Otimizando o problema da difusão de calor bidimensional

ALEXANDRE ESPOSTE SANTOS[1]*

[1] Curso de Bacharelado em Física com ênfase em Física Computacional,
Instituto de Ciências Exatas, Universidade Federal Fluminense,
27213-145 Volta Redonda - RJ, Brasil

Resumo

No presente trabalho otimizaremos a solução da equação de difusão de calor sobre uma placa cuja a malha é uma matriz de 1440x1440. Através de técnicas de programação, análises de gargalos, utilização de flags de otimização e de paralelismo obtivemos uma performance máxima com eficiência de 79,37%, o que resultou em um programa 4,84x mais rápido.

Palavras-chave: *Otimização, Calor, Paralelismo, Eficiência, Computação.*

Abstract

In the present work we will optimize the solution of the heat diffusion equation on a plate whose mesh is a matrix of 1440x1440. Through programming techniques, bottleneck analysis, use of optimization flags and through parallelism we obtained a maximum performance with efficiency of 79.37% which resulted in a 4.84x faster program.

Keywords: *Optimization, Heat, Parallelism, Efficiency, Computing.*

*alexandreesposte@id.uff.br

1 Introdução

Com o surgimento da computação, a programação vem contribuindo cada dia mais para o avanço da sociedade. Na física utilizamos da computação para solucionar problemas muito complexos de serem resolvidos analiticamente. A simulação de problemas físicos surge das teorias e leis que regem tal fenômeno que em geral se dá em uma equação diferencial, seja ela ordinária ou parcial.

A solução dessas equações pode ser um tanto quanto custoso, podendo levar horas, dias, meses ou até anos dependendo das características do problema ou até mesmo do resultado que se deseja obter. Portanto, devido ao anseio de se obter soluções em tempos cada vez menores, foram desenvolvidas tecnologias tais como a api openMP e a biblioteca openMPI, sendo ambas tecnologias voltadas para programação paralela onde tal técnica, em muitos casos, nos garante um maior desempenho na execução do código. Além da paralelização, se fez necessário também um melhor entendimento sobre o funcionamento da cpu e das linguagens de programação para que assim possamos escrever um código da forma mais otimizada possível, e por fim se faz o uso de flags de compilação e flags da cpu para uma otimização mais profunda de nossos códigos. Tais técnicas e tecnologias nos garantiu em muitas situações um maior desempenho de nos-

sos programas, sendo assim a execução rápida e realizada de forma eficiente.

Nos tópicos 2 e 3 encontram-se, respectivamente, o objetivo e a metodologia aplicada afim de resolver o problema deste trabalho.

Nas seções 4 e 5 apresentamos a implementação da lógica do código realizado e efetuamos algumas comparações relevantes acerca de otimizações empregadas.

No tópico 6 analisamos o profile do código para assim localizar gargalos. Enquanto isso, no tópico 7 discutimos sobre as técnicas de programação utilizadas.

Nas seções 8 e 9 fazemos, respectivamente, uma discussão mais aprofundada sobre as otimizações empregadas, assim como a utilização do openMP para paralelizar nossos gargalos.

No tópico 10 fazemos uma discussão acerca dos resultados obtidos, em 11 e 12 fazemos respectivamente, uma discussão mais aprofundada sobre a biblioteca openMPI assim como a implementação e discussão acerca dos resultados obtidos. Sendo nas duas últimas a validação de nossos resultados e as respectivas conclusões.

2 Objetivo

Neste trabalho temos três objetivos principais, onde o primeiro consiste em solucionar numericamente a equação da difusão de calor em duas dimensões.

Como segundo objetivo demonstraremos por sua vez que boas técnicas de programação aliadas a utilização de flags de compilação e de cpu já nos garantem um desempenho razoável.

Mostraremos por último um excelente desempenho devido às otimizações realizadas.

3 Metodologia

O problema abordado neste trabalho trata-se da equação de difusão do calor, eq.(1), dada por uma equação diferencial parcial categorizada como uma equação parabólica [1].

$$\frac{\partial u}{\partial t} = \alpha^2 \nabla^2 u, \tag{1}$$

onde u é a temperatura e α é o coeficiente de difusão térmica, conhecido também por difusividade térmica. Por vezes é encontrando na literatura α em vez de α^2.

O problema físico que desejamos solucionar consiste em uma placa quadrada de um material qualquer representado pela difusidade térmica α. Logo, devido a bidimensionalidade de nosso problema a eq.(1) pode ser reescrita como mostrada na expressão (2)

$$\frac{\partial u}{\partial t} = \alpha^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right). \tag{2}$$

Considera-se a difusidade térmica como sendo homogênea em toda a placa e com seu valor dado por $\alpha = 0.1\ m^2 s^{-1}$.

As condições de contorno e iniciais do problema podem ser vistas nas equações (3) e (4), respectivamente,

$$u(x,0,t) = u(0,y,t) = u(1,y,t) = u(x,1,t) = 293K, \tag{3}$$
$$0 \leq x \leq 1 \ \ e \ \ 0 \leq y \leq 1,$$

$$u(x,y,0) = 0K. \tag{4}$$

Tem-se na placa dois conjuntos de fontes, os quais são pontos equidistantes do centro. O primeiro e segundo conjunto de fontes estão localizados, respectivamente, a um raio de 100 e 300 milímetros do centro da placa.

A solução consiste de um looping principal responsável por executar as atualizações das temperaturas na placa em cada instante de tempo. O looping ocorre por 1 milhão de iterações até que então seja finalizado.

Da iteração 1 até a iteração 500 mil os dois conjuntos de fontes aquecem a placa. O conjunto de fontes de menor raio aquece a placa a $1000K$ enquanto o conjunto de fontes, de maior raio, aquece a mesma a $500K$. Da iteração 500 mil a 1 milhão os dois conjuntos, ambos a $400K$, refrigeram a placa.

Afim de obter uma solução aproximada da eq.(2), iremos utilizar o método da diferença direta, conhecido também como método explicito [1]. Inicialmente devemos discretizar nosso domínio, obtendo-se assim um conjunto finito de pontos, sendo esse conjunto de pontos conhecido como malha. A solução é obtida sobre os pontos que formam a malha [1].

Posteriormente devemos substituir as derivadas da eq.(2) por derivadas numéricas e, por fim, obter equações que quando resolvidas nos retorna o resultado desejado [1].

A discretização da eq.(2) é explicitada conforme a eq.(5) [1],

$$\frac{u_{i,j}^{t+1} - u_{i,j}^{t}}{\Delta t} = \alpha \left(\frac{u_{i+1,j}^{t} - 2u_{i,j}^{t} + u_{i-1,j}^{t}}{\Delta x^2} + \frac{u_{i,j+1}^{t} - 2u_{i,j}^{t} + u_{i,j-1}^{t}}{\Delta y^2} \right), \tag{5}$$

obtemos a solução isolando $u_{i,j}^{t+1}$. Tem-se também que $h = \Delta x = \Delta y$ [1], portanto,

$$u_{i,j}^{t+1} = \lambda \left(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{i,j}^{t} \right) + u_{i,j}^{t}, \tag{6}$$

sendo $\lambda = \dfrac{\alpha \Delta t}{h^2}$.

Observa-se da eq.(6) que a temperatura na posição (i,j) no tempo t+1 depende da temperatura da placa no tempo t. Este método é conhecido como método explicito ou direto, entretanto, existe também o método implícito e o método de Crank-Nicolson [1].

Para que haja convergência do método explicito, é necessário que [1],

$$\lambda \leq \frac{1}{2}. \tag{7}$$

Utilizou-se para a simulação da placa uma malha de 1440x1440, o tempo máximo de 1 segundo e uma variação por iteração de 1μs, o que nos garante um total de 1 milhão de iterações. Para a difusidade térmica utilizou-se 0.1 $m^2 s^{-1}$.

Para os valores utilizados, obteve-se $\lambda \approx 0.2$, garantindo-se assim a convergência do método.

Na Fig.(1) encontra-se a placa em seu instante inicial, na Fig.(2) encontra-se a placa na metade do tempo e na Fig.(3) encontra-se a placa no tempo final.

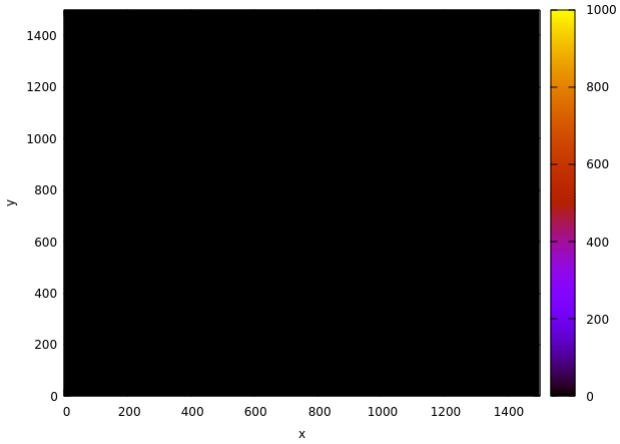

Figura 1: *Placa no instante t = 0s. Fonte: Autor*

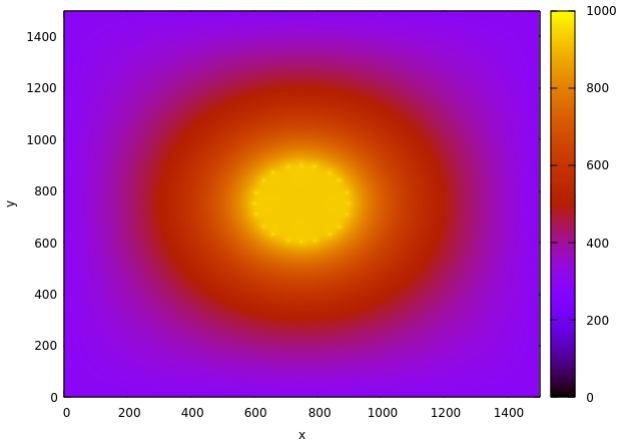

Figura 2: *Placa no instante t = 0.5s. Fonte: Autor*

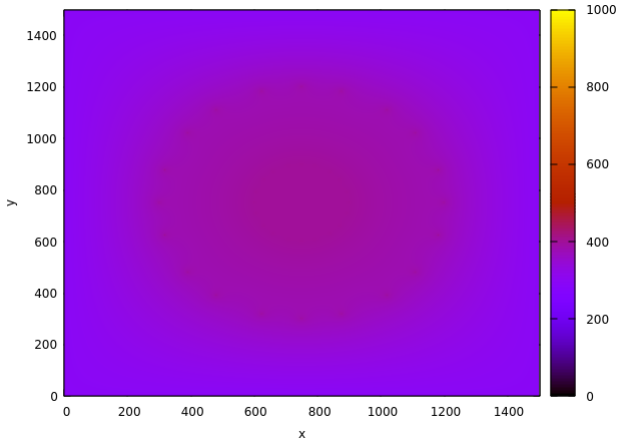

Figura 3: *Placa no instante t = 1s. Fonte: Autor*

Ao longo do desenvolvimento deste trabalho, analisaremos a performance do código utilizando algumas flags de otimização provindas do compilador e da Cpu. Após feita essa analise, iremos identificar os gargalos do código e paralelizar os mesmos via openMP e openMPI.

4 Implementação

Apresenta-se na fig.(4) o fluxograma com a lógica utilizada na criação do programa.

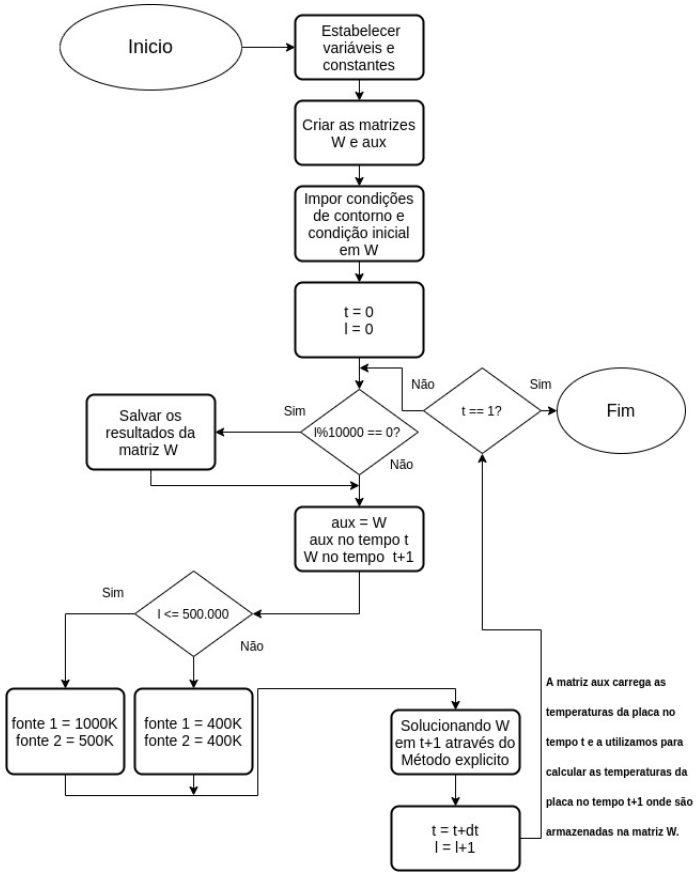

Figura 4: *Fluxograma do código. Fonte: Autor*

O código foi compilado pelo compilador gcc da gnu e executado na cpu Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz. Sendo 4 núcleos com a tecnologia Hyper-Threading totalizando um total

de 8 threads.

5 Benchmark serial

Neste tópico iremos realizar o benchmark do nosso programa em sua versão serial, aqui iremos aplicar algumas flags de otimização providas pelo compilador e pela cpu. Logo em seguida, vamos fazer um comparativo entre todas as flags para assim selecionar o conjunto delas que mais dão performance para o nosso programa.

As flags utilizadas estão citadas na lista abaixo, sendo as Ox são providas pelo compilador.

- O0
- O1
- O2
- O3
- fexpensive-optimizations
- m64
- foptimize-register-move
- funroll-loops
- ffast-math
- mavx
- mtune = native
- march = native

Informações sobre as flags citadas podem ser obtidas, em sua maioria, no seguinte endereço: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html .

Na fig.(5) encontram-se três gráficos na respectiva ordem, tempo versus flags, speedup versus flags e eficiência versus flags. Em cada gráfico temos duas linhas, sendo a linha azul referentes aos resultados obtidos utilizando somente as flags Ox do compilador e a linha laranja sendo os resultados obtidos utilizando as flags Ox do compilador em conjunto com as flags da cpu.

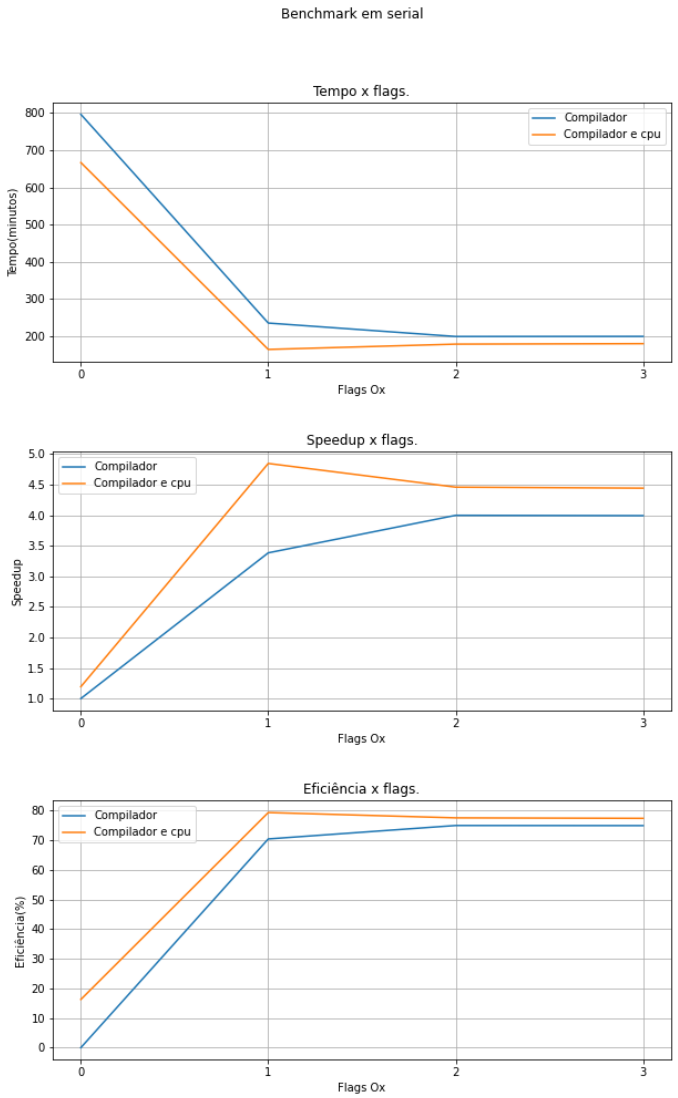

Figura 5: *Benchmark do código otimizado em serial. Fonte: Autor*

Não iremos nos aprofundar na analise deste benchmark no tópico presente, isto será tratado nos tópicos seguintes. Entretanto, somos capazes de observar que a flag O1 junto às flags da cpu foram as que melhor performaram nos garantindo um speedup de aproximadamente 5x e uma eficiência acima de 79%.

6 Profile

Através do profile somos capazes de identificar as regiões onde ocorrem gargalos em nosso programa. As regiões geralmente são funções em nossos programas e o gargalo é exatamente as funções que mais demandam tempo para terminar suas tarefas. Na fig.(6) encontra-se o profile do nosso programa.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 83.70  23445.68 23445.68  1000001    23.45    23.45  discretizar
 16.55  28080.86  4635.18  1000001     4.64     4.64  troca
  0.01  28082.81     1.96      101    19.36    19.36  salva_dados
  0.00  28083.08     0.27                             main
  0.00  28083.18     0.10        2    50.13    50.13  fonte
  0.00  28083.20     0.02  2067844     0.00     0.00  inicial
```

Figura 6: *Profile do código em serial. Fonte: Autor*

Observamos duas regiões onde ocorre gargalo, a primeira é a função discretizar tomando 83,7% do tempo de execução total do programa e a segunda é a função troca com 16,55%.

A função discretizar é onde todos o cálculos de atualização das temperaturas na placa são realizados. E a função troca é apenas uma função auxiliar que intercambia valores entre matrizes.

Através do profile fomos capazes de localizar as regiões do código que demandam mais tempo para serem executadas. Para conseguirmos uma maior performance do nosso código devemos paralelizar esses gargalos. A técnica de paralelização sera introduzida nos tópicos seguintes.

7 Técnicas de otimização de software

Para obtermos maiores performances em nossos códigos, são necessárias a utilização de técnicas de otimização.

Os computadores atuais trabalham utilizando números binários, isso significa que são capazes apenas de realizar operações de soma e multiplicação. Baseando-se nessa característica, a primeira otimização a ser feita é na escrita do código, deve-se sempre que possível substituir operações complexas por operações de adição e multiplicação.

Deve-se de modo geral ter boas técnicas de programação e visar sempre a otimização quando se desenvolve o programa. Devemos também, sempre que possível, utilizar as simetrias do problema.

A segunda técnica de otimização é a utilização de flags providas pelo compilador e cpu. As flags utilizadas nesse trabalho foram citadas no tópico 5.

A terceira forma para se ganhar performance é através da paralelização do código. Discutiremos nos próximos capítulos sobre a paralelização do programa.

8 Código base versus otimizado

No tópico atual analisaremos com maior nível de detalhes as performances obtidas com o programa em sua versão serial.

Na tabela (1) encontram-se os valores do tempo, speedup e eficiência para todas as flags Ox do compilador (-O0, -O1, -O2 e -O3) e para as flags do compilador em conjunto com as flags da cpu (-Ox -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mavx -mtune = native -march = native).

Tempo(minutos)		
n	Flags Ox	Flags Ox + cpu
O0	796,375	666,766
O1	235,238	164,268
O2	199,119	178,570
O3	199,379	179,819
Speedup		
n	Flags Ox	Flags Ox + cpu
O0	1,0	1,194
O1	3,385	4,848
O2	3,999	4,460
O3	3,994	4,444
Eficiência(%)		
n	Flags Ox	Flags Ox + cpu
O0	0,00	16,27
O1	70,46	79,37
O2	75,00	77,58
O3	74,96	77,42

Tabela 1: *Resultados do programa em serial utilizando o compilador gnu. Fonte: Autor*

Pode-se observar da tabela (1) que o código em serial (flag O0) demanda em torno de 796 minutos ($\approx$ 13 horas) para ser totalmente executado. O nosso primeiro objetivo antes de paralelizar é otimizar este programa o máximo possível, onde para isso utilizamos as flags citadas no tópico 5.

Nossa máxima performance a nível de compilação e hardware é dada utilizando as flags da cpu em conjunto com a flag O1 do compilador gnu. Tais otimizações fizeram com que o nosso código fosse executado em um total de 164 minutos ($\approx$ 2,7 horas) o que nos garante um speedup de 4,84x e uma eficiência de 79,37 %.

Os speedups e as eficiências foram calculados com base no tempo do programa em serial somente com a flag O0, que é o programa apenas com as otimizações feitas na escrita do código.

9 Implementação openMP

Neste tópico iremos utilizar do paralelismo junto às flags de otimização para obtermos uma melhor performance. Para isso utilizaremos a api openMP.

Conforme discutido no tópico 6 e observado na fig.(6), nosso código detém duas funções que demandam grande parte do tempo de execução. A forma de obtermos performance nesses gargalos é através da paralelização dessas funções, pois somente a otimização não é suficiente.

As funções discretizar e troca são o coração e a alma deste programa, pois através delas torna-se possível realizar a atualização das temperaturas na placa em cada instante de tempo. Tais funções devem acessar uma matriz de 1440 x 1440 e realizar operações em todos estes elementos e em todas as iterações. Tem-se portanto que por iteração são acessados e manipulados algo em torno de 4 milhões de elementos.

Utilizamos o openMP para paralelizar esses dois gargalos. Para isso, abrimos uma região paralela nessas duas funções e dentro delas utilizamos a diretiva "pragma for"para que as tarefas no interior do looping fossem divididas entre as diversas threads utilizadas. Utilizou-se um total de 1 à 8 threads

para que fosse possível realizar uma comparação de qual thread nos garantia um maior desempenho.

Nas figuras (7) e (8), respectivamente, são apresentadas as funções discretizar e troca com suas respectivas regiões paralelas.

```
void discretizar(double **w, double **aux , double lambda)
{

        int i,j;

        #pragma omp parallel shared(w,aux,lambda) private(i,j)
        {
                #pragma omp for schedule(dynamic)
                for(i = 1 ; i < X-1 ; i++)
                {
                        for(j = 1; j < Y-1 ; j++)
                        {

                                // Aqui verifica se o o conjunto (i,j) não se encontra sobre uma fonte, caso
 não seja uma fonte a conta é feita
                                if((i-X*0.5)*(i-X*0.5)+ (j-Y*0.5)*(j-Y*0.5) != 22500 && (i-X*0.5)*(i-X*0.5)+
(j-Y*0.5)*(j-Y*0.5) != 202500)
                                        w[i][j] = lambda*(aux[i+1][j]+aux[i-1][j]+aux[i][j+1]+aux[i]-
[j-1]-4*aux[i][j])+aux[i][j];

                        }
                }
        }

        return;
}
```

Figura 7: *Função discretizar com sua respectiva região paralela. Fonte: Autor*

```
//Funcao auxiliar
void troca(double **w , double **aux)
{
        int i,j;
        #pragma omp parallel shared(w,aux) private(i,j)
        {
                #pragma omp for schedule(dynamic)
                for(i=0; i< X ; i++)
                {
                        for(j=0; j < Y ; j++)
                        {
                                aux[i][j] = w[i][j];
                        }
                }
        }

        return;
}
```

Figura 8: *Função troca com sua respectiva região paralela. Fonte: Autor*

Discutiremos no tópico a seguir os resultados obtidos com a implementação do openMP.

10 Benchmark openMP - Comparando as Threads

Neste tópico discutiremos os resultados obtidos através da paralelização com a api openMP. Para este benchmark foram utilizadas duas configurações, onde na primeira fizemos uso apenas da flag O0 do compilador e na segunda da flag O1 em conjunto com as flags da cpu. Essa última configuração nos garantiu o maior desempenho no caso serial.

Na tabela (2) a seguir estão expostos todos os resultados obtidos e calculados.

O0		O1 + flags da cpu	
n	Tempo(min)	n	Tempo(min)
Thread 1	856,035	Thread 1	223,641
Thread 2	468,802	Thread 2	177,254
Thread 3	360,729	Thread 3	175,278
Thread 4	295,402	Thread 4	177,240
Thread 5	290,458	Thread 5	179,233
Thread 6	276,394	Thread 6	180,931
Thread 7	268,991	Thread 7	182,528
Thread 8	264,371	Thread 8	184,778
n	Speedup	n	Speedup
Thread 1	0,930	Thread 1	3,561
Thread 2	1,699	Thread 2	4,493
Thread 3	2,208	Thread 3	4,543
Thread 4	2,696	Thread 4	4,493
Thread 5	2,742	Thread 5	4,443
Thread 6	2,881	Thread 6	4,402
Thread 7	2,961	Thread 7	4,363
Thread 8	3,012	Thread 8	4,310
n	Eficiência(%)	n	Eficiência(%)
Thread 1	-7,49	Thread 1	71,92
Thread 2	41,13	Thread 2	77,74
Thread 3	54,70	Thread 3	77,99
Thread 4	62,91	Thread 4	77,74
Thread 5	63,53	Thread 5	77,49
Thread 6	65,29	Thread 6	77,28
Thread 7	66,22	Thread 7	77,08
Thread 8	66,80	Thread 8	76,80

Tabela 2: *Resultados obtidos após a implementação da api openMP, tomando o programa com a flag O0 e com o melhor conjunto de flags de maior performance do serial. Fonte: Autor*

Na fig.(9) estão presentes três gráficos com todas as informações contidas na tabela (2).

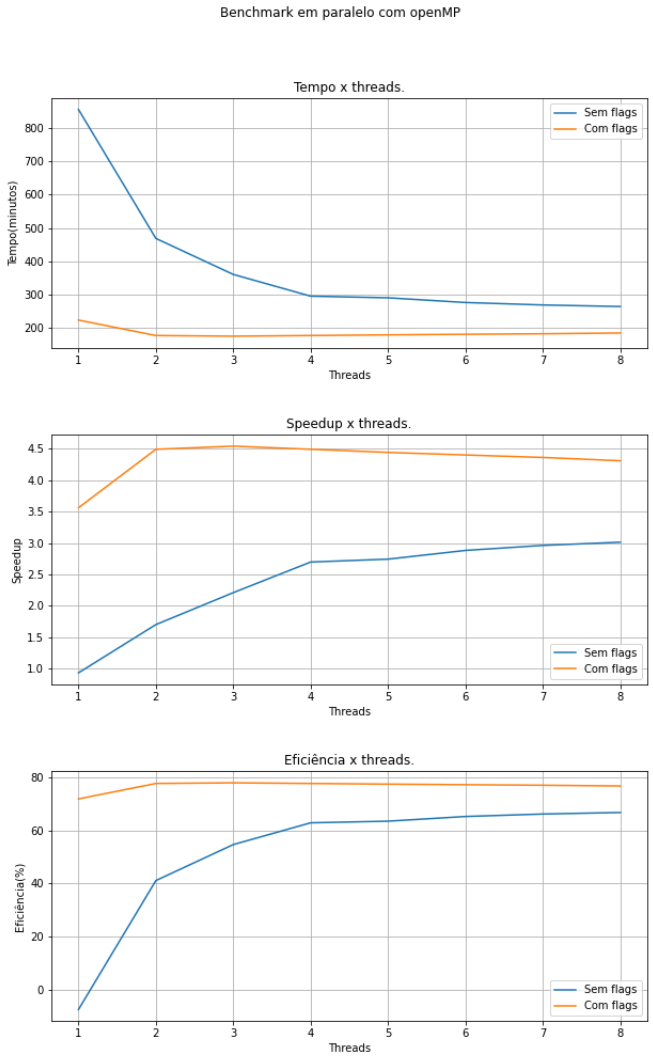

Figura 9: *Benchmark do código em paralelo com a implementação do openMP. Tem-se em azul as informações do código paralelo sem otimizações, em laranja tem-se o código em paralelo e munido do conjunto de flags de maior performance do caso serial. Fonte: Autor*

Os resultados de speedup e eficiência foram calculados com base no tempo de execução do programa em sua versão serial, sem ser feito uso de flags presentes na tabela (1), sendo equivalente à 796,375 minutos. Fazemos isso pois nosso objetivo é determinar qual método de paralelização e otimização nos garante uma maior performance. Logo, devemos então comparar todos os dados calculados com base nas informações de tempo do programa em sua versão serial.

Nossa primeira análise transcorrera sobre o programa em paralelo com openMP munido apenas da flag O0. Percebe-se que obtemos com apenas uma única thread um tempo de execução maior do que no caso serial, significando que obtivemos uma perca de desempenho devido aos processos internos da própria api, uma vez que os mesmos demandam recursos para serem executados.

Observamos com o aumento das threads a diminuição do tempo de execução e consequentemente o aumento do speedup e eficiência. Nessa configuração percebemos que a performance do nosso programa aumenta a cada nova thread adicionada, sendo a maior performance aquela atribuída à 8 threads. Portanto, um tempo de execução de 264,371 minutos sendo 3x mais rápido comparado ao serial e com uma eficiência de 66,80%.

Para a segunda configuração obtemos por sua vez comportamentos bem diferentes. Obtivemos para uma única thread um código 3,5x mais rápido quando comparado ao serial. A única diferença neste caso é a utilização de flags de otimização que, como visto, já nos garantiu maior desempenho quando comparado com 8 threads do paralelo openMP sem as flags de otimização. Isso nos indica que para uma maior performance se faz necessário utilizar a paralelização munido com as devidas otimizações.

Aqui obtivemos nossa maior performance para 3 threads. Observamos um aumento de speedup até

3 threads seguido da diminuição do mesmo a cada nova thread adicionada acima de 3. Nossa maior performance neste caso nos garantiu um tempo de execução de 175,254 minutos, sendo então 4,54x mais rápido quando comparado ao serial e apresentando uma eficiência de 77,99%.

Entretanto, conforme a tabela (1), o tempo de execução de maior performance serial fora de 164,268 minutos, sendo 4,85x mais rápido que o serial sem flags de otimização. Logo, torna-se mais intuitivo executar nosso programa em sua versão serial juntamente com a flag O1 e com as devidas flags da cpu, visto que a execução levará menor tempo para seu êxito.

Será discutido agora a escalabilidade do openMP junto ao nosso programa. Porém, faz-se necessário esclarecer inicialmente o modo com que estamos calculando os valores de speedup e eficiência encontrados nas tabelas (1) e (2). Para o speedup utilizamos da expressão

$$S(n) = \frac{T(serial)}{T(paralelo)}, \tag{8}$$

onde $T(serial)$ e $T(paralelo)$ representam, respectivamente, os tempos serial e paralelo. O speedup nos informa quantas vezes mais rápido nosso programa em paralelo executa quando comparado ao mesmo programa em serial.

Para a eficiência temos a equação

$$Eff = \left[1 - \frac{T(paralelo)}{T(serial)}\right] \cdot 100, \tag{9}$$

sendo $T(serial)$ e $T(paralelo)$ respectivamente os tempos serial e paralelo.

Na equação de eficiência (9) são considerados apenas os tempos de execução. Entretanto, gostaríamos de realizar uma analise mais técnica levando em consideração a quantidade de threads utilizadas. Para isso devemos calcular a eficiência de uma forma alternativa, sendo a sua expressão dada por

$$E(n) = \frac{S(n)}{n}, \tag{10}$$

onde $S(n)$ é o speedup de uma quantia n de threads e n é exatamente a quantidade de threads utilizada. No caso openMP os recursos se dão pela quantidade n.

A eficiência dada pela eq.(10) nos diz sobre a porção de tempo empregada de forma útil por cada thread. Faremos uso disso para analisarmos a escalabilidade do nosso programa no contexto da programação paralela via openMP. Na tabela (3) encontram-se o speedup por thread e na fig(9) tem-se o respectivo gráfico.

O0		O1 + flags da cpu	
n	S(n)/n	n	S(n)/n
Thread 1	0,9300	Thread 1	3,5610
Thread 2	0,8495	Thread 2	2,2465
Thread 3	0,7360	Thread 3	1,5143
Thread 4	0,6740	Thread 4	1,12325
Thread 5	0,5484	Thread 5	0,8886
Thread 6	0,4802	Thread 6	0,7337
Thread 7	0,4230	Thread 7	0,6233
Thread 8	0,3765	Thread 8	0,5388

Tabela 3: *Resultados obtidos após da implementação a api openMP, tomando o programa com a flag O0 e com o melhor conjunto de flags de maior performance do serial. Fonte: Autor*

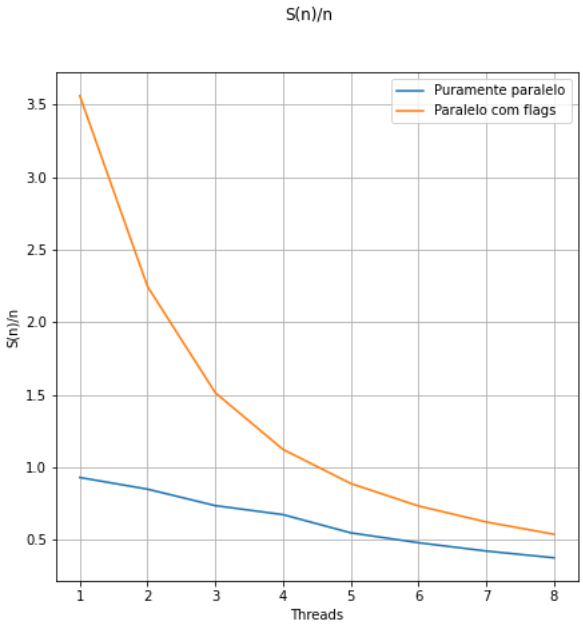

Figura 10: *Comparação do parâmetro S(n)/n para o openMP. A curva azul representa apenas o caso paralelo, já a laranja é o caso paralelo com o melhor conjunto de otimizações. Fonte: Autor*

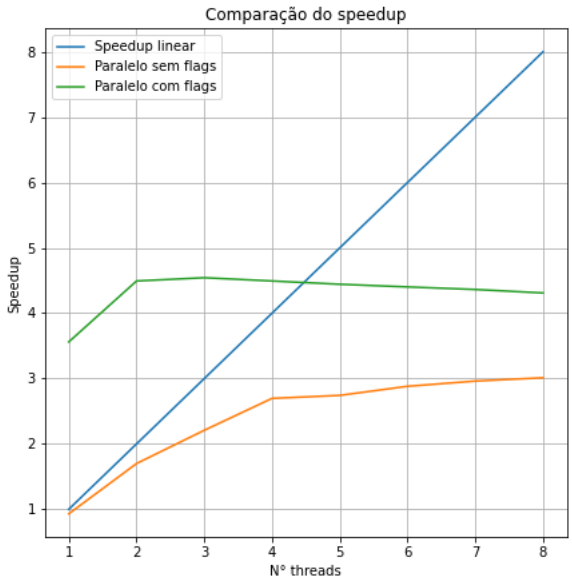

Figura 11: *Comparação com o speedup linear. Em azul temos o speedup linear teórico, em laranja temos o caso paralelo sem otimização por flags e em verde o paralelo com otimização.Fonte: Autor*

Observamos que a quantidade de tempo empregada de forma útil diminui com o acréscimo das threads. Isso significa que os recursos computacionais utilizados são de certa forma menos eficientes com o acréscimo de novas threads, sendo que a mesma não adiciona de forma significativa um decréscimo no tempo de execução e, portanto, nosso programa não é escalável. Deveria acontecer o oposto para que a escalabilidade fosse real.

Na fig(11) temos a comparação dos speedups com o speedup linear.

Observa-se que o caso paralelo sem as flags de otimização é sub-linear independentemente da quantidade de flags inseridas. Entretanto, temos um regime superlinear para o caso paralelo junto às flags de otimização. Tal superlinearidade ocorre até 4 threads, que é exatamente a região onde nosso parâmetro $S(n)/n$ se encontra acima de 1.

11 Implementação openMPI

Neste tópico iremos prosseguir de maneira análoga ao que fizemos para o openMP, exceto pelo fato de que agora utilizaremos a biblioteca openMPI. MPI é uma biblioteca que fornece recursos para a implementação paralela em ambiente de memória distribuída, onde seu significado se dá por message passing interface.

A estratégia aqui utilizada se dá por um processo master onde o mesmo terá a matriz representando a placa e uma matriz auxiliar para que seja possível atualizar as temperaturas. É neste processo onde as matrizes serão criadas, repartidas, enviadas, recebidas e sincronizadas.

Utilizamos a diretiva scatter para enviar tiras horizontais de nossa matriz principal para cada processo presente, onde tais tiras são calculadas com base nos processos presentes. A matriz auxiliar foi enviada com a diretiva broadcast, uma vez que a mesma não precisa de atualizações. Utilizamos a diretiva gather para a sincronização da matriz principal com as temperaturas atualizadas.

Para essa implementação utilizamos dois casos, primeiro apenas com o openMPI, sem as flags de otimização, e o segundo o openMPI com o melhor conjunto de flags de otimização do caso serial.

Discutiremos no próximo tópico os resultados obtidos. A implementação openMPI pode ser encontrada em anexo com este trabalho.

12 Benchmark openMPI - Comparação de processos

Neste tópico iremos discutir os resultados obtidos. Os mesmos podem ser encontrar na tabela (4) a seguir.

O0		O1 + flags da cpu	
n	Tempo(min)	n	Tempo(min)
Processo 1	780,610	Processo 1	349,327
Processo 2	577,329	Processo 2	319,355
Processo 3	523,299	Processo 3	328,110
Processo 4	522,019	Processo 4	345,484
Processo 5	701,816	Processo 5	459,879
Processo 6	723,304	Processo 6	481,348
Processo 8	762,814	Processo 8	508,230
n	Speedup	n	Speedup
Processo 1	1,020	Processo 1	2,280
Processo 2	1,379	Processo 2	2,494
Processo 3	1,522	Processo 3	2,427
Processo 4	1,526	Processo 4	2,305
Processo 5	1,135	Processo 5	1,732
Processo 6	1,101	Processo 6	1,654
Processo 8	1,044	Processo 8	1,567
n	S(n)/n	n	S(n)/n
Processo 1	1,02	Processo 1	2,28
Processo 2	0,69	Processo 2	1,25
Processo 3	0,51	Processo 3	0,81
Processo 4	0,38	Processo 4	0,58
Processo 5	0,23	Processo 5	0,35
Processo 6	0,18	Processo 6	0,28
Processo 8	0,13	Processo 8	0,20
n	Eficiência(%)	n	Eficiência(%)
Processo 1	1,98	Processo 1	56,14
Processo 2	27,51	Processo 2	59,90
Processo 3	34,29	Processo 3	58,80
Processo 4	34,45	Processo 4	56,62
Processo 5	11,87	Processo 5	42,25
Processo 6	9,18	Processo 6	39,56
Processo 8	4,21	Processo 8	36,18

Tabela 4: *Resultados obtidos após a implementação do mpi, tomando o programa com a flag O0 e com o melhor conjunto de flags de maior performance do serial. Fonte: Autor*

Na figura (12) temos os gráficos, respectivamente, de tempo, speedup, eficiência e parâmetro S/n.

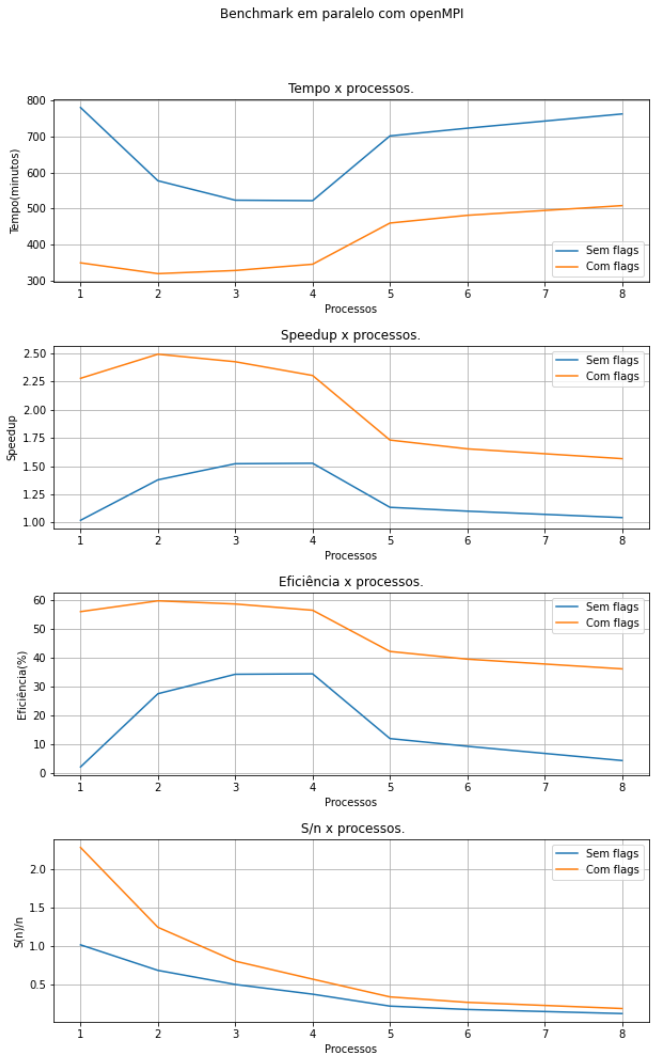

Figura 12: *Benchmark do código em paralelo com a implementação em mpi. Tem-se em azul as informações do código paralelo sem otimizações, em laranja tem-se o código em paralelo e munido do conjunto de flags de maior performance do caso serial. Fonte: Autor*

Todos os dados foram calculados com base no tempo de execução serial, sendo este o mais natural para o nosso propósito, que é analisar as performances obtidas por diversas técnicas de otimização e paralelização.

Inicialmente analisaremos o caso puramente paralelo, onde nossa melhor performance foi com 4 processos contabilizando um tempo de execução de 522,019 minutos, sendo, portanto, 1,52x mais rápido que o tempo de execução serial e nos garantindo uma eficiência de 34,45 %. Entretanto, em termos de eficiência por processo, conforme a eq.(10) não é tão vantajoso uma vez que tal quantidade de processos não acelera de forma significativa o tempo de execução do nosso programa quando comparado a uma quantidade menor de processos.

Para o caso paralelo e otimizado temos que a melhor performance se apresenta para dois processos, sendo o tempo de execução de 319,355 minutos, e portanto, 2,5x mais rápido que o programa em serial, nos garantindo assim uma eficiência de 59,90%.

De forma geral, o caso otimizado se mostra superior ao puramente paralelo, uma vez que o otimizado tem todos seus parâmetros superiores utilizando metade dos recursos. Entretanto, a otimização com mpi não se mostrou eficiente para a solução deste problema, visto que a implementação teve sua performance abaixo da implementação serial e da implementação com openMP.

A possível explicação para uma baixa performance é devido ao fato que com o aumento de processos se aumenta também a quantidade de mensagens a serem recebidas,enviadas e também o tempo de sincronização. Todos esses processos demandam uma quantidade de tempo que no total é executado por 1 milhão de iterações, levando portanto à uma menor performance.

Observa-se também que o aumento de processos faz com que o speedup diminua, ocasionando assim na diminuição da eficiência por processo, o que nos leva então a concluir que nosso programa não é escalável uma vez que a performance piora a cada novo processo adicionado.

Na figura (13) encontra-se a comparação dos speedups com o speedup linear.

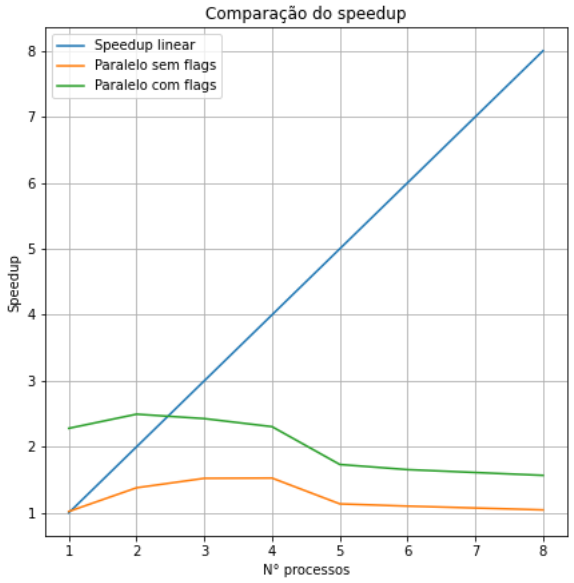

Figura 13: *Comparação com o speedup linear. Em azul temos o speedup linear teórico, em laranja temos o caso paralelo sem otimização por flags e em verde o paralelo com otimização.Fonte: Autor*

Percebe-se que novamente a implementação puramente paralela é um caso sub-linear enquanto o paralelo otimizado tem seu regime superlinear até 2 processos. Ambos tendem a valores menores de speedup com o aumento dos processos, o que reforça sobre o programa não ser escalável com a implementação mpi. Entretanto, sabe-se que a implementação paralela depende muito mais das estratégias utilizadas do que de fato da própria biblioteca. Logo, utilizar outra estratégia de paralelismo com mpi pode nos fornecer resultados extremamente diferentes.

13 Validação do resultado

Até o momento vimos diversas formas de se obter o desempenho de um programa, seja ela paralelizado ou apenas otimizado. Entretanto, tais modificações podem ser danosas e alterar os dados previstos, de modo que devemos então realizar a validação de nossos resultados.

O código nos retorna 100 arquivos de saída, os quais foram nomeados como 0.dat,1.dat,...,99.dat. Para validar nossos resultados, calculou-se o erro quadrático médio de cada um dos 100 arquivos e por fim obtivemos 100 pontos de erro.

Os erros são calculados referente ao código em serial e o código de maior performance. Para esses cálculos foi feito um programa em python. Na figura (14) encontra-se o erro quadrático médio por arquivo do código que nos deu a maior performance.

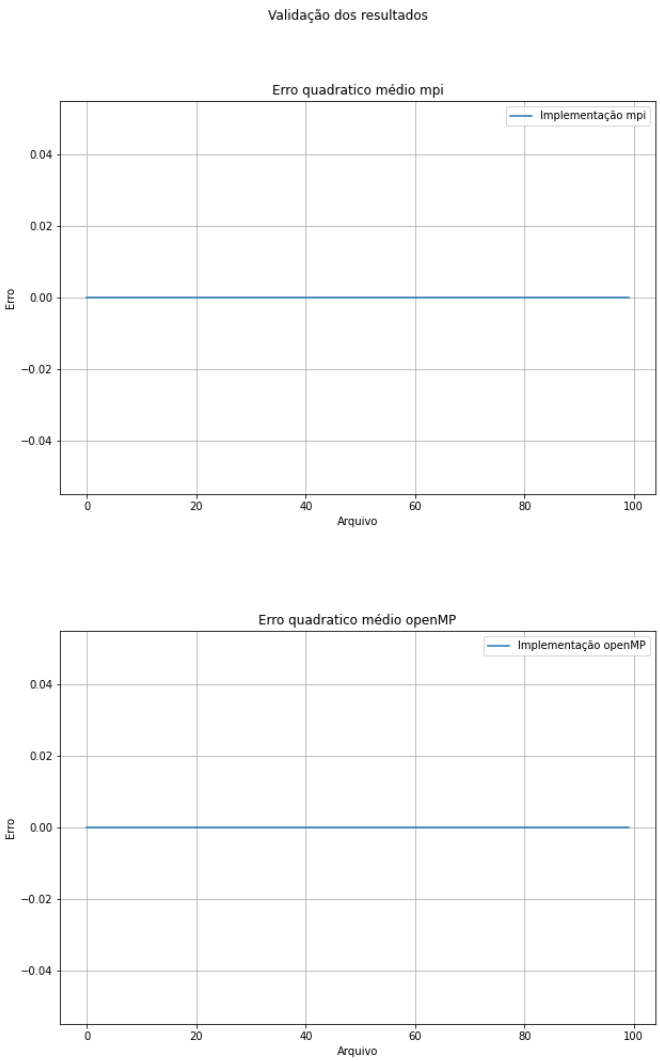

Figura 14: *Erro quadrático médio das implementações em paralelo. Fonte: Autor*

Obteve-se um erro quadrático médio nulo para todos os arquivos. Portanto, conclui-se que o código otimizado e paralelizado não modifica os resultados finais.

Não fizemos a validação para o serial caso houvesse algum desvio devido à introdução das flags de otimização, estes teriam sido apresentados também na validação em paralelo.

14 Conclusão

Nesse trabalho vimos como solucionar a equação de difusão de calor via métodos numéricos e vimos também técnicas de otimização para uma maior performance do código. Podemos concluir do presente trabalho que um programa devidamente otimizado nos garantiu 79,37% de eficiência em código serial. Através do profile fomos capazes de analisar os gargalos do nosso código e assim utilizar as devidas técnicas de paralelização nas regiões onde ocorrem os gargalos. Entretanto, tais implementações não surtiram o efeito desejado, onde nossa maior eficiência como a ai openMP fora de 77,99% e com a implementação do mpi nossa maior eficiência fora de 59,90%. O código mais otimizado tem seu tempo de execução em 164,268 minutos e um speedup de 4,84x mais rápido que o programa em serial sem otimização.

Conclui-se também há a necessidade de um código muito bem otimizado para surtir maiores efeitos da paralelização. Nota-se por fim que nem sempre a adição de mais recursos, como processos e threads, fará com que o programa ganhe performance. Observamos exatamente esse comportamento não escalável e, portanto, o acréscimo de recursos se torna prejudicial para a performance do mesmo.

Referências

[1] BURDEN, R. L.; FAIRES, J. D. Numerical Analysis. 9: Boston, USA, 2011..