Relatório APS01 - Supercomputação
Insper, Abril/2018
Pedro Cunial Campos

Introdução

Para a APS01 da disciplina de Super Computação do curso de Engenharia de Computação do Insper, era esperado o desenvolvimento de um programa de alto desempenho utilizando OpenMP para o cálculo da temperatura de uma placa metálica ao longo do tempo (problema já estudado na disciplina anterior de "Transferência de Calor e Mecânica dos Sólidos").

Sabendo que a temperatura de um dado ponto desta placa pode ser calculado a partir do seu valor anterior, tal como o de seus vizinhos, o cálculo pode ser traduzido para a seguinte equação:

$$T_{i,j}^{t+1} = T_{i,j}^{t} + F_0(T_{i-1,j}^{t} + T_{i+1,j}^{t} + T_{i,j-1}^{t} + T_{i,j+1}^{t} - 4T_{i,j}^{t})$$

Onde $T_{i,j}^{t+1}$ corresponde a temperatura em um dado ponto (i, j) da placa no instante seguinte ao tempo sabido (t).

O resultado desta simulação deve gerar um conjunto de imagens PGM, as quais devem ser unidas em um gif.

Implementação Sequencial

O processo de tradução do sistema para código fora a parte mais turbulenta do projeto. Tive problemas com o uso errado de macros (não estava colocando parênteses em volta das variáveis da mesma), alocação dinâmica de memória (estava alocando e dealocando uma matriz em cada iteração do cálculo, o que não funcionava para mais de três iterações, estourando o limite de endereços fornecidos para um programa comum em C) e no tratamento de casos específicos da placa (como as quinas e bordas).

Sobre o primeiro problema (macros), resolvi-o apenas colocando parênteses nas instâncias das variáveis dentro da mesma no caso da macro *itomi*. No caso da macro que estava utilizando para o cálculo da equação, substituí o mesmo por uma função *extern inline* (*upd_tmp*).

O problema da alocação dinâmica de memória foi resolvido ao reutilizar o array das diferenças (*diffs*) em todas as iterações.

Por fim, o cálculo de quinas e bordas foram tratados individualmente. Imaginando que a performance em paralelo do cálculo poderia ser prejudicada caso houvessem muitas condições em um único loop, o tratamento das bordas e quinas foi feito separado do resto da placa

(funções *calculate_borders* e *sequencial_calculate_borders*). O motivo de iterarmos em loops separados quanto a cada caso (topo, baixo e laterais) é para otimizações no uso do cache.

Implementação Paralela

Com o código sequencial pronto, a maneira mais trivial de paralelização seria o uso do *omp parallel for* no loop principal do código. No entanto, isso não melhorou a sua performance (na verdade, deixou o seu tempo de execução quase que o dobro do original).

Uma segunda tentativa foi o uso de tasks em loops pequenos (como os loops no começo da função do cálculo das bordas) para que pudessem ser executados em paralelo entre si. Novamente, esta otimização não surtiu o efeito esperado, quase que triplicando o tempo do programa.

Por fim, a solução “menos ruim” foi o uso de reductions na função de tirar a média das diferenças (*parallel_avg*).

No entanto, ao testar o desempenho do código em matrizes maiores, percebi que a diferença do desempenho entre o sequencial e o paralelo estava cada vez menor (sendo quase o mesmo tempo para uma matriz de 100x100). Desta forma, considerei as otimizações iniciais como úteis, e as mantive no código pelo desempenho em larga escala.

Considerações Finais

Após tentativas frustradas de otimização pela paralelização, percebi que boa parte dos atos que estava tomando com o intuito de melhorar o desempenho como aprendido em matérias anteriores estavam na verdade atrapalhando o desempenho do código paralelo (tal como o uso de flags de otimização do compilador, como o *-O2)*, como a retirada do uso de *loop unrolling* do código.

Pude perceber também a importância do overhead associado à paralelização de um código, vista a superior performance do código para entradas maiores.