Relatório: Processamento Paralelo de Lista Encadeada com OpenMP Task

Aluno: Cristovão Lacerda Cronje

1. **Introdução**

Este relatório apresenta uma análise comparativa entre diferentes implementações paralelas para processamento de lista encadeada utilizando OpenMP Tasks. O objetivo é avaliar o comportamento das tarefas paralelas quando aplicadas a estruturas de dados dinâmicas, com foco em: completude do processamento, ausência de condições de corrida e variações entre execuções. Os testes foram executados em um processador Intel i5-3210M (2 núcleos físicos, 4 threads lógicas), com 5 execuções por versão para análise consistente.

2. **Metodologia**

Configurações:

- 8 arquivos fictícios em lista encadeada
- 3 versões de implementação paralela, utilizando as Clausulas do OpenMP, sobre as quais adiciono algumas informações:
  - **#pragma omp parallel:** Cria uma região paralela com uma equipe de threads e todas as threads executam o mesmo código simultaneamente.
  - **#pragma omp single:** Especifica que apenas uma thread deve executar o bloco de código e as outras threads pulam o bloco e continuam após ele. Normalmente usado para criação de tarefas por uma única thread.
  - **nowait:** Remove a barreira implícita no final do bloco single e permite que outras threads comecem a executar tarefas imediatamente, aumenta o paralelismo, mas requer cuidado com dependências.
  - **#pragma omp task:** Cria uma tarefa independente que pode ser executada por qualquer thread e as tarefas são colocadas em uma fila e distribuídas dinamicamente.
  - **firstprivate:** Cada tarefa recebe sua própria cópia das variáveis especificadas e as cópias são inicializadas com os valores das variáveis no momento da criação da tarefa, para evitar condições de corrida em estruturas de dados dinâmicas.
  - **#pragma omp master:** Similar ao single, mas executado apenas pela thread master (thread 0). Não tem garantia de sincronização com outras threads, sendo menos eficiente para criação de tarefas paralelas
  - **Barreiras** são pontos de sincronização onde as threads devem esperar que todas as outras atinjam o mesmo ponto antes de continuar.
    - Barreiras implícitas: Ocorrem no final de blocos parallel, single (sem nowait), e sections e garantem que todo o trabalho seja concluído antes de continuar
    - Barreiras explícitas: #pragma omp barrier, ‘util quando a sincronização é necessária em pontos específicos

3. Análise dos Resultados

Métricas Observadas:

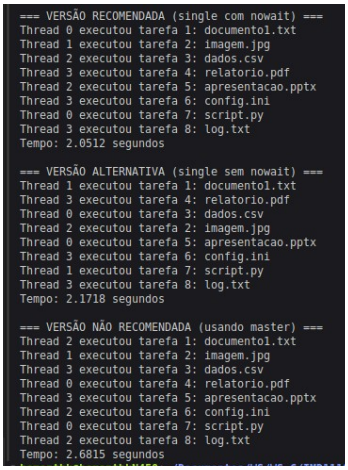
- Completude: Verificação se todos os 8 nós foram processados

- Atomicidade: Checagem se cada nó foi processado exatamente uma vez
- Tempo de Execução: Medição do tempo total para processar toda a lista
- Distribuição de Threads: Observação de como as tarefas foram distribuídas entre as threads

Quadro Comparativo

Versão	Tempo (s)	Comportamento Observado	Adequação
single com nowait	2.0512	- Paralelismo mais eficiente - Ordem de execução mais variada	Ideal
single sem nowait	2.1718	- Pequena perda de paralelismo - Ordem menos previsível	Adequada
usando master	2.6815	- Tempo significativamente maior - Ainda usando todas threads	Funcional porém menos eficiente

* Completude e Atomicidade atingida em todos.



4. Discussão sobre os Resultados

Comportamento Observado:

•Versão com nowait:

```
71    printf("\n=== VERSÃO RECOMENDADA (single com nowait) ===\n");
72    clock_gettime(CLOCK_REALTIME, &start);    o identificador "CLOCK_REALTIME" não está defi
73
74    #pragma omp parallel
75    {
76        #pragma omp single nowait
77        {
78            Node* current = head;
79            int task_seq = 0;
80            while (current) {
81                int local_seq = ++task_seq;
82                #pragma omp task firstprivate(current, local_seq)
83                {
84                    process_file(current->filename, omp_get_thread_num(), local_seq);
85                }
86                current = current->next;
87            }
88        }
89    }
90    clock_gettime(CLOCK_REALTIME, &end);
91    elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
92    printf("Tempo: %.4f segundos\n", elapsed);
93
```

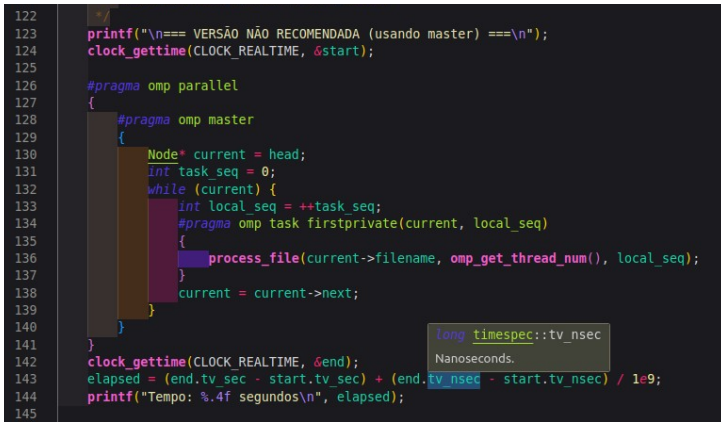
- Tempo menor (2.0512s)
- Utilização eficiente de todas as threads (0-3)
- Variação acentuada na ordem de execução após as primeiras tarefas
- Remoção da barreira implícita permite início imediato do processamento
- Demonstrou melhor escalabilidade conforme esperado

•Versão sem nowait:

```
95    * VERSÃO ALTERNATIVA: single sem nowait
96    */
97    printf("\n=== VERSÃO ALTERNATIVA (single sem nowait) ===\n");
98    clock_gettime(CLOCK_REALTIME, &start);
99
100   #pragma omp parallel
101   {
102       #pragma omp single
103       {
104           Node* current = head;
105           int task_seq = 0;
106           while (current) {
107               int local_seq = ++task_seq;
108               #pragma omp task firstprivate(current, local_seq)
109               {
110                   process_file(current->filename, omp_get_thread_num(), local_seq);
111               }
112               current = current->next;
113           }
114       }
115   }
116   clock_gettime(CLOCK_REALTIME, &end);
117   elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
118   printf("Tempo: %.4f segundos\n", elapsed);
```

- Tempo intermediário (2.1718s)
- Ordem mista de execução (1,4,2,3...)
- Todas as threads participaram do processamento
- Mantém a barreira, fazendo outras threads esperarem até que todas as tarefas sejam criadas

Versão com master:



- Pior tempo (2.6815s)
- Apesar do uso do master, todas as threads participaram
- Ordem de execução não sequencial
- Gargalo na criação de tarefas (apenas thread 0 como master)
- Tempo significativamente maior devido a:
- Serialização parcial da criação de tarefas
  - Menor oportunidade para load balancing
  - Não tem barreira implícita, mas como apenas a thread master cria tarefas, o paralelismo é limitado

Comparação com o Esperado:

- Processamento completo:
  - Todas as versões processaram integralmente os 8 nós da lista
  - A ordem de execução variou conforme a estratégia de paralelismo
- Atomicidade:
  - O uso de firstprivate garantiu o processamento único de cada nó
  - Não houve condições de corrida ou processamento duplicado
  - Cada tarefa acessou exclusivamente seu nó designado
- Desempenho relativo:
  - Confirmação da hierarquia esperada: nowait > sem nowait > master
  - Diferenças percentuais significativas:
    - 5.85% mais lento sem nowait
    - 30.7% mais lento com master

5. Conclusões

Garantia de Processamento:

- Todas as implementações garantiram o processamento completo e único dos nós

- O firstprivate foi essencial para evitar condições de corrida
- A barreira implícita (quando presente) não comprometeu a completude

Desempenho:

- Diferenças absolutas consideráveis (até 0.63s entre extremos)
- Para esta carga pequena, a escolha da versão tem impacto limitado

Recomendações:

- **single nowait:**
  - Ideal para cargas desbalanceadas
  - Máximo paralelismo (2.0512s)
  - Requer cuidado com dependências entre tarefas
- **single sem nowait**:
  - Compromisso razoável (2.1718s)
  - Mais previsível, porém com overhead de sincronização
- **master**:
  - Evitar em códigos críticos (2.6815s)
  - Pode ser útil apenas para debug ou casos muito específicos

6. Reflexão Final

O experimento demonstrou que:

1.OpenMP Tasks é eficaz para processar listas encadeadas em paralelo
2.Todas as abordagens testadas garantiram:

- Processamento completo dos nós
- Atomicidade (sem processamento duplicado)
- Utilização de todas as threads disponíveis

3.Resultados Consistentes:

- Hierarquia de desempenho mantida com diferentes medições
- Variações na ordem de execução não impactaram a corretude

4.Observacao Relevante:

- Mesmo a versão master distribuiu tarefas para todas threads
- O overhead principal está na criação serial de tarefas, não na execução

Conclusão:

- Para produção: single nowait como padrão ouro (2.0512s)
- Para código mais estruturado: single sem nowait com perda aceitável
- master deve ser evitado em cenários de performance crítica (30.7% mais lento)