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

PROJETO: COMPUTAÇÃO CIENTÍFICA DE ALTO DESEMPENHO II

Autor: Daniel Santiago da Silva
Professor: Rodrigo Amorim

Volta Redonda

2022

Sumário

1 INTRODUÇÃO 5

2 METODOLOGIA 6
2.1 O modelo de Ising 6
2.2 Método de Monte Carlo 6
2.2.1 Algoritmo de Metropolis 7

3 IMPLEMENTAÇÃO 8
3.1 Fluxograma 8
3.2 Condições iniciais 9

4 BENCHMARK 10

5 PROFILE 27

6 IMPLEMENTAÇÃO DO PROGRAMA COM OPENMP 28

7 IMPLEMENTAÇÃO DO PROGRAMA COM MPI 32

8 BENCHMARK DO PROGRAMA COM VÁRIAS THREADS 37
8.1 Lab107C 37
8.2 LNCC 42

9 BENCHMARK DO PROGRAMA COM VÁRIOS CORES 43
9.1 Lab107C 43
9.2 LNCC 46
9.2.1 B710 46
9.2.2 SequanaX 50

10 COMPARAÇÃO ENTRE AS PARALELIZAÇÕES 55
10.1 Lab107C 55

11 DISCUSSÃO DOS RESULTADOS 59
11.0.1 Magnetização 64
11.0.2 Energia 65
11.0.3 Calor específico 66
11.0.4 Susceptibilidade Magnética 66

12 CONCLUSÕES 68

REFERÊNCIAS 69

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. Além disso, algumas técnicas de otimização de compilação foram aplicadas com o objetivo de se reduzir o tempo de execução do programa. Após isso, 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. Furthermore, some compilation optimization techniques were applied in order to reduce the execution time of the program. After that, parallelism techniques are also applied in order to obtain an even greater gain in the execution time of the program (keeping the results reliable). The general results obtained will be analyzed in order to understand if the model was applied correctly and if the performance gain techniques applied 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 Metodologia

2.1 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

$$H = -J \sum_{<i,j>} \sigma_i \sigma_j - B \sum_i \sigma_i \tag{2.1}$$

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.

2.2 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.

2.2.1 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.

3 Implementação

3.1 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 [1].

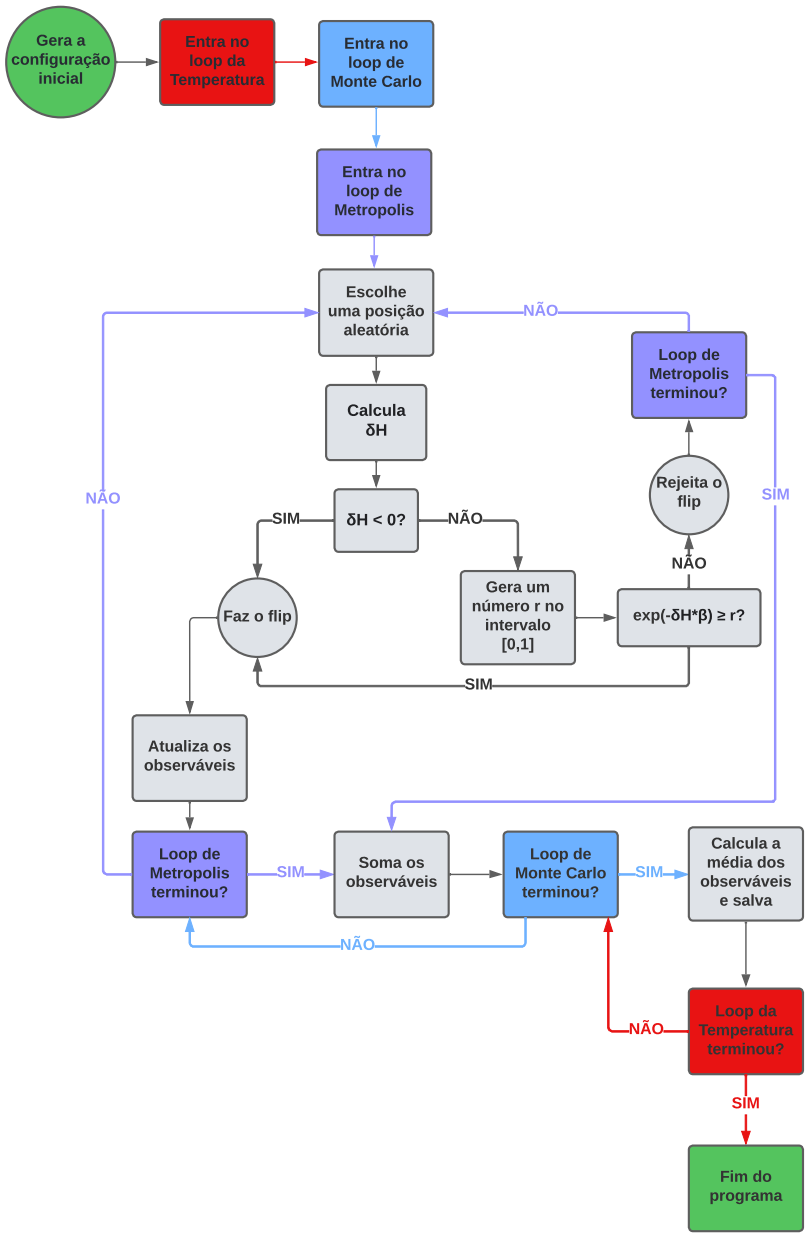

Figura 1 – Fluxograma

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

3.2 Condições iniciais

A Hamiltoniana do modelo de Ising geral é dada como mostra a equação (2.1), 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

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

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

$$\beta = \frac{1}{k_B T} = \frac{1}{T} \tag{3.2}$$

E a equação (3.1), efetivamente se torna

$$H = -\sum_{<i,j>} \sigma_i \sigma_j \tag{3.3}$$

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

$$P(\alpha) = e^{-\beta \delta H(\alpha)} \tag{3.4}$$

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.

4 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

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 $dT = 0.1$.

Compilador GCC

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	10505s	7297s	7338s	7266s
OX + Flags	10106s	6684s	6567s	6409s

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 [2].

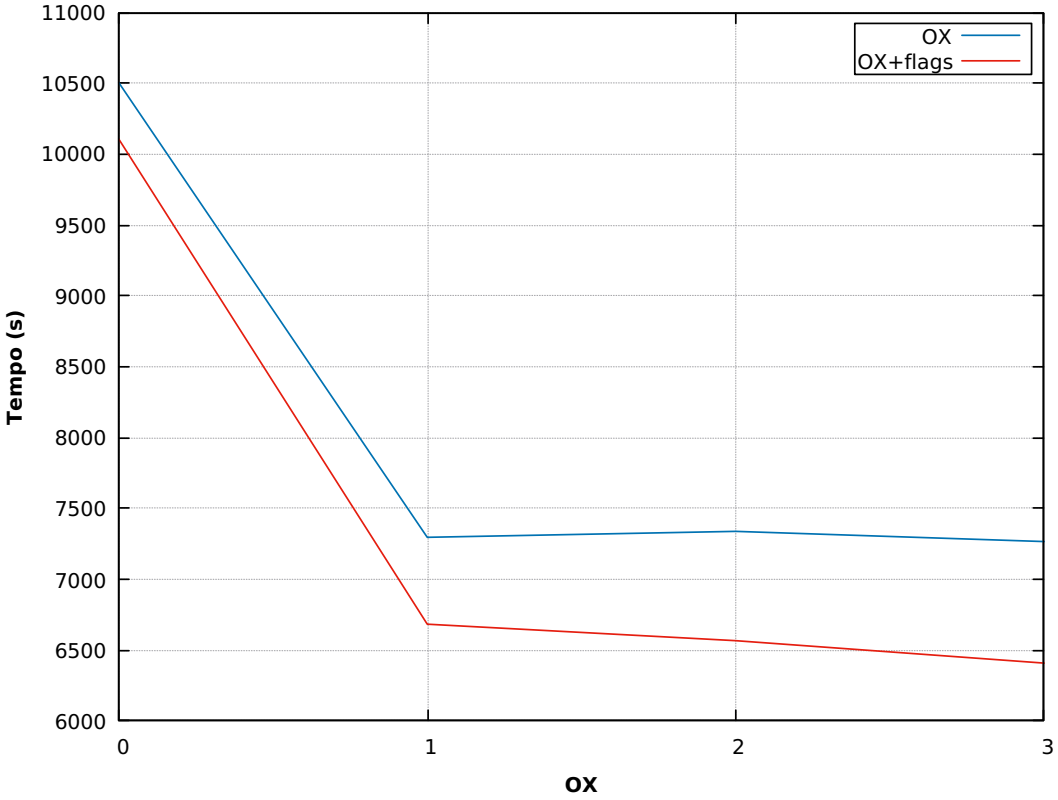

Figura 2 – Comparação geral entre o tempo pelo X

Aqui, pode-se inicialmente notar que, para todos valores de **X**, as compilações com as demais flags (além do **-OX**) resultaram em tempos menores (a maior diferença ficou com **857s** no caso **O3**). Além disso, o padrão é quase o mesmo: o pior tempo fica com **O0** e o melhor tempo sempre fica com **O3**, entretanto, **O1** é menor que **O2** só usando **-OX** mas é maior usando todas as flags (mesmo que a diferença seja bem pequena). 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 **6409**, que equivale a **1h:46m:49s**.

Fixando o pior tempo obtido com o programa, isto é, **10505s**, podemos calcular a eficiência que cada compilação resulta. Ou seja, plotar o gráfico da eficiência vs otimização, onde a eficiência é dada por

$$\eta[\%] = \left(1 - \frac{T_{opt}}{T_{no-opt}}\right) \cdot 100 \tag{4.1}$$

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

Portanto, neste caso, $T_{no-opt} = 10505$s e o gráfico da comparação entre as eficiências em função das flags é descrito em [3].

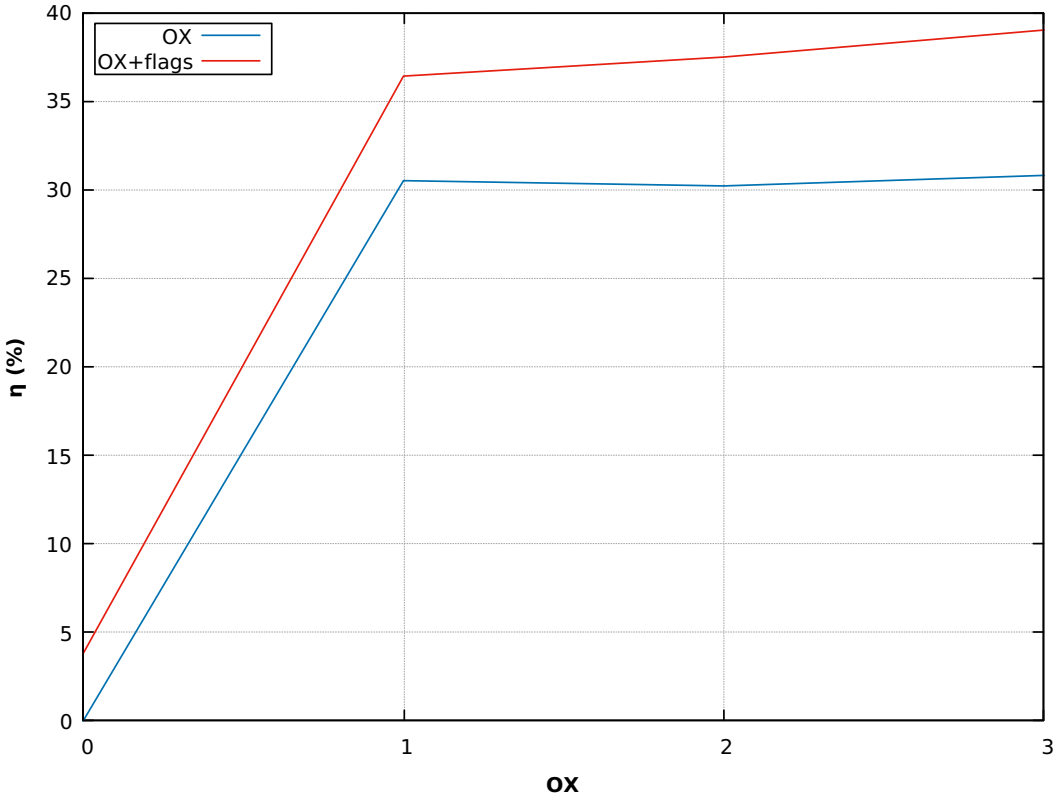

Figura 3 – Comparação geral entre a eficiência pelo X

Onde pode-se notar uma eficiência de aproximadamente 5% com as flags em **O0**, entre $30 - 37\%$ nos casos com **O1** e **O2** e no caso **O3** temos uma eficiência de 30.8% sem as demais flags e o melhor caso com as demais flags, mostrando uma eficiência de 39%, que equivale a uma redução de aproximadamente **1h:10m** no tempo de execução total usando a malha com $L = 528$.

Compilador Ifort

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 -w -mp1 -zero -xHOST -fast=2 -ipo**.

Os tempos obtidos nas 8 rodadas estão na tabela [3].

Flags	0	1	2	3
OX	6765s	6760s	6761s	6692s
OX + Flags	6317s	6547s	6549s	6263s

Tabela 3 – Tempos obtidos com as compilações usando o ifort

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

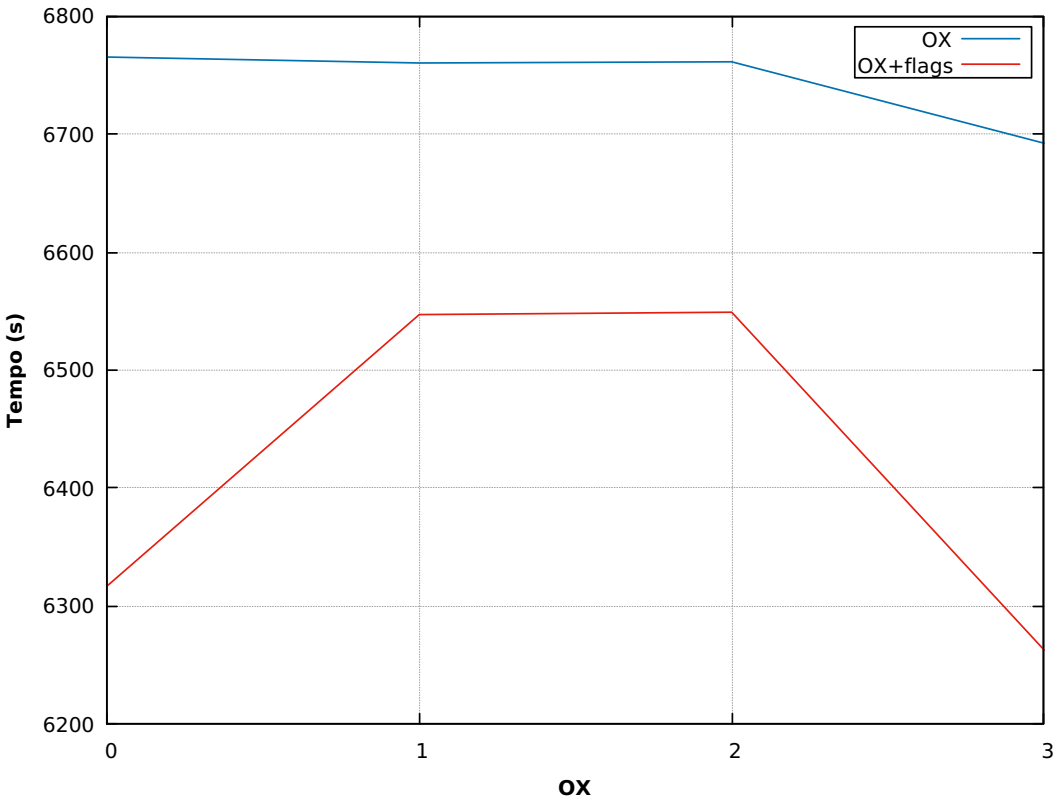

Figura 4 – Comparação geral entre o tempo pelo X

Aqui, pode-se inicialmente notar um comportamento praticamente todo diferente do obtido com o compilador gcc, a única semelhança é que o melhor tempo continua com os casos de **O3**. Um outro ponto interessante é que adicionar as demais flags além

do **-OX** tem um importância maior, isto é, reduzem mais o tempo. Nos casos usando somente **-OX** os valores para **X = 0, 1, 2** são quase os mesmos e a diferença só ocorre com **-O3**. Com as demais flags esse comportamento muda: os tempos com **-O1** e **-O2** são quase iguais, mas com **-O0** e **-O3** os tempos são bem menores. Contudo, o melhor tempo acontece no conjunto de flags **-O3 -w -mp1 -zero -xHOST -fast=2 -ipo** sendo de **6263s**, que equivale a **1h:44m:23s**. Um ponto interessante a se perceber é que apesar do comportamento praticamente 100% diferente do obtido com o gcc, o melhor tempo encontrado nos dois casos foi praticamente o mesmo.

Fixando o pior tempo obtido com o programa, isto é, **10505s**, podemos calcular a eficiência que cada compilação resulta usando a equação (4.1). O gráfico da comparação entre as eficiências em função das flags é descrito em [5].

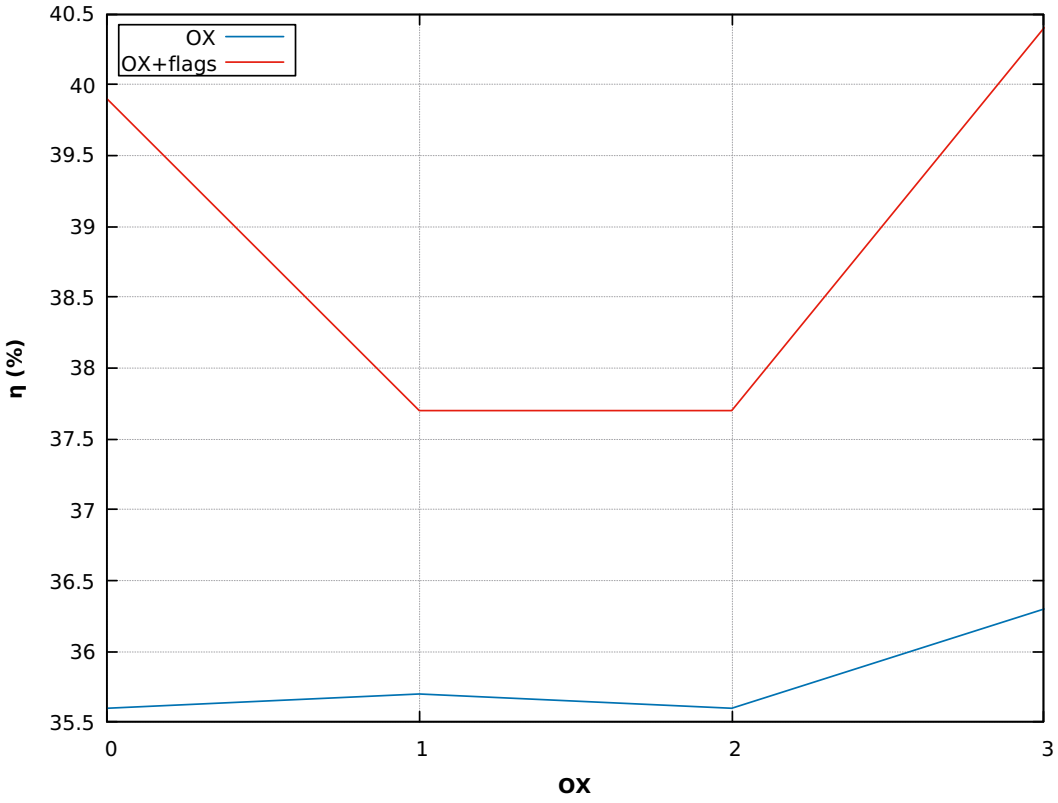

Figura 5 – Comparação geral entre a eficiência pelo X

As eficiências no geral oscilam em torno de $35 - 37\%$ para quase todos os casos, com exceção dos casos com **O0** e **O3** junto com as demais flags, que atingiram, respectivamente 39.9% e 40.4% de eficiência. O melhor caso, como já comentado anteriormente, foi com **O3** junto com as demais flags, que obteve uma redução que equivale a **1h:10m** no tempo de execução total usando a malha com $L = 528$.

LNCC - B710

A tabela [4] 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 4 – Configurações da máquina

As condições 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.

Compilador GCC

Os tempos obtidos nas 8 rodadas estão na tabela [5].

Flags	0	1	2	3
OX	581s	425s	428s	427s
OX + Flags	573s	396s	385s	367s

Tabela 5 – 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].

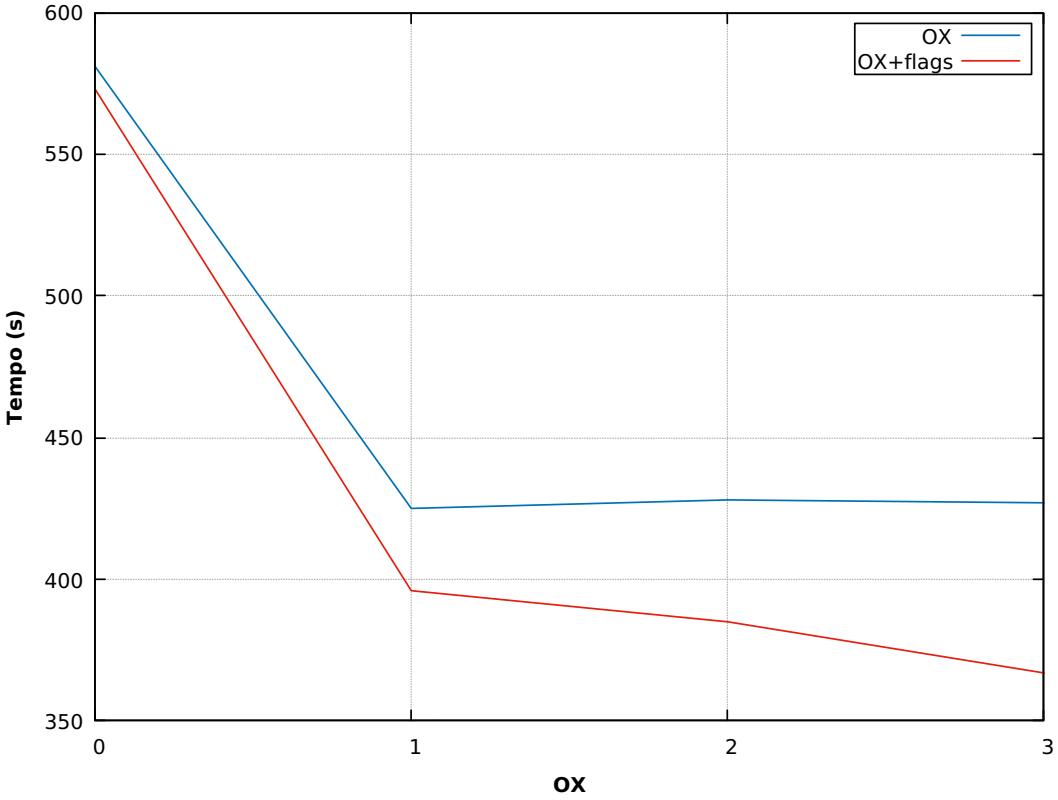

Figura 6 – Comparação geral entre o tempo pelo X

Aqui o padrão é um pouco diferente do encontrado no LAB 107C, nos casos só com a flags **OX** o pior tempo fica com **O0**, seguido de **O2**, depois **O3** e o melhor fica com **O1**. Esse resultado muda um pouco usando as flags **OX** junto com as demais flags: o pior tempo fica com **O0**, seguido de **O1**, depois **O2** e o melhor fica com **O3**. Dessa forma, o melhor caso ocorre com a utilização das flags **-O3 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native** sendo de **367s**.

Dada a equação (4.1) e os tempos obtidos em cada compilação, pode-se calcular as eficiências usando o $T_{no-opt} = 581$s. Isso está demonstrado em [7].

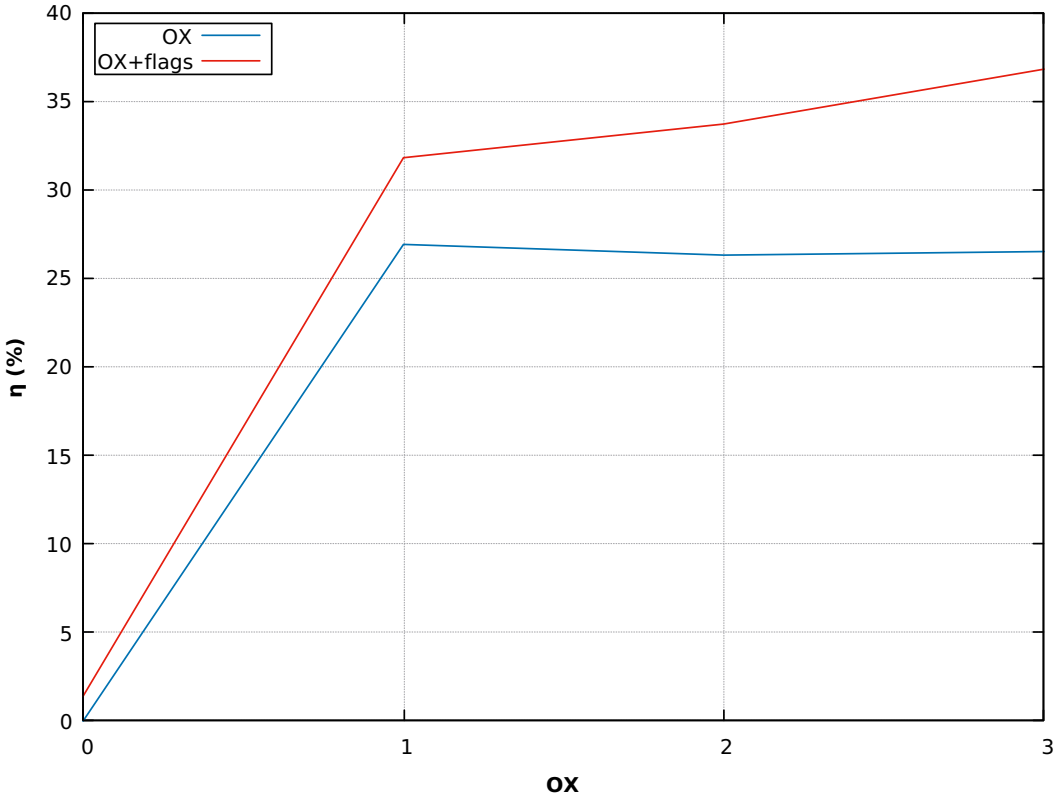

Figura 7 – Comparação geral entre a eficiência pelo X

Onde pode-se notar que, utilizando **-O1** as eficiências ficam entre 26 - 33% aproximadamente; entre 26 - 34% com **-O2**; aproximadamente 26% utilizando somente **-O3** e 37% utilizando **-O3 + flags**. Não é um resultado muito diferente do encontrado no LAB 107C (as pequenas flutuações provavelmente ocorrem pela malha utilizada aqui ser menor.

Compilador Ifort

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

Flags	0	1	2	3
OX	361s	355s	360s	361s
OX + Flags	335s	351s	335s	335s

Tabela 6 – Tempos obtidos com as compilações usando o ifort

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 [8].

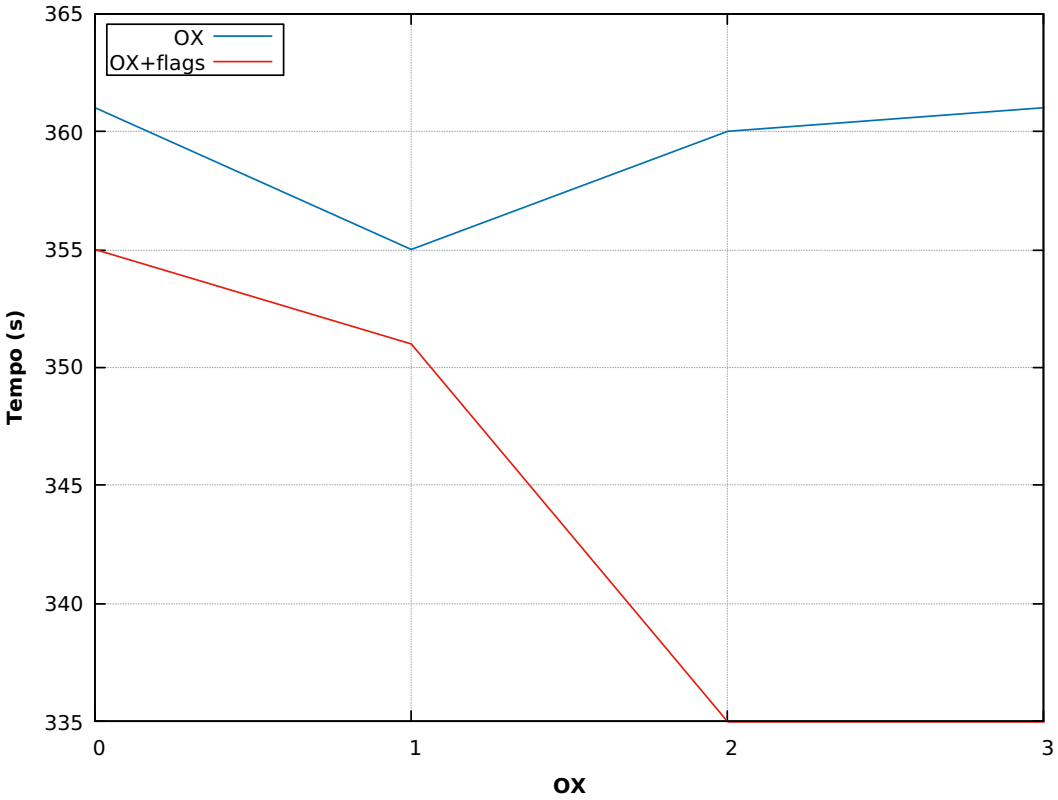

Figura 8 – Comparação geral entre o tempo pelo X

Aqui o padrão é bem diferente do encontrado no LAB 107C. Temos que, nos casos somente com a flags **OX**, o pior tempo fica com **O0** e **O3** empatados, seguido de **O2** e depois **O1**. Esse resultado muda bastante usando as flags **OX** junto com as demais flags: o pior tempo fica com **O0**, seguido de **O1**, depois **O2** e **O3** empatados. Dessa forma, o melhor caso ocorre com a utilização das flags **-O3 (ou -O2) -w -mp1 -zero -xHOST -fast=2 -ipo** sendo de **335s**.

Dada a equação (4.1) e os tempos obtidos em cada compilação, pode-se calcular as eficiências usando o $T_{no-opt} = 581$s. Isso está demonstrado em [9].

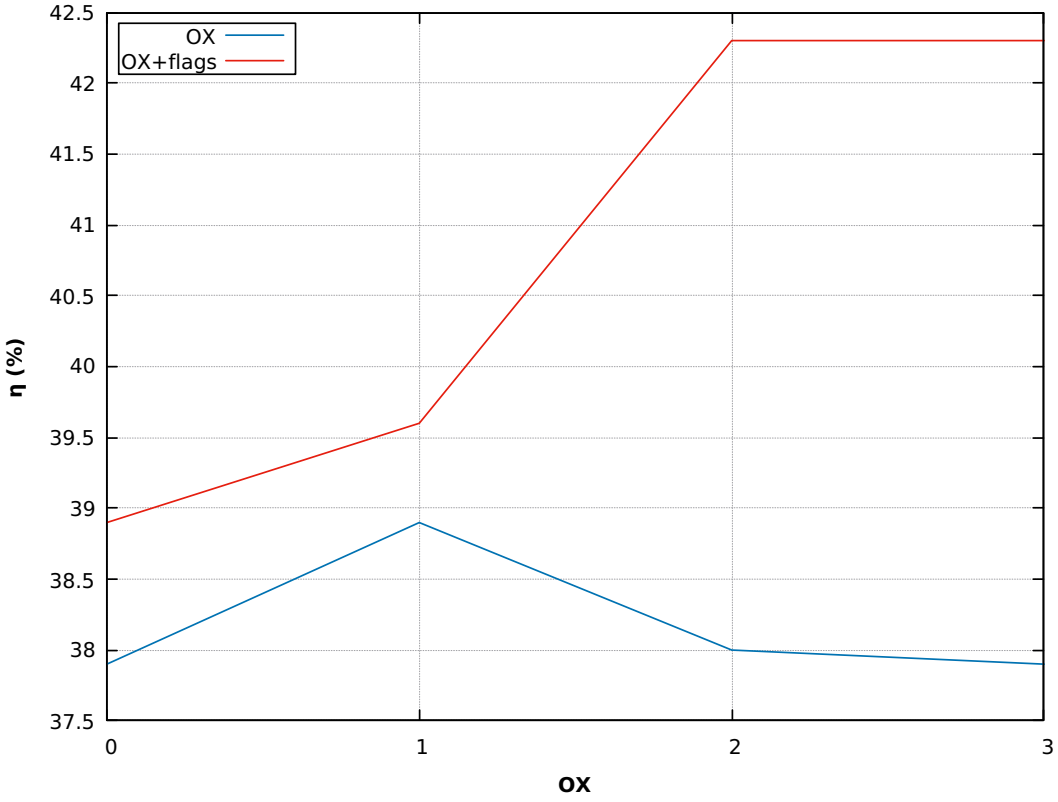

Figura 9 – Comparação geral entre a eficiência pelo X

Aqui pode-se notar que as eficiências em geral oscilam em torno de $37\% - 39\%$ com exceção dos casos com o melhor tempo, isto é **O2** ou **O3** junto com as demais flags, que alcançaram uma eficiência de mais de 42%. Apesar do comportamento bem diferente, a eficiência máxima encontrada ficou bem próxima da encontrada no LAB 107C (cerca de 40%).

LNCC - SequanaX

A tabela [7] 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 7 – 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.

Compilador GCC

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

Flags	0	1	2	3
OX	442s	338s	372s	380s
OX + Flags	425s	323s	305s	301s

Tabela 8 – 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 [10].

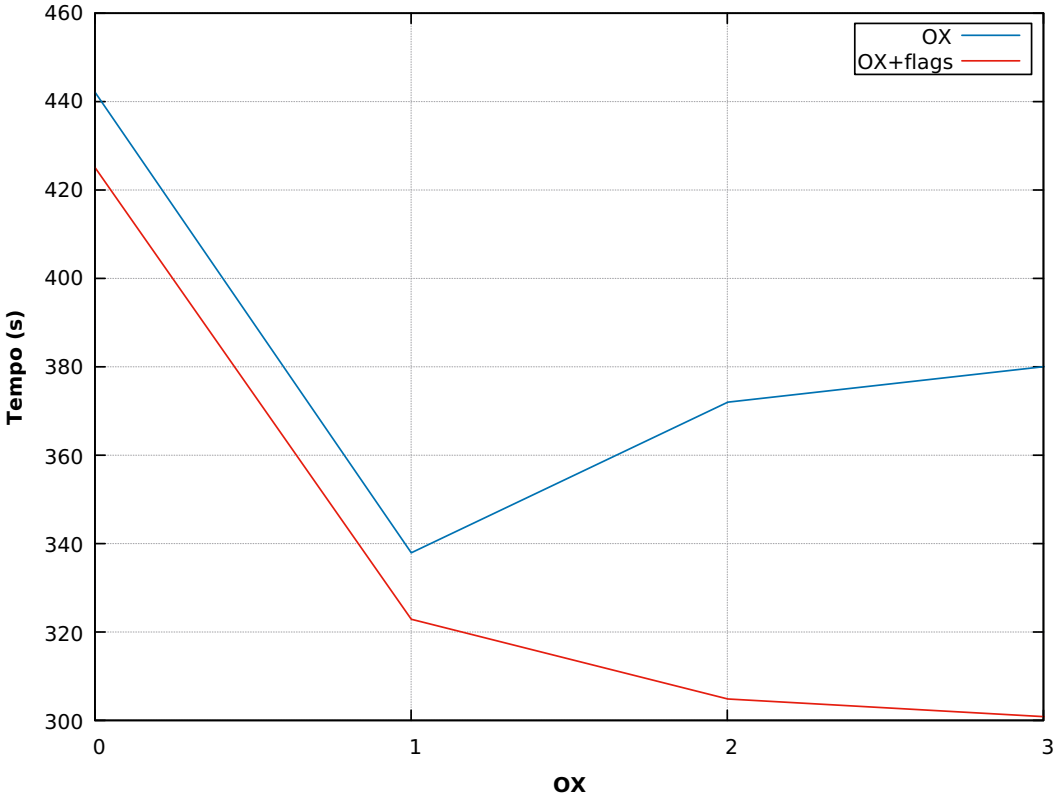

Figura 10 – Comparação geral entre o tempo pelo X

O padrão aqui também acontece de forma diferente ao usar somente a flag **OX** ou ao usar a flag **OX** em conjunto com as demais flags. No primeiro caso, somente com a flag **OX**, o pior tempo fica com **O0**, seguido de **O3**, depois **O2** e o melhor fica com **O1**. Usando a flag **OX** junto com as demais flags o tempo diminui conforme aumentamos o valor de **X**, desta forma, o melhor caso acontece com **-O3 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native** sendo de **301s**.

Dada a equação (4.1) e os tempos obtidos em cada compilação, pode-se calcular as eficiências usando o $T_{no-opt} = 442$s. Isso está demonstrado em [11].

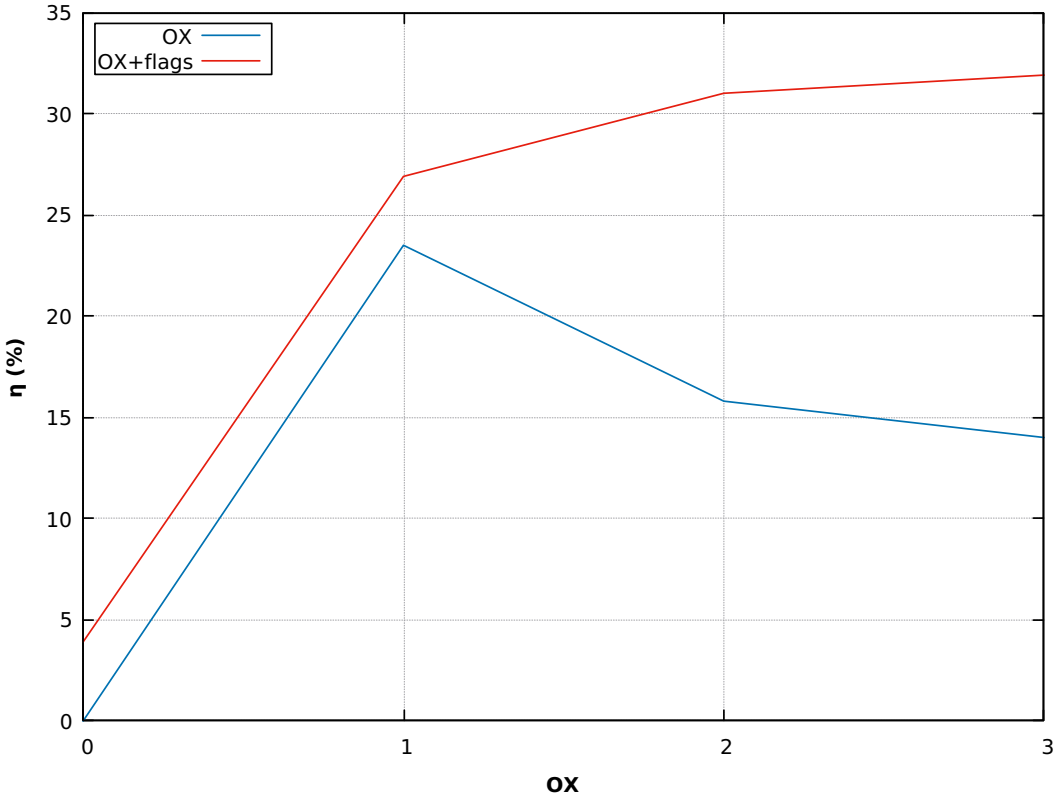

Figura 11 – Comparação geral entre a eficiência pelo X

Onde pode-se notar que, utilizando **-O1** as eficiências ficam entre 23 - 27% aproximadamente; entre 15 - 31% com **-O2**; aproximadamente 14% utilizando somente **-O3** e 32% utilizando **-O3** junto com as demais flags. Os ganhos na SequanaX com as flags usando o compilador gcc, em geral, foram menores do que no B710 ou no LAB107C.

Compilador Ifort

Os tempos obtidos nas 8 rodadas estão na tabela [9].

Flags	0	1	2	3
OX	239s	243s	238s	240s
OX + Flags	214s	244s	214s	212s

Tabela 9 – Tempos obtidos com as compilações usando o ifort

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 [12].

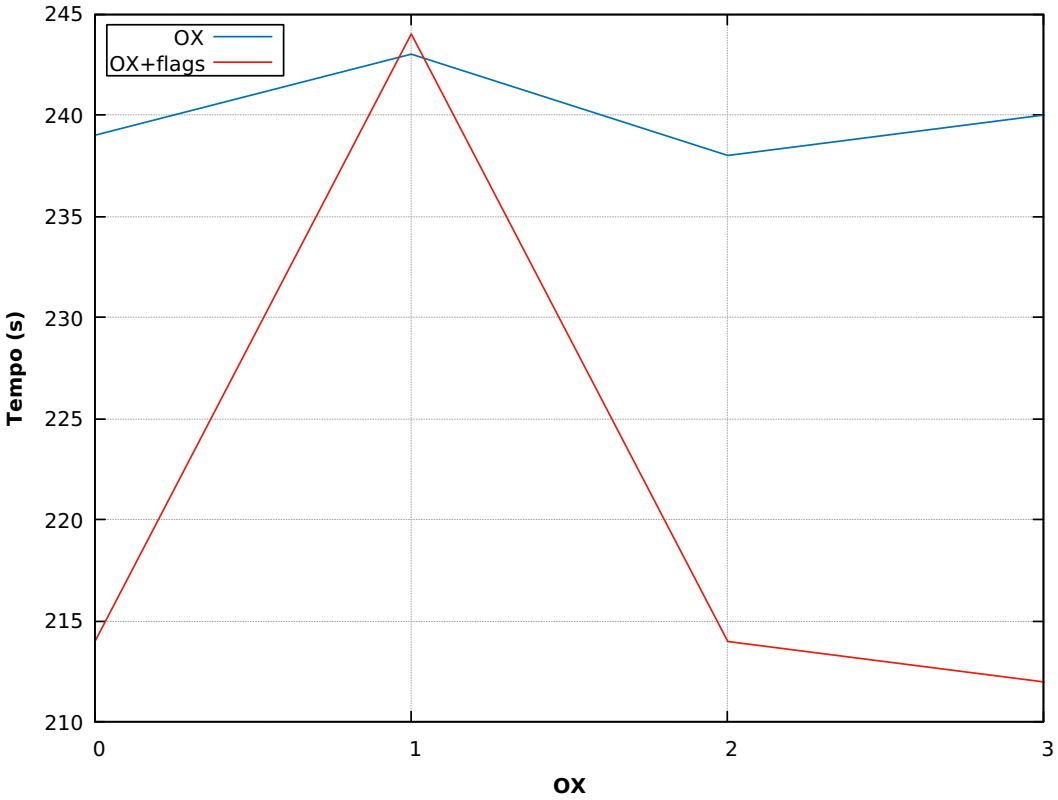

Figura 12 – Comparação geral entre o tempo pelo X

Usando o compilador ifort na SequanaX foi observado um caso não visto nas outras máquinas: o tempo piorou ao usar as demais flags de otimização, isto aconteceu no caso usando **O1**. Fora esse pequeno detalhe, o comportamento é o seguinte: usando somente a flag **OX** temos o pior tempo com **O1**, seguido de **O3**, depois **O0** e por fim **O2**; usando a flag **OX** em conjunto com as demais flags temos o pior tempo **O1**, seguido de **O0** e **O2** empatados e o melhor caso acontece com **O3**. Dessa forma, o melhor caso ocorre com a utilização das flags **-O3 -w -mp1 -zero -xHOST -fast=2 -ipo** sendo de **212s**.

Dada a equação (4.1) e os tempos obtidos em cada compilação, pode-se calcular as eficiências usando o $T_{no-opt} = 442$s. Isso está demonstrado em [13].

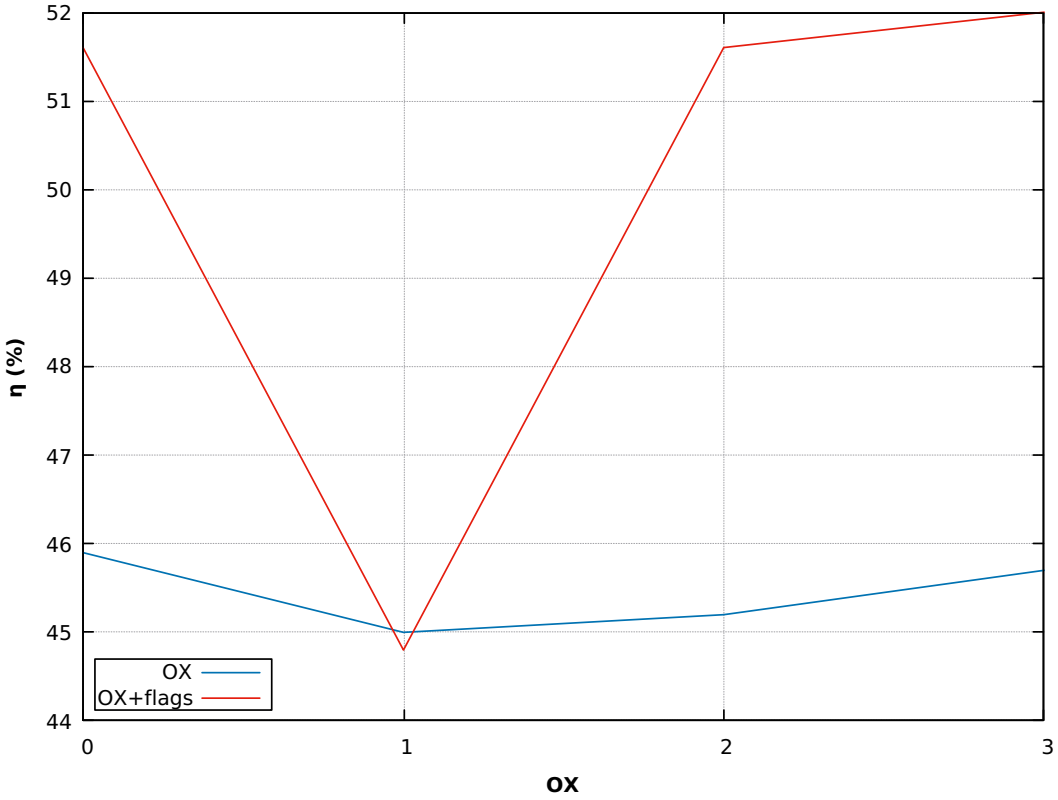

Figura 13 – Comparação geral entre a eficiência pelo X

As eficiências encontradas na SequanaX com o compilador ifort foram mais altas do que nas demais máquinas: oscilam em torno de $45\% - 50\%$, em geral. A melhor eficiência obtida foi de 52% com o **O3** em conjunto com as demais flags.

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, para isso, usaremos as flags **OX** em conjunto com as demais flags para os dois compiladores utilizados: gcc e ifort.

Compilador GCC

A comparação está demonstrada em [14].

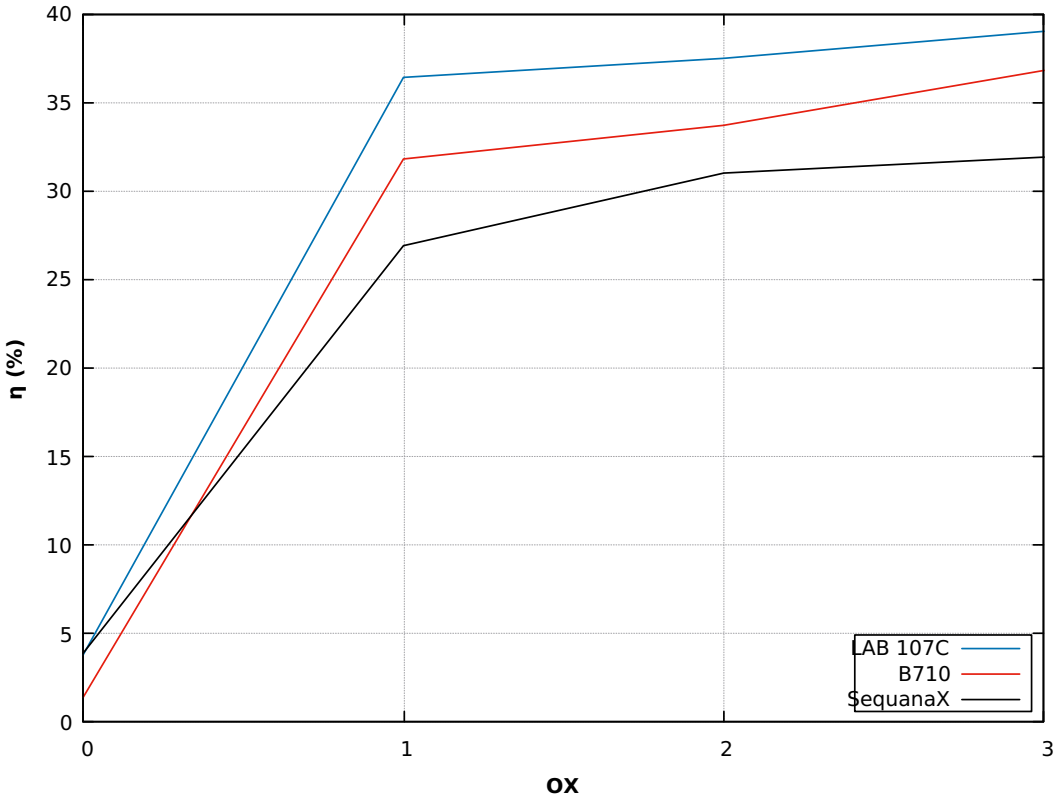

Figura 14 – Comparação das eficiências obtidas em diferentes máquinas

Quando analisamos as eficiências obtidas nas três máquinas é possível perceber a semelhança no comportamento, que não se destoa muito nas três máquinas (apesar dos valores serem diferentes). Apesar disso, surpreendentemente, os melhores resultados usando o compilador gcc foram obtidos no LAB 107C, isso talvez se dê pelo tamanho da malha ser diferente nas máquinas. Entretanto, com os testes feitos em 3 máquinas, é possível inferir que, no geral, aplicar a flag **-O3** seja a melhor tentativa em casos que não pode-se fazer um benchmark para encontrar de fato a melhor combinação em uma máquina qualquer.

Compilador Ifort

A comparação está demonstrada em [15].

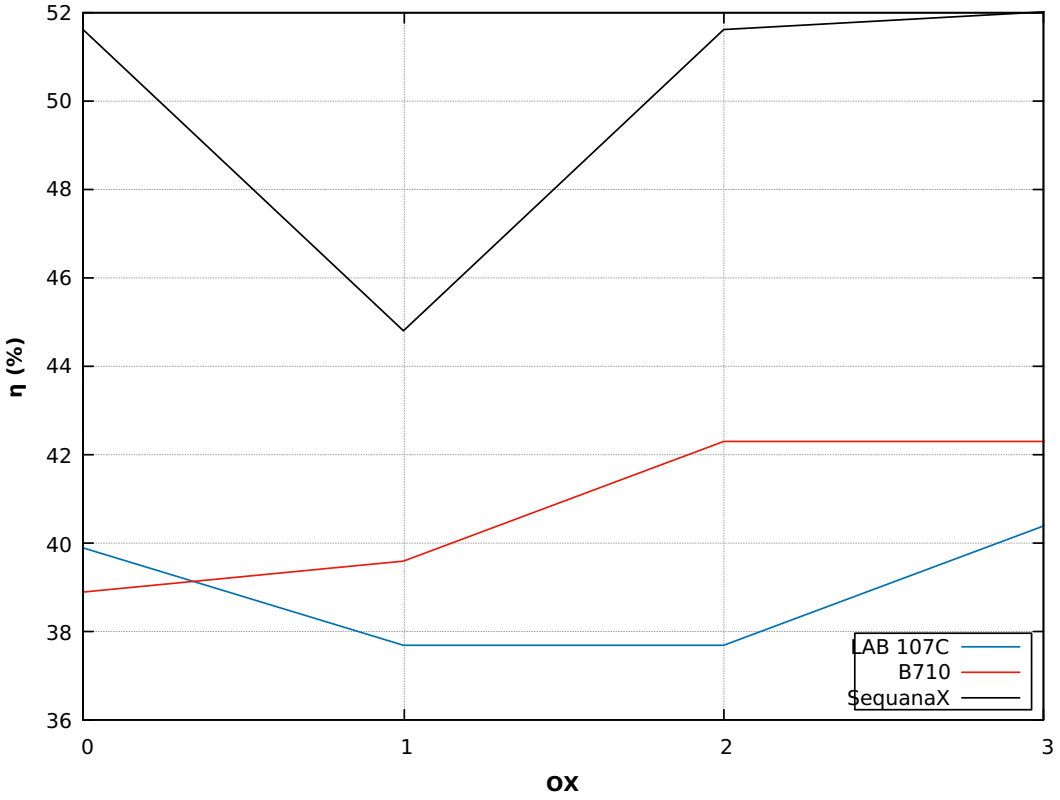

Figura 15 – Comparação das eficiências obtidas em diferentes máquinas

Com o compilador ifort não houve um padrão seguido nas três máquinas, com exceção de que, nas 3, o melhor caso ocorre com **-O3**. Um fator importante a ser percebido é que os valores encontrados na SequanaX foram bem mais altos, e, diferentemente do encontrado usando o gcc, o pior caso foi no LAB 107C. Isso talvez se dê pelo fato de que o ifort é um compilador da intel e provavelmente possui mais otimizações de funcionamento com máquinas mais novas, e como a SequanaX é a mais nova entre as três, ela se sai melhor. Ainda sim, pode-se concluir que, um bom palpite de compilação em uma máquina desconhecida é usar a flag **-O3**.

5 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 foi feito 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 [16].

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 42.68   2615.50  2615.50 4095864320     0.00     0.00  hamiltoniana
 27.88   4324.01  1708.51 4083040256     0.00     0.00  testa_flip
 12.41   5084.25   760.24 12249399552     0.00     0.00  Rand
 10.15   5706.23   621.98 4083040256     0.00     0.00  escolhe_pos
  7.20   6147.17   440.94                             main
  0.03   6149.09     1.92       46     0.04    37.39  equilibra
  0.01   6149.70     0.61        1     0.61     0.63  inicia_malha
  0.00   6149.76     0.06       46     0.00     0.00  magnetizacao_total
  0.00   6149.78     0.02       46     0.00     0.18  energia_total
```

Figura 16 – Profile programa base

Onde o tempo de execução do programa foi de **10505s**, que equivale a **2h:55m:5s**. Portanto, analisando o profile, pode-se notar que há cinco funções do programa que ocupam basicamente todo o tempo de execução: hamiltoniana, testa_flip, Rand, escolhe_pos e a main. Sendo a hamiltoniana o maior gargalo, ocupando mais de 42% do tempo total. Essa informação será útil no processo de paralelização do algoritmo posteriormente.

6 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 [17], 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 17 – 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 [18].

```
//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 18 – 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 [19].

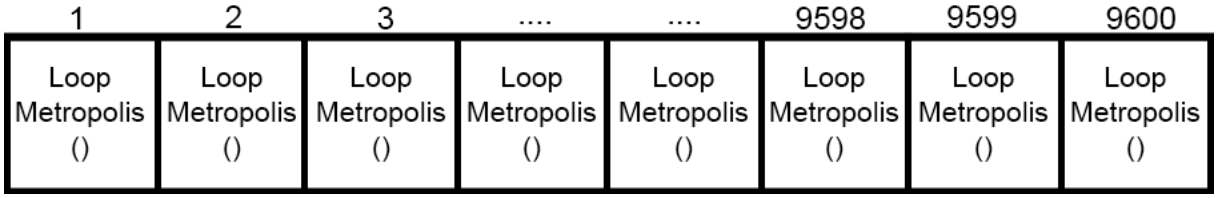

Figura 19 – Funcionamento do loop de Monte Carlo

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 [20].

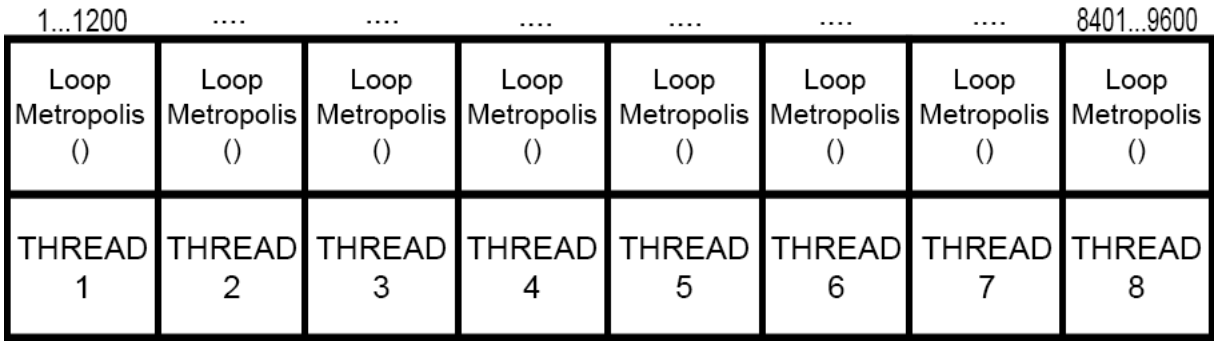

Figura 20 – Divisão da malha para uma máquina de 8 threads

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 [21].

```
//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 21 – 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.

7 Implementação do Programa com MPI

Como já visto anteriormente no capítulo sobre a paralelização com OpenMP, a parte de interesse para efetuar a paralelização são os loops do programa. Entretanto, com o MPI não há a restrição de só utilizar números inteiros como ocorria com o OpenMP e por isso o loop escolhido para efetuar a paralelização foi o loop da temperatura.

Uma vez escolhido qual loop a paralelização irá ser feita, discutiremos como fica a divisão da malha. O funcionamento do loop da temperatura pode ser melhor entendido pela figura [22].

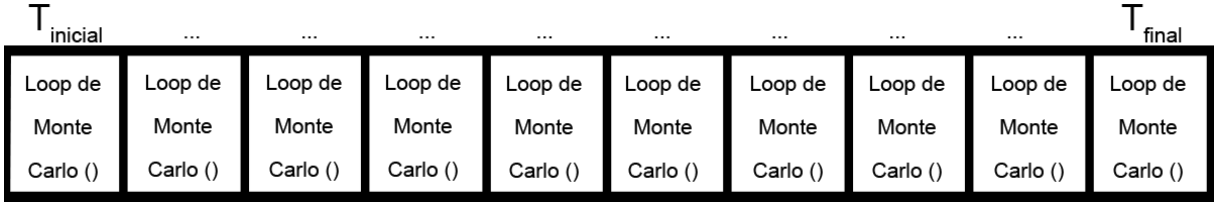

Figura 22 – Funcionamento do loop da temperatura

Isto é, durante cada iteração do loop da temperatura, o loop de Monte Carlo é chamado. Ou seja, na prática, a divisão será feita como mostra a figura [23].

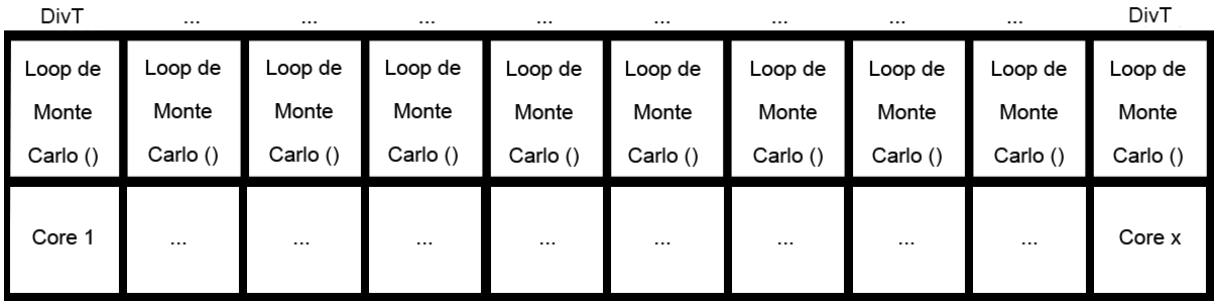

Figura 23 – Divisão da malha para uma máquina

Onde DivT é dado por

$$\text{DivT} = p_T/C$$

sendo p_T o número de iterações do loop da temperatura e C o número de cores disponíveis na máquina.

Para executar esse processo no algoritmo foram utilizadas três diretivas do MPI: MPI_Scatter, MPI_Gather e MPI_Barrier.

MPI_Scatter: Envia dados de um core para todos os outros cores em um comunicador.

MPI_Gather: Reúne valores de um grupo de cores em um só core.
MPI_Barrier: Cria um barreira que bloqueia o avanço até que todos os cores do comunicador atinjam esse ponto.

Além das diretivas, algumas mudanças no algoritmo foram necessárias para a paralelização ser possível. A seguir, elas serão discutidas.

A mudança básica é que as variáveis que antes era definidas no início da função main, agora são definidas dentro da zona paralela para que cada core tenha uma cópia da variável.

```
//inicia a rede aleatóriamente
inicia_malha(&seed, Rede);

//calcula a diferença entre T_f e T_i
deltaT = T - Tmin;

//calcula a quantidade de passos que o loop da temperatura executa
qtdT = deltaT/dT;

count = 0;

//checa se a quantidade de passos divide pelo número de cores, se não, cria passos "fantasmas"
while((qtdT%size)!=0){
    qtdT += 1;
    count++;
}

//calcula a quantidade de passos de temperatura que cada core irá executar
div = qtdT/size;
```

Figura 24 – Primeira mudança

Além disso, a parte [24] foi inserida no código. Ou seja, inicialmente se calcula a diferença entre T_f e T_i e divide-se pelo passo de interação dT, isso irá calcular o número de iterações no loop da temperatura. Logo em sequência é realizado uma checagem para saber se o número de iterações é divisível pelo número de cores disponíveis, se o resto não for 0, incrementa-se o número de iterações com iterações "fantasmas"(serão descartadas) até que essa divisão tenha resto 0 (isso é necessário para se utilizar o Scatter e o Gather da forma correta).

```
//variaveis internas de todos cores
double divEne[div], divMag[div], divCal[div], divSusc[div], divTemp[div];
double E_media[qtdT], Mabs_media[qtdT], calor_esp[qtdT], susc_mag[qtdT], Temperatura[qtdT];

//checa se o processo está sendo executado no core mestre, se sim, gera o vetor temperatura (contém os valores do loop)
if(rank == MESTRE){
    i=0;

    for(l=Tmin; l<=T; l+=dT){
        Temperatura[i] = l;
        i++;
    }
}

//divide as fatias do vetor temperatura para os cores
MPI_Scatter(Temperatura, div, MPI_DOUBLE, divTemp, div, MPI_DOUBLE, MESTRE, MPI_COMM_WORLD);
```

Figura 25 – Segunda mudança

Mais adiante, novas modificações ocorrem: os valores que eram salvos no arquivo anteriormente (que eram variáveis comuns) agora precisam ser vetores e são definidos

nesse ponto do algoritmo; um vetor para armazenar os valores da temperatura que serão utilizados no loop é definido; "pedaços"de vetores (que serão calculados em cada core e enviados ao core mestre no final) também são gerados.

Além disso, o vetor temperatura tem seus valores definidos no core mestre (cada espaço de memória do vetor recebe um valor de temperatura). Isto é, cada índice do vetor está associado a um valor do loop da temperatura, como pode ser demonstrado abaixo.

- $0 \rightarrow T_{min}$
- $1 \rightarrow T_{min} + dT$
- $2 \rightarrow T_{min} + 2dT$
  $\vdots$
- $N \rightarrow T_{max}$

Após o vetor temperatura ser definido, utiliza-se a diretiva MPI_Scatter que irá quebrar esse vetor em pedaços de tamanho igual e irá enviar esses pedaços para os cores disponíveis no sistema. Ou seja, cada core recebe uma faixa de valores de temperatura que irá ser utilizada no loop da temperatura individual de cada core.

```
//loop da temperatura
for(p=0; p<div; p++) {

    //termalização
    equilibra(&seed, J, B, k, divTemp[p], 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;

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

            if(testa_flip(pos, &dE, &seed, J, B, k, divTemp[p], 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
    divEne[p]=(E_total/(Msteps*N))*0.5;
    E2_media=(E2_total/(Msteps*N))*0.25;
    M_media=M_total/(Msteps*N);
    M2_media=M2_total/(Msteps*N);
    divMag[p]=Mabs_total/(Msteps*N);
    divCal[p] = (E2_media-(divEne[p]*divEne[p]*N))/(k*divTemp[p]*divTemp[p]);
    divSusc[p] = (M2_media-(M_media*M_media*N))/(k*divTemp[p]);
}
```

Figura 26 – Terceira mudança

Nesse ponto cada core, que já recebeu sua respectiva faixa de valores de temperatura, executa o loop com os valores dessa faixa recebida e salva as observáveis (energia, magnetização, calor específico e susceptibilidade magnética) nos seus vetores individuais.

```
//barreira para garantir que todos cores já terminaram seus processos
MPI_Barrier(MPI_COMM_WORLD);

//envia as fatias dos vetores gerados para o core MESTRE
MPI_Gather(divEne, div, MPI_DOUBLE, Mabs_media, div, MPI_DOUBLE, MESTRE, MPI_COMM_WORLD);
MPI_Gather(divMag, div, MPI_DOUBLE, E_media, div, MPI_DOUBLE, MESTRE, MPI_COMM_WORLD);
MPI_Gather(divCal, div, MPI_DOUBLE, calor_esp, div, MPI_DOUBLE, MESTRE, MPI_COMM_WORLD);
MPI_Gather(divSusc, div, MPI_DOUBLE, susc_mag, div, MPI_DOUBLE, MESTRE, MPI_COMM_WORLD);

//checa se o processo esta sendo executado no core mestre, se sim, salva os valores gerados no arquivo
if(rank == MESTRE) {
    arq = fopen("dados.dat", "w");                    //arquivo para armazenagem de dados

    for(i=0; i<=(qtdT-count); i++)
        fprintf(arq, "%.2lf %lf %lf %lf %lf\n", Temperatura[i], Mabs_media[i], E_media[i], calor_esp[i], susc_mag[i]);

    fclose(arq);

    printf("Fim do programa.\nObserváveis salvos no arquivo 'dados.dat'\nTabelas na ordem: Temperatura | <Magnetização
}

MPI_Finalize();

return 0;
```

Figura 27 – Quarta mudança

Na última modificação realizada, inicialmente se insere uma barreira para garantir que todos os cores já executaram suas tarefas. Após isso, cada core envia os "pedaços"de vetores que armazenam as observáveis de interesse para o core mestre, o qual forma o vetor "completo"com todos valores das observáveis para todos valores de temperaturas. Após isso, o core mestre salva os valores em um arquivo.

Todo esse processo pode ser exemplificado como mostra a figura [28].

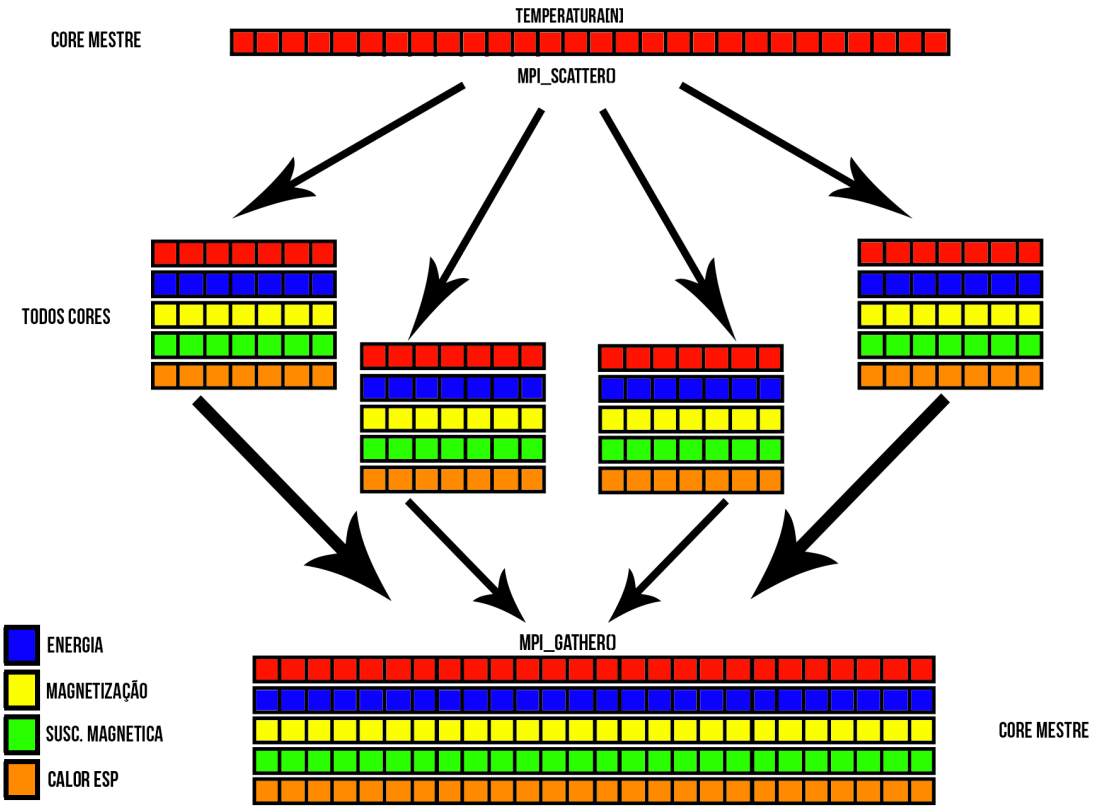

Figura 28 – Exemplificação da paralelização

8 Benchmark do Programa com Várias Threads

8.1 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. Como o compilador ifort demonstrou melhores resultados, a análise de performance da paralelização será feita com o mesmo.

Os tempos obtidos variando o número de threads estão na tabela [10].

Threads	8	7	6	5	4	3	2	1
Sem flags	2852s	3728s	4067s	4685s	4766s	5030s	5188s	6844s
Melhores flags	2337s	2880s	3142s	3161s	3327s	3693s	4586s	6181s

Tabela 10 – Tempos obtidos variando o número de threads usando o ifort

Pode-se comparar os tempos obtidos a fim de entender as diferenças variando o número de threads e isso está demonstrado em [29].

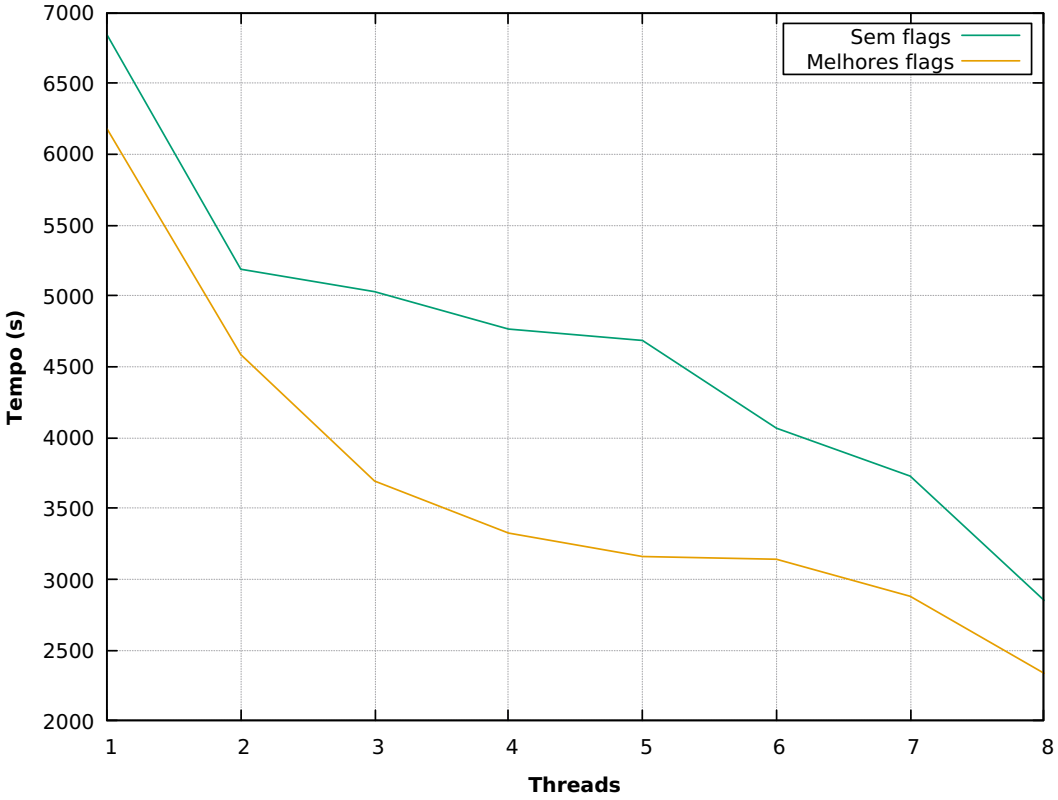

Figura 29 – Tempo pelo número de threads

Onde nota-se um padrão esperado: 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. Pode-se notar que a maior diferença ocorre quando mudamos de 1 para 2 thread, o que faz sentido, já que abrir ambientes paralelos demanda tempo e poder computacional, isso significa que usar 1 thread com um programa paralelizado não tem justificativa, mas com 2 já começa a valer a pena. Como era de se esperar, o melhor tempo acontece utilizando as melhores flags e 8 threads, sendo de **2337s**.

Afim de entender essa melhora no tempo de execução conforme o número de threads aumenta, pode-se calcular a eficiência usando a equação (4.1) e usando $T_{no-opt} = 10505$s (tempo de execução do programa em serial com o compilador GCC). Isso está descrito em [30].

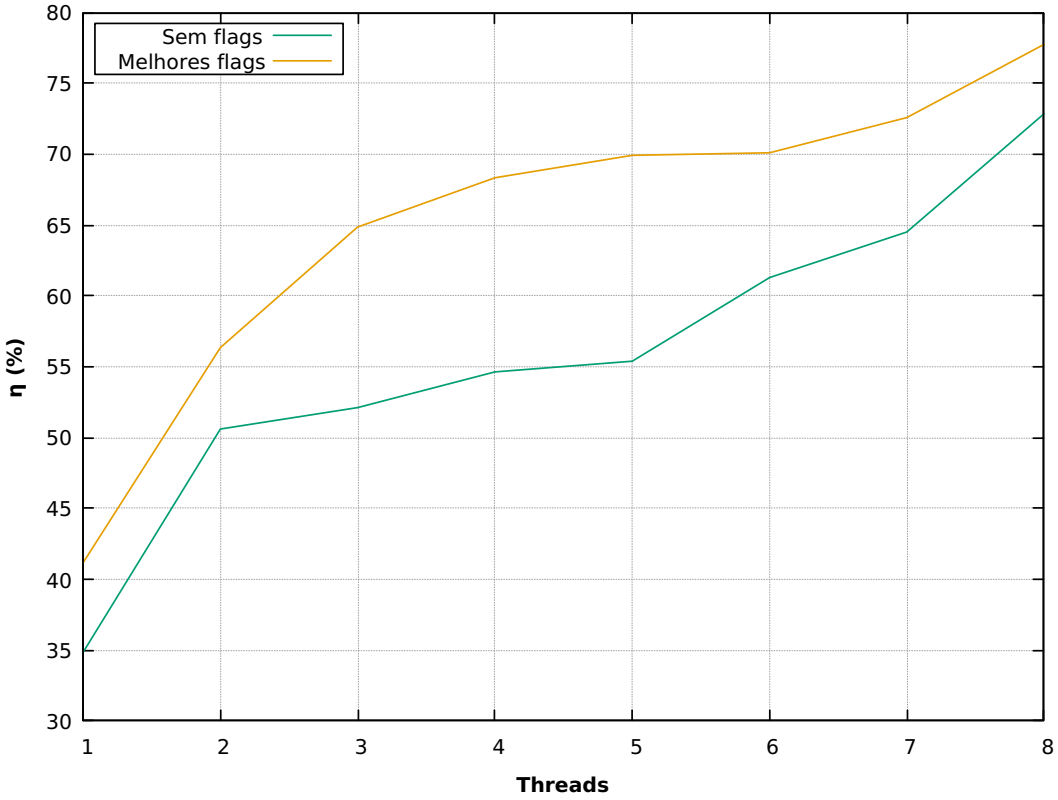

Figura 30 – Eficiência pelo número de threads

No primeiro momento 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 compilador utilizado para obter esse resultado foi o ifort, enquanto o tempo usado como "pior tempo"foi obtido utilizando o GCC, e, pelas análises do Benchmark, foi possível notar uma diferença razoável entre os tempos de execução dos dois compiladores (o ifort sempre gera menores tempos). Se compararmos o caso com 1 thread com o tempo de execução do algoritmo utilizando o compilador ifort, 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 utilizando o compilador GCC foi para manter coerência com as demais análises feitas anteriormente.

Portanto, analisando as eficiências, percebe-se que, no pior caso (1 thread) já existe um ganho de desempenho de aproximadamente 41% utilizando as melhores flags e essa eficiência só aumenta conforme o número de threads também aumenta, chegando, nos melhores casos (com 8 threads), em 73% sem flags e 78% com as melhores flags. Isso significa que o tempo do programa do programa reduz em torno de 3/4 aplicando todas as técnicas de otimização em conjunto com a paralelização em OpenMP.

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 [31].

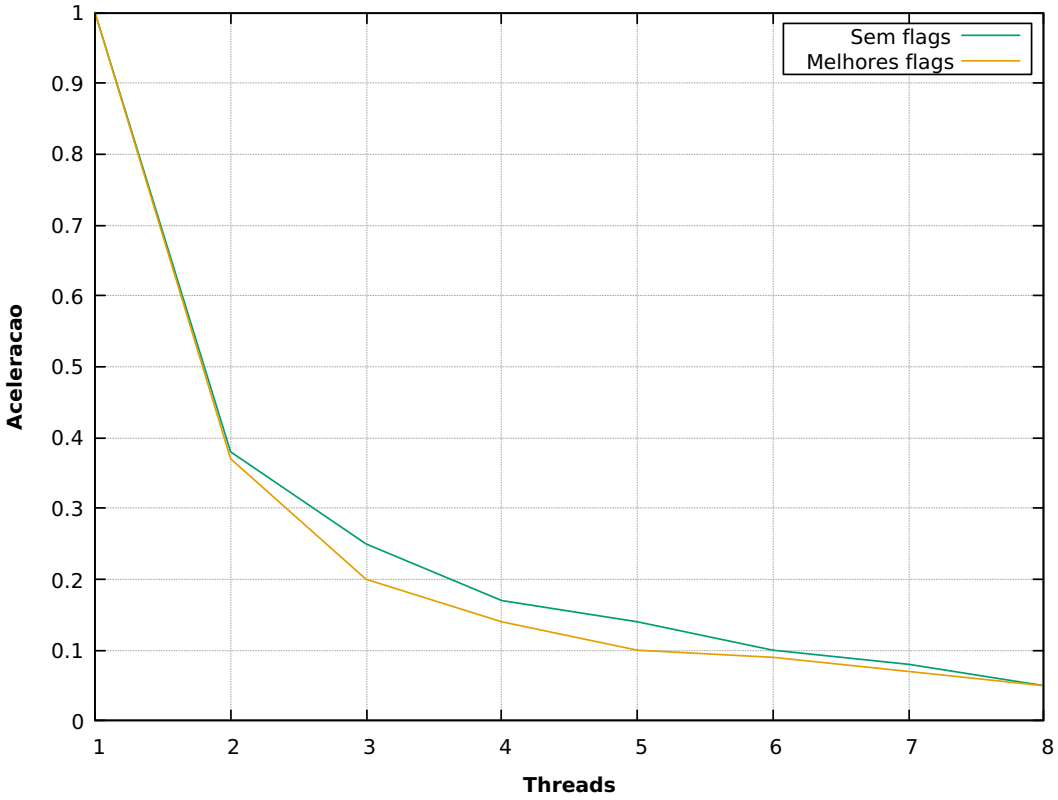

Figura 31 – Aceleração por thread (normalizada)

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. Isso demonstra que após um número X de threads, o valor da aceleração seria próximo a zero e teríamos a saturação do programa. Isto é, quanto mais threads são utilizadas, 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 0.05% do tempo, enquanto que, no caso utilizando 2 threads, cada thread é responsável por acelerar cerca de 0.38%. 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.

Outro resultado interessante é 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). Entretanto, como o algoritmo não funcionou no LNCC e não há um acesso possível a outra máquina com mais threads, isso fica somente como previsão.

Por fim, pode-se avaliar o *speed up* que o programa possui aumentando o número

de threads. O cálculo do *speed up* foi feito da seguinte forma

$$S = \frac{T_s}{T_p} \tag{8.1}$$

onde T_s é o tempo obtido na execução em serial e T_p é o tempo obtido em paralelo.

Evidentemente que o tempo utilizado foi o mesmo do que o utilizado para o cálculo da eficiência. O resultado obtido pode ser visto em [32].

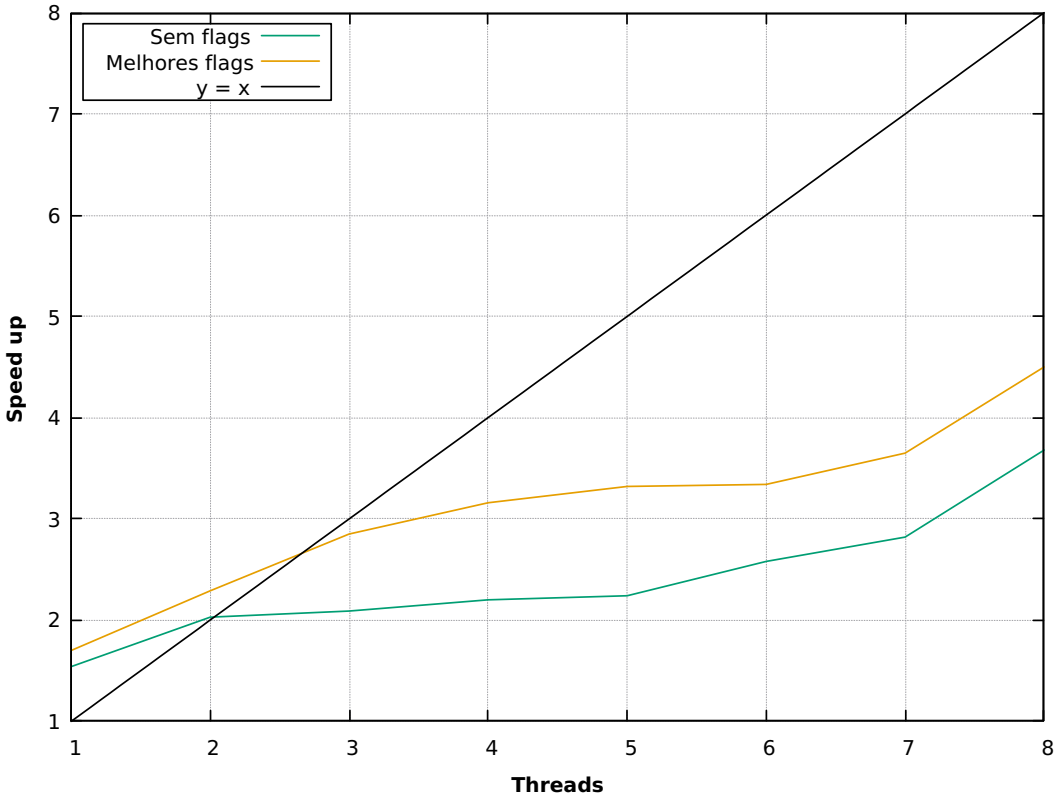

Figura 32 – *Speed up* por thread

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 sempre 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 $T = \frac{X}{N}$ para qualquer algoritmo e isso é claramente falso como pode-se ver no gráfico anterior. A proposição pode ser verdadeira, entretanto, não necessariamente é. O ganho de tempo está totalmente longe de ser linear e só ultrassa a função $f(x) = x$ nos casos com 1 e 2 threads. Logo, esse pensamento, apesar de ser algo natural de se acontecer, está equivocado.

8.2 LNCC

Infelizmente, mesmo após a mudança no código do algoritmo, o programa continua a não funcionar no B710 ou na SequanaX (o tempo aumenta conforme aumenta-se o número de threads) e por isso a análise não será efetuada.

9 Benchmark do Programa com Vários Cores

9.1 Lab107C

Analogamente ao processo feito com o código paralelizado em OpenMP, também pode-se realizar o processo com o algoritmo paralelizado em MPI. Da mesma forma, como o compilador ifort demonstrou melhores resultados, a análise de performance da paralelização será feita com o mesmo.

Os tempos obtidos variando o número de cores estão na tabela [11].

Cores	8	7	6	5	4	3	2	1
Sem flags	1651s	1677s	1890s	2062s	1991s	2452s	3647s	7081s
Melhores flags	1446s	1455s	1637s	1838s	1837s	2249s	3277s	6126s

Tabela 11 – Tempos obtidos variando o número de cores usando o ifort

Portanto, pode-se comparar os tempos obtidos a fim de entender as diferenças variando o número de cores e isso está demonstrado em [33].

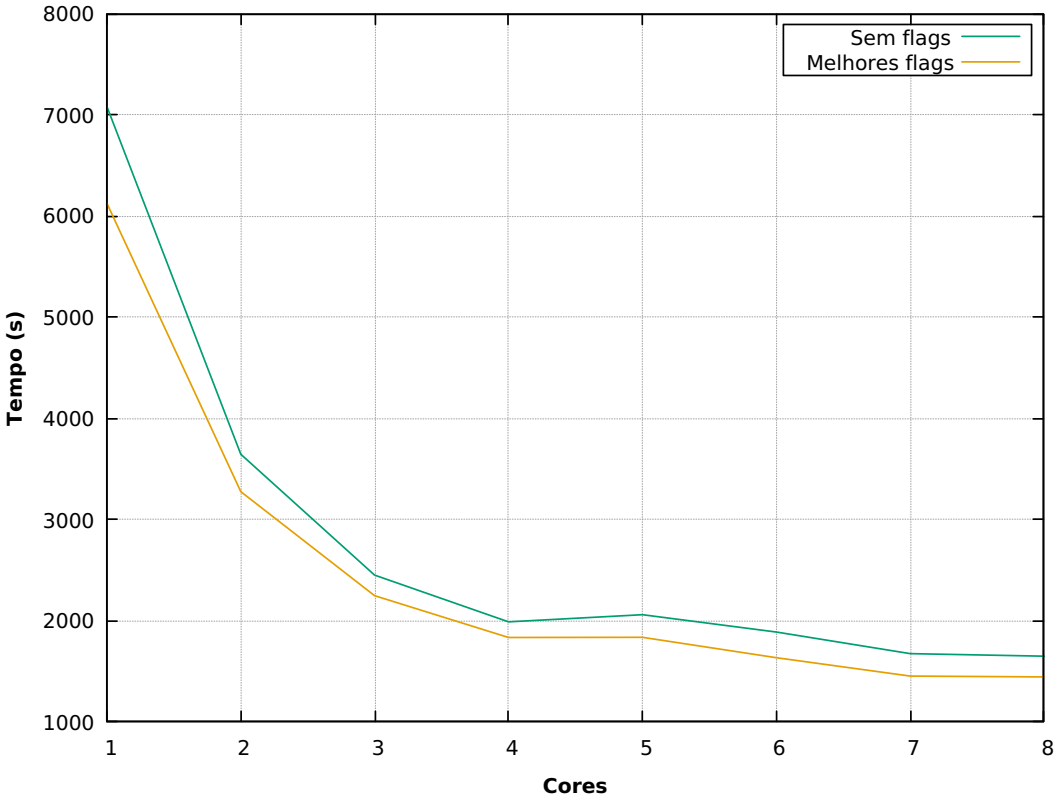

Figura 33 – Tempo pelo número de cores

Onde nota-se, no geral, um padrão esperado: conforme o número de cores aumenta, o tempo de execução diminui (com exceção do caso com 5 cores). Esse padrão acontece tanto nos casos sem flags quanto nos casos com as melhores flags. Pode-se notar que a maior diferença também ocorre quando mudamos de 1 para 2 cores (e é ainda mais acentuada), o que já mostra que, em MPI, mesmo com uma máquina com somente 2 cores, o ganho de desempenho já é bem interessante. Como era de se esperar, o melhor tempo acontece utilizando as melhores flags e 8 cores, sendo de **1446s**.

Afim de entender essa melhora no tempo de execução conforme o número de cores aumenta, pode-se calcular a eficiência usando a equação (4.1) e usando $T_{no-opt} = 10505$s (tempo de execução do programa em serial com o compilador GCC). Isso está descrito em [34].

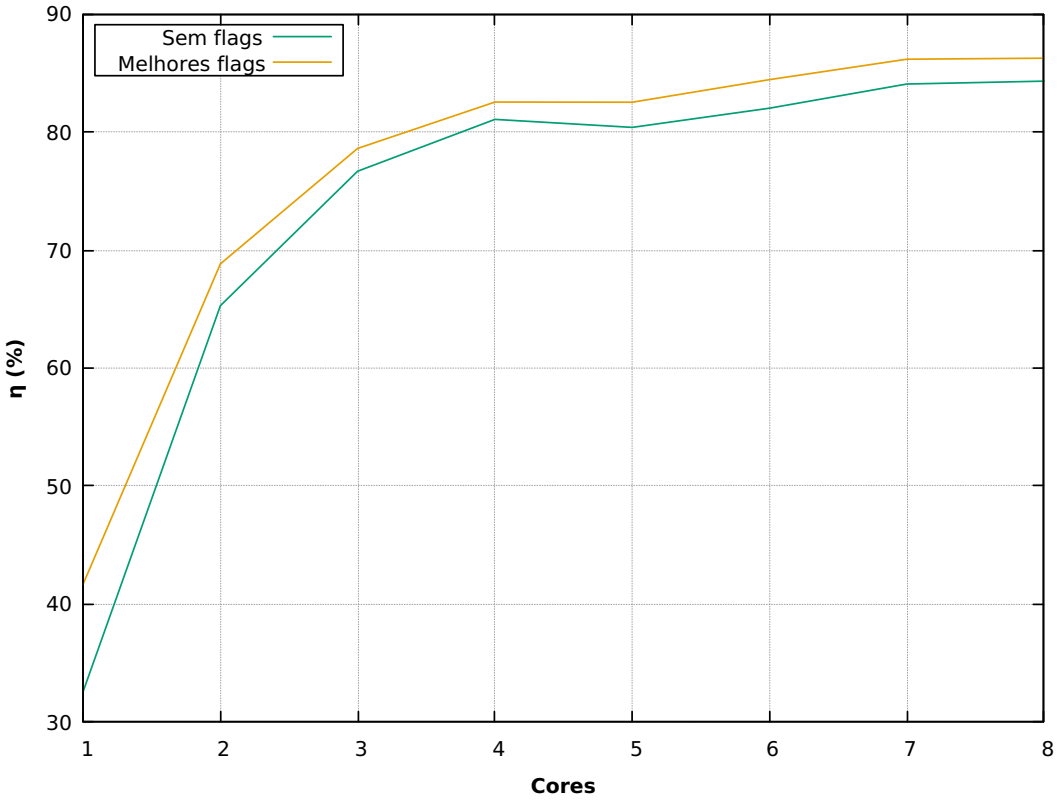

Figura 34 – Eficiência pelo número de cores

Portanto, analisando as eficiências, percebe-se que, no pior caso (1 core) já existe um ganho de desempenho de aproximadamente 42% utilizando as melhores flags e essa eficiência só aumenta conforme o número de cores também aumenta (com exceção do caso com 5 cores), chegando, nos melhores casos (com 8 cores), em 84% sem flags e 86% com as melhores flags. Este resultado mostra que com uma máquina razoavelmente acessível já é possível reduzir em quase 90% o tempo de execução do programa e isso já é um resultado

muito bom.

Após a análise da eficiência, pode-se analisar a aceleração por core (normalizada com o valor de 1 core), esse resultado pode ser visto em [35].

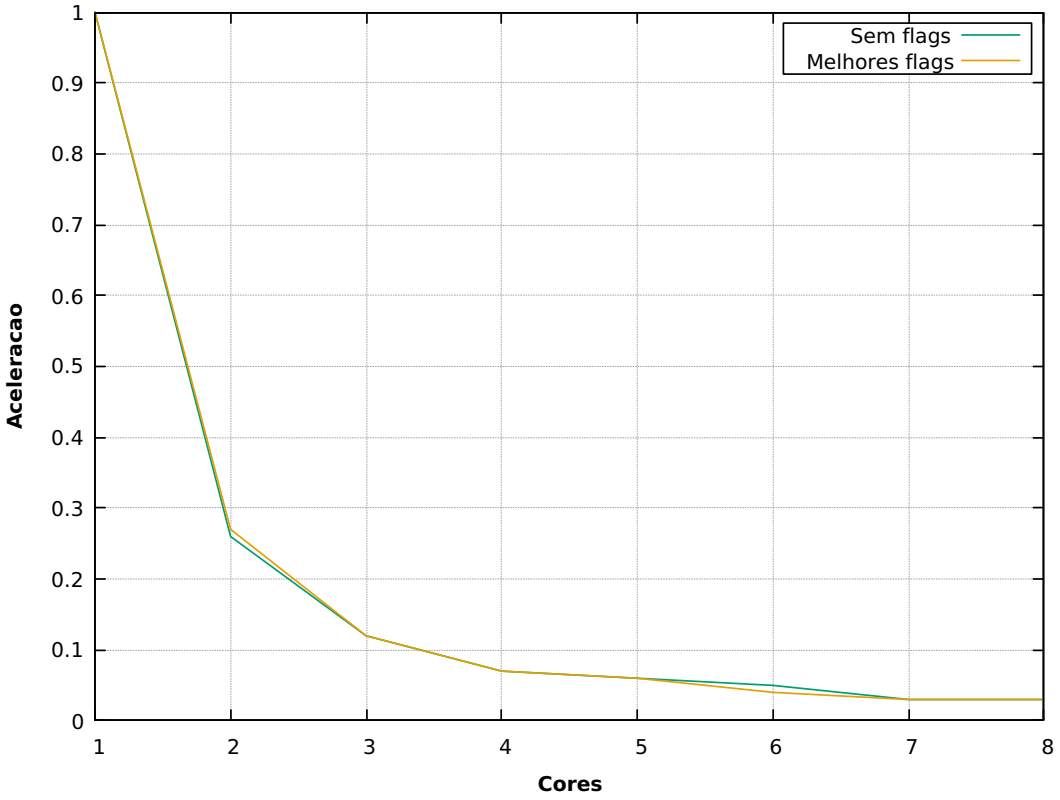

Figura 35 – Aceleração por core (normalizada)

O comportamento também é bem semelhante a uma exponencial e em ambos os casos o comportamento é quase idêntico (as curvas praticamente se sobrepõem), além disso, também é possível notar que conforme o número de cores aumenta, a aceleração individual dos cores perde potência, isto é, após um valor X de cores, isso estaria bem próximo de zero e teríamos a saturação do programa. Isso se mostra natural quando, com 8 cores, esse valor já está bem próximo a zero, ou seja, provavelmente a saturação ocorrerá não muito longe e isso poderá ser confirmado com os resultados no LNCC, que serão demonstrados a frente.

Por fim, pode-se avaliar o *speed up* utilizando a equação (8.1). Evidentemente que o tempo utilizado foi o mesmo do que o utilizado para o cálculo da eficiência e o resultado obtido pode ser visto em [36].

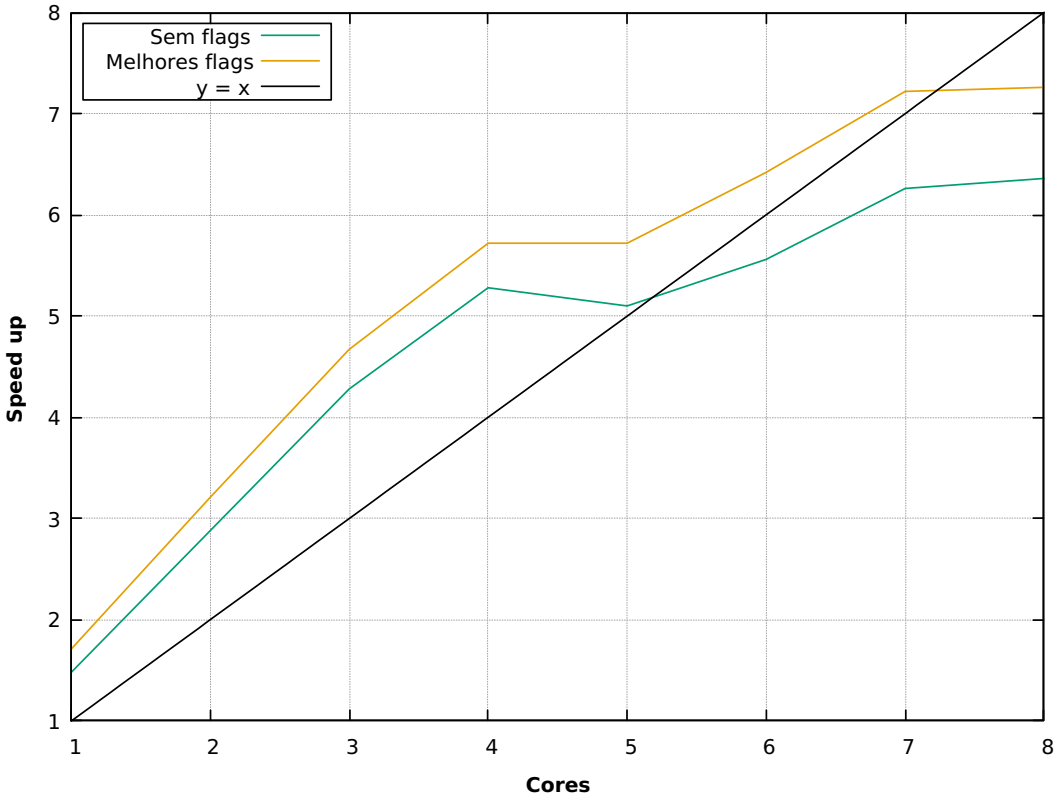

Figura 36 – *Speed up* por core

O resultado obtido na paralelização com o MPI é bem interessante: o *speed up* ultrapassa a função $f(x)$ em diversos casos (utilizando as melhores flags isso acontece até 7 cores e sem flags, até 5 cores). Ou seja, a ideia de que o tempo de execução seria dividido exatamente pelo número de cores utilizados em um programa paralelizado pode ser sim observado em alguns casos, como nessa implementação com MPI. Entretanto, é evidente (pelos contra exemplos) que isso não é um padrão observado no geral.

9.2 LNCC

O processo realizado no Lab107C também foi repetido no LNCC já que o código paralelizado em MPI funcionou corretamente nas máquinas presentes no LNCC (diferentemente do código paralelizado em OpenMP). A única diferença é que, assim como no benchmark, a malha utilizada foi com $L = 128$.

9.2.1 B710

Os tempos obtidos variando o número de cores na máquina B710 estão na tabela [12].

Cores	24	20	16	12	8	6	4	2	1
Sem flags	21s	31s	32s	40s	58s	74s	106s	203s	364s
Melhores flags	19s	27s	28s	36s	52s	67s	94s	178s	324s

Tabela 12 – Tempos obtidos variando o número de cores usando o ifort

Portanto, pode-se comparar os tempos obtidos a fim de entender as diferenças variando o número de cores e isso está demonstrado em [37].

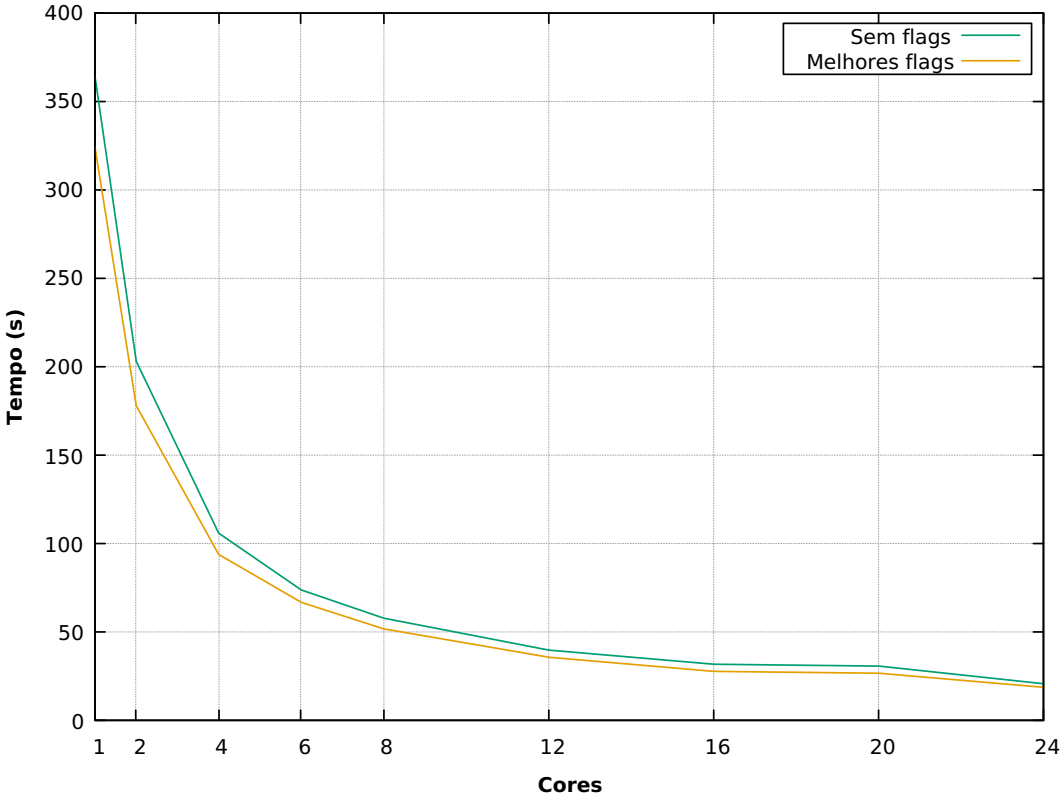

Figura 37 – Tempo pelo número de cores

Aqui nota-se que o comportamento do tempo é bem claro: conforme o número de cores aumenta, o tempo de execução diminui, assim como o resultado obtido no Lab107C, e esse padrão é repetido em ambos os casos (com ou sem flags). A maior diferença também ocorre quando mudamos de 1 para 2 cores (e também é acentuada), o que corrobora que, em MPI, mesmo com uma máquina com somente 2 cores, o ganho de desempenho já é bem interessante. Como era de se esperar, o melhor tempo acontece utilizando as melhores flags e 24 cores, sendo de **19s**. Um fato interessante a ser notado é que o ganho de tempo a partir de 16 cores já diminui bastante conforme o número de cores aumenta (a curva começa a ser mais horizontal).

Afim de entender essa melhora no tempo de execução conforme o número de cores aumenta, pode-se calcular a eficiência usando a equação (4.1) e usando $T_{no-opt} = 581$s

(tempo de execução do programa em serial com o compilador GCC). Isso está descrito em [38].

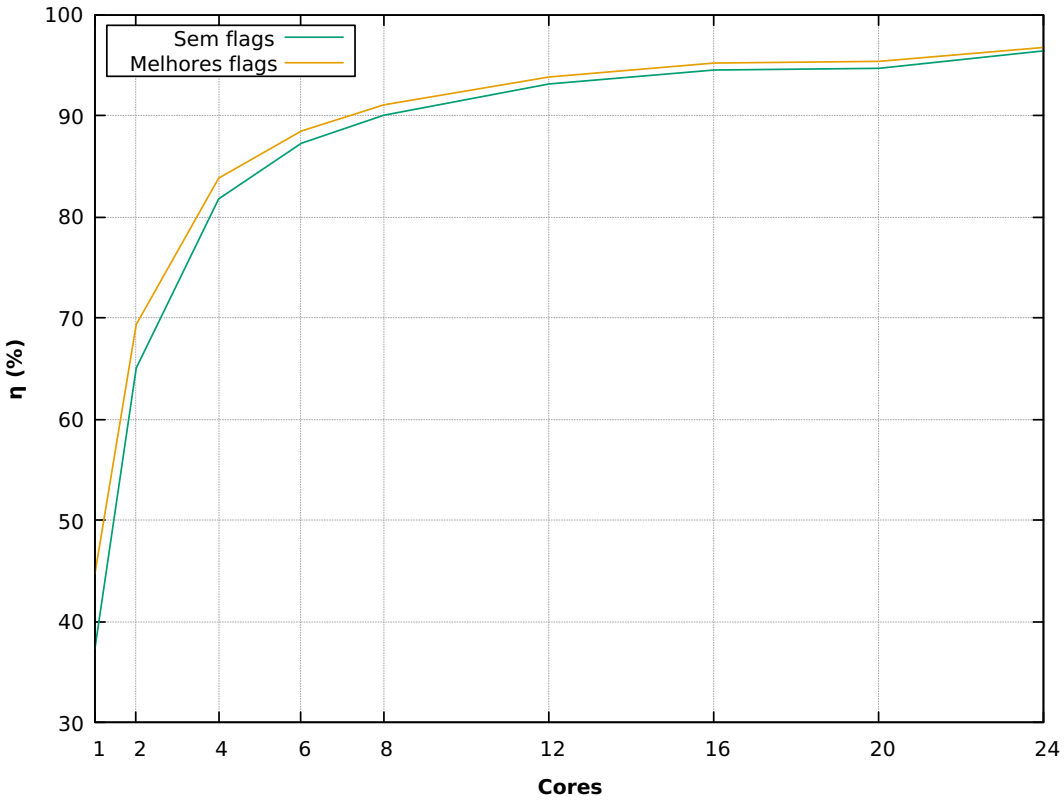

Figura 38 – Eficiência pelo número de cores

Onde pode-se notar que as eficiências obtidas não destoam muito do resultado obtido no Lab107C. Com 1 core a eficiência fica em cerca de 45% utilizando as melhores flags e também aumenta conforme o número de cores cresce. Chegando, nos melhores casos (com 24 cores) em 96% sem flags e 97% com as melhores flags. Ou seja, basicamente é possível melhorar a eficiência em 10% se tivermos +16 cores disponíveis em uma máquina (já que com 8 cores no Lab107C a eficiência alcançada foi de aproximadamente 86%). Obviamente todo ganho é útil se existir a possibilidade de acessar uma máquina melhor, entretanto, conforme pode ser notado, esse ganho passa a ser cada vez menor conforme mais cores são utilizados.

Após a análise da eficiência, pode-se analisar a aceleração por core (normalizada com o valor de 1 core), esse resultado pode ser visto em [39].

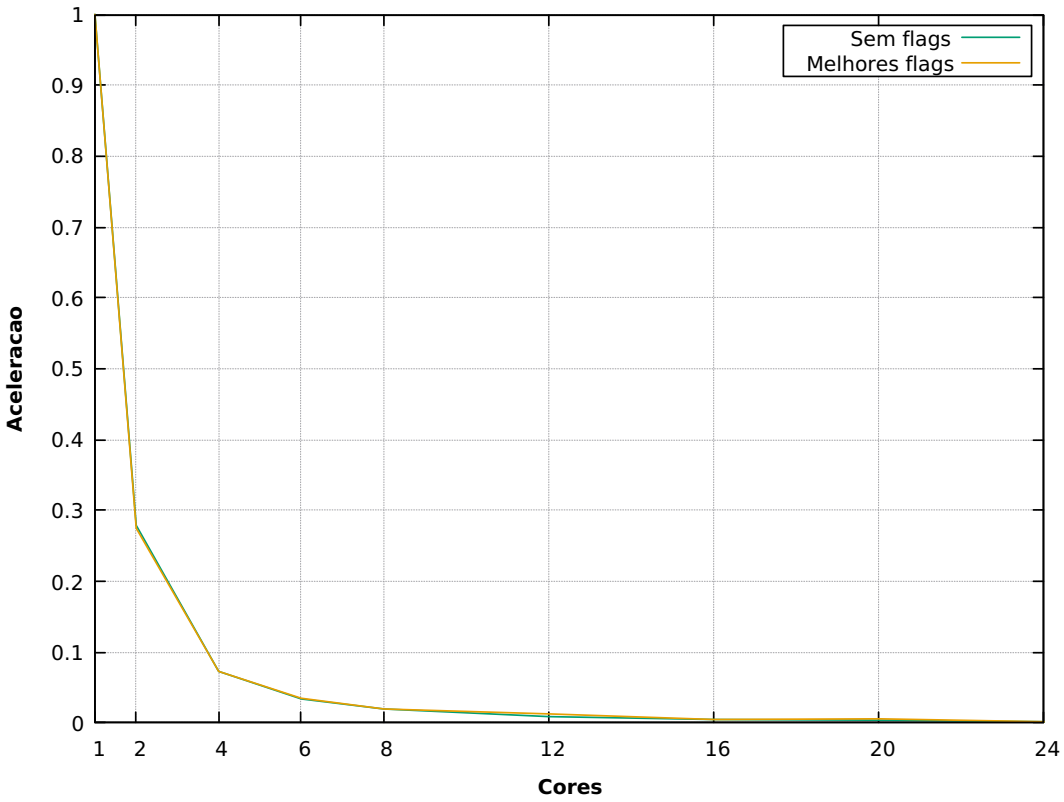

Figura 39 – Aceleração por core (normalizada)

O comportamento também é bem semelhante a uma exponencial e em ambos os casos (com ou sem flags) o comportamento é idêntico (as curvas se sobrepõem), além disso, novamente é possível notar que conforme o número de cores aumenta, a aceleração individual dos cores perde potência. Na B710 o número de cores já é mais expressivo e a previsão feita com os resultados do Lab107C já podem ser confirmadas: o algoritmo basicamente satura após 16 cores. A diminuição do tempo ainda ocorre, mas é praticamente ínfima. Isso nos diz que, utilizar uma máquina com mais de 16 cores é desnecessário e basicamente um desperdício.

Por fim, pode-se avaliar o *speed up* utilizando a equação (8.1). Evidentemente que o tempo utilizado foi o mesmo do que o utilizado para o cálculo da eficiência e o resultado obtido pode ser visto em [40].

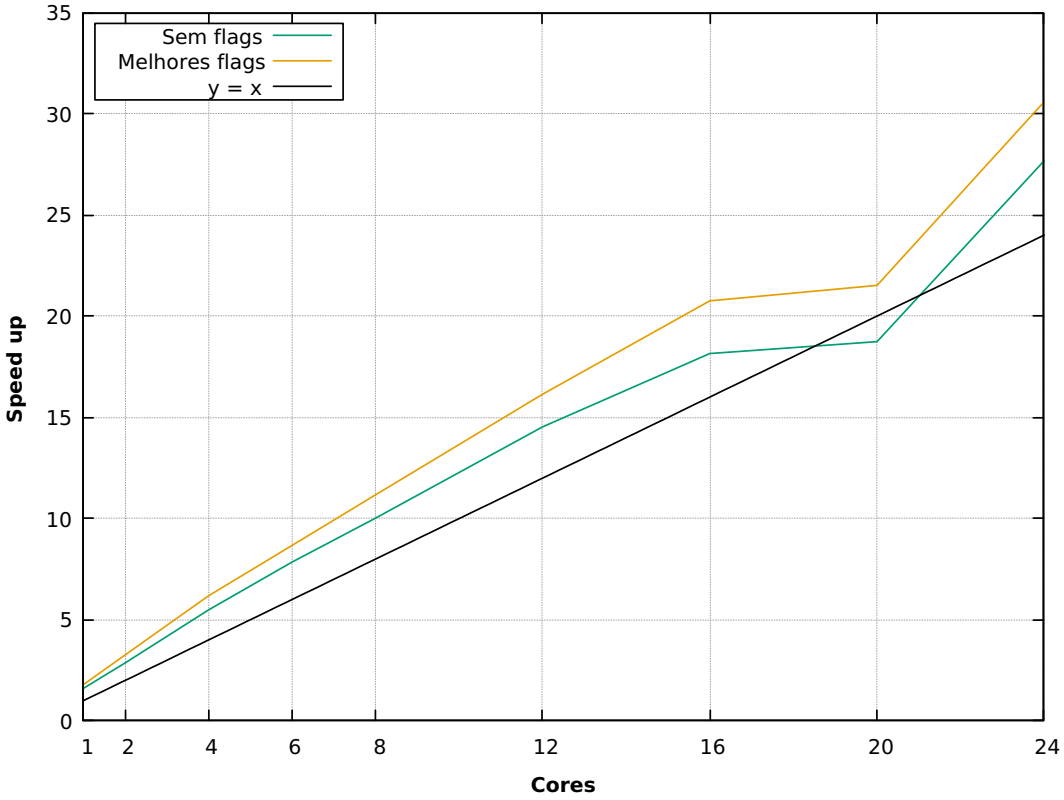

Figura 40 – *Speed up* por core

Assim como no caso obtido no Lab107C, o resultado obtido na paralelização com o MPI também se demonstrou parecido na B710: O *speed up* ultrapassa a função $f(x)$ em diversos casos (utilizando as melhores flags isso acontece em todos as combinações de cores e sem flags só não acontece com 20 cores). Ou seja, novamente demonstrando que a ideia de que o tempo de execução seria dividido exatamente pelo número de cores utilizados em um programa paralelizado pode ser sim observado em alguns casos.

9.2.2 SequanaX

Os tempos obtidos variando o número de cores na máquina SequanaX estão na tabela [13].

Cores	Sem flags	Melhores flags
48	11s	10s
44	19s	17s
40	18s	16s
36	17s	16s
32	17s	15s
28	18s	15s
24	16s	15s
20	24s	20s
16	22s	20s
12	30s	26s
8	43s	38s
6	54s	48s
4	77s	69s
2	153s	131s
1	242s	210s

Tabela 13 – Tempos obtidos variando o número de cores usando o ifort

Portanto, pode-se comparar os tempos obtidos a fim de entender as diferenças variando o número de cores e isso está demonstrado em [41].

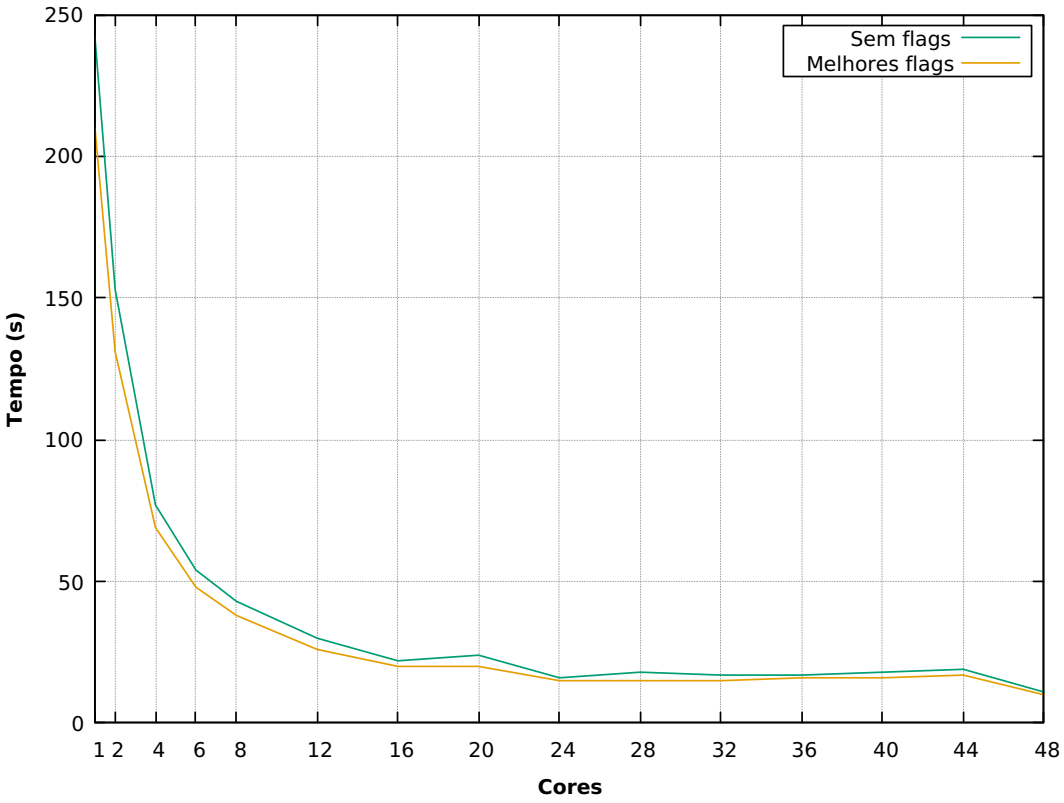

Figura 41 – Tempo pelo número de cores

O comportamento obtido na SequanaX basicamente só confirma os resultados

já vistos anteriormente no Lab107C e na B710: no geral, conforme o número de cores aumenta, o tempo de execução diminui e esse padrão é repetido em ambos os casos (com ou sem flags). A maior diferença também ocorre quando mudamos de 1 para 2 cores (e continua acentuada), novamente confirmando que, em MPI, mesmo com uma máquina com somente 2 cores, o ganho de desempenho já é bem interessante. Como era de se esperar, o melhor tempo acontece utilizando as melhores flags e 48 cores, sendo de **10s**. Outro comportamento repetido é o fato de que o ganho de tempo a partir de 16 cores começa a oscilar, corroborando com a saturação nesse número. Entretanto, em 48 cores, o tempo diminui novamente e isso muito possivelmente é devido especificamente ao número de passos utilizados no loop da paralelização (um comportamento bem individual e característico da forma de como o programa foi paralelizado e as constantes foram definidas, o que não anula a ideia de que o programa praticamente satura em 16 cores).

Afim de entender essa melhora no tempo de execução conforme o número de cores aumenta, pode-se calcular a eficiência usando a equação (4.1) e usando $T_{no-opt} = 442$s (tempo de execução do programa em serial com o compilador GCC). Isso está descrito em [42].

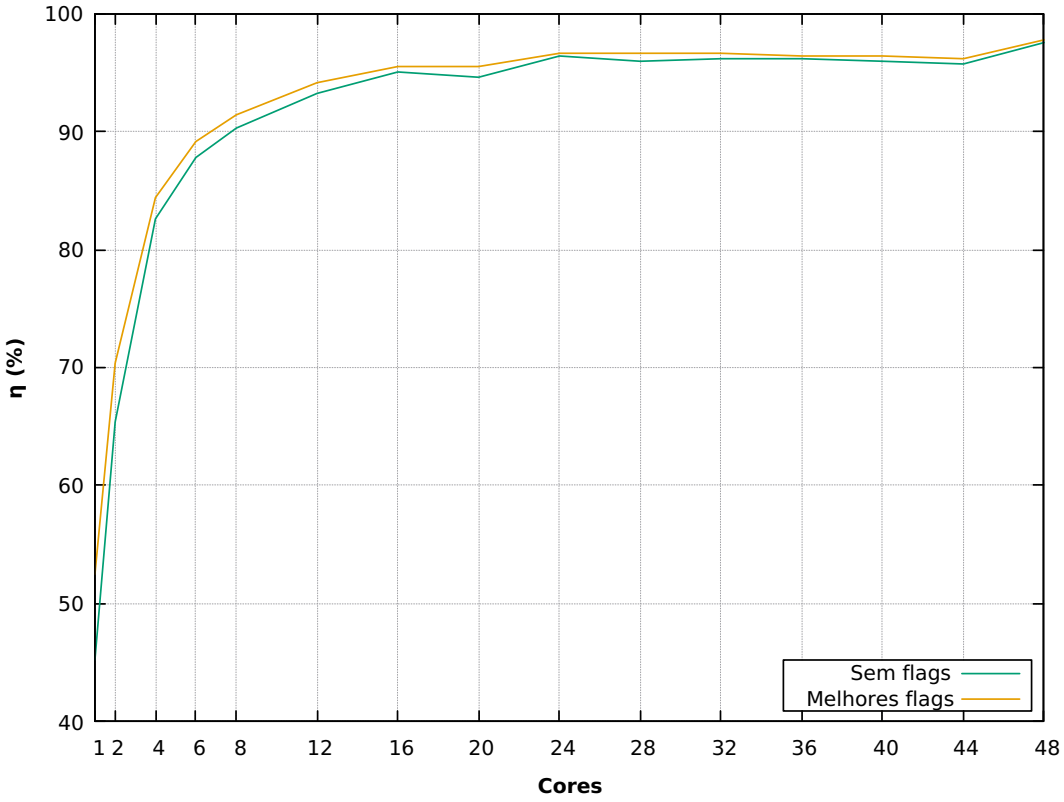

Figura 42 – Eficiência pelo número de cores

As eficiências obtidas na SequanaX são um pouco melhores que as obtidas anteriormente (muito provavelmente pelos componentes mais novos que a máquina possui)

mas também não destoam muito dos resultados obtidos anteriormente no Lab107C e na B710. Com 1 core a eficiência fica em cerca de 52% utilizando as melhores flags e também aumenta conforme o número de cores cresce. Chegando, nos melhores casos (com 48 cores) em 97% sem flags e 98% com as melhores flags. Ou seja, basicamente é possível melhorar a eficiência em aproximadamente 1% se tivermos +24 cores disponíveis em uma máquina (já que, com 24 cores na B710 a eficiência alcançada foi de aproximadamente 97%). Portanto, novamente entra o ponto de que utilizar uma máquina com muito mais cores não faz muito sentido nesse caso.

Após a análise da eficiência, pode-se analisar a aceleração por core (normalizada com o valor de 1 core), esse resultado pode ser visto em [43].

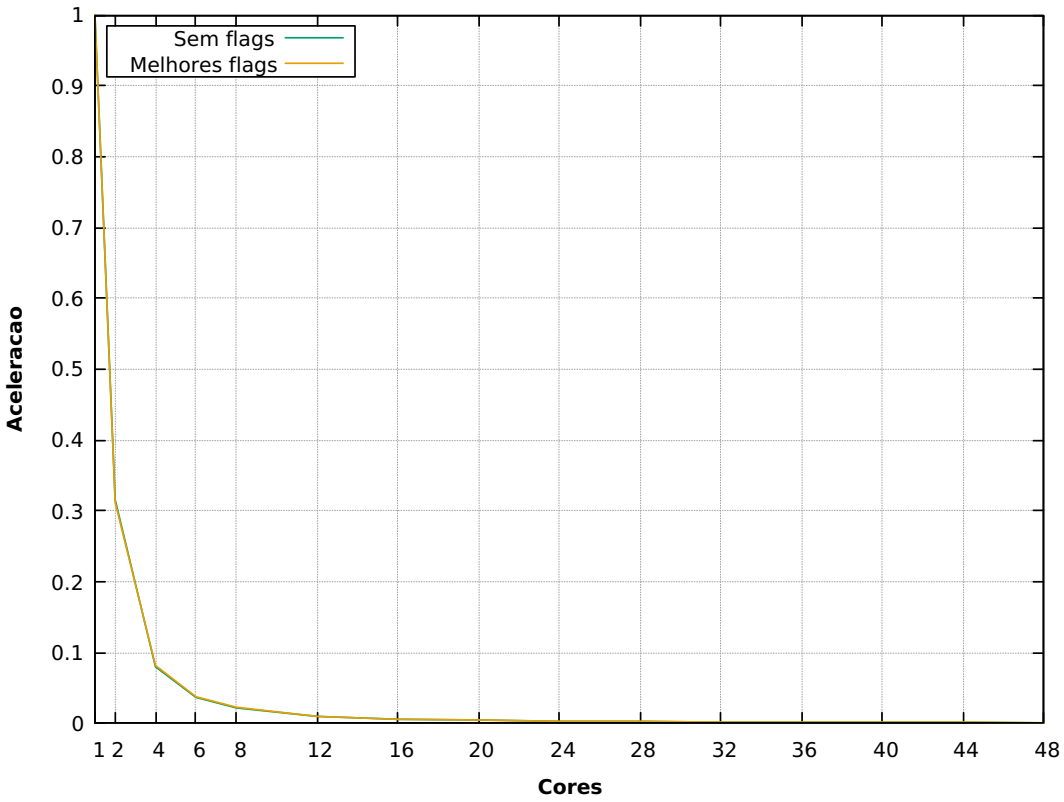

Figura 43 – Aceleração por core (normalizada)

Onde pode-se notar que o comportamento segue exatamente os resultados obtidos anteriormente no Lab107C e na B710: conforme o número de cores aumenta, a importância individual dos mesmos só diminui. Além disso, fica mais claro que utilizar uma máquina com mais de 16 cores não faz tanto sentido (já que a aceleração a partir de 16 cores tende a zero).

Por fim, pode-se avaliar o *speed up* utilizando a equação (8.1). Evidentemente que o tempo utilizado foi o mesmo do que o utilizado para o cálculo da eficiência e o resultado

obtido pode ser visto em [44].

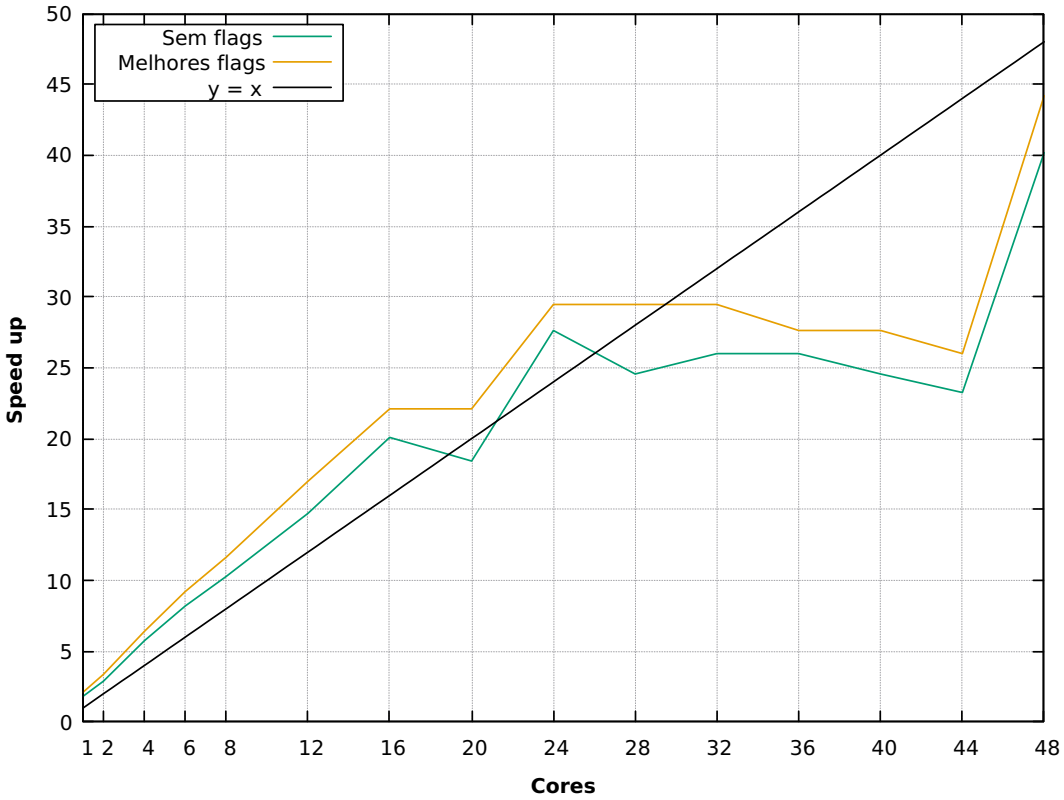

Figura 44 – *Speed up* por core

O resultado obtido na SequanaX foi um pouco diferente dos casos anteriores. O *speed up* ainda ultrapassa a função $f(x)$ em diversos casos (utilizando as melhores flags isso acontece até 28 cores e sem flags acontece até 16 e depois, excepcionalmente, com 24). Ou seja, novamente é possível visualizar que o tempo de execução pode ser dividido exatamente pelo número de cores utilizados em um programa paralelizado em alguns casos, entretanto, fica mais evidente na SequanaX que isso não acontece sempre e não poder ser adotado como um padrão.

10 Comparação entre as Paralelizações

10.1 Lab107C

Devido ao fato de que a paralelização em OpenMP não funcionou corretamente no LNCC, a comparação entre o desempenho das paralelizações em OpenMP e MPI só pode ser efetuada com os resultados obtidos no Lab107C. Para fins de comparação, somente os melhores resultados serão levados em conta, ou seja, os casos onde as melhores flags foram utilizadas.

Portanto, pode-se comparar os tempos obtidos em ambas formas de paralelização a fim de entender as diferenças. Isso está demonstrado em [45].

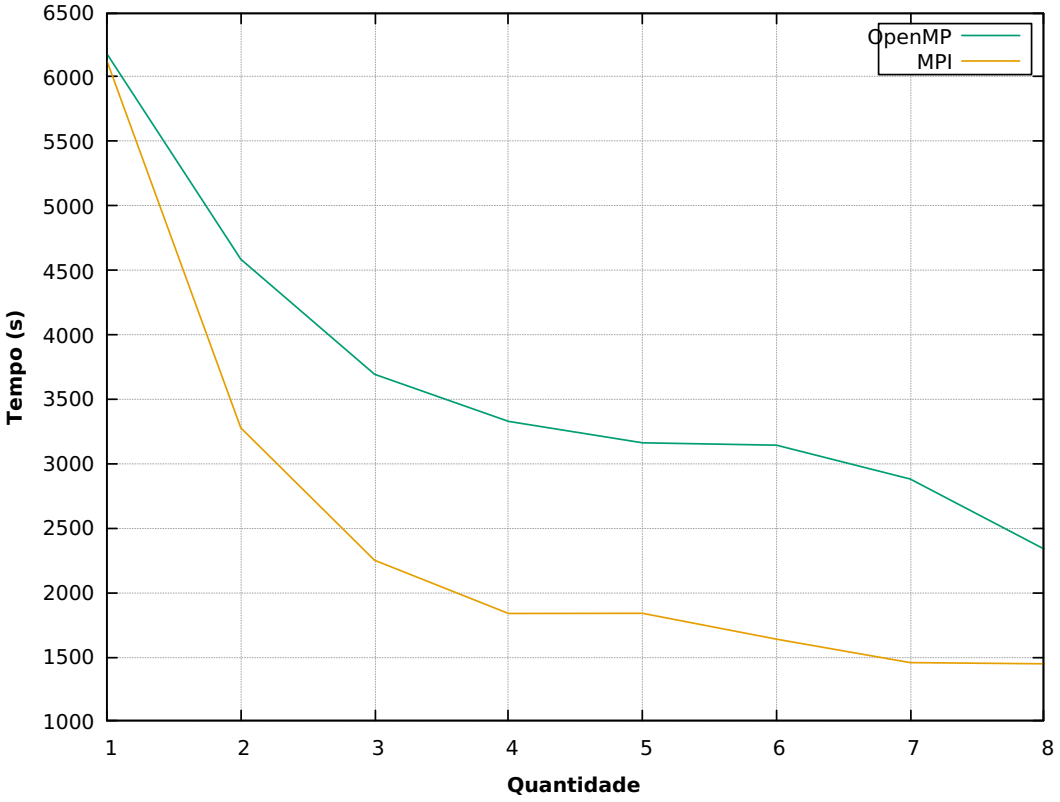

Figura 45 – Tempo pela quantidade (threads ou cores)

Onde é possível se notar que, indepentemente da quantidade (de cores ou threads) a API do MPI mostra melhores resultados nesse caso. Evidentemente que não é possível afirmar que isso é um padrão já que provavelmente depende da forma de como o programa foi paralelizado. No caso do algoritmo em questão é compreensível o OpenMP demonstrar resultados piores já que a paralelização nesse caso foi feita em um loop mais interno (isso custa tempo para abrir e fechar as seções paralelas) do que no caso com o MPI, que foi

feito no loop mais externo do algoritmo. Ainda sim, nesse caso particular, a utilização do MPI se mostrou mais vantajosa.

Afim de entender essa melhora no tempo de execução para os diferentes casos de paralelização, pode-se calcular a eficiência usando a equação (4.1) e usando $T_{no-opt} = 10505$s (tempo de execução do programa em serial com o compilador GCC). Isso está descrito em [46].

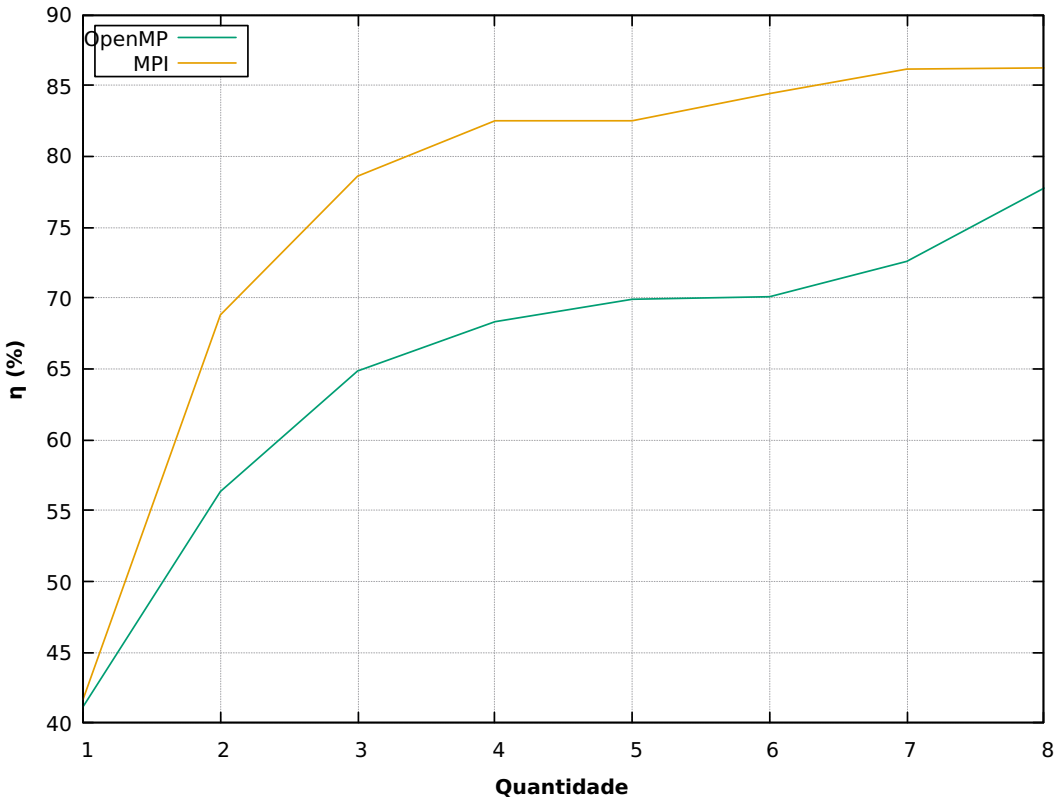

Figura 46 – Eficiência pela quantidade (threads ou cores)

Como era de se esperar, as eficiências ao se utilizar a API do MPI são maiores do que ao se utilizar o OpenMP. As eficiências só se aproximam no pior caso (com 1 core ou 1 thread) sendo de aproximadamente 42% em ambos casos, entretanto, no melhor caso (com 8 threads ou 8 cores) elas são de aproximadamente 86% e 78%, ou seja, o MPI se mostrou 8% mais eficiente no melhor caso. Contudo, essa diferença é ainda maior em outras quantidades, chegando a ser de aproximadamente 15% em quantidades intermediárias de threads ou cores.

Após a análise da eficiência, pode-se analisar a aceleração pela quantidade (normalizada com o valor de 1 core ou 1 thread), esse resultado pode ser visto em [47].

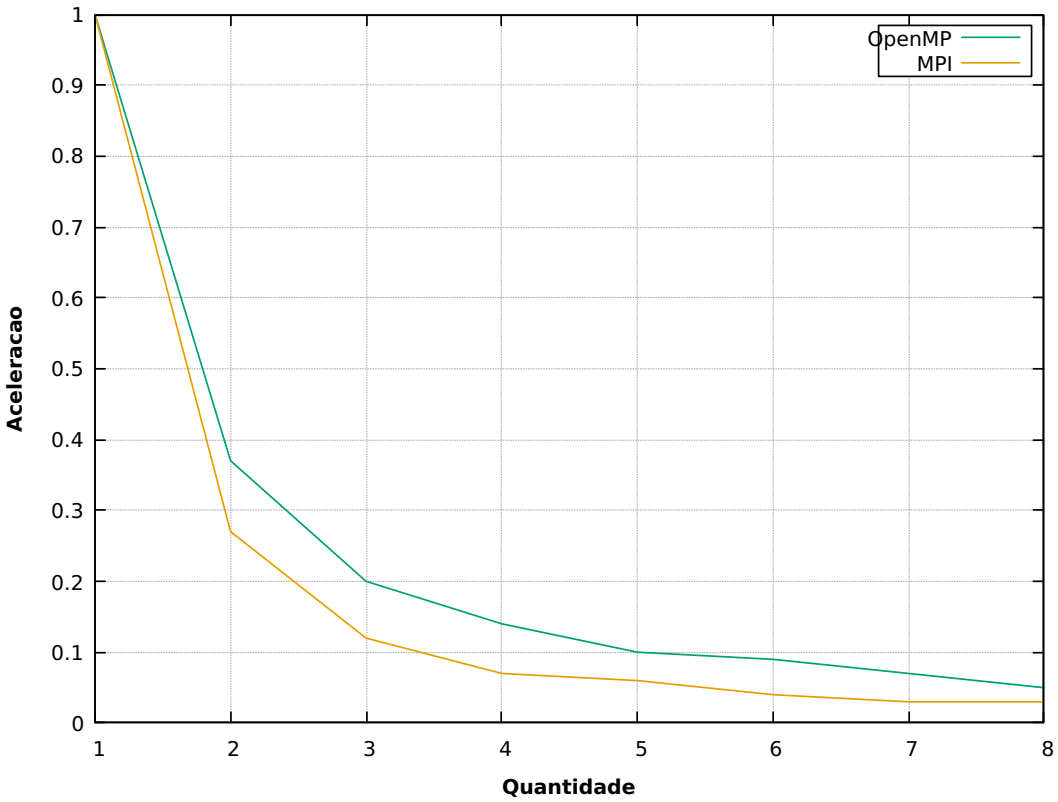

Figura 47 – Aceleração pela quantidade (normalizada)

Nesse caso, a aceleração utilizando o OpenMP se demonstrou maior em todos os casos se comparado quando o MPI foi utilizado. Isso nos diz duas coisas: (i) o programa deve demorar mais até atingir a saturação com o OpenMP (o que não pôde ser testado); (ii) a importância individual da threads se mantém por mais tempo se comparada com a importância individual dos cores.

Por fim, pode-se avaliar o *speed up* obtido com as duas formas de paralelização utilizando a equação (8.1). Evidentemente que o tempo utilizado foi o mesmo do que o utilizado para o cálculo da eficiência e o resultado obtido pode ser visto em [48].

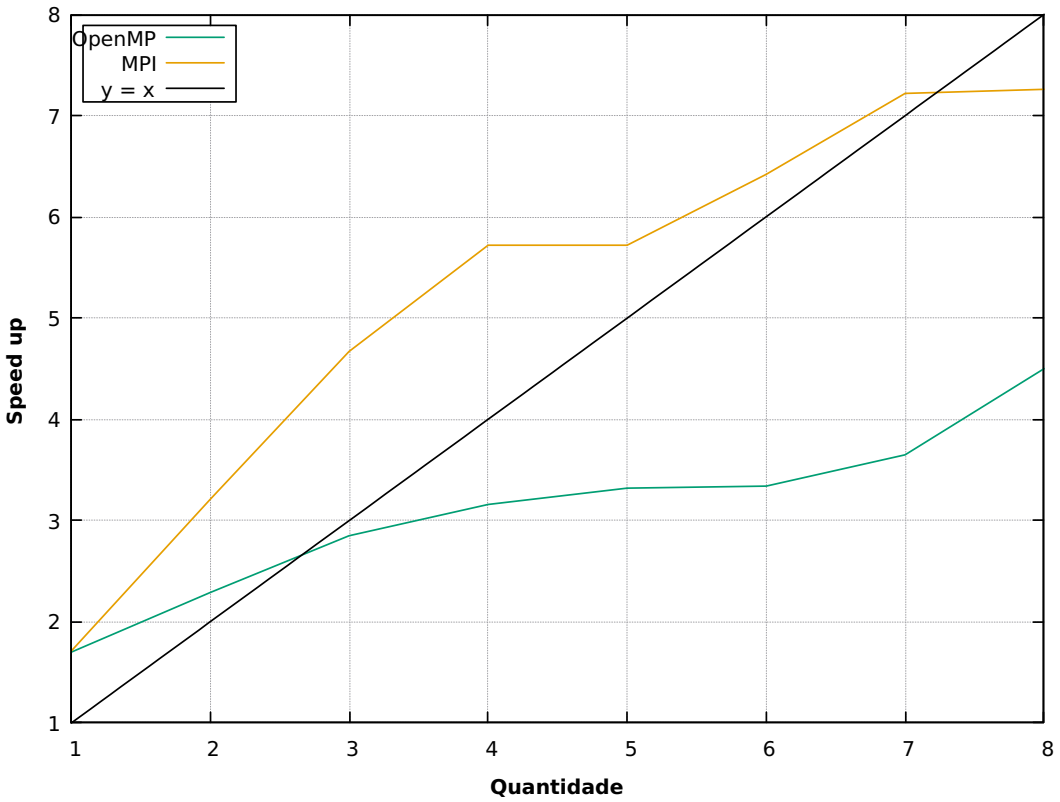

Figura 48 – *Speed up* pela quantidade (threads ou cores)

Por fim, o último resultado a ser comparado é o *speed up*, que, no caso quando o OpenMP foi utilizado mal ultrapassou a função $f(x)$ (somente em 1 caso, de 8), diferentemente do caso quando o MPI foi utilizado, que ultrapassou a função $f(x)$ em diversos pontos (em 7 de 8). Isso nos diz que o algoritmo paralelizado em MPI segue praticamente a ideia de que se utilizarmos uma maior quantidade, o tempo seria dividido por esse número, o que claramente não é o mesmo resultado observado ao se utilizar o OpenMP.

11 Discussão 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.

Então, o resultado obtido plotando a curva descrita pela solução analítica e o resultado numérico (do algoritmo em serial) é demonstrado nos gráficos [49] e [50].

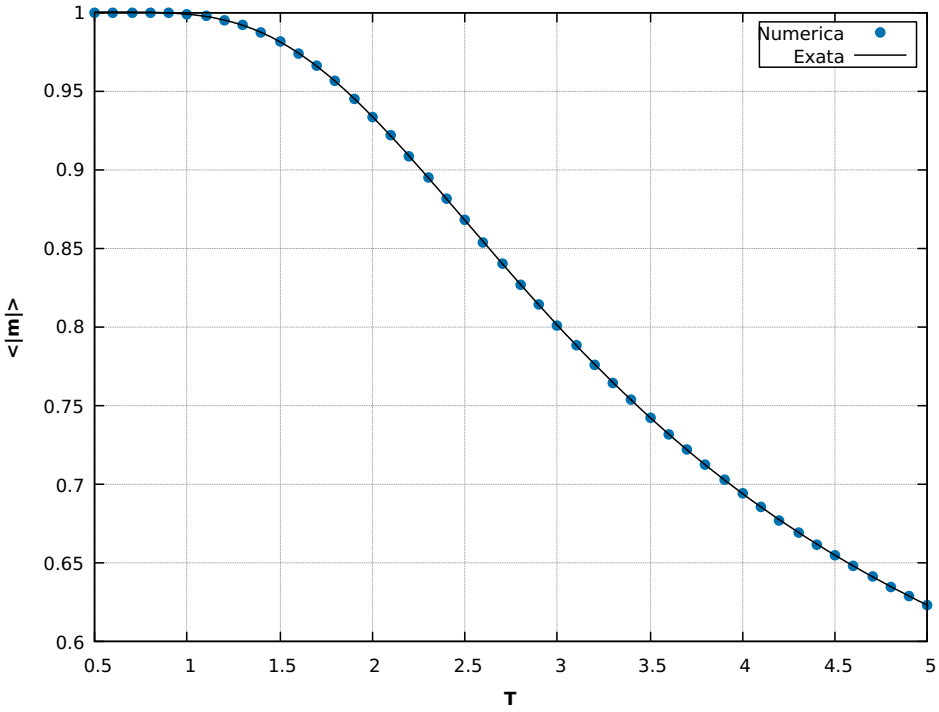

Figura 49 – Magnetização absoluta média por spin vs Temperatura - Solução numérica (Serial) e analítica

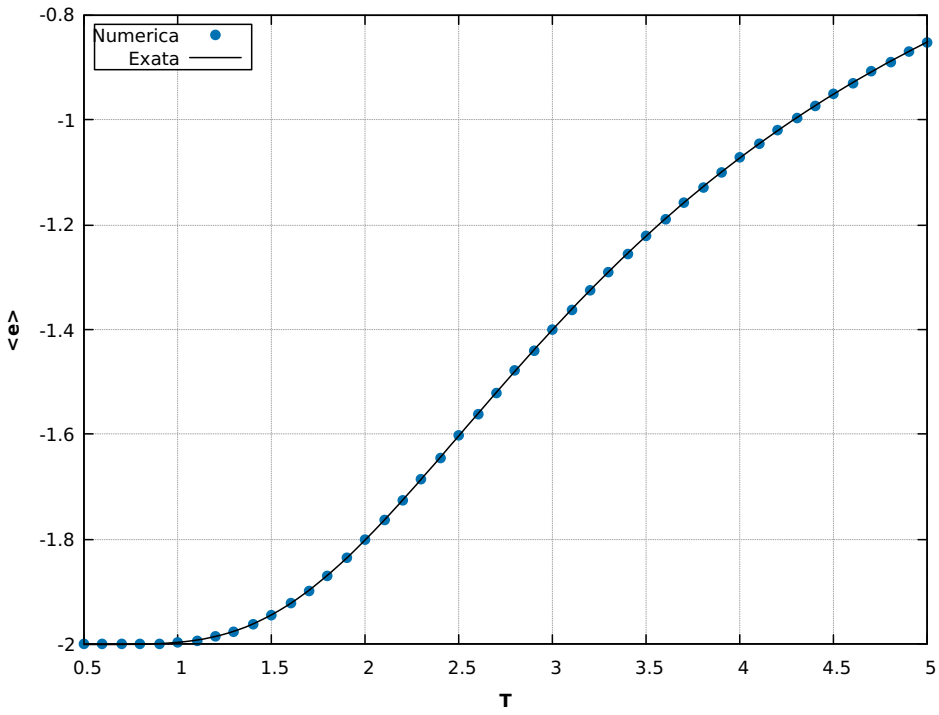

Figura 50 – Energia média por spin vs Temperatura - Solução numérica (Serial) e analítica

Onde nota-se que o modelo em serial 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.

Da mesma forma, pode-se repetir esse processo para as versões paralelizados em OpenMP e em MPI seguindo os mesmos critérios anteriores. O resultado para a versão paralelizada em OpenMP pode ser visto em [51] e [52]; e o resultado para a versão paralelizada em MPI pode ser visto em [53] e [54].

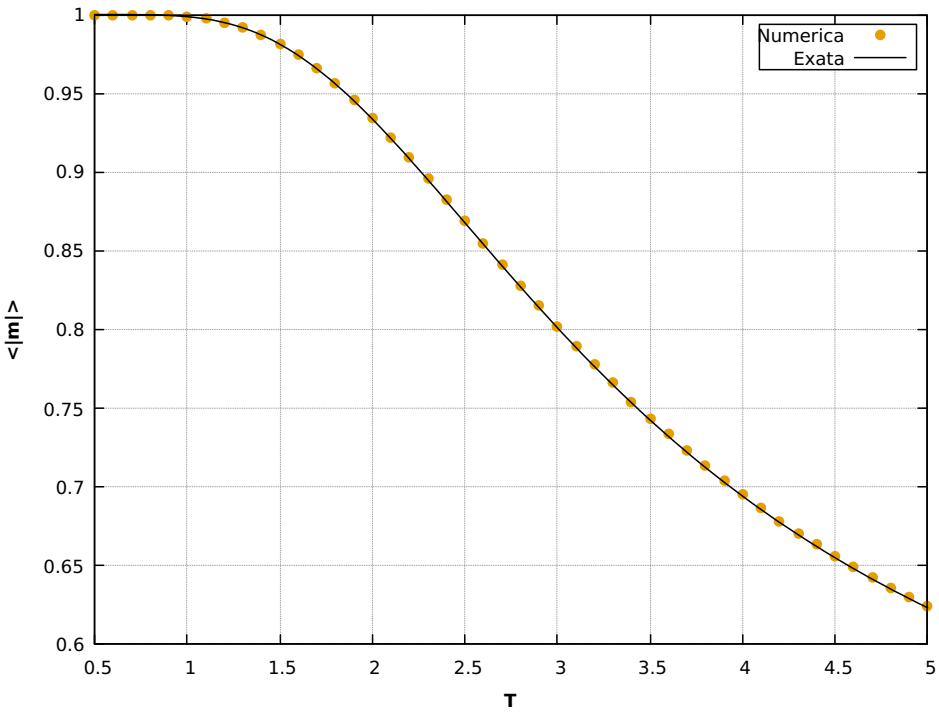

Figura 51 – Magnetização absoluta média por spin vs Temperatura - Solução numérica (OpenMP) e analítica

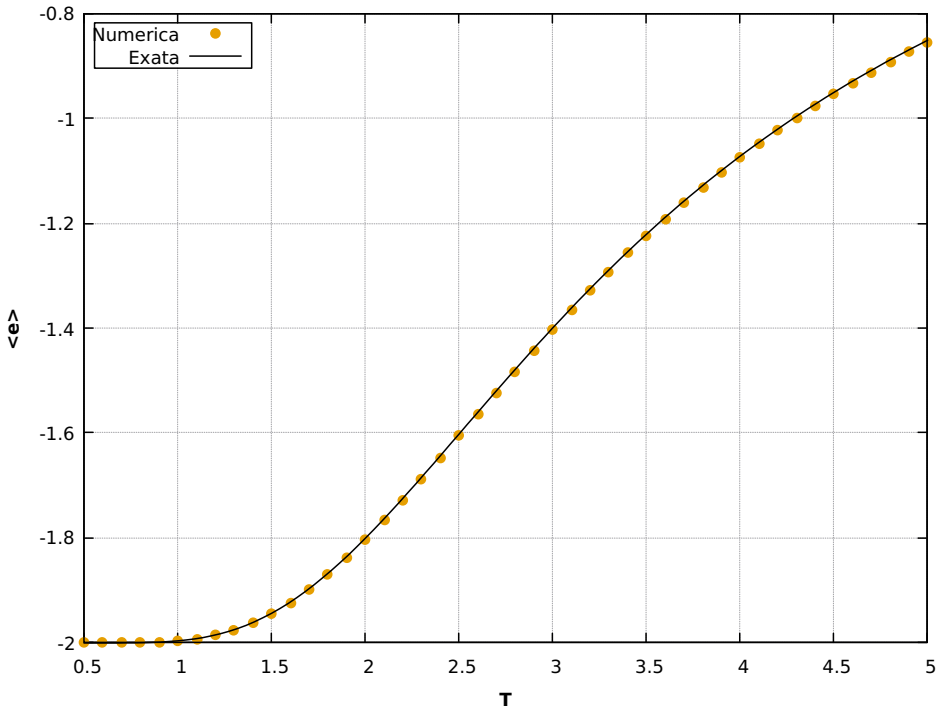

Figura 52 – Energia média por spin vs Temperatura - Solução numérica (OpenMP) e analítica

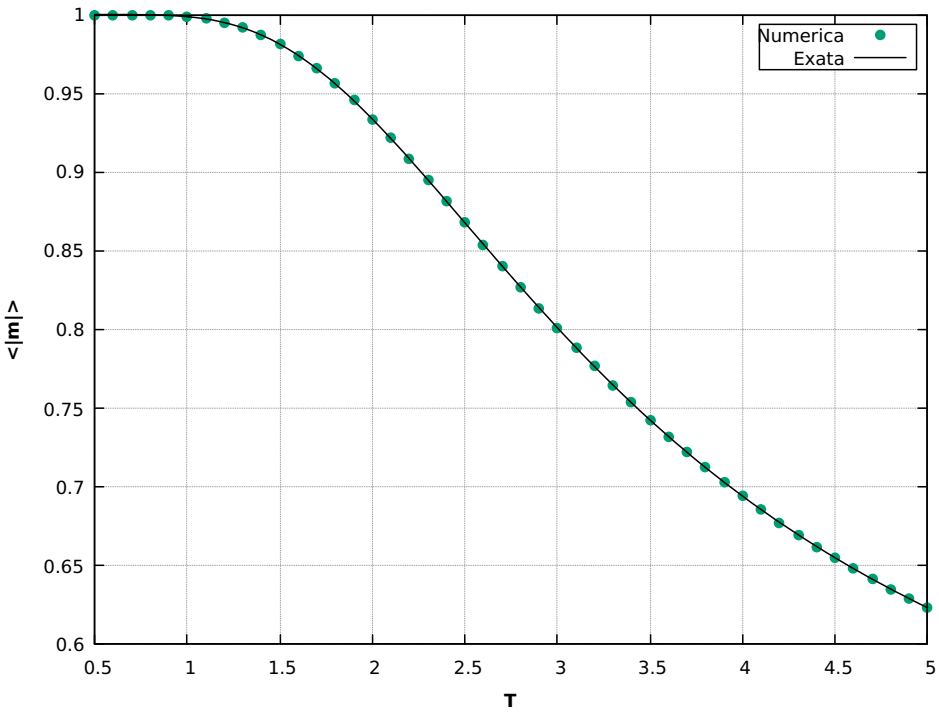

Figura 53 – Magnetização absoluta média por spin vs Temperatura - Solução numérica (MPI) e analítica

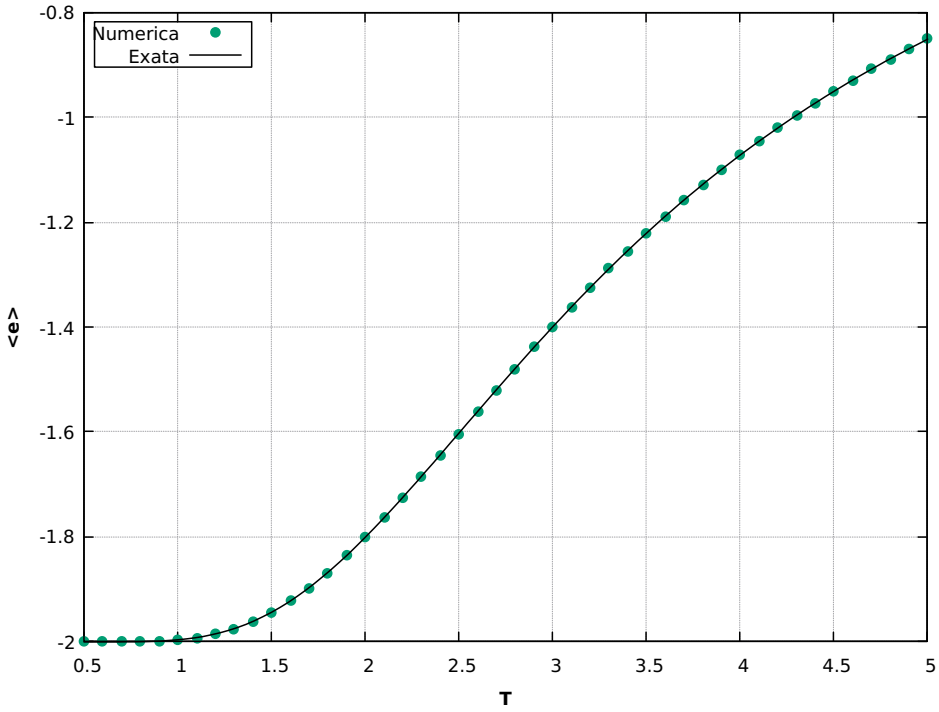

Figura 54 – Energia média por spin vs Temperatura - Solução numérica (MPI) e analítica

Ou seja, demonstrando que os algoritmos paralelizados, tanto em OpenMP quanto em MPI também seguem corretamente o resultado esperado.

Com a validação realizada, iremos agora a análise dos resultados para diversos tamanhos da rede de spins para demonstrar 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. A simulação foi feita utilizando 9600 passos no loop de Monte Carlo e 96 passos para o sistema chegar no equilíbrio. Além disso, a temperatura inicial utilizada foi $T_i = 0.5$, a temperatura final $T_f = 5$ e o incremento dT = 0.1, mantendo os demais atributos: B = 0, J = $k_B = 1$ e os valores dos spins ± 1.

11.0.1 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, que 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 demonstrados na figura [55]. Além disso, ele fica mais nítido conforme o tamanho da rede aumenta.

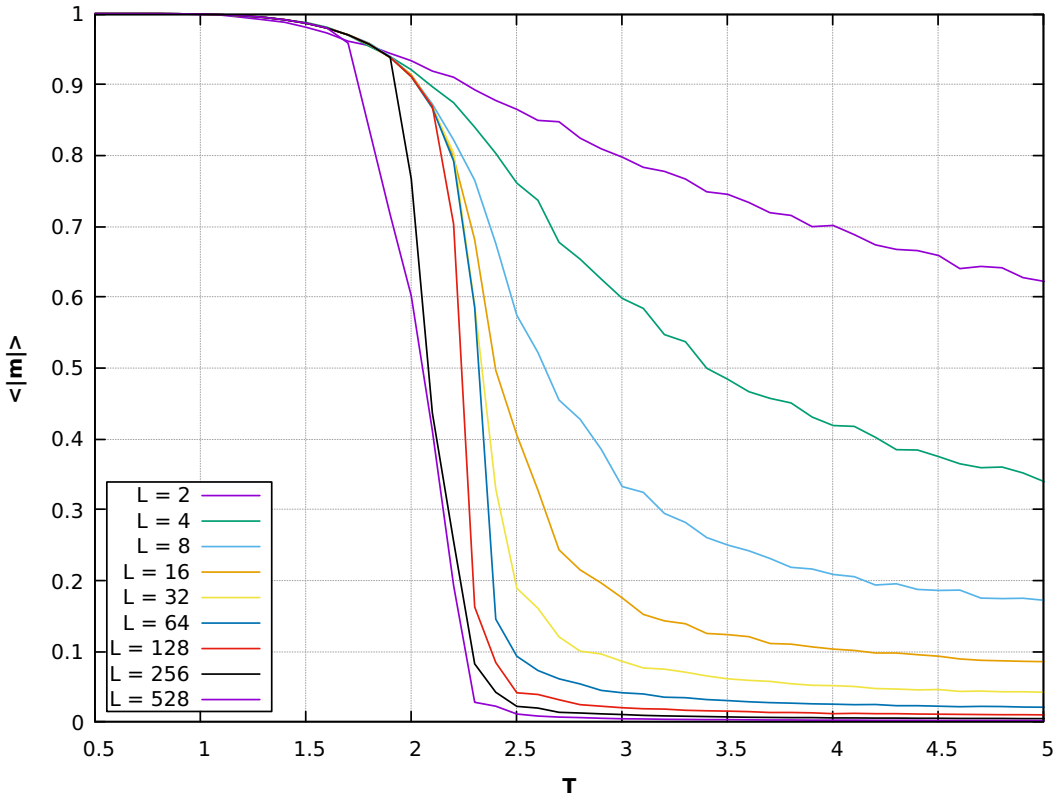

Figura 55 – Comparação da Magnetização absoluta por spin vs Temperatura para vários tamanhos da rede

11.0.2 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 [56], e, conforme o tamanho da rede aumenta esse comportamento fica mais nítido.

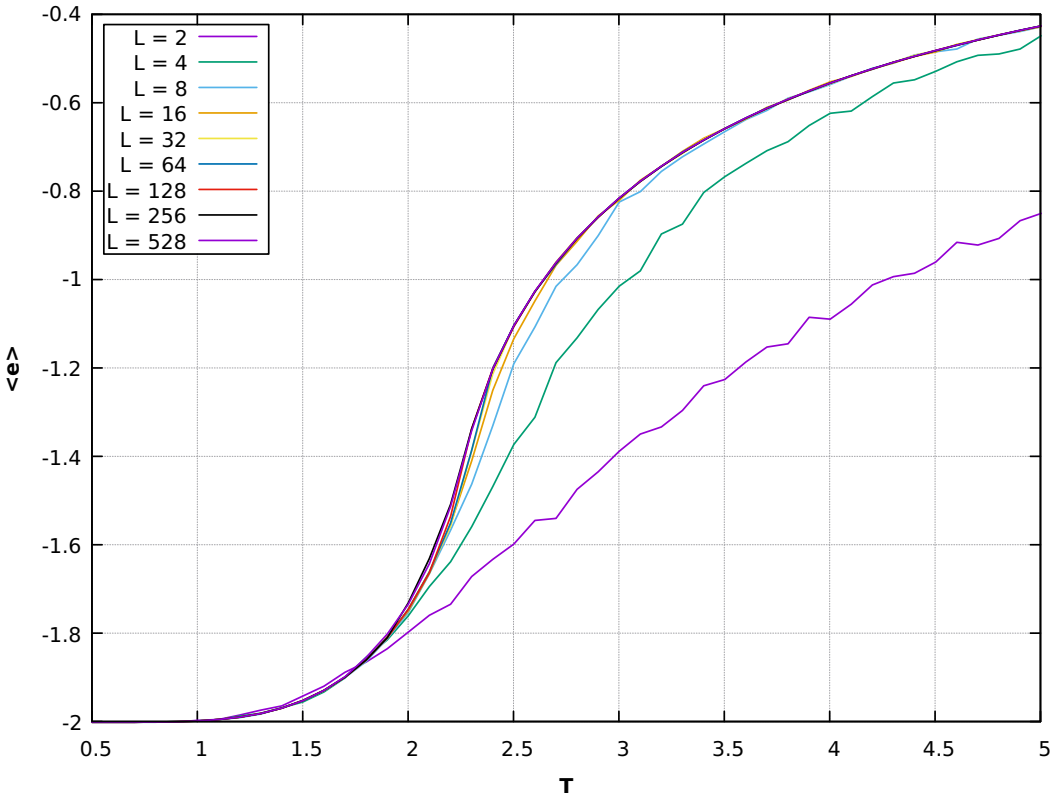

Figura 56 – Comparação da Energia por spin vs Temperatura para vários tamanhos da rede

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.

11.0.3 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 [57].

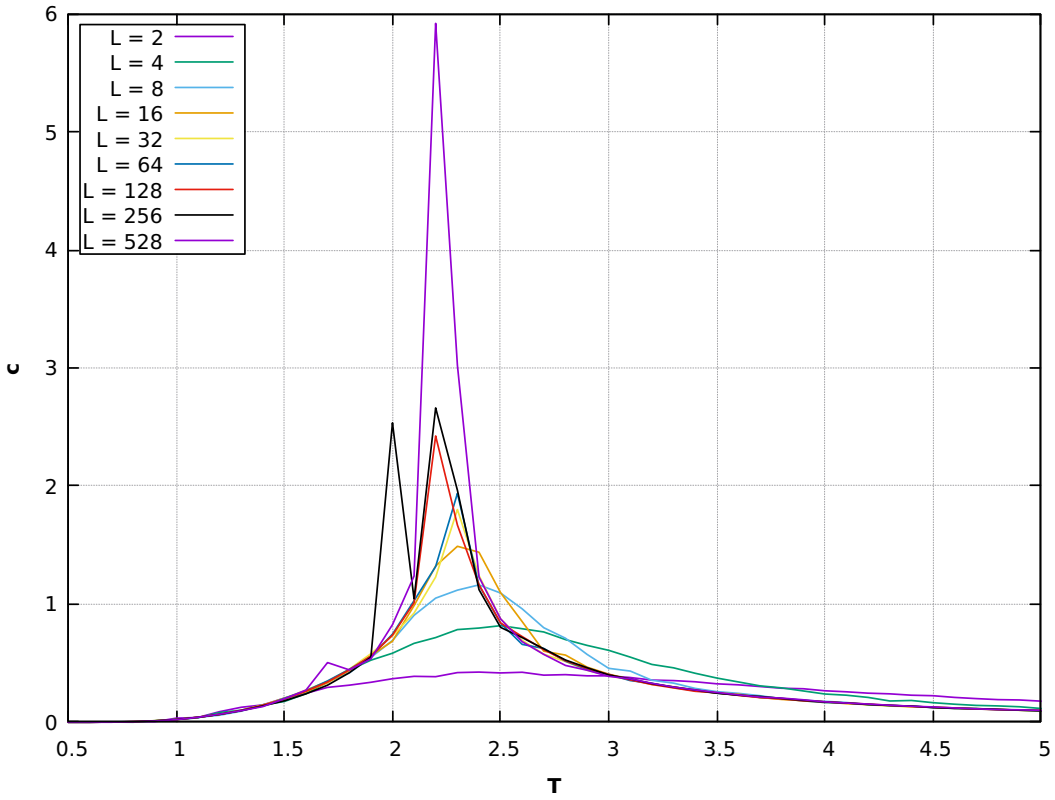

Figura 57 – Calor específico por spin vs Temperatura para vários tamanhos da rede

11.0.4 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 [58].

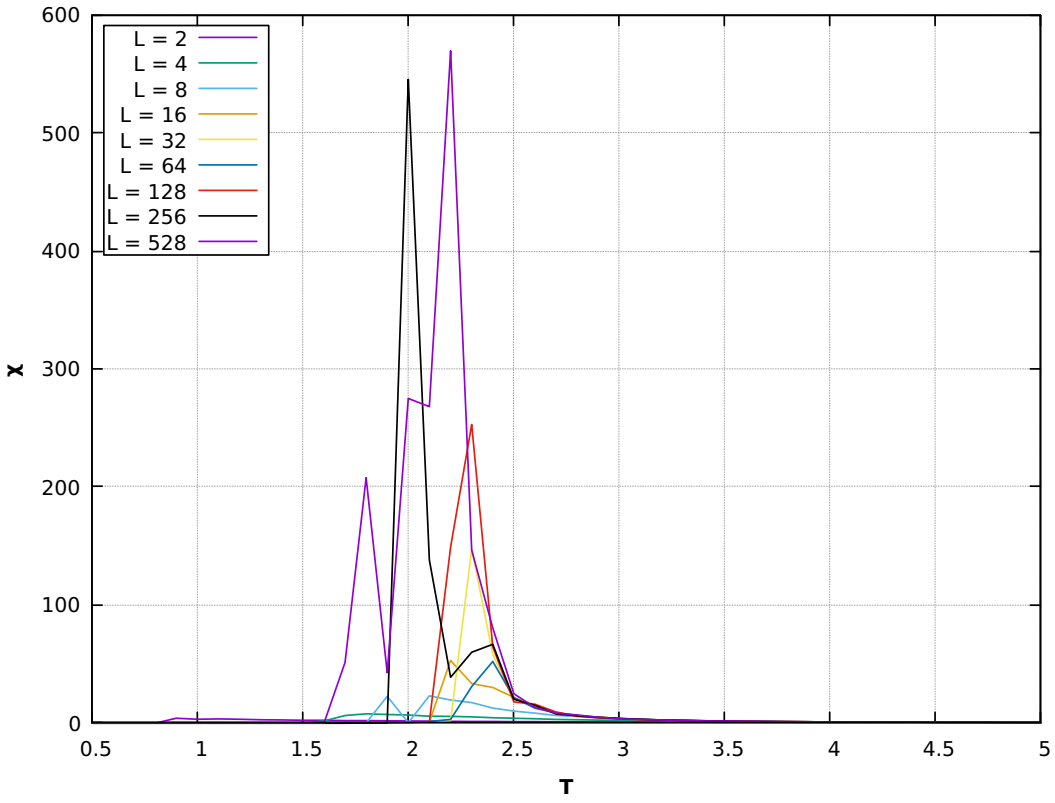

Figura 58 – Susceptibilidade magnética vs Temperatura para vários tamanhos da rede

12 Conclusões

Os resultados obtidos durante todo o trabalho mostram que todas as técnicas aplicadas, seja a nível de compilador ou técnicas de paralelização (em OpenMP ou MPI) geraram resultados positivos, isto é, geraram ganho de desempenho no tempo de execução do algoritmo e mantiveram a confiabilidade dos resultados.

Com as técnicas de otimização a nível de compilador já foi possível gerar bons resultados: No melhor caso houve um ganho de cerca de **52**% 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 essa técnica já seria possível ganhar bastante desempenho.

Em relação as técnicas de paralelização, ao se utilizar a API do OpenMP foi possível se obter, em uma máquina de 8 threads, um ganho de **73**% somente aplicando a paralelização e **78**% em união com as técnicas a nível de compilador. 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 a utilização da API poderia trazer. Ao se utilizar a API do MPI, foi possível se obter em uma máquina de 8 cores um ganho de **84**% somente aplicando a paralelização e **86**% em união com as técnicas a nível de compilador. Além disso, quando executamos em uma máquina de 24 cores, essas eficiências saltam para **96**% e **97**%, e para uma máquina de 48 cores, ficam em **97**% e **98**%. Entretanto, com os resultados vistos durante o trabalho é possível dizer que uma máquina de 16 cores provavelmente já seria o bastante para executar o algoritmo e ter ganhos expressivos. Contudo, é óbvio que se uma máquina mais potente estiver a disposição, o uso vai gerar resultados melhores e todo ganho importa.

Outro resultado interessante foi o obtido ao se comparar as API's utilizadas, nesse caso, o OpenMP e o MPI, onde foi possível notar que o MPI gerou melhores resultados. Entretanto, não é possível definir que esse é um resultado geral, muito pelo contrário, deve depender da forma de como a paralelização é executada. Mas ainda sim, no escopo deste trabalho a conclusão é que o MPI resultou em condições melhores.

De forma geral, diante de tudo, os resultados obtidos durante todo o processo 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 59.

GIORDANO, N. J. *Computacional Physics*. [S.l.: s.n.], 1997. Citado na página 6.

KOTZE, J. Introduction to monte carlo methods for an ising model of a ferromagnet. *Arxiv*, 2008. Citado na página 66.

SANTOS, M. L. *SIMULAÇÃO DE MONTE CARLO NO MODELO DE ISING NA REDE QUADRADA*. Dissertação (Mestrado) — UFMG, 2014. Citado na página 59.

SCHROEDER, D. V. *An Introduction to Thermal Physics*. [S.l.: s.n.], 1999. Citado na página 6.