Relatório da Tarefa 09

Regiões críticas nomeadas e Locks explícitos

20210093987 - Francisca Paula de Souza Braz
Data da entrega: 02/05/2025
Disciplina: DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.1)

Introdução

Em sistemas paralelos, o controle adequado de acesso a recursos compartilhados é essencial para garantir a integridade dos dados. Este relatório explora a sincronização de inserções concorrentes em listas encadeadas usando OpenMP, com foco na comparação entre regiões críticas nomeadas e locks explícitos.

O experimento busca ilustrar como diferentes abordagens afetam o desempenho e a escalabilidade conforme o número de estruturas de dados protegidas aumenta.

O objetivo foi Implementar e analisar um programa paralelo em C com OpenMP que realiza N inserções em listas encadeadas. Inicialmente, o programa utiliza duas listas, cada uma associada a uma thread, e emprega **regiões críticas nomeadas** para garantir que inserções em uma lista não bloqueiem a outra. Em seguida, o programa é generalizado para permitir um número arbitrário de listas definido pelo usuário, onde somente usar regiões críticas nomeadas não são mais suficiente e se faz necessário a utilização **locks explícitos** para manter a integridade das estruturas.

Metodologia

Duas listas com regiões críticas nomeadas: cada thread realiza inserções aleatórias em uma das duas listas. O uso de **#pragma omp critical (list1_lock)** assegura exclusão mútua em cada lista separadamente.

```c
#pragma omp parallel
{
    unsigned int seed = omp_get_thread_num() + 123;  // semente por thread

    #pragma omp for
    for (int i = 0; i < N; i++) {
        int value = rand_r(&seed) % 1000;    // número aleatório
        int target = rand_r(&seed) % 2;       // escolhe lista 0 ou 1

        if (target == 0) {
            #pragma omp critical(list1_lock)
            insert(&list1, value);
        } else {
            #pragma omp critical(list2_lock)
            insert(&list2, value);
        }
    }
}
```

Para lidar com a generização das listas arbitrárias, alocam-se vetores de ponteiros e vetores de omp_lock_t. O codigo não funcionava então para cada lista adicionou-se um lock independente, utilizado com omp_set_lock() e omp_unset_lock().

```
    omp_lock_t* locks = malloc(K * sizeof(omp_lock_t));
    for (int i = 0; i < K; i++)
        omp_init_lock(&locks[i]);

    #pragma omp parallel
    {
        unsigned int seed = omp_get_thread_num();
        int tid = omp_get_thread_num();

        #pragma omp for
        for (int i = 0; i < N; i++) {
            int value = rand_r(&seed) % 1000;
            int target = rand_r(&seed) % K;

            omp_set_lock(&locks[target]);
            insert(&listas[target], value);
            omp_unset_lock(&locks[target]);

            #pragma omp atomic
            contagem[tid][target]++;
        }
    }

    for (int i = 0; i < K; i++)
        omp_destroy_lock(&locks[i]);
    free(locks);
}
```

Durante a execução, o programa também rastreia quais threads acessam quais listas, registrando a distribuição de inserções por thread.

Resultados

Os testes foram realizados com diferentes números de listas (2, 4, 8, 16) e 100.000 inserções distribuídas aleatoriamente por threads. A implementação foi avaliada em termos de integridade, concorrência e escalabilidade dos mecanismos de sincronização.

```
Digite o número de listas: 4

--- Distribuição de inserções por thread ---
Thread  0: L0:  2034  L1:  2115  L2:  2077  L3:  2108
Thread  1: L0:  2068  L1:  2148  L2:  2013  L3:  2105
Thread  2: L0:  2156  L1:  2036  L2:  2030  L3:  2112
Thread  3: L0:  2145  L1:  2104  L2:  2038  L3:  2047
Thread  4: L0:  2059  L1:  2124  L2:  2078  L3:  2072
Thread  5: L0:  2067  L1:  2040  L2:  2069  L3:  2157
Thread  6: L0:  2032  L1:  2026  L2:  2128  L3:  2147
Thread  7: L0:  2074  L1:  2120  L2:  2064  L3:  2075
Thread  8: L0:  2061  L1:  2133  L2:  2092  L3:  2047
Thread  9: L0:  2127  L1:  2055  L2:  2064  L3:  2087
Thread 10: L0:  2113  L1:  2076  L2:  2115  L3:  2029
Thread 11: L0:  2029  L1:  2107  L2:  2150  L3:  2047

--- Número total de elementos por lista ---
Lista 0: 24965 elementos
Lista 1: 25084 elementos
Lista 2: 24918 elementos
Lista 3: 25033 elementos
paula@paula-Inspiron-15-3520:~/Documentos$
```

Saída das 04 Listas

```
Digite o número de listas: 8

--- Distribuição de inserções por thread ---
Thread  0: L0:   999  L1:  1114  L2:  1024  L3:  1053  L4:  1035  L5:  1001  L6:  1053  L7:  1055
Thread  1: L0:   995  L1:  1061  L2:   989  L3:  1052  L4:  1073  L5:  1087  L6:  1024  L7:  1053
Thread  2: L0:  1061  L1:   976  L2:  1017  L3:  1046  L4:  1095  L5:  1060  L6:  1013  L7:  1066
Thread  3: L0:  1078  L1:  1050  L2:  1029  L3:  1079  L4:  1067  L5:  1054  L6:  1009  L7:   968
Thread  4: L0:  1039  L1:  1083  L2:  1021  L3:   991  L4:  1020  L5:  1041  L6:  1057  L7:  1081
Thread  5: L0:   995  L1:  1009  L2:  1052  L3:  1079  L4:  1072  L5:  1031  L6:  1017  L7:  1078
Thread  6: L0:  1049  L1:  1019  L2:  1072  L3:  1056  L4:   983  L5:  1007  L6:  1056  L7:  1091
Thread  7: L0:  1026  L1:  1025  L2:  1089  L3:  1044  L4:  1048  L5:  1095  L6:   975  L7:  1031
Thread  8: L0:  1015  L1:  1062  L2:  1042  L3:  1082  L4:  1046  L5:  1071  L6:  1050  L7:   965
Thread  9: L0:  1034  L1:  1085  L2:  1036  L3:  1054  L4:  1093  L5:   970  L6:  1028  L7:  1033
Thread 10: L0:  1062  L1:  1028  L2:  1099  L3:  1021  L4:  1051  L5:  1048  L6:  1016  L7:  1008
Thread 11: L0:  1040  L1:  1066  L2:  1057  L3:  1044  L4:   989  L5:  1041  L6:  1093  L7:  1003

--- Número total de elementos por lista ---
Lista 0: 12393 elementos
Lista 1: 12578 elementos
Lista 2: 12527 elementos
Lista 3: 12601 elementos
Lista 4: 12572 elementos
Lista 5: 12506 elementos
Lista 6: 12391 elementos
Lista 7: 12432 elementos
```

Saída das 08 Listas

```
paula@paula-Inspiron-15-3520:~/Documentos$ ./tt
Digite o número de listas: 16

--- Distribuição de inserções por thread ---
Thread  0: L0:   480  L1:   563  L2:   492  L3:   522  L4:   512  L5:   494  L6:   522  L7:   496  L8:   510  L9:   551  L10:   532  L11:   531  L12:   523  L13:   507  L14:   531  L15:   559
Thread  1: L0:   486  L1:   556  L2:   487  L3:   541  L4:   506  L5:   537  L6:   514  L7:   538  L8:   509  L9:   505  L10:   502  L11:   511  L12:   567  L13:   550  L14:   510  L15:   515
Thread  2: L0:   528  L1:   496  L2:   509  L3:   546  L4:   535  L5:   536  L6:   521  L7:   510  L8:   533  L9:   480  L10:   508  L11:   500  L12:   560  L13:   524  L14:   492  L15:   556
Thread  3: L0:   537  L1:   504  L2:   518  L3:   516  L4:   540  L5:   520  L6:   492  L7:   465  L8:   541  L9:   546  L10:   511  L11:   563  L12:   527  L13:   534  L14:   517  L15:   503
Thread  4: L0:   523  L1:   548  L2:   502  L3:   505  L4:   501  L5:   560  L6:   542  L7:   548  L8:   516  L9:   535  L10:   519  L11:   486  L12:   519  L13:   481  L14:   515  L15:   533
Thread  5: L0:   485  L1:   536  L2:   533  L3:   526  L4:   551  L5:   483  L6:   527  L7:   532  L8:   510  L9:   473  L10:   519  L11:   553  L12:   521  L13:   548  L14:   490  L15:   546
Thread  6: L0:   542  L1:   503  L2:   515  L3:   513  L4:   477  L5:   517  L6:   513  L7:   515  L8:   507  L9:   516  L10:   557  L11:   543  L12:   506  L13:   490  L14:   543  L15:   576
Thread  7: L0:   522  L1:   539  L2:   530  L3:   507  L4:   519  L5:   561  L6:   495  L7:   499  L8:   504  L9:   486  L10:   559  L11:   537  L12:   529  L13:   534  L14:   480  L15:   532
Thread  8: L0:   502  L1:   566  L2:   538  L3:   570  L4:   511  L5:   544  L6:   524  L7:   484  L8:   513  L9:   496  L10:   504  L11:   512  L12:   535  L13:   527  L14:   526  L15:   481
Thread  9: L0:   504  L1:   528  L2:   497  L3:   518  L4:   558  L5:   471  L6:   525  L7:   534  L8:   530  L9:   557  L10:   539  L11:   536  L12:   535  L13:   499  L14:   503  L15:   499
Thread 10: L0:   534  L1:   508  L2:   572  L3:   516  L4:   525  L5:   529  L6:   506  L7:   503  L8:   528  L9:   520  L10:   527  L11:   505  L12:   526  L13:   519  L14:   510  L15:   505
Thread 11: L0:   536  L1:   545  L2:   514  L3:   523  L4:   518  L5:   524  L6:   544  L7:   499  L8:   504  L9:   521  L10:   543  L11:   521  L12:   471  L13:   517  L14:   549  L15:   504

--- Número total de elementos por lista ---
Lista 0: 6179 elementos
Lista 1: 6392 elementos
Lista 2: 6207 elementos
Lista 3: 6303 elementos
Lista 4: 6253 elementos
Lista 5: 6270 elementos
Lista 6: 6225 elementos
Lista 7: 6123 elementos
Lista 8: 6214 elementos
Lista 9: 6186 elementos
Lista 10: 6320 elementos
Lista 11: 6298 elementos
Lista 12: 6319 elementos
Lista 13: 6230 elementos
Lista 14: 6166 elementos
```

Saída das 16 Listas

Com **duas listas**, o uso de **regiões críticas nomeadas** foi eficiente, pois a cada thread inseria em sua respectiva lista sem bloqueio mútuo, permitindo paralelismo efetivo e mantendo a consistência dos dados. Como cada região crítica tinha um nome distinto, as inserções em listas diferentes não competiam pelo mesmo lock, evitando contenção desnecessária.

Ao generalizar para um número arbitrário de listas, tornou-se inviável o uso de regiões críticas nomeadas, já que o OpenMP exige que os nomes sejam definidos estaticamente. Para contornar isso, utilizamos um vetor de **locks explícitos** (`omp_lock_t`), permitindo a sincronização individualizada para cada lista.

Essa abordagem demonstraram-se eficazes para as inserções mantiveram integridade, e o paralelismo foi preservado mesmo com maior número de listas. A estratégia garantiu que múltiplas threads pudessem operar simultaneamente em listas distintas, sem conflito. Além disso, foi possível registrar quantas inserções cada thread fez em cada lista, o que permitiu verificar que a distribuição foi aproximadamente uniforme e que as threads acessaram as listas de forma balanceada.

Os tempos de execução foram similares entre os dois métodos para pequenas quantidades de listas, mas os **locks explícitos escalaram melhor** à medida que o número de listas aumentou, mantendo bom desempenho e evitando gargalos de sincronização.

Conclusão

A escolha do mecanismo de sincronização depende diretamente da escalabilidade necessária. **Regiões críticas nomeadas** são apropriadas para um número pequeno e fixo de recursos compartilhados, pela sua simplicidade e legibilidade. Entretanto, para um número arbitrário ou elevado de estruturas de dados, são necessárias soluções mais flexíveis. Os **locks explícitos** oferecem maior controle e possibilitam uma sincronização dinâmica e eficiente, sendo a opção recomendada quando o número de regiões críticas não pode ser definido estaticamente. Este experimento reforça a importância de adaptar a estratégia de sincronização à natureza e escala do problema.