UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE CIÊNCIAS EXATAS

PATRYCK ADRIEN DOS SANTOS

Equação de Poisson

Volta Redonda
2022

Resumo

Neste trabalho busca-se o estudo e aplicação de práticas de paralelismo usando a API (Interface de Programação de Aplicativo) OpenMP para otimização de performance de programas de resolução de EDPs da Equação de Poisson.

Palavras-chaves: Equação de Poisson. Paralelismo. OpenMP.

Abstract

This project searches to study and apply techniques of parallelism using the OpenMP API (Application Programming Interface) to optimize programs of EDPs resolution for Poisson Equation.

Keywords: Poisson Equation. Parallelism. OpenMP.

Sumário

1 Introdução

Métodos númericos computacionais para resolução de EDPs (Equações Diferenciais Parciais) geralmente trazem altos custos computacionais para sistemas grandes, como também, são métodos em geral paralelizavéis.

Neste trabalho utilizou-se a interface de programação OpenMP [OPENMP 2022] para aplicar técnicas de paralelismo visando a otimização do método númerico de Gauss-Seidel aplicado na resolução da EDP da Equação de Poisson.

2 Objetivos

Um objetivo natural inicial é o ganho de performace com relação ao tempo de execução do código, para isso, pretende-se além do estudo de técnicas de paralelismo e otimização, realizar análises sobre diferentes divisões de malha e suas implicações nos resultados finais.

Ao possuir conhecimento prévio das máquinas a serem utilizadas e suas especificações, busca-se utilizar malhas divisíveis pelo número de threads presente nas máquinas e então, comparar os resultados com valores de referência sem essa divisão.

Existindo a possibilidade de variar as máquinas utilizadas na obtenção dos dados para análise, onde tem-se as máquinas do Laboratório de Informática do ICEx (Instituto de Ciências Exatas) e as máquinas B700 e BullSequana X do LNCC (Laboratório Nacional de Computação Científica), prevê-se a comparação dos resultados obtidos entre as máquinas.

3 Metodologia

3.1 Máquinas

Uma das máquinas utilizadas pertence ao ICEx e possui como processador o Intel i7-2600 e CPU de 3.40GHz. Possui memórias de cache ilustradas na Figura 1 e memória RAM ilustradas na Figura 2.

```
Caches (sum of all):
  L1d:                    128 KiB (4 instances)
  L1i:                    128 KiB (4 instances)
  L2:                     1 MiB (4 instances)
  L3:                     8 MiB (1 instance)
```

Figura 1 – Caches

	total	used	free	shared	buff/cache	available
Mem:	11Gi	1,2Gi	6,6Gi	42Mi	3,9Gi	10Gi
Swap:	22Gi	0B	22Gi			

Figura 2 – Memória

Outra máquina utilizada foi a SDumont Base (B700) do LNCC possuindo CPU Intel Xeon E5-2695v2 Ivy Bridge de 2,4GHZ, 24 núcleos (12 por CPU) e mémoria RAM de 64GB modelo DDR3. Informações mais detalhadas podem ser encontradas no manual disponível no site SDumont[1].

Por fim, a última máquina utilizada também do LNCC, SDumont Expansão (BullSequana X), possui CPU Intel Xeon Cascade Lake Gold 6252 composta da tecnologia BullSequana X, 48 núcleos (24c por CPU) e 384Gb de memória RAM. Informações mais detalhadas podem ser encontradas no manual disponível no site SDumont.[1]

3.2 Método

A proposta para este trabalho é a resolução da EDP de Poisson

$$u_{xx} + u_{yy} = \frac{x}{y} + \frac{y}{x}, \ \ com \ \ 1 \leq x \leq 2 \ \ e \ \ 1 \leq y \leq 2 \tag{3.1}$$

[1] SDumont: <https://sdumont.lncc.br/support_manual.php?pg=support#6.3>. Acesso em: 3 de dezembro de 2022

Com as condições de contorno

$$u(x,1) = xln(x), \qquad u(x,2) = 2xln(2x)$$

$$u(1,y) = yln(y), \qquad u(2,y) = 2yln(2y)$$

Para isso discretiza-se o domínio em uma malha, onde

$$x = a + ih_x \ \ com \ \ i = 0,1,2,\ldots,m$$

$$y = c + jh_y \ \ com \ \ j = 0,1,2,\ldots,n$$

Tendo $h_x = \dfrac{b-a}{m}$ e $h_y = \dfrac{d-c}{n}$, onde m e n são as dimensões da malha, $a = c = 1$ e $b = d = 2$.

A partir da discretização, $u_{ij} = u(x_i, y_j)$ será o valor de $u(x,y)$ em (x_i, y_j). A aproximação de diferenças finitas para o laplaciano [GILAT e SUBRAMANIAM 2008] nos dá que,

$$u_{xx} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2}, \ \ i = 1,2,\ldots,m-1$$

$$u_{yy} = \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_x^2}, \ \ j = 1,2,\ldots,n-1$$

Portanto a malha ficará como ilustra a Figura 3.

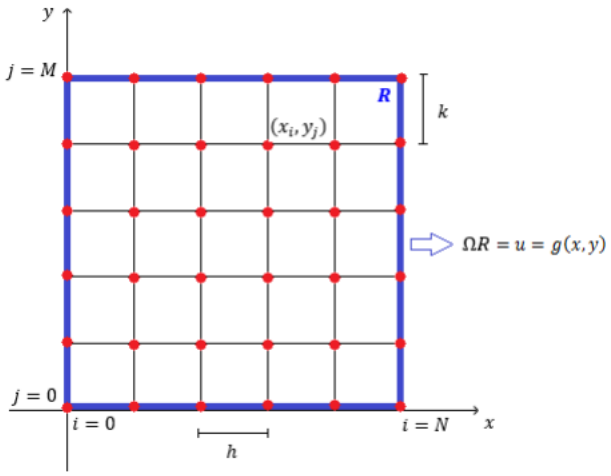

Figura 3 – Discretização da malha

Após a incialização da malha e aplicação das condições de contorno, será usado o método de Gauss-Seidel, descrito pela equação 3.2 para a iteração nos valores da malha, limitando essas iterações a um critério de parada, que será uma quantidade máxima de iterações.

$$u_{i,j}^{k+1} = \frac{u_{i-1,j}^{k+1} + u_{i+1,j}^{k} + u_{i,j-1}^{k+1} + u_{i,j}^{k} - h^2 f_{ij}}{4} \tag{3.2}$$

Ao final, os resultados podem ser comparados com os valores exatos dados pela função

$$u(x, y) = xyln(xy) \tag{3.3}$$

E validados pelo método de erro relativo [FRANCO], onde

$$\epsilon = \frac{\|u_{exato} - u_{solução}\|_\infty}{\|u_{exato}\|_\infty} \tag{3.4}$$

e a norma infinita de uma matriz u é dada por

$$\|u\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |u_{ij}|$$

Para uma melhor visualização do método foi desenvolvido um fluxograma, como ilustra a Figura 4.

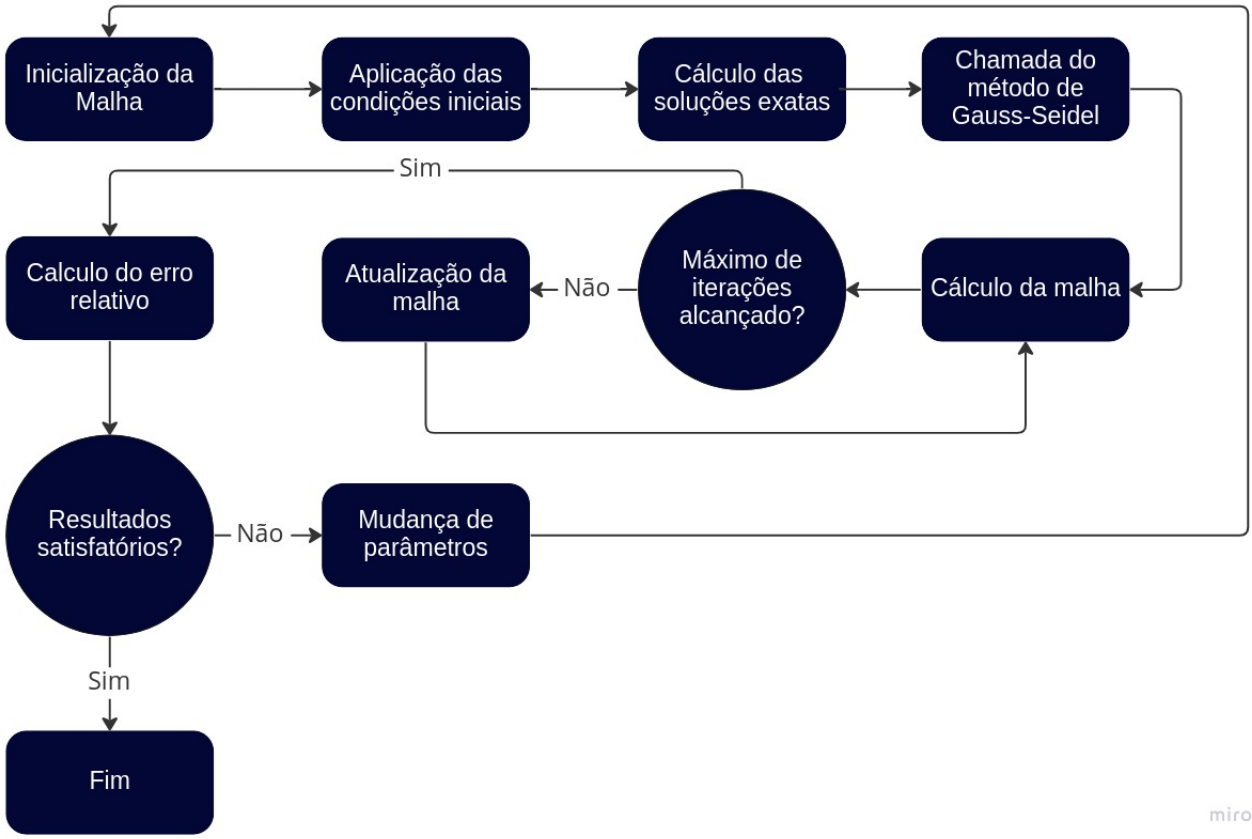

Figura 4 – Fluxograma

Onde a incialização é dada pela discretização da malha e aplicação das condições de contorno, calcula-se as solução exata dada pela equação 3.3, aplica-se o método de Gauss-Seidel dado pelo equação 3.2 na malha discretizada até que a quantidade máxima

de iterações seja alcançada, calcula-se o erro relativo dado pela equação 3.4 e analisa-se o resultado, caso não seja satisfatório, será feita uma refatoração no método de paralelização.

Inicialmente foi desenvolvido um código de referência para o cálculo da EDP que não visava o desempenho e sim a obtenção dos resultados. Assim o código foi executado para uma malha de 500×500 e marcado o tempo de execução, como mostra a Figura 5.

```
gfortran -O2  -o poisson inicializa.o cond_ini.o cond_con.o sol_ex.o met_gs.o poisson.o

real    23m55,731s
user    14m25,645s
sys     0m0,012s
```

Figura 5 – Primeiro Tempo

Após a aplicação de técnicas de otimização refatorando o código, obteve-se um ganho na eficiência de 8,85%. Sendo a eficiência η dada por,

$$\eta = 1 - \frac{T_{rod}}{T_{ref}} \tag{3.5}$$

Onde T_{rod} o tempo de rodada e T_{ref} o tempo de referência. Obtendo então, um menor tempo de execução, como mostra a Figura 6.

```
gfortran -O2 -pg -o poisson inicializa.o cond_ini.o cond_con.o sol_ex.o met_gs.o poisson

real    21m48,642s
user    14m1,564s
sys     0m0,008s
```

Figura 6 – Segundo Tempo

Foi feita então um profile do código, ou seja, uma ánalise dos gargalos presentes, utilizando a flag *-pg*, como mostra a Figura 7.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 83.56     682.63   682.63        1   682.63   816.92  __met_gs_MOD_sub_metodo_gauss_seidel
 11.87     779.62    96.99   391724     0.00     0.00  __met_gs_MOD_norma_infinita_mat
  4.57     816.92    37.30   195862     0.00     0.00  __met_gs_MOD_dif_rel
  0.00     816.93     0.01                             _init
  0.00     816.93     0.00        1     0.00   816.92  MAIN__
  0.00     816.93     0.00        1     0.00     0.00  __cond_con_MOD_condicoes_contorno
  0.00     816.93     0.00        1     0.00     0.00  __cond_ini_MOD_condicoes_iniciais
  0.00     816.93     0.00        1     0.00     0.00  __inicializa_MOD_inicializacao
  0.00     816.93     0.00        1     0.00     0.00  __sol_ex_MOD_solucao_exata
```

Figura 7 – Profile

Onde observa-se que 83,56% do tempo de execução foi gasto na subrotina do método de Gauss-Seidel, portanto, está é a melhor aposta para iniciar uma paraleliização no código.

Após estudos e testes de práticas de paralelização [HERMANNS], foi implementado a diretiva *do*, geralmente utilizada em loops, na subrotina do método de Gauss-Seidel. Também foram utilizadas as diretivas *default*, que pode definir o tipo (*shared* ou *private*) da maioria das variáveis, sem necessidade de classificação de uma por uma e *private* que contrária à última diretiva, dá a variáveis específicadas o tipo *private*, por fim foi utilizado a diretiva *schedule*, que escolhe como o loop será dividido entre as threads, o atributo *dynamic* divide o loop em tamanhos *chunk* e distribui dinamicamente os trabalhos entre as threads.

Para melhor visualização, foi feito um antes e depois da parte paralelizada do código, como mostram as Figuras 8 e 9.

```
    do while((delta >= tol))
        do i=1, m-1
            do j=1, n-1
                u_novo(i, j) = (u_novo(i-1, j) + u_ant(i+1, j) + &
                                u_novo(i, j-1) + u_ant(i, j+1) - &
                                ((h**2)*f(i, j, m, n, x, y)))/(4.d0)
            end do
        end do
        delta = dif_rel(u_novo, u_ant, m, n)
        k = k+1
        u_ant = u_novo
    end do
    print*, 'Quantidade de iterações:', k

 end subroutine sub_metodo_gauss_seidel
```

Figura 8 – Antes

```
          do while((it < max_it))

!$omp parallel do &
!$omp& default(shared) private(i, j) &
!$omp& schedule(dynamic, chunk)

               do i=1, (m-1)
                   do j=1, (n-1)
                       u_novo(i, j) = (u_novo(i-1, j) + u_ant(i+1, j) + &
                                       u_novo(i, j-1) + u_ant(i, j+1) - &
                                       ((h*h)*f(i, j, m, n, x, y)))*(0.25d0)
                  end do
               end do

!$omp end parallel do

               it = it + 1
               u_ant = u_novo
           end do

       end subroutine sub_metodo_gauss_seidel
```

Figura 9 – Depois

Além da paralelização, foi necessário uma mudança de críterio de parada do código não paralelizado para o código paralelizado, para fins de melhor convergência.

Após um estudo sobre os melhores valores para o chunk, obteve-se os melhores tempos fazendo a relação, chunk $= n \times 0.05$, onde n é a dimensão da malha. Como o valor de chunk depende do tamanho da malha, este varia em cada análise.

Foi utilizado o conjunto de flags *-O2 -fopenmp -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native* para o compilador gfortran da GNU e o conjunto de flags *-O2 -qopenmp -mp1 -zero -xHOST -ipo -align* [PILLA 2018] para o compilador ifort da Intel.

Foi executada uma simulação e gravado o tempo de execução para cada rodada, variando as threads. Ao final de cada tomada de tempo foi feito um gráfico de tempo versus threads, eficiência versus threads e escalabilidade versus threads. Sendo a escalabilidade a dada por,

$$a = \frac{T_{rod}}{T_{ref}} \tag{3.6}$$

4 Análises

Iniciou-se as ánalises de desempenho com uma malha de 160×160, possuindo chunk $= 8$. O erro relativo, dado pela equação 3.4, foi de $9,93 \times 10^{-8}$ e não obteve variância a partir do valor máximo de iterações de 1×10^{8}. Variando o número de threads e os compliladores da GNU e Intel, obteve-se o seguinte gráfico da Figura 10.

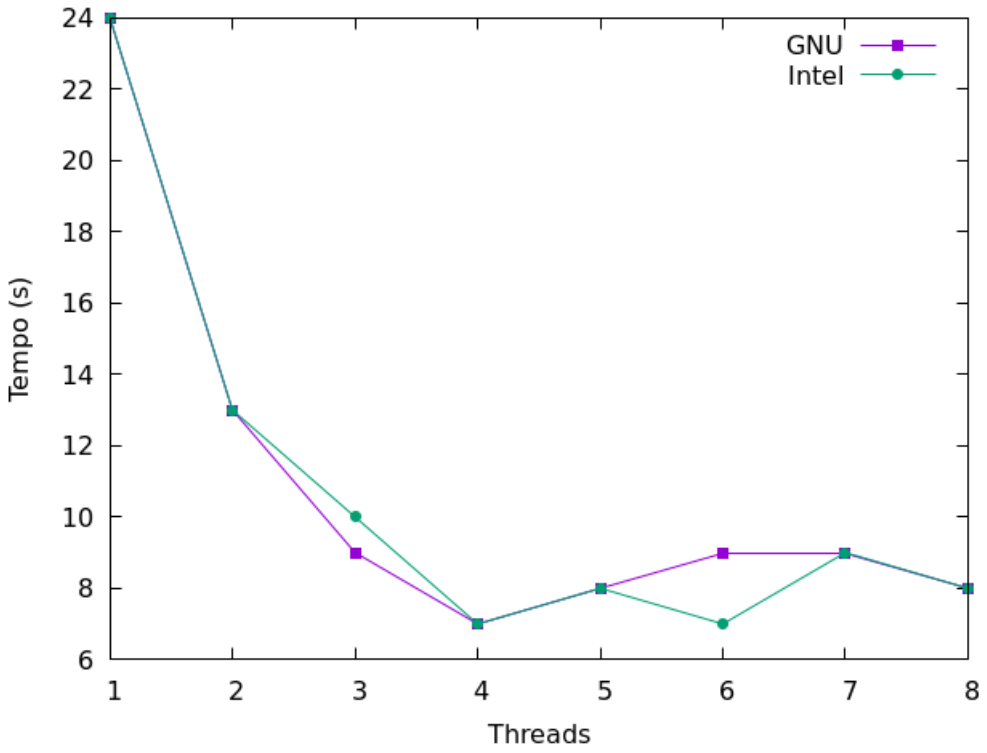

Figura 10 – Malha 160×160

Onde ambos compliladores obtiveram o menor tempo com 4 threads e após isso, apenas o compilador da Intel conseguiu outro tempo mínimo com 6 threads.

Foi feito então um gráfico de eficiência, dada pela equação 3.5, variando as threads, como mostra a Figura 11. Onde obteve-se uma eficiência um pouco maior que 70% em 4 threads, para os dois compiladores, como também, uma eficiência um pouco maior que 70% em 6 threads para o compilador da Intel.

Então foi feita a análise de escalabilidade do código, dada pela equação 3.6, variando as threads, como mostra a Figura 12. Onde o código perde escalabilidade até 4 threads e volta a ganhar a apartir dela em ambos compiladores, com o compilador da GNU possuindo uma escalabilidade maior que o da Intel.

Para a malha de 240×240 e chunk $= 12$, a menor erro relativo foi de $4,42 \times 10^{-8}$

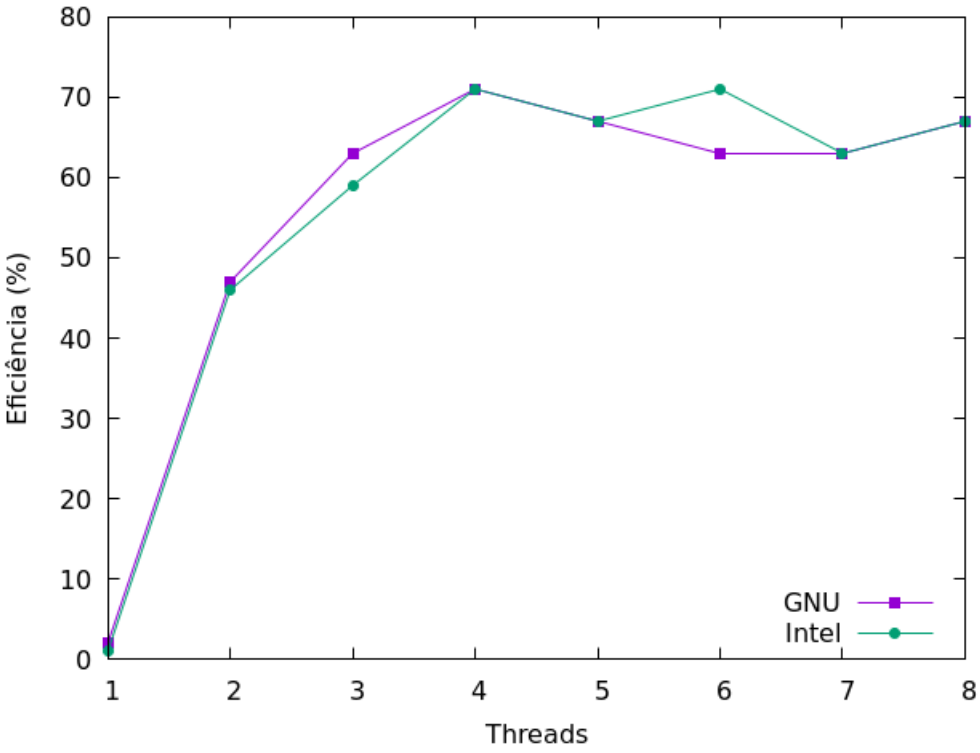

Figura 11 – Malha 160 × 160

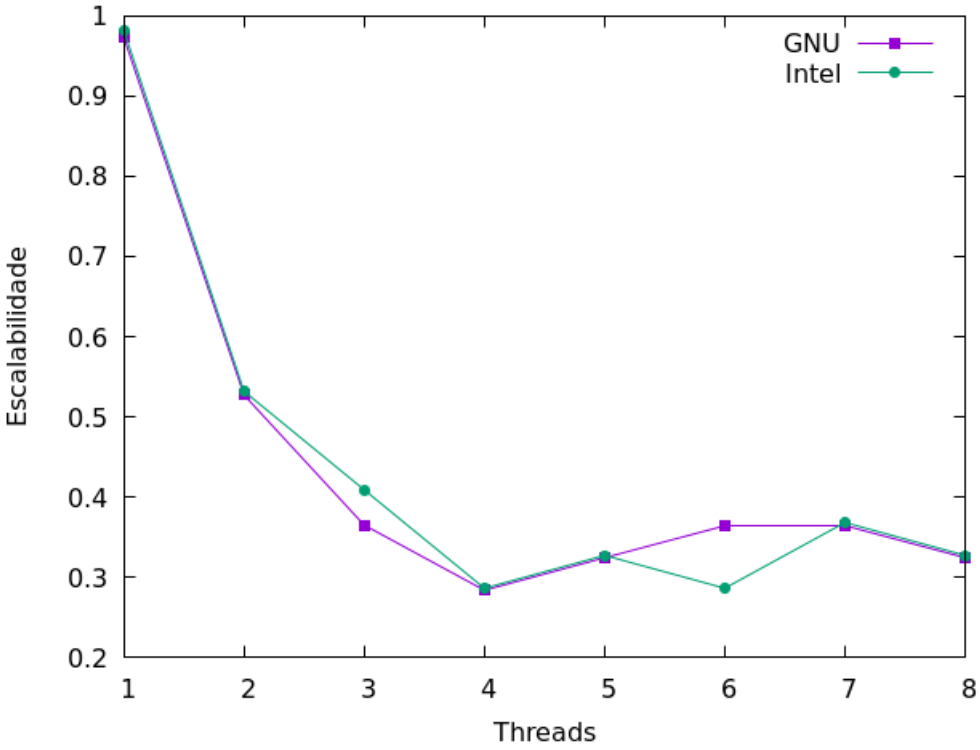

Figura 12 – Malha 160 × 160

com quantidade máxima de iterações de 2×10^5. Analogamente, foi feito o gráfico da Figura 13 variando o número de threads e os compiladores.

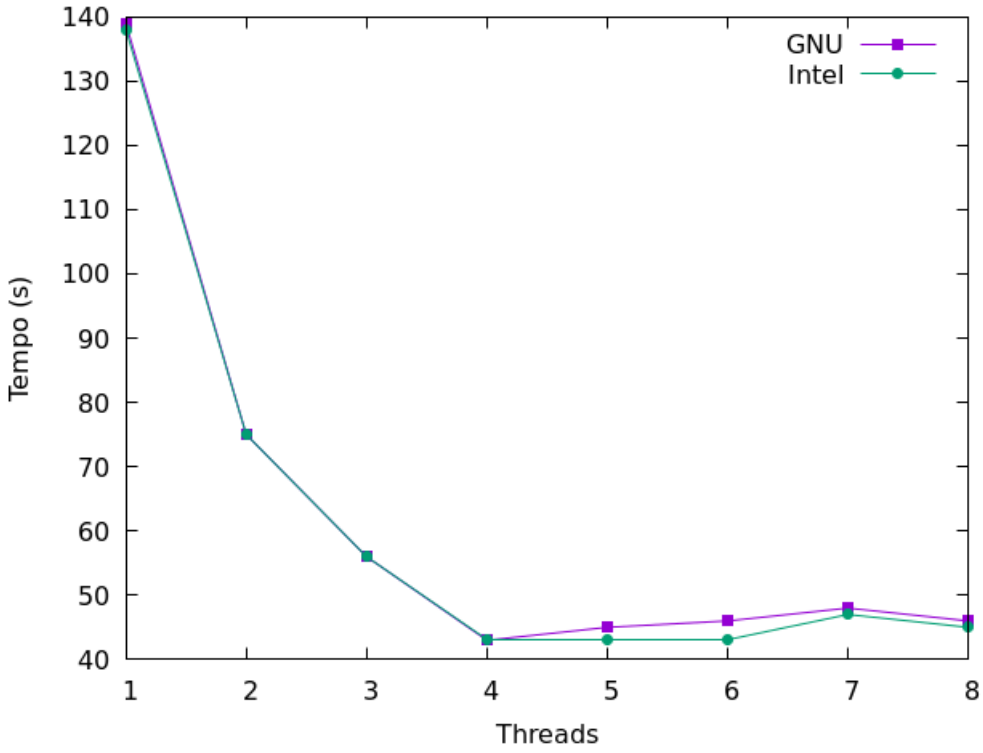

Figura 13 – Malha 240×240

Neste caso os dois compiladores permanecem praticamente invariantes até 4 threads e a partir dela o compilador da Intel obtém melhores tempos variando pouco com relação a seu tempo mínimo.

Na análise de eficiência, como mostra o gráfico da Figura 14, obteve-se um pouco menos de 70% em 4 threads desta vez, para ambos compiladores.

Na análise de escalabilidade do código, como mostra a Figura 15, a escalabilidade decaí muito com o aumento das threads, chegando quase a 0 a partir de 4 threads.

Para a malha de 320×320 e chunk $= 16$, o menor erro relativo foi de $2,92 \times 10^{-8}$ com quantidade máxima de iterações de 3×10^5. Analogamente, foi feito o gráfico da Figura 16 variando o número de threads e os compiladores.

Como na malha de 240×240 os dois compiladores permanecem invariantes até 4 threads e a partir dela o compilador da Intel obtém melhores tempos variando muito pouco com relação a seu tempo mínimo, permanecendo basicamente constante.

Na análise de eficiência, como mostra o gráfico da Figura 17, analogamente à malha de 240×240 obteve-se novamente um pouco menos de 70% em 4 threads, com o compilador do GNU tomando a frente na melhor eficiência, mas perdendo-a nas próximas threads. Na análise de escalabilidade do código, como mostra a Figura 18, a escalabilidade tem um aumento pequeno porém progressivo a partir de 4 threads para o compilador do GNU, já para o compilador da Intel a escalabilidade permanece constante.

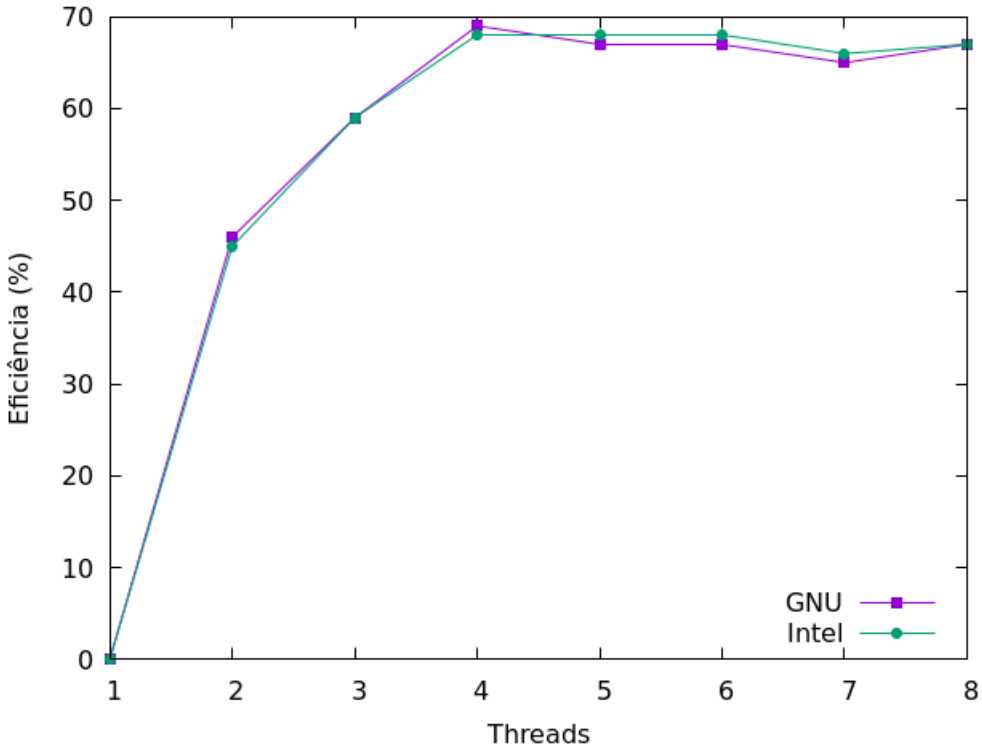

Figura 14 – Malha 240×240

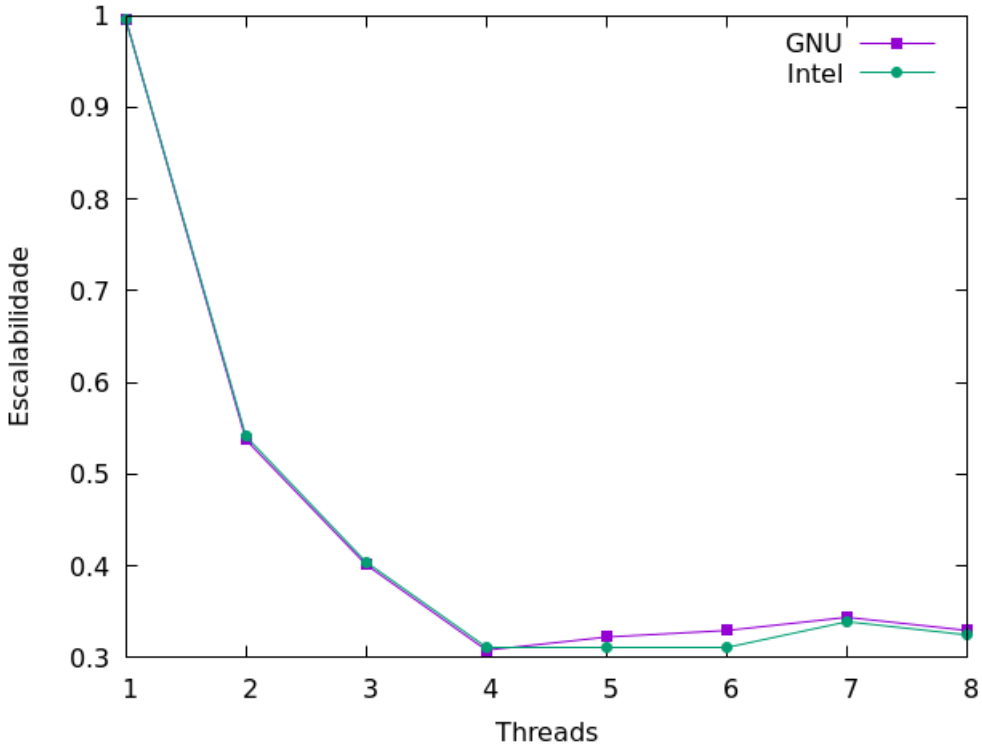

Figura 15 – Malha 240×240

Para a malha de 400×400 e chunk $= 20$, o menor erro relativo foi de $9,89 \times 10^{-9}$ com quantidade máxima de iterações de 3×10^5. Analogamente, foi feito o gráfico da

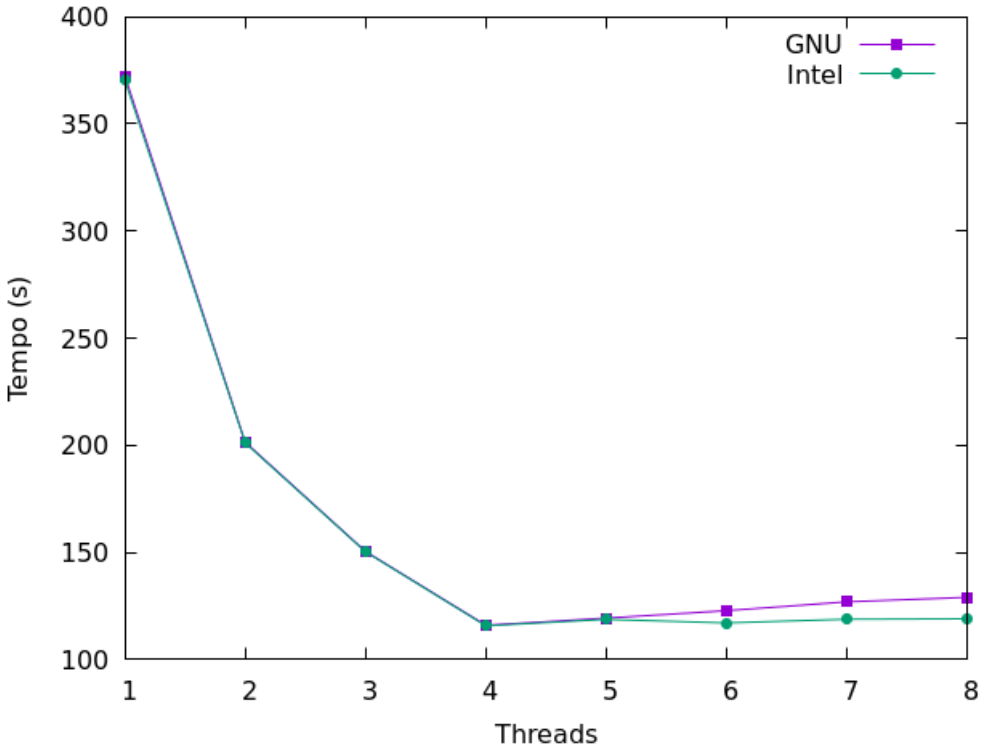

Figura 16 – Malha 320 × 320

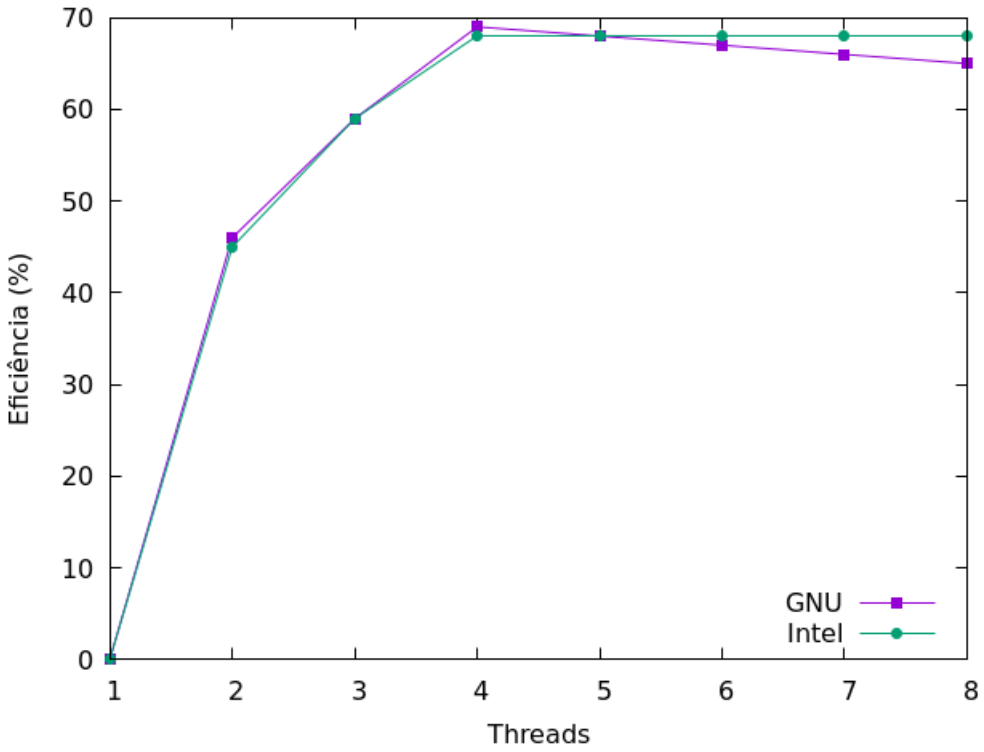

Figura 17 – Malha 320 × 320

Figura 19 variando o número de threads e os compiladores.

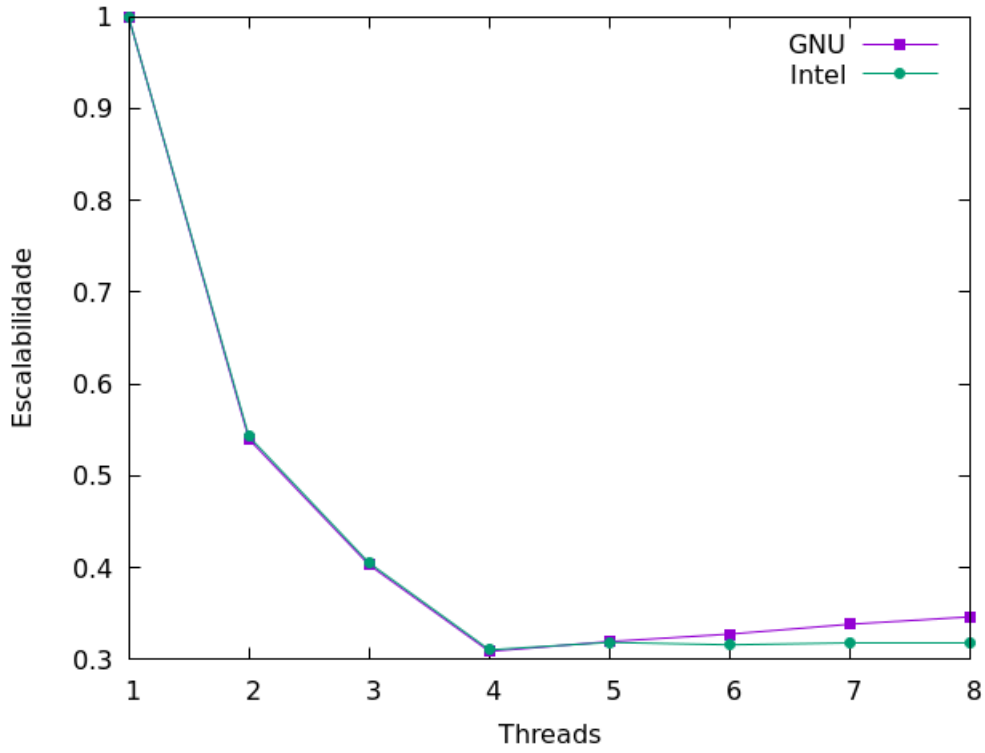

Figura 18 – Malha 320 × 320

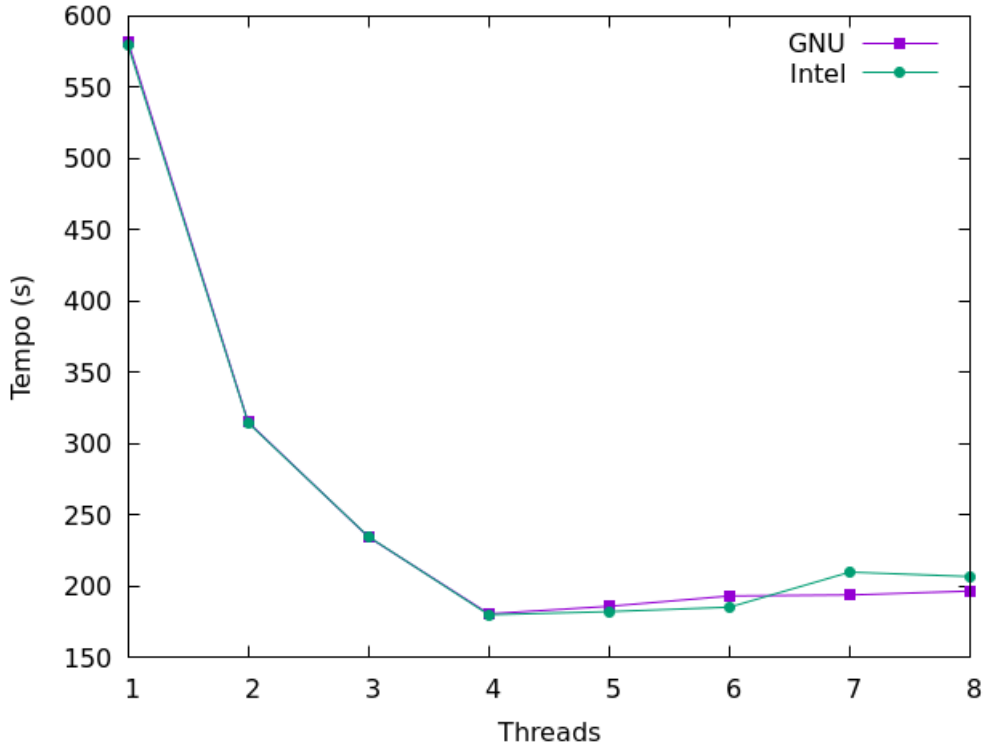

Figura 19 – Malha 400 × 400

Como visto nas malhas anteriores os dois compiladores permanecem invariantes até 4 threads e a partir dela o compilador da GNU varia menos desta vez, obtendo um

tempo menor que o compilador da Intel nas últimas 2 threads.

Na análise de eficiência, como mostra o gráfico da Figura 20, o compilador da Intel consegue ultrapassar a eficiência do compilador da GNU a partir de 4 threads, porém tem uma queda de eficincia a partir de 6 threads.

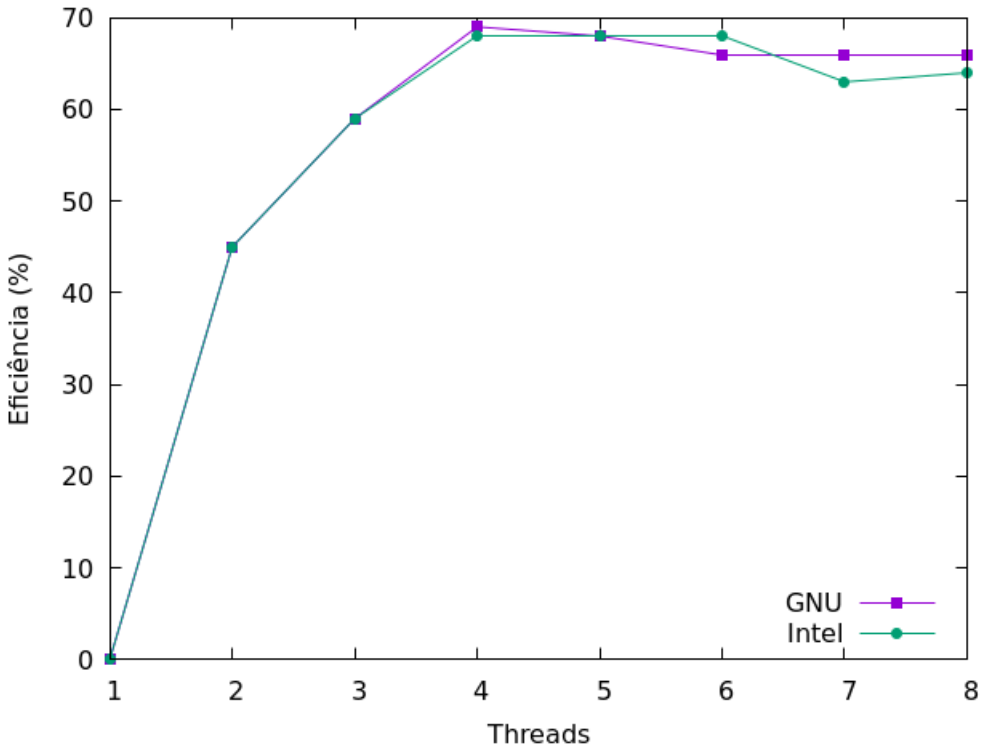

Figura 20 – Malha 400×400

Na análise de escalabilidade do código, como mostra a Figura 21, a escalabilidade tem um pequeno aumento apenas de 6 threads para 7 threads no compilador da Intel, já no compilador da GNU a escalabilidade permanece basicamente constante.

Para a malha de 480×480 e chunk $= 24$, o menor erro relativo foi de $9,97 \times 10^{-9}$ com quantidade máxima de iterações de 3×10^5. Analogamente, foi feito o gráfico da Figura 22 variando o número de threads e os compiladores.

Neste caso o compilador da GNU permaneceu com menores tempos em todas as threads, obtendo uma variância menor nas últimas 4 threads.

Na análise de eficiência, como mostra o gráfico da Figura 23, as variações entre os compiladores são muito pequenas, onde permanecem na faixa de 60% à 70% nas últimas 5 threads. Na análise de escalabilidade do código, como mostra a Figura 24, há um aumento de escalabilidade a partir de 4 threads em ambos compiladores, porém após ela a escalabilidade permanece basicamente constante.

Para a malha de 560×560 e chunk $= 28$, o menor erro relativo foi de $9,99 \times 10^{-8}$ com quantidade máxima de iterações de 5×10^5. Analogamente, foi feito o gráfico da

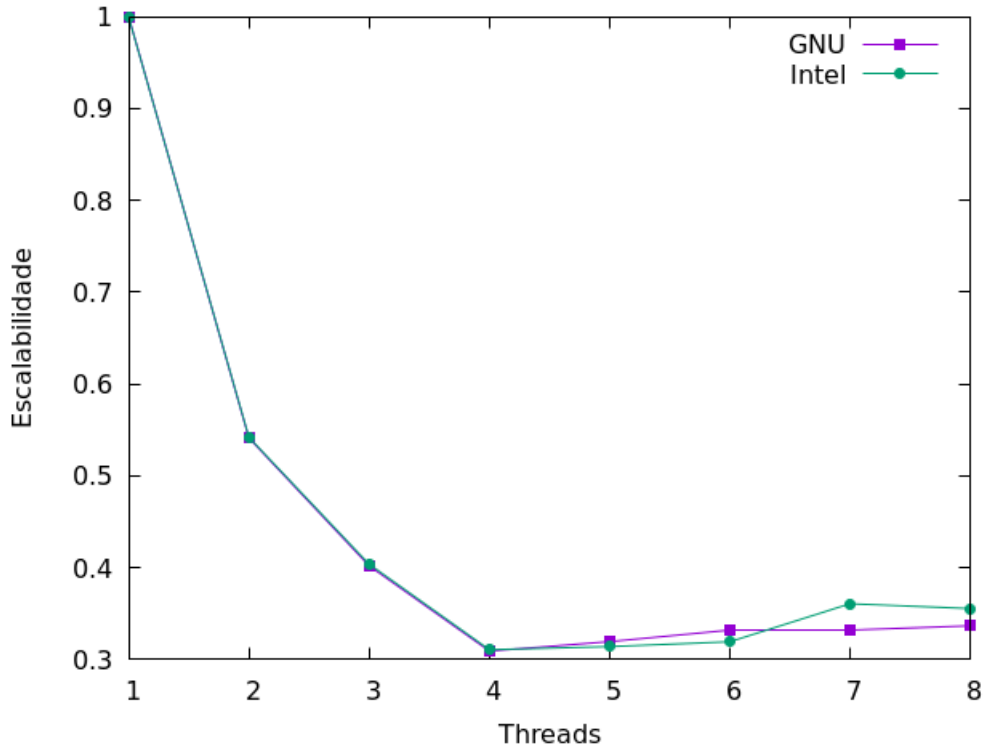

Figura 21 – Malha 400 × 400

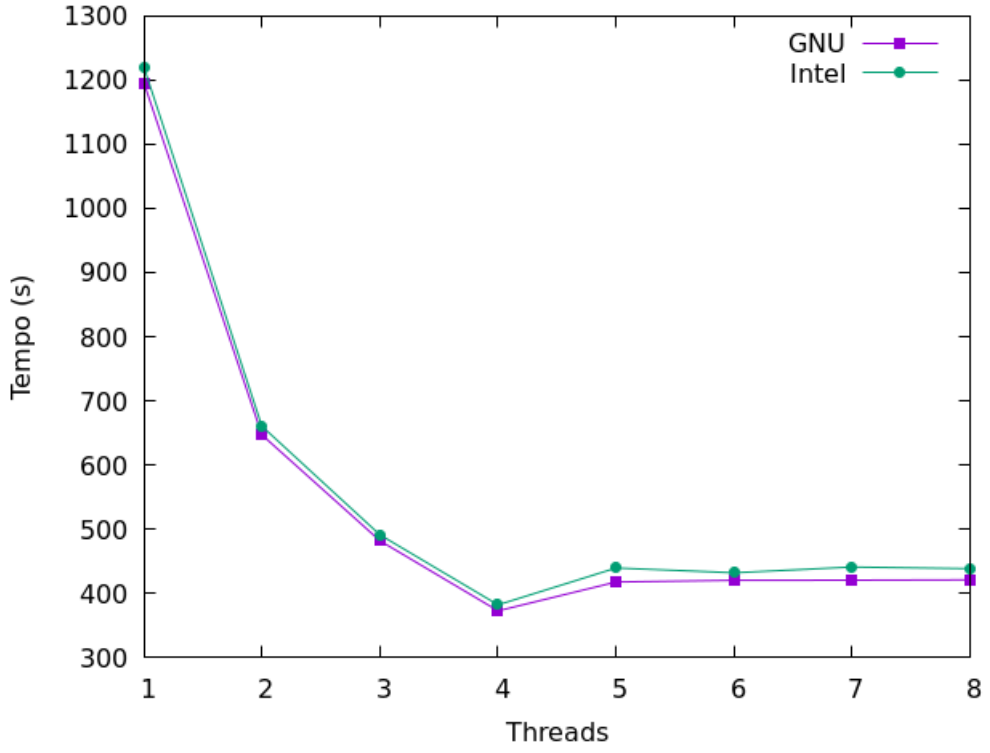

Figura 22 – Malha 480 × 480

Figura 25 variando o número de threads e os compiladores.

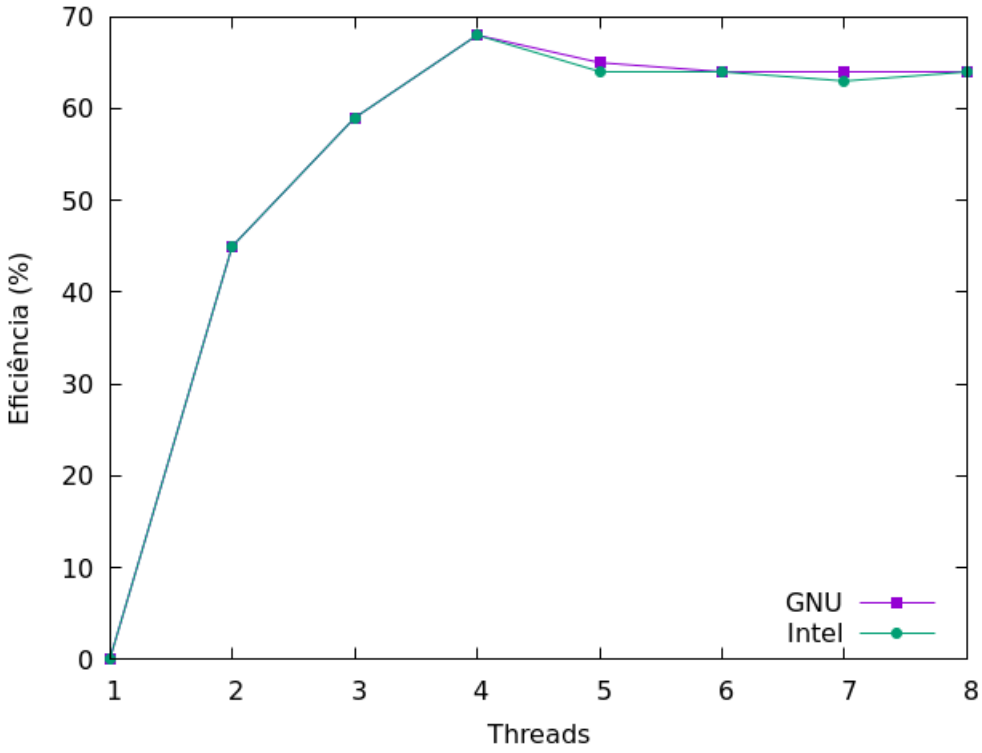

Figura 23 – Malha 480 × 480

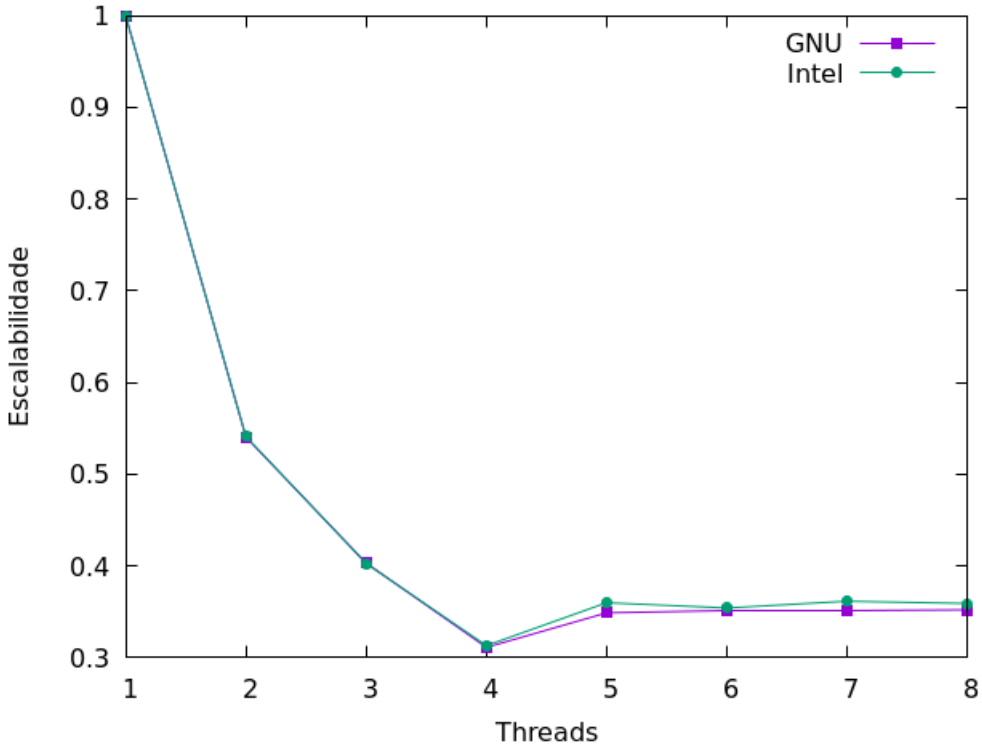

Figura 24 – Malha 480 × 480

Apesar da baixa variação entre os compiladores até 4 threads, a partir dela, o compilador da Intel mantém tempos menores que o compilador da GNU além de permanecer

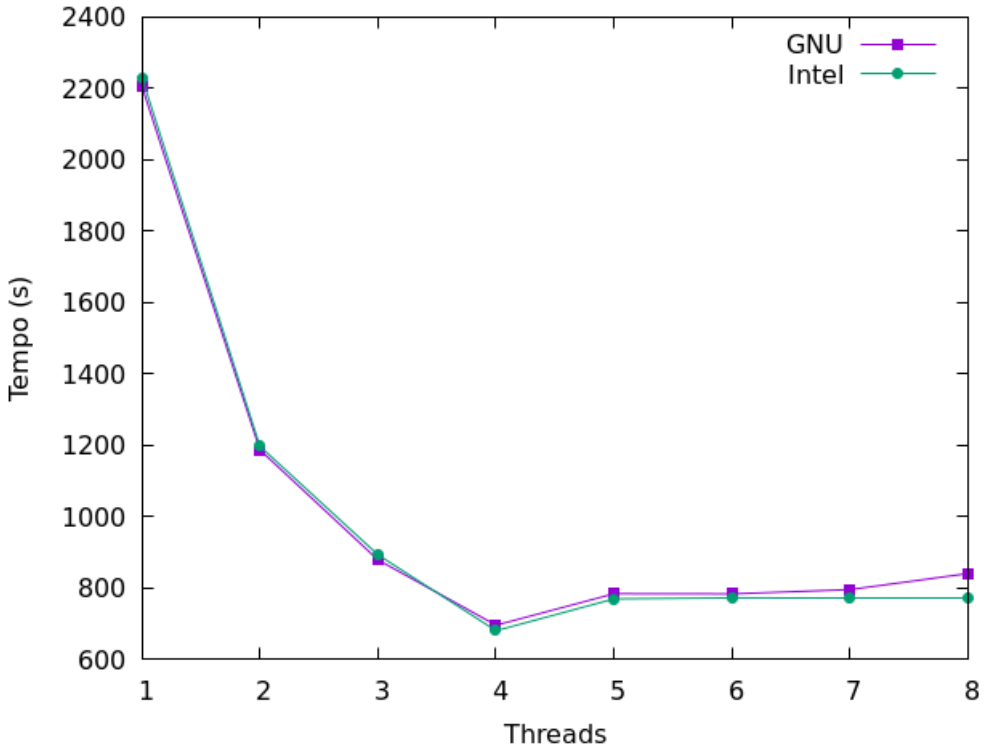

Figura 25 – Malha 560×560

com o tempo quase constante até 8 threads.

Na análise de eficiência, como mostra o gráfico da Figura 26, o compilador da Intel mantém a frente a partir de 4 threads nos melhores valores de eficiência.

Na análise de escalabilidade do código, como mostra a Figura 27, analogamente a malha de 480×480 há um aumento a partir de 4 threads para ambos compiladores na escalabilidade, como também um aumento na última thread para o compilador de GNU, já no compilador da Intel permanece constante após o aumento.

Ao observar que em maior parte das malhas, a partir de 4 threads, os valores de tempo de execução do programa variam pouco entre si, se mantendo muitas vezes quase constantes, uma análise para um número maior de threads se faz interessante. Com essa ideia em mente, utilizando as maquinas do LNCC, abordadas na seção 3.1, realizou-se um estudo de tempo de execução para 24 e 48 threads, comparando-as com 4 theads, que possuiram os menores tempos de execução.

Utilizando as mesmas configurações anteriores para a malha de 560×560, obteve-se uma erro relativo de $1,03 \times 10^{-7}$ e foi feito o gráfico da Figura 28, onde apesar de o compilador da GNU possuir um tempo maior que o da Intel para 4 threads, ele ganha a frente no menor tempo de execução ao aumentar a potência das máquinas.

Na análise de eficiência, como mostra o gráfico da Figura 29, o compilador da GNU ganha uma eficiência de 40% em 48 threads com relação ao melhor tempo adquirido na

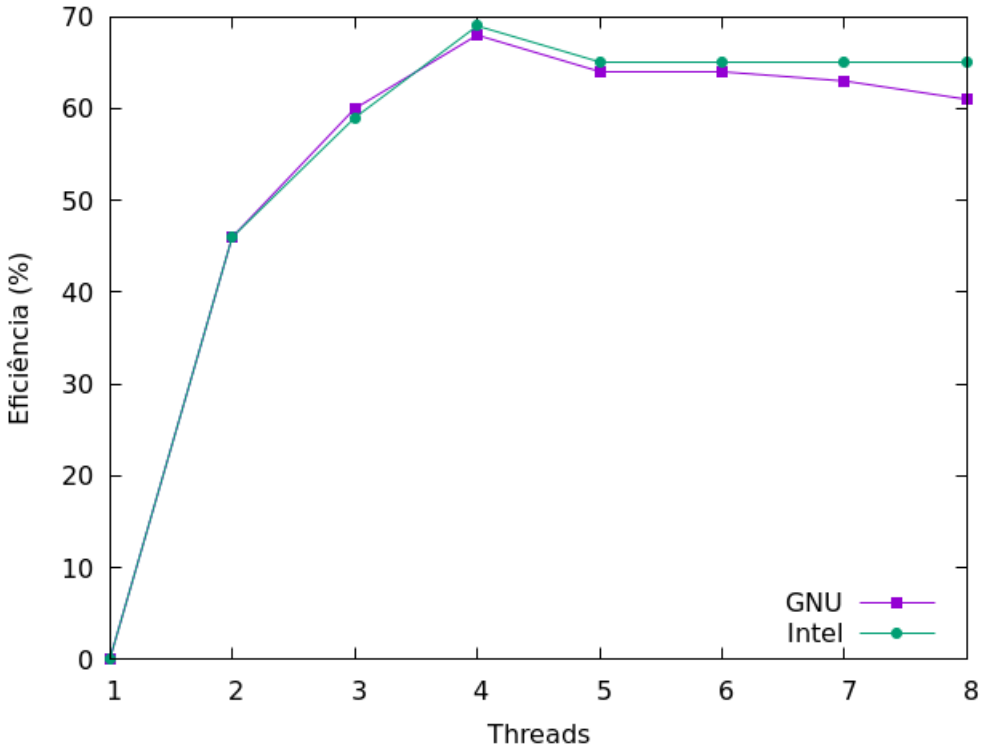

Figura 26 – Malha 560 × 560

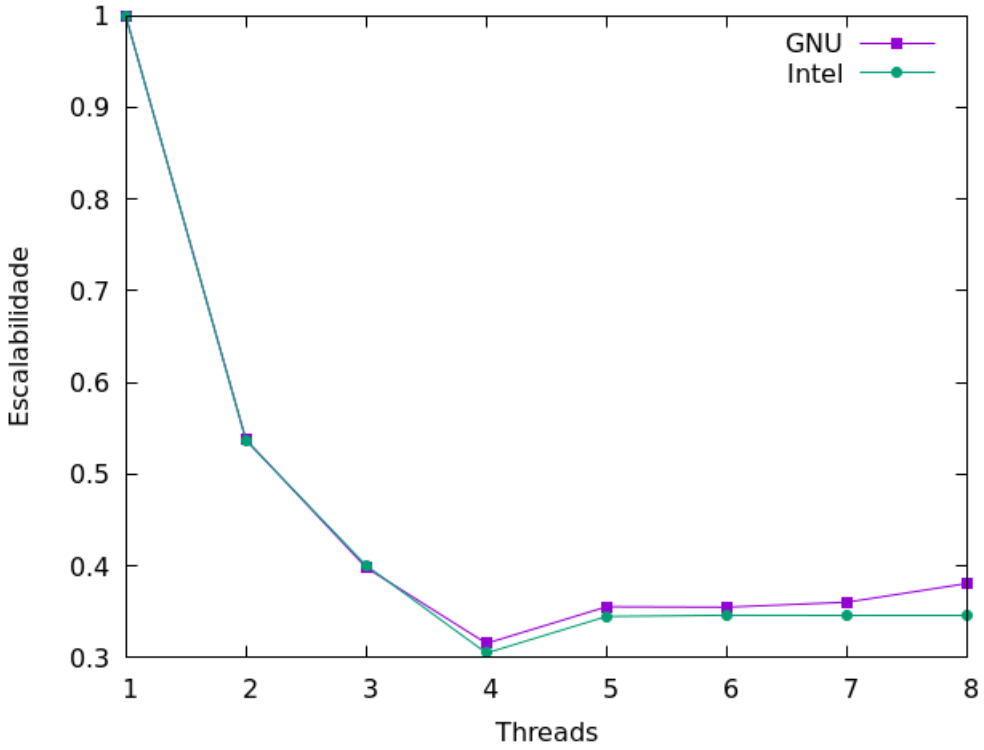

Figura 27 – Malha 560 × 560

malha de 560 × 560.

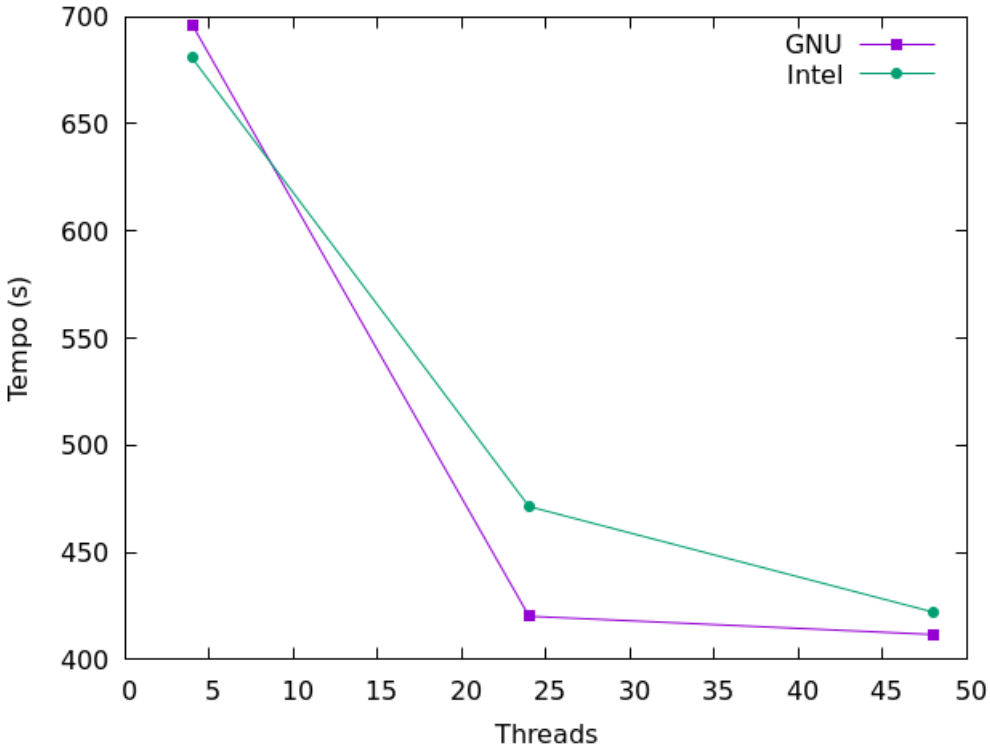

Figura 28 – Malha 560 × 560

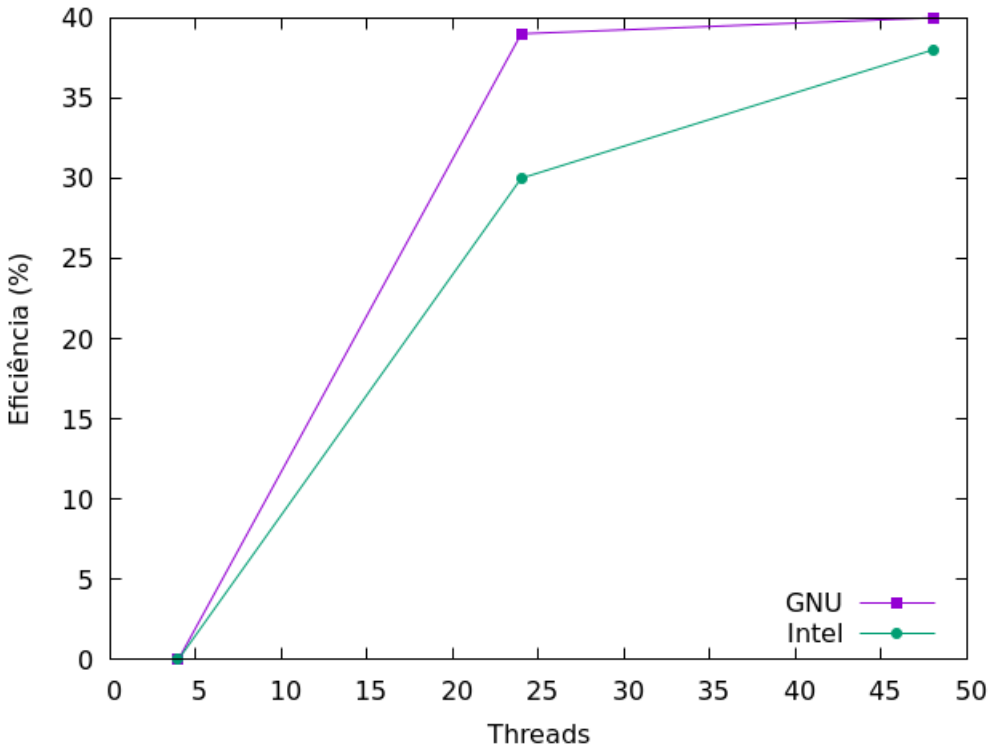

Figura 29 – Malha 560 × 560

Por fim, como mostra a Figura 30, a escalabilidade tem grande variância de 4 para 24 threads em ambos compiladores e baixa variância de 24 para 48 threads.

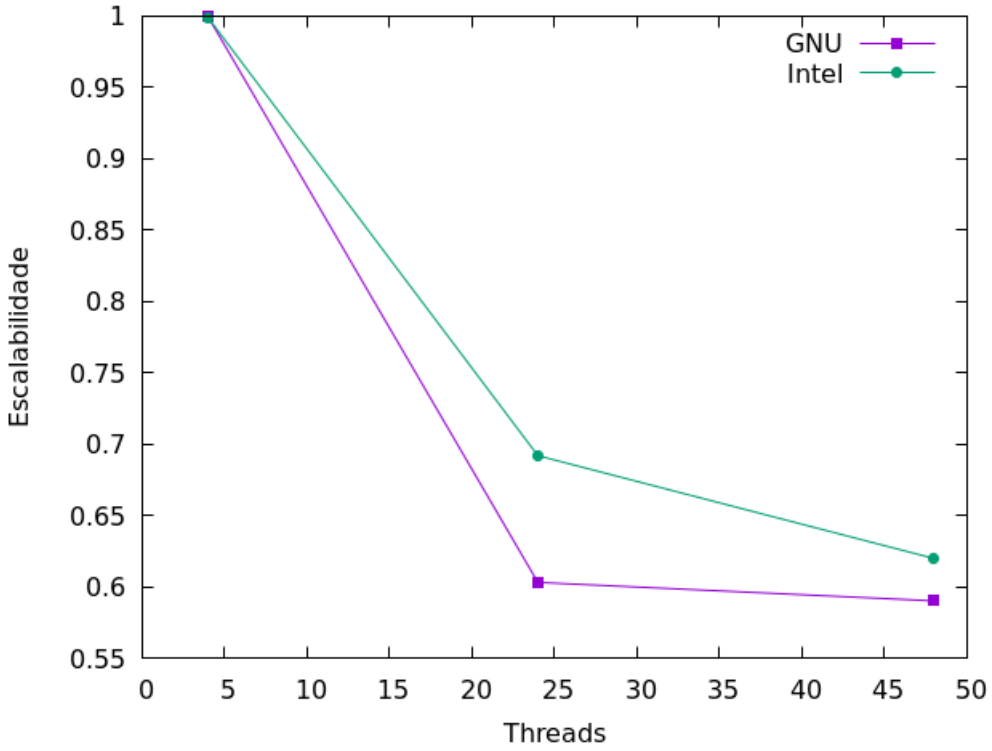

Figura 30 – Malha 560 × 560

5 Conclusões

Após todas as análises de threads e malhas, observa-se que os menores tempos foram todos obtidos com 4 threads, em todas as malhas, além disso, os tempos geralmente crescem ou permacem contantes após este número threads, isso mostra que não é necessário um número maior de threads nestas máquinas se o intuito for adquirir o menor tempo.

Obteve-se um ganho na faixa de 60% à 70% de eficiência em todas as malhas com as máquinas do ICEx e obteve-se de 35% à 40% de eficiência em relação aos menores tempos adquiridos nas máquinas do ICEx com as máquinas do LNCC.

Com relação ao erro relativo obtido entre as malhas, o menor foi de $9,97 \times 10^{-9}$ para a malha de 480×480. Onde constitui uma tolerância muito boa para uma malha grande, porém aumentando a potência das máquinas, utilizando as máquinas do LNCC este erro subiu de $9,99 \times 10^{-8}$ para $1,03 \times 10^{-7}$ para a malha de 560×560, portanto se faz interessante um estudo futuro visando obter erros ainda menores já que os tempos adquirido pelas máquinas do LNCC obtiveram ótimos valores de eficiência com relação as máquinas do ICEx.

Referências

FRANCO, N. M. B. ***Cálculo Numérico***. São Paulo: Universidade de São Paulo. Citado na página 9.

GILAT, A.; SUBRAMANIAM, V. ***Métodos Numéricos para Engenheiros e Cientistas: uma introdução com aplicações usando o MATLAB***. Porto Alegre: Bookman, 2008. Citado na página 8.

HERMANNS, M. ***Parallel Programming in Fortran 95 using OpenMP***. Espanha: Universidade Politécnica de Madrid. Disponível em: <https://www.openmp.org/wp-content/uploads/F95_OpenMPv1_v2.pdf>. Citado na página 11.

OPENMP. 2022. Disponível em: <https://www.openmp.org/>. Acesso em 5 dez. 2022. Citado na página 5.

PILLA, L. L. ***Basics of Vectorization for Fortran Applications***. 2018. Disponível em: <https://hal.inria.fr/hal-01688488/document>. Acesso em 03 dez. 2022. Citado na página 12.