Computação de Alto Desempenho

Felipe Schreiber Fernandes

May 22 2021

1 Introdução

A equação que o programe busca a solucionar é uma equação diferencial parcial elípitica:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{1}$$

Ela modela uma gama de problemas nos quais conhecemos os valores de uma função na borda e queremos avaliar o seu interior. Para resolvê-la do ponto de vista numérico, uma etapa crucial é discretizá-la, que envolve dividir o interior da região com o auxílio de um grid. Assim, cada o valor em cada ponto do grid pode ser expresso pela influência dos pontos adjacentes (acima, abaixo, a esquerda e direita). Contudo, como não conhecemos o valor dos pontos interiores e muitos pontos interiores são vizinhos entre si, então precisamos chutar um valor inicial e a cada iteração atualizar com um valor mais preciso. A figura abaixo mostra isso. A condição de parada pode ser o número de iterações ou caso a máxima diferença, ou seja considerando todos os pontos do interior, entre atualizações sucessivas seja menor que um certo valor. Esse método é conhecido na literatura como o método de Gauss-Seidel.

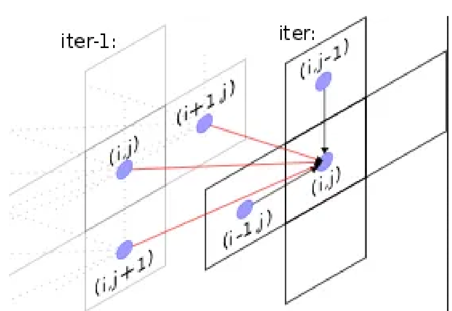

Figure 1: Atualização do valor da função no ponto interior pelo método iterativo de Gauss-Seidel

```
Real cte = 0.5/(dx2 + dy2);
#pragma omp parallel for shared(u) private(i,tmp) num_threads(this->NUM_THRDS)
for (i=1; i<nx-1; ++i) {
    #pragma omp parallel for shared(u) private(j,tmp) reduction(+:err) num_threads(this->NUM
    for (j=1; j<ny-1; ++j) {
        tmp = u[i][j];
        u[i][j] = ((u[i-1][j] + u[i+1][j])*dy2 +
                   (u[i][j-1] + u[i][j+1])*dx2)*cte;
        err += SQR(u[i][j] - tmp);
    }
}
return sqrt(err);
```

Figure 2: Loop que consumia maior tempo do programa, após modificação

Como o programa pode levar muitas iterações, de acordo com tão pequeno for o erro desejado e número de pontos no grid, a perfilagem de código é de extrema importância. Conforme resultado do trabalho anterior, principais hotspots são a função timeStep e a SQR. Uma análise mais detalhada do código nos leva a ver que ele não obedece aos padrões de boa prática de programação, pois ele chama função no loop interno, por exemplo.

2 Alterações realizadas

Elas consistiram em, principalmente:

- Acrescentar a diretiva **#pragma omp parallel for shared(u) private(i,tmp) num_threads(this->NUM_THRDS)** ao loop externo e,
- Acrescentar **#pragma omp parallel for shared(u) private(j,tmp) num_threads(this->NUM_THRDS)** ao loop interno

Importante notar que this->NUM_THRDS corresponde ao um atributo adicionado a classe. Ele informa quantas threads serão utilizadas. Dessa forma, o loop que mais consumia recursos computacionais ficou: As alterações acima foram possíveis pois relaxamos uma condição que antes estava presente: as atualizações de cada valor da matriz precisava seguir a ordem estabelecida. Uma vez que essa condição foi relaxada, podemos criar threads para paralelizar os loops interno e externo. Isso foi feito colocando pragma omp parallel for. Além disso, é necessário passar os parâmetros shared e private para determinar como os dados estarão disponíveis para cada thread.

O parâmetro shared(u) informa que matriz está disponível para todas as threads e portanto uma cópia não é necessária. A exclusão mútua não foi utilizada pois, apesar da matriz estar sendo atualizada por diferentes threads ao mesmo tempo, cada thread irá modificar apenas seu respectivo índice i,j passado, não havendo condição de corrida.

Por outro lado, as variáveis i,j e tmp precisam estar privadas para cada thread. Isso porque cada thread irá modificar apenas um índice específico da matriz. Caso o private não fosse utilizado, estaria sujeito a mais de uma thread ler o mesmo valor dos índices e tentar modificar a mesma posição na matriz, gerando uma condição de corrida. De maneira similar, a variável tmp poderia ser sobreescrita com o valor em outra thread antes que SQR(u[i][i]-tmp) na thread original fosse computado, gerando novamente uma condição de corrida.

Por fim o parâmetro reduction(+:err) informa que a variável privada err ao final do bloco paralelo é somada com o valor de todas as demais threads. O parâmetro num_threads(this->NUM_THRDS) apenas especifica o número de threads que serão utilizadas.

3 Resultados

De acordo com as especificações, foram gerados 4 variantes do código em questão: uma sem qualquer otimização, outra apenas incluindo a vetorização por meio da flag -O3, uma utilizando apenas o OMP com as modificações listadas acima, e por fim uma com a vetorização e o OMP. Cada um desses 4 programas foram executados modificando o tamanho do grid $NxN, N \in 512, 1024, 2048$ e, no caso daqueles que utilizam OMP, a quantidade de threads utilizadas $NUM\_THRDS \in 1, 2, ..., omp\_get\_max\_threads()$. Foram medidos o tempo de execução de cada uma dessas execuções, gerando a tabela 3.

4 Discussão dos resultados

A partir da tabela acima, podemos notar alguns fatos interessantes:

- Conforme duplicamos N, o tempo de execução é multiplicado por 4 no caso do programa sem qualquer otimização, o que é de se esperar pois o tamanho da matriz é na ordem de N^2. Porém nos demais esse aumento é um pouco menor.

- O programa com omp e vetorização, fixando a quantidade de threads em 1, era de se esperar que tivesse um resultado similar aquele apenas com a vetorização. Mas os dados apontam que obteve um resultado 2x melhor.

- Utilizando apenas o OMP, parece que aumentar o número de threads de 2 para 3 piora o resultado, possivelmente por conta do overhead criado. Utilizando OMP em conjunto com a vetorização o aumento da quantidade de threads impactou diretamente, melhorando bastante o tempo de execução.

- No geral, a combinação da paralelização com a vetorização foi o que gerou os melhores resultados, independente de N.

	program	nx	num_threads	tempo
0	./laplace	512	1	4.204004
1	./laplace	1024	1	17.548815
2	./laplace	2048	1	76.868279
3	./laplace_vec	512	1	3.248734
4	./laplace_vec	1024	1	13.591241
5	./laplace_vec	2048	1	60.024839
6	./laplace_omp	512	1	3.986706
7	./laplace_omp	512	2	2.481313
8	./laplace_omp	512	3	2.664023
9	./laplace_omp	1024	1	15.337704
10	./laplace_omp	1024	2	9.218728
11	./laplace_omp	1024	3	9.876598
12	./laplace_omp	2048	1	65.666409
13	./laplace_omp	2048	2	36.850189
14	./laplace_omp	2048	3	38.591171
15	./laplace_omp_vec	512	1	2.076559
16	./laplace_omp_vec	512	2	1.225685
17	./laplace_omp_vec	512	3	0.925183
18	./laplace_omp_vec	1024	1	7.618921
19	./laplace_omp_vec	1024	2	4.038153
20	./laplace_omp_vec	1024	3	3.063765
21	./laplace_omp_vec	2048	1	33.620965
22	./laplace_omp_vec	2048	2	15.217518
23	./laplace_omp_vec	2048	3	10.825928

Figure 3: Tabela de resultados