Thales Oliveira Arakawa

Aplicação do MPI ao Método das Diferenças Finitas

Universidade Federal Fluminense - UFF

Campus Volta Redonda

Instituto de Ciências Exatas

Brasil

2 de fevereiro de 2022

Resumo

Este trabalho vem propor uma análise sobre processamento paralelo através das da API MPI (*Message Passing Interface*). Utilizando-se de um algoritmo de resolução de Equações Diferenciais Parciais, através do método da diferença finita, poderemos observar como é impactado um processamento paralelo, via MPI, quando os dados que são distribuídos possuem dependência entre si. Para isso, o programa em sua versão serial deverá estar o mais otimizado possível, sempre considerando a aritmética realizada por um processador e também as computações desnecessárias. Dito isso veremos que o resultado obtido via MPI é o mesmo que o do OpenMP, sendo que cada um possuí sua eficiência para alcançar seu objetivo, que é a resolução da equação diferencial parcial de Laplace.

Palavras-chaves: Método das Diferenças Finitas, EDP, HPC, MPI, paralelismo.

Sumário

1 Introdução

Muita das vezes, nas ciências exatas, não é possível obter soluções analíticas para os problemas. Diante deste fato, muitos matemáticos buscam compreender e desenvolver ferramentas para obter um resultado aproximado.

Um exemplo é o método das diferenças finitas, que faz o uso das séries de Taylor; que é talvez a ferramenta de aproximação matemática mais comumente utilizada. Este método busca obter uma solução, não exata, para as equações diferenciais parciais. Porém a solução para uma EDP abrange os infinitos pontos em que esta é valida e da forma que o método das diferenças finitas é proposto deveríamos calcular o maior número de pontos possíveis, de forma que através de lápis e papel seja um processo árduo.

É para esses métodos recursivos e massivos que o computador se torna uma solução. Podendo realizar milhões de cálculos por segundo este é capaz de resolver este trabalho árduo num curto espaço de tempo.

Com a evolução dos processadores, que regem essas máquinas, chegamos ao ponto em que se desenvolve a computação paralela. Procedimento este em que existem diversos componentes realizando cálculos simultaneamente.

Para implementar a computação paralela foram desenvolvidas API's (Application Programming Interface), como o OpenMP e o MPI, que são capazes de serem implementados aos códigos, ditos seriais, e possibilitar a paralelização dos procedimentos matemáticos realizados pelo mesmo.

2 Objetivos

Este trabalho vem com o intuito de analisar a escalabilidade do método das diferenças finitas, para resolução de EDP's, através da utilização da API MPI e comparar suar performance com uma versão semelhante em OpenMP.

3 Fundamentos Teóricos

3.1 Método das Diferenças Finitas

3.1.1 A Diferença Finita

A Diferença Finita nada mais é do que uma aproximação, para as derivadas, utilizando-se da série em expansão de Taylor, ou seja:

$$f(x_i + h_x) = \sum_{n}^{\infty} f^n(x_i)\frac{h_x^n}{n!} = f(x_i) + h_x f'(x_i) + h_x^2\frac{f''(x_i)}{2} + ... \tag{3.1}$$

onde $h_x = x - x_i$. A Equação 3.1 é dita progressiva, temos também a sua forma regressiva, dada por:

$$f(x_i - h_x) = \sum_{n}^{\infty} f^n(x_i)\frac{(-h_x)^n}{n!} = f(x_i) - h_x f'(x_i) + h_x^2\frac{f''(x_i)}{2} + ... \tag{3.2}$$

Da diferença entre as Equações 3.1 e 3.2, obtemos a foma centrada (Eq. 3.3) que tem como característica uma convergência mais abrupta, uma vez que os termos h_x de ordem par se cancelam.

$$f(x_i + h_x) - f(x_i - h_x) = 2h_x f'(x_i) + 2h_x^3\frac{f'''(x_i)}{6} + ... \tag{3.3}$$

Determinando $h_x < 1$, de forma que $h_x^3 \ll 1$, teremos então uma aproximação de $f'(x_i)$ dada por:

$$f(x_i + h_x) - f(x_i - h_x) \approx 2h_x f'(x_i) \Rightarrow f'(x_i) \approx \frac{f(x_i + h_x) - f(x_i - h_x)}{2h_x}. \tag{3.4}$$

Definiremos então, a diferença finita centrada de ordem 1 (UFRGS, 2020), como:

$$D_{i,h_x} f(x_i) := \frac{f(x_i + h_x) - f(x_i - h_x)}{2h_x} \tag{3.5}$$

Para a derivada de segunda ordem, façamos a soma entre as Equações 3.1 e 3.2, obtendo-se assim:

$$f(x_i + h_x) + f(x_i - h_x) = 2f(x_i) + h_x^2 f''(x_i) + h_x^4\frac{f''''(x_i)}{12} + ..., \tag{3.6}$$

uma vez que $h_x < 1$ e, portanto, $h_x^4 \ll 1$, eliminaremos os termos de quarta ordem e superiores, obtendo a aproximação:

$$f''(x_i) \approx \frac{f(x_i + h_x) - 2f(x_i) + f(x_i - h_x)}{h_x^2} \tag{3.7}$$

que é por onde definiremos a diferença finita de ordem 2 (UFRGS, 2020), como:

$$D^2_{i,h_x} f(x_i) := \frac{f(x_i + h_x) - 2f(x_i) + f(x_i - h_x)}{h_x^2} \tag{3.8}$$

3.1.2 O Método

O método das diferenças finitas é utilizado para resolução de EDP's, principalmente, quando esta não possui solução analítica. O objetivo é obter uma solução aproximada através da implementação das diferenças finitas. Tomemos como exemplo o operador laplaciano para duas dimensões, onde

$$\nabla^2 f = \frac{\partial f}{\partial x}^2 + \frac{\partial f}{\partial y}^2 . \tag{3.9}$$

Substituindo as derivadas parciais pela diferença finita, dada pela equação 3.8, teremos então:

$$\begin{aligned} \nabla^2 f \approx & \frac{f(x_i + h_x, y_j) - 2f(x_i, y_j) + f(x_i - h_x, y_j)}{h_x^2} \\ & + \frac{f(x_i, y_j + h_y) - 2f(x_i, y_j) + f(x_i, y_j - h_y)}{h_y^2}. \end{aligned} \tag{3.10}$$

Para a equação de Laplace, teríamos então:

$$\begin{aligned} & \frac{f(x_i + h_x, y_j) - 2f(x_i, y_j) + f(x_i - h_x, y_j)}{h_x^2} \\ & + \frac{f(x_i, y_j + h_y) - 2f(x_i, y_j) + f(x_i, y_j - h_y)}{h_y^2} \approx 0 \end{aligned} \tag{3.11}$$

Uma vez que buscamos a solução de uma EDP, nosso objetivo então é determinar $f(x, y)$, logo o isolamos e obtemos:

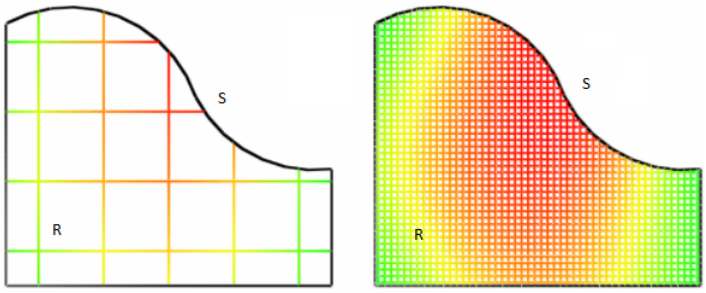
$$f(x_i, y_j) \approx \frac{h_y^2 \left[f(x_i + h_x, y_j) + f(x_i - h_x, y_j)\right] + h_x^2 \left[f(x_i, y_j + h_y) + f(x_i, y_j - h_y)\right]}{2(h_x^2 + h_y^2)}. \tag{3.12}$$

Como para toda EDP, esta possuí região($\mathbf{R}$) em que a solução é válida e também as condições de contorno($\mathbf{S}$). Uma vez que desconhecemos o comportamento da $f(x, y)$,

poderíamos atribuir quaisquer valores a ela e para cada ponto utilizaríamos a equação 3.12 para atualizar o valor relativo a este ponto. Será necessário então realizar uma varredura em todos os pontos da região em que a EDP se encontra até que, por convergência, a solução seja obtida.

O processo de determinar quais pontos serão computados, é chamado de discretização. É construído uma malha sobre região de tal forma que seu espaçamento horizontal seja de h_x e vertical de h_y

Figura 1 – Discretização de uma Região.



Fonte: Alto Qi.

3.1.3 OpenMP

O OpenMP é uma API, criada em 1997 para Fortran, que possibilita a paralelização de um código serial. Através de diretivas compostas por construtores e cláusulas adicionadas ao código e um compilador que possua suas bibliotecas o programa poderá ter sua atividade dividida entre os processadores virtuais (*threads*) disponíveis para computação (OPENMP, 2012a).

Atualmente o OpenMP possui suporte à Fortran, C/C++, Java, Python e outras linguagens mais (OPENMP, 2012b).

Figura 2 – Exemplo de criação de região paralela.

```
#pragma omp parallel
{
        do{
                erro=opDif(map,data,erro);
        }while(tol<erro);
}
```

Fonte: Autor.

A partir do momento que uma diretiva "#pragma omp parallel"é inserida no programa, é criada uma região paralela. Nesta região poderemos utilizar algumas funções

da biblioteca do OpenMP de forma a obter a quantidade de *threads* disponíveis e o número de identificação de cada uma. Fora dessa região o programa irá se comportar de forma serial. Além dessas funções básicas o OpenMP fornece diretivas mais inteligentes baseadas em algoritmos canônicos, como a varredura de vetores e matrizes através de *loops*.

3.1.4 MPI

O MPI é uma API, criada em 1993. Assim como o OpenMP ele é capaz de paralelizar um código, porém ao contrário do OpenMP, o MPI é capaz de estender este paralelismo para além de um único *socket*, chamado de nó. Uma outra diferença importante é o modelo topológico ao qual segue essas API's. Sendo uma de memória compartilhada (OpenMP), onde os *core's* tem acesso ao mesmo endereço de memória. a outra é de memória distribuída (MPI), onde os *core's* possuem um espaço reservado de memória para cada de forma que, quando necessário, uma comunicação deve ser atribuída entre estes para que um código seja processado.(FILHO, 2002)

Figura 3 – Exemplo programa com diretivas MPI.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if(rank==0){
        strcpy(message, "Hello, world");
        for(i=1; i<size; i++){
                MPI_Send(message,13,MPI_CHAR, i, type, MPI_COMM_WORLD);
        }
}
else{
        MPI_Recv(message, 13, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
}
printf("Message from node =%d :%.13s \n", rank, message);
MPI_Finalize();
```

Fonte: Autor.

O MPI distribuí para todos os trabalhadores/*core's* disponíveis o código a ser processado. Portanto, se não houver condições ligadas a identificação de cada trabalhador, todos irão realizar os mesmos procedimentos.

4 Metodologia

4.1 A Equação Diferencial Parcial

A EDP utilizada para implementação do método de diferenças finitas, será a equação de Laplace (Eq. 3.9). Esta é uma equação diferencial que pode ser encontrada em todas as áreas da física, uma vez que esta procura determinar o fluxo de um campo de forças conservativo, ou seja um campo tal que possua função potencial. O trabalho abordará o contexto da eletrostática, onde:

$$\nabla^2\Phi = \nabla \cdot \vec{\nabla}\Phi = -\nabla \cdot \vec{E} = 0, \tag{4.1}$$

onde Φ é o potencial eletrostático e a Equação 4.2 é a equação de Laplace para este potencial. Vale lembrar que, das equações de Maxwell, que:

$$\nabla^2\Phi = -\nabla \cdot \vec{E} = -\frac{\rho}{\epsilon_0}, \tag{4.2}$$

portanto no problema a ser desenvolvido estará ausente de cargas. Portanto para a implementação do método das diferenças finitas, a equação utilizada (Eq. 3.12) será aquela abordada na sessão 3.1.1

4.2 Discretização

Sendo a região a ser trabalhada bidimensional então uma estrutura matricial seria de grande conveniência. Porém, ao invés de se utilizar da sua forma convencional, foi criada uma variável do tipo *struct*, através da linguagem C, onde essa estrutura é apresentada na Figura 4.

Figura 4 – Monômero da estrutura matricial.

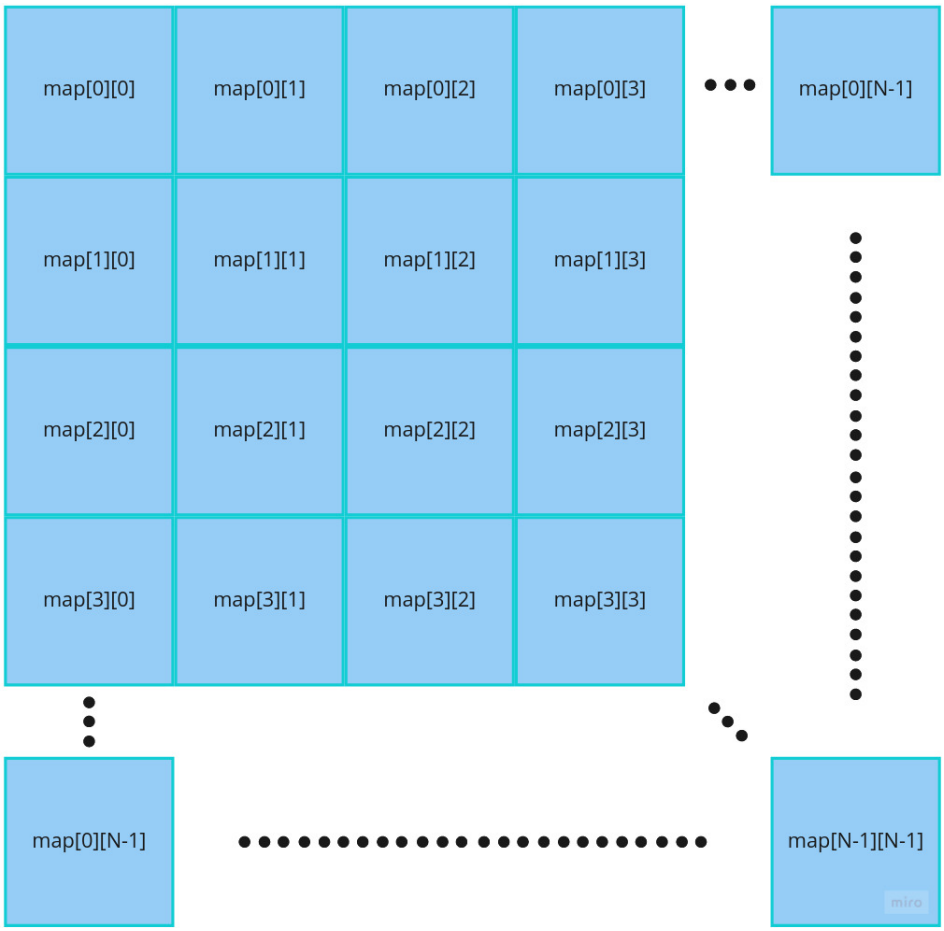
```
struct quad
{
        int cc;
        double a;
        double d;
};
```

Fonte: Autor.

Essa variável nada mais é que um monômero para uma estrutura maior, que se dá como uma matriz convencional em C, como na Figura 5. Porém, cada célula dessa matriz

irá carregar o status de ser um ponto da região de contorno (cc), o valor que este ponto possui atualmente (a) e o valor que possuirá após a varredura (d).

Figura 5 – Matriz de *struct*



Fonte: Autor.

A vantagem que essa abordagem fornece é a versatilidade de cada célula carregar os valores e condições que o método exige. Além de se comportar como uma matriz canônica em C, possibilitando uma compatibilidade com o MPI.

4.3 Método da Relaxação

Para solucionar a EDP, será utilizado o método da relaxação. Uma vez que já possuímos a relação matemática que cada ponto da malha deve obedecer, dado pela equação 3.12, bastará realizar varreduras sobre a mesma e ir atualizando os valores pontuais, sendo esperado a convergência, quando a variação dos valores atualizados forem suficientemente pequenos a solução numérica será entregue.

4.4 Paralelização - MPI

O MPI é uma API que possui diretivas que possibilitam em distribuir tarefas para cada *core*, sendo estes pertencentes a um mesmo *socket* ou não.

Pelo propósito do método utilizado será realizado uma divisão para cada *core* de forma que cada um deste receberá uma parcela da matriz principal.

Essa divisão se dá pela diretiva *Scatterv*, que possibilita uma distribuição não igualitária entre os mesmos.

Figura 6 – Separado a Matriz Principal.

```
//Trabalho único para o mestre
if(my_rank == 0){
        struct quad map[N][N];
        //Importa Condição de Contorno de um arquivo em ASCII
        importCC(map);
        //Vetor com o número de elementos que cada processo possuirá
        int counts[size];
        //Vetor com os deslocamentos sobre o buffer a ser aplicado o Scatterv
        int displacements[size];
        for(i=0;i<size;i++){
                counts[i]=(i < N%size) ? N*((int)(N/size) + 1) : N*(int)(N/size);
                displacements[i]=(i>(N%size)) ? N*(i*(int)(N/size) + N%size) : N*i*((int)(N/size) + 1) ;
                printf("\n i %d counts %d displa %d",i,counts[i],displacements[i]);
        }
        //Separando matriz por tamanhos diferentes
        MPI_Scatterv(map, counts, displacements, quad_type, own_map, trabalho, quad_type, 0, MPI_COMM_WORLD);
}
else
{
        MPI_Scatterv(NULL, NULL, NULL, quad_type, own_map, trabalho, quad_type, 0, MPI_COMM_WORLD);
}
```

Fonte: Autor.

4.5 Determinando Condição de Contorno

Para determinar a condição de contorno para a equação de Laplace, para a eletrostática, utilizou-se de um recurso que converte uma imagem em ASCII (protocolo de codificação). O recurso se encontra disponível na internet (<https://www.dcode.fr/binary-image>), basta realizar o *upload* da imagem e escolher a escala relativa, desejada. Neste trabalho foi utilizado a logo da UFF (Universidade Federal Fluminense), que na modelagem se encontrará em potencial constante de $100V$ e enclausurada a uma caixa aterrada.

4.6 Computadores

Para testar os programas e observar a escalabilidade quanto ao paralelismo, utilizamos:

1) Nós do supercomputador Santos Dumont (SDumont) do Laboratório Nacional de Computação Científica(LNCC,).

    a) B710

        * 2 x CPU Intel Xeon E5-2695v2 Ivy Bridge, 2,4Ghz

Figura 7 – Condição de contorno fora de escala.

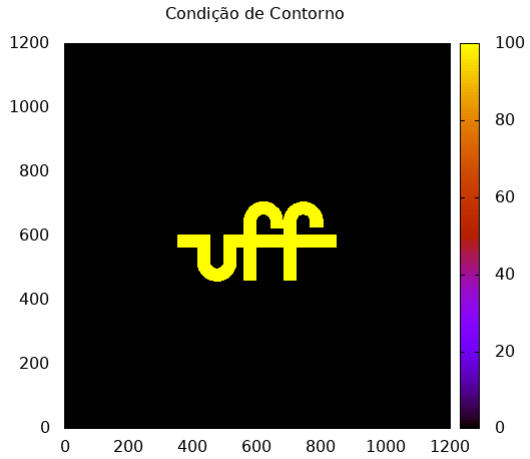


Figura 8 – Fonte: Autor.

* 24 núcleos (12 por CPU), totalizando de 12.096 núcleos
* 64GB DDR3 RAM

b) Sequana

* 2x Intel Xeon Cascade Lake Gold 6252
* 48 núcleos (24 por CPU)
* 384Gb de memória RAM

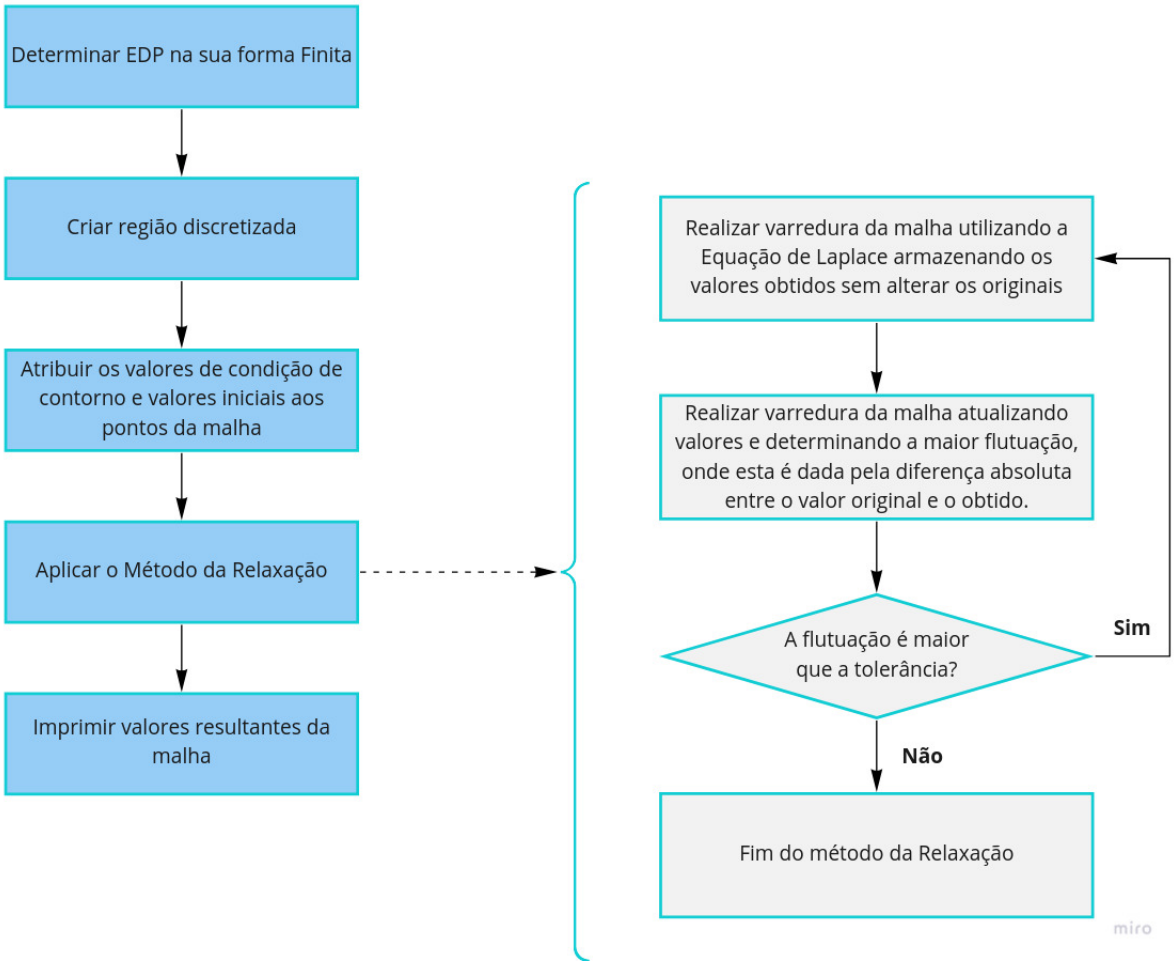
2) Computador do laboratório 107 da Universidade Federal Fluminense (UFF).

- Intel I7-200 Little Endian, 3.4Ghz
- 4 núcleos double-thread
- 4Gb de memória RAM

5 Fluxograma

A Figura 9 apresenta como se dará a implementação do programa.

Figura 9 – Fluxograma do método de diferenças finitas, com o uso da relaxação.



Fonte: Autor.

6 Análise do Programa Serial

6.1 Determinando as melhores *flags*

Foram testadas, apenas, as *flags* de otimização padrão do compilador *GNU Compiler Collection* (GCC).

Benchmark Serial	
Flag	Tempo (s)
O0	992,947
O1	261.831
O2	258,297
O3	262,498

Tabela 1

6.2 Perfil do Programa Serial

Como esperado, devido sua construção, as funções responsáveis pela varredura por trás do método da relaxação, chamada de *opDif* e *att*, é a que mais representa para o tempo total de execução, 99.9%.

Figura 10 – Perfil gerado pelo programa GNU profiler.

```
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00   95.72                 main [1]
               52.12    0.00   45416/45416       att [2]
               43.60    0.00   45416/45416       opDif [3]
                0.00    0.00       1/1           importCC [4]
                0.00    0.00       1/1           printMap [5]
-----------------------------------------------
               52.12    0.00   45416/45416       main [1]
[2]     54.4   52.12    0.00   45416         att [2]
-----------------------------------------------
               43.60    0.00   45416/45416       main [1]
[3]     45.6   43.60    0.00   45416         opDif [3]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[4]      0.0    0.00    0.00       1         importCC [4]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[5]      0.0    0.00    0.00       1         printMap [5]
-----------------------------------------------
```

Fonte: Autor.

6.3 Otimização do Programa Serial

Como podemos ver o perfil do programa através da Figura 10, $99,9\%$ do programa se concentrava em duas funções. Sendo necessária a varredura completa da matriz e a atualização da mesma.

Quanto aos cuidados sobre a otimização das operações matemática, estas já haviam sido implementadas, uma vez que é relativamente comum problemas dado pela divisão e multiplicação de uma variável de ponto flutuante por uma inteira. Portanto todas essas operações já estavam dadas por decimais.

No mais a alteração feita, foi a remoção de funções que auxiliavam a verificar o funcionamento do programa e os comentários com o mesmo fim.

7 Implementação do MPI

Uma vez que o Método das Diferenças Finitas possuí uma dependência dos dados com seus vizinhos, é necessário a implementação da comunicação dos trabalhadores. E como a intenção é paralelizar esse método, é exigido também a divisão da matriz para todos os trabalhadores/*threads* disponíveis.

Toda a comunicação feita através das diretivas do MPI é necessário declarar o tipo de variável que será transitada. A variável utilizada é do tipo *struct*. Para que o MPI reconheça esse tipo de variável é necessário parametrizá-lo como podemos ver na Figura 11.

Figura 11 – Matriz de *struct*

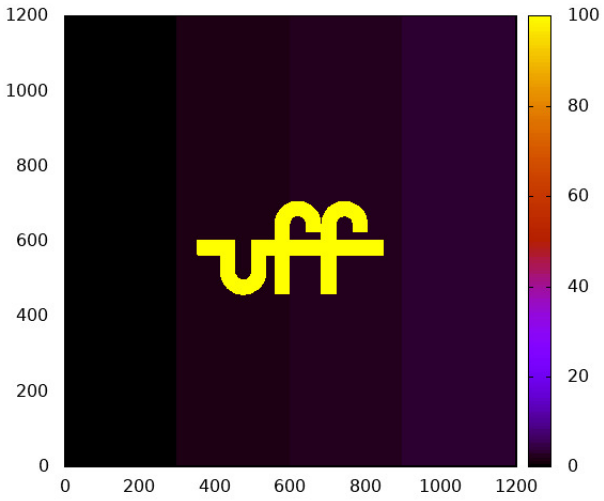
```
MPI_Datatype quad_type;
int lengths[3] = { 1, 1, 1 };
MPI_Aint displacements[3];
struct quad dummy_quad;
MPI_Aint base_address;
MPI_Get_address(&dummy_quad, &base_address);
MPI_Get_address(&dummy_quad.cc, &displacements[0]);
MPI_Get_address(&dummy_quad.a, &displacements[1]);
MPI_Get_address(&dummy_quad.d, &displacements[2]);
displacements[0] = MPI_Aint_diff(displacements[0], base_address);
displacements[1] = MPI_Aint_diff(displacements[1], base_address);
displacements[2] = MPI_Aint_diff(displacements[2], base_address);

MPI_Datatype types[3] = { MPI_INT, MPI_DOUBLE, MPI_DOUBLE };
MPI_Type_create_struct(3, lengths, displacements, types, &quad_type);
MPI_Type_commit(&quad_type);
```

Fonte: Autor.

Onde o vetor *lenghts* contém o tamanho de x variáveis para cada item do *struct* e o vetor *displacements* terá a posição de memória relativa para cada item do *struct*. Para isso é utilizado a função *Get address* que obtém o endereço de memória e posteriormente a função *Aint diff* nos dá o deslocamento de memória de um item em relação ao endereço base. Cria-se o vetor que contém os tipos de dados padrão para o MPI. Por final, com todos os parâmetros definidos, cria-se um tipo de dado para o MPI que se comporta como o *struct*, podendo-se assim utilizar as outras funções de comunicação do MPI. Para a divisão e união das informações distribuídas entre os trabalhadores, foi utilizado as diretivas *Scatterv* e *Gatherv*, que possibilita uma divisão não igualitária entre os mesmos. Esta versão do *Scatter* e *Gather* foi escolhida para que o programa possa ser escalado por qualquer número de *core's*. Podemos ver na Figura 12 a divisão sendo feita entre quatro trabalhadores.

Para a comunicação entre os trabalhadores, utilizou-se as diretivas *Send* e *Recv*, como na Figura 13. Essas funções de comunicação eram convocadas em toda a varredura que cada *core* fazia.

Figura 12 – Separação da Malha em *threads*.



Fonte: Autor.

Figura 13 – Separação da Malha em *threads*.

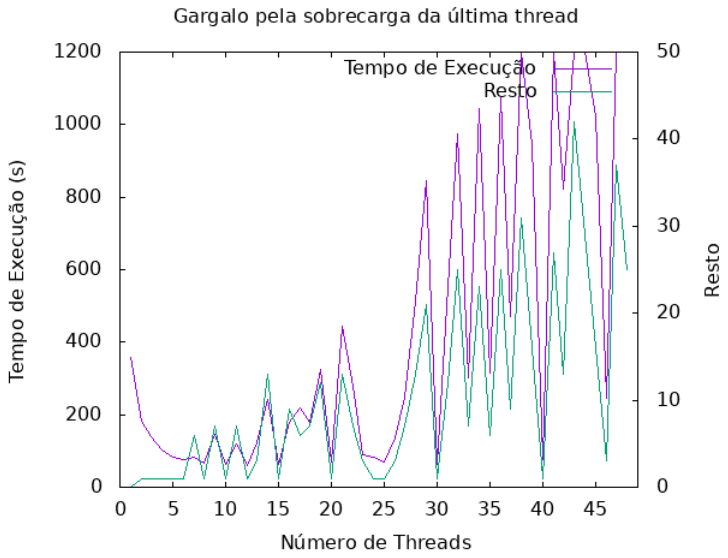
```
if(my_rank!=0){
        MPI_Send(&A[0][0],N,quad_type, my_rank-1, 99, MPI_COMM_WORLD);
        MPI_Recv(&aux1[0], N, quad_type, my_rank-1, 99, MPI_COMM_WORLD, &status);
}
if(my_rank!=size-1)
{
        MPI_Send(&A[nLin-1][0],N,quad_type, my_rank+1, 99, MPI_COMM_WORLD);
        MPI_Recv(&aux2[0], N, quad_type, my_rank+1, 99, MPI_COMM_WORLD, &status);
}
```

Fonte: Autor.

8 Resultados

A primeira versão do programa, em OpenMP, dividia o número de linhas (que graficamente se encontram na vertical) pelo número de *threads* e se houvesse resto nesta divisão este era todo acumulado na última *thread*. Dessa forma havia uma sobrecarga na última *thread* e portanto um gargalo proporcional ao resto dessa divisão, como podemos ver na Figura 14 o tempo de execução no SDumont (LNCC), correspondente ao Sequana.

Figura 14 – Ausência de escalabilidade devido a má divisão das regiões da malha. Pontos acima de 20 minutos foram execuções interrompidas por tempo limite.



Fonte: Autor. Dados gerados no Sequana/LNCC.

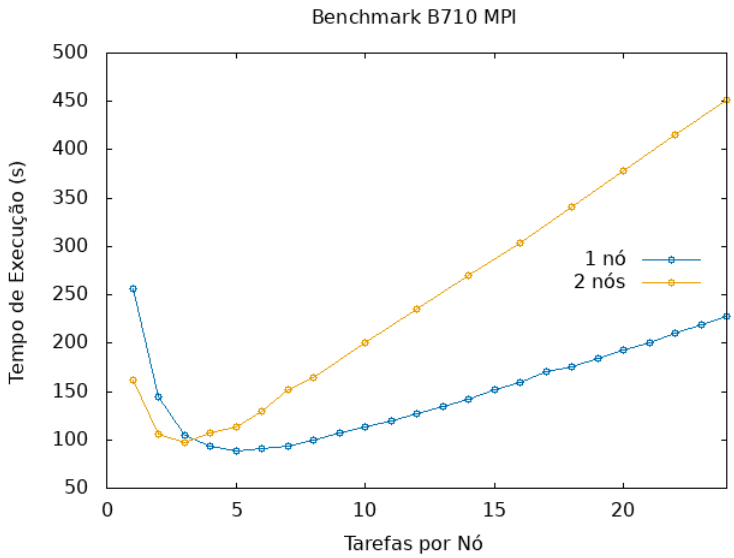
Através das diretivas *Scatterv* e *Gatterv* junto de seus parâmetros é possível, com o auxílio de um código, distribuir as tarefas sem que haja sobrecarga de algum trabalhador, como podemos ver na Figura 15, resultado do processamento no B710. A suavidade da curva de escalabilidade nos permite induzir que não há um acumulo de tarefa para algum *core*. De forma análoga também podemos ver o gráfico das informações do Sequana, Figura 16.

Para comparar a performance dos computadores disponíveis utilizando o MPI, foi gerado o gráfico dado pela Figura 17 que contém as curvas dos computadores utilizados em nó único.

A comparação entre a escalabilidade entre a versão OpenMP e MPI se dá entre as Figuras 17 e 18

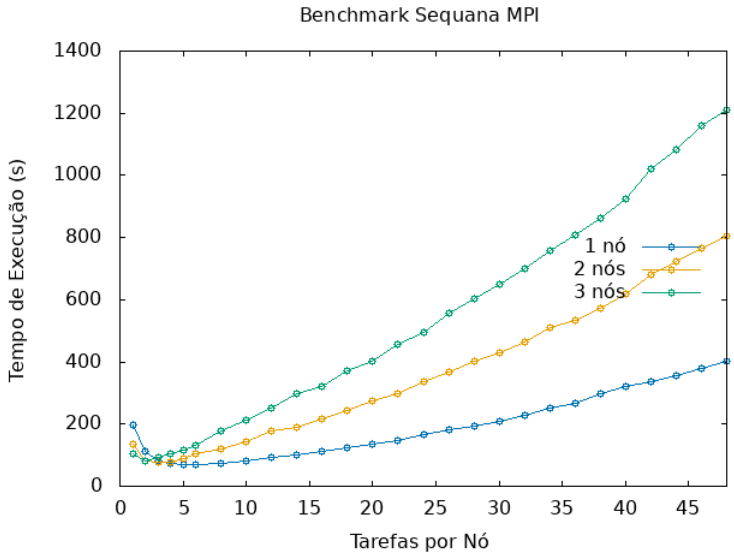
É de grande importância, ressaltar que os dados respectivos ao tempo de execução e eficiência, foram gerados sobre uma matriz 600x600 com uma tolerância sobre a máxima

Figura 15 – Escalabilidade do programa em MPI.



Fonte: Autor. Dados gerados no B710/LNCC.

Figura 16 – Escalabilidade do programa em MPI.



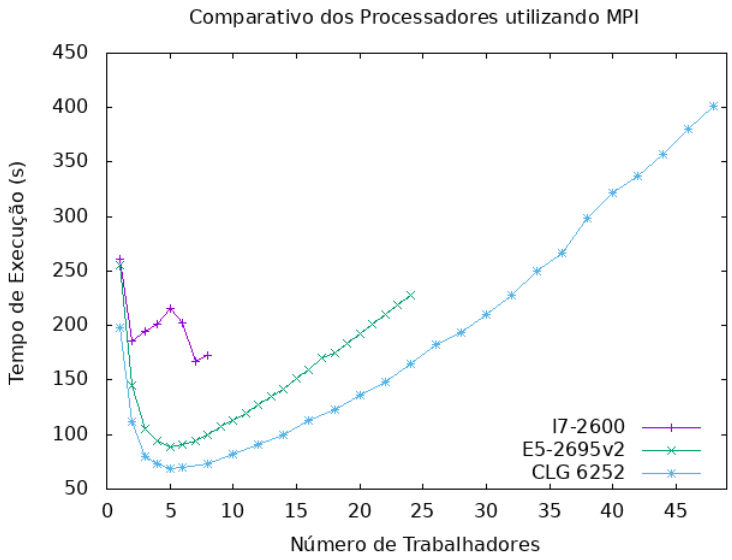
Fonte: Autor. Dados gerados no Sequana/LNCC.

flutuação de $10^{-5}V$.

Devido a dependência dos dados e a necessidade de comunicação durante as varreduras , podemos perceber uma menor eficiência do MPI quanto a escalabilidade. Figura 20 e 21.

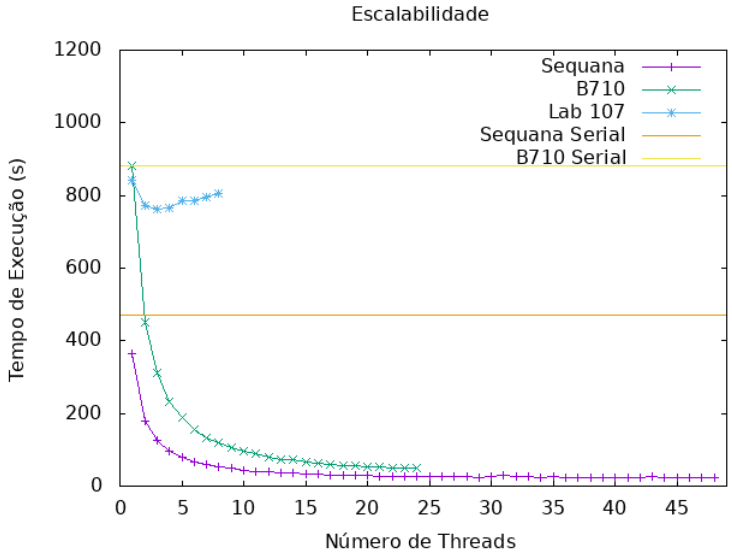
Partindo de um condição de contorno dada pela Figura 8, obtivemos os resultados dados pelas Figuras 22, 23 e 24, onde, respectivamente, suas tolerâncias quanto à flutuação eram de $10^{-}1V$, $10^{-}3V$ e $10^{-}5V$. Para esses resultados a matriz utilizada foi de dimensão 1200x1200.

Figura 17 – Tempo de Execução dos computadores do Laboratório 107(UFF) dos nós Sequana e B710(LNCC) com MPI.



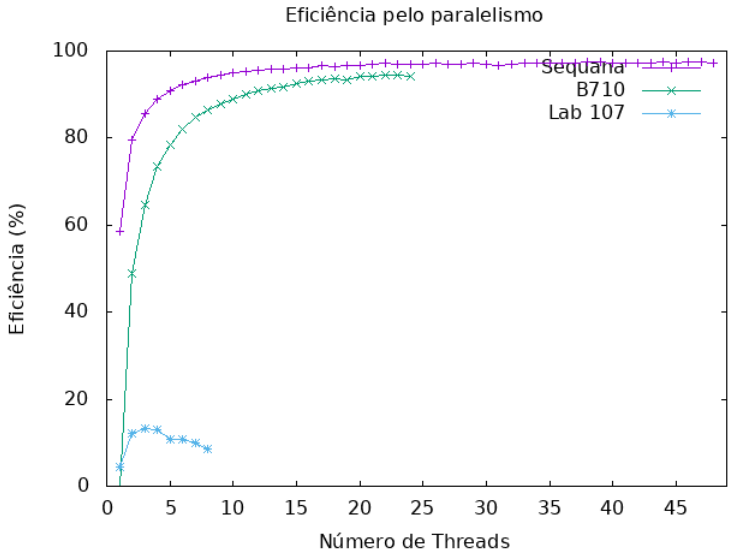
Fonte: Autor.

Figura 18 – Tempo de Execução dos computadores do Laboratório 107(UFF) dos nós Sequana e B710(LNCC) com OpenMP.



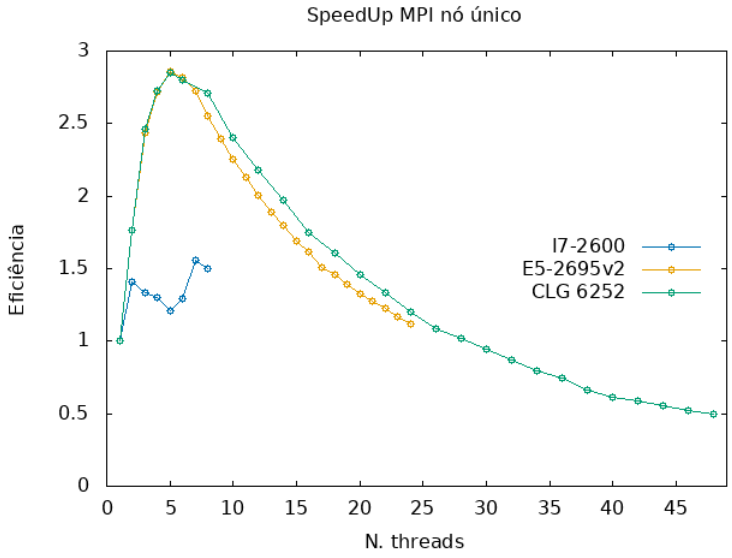
Fonte: Autor.

Figura 19 – Eficiência tendo como referência o tempo de um nó e tarefa única no Lab 107.



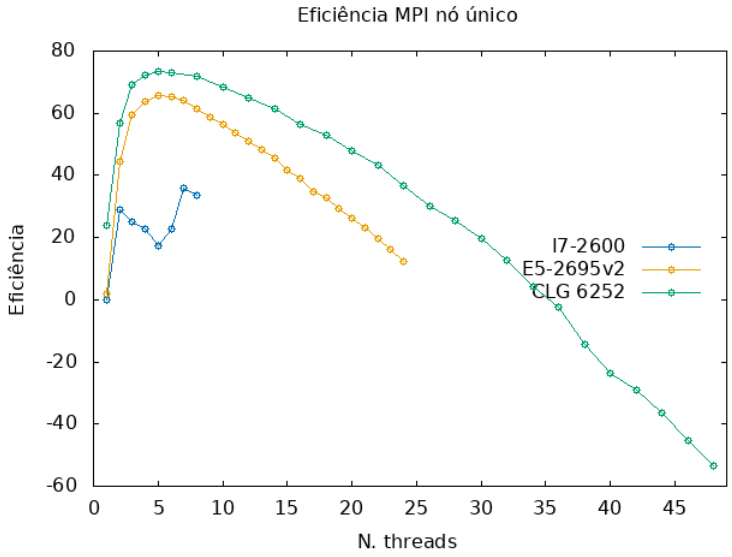
Fonte: Autor.

Figura 20 – SpeedUp do MPI em nó único.



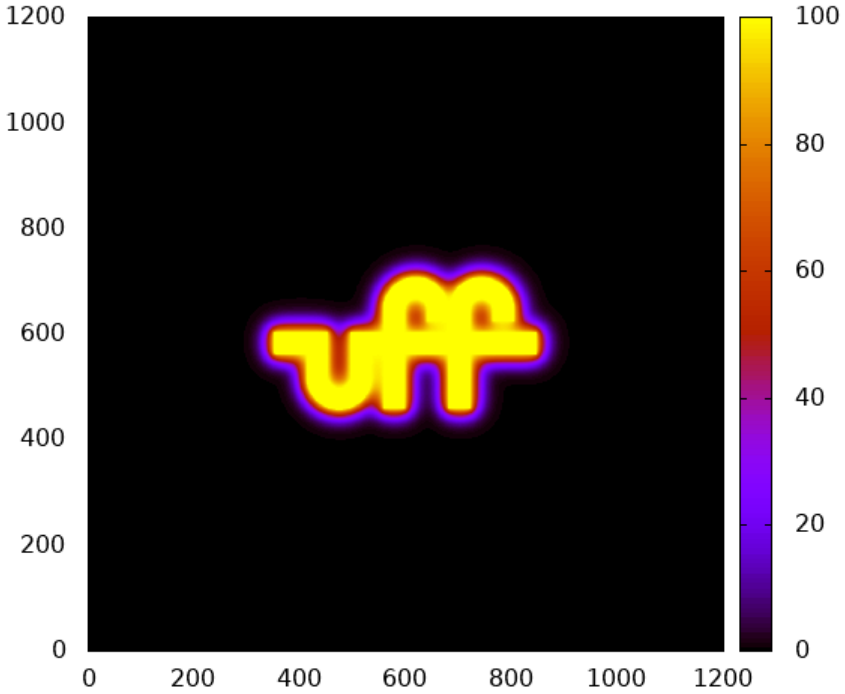
Fonte: Autor.

Figura 21 – Eficiência tendo como referência o tempo de um core no Lab 107.



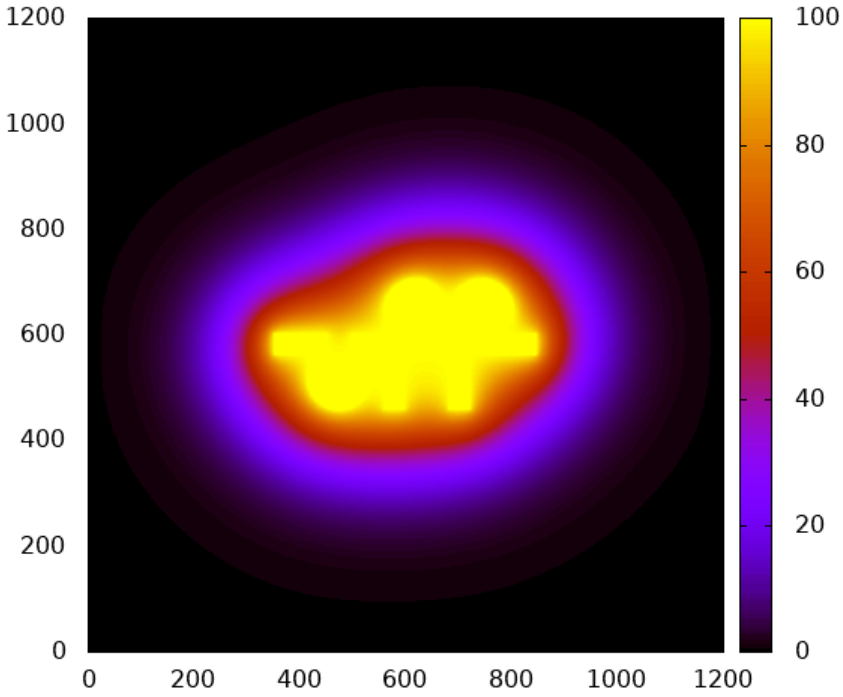
Fonte: Autor.

Figura 22 – Resultado com tolerância de $10^{-}1V$.



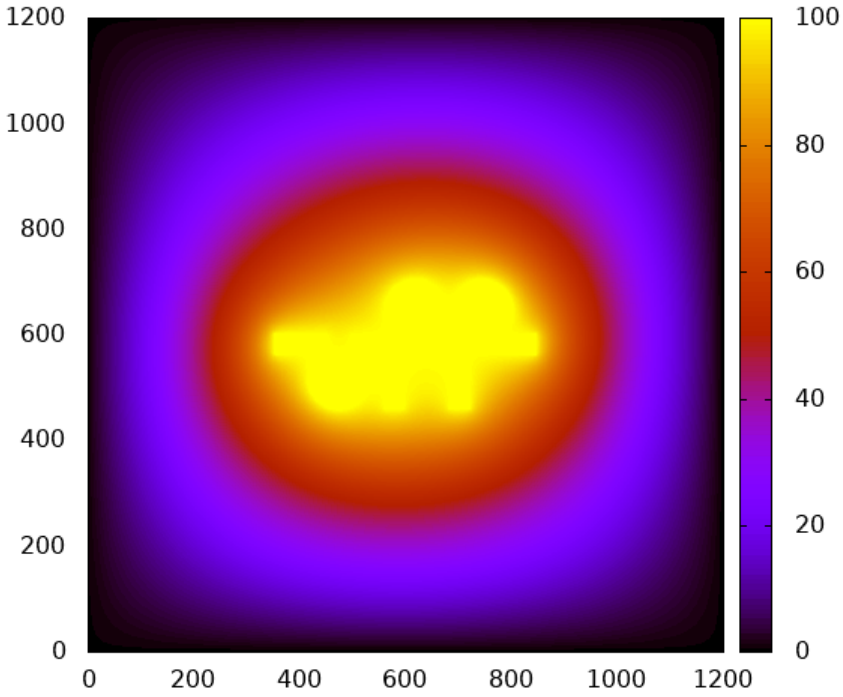
Fonte: Autor.

Figura 23 – Resultado com tolerância de $10^{-}3V$.



Fonte: Autor.

Figura 24 – Resultado com tolerância de $10^{-}5V$.

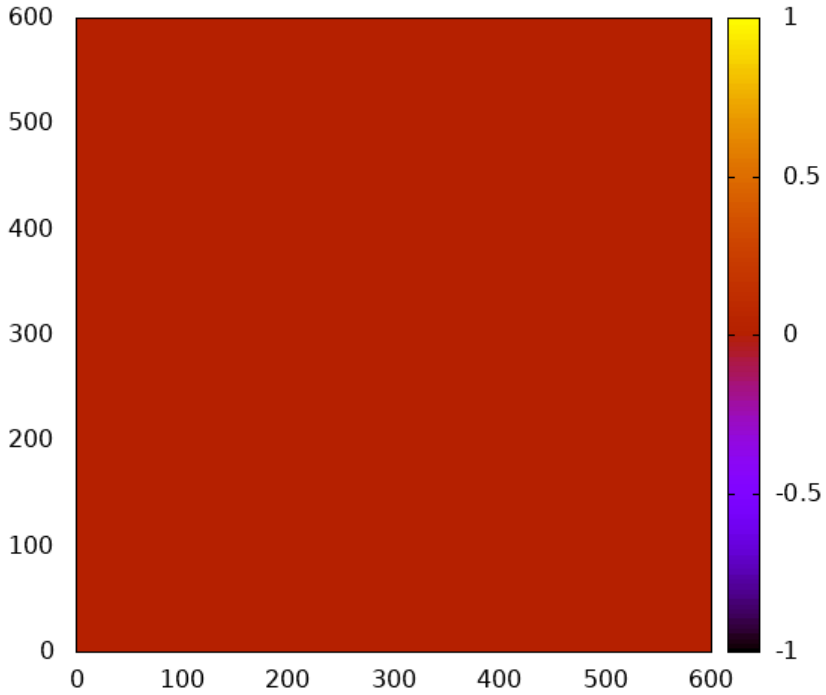


Fonte: Autor.

9 Validação dos Resultados

Tomando como referência o resultado obtido do programa na versão OpenMP, da diferença com o resultado em MPI, geramos o mapa de calor dado pela Figura 25. Portanto resultados numéricos idênticos.

Figura 25 – Diferença entre os resultados da versão em OpenMP e MPI.



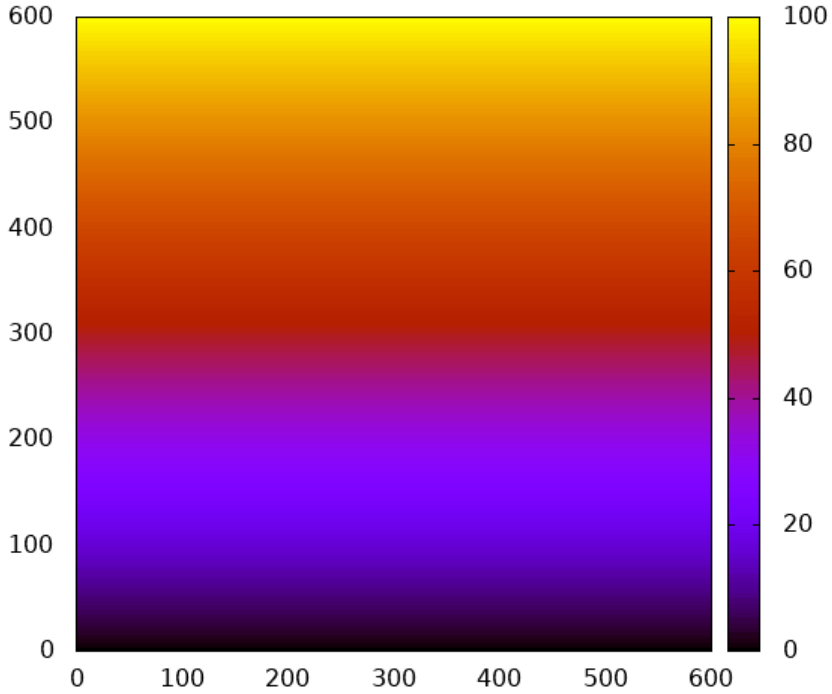
Fonte: Autor.

Para uma comparação do resultado do algoritmo utilizado com um resultado teórico conhecido, temos a Figura 26. Esta figura representa o potencial eletrostático de um sistema composto por uma placa em potencial constante ($100V$) e outra placa aterrada ($0V$), gerando-se assim o campo uniforme.

Da diferença entre o resultado teórico e o numérico, obtemos o mapa de calor dado pela Figura 27. Aqui podemos ver o comportamento difusivo do método do relaxamento junto das diferenças finitas, no qual a propagação da relaxação se dá das condições de contorno para os outros pontos. Resultado este que explica também o comportamento das Figuras 22, 23 e 24, onde era variado a tolerância da flutuação resultante do método da relaxação.

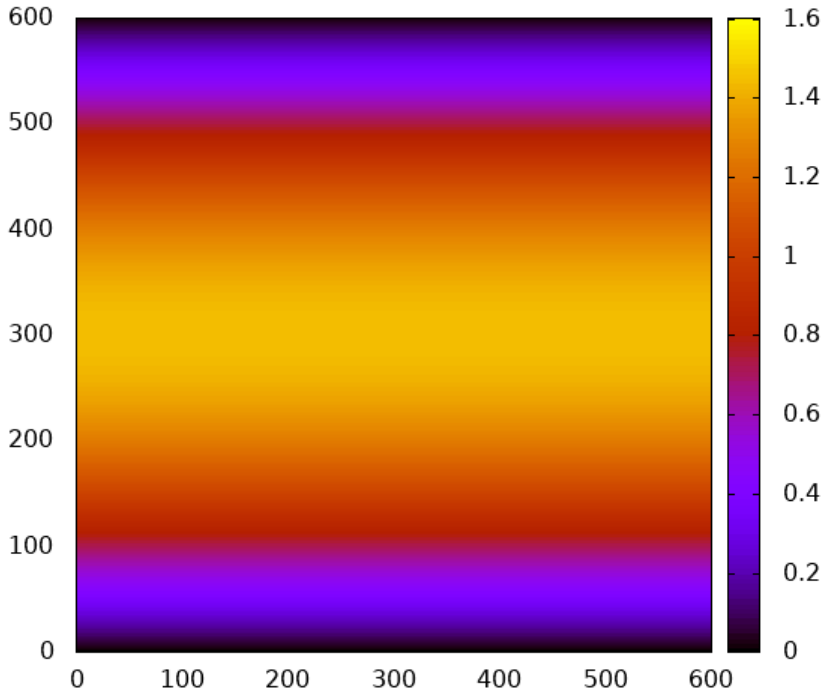
Outra característica interessante é quanto as simetrias que são compatíveis numa malha quadricular. Como podemos ver na Figura 28 a simetria radial deste campo começa a se esvair nas bordas da malha. Uma vez que neste problema o sistema deveria se estender

Figura 26 – Potencial de um sistema com uma placa a $100V$ e outra aterrada.



Fonte: Autor.

Figura 27 – Diferença entre o resultado teórico e o numérico.

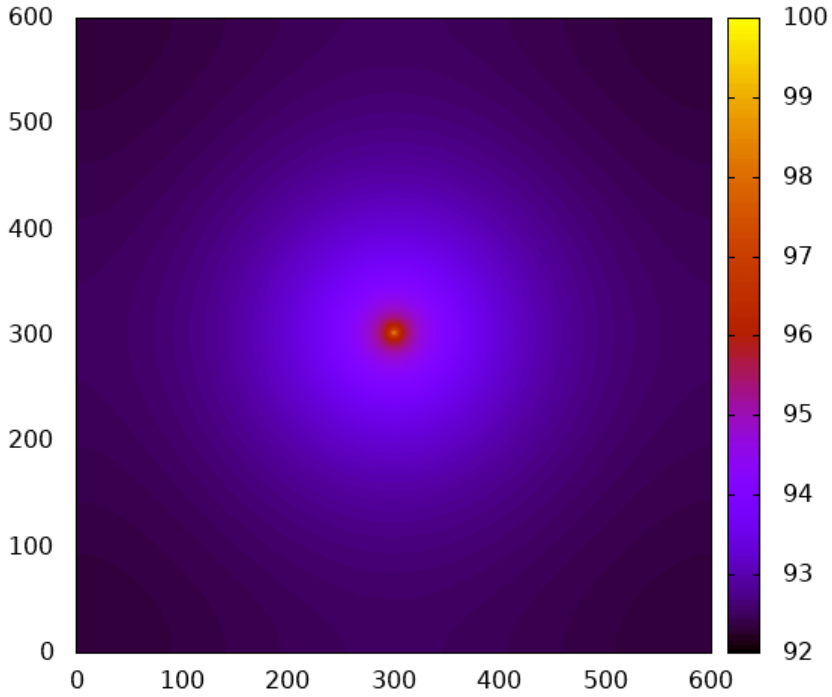


Fonte: Autor.

ao infinito, a forma comum de se contornar isso é de comunicar as laterais, opostas, entre si. Quando essa comunicação ocorre tanto no sentindo vertical quanto no horizontal o

topologia dessa malha acaba adquirindo uma característica irregular.

Figura 28 – Resultado numérico de uma carga pontual dentro de uma malha quadrada.



Fonte: Autor.

10 Conclusão

Apesar da anomalia, quanto a escalabilidade, dos processadores dos computadores de uso comum (Lab107), é notável a agilidade que pode ser obtida através da paralelização de um código, quando este assim o permite.

É intrínseco ao Método das Diferenças Finitas a dependência dos dados com seus vizinhos, exigindo-se assim a comunicação entre aqueles que processam esses dados. Sendo o OpenMP baseado numa paralelização de memória compartilhada, os cores são capazes de acessar os mesmos espaços de memória de forma simultânea. Já o MPI possuí o modelo de memória distribuída, sendo necessário uma comunicação entre os cores se assim for exigido pelo código.

Ao observar a escalabilidade da paralelização do MPI é possível notar uma perda de performance quando a divisão das tarefas é aumentada, ou seja, se dividirmos a tarefa em quantidade elevada a comunicação, que é intrínseca ao MPI, começa a se comportar como um gargalo.

Portanto conhecer o método numérico a ser utilizado é de vital importância para determinar as técnicas de otimização e de paralelização, de forma a obter um melhor resultado.

Referências

FILHO, A. A. V. I. V. J. M. F. *MPI: uma ferramenta para implementação paralela*. 2002. Disponível em: <https://doi.org/10.1590/S0101-74382002000100007>. Acesso em: 25 jan. 2022. Citado na página 9.

LNCC. *Supercomputador Santos Dumont (SDumont)*. Disponível em: <https://sdumont.lncc.br/support_manual.php?pg=support#1.1>. Acesso em: 10 set. 2021. Citado na página 12.

OPENMP. *About Us*. 2012. Disponível em: <https://www.openmp.org/about/about-us/>. Acesso em: 10 set. 2021. Citado na página 8.

OPENMP. *About Us*. 2012. Disponível em: <https://www.openmp.org/resources/openmp-compilers-tools/>. Acesso em: 10 set. 2021. Citado na página 8.

UFRGS. *Diferenças finitas*. 2020. Disponível em: <https://ciencia.ufla.br/reportagens/mercado/655-ufla-e-pioneira-em-laboratorio-remoto-para-ensino-de-fisica>. Acesso em: 2 set. 2021. Citado 2 vezes nas páginas 6 e 7.