UNIVERSIDADE FEDERAL FLUMINENSE

Método do Relaxamento para a Equação de Laplace

Aluno
Ezequiel Souza dos Santos

Volta Redonda - Rio de Janeiro
02 de fevereiro de 2021

Resumo

Nosso objetivo é analisar e melhorar a performace de um dado programa com diferentes técnicas e ferramentas. Neste caso, analisaremos o programa desenvolvido para resolver a equação de Laplace 2D.

A b s t r a c t

Our objective is to analyze and improve the performance of a given program with different techniques. In this case, we will analyze the program developed to solve the 2D Laplace equation.

1 Introdução

A Teoria do Potencial tem sua importância em áreas como Geodésia, Física, entre outras. A equação abaixo é descrita como *equação de Laplace*, sendo utilizada na solução de problemas físicos através da Teoria do Potencial.

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0.$$

E alguns desses problemas físicos são em Eletroestática, Temperatura e Fluidos. Sendo assim, para a temperatura, temos a equação de calor em estado estacionário.

Neste trabalho, iremos apresentar um algoritmo para resolver a equação de laplace (1), inicialmente criando uma condição de contorno.

2 Objetivo

Para a equação de laplace, é levado em consideração uma placa aquecida e isolada em todos os lugares (exceto em suas bordas), onde a temperatura é constante. Queremos (usando técnicas dos métodos numéricos) achar o menor erro para resolver a equação de laplace.

3 Metodologia

- Discretizando as variáveis independentes

  Seja $S \subset [-M,M] \times [-M,M] \subset \mathbb{R}^2$ uma superfície compacta e $f : S \longrightarrow \mathbb{R}^2$ uma função de classe C^∞. Queremos resolver o problema abaixo de forma computacional.

$$\begin{cases} \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0 \\ V|_S = f \end{cases}$$

  Seja $n \in N$. Aqui, assumiremos uma malha $(n+1) \times (n+1)$ em $\mathbb{R}^2$ sobre S como os conjuntos dos pontos (i, j) tais que

$$i = -M + m_1 \frac{2M}{n}, j = -M + m_2 \frac{2M}{n},$$

  onde $m_1 \geq 0$ e $m_2 \leq n$.

- Determinando V sobre a malha tomada.

  Pela fórmula de Taylor (considerando que f seja uma função diferenciável), temos:

$$f(x+h) = f(x) + f'(x) \cdot h + r(h),$$

  onde $\lim_{h \to 0} \frac{r(h)}{|h|} = 0$. Agora, usando a fórmula de Taylor a derivada parcial com $h = \Delta x = \frac{2M}{n}$, temos

$$\frac{\partial V}{\partial x}(i, j) \cong \frac{V(i+1, j) - V(i, j)}{\Delta x} \tag{3.1}$$

  Mas, por outro lado, podemos escrever a aproximação da derivada das seguintes formas (usando a variação de h na fórmula de Taylor):

$$\frac{\partial V}{\partial x}(i, j) \cong \frac{V(i, j) - V(i-1, j)}{\Delta x} \tag{3.2}$$

$$\frac{\partial V}{\partial x}(i, j) \cong \frac{V(i+1, j) - V(i-1, j)}{2\Delta x} \tag{3.3}$$

Veja que a equação (4) está centralizada no ponto $i+\frac{1}{2}$, e a equação (3.2) está centralizada no ponto $i-\frac{1}{2}$. Por consequência tomamos a segunda derivada sendo:

$$\frac{\partial^2 V}{\partial x^2} \cong \frac{1}{\Delta x}\left[\frac{\partial V}{\partial x}\left(i+\frac{1}{2}\right)-\frac{\partial V}{\partial x}\left(i-\frac{1}{2}\right)\right].$$

Logo,

$$\frac{\partial^2 V}{\partial x^2} \cong \frac{1}{\Delta x}\left[\frac{V(i+1,j)-V(i,j)}{\Delta x}-\frac{V(i,j)-V(i-1,j)}{\Delta x}\right] \cong \frac{V(i+1,j)+V(i-1,j)-2V(i,j)}{(\Delta x)^2}.$$

Temos que $\frac{\partial^2 V}{\partial y^2}$ sai de forma análoga. Por fim, aplicando as derivadas $\frac{\partial^2 V}{\partial x^2}$ e $\frac{\partial^2 V}{\partial y^2}$ na equação de Laplace e isolando $V(i,j)$, temos

$$V(i,j) \cong \frac{1}{4}\left[V(i+1,j)+V(i-1,j)+V(i,j+1)+V(i,j-1)\right]. \tag{3.4}$$

A equação (3.4) anterior mostra que o potencial em (i,j) é a média de V nos pontos próximos.

Nossa abordagem é usar técnicas dos métodos numéricos para determinar a função V (3.4). O método que usaremos é o *método da relaxação*. Sendo assim, tomaremos as seguintes técnicas: Inicializamos a placa e as condições de contorno usando W, onde $V = W$ no primeiro palpite. Após isso, aplicamos a função V (3.4) em W para obter um palpite melhor, gerando assim um novo palpite V_1. Repetimos este processo até gerar o palpite com menor erro.

4 Implementação

Veja a seguir a implementação do método da relaxação no algoritmo.

- **Construção**

  Basicamente, queremos calcular V até gerar o valor esperado (convergência), seguindo assim os seguintes passos:

  - Iniciamos a placa e as conções de contorno em W (sendo V=W no palpite inicial).
  - Através do palpite V, aplicamos a equação (3.4) para gerar um palpite melhor. Isto é, tomando V_n a n-ésima iteração, aplicamos a equação (3.4) para gerar V_{n+1}.
  - Atualizamos V repetidas vezes até que tenhamos o menor erro estipulado.

  Foram utilizados 3 funções no algoritmo: A primeira função, *inicializa_W*, inicializa a placa e as condições de contorno em W. A segunda função, *novo_V*, aplica V na equação (3.4) para gerar um valor mais preciso para V. Já a terceira função, *calcula_laplace*, chama a função novo_V para testar a convergência do algoritmo. Vejamos a seguir como funciona novo_V e calcula_laplace.

  1. **novo_V**

     Primeiramente, tomaremos os parâmetros Wmp_fim, que representa a n-ésima etapa de V, e Vmp, que é a $(n+1)$-ésima etapa obtida através da atualização de V. Assim,

     - Tome delta_t= 0, onde delta_t representa a mundança de V através da atualização.
     - Façamos um loop em (i,j):

       Vmp=$\frac{1}{4}$ [Wmp_fim $(i+1,j)$ + Wmp_fim $(i-1,j)$ + Wmp_fim $(i,j+1)$ + Wmp_fim $(i,j-1)$]
     - Ponhe $|\text{Vmp}(i,j) - \text{Wmp\_fim}(i,j)|$ em delta_t.

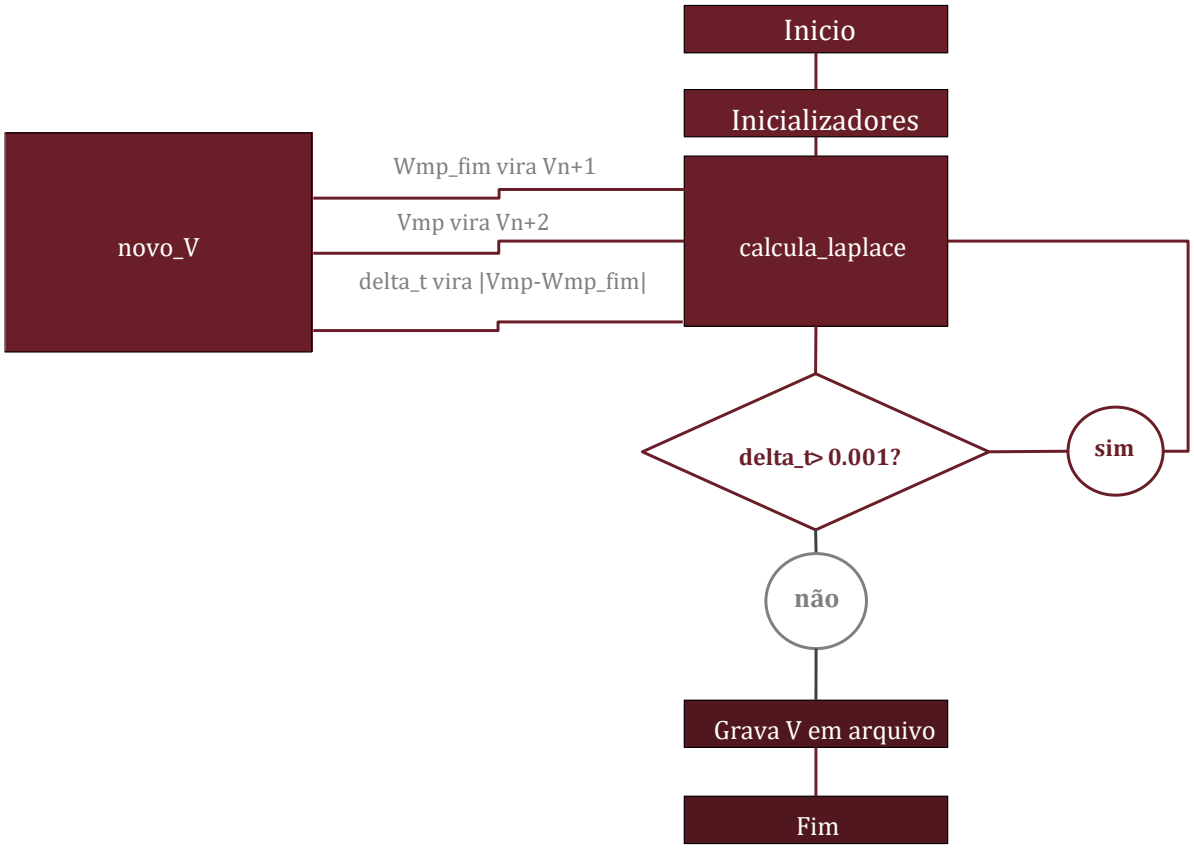
  2. **calcula_laplace**

- Ainda com os parâmetros Vmp e Wmp_fim:

  chame a função "novo_V", onde Wmp_fim vira um palpite melhor de V, isto é, Wmp_fim=Vmp.
- Calcule:

  $\text{Vmp}=\frac{1}{4}\left[\text{Wmp\_fim}\,(i+1,j)+\text{Wmp\_fim}\,(i-1,j)+\text{Wmp\_fim}\,(i,j+1)+\text{Wmp\_fim}\,(i,j-1)\right]$
- Ponhe delta_t= $|\text{Vmp}(i,j)-\text{Wmp\_fim}(i,j)|$.
- Verifique se delta_t<0.001. Caso sim, retorne delta_t e termine a função. Caso não, chame a função calcula_laplace novamente levando em consideração os mesmos parâmetros.

Vejamos abaixo o fluxograma.

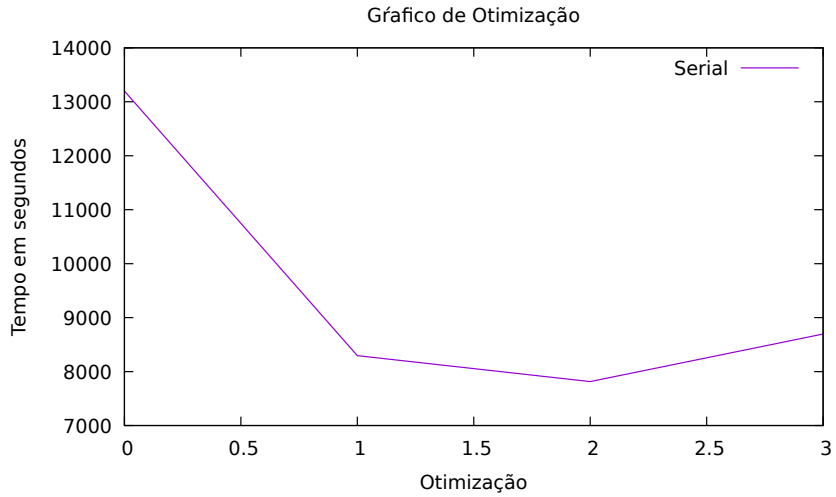


5 Benchmark e profile do código em serial

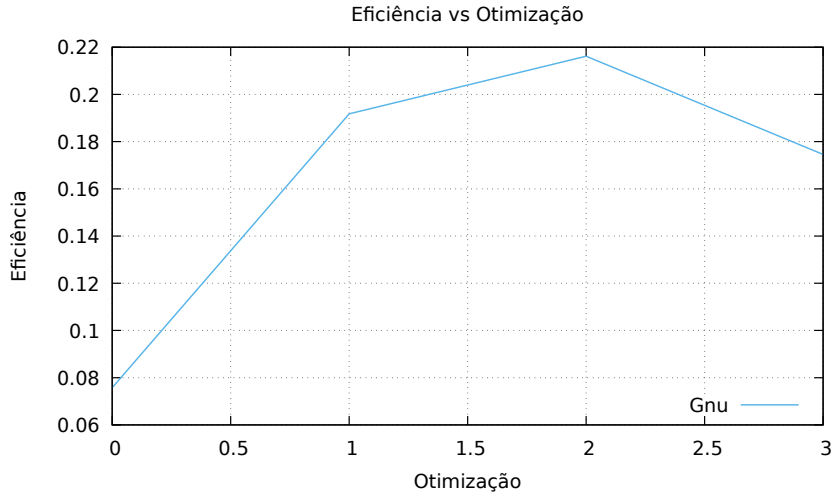
O primeiro código em serial obteve 266 minutos sendo rodado numa malha 10000×10000 com 100000000 de pontos. No nosso programa, é feito um laço simples para obter delta _ t e um valor mais preciso para V, o que faz o programa levar mais tempo de processamento. Mais precisamente, analisando o profile, o maior tempo de processamento (gargalo) no código está no processo que atualiza a matriz V (3.4). Visto isso, utilizamos diferentes flags (-O0, -O1, -O2, -O3) para melhorar o desempenho do código.

Otimização	Segundos
0	13200
1	8296
2	7814
3	8695

Podemos ver uma melhora significativa entre as flags -O0 e -O2. A flag -O0 roda o programa em 13200 segundos (220 minutos) e a flag -O2 roda o programa em 7814 segundos (130 minutos), implicando numa melhora de aproximadamente 65% no tempo de processamento do programa. Por outro lado, não vimos uma melhora entre as flags -O2 e -O3, pois a flag -O2 rodou o programa em 7814 segundos (130 minutos) e a flag -O3 rodou o programa em 8695 segundos (144 minutos), e este fato pode ocorrer pelo uso de memória ou processamento/gargalo do programa (sendo estes bastante comum).



Usando a flag -O2, a eficiência do programa é próxima de 22%. Já com a flag -O3, a eficiência do programa cai de 18%.



5.1 Máquina utilizada

A máquina utilizada para obter os resultados foi o Labotatório 107C.

```
(base) [aluno@localhost ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 42
Model name:            Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
Stepping:              7
CPU MHz:               3800.097
CPU max MHz:           3800,0000
CPU min MHz:           1600,0000
BogoMIPS:              6784.74
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi
 mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
 nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid
sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx lahf_lm epb ssbd ibrs ibpb stibp tpr_shadow vnmi f
lexpriority ept vpid xsaveopt dtherm ida arat pln pts md_clear spec_ctrl intel_stibp flush_l1d
```

Figura 1: Laboratório 107C

6 Otimizações do código em serial

Como dito anteriormente, no nosso programa anterior, era feito um laço simples para obter delta _ t e um valor mais preciso para V, o que fazia o programa levar mais tempo de processamento. Como mudança, criamos duas funções: "novo_V"e "calcula_laplace", usando uma estratégia recursiva na função "calcula_laplace". Como já visto, a função "novo_V"aplica V na equação (3.4) para gerar um valor mais preciso para V. Já a função "calcula_laplace", chama a função "novo_V"para testar a convergência do algoritmo. Essa modificação fez o código rodar em 215 minutos (sendo que o programa anterior roda em 266 minutos). Isso siginifica que o código otimizado melhorou 21% em relção ao código anterior.

```
while ( delta_t > tolerance && iter <= numero_iteracoes ) {

        delta_t = 0.0;

        for(i = 1; i <=  I; i++) {
                for(j = 1; j <=J; j++) {
                        Vmp[i][j] = 0.25*(Wmp_fim[i+1][j] + Wmp_fim[i-1][j] +
                        Wmp_fim[i][j+1] + Wmp_fim[i][j-1]);
                }
        }
        for(i = 1; i <= I; i++){
                for(j = 1; j <= J; j++){
                        delta_t = fmax( fabs(Vmp[i][j]-Wmp_fim[i][j]), delta_t);
                        Wmp_fim[i][j] = Vmp[i][j];
                }
        }
        iter++;
}
```

Figura 2: Programa anterior

```
void novo_V(double Vmp[I+2][J+2], double Wmp_fim[I+2][J+2]){

        int i, j;
        delta_t = 0.0;

        for(i = 1; i <= I; i++) {
                for(j = 1; j <= J; j++) {
                        Vmp[i][j] = 0.25*(Wmp_fim[i+1][j] + Wmp_fim[i-1][j] +
                        Wmp_fim[i][j+1] + Wmp_fim[i][j-1]);
                }
                for(j = 1; j <= J; j++){
                        delta_t = fmax( fabs(Vmp[i][j]-Wmp_fim[i][j]), delta_t);
                        Wmp_fim[i][j] = Vmp[i][j];
                }
        }

}

double calcula_laplace(double Vmp[I+2][J+2], double Wmp_fim[I+2][J+2]){

        novo_V(Vmp, Wmp_fim);
        iter++;
        if(delta_t > tolerance && iter <= numero_iteracoes) calcula_laplace(Vmp, Wmp_fim);
        return delta_t;
}
```

Figura 3: Programa novo

Observação. A máquina utilizada foi o Laboratório 107C. E o compilador foi o gfortran.

7 Implementação em OpenMP

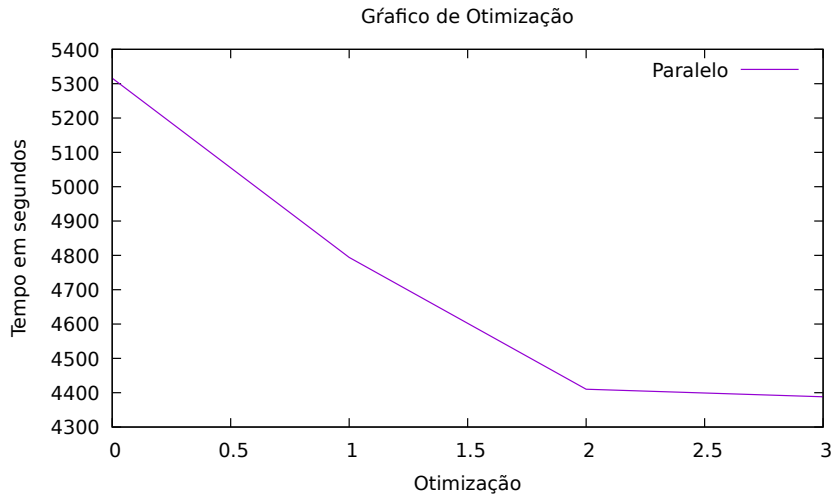
Fizemos apenas uma paralelização no código de um loop na função "novo_V".

```
void novo_V(double Vmp[I+2][J+2], double Wmp_fim[I+2][J+2]){

        int i, j;
        delta_t = 0.0;
        #pragma omp parallel for private(j) reduction(max:delta_t)
        for(i = 1; i <= I; i++) {
                for(j = 1; j <= J; j++) {
                        Vmp[i][j] = 0.25*(Wmp_fim[i+1][j] + Wmp_fim[i-1][j] +
                        Wmp_fim[i][j+1] + Wmp_fim[i][j-1]);
                }
                for(j = 1; j <= J; j++){
                        delta_t = fmax( fabs(Vmp[i][j]-Wmp_fim[i][j]), delta_t);
                        Wmp_fim[i][j] = Vmp[i][j];
                }
        }


}
```

Figura 4: Paralelização

7.1 Benchmark do programa em pararelo

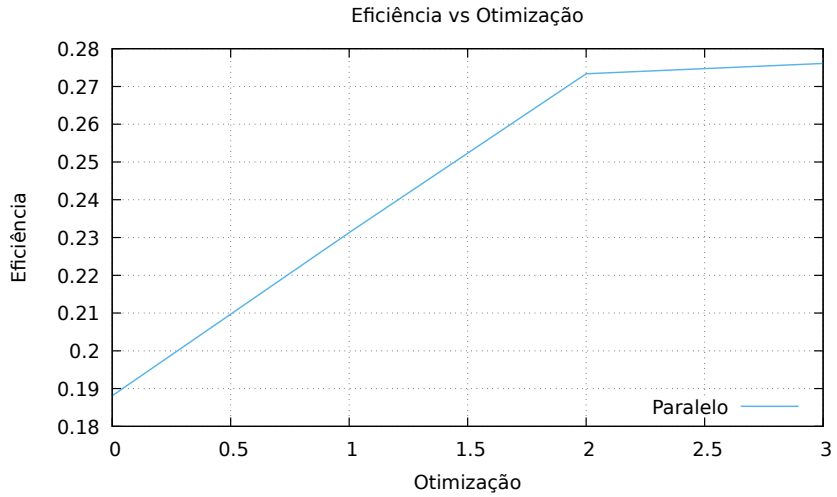
Após a paralelização do código, o mesmo rodou em aproximadamente 100 minutos, significando uma redução de aproximadamente 36% no tempo de processamento do código (em relação ao serial). Isso representa um desenpenho maior do código em paralelo se comparado ao código em serial. O mesmo código em paralelo foi rodado em diferentes flags, e percebeu-se que o aumento de desempenho do código está diretamente ligado ao aumento de flags.



Usando a flag -O0, o código rodou em 5316 segundos (88,6 minutos) e ao usar a flag -O3, o código foi rodado em 4388 segundos (73,13333 minutos), sendo isto uma redução de aproximadamente 20% no tempo de processamento. Vejamos abaixo a tabela que foi usada para gerar os dados.

Otimização	Segundos
0	5316
1	4794
2	4410
3	4388

Consequentemente, à medida que aumentamos o números de flags a eficiência do código aumenta. Usando a flag -O0, a eficiência do código é próxima de 19%. E usando a flag -O3, a eficiência do código é próxima de 28%.



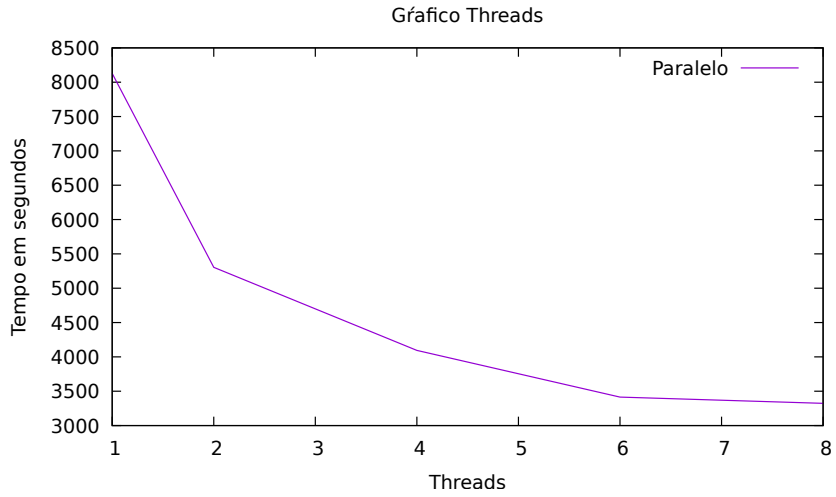
Observação. A máquina utilizada foi o Laboratório 107C.

8 Benchmark do programa com várias threads

Rodaremos o programa em paralelo com diferentes threads com o compilador gfortran (usando diferentes máquinas).

8.1 Máquina Laboratório 107C

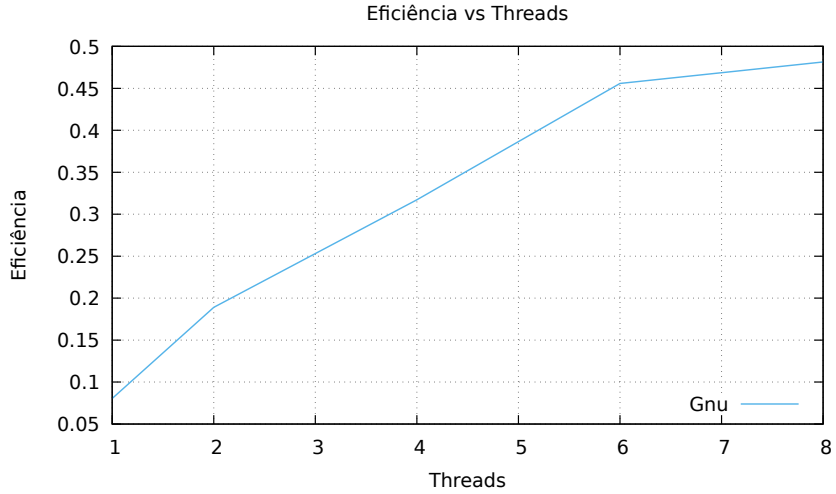
Como a máquina Laboratório 107C possui apenas 8 threads, rodaremos o programa em paralelo com as threads 1, 2, 4, 6 e 8 e a flag -O3. Conforme os resultados obtidos, o maior tempo de processamento do código foi de 8127 segundos usando 1 thread. Com 8 threads, o programa teve seu menor tempo de processamento, totalizando 3323 segundos.



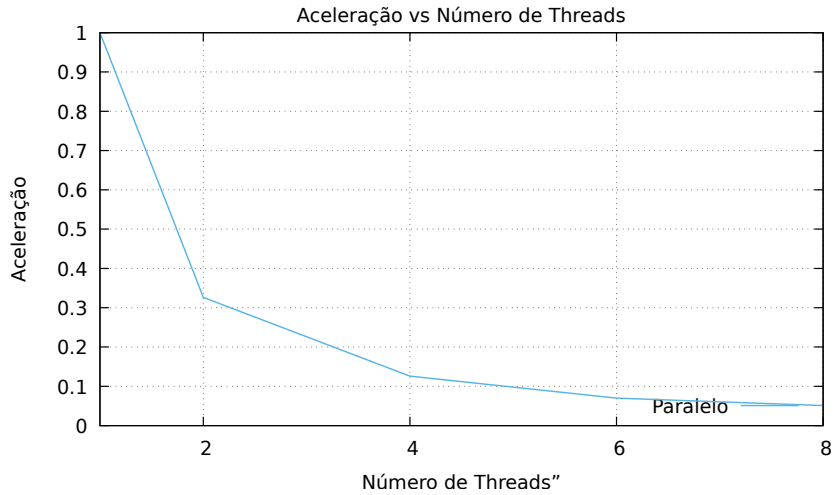
O que era de se esperar, o aumento de threads melhora significativamente a performace do programa. Veja abaixo a tabela que usamos para gerar o gráfico.

Threads	Segundos
1	8127
2	5304
4	4094
6	3415
8	3323

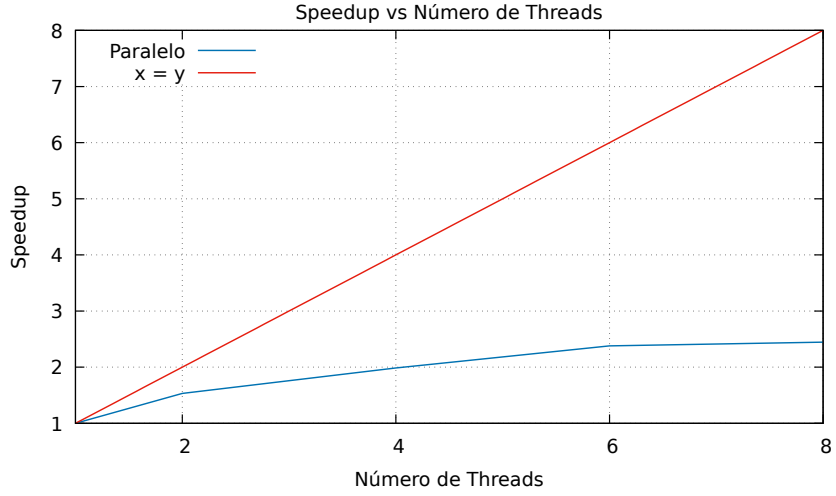
Por consequência, o aumento de threads faz o programa ter mais eficiêcia. Com 2 thread, a eficiêcia do código é de aproximadamente 20%. Já com 8 threads, a eficiência do código é próxima de 50%.



Agora, considerando a aceleração do código, normalizamos com o de uma thread. Dito isto, vimos que a aceleração do programa aumenta ao passo que aumentamos o número de threads.



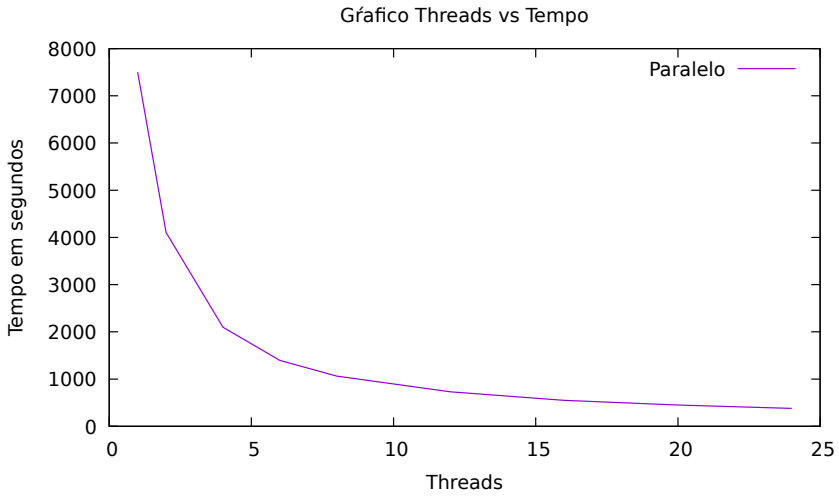
Além disso, podemos ver um aumento no speedup (à medida que aumentamos o número de threads). Isso implica numa melhoria significativa no desempenho do código na medida que aumentamos os threads. Além disso, vimos que o speedup é sub-linear, isso porque fica abaixo do gráfico $x = y$.



Lembrando que o speedup é o quociente entre o tempo em serial (sendo este de 1 thread) e o tempo em paralelo.

8.2 Máquina LNCC-B710

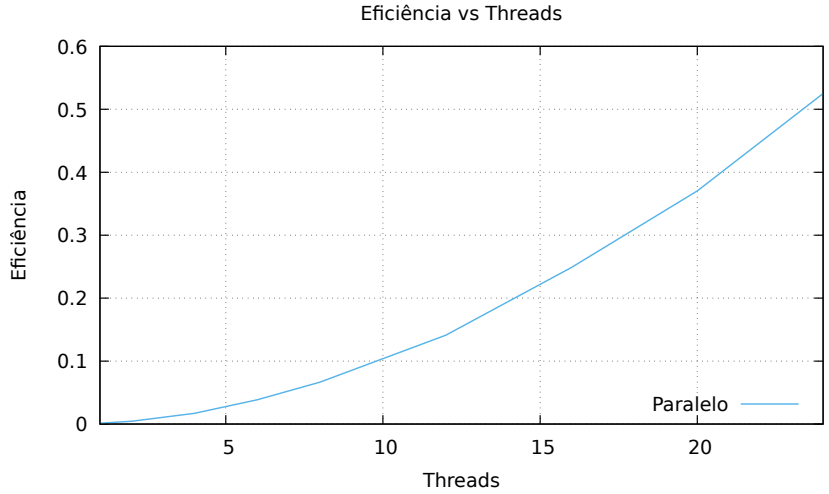
Como a máquina Laboratório B710 possui 24 threads, rodaremos o programa em paralelo com as threads 1, 2, 4, 6, 8, 12, 16, 20 e 24 e a flag -O2. Após a análise dos resultados: vimos que a máquina com 1 thread roda o código em 7501 segundos, sendo este o maior; o programa com 24 threads roda o código em 378 segundos, sendo este o menor.



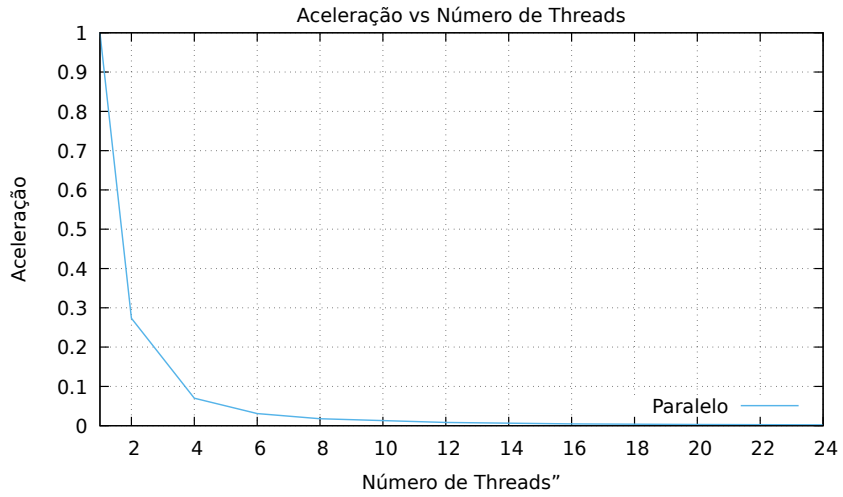
Isso mostra que o aumento de threads (nesta máquina) aumenta o desempenho do código.

Threads	Segundos
1	7501
2	4104
4	2097
6	1395
8	1062
12	729
16	549
20	450
24	378

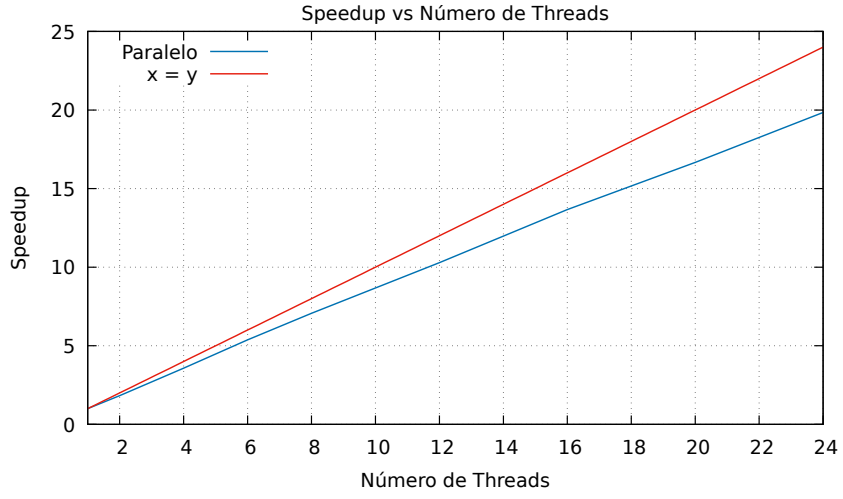
Percebemos também que com 2 thread o programa tem eficiência cgegando próximo de 10%. Já com 24 threads, o programa passa de 50% de eficiência.



É nitido também que o aumento de threads faz programa ficar mais acelerado. Veremos abaixo o gráfico "Aceleração vs Threads", ao qual foi normalizado com o de uma thread.



Por fim, vimos que o speedup tem um aumento significativo quando aumentamos o número de threads. Neste caso, o speedup é sub-linear.

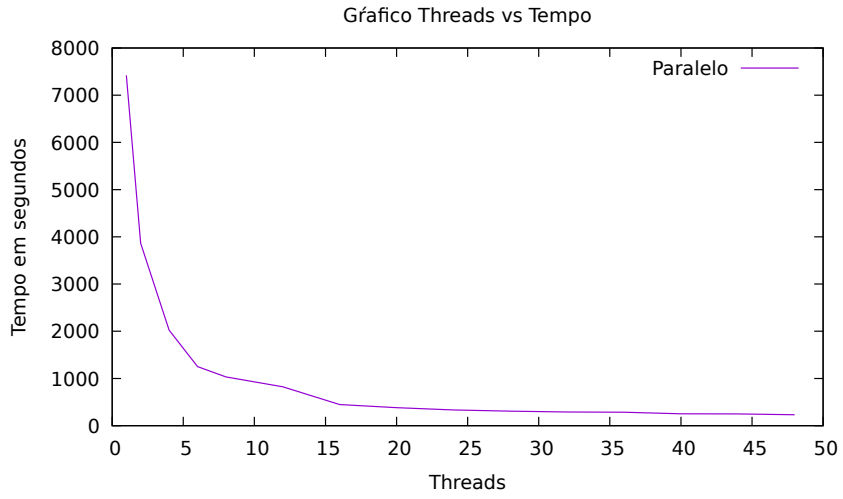


Observação. Máquina B710

```
[ezequiel.santos@sdumont11 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Stepping:              4
CPU MHz:               2957.080
CPU max MHz:           3200,0000
CPU min MHz:           1200,0000
BogoMIPS:              4799.78
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-11
NUMA node1 CPU(s):     12-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi
 mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl x
topology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pd
cm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm epb ssbd ibrs ibpb
 stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm ida arat pln pts md_clear spec_
ctrl intel_stibp flush_l1d
```

8.3 Máquina LNCC-SequanaX

Como a máquina Laboratório SequanaX possui 48 threads, rodaremos o programa em paralelo com as threads 1, 2, 4, 6, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44 e 48 e a flag -O2. Pelos resultados, observa-se que o programa com 1 thread roda em 7420 segundos. Por outro lado, o programa com 48 threads roda em 232 segundos.



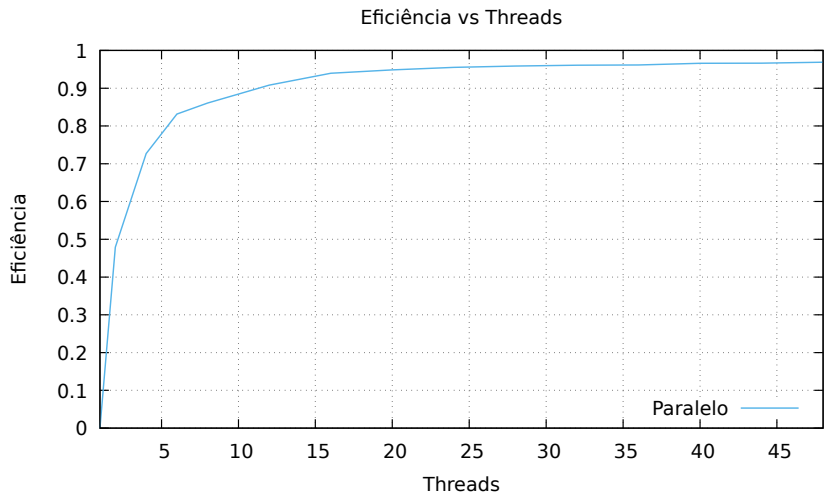
Basicamente, ao aumentarmos as threads, percebemos um aumento significativo no desempenho do programa.

Threads	Segundos
1	7420
2	3865
4	2025
6	1250
8	1032
12	681
16	448
20	382
24	333

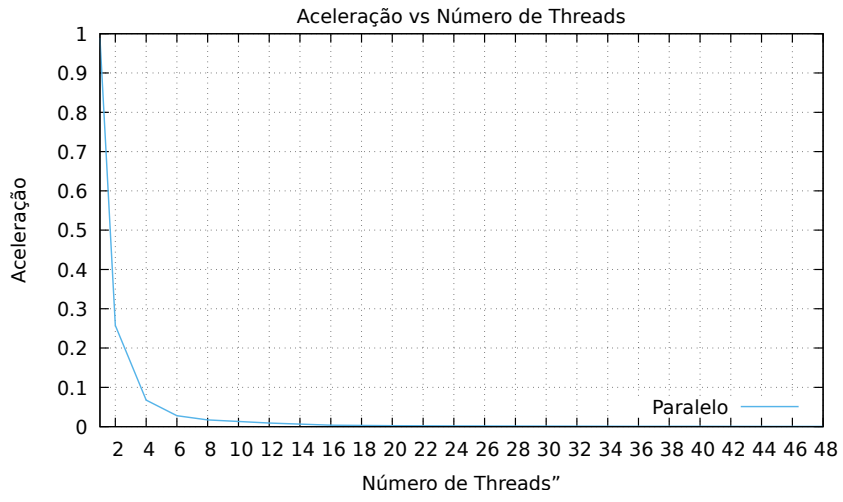
Threads	Segundos
28	307
32	291
36	285
40	253
44	250
48	232

Figura 5: Tabela de dados

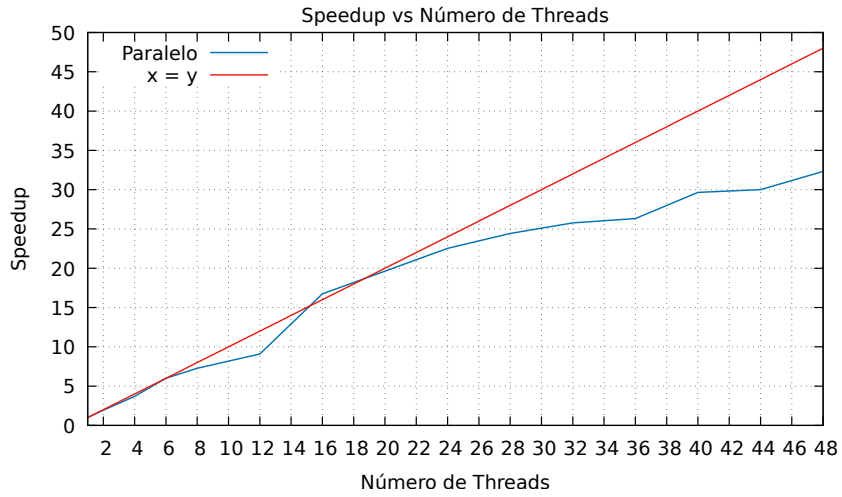
Com estes resultados, nota-se o seguinte: com 2 threads o programa teve aproximadamente 47% de eficiência; com 48 theads o programa chegou perto de 100% de eficiência.



Consequentemente, o aumento de threads faz o programa ficar mais acelerado, o que era de se esperar.



Finalizando, percebe-se que à medida que utilizamos o máximo de threads, o speedup aumenta. Visto isso, podemos dizer que o speedup é sub-linear. Porém, entre as threads 16 e 18, nota-se que o speedup fica um pouco acima da reta $x = y$.



Observação. Máquina SequanaX

```
[ezequiel.santos@sdumont18 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                88
On-line CPU(s) list:   0-87
Thread(s) per core:    2
Core(s) per socket:    22
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz
Stepping:              4
CPU MHz:               1561.798
CPU max MHz:           3700,0000
CPU min MHz:           1000,0000
BogoMIPS:              4200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              30976K
NUMA node0 CPU(s):     0-21,44-65
NUMA node1 CPU(s):     22-43,66-87
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi
 mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good no
pl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma
 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
 abm 3dnowprefetch epb cat_l3 cdp_l3 invpcid_single intel_ppin intel_pt ssbd mba ibrs ibpb stibp tpr_shadow vnm
i flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx5
12dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc c
qm_mbm_total cqm_mbm_local dtherm ida arat pln pts pku ospke md_clear spec_ctrl intel_stibp flush_l1d
```

9 Validação dos resultados

Temos que a máquina SequanaX é mais eficiente do que as máquinas 107C e B710. Além disso, usando o programa em serial com a flag -O2, a máquina SequanaX possui uma melhoria de 39% em relação a máquina Lab 107C e uma melhoria de 33% em relação a máquina B710. Agora, usando o programa em paralelo com a flag -O2, a maquina SequanaX possui uma melhoria de 93% em relação a máquina Lab 107C e uma melhoria de 45% em relação a máquina B710.

	Flag -O2	
	Paralelo	Serial
Lab 107C	4410	7814
Lab B710	574	7380
Lab SequanaX	310	4920

Figura 6: Comparações

Devemos ressaltar que o programa em paralelo é mais eficiente que o serial, visto que o paralelo roda em 100 minutos e o serial em 215 minutos (isso sem usar nenhuma flag).

10 Conclusões

Neste projeto, podemos perceber as diferenças entre os programas em serial e paralelo. O programa em paralelo possui uma eficiência melhor em relação ao serial, onde o serial foi rodado em 215 minutos e o paralelo em 100 minutos. Além do mais, o aumento de flags melhora o tempo de execução do programa (sendo a flag -O2 a que desempenhou um papel melhor). Com isso, também vimos que o aumento de threads traz muito mais eficiência para o programa. Combinando esses fatores ao uso de máquinas diferentes (Lab 107C, B710 e SequanaX), foi constatado um desempenho melhor ao usar a máquina SequanaX (dada a qualidade da máquina). Ou seja, em relação a métrica sobre a performace, a máquina SequanaX obteve um desempenho melhor em relação as outras máquinas.

Portanto, as técnicas e análises aqui apresentadas são de extrema importância no meio tecnológico/-científico. Tal importância vem do fato de que tais procedimentos podem diminuir custos de um projeto, tempo de desenvolvimento, tamanho de um dispositivo ou o quanto ele consome de energia, capacidade de processamento, entre outros.

Referências

[1] Sterling, Thomas and Brodowicz, Maciej and Anderson, Matthew. *High performance computing: modern systems and practices,*. 2017.

[2] Giordano, Nicholas J. *Computation Physics, Prentice Hall,Upper Saddle River,*. 2006.