Atividades - OpenMP Tasks Sections

Lab Computação Paralela

Table of contents

1 Introdução

Estas atividades visam aprofundar o entendimento prático das diretivas `#pragma omp sections` e `#pragma omp task` do OpenMP.

Para cada atividade:

1. **Analise o Problema:** Entenda o que o código inicial faz.

2. **Implemente o Desafio:** Modifique o código usando `sections` ou `tasks` conforme solicitado.
3. **Avalie a Clareza:** Compare a versão inicial com a sua modificação. O código ficou mais fácil ou mais difícil de entender? Por quê?
4. **Avalie o Desempenho:** Meça o tempo de execução de ambas as versões (Compile com `-O3 -fopenmp`). Execute várias vezes e considere a média. Houve ganho de desempenho? Justifique. Use `omp_get_wtime()`.
5. **Experimente:** Varie o número de threads (`OMP_NUM_THREADS`) e observe o impacto.

Compilação: `gcc -O3 -fopenmp seu_codigo.c -o seu_executavel -lm` (se usar math.h)

---

2 Atividade 1: Análise Estatística de Dados (`sections`)

2.1 Problema

Calcular várias métricas estatísticas (mínimo, máximo, soma e soma dos quadrados para desvio padrão) de um vetor muito grande. A versão inicial usa múltiplos loops paralelos, um para cada métrica principal.

2.2 Código Inicial (analise_inicial.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <limits.h> // Para INT_MAX, INT_MIN
#include <math.h>   // Para sqrt

#define N (1024 * 1024 * 100) // 100 Milhões de inteiros

int main() {
    int *data = (int*) malloc(N * sizeof(int));
    if (!data) { perror("malloc falhou"); return 1; }

    long long i;
    // Inicializa com dados (exemplo)
    printf("Inicializando dados...\n");
    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        data[i] = (i % 1000) - 500; // Valores entre -500 e 499
    }
    printf("Dados inicializados.\n");

    int min_val = INT_MAX;
    int max_val = INT_MIN;
```

```
    long long sum = 0;
    long long sum_sq = 0; // Soma dos quadrados

    double start_time, end_time;
    start_time = omp_get_wtime();

    // --- Bloco a ser otimizado ---
    printf("Calculando métricas (versão inicial)...\n");

    #pragma omp parallel for reduction(min:min_val)
    for (i = 0; i < N; i++) {
        if (data[i] < min_val) {
            min_val = data[i];
        }
    }
    printf("Min calculado.\n");

    #pragma omp parallel for reduction(max:max_val)
    for (i = 0; i < N; i++) {
        if (data[i] > max_val) {
            max_val = data[i];
        }
    }
    printf("Max calculado.\n");

    #pragma omp parallel for reduction(+:sum) reduction(+:sum_sq)
    for (i = 0; i < N; i++) {
        sum += data[i];
        sum_sq += (long long)data[i] * data[i]; // Evita overflow
    }
    printf("Soma e Soma Quadrados calculados.\n");
    // --- Fim do Bloco ---

    end_time = omp_get_wtime();

    double average = (double)sum / N;
    double variance = ((double)sum_sq / N) - (average * average);
    double std_dev = sqrt(variance);

    printf("\n--- Resultados ---\n");
    printf("Min: %d\n", min_val);
    printf("Max: %d\n", max_val);
    printf("Soma: %lld\n", sum);
    printf("Média: %.2f\n", average);
    printf("Desvio Padrão: %.2f\n", std_dev);
    printf("Tempo de execução (cálculo): %.4f segundos\n", end_time - start_time);

    free(data);
    return 0;
}
```

2.3 Desafio (`sections`)

Modifique o "Bloco a ser otimizado" para usar uma única região `#pragma omp parallel` contendo uma construção `#pragma omp sections`. Cada cálculo principal (min, max,

soma/soma_sq) deve ocorrer em uma `#pragma omp section` separada. Use as cláusulas de redução apropriadas na diretiva `parallel` ou `sections`.

3 Atividade 2: Simulação Simplificada de Partículas (`sections`)

3.1 Problema

Simular dois aspectos independentes do movimento de partículas: 1. Atualizar a posição baseado na velocidade atual. 2. Calcular uma nova velocidade baseado em uma força externa simples (ex: gravidade). A versão inicial faz isso em dois loops paralelos sequenciais.

3.2 Código Inicial (simulacao_inicial.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define NUM_PARTICLES (1024 * 1024 * 10)
#define DT 0.01 // Time step

typedef struct {
    double x, y, z; // Posição
    double vx, vy, vz; // Velocidade
} Particle;

int main() {
    Particle *particles = (Particle*) malloc(NUM_PARTICLES * sizeof(Particle));
    if (!particles) { perror("malloc falhou"); return 1; }

    long long i;
    // Inicializa partículas
    printf("Inicializando partículas...\n");
    #pragma omp parallel for
    for (i = 0; i < NUM_PARTICLES; i++) {
        particles[i].x = (double)rand() / RAND_MAX * 100.0;
        particles[i].y = (double)rand() / RAND_MAX * 100.0;
        particles[i].z = (double)rand() / RAND_MAX * 100.0;
        particles[i].vx = (double)rand() / RAND_MAX * 10.0 - 5.0;
        particles[i].vy = (double)rand() / RAND_MAX * 10.0 - 5.0;
        particles[i].vz = (double)rand() / RAND_MAX * 10.0 - 5.0;
    }
    printf("Partículas inicializadas.\n");

    double start_time, end_time;
    start_time = omp_get_wtime();

    // --- Bloco a ser otimizado ---
    printf("Simulando um passo (versão inicial)...\n");
    const double GRAVITY = -9.81;

    // 1. Atualiza Posições
```

```c
    #pragma omp parallel for
    for (i = 0; i < NUM_PARTICLES; i++) {
        particles[i].x += particles[i].vx * DT;
        particles[i].y += particles[i].vy * DT;
        particles[i].z += particles[i].vz * DT;
    }
     printf("  Posições atualizadas.\n");

    // 2. Atualiza Velocidades (baseado em força externa simples)
    #pragma omp parallel for
    for (i = 0; i < NUM_PARTICLES; i++) {
        // Exemplo: Gravidade agindo em Y
        particles[i].vy += GRAVITY * DT;
        // Outras forças poderiam ser calculadas aqui
    }
     printf("  Velocidades atualizadas.\n");
    // --- Fim do Bloco ---

    end_time = omp_get_wtime();

    printf("\nSimulação concluída.\n");
    // Verificar uma partícula (opcional)
    // printf("Partícula 0: Pos(%.2f, %.2f, %.2f) Vel(%.2f, %.2f, %.2f)\n",
    //        particles[0].x, particles[0].y, particles[0].z,
    //        particles[0].vx, particles[0].vy, particles[0].vz);
    printf("Tempo de execução (simulação): %.4f segundos\n", end_time - start_time);

    free(particles);
    return 0;
}
```

3.3 Desafio (`sections`)

Modifique o "Bloco a ser otimizado" para usar uma única região `#pragma omp parallel sections`. Coloque o loop de atualização de posição em uma seção e o loop de atualização de velocidade em outra seção. As duas atualizações podem ocorrer concorrentemente, pois a atualização de posição usa a velocidade *antiga* e a atualização de velocidade não depende da posição *nova*.

4 Atividade 3: Contagem de Linhas em Arquivos (`tasks`)

4.1 Problema

Contar o número total de linhas em todos os arquivos `.c` e `.h` dentro de um diretório e seus subdiretórios. A versão inicial faz isso com uma busca recursiva sequencial.

4.2 Código Inicial (contalinhas_inicial.c)

```c
#define _XOPEN_SOURCE 700 // Para scandir, strdup
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h> // Para operações de diretório
#include <sys/stat.h> // Para stat
#include <errno.h>
#include <omp.h> // Usaremos para timing inicialmente

// Função para contar linhas em um único arquivo
long long count_lines_in_file(const char *filepath) {
    FILE *fp = fopen(filepath, "r");
    if (!fp) {
        // fprintf(stderr, "Erro ao abrir arquivo %s: %s\n", filepath, strerror(errno));
        return 0; // Ignora arquivos que não pode abrir
    }
    long long count = 0;
    int ch;
    while ((ch = fgetc(fp)) != EOF) {
        if (ch == '\n') {
            count++;
        }
    }
    fclose(fp);
    // Adiciona 1 se o arquivo não terminar com newline mas não for vazio
    // Simplificação: vamos apenas contar os '\n'
    return count;
}

// Função recursiva para atravessar diretórios
long long traverse_directory(const char *dirpath) {
    long long total_lines = 0;
    struct dirent **namelist;
    int n;

    n = scandir(dirpath, &namelist, NULL, alphasort);
    if (n < 0) {
        fprintf(stderr, "Erro ao escanear diretório %s: %s\n", dirpath,
                strerror(errno));
        return 0;
    }

    // printf("Entrando em %s (%d itens)\n", dirpath, n);

    for (int i = 0; i < n; i++) {
        // Ignora "." e ".."
        if (strcmp(namelist[i]->d_name, ".") == 0
             || strcmp(namelist[i]->d_name, "..") == 0) {
            free(namelist[i]);
            continue;
        }

        // Monta o caminho completo
        char *fullpath = NULL;
        // Calcula tamanho necessário +2 para '/' e '\0'
        if (asprintf(&fullpath, "%s/%s", dirpath, namelist[i]->d_name) == -1) {
```

```
            fprintf(stderr, "Erro de alocação em asprintf\n");
            free(namelist[i]);
            continue; // Pula este item
        }


        struct stat path_stat;
        if (stat(fullpath, &path_stat) != 0) {
            fprintf(stderr, "Erro no stat para %s: %s\n", fullpath, strerror(errno));
            free(fullpath);
            free(namelist[i]);
            continue;
        }

        // Se for um diretório, chama recursivamente
        if (S_ISDIR(path_stat.st_mode)) {
            total_lines += traverse_directory(fullpath);
        }
        // Se for um arquivo regular e termina com .c ou .h
        else if (S_ISREG(path_stat.st_mode)) {
            size_t len = strlen(namelist[i]->d_name);
            if (len > 2 && (strcmp(namelist[i]->d_name + len - 2, ".c") == 0 ||
                            strcmp(namelist[i]->d_name + len - 2, ".h") == 0))
            {
                 // printf("  Processando arquivo: %s\n", fullpath);
                 total_lines += count_lines_in_file(fullpath);
            }
        }

        free(fullpath);
        free(namelist[i]);
    }
    free(namelist);
    // printf("Saindo de %s\n", dirpath);
    return total_lines;
}

int main(int argc, char *argv[]) {
    const char *start_dir = "."; // Diretório atual por padrão
    if (argc > 1) {
        start_dir = argv[1];
    }

    printf("Contando linhas em .c/.h a partir de: %s (versão inicial sequencial)\n",
           start_dir);

    double start_time, end_time;
    start_time = omp_get_wtime();

    // --- Bloco a ser otimizado ---
    long long total_lines = traverse_directory(start_dir);
    // --- Fim do Bloco ---

    end_time = omp_get_wtime();

    printf("\n--- Resultado ---\n");
    printf("Total de linhas encontradas: %lld\n", total_lines);
    printf("Tempo de execução: %.4f segundos\n", end_time - start_time);
```

```
    return 0;
}
```

4.3 Desafio (`tasks`)

Modifique a função `traverse_directory` para paralelizar a exploração. Quando um subdiretório é encontrado, em vez de chamá-lo recursivamente de forma direta, crie uma `#pragma omp task` para processar aquele subdiretório. Quando um arquivo `.c` ou `.h` é encontrado, crie uma `#pragma omp task` para chamar `count_lines_in_file`. Use `reduction(+:total_lines)` na região `parallel` ou some os resultados das tarefas manualmente usando `#pragma omp atomic` ou `#pragma omp critical` para atualizar um contador global. (A abordagem com redução pode ser mais complexa de integrar com tasks diretamente sem `taskloop`; a abordagem atômica é mais direta aqui). A chamada inicial a `traverse_directory` deve ser envolvida por `#pragma omp parallel` e `#pragma omp single`.

5 Conclusão das Atividades

Ao completar estas atividades, você deve ter uma compreensão mais concreta de:

- Como e quando aplicar `#pragma omp sections` para paralelismo funcional fixo.
- Como e quando aplicar `#pragma omp task` para paralelismo dinâmico e recursivo.
- A importância da sincronização (`taskwait`, barreiras implícitas, `atomic`).
- O conceito de overhead e a necessidade de "cutoffs" em algoritmos de tarefas recursivas.
- Como analisar o impacto dessas diretivas na clareza e no desempenho do código.

Lembre-se que a melhor abordagem (sections, tasks, for, ou mesmo sequencial) depende intrinsecamente da estrutura do problema e dos custos relativos de computação, sincronização e criação de threads/tarefas.

6 Entrega

- Você deve preparar um relatório para a atividade, indicando todas as decisões de projeto, e descrevendo as alterações realizadas e os *prints* da execução; entrega em PDF.
- A entrega será feita apenas no github (Classroom) da disciplina.