2D Finite Difference Time Domain (FDTD) kernel

Felipe Augusto Chatalov, Lucas Beluomini e Matheus Molina Dias

***Resumo*—Representação do problema 2D Finite Difference Time Domain (FDTD) kernel, usando a linguagem de programação C para o desenvolvimento de duas soluções do problema, a primeira com o uso de Threads e a segunda com MPI, para paralelização dos cálculos visando aumentar a velocidade de processamento utilizando programação concorrente.**

I. Introdução

O Método FDTD é há muito tempo popular método computacional para resolver as equações de Maxwell [1] para eletromagnetismo, que pode ser vista na equação 1 abaixo.

$$\begin{aligned}\frac{\partial \overline{\mathrm{E}}}{\partial \mathrm{t}} &= \frac{1}{\varepsilon}\nabla \times \overline{\mathrm{H}} - \frac{1}{\varepsilon}\left(\overline{\mathrm{J}}_{\text{source}} + \sigma \overline{\mathrm{E}}\right) \\ \frac{\partial \overline{\mathrm{H}}}{\partial \mathrm{t}} &= -\frac{1}{\mu}\nabla \times \overline{\mathrm{E}} - \frac{1}{\mu}\left(\overline{\mathrm{M}}_{\text{source}} + \sigma^{*} \overline{\mathrm{H}}\right)\end{aligned} \tag{1}$$

FDTD pode resolver modelos com geometrias arbitrárias compostas de materiais dielétricos [2]. Além disso, o FDTD pode lidar facilmente com fontes transitórias e pulsadas, tornando-se uma ferramenta útil para muitos problemas em eletromagnetismo. No entanto, simulações com grandes espaços de modelo ou longas formas de onda não senoidais podem exigir uma quantidade enorme de cálculos de ponto flutuante e tempos de execução de vários meses ou mais são possíveis mesmo em grandes sistemas HPC [3]. Para isso, algoritmos de programação concorrente são criados, podendo melhorar o tempo da multiplicação entre matrizes e vetores que são efetuadas no método FDTD.

O método FDTD, apesar de suas forças, tem algumas fraquezas. Ele requer a discretização completa dos campos elétricos e magnéticos em todo o domínio do volume, o que pode resultar em modelar uma quantidade significativa de "espaço em branco" [4]. Além disso, como o método é explícito, isso pode vir a se tornar uma desvantagem se o passo de tempo se tornar muito pequeno. Representando desafios ao aplicar o método FDTD em problemas grandes que devem ser modelados com alta fidelidade [5].

II. Métodos

O algoritmo base do método FDTD se trata de um vetor ($\_fict$) com o tamanho relativo a quantidade de iterações, e três matrizes quadradas principais (ex, ey e hz) com o tamanho proporções nx x ny.

O número de iterações é passado pelo usuário como argumento, podendo ser em três tamanhos predefinidos:

- small - 120 Iterações
- medium - 240 Iterações
- large - 360 Iterações

O número de iterações é correspondente a variável $tmax$ que define o tamanho do vetor $\_fict$, assim também é feito seu preenchimento, com valores em sequencia de inteiros que vai de 0 até o valor de $tmax$ no formato *double*.

Já o tamanho das matrizes principais, são colocadas como fixas em 10240 linhas por 10240 colunas. Essa configuração é mantida nos três tipos de métodos de soluções abordadas, o método sequencial, o método de *threads* e o método de MPI. Seus valores são definidos da seguinte forma:

- $ex(i, j) = i \cdot (j + 1)/nx$
- $ey(i, j) = i \cdot (j + 1)/ny$
- $hz(i, j) = i \cdot (j + 1)/nx$

Sendo que essas atribuições são colocadas dentro de dois *for*s alinhados, percorrendo todas as linhas e colunas de cada matriz.

*1) **Método Sequencial**:* A primeira etapa do conta do FDTD parte de um preenchimento da primeira linha da matriz ey que é baseada totalmente nos valores de $\_fict$, ou seja, na primeira iteração a primeira linha será toda preenchida com 0, na segunda com 1 e assim por diante, de acordo com a equação 2.

$$E_y(0, j) = F(t) \tag{2}$$

Sendo $E_y(0, j)$ o valor do campo elétrico E_y na primeira linha da matriz ey na posição $(0, j)$ e $F(t)$ a função matemática que retorna o valor $\_fict[t]$ no índice de tempo t. Portanto, $F(t) = \_fict[t]$.

A seguir, é feito o preenchimento de toda a matriz ey para o tempo (iteração) t em questão, para isso são colocados dois *for*s aninhados preenchendo todas as linhas(exceto a primeira) e todas as colunas de ey, semelhante a função 3

$$E_y(i, j) = E_y(i, j) - 0.5 \cdot (H_z(i, j) - H_z(i - 1, j)) \tag{3}$$

Sendo $E_y(i, j)$ o valor do campo elétrico de E_y na posição (i, j), $H_z(i, j)$ o valor do campo elétrico de H_z na posição (i, j), para todo $i \geq 1$ e $j \geq 0$, portanto os valores da matriz ey dependem dela mesma e de hz.

Para os valores de ex é um processo semelhante, no entanto, sua primeira coluna não é preenchida nessa conta e a ultima subtração da equação passa a ser na coluna anterior de hz e não em sua linha anterior, como demonstrado abaixo 4.

$$E_x(i, j) = E_x(i, j) - 0.5 \cdot (H_z(i, j) - H_z(i, j - 1)) \tag{4}$$

Sendo $E_x(i, j)$ o valor do campo elétrico de E_x na posição (i, j), $H_z(i, j)$ o valor do campo elétrico de H_z na posição (i, j), para todo $i \geq 0$ e $j \geq 1$, portanto os valores da matriz ex também dependem dela mesma e de hz.

Por fim, para o preenchimento da matriz de hz dependemos dos valores já calculados de ey e ex anteriormente, a matriz é percorrida por completo com dois *for*s alinhados e contem a seguinte equação para cada posição.

$$H_z(i,j) = H_z(i,j) - 0.7 \cdot (E_x(i,j+1) - E_x(i,j) + E_y(i+1,j) - E_y(i,j)) \quad (5)$$

Sendo $E_y(i,j)$ o valor do campo elétrico de E_y na posição (i,j), $E_x(i,j)$ o valor do campo elétrico de E_x na posição (i,j), $H_z(i,j)$ o valor do campo elétrico de H_z na posição (i,j), para todo $i \geq 0$ e $j \geq 0$, portanto os valores da matriz hz também dependem dela mesma e das matrizes ey e ex.

Dessa forma podemos calcular os valores dos três componentes do campo magnético no ponto (i,j) para cada iteração, com $E_y(i,j)$ para o componente y do campo, $E_x(i,j)$ para o componente x e $H_z(i,j)$ para o componente z.

*2) **Método de Pthreads**:* Para a solução em paralelo foi necessário o uso da biblioteca "pthread.h", com ela é possível dividir processos mais custosos entre *threads* na CPU, visando o aumento da velocidade de execução do algorítimo.

O processo inicial é semelhante ao sequencial, a definição do numero de interações continua sendo feita por passagem de argumento e o tamanho das matrizes continua fixo no mesmo valor, no entanto, para a execução em paralelo é preciso passar, também, a quantidade de *threads* que serão usadas para o paralelismo. Esse dado também é passado por argumento e segue da flag $-t$. Portanto, para a execução do código em paralelo é passado $-d < dataset >$ com o numero de iterações que será armazenado em $TMAX$ e $-t < threads >$ com o numero de *threads* a serem usadas, armazenadas em $NUMBER\_THREADS$. Nesta etapa, ex e ey são calculados em funções diferentes dentro do código. Para ex, a divisão de *threads* é realizada da seguinte maneira:

$$NY/(NUMBER\_THREADS/2) \quad (6)$$

Já para ey, as threads são calculadas de maneira semelhante, a diferença é que a conta realizada utiliza NX no lugar de NY, como apresentado a seguir:

$$NX/(NUMBER\_THREADS/2) \quad (7)$$

onde NX e NY representam o tamanho total da linha e o tamanho total da coluna da matriz, respectivamente.

Essas divisões são feitas para que cada *thread* trabalhe com uma parte da matriz, primeiramente calculando a primeira linha da matriz ey, com os valores de $\_fict$, esses valores serão usados apenas para o calculo da primeira parte da matriz, que estará com a *thread* 0, portanto não será necessário barreira. Ao calcular o valor de ey esperam na primeira barreira o termino do calculo de ex.

Simultaneamente a outra parte de *threads* calcula a matriz ex, mas desta vez nenhuma variável depende dos valores do vetor $\_fict$, portanto é possível paralelizar sem restrições. Por fim também para na primeira barreira.

Com os valores de ey e ex calculados e prontos na primeira barreira, é possível começar o calculo da matriz hz, que depende dos valores calculados de ey e ex. Essa grande ultima matriz é dividida entre todos as *threads* já criadas, sendo, as *threads* que calcularam ey responsáveis pelo calculo do inicio ate metade da matriz hz, e as que calcularam ex responsáveis pelo calculo do meio ate o fim de hz.

Por fim, todas as *threads* em execução esperam na segunda barreira que está depois do calculo e hz, que determina que a matriz está completa e a equação de FDTD para aquela iteração finalizou.

*3) **Método MPI**:* Ao pensar em um escopo um pouco maior, é possível dividir a tarefa entre vários processos, esses processos podem ser executados na mesma máquina, ou em máquinas diferentes, desde que conectadas na mesma rede. Para aplicar essa metodologia, é feito o uso da biblioteca "mpi.h"em C.

Para a execução o algoritmo com MPI é preciso, assim como os métodos já citados, a quantidade de iterações que serão efetuadas no problema ($-d$ <TMAX>). O tamanho das matrizes continua fixo no mesmo valor que os métodos anteriores mas é preciso definir a quantidade de processos que estão disponíveis para a execução ($-np$ <Size_Of_Cluster>) localizado antes do nome do arquivo a ser executado.

A compilação e a execução do algoritmo que usa MPI é feito pelo "mpicc"e pelo "mpirun"respectivamente, para a compilação não há nenhuma diferença significativa em relação ao gcc, exceto que ao invés do gcc é usado o mpicc. Mas para a execução do arquivo é preciso usar o formato:

$ mpirun $-np$ <processos> <file> $-d$ <dataset>

No algoritmo de MPI a divisão de tarefas é um pouco diferente, o processo 0 recebe a função de calcular o tempo de execução, já que, apesar dele participar na conta, ele quem envia e recebe os dados depois de calculados. Para a divisão de dados é usado a função $MPI\_Scatter$, essa função envia os dados inseridos em partes iguais para todos os processadores (incluindo o 0), essa função é chamada 3 vezes, uma para a matriz ex, outra para ey e outra para hz.

Para o calculo, o processo 0 fica responsável por manusear o $\_fict$ novamente, usando-o para o calculo da primeira linha de ey. Em todos os processos inicia-se enviando a ultima linha da matriz para o processo seguinte, isso é necessário pois o processo $n+1$ utiliza a ultima linha do processo n para o calculo da sua matriz, portanto, todos os processos fazem isso, com exceção do ultimo, pois ele não tem um processo subsequente.

Em seguido o calculo é o mesmo que nos outros métodos, no entanto, o uso da variável $H_z(i,j-1)$ é substituída pelo dado recebido da matriz anterior, portanto, antes de iniciar o calculo, todas os processos diferentes do processo 0 devem receber esse dado do seu processo antecedente. Após o calculo de ex e ey todos os processos param em uma primeira barreira, ate que todos os processos tenham completado a conta, precisa-se dos dados de ex e ey para o calculo de hz.

Para iniciar o calculo de hz, todos os processos, com exceção do processo 0, enviam os dados calculados de ey para o processo antecedente, e todos os processos recebem de seu sucessor, da recepção o processo 0 participa, mas o ultimo processo não. Esses dados são enviados e recebidos com as funções MPI_Send e MPI_Recv e são necessárias para o calculo da ultima matriz (hz) que usa a linha posterior à atual.

A conta do hz é a mesma em todos os métodos, os dados são divididos igualmente para todos os processos e após a finalização da conta é colocada outra barreira no final para sincronizar os dados e conclusão da iteração.

*4) **MPI + Threads**:* Uma outra forma de implementação seria uma estratégia híbrida, utilizando MPI e *Threads*, dessa

forma, cada processo é dividido em N *threads*, o que, em teoria, traria uma melhora exponencial à solução.

Para isso, a estrutura seria semelhante ao usado em MPI II-3, no entanto, ao chegar no momento do calculo em específico, a matriz seria dividida em *threads*, sendo N *threads* para cada processo. Isso não acarretaria em mais barreiras, pois os momentos de intersecção de dados são os mesmos.

Após o uso dessas *threads* para o calculo de ex e ey, elas seriam divididas para calcular os dados de hz, exatamente como foi feito no método de *Threads* II-2. Por fim, com hz calculado, conclui-se na ultima barreira e iniciando a próxima iteração até o fim da execução.

III. Resultados

Todos os algoritmos foram executados ao menos 5 vezes, foi calculado média do tempo de execução entre esses 5 resultados, a matriz foi estabelecida com a largura de 10240 e altura de 10240, e os valores de *small*, *medium* e *large* correspondentes ao número de iterações, os resultados da execução sequencial usadas para a análise foram:

A. Sequencial

- Small - 120 iterações - 125 segundos
- Medium - 240 iterações - 250 segundos
- Large - 360 iterações - 375 segundos

Para a execução dos testes, foi utilizada uma máquina com 16GB de memória RAM, um processador AMD Ryzen 5-5600H e com o sistema operacional Linux Mint.

Para comparar os resultados de forma visual, é construído um gráfico de *speedup*, ele se baseia na divisão entre o tempo de execução do código sequencial e o tempo do código executado com o método que está sendo utilizado para paralelizar (*threads* ou MPI).

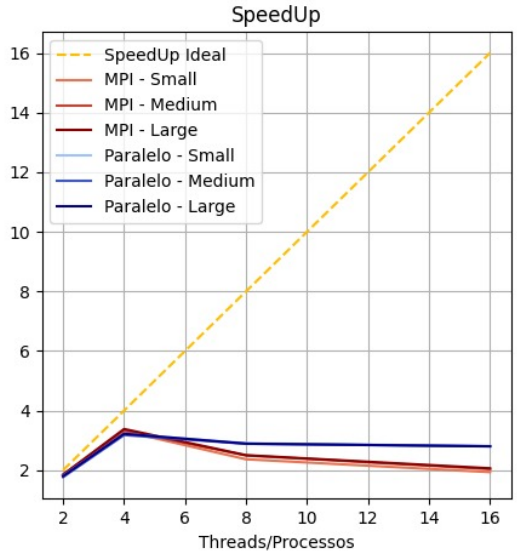

Figura 1: Comparação do *speedup* médio entre as execuções dos métodos MPI e Paralelo em cada tamanho

IV. Conclusões

É possível perceber uma melhora considerável de tempo de execução em ambas as estratégias de paralelização ate *threads* (no caso da estratégia de *threads*) ou processos (no caso da estratégia de MPI), as linhas se aproximam do *speedup* ideal, o significa que as estratégias de paralelização estão bem estruturados em ambos os métodos.

No entanto, há uma severa queda com uso superior a 4 *threads* ou processos. Isso se dá pelo fato do algoritmo ser executado em máquinas com núcleos insuficientes para tal tarefa. Com uma análise mais avançada utilizando a ferramenta *perf*, ferramenta utilizada para análise avançada de execução de programas, podemos perceber que utilizando mais de 4 processos ou *threads* há uma queda brusca do número de instruções por ciclo que vão de 4 (quanto utilizado 4 *threads* ou processos) chegando a 1 (quanto utilizado 16 *threads* ou processos).

Além disso, é possível reparar que após 4 threads há um aumento muito grande nas trocas de contexto de 6 por segundo com 4 *threads* ou processos para 24 trocas com 8 *threads* e 131 trocas com 16 *threads*. Outra mudança perceptível é a diminuição de ciclos do processador nos usos superiores a 4 *threads* ou processos, sendo menores a cada aumento de *thread* ou processo.

Referências

[1] H. Giefers, C. Plessl, and J. Förstner, "Accelerating finite difference time domain simulations with reconfigurable dataflow computers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 65–70, 2014.

[2] T. Stefański, S. Benkler, N. Chavannes, and N. Kuster, "Parallel implementation of the finite-difference time-domain method in open computing language," pp. 557–560, 2010.

[3] J. Saarelma, "Finite-difference time-domain solver for room acoustics using graphics processing units," *Master's thesis, Aalto University, Finland*, 2013.

[4] J. J. López, D. Carnicero, N. Ferrando, and J. Escolano, "Parallelization of the finite-difference time-domain method for room acoustics modelling based on cuda," *Mathematical and Computer Modelling*, vol. 57, no. 7-8, pp. 1822–1831, 2013.

[5] S. Adams, J. Payne, and R. Boppana, "Finite difference time domain (fdtd) simulations using graphics processors," pp. 334–338, 2007.