Trabalho 1 - Sistemas Operacionais de Redes

Integrantes: Diego Martins, Moisés Silva, Paulo Mascarenhas e Pedro Henrique

Introdução

- No que consiste o trabalho?

  “A ideia do trabalho é criar uma rendering-farm de forma que clientes possam utilizar servidores para computar imagens.”

Introdução

- **No que consiste o trabalho?**
- O que é rendering-farm?

  “Possui uma função clara: paralelizar a renderização de imagens geradas computacionalmente.”

Cenários

- Renderização local.

“Neste cenário, a renderização ocorrerá na máquina do próprio cliente, sem a dependência de computadores ou servidores externos. Isso é possível devido a criação de processos e threads. A complexidade neste cenário é bem reduzida, uma vez que não é necessário abrir comunicação, seja por socket ou MPI, com outros equipamentos (computadores ou servidores), ficando a cargo da máquina do cliente apenas renderizar.”

Executando (main.cpp)

```
/*----------TRABALHO 1 - Sistemas Operacionais de Redes----------

int resto = 0,      //para computar quando ainda falta para as threads pegarem no resto da divisao
    aux = 0,        //para auxiliar o resto
    n = NUM_THREADS; //para poder mexer com o numero maximo

char img[7] = {' ',' ',' ',' ',' ',' ','p','n','g'};

int altura = renderer.get_height();   // Pega altura da imagem final

if (NUM_THREADS > altura)           //se o n de threads for maior que a altura o programa limita isso
  n = altura;

int particao = altura/(n);          // Divide a altura em segmentos para a renderização separada

struct data_struct data_values[n];    // Cria uma struct para cada thread
pthread_t *thread = new pthread_t[n]; // Cria NUM_THREADS threads

for (int i = 0; i < n; i++) {         // Defini valores do struct e cria threads
  data_values[i].id = i;
```

Executando (main.cpp)

```
    resto = altura%n - aux;          //calcula um resto para que nao sobre partes sem fazer

    if(resto == 0){                          //as ultimas threads caem aqui
      data_values[i].start = particao*i + aux;          //e realizam somente o tamanho de uma particao de fato
      data_values[i].finish = particao*(i+1) + aux;
    }else{                                   //as primeiras threads devem cair aqui se a divisao nao for inteira
      data_values[i].start = particao*i + aux;          //aqui elas realizam a particao dela +1 (que sera para nao faltar nada)
      data_values[i].finish = particao*(i+1) + aux + 1;
      aux = aux + 1;
    }

    data_values[i].samples = samples;
    data_values[i].rend = &renderer;             // Ponteiro para a variavel renderer da classe Renderer
    pthread_create(&thread[i], NULL, &Renderer::renderStatic, &data_values[i]); // Cria thread enviando a struct como argumento
  }


  for (int i = 0; i < n; i++) {
    pthread_join(thread[i], NULL);               // Espera pela finalização de todas as threads criadas
  }

  //Ccoloca no nome da imagem o numero de threads que foi usado até 999 (maximo = altura = 720)
  img[0] = n/100 + 0x30;
  n = n%100;
  img[1] = n/10 + 0x30;
  img[2] = n%10 + 0x30;

  renderer.save_image(img);                  // Save image
```

Explicação (main.cpp)

- O objetivo das alterações neste arquivo são de criar e gerenciar as threads de renderização do objeto 3D.
- A ideia de resolução do problema foi congelar uma das variáveis da imagem, nesse caso a largura, e variar o valor da altura para possibilitar a renderização parcial em cada thread. Isso fará com que cada thread renderize fragmentos específicos da imagem simultaneamente com outras threads.
- O código foi alterado dividindo a altura da imagem pelo número de threads escolhidas, após isso o programa cria cada thread oferecendo um valor inicial e final da altura para renderização, fazendo com que, no final, toda a imagem seja renderizada por completo por multiprocessamento.

Executando (renderer.cpp)

```
void *Renderer::renderStatic (void *arg) {
  data *data_renderer = (data *)arg;

  int start = data_renderer->start;
  int finish = data_renderer->finish;
  int samples = data_renderer->samples;
  int id = data_renderer->id;
  Renderer *rend = (Renderer *)data_renderer->rend;
  rend->render(samples,id,start,finish);
```

Explicação (renderer.cpp)

- Esta função recebe os argumentos em uma struct de uma thread criada no arquivo “main.cpp” e as converte de volta à seus respectivos tipos de variáveis.
- Essas variáveis passadas na criação da thread serão usadas para limitar a área de renderização, definir o número de amostras para renderizar e adicionar o buffer de renderização calculado de volta à classe “renderer” que armazena o buffer de todas as threads em execução.

Executando (renderer.cpp)

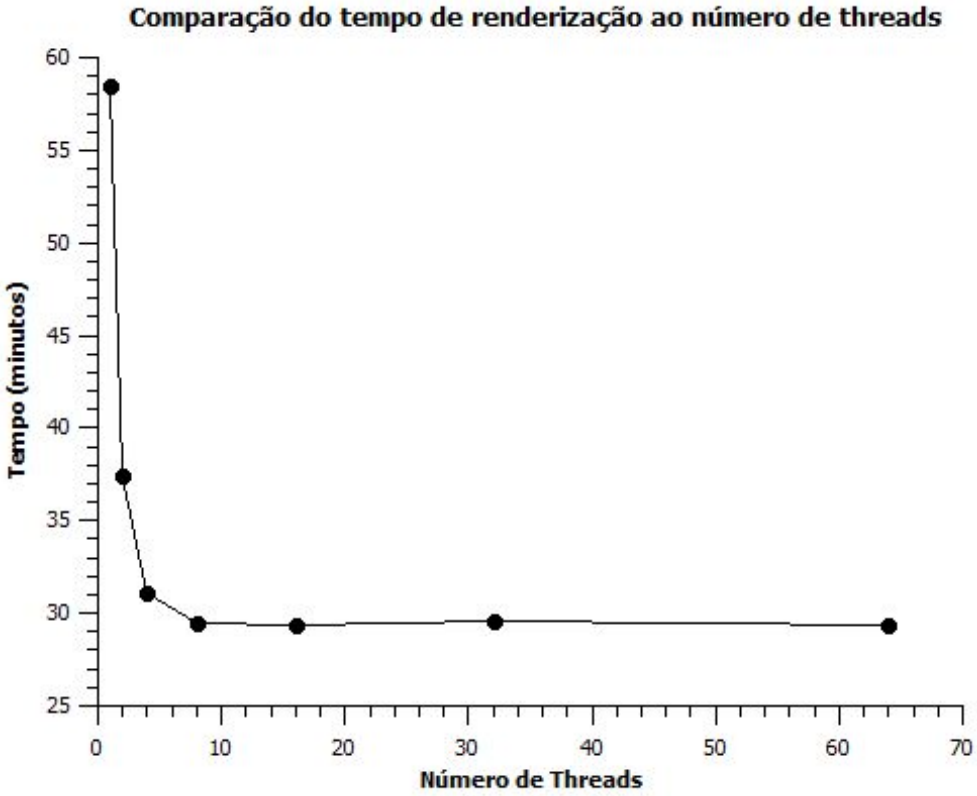
```
    // Main Loop
    for (int y = start; y < finish; y++) {
        unsigned short Xi[3]={0,0,y*y*y};                  // Stores seed for erand48

        fprintf(stderr, "\rRendering Thread '%i' (%i samples): %.2f%% ",// Prints
                id, samples, (double)y/finish*100);                     // progress

        for (int x=0; x<width; x++) {
            Vec col = Vec();

            for (int a=0; a<samples; a++){
                Ray ray = m_camera->get_ray(x, y, a>0, Xi);
                col = col + m_scene->trace_ray(ray,0,Xi);
            }
            m_pixel_buffer[(y)*width + x] = col * samples_recp;
        }
    }
}
```

Explicação (renderer.cpp)

- As alterações feitas na função principal do programa, a renderização, são a de limitação do começo e do fim da renderização parcial de cada thread. Como dito anteriormente, cada thread renderizará um intervalo definido da altura da imagem contendo, claramente, toda a largura da imagem para cada pedaço infinitesimal da altura.

Resultados de renderização (512 amostras)

Número de Threads	Tempo de Execução
1	58 minutos 44 segundos
2	37 minutos 43 segundos
4	31 minutos 15 segundos
8	29 minutos 52 segundos
16	29 minutos 37 segundos
32	29 minutos 53 segundos
64	29 minutos 41 segundos


Comparação do tempo de renderização ao número de threads
Tempo (minutos)
60
55
50
45
40
35
30
25
0
10
20
30
40
50
60
70
Número de Threads

Análise do Grupo

- Dado que o objetivo proposto pelo primeiro cenário seria realizar programação orientada ao multiprocessamento, pode-se dizer que o objetivo foi alcançado com sucesso.
- Analisando os resultados obtidos pelo grupo, pode-se afirmar que ocorreu o que se esperava. Aumentando o número de threads, diminui-se o tempo de renderização do objeto 3D, sendo que isto ocorre apenas em um intervalo limitado dependendo do processador utilizado. Neste caso, foi utilizado um processador com 4 núcleos físicos, ou seja, no máximo apenas 4 núcleos podem realizar a renderização em paralelo, o que implica no valor mínimo do tempo de renderização, que seria utilizando 4 threads.
- Utilizar um número de threads maior do que o número de núcleos do processador não modifica drasticamente o tempo de renderização, mas pelo contrário, às vezes pode até mesmo fazê-lo aumentar, dado que utilizará mais infraestrutura e processamento do que deveria.

Cenários

- **Renderização local;**
- Divisão de tarefas estaticamente.

Divisão de Tarefas

“O escalonamento de tarefas em ambientes distribuídos permite o gerenciamento do uso de um conjunto de recursos limitados para atendimento das demandas de consumo [Casavant and Kuhl 1988]”

Cenário 2: Passo-a-passo

1. A máquina do cliente recebe o modelo 3-D a ser renderizado. Caso fosse realizado localmente esse processo de renderização ocorreria neste instante.

Cenário 2: Passo-a-passo

1. A máquina do cliente recebe o modelo 3-D a ser renderizado. Caso fosse realizado localmente esse processo de renderização ocorreria neste instante;
2. O cliente envia o modelo aos servidores que ele possui acesso. Por ser estático, o programador no ato da codificação já indica como essa passagem se dará e quais partes devem ser tratadas por cada thread ou processo.

Cenário 2: Passo-a-passo

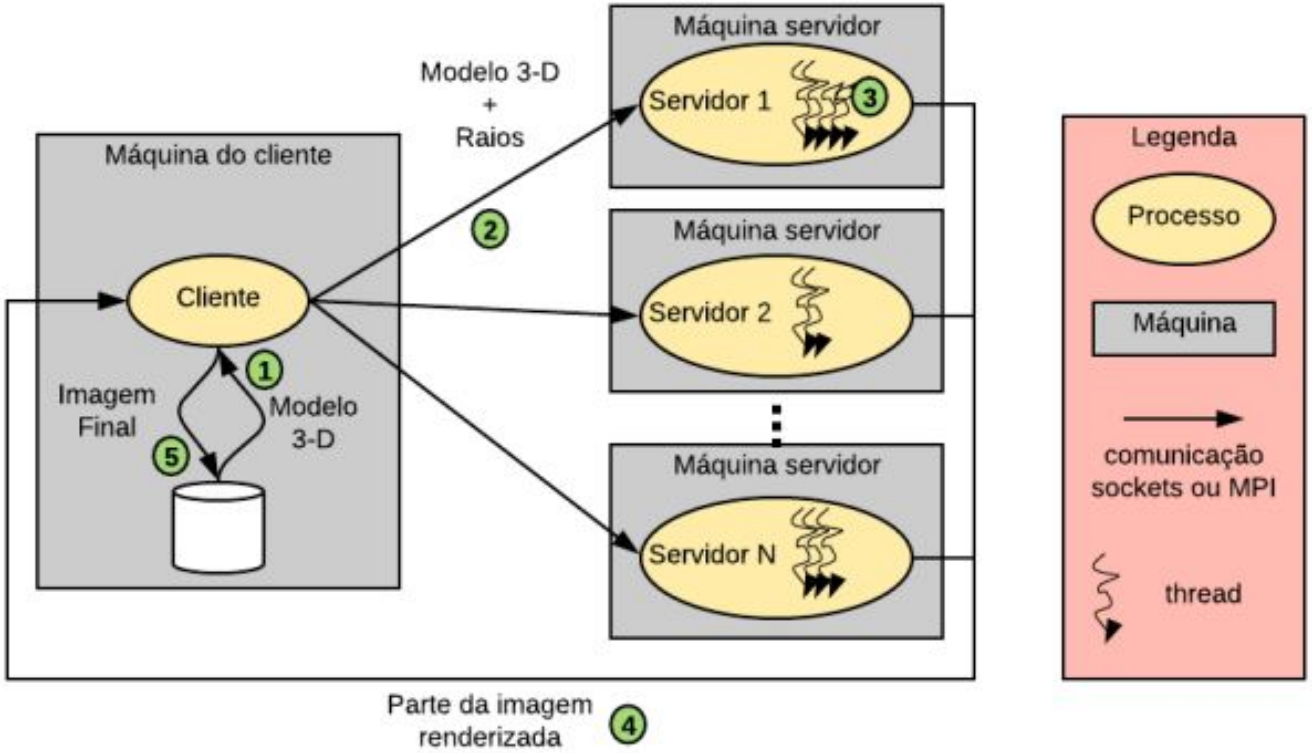
1. A máquina do cliente recebe o modelo 3-D a ser renderizado. Caso fosse realizado localmente esse processo de renderização ocorreria neste instante;
2. O cliente envia o modelo aos servidores que ele possui acesso. Por ser estático, o programador no ato da codificação já indica como essa passagem se dará e quais partes devem ser tratadas por cada thread ou processo;
3. Nesta etapa, ao receber o modelo do cliente, o servidor inicia o processo de renderização.

Cenário 2: Passo-a-passo

1. A máquina do cliente recebe o modelo 3-D a ser renderizado. Caso fosse realizado localmente esse processo de renderização ocorreria neste instante;
2. O cliente envia o modelo aos servidores que ele possui acesso. Por ser estático, o programador no ato da codificação já indica como essa passagem se dará e quais partes devem ser tratadas por cada thread ou processo;
3. Nesta etapa, ao receber o modelo do cliente, o servidor inicia o processo de renderização;
4. Após o término da renderização, o pedaço da imagem é encaminhado ao cliente novamente.

Cenário 2: Passo-a-passo

1. A máquina do cliente recebe o modelo 3-D a ser renderizado. Caso fosse realizado localmente esse processo de renderização ocorreria neste instante;
2. O cliente envia o modelo aos servidores que ele possui acesso. Por ser estático, o programador no ato da codificação já indica como essa passagem se dará e quais partes devem ser tratadas por cada thread ou processo;
3. Nesta etapa, ao receber o modelo do cliente, o servidor inicia o processo de renderização;
4. Após o término da renderização, o pedaço da imagem é enviada ao cliente novamente;
5. O cliente a recebe, monta (seguindo o algoritmo que o programador escolheu) e guarda em sua memória.


Máquina servidor
Servidor 1
3
Modelo 3-D
+
Raios
Máquina do cliente
2
Máquina servidor
Servidor 2
Cliente
1
Imagem
Final
Modelo
3-D
5
Máquina servidor
Servidor N
Parte da imagem
renderizada
4
Legenda
Processo
Máquina
comunicação
sockets ou MPI
thread

Cenários

- **Renderização local;**
- **Divisão de tarefas estaticamente;**
- Divisão de tarefas dinamicamente.

Cenário 3: Passo-a-passo

- A máquina do cliente recebe o modelo 3-D a ser renderizado (assim como no estático).

Cenário 3: Passo-a-passo

- A máquina do cliente recebe o modelo 3-D a ser renderizado (assim como no estático);
- Agora há um intermediário. A máquina proxy será responsável por guardar as tarefas a serem executadas e também os resultados obtidos posteriormente.

Cenário 3: Passo-a-passo

- A máquina do cliente recebe o modelo 3-D a ser renderizado (assim como no estático);
- Agora há um intermediário. A máquina proxy será responsável por guardar as tarefas a serem executadas e também os resultados obtidos posteriormente;
- Dinamicamente, cada servidor acessa a sacola de tarefas a serem realizadas. Esse acesso dependerá do poder de processamento de cada um deles, balanceando a carga entre cada um dos servidores.

Cenário 3: Passo-a-passo

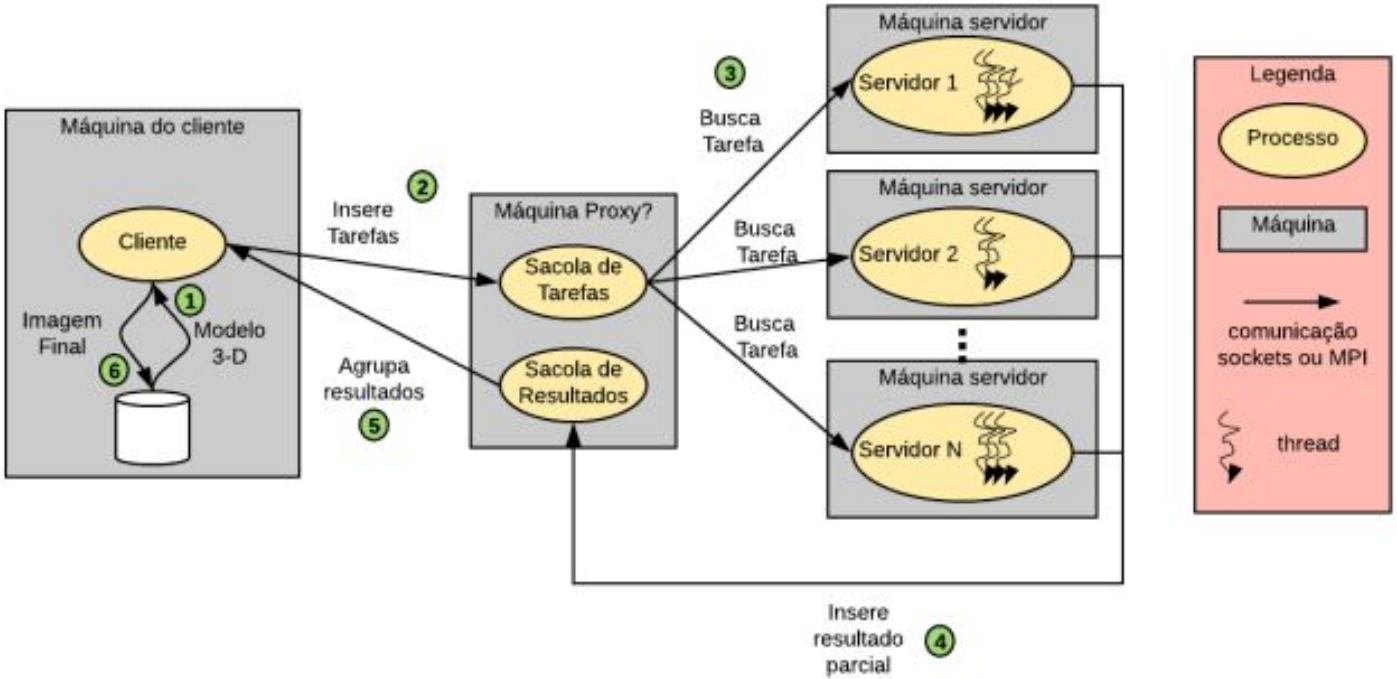
- O servidor renderiza a parte da imagem que lhe foi atribuída da sacola de tarefas (assim como no estático) porém envia à sacola de resultados da máquina proxy (intermediador).

Cenário 3: Passo-a-passo

- O servidor renderiza a parte da imagem que lhe foi atribuída da sacola de tarefas (assim como no estático) porém envia à sacola de resultados da máquina proxy (intermediador);
- A máquina cliente acessa os resultados das imagens renderizadas parcialmente.

Cenário 3: Passo-a-passo

- O servidor renderiza a parte da imagem que lhe foi atribuída da sacola de tarefas (assim como no estático) porém envia à sacola de resultados da máquina proxy (intermediador);
- A máquina cliente acessa os resultados das imagens renderizadas parcialmente;
- Monta a imagem e insere na memória.


Máquina do cliente
Cliente
1
Modelo 3-D
Imagem Final
6
2
Insere Tarefas
Agrupa resultados
5
Máquina Proxy?
Sacola de Tarefas
Sacola de Resultados
3
Busca Tarefa
Busca Tarefa
Busca Tarefa
Máquina servidor
Servidor 1
Máquina servidor
Servidor 2
Máquina servidor
Servidor N
Insere resultado parcial
4
Legenda
Processo
Máquina
comunicação sockets ou MPI
thread

Estático ou Dinâmico?

“Uma vez que as tarefas são realizadas sob demanda, a divisão delas dinamicamente promovem um balanceamento entre os processadores, de forma que eles operem de acordo com sua capacidade”.

“No estático, porém, cada processador recebe um conjunto de tarefas a serem executadas, não levando em conta o tempo de execução. Dessa forma, o processo fica dependente do processador com menos força.”

Bibliografia

- “Render Farm”. Disponível em: https://pt.wikipedia.org/wiki/Render_farm. Acessado por último em 16/10/2018.
- M. ANÇA, F. ANGELIN, G. CAVALHEIRO. “Modelo de Escalonamento Aplicativo para Bag of Tasks em Ambientes de Nuvem Computacional” Disponível em: http://www.lbd.dcc.ufmg.br/colecoes/erad/2017/026.pdf. Acessado por último em16/10/2018.
- Casavant, T. L. and Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Softw. Eng., 14(2):141–154.