Tarefa 7 - Processamento Paralelo com Tasks e Lista Encadeada

Descrição

Este programa demonstra o uso de **OpenMP Tasks** para processamento paralelo de uma lista encadeada. O programa cria uma lista de arquivos fictícios (nomeados com sobrenomes de cientistas famosos) e utiliza tasks para processar cada arquivo de forma paralela e assíncrona.

Funcionalidades

- **Lista Encadeada**: Estrutura de dados dinâmica contendo nomes de arquivos
- **OpenMP Tasks**: Criação de tarefas paralelas para processamento assíncrono
- **Distribuição Dinâmica**: Tasks são distribuídas automaticamente entre threads disponíveis
- **Sincronização Explícita**: Uso de `barrier`, `taskwait`, `single` e `master`
- **Gerenciamento de Memória**: Liberação adequada da memória alocada

Conceitos Demonstrados

1. OpenMP Tasks

```
#pragma omp task firstprivate(nome_local, task_id) { int thread_executora = omp_get_thread_num(); processar_arquivo(nome_local, thread_executora, task_id); }
```

2. Diretiva Single

```
#pragma omp single { // Apenas uma thread cria todas as tasks // Evita duplicação de trabalho }
```

3. Diretiva Master

```
#pragma omp master { // Executado apenas pela thread master (ID 0) // Usado para inicialização e finalização }
```

4. Barreira de Sincronização

```
#pragma omp barrier { // Todas as threads esperam aqui // Garante sincronização entre threads }
```

5. Task Wait

```
#pragma omp taskwait { // Espera todas as tasks criadas pela thread atual // Sincronização explícita de tasks }
```

Estrutura do Programa

Lista Encadeada

```
typedef struct No { char nome_arquivo[50]; struct No* proximo; } No;
```

Arquivos de Cientistas Famosos

```
Einstein.txt

Newton.txt

Darwin.txt

Curie.txt

Tesla.txt

Hawking.txt

Turing.txt
```

```
Galileo.txt

Mendel.txt

Pascal.txt
```

Fluxo de Execução

1. **Criação da Lista**: Adiciona 10 arquivos fictícios à lista encadeada
2. **Região Paralela**: Inicia região paralela com múltiplas threads
3. **Inicialização Master**: Thread master (ID 0) executa inicialização
4. **Barreira Inicial**: Todas as threads sincronizam antes do processamento
5. **Criação de Tasks**: Uma única thread percorre a lista e cria tasks
6. **Processamento Paralelo**: Tasks são executadas por diferentes threads
7. **Task Wait**: Aguarda explicitamente todas as tasks terminarem
8. **Barreira Final**: Sincronização após conclusão das tasks
9. **Finalização Master**: Thread master executa limpeza final
10. **Limpeza**: Libera memória da lista encadeada

Comandos OpenMP Utilizados

Comando	Função	Uso no Programa
`#pragma omp single`	Execução por apenas uma thread	Criação de tasks (evita duplicação)
`#pragma omp master`	Execução pela thread master (ID 0)	Inicialização e finalização do sistema
`#pragma omp barrier`	Sincronização de todas as threads	Pontos de sincronização no fluxo
`#pragma omp task`	Criação de tarefas assíncronas	Processamento paralelo dos arquivos

Comando	Função	Uso no Programa
`#pragma omp taskwait`	Aguardar conclusão das tasks	Sincronização explícita das tasks

Exemplo de Saída

```
=== PROCESSAMENTO PARALELO DE ARQUIVOS COM TASKS === Criando lista de arquivos fictícios... Número de threads disponíveis: 8 Iniciando processamento paralelo... Thread master 0 inicializando sistema... Thread 7 criando tasks para processamento... ==> Task 1 iniciada na Thread 6: Einstein.txt ==> Task 4 iniciada na Thread 1: Curie.txt -> Thread 1: Analisando conteúdo de Curie.txt... ==> Task 2 iniciada na Thread 3: Newton.txt -> Thread 3: Analisando conteúdo de Newton.txt... Todas as 10 tasks foram criadas! Aguardando conclusão de todas as tasks... -> Thread 3: Processamento de Newton.txt concluído! ==> Task 2 finalizada na Thread 3 Thread master 0 finalizando processamento... === PROCESSAMENTO CONCLUÍDO === Todos os arquivos foram processados com sucesso!
```

Vantagens das Tasks

1. Balanceamento Dinâmico

- Tasks são distribuídas automaticamente
- Threads ociosas pegam novas tasks
- Melhor utilização de recursos

2. Flexibilidade

- Número variável de tasks
- Criação condicional de tasks
- Aninhamento de tasks possível

3. Desacoplamento

- Criação e execução são independentes

- Uma thread cria, outras executam
- Escalabilidade natural

Análise de Performance

Características Observadas

- **Distribuição Não-Determinística**: A cada execução, tasks podem ser executadas por threads diferentes
- **Utilização Eficiente**: Múltiplas threads trabalham simultaneamente
- **Overhead Mínimo**: Tasks são criadas rapidamente com sincronização eficiente

Conceitos de Programação Paralela

- **Task Parallelism**: Diferentes threads executam diferentes tarefas
- **Work Stealing**: OpenMP implementa algoritmo de work stealing
- **Fork-Join Estendido**: Tasks estendem o modelo fork-join tradicional

Comparação: Tasks vs Parallel For

Aspecto	Tasks	Parallel For
Estrutura de Dados	Qualquer (lista, árvore)	Arrays/loops
Balanceamento	Dinâmico automático	Estático/manual
Flexibilidade	Alta	Limitada
Overhead	Ligeiramente maior	Menor
Casos de Uso	Trabalho irregular	Trabalho uniforme

Compilação e Execução

```
# Compilação gcc -fopenmp -o tarefa7 tarefa7.c # Execução ./tarefa7
```

Conclusão: Este programa demonstra como os 5 comandos OpenMP fundamentais (`single` , `master` , `barrier` , `task` , `taskwait`) trabalham em conjunto para criar um sistema de processamento paralelo robusto e bem sincronizado.

Visualização do Programa

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

// Estrutura do nó da lista encadeada
typedef struct No {
    char nome_arquivo[50];
    struct No* proximo;
} No;

// Função para criar um novo nó
No* criar_no(const char* nome) {
    No* novo_no = (No*)malloc(sizeof(No));
    if (novo_no != NULL) {
        strcpy(novo_no->nome_arquivo, nome);
        novo_no->proximo = NULL;
    }
    return novo_no;
}

// Função para adicionar um nó no final da lista
void adicionar_no(No** cabeca, const char* nome) {
    No* novo_no = criar_no(nome);
    if (*cabeca == NULL) {
        *cabeca = novo_no;
    } else {
        No* atual = *cabeca;
        while (atual->proximo != NULL) {
            atual = atual->proximo;
        }
        atual->proximo = novo_no;
    }
}

// Função para processar um arquivo (simulação)
void processar_arquivo(const char* nome_arquivo, int thread_id, int task_id) {
    printf("==> Task %d iniciada na Thread %d: %s\n", task_id, thread_id, nome_arquivo);

    // Simular algum processamento - sem critical, cada thread imprime independentemente
    printf("  -> Thread %d: Analisando conteúdo de %s...\n", thread_id, nome_arquivo);

    // Simular tempo de processamento variável
    for (volatile int i = 0; i < 1000000; i++);

    printf("  -> Thread %d: Processamento de %s concluído!\n", thread_id, nome_arquivo);
    printf("==> Task %d finalizada na Thread %d\n\n", task_id, thread_id);
}

// Função para liberar a memória da lista
void liberar_lista(No* cabeca) {
    No* atual = cabeca;
    while (atual != NULL) {
        No* temp = atual;
        atual = atual->proximo;
        free(temp);
    }
}

int main() {
    // Criar lista encadeada com nomes de arquivos baseados em cientistas famosos
    No* lista_arquivos = NULL;

    printf("=== PROCESSAMENTO PARALELO DE ARQUIVOS COM TASKS ===\n");
    printf("Criando lista de arquivos fictícios...\n\n");

    // Adicionar arquivos com nomes de cientistas famosos
    adicionar_no(&lista_arquivos, "Einstein.txt");
    adicionar_no(&lista_arquivos, "Newton.txt");
    adicionar_no(&lista_arquivos, "Darwin.txt");
    adicionar_no(&lista_arquivos, "Curie.txt");
    adicionar_no(&lista_arquivos, "Tesla.txt");
    adicionar_no(&lista_arquivos, "Hawking.txt");
```

```c
    adicionar_no(&lista_arquivos, "Turing.txt");
    adicionar_no(&lista_arquivos, "Galileo.txt");
    adicionar_no(&lista_arquivos, "Mendel.txt");
    adicionar_no(&lista_arquivos, "Pascal.txt");

    printf("Número de threads disponíveis: %d\n", omp_get_max_threads());
    printf("Iniciando processamento paralelo...\n\n");

    // Região paralela com tasks
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();

        // Master thread faz inicialização
        #pragma omp master
        {
            printf("Thread master %d inicializando sistema...\n", thread_id);
        }

        // Barrier para garantir que todas as threads estejam prontas
        #pragma omp barrier

        // Apenas uma thread cria as tasks (single)
        #pragma omp single
        {
            printf("Thread %d criando tasks para processamento...\n\n", omp_get_thread_num());

            No* atual = lista_arquivos;
            int contador_arquivos = 0;

            // Percorrer a lista e criar uma task para cada nó
            while (atual != NULL) {
                contador_arquivos++;

                // Capturar o nome do arquivo por valor para evitar race conditions
                char nome_local[50];
                strcpy(nome_local, atual->nome_arquivo);
                int task_id = contador_arquivos;

                // Criar task para processar este arquivo
                #pragma omp task firstprivate(nome_local, task_id)
                {
                    int thread_executora = omp_get_thread_num();
                    processar_arquivo(nome_local, thread_executora, task_id);
                }

                atual = atual->proximo;
            }

            printf("Todas as %d tasks foram criadas!\n", contador_arquivos);
            printf("Aguardando conclusão de todas as tasks...\n\n");
        }

        // Aguardar explicitamente todas as tasks terminarem
        #pragma omp taskwait

        // Barrier para sincronizar todas as threads após as tasks
        #pragma omp barrier

        // Master thread faz finalização
        #pragma omp master
        {
            printf("Thread master %d finalizando processamento...\n", thread_id);
        }
    }

    printf("\n=== PROCESSAMENTO CONCLUÍDO ===\n");
    printf("Todos os arquivos foram processados com sucesso!\n");

    // Liberar memória da lista
    liberar_lista(lista_arquivos);

    return 0;
}

//gcc -fopenmp -o tarefa7 tarefa7.c
//./tarefa7
```