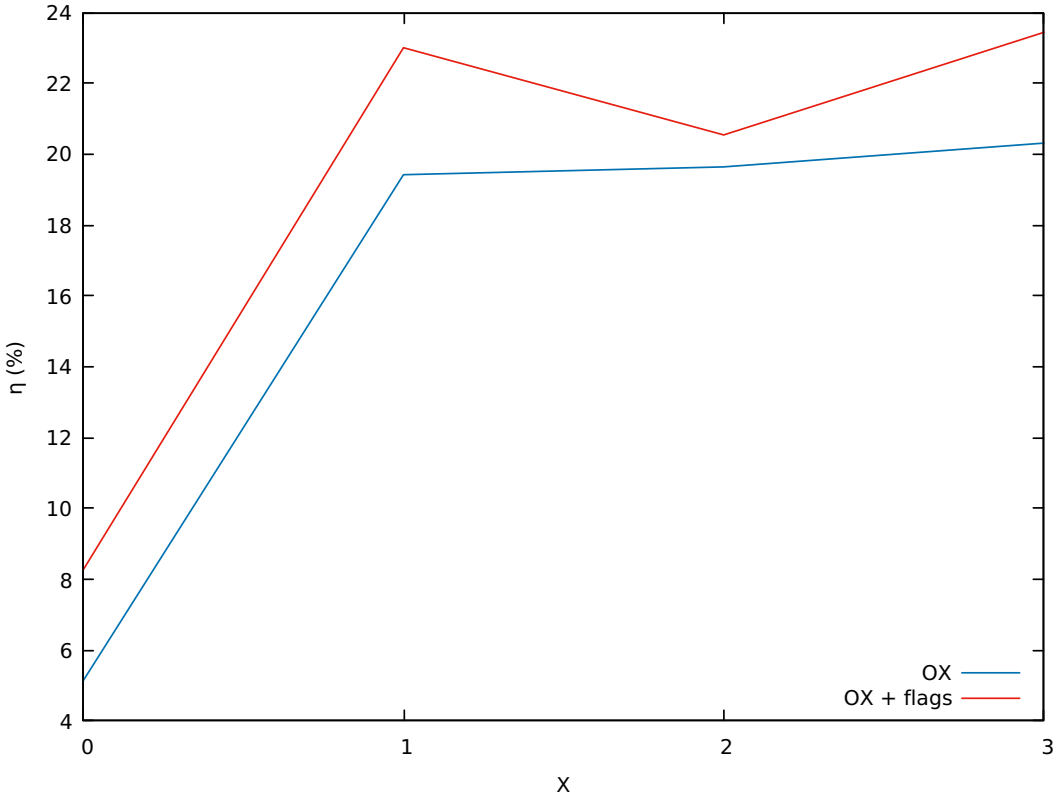

Figura 9 – Comparação geral entre a eficiência pelo X

Donde nota-se que, utilizando **-O0** as eficiências obtidas ficam entre 5 - 8%; com **-O1** ficam entre 19 - 23% aproximadamente; entre 19 - 21% com **-O2**; e, aproximadamente 20% utilizando somente **-O3** e 23% em conjunto com as demais flags.

5.1 Comparação entre as máquinas

Comparar os tempos obtidos com as compilações em cada máquina variando as flags não faz tanto sentido devido ao tamanho da malha ser bem diferente e isso gerar tempos com ordem de grandeza diferente. Mas, algo interessante a se comparar (apesar da tamanho da malha ser diferente) é a eficiência. Portanto, pegando os melhores casos (já que o padrão se repete pra todas máquinas), isto é, utilizando **-OX + -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native** pode-se fazer a comparação. Isso está demonstrado em [10].

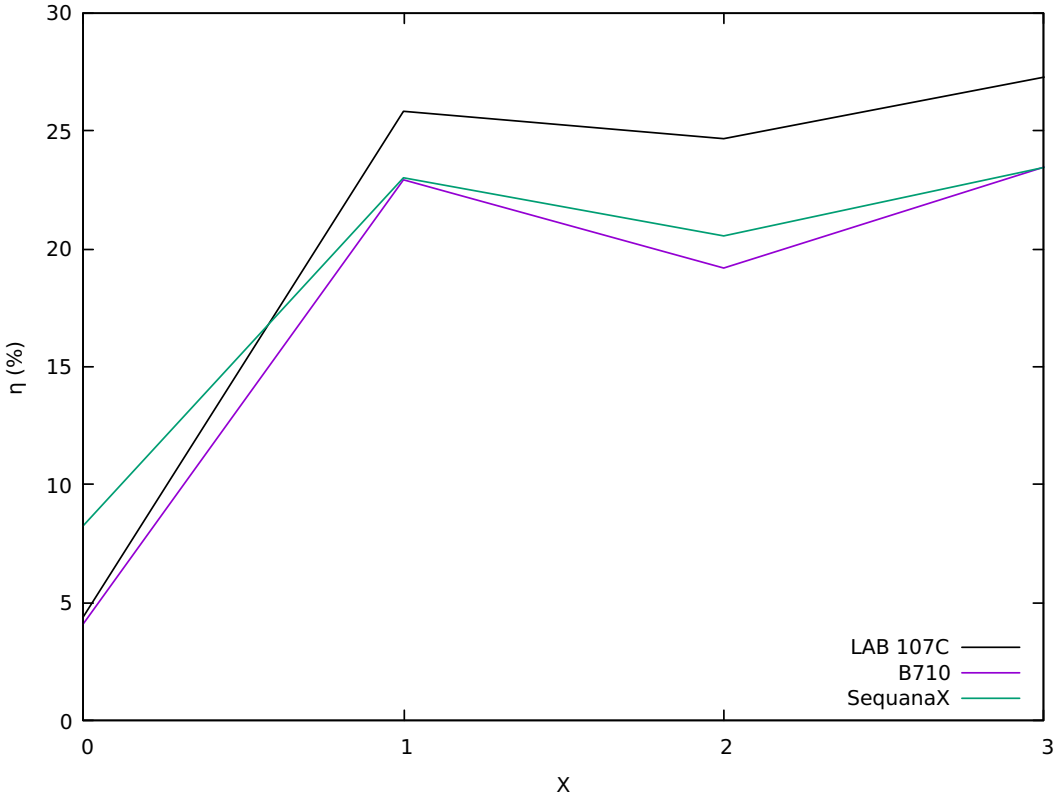

Figura 10 – Comparação das eficiências obtidas em diferentes máquinas

Aqui pode-se notar que, apesar das malhas utilizadas serem diferentes, as eficiências obtidas em cada caso foram bem próximas (em geral, não excedem 5%). Isso talvez sugere que caso a malha utilizada fosse a mesma, as eficiências ficassem mais próximas ainda (tal qual o caso do B710 com a SequanaX). Além disso, algo interessante a se notar é que o caso **-O0** na SequanaX se saiu bem melhor que os demais casos (quase o dobro).

Com os testes feitos em 3 máquinas, é possível inferir que, no geral, aplicar o caso **-O3 + flags** seja a melhor tentativa independente da máquina utilizada.

6 Profile

Realizar o profiling do algoritmo é importante para se entender os gargalos que ocorrem durante sua execução e encontrar os pontos que podem ser melhorados. O profile foi realizado na máquina do LAB 107C que tem suas configurações descritas em [1] e utilizando as mesmas condições já citadas anteriormente. Sendo assim, o resultado gerado após a aplicação da ferramenta de profiling no programa base é descrito na figura [11].

```
(base) [aluno@localhost ProfileCRU]$ gprof ising.x
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 55.22   3179.37  3179.37 686851584     0.00     0.00  hamiltoniana
 25.35   4638.77  1459.40 4968994816     0.00     0.00  testa_flip
 12.57   5362.32   723.55 4968994816     0.00     0.00  escolhe_pos
  6.86   5757.14   394.82                             main
  0.04   5759.55     2.42        46     0.05    29.80  equilibra
  0.02   5760.64     1.08         1     1.08     1.08  inicia_malha
  0.00   5760.68     0.04        46     0.00     0.00  magnetizacao_total
  0.00   5760.70     0.02        46     0.00     1.29  energia_total
```

Figura 11 – Profile programa base

Onde o tempo de execução do programa foi de **10480s**, que equivale a **2h:54m:40s**. Analisando o profile, pode-se notar que há quatro funções do programa que ocupam basicamente todo o tempo de execução: hamiltoniana, testa_flip, escolhe_pos e main. Sendo a hamiltoniana o maior gargalo, ocupando mais de 55% do tempo total. Essa informação será útil no processo de paralelização do algoritmo posteriormente.

7 Técnicas de Otimização de Software

Após o primeiro profiling, algumas técnicas de otimização de software foram aplicadas: Substituição de divisões por multiplicações (quando possíveis), substituição de chamadas da função *pow* por multiplicações e reservar memória corretamente para algumas variáveis. Isso está demonstrado nas figuras [12], [13], [14], [15], [16] e [17].

```
int main(){

    FILE *arq;
    int pos[2], dE=0;

    double J = 1;                                   //energia de interação
    double k = 1;                                   //constante de boltzman
    double B = 0;                                   //campo externo
    double T = 5;                                   //temperatura final
    double Tmin = 0.5;                              //temperatura inicial
    double dT = 0.1;                                //incremento na temperatura por iteração
    long unsigned int Msteps = 9600;                //número de iterações do método de Monte Carlo
    long unsigned int seed = 158235;                //semente para o gerador randômico
    double E=0, E2_media=0, E_media=0, E_total=0, E2_total=0;
    double M=0, M2_media=0, M_media=0, M_total=0, M2_total=0;
    double Mabs=0, Mabs_media=0, Mabs_total=0;
    double calor_esp = 0, susc_mag = 0;
```

Figura 12 – Variáveis base

```
int main(){

    FILE *arq;
    int i, j, pos[2], dE=0;
    int J = 1;                                      //energia de interação
    int k = 1;                                      //constante de boltzman
    int B = 0;                                      //campo externo
    double T = 5;                                   //temperatura final
    double Tmin = 0.5;                              //temperatura inicial
    double dT = 0.1;                                //incremento na temperatura por iteração
    unsigned int Msteps = 9600;                     //número de iterações do método de Monte Carlo
    unsigned int seed = 158235;                     //semente para o gerador randômico
    double E=0, E2_media=0, E_media=0, E_total=0, E2_total=0;
    double M=0, M2_media=0, M_media=0, M_total=0, M2_total=0;
    double Mabs_media=0, Mabs_total=0;
    double calor_esp = 0, susc_mag = 0;
```

Figura 13 – Variáveis modificadas

```
//loop do Monte Carlo
for(int i=1;i<=Msteps;i++){
    //loop de Metropolis
    for(int j=1;j<=N;j++){
        escolhe_pos(pos, r);

        if(testa_flip(pos, &dE, r, J, B, k, T)){
            //ajusta os observáveis
            E+=2*dE;
            M+=2*Rede[pos[0]][pos[1]];
        }
    }

    //soma dos observavéis
    E_total+=E;
    E2_total+= pow(E,2);
    M_total+=M;
    M2_total+=pow(M,2);
    Mabs_total+=abs(M);
}
```

Figura 14 – Chamadas da função *pow*

```
//loop do Monte Carlo
for(i=1;i<=Msteps;i++){
    //loop de Metropolis
    for(j=1;j<=N;j++){
        escolhe_pos(pos, r);

        if(testa_flip(pos, &dE, r, J, B, k, T)){
            //ajusta os observáveis
            E+=2*dE;
            M+=2*Rede[pos[0]][pos[1]];
        }
    }

    //soma dos observavéis
    E_total+=E;
    E2_total+= E*E;
    M_total+=M;
    M2_total+= M*M;
    Mabs_total+=abs(M);
}
```

Figura 15 – Sem chamadas da função *pow*

```
    //média dos observáveis
    E_media=E_total/(Msteps*N*2);                          //<E> - fator 1/2 pela cont
    E2_media=E2_total/(Msteps*N*4);                        //<E²>  - fator 1/4 idem (1
    M_media=M_total/(Msteps*N);                            //<M>
    M2_media=M2_total/(Msteps*N);                          //<M²>
    Mabs_media=Mabs_total/(Msteps*N);                      //<|M|>
    calor_esp = (E2_media-(pow(E_media,2)*N))/(k*pow(T,2));  //C_v = (<E²> - <E>²*N)/(k*
    susc_mag = (M2_media-(pow(M_media,2)*N))/(k*T);          //X = (<M²> - <M>²*N)/(k*T)

    fprintf(arq, "%.2lf %lf %lf %lf %lf\n", T, Mabs_media, E_media, calor_esp, susc_mag);
}
```

Figura 16 – Fazendo divisões

```
    //média dos observáveis
    E_media=(E_total/(Msteps*N))*0.5;                      //<E> - fator 1/2 pela cont
    E2_media=(E2_total/(Msteps*N))*0.25;                   //<E²>  - fator 1/4 idem (1
    M_media=M_total/(Msteps*N);                            //<M>
    M2_media=M2_total/(Msteps*N);                          //<M²>
    Mabs_media=Mabs_total/(Msteps*N);                      //<|M|>
    calor_esp = (E2_media-(E_media*E_media*N))/(k*T*T);    //C_v = (<E²> - <E>²*N)/(k*
    susc_mag = (M2_media-(M_media*M_media*N))/(k*T);       //X = (<M²> - <M>²*N)/(k*T)

    fprintf(arq, "%.2lf %lf %lf %lf %lf\n", T, Mabs_media, E_media, calor_esp, susc_mag);
}
```

Figura 17 – Fazendo multiplicações

Dessa forma, o novo resultado da ferramente de profiling é descrito na figura [18].

```
(base) [aluno@localhost ProfileOTI]$ gprof ising.x
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 54.02   3016.30  3016.30 686851584     0.00     0.00  hamiltoniana
 26.81   4513.25  1496.95 4968994816     0.00     0.00  testa_flip
 13.31   5256.72   743.47 4968994816     0.00     0.00  escolhe_pos
  6.06   5595.22   338.50                             main
  0.03   5596.72     1.50       46     0.03    29.21  equilibra
  0.01   5597.19     0.47        1     0.47     0.47  inicia_malha
  0.00   5597.22     0.03       46     0.00     1.22  energia_total
  0.00   5597.24     0.02       46     0.00     0.00  magnetizacao_total
```

Figura 18 – Profile programa otimizado

O tempo de execução do programa foi de **10039s**, que equivale a **2h:47m:19s**. Nota-se pelo profile que não ocorrem muitas mudanças, as funções hamiltoniana, testa_flip, escolhe_pos e main ainda ocupam praticamente todo o tempo de execução e a hamiltoniana continua sendo o maior gargalo, ocupando mais de 54% do tempo total.

8 Comparação do Programa Base vs Programa Otimizado

Comparando o primeiro profiling (programa base) e o segundo profiling (programa otimizado) pode-se notar que houve uma pequena diminuição do tempo nas funções hamiltoniana, testa_flip e main e um pequeno aumento nas demais funções. De qualquer forma, ao aplicarmos as técnicas de otimização obtivemos um resultado positivo: o tempo de execução diminuiu de **10480s** para **10039s**, o que corresponde a uma eficiência de aproximadamente **4.2**% (cerca de **8m** a menos no tempo de execução usando a malha com $L = 528$). A eficiência não é tão expressiva, mas se notarmos que essa melhora se deu apenas com simples modificações no algoritmo, aplicar as boas práticas de programação é uma maneira rápida de obter um ganho no tempo de execução.

9 Implementação do Programa com OpenMP

Ao analisar o profile vimos que há três funções que ocupam a maior parte do tempo de execução do algoritmo, então elas devem ser o foco da paralelização. O problema é: ao checar as funções, como mostra a figura [19], é fácil que notar que não há nada a ser paralelizado nelas.

```
//escolhe uma a posição aleatória de um spin
void escolhe_pos(int *p, gsl_rng *r){

    p[0]=(gsl_rng_uniform(r)*(L));
    p[1]=(gsl_rng_uniform(r)*(L));
}

//calcula a energia da posição escolhida
int hamiltoniana(int *p, int J, int B, int Rede[L][L]){

    int dir, esq, baixo, cima, E, bordas;

    //condições de contorno 2D
    dir=(p[0]+1)%L;
    esq=(p[0]+L-1)%L;
    cima=(p[1]+1)%L;
    baixo=(p[1]+L-1)%L;

    bordas = Rede[esq][p[1]]+Rede[dir][p[1]]+Rede[p[0]][cima]+Rede[p[0]][baixo];

    E = -J*Rede[p[0]][p[1]]*bordas - B*Rede[p[0]][p[1]];

    return E;
}

//decide se o spin da posição flipa ou não (com base na energia ou no número aleatório gerado)
int testa_flip(int *p, int *dE, gsl_rng *r, int J, int B, int k, double T, int Rede[L][L]){

    *dE=-2*hamiltoniana(p, J, B, Rede);

    if(*dE < 0 || (gsl_rng_uniform(r) <= exp(-(*dE)/(k*T)))){
        Rede[p[0]][p[1]]*=-1;
        return 1;
    }
    else
        return 0;
}
```

Figura 19 – Funções com maior participação no tempo de execução

Então pode-se concluir algo interessante: o motivo delas ocuparem a maior parte no tempo de execução é porquê elas são chamadas várias e várias vezes no algoritmo. Entender isso é importante porquê daqui é possível concluir que é necessário paralelizar então o(s) loop(s) que chamam essas funções e eles estão localizados na função main, como mostra a figura [20].

```
//loop da temperatura
for(;T>=Tmin;T-=dT)
{
    //termalização
    equilibra(r, J, B, k, T);

    //observáveis com valores no equilíbrio
    M = magnetizacao_total();
    E = energia_total(J, B);

    E_total=0;
    E2_total=0;
    M_total=0;
    M2_total=0;
    Mabs_total=0;

    //loop do Monte Carlo
    for(i=1;i<=Msteps;i++){
        //loop de Metropolis
        for(j=1;j<=N;j++){
            escolhe_pos(pos, r);

            if(testa_flip(pos, &dE, r, J, B, k, T)){
                //ajusta os observáveis
                E+=2*dE;
                M+=2*Rede[pos[0]][pos[1]];
            }
        }

        //soma dos observavéis
        E_total+=E;
        E2_total+= E*E;
        M_total+=M;
        M2_total+= M*M;
        Mabs_total+=abs(M);
    }

    //média dos observáveis
    E_media=(E_total/(Msteps*N))*0.5;                    //<E> - fator 1/2 pela contagem
    E2_media=(E2_total/(Msteps*N))*0.25;                 //<E²>  - fator 1/4 idem (1/2*1/
    M_media=M_total/(Msteps*N);                          //<M>
    M2_media=M2_total/(Msteps*N);                        //<M²>
    Mabs_media=Mabs_total/(Msteps*N);                    //<|M|>
    calor_esp = (E2_media-(E_media*E_media*N))/(k*T*T);  //C_v = (<E²> - <E>²*N)/(k*T²) |
    susc_mag = (M2_media-(M_media*M_media*N))/(k*T);     //X = (<M²> - <M>²*N)/(k*T) | <M

    fprintf(arq, "%.2lf %lf %lf %lf %lf\n", T, Mabs_media, E_media, calor_esp, susc_mag);
}
```

Figura 20 – Recorte da função main

Nesse ponto é necessário decidir em qual loop realizar a paralelização (já que temos 3 loops aninhados). O loop da temperatura foi logo descartado, uma vez que suas iterações são não inteiras e o OpenMP não trabalha com esse tipo de iteração. Dessa forma, restam apenas dois loops: o loop de Monte Carlo e o loop de Metropolis. Aqui a decisão é tomada com base no tempo que abrir seções paralelas demandam, ou seja, se paralelizar o loop mais interno (metropolis) as seções paralelas serão abertas mais vezes, o que deixaria o algoritmo mais lento. Por esse motivo, a paralelização foi feita no loop de Monte Carlo, isto é, o loop do meio.

Foi comentado anteriormente que utilizar $L = 528$ e $N_{MC} = 9600$ tinha uma explicação e é justamente nesse ponto que ela se justifica: os números 528 e 9600 dividem perfeitamente por 8, 24 e 48 (as maiores quantidades de threads disponíveis para os testes). Como ainda não era certo qual loop seria paralelizado no início, os dois números de iterações foram escolhidas de forma consciente para ser uma divisão perfeita em qualquer

um dos casos.

Uma vez escolhido qual loop a paralelização deve ser feita, discutiremos como fica a divisão da malha. O funcionamento do loop de Monte Carlo pode ser melhor entendido pela figura [21].

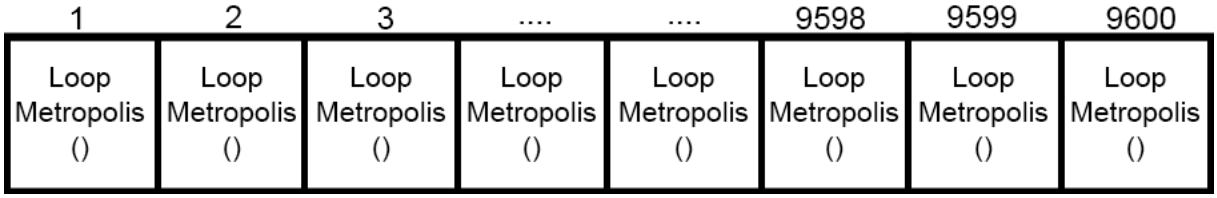

Figura 21 – Funcionamento do loop de Monte Carlo

Isto é, durante cada iteração do loop de Monte Carlo (nesse caso, 9600), o loop de Metropolis é chamado. Pegando como exemplo a paralelização feita em uma máquina de 8 threads, teríamos a divisão como mostra a figura [22].

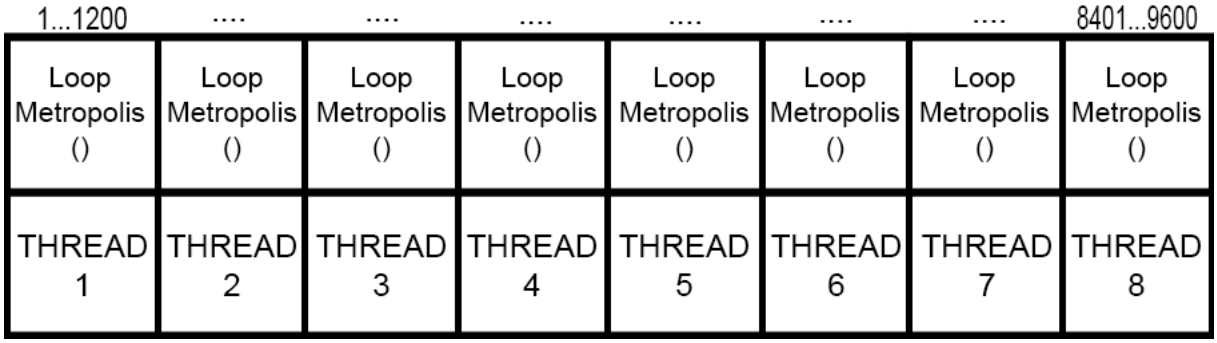

Figura 22 – Divisão da malha para uma máquina de 8 threads

Para a realização da paralelização com o OpenMP a diretiva utilizada foi *#pragma omp parallel for* aplicado no loop de Monte Carlo para realizar a paralelização do mesmo. Para manter o funcionamento coerente foi preciso definir algumas váriaveis como *shared, private* e *firstprivate* além de usar a diretiva *reduction(+:)*. Para a divisão das iterações foi utilizada a distribuição *schedule (dynamic)* com *chunk* = 1200 (para 8 threads).

shared: variáveis compartilhadas por todas as threads.
private: variáveis não compartilhadas e sem valor inicial.
firstprivate: variáveis não compartilhadas e com valor inicial.
reduction(+:): cada thread cria uma cópia local da variável solicitada que será somada após as iterações acabarem gerando uma única variável.
chunk: número de iterações que cada thread irá executar.
dynamic: número de iterações será atribuído dinâmicamente para uma thread caso ela já tenha terminando suas iterações.

O valor do *chunk* irá variar conforme o número de threads máxima disponível, por exemplo, para 48 threads ele seria de 200 e para 24 threads seria de 400.

A parte do código de interesse após a paralelização é mostrado na figura [23].

```
//loop da temperatura
for(;T>=Tmin;T-=dT)
{
    //termalização
    equilibra(r, J, B, k, T, Rede);

    //observáveis com valores no equilíbrio
    M = magnetizacao_total(Rede);
    E = energia_total(J, B, Rede);

    E_total=0;
    E2_total=0;
    M_total=0;
    M2_total=0;
    Mabs_total=0;

    #pragma omp parallel for schedule(dynamic, 1200) private(dE, pos) \
    shared(J, B, k, T) firstprivate(r, E, M, Rede) reduction(+: E_total, E2_total, M_total, M2_total, Mabs_total)
        //loop do Monte Carlo
        for(i=1;i<=Msteps;i++){
            //loop de Metropolis
            for(j=1;j<=N;j++){
                escolhe_pos(pos, r);

                if(testa_flip(pos, &dE, r, J, B, k, T, Rede)){
                    //ajusta os observáveis
                    E+=2*dE;
                    M+=2*Rede[pos[0]][pos[1]];
                }
            }

            //soma dos observavéis
            E_total+=E;
            E2_total+= E*E;
            M_total+=M;
            M2_total+= M*M;
            Mabs_total+=abs(M);
        }

    //média dos observáveis
    E_media=(E_total/(Msteps*N))*0.5;                    //<E> - fator 1/2 pela contagem dupla dos pares
    E2_media=(E2_total/(Msteps*N))*0.25;                 //<E²>  - fator 1/4 idem (1/2*1/2)
    M_media=M_total/(Msteps*N);                          //<M>
    M2_media=M2_total/(Msteps*N);                        //<M²>
    Mabs_media=Mabs_total/(Msteps*N);                    //<|M|>
    calor_esp = (E2_media-(E_media*E_media*N))/(k*T*T);  //C_v = (<E²> - <E>²*N)/(k*T²) | <E>² multiplicado
    susc_mag = (M2_media-(M_media*M_media*N))/(k*T);     //X = (<M²> - <M>²*N)/(k*T) | <M>² multiplicado N p

    fprintf(arq, "%.2lf %lf %lf %lf %lf\n", T, Mabs_media, E_media, calor_esp, susc_mag);
}
```

Figura 23 – Código paralelizado (recorte da função main)

OBS: Nesse ponto do trabalho uma alteração foi feita para a paralelização ser possível: a variável **Rede** (a matriz do sistema) era global e isso gerou problemas na paralelização, então ela foi definida localmente na função main.

10 Benchmark do Programa com Várias Threads

10.1 Lab107C

Para fazer a análise do ganho de desempenho no tempo de execução do algoritmo paralelizado as mesmas condições iniciais utilizadas anteriormente foram mantidas. Vale ressaltar aqui que o *chunk* definido para a análise no LAB 107C foi de 1200 (devido ao número de threads máximo ser 8). Além disso, a análise foi feita executando o programa sem nenhuma flag e com o melhor conjunto de flags descoberto no benchmark anterior.

Os tempos obtidos variando o número de threads estão na tabela [7].

Threads	8	7	6	5	4	3	2	1
Sem flags	3982s	5008s	5395s	5811s	6142s	7029s	7673s	10042s
Melhores flags	2763s	4165s	4624s	4808s	5057s	5725s	6934s	7662s

Tabela 7 – Tempos obtidos variando o número de threads

Pode-se comparar os tempos obtidos a fim de entender as diferenças variando o número de threads e isso está demonstrado em (24).

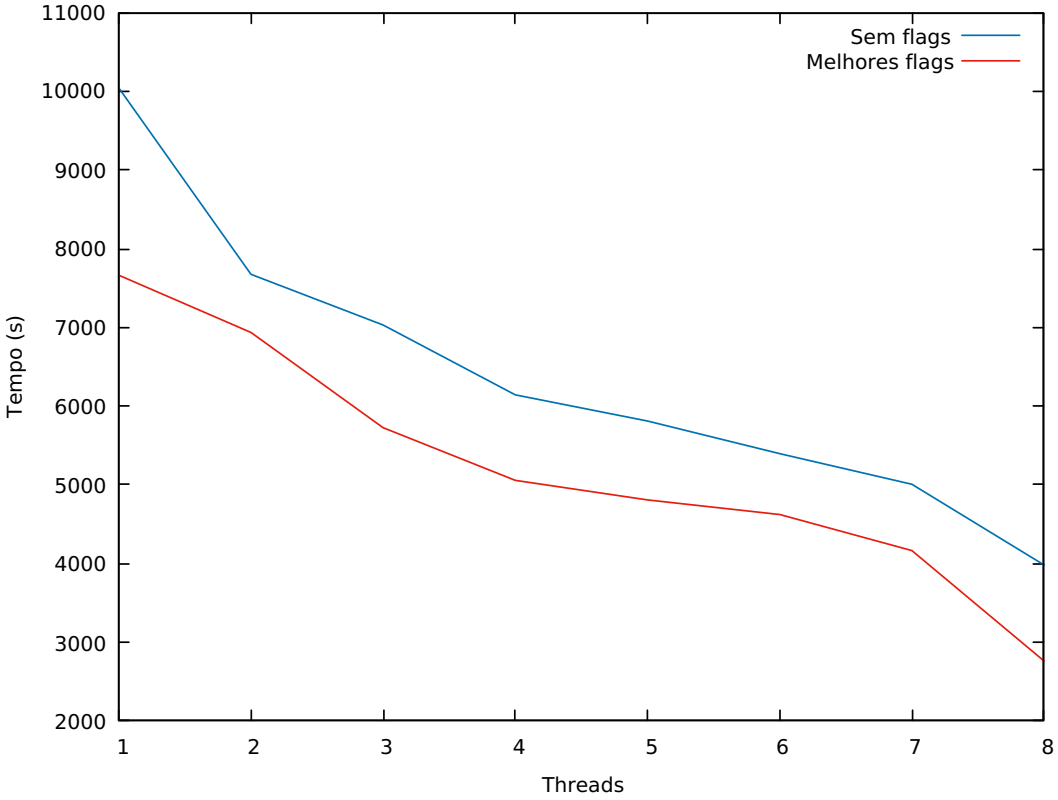

Figura 24 – Tempo pelo número de threads

Onde nota-se um padrão bem interessante: Conforme o número de threads aumenta, o tempo de execução sempre diminui. Esse padrão acontece tanto nos casos sem flags quanto nos casos com as melhores flags.

Afim de entender essa melhora no tempo de execução conforme o número de threads aumenta, pode-se calcular a eficiência usando a equação (5.1) e usando $T_{no-opt} = 10480s$ (tempo de execução do programa base). Isso está descrito em (25).

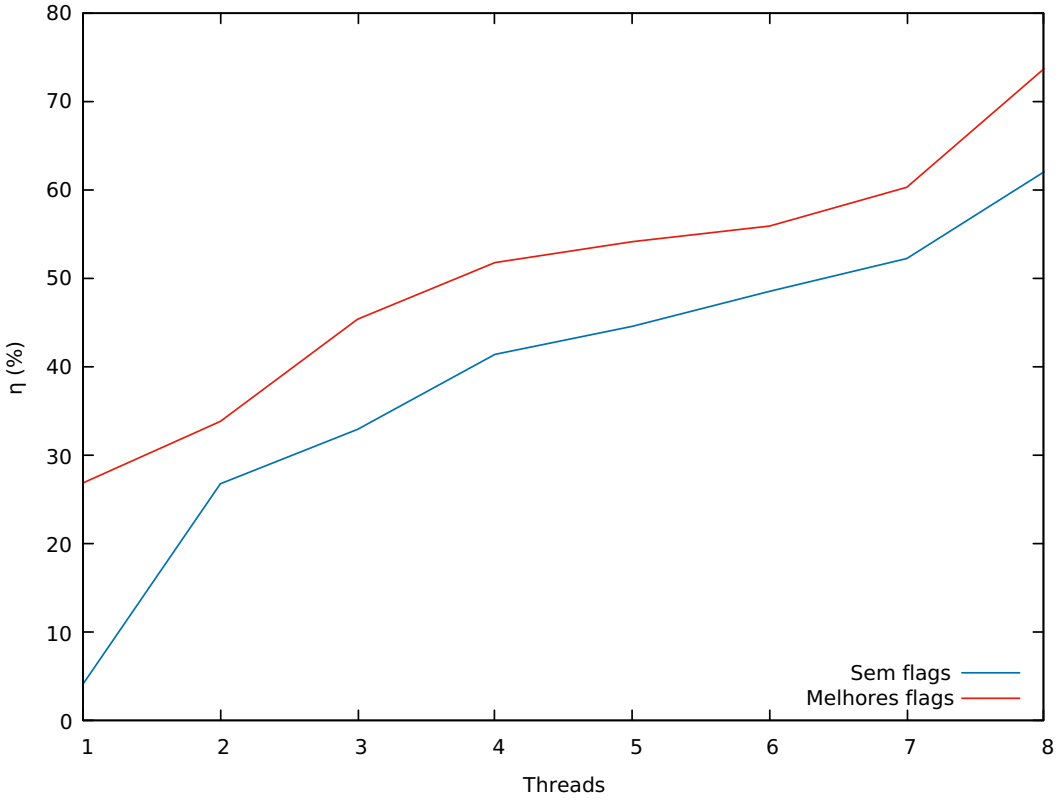

Figura 25 – Eficiência pelo número de threads

A primeira vista o gráfico pode parecer incorreto já que nos casos utilizando 1 thread já existe ganho na eficiência (o que não deveria ocorrer) mas a explicação é simples: O algoritmo utilizado na execução é o algoritmo com as boas práticas de programação aplicadas e o tempo de referência é o tempo do algoritmo base (que foi usado como referência em todas as outras análises) e por isso o ganho no tempo mesmo utilizando 1 thread. Se compararmos o caso com 1 thread com o tempo de execução do algoritmo com as boas práticas aplicadas percebe-se uma perca de eficiência (o que é natural devido ao tempo necessário para se abrir seções paralelas). A escolha da referência do tempo como o tempo do algoritmo base foi pra manter coerência com as demais análises feitas anteriormente.

Mas, analisando as eficiências percebe-se que, no pior caso (1 thread) já existe um ganho de desempenho de quase 30% utilizando as melhores flags e essa eficiência só aumenta conforme o número de threads também aumenta, chegando, nos melhores casos (com 8 threads), em 62% sem flags e 74% com as melhores flags. Isso significa que o tempo do programa do programa reduz em praticamente 3/4 aplicando todas as técnicas de otimização, o que é um resultado muito bom.

Após a análise da eficiência, pode-se analisar a aceleração por thread (normalizada com o valor de 1 thread), esse resultado pode ser visto em (26).

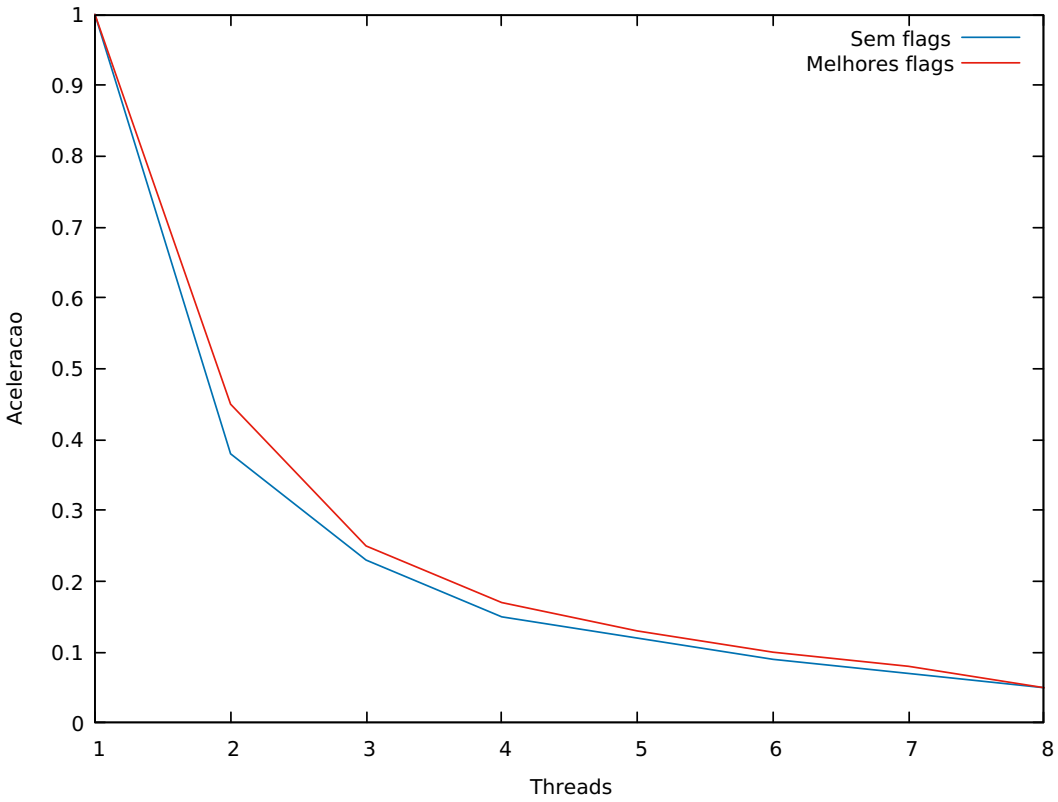

Figura 26 – Aceleração por thread (normalizada)

Aqui pode-se notar que o comportamento é bem semelhante a uma exponencial (em ambos casos) e isso demonstra algo interessante: conforme o número de threads aumenta, a aceleração individual das threads perde potência, o que demonstra que após um valor X de threads, isso estaria bem próximo de zero e teríamos a saturação do programa. Isto é, quanto mais threads são utilizados menor é a parcela individual no quesito aceleração do tempo. Por exemplo, rodando com 8 threads obtivemos o melhor tempo de execução, porém, em questão de aceleração, utilizando 8 threads cada thread seria responsável por acelerar cerca de 0.08% do tempo, em comparação, no caso utilizando 2 threads cada thread é responsável por acelerar cerca de 0.12%. Ou seja, apesar do tempo utilizando 2 threads ser maior que o tempo utilizando 8, a "importância"que cada thread tem é maior no caso utilizando 2 threads.

Pode-se notar que com 8 threads esse valor já está bem próximo a zero, isso demonstra que provavelmente não demoraria muito para o algoritmo saturar se aumentássemos mais o número de threads (provavelmente algo em torno de 16 - 24).

Por fim, pode-se avaliar o *speedup* que o programa possui aumentando o número de threads. O cálculo do *speedup* foi feito da seguinte forma

$$S = \frac{T_s}{T_p}$$

onde T_s é o tempo obtido na execução em serial (programa base) e T_p é o tempo obtido em paralelo.

O resultado obtido pode ser visto em (27).

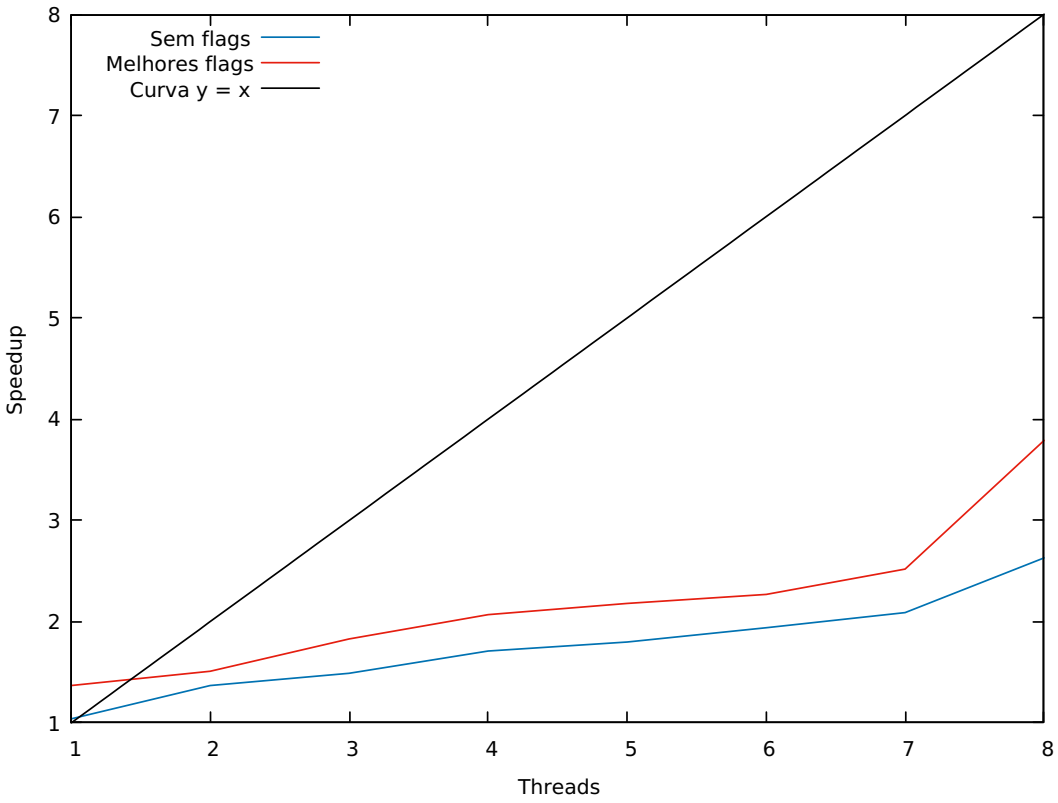

Figura 27 – *Speedup* por thread

A função $f(x) = x$ foi colocada no gráfico como referência e o motivo é que é natural pensar que conforme aumentamos o número de threads, o tempo seria diminuído por N, isto é, se inicialmente o programa foi executado em serial no tempo X, utilizando N threads é de esperar que o tempo de execução seja de $T = \frac{X}{N}$ e isso é claramente falso como podemos ver no gráfico anterior. O ganho de tempo está totalmente longe de ser linear e se ultrassa a função $f(x) = x$ no caso com 1 thread. Logo, esse pensamento, apesar de ser algo natural de se acontecer, está equivocado (pelo menos na maioria das vezes).

10.2 LNCC - B710

A execução no B710 também foi realizada mantendo as condições anteriores e utilizando a mesma lógica com as flags. Como o número de threads máximo era 24, o *chunk* foi definido em 400.

Os tempos obtidos variando o número de threads estão na tabela [8].

Threads	1	2	4	6	8	12	16	20	24
Sem flags	562s	648s	1100s	965s	1015s	973s	1067s	938s	940s
Melhores flags	545s	628s	1067s	936s	985s	944s	1035s	902s	920s

Tabela 8 – Tempos obtidos variando o número de threads

A comparação gráfica pode ser vista em [28].

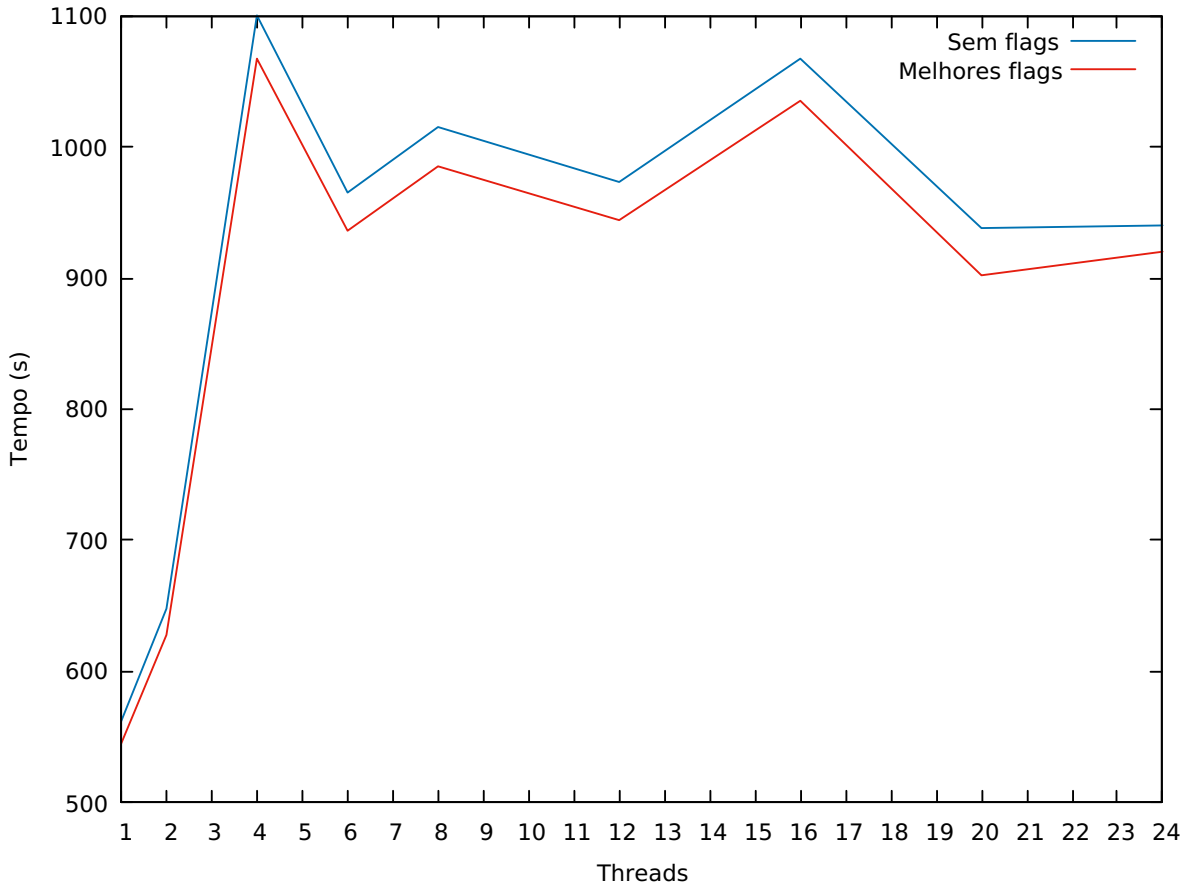

Figura 28 – Tempo pelo número de threads

Onde pode-se notar que o comportamento não faz sentido algum. Isso acontece devido a problemas de compatibilidade do algoritmo com o cluster utilizado cuja solução não foi possível de ser encontrada. O tempo deveria diminuir conforme o número de threads aumenta (até um certo ponto), conforme checado anteriormente no LAB 107C e

isso não ocorreu aqui. Por esse motivo, a análise do B710 se encerra aqui, já que continuar com os demais tópicos não agrega em nada.

10.3 LNCC - SequanaX

Assim como nas outras máquinas, as condições iniciais também foram mantidas e a utilização das flags se deu da mesma forma. Sendo assim, a única diferença é que o *chunk* foi definido em 200 já que o número de threads máximo era de 48.

Os tempos obtidos variando o número de threads estão na tabela [9].

Threads	Sem flags	Melhores flags
1	446s	437s
2	582s	564s
4	1060s	1028s
6	767s	751s
8	903s	883s
12	676s	649s
16	687s	660s
20	687s	662s
24	714s	680s
28	682s	655s
32	826s	795s
36	708s	666s
40	836s	810s
44	866s	848s
48	955s	917s

Tabela 9 – Tempos obtidos variando o número de threads

A comparação gráfica pode ser vista em [29].

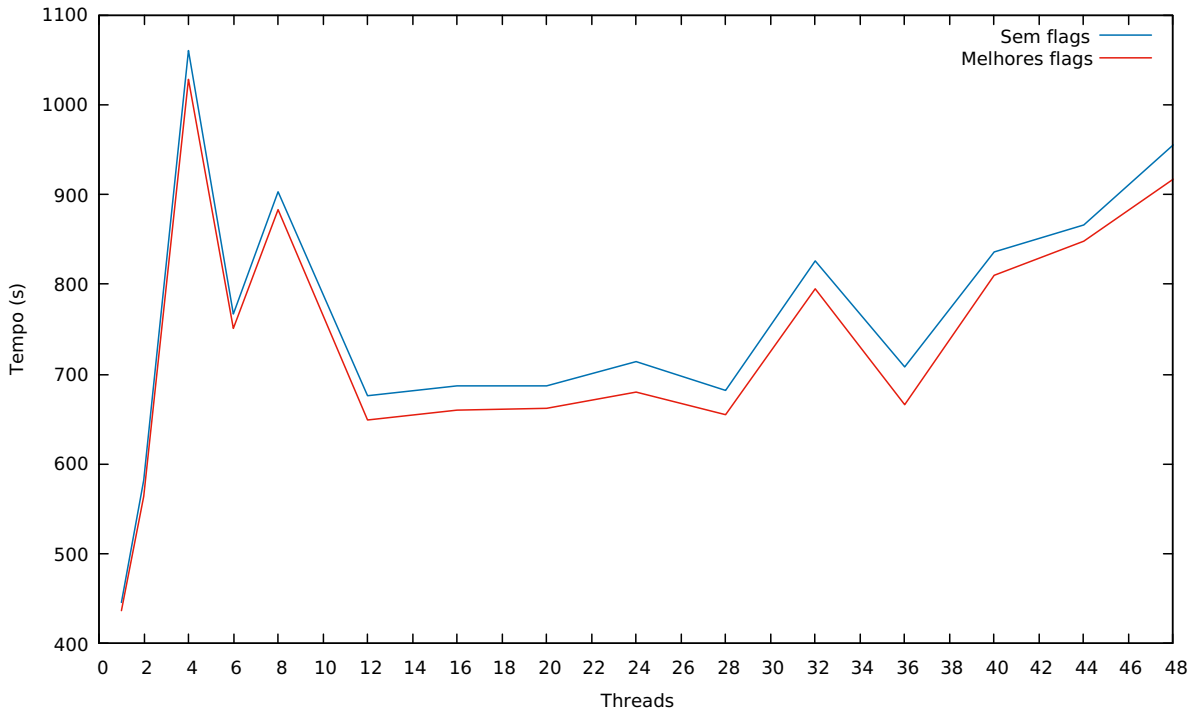

Figura 29 – Tempo pelo número de threads

Onde é possível notar que o mesmo problema que aconteceu na máquina B710 se repete na SequanaX e a análise também ficará limitada até aqui já que não faz sentido continuar.

11 Validação dos Resultados

O modelo nos fornece quatro observáveis: Magnetização absoluta média por spin, energia média por spin, calor específico por spin e susceptibilidade magnética por spin.

De acordo com (COSTA, 2006) e substituindo os valores iniciais utilizados, o calor específico por spin é obtido através de

$$c = \frac{1}{NT^2}\left[\left\langle E^2\right\rangle - \langle E\rangle^2\right] \tag{11.1}$$

A susceptibilidade magnética por spin através de

$$\chi = \frac{1}{NT}\left[\left\langle M^2\right\rangle - \langle M\rangle^2\right] \tag{11.2}$$

E a temperatura crítica por

$$T_c = \frac{2}{\ln(1+\sqrt{2})} = 2.2691853 \tag{11.3}$$

Como o modelo utilizado possui solução analítica, pode-se checar a eficiência do modelo usando essa solução. Para uma rede com L = 2 temos $N = 4$ spins e 16 microestados possíveis.

De acordo com (SANTOS, 2014) e substituindo os valores iniciais utilizados, a solução analítica dos observáveis para rede com L=2 são dadas por:

A magnetização absoluta média por spin

$$\langle|m|\rangle = \frac{2+\exp(8/T)}{6+2\cosh(8/T)} \tag{11.4}$$

onde $\langle|m|\rangle = \dfrac{\langle|M|\rangle}{N}$.

A energia média por spin

$$\langle e\rangle = -\frac{2\sinh(8/T)}{3+\cosh(8/T)} \tag{11.5}$$

onde $\langle e\rangle = \dfrac{\langle E\rangle}{N}$.

Como o calor específico e a susceptibilidade magnética dependem de E e M respectivamente, basta analisar se os resultados obtidos para E e M estão de acordo com o esperado.

Logo, podemos analisar inicialmente os resultados obtidos utilizando a versão em serial e comparar com a solução analítica e posteriormente checar se os resultados se mantém para a versão em paralelo.

11.1 Serial

Então, para a versão em serial o resultado obtido plotando a curva descrita pela solução analítica e o resultado numérico é demonstrado nos gráficos [30] e [31].

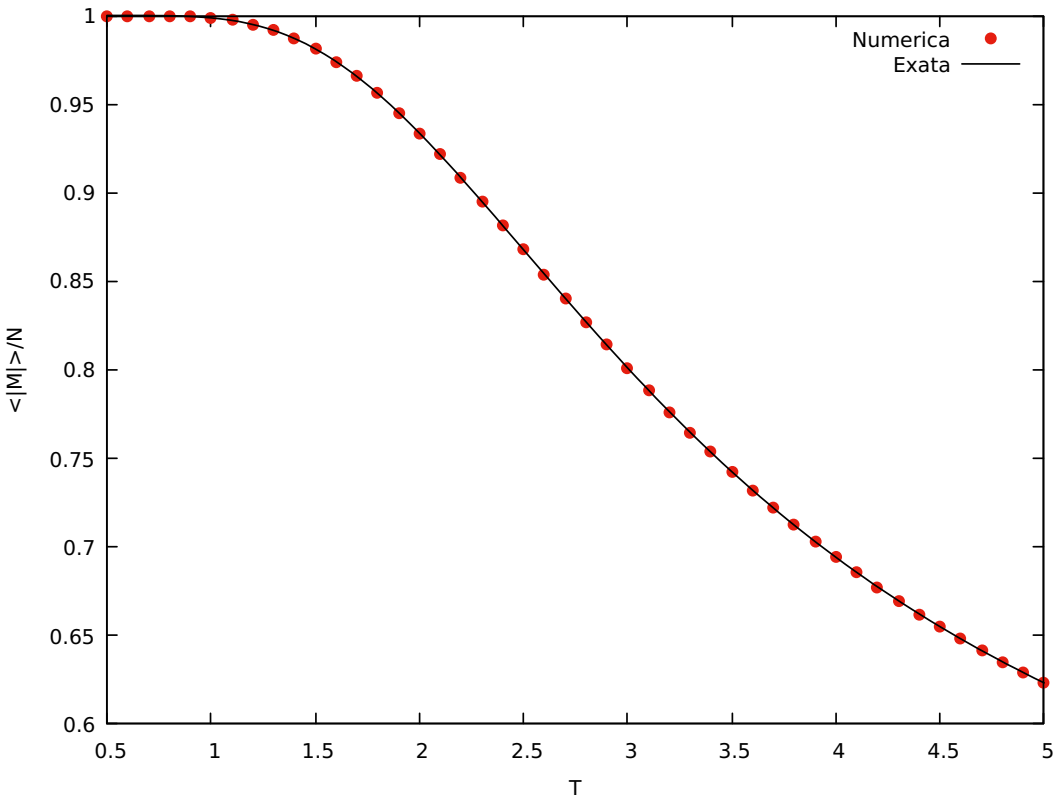

Figura 30 – Magnetização absoluta média por spin vs Temperatura - Solução numérica e analítica

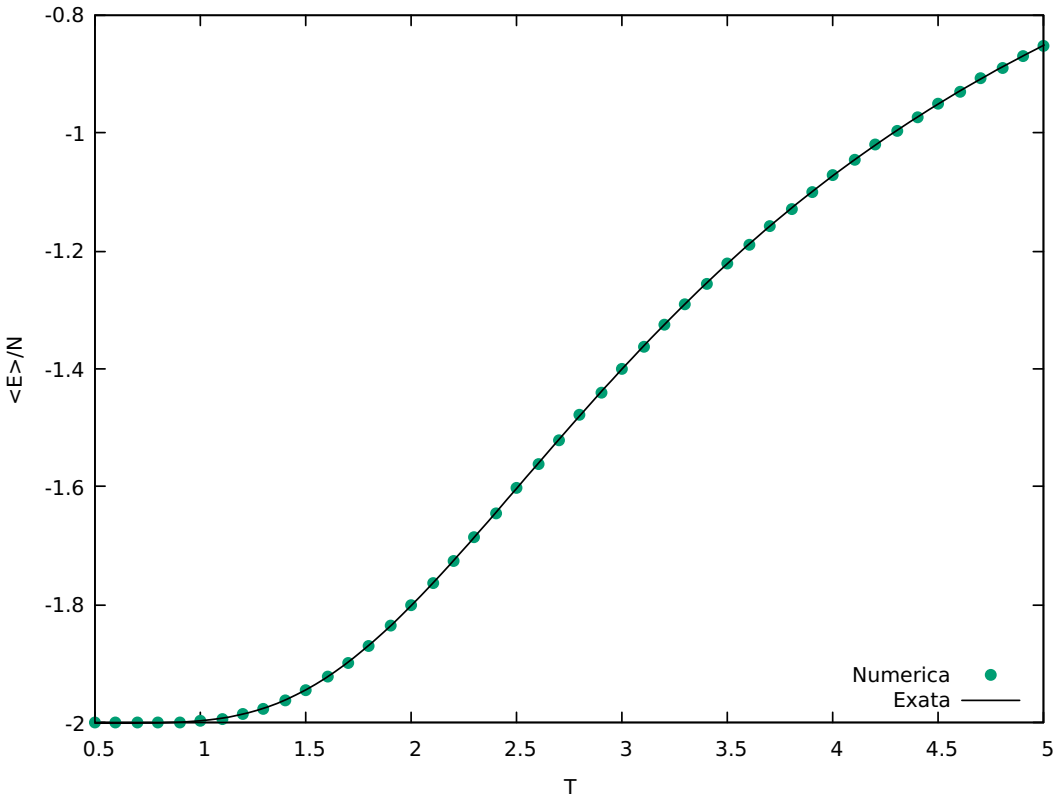

Figura 31 – Energia média por spin vs Temperatura - Solução numérica e analítica

Onde nota-se que o modelo segue perfeitamente os valores do resultado analítico. Essa precisão é obtida utilizando 10^7 passos no loop de Monte Carlo e usando $5 \cdot 10^4$ passos para o sistema chegar no equilíbrio. Mais passos foram utilizados para se obter uma maior precisão já que a rede nesse caso é bem pequena.

11.2 Paralelo

Para a versão em paralelo, o resultado obtido plotando a curva descrita pela solução analítica e o resultado numérico é demonstrado nos gráficos [32] e [33].

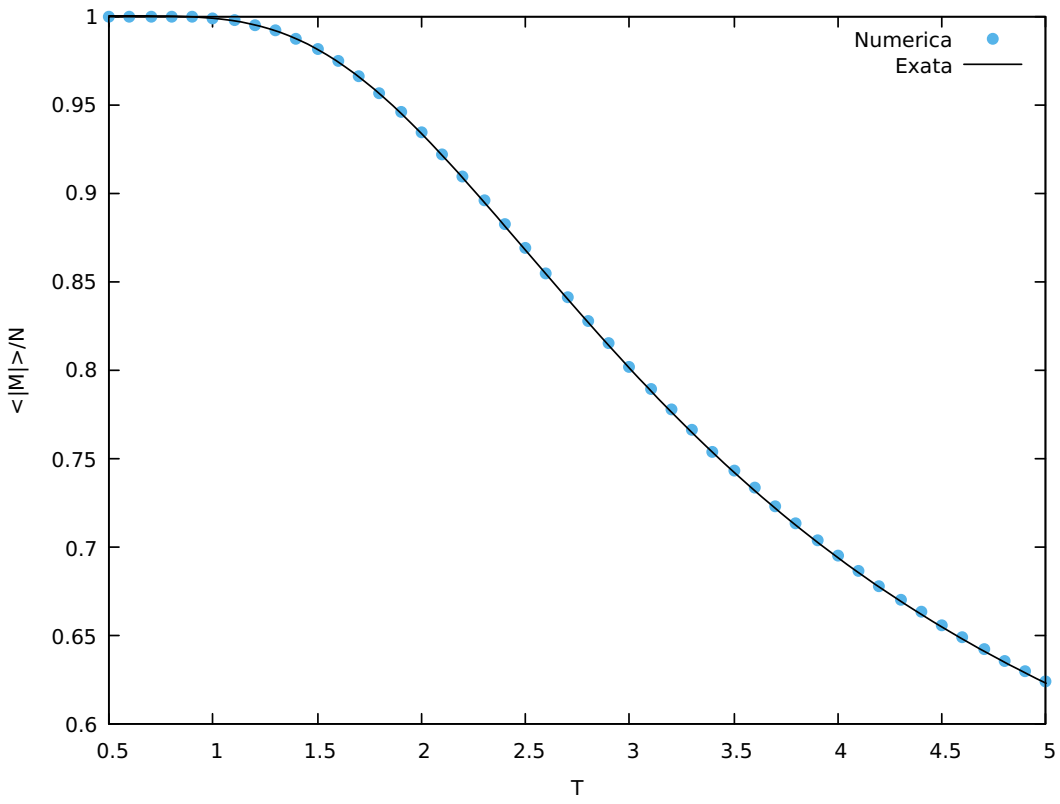

Figura 32 – Magnetização absoluta média por spin vs Temperatura - Solução numérica e analítica

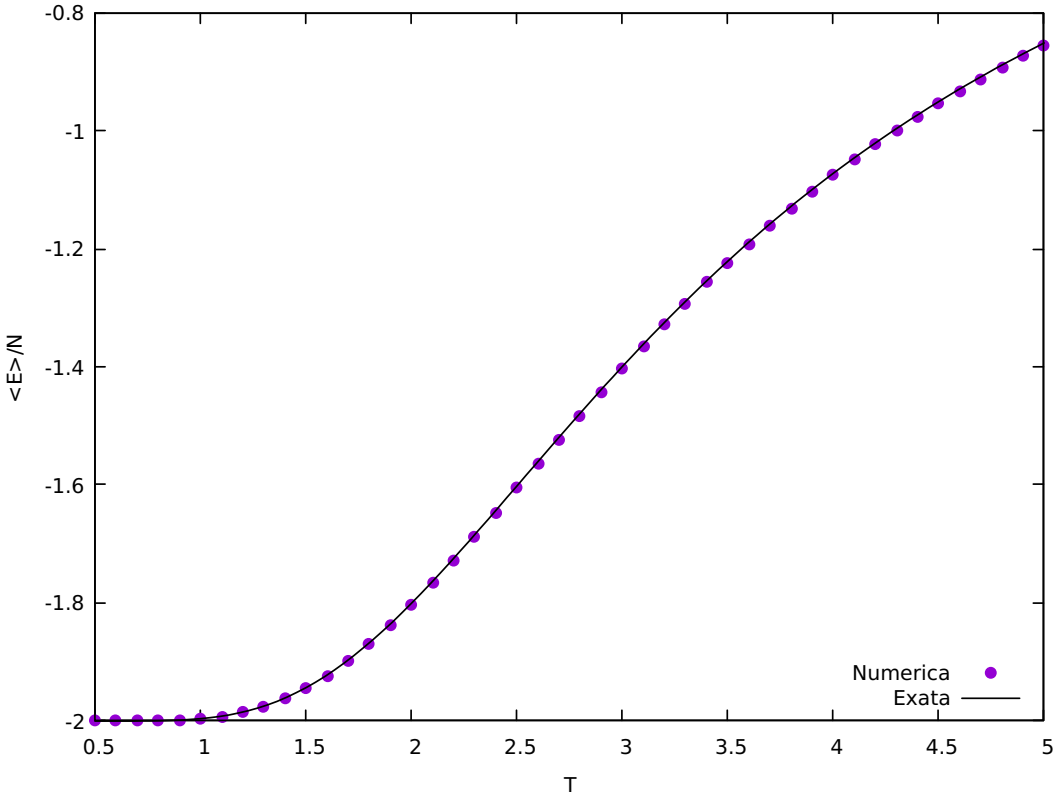

Figura 33 – Energia média por spin vs Temperatura - Solução numérica e analítica

Onde nota-se que o modelo em paralelo também segue perfeitamente os valores do resultado analítico.

11.3 Análise do comportamento do modelo

Como pode-se ver nas análises anteriores, tanto o modelo em serial como o modelo em paralelo apresenta excelentes resultados em comparação com a solução analítica (como deveria ser), visto isso, não há diferença entre utilizar o algoritmo em paralelo ou em serial (em questão de resultados do modelo). Com esses detalhes em mente, iremos agora a análise dos resultados para diversos tamanhos da rede de spins para analisar o comportamento do modelo. As condições iniciais foram as mesmas utilizadas anteriormente e os tamanhos da malha foram de $L = 2, 4, 8, 16, 32, 64, 128, 256$ e 528. Vale ressaltar que conforme aumentamos o tamanho da rede, o número de passos para a termalização também precisa ser aumentado, isto é, precisamos aumentar o número de N_{skip} para o modelo seguir seu funcionamento correto. Os resultados abaixo foram obtidos utilizando o algoritmo em paralelo (pelo menor tempo de execução e mesma confiabilidade de resultados).

11.3.1 Magnetização

O resultado esperado era que, após o equilíbrio e em baixas temperaturas, o sistema estivesse no seu estado de energia mínima (todos spins alinhados) e consequentemente, com a magnetização máxima (módulo igual a 1). Além disso, esse alinhamento se desfizesse conforme a temperatura aumenta, sendo que, as maiores flutuações devem ocorrem perto da temperatura crítica $T_c \approx 2.27$, já que exatamente nesse ponto ocorre a transição de fase (ferromagneto $\rightarrow$ paramagneto). Como pode-se notar, todos esses comportamentos são facilmente notados na figura [34]. Além disso, ele fica mais nítido conforme o tamanho da rede aumenta.

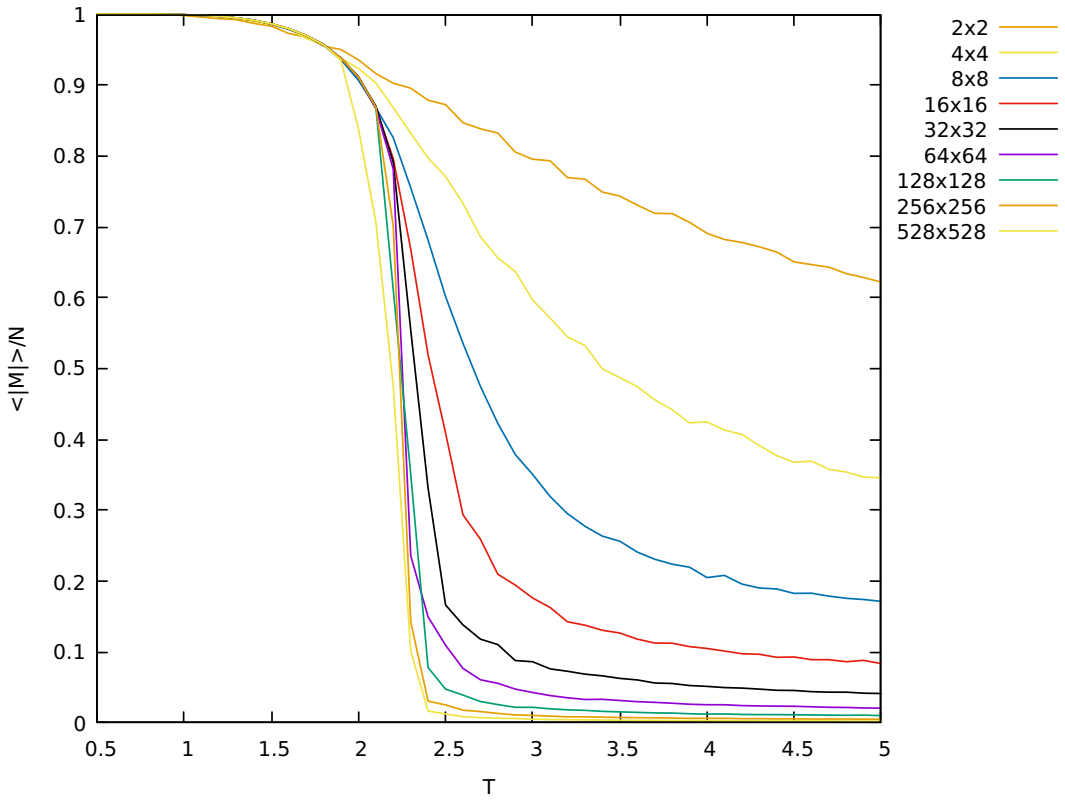

Figura 34 – Comparação da Magnetização absoluta por spin vs Temperatura para vários tamanhos da rede

11.3.2 Energia

O resultado esperado era que a energia inicial fosse mínima e que esse valor aumentasse conforme o aumento da temperatura. Após a temperatura crítica T_c o valor da energia deve tender a zero pelo alinhamento aleatório que um paramagneto possui. Além disso, como considera-se apenas os 4 vizinhos do spin, na temperatura $T = 0$ o valor da energia deveria ser $E = -4JN/2$, onde o fator $1/2$ vem da contagem dupla dos pares. Então, a energia por spin utilizando $J = 1$ deve ser -2 na temperatura $T = 0$. Esses comportamentos podem ser notados na figura [35], e, conforme o tamanho da rede aumenta

esse comportamento fica mais nítido.

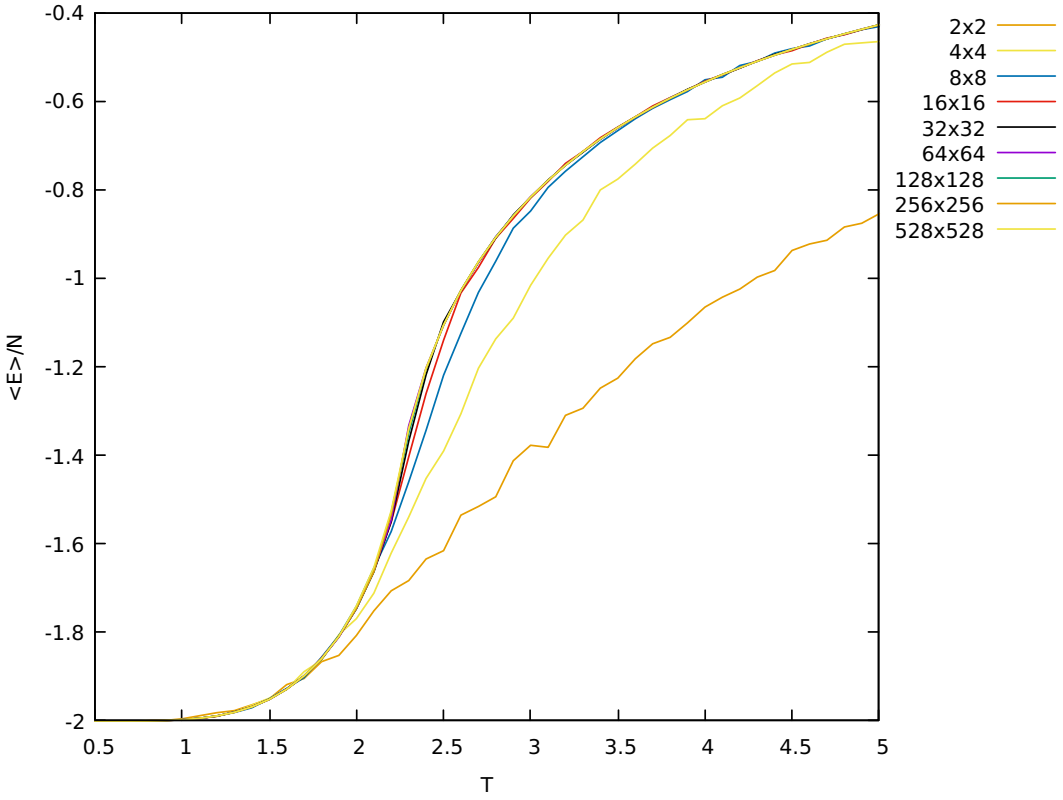

Figura 35 – Comparação da Energia por spin vs Temperatura para vários tamanhos da rede

Ademais, da simulação pode-se notar que na temperatura crítica a energia possui um ponto de inflexão e isso indica que o calor específico deve possuir uma divergência nessa temperatura, já que c está relacionado com a derivada parcial de E.

11.3.3 Calor específico

O calor específico deve possuir um pico em $T = T_c$ que aumenta conforme o tamanho da rede também aumenta (explodiria se $N \to \infty$). Esse comportamento pode ser notado em [36].

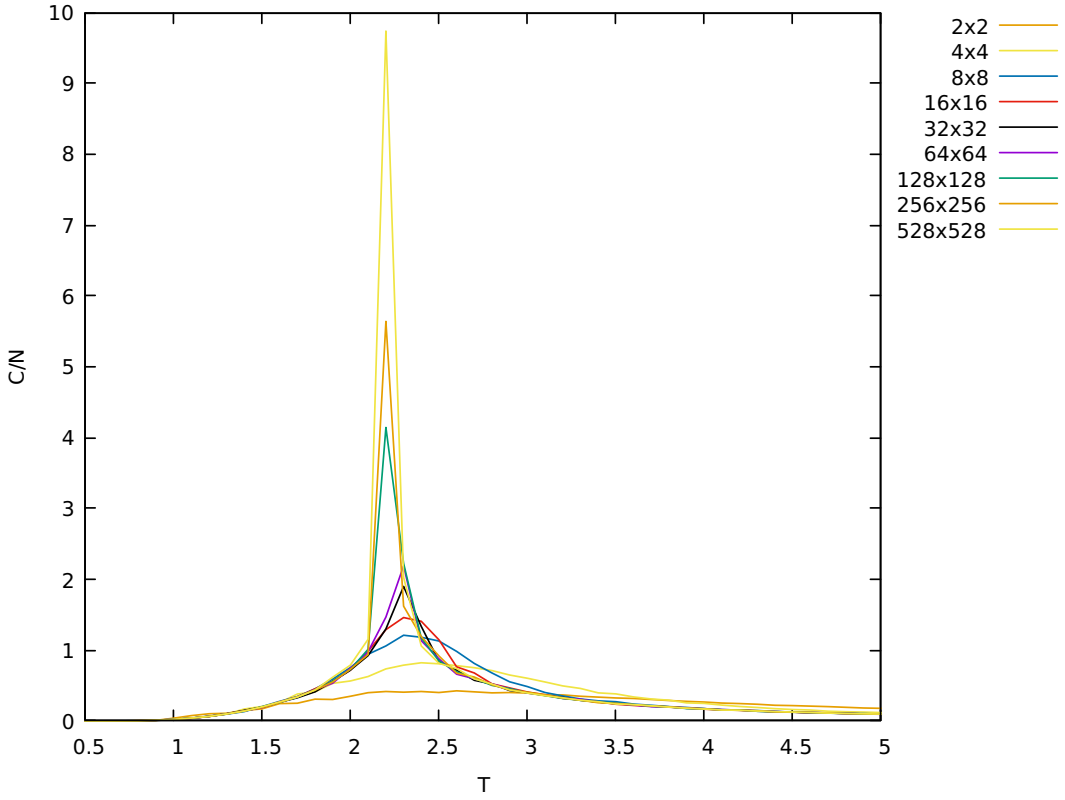

Figura 36 – Calor específico por spin vs Temperatura para vários tamanhos da rede

11.3.4 Susceptibilidade Magnética

Aqui encontra-se um problema já que analisar o comportamento magnético do sistema envolve o método de finite size scalling como mostra (KOTZE, 2008) e não será abordado. Porém, o resultado da simulação é demonstrado na figura [37].

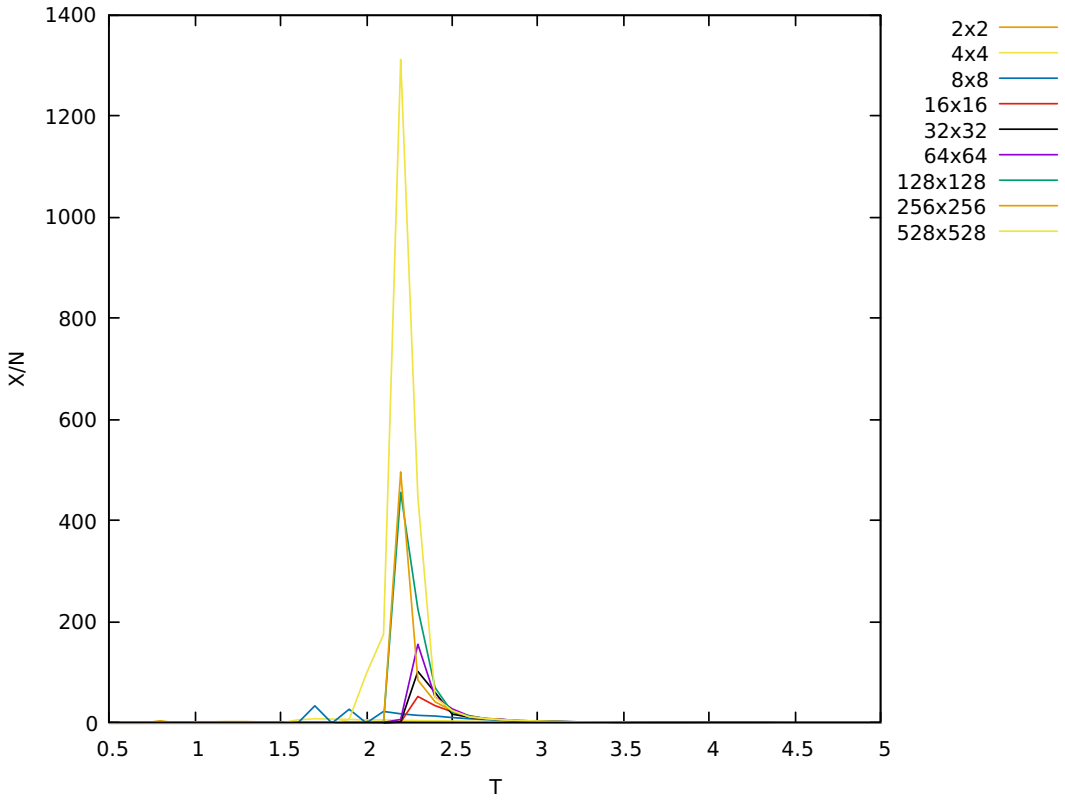

Figura 37 – Susceptibilidade magnética vs Temperatura para vários tamanhos da rede

12 Conclusões

Os resultados obtidos durante todo o trabalho mostram que todas as técnicas aplicadas, seja a nível de compilador, software ou técnicas de paralelização geraram resultados positivos, isto é, geraram ganho de desempenho no tempo de execução do algoritmo mantendo a confiabilidade dos resultados como discutido anteriormente.

As técnicas de otimização de software mostram que com simples práticas já é possível ganhar desempenho: Elas geraram um ganho no tempo de execução de **4.2**% em relação ao programa base. Somando as técnicas otimização a nível de compilador também foi possível gerar bons resultados: No melhor caso houve um ganho de cerca de **23**% em relação ao tempo base. Esse resultado é interesse já que nem sempre é possível/fácil aplicar a paralelização em alguns algoritmos, portanto, somente com essas técnicas já seria possível ganhar desempenho.

Com as técnicas de paralelização e com o auxílio da ferramente de profiling, foi possível obter em uma máquina de 8 threads um ganho de **62**% somente aplicando a paralelização e técnicas de software e **74**% em união com as técnicas a nível de compilador. Isso significa que aplicando todas as técnicas discutidas no trabalho foi possível reduzir o tempo de execução de um algoritmo em praticamente 3/4, o que é bem interessante. Infelizmente essa análise com mais threads não foi possível devido a problemas com a execução no cluster, isso prejudicou um pouco a estimativa máxima que as técnicas de paralelização poderiam trazer.

Mesmo assim, diante de tudo, os resultados obtidos durante todo o processo ainda foram muito interessantes e são de grande ajuda para pesquisas que dependem de métodos computacionais para serem realizadas.

Referências

COSTA, L. M. da. O modelo de ising 2d. 2006. Citado na página 40.

GIORDANO, N. J. *Computacional Physics*. [S.l.: s.n.], 1997. Citado na página 8.

KOTZE, J. Introduction to monte carlo methods for an ising model of a ferromagnet. *Arxiv*, 2008. Citado na página 47.

SANTOS, M. L. *SIMULAÇÃO DE MONTE CARLO NO MODELO DE ISING NA REDE QUADRADA*. Dissertação (Mestrado) — UFMG, 2014. Citado na página 40.

SCHROEDER, D. V. *An Introduction to Thermal Physics*. [S.l.: s.n.], 1999. Citado na página 8.