Computação Paralela

3D Fluid Simulation

Eduardo Cunha
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg55939@uminho.pt

Filipe Pereira
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg55941@uminho.pt

José Pacheco
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg55972@uminho.pt

I. Introdução

Para esta segunda fase do projeto, foi introduzida uma nova função, *lin_solve*, baseada num algoritmo *red-black*. Além disso, o tamanho dos dados foi aumentado de SIZE=42 para SIZE=84. O principal objetivo desta etapa é aplicar diretivas OpenMP para melhorar a eficiência do código, com foco na sua paralelização

II. Paralelização com OpenMP

A análise inicial do *profiler* identificou os principais *hot spots*: a função ***lin_solve*** (79,93% do tempo total de execução), e a função ***advect (***11,99%).

Na função ***lin_solve***, os dois *for loops* foram paralelizados independentemente utilizando a diretiva **#pragma omp parallel for schedule(static) private(old_x, change) reduction(max:max_c)**.

Utilizou-se ***reduction(max)***, que cria cópias da variável ***max_c*** para cada *thread*, agregando os valores no final para obter o maior. É mais eficiente que métodos manuais, como ***critical*** ou ***atomic***, graças à sincronização interna.

Analisámos os escalonamentos estático e dinâmico. O escalonamento estático reduz o *overhead* na gestão das *threads*, já que os blocos de iterações atribuídos a cada *thread* são pré-definidos. O escalonamento dinâmico proporciona um maior equilíbrio de carga entre as *threads*, no entanto, introduz um *overhead* adicional na gestão das *threads*. Como o trabalho entre *threads* é sempre semelhante, o escalonamento estático apresentou o melhor desempenho.

As variáveis ***old_x*** e ***change*** foram declaradas como *private*, para evitar *data races*. Após nova análise, a função ***lin_solve*** deixou de ser um *hotspot*. Para otimizar ainda mais o código, foram aplicados outros *pragmas*, ao longo do programa para explorar ao máximo o potencial de paralelização, como **pragma omp for schedule(static)**, tendo sempre em atenção a ausência de *data races*.

III. Ajustes no código

A tentativa de paralelização dos dois *loops* utilizando *tasks* foi inviável devido às dependências existentes entre os loops, essenciais para a corretude. Pelo mesmo motivo, o *do-while*, não pôde ser paralelizado, já que as iterações dependem das anteriores para garantir a convergência.

Foi testada uma redução da zona crítica, armazenando os valores de **change** num array temporário e aplicando a operação de redução após os dois loops. Contudo, verificou-se que as instruções adicionais relacionadas com a manipulação do array anularam os potenciais ganhos da redução da zona crítica.

Após uma leve vetorização dos *loops*, testou-se a diretiva **simd** para reforçar a vetorização com *threads*. Apesar das alterações observáveis no código *assembly* gerado (mais instruções vetoriais), estas mudanças não manifestaram uma melhoria significativa no tempo de execução. Presume-se que o *bottleneck* esteja nos acessos à memória, limitando os ganhos da vetorização devido aos *cache misses* causados pelas *threads* e pela diminuição de acesso sequencial à memória.

No fim, o grupo realizou verificações manuais extensivas para garantir a inexistência de *data races*.

Posto isto, as otimizações que demonstraram ser efetivas envolveram melhorias no desempenho de memória e ajustes na estrutura do código. A ordem dos ciclos foi alterada de i-j-k para k-j-i, reduzindo significativamente os cache misses e melhorando o tempo de execução graças a um acesso mais eficiente à memória. Adicionalmente, a função lin_solve foi adaptada para armazenar separadamente os somatórios sum_i, sum_j e sum_k, o que ajuda o compilador a identificar oportunidades de vetorização, e consequentemente, melhorar ainda mais o desempenho.

IV. Análise de Escalabilidade

O melhor tempo obtido foi de 1,48s com 20 threads. A partir das 21 threads o tempo de execução não diminui mais e, eventualmente, aumenta. Isto ocorre devido a diversos fatores como, *overhead* de paralelização relativo às trocas de contexto entre threads e à sincronização, a limitação imposta pela Lei de Amdahl, onde as partes sequenciais do código limitam os ganhos de desempenho, e também o aumento da competição pelo acesso à memória, que provoca atrasos significativos devido à limitação na largura de banda e na latência da memória.

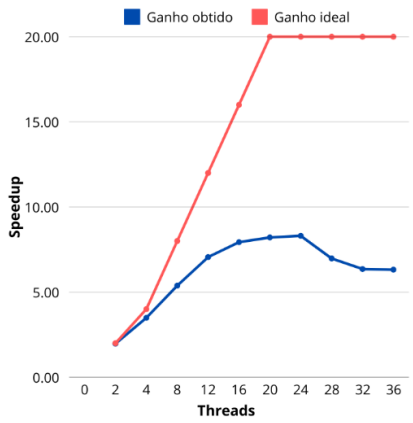

Figura 1 – Speedup relativamente ao numero de Thread