Universidade Federal do Rio Grande do Norte

Departamento de Engenharia de Computação e Automação
DCA0123 - Programação Concorrente e Distribuída

Relatório do VOLREND para o OpenMP

Alunos:
Felipe Oliveira Lins e Silva
Heitor Carlos de Medeiros Dantas
Luís Gabriel Pereira Condados
Matheus Oliveira de Freitas

3 de Janeiro de 2020

1 Descrição da aplicação e estrutura dos códigos original, portado e otimizado

O Volrend é uma aplicação do PARSEC que utiliza o Raytrace para a renderização de volumes. Esse algoritmo consiste em percorrer a imagem (projeção da cena tridimensional), traçando um raio para cada um de seus pontos a fim de calcular a sua cor –simulando a forma como nosso olho captura os raios de luz da cena.

Essa aplicação recebe um modelo tridimensional de um crânio humano e gera um conjunto de imagens, cada uma representando um quadro da rotação do volume em três eixos definidos pelo usuário.

O Volrend ainda faz uso de algumas otimizações no algoritmo de Raytrace: hierarchical opacity enumeration, early ray termination, and spatially adaptive image sampling.

- *Hierarchical opacity enumeration*: utiliza uma octree para evitar a renderização de regiões onde tem pouca presença de material do volume.
- *Early ray termination*: estabelece um limiar de opacidade para controlar quantas vezes o raio vai persistir.
- *Spatially adaptive image sampling*: Traça os raios de forma igualmente espaçada. Se a diferença de cor entre eles abaixo de um limiar, as cores do pixels entre os raios traçados são definidos por meio da interpolação do pixels vizinhos. Caso contrário, a região entre os raios traçados é subdividida e os raios são traçados nos cantos, repetindo em seguida o passo anterior.

Além disso, a aplicação faz uso do paralelismo de dados, dividindo a imagem a ser renderizada para as *threads* em regiões. Essas regiões ainda são subdivididas em blocos menores (*tiles*), de modo que se uma *thread* termina de processá-los, ele passa a processar os *tiles* das outras *threads* que ainda não foram finalizadas, a fim de equilibrar a carga de trabalho.

2 Descrição das estratégias de otimização do código em OpenMP

Como a função **Trace_ray** era a mais chamada e a que mais ocupava tempo na CPU, ela foi o primeiro alvo para ser otimizada. Porém, devido a sua alta complexidade em decorrência das mudanças de fluxos (**goto's** e **if's**), preferiu-se não alterá-la. Mas a função também continha alguns laços simples e com poucas iterações que, a primeira vista, não valia a pena otimizá-la. No entanto, como esta função é chamada milhões de vezes, foi efetuada a vetorização deles com **#pragma omp simd**.

Em seguida, foi analisada a função acima na hierarquia de chamadas de função: **Ray_Trace_Non_Adaptively**. Ela percorre cada *tile*, traçando os raios dos pixels, fazendo a distribuição de dados mencionada no tópico anterior. Esta estratégia foi substituída por apenas distribuir os pixels da imagem entre as *threads* de forma dinâmica usando **#pragma omp for schedule(dynamic, 32)**, que gerou os melhores resultados comparado a outros escalonadores testados.

No modo adaptativo, o **Trace_ray** é chamado pela função **Ray_Trace_Adaptive_Box**, que traça os raios esparsamente de forma recursiva pela imagem. Como os pixels da borda de cada *tile* eram compartilhados com as outras *tiles*, a aplicação implementou um busy wait para garantir seu acesso exclusivo. Porém isso ocasionava no travamento do programa no modo adaptativo para múltiplas *threads*. Assim, isso foi substituído por um lock para cada pixel.

Além disso, a divisão em regiões do **Ray_Trace_Adaptively** foi removida, de forma que no lugar de tratar a imagem como um conjunto de *tiles*, ela foi tratada como um vetor unidimensional para, assim, realizar um melhor escalonamento com menos *overhead* ocasionado pela divisão dos tiles – por meio do uso do **#pragma omp for schedule(dynamic, 32)**.

Na função **Ray_trace**, que efetua a renderização de cada frame, foi vetorizado apenas alguns *loops* simples com **#pragma omp simd**

Por fim, a última função relevante no perfilamento no código e que podia ser otimizada era a *Frame*, responsável por renderizar os frames, inicializar as imagens e gerá-las. Nela, foi apenas paralelizado os laços relativos a inicialização da imagem e da máscara, este último utilizado apenas no modo adaptativo para indicar o status de cada pixel.

3 Dificuldades encontradas e soluções aplicadas

A primeira dificuldade encontrada foi no processo de entendimento do código, a estrutura que foi implementada na aplicação e a forma que foi escrito o código, além das MACROS usadas para a paralelização, fizeram o código ser difícil de compreender. Para melhor entender o código tivemos que ler o artigo original do projeto, o qual foi de grande ajuda para o grupo.

Na portabilização para *OpenMP*, a maior dificuldade foi traduzir o gerenciamento de zonas criticas, no mais não apresentou grandes dificuldades, pois foi possível fazer uma tradução direta do código original para *OpenMP*.

4 Análise comparativa de escalabilidade para os 3 códigos

Para analisar a escalabilidade da aplicação, foi acrescentado ao código medidores de tempo, de "relógio de parede", por meio da função *runtime* do *OpenMP* (*omp_get_wtime()*), antes e depois da região paralela, que no caso ocorre na função *frame*, variando-se o número de *threads* e o tamanho do problema, que para essa aplicação o tamanho do problema é o número de rotações, que se traduz em número de rotações do modelo 3D, calculamos a eficiência e plotamos com mapas de calor. Vale salientar que todos os testes foram feitos no supercomputador do *NPAD*, com 10 execuções de cada configuração e obtido a mediana, como valor representativo.

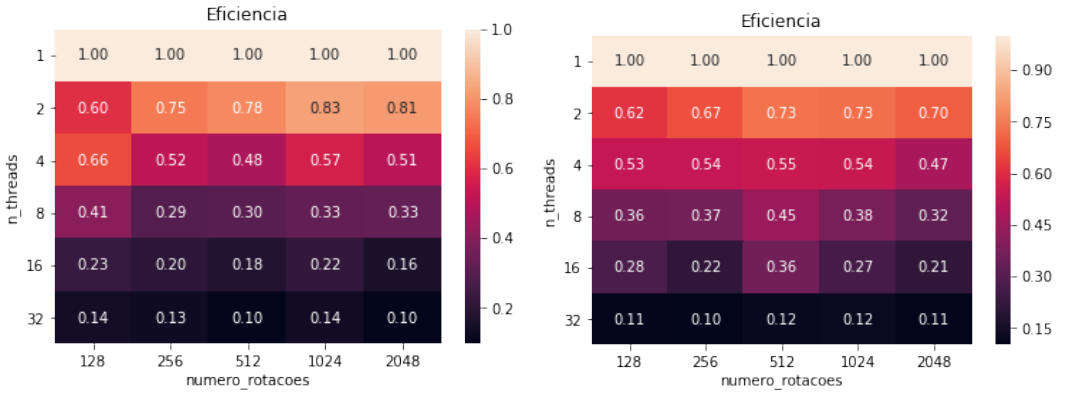

Figura 1: Versão original (à esquerda) e portada (à direta).

Observa-se na figura 1 que em ambas as versões da aplicação, original e portada, não houve características de escalabilidade, ou seja, a eficiência não se mantém constante, uma pequena diferença é notável entre essas versões. Já a versão portada apresentou eficiências um pouco menores na maioria das configurações.

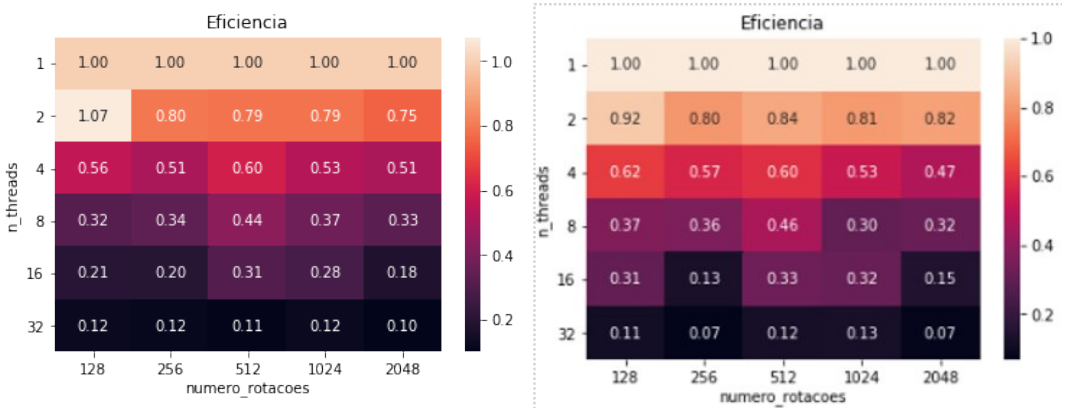

Figura 2: Versão Otimizada

Na figura 2 tem-se duas variações da versão otimizada, diferem entre si, pelo algoritmo de escalonamento utilizado na iteração que traça os raios, no modo não adaptativo, na figura mais a esquerda, foi utilizado o escalonador *guided* com tamanho mínimo de bloco igual a 32 e na mais a direita o escalonador *dynamic*, também com tamanho de bloco igual a 32. Observamos que ambas as versões otimizadas, apresentaram-se mais eficiente que as versões original e portada, embora ainda sem características fortes de escalabilidade. A otimizada com uso do *guided* apresentou uma eficiência maior que 1 (super eficiência) isso foi devido ao uso ótimo da cache.

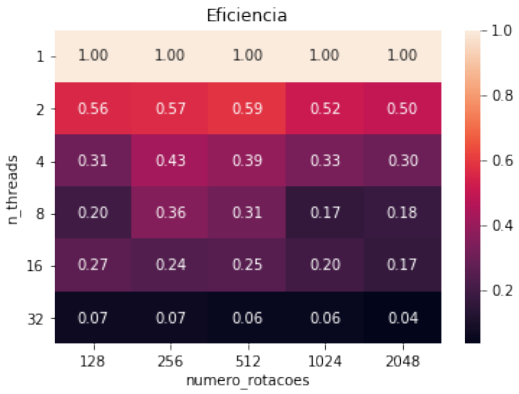

Figura 3: Versão adaptativa

Por fim a figura 3 apresenta os testes com a versão adaptativa do código otimizado, não foi possível realizar os mesmos testes com o código original, pois o mesmo apresenta *deadlock* em execução em paralelo. Podemos inferir que o adaptativo, embora apresenta menor tempo de execução para a versão serial, sua eficiência em função do número de *threads* x tamanho do problema, não mostrou-se escalável, assim como as demais versões, mas também apresentou-se menos eficiente que as apresentadas acima.

5 Conclusões

Por fim, a aplicação não se mostrou ser escalável, nem fracamente nem fortemente. Pela natureza do algoritmo, nossas hipóteses eram de que seria possível obter bons resultados. Uma possível causa para isto, é que o algoritmo foi elaborado com a proposta de ser otimizado para uma arquitetura em específico: a arquitetura DASH (*Directory Architeture for SHared Memory*). [2]

Referências

[1] PACHECO, Peter. An introduction to Parallel Programming. Burlington, MA, USA: Morgan Kaufmann, 2011.

[2] Nieh, Jason& Levoy, Marc. (2001). Volume Rendering on Scalable Shared-Memory MIMD Architectures. 10.1145/147130.147141.